Problems with Solutions in the Analysis of Algorithms

Minko Markov

Draft date May 8, 2014

Contents

Chapter 1

Notations: Θ, O, Ω, o, and ω

The functions we consider are assumed to have positive real domains and real codomains unless specified otherwise. Furthermore, the functions are assumed to be *asymptotically positive*. The function $f(n)$ is asymptotically positive iff $\exists n_0 : \forall n \geq n_0, f(n) > 0$.

Basic definitions:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\} \quad (1.1)$$

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) \leq c.g(n)\} \quad (1.2)$$

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq c.g(n) \leq f(n)\} \quad (1.3)$$

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) < c.g(n)\} \quad (1.4)$$

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq c.g(n) < f(n)\} \quad (1.5)$$

1.4 is equivalent to:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \quad (1.6)$$

if the limit exists. 1.5 is equivalent to:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \quad (1.7)$$

if the limit exists.

It is universally accepted to write "$f(n) = \Theta(g(n))$" instead of the formally correct "$f(n) \in \Theta(g(n))$".

Let us define the binary relations $\asymp$, $\preceq$, $\prec$, $\succeq$, and $\succ$ over functions as follows. For any two functions $f(n)$ and $g(n)$:

$$f(n) \asymp g(n) \quad \Leftrightarrow \quad f(n) = \Theta(g(n)) \tag{1.8}$$

$$f(n) \preceq g(n) \quad \Leftrightarrow \quad f(n) = O(g(n)) \tag{1.9}$$

$$f(n) \prec g(n) \quad \Leftrightarrow \quad f(n) = o(g(n)) \tag{1.10}$$

$$f(n) \succeq g(n) \quad \Leftrightarrow \quad f(n) = \Omega(g(n)) \tag{1.11}$$

$$f(n) \succ g(n) \quad \Leftrightarrow \quad f(n) = \omega(g(n)) \tag{1.12}$$

When the relations do not hold we write $f(n) \not\asymp g(n)$, $f(n) \not\preceq g(n)$, *etc.*

Properties of the relations:

1. Reflexivity: $f(n) \asymp f(n)$, $f(n) \preceq f(n)$, $f(n) \succeq f(n)$.
2. Symmetry: $f(n) \asymp g(n) \;\Leftrightarrow\; g(n) \asymp f(n)$.

   Proof: Assume $\exists c_1, c_2, n_0 > 0$ as necessitated by (1.1), so that $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$. Then $0 \leq \frac{1}{c_2} f(n) \leq g(n)$ and $g(n) \leq \frac{1}{c_1} f(n)$. Overall, $0 \leq \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$. So there exist positive constants $k_1 = \frac{1}{c_2}$ and $k_2 = \frac{1}{c_1}$, such that $0 \leq k_2.f(n) \leq g(n) \leq k_1.f(n)$ for all $n \geq n_0$. □
3. Transitivity:

   $f(n) \asymp g(n)$ and $g(n) \asymp h(n) \;\Rightarrow\; f(n) \asymp h(n)$

   $f(n) \preceq g(n)$ and $g(n) \preceq h(n) \;\Rightarrow\; f(n) \preceq h(n)$

   $f(n) \prec g(n)$ and $g(n) \prec h(n) \;\Rightarrow\; f(n) \prec h(n)$

   $f(n) \succeq g(n)$ and $g(n) \succeq h(n) \;\Rightarrow\; f(n) \succeq h(n)$

   $f(n) \succ g(n)$ and $g(n) \succ h(n) \;\Rightarrow\; f(n) \succ h(n)$.
4. Transpose symmetry:

   $f(n) \succeq g(n) \;\Leftrightarrow\; g(n) \preceq f(n)$

   $f(n) \succ g(n) \;\Leftrightarrow\; g(n) \prec f(n)$.

5. $f(n) \prec g(n) \Rightarrow f(n) \preceq g(n)$
   $f(n) \preceq g(n) \not\Rightarrow f(n) \prec g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \succeq g(n)$
   $f(n) \succeq g(n) \not\Rightarrow f(n) \succ g(n)$
6. $f(n) \asymp g(n) \Rightarrow f(n) \not\prec g(n)$
   $f(n) \asymp g(n) \Rightarrow f(n) \not\succ g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\asymp g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\succ g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\succeq g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\asymp g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\prec g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\preceq g(n)$
7. $f(n) \asymp g(n) \Leftrightarrow f(n) \preceq g(n)$ and $f(n) \succeq g(n)$
8. There do not exist functions $f(n)$ and $g(n)$, such that $f(n) \prec g(n)$ and $f(n) \succ g(n)$
9. Let $f(n) = f_1(n) \pm f_2(n) \pm f_3(n) \pm \ldots \pm f_k(n)$. Let

   $$\begin{aligned} &f_1(n) \succ f_2(n) \\ &f_1(n) \succ f_3(n) \\ &\quad \ldots \\ &f_1(n) \succ f_k(n) \end{aligned}$$

   Then $f(n) \asymp f_1(n)$.
10. Let $f(n) = f_1(n) \times f_2(n) \times \ldots \times f_k(n)$. Let some of the $f_i(n)$ functions be *positive* constants. Say, $f_1(n) = \text{const}$, $f_2(n) = \text{const}$, $\ldots$, $f_m(n) = \text{const}$ for some m such that $1 \leq m \leq n$. Then $f(n) \asymp f_{m+1}(n) \times f_{m+2}(n) \times \ldots \times f_k(n)$.
11. The statement "$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is equal to some L such that $0 < L < \infty$" is stronger than "$f(n) \asymp g(n)$":

    $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \quad \Rightarrow \quad f(n) \asymp g(n) \tag{1.13}$$
    $$f(n) \asymp g(n) \quad \not\Rightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ exists.}$$

    To see why the second implication does not hold, suppose $f(n) = n^2$ and $g(n) = (2 + \sin(n))n^2$. Obviously $g(n)$ oscillates between n^2 and $3n^2$ and thus $f(n) \asymp g(n)$ but $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist.

Problem 1 ([CLR00], pp. 24–25)**.** *Let* $f(n) = \frac{1}{2}n^2 - 3n$. *Prove that* $f(n) \asymp n^2$.

Solution:
For a complete solution we have to show some concrete positive constants c_1 and c_2 and a concrete value n_0 for the variable, such that for all $n \geq n_0$,

$$0 \leq c_1.n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2.n^2$$

Since $n > 0$ this is equivalent to (divide by n^2):

$$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

What we have here are in fact three inequalities:

$$0 \leq c_1 \tag{1.14}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \tag{1.15}$$

$$\frac{1}{2} - \frac{3}{n} \leq c_2 \tag{1.16}$$

(1.14) is trivial, any $c_1 > 0$ will do. To satisfy (1.16) we can pick $n_0' = 1$ and then any positive c_2 will do; say, $c_2 = 1$. The smallest integer value for n that makes the right-hand side of (1.15) positive is 7; the right-hand side becomes $\frac{1}{2} - \frac{3}{7} = \frac{7}{14} - \frac{6}{14} = \frac{1}{14}$. So, to saisfy (1.15) we pick $c_1 = \frac{1}{14}$ and $n_0'' = 7$. The overall n_0 is $n_0 = \max\{n_0', n_0''\} = 7$. The solution $n_0 = 7$, $c_1 = \frac{1}{14}$, $c_2 = 1$ is obviously not unique. □

Problem 2. *Is it true that* $\frac{1}{1000}n^3 \preceq 1000n^2$*?*

Solution:
No. Assume the opposite. Then $\exists c > 0$ and $\exists n_0$, such that for all $n \geq n_0$:

$$\frac{1}{1000}n^3 \leq c.1000n^2$$

It follows that $\forall n \geq n_0$:

$$\frac{1}{1000}n \leq 1000.c \Leftrightarrow n \leq 1000000.c$$

That is clearly false. □

Problem 3. *Is it true that for any two functions, at least one of the five relations* $\asymp$*,* $\preceq$*,* $\prec$*,* $\succeq$*, and* $\succ$ *holds between them?*

Solution:
No. Proof by demonstrating an counterexample ([CLR00, pp. 31]): let $f(n) = n$ and $g(n) = n^{1+\sin n}$. Since $g(n)$ oscillates between $n^0 = 1$ and n^2, it cannot be the case that $f(n) \asymp g(n)$ nor $f(n) \preceq g(n)$ nor $f(n) \prec g(n)$ nor $f(n) \succeq g(n)$ nor $f(n) \succ g(n)$.

However, this argument from [CLR00] holds only when $n \in \mathbb{R}^+$. If $n \in \mathbb{N}^+$, we cannot use the function $g(n)$ directly, *i.e.* without proving additional stuff. Note that $\sin n$ reaches its extreme values -1 and 1 at $2k\pi + \frac{3\pi}{2}$ and $2k\pi + \frac{\pi}{2}$, respectively, for integer k. As these are irrational numbers, the integer n cannot be equal to any of them. So, it is no longer true that $g(n)$ oscillates between $n^0 = 1$ and n^2. If we insist on using $g(n)$ in our counterexample we have to argue, for instance, that:

- for infinitely many (positive) values of the integer variable, for some constant $\epsilon > 0$, it is the case that $g(n) \geq n^{1+\epsilon}$;
- for infinitely many (positive) values of the integer variable, for some constant $\sigma > 0$, it is the case that $g(n) \leq n^{1-\sigma}$.

An alternative is to use the function $g'(n) = n^{1+\sin(\pi n + \pi/2)}$ that indeed oscillates between $n^0 = 1$ and n^2 for integer n. Another alternative is to use

$$g''(n) = \begin{cases} n^2, & \text{if } n \text{ is even,} \\ 1, & \text{else.} \end{cases}$$

□

Problem 4. *Let* $p(n)$ *be any univariate polynomial of degree* k*, such that the coefficient in the higherst degree term is positive. Prove that* $p(n) \asymp n^k$.

Solution:
$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ with $a_k > 0$. We have to prove that there exist positive constants c_1 and c_2 and some n_0 such that for all $n \geq n_0$, $0 \leq c_1 n^k \leq p(n) \leq c_2 n^k$. Since the leftmost inequality is obvious, we have to prove that

$$c_1 n^k \leq a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} \ldots + a_1 n + a_0 \leq c_2 n^k$$

For positive n we can divide by n^k, obtaining:

$$c_1 \leq a_k + \underbrace{\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \ldots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}}_{T} \leq c_2$$

Now it is obvious that any c_1 and c_2 such that $0 < c_1 < a_k$ and $c_2 > a_k$ are suitable because $\lim_{n\to\infty} T = 0$.

□

Problem 5. *Let* $a \in \mathbb{R}$ *and* $b \in \mathbb{R}^+$*. Prove that* $(n + a)^b \asymp n^b$

Solution:
Note that this problem does not reduce to Problem 4 except in the special case when b is integer. We start with the following trivial observations:

$n + a \leq n + |a| \leq 2n$, provided that $n \geq |a|$
$n + a \geq n - |a| \geq \frac{n}{2}$, provided that $\frac{n}{2} \geq |a|$, that is, $n \geq 2|a|$

It follows that:

$$\frac{1}{2} n \leq n + a \leq 2n, \text{ if } n \geq 2|a|$$

By raising to the b^{th} power we obtain:

$$\left(\frac{1}{2}\right)^b n^b \leq (n + a)^b \leq 2^b n^b$$

So we have a proof with $c_1 = \left(\frac{1}{2}\right)^b$, $c_2 = 2^b$, and $n_0 = \lceil 2|a| \rceil$.

Alternatively, solve this problem trivially using Problem 6. □

Problem 6. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$ *and any constant* $k \in \mathbb{R}^+$,

$$f(n) \asymp g(n) \Leftrightarrow (f(n))^k \asymp (g(n))^k$$

Solution:
In one direction, assume

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for some positive constants c_1 and c_2 and for all $n \geq n_0$ for some $n_0 > 0$. Raise the three inequalities to the k-th power (recall that k is positive) to obtain

$$0 \leq c_1^k (g(n))^k \leq (f(n))^k \leq c_2^k (g(n))^k, \text{ for all } n \geq n_0$$

Conclude that $(f(n))^k \asymp (g(n))^k$ since c_1^k and c_2^k are positive constants.
In the other direction the proof is virtually the same, only raise to power $\frac{1}{k}$. □

Problem 7. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$*, it is the case that* $\max(f(n), g(n)) \asymp f(n) + g(n)$.

Solution:
We are asked to prove there exist positive constants c_1 and c_2 and a certain n_0, such that for all $n \geq n_0$:

$$0 \leq c_1 (f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2 (f(n) + g(n))$$

As $f(n)$ and $g(n)$ are asymptotically positive,

$$\exists n_0' : \forall n \geq n_0', f(n) > 0$$
$$\exists n_0'' : \forall n \geq n_0'', g(n) > 0$$

Let $n_0''' = \max\{n_0', n_0''\}$. Obviously,

$$0 \leq c_1 (f(n) + g(n)) \text{ for } n \geq n_0''', \text{ if } c_1 > 0$$

It is also obvious that when $n \geq n_0'''$:

$$\frac{1}{2} f(n) + \frac{1}{2} g(n) \leq \max(f(n), g(n))$$
$$f(n) + g(n) \geq \max(f(n), g(n)),$$

which we can write as:

$$\frac{1}{2} (f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

So we have a proof with $n_0 = n_0'''$, $c_1 = \frac{1}{2}$, and $c_2 = 1$. □

Problem 8. *Prove or disprove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$ *such that* $f(n) - g(n)$ *is asymptotically positive, it is the case that* $\max(f(n), g(n)) \asymp f(n) - g(n)$.

Solution:
The claim is false. As a counterexample consider $f(n) = n^3 + n^2$ and $g(n) = n^3 + n$. In this case, $\max(f(n), g(n)) = n^3 + n^2 = f(n)$ for all sufficiently large n. Clearly, $f(n) - g(n) = n^2 - n$ which is asymptotically positive but $n^3 + n^2 \not\asymp n^2 - n$. □

Problem 9. *Which of the following are true:*

$$2^{n+1} \asymp 2^n$$
$$2^{2n} \asymp 2^n$$

Solution:
$2^{n+1} \asymp 2^n$ is true because $2^{n+1} = 2.2^n$ and for any constant c, $c.2^n \asymp 2^n$. On the other hand, $2^{2n} \asymp 2^n$ is not true. Assume the opposite. Then, having in mind that $2^{2n} = 2^n.2^n$, it is the case that for some constant c_2 and all $n \to +\infty$:

$$2^n.2^n \leq c_2.2^n \quad \Leftrightarrow \quad 2^n \leq c_2$$

That is clearly false. □

Problem 10. *Which of the following are true:*

$$\frac{1}{n^2} \prec \frac{1}{n} \tag{1.17}$$
$$2^{\frac{1}{n^2}} \prec 2^{\frac{1}{n}} \tag{1.18}$$

Solution:
(1.17) is true because

$$0 \leq \frac{1}{n^2} < c.\frac{1}{n} \quad \Leftrightarrow \quad 0 \leq \frac{1}{n} < c$$

is true for every positive constant c and sufficiently large n. (1.18), however, is not true. Assume the opposite. Then:

$$\forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq 2^{\frac{1}{n^2}} < c.2^{\frac{1}{n}} \quad \Leftrightarrow \quad 0 \leq \frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}} < c \tag{1.19}$$

But

$$\lim_{n\to\infty}\left(\frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}}\right) = \lim_{n\to\infty}\left(2^{\frac{1}{n^2}-\frac{1}{n}}\right) = 1 \text{ because} \tag{1.20}$$

$$\lim_{n\to\infty}\left(\frac{1}{n^2} - \frac{1}{n}\right) = \lim_{n\to\infty}\left(\frac{1-n}{n^2}\right) = \lim_{n\to\infty}\left(\frac{\frac{1}{n}-1}{n}\right) = 0 \tag{1.21}$$

It follows that (1.19) is false. □

Problem 11. *Which of the following are true:*

$$\frac{1}{n} \prec 1 - \frac{1}{n} \tag{1.22}$$

$$2^{\frac{1}{n}} \prec 2^{1-\frac{1}{n}} \tag{1.23}$$

Solution:
(1.22) is true because

$$\lim_{n\to\infty} \left(\frac{\frac{1}{n}}{1-\frac{1}{n}} \right) = \lim_{n\to\infty} \left(\frac{\frac{1}{\not{n}}}{\frac{n-1}{\not{n}}} \right) = \lim_{n\to\infty} \frac{1}{n-1} = 0 \tag{1.24}$$

(1.23) is false because

$$\lim_{n\to\infty} \left(\frac{2^{\frac{1}{n}}}{2^{1-\frac{1}{n}}} \right) = \lim_{n\to\infty} \left(\frac{2^{\frac{2}{n}}}{2^1} \right) = \text{const} \tag{1.25}$$

Problem 12. *Let* a *be a constant such that* $a > 1$*. Which of the following are true:*

$$f(n) \asymp g(n) \Rightarrow a^{f(n)} \asymp a^{g(n)} \tag{1.26}$$

$$f(n) \preceq g(n) \Rightarrow a^{f(n)} \preceq a^{g(n)} \tag{1.27}$$

$$f(n) \prec g(n) \Rightarrow a^{f(n)} \prec a^{g(n)} \tag{1.28}$$

for all asymptotically positive functions $f(n)$ *and* $g(n)$*.*

Solution:
(1.26) is not true – Problem 9 provides a counterexample since $2n \asymp n$ and $2^{2n} \not\asymp 2^n$. The same counterexample suffices to prove that (1.27) is not true – note that $2n \preceq n$ but $2^{2n} \not\preceq 2^n$.

Now consider (1.28).

case 1, $g(n)$ **is increasing and unbounded:** The statement is true. We have to prove that

$$\forall c > 0, \exists n' : \forall n \geq n',\ 0 \leq a^{f(n)} < c.a^{g(n)} \tag{1.29}$$

Since the constant c is positive, we are allowed to consider its logarithm to base a, namely $k = \log_a c$. So, $c = a^k$. Of course, k can be positive or negative or zero. We can rewrite (1.29) as

$$\forall k, \exists n' : \forall n \geq n',\ 0 \leq a^{f(n)} < a^k a^{g(n)} \tag{1.30}$$

Taking logarithm to base a of the two inequalities, we have

$$\forall k, \exists n' : \forall n \geq n',\ 0 \leq f(n) < k + g(n) \tag{1.31}$$

If we prove (1.31), we are done. By definition ((1.4) on page 1), the premise is

$$\forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) < c.g(n)$$

Since that holds for any $c > 0$, in particular it holds for $c = \frac{1}{2}$. So, we have

$$\exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) < \frac{g(n)}{2} \tag{1.32}$$

But $g(n)$ is increasing and unbounded. Therefore,

$$\forall k, \exists n_1 : \forall n \geq n_1,\ 0 < k + \frac{g(n)}{2} \tag{1.33}$$

We can rewrite (1.33) as

$$\forall k, \exists n_1 : \forall n \geq n_1,\ \frac{g(n)}{2} < k + g(n) \tag{1.34}$$

From (1.32) and (1.34) we have

$$\forall k, \exists n'' : \forall n \geq n'',\ 0 \leq f(n) < k + g(n) \tag{1.35}$$

Since (1.35) and (1.31) are the same, the proof is completed.

case 2, $g(n)$ is increasing but bounded: In this case (1.28) is not true. Consider Problem 11. As it is shown there, $\frac{1}{n} \prec 1 - \frac{1}{n}$ but $2^{\frac{1}{n}} \not\prec 2^{1-\frac{1}{n}}$.

case 3, $g(n)$ is not increasing: In this case (1.28) is not true. Consider Problem 10. As it is shown there, $\frac{1}{n^2} \prec \frac{1}{n}$ but $2^{\frac{1}{n^2}} \not\prec 2^{\frac{1}{n}}$. □

Problem 13. *Let a be a constant such that $a > 1$. Which of the following are true:*

$$a^{f(n)} \asymp a^{g(n)} \Rightarrow f(n) \asymp g(n) \tag{1.36}$$

$$a^{f(n)} \preceq a^{g(n)} \Rightarrow f(n) \preceq g(n) \tag{1.37}$$

$$a^{f(n)} \prec a^{g(n)} \Rightarrow f(n) \prec g(n) \tag{1.38}$$

for all asymptotically positive functions $f(n)$ and $g(n)$.

Solution:
(1.36) is true, if $g(n)$ is increasing and unbounded. Suppose there exist positive constants c_1 and c_2 and some n_0 such that

$$0 \leq c_1.a^{g(n)} \leq a^{f(n)} \leq c_2.a^{g(n)}, \forall n \geq n_0$$

Since $a > 1$ and $f(n)$ and $g(n)$ are asymptotically positive, for all sufficiently large n, the exponents have strictly larger than one values. Therefore, we can take logarithm to base a (ignoring the leftmost inequality) to obtain:

$$\log_a c_1 + g(n) \leq f(n) \leq \log_a c_2 + g(n)$$

First note that, provided that $g(n)$ is increasing and unbounded, for any constant k_1 such that $0 < k_1 < 1$, $k_1.g(n) \leq \log_a c_1 + g(n)$ for all sufficiently large n, regardless of whether the logarithm is positive or negative or zero. Then note that, provided that $g(n)$ is increasing and unbounded, for any constant k_2 such that $k_2 > 1$, $\log_a c_2 + g(n) \leq k_2.g(n)$ for all sufficiently large n, regardless of whether the logarithm is positive or negative or zero. Conclude there exists n_1, such that

$$k_1.g(n) \leq f(n) \leq k_2.g(n),\ \forall n \geq n_1$$

However, if $g(n)$ is increasing but bounded, (1.36) is not true. We already showed $2^{\frac{1}{n}} \asymp 2^{1-\frac{1}{n}}$ (see 1.25). However, since $\lim_{n\to\infty}\left(\frac{\frac{1}{n}}{1-\frac{1}{n}}\right) = 0$ (see (1.24)), it is the case that $\frac{1}{n} \prec 1 - \frac{1}{n}$ according to (1.6).
Furthermore, if $g(n)$ is not increasing, (1.36) is not true. We already showed (see (1.20)) that $\lim_{n\to\infty}\left(\frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}}\right) = 1$. According to (1.13), it is the case that $2^{\frac{1}{n^2}} \asymp 2^{\frac{1}{n}}$. However, $\frac{1}{n^2} \not\asymp \frac{1}{n}$ (see (1.21)).

Consider (1.37). If $g(n)$ is increasing and unbounded, it is true. The proof can be done easily as in the case with (1.36). If $g(n)$ is increasing but bounded, the statement is false. Let $g(n) = \frac{1}{n}$. As shown in Problem 11, $2^{1-\frac{1}{n}} \asymp 2^{\frac{1}{n}}$, therefore $2^{1-\frac{1}{n}} \preceq 2^{\frac{1}{n}}$, but $\frac{1}{n} \prec 1 - \frac{1}{n}$, therefore $1 - \frac{1}{n} \not\preceq \frac{1}{n}$. Suppose $g(n)$ is not increasing. Let $g(n) = \frac{1}{n}$. We know that $2^{\frac{1}{n^2}} \preceq 2^{\frac{1}{n}}$ but $\frac{1}{n^2} \not\asymp \frac{1}{n}$.

Now consider (1.38). It is not true. As a counterexample, consider that $2^n \prec 2^{2n}$ but $n \not\prec 2n$. □

Problem 14. *Let a be a constant such that $a > 1$. Which of the following are true:*

$$\log_a \phi(n) \asymp \log_a \psi(n) \Rightarrow \phi(n) \asymp \psi(n) \quad (1.39)$$
$$\log_a \phi(n) \preceq \log_a \psi(n) \Rightarrow \phi(n) \preceq \psi(n) \quad (1.40)$$
$$\log_a \phi(n) \prec \log_a \psi(n) \Rightarrow \phi(n) \prec \psi(n) \quad (1.41)$$
$$\phi(n) \asymp \psi(n) \Rightarrow \log_a \phi(n) \asymp \log_a \psi(n) \quad (1.42)$$
$$\phi(n) \preceq \psi(n) \Rightarrow \log_a \phi(n) \preceq \log_a \psi(n) \quad (1.43)$$
$$\phi(n) \prec \psi(n) \Rightarrow \log_a \phi(n) \prec \log_a \psi(n) \quad (1.44)$$

for all asymptotically positive functions $\phi(n)$ and $\psi(n)$.

Solution:
Let $\phi(n) = a^{f(n)}$ and $\psi(n) = a^{g(n)}$, which means that $\log_a \phi(n) = f(n)$ and $\log_a \psi(n) = g(n)$. Consider (1.26) and conclude that (1.39) is not true. Consider (1.36) and conclude that (1.42) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.27) and conlude that (1.40) is not true. Consider (1.37) and conclude that (1.43) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.28) and conclude that (1.41) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.38) and conlude that (1.44) is not true. □

Problem 15. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$, $f(n) \asymp g(n)$ *iff* $f(n) \preceq g(n)$ *and* $f(n) \succeq g(n)$.

Solution:
In one direction, assume that $f(n) \asymp g(n)$. Then there exist positive constants c_1 and c_2 and some n_0, such that:

$$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n),\ \forall n \geq n_0$$

It follows that,

$$0 \leq c_1.g(n) \leq f(n),\ \forall n \geq n_0 \tag{1.45}$$
$$0 \leq f(n) \leq c_2.g(n),\ \forall n \geq n_0 \tag{1.46}$$

In the other direction, assume that $f(n) \preceq g(n)$ and $f(n) \succeq g(n)$. Then there exists a positive constant c' and some n_0', such that:

$$0 \leq f(n) \leq c'.g(n), \forall n \geq n_0'$$

and there exists a positive constant c'' and some n_0'', such that:

$$0 \leq c''.g(n) \leq f(n), \forall n \geq n_0''$$

It follows that:

$$0 \leq c'.g(n) \leq f(n) \leq c''.g(n), \forall n \geq \max\{n_0', n_0''\}$$

□

Lemma 1 (Stirling's approximation)**.**

$$n! = \sqrt{2\pi n}\,\frac{n^n}{e^n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) \tag{1.47}$$

□

Here, $\Theta\left(\frac{1}{n}\right)$ means any function that is in the set $\Theta\left(\frac{1}{n}\right)$. A derivation of that formula—without specifying explicitly the $\sqrt{2\pi}$ factor—is found in Problem 143 on page 233.

Problem 16. *Prove that*

$$\lg n! \asymp n \lg n \tag{1.48}$$

Solution:
Use Stirling's approximation, ignoring the $\left(1 + \Theta\left(\frac{1}{n}\right)\right)$ factor, and take logarithm of both sides to obtain:

$$\lg(n!) = \lg(\sqrt{2\pi}) + \lg n + n \lg n - n \lg e$$

By Property 9 of the relations, $\lg(\sqrt{2\pi}) + \lg n + n \lg n - n \lg e \asymp n \lg n$. □

Problem 17. *Prove that for any constant* $a > 1$,

$$a^n \prec n! \prec n^n \tag{1.49}$$

Solution:
Because of the factorial let us restrict n to positive integers.

$$\lim_{n\to\infty}\left(\frac{n.(n-1).(n-2)\ldots 2.1}{\underbrace{a\,.\,a\,.\,a\,\ldots\,a\,.\,a}_{n \text{ times}}}\right) = \infty$$

$$\lim_{n\to\infty}\left(\frac{n.(n-1).(n-2)\ldots 2.1}{\underbrace{n\,.\,n\,.\,n\,\ldots\,n\,.\,n}_{n \text{ times}}}\right) = 0$$

□

Problem 18 (polylogarithm versus constant power of n)**.** *Let* a, k *and* ϵ *be any constants, such that* $k > 0$, $a > 1$, *and* $\epsilon > 0$. *Prove that:*

$$(\log_a n)^k \prec n^\epsilon \tag{1.50}$$

Solution:

$$\lim_{n\to\infty}\frac{n^\epsilon}{(\log_a n)^k} = \qquad \text{let } b \leftarrow \frac{\epsilon}{k}$$

$$\lim_{n\to\infty}\frac{(n^b)^k}{(\log_a n)^k} =$$

$$\lim_{n\to\infty}\left(\frac{n^b}{\log_a n}\right)^k = \qquad k \text{ is positive}$$

$$\lim_{n\to\infty}\frac{n^b}{\log_a n} = \qquad \text{use l'Hôpital's rule}$$

$$\lim_{n\to\infty}\frac{bn^{b-1}}{\left(\frac{1}{\ln a}\right)\left(\frac{1}{n}\right)} =$$

$$\lim_{n\to\infty}(\ln a)b\,n^b = \infty$$

□

Problem 19 (constant power of n versus exponent)**.** *Let* a *and* ϵ *be any constants, such that* $a > 1$ *and* $\epsilon > 0$. *Prove that:*

$$n^\epsilon \prec a^n \tag{1.51}$$

Solution:
Take $\log_a$ of both sides. The left-hand side yields $\epsilon.\log_a n$ and the right-hand side yields n. But $\epsilon.\log_a n \prec n$ because of Problem 18. Conclude immediately the desired relation holds. □

Definition 1 (log-star function, [CLR00], pp. 36)**.** *Let the function* $\lg^{(i)} n$ *be defined recursively for nonnegative integers* i *as follows:*

$$\lg^{(i)} n = \begin{cases} n, & \text{if } i = 0 \\ \lg\left(\lg^{(i-1)} n\right), & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0 \\ \text{undefined}, & \text{if } i > 0 \text{ and } \lg^{(i-1)} n < 0 \text{ or } \lg^{(i-1)} n \text{ is undefined} \end{cases}$$

Then

$$\lg^* n = \min\left\{i \geq 0 \,|\, \lg^{(i)} n \leq 1\right\}$$

□

According to this definition,

$$\begin{aligned} \lg^* 2 &= 1, \text{ since } \lg^{(0)} 2 = 2 \text{ and } \lg^{(1)} 2 = \lg\left(\lg^{(0)} 2\right) = \lg(2) = 1 \\ \lg^* 3 &= 2, \text{ since } \lg^{(0)} 3 = 3 \text{ and } \lg\left(\lg^{(0)} 3\right) = \lg(\lg 3) = 0.6644\ldots \\ \lg^* 4 &= 2 \\ \lg^* 5 &= 3 \\ &\ldots \\ \lg^* 16 &= 3 \\ \lg^* 17 &= 4 \\ &\ldots \\ \lg^* 65536 &= 4 \\ \lg^* 65537 &= 5 \\ &\ldots \\ \lg^* 2^{65536} &= 5 \\ \lg^*\left(2^{65536} + 1\right) &= 6 \\ &\ldots \end{aligned}$$

Obviously, every real number t can be represented by a *tower of twos*:

$$t = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^s}}}}}}$$

where s is a real number such that $1 < s \leq 2$. The *height of the tower* is the number of elements in this sequence. For instance,

number	*its tower of twos*	*the height of the tower*
2	2	1
3	$2^{1.5849625007\ldots}$	2
4	2^2	2
5	$2^{2^{1.2153232957\ldots}}$	3
16	2^{2^2}	3
17	$2^{2^{2^{1.0223362884\ldots}}}$	4
65536	$2^{2^{2^2}}$	4
65537	$2^{2^{2^{2^{1.0000051642167\ldots}}}}$	5

Having that in mind, it is trivial to see that $\lg^* n$ is the height of the tower of twos of n.

Problem 20 ([CLR00], problem 2-3, pp. 38–39)**.** *Rank the following thirty functions by order of growth. That is, find the equivalence classes of the "$\asymp$" relation and show their order by "$\succ$".*

$\lg(\lg^* n)$	$2^{\lg^* n}$	$\left(\sqrt{2}\right)^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln\ln n$	$\lg^* n$	$n.2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

Solution:

$2^{2^{n+1}} \succ 2^{2^n}$ because $2^{2^{n+1}} = 2^{2.2^n} = 2^{2^n} \times 2^{2^n}$.

$2^{2^n} \succ (n+1)!$ To see why, take logarithm to base two of both sides. The left-hand side becomes 2^n, the right-hand side becomes $\lg((n+1)!)$ By (1.47), $\lg((n+1)!) \asymp (n+1)\lg(n+1)$, and clearly $(n+1)\lg(n+1) \asymp n\lg n$. As $2^n \succ n\lg n$, by (1.41) we have $2^{2^n} \succ (n+1)!$

$(n+1)! \succ n!$ because $(n+1)! = (n+1) \times n!$

$n! \succ e^n$ by (1.49).

$e^n \succ n.2^n$. To see why, consider:

$$\lim_{n\to\infty} \frac{n.2^n}{e^n} = \lim_{n\to\infty} \frac{n}{\frac{e^n}{2^n}} = \lim_{n\to\infty} \frac{n}{\left(\frac{e}{2}\right)^n} = 0$$

$n.2^n \succ 2^n$

$2^n \succ \left(\frac{3}{2}\right)^n$. To see why, consider:

$$\lim_{n\to\infty} \frac{\left(\frac{3}{2}\right)^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{4}\right)^n = 0$$

$\left(\frac{3}{2}\right)^n \succ n^{\lg(\lg n)}$. To see why, take lg of both sides. The left-hand side becomes $n.\lg\left(\frac{3}{2}\right)$, the right-hand side becomes $\lg n.\lg(\lg n)$. Clearly, $\lg^2 n \succ \lg n.\lg(\lg n)$ and $n \succ \lg^2 n$ by (1.50). By transitivity, $n \succ \lg n.\lg(\lg n)$, and so $n.\lg\left(\frac{3}{2}\right) \succ \lg n.\lg(\lg n)$. Apply (1.41) and the desired conclusion follows.

$(\lg n)^{\lg n} = n^{\lg(\lg n)}$, which is obvious if we take lg of both sides. So, $(\lg n)^{\lg n} \asymp n^{\lg(\lg n)}$.

$(\lg n)^{\lg n} \succ (\lg n)!$ To see why, substitute $\lg n$ with m, obtaining $m^m \succ m!$ and apply (1.49).

$(\lg n)! \succ n^3$. Take lg of both sides. The left-hand side becomes $\lg((\lg n)!)$. Substitute $\lg n$ with m, obtaining $\lg(m!)$. By (1.48), $\lg(m!) \asymp m\lg m$, therefore $\lg((\lg n)!) \asymp (\lg n).(\lg(\lg n))$. The right-hand side becomes $3.\lg n$. Compare $(\lg n).(\lg(\lg n))$ with $3.\lg n$:

$$\lim_{n\to\infty} \frac{3.\lg n}{(\lg n).(\lg(\lg n))} = \lim_{n\to\infty} \frac{3}{\lg(\lg n)} = 0$$

It follows that $(\lg n).(\lg(\lg n)) \succ 3.\lg n$. Apply (1.41) to draw the desired conclusion.

$n^3 \succ n^2$.

$n^2 \asymp 4^{\lg n}$ because $4^{\lg n} = 2^{2\lg n} = 2^{\lg n^2} = n^2$ by the properties of the logarithm.

$n^2 \succ n\lg n$.

$\lg n! \asymp n\lg n$ (see (1.48)).

$n\lg n \succ n$.

$n \asymp 2^{\lg n}$ because $n = 2^{\lg n}$ by the properties of the logarithm.

$n \succ (\sqrt{2})^{\lg n}$ because $(\sqrt{2})^{\lg n} = 2^{\frac{1}{2}\lg n} = 2^{\lg\sqrt{n}} = \sqrt{n}$ and clearly $n \succ \sqrt{n}$.

$(\sqrt{2})^{\lg n} \succ 2^{\sqrt{2\lg n}}$. To see why, note that $\lg n \succ \sqrt{\lg n}$, therefore $\frac{1}{2}.\lg n \succ \sqrt{2}.\sqrt{\lg n} = \sqrt{2\lg n}$. Apply (1.28) and conclude that $2^{\frac{1}{2}\cdot\lg n} \succ 2^{\sqrt{2\lg n}}$, *i.e.* $(\sqrt{2})^{\lg n} \succ 2^{\sqrt{2\lg n}}$.

$2^{\sqrt{2\lg n}} \succ \lg^2 n$. To see why, take lg of both sides. The left-hand side becomes $\sqrt{2\lg n}$ and the right-hand side becomes $\lg(\lg^2 n) = 2.\lg(\lg n)$. Substitute $\lg n$ with m: the left-hand side becomes $\sqrt{2m} = \sqrt{2}\sqrt{m} = \sqrt{2}.m^{\frac{1}{2}}$ and the right-hand side becomes $2\lg m$. By (1.50) we know that $m^{\frac{1}{2}} \succ \lg m$, therefore $\sqrt{2}.m^{\frac{1}{2}} \succ 2\lg m$, therefore $\sqrt{2m} \succ 2\lg m$, therefore $\sqrt{2\lg n} \succ \lg(\lg^2 n)$. Having in mind (1.41) we draw the desired conclusion.

$\lg^2 n \succ \ln n$. To see this is true, observe that $\ln n = \frac{\lg n}{\lg e}$.

$\ln n \succ \sqrt{\lg n}$.

$\sqrt{\lg n} \succ \ln \ln n$. The left-hand side is $\sqrt{\frac{\ln n}{\ln 2}}$. Substitute $\ln n$ with m and the claim becomes $\frac{1}{\sqrt{\ln 2}} . \sqrt{m} \succ \ln m$, which follows from (1.50).

$\ln \ln n \succ 2^{\lg^* n}$. To see why this is true, note that $\ln \ln n \asymp \lg \lg n$ and rewrite the claim as $\lg \lg n \succ 2^{\lg^* n}$. Take lg of both sides. The left-hand side becomes $\lg \lg \lg n$, *i.e.* a triple logarithm. The right-hand side becomes $\lg^* n$. If we think of n as a tower of twos, it is obvious that the triple logarithm decreases the height of the tower with three, while, as we said before, the log-star measures the height of the tower. Clearly, the latter is *much* smaller than the former.

$2^{\lg^* n} \succ \lg^* n$. Clearly, for any increasing function $f(n)$, $2^{f(n)} \succ f(n)$.

$\lg^* n \asymp \lg^* (\lg n)$. Think of n as a tower of twos and note that the difference in the height of n and $\lg n$ is one. Therefore, $\lg^* (\lg n) = (\lg^* n) - 1$.

$\lg^* n \succ \lg (\lg^* n)$. Substitute $\lg^* n$ with $f(n)$ and the claim becomes $f(n) \succ \lg f(n)$ which is clearly true since $f(n)$ is increasing.

$\lg (\lg^* n) \succ 1$.

$1 \asymp n^{\frac{1}{\lg n}}$. Note that $n^{\frac{1}{\lg n}} = 2$: take lg of both sides, the left-hand side becomes $\lg \left(n^{\frac{1}{\lg n}} \right) = \frac{1}{\lg n} . \lg n = 1$ and the right-hand side becomes $\lg 2 = 1$. □

Problem 21. *Give an example of a function* $f(n)$*, such that for any function* $g(n)$ *among the thirty functions from Problem 20,* $f(n) \not\preceq g(n)$ *and* $f(n) \not\succeq g(n)$.

Solution:
For instance,

$$f(n) = \begin{cases} 2^{2^{n+2}}, & \text{if } n \text{ is even} \\ \frac{1}{n}, & \text{if } n \text{ is odd} \end{cases}$$

□

Problem 22. *Is it true that for any asymptotically positive functions* $f(n)$ *and* $g(n)$, $f(n) + g(n) \asymp \min (f(n), g(n))$*?*

Solution:
No. As a counterexample, consider $f(n) = n$ and $g(n) = 1$. Then $\min (f(n), g(n)) = 1$, $f(n) + g(n) = n + 1$, and certainly $n + 1 \not\asymp 1$. □

Problem 23. *Is it true that for any asymptotically positive function* $f(n)$, $f(n) \preceq (f(n))^2$*?*

Solution:
If $f(n)$ is increasing, it is trivially true. If it is decreasing, however, it may not be true: consider (1.17). □

Problem 24. *Is it true that for any asymptotically positive function* $f(n)$, $f(n) \asymp f(\frac{n}{2})$*?*

Solution:
No. As a counterexample, consider $f(n) = 2^n$. Then $f(\frac{n}{2}) = 2^{\frac{n}{2}}$. As we already saw, $2^n \not\asymp 2^{\frac{n}{2}}$. □

Problem 25. *Compare the growth of* $n^{\lg n}$ *and* $(\lg n)^n$.

Solution:
Take logarithm of both sides. The left-hand side becomes $(\lg n)(\lg n) = \lg^2 n$, the right-hand side, $n.\lg(\lg n)$. As $n.\lg(\lg n) \succ \lg^2 n$, it follows that $(\lg n)^n \succ n^{\lg n}$. □

Problem 26. *Compare the growth of* $n^{\lg\lg\lg n}$ *and* $(\lg n)!$

Solution:
Take lg of both sides. The left-hand side becomes $(\lg n).(\lg\lg\lg n)$, the right-hand side becomes $\lg((\lg n)!)$. Substitute $\lg n$ with m is the latter expression to get $\lg((m)!) \asymp m \lg m$. And that is $(\lg n).(\lg\lg n)$. Since $(\lg n).(\lg\lg n) \succ (\lg n).(\lg\lg\lg n)$, it follows that $(\lg n)! \succ n^{\lg\lg\lg n}$. □

Problem 27. *Let* $n!! = (n!)!$. *Compare the growth of* $n!!$ *and* $(n-1)!! \times ((n-1)!)^{n!}$.

Solution:
Let $(n-1)! = \nu$. Then $n! = n\nu$. We compare

$$n!! \quad \text{vs} \quad (n-1)!! \times ((n-1)!)^{n!}$$

$$(n\nu)! \quad \text{vs} \quad \nu! \times \nu^{n\nu}$$

Apply Stirling's approximation to both sides to get:

$$\sqrt{2\pi n\nu}\,\frac{(n\nu)^{n\nu}}{e^{n\nu}} \quad \text{vs} \quad \sqrt{2\pi\nu}\,\frac{\nu^\nu}{e^\nu} \times \nu^{n\nu}$$

$$\sqrt{2\pi n\nu}\,(n\nu)^{n\nu} \quad \text{vs} \quad \sqrt{2\pi\nu}\,e^{(n-1)\nu} \times \nu^\nu \times \nu^{n\nu}$$

Divide by $\sqrt{2\pi\nu}\,\nu^{n\nu}$ both sides:

$$\sqrt{n}\,n^{n\nu} \quad \text{vs} \quad e^{(n-1)\nu} \times \nu^\nu$$

Ignore the $\sqrt{n}$ factor on the left. If we derive without it that the left side grows faster, surely it grows even faster with it. So, consider:

$$n^{n\nu} \quad \text{vs} \quad e^{(n-1)\nu} \times \nu^\nu$$

Raise both sides to $\frac{1}{\nu}$:

$$n^n \quad \text{vs} \quad e^{n-1} \times \nu$$

That is,

$$n^n \quad \text{vs} \quad e^{n-1} \times (n-1)!$$

Apply Stirling's aproximation second time to get:

$$n^n \quad \text{vs} \quad e^{n-1} \times \sqrt{2\pi(n-1)}\,\frac{(n-1)^{n-1}}{e^{n-1}}$$

That is,

$$n^n \quad \text{vs} \quad \sqrt{2\pi(n-1)}\,(n-1)^{n-1}$$

Since $\sqrt{2\pi(n-1)}\,(n-1)^{n-1} \asymp (n-1)^{\left(n-\frac{1}{2}\right)}$, we have

$$n^n \quad \text{vs} \quad (n-1)^{\left(n-\frac{1}{2}\right)}$$

Clearly, $n^n \succ (n-1)^{\left(n-\frac{1}{2}\right)}$, therefore $n!! \succ (n-1)!! \times ((n-1)!)^{n!}$. □

Lemma 2. *The function series:*

$$S(x) = \frac{\ln x}{x} + \frac{\ln^2 x}{x^2} + \frac{\ln^3 x}{x^3} + \dots$$

is convergent for $x > 1$. *Furthermore,* $\lim_{x\to\infty} S(x) = 0$.

Proof:
It is well known that the series

$$S'(x) = \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \dots$$

called *geometric series* is convergent for $x > 1$ and $S'(x) = \frac{1}{x-1}$ when $x > 1$. Clearly, $\lim_{x\to\infty} S'(x) = 0$. Consider the series

$$S''(x) = \frac{1}{\sqrt{x}} + \frac{1}{(\sqrt{x})^2} + \frac{1}{(\sqrt{x})^3} + \dots \tag{1.52}$$

It is a geometric series and is convergent for $\sqrt{x} > 1$, *i.e.* $x > 1$, and $\lim_{x\to\infty} S''(x) = 0$. Let us rewrite $S(x)$ as

$$S(x) = \frac{1}{\sqrt{x}.\frac{\sqrt{x}}{\ln x}} + \frac{1}{(\sqrt{x})^2.\left(\frac{\sqrt{x}}{\ln x}\right)^2} + \frac{1}{(\sqrt{x})^3.\left(\frac{\sqrt{x}}{\ln x}\right)^3} + \dots \tag{1.53}$$

For each term $f_k(x) = \frac{1}{(\sqrt{x})^k.\left(\frac{\sqrt{x}}{\ln x}\right)^k}$ of $S(x)$, $k \geq 1$, for large enough x, it is the case that $f_k(x) < g_k(x)$ where $g_k(x) = \frac{1}{(\sqrt{x})^k}$ is the k^{th} term of $S''(x)$. To see why this is true, consider (1.50). Then the fact that $S''(x)$ is convergent and $\lim_{x\to\infty} S''(x) = 0$ implies the desired conclusion. □

Problem 28 ([Knu73], pp. 107)**.** *Prove that* $\sqrt[n]{n} \asymp 1$.

Solution:
We will show an even stronger statement: $\lim_{n\to\infty} \sqrt[n]{n} = 1$. It is known that:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Note that $\sqrt[n]{n} = e^{\ln \sqrt[n]{n}} = e^{\left(\frac{\ln n}{n}\right)}$.

$$e^{\left(\frac{\ln n}{n}\right)} = 1 + \underbrace{\frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots}_{T(n)}$$

Lemma 2 implies $\lim_{n\to\infty} T(n) = 0$. It follows that $\lim_{n\to\infty} \sqrt[n]{n} = 1$. □
We can also say that $\sqrt[n]{n} = 1 + O\left(\frac{\lg n}{n}\right)$, $\sqrt[n]{n} = 1 + \frac{\lg n}{n} + O\left(\frac{\lg^2 n}{n^2}\right)$, *etc*, where the big-Oh notation stands for any function of the set.

Problem 29 ([Knu73], pp. 107)**.** *Prove that* $n\left(\sqrt[n]{n} - 1\right) \asymp \ln n$.

Solution:
As

$$\sqrt[n]{n} = 1 + \frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots$$

it is the case that:

$$\sqrt[n]{n} - 1 = \frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots$$

Multiply by n to get:

$$n\left(\sqrt[n]{n} - 1\right) = \ln n + \underbrace{\frac{(\ln n)^2}{2!n} + \frac{(\ln n)^3}{3!n^2} + \dots}_{T(n)}$$

Note that $\lim_{n\to\infty} T(n) = 0$ by an obvious generalisation of Lemma 2. The claim follows immediately. □

Problem 30. *Compare the growth of* n^n, $(n+1)^n$, n^{n+1}, *and* $(n+1)^{n+1}$.

Solution:
$n^n \asymp (n+1)^n$ because

$$\lim_{n\to\infty} \frac{(n+1)^n}{n^n} = \lim_{n\to\infty} \left(\frac{n+1}{n}\right)^n = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$$

Clearly, $n^n \prec n^{(n+1)} = n.n^n$. And $n^{(n+1)} \asymp (n+1)^{(n+1)}$:

$$\lim_{n\to\infty} \frac{(n+1)^{n+1}}{n^{n+1}} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^{n+1} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \lim_{n\to\infty} \left(1 + \frac{1}{n}\right) = e.1 = e$$

□

Problem 31. *Let* k *be a constant such that* $k > 1$*. Prove that*

$$1 + k + k^2 + k^3 + \ldots + k^n = \Theta(k^n)$$

Solution:
First assume n is an integer variable. Then

$$1 + k + k^2 + k^3 + \ldots + k^n = \frac{k^{n+1} - 1}{k - 1} = \Theta(k^n)$$

The result can obviously be extended for real n, provided we define appropriately the sum. For instance, if $n \in \mathbb{R}^+ \setminus \mathbb{N}$ let the sum be

$$S(n) = 1 + k + k^2 + k^3 + \ldots + k^{\lfloor n-1 \rfloor} + k^{\lfloor n \rfloor} + k^n$$

By the above result, $S(n) = k^n + \Theta\left(k^{\lfloor n \rfloor}\right) = \Theta(k^n)$. □

Problem 32. *Let* k *be a constant such that* $0 < k < 1$*. Prove that*

$$1 + k + k^2 + k^3 + \ldots + k^n = \Theta(1)$$

Solution:

$$1 + k + k^2 + k^3 + \ldots + k^n < \sum_{t=0}^{\infty} k^t = \frac{1}{1 - k} = \Theta(1)$$ □

Corollary 1.

$$1 + k + k^2 + k^3 + \ldots + k^n = \begin{cases} \Theta(1), & \text{if } 0 < k < 1 \\ \Theta(n), & \text{if } k = 1 \\ \Theta(k^n), & \text{if } k > 1 \end{cases}$$ □

Problem 33. *Let* $f(x) = 2^{2^{\lfloor x \rfloor}}$ *and* $g(x) = 2^{2^{\lceil x \rceil}}$ *where* $x \in \mathbb{R}^+$*. Determine which of the following are true and which are false:*

1. $f(x) \asymp g(x)$
2. $f(x) \preceq g(x)$
3. $f(x) \prec g(x)$
4. $f(x) \succeq g(x)$
5. $f(x) \succ g(x)$

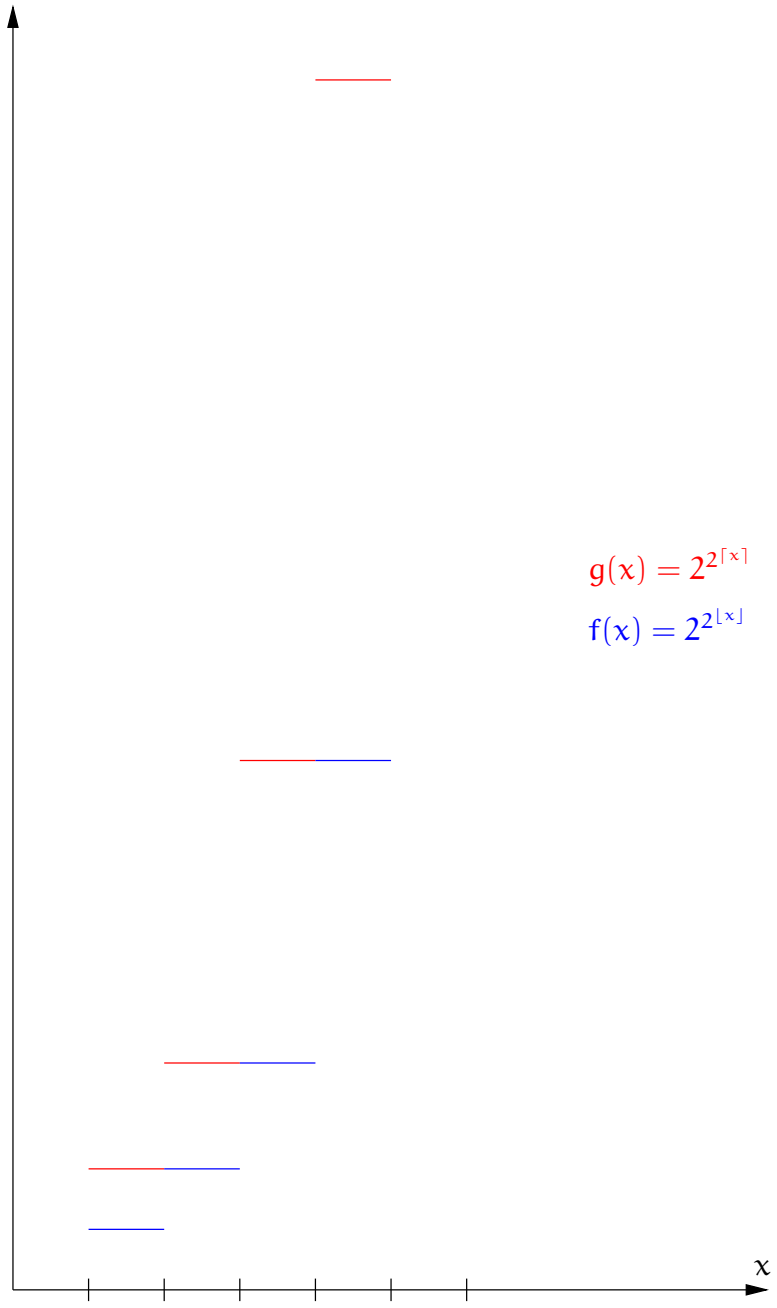


Figure 1.1: $f(x)$ and $g(x)$ from Problem 33.

Solution:
Note that $\forall x \in \mathbb{N}^+, \lfloor x \rfloor = \lceil x \rceil$, therefore $f(x) = g(x)$ whenever $x \in \mathbb{N}^+$. On the other hand, $\forall x \in \mathbb{R}^+ \setminus \mathbb{N}^+, \lceil x \rceil = \lfloor x \rfloor + 1$, therefore $g(x) = 2^{2^{\lfloor x \rfloor + 1}} = 2^{2.2^{\lfloor x \rfloor}} = \left(2^{2^{\lfloor x \rfloor}}\right)^2 = (f(x))^2$ whenever $x \in \mathbb{R}^+ \setminus \mathbb{N}^+$. Figure 1.1 illustrates the way that $f(x)$ and $g(x)$ grow.

First assume that $f(x) \prec g(x)$. By definition, for every constant $c > 0$ there exists x_0, such that $\forall x \geq x_0$, $f(x) < c.g(x)$. It follows for $c = 1$ there exists a value for the variable, say x', such that $\forall x \geq x'$, $f(x) < g(x)$. However,

$$\lceil x' \rceil \geq x'$$

Therefore,

$$f(\lceil x' \rceil) < g(\lceil x' \rceil)$$

On the other hand,

$$\lceil x' \rceil \in \mathbb{N}^+ \Rightarrow f(\lceil x' \rceil) = g(\lceil x' \rceil)$$

We derived

$$f(\lceil x' \rceil) = g(\lceil x' \rceil) \text{ and } f(\lceil x' \rceil) < g(\lceil x' \rceil) \lightning$$

We derived a contradiction, therefore

$$f(x) \not\prec g(x)$$

Analogously we prove that

$$f(x) \not\succ g(x)$$

To see that $f(x) \not\asymp g(x)$, note that $\forall \tilde{x} \in \mathbb{R}^+, \exists x'' \geq \tilde{x}$, such that $g(x'') = (f(x''))^2$. As $f(x)$ is a growing function, its square must have a higher asymptotic growth rate.

Now we prove that $f(x) \preceq g(x)$. Indeed,

$$\begin{aligned}
&\forall x \in \mathbb{R}^+, \ \lfloor x \rfloor \ \leq \lceil x \rceil \Rightarrow \\
&\forall x \in \mathbb{R}^+, \ 2^{\lfloor x \rfloor} \leq 2^{\lceil x \rceil} \Rightarrow \\
&\forall x \in \mathbb{R}^+, \ 2^{2^{\lfloor x \rfloor}} \leq 2^{2^{\lceil x \rceil}} \Rightarrow \exists c > 0, c = \text{const}, \text{ such that } \forall x \in \mathbb{R}^+, \ 2^{2^{\lfloor x \rfloor}} \leq c.2^{2^{\lceil x \rceil}}
\end{aligned}$$

Finally we prove that $f(x) \not\succeq g(x)$. Assume the opposite. Since $f(x) \preceq g(x)$, by property 7 on page 3 we derive $f(x) \asymp g(x)$ and that contradicts our result that $f(x) \not\asymp g(x)$. □

Problem 34. *Prove that* $\binom{n}{\lfloor \frac{n}{2} \rfloor} \asymp \frac{1}{\sqrt{n}} 2^n$. *You may assume* n *is even.*

Solution:
Let $m = \lfloor \frac{n}{2} \rfloor$, which is $\frac{n}{2}$ if we assume n is even. It is known that

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\left(\frac{n}{2}!\right)^2}$$

Apply Stirling's approximation (on page 11), ignoring the $\left(1+\Theta\left(\frac{1}{n}\right)\right)$ factor, on the three factorials to get

$$\frac{n!}{\left(\frac{n}{2}!\right)^2} = \frac{\sqrt{2\pi n}\,\frac{n^n}{e^n}}{\left(\sqrt{2\pi\frac{n}{2}}\,\frac{\frac{n}{2}^{\frac{n}{2}}}{e^{\frac{n}{2}}}\right)^2} = \sqrt{2\pi}\sqrt{n}\,\frac{n^n}{e^n} \times \frac{e^n}{\pi n\,\frac{n^n}{2^n}} = \sqrt{\frac{2}{\pi}} \times \frac{1}{\sqrt{n}} \times 2^n$$

□

Problem 35. *Compare the growth of* 2^{n^2+n} *and* 3^{n^2}.

Solution:

$$\lim_{n\to\infty} \frac{2^{n^2+n}}{3^{n^2}} = \lim_{n\to\infty} \frac{2^{n^2}2^n}{3^{n^2}} = \lim_{n\to\infty} \frac{2^n}{\left(\frac{3}{2}\right)^{n^2}} = 0$$

To see why the limit is 0, take lg of 2^n and $\left(\frac{3}{2}\right)^{n^2}$, namely n vs $n^2 \lg\left(\frac{3}{2}\right)$. We derived $2^{n^2+n} \prec 3^{n^2}$. □

Chapter 2

Iterative Algorithms

In this section we compute the asymptotic running time of algorithms that use the **for** and **while** statements but make no calls to other algorithms or themselves. The time complexity is expressed as a function of the size of the input, in case the input is an array or a matrix, or as a function of the upper bound of the loops. Consider the time complexity of the following trivial algorithm.

```
ADD-1(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      a ← a + i
4  return a
```

We make the folloing assumptions:

- the expression at line 3 is executed in constant time regardless of how large n is,
- the expression at line 1 is executed in constant time, and
- the loop control variable check and assignment of the **for** loop at line 2 are executed in constant time.

Since we are interested in the asymptotic running time, not in the precise one, it suffices to find the number of times the expression inside the loop (line 3 in this case) is executed as a function of the upper bound on the loop control variable n. Let that function be $f(n)$. The time complexity of ADD-1 will then be $\Theta(f(n))$. We compute $f(n)$ as follows. First we substitute the expression inside the loop with $a \leftarrow a + 1$ where a is the counter variable that is set to zero initially. Then find the value of a after the loop finishes as a function of n where n is the upper bound n of the loop control variable i. Using that approach, algorithm ADD-1 becomes ADD-1-MODIFIED as follows.

```
ADD-1-MODIFIED(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      a ← a + 1
4  return a
```

The value that ADD-1-MODIFIED outputs is $\sum_{i=1}^{n} 1 = n$, therefore its time complexity is $\Theta(n)$. Now consider another algorithm:

```
ADD-2(n: nonnegative integer)
1  return n
```

Clearly, ADD-2 is equivalent to ADD-1 but the running time of ADD-2 is, under the said assumptions, constant. We denote constant running time by $\Theta(1)$[†]. It is not incorrect to say the running time of both algorithms is $O(n)$ but the big-Theta notation is superior as it grasps precisely—in the asymptotic sense—the algorithm's running time.

Consider the following iterative algorithm:

```
ADD-3(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← 1 to n
4          a ← a + 1
5  return a
```

The value it outputs is $\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$, therefore its time complexity is $\Theta(n^2)$.

Algorithm ADD-3 has two *nested cycles*. We can generalise that the running time of k nested cycles as follows.

```
ADD-GENERALISED(n: nonnegative integer)
1  for i_1 ← 1 to n
2      for i_2 ← 1 to n
3          ...
4          for i_k ← 1 to n
5              expression
```

where **expression** is computed in $\Theta(1)$, has running time $\Theta(n^k)$.

Let us consider a modification of ADD-3:

```
ADD-4(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i to n
4          a ← a + 1
5  return a
```

[†]All constants are bit-Theta of each other so we might have as well used $\Theta(1000)$ or $\Theta(0.0001)$ but we prefer the simplest form $\Theta(1)$.

The running time is determined by the output a and that is:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i-1} 1}_{i-1}\Big) = \sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n}(n+1) - \sum_{i=1}^{n} i =$$

$$n(n+1) - \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$ (see Problem 4 on page 5.)

It follows that asymptotically ADD-4 has the same running time as ADD-3. Now consider a modification of ADD-4.

```
ADD-5(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i + 1 to n
4          a ← a + 1
5  return a
```

The running time is determined by the output a and that is:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i} 1}_{i}\Big) = \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n}(n) - \sum_{i=1}^{n} i =$$

$$n^2 - \frac{n(n+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$$

Consider the following algorithm:

```
A2(n: positive integer)
1  a ← 0
2  for i ← 1 to n − 1
3      for j ← i + 1 to n
4          for k ← 1 to j
5              a ← a + 1
6  return a
```

We are asked to determine a that A2 returns as a function of n. The answer clearly is

$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1$, we just need to find an equivalent closed form.

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n} j - \sum_{j=1}^{i} j\right) =$$
$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1) - \frac{1}{2}i(i+1)\right) =$$
$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1)\right) - \frac{1}{2}\sum_{i=1}^{n-1}(i^2+i) =$$
$$\frac{1}{2}n(n+1)(n-1) - \frac{1}{2}\sum_{i=1}^{n-1} i^2 - \frac{1}{2}\sum_{i=1}^{n-1} i$$

But $\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1)$, therefore $\sum_{i=1}^{n-1} i^2 = \frac{1}{6}(n-1)n(2n-1)$. Further, $\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$, so we have

$$\frac{1}{2}n(n-1)(n+1) - \frac{1}{12}n(n-1)(2n-1) - \frac{1}{4}n(n-1) =$$
$$\frac{1}{2}n(n-1)\left(n+1-\frac{1}{6}(2n-1)-\frac{1}{2}\right) =$$
$$\frac{1}{12}n(n-1)(6n+3-2n+1) = \frac{1}{12}n(n-1)(4n+4) =$$
$$\frac{1}{3}n(n-1)(n+1)$$

That implies that the running time of A2 is $\Theta(n^3)$. Clearly A2 is equivalent to the following algorithm.

```
A3(n: positive integer)
  1  return  n(n-1)(n+1)/3
```

whose running time is $\Theta(1)$.

```
A4(n: positive integer)
  1  a ← 0
  2  for i ← 1 to n
  3      for j ← i+1 to n
  4          for k ← i+j-1 to n
  5              a ← a+1
  6  return a
```

Problem 36. *Find the running time of algorithm* A4 *by determining the value of* a *it returns as a function of* n, f(n). *Find a closed form for* f(n).

Solution:

$$f(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=i+j-1}^{n} 1$$

Let us evaluate the innermost sum $\sum_{k=i+j-1}^{n} 1$. It is easy to see that the lower boundary $i+j-1$ may exceed the higher boundary n. If that is the case, the sum is zero because the index variable takes values from the empty set. More precisely, for any integer t,

$$\sum_{i=t}^{n} 1 = \begin{cases} n-t+1 & \text{, if } t \le n \\ 0 & \text{, else} \end{cases}$$

It follows that

$$\sum_{k=i+j-1}^{n} 1 = \begin{cases} n-i-j+2 & \text{, if } i+j-1 \le n \Leftrightarrow j \le n-i+1 \\ 0 & \text{, else} \end{cases}$$

Then

$$f(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n-i+1} (n+2-(i+j))$$

Now the innermost sum is zero when $i+1 > n-i+1 \Leftrightarrow 2i > n \Leftrightarrow i > \lfloor \frac{n}{2} \rfloor$, therefore

the maximum i we have to consider is $\lfloor \frac{n}{2} \rfloor$:

$$f(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} (n+2-(i+j)) =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} 1 - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i \left(\sum_{j=i+1}^{n-i+1} 1 \right) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} j =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-i+1-(i+1)+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-i+1-(i+1)+1) -$$

$$\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\sum_{j=1}^{n-i+1} j - \sum_{j=1}^{i} j \right) =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-2i+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-2i+1) -$$

$$\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\frac{(n-i+1)(n-i+2)}{2} - \frac{i(i+1)}{2} \right) =$$

$$(n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - 2(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i - (n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 -$$

$$\frac{1}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \Big((n+1)(n+2) - i(2n+3) + \cancel{i^2} - \cancel{i^2} - i \Big) =$$

$$(n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - (3n+5) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 -$$

$$\frac{(n+1)(n+2)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 + \frac{(2n+4)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i =$$

$$\left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) - (3n+5) \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} + 2 \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{6} -$$

$$\frac{1}{2} \left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) + (n+2) \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} =$$

$$\frac{\lfloor \frac{n}{2} \rfloor (n+1)(n+2)}{2} - \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) (2n+3)}{2} + \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{3}$$

When n is even, *i.e.* $n = 2k$ for some $k \in \mathbb{N}^+$, $\left\lfloor \frac{n}{2} \right\rfloor = k$ and so

$$f(n) = \frac{k(2k+1)(2k+2)}{2} - \frac{k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$\frac{k(k+1)(4k+2) - k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$k(k+1)\left(-\frac{1}{2} + \frac{2k+1}{3}\right) = \frac{k(k+1)(4k-1)}{6}$$

When n is odd, *i.e.* $n = 2k+1$ for some $k \in \mathbb{N}$, $\left\lfloor \frac{n}{2} \right\rfloor = k$ and so

$$f(n) = \frac{k(2k+2)(2k+3)}{2} - \frac{k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$\frac{k(k+1)(4k+6) - k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$k(k+1)\left(\frac{1}{2} + \frac{2k+1}{3}\right) = \frac{k(k+1)(4k+5)}{6}$$

Obviously, $f(n) = \Theta(n^3)$. □

A5(n: positive integer)
```
1  a ← 0
2  for i ← 1 to n
3      for j ← i to n
4          for k ← n + i + j − 3 to n
5              a ← a + 1
6  return a
```

Problem 37. *Find the running time of algorithm* A5 *by determining the value of* a *it returns as a function of* n, $f(n)$. *Find a closed form for* $f(n)$.

Solution:
We have three nested **for** cycles and it is certainly true that $f(n) = O(n^3)$. However, now $f(n) \neq \Theta(n^3)$. It is easy to see that for any large enough n, line 5 is executed for only three values of the ordered triple $\langle i, j, k\rangle$. Namely,

$$\langle i, j, k\rangle \in \{\langle 1, 1, n-1\rangle, \langle 1, 1, n\rangle, \langle 1, 2, n-1\rangle\}$$

because the condition in the innermost loop (line 5) requires that $i + j \leq 3$. So, $f(n) = 3$, thus $f(n) = \Theta(1)$. □

Problem 37 raises a question: does it make sense to compute the running time of an iterative algorithm by counting how many time the expression in the innermost loop is executed? At lines 2 and 3 of A5 there are condition evaluations and variable increments – can we assume they take no time at all? Certainly, if that was a segment of a real-world program, the outermost two loops would be executed $\Theta(n^2)$ times, unless some sort of optimisation

was applied by the compiler. Anyway, we *postulate* that the running time is evaluated by counting how many times the innermost loop is executed. Whether that is a realistic model for real-world computation or not, is a side issue.

```
A6(a_1, a_2, ... a_n: array of positive distinct integers, n ≥ 3)
 1  S: a stack of positive integers
 2  (* P(S) is a predicate that is evaluated in Θ(1) time. *)
 3  (* If there are less than two elements in S then P(S) is false. *)
 4  push(a_1, S)
 5  push(a_2, S)
 6  for i ← 3 to n
 7      while P(S) do
 8          pop(S)
 9      push(a_i, S)
```

Problem 38. *Find the asymptotic growth rate of running time of* A6. *Assume the predicate* P *is evaluated in* $\Theta(1)$ *time and the push and pop operations are executed in* $\Theta(1)$ *time.*

Solution:
Certainly, the running time is $O(n^2)$ because the outer loop runs $\Theta(n)$ times and the inner loop runs in $O(n)$ time: note that for each concrete i, the inner loop (line 8) cannot be executed more than $n-2$ times sinse there are at most n elements in S and each execution of line 8 removes one element from S.

However, a more precise analysis is possible. Observe that each element of the array is being pushed in S and *may be* popped out of S later but only once. It follows that line 8 cannot be exesuted more than n times altogether, *i.e.* for all i, and so the algorithm runs in $\Theta(n)$ time. □

```
A7(a_1, a_2, ... a_n: array of positive distinct integers, x: positive integer)
 1  i ← 1
 2  j ← n
 3  while i ≤ j do
 4      k ← ⌊(i+j)/2⌋
 5      if x = a_k
 6          return k
 7      else if x < a_k
 8          j ← k − 1
 9      else i ← k + 1
10  return −1
```

Problem 39. *Find the asymptotic growth rate of running time of* A7.

Solution:
The following is a loop invariant for A7:

For every iteration of the ***while*** *loop of* A7 *it is the case that:*

$$j - i + 1 \leq \frac{n}{2^t} \tag{2.1}$$

where the iteration number is t*, for some* $t \geq 0$*.*

We prove it by induction on t. The basis is $t = 0$, *i.e.* the first time the execution reaches line 3. Then j is n, i is 1, and indeed $n - 1 + 1 \leq \frac{n}{2^0}$ for all sufficiently large n. Assume that at iteration t, $t \geq 1$, (2.1) holds, and there is yet another iteration to go through. Ignore the possibility $x = a_k$ (line 5) because, if that is true then iteration $t+1$ never takes place. There are exactly two ways to get from iteration t to iteration $t+1$ and we consider them in separate cases.

Case I: the execution reaches line 8 Now j becomes $\left\lfloor \frac{i+j}{2} \right\rfloor - 1$ and i stays the same.

$$\begin{aligned}
\frac{j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}} && \text{divide (2.1) by 2}\\
\frac{j+i-2i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - i + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - i + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - 1 - i + \frac{1}{2} + 1 &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - 1 - i + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \frac{1}{2} > 0\\
\underbrace{\left\lfloor \frac{j+i}{2} \right\rfloor - 1}_{\text{the new } j} - i + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \lfloor m \rfloor \leq m,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

And so the induction step follows from the induction hypothesis.

Case II: the execution reaches line 9 Now j stays the same and i becomes $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$.

$$\begin{aligned}
\frac{j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}} && \text{divide (2.1) by 2}\\
\frac{2j-j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
j - \frac{j+i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
j - \frac{i+j}{2} + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^{t+1}}\\
j - \frac{i+j}{2} - \frac{1}{2} + 1 &\leq \frac{n}{2^{t+1}}\\
j - \left(\frac{i+j}{2} + \frac{1}{2} \right) + 1 &\leq \frac{n}{2^{t+1}}\\
j - \underbrace{\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1 \right)}_{\text{the new } i} + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \left\lfloor \frac{i+j}{2} \right\rfloor + 1 \geq \frac{i+j}{2} + \frac{1}{2},\ \forall i, j \in \mathbb{N}^+
\end{aligned}$$

And so the induction step follows from the induction hypothesis.

Having proven (2.1), we consider the maximum value that t reaches, call it $t_{\max}$. During that last iteration of the loop, the values of i and j become equal, because the loop stops executing when $j < i$. Therefore, $j - i = 0$ during the execution of iteration $t_{\max}$—before i gets incremented or j gets decremented. So, substituting t with $t_{\max}$ and $j - i$ with 0 in the invariant, we get $2^{t_{\max}} \leq n \Leftrightarrow t_{\max} < \lceil \lg n \rceil$. It follows that the running time of A7 is $O(\lg n)$.

The following claim is a loop invariant for A7:

> *For every iteration of the* ***while*** *loop of* A7*, if the iteration number is* t*,* $t \geq 0$*, it is the case that:*
>
> $$\frac{n}{2^{t+1}} - 4 < j - i \tag{2.2}$$

We prove it by induction on t. The basis is $t = 0$, *i.e.* the first time the execution reaches line 3. Then j is n, i is 1, and indeed $\frac{n}{2^{1+0}} - 4 = \frac{n}{2} - 4 < n - 1$, for all sufficiently large n. Assume that at iteration t, $t \geq 1$, (2.2) holds, and there is yet another iteration to go through. Ignore the possibility $x = a_k$ (line 5) because, if that is true then iteration $t + 1$ never takes place. There are exactly two ways to get from iteration t to iteration $t + 1$ and we consider them in separate cases.

Case I: the execution reaches line 8 Now j becomes $\left\lfloor \frac{i+j}{2} \right\rfloor - 1$ and i stays the same.

$$\begin{aligned}
&\frac{n}{2^{t+2}} - 2 < \frac{j-i}{2} \quad \text{divide (2.2) by 2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{j+i-2i}{2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{j+i}{2} - i \\
&\frac{n}{2^{t+2}} - 4 < \frac{j+i}{2} - 2 - i \\
&\frac{n}{2^{t+2}} - 4 < \underbrace{\left\lfloor \frac{j+i}{2} \right\rfloor - 1}_{\text{the new } j} - i \quad \text{since } m - 2 \leq \lfloor m \rfloor - 1,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

Case II: the execution reaches line 9 Now j stays the same and i becomes $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$.

$$\begin{aligned}
&\frac{n}{2^{t+2}} - 2 < \frac{j-i}{2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{2j-j-i}{2} \\
&\frac{n}{2^{t+2}} - 2 < j - \frac{j+i}{2} \\
&\frac{n}{2^{t+2}} - 4 < j - \frac{j+i}{2} - 2 \\
&\frac{n}{2^{t+2}} - 4 < j - \left(\frac{j+i}{2} + 2 \right) \\
&\frac{n}{2^{t+2}} - 4 < j - \underbrace{\left(\left\lfloor \frac{j+i}{2} \right\rfloor + 1 \right)}_{\text{the new } i} \quad \text{since } m + 2 \geq \lfloor m \rfloor + 1,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

Having proven (2.2), it is trivial to prove that in the worst case, *e.g.* when x is not in the array, the loop is executed $\Omega(\lg n)$ times. □

Problem 40. *Determine the asymptotic running time of the following programming segment:*

```
s = 0;
for(i = 1; i * i <= n; i ++)
  for(j = 1; j <= i; j ++)
    s += n + i - j;
return s;
```

Solution:
The segment is equivalent to:

```
s = 0;
for(i = 1; i <= floor(sqrt(n)); i ++)
  for(j = 1; j <= i; j ++)
    s += n + i - j;
return s;
```

As we already saw, the running time is $\Theta\left(\left(\sqrt{n}\right)^2\right)$ and that is $\Theta(n)$. □

Problem 41. *Assume that* $A_{n\times n}$, $B_{n\times n}$, *and* $C_{n\times n}$ *are matrices of integers. Determine the asymptotic running time of the following programming segment:*

```
 for(i = 1; i <= n; i ++)
  for(j = 1; j <= n; j ++) {
    s = 0;
    for(k = 1; k <= n; k ++)
      s += A[i][k] * B[k][j];
    C[i][j] = s; }
return s;
```

Solution:
Having in mind the analysis of ADD-3 on page 25, clearly this is a $\Theta(n^3)$ algorithm. However, if consider the order of growth as a function of the *length of the input*, the order of growth is $\Theta\left(m^{\frac{3}{2}}\right)$, where m is the length of the input, *i.e.* m is the order of the number of elements in the matrices and $m = \Theta(n^2)$. □

```
A8(a_1, a_2, ... a_n: array of positive integers)
 1  s ← 0
 2  for i ← 1 to n − 4
 3      for j ← i to i + 4
 4          for k ← i to j
 5              s ← s + a_i
```

Problem 42. *Determine the running time of algorithm* A8.

Solution:
The outermost loop is executed $n-4$ times (assume large enough n). The middle loop is executed 5 times precisely. The innermost loop is executed 1, 2, 3, 4, or 5 times for j equal to i, $i+1$, $i+2$, $i+3$, and $i+4$, respectively. Altogether, the running time is $\Theta(n)$. □

```
A9(n: positive integer)
 1  s ← 0
 2  for i ← 1 to n − 4
 3      for j ← 1 to i + 4
 4          for k ← i to j
 5              s ← s + 1
 6  return s
```

Problem 43. *Determine the running time of algorithm* A9. *First determine the value it returns as a function of* n.

Solution:
We have to evaluate the sum:

$$\sum_{i=1}^{n-4} \sum_{j=1}^{i+4} \sum_{k=i}^{j} 1$$

Having in mind that

$$\sum_{k=i}^{j} 1 = \begin{cases} j-i+1, & \text{if } j \geq i \\ 0, & \text{else} \end{cases}$$

we rewrite the sum as:

$$\sum_{i=1}^{n-4}\left(\underbrace{\sum_{j=1}^{i-1}\sum_{k=i}^{j}1}_{\text{this is 0}}+\sum_{j=i}^{i+4}\sum_{k=i}^{j}1\right)=$$

$$\sum_{i=1}^{n-4}\sum_{j=i}^{i+4}(j-i+1)=$$

$$\sum_{i=1}^{n-4}\big((i-i+1)+(i+1-i+1)+(i+2-i+1)+(i+3-i+1)+(i+4-i+1)\big)=$$

$$\sum_{i=1}^{n-4}(1+2+3+4+5)=15(n-4)$$

So, algorithm A9 returns $15(n-4)$. The time complexity, though, is $\Omega(n^2)$ because the outer two loops require $\Omega(n^2)$ work. □

```
A10(n: positive integer)
1  a ← 0
2  for i ← 0 to n − 1
3      j ← 1
4      while j < 2n do
5          for k ← i to j
6              a ← a + 1
7          j ← j + 2
8  return a
```

Problem 44. *Find the running time of algorithm* A10 *by determining the value of* a *it returns as a function* $f(n)$ *of* n. *Find a closed form for* $f(n)$.

Solution:

$$f(n)=\sum_{i=0}^{n-1}\sum_{j\in\{1,3,5,\ldots,2n-1\}}\sum_{k=i}^{j}1$$

Let $n'=n-1$. Then

$$j\in\{1,3,5,\ldots,2n-1\}\Leftrightarrow j\in\{1,3,5,\ldots,2n'+1\}$$

But $\{1,3,5,\ldots,2n'+1\}=\{2\times 0+1,2\times 1+1,2\times 2+1,\ldots,2\times n'+1\}$. So we can rewrite the sum as:

$$f(n)=\sum_{i=0}^{n-1}\sum_{l=0}^{n-1}\sum_{k=i}^{2l+1}1$$

We know that

$$\sum_{k=i}^{2l+1}1=\begin{cases}2l+1-i+1, & \text{if } 2l+1\geq i\Leftrightarrow l\leq\left\lceil\frac{i-1}{2}\right\rceil\\ 0, & \text{otherwise}\end{cases}$$

Let $\lceil \frac{i-1}{2} \rceil$ be called i'. it must be case that

$$
\begin{aligned}
f(n) &= \sum_{i=0}^{n-1} \left(\sum_{l=0}^{i'-1} \underbrace{\sum_{k=i}^{2l+1} 1}_{0} \quad + \quad \sum_{l=i'}^{n-1} \underbrace{\sum_{k=i}^{2l+1}}_{2l+2-i} \right) \\
&= \sum_{i=0}^{n-1} \sum_{l=i'}^{n-1} (2l+2-i) \\
&= \sum_{i=0}^{n-1} \left((2-i) \sum_{l=i'}^{n-1} 1 + 2 \sum_{l=i'}^{n-1} l \right) = \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-1-i'+1) + 2 \sum_{l=i'}^{n-1} l \right) = \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-i') + 2 \sum_{l=i'}^{n-1} l \right) = \quad /\!/\!/ \; since \sum_{k=p}^{q} k = \frac{1}{2}(q+p)(q-p+1) \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-i') + 2 \times \frac{1}{2} \times (n-1+i')(n-1-i'+1) \right) = \\
&= \sum_{i=0}^{n-1} \big((2-i)(n-i') + (n-1+i')(n-i') \big) = \\
&= \sum_{i=0}^{n-1} \Big((n-i') \big((2-i) + (n-1+i') \big) \Big) = \\
&= \sum_{i=0}^{n-1} (n-i')(n+(-i+1+i')) \qquad (2.3)
\end{aligned}
$$

But

$$
\begin{aligned}
&-i+1+i' = -i+1+\left\lceil \frac{i-1}{2} \right\rceil = \\
&\left\lceil -i+1+\frac{i-1}{2} \right\rceil = \left\lceil \frac{-2i+2+i-1}{2} \right\rceil = \left\lceil \frac{-i+1}{2} \right\rceil = \left\lceil -\frac{i-1}{2} \right\rceil = -\left\lfloor \frac{i-1}{2} \right\rfloor
\end{aligned}
$$

since $\forall x \in \mathbb{R}, \lceil -x \rceil = -\lfloor x \rfloor$. Therefore, (2.3) equals

$$\sum_{i=0}^{n-1} \left(n - \left\lceil \frac{i-1}{2} \right\rceil \right) \left(n - \left\lfloor \frac{i-1}{2} \right\rfloor \right) =$$

$$\sum_{i=0}^{n-1} \left(n^2 - n \left\lfloor \frac{i-1}{2} \right\rfloor - n \left\lceil \frac{i-1}{2} \right\rceil + \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor \right) =$$

$$n^2 \sum_{i=0}^{n-1} 1 - n \sum_{i=0}^{n-1} \left(\underbrace{\left\lfloor \frac{i-1}{2} \right\rfloor + \left\lfloor \frac{i-1}{2} \right\rfloor}_{i-1} \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^2(n) - n \sum_{i=0}^{n-1} (i-1) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - n \left(\sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - n \left(\frac{(n-1)n}{2} - n \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - \frac{n^2(n-3)}{2} + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$\frac{n^3 + 3n^2}{2} + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor \tag{2.4}$$

By (8.38) on page 255,

$$\sum_{i=0}^{n-1} \left\lfloor \frac{i-1}{2} \right\rfloor \left\lceil \frac{i-1}{2} \right\rceil = \begin{cases} \dfrac{(n-2)n(2n-5)}{24}, & n-1 \text{ odd } \Leftrightarrow n \text{ even} \\ \dfrac{(n-3)(n-1)(2n-1)}{24}, & n-1 \text{ even} \Leftrightarrow n \text{ odd} \end{cases}$$

I. Suppose n is even. Then

$$\begin{aligned} f(n) &= \frac{n^3 + 3n^2}{2} + \frac{(n-2)n(2n-5)}{24} \\ &= \frac{12(n^3 + 3n^2) + (2n^3 - 9n^2 + 10n)}{24} \\ &= \frac{14n^3 + 27n^2 + 10n}{24} \\ &= \frac{n(2n+1)(7n+10)}{24} \end{aligned}$$

II. Suppose n is odd. Then

$$\begin{aligned} f(n) &= \frac{n^3+3n^2}{2} + \frac{(n-3)(n-1)(2n-1)}{24} \\ &= \frac{12n^3+36n^2+2n^3-9n^2+10n-3}{24} \\ &= \frac{(n+1)(14n^2+13n-3)}{24} \end{aligned}$$

Obviously, $f(n) = \Theta(n^3)$ in either case. □

Asymptotics of bivariate functions

Our notations from Chapter 1 can be generalised for two variables as follows. A bivariate function $f(n,m)$ is asymptotically positive iff

$$\exists n_0 \exists m_0 : \forall n \geq n_0 \forall m \geq m_0, f(n,m) > 0$$

Definition 2. *Let $g(n,m)$ be an asymptotically positive function with real domain and codomain. Then*

$$\begin{aligned} \Theta(g(n,m)) = \big\{ f(n,m) \mid \exists c_1, c_2 > 0, \exists n_0, m_0 > 0 : \\ \forall n \geq n_0, \forall m \geq m_0,\ 0 \leq c_1.g(n,m) \leq f(n,m) \leq c_2.g(n,m) \big\} \end{aligned}$$ □

Pattern matching is a computational problem in which we are given a *text* and a *pattern* and we compute how many times or, in a more elaborate version, at what *shifts*, the pattern occurs in the text. More formally, we are given two arrays of characters $T[1..n]$ and $P[1..m]$, such that $n \geq m$. For any k, $1 \leq k \leq n-m+1$, we have a shift at position k iff:

$$\begin{aligned} T[k] &= P[1] \\ T[k+1] &= P[2] \\ &\cdots \\ T[k+m-1] &= P[m] \end{aligned}$$

The problem then is to determine all the valid shifts. Consider the following algorithm for that problem.

```
NAIVE-PATTERN-MATHING(T[1..n]: characters, P[1..m]: characters)
1  (* assume n ≥ m *)
2  for i ← 1 to n − m + 1
3       if T[i, i + 1, ..., i + m − 1] = P
4            print "shift at" i
```

Problem 45. *Determine the running time of algorithm* NAIVE-PATTERN-MATHING.

Solution:
The algorithm is ostensibly $\Theta(n)$ because it has a single loop with the loop control variable running from 1 to n. That analysis, however, is wrong because the comparison at line 3 cannot be performed in constant time. Have in mind that m can be as large as n. Therefore, the algorihm is in fact:

```
NAIVE-PATTERN-MATHING-1(T[1..n]: characters, P[1..m]: characters)
 1  (* assume n ≥ m *)
 2  for i ← 1 to n − m + 1
 3      Match ← TRUE
 4      for j ← 1 to m
 5          if T[i + j − 1] ≠ P[j]
 6              Match ← FALSE
 7      if Match
 8          print "shift at" i
```

For obvious reasons this is a $\Theta((n-m).m)$ algorithm: both the best-case and the worst-case running times are $\Theta((n-m).m)$[†]. Suppose we improve it to:

```
NAIVE-PATTERN-MATHING-2(T[1..n]: characters, P[1..m]: characters)
 1  (* assume n ≥ m *)
 2  for i ← 1 to n − m + 1
 3      Match ← TRUE
 4      j ← 1
 5      while Match AND j ≤ m do
 6          if T[i + j − 1] = P[j]
 7              j ← j + 1
 8          else
 9              Match ← FALSE
10      if Match
11          print "shift at" i
```

NAIVE-PATTERN-MATHING-2 has the advantage that once a mismatch is found (line 9) the inner loop "breaks". Thus the best-case running time is $\Theta(n)$. A best case, for instance, is:

$$T = \underbrace{a\,a\,\ldots\,a}_{n \text{ times}} \quad \text{and} \quad P = \underbrace{b\,b\,\ldots\,b}_{m \text{ times}}$$

However, the worst case running time is still $\Theta((n-m).m)$. A worst case is, for instance:

$$T = \underbrace{a\,a\,\ldots\,a}_{n \text{ times}} \quad \text{and} \quad P = \underbrace{a\,a\,\ldots\,a}_{m \text{ times}}$$

It is easy to prove that $(n-m).m$ is maximised when m varies and n is fixed for $m \asymp \frac{n}{2}$ and achieves maximum value $\Theta(n^2)$. It follows that all the naive string matchings are, at worst, quadratic algorithms. □

[†]Algorithms that have the same—in asymptotic terms—running time for all inputs of the same length are called *oblivious*.

It is known that faster algorithms exist for the pattern matching problem. For instance, the Knuth-Morris-Pratt [KMP77] algorithm that runs in $\Theta(n)$ in the worst case.

Problem 46. *For any two strings* x *and* y *of the same length, we say that* x is a circular shift of y *iff* y *can be broken into substrings—one of them possibly empty—*y_1 *and* y_2*:*

$$y = y_1\, y_2$$

such that $x = y_2\, y_1$*. Find a linear time algorithm,* i.e. $\Theta(n)$ *in the worst case, that computes whether* x *is a circular shift of* y *or not. Assume that* $x \neq y$*.*

Solution:
Run the linear time algorithm for string matching of Knuth-Morris-Pratt with input $y\,y$ (y concatenated with itself) as text and x as pattern. The algorithm will output one or more valid shifts iff x is a circular shift of y, and zero valid shifts, otherwise. To see why, consider the concatenation of y with itself when it is a circular shift of x for some y_1 and y_2, such that $y = y_1\, y_2$ and $x = y_2\, y_1$:

$$y \quad y \quad = \quad y_1 \quad \underbrace{y_2 \quad y_1}_{\text{this is } x} \quad y_2$$

The running time is $\Theta(2n)$, *i.e.* $\Theta(n)$, at worst. □

Chapter 3

Recursive Algorithms and Recurrence Relations

3.1 Preliminaries

A *recursive algorithm* is an algorithm that calls itself, one or more times, on smaller inputs. To prevent an infinite chain of such calls there has to be a value of the input for which the algorithm does not call itself.

A *recurrence relation in one variable* is an equation, *i.e.* there is an "=" sign "in the middle", in which a function of the variable is equated to an expression that includes the same function on smaller value of the variable. In addition to that for some basic value of the variable, typically one or zero, an explicit value for the function is defined – that is the initial condition[†]. The variable is considered by default to take nonnegative integer values, although one can think of perfectly valid recurrence relations in which the variable is real.

Typically, in the part of the relation that is not the initial condition, the function of the variable is written on the left-hand side of the "=" sign as, say, $T(n)$, and the expression, on the right-hand side, *e.g.* $T(n) = T(n-1) + 1$. If the initial condition is, say, $T(0) = 0$, we typically write:

$$\begin{aligned} &T(n) = T(n-1) + 1,\ \forall n \in \mathbb{N}^+ \\ &T(0) = 0 \end{aligned} \tag{3.1}$$

It is not formally incorrect to write the same thing as:

$$\begin{aligned} &T(n-1) = T(n-2) + 1,\ \forall n \in \mathbb{N}^+, n \neq 1 \\ &\quad T(0) = 0 \end{aligned}$$

The equal sign is interpreted as an assignment from right to left, just as the equal sign in the C programming language, so the following "unorthodox" way of describing the same

[†]Note there can be more than one initial condition as in the case with the famous Fibonacci numbers:

$$\begin{aligned} &F(n) = F(n-1) + F(n-2),\ \forall n \in \mathbb{N}^+, n \neq 1 \\ &F(1) = 1 \\ &F(0) = 0 \end{aligned}$$

The number of initial conditions is such that the initial conditions prevent "infinite descent".

relation is *discouraged*:

$$T(n-1)+1 = T(n),\ \forall n \in \mathbb{N}^+$$
$$0 = T(0)$$

Each recurrence relation defines an infinite numerical sequence, provided the variable is integer. For example, (3.1) defines the sequence $0, 1, 2, 3, \ldots$. Each term of the relation, except for the terms defined by the initial conditions, is defined recursively, *i.e.* in terms of smaller terms, hence the name. To *solve* a recurrence relation means to find a non-recursive expression for the same function – one that defines the same sequence. For example, a solution of (3.1) is $T(n) = n$.

It is natural to describe the running time of a recursive algorithm by some recurrence relation. However, since we are interested in asymptotic running times, we do not need the precise solution of a "normal" recurrence relation as described above. A normal recurrence relation defines a sequence of numbers. If the time complexity of an algorithm as a worst-case analysis was given by a normal recurrence relation then the number sequence a_1, a_2, a_3, ..., defined by that relation, would describe the running time of algorithm precisely, *i.e.* for input of size n, the maximum number of steps the algorithm makes over all inputs of size n is precisely a_n. We do not need such a precise analysis and often it is impossible to derive one. So, the recurrence relations we use when analysing an algorithm typically have bases $\Theta(1)$, for example:

$$T(n) = T(n-1)+1,\ n \geq 2 \tag{3.2}$$
$$T(1) = \Theta(1)$$

Infinitely many number sequences are solutions to (3.2). To solve such a recurrence relation means to find the asymptotic growth of any of those sequences. The best solution we can hope for, asymptotically, is the one given by the Θ notation. If we are unable to pin down the asymptotic growth in that sense, our second best option is to find functions $f(n)$ and $g(n)$, such that $f(n) = o(g(n))$ and $T(n) = \Omega(f(n))$ and $T(n) = O(g(n))$. The best solution for the recurrence relation (3.2), in the asymptotic sense, is $T(n) = \Theta(n)$. Another solution, not as good as this one, is, for example, $T(n) = \Omega(\sqrt{n})$ and $T(n) = O(n^2)$.

In the problems that follow, we distinguish the two types of recurrence relation by the initial conditions. If the initial condition is given by a precise expression as in (3.1) we have to give a precise answer such as $T(n) = n$, and if the initial condition is $\Theta(1)$ as in (3.2) we want only the growth rate.

It is possible to omit the initial condition altogether in the description of the recurrence. If we do so we assume tacitly the initial condition is $T(c) = \Theta(1)$ for some positive constant c. The reason to do that may be that it is pointless to specify the usual $T(1)$; however, it may be the case that the variable never reaches value one. For instance, consider the recurrence relation

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor + 17\right) + n$$

which we solve below (Problem 53 on page 57). To specify "$T(1) = \Theta(1)$" for it is *wrong*.

3.1.1 Iterators

The recurrence relations can be partitioned into the following two classes, assuming T is the function of the recurrence relations as above.

1. The ones in which T appears only once on the right-hand side as in (3.1).
2. The ones in which T appears mutiple times on the right-hand side, for instance:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + n \tag{3.3}$$

We will call them relations with *single occurrence* and with *multiple occurrences*, respectively. Of course, that distinction is somewhat fuzzy because the following recurrence relations: $T(n) = 2T(n-1)$ and $T(n) = T(n-1) + T(n-1)$ are clearly equivalent—they define the same sequence if the initial condition is the same—and yet the second one is superficially with multiple occurrences while the first one is not. Therefore, those definitions are not purely syntactic. With a certain lack of precision we define that recurrence relation with a single occurrence is one which can easily be transformed into an equivalent recurrence relation in which the symbol T appears precisely once on the right-hand side.

We find it helpful to make that distinction because in general only the relations with single occurrence are ameaneable to the method of unfolding (see below). If the relation is with single occurrence we define *the iterator* of the relation as the iterative expression that shows how the variable decreases. For example, the iterator of (3.1) is:

$$n \to n - 1 \tag{3.4}$$

It is not practical to define iterators for relations with multiple occurrences. If we wanted to define iterators for them as well, they would have a set of functions on the right-hand side, for instance the iterator of (3.3) would be

$$n \to \{n-1, n-2, n-3\}$$

and that does help the analysis of the relation. So, we define iterators only for relations with single occurrence. The iterators that are easiest to deal with (and, fortunately, occur often in practice) are the ones in which the function on the right-hand side is subtraction or division (by constant > 1):

$$n \to n - c,\ c > 0 \tag{3.5}$$

$$n \to \frac{n}{b},\ b > 1 \tag{3.6}$$

Another possibility is that function to be some root of n:

$$n \to \sqrt[d]{n},\ d > 1 \tag{3.7}$$

Note that the direction of the assignment in the iterator is the opposite to the one in the recurrence relation (compare (3.1) with (3.4)). The reason is that a recurrence has to phases: descending and ascending. In the descending phase we start with some value n for the variable and decrease it in successive steps till we reach the initial condition; in the ascending phase we go back from the initial condition "upwards". The left-to-right direction of the iterator refers to the descending phase, while the right-to-left direction of the assignment in the recurrence refers to the ascending phase.

It is important to be able to estimate the number of times an iterator will be executed before its variable becomes 1 (or whatever value the initial conditions specify). If the variable n

is integer, the iterator $n \to n-1$ is the most basic one we can possibly have. The number of times it is executed before n becomes any *a priori* fixed constant is $\Theta(n)$. That has to be obvious. Now consider (3.5). We ask the same question: how many times it is executed before n becomes a constant. Substitute n by cm and (3.5) becomes:

$$cm \to c(m-1) \tag{3.8}$$

The number of times (3.8) is executed (before m becomes a constant) is $\Theta(m)$. Since $m = \Theta(n)$, we conclude that (3.5) is executed $\Theta(n)$ times.

Consider the iterator (3.6). To see how many times it is executed before n becomes a constant (fixed *a priori*)) can be estimated as follows. Substitute n by b^m and (3.6) becomes

$$b^m \to b^{m-1} \tag{3.9}$$

(3.9) is executed $\Theta(m)$ times because $m \to m-1$ is executed $\Theta(m)$ times. Since $m = \log_b n$, we conclude that (3.6) is executed $\Theta(\log_b n)$ times, *i.e.* $\Theta(\lg n)$ times. We see that the concrete value of b is immaterial with respect to the asymptotics of the number of executions, provided $b > 1$.

Now consider (3.7). To see how many times it is executed before n becomes a constant, substitute n by d^{d^m}. (3.7) becomes

$$d^{d^m} \to d^{\frac{d^m}{d}} = d^{d^{m-1}} \tag{3.10}$$

(3.10) is executed $\Theta(m)$ times. As $m = \log_d \log_d n$, we conclude that (3.7) is executed $\Theta(\log_d \log_d n)$ times, *i.e.* $\Theta(\lg \lg n)$ times. Again we see that the value of the constant in the iterator, namely d, is immaterial as long as $d > 1$.

Let us consider an iterator that decreases even faster than (3.7):

$$n \to \lg n \tag{3.11}$$

The number of times it is executed before n becomes a constant is $\lg^* n$, which follows right from Definion 1 on page 13.

Let us summarise the rates of decrease of the iterators we just considered assuming the mentioned "constants of decrease" b and d are 2.

iterator	*asymptotics of the number executions*	*alternative form (see Definion 1)*
$n \to n-1$	n	$\lg^{(0)} n$
$n \to n/2$	$\lg n$	$\lg^{(1)} n$
$n \to \sqrt{n}$	$\lg \lg n$	$\lg^{(2)} n$
$n \to \lg n$	$\lg^* n$	$\lg^* n$

There is a gap in the table. One would ask, what is the function $f(n)$, such that the iterator $n \to f(n)$ is executed, asymptotically, $\lg \lg \lg n$ times, *i.e.* $\lg^{(3)} n$ times. To answer that question, consider that $f(n)$ has to be such that if we substitute n by 2^m, the number of executions is the same as in the iterator $m \to \sqrt{m}$. But $m \to \sqrt{m}$ is the same as $\lg n \to \sqrt{\lg n}$, *i.e.* $n \to 2^{\sqrt{\lg n}}$. We conclude that $f(n) = 2^{\sqrt{\lg n}}$. To check this, consider the iterator

$$n \to 2^{\sqrt{\lg n}} \tag{3.12}$$

Substitute n by $2^{2^{2^m}}$ in (3.12) to obtain:

$$2^{2^{2^m}} \to 2^{\sqrt{\lg 2^{2^{2^m}}}} = 2^{\sqrt{2^{2^m}}} = 2^{2^{\frac{2^m}{2}}} = 2^{2^{2^m-1}} \tag{3.13}$$

Clearly, (3.13) is executed $m = \lg\lg\lg n = \lg^{(3)} n$ times.

A further natural question is, what the function $\phi(n)$ is, such that the iterator $n \to \phi(n)$ is executed $\lg^{(4)} n$ times. Applying the reasoning we used to derive $f(n)$, $\phi(n)$ has to be such that if we substitute n by 2^m, the number of executions is the same as in $m \to 2^{\sqrt{\lg m}}$. As $m = \lg n$, the latter becomes $\lg n \to 2^{\sqrt{\lg\lg n}}$, *i.e.* $n \to 2^{2^{\sqrt{\lg\lg n}}}$. So, $\phi(n) = 2^{2^{\sqrt{\lg\lg n}}}$. We can fill in two more rows in the table:

iterator	*asymptotics of the number executions*	*alternative form (see Definion 1)*
$n \to n-1$	n	$\lg^{(0)} n$
$n \to n/2$	$\lg n$	$\lg^{(1)} n$
$n \to \sqrt{n}$	$\lg\lg n$	$\lg^{(2)} n$
$n \to 2^{\sqrt{\lg n}}$	$\lg\lg\lg n$	$\lg^{(3)} n$
$n \to 2^{2^{\sqrt{\lg\lg n}}}$	$\lg\lg\lg\lg n$	$\lg^{(4)} n$
$n \to \lg n$	$\lg^* n$	$\lg^* n$

Let us define, analogously to Definion 1, the function base-two iterated exponent.

Definition 3 (iterated exponent)**.** *Let i be a nonnegative integer.*

$$\text{itexp}^{(i)}(n) = \begin{cases} n, & \text{if } i = 0 \\ 2^{\text{itexp}^{(i-1)}(n)}, & \text{if } i > 0 \end{cases}$$ □

Having in mind the results in the table, we conjecture, and it should not be too difficult to prove by induction, that the iterator:

$$n \to \text{itexp}^{(k)}\left(\sqrt{\lg^{(k)} n}\right) \tag{3.14}$$

is executed $\lg^{(k+2)} n$ times for $k \in \mathbb{N}$.

3.1.2 Recursion trees

Assume we are given a recurrence relation of the form:

$$T(n) = k_1 T(f_1(n)) + k_2 T(f_2(n)) + \ldots + k_p T(f_p(n)) + \phi(n) \tag{3.15}$$

where k_i, $i \le i \le p$ are positive integer constants, $f_i(n)$ for $1 \le i \le p$ are integer-valued functions such that $n > f(n)$ for all $n \ge n_0$ where n_0 is the largest (constant) value of the argument in any initial condition, and $\phi(n)$ is some positive function. It is not necessary $\phi(n)$ to be positive as the reader will see below; however, if $T(n)$ describes the running time of a recursive algorithm then $\phi(n)$ has to be positive. We build a special kind of rooted tree that corresponds to our recurrence relation. Each node of the tree corresponds to one particular value of the variable that appears in the process of unfolding the relation,

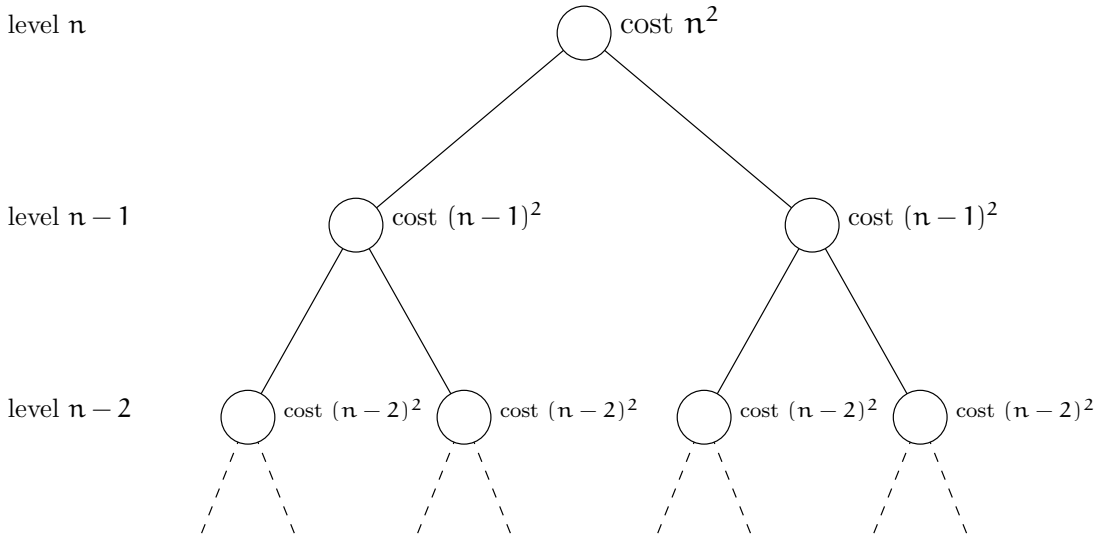


Figure 3.1: The recursion tree of $T(n) = 2T(n-1) + n^2$.

the value that corresponds to the root being n. That value we call *the level* of the node. Further, with each node we associate $\phi(m)$ where m is the level of that node. We call that, *the cost* of the node. Further, each node—as long as no initial condition has been reached yet—has $k_1 + k_2 + \ldots + k_p$ children, k_i of them being at level defined by f_i for $1 \leq i \leq p$. For example, if our recurrence is

$$T(n) = 2T(n-1) + n^2$$

the recursion tree is as shown on Figure 3.1. It is a complete binary tree. It is binary because there are two invocations on the right side, *i.e.* $k_1 + k_2 + \ldots + k_p = 2$ in the above terminology. And it is complete because it is a recurrence with a single occurrence. Note that if $k_1 + k_2 + \ldots + k_p$ equals 1 then the recursion tree degenerates into a path.

The size of the tree depends on n so we can not draw the whole tree. The figure is rather a suggestion about it. The bottom part of the tree is missing because we have not mentioned the initial conditions. The solution of the recursion—and that is the goal of the tree, to help us solve the recursion—is the total sum of all the costs. Typically we sum by levels, so in the current example the sum will be

$$n^2 + 2(n-1)^2 + 4(n-2)^2 + \ldots$$

The general term of this sum is $2^k(n-k)^2$. The "..." notation hides what happens at the right end, however, we agreed the initial condition is for some, it does not matter, what constant value of the variable. Therefore, the sum

$$\sum_{k=0}^{n} 2^k(n-k)^2$$

has the same growth rate as our desired solution. Let us find a closed form for that sum.

$$\sum_{k=0}^{n} 2^k(n-k)^2 = n^2 \sum_{k=0}^{n} 2^k - 2n \sum_{k=0}^{n} 2^k k + \sum_{k=0}^{n} 2^k k^2$$

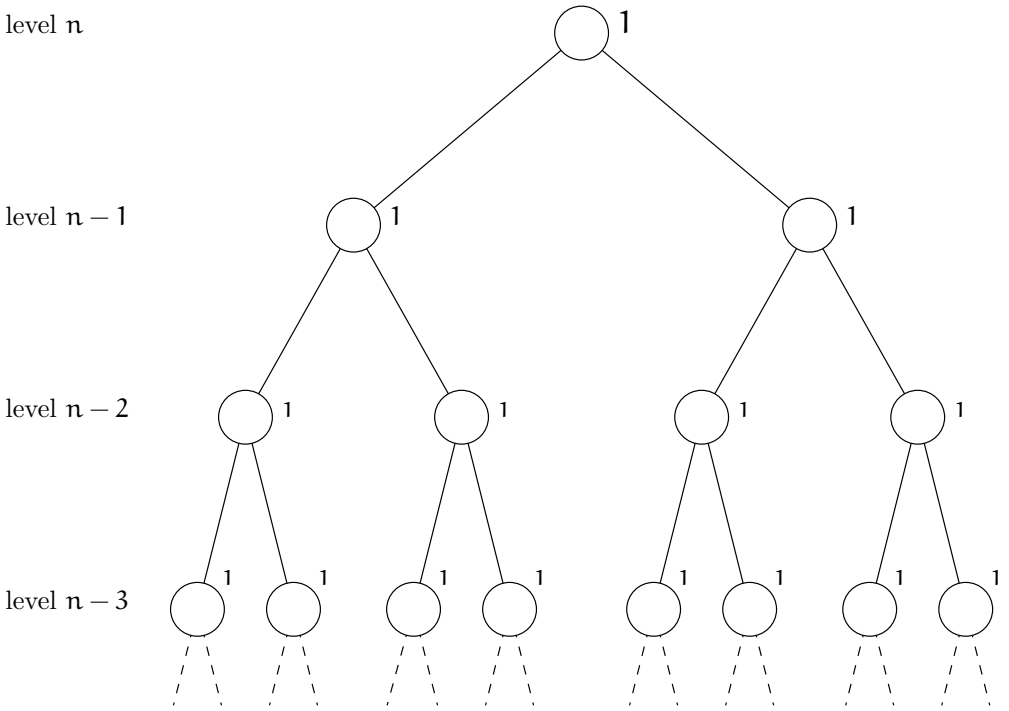


Figure 3.2: The recursion tree of $T(n) = 2T(n-1) + 1$.

Having in mind Problem 144 on page 237 and Problem 145 on page 237, that expression becomes

$$n^2(2^{n+1} - 1) - 2n((n-1)2^{n+1} + 2) + n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$n^2.2^{n+1} - n^2 - 2n(n.2^{n+1} - 2^{n+1} + 2) + n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$2.n^2.2^{n+1} - n^2 - 2.n^2.2^{n+1} + 2n.2^{n+1} - 4n - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$4.2^{n+1} - n^2 - 4n - 6$$

It follows that $T(n) = \Theta(2^n)$.

The correspondence between a recurrence relation and its recursion tree is not necessarily one-to-one. Consider the recurrence relation

$$\begin{aligned} &T(n) = 2T(n-1) + 1,\ n \geq 2 \\ &T(1) = \Theta(1) \end{aligned} \tag{3.16}$$

and its recursion tree (Figure 3.2). The cost at level n is 1, at level $n-1$ is 2, at level $n-2$ is 4, at level $n-3$ is 8, *etc.* The tree is obviously complete. Let us now rewrite (3.2) as follows.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \ \Leftrightarrow\ T(n) = T(n-1) + T(n-1) + 1 \\ T(n-1) &= 2T(n-2) + 1 \\ T(n) &= T(n-1) + 2T(n-2) + 2 \end{aligned}$$

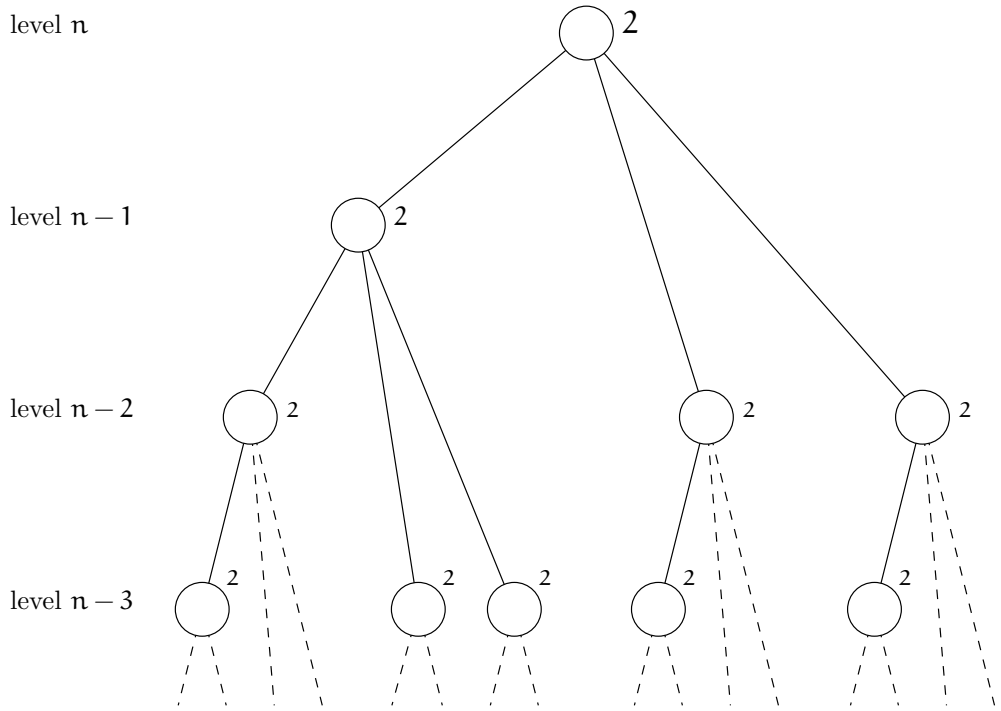


Figure 3.3: The recursion tree of $T(n) = T(n-1) + 2T(n-2) + 2$.

We have to alter the initial conditions for this rewrite, adding $T(2) = 3$. Overall the recurrence becomes

$$\begin{aligned} &T(n) = T(n-1) + 2T(n-2) + 2 \\ &T(2) = 3 \\ &T(1) = 1 \end{aligned} \tag{3.17}$$

Recurrences (3.2) and (3.17) are equivalent. One can say these are different ways of writing down the same recurrence because both of them define one and the same sequence, namely $1, 3, 7, 15, \ldots$ However, their recursion trees are neither the same nor isomorphic. Figure 3.3 shows the tree of (3.17). To give a more specific example, Figure 3.4 shows the recursion tree of (3.17) for $n = 5$. It shows the whole tree, not just the top, because the variable has a concrete value. Therefore the initial conditions are taken into account. The reader can easily see the total sum of the costs over the tree from Figure 3.4 is 31, the same as the tree from Figure 3.2 for $n = 5$. However, the sum 31 on Figure 3.2 is obtained as $1+2+4+8+16$, if we sum by levels. In the case with Figure 3.4 we do not have obvious definition of levels.

- If we define the levels as the vertices that have the same value of the variable, we have 5 levels and the sum is derived, level-wise, as $2 + 2 + 6 + 15 + 6 = 31$.
- If we define the levels as the vertices that are at the same distance to the root, we have only 4 levels and the sum is derived, level-wise, as $2 + 6 + 18 + 5 = 31$.

Regardless of how we define the levels, the derivation is not $1 + 2 + 4 + 8 + 16$.

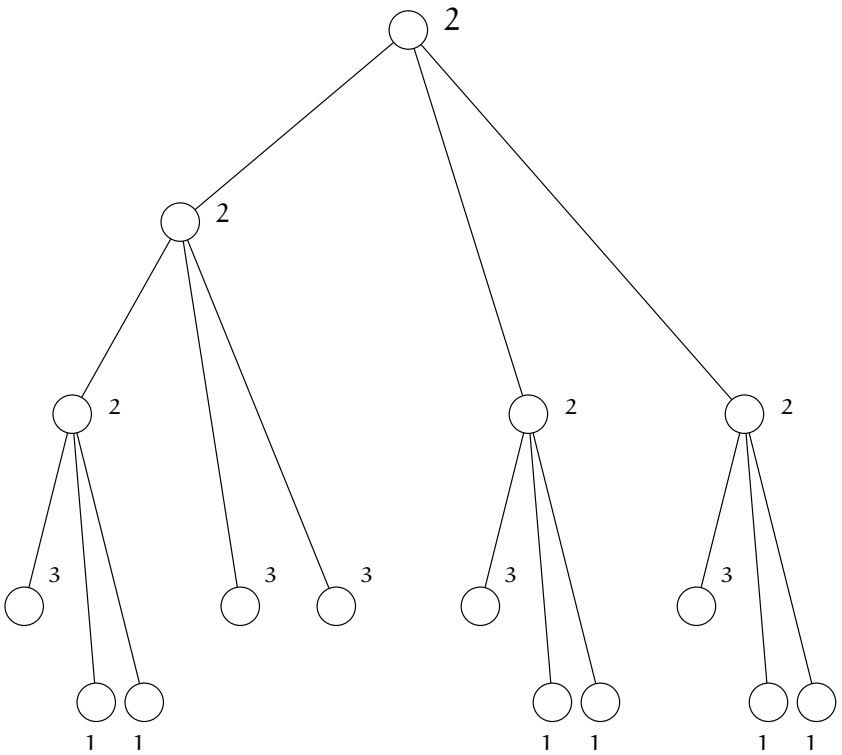


Figure 3.4: The recursion tree of $T(n) = T(n-1) + 2T(n-2) + 2$, $T(2) = 3$, $T(1) = 1$, for $n = 5$.

3.2 Problems

Our repertoire of methods for solving recurrences is:

- by induction,
- by unfolding,
- by considering the recursion tree,
- by the Master Theorem, and
- by the method of the characteristic equation.

3.2.1 Induction, unfolding, recursion trees

Problem 47. *Solve*

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ T(0) &= 0 \end{aligned} \tag{3.18}$$

Solution:
We guess that $T(n) = 2^n - 1$ for all $n \geq 1$ and prove it by induction on n.

Basis: $n = 1$. We have $T(1) = 2T(0) + 1$ by substituting n with 1. But $T(0) = 0$, thus $T(1) = 2 \times 0 + 1 = 1$. On the other hand, substituting n with 1 in our guessed solution, we have $2^1 - 1 = 1$.

Inductive hypothesis: assume $T(n) = 2^n - 1$ for some $n > 1$.

Inductive step: $T(n+1) = 2T(n)+1$ by definition. Apply the inductive hypothesis to obtain $T(n+1) = 2(2^n - 1) + 1 = 2^{n+1} - 1$. □

The proofs by induction have one major drawback – making a good guess can be a form of art. There is no recipe, no algorithm for making a good guess in general. It makes sense to compute several initial values of the sequence defined by the recurrence and try to see a pattern in them. In the last problem, $T(1) = 1$, $T(2) = 3$, $T(3) = 7$ and it is reasonable to assume that $T(n)$ is $2^n - 1$. Actually, if we think about (3.18) in terms of the binary representation of $T(n)$, it is pretty easy to spot that (3.18) performs a shift-left by one position and then turns the least significant bit from 0 into 1. As we start with $T(1) = 1$, clearly

$$T(n) = \underbrace{111\ldots1}_{n \text{ times}}{}_{\text{b}}$$

For more complicated recurrence relations, however, seeing a pattern in the initial values of the sequence, and thus making a good guess, can be quite challenging. If one fails to see such a pattern it is a good idea to check if these numbers are found in *The On-Line Encyclopedia of Integer Sequences* [Slo]. Of course, this advice is applicable when we solve precise recurrence relations, not asymptotic ones.

Problem 48. *Solve by unfolding*

$$\begin{aligned} T(n) &= T(n-1) + n \\ T(0) &= 1 \end{aligned} \tag{3.19}$$

Solution:
By unfolding (also called unwinding) of the recurrence down to the initial condition.

$$\begin{aligned} T(n) &= T(n-1) + n \quad \text{directly from (3.19)} \\ &= T(n-2) + n - 1 + n \quad \text{substitute } n \text{ with } n-1 \text{ in (3.19)} \\ &= T(n-3) + n - 2 + n - 1 + n \quad \text{substitute } n-1 \text{ with } n-2 \text{ in (3.19)} \\ &\ldots \\ &= T(0) + 1 + 2 + 3 + \ldots + n - 2 + n - 1 + n = \\ &= 1 + 1 + 2 + 3 + \ldots + n - 2 + n - 1 + n = \\ &= 1 + \frac{n(n+1)}{2} \end{aligned}$$

This method is considered to be not as formally precise as the induction. The reason is that we inevitably skip part of the derivation—the dot-dot-dot "..." part—leaving it to the imagination of the reader to verify the derived closed formula. Problem 48 is trivially simple and it is certain beyond any doubt that if we start with $T(n-3) + n - 2 + n - 1 + n$ and systematically unfold $T(i)$, decrementing by one values of i, eventually we will "hit"

the initial condition $T(0)$ and the "tail" will be $1+2+3+\ldots+n-2+n-1+n$. The more complicated the expression is, however, the more we leave to the imagination of the reader when unfolding.

One way out of that is to use the unfolding to derive a closed formula and then prove it by induction. □

Problem 49. *Solve*

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{3.20}$$

$$T(1) = \Theta(1) \tag{3.21}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$.

Part i: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.22}$$

There is a potential problem with the initial condition because for $n = 1$ the right-hand side of (3.22) becomes $c.1. \lg 1 = 0$, and $0 \neq \Theta(1)$. However, it is easy to deal with that issue, just do not take $n = 1$ as basis. Taking $n = 2$ as basis works as $c.2. \lg 2$ is not zero. However, note that $n = 2$ is not sufficient basis! There are certain values for n, for example 3, such that the iterator of this recurrence, namely

$$n \to \left\lfloor \frac{n}{2} \right\rfloor$$

"jumps over" 2, having started from one of them. Indeed, $\left\lfloor \frac{3}{2} \right\rfloor = 1$, therefore the iterator, starting from 3, does

$$3 \to 1$$

and then goes infinite descent. The solution is to take two bases, for both $n = 2$ and $n = 3$. It is certain that no matter what n is the starting one, the iterator will at one moment "hit" either 2 or 3. So, the bases of our proof are:

$$T(2) = \Theta(1) \tag{3.23}$$

$$T(3) = \Theta(1) \tag{3.24}$$

Of course, that does not say that $T(2) = T(3)$, it says there exist constants c_2 and c_3, such that:

$$c_2 \leq c.2 \lg 2$$
$$c_3 \leq c.3 \lg 3$$

Our induction hypothesis is that relative to some sufficiently large n, (3.22) holds for some positive constant c all values of the variable between 3 and n, excluding n. The induction step is to prove (3.22), using the hypothesis. So,

$$\begin{aligned}
T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \qquad \text{this is the defintion of } T(n) \\
&\leq 2.c.\left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \qquad \text{from the inductive hypothesis} \\
&\leq 2.c.\frac{n}{2} \lg \frac{n}{2} + n \\
&= cn(\lg n - 1) + n \\
&= cn \lg n + (1-c)n \qquad (3.25) \\
&\leq cn \lg n \qquad \text{provided that } (1-c) \leq 0 \Leftrightarrow c \geq 1 \qquad (3.26)
\end{aligned}$$

If $c \geq 1$, the proof is valid. If we want to be perfectly precise we have to consider the two bases as well to find a value for c that works. Namely,

$$c = \max\left\{1, \frac{c_2}{2\lg 2}, \frac{c_3}{3\lg 3}\right\}$$

In our proofs from now on we will not consider the initial conditions when choosing an appropriate constant.

Part ii: Proof that $T(n) = \Omega(n \lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn \lg n \qquad (3.27)$$

We will ignore the basis of the induction and focus on the hypothesis and the inductive step only. Applying the inductive hypothesis to (3.27), we get:

$$\begin{aligned}
T(n) &\geq 2d\left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \qquad \text{from the inductive hypothesis} \\
&\geq 2d\left(\frac{n}{2} - 1\right) \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\
&= d(n-2) \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\geq d(n-2) \lg \left(\frac{n}{4}\right) + n \\
&= d(n-2)\,(\lg n - 2) + n \\
&= dn \lg n + n(1-2d) - 2d \lg n + 4d \\
&\geq dn \lg n \qquad \text{provided that } n(1-2d) - 2d \lg n + 4d \geq 0
\end{aligned}$$

So (3.27) holds when

$$n(1-2d) - 2d \lg n + 4d \geq 0 \qquad (3.28)$$

Observe that for $d = \frac{1}{4}$ inequality (3.28) becomes

$$\frac{n}{2} + 1 \geq \frac{1}{2} \lg n$$

It certainly holds $\forall n \geq 2$, therefore the choice $d = \frac{1}{4}$ and $n_1 = 2$ suffices for our proof. □

Problem 50. *Solve*

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \tag{3.29}$$

$$T(1) = \Theta(1) \tag{3.30}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$. We ignore the basis of the induction – the solution of Problem 49 gives us enough confidence that we can handle the basis if we wanted to.

Part i: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.31}$$

From the inductive hypothesis

$$\begin{aligned}
T(n) &\leq 2.c. \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\
&\leq 2.c. \left(\frac{n}{2} + 1\right) \lg \left\lceil \frac{n}{2} \right\rceil + n && (3.32) \\
&\leq 2.c. \left(\frac{n}{2} + 1\right) \lg \left(\frac{3n}{4}\right) + n \qquad \text{because } \frac{3n}{4} \geq \left\lceil \frac{n}{2} \right\rceil \ \forall n \geq 2 && (3.33) \\
&= c(n+2)(\lg n + \lg 3 - 2) + n \\
&= cn \lg n + cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \\
&\leq cn \lg n \qquad \text{if } \ cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \leq 0
\end{aligned}$$

Consider

$$cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n = (c(\lg 3 - 2) + 1)n + 2c \lg n + 2c(\lg 3 - 2)$$

Its asymptotic growth rate is determined by the linear term. If the constant $c(\lg 3 - 2) + 1$ is negative then the whole expression is certainly negative for all sufficiently large values of n. In other words, for the sake of brevity we do not specify precisely what n_0 is. In order to have $c(\lg 3 - 2) + 1 < 0$ it must be the case that $c > \frac{1}{2 - \lg 3}$. So, any $c > \frac{1}{2 - \lg 3}$ works for our proof.

❣❣ NB ❣❣ *In (3.32) we substitute $\left\lceil \frac{n}{2} \right\rceil$ with $\frac{3n}{4}$. We could have used any other fraction $\frac{pn}{q}$, provided that $\frac{1}{2} < \frac{p}{q} < 1$. It is easy to see why it has to be the case that $\frac{1}{2} < \frac{p}{q}$: unless that is fulfilled we cannot claim there is a "$\leq$" inequality between (3.32) and (3.33). Now we argue it has to be the case that $\frac{p}{q} < 1$. Assume that $\frac{p}{q} = 1$,* i.e., *we substitute $\left\lceil \frac{n}{2} \right\rceil$ with n. Then (3.33) becomes:*

$$c(n+2)(\lg n) + n = cn \lg n + 2c \lg n + n$$

Clearly, that is bigger than $cn \lg n$ for all sufficiently large n and we have no proof.

Part ii: Proof that $T(n) = \Omega(n \lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn \lg n \tag{3.34}$$

From the inductive hypothesis

$$\begin{aligned} T(n) &\geq 2.d. \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\ &\geq 2.d. \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) + n \\ &= dn(\lg n - 1) + n \\ &= dn \lg n + (1-d)n \\ &\geq dn \lg n \qquad \text{provided that } (1-d)n \geq 0 \end{aligned} \tag{3.35}$$

It follows that any d such that $0 < d \leq 1$ works for our proof. □

⁘ NB ⁘ *As explained in [CLR00, pp. 56–57], it is easy to make a wrong "proof" of the growth rate by induction if one is not careful. Suppose one "proves" the solution of (3.20) is* $T(n) = O(n)$ *by first guessing (incorrectly) that* $T(n) \leq cn$ *for some positive constant* c *and then arguing*

$$\begin{aligned} T(n) &\leq 2c \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq cn + n \\ &= (c+1)n \\ &= O(n) \end{aligned}$$

While it is certainly true that $cn + n = O(n)$*, that is irrelevant to the proof. The proof started relative to the constant* c *and has to finish relative to it. In other words, the proof has to show that* $T(n) \leq cn$ *for the choice of* c *in the inductive hypothesis, not that* $T(n) \leq dn$ *for some positive constant* d *which is not* c*. Proving that* $T(n) \leq (c+1)n$ *does* not *constitute a proof of the statement we are after.*

Problem 51. *Solve*

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \tag{3.36}$$

$$T(1) = \Theta(1) \tag{3.37}$$

Solution:
We prove that $T(n) = \Theta(\lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$.

Part i: Proof that $T(n) = O(\lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq c \lg n \tag{3.38}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\le c \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&\le c \lg\left(\frac{n}{2}\right) + 1 \\
&= c(\lg n - 1) + 1 \\
&= c \lg n + 1 - c \\
&\le c \lg n \qquad \text{provided that } 1 - c \le 0 \Leftrightarrow c \ge 1
\end{aligned}$$

Part ii: Proof that $T(n) = \Omega(\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \ge n_1$,

$$T(n) \ge d \lg n \tag{3.39}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\ge d \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&\ge d \lg\left(\frac{n}{4}\right) + 1 \qquad \text{since } \frac{n}{4} \le \left\lfloor \frac{n}{2} \right\rfloor \text{ for all sufficiently large } n \\
&= d \lg n - 2d + 1 \\
&\ge d \lg n \qquad \text{provided that } -2d + 1 \ge 0 \;\Leftrightarrow\; d \le \frac{1}{2}
\end{aligned}$$

□

Problem 52. *Solve*

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \tag{3.40}$$

$$T(1) = \Theta(1) \tag{3.41}$$

Solution:
We prove that $T(n) = \Theta(\lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$.

Part i: Proof that $T(n) = O(\lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \ge n_0$,

$$T(n) \le c \lg n \tag{3.42}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\le c \lg\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\
&\le c \lg\left(\frac{3n}{4}\right) + 1 \\
&= c(\lg n + \lg 3 - 2) + 1 \\
&= c \lg n + c(\lg 3 - 2) + 1 \\
&\le c \lg n \qquad \text{provided that } c(\lg 3 - 2) + 1 \le 0 \Leftrightarrow c \ge \frac{1}{2 - \lg 3}
\end{aligned}$$

Part ii: Proof that $T(n) = \Omega(\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq d \lg n \tag{3.43}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq d \lg \left(\left\lceil \frac{n}{2} \right\rceil \right) + 1 \\ &\geq d \lg \left(\frac{n}{2} \right) + 1 \\ &= d \lg n - d + 1 \\ &\geq d \lg n \qquad \text{provided that } -d + 1 \geq 0 \Leftrightarrow d \leq 1 \end{aligned}$$

□

Problem 53. *Solve*

$$T(n) = 2T \left(\left\lfloor \frac{n}{2} + 17 \right\rfloor \right) + n \tag{3.44}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$. Note that the initial condition in this problem is *not* $T(1) = \Theta(1)$ because the iterator

$$n \to \left\lfloor \frac{n}{2} \right\rfloor + 17$$

never reaches 1 when starting from any sufficiently large n. Its fixed point is 34 but we avoid mentioning the awkward initial condition $T(34) = \Theta(1)$.

Part i: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.45}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c \left(\left\lfloor \frac{n}{2} \right\rfloor + 17 \right) \lg \left(\left\lfloor \frac{n}{2} \right\rfloor + 17 \right) + n \\ &= 2c \left(\frac{n}{2} + 17 \right) \lg \left(\frac{n}{2} + 17 \right) + n \\ &= c(n + 34) \lg \left(\frac{n + 34}{2} \right) + n \\ &= c(n + 34) \big(\lg (n + 34) - 1 \big) + n \\ &\leq c(n + 34) \big(\lg (\sqrt{2}n) - 1 \big) + n \end{aligned} \tag{3.46}$$

because for all sufficiently large values of n, say $n \geq 100$, it is the case that $\sqrt{2}n \geq n + 34$.

$$\begin{aligned}
T(n) &\leq c(n+34)\big(\lg(\sqrt{2}n) - 1\big) + n \qquad \text{from (3.46)} \\
&= c(n+34)\left(\lg n + \frac{1}{2}\lg 2 - 1\right) + n \\
&= c(n+34)\left(\lg n - \frac{1}{2}\right) + n \\
&= cn\lg n + 34c\lg n - \frac{cn}{2} - 17c + n \\
&\leq cn\lg n \qquad \text{provided that } 34c\lg n - \frac{cn}{2} - 17c + n \leq 0
\end{aligned}$$

In order $34c\lg n - \frac{cn}{2} - 17c + n = n\left(1 - \frac{c}{2}\right) + 34c\lg n - 17c$ to be non-positive for all sufficiently large n it suffices $\left(1 - \frac{c}{2}\right)$ to be negative because the linear function dominated the logarithmic function. A more detailed analysis is the following. Fix $c = 4$. The expression becomes $(-1)n + 136\lg n - 136$.

$$(-1)n + 136\lg n - 136 \leq 0 \;\Leftrightarrow\; n \geq 136(\lg n - 1) \;\Leftrightarrow\; \frac{n}{\lg n - 1} \geq 136$$

For $n = 65536 = 2^{16}$ the inequality holds:

$$\frac{2^{16}}{15} \geq 136$$

so we can finish the proof with choosing $n_0 = 65536$ and $c = 4$.

Part ii: Proof that $T(n) = \Omega(n\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn\lg n$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\geq 2d\left(\left\lfloor\frac{n}{2}\right\rfloor + 17\right)\lg\left(\left\lfloor\frac{n}{2}\right\rfloor + 17\right) + n \\
&\geq 2d\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + n \\
&= dn(\lg n - 1) + n \\
&= dn\lg n + (1-d)n \\
&\geq dn\lg n \qquad \text{provided that } 1 - d \geq 0 \;\Leftrightarrow\; d \leq 1 \qquad \square
\end{aligned}$$

Problem 54. *Solve*

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \qquad (3.47) \\
T(1) &= \Theta(1)
\end{aligned}$$

by the method of the recursion tree.

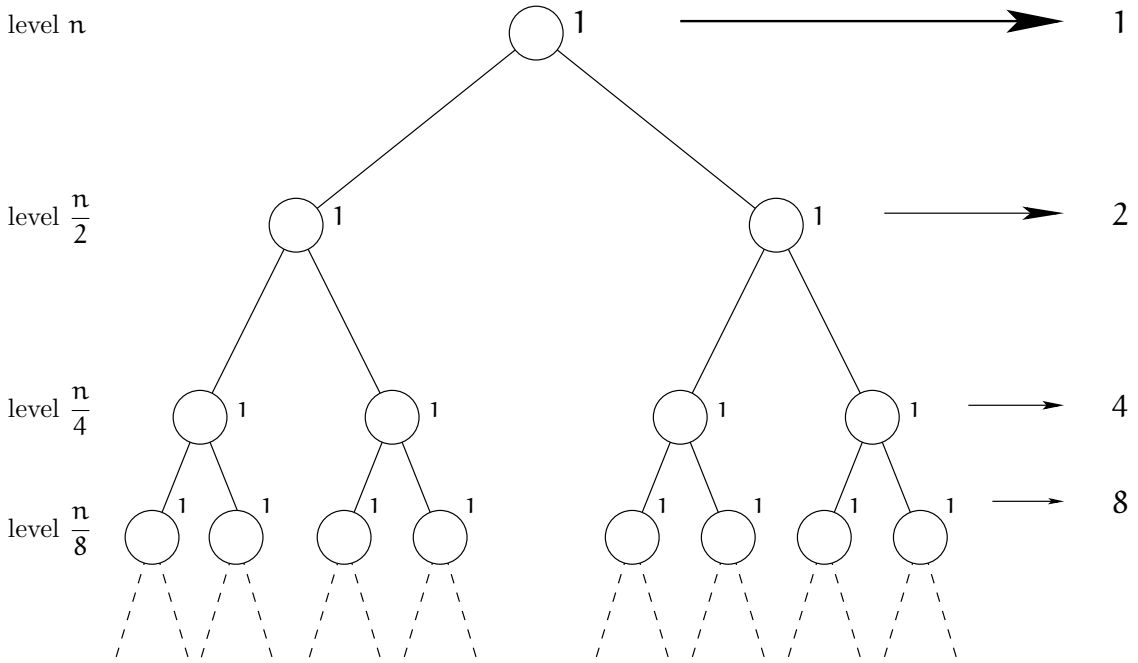


Figure 3.5: The recursion tree of $T(n) = 2T\left(\frac{n}{2}\right) + 1$.

Solution:
The recursion tree is shown on Figure 3.5. The solution is the sum over all levels:

$$T(n) = \underbrace{1 + 2 + 4 + 8 + \ldots}_{\text{the number of terms is the height of the tree}} \tag{3.48}$$

The height of the tree is the number of times the iterator

$$n \to \frac{n}{2}$$

is executed before the variable becomes 1. As we already saw, that number is $\lg n$[†]. So, (3.48) in fact is

$$\begin{aligned} T(n) &= \underbrace{1 + 2 + 4 + 8 + \ldots}_{(\lg n + 1)\text{ terms}} = 1 + 2 + 4 + 8 + \ldots + \frac{n}{2} + n \\ &= \sum_{i=0}^{\lg n} \frac{n}{2^i} = n\left(\sum_{i=0}^{\lg n} \frac{1}{2^i}\right) \leq n \underbrace{\left(\sum_{i=0}^{\infty} \frac{1}{2^i}\right)}_{2} = 2n \end{aligned}$$

We conclude that $T(n) = \Theta(n)$. □

However, that proof by the method of the recursion tree can be considered insufficiently precise because it involves several approximations and the use of imagination—the dot-dot-dot notations. Next we demonstrate a proof by induction of the same result. We may think

[†]Actually it is $\lfloor \lg n \rfloor$ but that is immaterial with respect to the asymptotic growth of $T(n)$.

of the proof with recursion tree as a mere way to derive a good guess to be verified formally by induction.

Problem 55. *Prove by induction on* n *that the solution to*

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\ T(1) &= \Theta(1) \end{aligned} \tag{3.49}$$

is $T(n) = \Theta(n)$.

Solution:
We prove separately that $T(n) = O(n)$ and $T(n) = \Omega(n)$.

Part i: Proof that $T(n) = O(n)$. For didactic purposes we will first make an unsuccessful attempt.

Part i, try 1: Assume there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \tag{3.50}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2} + 1 \\ &= cn + 1 \end{aligned}$$

Our proof ran into a problem: no matter what positive c we choose, it is not true that $cn + 1 \leq cn$, and thus (3.50) cannot be shown to hold. Of course, that failure does *not* mean our claim $T(n) = \Theta(n)$ is false. It simply means that (3.50) is inappropriate. We amend the situation by a technique known as *strenthening the claim.* It consists of stating an appropriate claim that is stronger than (3.50) and then proving it by induction. Intuitively, that stronger claim has to contain some minus sign in such a way that after applying the inductive hypothesis, there is a term like $-c$ that can "cope with" the $+1$.

Part i, try 2: Assume there exists positive constants b and c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn - b \tag{3.51}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2\left(c\frac{n}{2} - b\right) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \qquad \text{for any } b \text{ such that } -b + 1 \leq 0 \Leftrightarrow b \geq 1 \end{aligned}$$

Part ii: Proof that $T(n) = \Omega(n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq 2\left(d\frac{n}{2}\right) + 1 \\ &= dn + 1 \\ &\geq dn \end{aligned}$$

□

Problem 56. *Prove by induction on* n *that the solution to*

$$\begin{aligned} T(n) &= 2T(n-1) + n \\ T(1) &= \Theta(1) \end{aligned} \tag{3.52}$$

is $T(n) = \Theta(2^n)$.

Solution:
We prove separately that $T(n) = O(2^n)$ and $T(n) = \Omega(2^n)$.

Part i: Proof that $T(n) = O(2^n)$. For didactic purposes we will first make several unsuccessful attempts.

Part i, try 1: Assume there exists a positive constant c such that for all large enough n,

$$T(n) \leq c2^n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c2^{n-1} + n \\ &= c2^n + n \\ &\not\leq c2^n \text{ for any choice of positive } c \end{aligned}$$

Our proof failed so let us strenghten the claim.

Part i, try 2: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^n - b$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + n \\ &= c2^n - 2b + n \\ &\not\leq c2^n - b \text{ for any choice of positive } c \end{aligned}$$

The proof failed once again so let us try another strenghtening of the claim.

Part i, try 3: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^{n-b}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-b-1}) + n \\ &= c2^{n-b} + n \\ &\not\leq c2^{n-b} \text{ for any choice of positive } c \end{aligned}$$

Yet another failure and we try yet another strenghtening of the claim.

Part i, try 4: Assume there exists a positive constant c such that for all large enough n,

$$T(n) \leq c2^n - n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - (n-1)) + n \\ &= c2^n - n + 2 \\ &\not\leq c2^n - n \text{ for any choice of positive } c \end{aligned}$$

Part i, try 5: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^n - bn$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b(n-1)) + n \\ &= c2^n - 2bn + 2b + n \\ &= c2^n - bn + (1-b)n + 2b \\ &\leq c2^n - bn \text{ for any choice of } c > 0 \text{ and } b > 1 \end{aligned}$$

Success! At last we have managed to formulate a provable hypothesis.

Part ii: Proof that $T(n) = \Omega(n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \geq d2^n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq 2(d2^{n-1}) + n \\ &= d2^n + n \\ &\geq d2^n \end{aligned}$$

Success! Again we see that the strengthening of the claim is required only in one direction of the proof. □

The next three problems have the iterator

$$n \to \sqrt{n}$$

According to the table on page 46, that number of times this iterator is executed before n becomes some fixed constant is $\Theta(\lg\lg n)$. Note, however, that unless n is integer, this constant cannot be 1 because for real n, it is the case that $n > 1$ after any iteration. Therefore "$T(1) = \Theta(1)$" cannot be the initial condition if n is real. One way out of that is to change the initial conditions to

$$T(n) = \Theta(1) \text{ for } 2 \le n \le 4$$

Problem 57. *Solve*

$$T(n) = 2T(\sqrt{n}) + 1 \tag{3.53}$$

Solution:
Substitute n by 2^{2^m}, *i.e.* $m = \lg\lg n$ and $2^m = \lg n$. Then (3.53) becomes

$$T\left(2^{2^m}\right) = 2T\left(2^{\frac{2^m}{2}}\right) + 1$$

which is

$$T\left(2^{2^m}\right) = 2T\left(2^{2^{m-1}}\right) + 1 \tag{3.54}$$

Further substitute $T\left(2^{2^m}\right)$ by $S(m)$ and (3.54) becomes

$$S(m) = 2S(m-1) + 1 \tag{3.55}$$

But we know the solution to that recurrence. According to Problem 47, $S(m) = \Theta(2^m)$. Let us go back now to the original n and $T(n)$.

$$S(m) = \Theta(2^m) \;\Leftrightarrow\; T\left(2^{2^m}\right) = \Theta(\lg n) \;\Leftrightarrow\; T(n) = \Theta(\lg n)$$

□

Problem 58. *Solve*

$$T(n) = 2T(\sqrt{n}) + \lg n \tag{3.56}$$

Solution:
Substitute n by 2^m, *i.e.* $m = \lg n$. Then (3.56) becomes

$$T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m \tag{3.57}$$

Further substitute $T(2^m)$ by $S(m)$ and (3.57) becomes

$$S(m) = 2S\left(\frac{m}{2}\right) + m \tag{3.58}$$

Consider Problem 49 and Problem 50. They have solve the same recurrence, differing from (3.58) only in the way the division is rounded to integer. In Problem 49 the iterator is

$$n \to \left\lfloor \frac{n}{2} \right\rfloor$$

and in Problem 50 the iterator is

$$n \to \left\lceil \frac{n}{2} \right\rceil$$

Both Problem 49 and Problem 50 have $\Theta(n \lg n)$ solutions. We conclude the solution of (3.58) is $S(m) = \Theta(m \lg m)$, which is equivalent to $T(n) = \Theta(\lg n \lg\lg n)$. □

Problem 59. *Solve*

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.59}$$

Solution:
Let us unfold the recurrence:

$$T(n) = n + n^{\frac{1}{2}} T\left(n^{\frac{1}{2}}\right) \tag{3.60}$$

$$= n + n^{\frac{1}{2}} \left(n^{\frac{1}{2}} + n^{\frac{1}{4}} T\left(n^{\frac{1}{4}}\right)\right) \tag{3.61}$$

$$= 2n + n^{\frac{3}{4}} T\left(n^{\frac{1}{4}}\right) \tag{3.62}$$

$$= 2n + n^{\frac{3}{4}} \left(n^{\frac{1}{8}} + T\left(n^{\frac{1}{8}}\right)\right) \tag{3.63}$$

$$= 3n + n^{\frac{7}{8}} T\left(n^{\frac{1}{8}}\right) \tag{3.64}$$

$$\ldots \tag{3.65}$$

$$= in + n^{\left(1-\frac{1}{2^i}\right)} T\left(n^{\frac{1}{2^i}}\right) \tag{3.66}$$

As we already said, the maximum value of i, call it $i_{\max}$, is $i_{\max} = \lg\lg n$. But then $2^{i_{\max}} = \lg n$, therefore

$$n^{\left(1-\frac{1}{2^{i_{\max}}}\right)} = \frac{n}{n^{\frac{1}{2^{i_{\max}}}}} = \frac{n}{n^{\frac{1}{\lg n}}} = \frac{n}{2}$$

The derivation of the fact that $n^{\frac{1}{\lg n}} = 2$ is on page 16. So, for $i = i_{\max}$,

$$T(n) = (\lg\lg n)n + \frac{n}{2}T(c) \qquad c \text{ is some number such that } 2 \le c \le 4$$

But $T(c)$ is a constant, therefore $T(n) = \Theta(n \lg\lg n)$.

Let us prove the same result by induction.
Part i: Prove that $T(n) = O(n \lg\lg n)$, that is, there exists a positive constant c such that for all sufficiently large n,

$$T(n) \le cn \lg\lg n \tag{3.67}$$

Our inductive hypothesis then is

$$T(\sqrt{n}) \le c\sqrt{n} \lg\lg \sqrt{n} \tag{3.68}$$

We know by the definition of the problem that

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.69}$$

Apply (3.68) to (3.69) to get

$$\begin{aligned}
T(n) &\leq \sqrt{n}(c\sqrt{n}\lg\lg\sqrt{n}) + n \\
&= cn\lg\lg\sqrt{n} + n \\
&= cn\lg\left(\frac{1}{2}\lg n\right) + n \\
&= cn\lg\left(\frac{\lg n}{2}\right) + n \\
&= cn(\lg\lg n - 1) + n \\
&= cn\lg\lg n - cn + n \\
&\leq cn\lg\lg n \qquad \text{provided that } -cn + n \leq 0 \Leftrightarrow c \geq 1
\end{aligned}$$

Part ii: Prove that $T(n) = \Omega(n\lg\lg n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \geq dn\lg\lg n \tag{3.70}$$

Our inductive hypothesis then is

$$T(\sqrt{n}) \geq d\sqrt{n}\lg\lg\sqrt{n} \tag{3.71}$$

We know by the definition of the problem that

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.72}$$

Apply (3.71) to (3.72) to get

$$\begin{aligned}
T(n) &\geq \sqrt{n}(d\sqrt{n}\lg\lg\sqrt{n}) + n \\
&= dn\lg\lg\sqrt{n} + n \\
&= dn\lg\left(\frac{1}{2}\lg n\right) + n \\
&= dn\lg\left(\frac{\lg n}{2}\right) + n \\
&= dn(\lg\lg n - 1) + n \\
&= dn\lg\lg n - dn + n \\
&\geq dn\lg\lg n \qquad \text{provided that } -dn + n \geq 0 \Leftrightarrow d \leq 1
\end{aligned}$$

□

Problem 60. *Solve*

$$T(n) = n^2T\left(\frac{n}{2}\right) + 1 \tag{3.73}$$

Solution:
Unfold the recurrence:

$$
\begin{aligned}
T(n) &= n^2 T\left(\frac{n}{2}\right) + 1 \\
&= n^2 \left(\frac{n^2}{4} T\left(\frac{n}{4}\right) + 1\right) + 1 \\
&= \frac{n^4}{2^2} T\left(\frac{n}{2^2}\right) + n^2 + 1 \\
&= \frac{n^4}{2^2} \left(\frac{n^2}{2^4} T\left(\frac{n}{2^3}\right) + 1\right) + n^2 + 1 \\
&= \frac{n^6}{2^6} T\left(\frac{n}{2^3}\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^6}{2^6} \left(\frac{n^2}{2^6} T\left(\frac{n}{2^4}\right) + 1\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}} T\left(\frac{n}{2^4}\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}} \left(\frac{n^2}{2^8} T\left(\frac{n}{2^5}\right) + 1\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + \frac{n^2}{2^0} + \frac{n^0}{2^0} \\
&= \frac{n^{10}}{2^{5.4}} T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}} \\
&= \dots \\
&= \underbrace{\frac{n^{2i}}{2^{i(i-1)}} T\left(\frac{n}{2^i}\right)}_{A} + \underbrace{\frac{n^{2(i-1)}}{2^{(i-1)(i-2)}} + \frac{n^{2(i-2)}}{2^{(i-2)(i-3)}} \dots + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}}}_{B}
\end{aligned}
$$

The maximum value of i is $i_{max} = \lg n$. First we compute A with $i = \lg n$, having in mind $T(1)$ is some positive constant c.

$$A = \frac{n^{2\lg n}}{2^{\lg^2 n - \lg n}} T(1) = \frac{c 2^{\lg n} n^{2\lg n}}{2^{\lg^2 n}} = \frac{c.n.n^{2\lg n}}{2^{\lg^2 n}}$$

But

$$2^{\lg^2 n} = 2^{\lg n . \lg n} = 2^{\lg\left(n^{\lg n}\right)} = n^{\lg n} \tag{3.74}$$

Therefore

$$A = \frac{c.n.n^{2\lg n}}{n^{\lg n}} = \Theta(n^{1+\lg n}) \tag{3.75}$$

Consider B. Obviously, we can represent it as a sum in the following way:

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{n^{2((\lg n)-j)}}{2^{((\lg n)-j)((\lg n)-j-1)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{n^{2\lg n}}{2^{(\lg^2 n - j\lg n - j\lg n + j^2 - \lg n + j)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(2^{2j\lg n}\right)\left(2^{\lg n}\right)}{\left(2^{\lg^2 n}\right)\left(2^{j^2+j}\right)} \end{aligned} \tag{3.76}$$

But

$$2^{2j\lg n} = 2^{\lg\left(n^{2j}\right)} = n^{2j} \tag{3.77}$$

and

$$2^{\lg n} = n \tag{3.78}$$

Apply (3.74), (3.77), and (3.78) on (3.76) to obtain

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(n^{2j}\right)(n)}{\left(n^{\lg n}\right)\left(2^{j^2+j}\right)} \\ &= \sum_{j=1}^{\lg n} \frac{n^{1+\lg n}}{2^{j^2+j}} \\ &= \left(n^{1+\lg n}\right) \sum_{j=1}^{\lg n} \frac{1}{2^{j^2+j}} \\ &\le \left(n^{1+\lg n}\right) \sum_{j=1}^{\infty} \frac{1}{2^{j^2+j}} \\ &\le n^{1+\lg n} \qquad \text{since we know that } \sum_{j=1}^{\infty} \frac{1}{2^j} = 1 \\ &= \Theta\left(n^{1+\lg n}\right) \end{aligned} \tag{3.79}$$

From (3.75) and (3.79) it follows that

$$T(n) = \Theta\left(n^{1+\lg n}\right)$$

□

Problem 61. *Solve by unfolding*

$$T(n) = T(n-2) + 2\lg n \tag{3.80}$$

Solution:
Let us unfold the recurrence:

$$\begin{aligned} T(n) &= T(n-2) + 2\lg n \\ &= T(n-4) + 2\lg(n-2) + 2\lg n \\ &= T(n-6) + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \\ &= \ldots \\ &= T(c) + \ldots + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \end{aligned} \tag{3.81}$$

where c is either 1 or 2[†].

Case I: n is odd. Then $c = 1$ and (3.81) is:

$$2\lg n + 2\lg(n-2) + 2\lg(n-4) + \ldots + 2\lg 3 + T(1) \tag{3.82}$$

We approximate $T(1)$ with $0 = \lg 1$, which does not alter the asymptotic growth rate of (3.82), and thus (3.82) becomes:

$$\begin{aligned} &\lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \ldots + \lg 3^2 + \lg 1 = \\ &\lg\left(n^2(n-2)^2(n-4)^2 \ldots 3^2.1\right) = \\ &\lg\Big(\underbrace{n.n(n-2)(n-2)(n-4)(n-4)\ldots 5.5.3.3.1}_{n \text{ factors}}\Big) = T(n) \end{aligned} \tag{3.83}$$

Define

$$X(n) = \lg\Big(\underbrace{n(n-1)(n-2)(n-3)\ldots 3.2.1}_{n \text{ factors}}\Big) = \lg n!$$
$$Y(n) = \lg\Big(\underbrace{(n+1)n(n-1)(n-2)\ldots 4.3.2}_{n \text{ factors}}\Big) = \lg(n+1)!$$

and note that

$$X(n) \le T(n) \le Y(n) \tag{3.84}$$

because of the following inequalities between the corresponding factors inside the logarithms

$$\begin{array}{lccccccccc} X(n) = \lg(& n & n-1 & n-2 & n-3 & \ldots & 3 & 2 & 1 &) \\ & /\!/\backslash & /\!/\backslash & /\!/\backslash & /\!/\backslash & & /\!/\backslash & /\!/\backslash & /\!/\backslash & \\ T(n) = \lg(& n & n & n-2 & n-2 & \ldots & 3 & 3 & 1 &) \\ & /\!/\backslash & /\!/\backslash & /\!/\backslash & /\!/\backslash & & /\!/\backslash & /\!/\backslash & /\!/\backslash & \\ Y(n) = \lg(& n+1 & n & n-1 & n-2 & \ldots & 4 & 3 & 2 &) \end{array}$$

However, $X(n) = \Theta(n\lg n)$ and $Y(n) = \Theta((n+1)\lg(n+1)) = \Theta(n\lg n)$ by (1.48). Having in mind that and (3.84), $T(n) = \Theta(n\lg n)$ follows immediately.

[†]The initial conditions that define $T(1)$ and $T(2)$ are omitted.

Case II: n is even. Then $c = 2$ and (3.81) is:

$$2\lg n + 2\lg(n-2) + 2\lg(n-4) + \ldots + 2\lg 4 + T(2) \tag{3.85}$$

We approximate $T(2)$ with $1 = \lg 2$, which does not alter the asymptotic growth rate of (3.82), and thus (3.82) becomes:

$$\begin{aligned}
&\lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \ldots + \lg 4^2 + \lg 2 = \\
&\lg\left(n^2(n-2)^2(n-4)^2 \ldots 4^2.2\right) = \\
&\lg\Big(\underbrace{n.n(n-2)(n-2)(n-4)(n-4)\ldots 6.6.4.4.2}_{n-1 \text{ factors}}\Big) = T(n)
\end{aligned} \tag{3.86}$$

Define

$$X(n) = \lg\Big(\underbrace{n(n-1)(n-2)(n-3)\ldots 4.3.2}_{n-1 \text{ factors}}\Big) = \lg n!$$

$$Y(n) = \lg\Big(\underbrace{(n+1)n(n-1)(n-2)\ldots 5.4.3}_{n-1 \text{factors}}\Big) = \lg\frac{(n+1)!}{2} = \lg(n+1)! - 1$$

and note that

$$X(n) \le T(n) \le Y(n) \tag{3.87}$$

because of the following inequalities between the corresponding factors inside the logarithms

$$\begin{array}{lccccccccc}
X(n) = \lg\big(& n & n-1 & n-2 & n-3 & \ldots & 4 & 3 & 2 & \big) \\
 & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & \\
T(n) = \lg\big(& n & n & n-2 & n-2 & \ldots & 4 & 4 & 2 & \big) \\
 & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & \\
Y(n) = \lg\big(& n+1 & n & n-1 & n-2 & \ldots & 5 & 4 & 3 & \big)
\end{array}$$

However, $X(n) = \Theta(n\lg n)$ and $Y(n) = \Theta((n+1)\lg(n+1)) = \Theta(n\lg n)$ by (1.48). Having in mind that and (3.84), $T(n) = \Theta(n\lg n)$ follows immediately. □

Problem 62. *Solve by induction*

$$T(n) = T(n-2) + 2\lg n \tag{3.88}$$

Solution:
We use Problem 61 to guess the solution $T(n) = \Theta(n\lg n)$.

Part i: Proof that $T(n) = O(n\lg n)$, that is, there exists a positive constant c such that for all sufficiently large n,

$$T(n) \le cn\lg n \tag{3.89}$$

The following inequalities hold

$$\begin{aligned}
T(n) &\leq c(n-2)\lg(n-2) + 2\lg n \qquad \text{from the induction hypothesis} \\
&\leq c(n-2)\lg n + 2\lg n \\
&= cn\lg n - 2c\lg n + 2\lg n \\
&\leq cn\lg n \qquad \text{provided that } -2c\lg n + 2\lg n \leq 0 \Leftrightarrow c \geq 1
\end{aligned}$$

Part ii: Proof that $T(n) = \Omega(n\lg n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \geq dn\lg n \tag{3.90}$$

It is the case that

$$\begin{aligned}
T(n) &\geq d(n-2)\lg(n-2) + 2\lg n \qquad \text{from the induction hypothesis} \\
&= (dn - 2d)\lg(n-2) + 2\lg n \\
&= dn\lg(n-2) + 2(\lg n - d\lg(n-2)) \qquad (3.91)
\end{aligned}$$

Having in mind (3.90) and (3.91), our goal is to show that

$$\begin{aligned}
dn\lg(n-2) + 2(\lg n - d\lg(n-2)) &\geq dn\lg n \Leftrightarrow \\
dn\lg(n-2) - dn\lg n + 2(\lg n - d\lg(n-2)) &\geq 0 \Leftrightarrow \\
\underbrace{d\lg\left(\frac{n-2}{n}\right)^n}_{A} + \underbrace{2\lg\frac{n}{(n-2)^d}}_{B} &\geq 0 \qquad (3.92)
\end{aligned}$$

Let us first evaluate A when n grows infinitely:

$$\lim_{n\to\infty} d\lg\left(\frac{n-2}{n}\right)^n = d\lim_{n\to\infty}\lg\left(1+\frac{-2}{n}\right)^n = d\lg\lim_{n\to\infty}\left(1+\frac{-2}{n}\right)^n = d\lg e^{-2} = -2d\lg e$$

Now consider B when n grows infinitely:

$$\lim_{n\to\infty} 2\lg\frac{n}{(n-2)^d} = 2\lg\lim_{n\to\infty}\frac{n}{(n-2)^d} \tag{3.93}$$

Note that for any d such that $0 < d < 1$, (3.93) is $+\infty$. For instance, for $d = \frac{1}{2}$, (3.93) becomes

$$2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\frac{n^{\frac{1}{2}}}{(n-2)^{\frac{1}{2}}}\right) =$$

$$2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\left(\frac{n}{n-2}\right)^{\frac{1}{2}}\right) =$$

$$2\lg\left(\left(\lim_{n\to\infty} n^{\frac{1}{2}}\right)\underbrace{\left(\lim_{n\to\infty}\left(\frac{1}{1-\frac{2}{n}}\right)^{\frac{1}{2}}\right)}_{1}\right) = +\infty$$

It follows inequality (3.92) is true for any choice of d such that $0 < d < 1$, say, $d = \frac{1}{2}$, because A by absolute value is limited by a constant, and B grows infinitely. And that concludes the proof of (3.89). □

⁘ NB ⁘ *The proof by induction in **Part ii** of the solution to Problem 61 is tricky. Consider (3.91):*

$$dn \lg (n-2) + 2(\lg n - d \lg (n-2))$$

Typically, we deal with logarithms of additions or differences by approximating the additions or differences with multiplications or fractions in such a way that the inequality holds in the desired direction. But notice that if we approximate $n-2$ inside the above logarithms with any *fraction $\frac{n}{\alpha}$, for any positive constant α, it must be the case that $\alpha > 1$, otherwise the inequality would not be in the direction we want. Here is what happens when we substitute $n-2$ with $\frac{n}{\alpha}$ in the logarithm on the left:*

$$dn \lg \frac{n}{\alpha} + 2(\lg n - d \lg (n-2)) = dn \lg n - dn \lg \alpha + 2(\lg n - d \lg (n-2))$$

To accomplish the proof, we have to show the latter is greater than or equal to $dn \lg n$; and to show that, we have to show that the term $-dn \lg \alpha + 2(\lg n - d \lg (n-2))$ is positive. But that is not true! $d > 0$ and $\alpha > 1$, therefore $-dn \lg \alpha < 0$ for all $n > 0$. And the asymptotic behaviour of $-dn \lg \alpha + 2(\lg n - d \lg (n-2))$ is determined by $-dn \lg \alpha$ because the linear function dominates the logarithmic function for all sufficiently large n. Therefore, we need a more sophisticated technique, based on analysis.

Problem 63. *Solve by unfolding*

$$T(n) = T(n-1) + \lg n$$

Solution:

$$\begin{aligned}
T(n) &= T(n-1) + \lg n \\
&= T(n-2) + \lg (n-1) + \lg n \\
&= T(n-3) + \lg (n-2) + \lg (n-1) + \lg n \\
&\dots \\
&= \underbrace{T(1)}_{\Theta(1)} + \lg 2 + \lg 3 + \ldots + \lg (n-2) + \lg (n-1) + \lg n \\
&= \Theta(1) + \lg (2.3 \ldots (n-2)(n-1)n) \\
&= \Theta(1) + \lg n! \\
&= \Theta(1) + \Theta(n \lg n) \qquad \text{by (1.48)} \\
&= \Theta(n \lg n)
\end{aligned}$$

□

Problem 64. *Solve by unfolding*

$$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n \tag{3.94}$$

Solution:

$$\begin{aligned}
T(n) &= n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) \\
&= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{\left\lfloor \frac{n}{4} \right\rfloor}{4} \right\rfloor\right)\right) \\
&= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{n}{16} \right\rfloor\right)\right) \qquad \text{because } \left\lfloor \frac{\left\lfloor \frac{n}{4} \right\rfloor}{4} \right\rfloor = \left\lfloor \frac{n}{16} \right\rfloor \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9T\left(\left\lfloor \frac{n}{16} \right\rfloor\right) \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9\left\lfloor \frac{n}{16} \right\rfloor + 27T\left(\left\lfloor \frac{n}{64} \right\rfloor\right) \\
&\dots \\
&= \underbrace{3^0\left\lfloor \frac{n}{4^0} \right\rfloor + 3^1\left\lfloor \frac{n}{4^1} \right\rfloor + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + \ldots + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor}_{P(n)} + \underbrace{3^i T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right)}_{\text{remainder}} \qquad (3.95)
\end{aligned}$$

The maximum value for i, let us call it i_{max}, is achieved when $\left\lfloor \frac{n}{4^i} \right\rfloor$ becomes 1. It follows $i_{max} = \lfloor \log_4 n \rfloor$. Let us estimate the main part $P(n)$ and the remainder of (3.95) for $i = i_{max}$.

- To estimate $P(n)$, define

$$\begin{aligned}
X(n) &= 3^0\left(\frac{n}{4^0}\right) + 3^1\left(\frac{n}{4^1}\right) + 3^2\left(\frac{n}{4^2}\right) + \ldots + 3^{i_{max}-1}\left(\frac{n}{4^{i_{max}-1}}\right) \\
Y(n) &= 3^0\left(\frac{n}{4^0} - 1\right) + 3^1\left(\frac{n}{4^1} - 1\right) + 3^2\left(\frac{n}{4^2} - 1\right) + \ldots + 3^{i_{max}-1}\left(\frac{n}{4^{i_{max}-1}} - 1\right)
\end{aligned}$$

  Clearly, $X(n) \geq P(n) \geq Y(n)$. But

$$\begin{aligned}
X(n) &= n\left(\sum_{j=0}^{i_{max}-1}\left(\frac{3}{4}\right)^j\right) \\
&\leq n\sum_{j=0}^{\infty}\left(\frac{3}{4}\right)^j \\
&= n\frac{1}{1-\frac{3}{4}} \\
&= \Theta(n)
\end{aligned}$$

  and

$$\begin{aligned}
Y(n) &= n\left(\sum_{j=0}^{i_{max}-1}\left(\frac{3}{4}\right)^j\right) - \sum_{j=0}^{i_{max}-1} 3^j \\
&= n\,\Theta(1) - \Theta\left(3^{i_{max}-1}\right) \qquad \text{by Corollary 1 on page 20} \\
&= \Theta(n) - \Theta\left(3^{\log_4 n}\right) \qquad \text{since } i_{max} = \lfloor \log_4 n \rfloor \\
&= \Theta(n) - \Theta(n^{\log_4 3}) = \Theta(n)
\end{aligned}$$

  Then it has to be the case that $P(n) = \Theta(n)$.

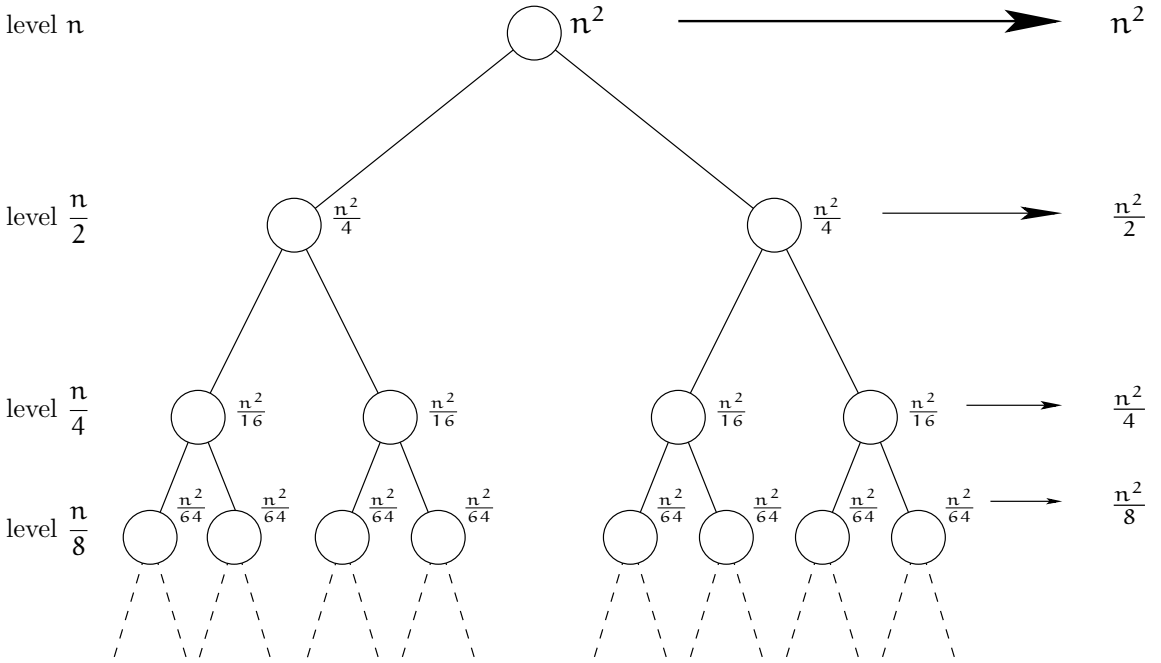


Figure 3.6: The recursion tree of $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

- To estimate the remainder, consider the two factors in it:

$$3^{i_{\max}} = 3^{\lfloor \log_4 n \rfloor} = \Theta(3^{\log_4 n}) = \Theta(n^{\log_3 4})$$
$$T\left(\left\lfloor \frac{n}{4^{i_{\max}}} \right\rfloor\right) = T(1) = \Theta(1)$$

  It follows the remainder is $\Theta(3^{\log_4 n}) = o(n)$.

Therefore, $T(n) = \Theta(n) + o(n) = \Theta(n)$. □

Problem 65. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

by the method of the recursion tree.

Solution:
The recursion tree is shown on Figure 3.6. The solution is the sum

$$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \ldots \leq n^2 \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n^2$$

It follows $T(n) = \Theta(n^2)$. □

Problem 66. *Solve*

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

by the method of the recursion tree.

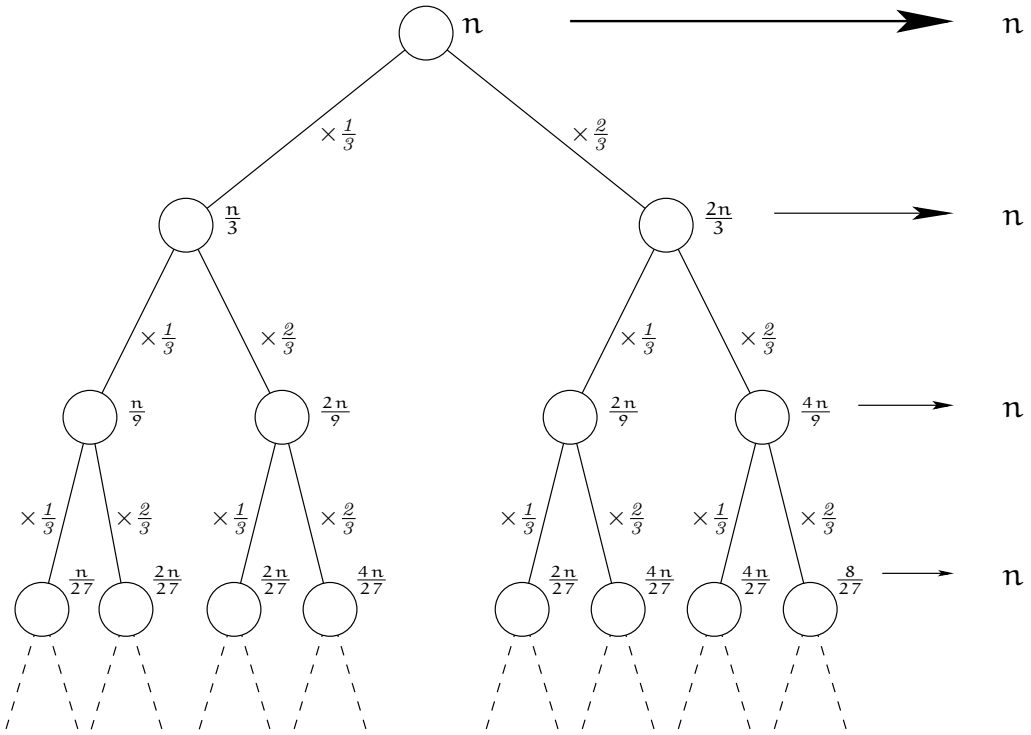


Figure 3.7: The recursion tree of $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$.

Solution:
The recursion tree is shown on Figure 3.7. This time the tree is not complete so we do not write the levels on the left side in terms of n (as we did on Figure 3.6). Rather, the level of each node is the distance between it and the root. Thus the equidistant with respect to the root nodes are at the same level. Think of the tree as an ordered tree. That is, if a node has any children we distinguish between the left and the right child. The value of the left child is the value of the parent multiplied by $\frac{1}{3}$ and the value of the right child is the value of the parent multiplied by $\frac{2}{3}$. It is trivial to prove by induction that for each level such that all the nodes at this level exist, the sum of the values at that level is n. However, we cannot obtain the answer immediately through mulitplying n by the height because the tree is not balanced. The maximum distance between the root and any leaf is achieved along the rightmost path (starting at the root, always take the right choice; see Figure 3.7) and the minimum distance, by the leftmost path. The length of the leftmost path is determined by the iterator

$$n \to \frac{n}{3}$$

which is executed $\Theta(\log_3 n)$ times before reaching any fixed in advance constant. The length of the rightmost path is determined by the iterator

$$n \to \frac{2n}{3} = \frac{n}{\frac{3}{2}}$$

which is executed $\Theta\left(\log_{\frac{3}{2}} n\right)$ times before reaching any fixed in advance constant.

Let $\mathcal{T}$ be the recursion tree. Construct two balanced trees $\mathcal{T}_1$ and $\mathcal{T}_2$ such that the height of $\mathcal{T}_1$ is $\Theta(\log_3 n)$ and the height of $\mathcal{T}_2$ is $\Theta\left(\log_{\frac{3}{2}} n\right)$. Suppose that each level in $\mathcal{T}_1$ and $\mathcal{T}_2$ is associated with some value n – it does not matter for what reason, just assume each level "costs" n. Let $S_i(n)$ be the sum of those costs in $\mathcal{T}_i$ over all levels for $i = 1, 2$. Clearly,

$$S_1(n) = n \times \Theta(\log_3 n) = \Theta(n \log_3 n) = \Theta(n \lg n)$$
$$S_2(n) = n \times \Theta\left(\log_{\frac{3}{2}} n\right) = \Theta\left(n \log_{\frac{3}{2}} n\right) = \Theta(n \lg n)$$

To conlude the solution, note that $S_1(n) \leq T(n) \leq S_2(n)$ because $\mathcal{T}_1$ can be considered a subtree of $\mathcal{T}$ and $\mathcal{T}$ can be considered a subtree of $\mathcal{T}_2$. Then $T(n) = \Theta(n \lg n)$. □

Problem 67. *Solve by unfolding*

$$T(n) = T(n - a) + T(a) + n \qquad a = const,\ a \geq 1$$

Solution:
We assume a is integer[†] and the initial conditions are

$$T(1) = \Theta(1)$$
$$T(2) = \Theta(1)$$
$$\cdots$$
$$T(a) = \Theta(1)$$

Let us unfold the recurrence.

$$\begin{aligned}
T(n) &= T(n-a) + T(a) + n \\
&= (T(n-2a) + T(a) + n - a) + T(a) + n \\
&= T(n-2a) + 2T(a) + 2n - a \\
&= (T(n-3a) + T(a) + n - 2a) + 2T(a) + 2n - a \\
&= T(n-3a) + 3T(a) + 3n - 3a \\
&= (T(n-4a) + T(a) + n - 4a) + 3T(a) + 3n - 3a \\
&= T(n-4a) + 4T(a) + 4n - 6a \\
&= (T(n-5a) + T(a) + n - 4a) + 4T(a) + 4n - 6a \\
&= T(n-5a) + 5T(a) + 5n - 10a \\
&\cdots \\
&= T(n-ia) + iT(a) + in - \frac{1}{2}i(i-1)a
\end{aligned} \tag{3.96}$$

Let the maximum value i takes be $i_{\max}$. Consider the iterator

$$n \to n - a$$

[†]It is not essential to postulate a is integer. The problems makes sense even if a is just a positive real. If that is the case the initial conditions have to be changed to cover some interval with length a, *e.g.* $T(i) = \text{const.}$ if $i \in (0, a]$.

It maps every $n > a$, $n \in \mathbb{N}$, to a unique number from $\{1, 2, \ldots, a\}$. Let that number be called k. So $i_{\max}$ is the number of times the iterator is executed until the variable becomes k. If $n \bmod a \neq 0$ then k is $n \bmod a$, otherwise k is a[†]. It follows that

$$i_{\max} = \begin{cases} \left\lfloor \frac{n}{a} \right\rfloor, & \text{if } n \bmod a \neq 0 \\ \frac{n}{a} - 1, & \text{else} \end{cases}$$

That is equivalent to

$$i_{\max} = \left\lceil \frac{n}{a} \right\rceil - 1$$

Subsituting i with $\left\lceil \frac{n}{a} \right\rceil - 1$ in (3.96), we get

$$T(k) + \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) T(a) + \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) n - \frac{1}{2} \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) \left(\left\lceil \frac{n}{a} \right\rceil - 1 - 1 \right) a \tag{3.97}$$

The growth rate of (3.97) is determined by

$$n \left\lceil \frac{n}{a} \right\rceil - \frac{1}{2} \left\lceil \frac{n}{a} \right\rceil \left\lceil \frac{n}{a} \right\rceil = \Theta(n^2)$$

It follows $T(n) = \Theta(n^2)$. □

Problem 68. *Solve*

$$T(n) = T(\alpha n) + T((1-\alpha)n) + n, \qquad \alpha = const.,\ 0 < \alpha < 1 \tag{3.98}$$

by the method of the recursion tree.

Solution:
Define that $1 - \alpha = \beta$. Obviously, $0 < \beta < 1$ and (3.98) becomes

$$T(n) = T(\alpha n) + T(\beta n) + n \tag{3.99}$$

The recursion tree of (3.99) is shown on Figure 3.8. The solution is completely analogous to the solution of Problem 66. The level of each node is the distance between it and the root. The sum of the costs at every level such that all nodes at that levels exist, is n. More precisely, at level i the sum is $(\alpha + \beta)^i n = n$. The tree is not complete. Assume without loss of generality that $\alpha \leq \beta$ and think of the tree as an ordered tree. The shortest path from the root to any leaf is the leftmost one, *i.e.* "follow the alphas", and the longest path is the rightmost one. The length of the shortest path is $\log_{\left(\frac{1}{\alpha}\right)} n$ and of the longest path, $\log_{\left(\frac{1}{\beta}\right)} n$. We prove that $T(n) = \Theta(n \lg n)$ just as in Problem 66 by considering two other trees, one that is a subgraph of the current one and one that isa supergraph of the current one. Since the first of then has sum of the costs $n \times \Theta\left(\log_{\left(\frac{1}{\alpha}\right)} n\right) = \Theta(n \lg n)$ and the second one, $n \times \Theta\left(\log_{\left(\frac{1}{\beta}\right)} n\right) = \Theta(n \lg n)$, it follows $T(n) = \Theta(n \lg n)$. □

[†]Not $n \bmod a$, which is 0.

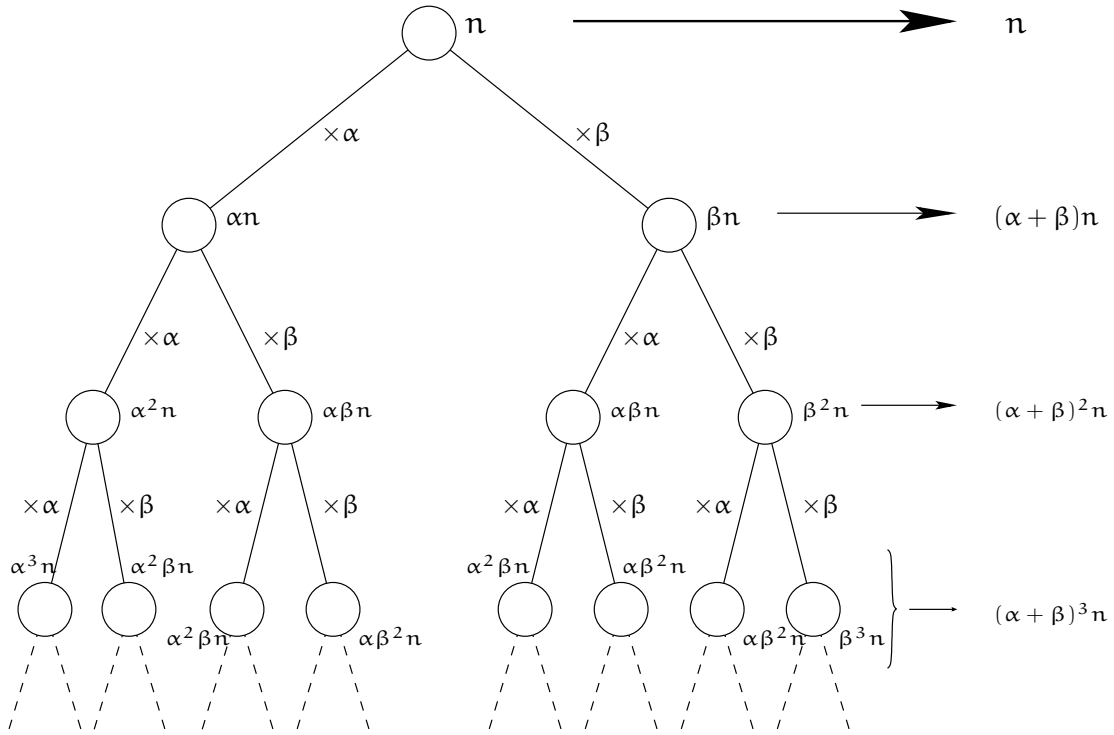


Figure 3.8: The recursion tree of $T(n) = T(\alpha n) + T(\beta n) + n$ where $0 < \alpha, \beta < 1$ and $\alpha + \beta = 1$.

Problem 69. *Solve by unfolding*

$$T(n) = T(n-1) + \frac{1}{n} \tag{3.100}$$

Solution:
Before we commence the unfolding check the definition of the harmonic series, the partial sum H_n of the harmonic series, and its order of growth $\Theta(\lg n)$ on page 248.

$$\begin{aligned}
T(n) &= T(n-1) + \frac{1}{n} \\
&= T(n-2) + \frac{1}{n-1} + \frac{1}{n} \\
&= T(n-3) + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n} \\
&\dots \\
&= T(1) + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n} \\
&= T(1) - 1 + \underbrace{1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}}_{H_n} \\
&= O(1) + H_n \\
&= O(1) + \Theta(\lg n) \\
&= \Theta(\lg n)
\end{aligned}$$

□

Problem 70. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + 1$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + 1 \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + 1\right) + 1 \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1} + 1 \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + 1\right) + \frac{n}{n+1} + 1 \\
&= \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4) + 1\right) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{n-3}{n+1}T(n-4) + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \qquad (3.101)
\end{aligned}$$

If we go on like that down to $T(1)$, (3.101) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \ldots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \ldots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + \frac{n+1}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=3}^{n+1} i \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n+1} i\right) - 3\right) \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\frac{(n+1)(n+2)}{2} - 3\right) \\
&= \frac{1}{n+1}\left(4T(1) + (n^2+3n+2) - 6\right) \\
&= \frac{n^2+3n+4T(1)-4}{n+1} \\
&= \underbrace{\frac{n^2}{n+1}}_{\Theta(n)} + \underbrace{\frac{3n}{n+1}}_{\Theta(1)} + \underbrace{\frac{4T(1)-4}{n+1}}_{O(1)} \\
&= \Theta(n)
\end{aligned}$$

So, $T(n) = \Theta(n)$. □

Problem 71. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + n^2$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + n^2 \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + (n-1)^2\right) + n^2 \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + (n-2)^2\right) + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-2}{n+1}T(n-3) + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4) + (n-3)^2\right) + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-3}{n+1}T(n-4) + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2
\end{aligned} \tag{3.102}$$

If we go on like that down to $T(1)$, (3.102) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3.2^2}{n+1} + \frac{4.3^2}{n+1} + \ldots \\
&\qquad + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{2}{n+1}T(1) + \frac{3.2^2}{n+1} + \frac{4.3^2}{n+1} + \ldots \\
&\qquad + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + \frac{(n+1)n^2}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=3}^{n+1} i(i-1)^2 \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n+1} i(i-1)^2\right) - 2\right) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n+1} i(i-1)^2 \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n+1} (i^3 - 2i^2 + i) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\sum_{i=1}^{n+1} i^3 - 2\sum_{i=1}^{n+1} i^2 + \sum_{i=1}^{n+1} i\right)
\end{aligned} \tag{3.103}$$

Having in mind (8.26), (8.27), and (8.28) on page 249, (3.103) becomes

$$\begin{aligned}
&\frac{2T(1)-2}{n+1}+\frac{1}{n+1}\left(\frac{(n+1)^2(n+2)^2}{4}-2\frac{(n+1)(n+2)(2n+3)}{6}+\frac{(n+1)(n+2)}{2}\right)\\
&=\underbrace{\frac{2T(1)-2}{n+1}}_{O(1)}+\underbrace{\frac{(n+1)(n+2)^2}{4}}_{\Theta(n^3)}-\underbrace{\frac{(n+2)(2n+3)}{3}}_{\Theta(n^2)}+\underbrace{\frac{n+2}{2}}_{\Theta(n)}\\
&=\Theta(n^3)
\end{aligned}$$

So, $T(n)=\Theta(n^3)$. □

Problem 72. *Solve by unfolding*

$$T(n)=\frac{n}{n+1}T(n-1)+\sqrt{n} \tag{3.104}$$

where $\sqrt{n}$ *stands for either* $\lfloor\sqrt{n}\rfloor$ *or* $\lceil\sqrt{n}\rceil$.

Solution:

$$\begin{aligned}
T(n)&=\frac{n}{n+1}T(n-1)+\sqrt{n}\\
&=\frac{n}{n+1}\left(\frac{n-1}{n}T(n-2)+\sqrt{n-1}\right)+\sqrt{n}\\
&=\frac{n-1}{n+1}T(n-2)+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3)+\sqrt{n-2}\right)+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-2}{n+1}T(n-3)+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4)+\sqrt{n-3}\right)+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-3}{n+1}T(n-4)+\frac{(n-2)\sqrt{n-3}}{n+1}+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}
\end{aligned} \tag{3.105}$$

If we go on like that down to $T(1)$, (3.105) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3\sqrt{2}}{n+1} + \frac{4\sqrt{3}}{n+1} + \ldots \\
&\qquad + \frac{(n-2)\sqrt{n-3}}{n+1} + \frac{(n-1)\sqrt{n-2}}{n+1} + \frac{n\sqrt{n-1}}{n+1} + \sqrt{n} \\
&= \frac{2}{n+1}T(1) + \frac{3\sqrt{2}}{n+1} + \frac{4\sqrt{3}}{n+1} + \ldots \\
&\qquad + \frac{(n-2)\sqrt{n-3}}{n+1} + \frac{(n-1)\sqrt{n-2}}{n+1} + \frac{n\sqrt{n-1}}{n+1} + \frac{(n+1)\sqrt{n}}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=2}^{n}(i+1)\sqrt{i} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n}(i+1)\sqrt{i}\right) - 2\right) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n}(i+1)\sqrt{i} \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n}(i\sqrt{i}+\sqrt{i}) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\sum_{i=1}^{n} i\sqrt{i} + \sum_{i=1}^{n}\sqrt{i}\right) \qquad (3.106)
\end{aligned}$$

But we know that

$$\sum_{i=1}^{n}\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{3}{2}}\right) \qquad \text{by (8.10) on page 240.}$$

$$\sum_{i=1}^{n}\left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{3}{2}}\right) \qquad \text{by (8.12) on page 241.}$$

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{5}{2}}\right) \qquad \text{by (8.15) on page 244.}$$

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{5}{2}}\right) \qquad \text{by (8.19) on page 247.}$$

Therefore, regardless of whether "$\sqrt{n}$" in (3.104) stands for $\lfloor\sqrt{n}\rfloor$ or $\lceil\sqrt{n}\rceil$,

$$\begin{aligned}
T(n) &= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\Theta\left(n^{\frac{5}{2}}\right) + \Theta\left(n^{\frac{5}{2}}\right)\right) \text{ by substituting into (3.106)} \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\Theta\left(n^{\frac{5}{2}}\right)\right) \\
&= O(1) + \Theta\left(n^{\frac{3}{2}}\right)
\end{aligned}$$

So, $T(n) = \Theta\left(n^{\frac{3}{2}}\right)$. □

Problem 73. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + \lg n \tag{3.107}$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + \lg n \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + \lg(n-1)\right) + \lg n \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + \lg(n-2)\right) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \dots \\
&= \underbrace{\frac{2}{n+1}T(1)}_{A} + \underbrace{\frac{3}{n+1}\lg 2 + \frac{4}{n+1}\lg 3 + \dots + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n}_{B}
\end{aligned}$$

Clearly, $A = O(1)$. Consider B.

$$\begin{aligned}
B &= \frac{3}{n+1}\lg 2 + \frac{4}{n+1}\lg 3 + \dots + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \frac{n+1}{n+1}\lg n \\
&= \frac{1}{n+1}\underbrace{(3\lg 2 + 4\lg 3 + \dots + (n-1)\lg(n-2) + n\lg(n-1) + (n+1)\lg n)}_{C}
\end{aligned}$$

Now consider C.

$$\begin{aligned}
C &= 3\lg 2 + 4\lg 3 + \dots + (n-1)\lg(n-2) + n\lg(n-1) + (n+1)\lg n \\
&= \underbrace{\lg 2 + \lg 3 + \dots + \lg(n-1) + \lg n}_{D} + \underbrace{2\lg 2 + 3\lg 3 + \dots + (n-1)\lg(n-1) + n\lg n}_{E}
\end{aligned}$$

But $D = \Theta(n \lg n)$ (see Problem 141 on page 232) and $E = \Theta(n^2 \lg n)$ (see Problem 142 on page 232). It follows that $C = \Theta(n^2 \lg n)$, and so $B = \Theta(n \lg n)$. We conclude that

$$T(n) = \Theta(n \lg n)$$ □

Problem 74. *Solve*

$$T(1) = \Theta(1) \tag{3.108}$$
$$T(2) = \Theta(1) \tag{3.109}$$
$$T(n) = T(n-1).T(n-2) \tag{3.110}$$

Solution:
Unlike the problems we encountered so far, the aymptotic growth rate of $T(n)$ in this problem depends on the concrete values of the constants in (3.108) and (3.109). It is easy to see that if $T(1) = T(2) = 1$ then $T(n) = 1$ for all positive n. So let us postulate that

$$T(1) = c \tag{3.111}$$
$$T(2) = d \tag{3.112}$$

where c and d are some positive constants. Then

$$\begin{aligned}
&T(3) = T(2).T(1) = cd \\
&T(4) = T(3).T(2) = cd^2 \\
&T(5) = T(4).T(3) = c^2d^3 \\
&T(6) = T(5).T(4) = c^3d^5 \\
&T(7) = T(6).T(5) = c^5d^8 \\
&\ldots
\end{aligned}$$

The degrees that appear in this sequence look like the Fibonacci number (see the definition on page 248). Indeed, it is trivial to prove by induction that

$$\begin{aligned}
&T(1) = c \\
&T(n) = d^{F_{n-1}}c^{F_{n-2}}, \text{ for all } n > 1
\end{aligned} \tag{3.113}$$

Define

$$\begin{aligned}
a &= c^{\frac{1}{\sqrt{5}}} \\
b &= d^{\frac{1}{\sqrt{5}}}
\end{aligned}$$

and derive

$$\begin{aligned}
T(n) &= \Theta\left(b^{\phi^{n-1}}\right)\Theta\left(a^{\phi^{n-2}}\right) && \text{applying (8.20) on page 248 on (3.113)} \\
&= \Theta\left(b^{\phi^{n-1}}a^{\phi^{n-2}}\right) \\
&= \Theta\left(b^{\phi.\phi^{n-2}}a^{\phi^{n-2}}\right) \\
&= \Theta\left(k^{\phi^{n-2}}a^{\phi^{n-2}}\right) && \text{defining that } b^{\phi} = k \\
&= \Theta\left((ak)^{\phi^{n-2}}\right)
\end{aligned} \tag{3.114}$$

Depending on how detalied analysis we need, we may stop right here. However, we can go on a little further because depending on what a and k are, (3.113) can have dramatically different asymptotic growth.

- If $ak > 1$, $T(n) \xrightarrow[n \to +\infty]{} \infty$.
- If $ak = 1$, $T(n) = 1$ for all positive n, thus $T(n) = \Theta(1)$.
- If $ak < 1$, $T(n) \xrightarrow[n \to +\infty]{} 0$, thus $T(n) = O(1)$. □

Problem 75. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{n-1} T(i) + 1$$

Solution:
By definition,

$$T(n) = T(n-1) + T(n-2) + \ldots + T(2) + T(1) + 1 \quad (3.115)$$
$$T(n-1) = \quad T(n-2) + \ldots + T(2) + T(1) + 1 \quad (3.116)$$

Subtract 3.116 from 3.115 to obtain

$$T(n) - T(n-1) = T(n-1)$$

So, the original recurrence is equivalent to the following one:

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n-1)$$

It is trivial to show that $T(n) = \Theta(2^n)$, either by induction or by the method with the characteristic equation. □

Problem 76. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{n-1} (T(i) + T(n-i)) + 1$$

Solution:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} (T(i) + T(n-i)) + 1 \\ &= \underbrace{\sum_{i=1}^{n-1} T(i)}_{T(1)+T(2)+\ldots+T(n-1)} + \underbrace{\sum_{i=1}^{n-1} T(n-i)}_{T(n-1)+T(n-2)+\ldots+T(1)} + 1 \\ &= 2\sum_{i=1}^{n-1} T(i) + 1 \end{aligned}$$

Having in mind the latter result, we proceed as in the previous problem.

$$T(n) = 2T(n-1) + 2T(n-2) + \ldots + 2T(2) + 2T(1) + 1 \quad (3.117)$$
$$T(n-1) = \quad 2T(n-2) + \ldots + 2T(2) + 2T(1) + 1 \quad (3.118)$$

Subtract 3.118 from 3.117 to obtain

$$T(n) - T(n-1) = 2T(n-1)$$

So, the original recurrence is equivalent to the following one:

$$T(1) = \Theta(1)$$
$$T(n) = 3T(n-1)$$

It is trivial to show that $T(n) = \Theta(3^n)$, either by induction or by the method of the characteristic equation. □

Problem 77. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = nT(n-1) + 1$$

Solution:

$$\begin{aligned}
T(n) &= nT(n-1) + 1 \\
&= n((n-1)T(n-2) + 1) + 1 \\
&= n(n-1)T(n-2) + n + 1 \\
&= n(n-1)((n-2)T(n-3) + 1) + n + 1 \\
&= n(n-1)(n-2)T(n-3) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)((n-3)T(n-4) + 1) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2) + n(n-1) + n + 1 \\
&\dots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!}{(n-i+1)!} + \frac{n!}{(n-i+2)!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \qquad (3.119)
\end{aligned}$$

Clearly, the maximum value i achieves is $i_{max} = n-1$. For $i = i_{max}$, (3.119) becomes:

$$\begin{aligned}
T(n) &= \frac{n!}{1!}T(1) + \frac{n!}{2!} + \frac{n!}{3!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&= n! \times \underbrace{\left(\frac{T(1)}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{(n-1)!} + \frac{1}{n!}\right)}_{A}
\end{aligned}$$

We claim A is bounded by a constant. To see why, note that the series $\sum_{i=1}^{\infty} \frac{1}{i!}$ is convergent because the geometric series $\sum_{i=1}^{\infty} \frac{1}{2^i}$ is convergent and $i! > 2^i$ for all $i > 3$. Therefore,

$$T(n) = \Theta(n!)$$

□

Problem 78. *Solve by unfolding*

$$T(1) = \Theta(1)$$
$$T(n) = nT(n-1) + 2^n$$

Solution:

$$\begin{aligned}
T(n) &= nT(n-1) + 2^n \\
&= n((n-1)T(n-2) + 2^{n-1}) + 2^n \\
&= n(n-1)T(n-2) + n2^{n-1} + 2^n \\
&= n(n-1)((n-2)T(n-3) + 2^{n-2}) + n2^{n-1} + n2^n \\
&= n(n-1)(n-2)T(n-3) + n(n-1)2^{n-2} + n2^{n-1} + n2^n \\
&= n(n-1)(n-2)((n-3)T(n-4) + 2^{n-3}) + n(n-1)2^{n-2} + n2^{n-1} + n2^n \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2)2^{n-3} + n(n-1)2^{n-2} + n2^{n-1} + n2^n \\
&\dots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!\,2^{n-i+1}}{(n-i+1)!} + \frac{n!\,2^{n-i+2}}{(n-i+2)!} + \dots + \frac{n!\,2^{n-1}}{(n-1)!} + \frac{n!\,2^n}{n!} \qquad (3.120)
\end{aligned}$$

Clearly, the maximum value i achieves is $i_{\max} = n-1$. For $i = i_{\max}$, (3.120) becomes:

$$\begin{aligned}
T(n) &= \frac{n!}{1!}T(1) + \frac{n!\,2^2}{2!} + \frac{n!\,2^3}{3!} + \dots + \frac{n!2^{n-1}}{(n-1)!} + \frac{n!\,2^n}{n!} \\
&= n! \times \left(T(1) + \sum_{k=2}^{n} \frac{2^k}{k!} \right)
\end{aligned}$$

But the series $\sum_{k=1}^{\infty} \frac{2^k}{k!}$ is convergent. To see why, apply d'Alambert criterion:

$$\frac{\frac{2^{k+1}}{(k+1)!}}{\frac{2^k}{k!}} = \frac{2}{k+1} < \epsilon \text{ for any } \epsilon > 0 \text{ for all sufficiently large } k$$

It follows $\sum_{k=2}^{n} \frac{2^k}{k!}$ is bound by a constant, therefore $T(n) = \Theta(n!)$. □

The following problem is presented without solution in [Par95, pp. 40, Problem 248].

Problem 79. *Solve the recurrence*

$$T(n) = T\left(\left\lfloor \frac{n}{\lg n} \right\rfloor \right) + 1$$

Solution:
This recurrence is with a single occurrence (see the definition on page 44) and its solution is the asymptotic number of times it takes to execute the iterator

$$n \to \left\lfloor \frac{n}{\lg n} \right\rfloor$$

in order n to become some constant, say 1. However, the argument at the right-hand side decreases in a complicated manner and it is unlikely a pattern will emerge if we unfold the recurrence several times, so a solution by unfolding seems impractical. Let us try to estimate $T(n)$. Consider another recurrence $S(n)$ in which the argument is always divided by the same quantity m: $S(n) = S\left(\left\lfloor \frac{n}{m} \right\rfloor\right) + 1$. If m is a constant, we know (see (3.6) on page 44) its solution is $S(n) = \log_m n$. It is easy to see the solution is the same even when

m is not a constant. So, if m equals $\lg n$ with respect to *the initial value of* n and m stays the same during the recurrence's unfolding, the solution is

$$S(n) = \Theta(\log_{\lg n} n) = \Theta\left(\frac{\lg n}{\lg\lg n}\right)$$

However, since n decreases, at each execution of the iterator the argument is divided by a quantity that gets smaller and smaller, approaching and eventually becoming constant. With those considerations in mind we can claim the following bounds on $T(n)$:

$$\frac{\lg n}{\lg\lg n} \preceq T(n) \preceq \lg n$$

We prove by induction on n that $T(n) = \Theta\left(\frac{\lg n}{\lg\lg n}\right)$. We omit the floor notation for simplicity, thus

$$T(n) = T\left(\frac{n}{\lg n}\right) + 1 \tag{3.121}$$

Part i: Proof that $T(n) = O\left(\frac{\lg n}{\lg\lg n}\right)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq c\frac{\lg n}{\lg\lg n} \tag{3.122}$$

The inductive hypothesis is

$$T\left(\frac{n}{\lg n}\right) \leq c\frac{\lg\left(\frac{n}{\lg n}\right)}{\lg\lg\left(\frac{n}{\lg n}\right)} \tag{3.123}$$

Substitute (3.123) into (3.121) to obtain

$$\begin{aligned} T(n) &\leq c\frac{\lg\left(\frac{n}{\lg n}\right)}{\lg\lg\left(\frac{n}{\lg n}\right)} + 1 \\ &= c\frac{\lg n - \lg\lg n}{\lg(\lg n - \lg\lg n)} + 1 \end{aligned} \tag{3.124}$$

Substitute $\lg n$ by 2^m, thus $\lg\lg n$ by m, in (3.124) to obtain

$$\begin{aligned} &c\frac{2^m - m}{\lg(2^m - m)} + 1 \leq c\frac{2^m - m}{\lg 1.9^m} + 1 = c\frac{2^m - m}{m\lg 1.9} + 1 = \frac{c}{1.9}\frac{2^m - m}{m} + 1 = \\ &\frac{c}{1.9}\frac{2^m}{m} - \frac{c}{1.9} + 1 \leq c\frac{2^m}{m} = c\frac{\lg n}{\lg\lg n} \end{aligned} \tag{3.125}$$

(3.124) and (3.125) yield

$$T(n) \leq c\frac{\lg n}{\lg\lg n}$$

for any c, say $c = 1$, and all large enough n.

Part ii: Proof that $T(n) = \Omega\left(\frac{\lg n}{\lg\lg n}\right)$, that is, there exists a positive constant d and some n_0, such that for all $n \geq n_0$,

$$T(n) \geq d\frac{\lg n}{\lg\lg n} \tag{3.126}$$

The inductive hypothesis is

$$T\left(\frac{n}{\lg n}\right) \geq d\frac{\lg\left(\frac{n}{\lg n}\right)}{\lg\lg\left(\frac{n}{\lg n}\right)} \tag{3.127}$$

Substitute (3.127) into (3.121) to obtain

$$\begin{aligned} T(n) &\geq d\frac{\lg\left(\frac{n}{\lg n}\right)}{\lg\lg\left(\frac{n}{\lg n}\right)} + 1 \\ &= d\frac{\lg n - \lg\lg n}{\lg\left(\lg n - \lg\lg n\right)} + 1 \end{aligned} \tag{3.128}$$

Substitute $\lg n$ by 2^m, thus $\lg\lg n$ by m, in (3.128) to obtain

$$d\frac{2^m - m}{\lg\left(2^m - m\right)} + 1 \geq d\frac{2^m - m}{\lg 2^m} + 1 = d\frac{2^m - m}{m} + 1 = d\frac{2^m}{m} - d + 1 \geq d\frac{2^m}{m} \tag{3.129}$$

for, say, $d = 1$. Going back to n, (3.128) and (3.129) prove that

$$T(n) \geq d\frac{\lg n}{\lg\lg n}$$

for $d = 1$ and all large enough n. □

Problem 80. *Prove that the recurrence relation*

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= \sum_{i=1}^{m} T(n_i) + \Theta(m) \end{aligned} \tag{3.130}$$

for any numerical partition $n_1, n_2, \ldots, n_m$ *of* $n - 1$*, has solution* $T(n) = O(n)$.

Proof:
By induction on n. Assume there are positive constants b and c such that $T(n) \leq c.n - b$. Assume the bigger hidden constant in the "$\Theta(m)$" expression is k. By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq \sum_{i=1}^{m}(c.n_i - b) + k.m \\ &= c\sum_{i=1}^{m}(n_i) - b.m + k.m \\ &= c(n-1) + m(k-b), \text{ since } n_1 + n_2 + \ldots n_m = n - 1 \\ &= c.n - c + m(k-b) \\ &\leq c.n - b, \text{ if } k - b < 0 \text{ and } c > b \end{aligned}$$

□

Problem 81. *Solve the recurrence*

$$T(n) = T(n-1) + \frac{2(n-1)}{n(n+1)} \tag{3.131}$$

Solution:

$$\begin{aligned} T(n) &= T(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= T(n-1) + T(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &= T(n-1) + T(n-2) + T(n-3) + \frac{2(n-3)}{(n-2)(n-1)} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\dots \\ &= \underbrace{T(1)}_{\Theta(1)} + \frac{2\cdot 1}{2\cdot 3} + \frac{2\cdot 2}{3\cdot 4} + \dots \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \end{aligned} \tag{3.132}$$

It follows

$$T(n) \asymp \sum_{k=2}^{n} \frac{2(k-1)}{k(k+1)} = 2\left(\sum_{k=2}^{n} \frac{k}{k(k+1)} - \sum_{k=2}^{n} \frac{1}{k(k+1)}\right) \tag{3.133}$$

First consider $\sum_{k=2}^{n} \frac{k}{k(k+1)}$:

$$\sum_{k=2}^{n} \frac{k}{k(k+1)} = \sum_{k=2}^{n} \frac{1}{k+1} \asymp \lg n \text{ by } \textbf{Fact } 8 \text{ on page } 248 \tag{3.134}$$

Now consider $\sum_{k=2}^{n} \frac{1}{k(k+1)}$. It is well-known the series $\sum_{k=1}^{\infty} \frac{1}{k(k+1)}$ is convergent and its sum is 1. A trivial proof of that fact is

$$\sum_{k=1}^{\infty} \frac{1}{k(k+1)} = \sum_{k=1}^{\infty} \frac{(k+1)-k}{k(k+1)} = \sum_{k=1}^{\infty} \left(\frac{1}{k} - \frac{1}{k+1}\right) =$$
$$\frac{1}{1} - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \frac{1}{3} + \dots = \frac{1}{1} + 0 + 0 + \dots = 1$$

It follows that

$$\sum_{k=2}^{n} \frac{1}{k(k+1)} = \Theta(1) \tag{3.135}$$

Plug in (3.135) and (3.134) into (3.133) to obtain

$$T(n) \asymp \lg n \tag{3.136}$$

□

The average complexity of QUICKSORT (the pseudocode is on page 130) is given by recurrence (3.137). The reasoning is the following. We consider only the number of comparison,

i.e. we ignore the swaps. Given an input $A[1,\ldots,n]$, there are precisely $n-1$ comparisons performed by the PARTITION function (see the pseudocode on page 127). The pivot position returned by PARTITION can be any number from 1 to n inclusive. Let k denote that value. Two recursive calles are performed, one on the subarray $A[1,\ldots,k-1]$ and the other one on the subarray $A[k+1,\ldots,n]$. The first one has precisely $k-1-1+1=k-1$ elements, the second one has precisely $n-(k+1)+1=n-k$ elements.

Problem 82. *Solve the recurrence*

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}\big(T(k-1)+T(n-k)\big)+(n-1) \tag{3.137}$$

Solution:

$$\begin{aligned}T(n)&=\frac{1}{n}\sum_{k=1}^{n}\big(T(k-1)+T(n-k)\big)+(n-1)\\&=\frac{1}{n}\left(\sum_{k=1}^{n}T(k-1)+\sum_{k=1}^{n}T(n-k)\right)+(n-1)\\&=\frac{2}{n}\left(\sum_{k=1}^{n}T(k-1)\right)+(n-1)\end{aligned} \tag{3.138}$$

Multiply both sides of (3.138) by n to obtain:

$$nT(n)=2\left(\sum_{k=1}^{n}T(k-1)\right)+n(n-1) \tag{3.139}$$

Then

$$(n-1)T(n-1)=2\left(\sum_{k=1}^{n-1}T(k-1)\right)+(n-1)(n-2) \tag{3.140}$$

Subtract (3.139) minus (3.140) to obtain:

$$\begin{aligned}nT(n)-(n-1)T(n-1)=2T(n-1)+n(n-1)-(n-1)(n-2)=\\2T(n-1)+2(n-1)\quad\Leftrightarrow\\nT(n)=(n+1)T(n-1)+2(n-1)\end{aligned} \tag{3.141}$$

Divide both sides of (3.141) by $n(n+1)$ to obtain:

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{2(n-1)}{n(n+1)} \tag{3.142}$$

Let $\frac{T(n)}{n+1}=S(n)$, *i.e.* $T(n)=(n+1)S(n)$. In terms of $S(\,)$, (3.142) becomes:

$$S(n)=S(n-1)+\frac{2(n-1)}{n(n+1)} \tag{3.143}$$

But $S(n)\asymp\lg n$ by (3.136) on the previous page. It follows that $T(n)=\Theta(n\lg n)$. □

3.2.2 The Master Theorem

There are several theoretical results solving a broad range of recurrences corresponding to divide-and-conquer algorithms that are called master theorems. The one stated below is due to [CLR00]. There is a considerately more powerful master theorem due to Akra and Bazzi [AB98]. See [Lei96] for a detailed explanation.

Theorem 1 (Master Theorem, [CLR00], pp. 62)**.** *Let* $a \geq 1$ *and* $b > 1$ *be constants, let* $k = \lg_b a$*, and let* $f(n)$ *be a positive function. Let*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = \Theta(1)$$

where $\frac{n}{b}$ *is interpreted either as* $\left\lfloor \frac{n}{b} \right\rfloor$ *or* $\left\lceil \frac{n}{b} \right\rceil$*. Then* $T(n)$ *can be bounded asymptotically as follows.*

Case 1 *If* $f(n) = O\left(n^{k-\epsilon}\right)$ *for some positive constant* ϵ *then* $T(n) = \Theta(n^k)$*.*

Case 2 *If* $f(n) = \Theta(n^k)$ *then* $T(n) = \Theta\left(n^k . \lg n\right)$*.*

Case 3 *If both*

1. $f(n) = \Omega\left(n^{k+\epsilon}\right)$ *for some positive constant* ϵ*, and*
2. $a.f\left(\frac{n}{b}\right) \leq c.f(n)$ *for some constant* c *such that* $0 < c < 1$ *and for all sufficiently large* n*,*

then $T(n) = \Theta(f(n))$*.* □

Condition 3-2 is known as *the regularity condition.*

Note that the condition $f(n) = O\left(n^{k-\epsilon}\right)$ is stronger than $f(n) = o(n^k)$ and $f(n) = \Omega\left(n^{k+\epsilon}\right)$ is stronger than $f(n) = \omega\left(n^k\right)$:

$$f(n) = O(n^{k-\epsilon}) \Rightarrow f(n) = o(n^k)$$
$$f(n) = o(n^k) \not\Rightarrow f(n) = O(n^{k-\epsilon})$$
$$f(n) = \Omega(n^{k+\epsilon}) \Rightarrow f(n) = \omega(n^k)$$
$$f(n) = \omega(n^k) \not\Rightarrow f(n) = \Omega(n^{k+\epsilon})$$

For example, consider that

$$n \lg n = \omega(n) \tag{3.144}$$
$$n \lg n \neq \Omega(n^{1+\epsilon}) \text{ for any } \epsilon > 0 \text{ because } \lg n \neq \Omega(n^{\epsilon}) \text{ by (1.50)} \tag{3.145}$$
$$\frac{n}{\lg n} = o(n) \tag{3.146}$$
$$\frac{n}{\lg n} \neq O(n^{1-\epsilon}) \text{ for any } \epsilon > 0 \text{ because } \frac{1}{\lg n} \neq O(n^{-\epsilon}) \tag{3.147}$$

To see why $\frac{1}{\lg n} \neq O(n^{-\epsilon})$ in (3.147) consider that

$$\lim_{n\to\infty} \frac{\lg n}{n^{\epsilon}} = 0 \Rightarrow \lim_{n\to\infty} \left(\frac{\frac{1}{n^{\epsilon}}}{\frac{1}{\lg n}} \right) = 0 \Rightarrow \frac{1}{n^{\epsilon}} = o\left(\frac{1}{\lg n}\right) \text{ by (1.6)} \Rightarrow$$

$$\frac{1}{\lg n} = \omega\left(\frac{1}{n^{\epsilon}}\right) \text{ by the transpose symmetry}$$

Problem 83. *Solve by the Master Theorem*

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Solution:
Using the terminology of the Master Theorem, a is 4, b is 2, thus $\log_b a$ is $\log_2 4 = 2$ and $n^{\log_b a}$ is n^2. The function $f(n)$ is n. The theorem asks us to compare $f(n)$ and $n^{\log_b a}$, which, in the current case, is to compare n with n^2. Clearly, $n = O(n^{2-\epsilon})$ for some $\epsilon > 0$, so Case 1 of the Master Theorem is applicable and $T(n) = n^2$. □

Problem 84. *Solve by the Master Theorem*

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

Solution:
Rewrite the recurrence as

$$T(n) = 1.T\left(\frac{n}{\frac{3}{2}}\right) + 1$$

Using the terminology of the Master Theorem, a is 1, b is $\frac{3}{2}$, thus $\log_b a$ is $\log_{\frac{3}{2}} 1 = 0$ and $n^{\log_b a}$ is $n^0 = 1$. The function $f(n)$ is 1. Clearly, $1 = \Theta(n^0)$, so Case 2 of the Master Theorem is applicable. Assording to it, $T(n) = \Theta(1. \lg n) = \Theta(\lg n)$. □

Problem 85. *Solve*

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

Solution:
Using the terminology of the Master Theorem, a is 3, b is 4, thus $\log_b a$ is $\log_4 3$, which is approximately 0.79, and the function $f(n)$ is $n \lg n$. It certainly is true that $n \lg n = \Omega(n^{\log_4 3+\epsilon})$ for some $\epsilon > 0$, for instance $\epsilon = 0.1$. However, we have to check the regularity condition to see if Case 3 of the Master Theorem is aplicable. The regularity condition in this case is:

$$\exists c \text{ such that } 0 < c < 1 \text{ and } 3\frac{n}{4} \lg \frac{n}{4} \leq cn \lg n \text{ for all sufficiently large } n$$

The latter clearly holds for, say, $c = \frac{3}{4}$, therefore Case 3 is applicable and according to it, $T(n) = \Theta(n \lg n)$. □

Problem 86. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$

Solution:
Let us the try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 2 and b is 2, thus $\log_b a$ is $\log_2 2 = 1$, therefore $n^{\log_b a}$ is $n^1 = n$. The function $f(n)$ is $n \lg n$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $n \lg n = O(n^{1-\epsilon})$ for some $\epsilon > 0$? No, because $n \lg n = \omega(n^1)$.

try Case 2 Is it true that $n \lg n = \Theta(n^1)$? No, because $n \lg n = \omega(n^1)$.

try Case 3 Is it true that $n \lg n = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$? No, see (3.145).

Therefore this problem cannot be solved using the Master Theorem as stated above. We solve it by Theorem 2 on page 96 and the answer is $T(n) = \Theta(n \lg^2 n)$. □

Problem 87. *Solve*

$$T(n) = 4T\left(\frac{n}{2}\right) + n \tag{3.148}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \tag{3.149}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \tag{3.150}$$

$$\tag{3.151}$$

Solution:
Using the terminology of the Master Theorem, a is 4 and b is 2, thus $\log_b a$ is $\log_4 2 = 2$, therefore $n^{\log_b a}$ is n^2. With respect to (3.148), it is the case that $n = O(n^{2-\epsilon})$ for some $\epsilon > 0$, therefore the solution of (3.148) is $T(n) = \Theta(n^2)$ by Case 1 of the Master Theorem. With respect to (3.149), it is the case that $n^2 = \Theta(n^2)$, therefore the solution of (3.149) is $T(n) = \Theta(n^2 \lg n)$ by Case 2 of the Master Theorem. With respect to (3.150), it is the case that $n^3 = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$, therefore the solution of (3.150) is $T(n) = \Theta(n^3)$ by Case 3 of the Master Theorem, *provided* the regularity condition holds. The regularity condition here is

$$\exists c \text{ such that } 0 < c < 1 \text{ and } 4\left(\frac{n}{2}\right)^3 \leq cn^3 \text{ for all sufficiently large } n$$

Clearly that holds for any c such that $\frac{1}{2} \leq c < 1$. Therefore, by Case 3 of the Master Theorem, the solution of (3.150) is $T(n) = \Theta(n^3)$. □

Problem 88. *Solve*

$$T(n) = T\left(\frac{n}{2}\right) + \lg n \tag{3.152}$$

Solution:
Let us try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 1 and b is 2, thus $\log_b a$ is $\log_2 1 = 0$, therefore $n^{\log_b a}$ is $n^0 = 1$. The function $f(n)$ is $\lg n$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $\lg n = O(n^{0-\epsilon})$ for some $\epsilon > 0$? No, because $\lg n$ is an increasing function and $n^{-\epsilon} = \frac{1}{n^\epsilon}$ is a decreasing one.

try Case 2 Is it true that $\lg n = \Theta(n^0)$? No, because $\lg n = \omega(n^0)$.

try Case 3 Is it true that $\lg n = \Omega(n^{0+\epsilon})$ for some $\epsilon > 0$? No, see (1.50) on page 12.

So the Master Theorem is not applicable and we seek other methods for solving. Substitute n by 2^m, *i.e.* $m = \lg n$ and $m = \lg n$. Then (3.152) becomes

$$T(2^m) = T\left(2^{m-1}\right) + m \tag{3.153}$$

Further substitute $T(2^m)$ by $S(m)$ and (3.153) becomes

$$S(m) = S(m-1) + m \tag{3.154}$$

But that recurrence is the same as (3.19), therefore its solution is $S(m) = \Theta(m^2)$. Let us go back now to the original n and $T(n)$.

$$S(m) = \Theta(m^2) \;\Leftrightarrow\; T(2^m) = \Theta(\lg^2 n) \;\Leftrightarrow\; T(n) = \Theta(\lg^2 n)$$

□

Problem 89. *Solve by the Master Theorem*

$$T(n) = 2T\left(\frac{n}{2}\right) + n^3 \tag{3.155}$$

$$T(n) = T\left(\frac{9n}{10}\right) + n \tag{3.156}$$

$$T(n) = 16T\left(\frac{n}{4}\right) + n^2 \tag{3.157}$$

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \tag{3.158}$$

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \tag{3.159}$$

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} \tag{3.160}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n} \tag{3.161}$$

$$T(n) = 8T\left(\frac{n}{2}\right) + n^3 \tag{3.162}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + 2n^2 \tag{3.163}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + n\lg n \tag{3.164}$$

Solution:
(3.155): as $n^3 = \Omega\left(n^{\log_2 2+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $2\left(\frac{n}{2}\right)^3 \leq cn^3 \Leftrightarrow \frac{1}{4} \leq c$. So, any c such that $\frac{1}{4} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^3)$.

(3.156): rewrite the recurrence as $T(n) = 1.T\left(\frac{n}{\frac{10}{9}}\right) + n$. As $n = \Omega\left(n^{\left(\log_{\frac{10}{9}} 1\right)+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $1\left(\frac{n}{\frac{10}{9}}\right) \leq cn \Leftrightarrow \frac{9}{10} \leq c$. So, any c such that $\frac{9}{10} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n)$.

(3.157): As $n^2 = \Theta\left(n^{\log_4 16}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = n^2 \lg n$.

(3.158): as $n^2 = \Omega\left(n^{\log_3 7+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $7\left(\frac{n}{3}\right)^2 \leq cn^2 \Leftrightarrow \frac{7}{9} \leq c$. So, any c such that $\frac{7}{9} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^2)$.

(3.159): as $n^2 = O\left(n^{\log_2 7-\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 1 of the Master Theorem and so $T(n) = \Theta\left(n^{\log_2 7}\right)$.

(3.160): as $\sqrt{n} = \Theta\left(n^{\log_4 2}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = \Theta(\sqrt{n} \lg n)$.

(3.161): as $n^{\frac{5}{2}} = \Omega\left(n^{\log_2 4+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $4\left(\frac{n}{2}\right)^{\frac{5}{2}} \leq cn^{\frac{5}{2}} \Leftrightarrow \frac{1}{\sqrt{2}} \leq c$. So, any c such that $\frac{1}{\sqrt{2}} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^2\sqrt{n})$.

(3.162): As $n^3 = \Theta\left(n^{\log_2 8}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = n^3 \lg n$.

(3.163): as $2n^2 = \Omega\left(n^{\log_2 3+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $3\left(2\left(\frac{n}{2}\right)^2\right) \leq c2n^2 \Leftrightarrow 3 \leq 4c$. So, any c such that $\frac{3}{4} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(2n^2) = \Theta(n^2)$.

(3.164): as $n \lg n = O\left(n^{\log_2 3-\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 1 of the Master Theorem and so $T(n) = \Theta\left(n^{\log_2 3}\right)$. □

The following result extends Case 2 of the Master Theorem.

Theorem 2. *Under the premises of Theorem 1, assume*

$$f(n) = \Theta(n^k \lg^t n) \tag{3.165}$$

for some constant $t \geq 0$. *Then*

$$T(n) = \Theta(n^k \lg^{t+1} n)$$

Proof:
Theorem 1 itself is not applicable because the recurrence for the said $f(n)$ cannot be classified into any of the three cases there. To solve the problem we use unfolding. For simplicity we assume that n is an exact power of b, *i.e.* $n = b^m$ for some integer $m > 0$. The same technique is used in [CLR00] for proving the Master Theorem: first prove it for exact powers of b and then prove the result holds for any positive n. Here we limit our proof to the case that n is an exact power of b and leave it to the reader to generalise for any positive n.

Assume that the logarithm in (3.165) is base-b and note we can rewrite what is inside the Θ-notation on the right-hand side of (3.165) in the following way:

$$n^k \log_b^t n = n^{\log_b a} (\log_b b^m)^t = b^{(m \log_b a)} m^t = b^{(\log_b a^m)} m^t = a^m m^t \tag{3.166}$$

Then (3.165) is equivalent to saying that

$$c_1 a^m m^t \leq f(b^m) \leq c_2 a^m m^t$$

for some positive constants c_1 and c_2 and all sufficiently large values of m. However, for the sake of simplicity, we will assume in the remainder of the proof that

$$f(b^m) = a^m m^t \tag{3.167}$$

The reader is invited to construct a proof for the general case.

By the definition of the Master Theorem, $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Using (3.167) we rewrite that as follows.

$$\begin{aligned}
T(b^m) &= aT\left(\frac{b^m}{b}\right) + a^m m^t \\
&= aT(b^{m-1}) + a^m m^t \;\Leftrightarrow \\
S(m) &= aS(m-1) + a^m m^t \qquad \text{substituting } T(b^m) \text{ with } S(m) \\
&= a\big(aS(m-2) + a^{m-1}(m-1)^t\big) + a^m m^t \\
&= a^2 S(m-2) + a^m (m-1)^t + a^m m^t \\
&= a^2\big(aS(m-3) + a^{m-2}(m-2)^t\big) + a^m (m-1)^t + a^m m^t \\
&= a^3 S(m-3) + a^m (m-2)^t + a^m (m-1)^t + a^m m^t \\
&\dots \\
&= a^{m-1} S(1) + a^m 2^t + a^m 3^t + \ldots + a^m (m-2)^t + a^m (m-1)^t + a^m m^t \\
&= a^{m-1} S(1) - a^m + a^m \underbrace{(1^t + 2^t + 3^t + \ldots + (m-2)^t + (m-1)^t + m^t)}_{\Theta(m^{t+1}) \text{ by (8.25) on page 249}} \\
&= a^{m-1} S(1) - a^m + a^m \Theta(m^{t+1}) \\
&= a^{m-1} S(1) - a^m + \Theta(a^m m^{t+1})
\end{aligned} \tag{3.168}$$

But (3.168) is $\Theta(a^m m^{t+1})$ because $a^m m^{t+1} = \omega(|a^{m-1}S(1) - a^m|)$. So,

$$S(m) = \Theta(a^m m^{t+1}) \Leftrightarrow T(n) = \Theta\left(a^{\log_b n}(\log_b n)^{t+1}\right)$$

Having in mind that $a^{\log_b n} = n^{\log_b a}$ and $\log_b n = \Theta(\lg n)$, we conclude that

$$T(n) = \Theta\left(n^{\log_b a} \lg^{t+1} n\right)$$

□

Problem 90. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n$$

Solution:
Since $\lg n = O\left(n^{\log_2 2-\epsilon}\right)$ for some $\epsilon > 0$, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$ by Case 1 of the Master Theorem. □

Problem 91. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$$

Solution:
Let us the try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 2 and b is 2, thus $\log_b a$ is $\log_2 2 = 1$, therefore $n^{\log_b a}$ is $n^1 = n$. The function $f(n)$ is $\frac{n}{\lg n}$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $\frac{n}{\lg n} = O(n^{1-\epsilon})$ for some $\epsilon > 0$? No, see (3.147) on page 91.

try Case 2 Is it true that $\frac{n}{\lg n} = \Theta(n^1)$? No, because $n \lg n = o(n^1)$.

try Case 3 Is it true that $\frac{n}{\lg n} = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$? No, because $n \lg n = o(n^1)$.

Therefore this problem cannot be solved using the Master Theorem as stated above. Furthermore, Theorem 2 on the previous page cannot be applied either because it is not true that $\frac{n}{\lg n} = \Theta(n^{\log_2 2} \lg^t(n))$ for any $t \geq 0$.

We solve the problem by unfolding.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{\frac{n}{2}}{\lg \frac{n}{2}}\right) + \frac{n}{\lg n} \\
&= 4T\left(\frac{n}{4}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= 4\left(2T\left(\frac{n}{8}\right) + \frac{\frac{n}{4}}{\lg \frac{n}{4}}\right) + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= 8T\left(\frac{n}{8}\right) + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&\dots \\
&= nT(1) + \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{(\lg n) - 2} + \frac{n}{(\lg n) - 1} + \frac{n}{\lg n} \\
&= nT(1) + \underbrace{n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{(\lg n) - 2} + \frac{1}{(\lg n) - 1} + \frac{1}{\lg n}\right)}_{B}
\end{aligned}$$

Clearly, $|A| = O(n)$. Now observe that $B = n.H_{\lg n}$ because inside the parentheses is the $(\lg n)^{\text{th}}$ partial sum of the harmonic series (see 8.21 on page 248). By (8.22), $H_{\lg n} = \Theta(\lg \lg n)$, therefore $B = \Theta(n \lg \lg n)$, therefore $T(n) = \Theta(n \lg \lg n)$. □

Problem 92 ([CLR00], Problem 4.4-3)**.** *Show that case 3 of the master theorem is overstated, in the sense that the regularity condition* $af\left(\frac{n}{b}\right) \leq cf(n)$ *for some constant* $c < 1$ *implies that there exists a constant* $\epsilon > 0$ *such that* $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Solution:
Assume for some constant c such that $0 < c < 1$,

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right)$$

Then

$$\begin{aligned}
f(n) &\geq \frac{a^2}{c^2} f\left(\frac{n}{b^2}\right) \\
f(n) &\geq \frac{a^3}{c^3} f\left(\frac{n}{b^3}\right) \\
&\dots \\
f(n) &\geq \frac{a^t}{c^t} f\left(\frac{n}{b^t}\right) \qquad (3.169)
\end{aligned}$$

For some constant value n_0 for n, that process stops. So, $\frac{n}{b^t} = n_0$, therefore $t = \log_b\left(\frac{n}{n_0}\right) = \log_b n - \log_b n_0$. Let $s = \frac{1}{c}$. Since $c < 1$, it is the case that $s > 1$. Substitute t, $\frac{n}{b^t}$, and c in (3.169) to obtain

$$f(n) \geq \frac{a^{\log_b n}}{a^{\log_b n_0}} \times \frac{s^{\log_b n}}{s^{\log_b n_0}} \times f(n_0)$$

Note that $\frac{1}{a^{\log_b n_0}} \times \frac{1}{s^{\log_b n_0}} \times f(n_0)$ is a constant. Call that constant, β. So,

$$f(n) \geq a^{\log_b n} \times s^{\log_b n} \times \beta$$

Having in mind that $a^{\log_b n} = n^{\log_b a}$ and $s^{\log_b n} = n^{\log_b s}$, we see that

$$f(n) \geq \beta \times n^{\log_b a} \times n^{\log_b s}$$

Since $s > 1$, $\log_b s > 0$. Let $\epsilon = \log_b s$. We derive the desired result: for all sufficiently large n and some constant β

$$f(n) \geq \beta n^{\log_b a + \epsilon} \Rightarrow f(n) = \Omega(n^{\log_b a + \epsilon})$$

□

3.2.3 The Method with the Characteristic Equation

Theorem 3 ([Man05], pp. 57). *Let the infinite sequence $\widetilde{a} = a_0, a_1, a_2, \ldots$ be generated by the linear recurrence relation*

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_{r-1} a_{n-(r-1)} + c_r a_{n-r} \tag{3.170}$$

Let α_1, α_2, ..., α_s be the distinct complex roots of the characteristic *equation*

$$x^r - c_1 x^{r-1} - c_2 x^{r-2} - \ldots - c_{r-1} x - c_r = 0 \tag{3.171}$$

where α_i has multiplicity k_i for $1 \leq i \leq s$[†]*. Then*

$$a_n = P_1(n)\alpha_1^n + P_2(n)\alpha_2^n + \ldots + P_s(n)\alpha_s^n \tag{3.172}$$

where $P_i(n)$ is a polynomial of n of degree $< k_i$. The polynomials $P_1(n)$, $P_2(n)$, ..., $P_s(n)$ have r coefficients altogether which coefficients are determined uniquely by the first r elements of $\widetilde{a}$. □

Using our terminology, (3.170) would be rewritten as

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \ldots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \tag{3.173}$$

This is a generic instance of *homogeneous* linear recurrence. It is hardly possible such a recurrence relation to describe the running time of a recursive algorithm since after the recursive calls finish, at least some constant-time work must be perfomed. Therefore, we will consider more general recurrence relations that are *nonhomogeneous* and whose generic form is:

$$\begin{aligned} T(n) = {} & c_1 T(n-1) + c_2 T(n-2) + \ldots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \\ & + b_1^n Q_1(n) + b_2^n Q_2(n) + \ldots + b_m^n Q_m(n) \end{aligned} \tag{3.174}$$

b_1, b_2, ..., b_m are distinct positive constants and $Q_1(n)$, $Q_2(n)$, ..., $Q_m(n)$ are polynomials of n of degrees d_1, d_2, ..., d_m, respectively.

[†]Clearly, $k_i \geq 1$ for $1 \leq i \leq s$ and $k_1 + k_2 + \ldots + k_s = r$.

Let us denote multisets by $\{\ \}_M$ brackets, *e.g.* $\{1,1,2,3,3,3\}_M$. For each element a from some mulitset A, let $\#(a,A)$ denote the number of occurrences of a in A. For example, $\#(1,\{1,1,2,3,3,3\}) = 2$. The union of two multisets A and B adds the multiplicities of the elements, that is,

$$A \cup B = \{x \,|\, (x \in A \text{ or } x \in B) \text{ and } (\#(x, A \cup B) = \#(x, A) + \#(x, B))\}_M$$

The cardinality of a multiset A is the sum of the multiplicities of its elements and is denoted by $|A|$. For example, $|\{1,1,2,3,3,3\}_M| = 6$.

The solution of (3.174) is the following. Let the multiset of the roots of the characteristic equation be A. Clearly, $|A| = r$. Let $B = \{b_i \,|\, \#(b,B) = d_i + 1\}_M$. Let $Y = A \cup B$. Clearly, $|Y| = r + \sum_{i=1}^{m}(d_i + 1)$. Let us rename the distinct elements of Y as y_1, y_2, ..., y_t and define that $\#(y_i, Y) = z_i$, for $1 \le i \le t$. Then

$$\begin{aligned} T(n) = {} & \beta_{1,1}\, y_1^n + \beta_{1,2}\, n\, y_1^n + \ldots + \beta_{1,z_1}\, n^{z_1-1}\, y_1^n + \\ & \beta_{2,1}\, y_2^n + \beta_{2,2}\, n\, y_2^n + \ldots + \beta_{2,z_2}\, n^{z_2-1}\, y_2^n + \\ & \ldots \\ & \beta_{t,1}\, y_1^n + \beta_{t,2}\, n\, y_t^n + \ldots + \beta_{t,z_t}\, n^{z_t-1}\, y_1^n \end{aligned} \tag{3.175}$$

The indexed β's are constants, $|Y|$ in number. Since we are interested in the asymptotic growth rate of $T(n)$ we do not care what are the precise values of those constants. As we do not specify concrete initial conditions we are not able to compute them precisely anyways. The asymptotic growth rate is determined by precisely one of all those terms—namely, the biggest y_i, multiplied by the biggest degree of n.

Problem 93. *Solve*

$$T(n) = T(n-1) + 1$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^0$ to make sure its form is as required by (3.174). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 0$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1,1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n$ for some constants A and B, therefore $T(n) = \Theta(n)$. □

Problem 94. *Solve*

$$T(n) = T(n-1) + n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^1$ to make sure its form is as required by (3.174). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset

of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n$ for some constants A, B, and C, therefore $T(n) = \Theta(n^2)$. □

Problem 95. *Solve*

$$T(n) = T(n-1) + n^4$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^4$ to make sure its form is as required by (3.174). The characteristic equation is $x-1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 4$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 5$ to $\{1\}_M$, obtaining $\{1, 1, 1, 1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n + D\,n^3\,1^n + E\,n^4\,1^n + F\,n^5\,1^n$ for some constants A, B, C, D, and F, therefore $T(n) = \Theta(n^5)$. □

Problem 96. *Solve*

$$T(n) = T(n-1) + 2n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n(2n^1)$ to make sure its form is as required by (3.174). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n$ for some constants A, B, and C, therefore $T(n) = \Theta(n^2)$. □

Problem 97. *Solve*

$$T(n) = T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 2^n n^0$ to make sure its form is as required by (3.174). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1, 2\}_M$. Then $T(n) = A\,1^n + B\,2^n$ for some constants A and B, therefore $T(n) = \Theta(2^n)$. □

Problem 98. *Solve*

$$T(n) = 2T(n-1) + 1$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 1^n n^0$ to make sure its form is as required by (3.174). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 0$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1, 2\}_M$. Then $T(n) = A\,1^n + B\,2^n$ for some constants A and B, therefore $T(n) = \Theta(2^n)$. □

Problem 99. *Solve*

$$T(n) = 2T(n-1) + n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 1^n n^1$ to make sure its form is as required by (3.174). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 2\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,2^n$ for some constants A, B, and C, therefore $T(n) = \Theta(2^n)$. □

Problem 100. *Solve*

$$T(n) = 2T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 2^n n^0$ to make sure its form is as required by (3.174). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 0$ to $\{2\}_M$, obtaining $\{2, 2\}_M$. Then $T(n) = A\,2^n + B\,n\,2^n$ for some constants A and B, therefore $T(n) = \Theta(n2^n)$. □

Problem 101. *Solve*

$$T(n) = 3T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 3T(n-1) + 2^n n^0$ to make sure its form is as required by (3.174). The characteristic equation is $x - 3 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{3\}_M$. In the naming convention of (3.174), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 0$ to $\{3\}_M$, obtaining $\{2, 3\}_M$. Then $T(n) = A\,2^n + B\,3^n$ for some constants A and B, therefore $T(n) = \Theta(3^n)$. □

Problem 102. *Solve*

$$T(n) = T(n-1) + T(n-2)$$

using the method of the characteristic equation.

Solution:
This recurrence is homogeneous but is nevertheless interesting. The characteristic equation is $x^2 - x - 1 = 0$. The two roots are $x_1 = \frac{1+\sqrt{5}}{2}$ and $x_2 = \frac{1-\sqrt{5}}{2}$. Then $T(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$ for some constants A and B. Note that $\left|\frac{1-\sqrt{5}}{2}\right| < 1$ therefore $T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. □

Problem 103. *Solve*

$$T(n) = T(n-1) + 2T(n-2)$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - x - 2 = 0$. The two roots are $x_1 = \frac{1+3}{2} = 2$ and $x_2 = \frac{1-3}{2} = -1$. Then $T(n) = A\,2^n + B\,(-1)^n$ for some constants A and B. Therefore, $T(n) = \Theta(2^n)$. □

Problem 104. *Solve*

$$T(n) = 3T(n-1) + 4T(n-2) + 1 \tag{3.176}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - 3x - 4 = 0$. The two roots are $x_1 = \frac{3+5}{2} = 4$ and $x_2 = \frac{3-5}{2} = -1$. Then $T(n) = A\,4^n + B\,(-1)^n + C\,1^n$ for some constants A, B, and C. Therefore, $T(n) = \Theta(4^n)$. □

Problem 105. *Solve*

$$T(n) = 4T(n-1) + 3T(n-2) + 1 \tag{3.177}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - 4x - 3 = 0$. The two roots are $x_1 = 2 + \sqrt{7}$ and $x_2 = 2 - \sqrt{7}$. Then $T(n) = A\,(2+\sqrt{7})^n + B\,(2-\sqrt{7})^n + C\,1^n$ for some constants A, B, and C. Therefore, $T(n) = \Theta((2+\sqrt{7})^n)$. □

Problem 106. *Solve*

$$T(n) = 5T(n-1) + 6T(n-2) + 1 \tag{3.178}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - 5x - 6 = 0$. The two roots are $x_1 = \frac{5+\sqrt{5^2+24}}{2} = \frac{5+\sqrt{49}}{2} = \frac{5+7}{2} = 6$ and $x_2 = \frac{5-7}{2} = -1$. Then $T(n) = A.6^n + B.(-1)^n + C.1^n$ for some constants A, B, and C. Therefore, $T(n) = \Theta(6^n)$. □

Problem 107. *Solve*

$$T(n) = 4T(n-3) + 1 \tag{3.179}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is

$$x^3 - 4 = 0$$

Its roots are

$$x_1 = \sqrt[3]{4}$$
$$x_2 = \sqrt[3]{4}\, e^{i\frac{2\pi}{3}}$$
$$x_3 = \sqrt[3]{4}\, e^{i\frac{-2\pi}{3}}$$

If A, B, C, and D are some complex constants the solution is

$$\begin{aligned}
T(n) &= A\left(\sqrt[3]{4}\right)^n + B\left(\sqrt[3]{4}\right)^n e^{\frac{2n\pi i}{3}} + C\left(\sqrt[3]{4}\right)^n e^{\frac{-2n\pi i}{3}} + D1^n = \\
&= A\left(\sqrt[3]{4}\right)^n + B\left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{2n\pi}{3}\right) + i\sin\left(\frac{2n\pi}{3}\right)\right) + \\
&\quad C\left(\sqrt[3]{4}\right)^n \left(\cos\left(\frac{-2n\pi}{3}\right) + i\sin\left(\frac{-2n\pi}{3}\right)\right) + D \\
&= A\left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2n\pi}{3}\right)(B+C) + \\
&\quad \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2n\pi}{3}\right)(B-C)i + D
\end{aligned}$$

If we take $B = C = \frac{1}{2}$, we get one solution

$$T_1(n) = A\left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \cos\left(\frac{2n\pi}{3}\right) + D$$

If we take $B = -\frac{1}{2}i$ and $C = \frac{1}{2}i$, we get another solution

$$T_2(n) = A\left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \sin\left(\frac{2n\pi}{3}\right) + D$$

By the superposition principle[†], we have a general solution

$$T(n) = A_1\left(\sqrt[3]{4}\right)^n + A_2\left(\sqrt[3]{4}\right)^n \cos\left(\frac{2n\pi}{3}\right) + A_3\left(\sqrt[3]{4}\right)^n \sin\left(\frac{2n\pi}{3}\right) + A_4$$

for some constants A_1, A_2, A_3, A_4. The asymptotics of the solution is $T(n) = \Theta\left(\left(\sqrt[3]{4}\right)^n\right)$.
□

[†]The superposition principle says if we have a linear recurrence and we know that some functions $g_i()$, $1 \leq i \leq k$, are solutions to it, then any linear combination of them is also a solution. See [Bal91], pp. 97.

Chapter 4

Proving the correctness of algorithms

4.1 Preliminaries

Every algorithm implements a total function that maps the set of the inputs to the set of the outputs. To prove the algorithm is correct is to prove that

1. the algorithm halts on every input, and
2. for every input, the corresponding output is precisely the one that is specified by the said function.

Proving facts about the "behaviour" of algorithms is not easy even for trivial algorithms. It is well known that in general such proofs cannot be automated. For instance, provably there does not exist an algorithm that, given as input a program and its input, always determines (using, of course, a finite number of steps) whether that program with that input halts. For details, see the HALTING PROBLEM and Theorem 5 on page 255.

That famous undecidability result, due to Alan Turing, sets certain limitations on the power of computation in general. Now we show that even for a specific simple program deciding whether it halts or not can be extremely difficult. Consider the following program.

```
k = 3;
for (;;) {
  for(a = 1; a <= k; a ++)
   for(b = 1; b <= k; b ++)
    for(c = 1; c <= k; c ++)
     for(n = 3; n <= k; n ++)
      if (pow(a,n) + pow(b,n) == pow(c,n))
       exit();
  k++; }
```

Clearly, the program does not halt if and only if Fermat's Last Theorem[†] is true. However, for several hundred years some of the best mathematicians in the world were unable to prove that theorem. The theorem was indeed proved after a huge effort by Sir Andrew

[†]It says: "$a^n + b^n = c^n$ has no positive integer solutions for $n \geq 3$".

Wiles, an effort that spanned many years and led to some initial frustrations as the first proof turned out to be incorrect. For a detailed account of these events, see [CSS97] or the review of that book by Buzzard [Buz99].

We emphasise that to prove Fermat's Last Theorem and to prove that the abovementioned program halts are essentially the same thing. Furthermore, determining whether an algorithm halts or not is but only one aspect of the analysis of algorithms. If telling whether the execution of a well-defined sequence of instructions halts or not is hard, determining whether it indeed returns the desired output cannot be any easier in general. So, determining the behaviour of even simple algorithms can necessitate profound mathematical knowledge and skills.

4.2 Loop Invariants – An Introduction

It is possible to prove assertions about algorithms—correctness or time complexity—using *loop invariants*. That technique is applicable when the algorithm is iterative. It may be a simple (not nested) loop or something more complicated. The gist of the algorithm has to be a **for** or **while** loop.

Proving assertions with loop invariants is essentially proving assertions by induction, with the notable exception that normally proofs by induction are done for infinite sequences, while loop invariants concern algorithm that take only finite number of steps. Here is an example of a proof of correctness of a very simple algorithm that uses a loop invariant. The proof is very detailed and the invariant itself is outlined as a nested statement (sub-lemma).

```
MAXIMAL SEQUENTIAL(A[1,2,...,n]: array of integers)
1  max ← A[1]
2  i ← 2
3  while i ≤ n do
4      if A[i] > max
5          max ← A[i]
6      i ← i + 1
7  return max
```

Lemma 3. *Algorithm* MAXIMAL SEQUENTIAL *returns the value of a maximum element of* $A[1, 2, \ldots, n]$.

Proof:
In order to prove the desired desired result we first prove a sub-lemma:

> **Sub-lemma:** *Every time the execution of* MAXIMAL SEQUENTIAL *is at line 3,* max *contains the value of a maximum element in the subarray* $A[1, \ldots, i-1]$.
>
> The proof of the sub-lemma is done by induction on the number of times the execution reaches line 3.
>
> **Basis.** The first time the execution is at line 3, i equals 2 because of the previous assignment at line 2. So, the subarray $A[1, \ldots, i-1]$ is in fact $A[1, \ldots, 1]$. But max is $A[1]$ because of the previous assignment at line 1 and indeed this is a maximum element in $A[1, \ldots, 1]$.

Inductive Hypothesis. Assume the claim is true at a certain moment when the execution is at line 3, such that the loop is to be executed at least once more[†]. Let us define that the current value of $\mathtt{i}$ is called $\mathtt{i}'$.

Induction Step. The following two possibilities are exhaustive.

- $A[\mathtt{i}']$ is the maximum element in $A[1,\ldots,\mathtt{i}']$. By the inductive hypothesis, *max* equals a maximum element in $A[1,\ldots,\mathtt{i}'-1]$. It must be the case that $\mathit{max} < A[\mathtt{i}']$. So, the condition at line 4 is TRUE and the assignment at line 5 takes place. Thus *max* becomes equal to the maximum element in $A[1,\ldots,\mathtt{i}']$ immediately after that assignment. Then $\mathtt{i}$ gets incremenented to $\mathtt{i}'+1$ (line 6). So, the next time the execution reaches line 3 again, *max* is indeed equal to a maximum element in $A[1,\ldots,(\mathtt{i}'+1)-1]$. We see the invariant is preserved in the current case.
- It is not the case that $A[\mathtt{i}']$ is the maximum element in $A[1,\ldots,\mathtt{i}']$. Then $A[\mathtt{i}']$ is smaller than or equal to a maximum element in $A[1,\ldots,\mathtt{i}'-1]$, which in its turn is equal to *max* by the inductive hypothesis. It follows *max* is equal to a maximum element in $A[1,\ldots,\mathtt{i}']$. The condition at line 4 is FALSE and the assignment at line 5 does not take place. Then $\mathtt{i}$ gets incremenented to $\mathtt{i}'+1$ (line 6). So, the next time the execution reaches line 3 again, *max* is indeed equal to a maximum element in $A[1,\ldots,(\mathtt{i}'+1)-1]$. We see the invariant is preserved in the current case, too.

The proof of the sublemma is done.

Corollary of the sub-lemma. Consider the last time line 3 is executed. Then $\mathtt{i}$ equals $n+1$. Substitute $\mathtt{i}$ with $n+1$ in the sub-lemma and conclude that *max* contains the value of a maximum element in $A[1,\ldots,(n+1)-1]$, *i.e.* *max* equals a maximum element in $A[1,\ldots,n]$. We observe that at line 7 the algorithm returns *max*, and that concludes the proof of the lemma. □

In the subsequent proofs we will not maintain as separate claims the invariant and its corollary relative to the last execution of the loop. The latter will be the last step of the proof, called *termination*. The proofs will have the following structure (after [CLR00]). The claim is a one-place predicate, call it P, that is associated with the line of the algorithm that contains the loop's conditional statement. In the most general case the variable of the predicate is the number of times the conditional statement has been evaluated. Say, the loop's conditional statement is at line k. Generally speaking, the proof goes like this:

- Show P holds the first time the execution is at line k.
- Assuming P holds at certain time when the execution is at line k and the loop is to be executed at least once more (*i.e.,* the condition is true), prove P holds the next time the execution is at line k.
- Consider P at the moment when the execution is at line k and loop's body is not to be executed any more (*i.e.,* the condition is false) and use P to

[†]It would be an error to omit the proviso "the loop is to be executed at least once more". We have to prove the invariant is preserved during any execution of the loop. Therefore, if we consider the very last execution of line 3, we cannot establish the preservation of the invariant in the body of the loop.

prove the desired result immediately. Such a moment must exists since the algorithm terminates.

An important special case of that proof scheme is the following one. The variable of the predicate is the control variable of the loop. Say, the loop is a **for**-loop and its conditional statement is at line k:

k **for** $i \leftarrow a$ to b

Generally speaking, the proof goes like this:

- Show $P(a)$ holds the first time the execution is at line k.
- Assuming $P(i)$ holds at certain time when the execution is at line k and the loop is to be executed at least once more (*i.e.,* $i \leq b$), prove $P(i+1)$ holds the next time the execution is at line 12.
- At the moment when the execution is at line 12 and the loop is not to be executed any more, i equals $b+1$. Plug that value into the predicate: $P(b+1)$ must yield *immediately* desired result.

We emphasise two points.

- First, there can be multiple ways of going through the body of the loop depending on the values of both the control variable and other variables. For instance, the body can have complicated nested **if**–**else**–**if** structure. The proof has to follow any possible path of execution. Say, the algorithm has positive integer variables a, b, and t, and the body of the loop has the following general structure:

```
if i mod 5 = 1 then
    if t is a perfect square then
        a ← b + 1
    else if t < 200 then
        a ← b + 2
    else
        a ← b + 3
else if i mod 5 = 2 then
        b ← b^2
else
        b ← b − t
```

  A complete proof must "trace" all possible ways of going through these cases and subcases that depend on the divisibility of i by 5 and, in case the remainder is 1, on the values of t.

- Obviously, the predicate must not only be true but useful as well. Not every true statement is useful. For instance, consider a purported algorithm that sorts using at the top level a **for**-loop the conditional statement of which is at line 12:

  12 **for** $i \leftarrow 1$ **to** n

  Will the following invariant do?

Every time the execution is at line 12, the subarray $A[1,\ldots,i-1]$ is sorted.

The answer is definitely no, it is too weak. That invariant can be demonstrably true but its truth does not imply the algorithm sorts. To see why, consider the following algorithm that does not sort:

```
NO SORT(A[1,...,n])
  1  for i ← 1 to n
  2        A[i] ← i
```

Note that the above invariant holds for it: the subarray $A[1,\ldots,i-1]$ is in fact $[1, 2,\ldots,i-1]$ and is sorted. However, 1, 2, ..., $i-1$ are not necessarily elements of the original $A[]$. A sorting algorithm must sort its own input. If it outputs sorted sequence that is unrelated to the input it is not a sorting algotithm.

Once more: the invariant must not only be true but also useful for our proof.

4.3 Practicing Proofs with Loop Invariants

4.3.1 Elementary problems

Problem 108. *Consider the following two program fragments written in* `C`*. Prove using invariants that each of the functions* `sum1( )` *and* `sum2( )` *returns the sum of the elements of the array* $A[0,1,\ldots,n-1]$*:*

```
int A[n];
int sum1(int n) {
   int i, s = 0;
   for(i = 0; i < n; i ++) {
        s += A[i]; }
   return s; }
```

```
int A[n];
int sum2(int n) {
   int i, s = 0;
   for(i = 0; i < n; i ++) {
      if (i%2 == 0) {
         s += A[ i/2 ]; }
      else {
         s += A[n - 1 - i/2]; } }
   return s; }
```

Have in mind that the integer division `i/2` *returns* $\lfloor \frac{i}{2} \rfloor$.

Solution for `sum1( )`:
An invariant for `sum1( )` is:

Every time line 4 is reached, $s = \sum_{j=0}^{i-1} A[j]$.

Basis When line 4 is reached intially, $s = 0$ because of the assignment at line 3. On the other hand, $\sum_{j=0}^{i-1} A[j] = 0$, because $i = 0$, which in its turn implies the set $\{0,\ldots,i-1\}$ is empty. A sum whose index variable gets its values from an empty set is zero. ✓
Maintenance Assume the claim holds at some moment the execution is at line 4 and the body of the loop is to be executed at least once more. Before the assignment at line 5 it is the case that $s = \sum_{j=0}^{i-1} A[j]$ by the assumption. After the assignment at line 5 it is the case

that $s = \left(\sum_{j=0}^{i-1} A[j]\right) + A[i]$. The next time the execution is at line 4, i gets incremented by one, which means that with respect to the new i, it is the case that $s = \sum_{j=0}^{i-1} A[j]$.
Termination The last time the execution is at line 4, it is the case that

$$i = n$$
$$s = \sum_{j=0}^{i-1} A[j]$$

Therefore, $s = \sum_{j=0}^{n-1} A[j]$.

Solution for `sum2( )`:
An invariant for `sum2( )` is:

Every time line 4 is reached, $s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$.

Basis The first time the execution is at line 4, $s = 0$ because of the assignment at line 3. On the other hand, $s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] = 0 + 0 = 0$, because $\lfloor \frac{0+1}{2} \rfloor = 0$ and $\lfloor \frac{0}{2} \rfloor = 0$, which in its turn means the sets $\{0, \ldots, \lfloor \frac{i+1}{2} \rfloor - 1\} = \{0, \ldots, -1\}$ and $\{n - \lfloor \frac{i}{2} \rfloor, \ldots, n-1\} = \{n, \ldots, n-1\}$ are empty. ✓
Maintenance Assume the claim holds at some moment the execution is at line 4 and the body of the loop is to be executed at least once more.
Case 1 i is even. The condition at line 5 is true and the execution goes to line 6. Before the assignment at line 6, it is the case that $s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$ by the assumption.
After the assignment, it is the case that

$$s = \left(\sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \right) + A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$$
$$= \left(\sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \right) + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \qquad (4.1)$$

Since i is even, $i+1$ is odd. It is easy to see that $\lfloor \frac{i+1}{2} \rfloor = \lfloor \frac{i}{2} \rfloor$, therefore the expression (4.1) is equivalent to

$$\left(\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor - 1} A[j] + A\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \right) + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] = \sum_{j=0}^{\lfloor \frac{i}{2} \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] =$$
$$\sum_{j=0}^{\lfloor \frac{(i+1)+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j] \qquad (4.2)$$

The next time the execution is at line 4, i is incremented by one. It is obvious that with respect to the new value of i, (4.2) is:

$$\sum_{j=0}^{\lfloor \frac{i+1}{2}-1 \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

And so the invariant holds.

Case 2 i is odd. The condition at line 5 is false and the execution reaches line 8. Before the assignment at line 8 it is the case that $s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$ by the assumption. After the assignment it is the case that

$$\begin{aligned} s &= \left(\sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \right) + A\left[n - 1 - \left\lfloor \frac{i}{2} \right\rfloor \right] \\ &= \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \left(\left(\sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \right) + A\left[n - 1 - \left\lfloor \frac{i}{2} \right\rfloor \right] \right) \\ &= \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-1-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j] \end{aligned} \tag{4.3}$$

When i is odd it is the case that $\lfloor \frac{i+1}{2} \rfloor = \lfloor \frac{i+2}{2} \rfloor$. Besides that, for any i it is the case that $n - 1 - \lfloor \frac{i}{2} \rfloor = n - 1 + \lceil -\frac{i}{2} \rceil = n + \lceil -1 - \frac{i}{2} \rceil = n - \lfloor 1 + \frac{i}{2} \rfloor = n - \lfloor \frac{i+2}{2} \rfloor$. If i is odd the last expression equals $n - \lfloor \frac{i+1}{2} \rfloor$. Therefore (4.3) is equivalent to

$$\sum_{j=0}^{\lfloor \frac{(i+1)+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i+1}{2} \rfloor}^{n-1} A[j] \tag{4.4}$$

The next time the execution is at line 4, i gets incremented by one. With respect to the new i, (4.4) is:

$$\sum_{j=0}^{\lfloor \frac{i+1}{2}-1 \rfloor} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

And so the invariant holds.

Termination The last time the execution is at line 4, it is the case that

$$i = n$$

$$s = \sum_{j=0}^{\lfloor \frac{i+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{i}{2} \rfloor}^{n-1} A[j]$$

Therefore,

$$s = \sum_{j=0}^{\lfloor \frac{n+1}{2} \rfloor - 1} A[j] + \sum_{j=n-\lfloor \frac{n}{2} \rfloor}^{n-1} A[j] \tag{4.5}$$

We will prove that $\lfloor \frac{n+1}{2} \rfloor - 1$ and $n - \lfloor \frac{n}{2} \rfloor$ are two adjacent integers, increasing in that order, for every natural n. First assume n is even, *i.e.* $n = 2k$ for some natural k. Then

$$\left\lfloor \frac{n+1}{2} \right\rfloor - 1 = \left\lfloor \frac{2k+1}{2} \right\rfloor - 1 = \left\lfloor \frac{2k}{2} \right\rfloor - 1 = k - 1$$

and

$$n - \left\lfloor \frac{n}{2} \right\rfloor = 2k - \left\lfloor \frac{2k}{2} \right\rfloor = 2k - k = k$$

Now assume n is odd, *i.e.* $n = 2k + 1$ for some natural k. Then

$$\left\lfloor \frac{n+1}{2} \right\rfloor - 1 = \left\lfloor \frac{2k+2}{2} \right\rfloor - 1 = k + 1 - 1 = k$$

and

$$n - \left\lfloor \frac{n}{2} \right\rfloor = 2k + 1 - \left\lfloor \frac{2k+1}{2} \right\rfloor = 2k + 1 - \left\lfloor \frac{2k}{2} \right\rfloor = 2k + 1 - k = k + 1$$

Since $\lfloor \frac{n+1}{2} \rfloor - 1$ and $n - \lfloor \frac{n}{2} \rfloor$ are adjacent values in that order, the intervals

$$\left[0, 1, \ldots, \left\lfloor \frac{n+1}{2} \right\rfloor - 1\right] \text{ and } \left[n - \left\lfloor \frac{n}{2} \right\rfloor, n - \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n - 1\right]$$

are a partitioning of $[0, 1, \ldots, n-1]$. Therefore, (4.5) is equivalent to

$$s = \sum_{j=0}^{n-1} A[j]$$

□

Problem 109. The mode *of an array of data is any most frequently occurring element. It is not necessarily unique. In fact, if all elements are are unique then any element is mode. Design a simple iterative algorithm that computes a mode of an array of integers. Break ties arbitrarily. Your algorithm should call once function* SORT *that sorts the array. Apart from that call, the algorithm should have linear time complexity and constant space complexity. Verify the correctness of your algorithm by loop invariant.*

Solution:
The following algorithm clearly satisfies the complexity requirements.

```
COMPUTE MODE(A[1, 2, ..., n])
  1  SORT(A[])
```

```
 2  mode ← A[1]
 3  m ← 1
 4  s ← 1
 5  for i ← 2 to n
 6      if A[i] = mode
 7          m ← m + 1
 8          s ← s + 1
 9      else if A[i] ≠ A[i − 1]
10          s ← 1
11      else
12          s ← s + 1
13          if s > m
14              mode ← A[i]
15              m ← s
16  return mode
```

Now we prove its correctness. COMPUTE MODE returns *the leftmost mode*, that is, if there is a unique mode it is the leftmost mode, otherwise the leftmost mode is the mode whose leftmost appearance in $A[\,]$ is to the left of any other appearance of any other mode.

The assignments at lines 4 and 8 are, of course, unnecessary but they facilitate the correctness proof. We start with a definition and simple observation which we do not prove formally.

Definition 4 (run in a sequence)**.** *Suppose* $A[1, 2, \ldots, n]$ *is an array of numbers.* A run in $A[\,]$ *is every maximal subarray* $A[i, i+1, \ldots, i+k]$ *where* $1 \le i \le i+k \le n$ *such that* $A[i] = A[i+1] = \ldots = A[i+k]$. The length of a run *is the number of its elements.* The representative of a run *is the value of any of its elements.* □

Observation 1. *Assume* $A[1, 2, \ldots, n]$ *has precisely* t *unique elements and it is sorted. Then* $A[\,]$ *consists of precisely* t *runs whose lengths sum up to* n *and whose representatives appear in strictly ascending order.* □

Let there be t runs in the sorted array $A[\,]$ after the execution of line 1. Let their lengths, in order from left to right, be n_1, n_2, ..., n_t, where $n_1 + n_2 + \ldots + n_t = n$. The following is a loop invariant for the **for**-loop of COMPUTE MODE (lines 5–14):

> Every time the execution is at line 5, *mode* contains the value of the leftmost mode of the subarray $A[1, \ldots, i-1]$, *m* is the number of times it occurs there, and *s* contains the length of the rightmost run of $A[1, \ldots, i-1]$.

Basis. Suppose the execution is at line 5 for the first time. Then $i = 2$ and so the subarray $A[1, \ldots, i-1]$ is in fact $A[1]$. Obviously, the invariant holds. ✓

Maintenance. Assume the invariant holds at some execution of line 5 such that the body of the loop is to be executed once more.

Case I $A[i]$ equals the leftmost mode of $A[1, \ldots, i-1]$. Clearly, the following hold.

- $A[i]$ equals the leftmost mode of $A[1, \ldots, i]$.

- The number of the occurrences of the leftmost mode in $A[1,\ldots,i]$ is the number of the occurrences of the leftmost mode in $A[1,\ldots,i-1]$ plus one.
- Since $A[\,]$ is sorted, clearly $A[i]$ and $A[i-1]$ are in the same run in $A[1,\ldots,i]$, therefore the length of the rightmost run of $A[1,\ldots,i]$ equals the length of the rightmost run of $A[1,\ldots,i-1]$ plus one.

Now use the inductive hypothesis. According to it, the leftmost mode of $A[1,\ldots,i-1]$ equals *mode*, so the condition at line 6 is TRUE and lines 7 and 8 are executed.

- Since *mode* equals the leftmost mode of $A[1,\ldots,i-1]$, it is the case that *mode* equals the leftmost mode of $A[1,\ldots,i]$.
- Since *m* equals the number of the occurrences of the leftmost mode in $A[1,\ldots,i-1]$ before the assignment at line 7, it is the case that *m* equals the number of the occurrences of the leftmost mode in $A[1,\ldots,i]$ after the assignment at line 7.
- Since *s* equals the length of the rightmost run of $A[1,\ldots,i-1]$ before the assignment at line 9, it is the case that *s* equals the length of the rightmost run of $A[1,\ldots,i]$ after the assignment at line 8.

As the execution goes to line 5 again, i gets incremented by one. Relative to the new value of i, it is the case that *mode* equals the leftmost mode of $A[1,\ldots,i-1]$, *m* equals the number of the occurrences of the leftmost mode in $A[1,\ldots,i-1]$, and *s* equals the length of the rightmost run of $A[1,\ldots,i-1]$. And so the invariant holds.

Case II $A[i]$ does not equal the leftmost mode of $A[1,\ldots,i-1]$. Since $A[\,]$ is sorted, it is the case that $A[i]$ is greater than the leftmost mode of $A[1,\ldots,i-1]$. By the inductive hypothesis, the leftmost mode of $A[1,\ldots,i-1]$ equals *mode*, so the condition at line 6 is FALSE and the execution goes to line 9.

Case II.a $A[i]$ is the leftmost element of some run in $A[\,]$, *i.e.*, $A[i] \neq A[i-1]$. Clearly, the following hold.

- $A[i]$ does not equal the leftmost mode of $A[1,\ldots,i]$ and the leftmost mode of $A[1,\ldots,i]$ is the same as the leftmost mode of $A[1,\ldots,i-1]$.
- The number of the occurrences of the leftmost mode in $A[1,\ldots,i]$ is the same as the number of the occurrences of the leftmost mode in $A[1,\ldots,i-1]$.
- The length of the rightmost run of $A[1,\ldots,i]$ is one.

Now use the inductive hypothesis. According to it,

- *mode* contains the value of the leftmost mode in $A[1,\ldots,i-1]$.
- *m* contains the number of occurrences of the leftmost mode in $A[1,\ldots,i-1]$.

We conclude that

- *mode* contains the value of the leftmost mode in $A[1,\ldots,i]$.
- *m* contains the number of occurrences of the leftmost mode in $A[1,\ldots,i]$.

Since the condition at line 9 is TRUE, the assignment at line 10 is executed and the execution goes to line 5, leaving *mode* and *m* unchanged and setting *s* to one. At line 5 i is incremented by one. Relative to the new value of i,

- *mode* contains the value of the leftmost mode in $A[1,\ldots,i-1]$.
- *m* contains the number of occurrences of the leftmost mode in $A[1,\ldots,i-1]$.
- *s* equals the length of the rightmost run of $A[1,\ldots,i-1]$.

And so the invariant holds.

Case II.b A[i] is not the leftmost element of any run in A[], *i.e.*, $A[i] = A[i-1]$. Since the condition at line 9 is FALSE, the execution goes to line 12. There *s* gets incremented by one. Before the increment the value of *s* equaled the length of the rightmost run in $A[1,\ldots,i-1]$. After the increment it equals the length of the rightmost run of $A[1,\ldots,i]$.

Note that in **Case II.b**, A[i] may or may not be equal to the leftmost mode in $A[1,\ldots,i]$. Let us define a necessary and sufficient condition under which A[i] equals the leftmost mode in $A[1,\ldots,i]$.

Let the rightmost run in $A[1,\ldots,i-1]$ be called Y. Let the run in $A[1,\ldots,i-1]$ whose representative is the leftmost mode be called X. From general considerations we can say A[i] equals the leftmost mode in $A[1,\ldots,i]$ iff the Y is longer than any other run in A[]. In **Case II**, however, A[i] does not equal the leftmost mode of $A[1,\ldots,i-1]$. Conclude that X is to the left of Y and A[i] equals the leftmost mode in $A[1,\ldots,i]$ iff the lengths of X and Y are the same.

Y cannot be longer than any other run in $A[1,\ldots,i-1]$ for if that were the case, A[i] would have been equal to the leftmost mode of $A[1,\ldots,i-1]$, contrary to the assumption of **Case II**. So, the following possibilities are exhaustive: either Y is shorter than X or Y and X have the same length.

Suppose Y is shorter than X. In that case A[i] is not the leftmost mode of $A[1,\ldots,i]$, so the leftmost mode of $A[1,\ldots,i]$ is the same as the leftmost mode of $A[1,\ldots,i-1]$ and the number of occurences are the same in $A[1,\ldots,i-1]$ and $A[1,\ldots,i]$. On the other hand, by the inductive hypothesis *s* before the increment at line 12 equals the length of Y so it is the case that $s < m$ before the increment at line 12. After the increment at line 12 it is the case that $s \leq m$. So the condition at line 13 is FALSE and lines 14 and 15 are not executed and *mode* and *m* remain unchanged. It follows that *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,i]$, *m* is the number of times it occurs there, and *s* contains the length of the rightmost run of $A[1,\ldots,i]$. Then the execution returns to line 5 and i gets incremented by one. Relative to the new value of i, *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,i-1]$, *m* is the number of times it occurs there, and *s* contains the length of the rightmost run of $A[1,\ldots,i-1]$. And so the invariant is preserved.

Suppose Y is as long as X. In that case A[i] is the leftmost mode of $A[1,\ldots,i]$ and the number of its occurrences equals the length of Y plus one. On the other hand, by the inductive hypothesis *s* before the increment at line 12 equals the length of Y so it is the case that $s = m$ before the increment at line 12. After the increment at line 12 it is the case that $s > m$. So the condition at line 13 is TRUE and lines 14 and 15 are executed. Now *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,i]$, *m* is the number of times it occurs there, and *s* contains the length of the rightmost run of $A[1,\ldots,i]$. Then

the execution returns to line 5 and i gets incremented by one. Relative to the new value of i, *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,i-1]$, *m* is the number of times it occurs there, and *s* contains the length of the rightmost run of $A[1,\ldots,i-1]$. And so the invariant is preserved.

Termination. Substitute i with $n+1$ in the first part of the invariant to obtain "At the final execution of line 5, *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,n]$." So at line 16 the retuned value is precisely the leftmost mode of $A[]$. □

⁘ NB ⁘ It is tempting to get rid of the unnecessary assignments to *s*. Consider the following version of the algorithm.

```
COMPUTE MODE OPTIMISED(A[1,2,...,n])
 1  SORT(A[])
 2  mode ← A[1]
 3  m ← 1
 4  for i ← 2 to n
 5      if A[i] = mode
 6          m ← m + 1
 7      else if A[i] ≠ A[i − 1]
 8          s ← 1
 9      else
10          if s > m
11              mode ← A[i]
12              m ← s
13  return mode
```

However, if we are to prove its correctness directly rather than use the correctness of COMPUTE MODE, the loop invariant would be:

> Every time the execution is at line 4, *mode* contains the value of the leftmost mode of the subarray $A[1,\ldots,i-1]$ and *m* is the number of times it occurs there. Furthermore, if $A[i-1]$ does not equal the leftmost mode of $A[1,\ldots,i-1]$ then *s* contains the length of the rightmost run of $A[1,\ldots,i-1]$.

Of course, if $A[i-1]$ equals the leftmost mode of $A[1,\ldots,i-1]$ then *s* is undefined or meaningless.

The proof of that invariant is more involved. For starters, it requires two bases: one for the first execution of line 4 and another one for the first time the condition at line 7 is evaluated and the assignment at line 8 executed. See Problem 157 on page 264 for clarification on why a second basis is needed.

Second Solution:
The code is written by Georgi Georgiev.

```
ALGM(A[1...n])
 1  SORT(A)
 2  i ← 1
```

```
 3  m ← 0
 4  mode ← A[1]
 5  while i ≤ n do
 6      t ← i + 1
 7      while t ≤ n and A[i] = A[t] do
 8          t ← t + 1
 9      if t − i > m
10          m ← t − i
11          mode ← A[i]
12      i ← t
13  return mode
```

Lemma 4. *The inner **while**-loop (lines 7–8) has the following effect. Under the assumption that* $A[i]$ *is the leftmost element of some run in* $A[\,]$*, it is the case that* $A[i, \ldots, t-1]$ *constitutes one run in* $A[\,]$ *after the inner **while**-loop terminates.*

Proof: Relative to any execution of the outer **while**-loop (lines 5–12), the following is an invariant for the inner **while**-loop (lines 7–8):

> Every time the execution is at line 7, the subarray $A[i, \ldots, t-1]$ is subarray of a single run in $A[\,]$.

Basis. The claim is trivially true the first time the execution is at line 7.

Maintenance. Line 7 can be executed only once. In that case there is no maintenance to prove and the proof proceeds with the termination phase.

Assume line 7 is executed more than once and consider an execution of it that is not the last one. It is the case that both $t \leq n$ and $A[i]$ equals $A[t]$. Having assumed $A[i] = A[i+1] = \ldots A[t-1]$, it follows that $A[i] = A[i+1] = \ldots A[t-1] = A[t]$ and so the subarray $A[i, \ldots, t]$ is a subarray of a single run in $A[\,]$. As t gets incremented at line 8, the next time the execution is at line 7, it is the case that $A[i, \ldots, t-1]$ is subarray of a single run in $A[\,]$.

Termination. The last time the execution is at line 7, either $t < n$ and $A[t]$ does not equal $A[i]$, or $t = n + 1$. In both cases $A[t-1]$ is the rightmost element of some run that $A[i]$ is in. Under the assumption that $A[i]$ is the leftmost element of some run, the proof of the lemma follows immediately. □

Lemma 5. *Algorithm* ALGM *returns a mode of* $A[\,]$.

Proof: The following is an invariant for the outer **while**-loop (lines 5–12):

> Every time the execution is at line 12[†], mode contains the value of a mode of $A[1, \ldots, t-1]$ and m contains the number of its occurrences in that subarray. Furthermore, $A[t-1]$ is the rightmost element of some run in $A[\,]$.

Basis. Consider the first execution of the body of the outer **while**-loop. By Lemma 4, when the execution is at line 9, it is the case that $A[1, \ldots, t-1]$ is the leftmost run of

[†]Before the assignment at line 12 takes place.

$A[]$. Its length is $t-1$, that is, $t-i$ because i was assigned 1 at line 2 and it was not modified afterwards. Furthermore, m equals 0 because of the assignment at line 3 and thus the condition at line 9 is necessarily true. Then m is assigned the length of the leftmost run of $A[]$, that run being $A[1,\ldots,t-1]$, and mode is assigned the representative of that run. The claim clearly holds.

Maintenance. If the loop is not executed a second time proceed directly with the termination phase of the proof.

Suppose the claim holds before certain execution of line 12 and the outer **while**-loop is to be executed at least once more. By the inductive hypothesis, $t-1$ is the index of the rightmost element of some run in $A[]$ and it is not the rightmost run. It follows that after the assignment at line 12, i contains the index of the leftmost element of the next run. Call that next run Y. By Lemma 4, the next time the inner **while**-loop is executed, $A[i,\ldots,t-1]$ is a single run. Clearly, $A[i,\ldots,t-1]$ is Y. If Y is longer than any run prior to it then the unique mode of $A[1,\ldots,t-1]$ is the representative of Y, otherwise the representative of some other run is a mode of $A[1,\ldots,t-1]$. According to the inductive hypothesis, such a representative is stored in the variable mode and m is the number of its occurrences.

It follows that the condition at line 9 is true iff Y is the longest run in $A[1,\ldots,t-1]$. In that case the assignments at line 10 and 11 store the representative of Y into mode and the length of Y into m, so the invariant is preserved. Otherwise, the unchanged values of the variable mode and m store a mode of $A[1,\ldots,t-1$ and the number of its occurrences, so the invariant is preserved.

Termination. The last time the execution is at line 12, t equals $n+1$. Then i is assigned $n+1$ and the outer loop terminates. Substitute i with $n+1$ in the invariant to obtain "mode contains the value of a mode of $A[1,\ldots,n]$". Conclude that the returned value is a mode of $A[]$. □

4.3.2 Proving the correctness of Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, and Quick Sort

Consider the following proof of correctness by loop invariant of Insertion Sort, based on [CLR00]. The goal is to prove that Insertion Sort indeed sorts its input.

```
Insertion Sort(A[1,2,...n]: array of integers)
1  for i ← 2 to n
2      key ← A[i]
3      j ← i − 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← key
```

There are two loops in that algorithm. A very precise formal proof should consist of two separate invariants:

- an invariant concerning the inner **while** loop—it should be stated and proved relative to j, the control variable of the inner loop;

- an invariant concerning the outer **for** loop—it should be stated and proved relative to $\mathtt{i}$.

The algorithm moves the elements of A[ ] around. Because of that, making a precise argument can be tricky: for instance, when we mention A[i] in Lemma 6, do we mean the element of A at position $\mathtt{i}$ before, or after, the inner **while** loop has shifted a certain subarray "upwards" by one position? Position $\mathtt{i}$ can be affected by the shift, so in general those are different elements. We overcome that potential ambiguity by giving different names to the whole array, or parts of it, in different moments of the execution.

Lemma 6. *Consider algorithm* Insertion Sort. *Relative to any execution of the* ***for*** *loop (lines 1–7), let us call the subarray* $A[1,\ldots,i]$ *before the* ***while*** *loop (lines 4–6) starts being executed,* $A'[1,\ldots,i]$*, and let us call it* $A''[1,\ldots,i]$*, after it is executed. Assume the subarray* $A'[1,\ldots,i-1]$ *is sorted. The* ***while*** *loop has the following effects:*

- $\mathtt{j}$ *is assigned the biggest number from* $\{1,2,\ldots,i-1\}$ *such that* $A'[j] \leq \mathtt{key}$*, if such a number exists, or is assigned* 0*, otherwise.*
- *with respect to that value of* $\mathtt{j}$*, it shifts the subarray* $A'[j+1,\ldots,i-1]$ *by one position upwards.*

Proof: The following is a loop invariant for the **while** loop:

Every time the execution reaches line 4:

- for every element $\mathtt{x}$ of the current subarray $A[j+2,\ldots,i]$, $\mathtt{x} > \mathtt{key}$, and
- the current subarray $A[j+2,\ldots,i]$ is the same as $A'[j+1,\ldots,i-1]$.

Basis. The first time the execution reaches line 4, it is the case that $\mathtt{j} = \mathtt{i}-1$. So, the current subarray $A[j+2,\ldots,i]$ is in fact $A[i+1,\ldots,i]$. Since that is an empty subarray, the first part of the invariant is vacuously true. The second part of the invariant is true as well because both subarrays it concerns are empty.

Maintenance. Assume the claim holds at a certain moment $\mathtt{t}$ when the execution is at line 4 and the **while** loop is to be executed at least once more. The latter means that $\mathtt{j} > 0$ and $A[j] > \mathtt{key}$. After line 5, it is the case that $A[j+1] > \mathtt{key}$, and that is relative to the value of $\mathtt{j}$ that the current iteration began with. By the first part of the invariant, for every element $\mathtt{x}$ of the current subarray $A[j+2,\ldots,i]$, $\mathtt{x} > \mathtt{key}$. We conclude that for every element $\mathtt{x}$ of the current subarray $A[j+1,\ldots,i]$, $\mathtt{x} > \mathtt{key}$. At line 6, $\mathtt{j}$ is decremented. Relative to the new value of $\mathtt{j}$, the previously stated conclusion becomes, for every element $\mathtt{x}$ of the current subarray $A[j+2,\ldots,i]$, $\mathtt{x} > \mathtt{key}$. We proved the first part of the invariant.

Consider the execution at moment $\mathtt{t}$ again. Consider the second part of the invariant. According to it, the current subarray $A[j+2,\ldots,i]$ is the same as $A'[j+1,\ldots,i-1]$. Then the execution of the loop body commences. At line 5, the value of the current A[j] is assigned to $A[j+1]$. But element A[j] has not been modified by the **while** loop so far[†]—a fact which is fairly obvious and does not necessitate inclusion in the invariant—so A[j] is in fact $A'[j]$. It follows the current subarray $A[j+1,\ldots,i]$ is the same as $A'[j,\ldots,i-1]$ after the assignment at line 5 takes place. Then $\mathtt{j}$ gets decremented (line 6). When the execution

[†]Remember that here we consider only the executions of the **while** loop relative to the current execution of the outer **for** loop, not all executions of the **while** loop since the start of the algorithm.

is at line 4 again, relative to the new value of j, it is the case that $A[j+2,\ldots,i]$ is the same as $A'[j+1,\ldots,i-1]$. We proved the second part of the invariant.

Termination. Consider the moment when the execution is at line 4 and the condition there is **False**. That is, $j \leq 0$ or $A[j] \leq \mathit{key}$.

Case i. First assume that $j \leq 0$. Since j is decremented by one, it cannot be negative, so it is the case that $j = 0$. Plug the value 0 for j in the invariant to obtain that:

- for every element x of the current subarray $A[2,\ldots,i]$, $x > \mathit{key}$, and
- the current subarray $A[2,\ldots,i]$ is the same as $A'[1,\ldots,i-1]$.

The first part of the invariant means there is no number j from $\{1,2,\ldots,i-1\}$ such that $A'[j] \leq \mathit{key}$. But j is assigned 0 and so the first claim of this Lemma is true. The second part of the invariant means that, relative to the value 0 for j, the original subarray $A[j+1,\ldots,i-1]$ has been shifted one position upwards. So, the Lemma holds when $j = 0$.

Case ii. Now assume that $j > 0$ and $A[j] \leq \mathit{key}$. But $A[j]$ has never been modified by the **while** loop. Therefore, it is the case that $A'[j] \leq \mathit{key}$.

The invariant says that, on the one hand, $A[j+2] > \mathit{key}$, $A[j+3] > \mathit{key}$, *etc.*, $A[i] > \mathit{key}$, and on the other hand, that $A[j+2] = A'[j+1]$, $A[j+3] = A'[j+2]$, $A[i] = A'[i-1]$. It follows that $A'[j+1] > \mathit{key}$, $A'[j+2] > \mathit{key}$, *etc.*, $A'[i-1] > \mathit{key}$.

The two facts above imply that indeed j is assigned the biggest number from $\{1,2,\ldots,i-1\}$ such that $A'[j] \leq \mathit{key}$. So, the first claim of the Lemma holds. The second claim of the Lemma is literally the same as the the second part of the invariant. That concludes the proof of the Lemma. □

Lemma 7. *Algorithm* Insertion Sort *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1,\ldots,n]$, and let us call $A''[1,\ldots,n]$ the array after the algorithm halts. The following is a loop invariant for the **for** loop:

> Every time the execution of Insertion Sort is at line 1, the current subarray $A[1,\ldots,i-1]$ consists of the same elements as $A'[1,\ldots,i-1]$, but in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i = 2$. The subarray $A[1,\ldots,1]$ consists of a single element that is clearly the same as $A'[1]$ and it is, in a trivial sense, sorted.

Maintenance. Assume the claim holds at a certain execution of line 1 and the **for** loop is to be executed at least once more. Let $\widetilde{A}[1,\ldots,i]$ be the name of $A[\,]$ when the execution of the **for** loop commences. The current value of $A[i]$, *i.e.* $\widetilde{A}[i]$, is stored in key, j is set to $i-1$, and the inner **while** loop is executed. By Lemma 6, the effects of the **while** loop are the following:

- j is assigned the biggest number from $\{1,2,\ldots,i-1\}$ such that $\widetilde{A}[j] \leq \mathit{key}$, if such a number exists, or is assigned 0, otherwise.
- with respect to that value of j, the subarray $\widetilde{A}[j+1,\ldots,i-1]$ is shifted by one position upwards.

If there are elements in $\widetilde{A}[1,\ldots,i-1]$ that are bigger than $\texttt{key} = \widetilde{A}[i]$, they are stored in a contiguous sorted subsequence – that follows from the assumption at the beginning of the **Maintenance** phase. Lemma 6 implies that the index of the smallest of those is $j+1$. Lemma 6 further implies that $\widetilde{A}[j+1,\ldots,i-1]$ is shifted into the current $A[j+2,\ldots,i]$. Thus in the current $A[\,]$, $A[j+1] = A[j+2]$ and therefore the assignment at line 7 overwrites a value that has already been copied into another position. Clearly, at the end of the **for** loop, $A[1,\ldots,i]$ consists of the same elements as $\widetilde{A}[1,\ldots,i]$ but in sorted order.

It remains to consider the case when no elements in $\widetilde{A}[1,\ldots,i-1]$ are bigger than $\texttt{key} = \widetilde{A}[i]$. By Lemma 6, j equals $i-1$ at the end of the **while** loop and nothing has been shifted upwards, thus the assignment at line 7 overwrites the i-th element of A with the value it had at the start of the **for** loop, namely $\widetilde{A}[i]$. Clearly, at the end of the **for** loop, $A[1,\ldots,i]$ consists of the same elements as $\widetilde{A}[1,\ldots,i]$ but in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, i equals $n+1$. Plug the value $n+1$ in place of i in the invariant to obtain "the current subarray $A[1,\ldots,(n+1)-1]$ consists of the same elements as $A'[1,\ldots,(n+1)-1]$, but in sorted order". □

```
SELECTION SORT(A[1,2,...n]: array of integers)
1  for i ← 1 to n − 1
2      for j ← i + 1 to n
3          if A[j] < A[i]
4              swap(A[i], A[j])
```

The following two lemmas concern the correctness of SELECTION SORT. It is obvious that SELECTION SORT permutes the elements of the input because the only changes is does to $A[\,]$ happen at line 4, using swaps. That is unlike INSERTION SORT where it is not (so) obviuos that at the end, the elements in the array are the same as the original ones. So in the analysis of INSERTION SORT we did concern ourselves with proving that no original element gets overwritten before its value is stored safely somewhere else. In the analysis of SELECTION SORT we have no such concerns.

Lemma 8. *With respect to a particular execution of the outer* ***for*** *loop (lines 1–4) of* SELECTION SORT*, the execution of the inner* ***for*** *loop (lines 2–4) has the effect that* $A[i]$ *is a smallest element in* $A[i,\ldots,n]$.

Proof:
With respect to a certain execution of the outer **for** loop, the following is a loop invariant for the inner **for** loop:

> Every time the execution reaches line 2, the current $A[i]$ holds the value of a minimum element from $A[i,\ldots,j-1]$.

Basis. The first time the execution of the inner **for** loop is at line 2, it is the case that $j = i+1$. Then $A[i]$ is trivially a minimum element in $A[i,\ldots,(i+1)-1]$.

Maintenance. Assume the claim is true at some moment when the execution is at line 2 and the inner **for** loop is to be executed once more. The following two cases are exhaustive.

Case i. $A[j] < A[i]$. The condition at line 3 is TRUE and the swap at line 4 takes place. By the maintenance hypothesis, $A[i]$ is a minimum in $A[i, \ldots, j-1]$. Since $A[j] < A[i]$, by the transitivity of the $<$ relation, $A[j]$ is the minimum element in $A[i, \ldots, j]$ before the swap. We conclude $A[i]$ is the minimum element in $A[i, \ldots, j]$ after that swap. Then j gets incremented by one and the execution goes to line 2. With respect to the new value of j, it is the case that $A[i]$ is the minimum element in $A[i, \ldots, j-1]$.

Case ii. $A[j] \not< A[i]$. Then the condition at line 3 is FALSE and the swap at line 4 does not take place. By the maintenance hypothesis, $A[i]$ is a minimum element in $A[i, \ldots, j-1]$. Since $A[i] \leq A[j]$, clearly $A[i]$ is a minimum element in $A[i, \ldots, j]$. Then j gets incremented by one and the execution goes to line 2. With respect to the new value of j, it is the case that $A[i]$ is a minimum element in $A[i, \ldots, j-1]$.

Termination. Consider the moment when the execution is at line 2 for the last time. Clearly, j equals $n+1$. Plug the value $n+1$ in place of j in the invariant to obtain "the current $A[i]$ holds the value of a minimum element from $A[i, \ldots, (n+1)-1]$". □

Lemma 9. *Algorithm* SELECTION SORT *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1, \ldots, n]$. The following is a loop invariant for the outer **for** loop (lines 1–4):

> Every time the execution of SELECTION SORT is at line 1, the current subarray $A[1, \ldots, i-1]$ consists of $i-1$ in number smallest elements from $A'[1, \ldots, n]$, in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i = 1$. The current subarray $A[1, \ldots, i-1]$ is empty and thus, vacuously, it consists of the zero in number smallest elements from $A'[1, \ldots, n]$, in sorted order.

Maintenance. Assume the claim holds at a certain execution of line 1 and the outer **for** loop is to be executed at least once more. Let us call the array $A[\,]$ at that moment, $A''[\,]$. By Lemma 8, the effect of the inner **for** loop is that it stores into the i^{th} position a smallest value from $A''[i, \ldots, n]$. On the other hand, by the maintenance hypothesis, $A''[1, \ldots, i-1]$ consists of $i-1$ in number, smallest elements from $A'[1, \ldots, n]$, in sorted order. We conclude that at the end of that execution of the outer **for** loop, the current $A[1, \ldots, i]$ consists of i in number, smallest elements from $A'[1, \ldots, n]$, in sorted order. Then i gets incremented by one and the execution goes to line 1. With respect to the new value of i, it is the case that the current $A[1, \ldots, i-1]$ consists of $i-1$ in number, smallest elements from $A'[1, \ldots, n]$, in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, i equals n. Plug the value n in place of i in the invariant to obtain "the current subarray $A[1, \ldots, n-1]$ consists of the smallest, $n-1$ in number, elements from $A'[1, \ldots, n]$, in sorted order". But then $A[n]$ has to be a maximum element from $A'[1, \ldots, n]$. And that concludes the proof of the correctness of SELECTION SORT. □

```
BUBBLE SORT(A[1, 2, ... n]: array of integers)
1  for i ← 1 to n
2      for j ← n downto i + 1
```

```
3          if A[j − 1] > A[j]
4              swap(A[j − 1], A[j])
```

Lemma 10. *With respect to a particular execution of the outer* ***for*** *loop (lines 1–4) of* Bubble Sort*, the execution of the inner* ***for*** *loop (lines 2–4) has the effect that* $A[i]$ *is a smallest element in* $A[i, \ldots, n]$.

Proof:
With respect to a certain execution of the outer **for** loop, the following is a loop invariant for the inner **for** loop:

> Every time the execution reaches line 2, the current $A[j]$ holds the value of a minimum element from $A[j, \ldots, n]$.

Basis. The first time the execution of the inner **for** loop is at line 2, it is the case that $j = n$. Then $A[n]$ is trivially a minimum element in $A[n, \ldots, n]$.

Maintenance. Assume the claim is true at some moment when the execution is at line 2 and the inner **for** loop is to be executed once more. The following two cases are exhaustive.

Case i. $A[j-1] > A[j]$. The condition at line 3 is True and the swap at line 4 takes place. By the maintenance hypothesis, $A[j]$ is a minimum in $A[j, \ldots, n]$ at the before the swap. Since $A[j-1] > A[j]$, $A[j]$ is a minimum element in $A[j-1, \ldots, n]$ before the swap. After the swap, clearly $A[j-1]$ is a minimum element in $A[j-1, \ldots, n]$. Since j is decremented by one the next time the execution is at line 2, with respect to the new j, it is the case that $A[j]$ is a minimum element from $A[j, \ldots, n]$.

Case ii. $A[j-1] \not> A[j]$, *i.e.* $A[j-1] \leq A[j]$. The condition at line 3 is False and the swap at line 4 does not take place. By the maintenance hypothesis, $A[j]$ is a minimum in $A[j, \ldots, n]$ at the before the evaluation at line 3. By the transitivity of the $\leq$ relation, $A[j-1]$ is a minimum element in $A[j-1, \ldots, n]$. Since j is decremented by one the next time the execution is at line 2, with respect to the new j, it is the case that $A[j]$ is a minimum element from $A[j, \ldots, n]$.

Termination. Consider the moment when the execution is at line 2 for the last time. Then j equals i. Plug the value i in place of j in the invariant to obtain "$A[i]$ holds the value of a minimum element from $A[i, \ldots, n]$". □

Lemma 11. *Algorithm* Bubble Sort *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1, \ldots, n]$. The following is a loop invariant for the outer **for** loop (lines 1–4):

> Every time the execution of Bubble Sort is at line 1, the current subarray $A[1, \ldots, i-1]$ consists of $i-1$ in number smallest elements from $A'[1, \ldots, n]$, in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i = 1$. Obviously, the current $A[1, \ldots, i-1]$ is empty and the claim is vacuously true.

Maintenance. Assume the claim holds at a certain execution of line 1 and the outer **for** loop is to be executed at least once more. Let us call the array $A[\,]$ at that moment, $A''[\,]$.

By Lemma 10, the effect of the inner **for** loop is that it stores into the i^{th} position a smallest value from $A''[i,\ldots,n]$. On the other hand, by the maintenance hypothesis, $A''[1,\ldots,i-1]$ consists of $i-1$ in number, smallest elements from $A'[1,\ldots,n]$, in sorted order. We conclude that at the end of that execution of the outer **for** loop, the current $A[1,\ldots,i]$ consists of i in number, smallest elements from $A'[1,\ldots,n]$, in sorted order. Then i gets incremented by one and the execution goes to line 1. With respect to the new value of i, it is the case that the current $A[1,\ldots,i-1]$ consists of $i-1$ in number, smallest $i-1$ elements from $A'[1,\ldots,n]$, in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, j equals $n+1$. Plug the value n in place of i in the invariant to obtain "the current subarray $A[1,\ldots,(n+1)-1]$ consists of the smallest, $(n+1)-1$ in number, elements from $A'[1,\ldots,n]$, in sorted order.". □

MERGE($A[1,2,\ldots n]$: array of integers; l, mid, h: indices in $A[]$)

```
 1  (* the subarrays A[l,...,mid] and A[mid+1,...,h] are sorted *)
 2  n1 ← mid − l + 1
 3  n2 ← h − mid
 4  create L[1,...,n1+1] and R[1,...,n2+1]
 5  L ← A[l,...,mid]
 6  R ← A[mid+1,...,h]
 7  L[n1+1] ← ∞
 8  R[n2+1] ← ∞
 9  i ← 1
10  j ← 1
11  for k ← l to h
12      if L[i] ≤ R[j]
13          A[k] ← L[i]
14          i ← i+1
15      else
16          A[k] ← R[j]
17          j ← j+1
```

Lemma 12. *On the assumption that the subarrays $A[l,\ldots,mid]$ and $A[mid+1,\ldots,h]$ are sorted, the whole array $A[l,\ldots,h]$ is sorted after* MERGE *terminates.*

Proof:
The following is a loop invariant for the **for** loop (lines 11–17):

> **part i** Every time the execution of MERGE is at line 11, $A[l,\ldots,k-1]$ contains $k-l$ smallest elements of $L[]$ and $R[]$, in sorted order.
>
> **part ii** Furthermore, $L[i]$ and $R[j]$ are smallest elements in $L[]$ and $R[]$, respectively, that have not been copied into $A[]$ yet.

Basis. When the execution is at line 11 for the first time, it is the case that $k=l$. Then the subarray $A[l,\ldots,k-1]$ is in fact $A[l,\ldots,l-1]$, *i.e.* an empty subarray. Vacuously, it is sorted and contains the $k-l=0$ smallest elements of $L[]$ and $R[]$, in sorted order. Furthermore, $L[1]$ and $R[1]$ are smallest elements of $L[]$ and $R[]$, respectively, that have not been copied into $A[]$.

Maintenance. Assume the claim holds at a certain execution of line 11 and the **for** loop is to be executed at least once more. There are two alternative ways the execution can take through the body of the loop. We consider them both. Before we do that we prove an useful auxilliary result; both L[] and R[] contain a ∞ value so we want to be sure that those are never compared.

> *In the comparison at line 12, it cannot be the case that two ∞ values are compared.*
>
> **Proof:** Assume the opposite. By the maintenance hypothesis, $k - l$ elements are copied into A from L[] and R[]. By the maintenance hypothesis, the body of the loop is to be executed once more, so the number of the copied elements is $\leq h - l$. By the assumption we made, we have copied all $n_1 + n_2$ elements that are not ∞, so the number of copied elements is $\geq n_1 + n_2 = mid - l + 1 + h - mid = h - l + 1 > h - l$. $\lightning$

Consider the comparison at line 12. Since both L[i] and R[j] cannot be ∞, the result of the comparison is always defined. First assume L[i] $\leq$ R[j]. Clearly, L[i] $< \infty$. By **part ii** of the maintenance hypothesis and the assumption that L[i] $\leq$ R[j] we conclude L[i] is a smallest element in L[] and R[] that has not been copied yet. By **part i** of the maintenance hypothesis, L[i] is not smaller than any element in $A[l, \ldots, k\text{-}1]$. The execution goes to line 13. So, now $A[k]$ is not smaller that any element in $A[l, \ldots, k\text{-}1]$. It follows $A[l, \ldots, k]$ is sorted and contains the $k - l + 1 = (k + 1) - l$ smallest elements of L[] and R[]. But k gets incremented the next time the execution is at line 11. Relative to the new value of k, it is the case that $A[l, \ldots, k\text{-}1]$ contains $k - l$ smallest elements of L[] and R[], in sorted order. So, **part i** of the invariant holds.

Now we prove that **part ii** of the invariant holds as well. By assumption, L[] and R[] are sorted. Before the assignment at line 13, L[i] was a smallest element from L[] not copied into A[] yet. After that assignment, L[i + 1] is a smallest element from L[] not copied into A[] yet. But i get incremented at line 14. With respect to the new value of i, L[i] is a smallest element from L[] not copied into A[] yet.

Now assume the execution is still at line 12 and L[i] $\not\leq$ R[j], *i.e.* L[i] $>$ R[j]. The proof is completely analogous to the proof above.

Termination. The loop control variable k equals $h + 1$ the last time the execution is at line 11. Plug that value into the invariant to obtain "the subarray $A[l, \ldots, h - 1]$ contains $k - l$ smallest elements of L[] and R[], in sorted order". □

MERGE SORT(A[1, 2, ..., n]: array of integers; l, h: indices in A[])
1 **if** $l < h$
2 $\quad mid \leftarrow \lfloor \frac{l+h}{2} \rfloor$
3 $\quad$ MERGE SORT(A, l, mid)
4 $\quad$ MERGE SORT(A, $mid + 1$, h)
5 $\quad$ MERGE(A, l, mid, h)

Lemma 13. *Algorithm* MERGE SORT *is a correct sorting algorithm if the initial call is* MERGE SORT(A, 1, n).

Proof:

By induction on the difference $h - l$[†]. We consider it obvious that $h - l$ can get as small as zero but not any smaller. So, the basis is $h - l = 0$.

Basis. $h = l$. On the one hand, the array $A[l]$ is trivially sorted. On the other hand, MERGE SORT does nothing when $h = l$. So, the one element array remains sorted at the end.

Maintenance. Assume that MERGE SORT sorts correctly the subarrays $A[l, \ldots, mid]$ and $A[mid + 1, \ldots, h]$ (lines 3 and 4) during any recursive call such that $h > l$. Use Lemma 12 to conclude that at the end of the current call, the whole $A[l, \ldots, h]$ is sorted.

Termination. When proving facts about recursive algorithms, using induction rather than loop invariant, the termination step of the proof concerns the termination of the very first recursive call. In this proof that is MERGE SORT$(A[1, \ldots, n])$. With MERGE SORT, this step is trivial: simply observe that after the algorithm finishes altogether, $A[1, \ldots, n]$ is sorted. □

PARTITION($A[1, 2, \ldots, n]$: array of integers; l, h: indices in $A[]$)

```
1  pivot ← A[h]
2  pp ← l
3  for i ← l  to  h − 1
4      if A[i] < pivot
5          swap(A[i], A[pp])
6          pp ← pp + 1
7  swap(A[pp], A[h])
8  return pp
```

Lemma 14. *The value* pp *returned by* PARTITION *is such that* $l \leq pp \leq h$ *and*

$$\forall x \in A[l, \ldots, pp - 1], \forall y \in A[pp + 1, \ldots, h] : x < A[pp] \leq y$$

Proof:
At any moment, let S denote the current set $\{x \in A[l, \ldots, i-1] \mid x < pivot\}$. The following is a loop invariant for the **for** loop (lines 3–6):

> Every time the execution of PARTITION is at line 3, $pp \leq i$ and the elements of S are precisely the elements in $A[l, \ldots, pp - 1]$.

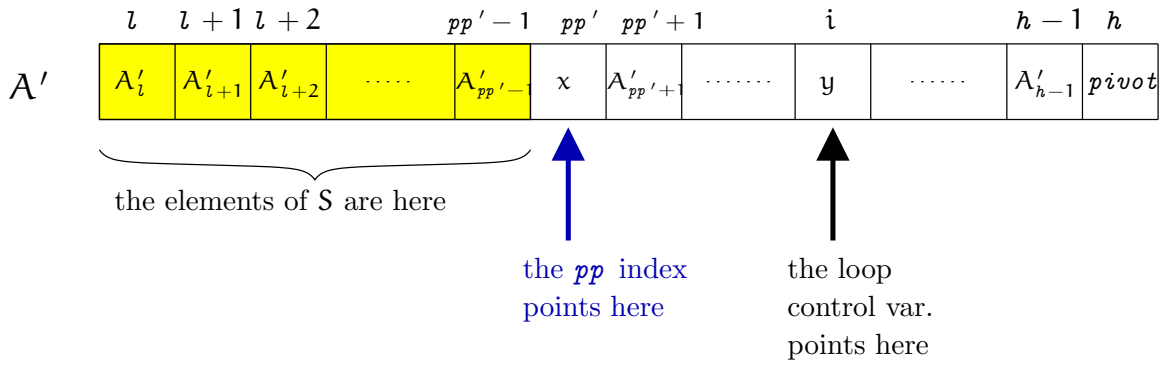
Basis: $i = l$. Because of the assignement at line 2, $pp \leq i$ holds. The subarray $A[l, \ldots, i-1]$ is $A[l, \ldots, l - 1]$, an empty subarray, therefore $S = \emptyset$, so it is vacuously true that the elements of S are precisely the elements in $A[l, \ldots, pp - 1]$.

Maintenance: Assume the claim holds at a certain moment when the execution is at line 3 and $i \leq h - 1$. Let $A'[]$ and pp' denote the array $A[]$ and the index variable pp, respectively, at the beginning of that the **for** loop execution.

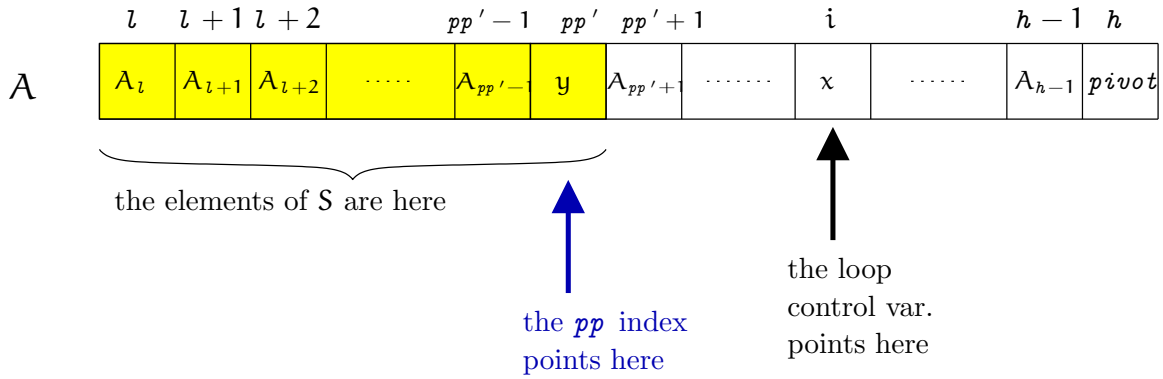
Case 1: $A'[i] < pivot$. By the maintenance hypothesis, $A'[pp]$ is the leftmost element that is left of the current i^{th} position and is not smaller than $pivot$. All the elements that are smaller than $pivot$ and left of the current i^{th} position—namely, the elements of S—consitute the subarray left of the current pp^{th} position. Let the values of $A'[pp]$ and

[†]Note that this is not a proof with loop invariant because MERGE SORT is not an iterative algorithm.

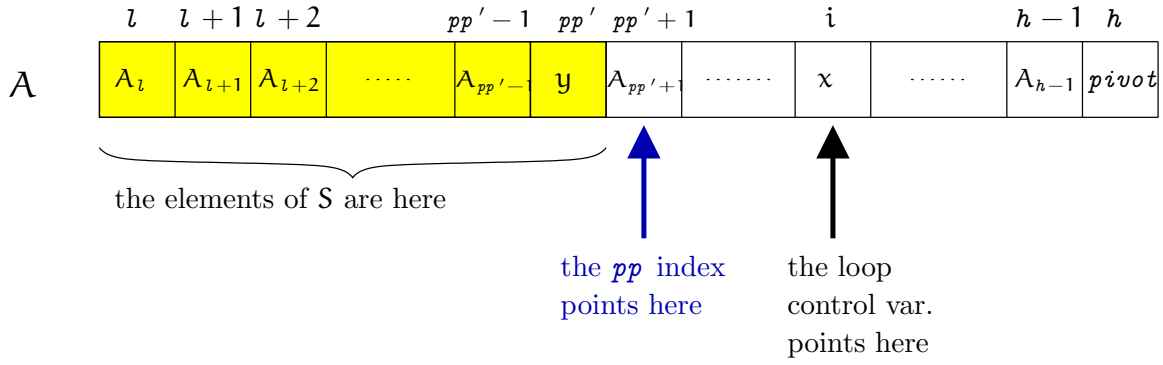
A′[i] be called x and y, respectively. See the following figure. The elements of S are outlined in yellow. For brevity we write A_l rather than $A[l]$, *etc.*



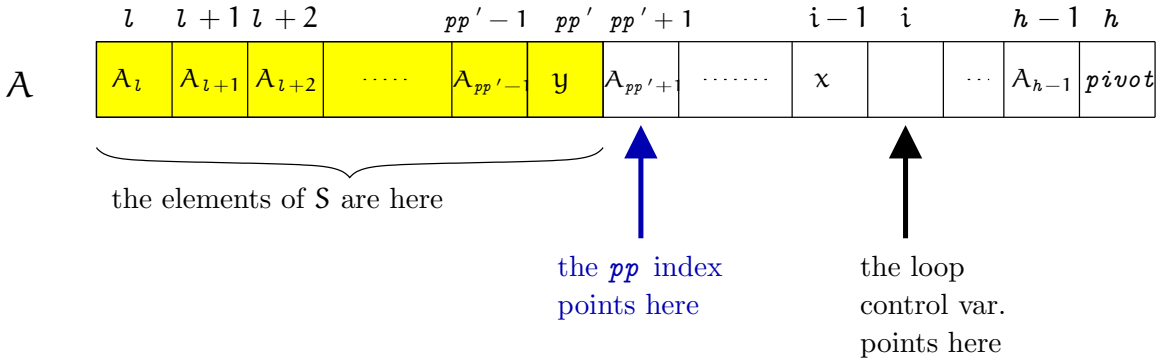
The condition at line 4 is TRUE and so the execution proceeds to line 5 where x and y get swapped. Note that S "grows" by one element, namely y:



At line 6, *pp* is incremented by one:

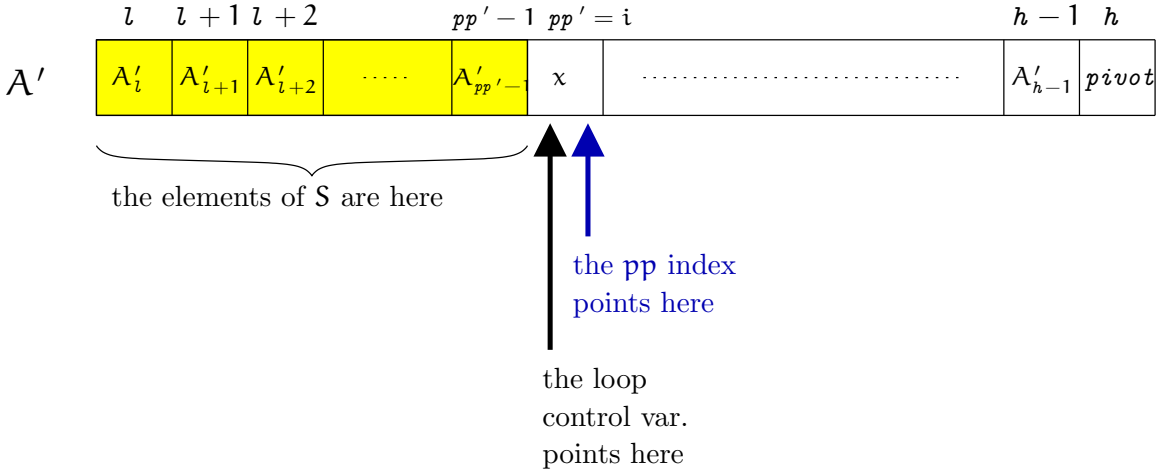


Next the execution is at line 3 again, with i incremented by one. Relative to the current i, we have the following picture:

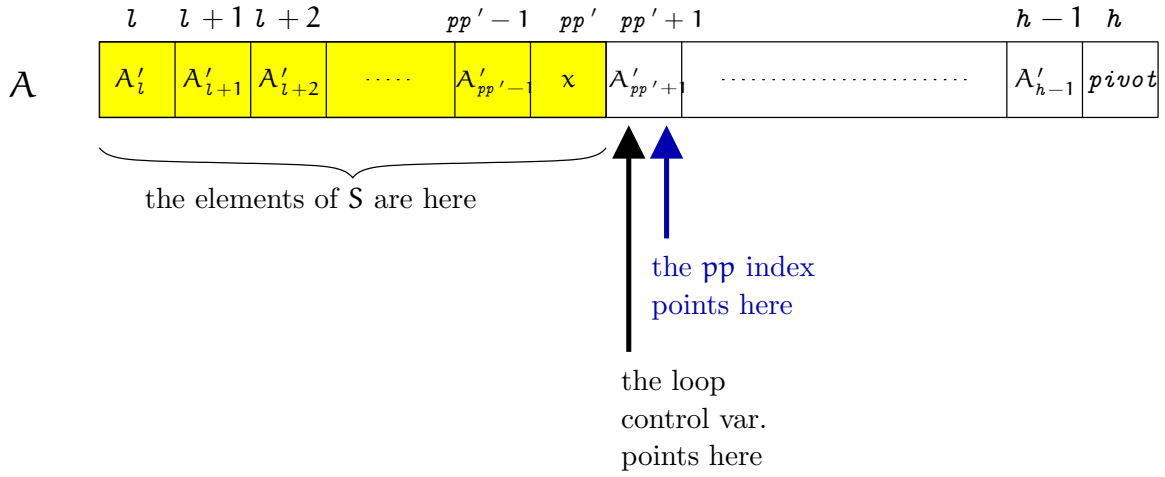


Relative to the current values of i and pp, the elements in $A[l, \ldots, i-1]$ smaller than $pivot$ are precisely the elements in $A[l, \ldots, pp-1]$. Furthermore, $pp \leq i$. And so the invariant holds.

There is no loss of generality in considering the case $pp' < i$ as we did here. However, if the reader is not convinced, here is what happens when $pp' = i$ at the start of the iteration whose execution we follows. Let x be the value of the element at position $pp' = i$:



The condition at line 4 is true under the premise of **Case 1**, so the swap at line 5 takes place. However, nothing changes because x is swapped with itself. Then both pp and i get incremented by one. Relative to their new values, S is one element bigger than it was at the beginning of the current iteration – it has "grown", x being added to it. However, it is still the case that $pp \leq i$ and the elements of S are precisely the elements in $A[l, \ldots, pp-1]$:



Case 2: $A'[i] \geq pivot$. In this case, the condition at line 4 is FALSE and so the execution proceeds directly to line 3 with the loop control variable being incremented by one. The invariant holds because of the maintenance hypothesis and the facts that no element in the array is moved, pp equals pp', and $pp \leq i$ relative to the new value of i, and S is the same relative to the new value of i.

Termination: When the **for** loop terminates, it is the case that $i = h$. Now S is the set of the elements in $A[l, \ldots, h-1]$ smaller than $pivot$. But it is also true that S consists of the elements in $A[l, \ldots, h]$ smaller than $pivot$. By the loop invariant, the elements of S form the contiguous subarray $A[l, \ldots, pp-1]$. After the assignment at line 7, it is the case that

$$\forall x \in A[l, \ldots, pp-1], \forall y \in A[pp+1, \ldots, h] : x < A[pp] \leq y$$

And finally at line 8, PARTITION returns pp, so the claim of this Lemma is true. □

QUICK SORT($[A1, 2, \ldots, n]$: array of integers; l, h: indices in $A[]$)
1 **if** $l < h$
2 $\quad mid \leftarrow$ PARTITION(A, l, h)
3 $\quad$ QUICK SORT($A, l, mid - 1$)
4 $\quad$ QUICK SORT($A, mid + 1, h$)

Lemma 15. *Algorithm* QUICK SORT *is a correct sorting algorithm if the initial call is* QUICK SORT$(A, 1, n)$.

Proof:
By induction on the difference $h - l$. We consider it obvious that $h - l$ can get as small as zero but not any smaller. So, the basis is $h - l = 0$.

Basis. $h = l$. On the one hand, the array $A[l]$ is trivially sorted. On the other hand, QUICK SORT does nothing when $h = l$. So, the one element array remains sorted at the end.

Maintenance. Assume that $h > l$. Use Lemma 14 to conclude that after the call to PARTITION (line 2) returns mid, it is the case that $A[]$ is modified in such a way that left

of $A[mid]$ are precisely the elements of the original $A[]$ smaller than $A[mid]$, and right of $A[mid]$ are the elements not smaller than it. Assuming that the two recursive calls at lines 3 and 4 sort correctly the respective subarrays, clearly $A[l, \ldots, h]$ is sorted at the end of the current recursive call.

Termination. Consider the termination of the first recursive call QUICK SORT$(A[1, \ldots, n])$ and conclude that $A[1, \ldots, n]$ is sorted. □

4.3.3 The correctness of algorithms on binary heaps

First let us clarify that by heaps we mean binary max heaps. The algorithms concerning heaps use the following primitive operations:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

In the following definitions and discussion we assume the array $A[]$ is used to represent a complete binary tree. *A complete binary tree* is a binary tree such that every level, except possibly for the last level, is completely filled. If the last level is incomplete then its nodes must be as left as possible. *A perfect binary tree* is a complete binary tree in which the last level is complete. For the classification of trees as data structures, see [NIS]. When we say tree, we always mean binary tree.

Definition 5. *Let* $A[1, 2, \ldots, n]$ *be an array of integers*[†] *and* i *be an index such that* $1 \leq i \leq n$. *We call* "the complete subtree in $A[]$ rooted at i" *—denoted by* $A[\![i]\!]$*—the not necessarily contiguous subarray of* $A[]$ *induced by the indices generated by the following function:*

```
GENERATE INDICES(A[1, ..., n]: array of integers, i: index in A[])
1  print i
2  left ← LEFT(i)
3  right ← RIGHT(i)
4  if left ≤ n
5      GENERATE INDICES(A[], left )
6  if right ≤ n
7      GENERATE INDICES(A[], right)
```

□

Clearly, $A[\![1]\!]$ is the array $A[1, \ldots, n]$ itself. Since $A[]$ represents a tree, $A[1]$ is the root and $A[\![i]\!]$ is the subtree rooted at vertex $A[i]$. When we use Graph Theory terminology and more specifically, terminology that concerns rooted trees, on arrays, we of course have in

[†]The heaps we consider have elements–integers. Of course, any other data type whose elements take $O(1)$ memory and allows comparing and moving elements around in $O(1)$ time could be used.

mind that the array represents a rooted tree. So the leaves of A[] are the elements that correspond to the leaves of the tree that A[] represents. By Problem 153 on page 251, the number of leaves is precisely $\lceil \frac{n}{2} \rceil$ and thus the number of non-leaves is $\lfloor \frac{n}{2} \rfloor$. It follows the leaves of $A[1, \ldots, n]$ are precisely $A\left[\lfloor \frac{n}{2} \rfloor + 1\right]$, $A\left[\lfloor \frac{n}{2} \rfloor + 2\right]$, $\ldots$, $A[n]$. Furthermore, the predicate that indicates whether an array element is a leaf or not is

$$\textsc{IsLeaf}(i) = \begin{cases} \textsc{True}, & \text{if } i > \lfloor \frac{n}{2} \rfloor \\ \textsc{False}, & \text{else} \end{cases}$$

Definition 6. *Let* T *be any rooted tree and* u *be any vertex in it.* The height *of* u *is the distance between it and the root. The height of* T *is the maximum height of any vertex in* T. The depth[†] *of* u *is the length of any longest path between* u *and a leaf which path does not contain vertices of height smaller than the height of* u. □

The depths of the vertices in a complete binary tree is illustrated on Figure 8.3 on page 253. When we speak of the depth of an element of A[] we have in mind the depth of the corresponding vertex of the tree A[] represents. Of course, the elements of depth 0 are the leaves. As an example consider the following 26-element array A[]:

A	6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

The leaves of A[] are elements A[13]–A[26], shown in yellow:

A	6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Suppose $i = 3$. A⟦3⟧ is the (non-contiguous) subarray of A[] outlined in green:

6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

The leaves of A⟦3⟧ are the five elements A[14], A[15], A[24]–A[26], outlined in blue:

6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Definition 7 (heap)**.** A⟦i⟧ *is a heap iff:*

- *either* $\textsc{IsLeaf}(i)$,
- *or*
  - $A⟦\textsc{Left}(i)⟧$ *is a heap and* $A[i] \geq A[\textsc{Left}(i)]$,
  - $A⟦\textsc{Right}(i)⟧$*, if it exists, is a heap, and* $A[i] \geq A[\textsc{Right}(i)]$. □

[†] In [CLR00], the authors call "height of a node" what we call "depth".

We consider the following fact obvious.

Fact: If $A[\![i]\!]$ is a heap then for any index j such that $A[j]$ is in $A[\![i]\!]$, $A[\![j]\!]$ is a heap. □

In the context of heaps, relative to some $A[\,]$ and index i in it, let us call *a heap inversion* every pair of indices $\langle j, k\rangle$ such that:

- $A[j]$ and $A[k]$ are in $A[\![i]\!]$,
- $A[j] < A[k]$, and
- $k = \textsc{Left}(j)$ or $k = \textsc{Right}(j)$.

Clearly, $A[\![i]\!]$ is a heap iff it has no heap inversions.

```
ITERATIVE HEAPIFY(A[1,2,...,n]: array of integers, i: index in A[])
 1  j ← i
 2  while j ≤ ⌊n/2⌋ do
 3      left ← LEFT(j)
 4      right ← RIGHT(j)
 5      if left ≤ n and A[left] > A[j]
 6          largest ← left
 7      else
 8          largest ← j
 9      if right ≤ n and A[right] > A[largest]
10          largest ← right
11      if largest ≠ j
12          swap(A[j], A[largest])
13          j ← largest
14      else
15          break
```

Lemma 16. *Under the assumption that* $A[1,\ldots,n]$ *and* i *are such that each of* $A[\![\textsc{Left}(i)]\!]$ *and* $A[\![\textsc{Right}(i)]\!]$*, if it exists, is a heap, the effect of algorithm* ITERATIVE HEAPIFY *is that* $A[\![i]\!]$ *is heap at its termination.*

Proof:
The following is a loop invariant for the **while** loop (lines 1–15):

> Every time the execution is at line 2, the only possible heap inversions in $A[\![i]\!]$ are $\langle j, \textsc{Left}(j)\rangle$ and $\langle j, \textsc{Right}(j)\rangle$, if $A[\textsc{Left}(j)]$ and $A[\textsc{Right}(j)]$ exist.

Basis. $j = i$. By the premises, each of $A[\![\textsc{Left}(i)]\!]$ and $A[\![\textsc{Right}(i)]\!]$, if it exists, is a heap, so the claim holds trivially.

Maintenance. Assume that the claim holds at some moment when the execution is at line 2 and execution will reach line 2 at least once more.

:: NB :: *The latter proviso implies line 15 is not reached during the current execution.*

So, it is the case that $j \le \lfloor \frac{n}{2} \rfloor$, and thus at least one of $\textsc{Left}(j) \le n$ and $\textsc{Right}(j) \le n$ is true, namely $\textsc{Left}(j) \le n$. So, at least $A[\textsc{Left}(j)]$ is defined. Without loss of generality, assume both $\textsc{Left}(j) \le n$ and $\textsc{Right}(j) \le n$ to avoid considering unnecessary subcases.

Case i: Both A[*left*] and A[*right*] are bigger than A[j] at the beginning and A[*right*] > A[*left*]. The evaluation at line 5 yields TRUE and the assignment at line 6 takes place. The evaluation at line 9 yields TRUE, too, so the assignment at line 10 takes place and when the execution is at line 11, *largest* equals *right*. The evaluation at line 11 yields TRUE and at line 12, A[j] and A[*largest*] get exchanged. We claim the only possible heap inversions in A⟦i⟧ after the exchange are ⟨*right*, LEFT(*right*)⟩ and ⟨*right*, RIGHT(*right*)⟩:

- no element in A⟦i⟧ outside A⟦j⟧ has been changed;
- currently A[j] > A[*left*] so ⟨j, LEFT(j)⟩ cannot be a heap inversion;
- currently A[j] > A[*right*] so ⟨j, RIGHT(j)⟩ cannot be a heap inversion;
- A⟦*left*⟧ has not been modified.
- none of A⟦LEFT(*right*)⟧ and A⟦RIGHT(*right*)⟧ has been modified.

But *right* is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.

Case ii: Both A[*left*] and A[*right*] are bigger than A[j] at the beginning and A[*right*] $\not>$ A[*left*]. The evaluation at line 5 yields TRUE and the assignment at line 6 takes place. The evaluation at line 9 yields FALSE, so the assignment at line 10 does not take place and when the execution is at line 11, *largest* equals *left*. The evaluation at line 11 yields TRUE and at line 12, A[j] and A[*largest*] get exchanged. We claim the only possible heap inversions in A⟦i⟧ after the exchange are ⟨*left*, LEFT(*left*)⟩ and ⟨*left*, RIGHT(*left*)⟩:

- no element in A⟦i⟧ outside A⟦j⟧ has been changed;
- currently A[j] > A[*left*] so ⟨j, LEFT(j)⟩ cannot be a heap inversion;
- currently A[j] ≥ A[*right*] so ⟨j, RIGHT(j)⟩ cannot be a heap inversion;
- A⟦*right*⟧ has not been modified.
- none of A⟦LEFT(*left*)⟧ and A⟦RIGHT(*left*)⟧ has been modified.

But *left* is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.

Case iii: A[*left*] $\not>$ A[j] and A[*right*] > A[j] at the beginning. The evaluation at line 5 yields FALSE and the assignment at line 8 takes place. The evaluation at line 9 yields TRUE, so the assignment at line 10 takes place and when the execution is at line 11, *largest* equals *right*. The evaluation at line 11 yields TRUE and at line 12, A[j] and A[*largest*] get exchanged. We claim the only possible heap inversions in A⟦i⟧ after the exchange are ⟨*right*, LEFT(*right*)⟩ and ⟨*right*, RIGHT(*right*)⟩. The proof is precisely the same as the proof of the analogous claim is **Case i**. But *largest* is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.

Case iv: A[*left*] > A[j] and A[*right*] $\not>$ A[j] at the beginning. The evaluation at line 5 yields TRUE and the assignment at line 6 takes place. The evaluation at line 9 yields FALSE, so the assignment at line 10 does not take place and when the execution is at line 11, *largest* equals *left*. The evaluation at line 11 yields TRUE and at line 12, A[j] and A[*largest*] get exchanged. We claim the only possible heap inversions in A⟦i⟧ after

the exchange are $\langle \mathit{left}, \textsc{Left}(\mathit{left})\rangle$ and $\langle \mathit{left}, \textsc{Right}(\mathit{left})\rangle$. The proof is precisely the same as the proof of the analogous claim is **Case ii**. But left is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.

Case v: $A[\mathit{left}] \not> A[j]$ and $A[\mathit{right}] \not> A[j]$ at the beginning. But that is impossible under the current assumptions, because clearly line 15 would be reached and the execution would never get to line 2 again.

Termination. Unlike the examples we have see so far, this loop can be exited in two ways: via line 2 when $j > \left\lfloor \frac{n}{2} \right\rfloor$ and via line 15. Let us consider the first possibility. Then both $\textsc{Left}(j)$ and $\textsc{Right}(j)$ point outside $A[1, \ldots, n]$. The invariant, therefore, says there are no heap inversions in $A[\![i]\!]$ at all since $A[\textsc{Left}(j)]$ and $A[\textsc{Right}(j)]$ do not those exist. And so $A[\![i]\!]$ is heap.

Consider the second possibility, *viz.* the **while** loop starts executing but the execution reaches line 15. It is obvious that, in order line 15 to be reached, it has to be the case that $\mathit{largest} = j$ at line 11. In order that to happen, both $A[\mathit{left}] \leq A[j]$ and $A[\mathit{right}] \leq A[j]$ must be true at the beginning of that execution since that is the only way that line 8 is reached and line 10 is not reached. But if $A[\mathit{left}] \leq A[j]$ and $A[\mathit{right}] \leq A[j]$, there are no heap inversions in $A[\![i]\!]$ at all, so $A[\![i]\!]$ is a heap. □

The function Heapify in the following algorithm is either Recursive Heapify on page 148 or Iterative Heapify on page 133.

```
Build Heap(A[1,2,...,n]: array of integers)
1  for i ← n downto 1
2      Heapify(A[], i)
```

Lemma 17. *When* Build Heap *terminates,* $A[]$ *is a heap.*

Proof:
The following is a loop invariant for the **while** loop (lines 1–2):

> Every time the execution is at line 1 and d is the depth of $A[i]$, then for every element $A[j]$ of depth $d-1$, $A[\![j]\!]$ is a heap.

We make use the of the fact—without proving is—that for every depth d, the elements of $A[]$ of depth d form a continuous subarray in $A[]$. If we call that subarray, *the* d*-block*, clearly those blocks appear in reverse sorted order:

- $A[1]$ is the $\lfloor \lg n \rfloor$-block[†].
- ...
- $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n\right]$ is the 0-block (the leaves).

So, as the index i starts from n and goes down to 1, the depths of the $A[i]$ elements take all values from $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ in ascending order. Furthermore, Lemma 35 on page 253 implies that the d-block is precisely $A\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1, \ldots, \left\lfloor \frac{n}{2^d} \right\rfloor\right]$.

[†]Recall that the height of any n element heap is $\lfloor \lg n \rfloor$ (see eq. (8.31) on page 250) and note that the height of the heap equals the depth of the root.

In order to make the proof work, assume that $A[\,]$ has an dummy element at position 0. The value of that $A[0]$ does not concern us. What is important is to consider the remainder of the array, namely $A[1,\ldots,n]$, as a subtree of $A[0]$. In other words, the depth of $A[0]$ is $\lfloor \lg n \rfloor + 1$. Without that technicality, the termination phase of the proof could not be formulated because i equals 0 when the execution is at line 2 for the last time.

Basis. Unlike the previous proofs, here the basis is not for a single value of the loop control variable but for a multitude of values. To see why, note that the induction is on the depth of the elements, not on the number of times line 2 is reached. It is natural to take for basis depth 0. But that means that the basis is over all leaves of $A[1,\ldots,n]$. The leaves are precisely the elements $A[i]$ such that $\left\lfloor \frac{n}{2} \right\rfloor + 1 \le i \le n$ (see Problem 153 on page 251).

Consider any i be such that $\left\lfloor \frac{n}{2} \right\rfloor + 1 \le i \le n$. Then $A[i]$ has depth 0. But elements of depth -1 do not exist. So the claim is vacuously true.

Maintenance. Relative to some number d such that $0 \le d \le \lfloor \lg n \rfloor$[†], consider the first time the execution is at line 2 and the depth of the current $A[i]$ is d. Since the depths of the $A[i]$ elements take all values from $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ in ascending order, such a moment will occur. As implied by Lemma 35 on page 253, at that moment $i = \left\lfloor \frac{n}{2^d} \right\rfloor$. Assume that the claim holds at that moment.

⁘ NB ⁘ *Our goal now is **not** to prove, using this assumption, that the claim holds the next time the execution is at line 2. Such an implication **does not exist**.*

Our goal is to prove that at the subsequent moment when the execution is at line 2 and for the first time, the depth of the current $A[i]$ is $d+1$, it is the case that for all indices j such that $A[j]$ is at depth d, $A[\![j]\!]$ is a heap. As implied by Lemma 35 on page 253, that subsequent moment is when $i = \left\lfloor \frac{n}{2^{d+1}} \right\rfloor$.

In other words, the maintenance phase consists of the following:

- assuming that

$$\underbrace{A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor + 1\right]\!\right], A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor + 2\right]\!\right], \ldots, A\left[\!\left[\left\lfloor \frac{n}{2^{d-1}} \right\rfloor\right]\!\right]}_{\text{the indices here are from the } d-1 \text{ block}}$$

  are heaps,

- show that

$$\underbrace{A\left[\!\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1\right]\!\right], A\left[\!\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 2\right]\!\right], \ldots, A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor\right]\!\right]}_{\text{the indices here are from the } d \text{ block}}$$

  are heaps.

It is rather obvious that the the map

$$j \to \{\textsc{Left}(j), \textsc{Right}(j)\}$$

maps the d-block on the $(d-1)$-block in the sense that the sets $\{\textsc{Left}(j), \textsc{Right}(j)\}$ are partitioning of the indices of the $(d-1)$-block, if j takes its values from the set of indices of the d-block. Of course, that holds if $d > 0$.

[†]In other words, d can be the depth of any vertex from $A[1,\ldots,n]$.

Apply Lemma 16 or Lemma 23, whichever one is applicable (depending on whether the recursive or the iterative HEAPIFY is used) on every one of

$$A\left[\left[\left\lfloor\frac{n}{2^{d+1}}\right\rfloor+1\right]\right], A\left[\left[\left\lfloor\frac{n}{2^{d+1}}\right\rfloor+2\right]\right], \ldots, A\left[\left[\left\lfloor\frac{n}{2^{d}}\right\rfloor\right]\right]$$

For each index

$$j \in \left\{\left\lfloor\frac{n}{2^{d+1}}\right\rfloor+1, \left\lfloor\frac{n}{2^{d+1}}\right\rfloor+2, \ldots, \left\lfloor\frac{n}{2^{d}}\right\rfloor\right\}$$

the premises of the Lemma include the assumption that $A[\![\textsc{Left}(j)]\!]$ an $A[\![\textsc{Right}(j)]\!]$ are heaps. But those two subtrees are from the set

$$\left\{A\left[\left[\left\lfloor\frac{n}{2^{d}}\right\rfloor+1\right]\right], A\left[\left[\left\lfloor\frac{n}{2^{d}}\right\rfloor+2\right]\right], \ldots, A\left[\left[\left\lfloor\frac{n}{2^{d-1}}\right\rfloor\right]\right]\right\}$$

and we did assume the elements of this set are heaps. So, the said Lemma provides the desired result.

Termination. When the execution is at line 2 for the last time, $i = 0$. We agreed that the dummy $A[0]$ is of depth $\lfloor \lg n \rfloor + 1$. Plug that value in the invariant to derive that every element of $A[\,]$ of depth $\lfloor \lg n \rfloor$, and the only such element is $A[1]$, it is the case that $A[\![1]\!]$ is a heap. □

The following pseudocode uses the notation "A.size". Assume that is a number such that $1 \leq A.\text{size} \leq n$, and HEAPIFY works using it as an upper index of the array–heap, **not** n as the pseudocode of HEAPIFY says.

```
HEAP SORT(A[1, 2, ..., n])
 1  BUILD HEAP(A[])
 2  A.size ← n
 3  for i ← n downto 2
 4      swap(A[1], A[i])
 5      A.size ← A.size − 1
 6      HEAPIFY(A[], 1)
```

Lemma 18. HEAP SORT *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1, \ldots, n]$. The following is a loop invariant for the **for** loop (lines 3–6):

> Every time the execution of HEAP SORT is at line 3, the current subarray $A[i+1, \ldots, n]$ consists of $n - i$ in number biggest elements of $A'[1, \ldots, n]$. Furthermore, the current $A[1, \ldots, i]$ is a heap.

Basis. The first time the execution reaches line 3, it is the case that $i = n$. The current subarray $A[i+1, \ldots, n]$ is empty and thus, vacuously, it consists of zero in number biggest elements from $A'[1, \ldots, n]$, in sorted order. $A[1, \ldots, n]$ is a heap by Lemma 18, applied to line 1.

Maintenance. Assume the claim holds at a certain execution of line 3 and the **for** loop is to be executed at least once more. Let us call the array $A[\,]$ at that moment, $A''[\,]$. By

the maintenance hypothesis, $A''[i+1,\ldots,n]$ are $n-i$ in number maximum elements of $A'[]$, in sorted order. By the maintenance hypothesis again, $A''[1]$ is a maximum element of $A''[1,\ldots,i]$. After the swap at line 4, $A''[i,\ldots,n]$ are $n-i$ in number maximum elements of $A'[]$, in sorted order. Relative to the new value of i the next time the execution is at line 3, the first sentence of the invariant holds.

The second sentence holds, too, by applying Lemma 16 or Lemma 23, whichever one is applicable (depending on whether the recursive or the iterative HEAPIFY is used), at line 6. Just keep in mind that HEAPIFY considers the heap to be $A[1,\ldots,i-1]$ because i equals A.size when the execution is at line 6; note that because of line 5, A.size is $i-1$ at line 6. Thus at line 6, the current $A[i]$ is outside the scope of the current heap.

Termination. Consider the moment when the execution is at line 3 for the last time. Clearly, i equals 1. Plug the value 1 in place of i in the invariant to obtain "the current subarray $A[2,\ldots,n]$ consists of $n-1$ in number biggest elements of $A'[1,\ldots,n]$.". But then $A[1]$ has to be a minimum element from $A'[1,\ldots,n]$. And that concludes the proof of the correctness of HEAP SORT. □

4.3.4 The correctness of Dijkstra's algorithm

We assume the reader is familiar with the terminology concerning weighted digraphs. When we talk about path lengths or distances, we mean weighted lengths or distances. Note the distance is not necessarily symmetric in digraphs. Let the weight function be $w : E \to \mathbb{R}^+$. The proof of correctness of the algorithm below is a detailed version of the proof in [Man05]. If $G(V,E)$ is a graph, for any $u \in V$ we denote the set $\{v \in V \mid (u,v) \in E\}$ by "adj(u)", and for any $u,v \in V$ we denote the distance from u to v in G by "$\text{dist}_G(u,v)$". The subscript G in that notation is useful when u and v are vertices in more than one graph under consideration and we want to emphasise we mean the distance in that particular graph. We postulate that $\text{dist}_G(u,v) = \infty$ iff there is no path from u to v in G.

```
DIJKSTRA(G(V,E): graph; w: weight function; s: vertex from V)
 1  (* U is a variable of type vertex set *)
 2  foreach u ∈ V
 3      dist[u] ← ∞
 4      π[u] ← 0
 5  U ← {s}
 6  dist[s] ← 0
 7  foreach x ∈ adj(s)
 8      dist[x] ← w((s,x))
 9      π[x] ← s
10  while ({v ∈ V \ U | dist[v] < ∞} ≠ ∅) do
11      select any x ∈ {v ∈ V \ U | dist[v] < ∞} such that dist[x] is minimum
12      U ← U ∪ {x}
13      foreach y ∈ adj(x)
14          if dist[y] > dist[x] + w((x,y))
15              dist[y] ← dist[x] + w((x,y))
16              π[y] ← x
```

It is obvious that Dijkstra's algorithm terminates because at each iteration of the **while**

loop (lines 10–16) precisely one vertex is added to U and since V is finite, inevitably the set $\{v \in V \setminus U \mid \text{dist}[v] < \infty\}$ will become $\emptyset$. Now we prove Dijkstra's algorithm computes correctly the shortest paths in G from s to all vertices.

Lemma 19. *At the termination of* DIJKSTRA*, it is the case that*

- $\forall u \in V$: *the value* $dist[u]$ *equals* $dist_G(s, u)$*, and*
- *the array* $\pi[\,]$ *represents a shortest-paths tree in* G*, rooted at* s*.*

Proof:
Let us first make several definitions. *A* u*-path* for any $u \in V$ is a any path from s to u. During the execution of DIJKSTRA, relative to the current value of U, for any $z \in V \setminus U$, *a* z*-special path in* G is any path $p = s, u, \ldots, w, z$, such that $|p| > 0$ and precisely one vertex in p, namely z, is not from U. *A special path* is any path that is a z-special path for some $z \in V \setminus U$. Relative to the current value of U, *the fringe* $F(U)$ is the set $\{z \in V \setminus U \mid \text{there exists a } z\text{-special path}\}$.
The following is a loop invariant for the **while** loop (lines 10–16):

> Every time the execution of DIJKSTRA is at line 10 the following conjunction holds:
>
> **part i:** $\forall u \in U : \text{dist}[u] = \text{dist}_G(s, u)$, and
>
> **part ii:** $\forall u \in U \setminus \{s\} : \pi[u]$ is the neighbour of u is some shortest u-path and $\pi[u] \in U$, and
>
> **part iii:** $\forall u \in V \setminus U : \text{dist}[u] < \infty$ iff $u \in F(U)$, and
>
> **part iv:** $\forall u \in F(U) : \text{dist}[u]$ is the length of a shortest u-special path and $\pi[u]$ is the neighbour of u in such a path.

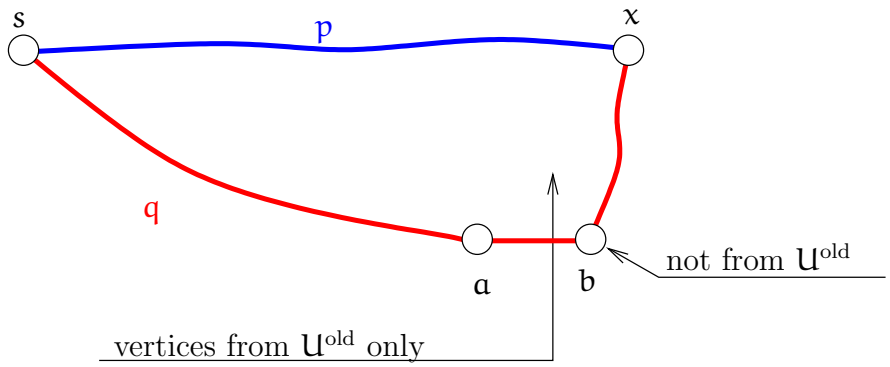
Basis. The first time the execution reaches line 10, it is the case that $U = \{s\}$ because of the assignment at line 5. **part i** holds because $\text{dist}[s] = 0$ (line 6) and $\text{dist}_G(s, s) = 0$ (by definition). **part ii** holds vacuously since $U \setminus \{s\} = \emptyset$. **part iii** holds because on the one hand $F(U) = \text{adj}(s)$ and on the other hand the assignments at lines 3 and 8 imply $\text{adj}(s)$ are the only vertices with $\text{dist}[\,] < \infty$. **part iv** holds because $\forall u \in F(U)$ the only u-special path is the edge (s, u); at lines 8 and line 9, $\text{dist}[u]$ and $\pi[u]$ are set accordingly.

Maintenance. Assume the claim holds at a certain execution of line 10 and the **while** loop is to be executed at least once more. Let us call the set U at that moment, U^{old} and after the assignment at line 12, U^{new}. So, $\{x\} = U^{\text{new}} \setminus U^{\text{old}}$. We first prove **part i** and **part ii**. We do that by considering U^{old} and x separately. We claim that for all vertices in U^{old}, their $\text{dist}[\,]$ and $\pi[\,]$ values do not change during the current iteration of the **while** loop. But that follows trivially from the fact that by **part i** of the inductive hypothesis their $\text{dist}[\,]$ values are optimal and the fact that, if the **while** loop changes the $\text{dist}[\,]$ and $\pi[\,]$ values of any vertex, that implies decreasing its $\text{dist}[\,]$ value.

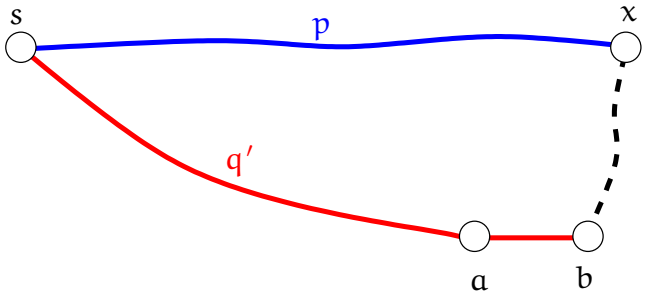
Consider vertex x. Before the assignment at line 12, by **part iii** of the inductive hypothesis x is a fringe vertex and so by **part iv**, its $\text{dist}[\,]$ value is the length of a shortest x-special—with respect to U^{old}—path p and $\pi[x]$ is the neighbour of u in p. Now we prove that p is a shortest x-path in G.

Assume the contrary. Then there exists an x-path q that is shorter than p. Since one endpoint of q, namely s, is from U^{old}, and the other endpoint x is not from U^{old}, there is

at least one pair of neighbour vertices in q such that one is from U^{old} and the other one is not from U^{old}. Among all such pairs, consider the pair a, b that is closest to s in the sense that between s and a inclusive there are only vertices from U^{old} and b is the first vertex (in direction away from s) not from U^{old}. Note that $b \neq x$, for if b were x then q would be an x-special—with respect to U^{old}—path shorter than p, and by **part iv** that is not possible. Let the subpath of q between s and b be called q'. Note that $|q'| < |q|$. By assumption, $|q| < |p|$, therefore $|q'| < |p|$. Then note q' is a special path with respect to U^{old} and $b \in F(U^{\text{old}})$. By **part iv**, dist[b] is at most $|q'|$ at the beginning of the current iteration of the **while**-loop, thus dist[b] < dist[x] and DIJKSTRA would have selected b rather than x at line 11. This contradiction refutes the assumption there exists any x-path shorter than p. So, dist[x] indeed equals $\text{dist}_G(s, x)$, therefore **part i** and **part ii** hold the next time the execution reaches line 10. The following two figures illustrate the contradiction we just derived. Initially we assumed the existence of a path q shorter than p and defined its rightmost neighbour pair a, b such that b is the vertex closest to s and not from U^{old}:



Then we concluded the subpath q' between s and b must be special with respect to U^{old} and, furthermore, shorter than p:



Immediately we concluded the algorithm whould have picked b rather than x.

It remains to prove that **part iii** and **part iv** hold after the current iteration. Have in mind that x was a fringe vertex at the beginning of the current iteration of the **while** loop but at the end of it x is in U, we exclude x from consideration. The proof of **part iii** is straightforward. As just said, x is no longer in $V \setminus U$.

- In one direction, partition the remaining vertices of $V \setminus U$ into those whose dist[] value was $< \infty$ at the beginning of the current iteration and those whose dist[] value was equal to ∞ at the at the beginning of the current iteration. By **part iii** of the

inductive hypothesis, the former set were neighbours of vertices from U^{old}, so at the end of the iteration they are neighbours of vertices from U^{new} which makes them fringe vertices. On the other hand, the latter set are neighbours to x, which makes them fringe vertices with respect to U^{new}.

- In the other direction, consider the vertices from $V \setminus U$ whose dist[] value is equal to ∞ at the end of the current iteration. They can neither be neighbours to vertices from U^{old}, otherwise they would have dist[] value $< \infty$ at the beginning of the current iteration of the **while**-loop, nor can they be neighbours of x, otherwise their dist[] values would be set to some positive reals by **for**-loop at lines 13–16. Therefore, they are not fringe vertices at the end of the current iteration of the **while**-loop.

To prove **part iv**, partition $F(U^{\text{new}})$ into $F(U^{\text{old}}) \setminus \{x\}$ and $F(U^{\text{new}}) \setminus (F(U^{\text{old}}) \setminus \{x\})$—the vertices added during the current iteration of the **while**-loop. It is obvious that for every vertex u from $F(U^{\text{new}}) \setminus (F(U^{\text{old}}) \setminus \{x\})$ its only neighbour from U^{new} is x—otherwise, dist[u] would not be ∞ at the beginning of the current iteration of the **while**-loop. Then every shortest u-special path p is such that the path neighbour of u is x and so $|p|$ equals $\text{dist}_G(s, x) + w((x, u))$. We already showed that $\text{dist}_G(s, x)$ equals dist[x] during the current iteration of the **while**-loop. We note that at line 16, dist[u] is assigned precisely dist[x] + $w((x, u))$. Clearly, **part iv** holds for u.

Now consider any vertex u in $F(U^{\text{old}}) \setminus \{x\}$. If dist[$u$] is not changed, in other words decreased, during the current iteration of the **while**-loop, **part iv** holds by the induction hypothesis regardless of whether u is or is not a neighbour of x. Suppose dist[u] is decreased during the current iteration of the **while**-loop. u must be a neighbour of x because the only place dist[u] can be altered is at line 15 that is executed within the **for**-loop (lines 13–16). However, u is a neighbour of at least one vertex from U^{old}, otherwise u would not be a vertex from $F(U^{\text{old}})$. By **part iv** of the induction hypothesis, at the beginning of the current iteration of the **while**-loop, it is the case that dist[u] = $|p|$, where p is a shortest u-special path with respect to U^{old}. The fact that dist[u] was altered at line 15 means that

$$\text{dist}[x] + w((x, u)) < |p|$$

It follows there is a u-special path q with respect to U^{new} such that the path neighbour of u in q is x, and $|q| = \text{dist}[x] + w((x, u))$. Furthermore, $|q|$ is the minimum length of any u-special path q with respect to U^{new} such that the path neighbour of u in q is x because dist[x] = $\text{dist}_G(s, x)$ as we already proved. There cannot be a shorter than q, u-special path with respect to U^{new}—assuming the opposite leads to a contradiction because the path neighbour of u in that alleged path cannot be x and cannot be any other vertex from U^{new}. That concludes the proof of **part iv**.

Termination. Consider the moment when the execution is at line 10 for the last time. It is either the case that $U = V$, or $U \subset V$ but all vertices in $V \setminus U$ have dist[] values equal to ∞. In the former case, the claim of this lemma follows directly from **part i** and **part ii** of the invariant. In the latter case, it is easy to see that every vertex w such that dist[w] = ∞ is such that no path exists from s to w—assuming the opposite leads to a contradiction because that alleged path must have neighbour vertices a and b, such that dist[a] $< \infty$, dist[b] = ∞, and $(a, b) \in E(G)$; clearly, b would have gotten finite dist[] value as a neighbour of a during the iteration of the **while**-loop when the value of the x variable at line 11 was a. It follows that DIJKSTRA assigns ∞ precisely to the dist[] of those vertices that are not reachable from s, and all other ones are dealt with correctly. □

4.3.5 The correctness of Counting Sort

The input is an array of positive integers $A[1, 2, \ldots, n]$. It is known there is some $k \in \mathbb{N}^+$ such that $1 \leq A[i] \leq k$ for all i, $1 \leq i \leq n$. We are going to sort $A[\,]$ in time $\Theta(n + k)$ by *counting* the number of appearances of each element. That sorting algorithm is not comparison-based and thus the $\Omega(n \lg n)$ lower bound does not hold for it. The algorithm uses an auxiliary array $C[0, 1, \ldots, k]$ as a workpad and another array $B[1, 2, \ldots, n]$ for writing the output.

```
Counting Sort(A[1,2,...,n]: positive integers; k: a positive integer)
 1  (* k is such that 1 ≤ A[i] ≤ k for all i *)
 2  for i ← 0 to k
 3      C[i] ← 0
 4  for i ← 1 to n
 5      C[A[i]] ← C[A[i]] + 1
 6  for i ← 1 to k
 7      C[i] ← C[i] + C[i − 1]
 8  for i ← n downto 1
 9      B[C[A[i]]] ← A[i]
10      C[A[i]] ← C[A[i]] − 1
```

Obviously, the first **for**-loop (lines 2–3) initialises $C[\,]$ with zeroes. Let us introduce the following notation: if $Z[1, \ldots, m]$ is an array, j is an index such that $1 \leq j \leq m$, and x is an element that may occur in $Z[\,]$, then $\#(x, j, Z)$ denotes the number of occurrences of x in the subarray in the subarray $Z[1, \ldots, j]$.

Lemma 20. *After the second **for**-loop (lines 4–5) terminates, for $1 \leq j \leq k$, element $C[j]$ contains the number of occurrences of j in $A[\,]$.*

Proof:
The following is a loop invariant for the second **for**-loop:

> Every time the execution is at line 4, for every element $C[j]$ where $1 \leq j \leq k$, it is the case that $C[j] = \#(j, i - 1, A)$.

Basis. The first time the execution reaches line 4, all elements of $C[\,]$ are zeroes. On the other hand, i is 1, and thus $A[1, \ldots, i - 1]$ is empty. The claim is vacuously true.

Maintenance. Assume the claim holds at a certain execution of line 4 and the second **for**-loop is to be executed at least once more. Let the value of $C[A[i]]$ be y at the moment before the execution of line 5. By the inductive hypothesis, is $y = \#(A[i], i - 1, A)$. An obvious mathematical fact is that $\#(A[i], i - 1, A) + 1 = \#(A[i], i, A)$. Indeed, after the execution of line 5, $C[A[i]]$ gets incremented by one, so now $C[A[i]]$ equals $\#(A[i], i, A)$. Since the other elements of $C[\,]$ (apart from $C[A[i]]$) are not affected by the current execution of the **for**-loop, it is the case that:

- for every element $C[j]$ apart from $C[A[i]]$, $C[j] = \#(j, i-1, A)$, and also $C[j] = \#(j, i, A)$
- $C[A[i]] = \#(A[i], i, A)$.

Overall, for every element C[j], $C[j] = \#(j, i, A)$. That is before i gets incremented. After i gets incremented, $C[j] = \#(j, i-1, A)$ for every j such that $1 \leq j \leq k$.

Termination. Consider the moment when the execution is at line 4 for the last time. Clearly, i equals $n+1$. Plug the value $n+1$ in place of i in the invariant to obtain "for every element C[j] where $1 \leq j \leq k$, it is the case that $C[j] = \#(j, n, A)$." □

Lemma 21. *After the third* ***for****-loop (lines 6–7) terminates, for* $1 \leq j \leq k$*, element* C[j] *contains the number of elements of* A[] *that are* $\leq j$.

Proof:
The following is a loop invariant for the third **for**-loop:

> Every time the execution is at line 6, for every j such that $0 \leq j \leq i-1$, $C[j] = \sum_{t=1}^{j} \#(t, n, A)$.

Basis. The first time the execution reaches line 6, i equals 1, so the claim is that $C[0] = \sum_{t=1}^{0} \#(t, n, A) = 0$. But C[0] is indeed 0 since it is initialised to 0 by the first **for**-loop and the second **for**-loop does not assign anything to it as A[i] cannot be 0.✓

Maintenance. Assume the claim holds at a certain execution of line 6 and the third **for**-loop is to be executed at least once more. The element C[i] has not been affected by the execution of the **for**-loop so far, therefore by Lemma 20, $C[i] = \#(i, n, A)$. By the inductive hypothesis, $C[i-1] = \sum_{t=1}^{i-1} \#(t, n, A)$. After the execution of line 7, it is the case that $C[i] = \sum_{t=1}^{i} \#(t, n, A)$. With respect to the new value of i, for every j such that $0 \leq j \leq i-1$, $C[j] = \sum_{t=1}^{j} \#(t, n, A)$.

Termination. Consider the moment when the execution is at line 6 for the last time. Clearly, i equals $n+1$. Plug the value $n+1$ in place of i in the invariant to obtain "for every j such that $0 \leq j \leq n$, $C[j] = \sum_{t=1}^{j} \#(t, n, A)$." □

For every integer j such that $1 \leq j \leq k$, let us call j *essential* if there is at least one element of A[] with value j. By Lemma 20, the non-zero elements of C[] after the second **for**-loop are precisely the elements whose indices are essential. For every element x of A[] we define the concept *the proper place of* x. The proper place of x is the total number of elements of A[] smaller than x, plus the number of elements equal to x that are left of x, plus one. In other words, the proper place of x is its index in the array after a stable sorting algorithm has been executed on the array.

Lemma 22. Counting Sort *is a stable sorting algorithm.*

Proof:
The following is a loop invariant for the fourth **for**-loop (lines 8–10):

> Every time the execution is at line 8, for every j such that $1 \leq j \leq k$ and j is essential, C[j] is the proper place for the rightmost element from the subarray $A[1, \ldots, i]$ that has value j. Furthermore, all elements from $A[i+1, \ldots, n]$ are at their proper places in B[].

Basis. The first time the execution reaches line 6, i equals n. The first part of the invariant is, "for every j such that $1 \leq j \leq k$ and j is essential, C[j] is the proper place for the rightmost element from the subarray $A[1, \ldots, n]$ that has value j". However, C[] has

not been modified yet by the fourth loop, therefore Lemma 21 holds. For every distinct value j that occurs in $A[]$, the proper place of the rightmost j in $A[]$ is at position that equals the sum of the numbers of all elements with value $\leq j$. According to Lemma 21, $C[j]$ equals precisely that sum.

Consider the second part of the invariant. The subarray $A[i+1,\ldots,n] = A[n+1,\ldots,n]$ is empty, therefore the claim holds vacuously.

Maintenance. Assume the claim holds at a certain execution of line 8 and the fourth **for**-loop is to be executed at least once more. The value of $A[i]$ is some essential integer j between 1 and k. Moreover, it is **the rightmost** element in $A[1,\ldots,i]$ with value j. By the induction hypothesis, its proper place is $C[A[i]]$ and that is where the algorithm copies it (line 9).

Assume there are other elements in $A[1,\ldots,i]$ with value j. At line 10, $C[A[i]]$ gets decremented, so now it contains the proper position of the rightmost j in $A[1,\ldots,i-1]$. After i gets decremented, it is the case the element of $C[]$ that just got modified (decremented) contains the proper position of the rightmost j in $A[1,\ldots,i]$. Since all the other elements of $C[]$ are unmodified, the first part of the invariant holds.

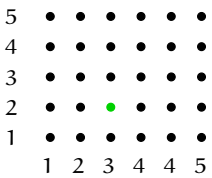
Now assume there are no other elements in $A[1,\ldots,i]$ with value j. Then element j is not essential with respect to the remainder of $A[]$ that is still to be scanned, so the value of $C[A[i]]$ is inconsequential for the remainder of the algorithm. Indeed, after the decrement at line 10, $C[A[i]]$ points to a location in B where another element, not j, belongs. But, as we said, the value of that $C[A[i]]$ will never be used again for that value of $A[i]$ will occur no more. The first part of the invariant holds in this case, too.

Now we prove the second part of the invariant is preserved as well. Consider the moment the execution is at line 8 at the beginning of the current iteration. By the induction hypothesis, all elements from $A[i+1,\ldots,n]$ are at their proper places in $B[]$. We just proved that during the execution of the loop, element $A[i]$ is put in its proper place. So all elements from $A[i,\ldots,n]$ are at their proper places in $B[]$. With respect to the new value of i, it is the case that all elements from $A[i+1,\ldots,n]$ are at their proper places in $B[]$.

Termination. When the loop terminates, $i = 0$. Plug in that value in the second part of the invariant to obtain "all elements from $A[1,\ldots,n]$ are at their proper places in $B[]$." □

4.3.6 Correctness proofs of miscellaneous algorithms

Problem 110. *Two players, call them Red and Blue, are given a rectangular grid of m rows and n dots, for instance for $m = n = 5$:*



Every dot is refered to as (i,j) where $1 \leq i \leq m$ and $1 \leq i \leq n$, i being the row and j, the column. For instance, the green dot is $(2,3)$. Red and Blue take alternating turns, starting with Red. In his or her turn, each player puts either a horizontal or a vertical line segment with unit length, connecting two adjacent grid dots that have not been previously connected. Red's segments have red colour and Blue's segments have blue colour. Red wins iff he or

she creates a closed curve (of red segments only). Otherwise, Blue wins. Since the number of unused adjacent pairs decreases with every turn, the game terminates. Decide whether one player, Red or Blue, has a winning strategy. If yes, what is it?

Solution:
Blue has a winning strategy. Suppose Red has just placed a segment. That segment can either be horizontal or vertical.

BLUE'S ANSWER($m \times n$ dot grid)
```
1  Red places a segment between some pair of adjacent dots
2  while there are unused pairs of adjacent dots do
3      if Red has placed a horizontal segment
4          let Red's last segment be ((i,j),(i,j+1))
5          if i > 1  and ((i-1,j+1),(i,j+1)) is unused
6              Blue places the segment ((i-1,j+1),(i,j+1))
7          else
8              Blue places a segment between an arbitrary unused pair of adj. dots
9      else
10         let Red's last segment be ((i,j),(i+1,j))
11         if j > 1  and ((i+1,j-1),(i+1,j)) is unused
12             Blue places the segment ((i+1,j-1),(i+1,j))
13         else
14             Blue places a segment between an arbitrary unused pair of adj. dots
15     Red places a segment between some unused pair of adjacent dots
16     if a closed red-segment curve is formed
17         Red wins, game over
18 Blue wins, game over
```

Informally speaking, Blue's strategy is to prevent an upper right corner consisting of two red segments. Red cannot make a red closed curve unless he or she can make a two-segment red upper right corner.

Let us define that during the execution of BLUE'S ANSWER, any red segment is *free* iff

- it is horizontal, say $((i,j),(i,j+1))$, $i > 1$, and $((i-1,j+1),(i,j+1))$ is unused, or
- it is vertical, say $((i,j),(i+1,j))$, $j > 1$, and $((i+1,j-1),(i+1,j))$ is unused.

For example, the following two figures illustrate a free vertical segment (left) and a free horizontal segment (right).

The following is a loop invariant for the **while** loop (line 2–17):

> Every time the execution of BLUE'S ANSWER is at line 2, there is at most one free red segment. Furthermore, it is placed by the most recent addition of red segment.

Basis. The first time the execution reaches line 2, Red has just placed a red segment, either a horizontal or a vertical one. If it is $((i,j),(i,j+1))$, if $i=1$ it is not free, otherwise it is free. If it is $((i,j),(i+1,j))$, if $j=1$ it is not free, otherwise it is free. The invariant holds in any event.

Maintenance. Assume the claim holds at a certain execution of line 2 and the **while** loop is to be executed at least once more. Note that the placement of the blue segment at line 12 or line 14, whichever is applicable, is such that if there was a free segment at the beginning of the current execution of the **while** loop, there is no free segment after the blue segment is placed. Then the adding of another red segment at line 15 may or may not lead to a free segment. The invariant holds.

Based on this invariant, it is trivial to prove that during any execution of BLUE'S ANSWER, an upper right corner of two red segments is never made. Such a corner can only be made after the addition of a red segment at line 15. However, the addition of a single red segment can lead to such a corner only if there has been a free segment before. By the invariant, at the beginning of any execution of the **while** loop there is at most one free segment; even if there is one, a blue segment is placed in such a way that there is no free segment at line 15. It follows that an upper right corner of two red segments is never made, and so line 17 is unreachable. It follows that the execution always reaches line 17 and Blue wins. □

Problem 111. *A deck of* 52 *cards are piled on a stack. Two players,* A *and* B*, play in turns by taking one or two cards from the top of the stack. Once taken away, a card is never returned to the stack. The winner is the one taking a card or two cards last.* A *plays first. Does* A *have a winning strategy? If yes, how does that depend on the number of cards.*

Solution:
Let us think backwards. Suppose it is A's turn. If the cards in the stack are one or two, A clearly wins.

If the cards are three, A loses (assuming B plays optimally) since after his or her turn, the number of cards left is one or two, and B wins as A in the former case.

If the cards are four or five, A takes one or two cards away, respectively, and places B in the losing situation with three cards. If the cards are six, A loses. And so on.

Intuitively, it is clear that A wins whenever the initial number of cards is not a multiple of three. Here is a precise argument in support of that claim. Let A execute the following algorithm. Assume that the call BPLAYS removes either one or two cards from the stack and outputs an appropriate message.

```
REMOVE LAST CARD(n: pos. integer such that n mod 3 ≠ 0)
1  while n > 3 do
2      k ← n mod 3
3      print "A takes " k " cards."
4      n ← n − k
5      BPLAYS(n)
6  (* n = 1 or n = 2 *)
7  print "A takes " n " cards and wins!"
```

The following is a loop invariant for the **while** loop (line 2–17):

Every time the execution of REMOVE LAST CARD is at line 1, $n \bmod 3 \neq 0$.

Basis. The first time the execution is at line 1, the claim is true by the definition of the algorithm.

Maintenance. Assume the claim holds at a certain execution of line 1 and the **while** loop is to be executed at least once more. After the assignments at lines 2 and 4, it is the case that $n \bmod 3 = 0$ but still $n \geq 3$. Then BPLAYS causes the decrement of n by one or two, thus $n \bmod 3 \neq 0$ the next time the execution of REMOVE LAST CARD is at line 1.
Termination. The last time the execution of REMOVE LAST CARD is at line 1, it must be the case that $n \leq 3$. By the invariant, n cannot be 3, so $n \in \{1, 2\}$. Indeed, A wins immediately. □

4.4 Proving algorithm correctness by induction

The correctness of recursive algorithms is typically proven by induction. Although the proofs with loop invariants are essentially proofs by induction they differ from the proofs that use "pure" induction as shown in the following examples.

4.4.1 Correctness proofs of miscellaneous algorithms

```
STRANGE INCREMENT(n: natural number)
1  if n = 0
2      return 1
3  else
4      if n mod 2 = 0
5          return n + 1
6      else
7          return 2 × STRANGE INCREMENT(⌊n/2⌋)
```

Problem 112. *Prove that* STRANGE INCREMENT *returns* $n + 1$ *for any natural* n.

Solution:
The algorithm is recursive so the proof is by induction on n.

Basis. $n = 0$. The condition at line 1 is TRUE and the algorithm returns $0 + 1 = 1$ via the assignment at line 2. ✓

Inductive Hypothesis. Assume $\forall m < n$, the algorithm returns $m + 1$.

Inductive Step. Consider the work of STRANGE INCREMENT on input n. First suppose n is even. The condition at line 4 is TRUE, therefore line 5 is executed and the algorithm returns $n + 1$. ✓

Now suppose n is odd. The condition at line 4 is FALSE, therefore line 7 is executed and the algorithm returns $2 \times$ STRANGE INCREMENT $\left(\lfloor \frac{n}{2} \rfloor\right)$. As $\lfloor \frac{n}{2} \rfloor$, by the inductive

hypothesis it is the case that STRANGE INCREMENT $\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ returns $\left\lfloor \frac{n}{2} \right\rfloor + 1$. Since n is odd, $n = 2k + 1$ for some $k \in \mathbb{N}$. The returned value is

$$2 \times \left(\left\lfloor \frac{2k+1}{2} \right\rfloor + 1 \right) = 2 \times \left(\left\lfloor k + \frac{1}{2} \right\rfloor + 1 \right) = 2 \times (k+1) = 2k + 2 = 2n + 1$$

□

4.4.2 The correctness of algorithms on binary heaps

```
RECURSIVE HEAPIFY(A[1,2,...,n]: array of integers, i: index in A[])
 1   left ← LEFT(i)
 2  right ← RIGHT(i)
 3  if left ≤ n  and A[left] > A[i]
 4      largest ← left
 5  else
 6      largest ← i
 7  if right ≤ n  and A[right] > A[largest]
 8      largest ← right
 9  if largest ≠ i
10      swap(A[i], A[largest])
11      RECURSIVE HEAPIFY(A[], largest)
```

Lemma 23. *Under the assumption that* $A[1, \ldots, n]$ *and* i *are such that each of* $A[\![\text{LEFT}(i)]\!]$ *and* $A[\![\text{RIGHT}(i)]\!]$*, if it exists, is a heap, the effect of algorithm* RECURSIVE HEAPIFY *is that* $A[\![i]\!]$ *is heap.*

Proof:
By induction on the height h of $A[\![i]\!]$.

Basis. $h = 0$. That means that $A[\![i]\!]$ consists of a single element from $A[]$. So, both LEFT(i) and RIGHT(i) point outside $A[]$. Let us follow the execution of RECURSIVE HEAPIFY: the condition at line 3 is false and so the assignment at line 6 takes place. The condition at line 7 is false, too, so line 8 is not executed and the execution goes to line 9. The evaluation there yields FALSE and so the current recursive call terminates. Clearly, $A[\![i]\!]$ is a heap when that recursive call terminates.

Inductive Hypothesis. Assume that for every $A[\![j]\!]$ of height $\leq h - 1$ rooted at some j such that $A[j]$ belongs to $A[\![i]\!]$, it is the case that RECURSIVE HEAPIFY$(A[], j)$ constructs a heap out of $A[\![j]\!]$.

Inductive Step. Consider the execution of RECURSIVE HEAPIFY$(A[], i)$. Without loss of generality, assume that LEFT$(i) \leq n$ and RIGHT$(i) \leq n$, so lines 3 and 7 are

if $A[left] > A[i]$

and

if $A[right] > A[largest]$

respectively.
Case i: Both A[$left$] and A[$right$] are bigger than A[i] at the beginning and A[$right$] > A[$left$]. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place. The evaluation at line 7 yields TRUE, too, so the assignment at line 8 takes place and when the execution is at line 9, $largest$ equals $right$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $right$. By the inductive hypothesis, that recursive call constructs a heap out of A⟦$right$⟧. On the other hand, A⟦$left$⟧ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$right$], is

- bigger than the current A[$left$] because of the premises
- not smaller than the current A[$right$] for the following reasons. At the beginning, A⟦$right$⟧ was a heap so the former A[$right$] was not smaller than any other element of A⟦$right$⟧ then (at the beginning). Now, at the end, the elements of A⟦$right$⟧ consist of the initial elements minus the initial A[$right$] plus the initial A[i]. Since the initial A[$right$] is bigger than the initial A[i] and not smaller that any other element of A⟦$right$⟧, the current A[i] is not smaller than the current A[$right$].

Therefore, by definition the current A⟦i⟧ is a heap.
Case ii: Both A[$left$] and A[$right$] are bigger than A[i] at the beginning and A[$right$] $\not>$ A[$left$]. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place. The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $largest$ equals $left$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $left$. By the inductive hypothesis, that recursive call constructs a heap out of A⟦$left$⟧. On the other hand, A⟦$right$⟧ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$left$], is

- not smaller than the current A[$right$] because of the premises
- not smaller than the current A[$left$] for the following reasons. At the beginning, A⟦$left$⟧ was a heap so the former A[$left$] was not smaller than any other element of A⟦$left$⟧ then (at the beginning). Now, at the end, the elements of A⟦$left$⟧ consist of the initial elements minus the initial A[$left$] plus the initial A[i]. Since the initial A[$left$] is bigger than the initial A[i] and not smaller that any other element of A⟦$left$⟧, the current A[i] is not smaller than the current A[$left$].

Therefore, by definition the current A⟦i⟧ is a heap.
Case iii: A[$left$] $\not>$ A[i] and A[$right$] > A[i]. The evaluation at line 3 yields FALSE and the assignment at line 6 takes place. The evaluation at line 7 yields TRUE, so the assignment at line 8 takes place and when the execution is at line 9, $largest$ equals $right$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $right$. By the inductive hypothesis, that recursive call constructs a heap out of A⟦$right$⟧. On the other hand, A⟦$left$⟧ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$right$], is

- bigger than the current A[$left$] because of the premises and the transitivity of the inequalities

- not smaller than the current A[$right$] for the following reasons. At the beginning, A⟦$right$⟧ was a heap so the former A[$right$] was not smaller than any other element of A⟦$right$⟧ then (at the beginning). Now, at the end, the elements of A⟦$right$⟧ consist of the initial elements minus the initial A[$right$] plus the initial A[i]. Since the initial A[$right$] is bigger than the initial A[i] and not smaller that any other element of A⟦$right$⟧, the current A[i] is not smaller than the current A[$right$].

Therefore, by definition the current A⟦i⟧ is a heap.
Case iv: A[$left$] > A[i] and A[$right$] $\not>$ A[i]. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place. The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $largest$ equals $left$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $left$. By the inductive hypothesis, that recursive call constructs a heap out of A⟦$left$⟧. On the other hand, A⟦$right$⟧ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$left$], is

- bigger than the current A[$right$] because of the premises and the transitivity of the inequalities

- not smaller than the current A[$left$] for the following reasons. At the beginning, A⟦$left$⟧ was a heap so the former A[$left$] was not smaller than any other element of A⟦$left$⟧ then (at the beginning). Now, at the end, the elements of A⟦$left$⟧ consist of the initial elements minus the initial A[$left$] plus the initial A[i]. Since the initial A[$left$] is bigger than the initial A[i] and not smaller that any other element of A⟦$left$⟧, the current A[i] is not smaller than the current A[$left$].

Therefore, by definition the current A⟦i⟧ is a heap.
Case v: A[$left$] $\not>$ A[i] and A[$right$] $\not>$ A[i]. The evaluation at line 3 yields FALSE and the assignment at line 6 takes place. The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $largest$ equals i. The evaluation at line 9 yields FALSE and the execution terminates, leaving A⟦i⟧ untouched. By the premises, A⟦$left$⟧ and A⟦$right$⟧ are heaps and A[$left$] > A[i] and A[$right$] $\not>$ A[i], so the current A⟦i⟧ is a heap by definition. □

Chapter 5

Lower Bounds for Computational Problems

5.1 Comparison-based sorting

We prove an $\Omega(n \lg n)$ bound for the time complexity of any *comparison-based* sorting algorithm. A sorting algorithm on input $a_1, a_2, \ldots, a_n$ is comparison-based if it accesses the input elements in only one way – by performing a comparison operation $a_i \stackrel{?}{<} a_j$ and acting according to the result of it. The outcome of the comparison operation is binary. Not all thinkable sorting algorithms are comparison based, for instance if it is known that $\forall i : a_i \in \{1, 2\}$ we can sort the input by counting the ones and filling in the output with ones and twos accordingly in $O(n)$ time. That algorithm, however, is not comparison-based and so the lower bound $\Omega(n \lg n)$ is not applicable to it. Recall Insertion Sort.

```
Insertion Sort(A[1, ..., n])
1  for i ← 2 to n
2      key ← A[i]
3      j ← i − 1
4      while j > 0 and key < A[j] do
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← key
```

Consider the work of Insertion Sort on an input of size three. Let the input be

$$a_1, a_2, a_3$$

We denote by lower case letter, *e.g.* "a_1", the elements of the input and with capital letters, *e.g.* "$A[1]$", the elements of the current array $A[\,]$. That means at the beginning necessarily $A[\,] = [a_1, a_2, a_3]$ but after several steps of the execution it may be the case that $A[\,] = [a_2, a_3, a_1]$. Let us assume the elements of the input are pairwise distinct. Precisely

one of the following is the case:

$$a_1 < a_2 < a_3 \tag{5.1}$$
$$a_1 < a_3 < a_2 \tag{5.2}$$
$$a_2 < a_1 < a_3 \tag{5.3}$$
$$a_2 < a_3 < a_1 \tag{5.4}$$
$$a_3 < a_1 < a_2 \tag{5.5}$$
$$a_3 < a_2 < a_1 \tag{5.6}$$

Let us call those, *the permutations*. To sort the input is the same as to determine which one of (5.1), ..., (5.6) is the case. INSERTION SORT is a comparison-based algorithm. The comparisons happer at one place only: the colour box at line 4 key < A[j] . The fist comparison that takes place is necessarily $a_2 < a_1$.

Case I If $a_2 < a_1$ is TRUE then precisely three permutations (out of the original six) are possible:

$$a_2 < a_1 < a_3$$
$$a_2 < a_3 < a_1$$
$$a_3 < a_2 < a_1$$

In that case the algorithm changes A[] during the current execution of the **for**-loop so that A[] becomes $[a_2, a_1, a_3]$. No more comparisons take place and the algorithm proceeds with the next execution of the **for**-loop. The next comparison in $a_3 < a_1$.

Case II If $a_2 < a_1$ is FALSE then precisely three permutations (out of the original six) are possible:

$$a_1 < a_2 < a_3$$
$$a_1 < a_3 < a_2$$
$$a_3 < a_1 < a_2$$

In this case the algorithm does not change A[] during the current execution of the **for**-loop and therefore A[] remains $[a_1, a_2, a_3]$. No more comparisons are done at line 4 and the algorithm proceeds with the next iteration of the **for**-loop.

Case I.1 If $a_3 < a_1$ is TRUE there are two possible permutations:

$$a_2 < a_3 < a_1$$
$$a_3 < a_2 < a_1$$

A[] becomes $[a_2, a_1, a_1]$, the value a_3 being stored in the variable key. The next comparison is $a_3 < a_2$.

Case I.1.a If $a_3 < a_2$ is TRUE there remains a single possible permutation:

$$a_3 < a_2 < a_1$$

Speaking of the algorithm, the**while**-loop is executed precisely once more and A[] becomes $[a_2, a_2, a_1]$. Tnen a_3 is placed at the first position in A[] (line 7) and A[] becomes $[a_3, a_2, a_1]$. Then the algorithm terminates.

I.1.b If $a_3 < a_2$ is FALSE there remains a single possible permutation:

$$a_2 < a_3 < a_1$$

Speaking of the algorithm, the**while**-loop is not executedany more. a_3 is placed at the second position in A[] (line 7)) and A[] becomes $[a_2, a_3, a_1]$.

Case I.2 $a_3 < a_1$ is FALSE there remains a single possible permutation:

$$a_2 < a_1 < a_3$$

Speaking of the algorithm, that is the end. A[] remains $[a_2, a_1, a_3]$.

The four subcases of **Case I** can be presented succinctly by the tree-like sttructure on Figure 5.1. There are two types of vertices – comparison vertices, corresponding to the questions of the kind $a_i \overset{?}{<} a_j$, and the leaves (in green). Above every comparison vertex we have recorded the permutations that are possible before that comparison has taken place. Naturally, above the root are all six possible permutations that are possible initially. The leaves correspond to all possible outputs of the algorithm. If we perform the analogous analysis of the subcases of **Case II**, we obtain the structure shown at Figure 5.2. The tree T is binary becase there are two possible outcomes for every element comparison.

We derived the tree-like structure by meticulous analysis of the work of INSERTION SORT on all possible inputs of size three. Such a tree-like scheme of questions and answers is called *a decision tree*. The definition of that concept in [CLR00] is:

> *A decision tree* is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored.

The decision trees do not have to binary in general and do not have to be limited to the sorting problem. The famous balance puzzle and the twelve-coin puzzle (see Problem 113 on page 156 and Problem 114 on page 158) are based on decision trees, though their trees are not binary but ternary. The balance puzzle, the twelve-coin puzzle, and INSERTION SORT on input of size three have a single decision tree because they consider fixed input size. Clearly, INSERTION SORT in general is characterised not by a single decision tree but by an infinite collection of decision trees, one for any input size. Both the balance puzzle and the twelve-coin puzzle have obvious generalisations for input of arbitrary size, in which case they have solutions with infinitely many decision trees, one for each input size.

It is important to realise we can think of the decision tree as as the primary object of consideration rather that a derived object. In the balance puzzle and the twelve-coin puzzle that is obvious because the solution is the decision tree. In the case of the the sorting problem that may not be obvious; however, note that decision trees *can* sort in some sense, not by moving elements around but by determining the sorted permutation after asking appropriate questions (without moving anything).

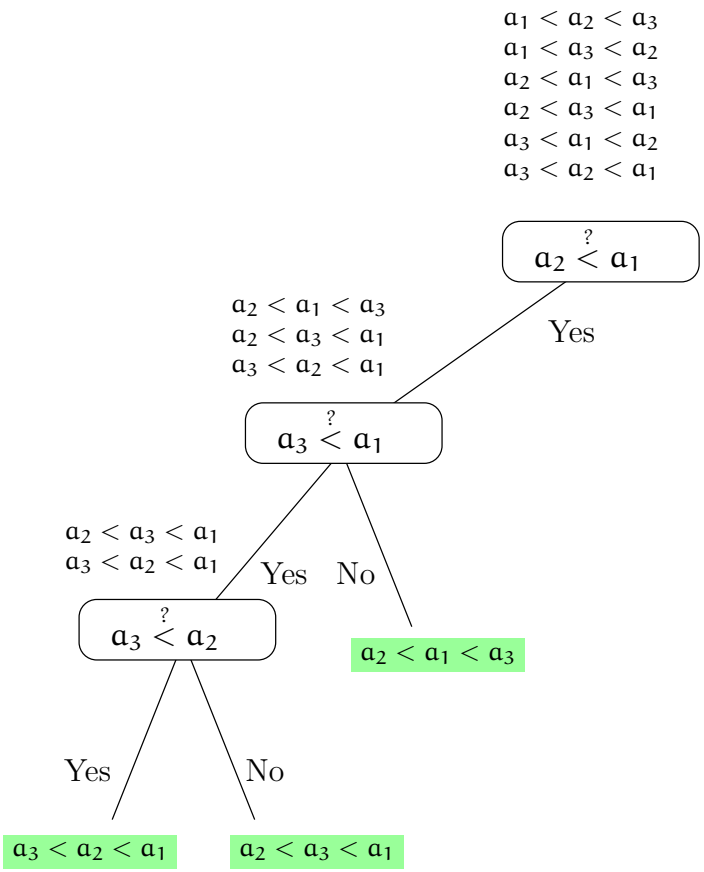


Figure 5.1: The subcases in which $a_2 < a_1$ is TRUE.

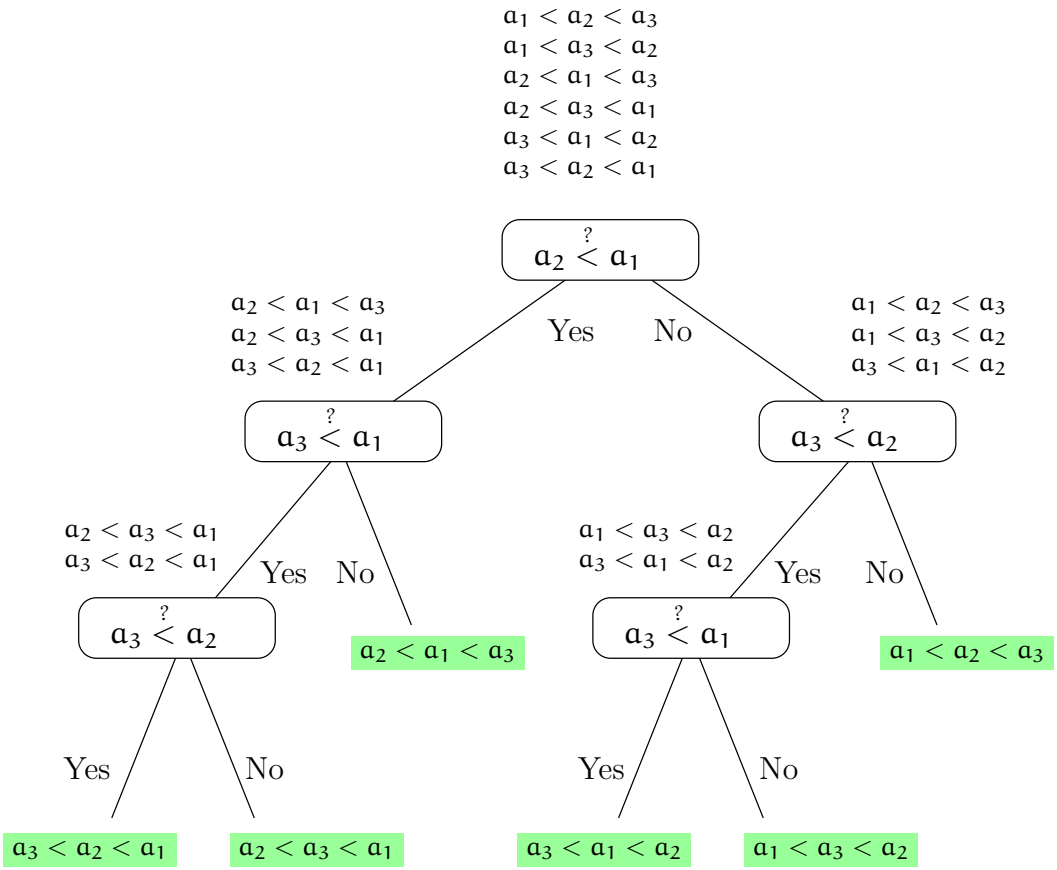


Figure 5.2: This is whole tree T of the comparisons. It represents a scheme $\mathcal{S}$ of questions and answers for determining the sorted permutation of three given numbers.

Now we show that decision trees provide us with a mechanism for establishing lower bounds. First consider the concrete tree T derived from Insertion Sort for input size three (Figure 5.2). The fact that there are leaves at distances two and three from the root means for certain inputs we determine the sorted permutation with two questions and for other inputs, with three questions. The height of the tree is the maximum number of questions we need—with respect to the particular scheme of questions and answers we have chosen—to detemine the sorted permutation. It has to be fairly clear that for input size three, the height of the tree has to be at least three. Assume there is another scheme $\mathcal{S}'$ of questions and answers for input size three that always determines with at most two questions the sorted permutation. The tree T' that corresponds to $\mathcal{S}'$ has height at most 2 and so it is different from T. However, just like T, it is a binary tree because the questions asked have precisely two possible outcomes. It is a trivial observation that T' cannot have more than four leaves, being a binary tree of height at most 2. But then $\mathcal{S}'$ cannot distinguish more than four different permutations. However, in order to be a correct scheme of questions and answers for determining the sorted permutation of three numbers, $\mathcal{S}'$ has to identify one out of *six*, not four, possible permutations. That refutes the claim such $\mathcal{S}'$ exists – there is no way its tree can identify uniquely one possibility among six ones. Note that analysis does not consider at all the specifics of the hypothetical tree T'. Surely we could investigate exhaustively all decision trees for sorting three elements and we would observe all of them have height at least three. However, that would very tedious and would hardly scale for arbitrary number of elements. The analysis we did is non-constructive and is based on the pigeonhole principle: the leaves of the hypothetical tree are fewer than the possibilities so it must be the case at least one leaf is associated with more than one possibility.

The non-constructive anaylsis scales easily for arbitrary many elements. For every input size n there is a scheme of questions and answers that determines the sorted permutation, the questions being of the type $a_i \stackrel{?}{<} a_j$. We know such schemes exist because there exist comparison-based sorting algorithms and to every comparison-based sorting algorithm there corresponds an infinite collection of decision trees, one for each input size. The sizes of these trees grows explosively with n because the number of the leaves is $\Omega(n!)$ so it is not feasible to even draw them for $n > 3$ but that is not important. What is important is that if we consider the set $\mathfrak{T}_n$ of all possible decision trees for sorting on input size n, for any n, the tree from $\mathfrak{T}_n$ with minimum height yields a lower bound for Sorting for in the worst case that many questions must be answered in order to compute the sorted permutation. A lower bound for the minimum height of tree from $\mathfrak{T}_n$ as a function of n is a lower bound for the computational problem comparison-based sorting.

Any tree from $\mathfrak{T}_n$ must have at least $n!$ leaves because the process of asking questions has to determine uniquely one object from $n!$ objects altogether. Since the tree is binary, its height is at least logarithmic (base 2) in the number of leaves. So every tree from $\mathfrak{T}_n$ has height at least $\log_2 n!$, and according to Problem 1.48, $\log_2 n! = \Theta(n \lg n)$. The lower bound $\Omega(n \lg n)$ for the comparison-based sorting follows right away.

5.2 The Balance Puzzle and the Twelve-Coin Puzzle

Problem 113 (The balance puzzle)**.** *We are given a set of 9 numbered items, say balls. From them 8 have identical weights and one is heavier. Our task is to identify the heavier ball using balance scales with the restriction that no standerd weights are provided. Thus the*

only possible way to perform a measurement is to put some balls in one pan, some other balls in the other pan, and observe the result. There are three possible outcomes from any such measurement: the left pan goes down, the right pan goes down, or both pans are balanced. We have to achieve our goal with as few measurements as possible.

Propose a scheme of measurements that identifies the heavy ball using the scales at most twice. Prove that two is a lower bound on the necessary number of measurements.

Solution:
Call the nine balls $\mathtt{b}_1$, ..., $\mathtt{b}_9$. Use the scales once measuring $\mathtt{b}_1$, $\mathtt{b}_2$, and $\mathtt{b}_3$ *versus* $\mathtt{b}_4$, $\mathtt{b}_5$, and $\mathtt{b}_6$. There are precisely three possible outcomes.

- If $\mathtt{b}_1$, $\mathtt{b}_2$, and $\mathtt{b}_3$ collectively are heavier than $\mathtt{b}_4$, $\mathtt{b}_5$, and $\mathtt{b}_6$, use the scales for a second time with $\mathtt{b}_1$ against $\mathtt{b}_2$. There are precisely three possible outcomes.
  - If $\mathtt{b}_1$ is heavier than $\mathtt{b}_2$, report *"$\mathtt{b}_1$ is the heavier ball".*
  - If $\mathtt{b}_2$ is heavier than $\mathtt{b}_1$, report *"$\mathtt{b}_2$ is the heavier ball".*
  - If none of $\mathtt{b}_1$ and $\mathtt{b}_2$ is heavier than the other one, report *"$\mathtt{b}_3$ is the heavier ball".*
- If $\mathtt{b}_4$, $\mathtt{b}_5$, and $\mathtt{b}_6$ collectively are heavier than $\mathtt{b}_1$, $\mathtt{b}_2$, and $\mathtt{b}_3$, use the scales for a second time with $\mathtt{b}_4$ against $\mathtt{b}_5$. There are precisely three possible outcomes.
  - If $\mathtt{b}_4$ is heavier than $\mathtt{b}_5$, report *"$\mathtt{b}_4$ is the heavier ball".*
  - If $\mathtt{b}_5$ is heavier than $\mathtt{b}_4$, report *"$\mathtt{b}_5$ is the heavier ball".*
  - If none of $\mathtt{b}_4$ and $\mathtt{b}_5$ is heavier than the other one, report *"$\mathtt{b}_6$ is the heavier ball".*
- If $\mathtt{b}_1$, $\mathtt{b}_2$, and $\mathtt{b}_3$ collectively are as heavy as $\mathtt{b}_4$, $\mathtt{b}_5$, and $\mathtt{b}_6$, use the scales for a second time with $\mathtt{b}_7$ against $\mathtt{b}_8$. There are precisely three possible outcomes.
  - If $\mathtt{b}_7$ is heavier than $\mathtt{b}_8$, report *"$\mathtt{b}_7$ is the heavier ball".*
  - If $\mathtt{b}_8$ is heavier than $\mathtt{b}_7$, report *"$\mathtt{b}_8$ is the heavier ball".*
  - If none of $\mathtt{b}_7$ and $\mathtt{b}_8$ is heavier than the other one, report *"$\mathtt{b}_9$ is the heavier ball".*

Figure 5.3 illustrates that measurements scheme. It is very useful to think in terms of possibilities, or in other words, possible states. Initially there are precisely nine possibilities with respect to the balance puzzle: either $\mathtt{b}_1$ is heavier, or $\mathtt{b}_2$ is heavier, ..., or $\mathtt{b}_9$ is heavier. Let us denote those by *i*, *ii*, ..., *ix*, respectively. The initial set of possibilities is $\{i, ii, \ldots, ix\}$. Every time we measure, the set of possibilities grows smaller. The puzzle is solved iff we reduce the set of possibilities to sets of size at most one. On Figure 5.3 we write the set of possibilities next to nodes that represent measurements. That means, those sets are possible *before* the measurement takes place.

The proof of the lower bound is trivial: we have to distinguish one out of nine possibilities. We use a ternary decision tree. A ternary tree with nine leaves must have height at least two, and therefore any scheme ot measurements must use the scales at least twice.

□

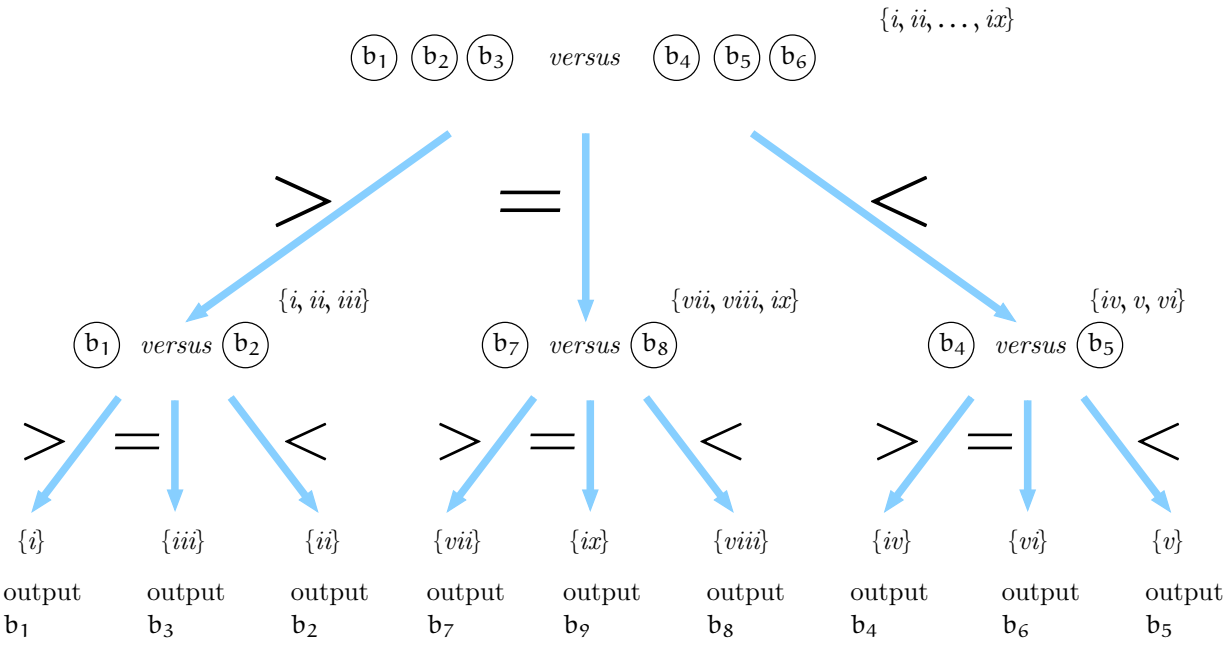


Figure 5.3: A measurement scheme with two questions for the balance puzzle. The possibilities are written with roman numerals. To solve the puzzle is to decide which of all nine possibilities is the case.

Problem 114 (The twelve-coin puzzle)**.** *We are given a set of* 12 *numbered items, say coins. From them* 11 *have identical weights and one—call it the odd coin—is heavier or lighter. Our task is to identify the odd coin using balance scales as those in Problem 113. Propose a scheme of measurements that identifies the heavy coin using the scales at most three times. Prove that three is a lower bound on the necessary number of measurements.*

Solution:
This puzzle is a bit more complicated than the balance puzzle. The number of possibilities now is 24, twice the number of coins. Denote by *i'* the possibility "b_1 is lighter", by *i"* the possibility "b_1 is heavier", by *ii'* the possibility "b_2 is lighter", by *ii"* the possibility "b_2 is heavier", *etc.*, by *xii'* the possibility "b_{12} is lighter", and by *xii"* the possibility "b_{12} is heavier". Figure 5.4 shows a measurement scheme with at most three measurements determining which of the twenty four possibilities is the case.

Now we prove that three measurements are necessary. Since there are 24 possibilities altogether and we have to find out which one of them is the case, a lower bound for the number of measurements is $\lceil \log_3 24 \rceil = 3$, the reason being that a ternary decision tree with ≥ 24 leaves has height at least three.

Note that the derivation of a lower bound using a decision tree argument does *not* imply that bound is achievable. In the case of the twelve-coin puzzle, the lower bound three is indeed achievable as demonstrated by Figure 5.4. However, the same problem with thirteen coins *cannot* be solved with at most three measurements in the worst case. To see why, consider that

- it makes no sense to weigh two sets of coins with different cardinalities during the first

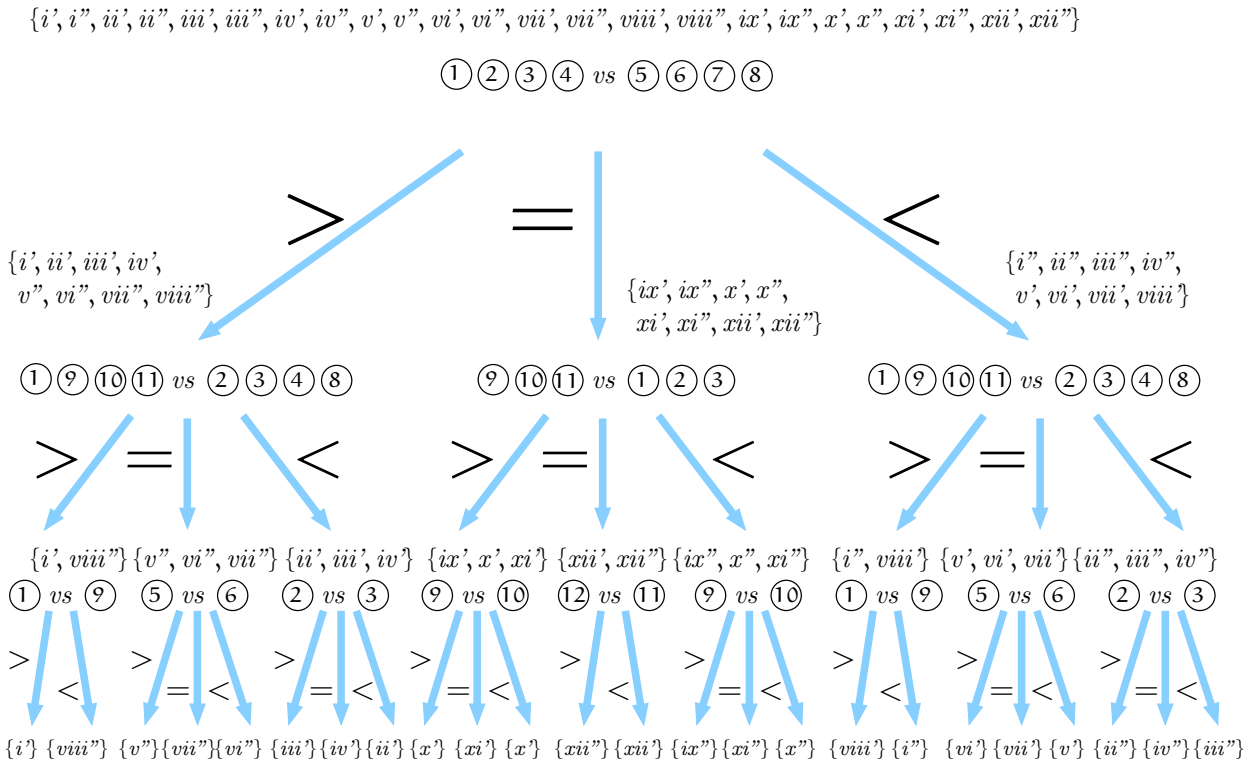


Figure 5.4: A measurement scheme for the twelve-coin puzzle. Note that for some measurements, only two of all three outcomes are possible.

measurement because if the pan with the bigger set goes down we gain *no information whatsoever* and the possibilities remain the same;

- having in mind the previous consideration, for any choice of two subsets of the same cardinality of the thirteen coins, either those sets have cardinality ≥ 5, or the remaining set has cardinality ≥ 5;
- a set of cardinality ≥ 5 has ≥ 10 possibilities associated with it;
- there is no way to determine which of the ≥ 10 possibilities is the case using ≤ 2 measurements.

But we can derive a lower bound three for the thirteen-coin puzzle: $\lceil \log_3 26 \rceil = 3$. There is no contradiction between the fact that $\lceil \log_3 26 \rceil = 3$ and the above proof that the thirteen-coin puzzle cannot be solved using three measurements at worst. Those are separate arguments – the lower bound three is still valid but, unlike the twelve-coin puzzle, now it is not achievable. □

5.3 Comparison-based element uniqueness

Computational Problem Element Uniqueness
Generic Instance: A list of numbers $a_1, a_2, \ldots, a_n$
Question: Are all numbers unique? □

An algorithm for Element Uniqueness is comparison based if it accesses the input elements in only one way – by performing a comparison operation $a_i \stackrel{?}{<} a_j$ and acting according to the result of it, the outcome of the comparison operation being binary. The naive comparison-based algorithm for Element Uniqueness is to compare all unordered pairs from the input in time $\Theta(n^2)$. A more sophisticated algorithm is to sort the input first in $\Theta(n \lg n)$ and then determine whether there are repeating numbers or not by a single linear sweep: if there are repeating elements they will form a contiguous subsequence after the sorting. The overall running time is $\Theta(n \lg n)$.

```
Comparison-based Element Uniqueness(A[1,...,n]: array of integers)
1  Sort(A[])
2  are_unique ← True
3  for i ← 2 to n
4      if A[i − 1] = A[i]
5          are_unique ← False
6  return are_unique
```

Now we prove that any algorithm for comparison-based Element Uniqueness has time complexity $\Omega(n \lg n)$ using a decision tree argument. Any algorithm for that problem has a binary decision tree associated with it. In fact, as we pointed out above, a particular algorithm has an infinite set of decision trees associated with it, one for each input size. Speaking of input size n, the tree is only one. In order to prove the lower bound it suffices to prove that the tree has $\Omega(n!)$ leaves and then apply the argument we made on page 156

for comparison-based sorting. In its turn, in order to prove the tree has $\Omega(n!)$ leaves it suffices to prove the tree distinguishes all permutations of the input in the sense that every leaf is associated with at most one permutation.

With respect to the sorting problem that task is trivial: if the decision tree has a leaf associated with ≥ 2 permutations then the algorithm does not sort correctly. With respect to the uniqueness of the elements it is not so obvious the leaves should be associated with different permutations. The proof below follows a proof from [SW11].

Lemma 24. *Suppose* X *is a comparison-based algorithm for* Element Uniqueness *and its decision tree for input of size* n *is* T. *Every leaf of* T *is associated with at most one permutation of the input elements.*

Proof:
We can assume without loss of generality that the input elements are unique because we perform worst-case analysis. If we prove the claim for inputs of unique elements we are done because that restriction scales: for every n there is an input of unique elements and since X is a correct algorithm it has to work correctly on that instance.

Assume that the smallest input value is b_1, the second smallest is b_2, *etc.*, the largest input value is b_n. Clearly, there are n distinct b values under the assumption all input elements are unique. We emphasise that

$$b_1 < b_2 < \ldots < b_n$$

First we prove that for every k such that $1 \leq k < n$, X compares b_{k-1} and b_k at some moment of its execution. Assume the opposite. Namely, there exists an input $A[1, \ldots, n]$ of unique elements such that X does not compare the $(k-1)$-th and the k-th smallest elements. Since X is a correct algorithm and all elements of $A[]$ are unique, $X(A[1, \ldots, n])$ returns True. Let $A[i]$ be the input element with value b_{k-1} and $A[j]$ be the input element with value b_k. Now transform $A[1, \ldots, n])$ into $A'[1, \ldots, n])$ by changing the value of the $A[i]$ to b_k and run $X(A'[])$. As X is a correct algorithm it has to return No because $A'[]$ has repeating elements. However, the work of X on $A'[]$ is identical to its work on $A[]$ because, by assumption, it never compares $A[i]$ with $A[j]$ and thus all the comparisons it performs and their outcomes are precisely the same for both inputs. Then $X(A[1, \ldots, n])$ and $X(A'[1, \ldots, n])$ return the same value, contradicting the former assumption that X is a correct algorithm.

And so we proved that X has to compare every pair (there are $n-1$ of them) of elements with adjacent values. Now we prove a crucial fact. Suppose a_1, a_2, $\ldots$, a_n are distinct numbers and a'_1, a'_2, $\ldots$, a'_n is any permutation on them different from the identity. Suppose that b_1 is the smallest value, b_2 is the second smallest value, *etc.*, b_n is the largest value. Then there exists a k such that $1 \leq k < n$, such that if a_i has value b_{k-1} and a_j has value b_k, which means $a_i < a_j$, it is the case that $a'_i > a'_j$. This fact is trivial though it may sound intimidating. Reworded, it says for any "true" permutation there exist adjacent values (not adjacent positions but adjacent *values*) in the original list such that after the permutation, the inequality sign between the elements at their *position* is the opposite. For instance, if the original list is

$$2, 5, 3, 1, 4, 6$$

and the permutation is

$$3, 5, 4, 1, 6, 2$$

note that $a_5 = 4$ and $a_2 = 5$ form such pair: a'_5 is 6 and a'_2 is 5, thus $a'_5 > a'_2$. The proof of the fact is easily done by assuming the opposite. The opposite is, for every two positions containing elements with adjacent values in the original list, after the permutation takes place, the inequality sign between the elements in these positions is the same as in the original list. But then it follows trivially the permutation keeps all elements in their places, in contradiction with the initial assumption the permutation is different from the identity.

Having proved the two auxiliary facts, the proof of the lemma follows easily. Assume that the decision tree T of X associates some leaf u with two or more permutations of the input. Consider any two of them. Since they are different permutations, one of them can be thought of as non-identity permutation of the other one. By the second fact above, there exist adjacent values in the first permutation at positions i and j such that in the second permutation the values at positions i and j are in the opposite order. Let us clarify that point. Leaf u has two permutations $a_1, a_2, \ldots, a_n$ and $a'_1, a'_2, \ldots, a'_n$ of the input associated with it. That means, both of them are consistent with the set of questions and answers so far (*i.e.* with the non-leaf vertices on the path from the root to u). If $b_1, \ldots, b_n$ are the values of the input in increasing order, the second fact says that for some adjacent values b_{k-1} and b_k, if they are at positions i and j, respectively, in $a_1, a_2, \ldots, a_n$, then in $a'_1, a'_2, \ldots, a'_n$ the elements at positions i and j, namely a'_i and a'_j are such that $a'_i > a'_j$. However, in the other permutation, $a_i < a_j$. The first fact implies that X must have asked the question *"Is it the case that* $b_{k-1} < b_k$*?"*. However, X cannot ask questions about adjacent values directly: if it could ask such questions in constant time, the problem would be solvable in linear time. The algorithm rather asks questions of the type $a_i \overset{?}{<} a_j$. The first fact implies the algorithm has to be designed in a way such that b_{k-1} and b_k are unavoidably compared, regardless of their positions in the list. And the crucial observation is that X must have compared b_{k-1} with b_k by asking question of the type $a_i \overset{?}{<} a_j$. For one of the two said permutations the answer is YES, for the other one, NO. It follows that no leaf can be associated with both permutations. □

Chapter 6

Algorithmic Problems

6.1 Programming fragments

Problem 115. *Determine the asymptotic running time of the following programming fragment. Function* `q()` *is an unspecified function, it works in* $\Theta(1)$ *time and can swap elements of the array* `A`. `Heapsort(A, i, j)` *sorts* `A[i..j]` *by the eponymous sorting algorithm.*

```
int A[MAXINT];

int p(int, int);
void q(int, int);

int main() { return p(1, n); }

int p(int i, int j) {
  int mid, a, b;
  if (j > i) {
   Heapsort(A, i, j);
   q(i, j);
   mid = (i+j)/2;
   a = p(i, mid);
   b = p(mid + 1, j);
   return a + b; }
  return 1; }
```

Solution:
It is well known that `Heapsort` has $\Theta(n \lg n)$ worst case time complexity. We have to assume that at every recursive level its complexity is the worst possible because we do not know how `q()` works. There are two resursive calls at each execution of `p()`, therefore the running time is determined by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \lg n)$$

Using Theorem 2 on page 96, we conclude $T(n) = \Theta(n \lg^2 n)$. □

Problem 116. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
void r(int, int, int);

int main() { return r(1, n, n*n); }

void r(int a, int b, int c) {
  int k;
  if (a + b + c > a + b + 1) {
   for(k = 1; k < a + b + c; k = (k << 2) - 1) {
    if(k % 3 == 0) break;
    r(a, b, c - 1); }
   for(k = 1; k < a + b + c; k <<= a + b + c)
    r(a, b, c - 1); } }
```

Solution:
Only the third parameter of `r()` determines the recursion; the first two parameters are insignificant. The body of the first `for` loop is executed precisely once because the second value that `k` gets is $(1 * (2^2) - 1) = 3$, then the condition of the `if` operator is evaluated to TRUE, and after the `break` operator the loop is executed no more.

The second `for` loop is executed only once, too: the second value that `k` gets is 2^{a+b+c}, and certainly $2^{a+b+c} > a + b + c$. There are two resursive calls at each execution of `r()`, therefore the running time is determined by the recurrence

$$T(m) = 2T(m-1) + \Theta(1)$$

where m is the size of the input. The solution is known to be $T(m) = \Theta(2^m)$. Having in mind that $m = n^2$, it is easy to see that $T(n) = \Theta\left(2^{n^2}\right)$. □

Problem 117. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int foo(int n) {
  int s;
  if (n == 0) s = 1;
  else s = foo(n - 1) * 2;
  bar(s);
  return s; }

void bar(int m) {
  int i;
  for (i = m; i > 0; i /= 2)
    cout << i % 2; }
```

Solution:
First we prove that `foo(n)` returns 2^n. Note that the `bar()` function, while affecting the running time, does not affect the value of `s` is any way. We prove the claim by induction

on `n`. For $n = 0$ it is obviously the case that `foo(n)` returns $1 = 2^0$. Assuming `foo(n-1)` returns 2^{n-1}, it is clear `foo(n)` returns $2 \times 2^{n-1} = 2^n$.

Note that `bar()` runs in time $\Theta(\lg m)$. Having in mind that `s` is 2^n and `bar()` runs in logarithmic time with respect to its input, it is clear that `bar(s)` runs in time $\Theta(\lg(2^n)) = \Theta(n)$. We conclude the time complexity of `foo()` is determined by the recurrence

$$T(n) = T(n-1) + n$$

According to (3.19) on page 51, $T(n) = \Theta(n^2)$. □

Problem 118. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int recf(int n) {
  int s;
  if (n == 0) s = 1;
  else s = foo(n - 1) * 2;
  bar(s);
  return s; }

void bar(int m) {
  int i;
  for (i = m; i > 0; i /= 2)
    cout << i % 2; }
```

Solution:
First we prove that `foo(n)` returns 2^n. Note that the `bar()` function, while affecting the running time, does not affect the value of `s` is any way. We prove the claim by induction on `n`. For $n = 0$ it is obviously the case that `foo(n)` returns $1 = 2^0$. Assuming `foo(n-1)` returns 2^{n-1}, it is clear `foo(n)` returns $2 \times 2^{n-1} = 2^n$.

Note that `bar()` runs in time $\Theta(\lg m)$. Having in mind that `s` is 2^n and `bar()` runs in logarithmic time with respect to its input, it is clear that `bar(s)` runs in time $\Theta(\lg(2^n)) = \Theta(n)$. We conclude the time complexity of `foo()` is determined by the recurrence

$$T(n) = T(n-1) + n$$

According to (3.19) on page 51, $T(n) = \Theta(n^2)$. □

Problem 119. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int recf(int n) {
  int i, s = 1;
  if (n == 1) return s;
  for (i = 0; i < 3; i ++) s += recf(n-1) * (i + 1);
  for (i = 0; i < 4; i ++) s += recf(n-2) * (i + 1);
  return s; }
```

Solution:
The time complexity of `recf()` is determined by the recurrence

$$T(n) = 3T(n-1) + 4T(n-2) + 1$$

To see why, note there are three recursive calls with input of size $n-1$ and four, with input of size $n-2$. According to Problem 104 on page 103, $T(n) = \Theta(4^n)$. □

⁘ NB ⁘ *We should not try to "optimise" the number of recursive calls. One may indeed be tempted to think we can make only one recursive call with input $n-1$ and then use the obtained value three times, rather than making three consecutive recursive calls (and likewise, only one call with input $n-2$ and then use the result four times. Such "optimisations" are not allowed: the algorithm should be investigated as it is. Furthermore, it is possible that the shown fragment is an abbreviated version of a program that does a lot more, for instance it may change a global variable. If that is the case, we cannot substitute a multitude of recursive calls by a single call and claim that the new "optimised" program is necessarily equivalent.*

Problem 120. *Determine the asymptotic running time of the following programming fragment as a function of n.*

```
int r(int n, int m) {
  int i, s = 0;
  if (n < 2) return n*m + 1;
  for(i = 0; i < 3; i ++) {
    s += r(n-1,m+i) * r(n-2,m-i); }
  s += r(n-1,m);
  return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 4T(n-1) + 3T(n-2) + 1$$

To see why that is true note that the variable that determines the recursive calls is n. The other variable m does not affect the time complexity, though it surely affects the returned quantity. There are three recursive calls with $n-2$ and four with $n-1$. The fact that there is a multiplication $r(n-1, m+i) * r(n-2, m-i)$ is immaterial with respect to the structure of the recursive calls. According to Problem 105 on page 103, $T(n) = \Theta((2+\sqrt{7})^n)$. □

Problem 121. *Determine the asymptotic running time of the following programming fragment as a function of n.*

```
int f(int n, int m) {
  int i, s = 0;
  if (n == 0 || n == 1)
    return m;
  for(i = 0; i < 5; i++) {
```

```
    s += f(n-1, m + i);
    s += f(n-2, m + 2*i); }
  s += f(n-2, 2*m)*3;
 return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 5T(n-1) + 6T(n-2) + 1$$

To see why, note that the variable that controls the recursive calls is n. The other variable m does not affect the time complexity, though it surely affects the returned quantity. There are six recursive calls with $n-2$ and five with $n-1$. According to Problem 106 on page 104, $T(n) = \Theta(6^n)$. □

Problem 122. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int r(int n) {
  int i, s = 2;
  if (n == 1)
    return 2;
  for(i = n; i > 0; i /= 2) {
    s += 2; }
  s += r(n/2)*r(n/2);
  return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n$$

To see why, note the `for` loop runs in $\Theta(\lg n)$ time and afterwards two recursive calls are made, each one on an input that is half the size of the original one. According to Problem 90 on page 97, $T(n) = \Theta(n)$. □

Problem 123. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int bf(int m, int v) {
  if (m == 1)
    return v;
  return bf(m - 1, 1) * v && bf(m-1, 0) * (1 - v); }

int main() {
  return bf(n, x); }
```

Solution:
At a first glance, it seems the fragment has exponential time complexity: two recursive calls are made, each with input of size $n-1$, and only constant work is done in addition to them, therefore the recurrence $T(n) = 2T(n-1)+1$ determines the complexity, and we know (see Problem 98 on page 102) this recurrence has solution $T(n) = \Theta(2^n)$.

However, if we take into account the operator precedence of the `C` language[†] and the evaluation of the logical operators[‡], the analysis is quite different. Suppose we make the initial call `bf(n, x)` with a sufficiently large value of `n`. The first recursive call is `bf(n - 1, 1)`. Inside it, the first recursive call is `bf(n - 2, 1)`, *etc.*, until `bf(2, 1)` calls `bf(1, 1)`. Clearly, `bf(1, 1)` returns `1` and then `bf(2, 1)` makes its second recursive call, `bf(1, 0)`. The latter returns `0` and the `&&` operator within `bf(2, 1)` evaluates `1 && 0` to `0` and passes `0` upwards to `bf(3, 1)`. Now `bf(3, 1)` does not call `bf(2, 0)` because, by the rules of `C`, the evaluation of the `&&` operator is left to right and it stops once the value is known—`bf(2, 1)` returning `0` implies the result within `bf(3, 1)` is necessarily `0`. Thus `bf(3, 1)` passes `0` upwards, `bf(4, 1)` does not call `bf(3, 0)` but passes `0` upwards directly, *etc.*, until `bf(n, x)` returns `0` without calling `bf(n - 1, 0)`. In other words, the recursion tree is in fact a path, except that at the very bottom, `bf(2, 1)` has two children. It follows that the time complexity is linear in `n` rather than exponential. The recursion tree is shown on Figure 6.1.

Of course, that is the case only under the assumption that the rules of the `C` language apply. If that fragment were written in pseudocode the time complexity would be proved to be exponential since we have no standard rules for the direction and early stopping of the evaluation of logical expressions in pseudocode. □

Problem 124. *Determine the asymptotic running time of the following programming fragment as a function of* `n`.

```
unsigned n;
int main() {
  return recf(1, n); }

int recf(unsigned i, unsigned j) {
  int k, s = 0;
  if (j - i > 3) {
    for(k = 0; k < 4; k ++) {
      s += recf(i + k, j + k - 3); }
    return s; }
  else
  return 1; }
```

Solution:
The execution of the recursion is controlled by the difference `j-i` of the two input variables. Let us call it, *the control difference*. For all large enough values of the control difference, *i.e.* whenever $j - i > 3$, there are exactly four recursive calls, each one having control difference

[†] See [KR88], pp. 53.

[‡] *ibid.*, pp. 41: "*More interesting are the logical operators* `&&` *and* `||`. *Expressions connected by* `&&` *or* `||` *are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.*"

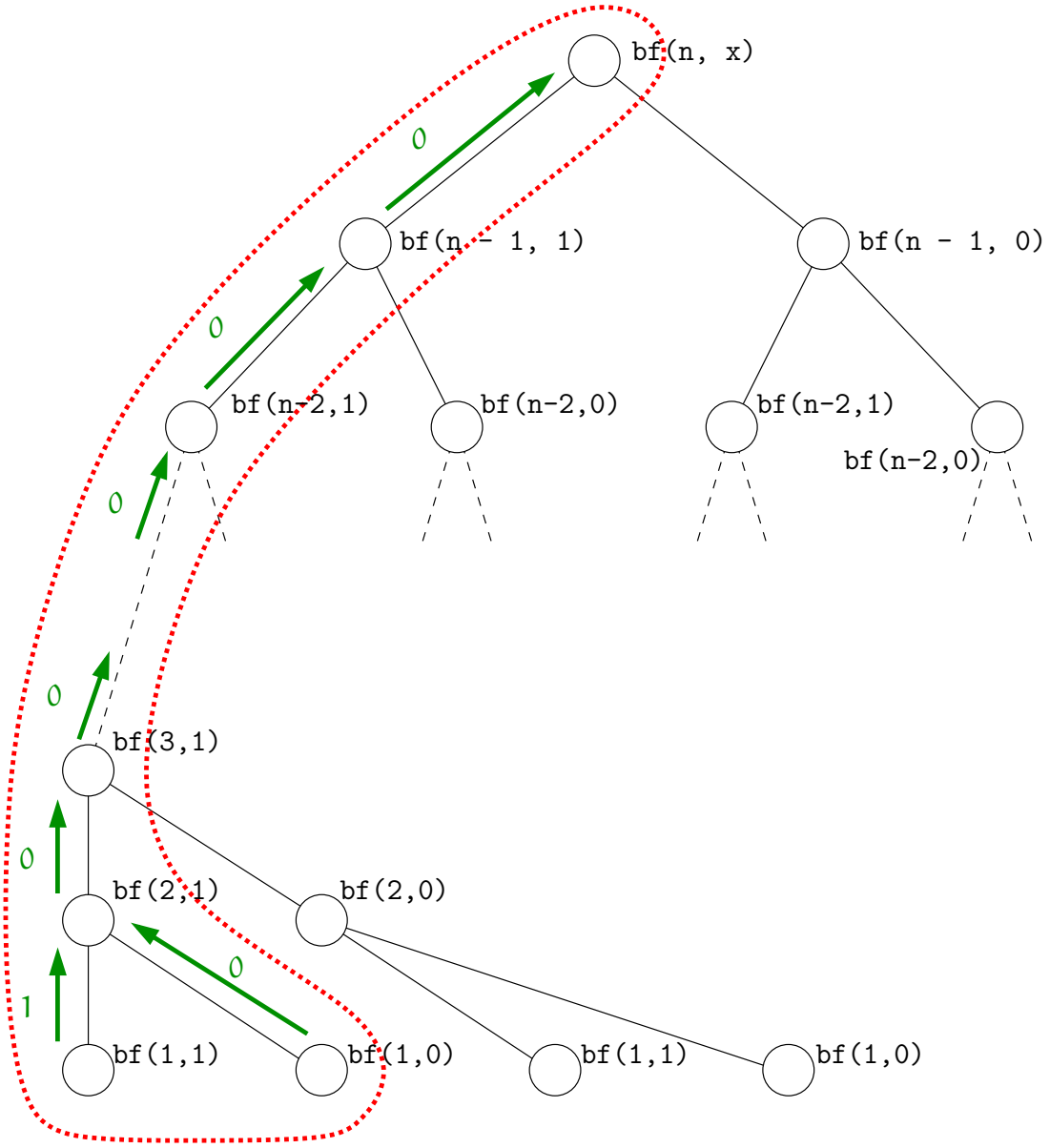


Figure 6.1: The recursion tree of the fragment in Problem 123. We draw the whole tree, *i.e.* what the tree would be if the considerations about the `&&` operator were not taken into account. The part of the tree that corresponds to the execution of the fragment when these considerations are taken into account, is outlined with dashed red line. The flow of the returned values is drawn in green.

that is smaller by three since $j+k-3-(i+k) = j-i-3$ for $k \in \{0,1,2,3\}$. Therefore, the following recurrence relation determines the asymptotic running time:

$$T(n) = 4T(n-3) + 1$$

According to Problem 107 on page 104, $T(n) = \Theta\left(4^{\frac{n}{3}}\right)$. □

6.2 Arrays and sortings

Problem 125. *Let* $C_1, C_2, \ldots, C_n$ *be* n *cities placed along a straight line. Let* d_i *be the distance between* C_i *and* C_{i+1}*, for* $1 \leq i < n$*. Initially each city* C_i *possesses some amount* $x_i \in \mathbb{R}$ *of a certain resource, say water. If* $x_i < 0$*, what* C_i *possesses is not real water but deficiency of water,* e.g. *if* C_i *has* -5.5 *units of water and afterwards we transport* 6 *units of water to it, it is going to have* $+0.5$ *units. Each city* C_i *needs some amount* $l_i \in \mathbb{R}^+$ *of water. We say* C_i *is* satisfied *iff* $l_i \leq x_i$*.*

Water can be transported between any two adjacent cities but the transportation is lossy: if we start transporting amount z *from* C_i *to* C_{i+1} *or* vice versa*, the amount that is going to be delivered is* $\max\{z - d_i, 0\}$*.*

Design an algorithm that outputs TRUE*, in case there is a way to transport water between the cities so that every city is satisfied, or* FALSE*, otherwise. Assume that the amount of water in any city is constant unless water is transported to, or from, it. Prove the correctness of your algorithm and analyse its time complexity.*

Solution: Define d_n to be zero. Consider the following algorithm:

```
TRANSPORT(x_1, ..., x_n, d_1, ..., d_n, l_1, ..., l_n)
 1  s ← 0
 2  for i ← 1 to n
 3      Δ ← x_i − l_i
 4      if s + Δ ≥ 0
 5          s ← max{s + Δ − d_i, 0}
 6      else
 7          s ← s + Δ − d_i
 8  if s ≥ 0
 9      return TRUE
10  else
11      return FALSE
```

For any i such that $1 \leq i \leq n$, let $\mathcal{A}_i$ be the subarray of cities $[C_1, C_2, \ldots, C_i]$. $\mathcal{A}_i$ is called *good* if there is a way to satisfy its cities by transporting water only between them. Otherwise, $\mathcal{A}_i$ is called *wanting*. Suppose q is a positive amount. When we say $\mathcal{A}_i$ is q-*redundant* we mean that:

- $\mathcal{A}_i$ is good and it remains good even if the amount in C_i is decreased by q units beforehand.
- However, if the amount in $\mathcal{A}_i$ is decreased by $q + \epsilon$ units beforehand, for any $\epsilon > 0$, $\mathcal{A}_i$ becomes wanting.

When we say $\mathcal{A}_i$ is *isolated* we mean that:

- $\mathcal{A}_i$ is good.
- However, if the amount in $\mathcal{A}_i$ is decreased by $d_i + \epsilon$ units beforehand, for any $\epsilon > 0$, $\mathcal{A}_i$ becomes wanting.

When we say $\mathcal{A}_i$ is q-*deficient* we mean:

- $\mathcal{A}_i$ is wanting but it becomes good if the amount in C_i is increased by q units beforehand.
- However, if the amount in $\mathcal{A}_i$ is increased by $q - \epsilon$ units beforehand, for any $\epsilon > 0$, $\mathcal{A}_i$ remains wanting.

Note the distintion between isolated and redundant: isolatedness is not a special case of redundancy although it may sound like being a special case, namely for $q = d_i$. Now we point out the difference. If $\mathcal{A}_i$ is d_i-redundant then it is good and d_i is precisely equal to the largest quantity that can transported out of C_i beforehand (keeping $\mathcal{A}_i$ good). So, d_i is a threshold value. On the other hand, if $\mathcal{A}_i$ is isolated then it is good and with certainty transporting any amount $\geq d_i$ out of C_i ruins its goodness; the key observation is, it is possible that transporting even the smallest quantity out of C_i may ruin the goodness – the definition of isolatedness allows that. So, in the case with isolatedness, there is no certain threshold quantity that can safely be transported out of C_i.

Lemma 25. *Algorithm* TRANSPORT *returns* TRUE *iff* $\mathcal{A}_n$ *is good.*

Proof:
The following is a loop invariant for the **for**-loop (lines 2–7):

Every time the execution of TRANSPORT is at line 2,

- if $s > 0$ then $\mathcal{A}_{i-1}$ is $(s + d_{i-1})$-redundant.
- if $s = 0$ then $\mathcal{A}_{i-1}$ is isolated.
- if $s < 0$ then $\mathcal{A}_{i-1}$ is $|s + d_{i-1}|$-deficient.

Basis. The first time the execution is at line 2, i equals 1 and so $\mathcal{A}_{i-1}$ is empty. The empty subarray of cities is vacuously isolated because it is vacuously good (no deficiency) and its water—of which there is none—cannot be decreased by any amount. On the other hand, s equals 0 because of the assignment at line 1. So, the invariant holds.

Maintenance. Assume the claim holds at a certain execution of line 2 and the **for** loop is to be executed at least once more. Δ is set to $x_i - l_i$ at line 3. Let us call the value of s prior to the execution of line 5 or line 7, s_{old}, and the value of s after that execution, s_{new}.

Case I: Suppose $s_{old} > 0$. By the assumption, $\mathcal{A}_{i-1}$ is $(s_{old} + d_{i-1})$-redundant. That means we can deliver s_{old} units of water to C_i by transporting $s_{old} + d_{i-1}$ out of C_{i-1} to C_i (losing d_{i-1} quantity along the way), $\mathcal{A}_{i-1}$ remaining good. However, if we transport any more water out of C_{i-1} to C_i, $\mathcal{A}_{i-1}$ becomes wanting. So, C_i can get at most s_{old} water more (keeping $\mathcal{A}_{i-1}$ good).

Case I.1: Suppose $s_{old} + \Delta \geq 0$. Then the assignment at line 5 takes place. The following two facts:

1. $\mathcal{A}_{i-1}$ is good and C_i can get s_{old} water more, $\mathcal{A}_{i-1}$ staying good.

2. $s_{old} + \Delta \geq 0$.

imply $\mathcal{A}_i$ is good. Furthermore, $s_{old} + \Delta$ is the surplus in C_i that can be transported out of C_i (towards C_{i+1}), keeping $\mathcal{A}_i$ good; that is the threshold value, anything more out of C_i makes $\mathcal{A}_i$ wanting. So, $\mathcal{A}_i$ is $(s_{old} + \Delta)$-redundant.

Case I.1.a: Suppose $s_{old} + \Delta - d_i > 0$. Then $s_{new} = s_{old} + \Delta - d_i$. Now we prove $\mathcal{A}_i$ is $(s_{new} + d_i)$-redundant. We know $\mathcal{A}_i$ is good and $(s_{old} + \Delta)$-redundant. Observe that $s_{old} + \Delta = s_{old} + \Delta - d_i + d_i = s_{new} + d_i$ and conclude $\mathcal{A}_i$ is $(s_{new} + d_i)$-redundant. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $(s + d_{i-1})$-redundant.

Case I.1.b: Suppose $s_{old} + \Delta - d_i = 0$. Then $s_{new} = 0$. Recall that $\mathcal{A}_i$ is good. Note that $s_{old} + \Delta - d_i - \epsilon < 0$ for any $\epsilon > 0$. It follows $\mathcal{A}_i$ is isolated[†]. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is isolated.

Case I.2: Suppose $s_{old} + \Delta < 0$, which means $\Delta < 0$ since $s_{old} > 0$. But then $\mathcal{A}_i$ is wanting because, as we already pointed out, if C_i gets any more water than s_{old}, $\mathcal{A}_{i-1}$ becomes wanting. In order to make $\mathcal{A}_i$ good, at least $|s_{old}+\Delta|$ has to be transported into C_i. By defintion, $\mathcal{A}_i$ is $|s_{old} + \Delta|$-deficient. Having in mind that $s_{old} + \Delta = s_{old} + \Delta - d_i + d_i = s_{new} + d_i$, it follows $\mathcal{A}_i$ is $|s_{new} + d_i|$-deficient. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $|s_{new} + d_{i-1}|$-deficient.

Case II: Suppose $s_{old} = 0$. By the induction hypothesis, $\mathcal{A}_{i-1}$ is closed. That means $\mathcal{A}_{i-1}$ is good but we cannot transport $d_{i-1} + \epsilon$ units of water out of C_{i-1} (towards C_i), for any $\epsilon > 0$, and keep $\mathcal{A}_{i-1}$ good. So, no amount of water from $\mathcal{A}_{i-1}$ can go into C_i if we are to keep $\mathcal{A}_{i-1}$ good. It follows the status of $\mathcal{A}_i$ depends entirely on Δ and d_i.

Case II.1: Suppose $s_{old} + \Delta \geq 0 \Leftrightarrow \Delta \geq 0$. Then the assignment at line 5 takes place.

Case II.1.a: Suppose $\Delta - d_i > 0$. Then $s_{new} = \Delta - d_i$. $\mathcal{A}_i$ is good and it is possible to transport Δ water out of C_i (towards C_{i+1}), keeping $\mathcal{A}_i$ good; furthermore, Δ is the threshold value, any more will make $\mathcal{A}_i$ wanting. Then $\mathcal{A}_i$ is Δ-redundant, *i.e.* $(s_{new} + d_i)$-redundant. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $(s + d_{i-1})$-redundant.

Case II.1.b: Suppose $\Delta - d_i = 0$. Then $s_{new} = 0$. Clearly, $\mathcal{A}_i$ is good but no water can be transported out of C_i, if we are to keep $\mathcal{A}_i$ good. It follows $\mathcal{A}_i$ is isolated. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is isolated.

Case II.2: Suppose $s_{old} + \Delta < 0 \Leftrightarrow \Delta < 0$. Then the assignment at line 7 takes place and $s_{new} = \Delta - d_i$, which is a negative amount since $d_i > 0$. Clearly, $\mathcal{A}_i$ is wanting. It becomes good if at least $|\Delta|$ water is transported into C_i; any less amount will keep it wanting. By defintion, $\mathcal{A}_i$ is $|\Delta|$-deficient, *i.e.* $|s_{new} + d_i|$-deficient. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $|s_{new} + d_{i-1}|$-deficient.

Case III: Suppose $s_{old} < 0$. By the induction hypothesis, $\mathcal{A}_{i-1}$ is $|s_{old}+d_{i-1}|$-deficient. That means $\mathcal{A}_{i-1}$ is wanting and unless we deliver at least $|s_{old} + d_{i-1}|$ units of water into

[†]As noted before, $\mathcal{A}_i$ can be both isolated and redundant.

C_{i-1}, it stays wanting. To deliver at least $|s_{old} + d_{i-1}|$ units of water into C_{i-1} means to transport at least $|s_{old} + d_{i-1}| + d_{i-1}$ units of water out of C_i (towards C_{i-1}) in order to compensate for the loss of d_{i-1} units along the way.

We claim that $|s_{old}| > d_{i-1}$. To see why, note that the negative value s_{old} was assigned to s during the previous execution of the **for**-loop at line 7. Now we discuss the previous iteration, so let us call i_{old} the value of the variable i then. In order to reach line 7, it must have been the case that $s + \Delta$ was negative. But $-d_{i_{old}}$ at line 7 was negative, too. It follows the absolute value of what was assigned to s at line 7 was strictly larger than $d_{i_{old}}$. In other words, $|s_{old}| > d_{i_{old}}$. Finally, note that i got incremented by one since the previous iteration, so $i_{old} = i - 1$, with respect to the current i.

Having proved that $|s_{old}| > d_{i-1}$ and having in mind that $s_{old} < 0$, it is obvious that $|s_{old} + d_{i-1}| + d_{i-1} = |s_{old}|$. So, the threshold amount of water to transport out of C_i (towards C_{i-1}) to make $\mathcal{A}_{i-1}$ good is $|s_{old}|$, anything less will keep $\mathcal{A}_{i-1}$ wanting.

Case III.1: Suppose $s_{old} + \Delta \geq 0$. Then the assignment at line 5 takes place.

Case III.1.a: Suppose $s_{old} + \Delta - d_i > 0$. Then $s_{new} = s_{old} + \Delta - d_i$. Since s_{old} is negative and $-d_i$ is negative, it must be the case that Δ is positive; furthermore, it must be the case that $\Delta > |s_{old}| + d_i$. Now we show $\mathcal{A}_i$ is $(s_{old} + \Delta)$-redundant. There are Δ units of water in C_i. Transport $|s_{old}|$ units to C_{i-1}, thus making $\mathcal{A}_{i-1}$ good. That amounts to reducing the water in C_i down to $s_{old} + \Delta$ units (recall that s_{old} is negative so $s_{old} + \Delta$ is smaller than Δ, still being a positive quantity). Since $|s_{old}|$ is a threshold quantity, there is no way to have more than $s_{old} + \Delta$ units in C_i and making $\mathcal{A}_{i-1}$ good.

Since $s_{old} + \Delta$, the remaining quantity in C_i, is nonnegative, and $\mathcal{A}_{i-1}$ is good, $\mathcal{A}_i$ is $(s_{old} + \Delta)$-redundant. In other words, $\mathcal{A}_i$ is $(s_{new} + d_i)$-redundant. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $(s + d_{i-1})$-redundant.

Case III.1.b: Suppose $s_{old} + \Delta - d_t = 0$. Then $s_{new} = 0$. We prove that $\mathcal{A}_i$ is closed. Note that the water in C_i is $\Delta = -s_{old} + d_i$, $-s_{old}$ being a positive amount, and $\mathcal{A}_{i-1}$ is wanting. We transport $|s_{old}|$ units of water from C_i back to C_{t-1}, making $\mathcal{A}_{i-1}$ good. We know that transporting anything less will keep $\mathcal{A}_{i-1}$ wanting. After the transportation C_i will be satisfied, too, having d_i water in it. So, $\mathcal{A}_i$ is good after the transportation. However, it is not possible to make $\mathcal{A}_{i-1}$ good *and* keep C_i satisfied *and* deliver ϵ units of water into C_{i+1}, for any $\epsilon > 0$, using the water in C_i. It follows $\mathcal{A}_i$ is isolated. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is isolated.

Case III.2: Suppose $s_{old} + \Delta < 0$. Then the assignment at line 7 takes place and $s_{new} = s_{old} + \Delta - d_i$, which is a negative amount since $d_i > 0$. We prove $\mathcal{A}_i$ is $|s_{old} + \Delta|$-deficient. Recall that $\mathcal{A}_{i-1}$ is wanting and unless $|s_{old}|$ units are transported into C_{i-1}, it remains waning. The quantity in C_i, *viz.* Δ, may be positive, zero, or negative, we do not know that; what matters is that Δ decremented by $|s_{old}|$, *i.e.* $s_{old} + \Delta$, is negative—that fact implies $\mathcal{A}_i$ is wanting. Furthermore, in order to make $\mathcal{A}_i$ good, the amount of water in C_i must be increased by at least $|s_{old} + \Delta|$, any smaller increase leaves $\mathcal{A}_i$ wanting. To see why that is true, note that under the current assumptions, $\Delta - |s_{old}| + |s_{old} + \Delta| = 0$. And the latter is true since

$$\forall x, y \in \mathbb{R}, \text{ such that } x < 0 \text{ and } x + y < 0, \quad y - |x| + |x + y| = 0$$

Since $|s_{old} + \Delta|$ is the threshold amount to be added to the water in C_i in order to make $\mathcal{A}_i$ good, by definition $\mathcal{A}_i$ is $|s_{old} + \Delta|$-deficient. Clearly, that is equivalent to saying that $\mathcal{A}_i$

is $|s_{new} - d_i|$-deficient. The next time the execution is at line 2, i gets incremented by one. With respect to the new value of i, it is the case that $\mathcal{A}_{i-1}$ is $|s_{new} + d_{i-1}|$-deficient.

Termination. Consider the moment when the execution is at line 2 for the last time. Clearly, i equals $n+1$. If the current s is non-negative then $\mathcal{A}_{i-1} = \mathcal{A}_n$ is either s-redundant (recall then d_n is defined to be zero) or isolated, therefore it is good. Accordingly, the retuned value (line 9) is TRUE. If s is negative then it is wanting. Accordingly, the retuned value (line 11) is FALSE.

The time complexity is obviously $\Theta(n)$. □

Problem 126. *Imagine two rooms with no visibility between them. In one room there n numbered light switches s_1, s_2, ..., s_n. In the other room there are n numbered light bulbs l_1, l_2, ..., l_n. It is known that each switch turns on and off to exactly one bulb but we do not know anything about the wiring between the switches and the bulbs. Initially we are in the room with the switches. Our job is to tell the exact wiring,* i.e. *which switch operates which bulb. We are allowed to press any switches and then go to the room with the bulbs and perform an observation. We are not allowed to touch the bulbs – our only source of information is the observation of the bulbs.*

The switches are such that their physical appearance does not change when toggled so we have no way of knowing beforehand whether pressing a certain switch leads to turning on or turning off of a bulb. Every swtich has, of course, two states only, as any normal light switch.

Describe an algorithm that discovers the wiring with minimum number of observations, i.e. *with minimum visits to the room with the bulbs. The algorithm should work iteratively, at each iteration simulating toggling some switches and then simulating an observation by calling some function* OBSERVE. *The toggling is simulated by writing into a 0-1 array $P[1, \ldots, n]$. Say, $P[i] = 1$ means s_i is toggled, and $P[i] = 0$ means s_i is not toggled. The result of the "observation" is written in some 0-1 array $L[1, \ldots, n]$. Say, $L[i] = 1$ means l_i is on, and $L[i] = 0$ means l_i is off. After every call to* OBSERVE, *the algorithm temporarily halts, the execution is supposed to be transfered to an outside agent and the algorithm resumes after the outside agent finishes writing into L.*

Prove an asymptotic lower bound for the number of observations. Is your algorithm optimal in the asymptotic sense?

Solution:
Our algorithm maintains the following data structures:

- a 0-1 array $A[1, \ldots, n]$ to keep the result of the previous observation,
- an array $S[1, \ldots, n]$ that refers to the switches. Every element of S is a pointer. Namely, $S[i]$ points to a (doubly) linked list that represents the set of the bulbs, each of which can possibly be connected to s_i according to currently available information. We call this set of bulbs, *the candidate set for* s_i.
- an array B of positive integers of size $2n$. During iteration i, B contains $2i$ elements that determine a partitioning of S into subarrays. $B[2j]$ and $B[2j+1]$ are numbers such that $B[2j] \leq B[2j+1]$ and the pair $\langle B[2j], B[2j+1] \rangle$ represents the subarray $S[\, B[2j], \ldots, B[2j+1] \,]$.

- a multitide of doubly linked lists with n elements altogether. They represent a partition of the set of the bulbs. Each element contains an integer that corresponds to the ID of precisely one bulb, and each list represents precisely one candidate set. Initially, there is only one list in this multitude, call this list C. That reflects the fact that at the beginning we have no restrictions on the possible connections between bulbs and switches. At the end, there are n non-empty lists in this multitude. That reflects the fact that at the end we know precisely the wiring between the switches and the bulbs.

Here is the pseudocode. Initially P[ ] is arbitrary.

```
SWITCHES AND BULBS()
 1  create doubly linked list C of n elements, one for each bulb
 2  create S and set every pointer in it to C
 3  B ← [1, n]
 4  OBSERVE()
 5  copy L into A
 6  while the are less than 2n entities in B do
 7      foreach pair ⟨B[2j], B[2j + 1]⟩ such that B[2j] < B[2j + 1]
 8          mid ← (1/2)(B[2j] + B[2j + 1])
 9          set P[ B[2j], ..., mid ] to ones
10          set P[ mid + 1, ..., B[2j + 1] ] to zeros
11      update B so that for each applicable pair ⟨B[2j], B[2j + 1]⟩, it
12            is substituted by two pairs ⟨B[2j], mid⟩ and ⟨mid + 1, B[2j + 1]⟩
13      OBSERVE()
14      for i ← 1 to n
15          if A[i] ≠ L[i]
16              mark bulb i as changed
17          foreach list of bulbs
18              split the list, if necessary, into two lists: changed and unchanged bulbs
19          foreach element of S
20              update the pointer to the relevant list of bulbs
21      copy L into A
22  for i ← 1 to n
23      print the sole element of the list pointed to by S[i]
```

The query complexity of the algorithm, *i.e.* the number of calls of OBSERVE, is the number of executions of the **while** loop plus one. The number of executions of the **while** loop is logarithmic in n because we split each subarray, delineated by a couple from B, roughly in half, with each execution. So, the number of queries is $\Theta(\lg n)$.

Now we prove an $\Omega(\lg n)$ lower bound of the number of such queries. We use the decision tree model. The decision tree model is used, for instance, for proving an $\Omega(n \lg n)$ lower bound for comparison based sortings (see [CLR00]). However, the decision trees for comparison based sortings are binary because there are precisely two possible outcomes of each comparison of the kind $a_i \overset{?}{<} a_j$. In contrast to that, any decision tree that corresponds to the current problem of switches and bulbs has branching factor of 2^n. To why this is true, note that there are precisely 2^n possible outcomes from each observation of the n bulbs.

The current problem is, essentially, computing a permutation, because the mapping from switches to bulbs is a bijection. It follows that any decision tree for that problem has to distinguish all possible $n!$ permutations of n elements: if the decision tree has a leaf labeled by at least two permutations then the corresponding algorithm is not correct. It follows that the leaves must be at least $n!$.

The height of the tree is approximately logarithm to base the branching factor of the number of leaves:

$$\log_{2^n} n! = \frac{\log_2 n!}{\log_2 2^n} = \frac{\Theta(n \lg n)}{n} = \Theta(\lg n)$$

The height of the tree is a lower bound for the query complexity of any observation-based algorithm for the problem of switches and bulbs. It follows that $\Theta(\lg n)$ observations are required if the only testing allowed is direct observation. It follows that algorithm SWITCHES AND BULBS is asymptotically optimal with respect to the number of performed ovservations. □

Problem 127 ([CLR00], Problem 4-2, Finding the missing integer)**.** *An array* $A[1, \ldots, n]$ *contains all the integers from* 0 *to* n *except one. It would be easy to determine the missing integer in* $O(n)$ *time by using an auxiliary array* $B[0, \ldots, n]$ *to record which numbers appear in* A*. In this problem, however, we cannot access an entire integer in* A *with a single operation. The elements of* A *are represented in binary, and the only operation we can use to access them is "fetch the* j*-th bit of* $A[i]$*," which takes constant time. Show that if we use only this operation, we can still determine the missing integer in* $O(n)$ *time.*

Solution:
Let m be the number of bits required to represent n in binary. It is well known that $m = \lfloor \log_2 n \rfloor + 1$. In this problem we think of A as an $m \times n$, 0-1 matrix. Row number m of A consists of the least significant bits of the numbers in A, row number $m - 1$ consists of the second least significant bits, *etc.*, row number 1 consists of the most significant bits. For instance, if $n = 10$ and the missing number is $6 = 0110_{\text{b}}$, A may look like:

$A =$

0	0	1	0	0	0	0	1	1	0
0	0	0	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0	0	1
1	1	0	1	0	1	0	1	0	0

The only constant time access to it is of the form A[j][i], which the authors of [CLR00] call "fetch the j-th bit of A[i]", assuming the first bit is the most significant, *etc.*

Consider the following program in C.

```
int m = floor(logb(n)) + 1;
int A[m][n];

int main() {
  int i, j, t, numrow, n0, n1, ntemp = n;
  int B[n], res[m];
  for(i = 0; i < n; i ++)
    B[i] = i;
```

```
for(i = m - 1; i >= 0; i --) {
  n0 = n1 = 0;
  for(j = 0; j < ntemp; j ++) {
    if (A[i][B[j]] == 0) n0 ++;
    else n1 ++; }
  if(n0 - n1 == 2 || n0 - n1 = 1) res[i] = 1;
  if(n0 - n1 == 0 || n0 - n1 == -1) res[i] = 0;
  for(j = 0, t = 0; j < ntemp; j ++)
    if((res[i] == A[i][B[j]])) {
      B[t] = B[j];
      t ++; }
  if(ntemp % 2 == 0) ntemp = (ntemp / 2);
  else if(res[i] == 0) ntemp = floor(ntemp / 2);
  else ntemp = ceil(ntemp / 2);
}
for(i = 0; i < n; i ++)
  printf("%d", res[i]);
}
```

We claim the algorithm implemented by this program solves correctly the problem of determining the missing bit. First we prove it is correct.

Define that a *complete array* of size n is a two dimensional bit array similar to the above A but without any missing column from it. Clearly, such an array has $n+1$ columns. For instance, a complete array of size 10 would be the following:

0	0	1	0	0	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	1
1	1	0	1	0	1	0	1	0	0	0

Define that an *almost complete array* of size n is a two dimensional bit array with precisely one missing column from it[†]. Now consider any complete array $\tilde{A}$ of size n. Call $\tilde{L}$ the bottom row of $\tilde{A}$. That $\tilde{L}$ consists of the least significant bits of the numbers fron $\tilde{A}$. Let $\tilde{n_0}$ be the number of zeros and $\tilde{n_1}$, the number of ones, in $\tilde{L}$. Let $\tilde{\Delta} = \tilde{n_0} - \tilde{n_1}$. We claim that:

$$\tilde{\Delta} = \begin{cases} 0, & \text{if } n \text{ is odd} \\ 1, & \text{if } n \text{ is even} \end{cases}$$

Indeed, it is trivial to prove by induction that if the number $n+1$ of columns in $\tilde{A}$ is even then $\tilde{\Delta} = 0$ and if it is odd, $\tilde{\Delta} = 1$.

Now consider A: any almost complete array of size n, obtained from $\tilde{A}$ by deleting a column, *i.e.*, the missing number. Let L be the bottom row of A. Let n_0 be the number of zeros and n_1, the number of ones, in L. Let $\Delta = n_0 - n_1$. We claim that:

$$\Delta = \begin{cases} \tilde{\Delta} + 1, & \text{if the missing number is odd} \\ \tilde{\Delta} - 1, & \text{if the missing number is even} \end{cases}$$

[†]It follows the array A in the current problem is an almost complete array of size n.

Indeed, if the missing number is even there is a 0 less in L in comparison with $\tilde{L}$, while the number of ones is the same; that is, $n_0 = \tilde{n_0} - 1$ and $n_1 = \tilde{n_1}$. Likewise, if the missing number is odd there is a 1 less in L in comparison with $\tilde{L}$, while the number of zeros is the same; that is, $n_0 = \tilde{n_0}$ and $n_1 = \tilde{n_1} - 1$. Having in mind the above considerations, it is clear that:

$$\Delta = \begin{cases} 2, & \text{if } n \text{ is even and the missing number is odd} \\ 1, & \text{if } n \text{ is odd and the missing number is odd} \\ 0, & \text{if } n \text{ is even and the missing number is even} \\ -1, & \text{if } n \text{ is odd and the missing number is even} \end{cases}$$

We conclude that:

$$\Delta \in \{1, 2\} \Rightarrow \text{the least significant bit of the missing number is } 1 \tag{6.1}$$

$$\Delta \in \{-1, 0\} \Rightarrow \text{the least significant bit of the missing number is } 0 \tag{6.2}$$

So, with one linear scan along the bottom row of A we can compute Δ and then in constant time we can compute the least significant bit of the missing number. However, if we attempt a similar approach for the other bits of the missing number, we will end up with $\Omega(n \lg n)$ computation because the number of rows is logarithmic in n. The key observation is that in order to derermine the second least significant bit of the missing number, we need to scan approximately half the columns of A. Namely, if the least significant bit was determined to be 1, for the computation of the second least significant bit we need to scan only the columns having 1 at the bottom row. Likewise, if the least significant bit was determined to be 0, for the computation of the second least significant bit we need to scan only the columns having 0 at the bottom row. Next we explain why this is true.

The number of rows of A is $m = \lfloor \log_2 n \rfloor + 1$. Define that $A_0^{(m-1)}$ is the two dimensional array obtained from A by deleting the columns that have 0 in row $m-1$, and then deleting row $m-1$. Define that $A_1^{(m-1)}$ is the two dimensional array obtained from A by deleting the columns that have 1 in row $m-1$, and then deleting row $m-1$. Call the process of deriving $A_0^{(m-1)}$ and $A_1^{(m-1)}$, *the reduction of* A, and the two obtained arrays, *the reduced arrays*. Let b be the least significant bit of the missing number and $\overline{b}$ be its complement.

Lemma 26. *Under the current naming conventions,* $A_b^{(m-1)}$ *is complete, and* $A_{\overline{b}}^{(m-1)}$ *is almost complete. Furthermore, the missing number in* $A_{\overline{b}}^{(m-1)}$ *is obtained from the missing number in* A *by removing the least significant bit (*i.e.*, shift right).*

Proof:
First we will see an example and then make a formal proof. To use the previously given example with $n = 10$, $m = 4$, missing number $6 = 0110_b$, and

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

the two derived subarrays are

$A_0^{(3)} =$ | 0 | 0 | 0 | 0 | 1 |
| --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |

$A_1^{(3)} =$ | 1 | 0 | 0 | 1 | 0 |
| --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |

Obviously, $A_0^{(3)}$ is complete and $A_1^{(3)}$ is almost complete: column $\begin{matrix}0\\1\\1\end{matrix}$ is missing from it. The least significant bit of the missing number in A is 0; if we did not know which is the missing number, we could deduce that its least significant bit is 0 by computing the aforementioned $\Delta = 5 - 5 = 0$. Indeed $A_{\bar{0}}^{(3)} = A_1^{(3)}$ is the array that is almost complete. And indeed the missing number in it 011 is obtained from 0110 by shift right.

Let us prove the lemma. It is clear that if $\tilde{A}$ is a complete array of size n, both $\tilde{A}_0^{(m-1)}$ and $\tilde{A}_1^{(m-1)}$ are complete. If n is odd then they contain the same columns (possibly in different order left to right), otherwise $\tilde{A}_1^{(m-1)}$ contains one column more and the other columns are the same. Now imagine A—the array obtained from $\tilde{A}$ by the deletion of precisely one column. If the bit at the bottom row of the deleted column is 0 then $A_0^{(m-1)}$ is the same as $\tilde{A}_0^{(m-1)}$, so $A_0^{(m-1)}$ is complete. However, $A_1^{(m-1)}$ is not the same as $\tilde{A}_1^{(m-1)}$: $\tilde{A}_1^{(m-1)}$ contains one more column that corresponds to the missing number. It follows that $A_1^{(m-1)}$ is almost complete. Alternatively, if the bit at the bottom row of the deleted column is 1 then $A_1^{(m-1)}$ is the same as $\tilde{A}_1^{(m-1)}$, so $A_1^{(m-1)}$ is complete, but $A_0^{(m-1)}$ is almost complete. That concludes the proof of the lemma. □

Having all that in mind, the verification of the algorithm is straightforward. `m` is the number of bits, *i.e.* the number of rows of A. The array `res` is the output: at each iteration of the main **for** loop, one bit of `res` is computed, the direction being from the least significant bit "upwards". `B` is an auxilliary array of integers. The definition of the problem requires bitwise access only to the elements of `A`; the array `B` can be accessed "normally". `B` keeps the indices of the columns whose `i`-th row we scan at every iteration of the main **for** loop. Initially, of course, `B` contains the indices `0`, `1`, ..., `n-1`, in that order, so when `i` is `m-1` we simply scan the bottom row of `A`. At every iteration of the main **for** loop, `ntemp` is the number of columns in the almost complete array whose last row we scan. Initially, `ntemp` is `n`, which reflects the fact that at the first iteration we scan the bottom row of `A`. We will verify the assignment of new value to `ntemp` later on.

Within the main **for** loop, the first nested **for** loop simply counts the zeros and ones and stores the results in `n0` and `n1`, respectively.

The difference `n0 - n1` determines the `i`-th least significant bit `res[i]` according to 6.1 and 6.2.

The second nested **for** loop discovers the indices of the columns that correspond to the columns of the next reduced array that is almost complete. To see why we consider only values (of the last row of `A`) equal to `res[i]`, check the above Lemma.

Finally, the asignment of new value to `ntemp` is done in accordance to the following considerations. If `ntemp` is even then both derived arrays have the same length `ntemp / 2`. Otherwise, note that we are interested in that derived subarray that is almost complete. If `res[i]` is one then that subarray is the one obtained by deleting the columns with zeros at the bottom; it has one more column than the complete derived subarray, so `res[i]` should be `ceil(ntemp / 2)`. Analogously, if `res[i]` is zero then `res[i]` should be `floor(ntemp`

`/ 2)`. That concludes the verification of the algorithm. The reader is invited to make an even more rigorous proof of correctness using loop invariant.

The number of accesses to `A` at each iteration of the main **for** loop is proportional to the current value of `ntemp`. Clearly, the total number of accesses is proportional to

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \leq 2n = \Theta(n)$$

□

Problem 128. *Consider Problem 127 under the additional assumption that the numbers in* A*, that is, the columns, appear in sorted order. Find the missing number with* $O(\lg n)$ *bit accesses to* A*.*

Solution:
If the numbers in A are sorted the problem can be solved by first determining the most significant bit, then the second most significant bit, *etc.*, the least significant bit, of the missing number, with precisely one access to A for each bit.

Suppose $\tilde{A}$ is the complete array of size n (see the definition of "complete array" in the solution to Problem 127), *i.e.* there is no missing number. For instance, if $n = 10$ then $\tilde{A}$ is:

$\tilde{A} =$

0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	0	0	1	1	0	0	1
0	1	0	1	0	1	0	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

Consider the boundary on the top row between the zeros and the ones. The rightmost zero is in column 7 ($7 = 2^{\lfloor \log_2 10 \rfloor} - 1$) and the leftmost one is in column 8 ($8 = 2^{\lfloor \log_2 10 \rfloor}$). It is easy to generalise that the boundary is between columns $2^{\lfloor \log_2 n \rfloor} - 1$ and $2^{\lfloor \log_2 n \rfloor}$, provided the leftmost column is number 0.

Now consider the boundary on the top row between the zeros and the ones in an almost complete array A of size n. For instance, if $n = 10$ and the missing number is $6 = 0110_b$ then A is:

$A =$

0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	0	0	0
0	0	1	1	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Consider positions $7 = 2^{\lfloor \log_2 10 \rfloor} - 1$ and $8 = 2^{\lfloor \log_2 10 \rfloor}$ on the top row. Now the boundary is not between them because the missing number has most significant bit 0, so the boundary is "shifted" one position to the left in comparison with $\tilde{A}$. However, if we do not know what the missing number's most significant bit is, we can deduce it is 0 from the fact that there are two 1's at positions 7 and 8 on the top row.

Clearly, if there were 0 and 1 at positions 7 and 8, respectively, on the top row, that would mean the missing number's most significant bit is 1, as in the following example where the array is called B, $n = 10$ and the missing number is $9 = 1001_b$:

$B =$

0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	1	1	1	0	0
0	0	1	1	0	0	1	1	0	1
0	1	0	1	0	1	0	1	0	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

So, by inspecting only position $2^{\lfloor \log_2 n \rfloor} - 1$ on the top row, we can deduce the most significant bit of the missing number as follows:

$$\text{the missing number's most significant bit } = \begin{cases} 0, & \text{if } A[0][2^{\lfloor \log_2 n \rfloor} - 1] = 1 \\ 1, & \text{if } A[0][2^{\lfloor \log_2 n \rfloor} - 1] = 0 \end{cases}$$

Having computed the most significant bit of the missing number in $\Theta(1)$ time, we compute the second most significant bit in $\Theta(1)$ time, *etc.*, until we compute all bits of the missing number with $\Theta(\lg n)$ attempts, each taking $\Theta(1)$ time.

Case I: If the most significant bit is 0, to compute the second most significant bit we consider only the subarray of columns $0, 1, \ldots, 2^{\lfloor \log_2 n \rfloor} - 2$ and rows $1, 2, \ldots, \lfloor \log_2 n \rfloor$. Using the above A as an example, that subarray is, say, A′:

$A =$

0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	0	0	0
0	0	1	1	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$\longmapsto$ $A' =$

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*

We proceed recursively with A′ exactly as with A because A′ is an almost complete array. Let us compute the size of A′. The initial A can be of any size but the size of A′ is uniquely determined by n. Let $m = \lfloor \log_2 n \rfloor + 1$, that is, the number of bits necessary to represent n in binary (thus m is the number of rows in A). Let $m' = m - 1$. The derived A′ is of size $2^{m'} - 1$. The -1 comes from the fact that A′ has a missing number.

Case II: If the most significant bit is 1, to compute the second most significant bit we consider only the subarray whose columns have 1's in the top row. Using the above B as an example, that subarray is, say, B′:

$B =$

0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	1	1	1	0	0
0	0	1	1	0	0	1	1	0	1
0	1	0	1	0	1	0	1	0	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$\longmapsto$ $B' =$

0	0
0	1
0	0
0	*1*

The number of columns, call it c′, in B′ is easy to compute: it is $c' = n - 2^{m'}$ where $m' = \lfloor \log_2 n \rfloor$. In the concrete example, $n = 10$, $m' = 3$, thus $c' = n - 2^{m'} = 10 - 8 = 2$.

However, B' is not necessarily an almost complete array—in order to be an almost complete array it has to have at least one 1 at its top row. In fact, B' is an almost complete array only when $c' > \frac{1}{2}(2^{m'}) = 2^{m'-1}$. In this example, B' is not an almost complete array, it has one row too many. We can conlude the second most significant bit of the missing number is 0 (the missing number is $9 = 1001_b$) just by knowing the dimensions of B'; we do not have to scan, or even examine bits of, the top row to make that conclusion. The rule is, while $c' \leq 2^{m'-1}$, write 0's to into the missing number's bit positions and perform $m' \leftarrow m' - 1$. This process is equivalent to removing the necessary number of top rows from B'. Once the process is over and B' is reduced as necessary, it can be dealt with recursively.

Consider the following program in C.

```
int m = floor(logb(n)) + 1;
int A[m][n], res[m];
void find(int, int, int) ;

int main() {
  find(0, n-1, 0);
  return 0; }

void find(int low, int high, int row) {
  int j, c, n1, n2, numel = high - low + 1;
  if(row == m-1) {
    res[row] = !(A[i][0]);
    PrintResult();
    return; }
  n1 = floor(logb(numel));
  n2 = 1 << n1;
  if(A[row][low+n2-1] == 1) {
    res[row] = 0;
    find(low, n2-2, row+1); }
  else {
    j = 1;
    res[row] = 1;
    c = numel - n2;
    while((n1 >= 0) & & c <= (1<<(--n1))) {
      j ++;
      res[row+j] = 0; }
    if(row+j == m-1) {
      PrintResult();
      return; }
    find(n2, high, row+j); } }
```

The correctness of the fragment follows from the previous discussion. The time complexity is obviously $\Theta(\lg n)$. □

Problem 129. A circular array $A[1, \ldots, n]$ *is an array such that* $n \geq 3$ *and* $A[1]$ *and* $A[n]$ *are considered to be adjacent elements just like* $A[1]$ *and* $A[2]$, $A[2]$ *and* $A[3]$, etc.,

are adjacent. We are given a circular array $A[1,\ldots,n]$ *of nonnegative integers. For any* $i,j \in \{1,2,\ldots,n\}$ *such that* $i \neq j$, *dist*$(i,j) = \max\{|i-j|, n-|i-j|\}$. *Design a linear time algorithm that computes a maximum number* t *such that for some* $i,j \in \{1,2,\ldots,n\}, i \neq j$, $t = A[i] + A[j] + dist(i,j)$.

Solution:
Consider the following algorithm, due to Mugurel Ionuţ Andreica [MAMc].

CIRCULAR ARRAY($A[1,\ldots,n]$: circular array of nonnegative integers)
1 let $B[0\ldots n]$ and $C[1\ldots n]$ be linear arrays of nonnegative integers
2 $B[0] \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** n
4 $\quad B[i] \leftarrow \max\{B[i-1], A[i] - (i-1)\}$
5 $\quad C[i] \leftarrow B[i-1] + A[i] + (i-1)$
6 $x \leftarrow \max\{C[i] \mid 1 \leq i \leq n\}$
7 **for** $i \leftarrow 1$ **to** n
8 $\quad B[i] \leftarrow \max\{B[i-1], A[i] + (i-1)\}$
9 $C[n] \leftarrow A[n] + 1$
10 **for** $i \leftarrow n-1$ **downto** 2
11 $\quad C[i] \leftarrow \max\{C[i+1], A[i] + n - (i-1)\}$
12 $y \leftarrow \max\{B[i] + C[i+1] \mid 1 \leq i \leq n-1\}$
13 **return** $\max\{x, y\}$

It is obvious that the time complexity is $\Theta(n)$. Now we prove the correctness.

Lemma 27. *Whenever the execution of the first* ***for*** *loop (lines 3–5) of* CIRCULAR ARRAY *is at line 5 and* $i \geq 2$, $C[i]$ *is assigned* $\max\{A[k] + A[i] + i - k \mid 1 \leq k < i\}$.

Proof:
It is fairly obvious that at line 5 the value $B[i-1]$ is such that

$$B[i-1] = \begin{cases} 0, \text{ if } A[k] - (k-1) \leq 0 \ \ \forall k \text{ such that } 1 \leq k < i \\ \max\{A[k] - (k-1) \mid 1 \leq k < i\}, \text{ else} \end{cases}$$

However, $A[1] - (1-1)$ cannot be negative, therefore there is at least one non-negative value in the sequence $A[k] - (k-1), 1 \leq k < i$, so we can say simply that $B[i-1]$ at line 5 is $B[i-1] = \max\{A[k] - k + 1 \mid 1 \leq k < i\}$. It follows that $C[i]$ is assigned the value $\max\{A[k] - k + 1 \mid 1 \leq k < i\} + A[i] + i - 1 = \max\{A[k] + A[i] + i - k \mid 1 \leq k < i\}$. □

It follows that x is assigned the value $\max\{A[i] + A[j] + j - i \mid 1 \leq i < j \leq n\}$ at line 6 of CIRCULAR ARRAY.

Lemma 28. y *is assigned the value* $\max\{A[i] + A[j] + n - (j-i) \mid 1 \leq i < j \leq n\}$ *at line 12 of* CIRCULAR ARRAY.

Proof:
Consider the second **for** loop (lines 7–8). Since $A[1] + (1-1) \geq 0$, it is the case that $\forall i, 1 \leq i \leq n, B[i] = \max\{A[k] + (k-1) \mid 1 \leq k \leq i\}$ after the second **for** loop terminates. Now consider the third **for** loop at lines 10–11. Think of the assignment at line 9 as $C[n] = A[n] + n - (n-1)$. Having that in mind, it is fairly obvious that after that **for**

loop terminates, it is the case that $C[i] = \max\{A[k] + n - (k-1) \mid i \le k \le n\}$, $\forall i, 2 \le i \le n$ From these two considerations it follows immediately that at line 12, y is assigned the value

$$\max\{A[i] + (i-1) + A[j] + n - (j-1) \mid 1 \le i < j \le n\} =$$
$$\max\{A[i] + A[j] + n - (j-i) \mid 1 \le i < j \le n\}$$

□

It follows immediately that CIRCULAR ARRAY indeed returns the maximum number t such that for some $i, j \in \{1, 2, \ldots, n\}, i \ne j$, $t = A[i] + A[j] + \text{dist}(i, j)$. □

Problem 130 ([CLR00], Problem 10.3-8)**.** *Let* $X[1, \ldots, n]$ *and* $Y[1, \ldots, n]$ *be two arrays, each containing* n *numbers already in sorted order. Give an* $O(\lg n)$*-time algorithm to find the median of all* $2n$ *elements in arrays* X *and* Y*.*

Solution:
Assume that when n is even the median of X is $X\left[\frac{n}{2} + 1\right]$. If the arrays are of equal size, and that is the current case, we can solve the problem by a divide and conquer algorithm that compares the medians of the two arrays and then discards the lower half of the array with the smaller median and the upper half of the array with the bigger median. The algorithm proceeds likewise until both arrays are reduced to 2 elements each. Then we solve the reduced problem in constant time. In case the size is odd, by upper and lower half we mean, the subarray from one end until and excluding the median. It is easy to show this dichotomy brings the size of the array down to 2 regardless of what the initial n is, because the iterator $n \to \left\lceil \frac{n}{2} \right\rceil$ reaches 2 regardless of the starting value of n.

Now consider a more general version of this problem where the arrays are $X[1, \ldots, p]$ and $Y[1, \ldots, q]$ for possibly unequal values of p and q. The following solution is based on [LD05]. Let us call Z the array that would be obtained if we merged X and Y. Let $m = p + q$. The essence is the fact that we can check in $\Theta(1)$ time whether $X[i]$ is the median of Z, for any i such that $1 \le i \le p$. According to our definition of median, the median is greater than or equal to $\left\lfloor \frac{m}{2} \right\rfloor$ elements of an m-element array. Having that in mind, clearly if $X[i]$ is the median then:

- $X[i]$ is greater than or equal to $i - 1$ elements of X.
- $X[i]$ is greater than or equal to $j = \left\lfloor \frac{m}{2} \right\rfloor - i + 1$ elements of Y.

It takes only constant time to check if $Y[j] \le X[i] \le Y[j+1]$[†] If that is fulfilled we have found the median and it is $X[i]$. Otherwise, we binary search in X to see if the median is in X. If that fails, the median must be from Y, and we can repeat the analogous process with X and Y swapped.

COMMON MEDIAN($X[1, \ldots, p]$, $Y[1, \ldots, q]$: sorted arrays)
1 $m \leftarrow p + q$
2 $k \leftarrow$ MEDIAN BIN SEARCH($X, Y, 1, p$)
3 **if** $k > 0$

[†]To avoid excessive boundary checks, pad X and Y at the left side with $-\infty$ and with ∞ at the right side.

```
4       return X, k
5   k ← MEDIAN BIN SEARCH(Y, X, 1, q)
6   return Y, k
```

```
MEDIAN BIN SEARCH(A, B: sorted arrays, l, r: integers)
1   if l > r
2       return −1
3   i ← ⌊(l+r)/2⌋
4   j ← ⌊m/2⌋ − i + 1
5   if B[j] ≤ A[i] ≤ B[j + 1]
6       return i
7   if A[i] < B[j]
8       MEDIAN BIN SEARCH(A, B, l, i)
9   if A[i] > B[j + 1]
10      MEDIAN BIN SEARCH(A, B, i + 1, r)
```

A not too formal proof of correctness of COMMON MEDIAN is simply pointing out the preceding discussion and knowing that the binary search idea is correct. The time complexity is obviously $\Theta(\lg m)$. Alternatively, we can say the complexity is $\Theta(\max\{\lg p, \lg q\})$. □

6.3 Graphs

Whenever we say "graph" without any qualifiers, we mean undirected graph without loops and without edge weights. The edges of graphs are denoted as, for example (u, v), although typically parentheses denote ordered pairs and in undirected graphs the edges are in fact vertex sets of size two. Whenever we say "weighted graph" we mean that the edges have positive weights and the vertices, no weights. Unless otherwise specified, n is the number of vertices of the graph under consideration and m is the number of its edges. If G is a graph, we denote its vertex set by $V(G)$ and its edge set, by $E(G)$. $\text{adj}(u)$ denotes the adjacency list of vertex u. By $\overline{G}$ we denote its complement: $\overline{G} = (V, \overline{E})$, where $\overline{E} = \mathcal{V}_2 \setminus E$, where $\mathcal{V}_2 = \{X \in 2^V \mid |X| = 2\}$.

To *delete* a vertex v from a graph $G(V, E)$ means to delete v from V and to delete all edges with one endpoint v from E. The vertex deletion operation is denoted by $G - v$. To *remove* an edge e from a graph $G(V, E)$ means to delete e from E without deleting its endpoints from V. The edge deletion operation is denoted by $G - e$. To *add* an edge $e' = (u, v)$ to G means that $(u, v) \notin E$ and then the operation $E \leftarrow E \cup \{(u, v)\}$ is performed. The edge addition operation is denoted by $G + e'$. To delete a subset $U \in V$ from V means to delete all vertices from U and all edges with at least one endpoint in U.

By "path" we mean a simple path, *i.e.* without repeating vertices. Likewise, by "cycle" we mean a simple cycle. The *degree* of a vertex u in undirected graph G is denoted by $\deg(u)$ and is defined as $\deg(u) = |\text{adj}(u)|$. If u is a vertex in multiple graphs, we write $\deg_G(u)$ to emphasise we mean the degree of u in G.

6.3.1 Graph traversal related algorithms

Definition 8. *Let* $G(V, E, w)$ *be a weighted connected graph. The* eccentricity *of any vertex* $v \in V$ *is* $ecc(v) = \max\{dist(v, u) \mid u \in V \setminus \{v\}\}$. *The* diameter *of* G *is* $diam(G) = \max\{ecc(v) \mid v \in V\}$. □

The term "diameter" is overloaded, meaning either the maximum eccentricity, or any path of such length whose endpoints are two vertices of maximum eccentricity. We encourage the reader to have in mind that diameter is completely different from longest path: the diameter is the longest one among the shortest paths between any two vertices in the graph, while the longest path is the longest one among the longest paths between any two vertices in the graph. As an extreme example, consider the complete graph K_n (with edge weights ones). Its diameter is 1 because every vertex is connected to every other vertex but its longest path is of length $n - 1$ because K_n is Hamiltonian. A notable exception are trees. In any tree, "diameter" and "longest path" are the same thing, *i.e.* paths of the same length. To see why, note that in trees there is a unique path between any two vertices. Therefore, the longest path between any two vertices u and v has the same length as the shortest path between u and v, because there is only one such path to begin with.

A *cut vertex* in a graph G with k connected components is any vertex $u \in V(G)$ such that the deletion of u leads to graph G' with $\geq k + 1$ connected components. A *bridge* in a graph G with k connected components is any edge $e \in V(G)$ such that the deletion of u leads to graph G' with $k + 1$ connected components.

Now we present the well known algorithm DFS for graph traversal, in the version of Cormen *et al.* [CLR00].

```
DFS(G(V, E): directed graph)
 1  foreach u ∈ V
 2      color[u] ← WHITE
 3      π[u] ← NIL
 4  time ← 0
 5  foreach u ∈ V
 6      if color[u] = WHITE
 7          DFS VISIT(G, u)
```

```
DFS VISIT(G(V, E): directed graph, u: vertex from V)
 1  color[u] ← GRAY
 2  time ← time + 1
 3  d[u] ← time
 4  foreach v ∈ adj(u)
 5      if color[v] = WHITE
 6          π[v] ← u
 7          DFS VISIT(G, v)
 8  color[u] ← BLACK
 9  time ← time + 1
10  f[u] ← time
```

It is well known that DFS on undirected graphs partitions the edges into *tree edges* and *back edges* according to the following rules: if the colour of v at line 5 is WHITE then (u, v) is a tree edge and if the colour of v at line 5 is GRAY then (u, v) is a back edge. Also, it is known it is not possible the said colour to be BLACK, so no other type of edge is possible in undirected graphs.

Problem 131. *Design a fast algorithm to compute the diameter of weighted tree. Analyse its correctness and time complexity.*

Solution:
We use a modified DFS as follows. The original DFS on the preceding page works on nonweighted graphs. Assume the input graph is weighted and connected. Let the algorithm use an additional array dist$[1, \ldots, n]$. Consider the following modification of DFS VISIT that does not use $d[\,]$, $f[\,]$, and the variable time.

```
ECCENTRICITY(T(V, E, w): weighted tree, u: vertex from V)
 1  (* Returns an ordered pair ⟨α, β⟩ where α = ecc(u) and *)
 2  (* β is a vertex at distance α from u. *)
 3  foreach x ∈ V
 4      color[x] ← WHITE
 5      π[x] ← NIL
 6      dist[x] ← 0
 7  ECC1(T, u)
 8  α ← max{dist[x] | x ∈ V}
 9  β ← any x ∈ V such that dist[x] = α
10  return ⟨α, β⟩
```

```
ECC1(T(V, E, w): weighted tree, u: vertex from V)
1  color[u] ← GRAY
2  foreach v ∈ adj(u)
3      if color[v] = WHITE
4          π[v] ← u
5          dist[v] ← dist[u] + w(u, v)
6          ECC1(T, v)
7  color[u] ← BLACK
```

It is trivial to prove by induction that after the call at line 7 in ECCENTRICITY, for every vertex $x \in V$, dist[x] contains the distances betwen x and u in T. Using the definition of ecc(u), conclude that ECCENTRICITY returns the eccentricity of u and a vertex that is at that distance from u. ECCENTRICITY has the time complexity of DFS and that is $\Theta(m+n)$.

Lemma 29. *Let* G *be a connected graph, weighted or non-weighted. Any two paths of maximum length in* G *share a vertex.*

Proof:
Assume there are two paths of maximum length that are independent. It is not difficult to show there is a path of even greater length, contrary to the assumption just made. We leave the details to the reader. □

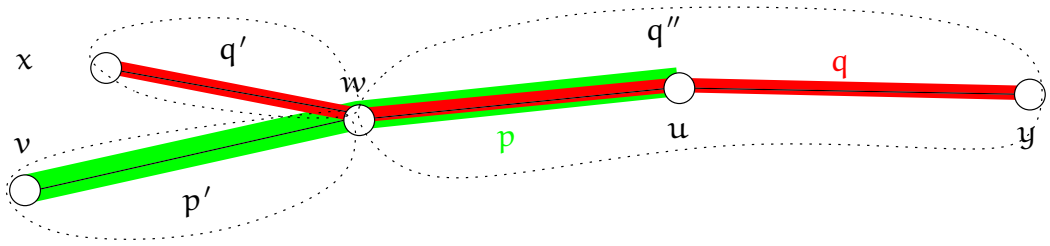


Figure 6.2: The paths p, q, p', and p'' under the assumption that u is an internal vertex of the backbone T'.

Corollary 2. *Let T be a tree, weighted or non-weighted. Any two diameters in T share a vertex.* □

Corollary 3. *Let T be a tree, weighted or non-weighted. Let V' be the union of the vertex sets of all diameters in T. V' induces a subtree T' of T. Every leaf of T' is a leaf of T, too.* □

We call the subtree T' from Corollary 3, *the backbone of T*. Note that T can coincide with its backbone, *e.g.* if T is a star.

Lemma 30. *Let T be a tree, weighted or non-weighted. Let T' be the backbone of T. For any vertex $\mathsf{u} \in \mathsf{V}(\mathsf{T})$, the eccentricity of u is the length of a path p such that one endpoint of p is u and the other endpoint, call it v, of p is some leaf of T'.*

Proof:
Assume the opposite. First assume u is a leaf of T'. By the definition of backbone, all vertices at maximum distance from u are endpoints of some diameters in T, *i.e.* they are leaves of T, hence the contradiction. Assume u is an internal vertex of T'. Then there is a diameter q of T such that u is an internal vertex in q. Having assumed that v is not a leaf of T', it follows v is not a vertex of q. But p and q share at least one vertex, namely u. Let the maximum contiguous subsequence (subpath) of p and q be $\mathsf{u}, \ldots, \mathsf{w}$. Let p' be the subpath $\mathsf{w}, \ldots, \mathsf{v}$ of p. Let q' be the subpath $\mathsf{w}, \ldots, \mathsf{x}$ of q such that x is am endpoint of q and u is not an internal vertex of q'. Let q'' be the subpath of q from w to y where y is the other endpoint of q (*i.e.*, not x). It must be case that $|\mathsf{p}'| > |\mathsf{q}'|$ according to the current assumptions. It follows that using p' and q'' we can construct a path in T that is longer than the diameter q, contrary to the fact that q is a longest path in T. See Figure 6.2.

Finally, assume u is not in T'. It is clear there exists a unique vertex z in T' such that, for every other vertex a from T', z is an internal vertex in the path between a and u. The proof reduces to the previous one with z instead of u. See Figure 6.3. □

Having in mind Lemma 30 and the correctness of algorithm Eccentricity, it is obvious that Eccentricity returns a vertex that is a leaf of the backbone of T. Now consider the following algorithm.

```
Diameter of Tree(T(V, E, w): weighted tree)
1  u ← arbitrary vertex from V
2  ⟨x, y⟩ ← Eccentricity(T, u)
3  ⟨w, z⟩ ← Eccentricity(T, y)
4  return ⟨w, z⟩
```

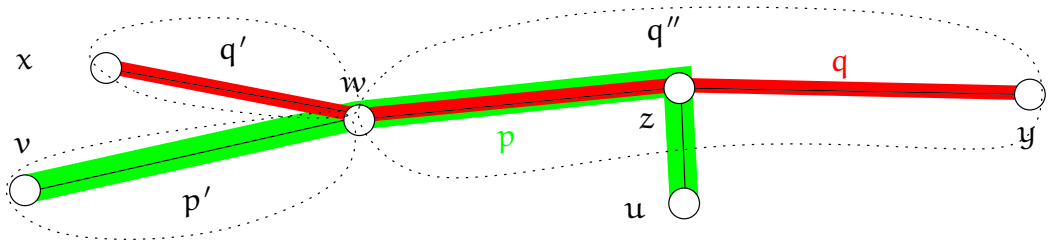


Figure 6.3: The case when u is not in the backbone T'.

Using tha names of DIAMETER OF TREE, y is a leaf of the backbone of T. Then the second call of ECCENTRICITY, namely ECCENTRICITY(T, y), returns $\langle w, z\rangle$ such that z is another leaf of the backbone, one that is at maximum distance from y, and w is the distance between them, *i.e.* the diameter. That proves the correctness of the algorithm. Clearly, the time complexity is $\Theta(n + m)$. □

Problem 132. *Design a fast algorithm to output all cut vertices of a connected graph. Analyse its correctness and time complexity.*

Solution:
The solution is based on the solution in [Ski08, Section 5.9.2, pp. 173–177]. The main idea is to modify DFS and thus to use its optimal time complexity.

Each execution of DFS on a connected graph $G(V, E)$ generates a tree T. $V(T) = V$ and $E(T) = \{e \in E \mid e$ is classified as a tree edge by DFS.$\}$. Clearly, the leaves of T cannot be cut vertices. To see why, consider any vertex $v \in V$ that is a leaf of T. If $\deg_G(v) = 1$ then the deletion of u from G leads to one connected component because such a vertex connects only itself to the remainder of the graph. If $\deg_G(v) \geq 2$, the tree edges are sufficient to hold the remainder of the graph together.

The root r of the tree may or may not be a cut vertex of G: if $\deg_T(r) = 1$ then r is not a cut vertex, otherwise it is a cut vertex.

Each of the remaining vertices may or may not be a cut vertex according to the following lemma.

Lemma 31. *Using the above names, for every vertex u that is not the root and is not a leaf of T, u is a cut vertex iff there exists a child v of u in T such that, if T_v is the subtree of T rooted at v, there is no back edge from any vertex from T_v to a proper ancestor of u.*

Proof:
Suppose there is no back edge from a vertex from $V(T_v)$ to a proper ancestor of u. Then for every path p with endpoints x and y such that $x \in V(T_v)$ and $y \in V(G) \setminus (V(T_v) \cup \{u\})$, u is an internal vertex in p. So, u is a cut vertex in G. Suppose the opposite: there is a back edge from a vertex from $V(T_v)$ to a proper ancestor of u. Then there exists a path p with endpoints x and y such that $x \in V(T_v)$ and $y \in V(G) \setminus (V(T_v) \cup \{u\})$, such that u is not in p. So, u is not a cut vertex in G. □

The algorithm that implements these ideas is a modifed DFS. The arrays $d[\,]$, $f[\,]$, and $\pi[\,]$, and the variable "time" are not used. There is, however, an array level$[1, \ldots, n]$ of natural numbers that keeps record of the levels of the vertices in the DFS tree. That is, level[i] means the distance between vertex i and the root r. Obviously, the level of the root is 0.

```
FIND CUT VERTICES(G(V,E): undirected graph)
 1  foreach u ∈ V
 2      color[u] ← WHITE
 3  let u be an arbitrary vertex from V
 4  FCV1(G,u,0);


FCV1(G(V,E): undirected graph, u: vertex from V, l: integer)
 1  color[u] ← GRAY
 2  level[u] ← l
 3  minback ← level[u]
 4  if level[u] = 0
 5      count ← 0
 6  IsCut ← FALSE
 7  foreach v ∈ adj(u)
 8      if color[v] = WHITE {
 9          if level[u] = 0
10              count ← count + 1
11          x ← FCV1(G,v,l+1)
12          if x ≥ level[u] and level[u] ≥ 1
13              IsCut ← TRUE
14          minback ← min{minback,x} }
15      if color[v] = GRAY {
16          if level[v] < minback and level[v] ≠ level[u] − 1
17              minback ← level[v] }
18  if IsCut
19      print u
20  if level[u] = 0 and count ≥ 2
21      print u
22  color[u] ← BLACK
23  return minback
```

We argue that the printing of cut vertices (line 19 or line 21) is correct. First, recall that the root vertex is treated differently: the root, *i.e.* the starting vertex of the DFS, is a cut vertex iff there are at least two tree edges incident to it. The number of tree edges incident to the root is recorded in the `count` variable[†]. It is incremented at line 10 precisely when the current u is 0, *i.e.* u is the root, and v is WHITE, *i.e.* (u,v) is a tree egde. It follows that after the **for** loop (lines 7–17) finishes, `count` ≥ 2 iff the current u is the root of the DFS tree and it is indeed a cut vertex, and so the printing at line 21 is correct.

On the other hand, line 13 is reached iff

- vertex u is not the root, because level[u] ≥ 1 implies that, and
- there is no back edge from any vertex from T_v to a proper ancestor of u, because at line 11, x is assigned the number of the lowest level proper ancestor of v that is

[†]We emphasise that the number of tree edges incident to that vertex is completely different from the degree of that vertex in G.

incident to a vertex from T_v; if that number is $\geq$ level[u] that vertex must be u or a vertex from T_v, so by Lemma 31, u is a cut vertex.

Those two conditions imply u is a cut vertex. Of course, in order to give a complete formal proof one has to prove by induction that FCV1 returns the number of the lowest level proper ancestor of u that is incident to a vertex from T_u. We leave that job to the inquisitive reader.

We point out that the variable minback at line 17 is set to level[v] only if level[v] $\neq$ level[u] $-$ 1 for the following reason. We know that DFS in the current implementation visits every edge of an undirected graph twice because that edge is in two different adjacency lists (one list for each endpoint). So, it makes sense to consider (u, v) at lines 16 and 17 as a back edge only when u is not the immediate ancestor of v. In other words, when level[v] $\neq$ level[u] $-$ 1. We also point out that the code at lines 16–17 is executed for each back edge (u, v).

That concludes the proof of the correctness of algorithm Find Cut Vertices. The time complexity is, obviously, the same as that of DFS: $\Theta(m + n)$. □

Problem 133. *Design a fast algorithm to output all bridges of a connected graph. Analyse its correctness and time complexity.*

Solution:
This problem is similar to the previous one and the solution is quite close to algorithm Find Cut Vertices. Again we use a modification of DFS and again we consider the partition of the edges into tree edges and back edges.

Lemma 32. *With respect to the work of DFS and the classification of edges into tree edges and back edges, any edge (u, v) is a bridge iff u is the father of v in T and the subtree T_v rooted at v is such that there is no back edge from a vertex from T_v to u or a proper ancestor of u.*

Proof:
The claim is obvious, having in mind that there is no back edge from a vertex from T_v to u or a proper ancestor of u if and only if every path from a vertex from T_v to a vertex outside T_v must contain the edge (u, v). □

Unlike the problem of finding the cut vertices, now the root and the leaves of the DFS tree do not have to be treated differently from the other vertices.

```
Find Bridges(G(V, E): undirected graph)
 1  foreach u ∈ V
 2      color[u] ← White
 3  let u be an arbitrary vertex from V
 4  FBR1(G, u, 0);


FBR1(G(V, E): undirected graph, u: vertex from V, l: integer)
 1  color[u] ← Gray
 2  level[u] ← l
 3  minback ← level[u]
 4  foreach v ∈ adj(u)
```

```
 5      if color[v] = WHITE {
 6          x ← FBR1(G, v, l + 1)
 7          if x > level[u]
 8              print (u, v)
 9          else if x < minback
10                  minback ← x }
11      if color[v] = GRAY {
12          if level[v] < minback and level[v] ≠ level[u] − 1
13              minback ← level[v] }
14  color[u] ← BLACK
15  return minback
```

The proof of the correctness of algorithm FIND BRIDGES is simpler than that of FIND CUT VERTICES. The edge (u, v) is printed (line 8) iff the condition specified in Lemma 32 is fulfilled. Actually, the condition at line 7 is the main difference betweem this algorithm and FIND CUT VERTICES (see line 12 there). The time complexity is, obviously, the same as that of DFS: $\Theta(m + n)$. □

6.3.2 $\mathcal{NP}$-hard problems on restricted graphs

One way of dealing with computational intractability on graphs is designing fast algorithms for classes of graphs, graphs whose structure is restricted enough to permit fast algorithms for problems that are hard in general. A very natural candidate for a simple graph class are trees. Indeed, most intractable graph problems have fast, often linear-time, algorithms on trees, with few notable exceptions such as GRAPH BANDWIDTH that remains $\mathcal{NP}$-complete on trees, even on trees with maximum degree ≤ 3 (see [GGJK78]).

Definition 9. *Let* $G = (V, E)$ *be a graph. Any subset* $U \subseteq V$ *is called:*

- vertex cover *if* $\forall (u, v) \in E : u \in U$ *or* $v \in U$.
- dominating set *if* $\forall v \in V : v \in U$ *or* $\exists w \in U$ *such that* $(v, w) \in E$.
- independent set *if* $\forall u \in U \, \forall v \in U : (u, v) \notin E$.
- clique *if* $\forall u \in U \, \forall v \in U : (u, v) \notin E$. □

The concepts from Definition 9 have their counterparts–computational problems. Here we list the optimisation versions of the problems.

Computational Problem VERTEX COVER
Generic Instance: A graph G
Objective: Compute the size of a minimum vertex cover of G □

Computational Problem DOMINATING SET
Generic Instance: A graph G
Objective: Compute the size of a minimum dominating set of G □

Computational Problem INDEPENDENT SET
Generic Instance: A graph G
Objective: Compute the size of a maximum independent set of G □

Computational Problem MAXIMUM CLIQUE
Generic Instance: A graph G
Objective: Compute the size of a maximum clique of G □

It is immediately obvious that with respect to tractability or intractability, INDEPENDENT SET and MAXIMUM CLIQUE are in the same group.

Observation 2. *For every graph $G = (V, E)$, for any $U \subseteq V$, U is a clique iff U is an independent set in $\overline{G}$.*

The same holds for VERTEX COVER and INDEPENDENT SET though that may not be so obvious.

Theorem 4. *For any graph $G = (V, E)$, for any $U \subseteq V$, U is an independent set iff $V \setminus U$ is a vertex cover.* □

Proof:
First assume U is an independent set. Assume $V \setminus U$ is not a vertex cover. It follows there is an edge (u, v) such that $u \notin V \setminus U$ and $v \notin V \setminus U$. But that is equivalent to saying there is an edge (u, v) such that $u \in U$ and $v \in U$. Then U is not an independent set, contrary to the initial assumption.
Now assume $V \setminus U$ is a vertex cover. Assume U is not an independent set. Negating the definition of independent set, we derive $\exists u \in U \, \exists v \in U : (u, v) \in E$. The definition of vertex cover says that for every edge, at least one of its vertices is in the cover. Since $V \setminus U$ is a vertex cover and (u, v) is an edge, $u \in V \setminus U$ or $v \in V \setminus U$. It follows $u \notin U$ or $v \notin U$, contrary to the previous conclusion that $u \in U$ and $v \in U$. □

Corollary 4. *For every graph G, every minimum vertex cover U induces a maximum independent set, namely $V \setminus U$, and* vice versa. □

Problem 134. *Construct a linear time algorithm for* VERTEX COVER *on trees using a greedy approach.*

Solution:

```
VC ON TREES(T = (V, E): tree)
1  A ← ∅
2  while T has at least one edge do
3      if T has a single edge e = (u, v)
4          let x be an arbitrary vertex from {u, v}
5          A ← A ∪ {x}
6          delete x from T
7      else let U ⊂ V be the set of the leaves of T
8          B ← ∅
9          foreach u ∈ U
```

```
10              let e = (u, v) be the edge incident with u
11              A ← A ∪ {v}
12              B ← B ∪ {u, v}
13      delete B from T
14  return A
```

Note that A and B are sets so multiple additions of vertices to them at line 11 or line 12 does not matter.

The verification is straightforward. We are going to prove the returned A is a minimum vertex cover. If T has a single edge, clearly any vertex x from it covers T and {x} is a minimum vertex cover. Otherwise, by the well-known fact that in any tree with at least two edges:

- there are two leaves,
- there is at least one non-leaf, and
- for every edge incident with a leaf, the other vertex in non-leaf,

we conclude that there are at least two edges incident with leaves, and for each such edge, one endpoint is a non-leaf vertex. Clearly, each of those edges must be covered at the end, and it can only be covered by at least one of its endpoints. Therefore, if we choose the non-leaf endpoint (vertex v at line 11) to add to A, we cannot go wrong with respect to the number of vertices in A.

The latter statement is fairly obvious but if the reader is not convinced, here is a proof by contradiction. Assume for at least one edge $e = (u, v)$ where u is a leaf and v, non-leaf, the algorithm make wrong choice putting v (instead of u) in A. So, there is no minimum vertex cover of T containing v.

Consider any minimum vertex set $A' \subset V$. However, since at least one of u and v must be in any vertex cover, we conclude $u \in A'$. Now let $A'' = (A' \setminus \{u\}) \cup \{v\}$. Clearly, edge e is covered by A'' and, furthermore, any edge covered by A' is covered by A'', too, and $|A''| = |A'|$. That contradicts the assumption that no minimum vertex cover contains v.

The algorithm can easily be implemented in linear time using a postorder traverse of T. □

According to Theorem 4, that algorithm solves Independent Set as well. However, we can use dynamic programming to the same effect. The dynamic programming algorithm is linear-time as well but it has the advantage it can be used with a minor modification to solve Weighted Independent Set.

Problem 135. *Construct a linear time algorithm for* Independent Set *on trees using dynamic programming.*

Solution:

```
IS on trees, dynamic(T = (V, E): tree)
1  the algorithm uses arrays A[1, ..., n] and B[1, ..., n]
2  let r be an arbitrary vertex from V
3  make T rooted tree with root r
4  work from the leaves upwards in the following way
```

```
 5  foreach leaf vertex u
 6      A[u] = 1
 7      B[u] = 0
 8  foreach non-leaf vertex u
 9      let v_1, v_2, ..., v_k be the children of u
10      A[u] = 1 + sum_{i=1}^{k} B[v_i]
11      B[u] = sum_{i=1}^{k} max{A[v_i], B[v_i]}
12  return  max{A[r], B[r]}
```

The arrays A[] and B[] keep the following information with respect to the *rooted* tree T. Let T_u denote the subtree rooted at u, for every u. For every vertex u,

- $A[u]$ is the size of a maximum independent set in T_u that contains u, and
- $B[u]$ is the size of a maximum independent set in T_u that does not contain u.

The verification of the algorithm is based on the trivial fact that an optimum independent set either contains a certain vertex, or does not contain that vertex. In particular, the root is in some optimum independent set or not. The assignments at lines 6 and 7 are obviously correct. In a recursive implementation of the algorithm those lines correspond to the bottom of the recursion. The assignment at line 10 is correct because if u is necessarily contained in any independent set, then $v_1, \ldots, v_k$ are necessarily not in that set; therefore, we choose max independent sets in $T_{v_1}, \ldots, T_{v_k}$ that do not contain the respective roots $v_1, \ldots, v_k$. Consider the assignment at line 11. If u is not in the independent set, for any child v_i we can pick the maximum independent set in T_{v_I} regardless of whether it contains v_i or not.

The key observation, unmentioned so far, with respect to the correctness, is that the optimum for any internal vertex is obtained from the optimuma obtained for its children, those optima being *independent* of one another.

The proposed algorithm can be implemented with linear time complexity if we traverse the tree in postorder. The recurrence relation of its time complexity is

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{m} T(n_i) + \Theta(m)$$

where $n_1, n_2, \ldots, n_m$ is a numerical partition of $n-1$. The reason is that at each internal vertex the algorithm performs work proportional to the number of the children of that vertex, in addition to the recursive calls, one recursive call per child. The number of children is assumed to be m and the total number of vertices in all subtrees rooted at the children is $n-1$. According to Problem 80 on page 88, this recurrence relation has solution $O(n)$. The fact that it is also $\Theta(n)$ is obvious. □

Problem 136. *Construct a linear time algorithm for* Dominating Set *on trees using dynamic programming.*

Solution: For each vertex x is a rooted tree, let $ch(x)$ denote the set of its children.

DS ON TREES($T = (V, E)$: tree)

1 the algorithm uses arrays $A[1, \ldots, n]$ and $B[1, \ldots, n]$
2 let r be an arbitrary vertex from V
3 make T rooted tree with root r
4 work from the leaves upwards in the following way
5 **foreach** leaf vertex u
6 $\quad A[u] = 1$
7 $\quad B[u] = 0$
8 **foreach** non-leaf vertex u
9 $$A[u] = 1 + \min\left\{\sum_{v \in ch(u)} B[v], \min_{v \in ch(u)}\left\{\sum_{w \in ch(v)} B[w] + \sum_{z \in (ch(u) \setminus \{v\})} A[z]\right\}\right\}$$
10 $$B[u] = \min\left\{1 + \sum_{v \in ch(u)} B[v], \sum_{v \in ch(u)} A[v]\right\}$$
11 **return** $\min\{A[r], B[r]\}$

The arrays $A[]$ and $B[]$ keep the following information with respect to the *rooted* tree T. Let T_u denote the subtree rooted at u, for every u. For every vertex u,

- $A[u]$ is the size of a minimum dominating set in T_u, and
- $B[u]$ is the size of a minimum set in T_u that dominates every vertex in T_u except possibly u.

The verification of the assignments at lines 6 and 7 is trivial. Consider line 9, which can be written as

$$A[u] = \min\left\{1 + \sum_{v \in ch(u)} B[v], 1 + \min_{v \in ch(u)}\left\{\sum_{w \in ch(v)} B[w] + \sum_{z \in (ch(u) \setminus \{v\})} A[z]\right\}\right\}$$

Indeed, a minimum dominating set in T_u either includes u, or not. If it includes u then for all children of u it suffices to consider (that is, to sum together) the sizes of their vertex subsets that dominate all vertices except the children of u, for if u is included in the dominating set then its children are dominated by it. The value $1 + \sum_{v \in ch(u)} B[v]$ corresponds to that possibility.

If a dominating set in T_u does not include u then u has to be dominated by one of its children. The minimum size of the dominating set for T_u in this case is the minimum over every child v of the sum of those three:

- 1, which is the contribution of vertex v,
- the sum over all children of v of the sizes of the dominating sets of the subtrees rooted at them, those dominating sets possibly leaving those children (of v) undominated; the reason we can afford that is although they are undominated with respect to the subtrees rooted at them, they get dominated with respect to the subtree rooted at v because v is in a dominating set, and
- the sum over all other children of u (not v) of the sizes of the minimum dominating sets, each one containing the corresponding child; the reason the child must be included is that u is not in the overall dominating set of T_u.

The value $1+\min_{v\in ch(u)}\left\{\sum_{w\in ch(v)} B[w] + \sum_{z\in(ch(u)\setminus\{v\})} A[z]\right\}$ corresponds to the current possibility.

Now consider line 10. With respect to $B[u]$, u can be left undominated but that does not mean it is *necessarily* undominated. Thus, it is possible that u is included in the set. If u is included then all vertices in T_u are dominated, so a minimum dominating set for T_u has size $1 + \sum_{v\in ch(u)} B[v]$, the 1 reflecting the contribution of u and the $B[\,]$ values of the children of u reflecting the fact that all those children are dominated by u anyways. However, if u is not included in the dominating set, we have to sum over all children the sizes of the minimum dominating sets in their subtrees that ensure the domination of the said children because now u cannot dominate them. The correct value in this case is $\sum_{v\in ch(u)} A[v]$.

The proposed algorithm can be implemented with linear time complexity if we traverse the tree in postorder. The recurrence relation of its time complexity is

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{m} T(n_i) + \Theta(m)$$

where $n_1, n_2, \ldots, n_m$ is a numerical partition of $n-1$. In contrast with IS ON TREES, DYNAMIC, it may not be immediately obvious that this recurrence relation describes the time complexity of DS ON TREES. To see that recurrence is applicable to DS ON TREES, note that all obtained $A[\,]$, $B[\,]$, $\sum A[\,]$, and $\sum B[\,]$ values up to a certain moment can be stored and used later, *i.e.* when the current execution is further up in the tree. The value $\sum_{z\in(ch(u)\setminus\{v\})} A[z]$ therefore can be obtained as a difference in constant time. Having established that this recurrence relation is applicable, we use Problem 80 on page 88 to conclude the algorithm is linear time. □

6.3.3 Dynamic Programming

Consider the following problem from The Algorithm Design Manual of Steven Skiena [Ski08, pp. 315, Problem 8-22].

Problem 137. *Consider the problem of examining a string* $x = x_1x_2\ldots x_n$ *from an alphabet of* k *symbols, and a multiplication table over this alphabet. Decide whether or not it is possible to parenthesize* x *in such a way that the value of the resulting expression is* a*, where* a *belongs to the alphabet. The multiplication table is neither commutative or associative, so the order of multiplication matters.*

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

For example, consider the above multiplication table and the string bbbba*. Parenthesizing it* (b(bb))(ba) *gives* a*, but* ((((bb)b)b)a) *gives* c*. Give an algorithm, with time polynomial in* n *and* k*, to decide whether such a parenthesization exists for a given string, multiplication table, and goal element.*

Solution:

```
STRING EVALUATION(Σ: finite alphabet, x ∈ Σ*, ⊗: binary operation over Σ, a ∈ Σ)
 1  (* The algorithm computes whether there exists a parenthesisation *)
 2  (* of x such that the application of ⊗ according to it *)
 3  (* results in symbol a *)
 4  let x be σ_1, σ_2, ..., σ_n
 5  let T be an n × n table with elements-subsets of Σ
 6  for i ← 1 to n
 7      T[i, i] ← σ_i
 8  for diag ← 1 to n − 1
 9      for i ← 1 to n − diag
10          T[i, i + diag] ← ∅
11          for k ← i to i + diag − 1
12              foreach p ∈ T[i, k]
13                  foreach q ∈ T[k + 1, i + diag]
14                      T[i, i + diag] ← T[i, i + diag] ∪ {p ⊗ q}
15  if a ∈ T[1, n]
16      return Yes
17  else
18      return No
```

The proposed algorithm resembles the well-known MATRIX-CHAIN-ORDER (see [CLR00]), which is not coincidental because in both cases the goal is to compute a parenthesisation with certain properties of a linear order. Combinatorially, the number of all possible parenthesisations of a linear order of n objects is

$$P_n = \begin{cases} 1, & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P_k P_{n-k}, & \text{if } n \geq 2 \end{cases}$$

and therefore the brute force algorithm to both MATRIX-CHAIN-ORDER and the current problem would have time complexity $\Omega(P_n)$. It is well-known that P_n equals C_{n-1}, the $(n-1)^{\text{th}}$ Catalan number, and $C_n \asymp \frac{4^n}{n\sqrt{n}}$.

The proposed solution to the current problem uses a function

$$T : \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\} \to 2^{\Sigma}$$

defined as follows. For all $1 \leq i \leq j \leq n$, $T(i, j)$ is the set of all $p \in \Sigma$ such that there is a parenthesisations of the substring $\sigma_i, \ldots, \sigma_j$ such that the application of $\otimes$ according to this parenthesisation yields p. That function can be computed efficiently using a two dimensional table for storing partial results in the following way:

$$T[i, j] = \begin{cases} \sigma_i, & \text{if } i = j \\ \bigcup_{k=i}^{j-1} \{p \otimes q \mid p \in T[i, k] \wedge q \in T[k + 1, j]\}, & \text{else} \end{cases} \tag{6.3}$$

The answer is Yes if and only if $T[1, n]$ contains the given a. Algorithm STRING EVALUATION is a straightforward application of (6.3). Its time complexity is $O(n^3 k^2)$.

Computational Problem PARTITION
Generic Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$, a weight function $w : A \to \mathbb{Z}^+$

Question: Does there exist a partition A_1, A_2 of A such that $\sum_{a \in A_1} w(a) = \sum_{a \in A_2} w(a)$? □

Computational Problem EQUAL SUM SUBSETS (ESS)
Generic Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$, a weight function $w : A \to \mathbb{Z}^+$
Question: Do there exist two non-empty non-intersecting subsets A_1, A_2 of A such that $\sum_{a \in A_1} w(a) = \sum_{a \in A_2} w(a)$? □

Problem 138. *Design a reasonably fast algorithm for* PARTITION.

Solution: The proposed solution is well-known.

```
PARTITION(A: set, w : A → ℤ⁺)
 1  if (Σ_{a∈A} w(a)) mod 2 = 1
 2      return No
 3  S ← (1/2) Σ_{a∈A} w(a)
 4  let T be a two-dimensional, n × (S + 1) boolean array T[1...n, 0...S]
 5  for j ← 0 to S
 6      if j = 0 or j = w(a_1)
 7          T[1, j] ← 1
 8      else
 9          T[1, j] ← 0
10  for i = 2 to n
11      for j = 0 to S
12          T[i, j] ← T[i − 1, j] ∨ T[i − 1, j − w(a_i)]
13  if T[n, S] = 1
14      return Yes
15  else
16      return No
```

The verification is straightforward. Let A^i denote the subset $\{a_1, a_2, \ldots, a_i\}$. Let $T(i, j)$ be the following predicate, where the domain of i is $\{1, 2, \ldots, n\}$ and the domain of j is $\{0, 1, \ldots, S\}$:

$$T(i, j) = \begin{cases} \text{TRUE}, & \text{if } \exists B \subseteq A^i : \sum_{a \in B} w(a) = j \\ \text{FALSE}, & \text{else} \end{cases}$$

The proposed algorithm is a direct algorithmic implementation of that predicate using the dynamic programming approach with a two-dimensional table. First note that if the sum of all weights is odd there cannot exist such a partition, therefore the preprocessing at lines 1 and 2 is correct. Then note that $T(1, j)$ is TRUE for precisely two values of j:

- for $j = 0$ because there exists a subset B of A^1, namely $B = \emptyset$, such that the sum $\sum_{a \in B} w(a)$ over its elements, namely the empty sum, equals $j = 0$.
- for $j = w(a_1)$ because there exists a subset B of A^1, namely $B = \{a_1\} = A^1$, such that the sum $\sum_{a \in B} w(a)$ over its elements equals $j = w(a_1)$.

For all other values of j, $T(1,j)$ is FALSE. It follows the first row of the table is filled in correctly (lines 5–9). The key observation is that $T(i,j)$ for $2 \leq i \leq n$ is TRUE iff $T(i-1,j)$ is TRUE or $T(i-1,j-w(a_i))$ is TRUE, the reason being that A^i has a subset whose elements' weights sum up to j iff at least one of the following is the case:

- there is a subset of A^{i-1} whose elements' weights sum up to j, in which case we do not use element a_i,
- there is a subset of A^{i-1} whose elements' weights sum up to $j-w(a_i)$, in which case we do use element a_i to obtain a subset with sum equal to j.

The time complexity of PARTITION is $\Theta(nS)$. Note this is *not* a polynomial complexity because S can be (at worst) exponential in the size of the input if the input is encoded succinctly, *e.g.* in binary. However, if S is small, *i.e.* at most polynomial in the size of the input[†], the time complexity would be polynomial. Algorithms whose time complexity depends on some numbers (number S in this case) in such a way that if the numbers are at most polynomial in the input size the time complexity is polynomial but if the numbers are superpolynomial then the time complexity becomes superpolynomial, are called *pseudopolynomial time* algorithms. □

Problem 139. *Design a reasonably fast algorithm for* ESS.

Solution: We point out that ESS is a generalisation of PARTITION. ESS becomes PARTITION if we impose the additional requirement that $A_1 \cup A_2 = A$. The proposed solution to ESS is based on [CEPS08]

```
2ESS(A: set, w : A → ℤ⁺)
 1  S ← (1/2) Σ_{a∈A} w(a)
 2  let T be a three-dimensional, n×(S+1)×(S+1) boolean array T[1...n, 0...S, 0...S]
 3  for j ← 0 to S
 4      for k ← 0 to S
 5          if (j = k = 0) or (j = w(a_1) and k = 0) or (j = 0 and k = w(a_1))
 6              T[1, j, k] ← 1
 7          else
 8              T[1, j, k] ← 0
 9  for i = 2 to n
10      for j = 0 to S
11          for k = 0 to S
12              T[i, j, k] ← T[i − 1, j, k] ∨ T[i − 1, j − w(a_i), k] ∨ T[i − 1, j, k − w(a_i)]
13  for j ← 1 to S
14      if T[n, j, j] = 1
15          return Yes
16  return No
```

Let A^i denote the subset $\{a_1, a_2, \ldots, a_i\}$. Let $T(i,j,k)$ be the following predicate, where

[†] S is at most polynomial in the input size if its representation under a sane encoding is at most logarithmic in the input size.

the domain of i is $\{1, 2, \ldots, n\}$, the domains of both j and k is $\{0, 1, \ldots, S\}$:

$$T(i,j,k) = \begin{cases} \text{TRUE}, & \text{if } \exists B, C \subseteq A^i : B \cap C = \emptyset \wedge \sum_{a \in B} w(a) = j \wedge \sum_{a \in C} w(a) = k \\ \text{FALSE}, & \text{else} \end{cases}$$

The proposed algorithm is a direct algorithmic implementation of that predicate using the dynamic programming approach with a three-dimensional table. Note that $T(1,j,k)$ is TRUE for precisely three values of the ordered pair $\langle j,k\rangle$:

- for $\langle 0,0\rangle$ because there exist subsets B, C of A^1, namely $B = \emptyset$ and $C = \emptyset$ such that the sum $\sum_{a \in B} w(a)$ equals $j = 0$ and the sum $\sum_{a \in C} w(a)$ equals $k = 0$.
- for $\langle w(a_1), 0\rangle$ because there exists a subset B of A^1, namely $B = \{a_1\} = A^1$, such that the sum $\sum_{a \in B} w(a)$ equals $j = w(a_1)$ and there exists a subset $C = \emptyset$ of A^1 such that the sum $\sum_{a \in C} w(a)$ equals $k = 0$.
- for $\langle 0, w(a_1)\rangle$, the argument being completely analogous to the argument above.

For all other values of $\langle j$, $T(1,j)$ is FALSE. It follows the first row of the table is filled in correctly (lines 3–8). The key observation is that $T(i,j,k)$ for $2 \leq i \leq n$ is TRUE iff $T(i-1,j,k)$ is TRUE or $T(i-1,j-w(a_i),k)$ is TRUE or $T(i-1,j,k-w(a_i))$ is TRUE. The reason is that A^i has two subsets B and C whose elements' weights sum up to j and k, respectively, iff at least one of the following is the case:

- there are such subsets of A^{i-1}, in which case we do not use element a_i,
- there is a subset B' of A^{i-1} whose elements' weights sum up to $j-w(a_i)$ and a subset C' of A^{i-1} whose elements' weights sum up to k, in which case $B = B' \cup \{a_i\}$ and $C = C'$.
- there is a subset B' of A^{i-1} whose elements' weights sum up to j and a subset C' of A^{i-1} whose elements' weights sum up to $k - w(a_i)$, in which case $B = B'$ and $C = C' \cup \{a_i\}$.

The time complexity of 2ESS is $\Theta(nS^2)$ and so it is another pseudopolynomial time algorithm. □

Chapter 7

Intractability

7.1 Several $\mathcal{NP}$-complete decision problems

Suppose Π_1 and Π_2 are computational decision problems version, *i.e.* with YES/NO answers. *A polynomial reduction from* Π_1 *to* Π_2 is a polynomial-time computable (total) function $f : \Pi_1 \to \Pi_2$ such that $\forall x \in \Pi_1 : x \in Y_{\Pi_1}$ iff $f(x) \in Y_{\Pi_2}$. The fact that Π_1 reduces to Π_2 by a polynomial reduction is denoted by $\Pi_1 \propto \Pi_2$. Here is a list of several decision problems:

Computational Problem SAT
Generic Instance: A set of boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, a CNF[†] Q on them
Question: Does there exist a truth assignment for X that satisfies Q? □

Computational Problem 3SAT
Generic Instance: A set of boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, a CNF Q on them with precisely 3 literals per clause
Question: Does there exist a truth assignment for X that satisfies Q? □

Computational Problem VERTEX COVER (VC)
Generic Instance: A graph G, $k \in \mathbb{N}$
Question: Does G have a vertex cover of size $\leq k$? □

Computational Problem DOMINATING SET (DS)
Generic Instance: A graph G, $k \in \mathbb{N}$
Question: Does G have a dominating set of size $\leq k$? □

Computational Problem INDEPENDENT SET (IS)
Generic Instance: A graph G, $k \in \mathbb{N}$
Question: Does G have an independent set of size $\geq k$? □

[†]CNF means Conjunctive Normal Form.

Computational Problem CLIQUE
Generic Instance: A graph G, $k \in \mathbb{N}$
Question: Does G have a clique of size $\geq k$? □

Computational Problem 3-DIMENSIONAL MATCHING (3DM)
Generic Instance: A set $A \subseteq B \times C \times D$ where B, C, and D are pairwise disjoint sets such that $|B| = |C| = |D| = n$.
Question: Does A contain *matching*, *i.e.* a subset $A' \subseteq A$ such that $|A'| = n$ and

$$\forall \alpha = (a_\alpha, b_\alpha, c_\alpha) \in A' \quad \forall \beta = (a_\beta, b_\beta, c_\beta) \in A' : \alpha \neq \beta \rightarrow a_\alpha \neq a_\beta \wedge b_\alpha \neq b_\beta \wedge c_\alpha \neq c_\beta \ ?$$

□

For brevity, let us call such ordered triples *component-wise distinct*. In that terminology, the 3DM problem is: do there exist component-wise distinct triples in A such that every element from B, C, and D is in (precisely) one of them? Obviously, this is the same question as: do there exist n component-wise distinct triples in A?

Computational Problem HAMILTONIAN CYCLE (HC)
Generic Instance: A graph $G = (V, E)$
Question: Is there a Hamiltonian cycle in G? □

Computational Problem PARTITION
Generic Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$, a weight function $w : A \rightarrow \mathbb{Z}^+$
Question: Does there exist a partition A_1, A_2 of A such that $\sum_{a \in A_1} w(a) = \sum_{a \in A_2} w(a)$?
□

Computational Problem KNAPSACK
Generic Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$, a weight function $w : A \rightarrow \mathbb{Z}^+$, a value function $v : A \rightarrow \mathbb{Z}^+$, a size constraint $B \in \mathbb{Z}^+$, a value goal $K \in \mathbb{Z}^+$
Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} w(a) \leq B \quad \text{and} \quad \sum_{a \in A'} v(a) \geq K \ ?$$

□

Computational Problem TRAVELING SALESMAN PROBLEM (TSP)
Generic Instance: A finite set of locations $C = \{c_1, c_2, \ldots, c_n\}$, a distance function $\text{dist} : C \times C \rightarrow \mathbb{Z}^+$[†], a bound $B \in \mathbb{N}^+$
Question: Does there exist a permutation π of all locations

$$c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)}$$

such that $\left(\sum_{i=1}^{n} \text{dist}(c_{\pi(i)}, c_{\pi(i+1)})\right) + \text{dist}(c_{\pi(n)}, c_{\pi(1)}) \leq B$? □

[†]We can assume $\text{dist}(x, y)$ is 0 for $x = y$ and also equals $\text{dist}(y, x)$.

Definition 10. *For any two paths in a directed or undirected graph we say they are* edge-disjoint *if they have no common edges, they are* vertex-disjoint *if they have no common vertices, and they are* internally vertex-disjoint *if they have no common vertices except possibly for common endpoints.* □

Computational Problem Edge-Disjoint Paths (EDP)
Generic Instance: A directed graph $G = (V, E)$, a list of tuples (s_1, t_1), ..., (s_k, t_k) of vertices from V.
Question: Do there exist k edge-disjoint directed paths

$$p_1 = s_1, \ldots, t_1$$
$$p_2 = s_2, \ldots, t_2$$
$$\ldots$$
$$p_k = s_k, \ldots, t_k$$

in G? □

Computational Problem Vertex-Disjoint Paths (VDP)
Generic Instance: A directed graph $G = (V, E)$, a list of tuples (s_1, t_1), ..., (s_k, t_k) of vertices from V.
Question: Do there exist k internally vertex-disjoint[†] directed paths

$$p_1 = s_1, \ldots, t_1$$
$$p_2 = s_2, \ldots, t_2$$
$$\ldots$$
$$p_k = s_k, \ldots, t_k$$

in G? □

Subsections 7.2.11 on page 229 and 7.2.12 on page 230 prove the well-known fact that EDP and VDP are computationally equivalent modulo a polynomial factor. Both EDP and VDP can be defined on undirected graphs. The directed and undirected versions of the problem have dramatically different complexities: while the directed versions are $\mathcal{NP}$-complete (see subsection 7.2.10 on page 224 for the $\mathcal{NP}$-hardness; the fact that the problems are in $\mathcal{NP}$ is trivial) and remain so even for $k = 2$ [FHW80], the undirected version is in $\mathcal{P}$ for every fixed k [RS95]. A survey of several versions of the disjoint paths problems can be found in [Pap95, pp. 214–215].

[†]It is necessary to use the weaker concept of vertex-disjointness, *viz.* "internally vertex-disjoint", because the tuples may have vertices in common.

7.2 Polynomial Reductions

7.2.1 SAT $\propto$ 3SAT

Suppose $X = \{x_1, x_2, \ldots, x_n\}$ is a set of boolean variables and $Q = \{q_1, q_2, \ldots, q_m\}$ is a CNF on them. For each q_i, $1 \leq i \leq m$, we do the following contruction.

Case i: If q_i has a single literal, *i.e.* $q_i = (y)$ for some literal y over X, add two variables $u_{1,i}$, $u_{2,i}$ that are not in X and are not going to be used anywhere else in our construction, and construct the four clauses:

$$q_i^1 = (y, u_{1,i}, u_{2,i}), q_i^2 = (y, u_{1,i}, \overline{u_{2,i}}), q_i^3 = (y, \overline{u_{1,i}}, u_{2,i}), q_i^4 = (y, \overline{u_{1,i}}, \overline{u_{2,i}})$$

Let $Q' = (Q \setminus \{q_i\}) \cup \{q_i^1, q_i^2, q_i^3, q_i^4\}$. We claim that Q is satisfiable iff Q' is satisfiable. Indeed, if Q is satisfiable then every satisfying assignment t for Q must set literal y to TRUE. Then any assignment t' for Q' that agrees with t on all variables except for $u_{i,1}$ and $u_{i,2}$ satisfies each one of q_j^1 for $1 \leq j \leq 4$. It follows t' is a satisfying assignment for Q'. In the other direction, a satisfying assignment t' for Q' implies a satisfying assignment t for Q, t being the restriction of t' on the variables from X. To see why note that t' must set y to TRUE because if y is set to FALSE there is no way to satisfy q_j^1 for $1 \leq j \leq 4$ simultaneously with $u_{1,i}$ and $u_{2,i}$.

Case ii: If q_i has precisely two literals, *i.e.* $q_i = (y, z)$ for some literals y and z over X, add a single variable v_i that is not in X and is not going to be used anywhere else in our construction, and construct the two clauses:

$$q_i^1 = (y, z, v_i), q_i^2 = (y, z, \overline{v_i})$$

The justification of that construction is analogous to above.

Case iii: If q_i has precisely three literals we leave it as it is.

Case iv: And finally suppose q_i has > 3 literals. Let $q_i = (y_1, y_2, \ldots, y_k)$ for some $k > 3$. Add $k-3$ new variables $u_{1,i}$, $u_{2,i}$, $\ldots$, $u_{k-3,i}$ that are not in X and are not going to be used anywhere else in our construction. Construct $k-2$ new clauses:

$$\begin{aligned}
q_i^1 &= (y_1, y_2, u_{1,i}),\\
q_i^2 &= (\overline{u_{1,i}}, y_3, u_{2,i}),\\
q_i^3 &= (\overline{u_{2,i}}, y_4, u_{3,i}),\\
&\ldots\\
q_i^{k-4} &= (\overline{u_{k-5,i}}, y_{k-3}, u_{k-4,i}),\\
q_i^{k-3} &= (\overline{u_{k-4,i}}, y_{k-2}, u_{k-3,i}),\\
q_i^{k-2} &= (\overline{u_{k-3,i}}, y_{k-1}, y_k)
\end{aligned}$$

Let $Q' = (Q \setminus \{q_i\}) \cup \{q_i^1, q_i^2, \ldots, q_i^{k-2}\}$. We prove that Q is satisfiable iff Q' is satisfiable.

$\Longrightarrow$ Suppose there is a satisfying assignment t for Q. At least one of the literals y_1, y_2, $\ldots$, y_k must be set to TRUE by t. We construct a truth assignment t' for $X \cup \{u_{1,i}, u_{2,i}, \ldots, u_{k-3,i}\}$ that agrees with t on all variables from X and assigns the following values to $u_{1,i}$, $u_{2,i}$, $\ldots$, $u_{k-3,i}$.

- If $t(y_1) = \textsc{True}$ or $t(y_2) = \textsc{True}$ then

$$\begin{aligned} t'(u_{1,i}) &= \textsc{False} \\ t'(u_{2,i}) &= \textsc{False} \\ &\dots \\ t'(u_{k-3,i}) &= \textsc{False} \end{aligned}$$

  In this case, q_i^1 is satisfied by y_1 or y_2, and q_i^2, ..., q_i^{k-2} are satisfied by $\overline{u_{1,i}}$, ..., $\overline{u_{k-3,i}}$, respectively.

- Suppose $t(y_j) = \textsc{True}$ for some j such that $3 \le j \le k-2$. Note y_j is in the clause $q_i^{j-1} = (\overline{u_{j-2,i}}, y_j, u_{j-1,i})$. The truth assignments for the $u_{j,i}$ variables are

$$\begin{aligned} t'(u_{1,i}) &= \textsc{True} \\ t'(u_{2,i}) &= \textsc{True} \\ &\dots \\ t'(u_{j-2,i}) &= \textsc{True} \\ t'(u_{j-1,i}) &= \textsc{False} \\ t'(u_{j,i}) &= \textsc{False} \\ &\dots \\ t'(u_{k-3,i}) &= \textsc{False} \end{aligned}$$

  Note that

  - q_i^{j-1} is satisfied by y_j,
  - q_i^1, ..., q_i^{j-2} are satisfied by the literals $u_{1,i}$, ..., $u_{j-2,i}$, respectively, and
  - q_i^j, ..., q_i^{k-2} are satisfied by $\overline{u_{j-1,i}}$, ..., $\overline{u_{k-3,i}}$, respectively.

- If $t(y_{k-1}) = \textsc{True}$ or $t(y_k) = \textsc{True}$ then

$$\begin{aligned} t'(u_{1,i}) &= \textsc{True} \\ t'(u_{2,i}) &= \textsc{True} \\ &\dots \\ t'(u_{k-3,i}) &= \textsc{True} \end{aligned}$$

  In this case, q_i^{k-2} is satisfied by y_{k-1} or y_k, and q_i^1, ..., q_i^{k-2} are satisfied by $u_{1,i}$, ..., $u_{k-3,i}$, respectively.

$\Longleftarrow$ Suppose there is a satisfying assignment t' for Q'. But at least one of y_1, ..., y_k is assigned $\textsc{True}$ by $t'(\,)$ – assume the opposite and see there is no way the clauses q_i^1, ..., q_i^{k-2} are satisfied simultaneously by the $u_{1,i}$ and $\overline{u_{j,i}}$ literals.

That concludes the proof that Q is satisfiable iff Q' is satisfiable. The fact that the construction can be carried out in polynomial time is patently obvious.

7.2.2 3SAT $\propto$ 3DM

Consider an arbitrary instance of 3SAT: a set $X = \{x_1, x_2, \ldots, x_n\}$ of boolean variables and CNF $Q = \{q_1, q_2, \ldots, q_m\}$, each clause q_j being a disjunction of precisely 3 literals over X. We are going to construct three pairwise disjoint sets B, C, and D such that $|B| = |C| = |D|$ and a set $A \subseteq B \times C \times D$ such that A contains $|B|$ componentwise-distinct triples iff Q is satisfiable. The sets B, C, and D are

$$\begin{aligned}
B = &\{u_{i,j} \,|\, 1 \le i \le n, 1 \le j \le m\} \cup \\
&\{v_{i,j} \,|\, 1 \le i \le n, 1 \le j \le m\} \\
C = &\{y_{i,j} \,|\, 1 \le i \le n, 1 \le j \le m\} \cup \\
&\{s_{1,j} \,|\, 1 \le j \le m\} \cup \\
&\{g_{1,j} \,|\, 1 \le j \le m(n-1)\} \\
D = &\{z_{i,j} \,|\, 1 \le i \le n, 1 \le j \le m\} \cup \\
&\{s_{2,j} \,|\, 1 \le j \le m\} \cup \\
&\{g_{2,j} \,|\, 1 \le j \le m(n-1)\}
\end{aligned}$$

with $6mn$ elements altogether. The set A is

$$A = (\cup_{i=1}^{n} A_{1,i}) \bigcup (\cup_{j=1}^{m} A_{2,j}) \bigcup A_{3,u} \bigcup A_{3,v}$$

where

$$\begin{aligned}
A_{1,i} = &\{(u_{i,j}, y_{i,j}, z_{i,j}) \,|\, 1 \le j \le m\} \cup \\
&\{(v_{i,j}, y_{i,j+1}, z_{i,j}) \,|\, 1 \le j < m\} \cup \{(v_{i,m}, y_{i,1}, z_{i,m})\}
\end{aligned} \tag{7.1}$$

$$A_{2,j} = \{(u_{i,j}, s_{1,j}, s_{2,j}) \,|\, \overline{x_i} \text{ is a literal in } q_j\} \cup \{(v_{i,j}, s_{1,j}, s_{2,j}) \,|\, x_i \text{ is a literal in } q_j\} \tag{7.2}$$

$$\begin{aligned}
A_{3,u} = \big\{&(u_{i,j}, g_{1,k}, g_{2,k}) \,|\, 1 \le i \le n, 1 \le j \le m, 1 \le k \le m(n-1), \\
&u_{i,j} \text{ does not occur in any triple of } A_{3,j}, \text{ for any } j\big\}
\end{aligned}$$

$$\begin{aligned}
A_{3,v} = \big\{&(v_{i,j}, g_{1,k}, g_{2,k}) \,|\, 1 \le i \le n, 1 \le j \le m, 1 \le k \le m(n-1), \\
&v_{i,j} \text{ does not occur in any triple of } A_{3,j}, \text{ for any } j\big\}
\end{aligned}$$

It is obvious the construction can be performed in linear time. Now we prove that Q is satisfiable iff A has a subset A' that is a three-dimentional matching by giving a small example and generalising it. The reduction is based on two main ideas, the first of which is the following. Suppose we are given a set of triples that contains a circular-shaped substructure like the one shown on Figure 7.1. Suppose that the red elements do not appear in any other triple while the black elements appear in other triples. The only way to achieve a 3DM is to choose every other triple from that substructure with respect to the circular arrangement. Needless to say, if the number of triples in this structure is odd then the task cannot be accomplished. We discuss the second idea further on. Consider the

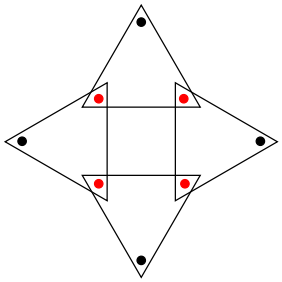


Figure 7.1: A circular arrangements of triples that is a subset of a larger set of triples. The red elements do not appear in other triples while the black elements do. There can be no matching in the larger set (not shown) unless it contains two of the shown triples that are not adjacent in the circular arrangement, *i.e.* have empty intersection.

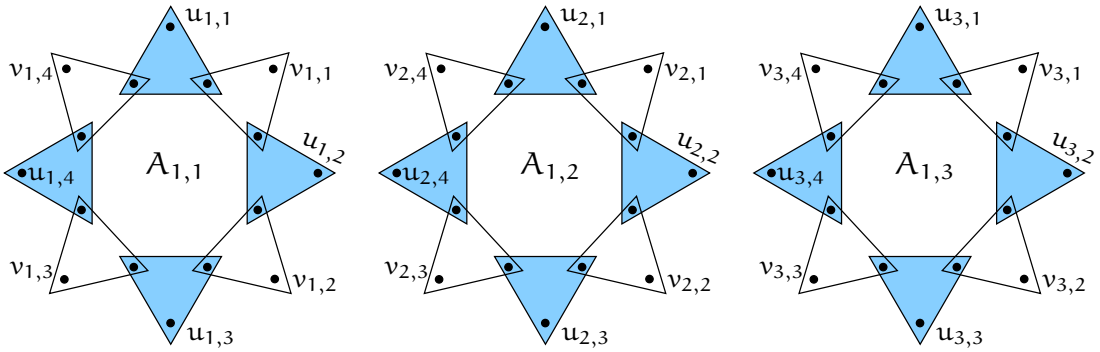


Figure 7.2: The three stars in the construction for $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. $A_{1,1}$ corresponds to x_1, $A_{1,2}$ corresponds to x_2, and $A_{1,3}$ corresponds to x_3.

following CNF:

$$Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

A satisfying truth assignment for it is, say $t(x_1) = t(x_2) = 1$, $t(x_3) = 0$. Now $n = 3$ and $m = 4$. Our construction uses $6 \times 4 \times 3 = 72$ set elements, partitioned among B, C, and D. We call the sets $A_{1,i}$, *the stars*. Each of the three stars $A_{1,1}$, $A_{1,2}$, and $A_{1,3}$ has $2 \times 4 = 8$ triples as dictated by equation (7.1) on the preceding page, arranged in a circular order as shown on Figure 7.2. Star $A_{1,i}$ corresponds to variable x_i. In each star only the $u_{i,j}$ and $v_{i,j}$ elements are named because they will take part in other triples further on. The anonymous elements do not take part in any other triples. Those anonymous elements are the $y_{i,j}$ and the $z_{i,j}$ elements. In each star $A_{1,i}$, the $u_{i,j}$ elements for $1 \leq j \leq m$ model the appearance of the literal x_i in clause q_j while the $v_{i,j}$ elements for $1 \leq j \leq m$ model the appearance of the literal $\overline{x_i}$ in clause q_j[†]. In order to obtain a valid 3DM, for each star either all shaded, or all unshaded, triples must be put in the matching because the $y_{i,j}$'s and $z_{i,j}$'s (the anonymous dots around the inside on Figure 7.2) are not in any other triples, so if we leave some of them "uncovered" now they will never be "covered". In other words,

[†]Of course, not every possible literal appears in every clause. The "garbage collection" triples we introduce further on take care of that.

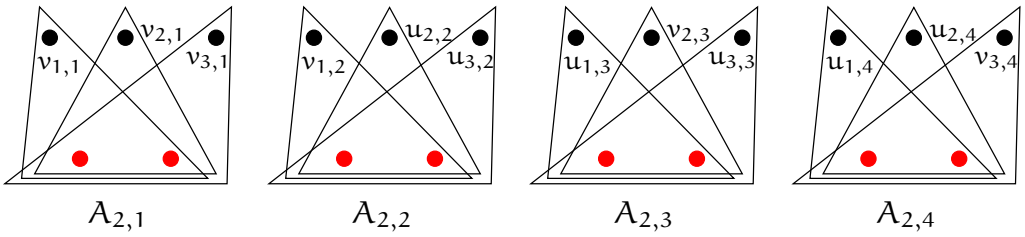


Figure 7.3: The four crowns in the construction for $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})$ $(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. $A_{2,1}$ corresponds to clause q_1 and so on. The red dots represent the common anonymous elements.

for each variable x_i, either we assign it TRUE in every clause, or we assign it FALSE in every clause. Which, of course, it the right thing to do. The convention we follow is that picking the u values for variable x_i assigns FALSE to it and picking the v values assigns TRUE to it.

Now consider the sets of triples $A_{2,j}$ defined by (7.2). We call those sets, "the crowns". In this example the crowns are four and crown $A_{2,j}$ corresponds to clause q_j for $1 \leq j \leq 4$. Each crown is the overlap of three triples, each two of them sharing the same two common elements $s_{1,j}$, $s_{2,j}$. Now those common elements are the anonymous elements, the ones that do not appear in any other triples. See Figure 7.3 for illustration. The anonymous red dots represent the elemens $s_{1,j}$, $s_{2,j}$ that do not appear in any other triple. The u and v elements from the crowns are shared with the hitherto defined stars. See Figure 7.4 which illustrates the joining together of the stars and the crowns. The purpose of the crowns is to make sure the matching, if one exists, corresponds to a satisfying assignment. Obviuosly, exactly one of the triples from each crown participates in the matching, which corresponds to the fact that each clause must be satisfied. For each crown, "the spike" of its participating triple, *i.e.* the element of the participating triple that is u or v, is of crucial importance. That spike is shared with some star. Because of that sharing, in the said star, the triples from that star that are in the matching have *opposite* truth values to the truth value of the spike. Recall that the truth values are encoded by the correspondence

u corresponds to FALSE, v corresponds to TRUE.

The other two spikes of the crown are elements that, in order to be covered in the matching, must be covered by the stars they are in.

Now we argue that a satisfying assignment of Q corresponds to a 3DM in the structure we built. Recall that in our examples the CNF is:

$$Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

In each clause that has to be a literal that is TRUE. Let x_1 be the literal that is TRUE in $q_1 = (x_1 \vee x_2 \vee x_3)$. We model that by chosing the triple $(v_{1,1}, s_{1,1}, s_{2,1})$ from $A_{2,1}$. That choice has the following consequences with respect to the matching. See Figure 7.5 for an illustration. All colour-coding on the figure is removed except for the green colour of $(v_{1,1}, s_{1,1}, s_{2,1})$. Now $v_{1,1}$ in $A_{1,1}$ is covered and thus the triple of $A_{1,1}$ containing $v_{1,1}$ cannot be in the matching. The triples of $A_{1,1}$ that must be in the matching are shown in blue on Figure 7.6. Putting the blue triples into the matching in its turn has consequences. Since the blue triples have u elements in them, all u values from $A_{1,1}$ that are shared with other crowns become unusable for connecting $A_{1,1}$ to other crowns. For instance, $u_{1,3}$ is

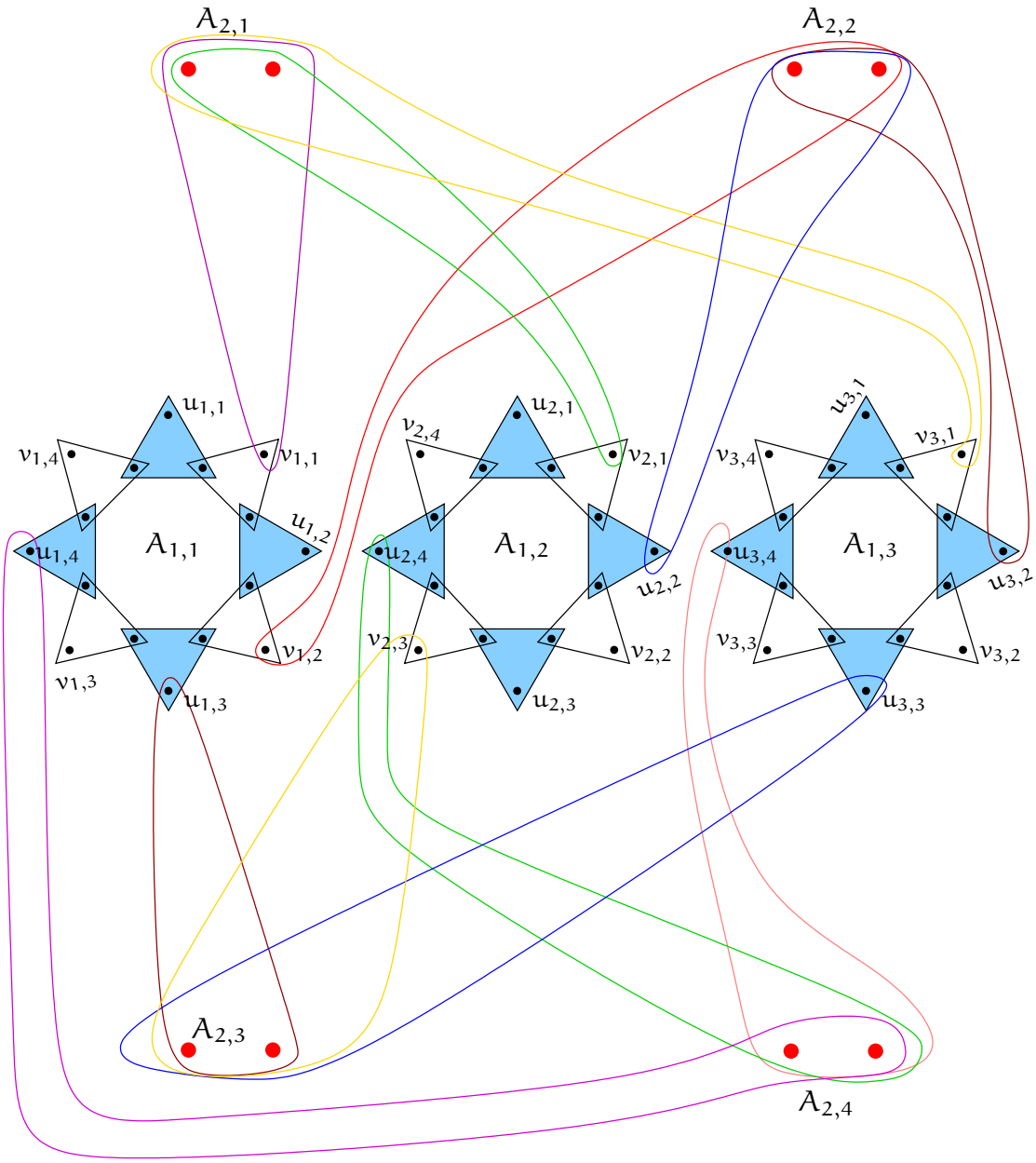


Figure 7.4: The stars and the crowns together. The CNF is $Q = (x_1 \vee x_2 \vee x_3)$ $(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

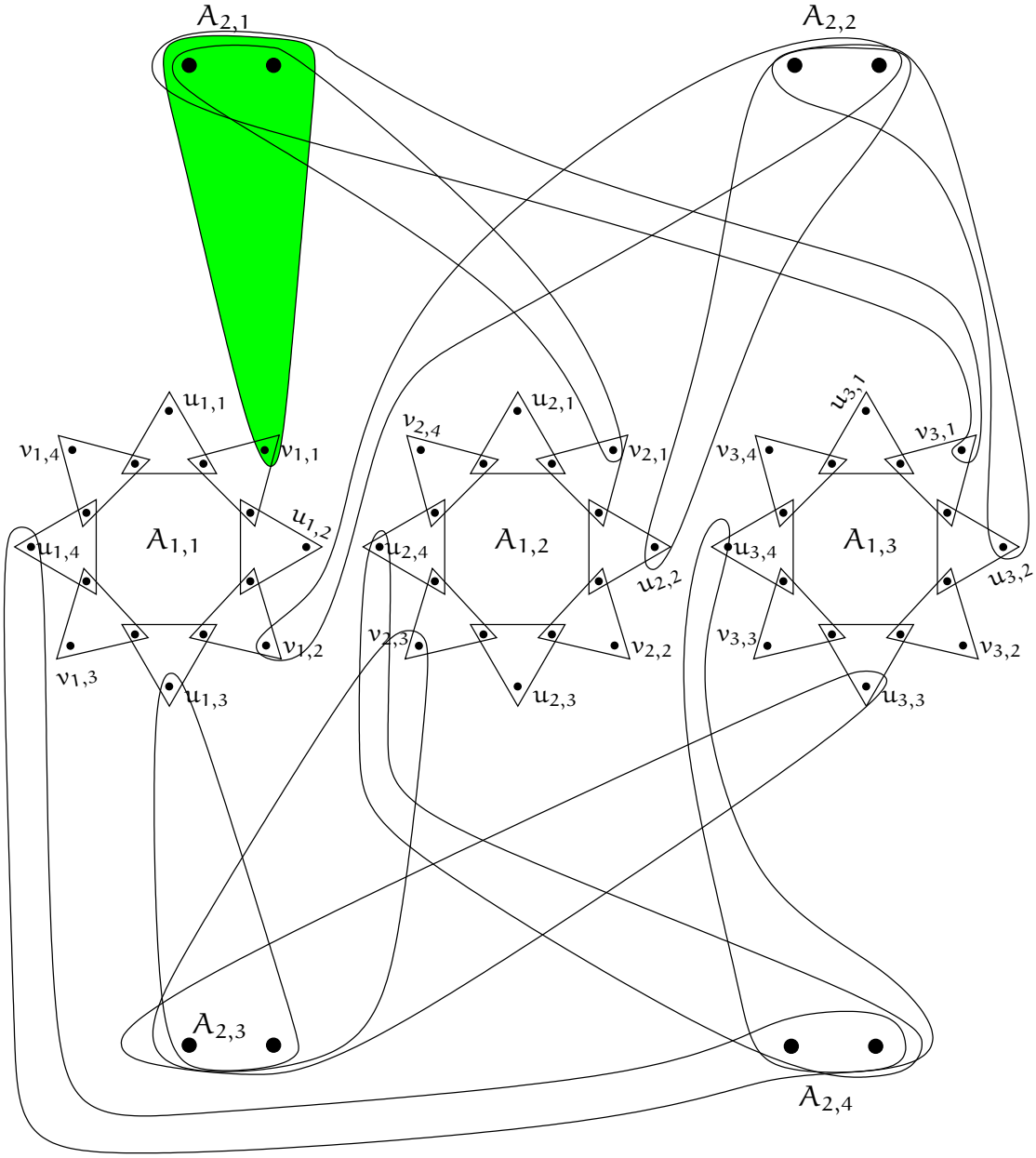


Figure 7.5: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)$ $(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. x_1 is assigned TRUE in the first clause $(x_1 \vee x_2 \vee x_3)$. We model that by choosing $(v_{1,1}, s_{1,1}, s_{2,1})$ from $A_{2,1}$ (in green).

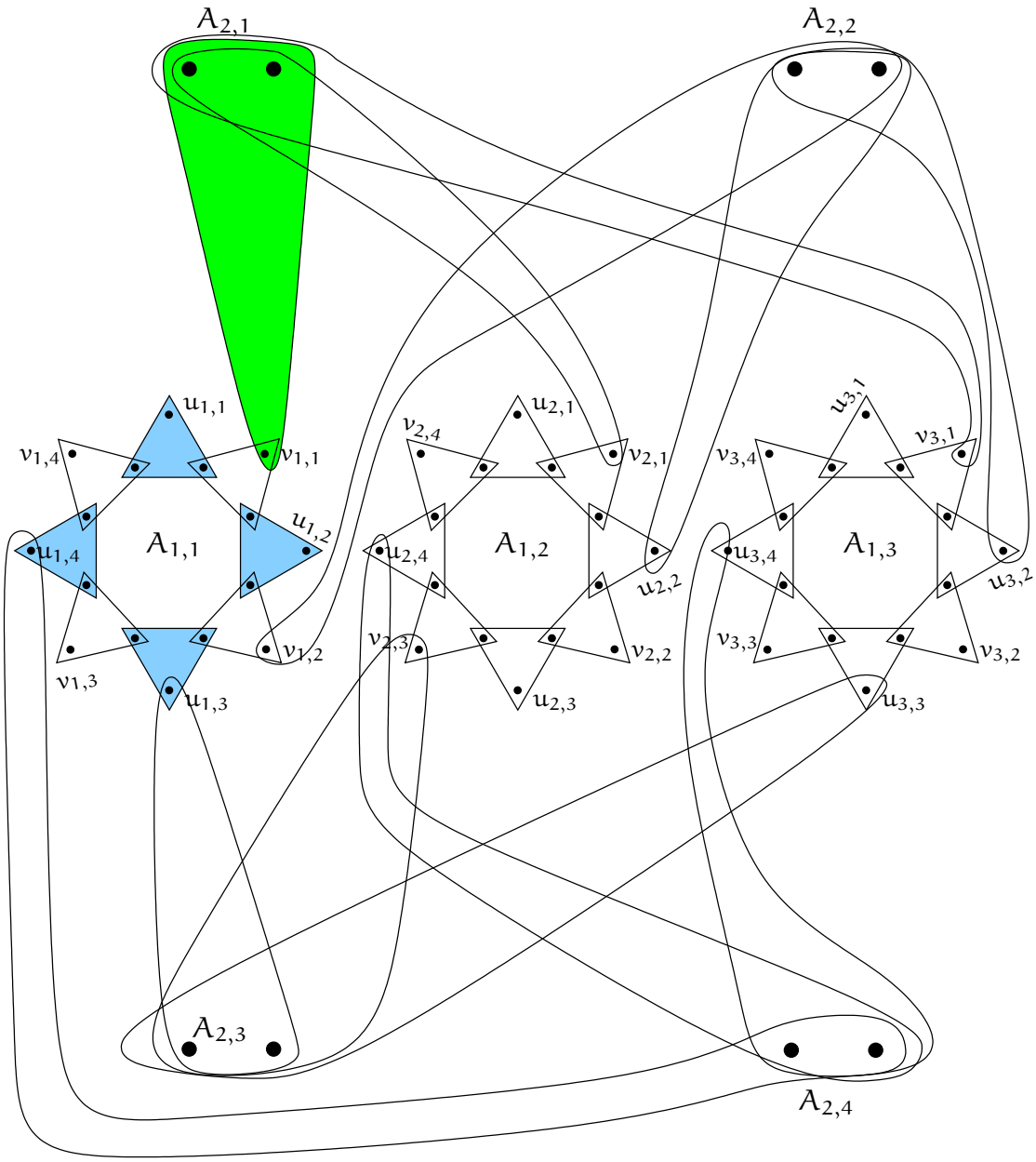


Figure 7.6: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. The green triple forces the blue triples.

shared with $A_{2,3}$ and now it is certain the triple from $A_{2,3}$ that has $u_{1,3}$ in its spike is *not* going to be in the matching. The same fact, translated into the language of the CNF, says that the choice of x_1 as literal to satisfy q_1 forbids the choice of $\overline{x_1}$ as a literal to satisfy q_3. Likewise, the fact that the triple of $A_{1,1}$ having $u_{1,4}$ is in the matching means that the triple of $A_{2,4}$ having $u_{1,4}$ is not in the matching. Figure 7.7 shows those "forbidden triples".

There has to be a literal in the second clause that is TRUE. Suppose that is x_1 again. We model that by placing $(v_{1,2}, s_{2,1}, s_{2,2})$ in the matching (see Figure 7.8). That choice forces no consequences. It *is* consistent with the previous choice but that is all. There has to be a literal in the third clause that is TRUE. Suppose that is x_2. We model that by placing $(v_{2,3}, s_{1,3}, s_{2,3})$ in the matching (see Figure 7.9) and that forces the placement of half the triples from $A_{1,2}$ in the matching. Finally we choose $\overline{x_3}$ from the fourth clause to be TRUE and model that by placing $(v_{3,4}, s_{1,4}, s_{2,4})$ in the matching (see Figure 7.10)

It is not difficult to the converse. Namely, if there is a matching, *i.e.* a subset of the constructed triples, there is a satisfying truth assignment for the CNF.

To complete the argument we have to consider a certain technicality. Not every spike of every star is covered by the matching, if one exists. On Figure 7.10 precisely 8 star spikes are outside the matching. The reader is invited to check that the triples $A_{3,u}$ and $A_{3,v}$, called collectively *garbage collection gadget*, "take care" of all star spikes that are left unshared by the construction so far.

7.2.3 3SAT $\propto$ VC

Consider an arbitrary instance of 3SAT: a set $X = \{x_1, x_2, \ldots, x_n\}$ of boolean variables and CNF $Q = \{q_1, q_2, \ldots, q_m\}$, each clause q_j being a disjunction of precisely 3 literals over X. We are going to construct a graph $G = (V, E)$ and a number k such that G has vertex cover of size $\leq k$ iff Q is satisfiable. The vertex set V is

$$V = \{u_i \mid 1 \leq i \leq n\} \cup \{v_i \mid 1 \leq i \leq n\} \cup \{z_{i,j} \mid 1 \leq i \leq 3, 1 \leq j \leq m\}$$

The edge set E is

$$E = \{(u_i, v_i) \mid 1 \leq i \leq n\} \cup \{(z_{i,j}, z_{k,j}) \mid i, k \in \{1,2,3\}, i \neq k, 1 \leq j \leq m\} \cup E'$$

where $E' = \cup_{i=1}^{m} E'_j$, where for any $1 \leq i \leq m$, E'_j is:

$$E'_j = \{(z_{1,j}, a_j), (z_{1,j}, b_j), (z_{1,j}, c_j)\}$$

The vertices a_j, b_j, and c_j correspond to the three distinct literals of clause q_j as follows. Assume some order, it does not matter what, is imposed on the literals of q_j, say $q_j = (y_1, y_2, y_3)$. If y_1 is without negation then it is some x_k and so a_j is set to u_k; otherwise a_j is set to v_k. Likewise, if y_2 is without negation then it is some x_p and so b_j is set to u_p; otherwise b_j is set to v_p; and if y_3 is without negation then it is some x_q and so c_j is set to u_q; otherwise c_j is set to v_q.

To complete the construction we must specify k as well. k is $n + 2m$.

It is obvious the construction can be done in polynomial time. Before we argue about its correctness, consider a small example. Let

$$Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

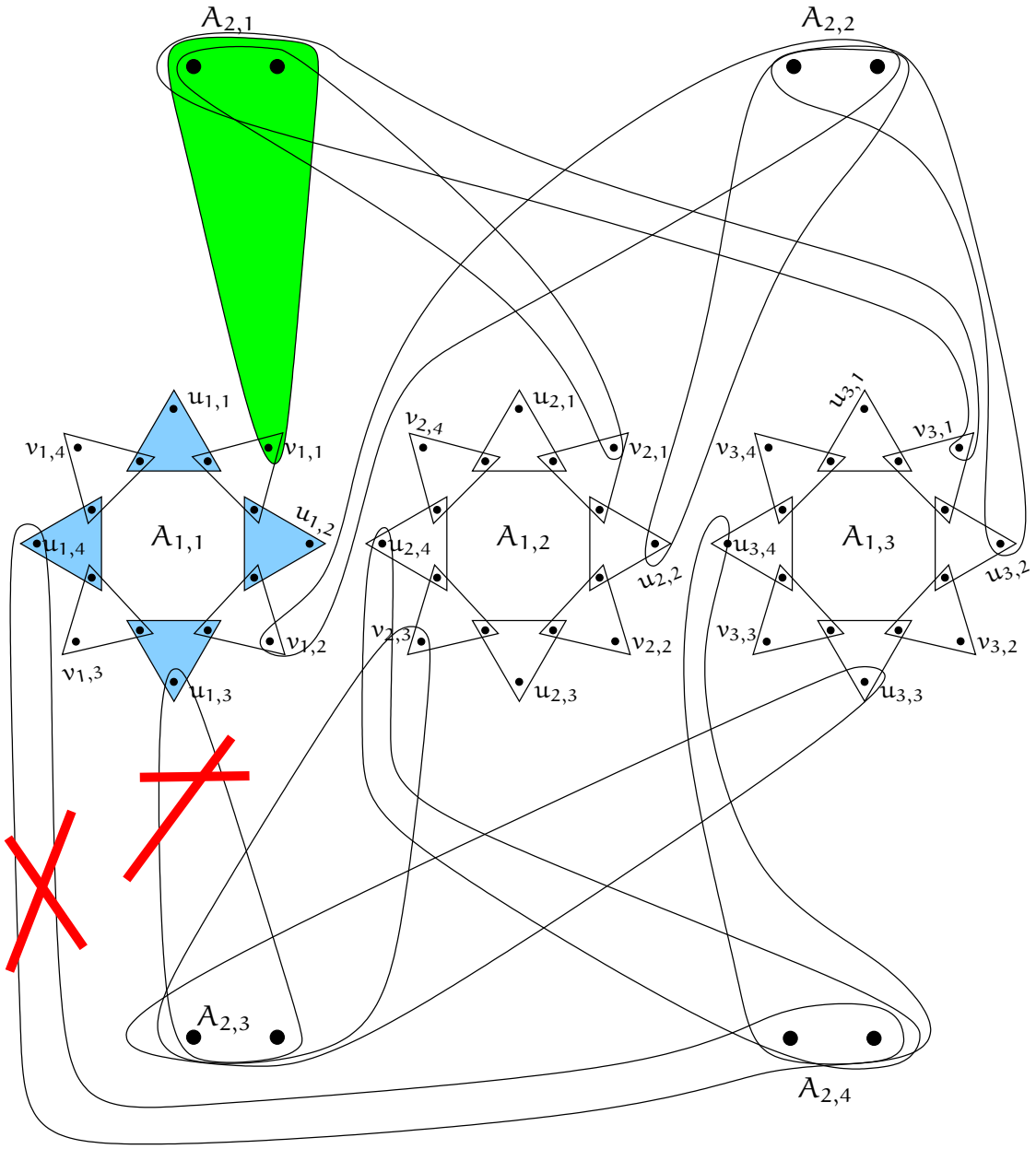


Figure 7.7: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. Once the blue triples are in the matching, the two triples crossed with red lines are out of the question.

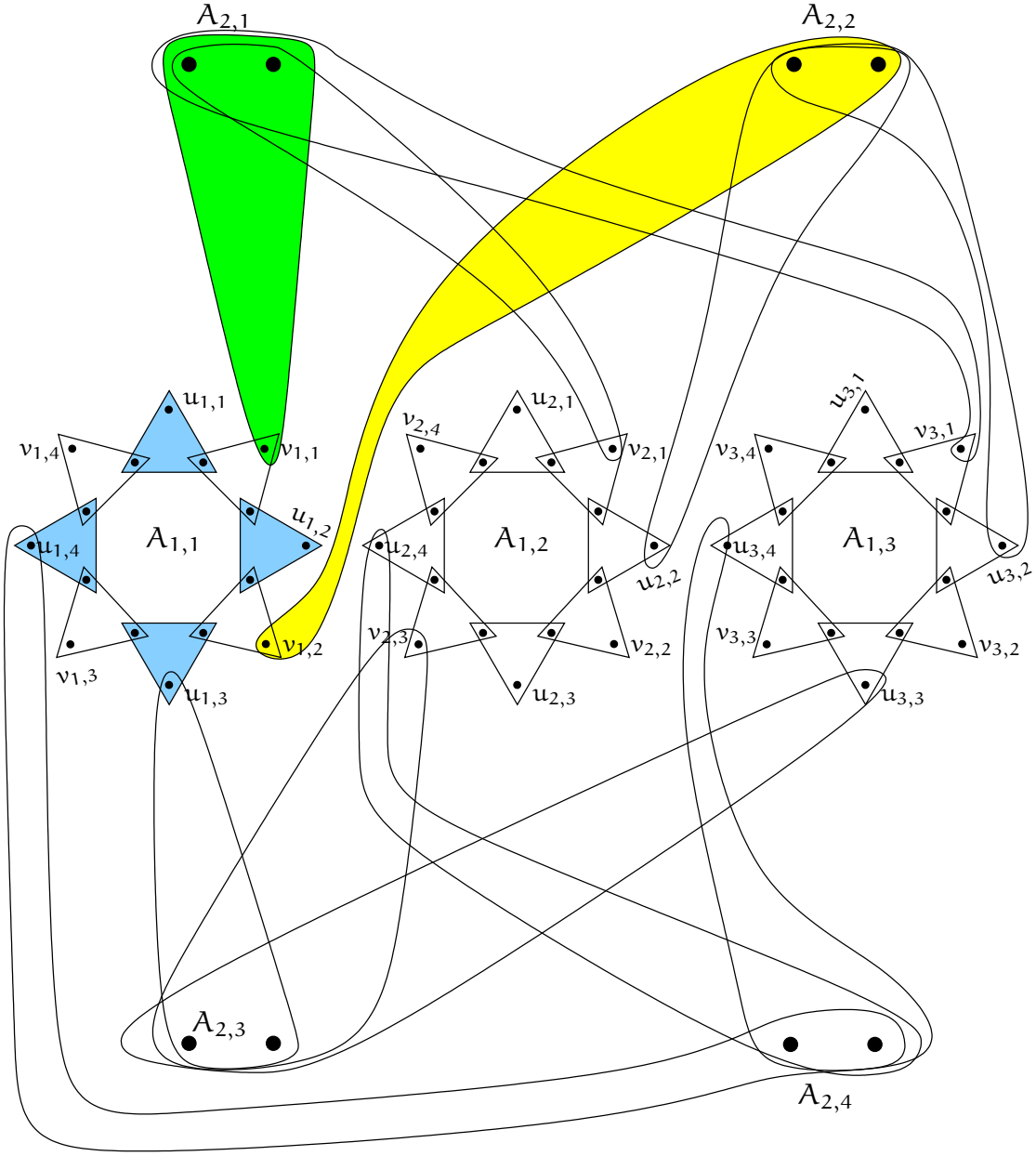


Figure 7.8: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. We put $(v_{1,2}, s_{1,2}, s_{2,2})$ in the matching (in yellow). That choice is consistent with the former choice (in green) but forces nothing.

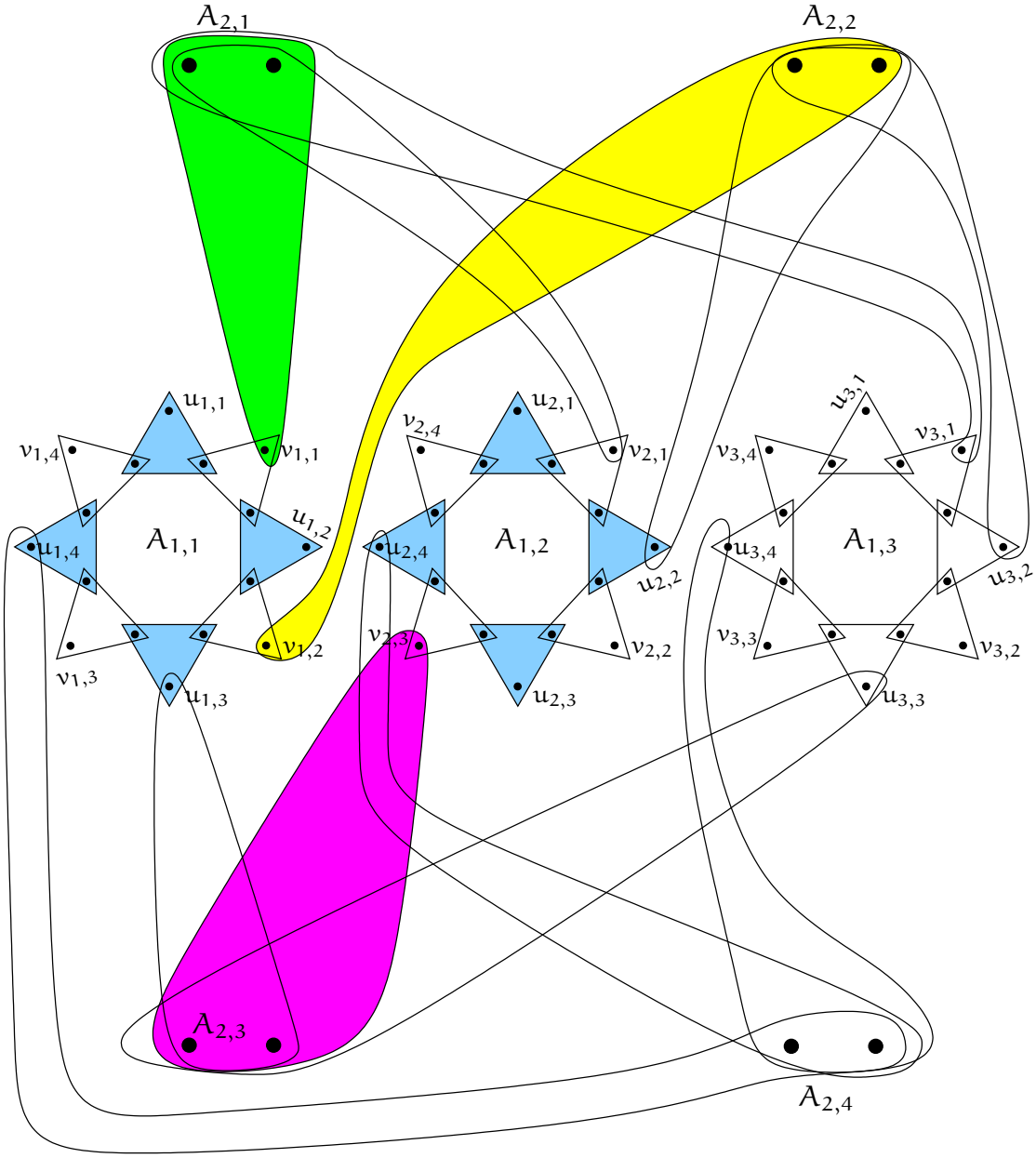


Figure 7.9: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)$ $(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. We put $(v_{2,3}, s_{1,3}, s_{2,3})$ in the matching (in magenta). That forces the addition of the blue triples from $A_{1,2}$ to the matching.

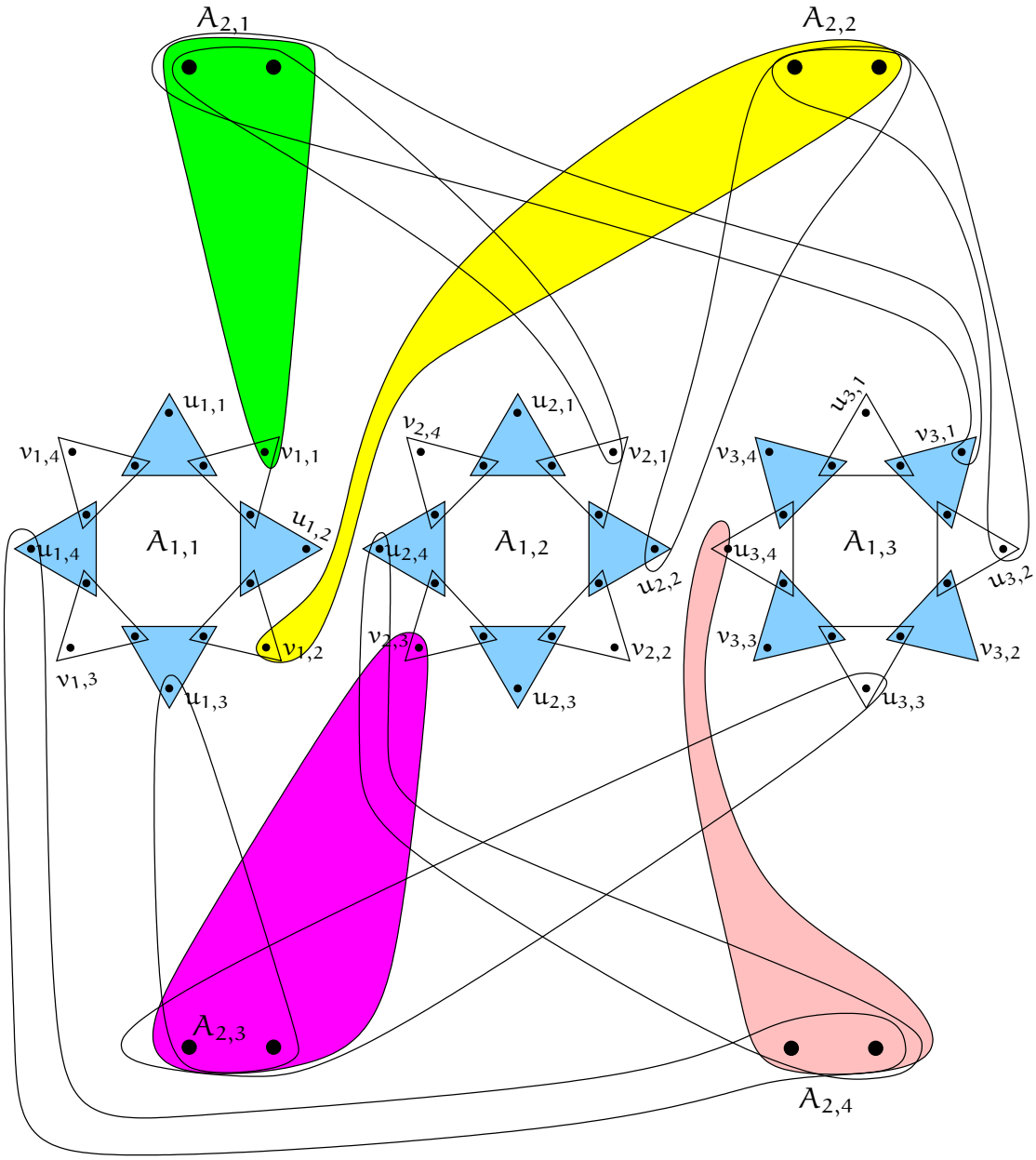


Figure 7.10: The reduction 3SAT $\propto$ 3DM. The CNF is $Q = (x_1 \vee x_2 \vee x_3)$ $(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. We put $(v_{3,4}, s_{1,4}, s_{2,4})$ in the matching (in). That forces the addition of the blue triples from $A_{1,3}$ to the matching.

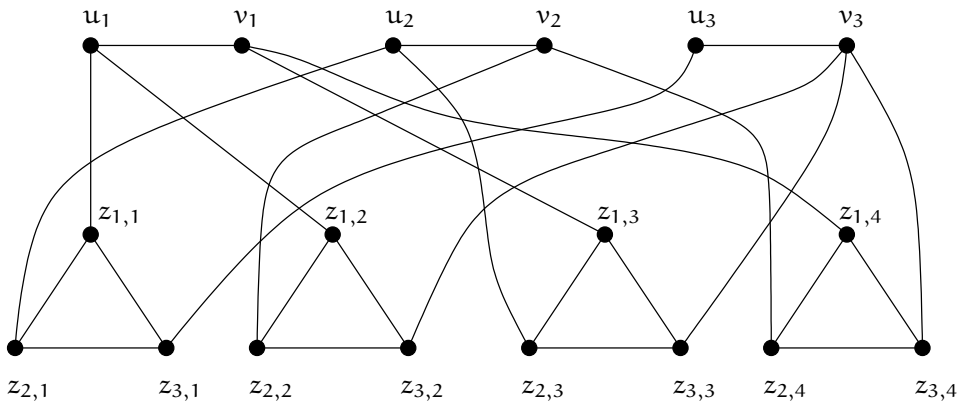


Figure 7.11: The reduction 3SAT $\propto$ VC: that is the graph corresponding to $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

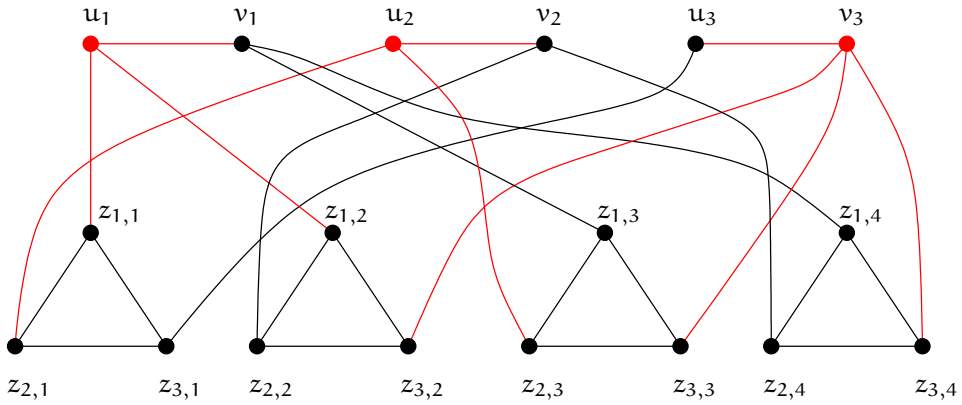


Figure 7.12: The reduction 3SAT $\propto$ VC: having chosen u_1, u_2, and v_3 to be in the vertex cover. The red edges are the "covered" ones, the black edges are yet to be covered.

Figure 7.11 shows the graph we construct. A satisfying truth assignment for it is, say $t(x_1) = t(x_2) = 1$, $t(x_3) = 0$. For each of the three edges on the top row, at least one vertex from it has to be in the vertex cover. Choosing a vertex among, say, u_1 and v_1 translates to choosing a boolean value, TRUE or FALSE, respectively, for x_1. The chosen correspondence

the u vertex corresponds to TRUE, the v vertex corresponds to FALSE

means we choose u_1, u_2, and v_3 on the top row. As a consequence, all edges with at least one endpoint in that vertex set are "covered". See Figure 7.12 for illustration. It is obvious that in each of the four 3-cliques on the bottom, we have to choose 2 vertices and place them in the vertex cover – choosing less than 2 vertices would not do. In the concrete example we can proceed as suggested by Figure 7.13. The key observation is that if the maximum number of vertices in the vertex cover is $n + 2m$, in the concrete example that is 11, we cannot afford to choose all three vertices from any of the 3-cliques. Having left out of the vertex cover precisely one vertex from each 3-clique, the edges incident to that vertex have to be taken care of by their other endpoints, namely the endpoints from the edges above. Recall that choosing vertices from the edges above translates as choosing boolean values for

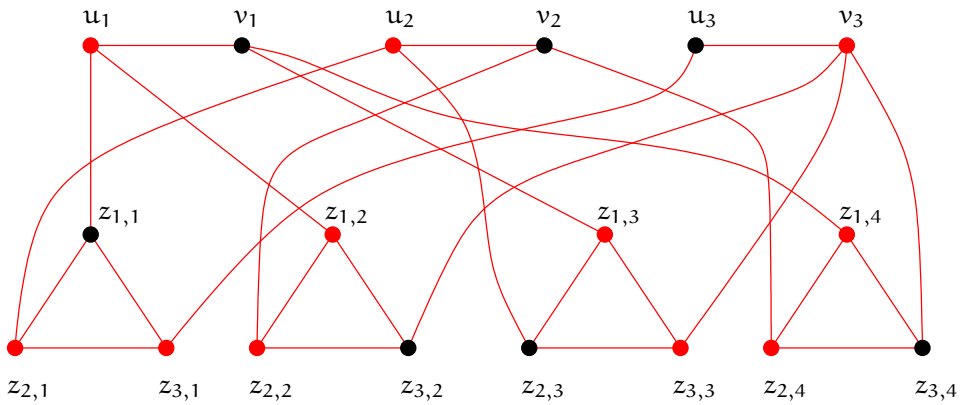


Figure 7.13: The reduction 3SAT $\propto$ VC: having chosen appropriately two vertices from each 3-clique, the graph is covered.

every variable. It follows that if there is a satisfying assignment we can make such choice for each edge above that all 3-cliques at the bottom will have a vertex, such that all edges incident to it with other endpoint some vertex from the top row, can safely be left out of a vertex cover. The argument in the opposite direction is trivial. □

7.2.4 VC $\propto$ HC

7.2.5 3DM $\propto$ Partition

Suppose $A \subseteq B \times C \times D$ is an instance of 3DM such that $|B| = |C| = |D| = n$ and $A = \{a_1, a_2, \ldots, a_m\}$. We construct a set S and function $w : S \to \mathbb{Z}^+$ such that $\exists S' \subset S$ such that $\sum_{x \in S'} w(s) = \sum_{x \in S \setminus S'} w(s)$ iff A has a matching as specified by 3DM. S contains $m + 2$ elements altogether. Say, $S = \{s_1, s_2, \ldots, s_{m+2}\}$. All we have to do is specify the weight function $w(\,)$.

First we specify the weight function for s_1, s_2, …, s_m. Assume s_j corresponds to a_j, for $1 \leq j \leq m$. Let $p = \lfloor \log_2 n \rfloor + 1$, *i.e.* the number of bits necessary to represent n in binary. Let σ_j for $1 \leq j \leq m$ be binary strings of length $3np$ that we define below. For any binary string x let $\nu(x)$ be number encoded by x in binary. We define $w(s_j)$ to be $\nu(\sigma_j)$. σ_j is the concatenation of three 3 substrings of equal length

$$\sigma_j = \sigma_{j,1}\sigma_{j,2}\sigma_{j,3}$$

$\sigma_{j,1}$ corresponds to B, $\sigma_{j,2}$ to C, and $\sigma_{j,3}$, to D. Each $\sigma_{j,i}$ is further subdivided into n substrings of length p:

$$\sigma_{j,i} = \sigma_{j,i,1}\sigma_{j,i,2}\ldots\sigma_{j,i,n}$$

$\sigma_{i,1,k}$ corresponds to b_k for $1 \leq k \leq n$, $\sigma_{i,2,k}$ corresponds to c_k for $1 \leq k \leq n$, and $\sigma_{i,3,k}$ corresponds to d_k for $1 \leq k \leq n$. In every σ_j for $1 \leq j \leq m$ we place exactly three symbols 1 in s_j; the remainder is filled in with 0's. Now we define in which positions we place the 1's. Suppose that

$$f, g, h : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n\}$$

are functions such that:

- $f(j)$ is the index of the b-element of a_j,
- $g(j)$ is the index of the c-element of a_j, and
- $h(j)$ is the index of the d-element of a_j,

for $1 \leq j \leq m$. We put 1's in σ_j at positions $b_{f(j)}$, $c_{g(j)}$, and $d_{h(j)}$. It is not difficult to see that

$$\nu(\sigma_j) = 2^{p(3n-f(j))} + 2^{p(2n-g(j))} + 2^{p(n-h(j))}$$

Let $Z = \{\sigma_j \mid 1 \leq j \leq m\}$. For each string $\sigma_j \in Z$ we define *the zones of* σ_j as the $3n$ substrings, each of length p, in it. Collectively, *the zones* are the indices of those substrings, the substrings being the abovementioned $\sigma_{j,i,k}$. If we add all σ_j strings together in the sense of binary addition, there will never be carry from one zone to the next because all zones have length p and by construction that suffices to accomodate the sum of at most n 1's. Consequently, for any subset of the set of strings, adding the elements together does not lead to carry. Let

$$H = \sum_{t=0}^{3n-1} 2^{pt}$$

H is the number that has, in binary, is represented by a string of length $3np$ that has 1 in the rightmost position of every zone and 0 elsewhere. The key observation is that for any $X \subseteq Z$, it is the case that

$$\sum_{x \in X} \nu(x) = H \text{ iff the subset of } A \text{ corresponding to } X \text{ is a matching.} \tag{7.3}$$

That observation is straightforward: if the subset of A corresponding to X is a matching then the strings of X will "cover" precisely once with 1 each position (among the $3np$ positions) that is the rightmost position of a zone; conversely, if the subset of A corresponding to X is not a matching then at least one rightmost position of a zone will not be covered or will be covered more than once – in both cases $\sum_{x \in X} \nu(x)$ cannot be H.

Finally we define the weights of the last two elements of S that we called s_{m+1} and s_{m+2}:

$$w(s_{m+1}) = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H$$

$$w(s_{m+2}) = \left(\sum_{j=1}^{m} \nu(\sigma_j)\right) + H$$

That is the end of our construction. Clearly, it can be done in polynomial time.

Now we argue about the correctness of the reduction. Note that if S has a triple that contains b_1, which certainly is the case if S has a matching, then $w(s_{m+1})$ is positive. Also note that $w(s_{m+1}) + w(s_{m+2}) = 3\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. Then

$$\sum_{j=1}^{m+2} \nu(\sigma_j) = 4\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) \tag{7.4}$$

Now suppose $\exists S' \subset S$ such that $\sum_{x\in S'} w(x) = \sum_{x\in S\setminus S'} w(x)$. Considering (7.4), it must be the case that $\sum_{x\in S'} w(x) = \sum_{x\in S\setminus S'} w(x) = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. Elements s_{m+1} and s_{m+2} cannot be both in S' or in $S\setminus S'$ because $w(s_{m+1})+w(s_{m+2}) = 3\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$. So, precisely one of S' or in $S\setminus S'$ contains s_{m+1}. Without loss of generality let S' contain s_{m+1}. Then

$$\sum_{z\in S'\setminus\{s_{m+1}\}} w(z) = \underbrace{2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)}_{\text{the total weight of } S'} - \left(\underbrace{2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H}_{\text{the value of } s_{m+1}}\right) = H$$

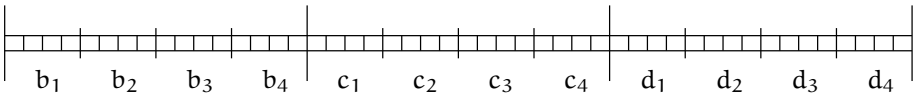
So, if there is a partition then the remaining elements from one subset must have weights that sum to H. By (7.3), a subset whose weights sum to H in the PARTITION instance is equivalent to the existence of matching in the 3DM instance.

On the other hand, if $A' \subseteq A$ is a matching, then the sum of the weights of its corresponding strings is H by (7.3), therefore the set $\{s_{m+1}\} \cup \{\sigma_j \mid a_j \in A'\}$ has total weight $H + 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right) - H = 2\left(\sum_{j=1}^{m} \nu(\sigma_j)\right)$ and thus it is an yes-instance of PARTITION.

Now we give an example. Suppose $B = \{b_1, b_2, b_3, b_4\}$, $C = \{c_1, c_2, c_3, c_4\}$, and $D = \{d_1, d_2, d_3, d_4\}$ are sets and A is an instance of 3DM:

$$A = \{(b_1, c_1, d_1), (b_3, c_2, d_2), (b_2, c_3, d_4), (b_4, c_4, d_3),$$
$$(b_1, c_2, d_2), (b_2, c_3, d_3), (b_3, c_1, d_1), (b_4, c_4, d_3)\}$$

So, $n = 4$, $m = 8$, and $p = 4$. We are going to define 8 binary strings, each one of length $3 \times 4 \times 4 = 48$. Each string is divided into 12 zones, each zone 4 positions long. The correspondence between the zones and the elements of B, C, and D is shown below:



The 8 strings are shown on Figure 7.14 with the correspondence between them and the elements of A. Rather than writing zeroes and ones, we represent the strings as arrays of positions that are empty (zero) or filled in (one). The rightmost position is the least significant bit, *etc.*. The concrete numerical values of the strings are not important and we leave them out. The reader may verify that the string on top, if read in binary, has numerical value $2^{12} + 2^{12+16} + 2^{12+32} = 17\,592\,454\,483\,968$, *etc.* Figure 7.14 also shows the result of the binary addition denoted by Σ. Clearly, there is no carry out of any of the zones. The 12 zones are not shown explicitly, only the regions corresponding to B, C, and D are outlined. The value H written as a binary string is shown on Figure 7.15. Verify the crucial observation (7.3): the instance has a matching, namely

$$A' = \{(b_1, c_1, d_1), (b_3, c_2, d_2), (b_2, c_3, d_4), (b_4, c_4, d_3)\}$$

and indeed the strings corresponding to the triples in A' (the four top rows on Figure 7.14) sum precisely to H.

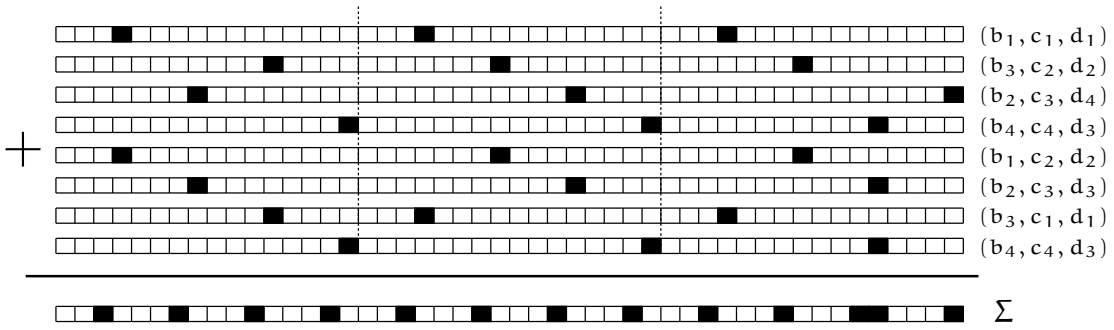


Figure 7.14: Each array of boxes represents a binary string, an empty box corresponding to a zero and a filled-in box corresponding to a one.

Figure 7.15: The number $H = \sum_{t=0}^{3n-1} 2^{pt}$ in binary.

7.2.6 VC $\propto$ DS

Assume $\langle G = (V, E), k\rangle$ is an instance of VC. Assume without loss of generality G has no vertices of degree 0 – if there were such isolated vertices, each of them would be necessarily in any dominating set. Construct the following graph $G' = (V \cup V', E \cup E')$ where V' and E' are obtained as follows. For each edge $e = (u, v)$ from E, add a new vertex w_e to V' and add two new edges (u, w_e) and (w_e, v) to E'. We claim G' has dominating set of size s iff G has vertex cover of size k.

In one direction, assume $U \subset V$ is a vertex cover for G. We prove U is a dominating set for G'. Consider any vertex z of G'. If z is from U then z is dominated, being in the dominating set. If z is one of the newly added vertices V' then z has a neighbour in U because by construction z has two neighbours precisely and they are the endpoints of some original edge e from E and since U is a vertex cover for G at least one endpoint of e is in U. Finally, if z is from $V \setminus U$ note that z has a neighbour in U because there is at least one edge incident with z and it must have an endpoint in U.

In the other direction, assume U is a dominating set for G'. It is easy to see that if there are vertices from V' in U, for each $w_e \in V' \cap U$ we can substitute w_e with either of its two neighbours in the dominating set; precisely the same vertices are dominated after that operation and the size of the dominating set does not increase. So we can assume without loss of generality that $V' \cap U = \emptyset$. Since U is a dominating set for G',

> for every vertex from $V \cup V'$, it is in U or has a neighbour in U.

In particular,

> for every vertex from V', it is in U or has a neighbour in U.

But we just pointed out a construction, according to which it is possible to assume $V' \cap U = \emptyset$. The statement becomes

> for every vertex from V' it has a neighbour in U.

Clearly, that implies for every edge from E at least one of its endpoints in is U. Then by definition U is a vertex cover for G.

7.2.7 HC $\propto$ TSP

Suppose $G(V, E)$ is an instance of HC and $V = \{v_1, v_2, \ldots, v_n\}$. Construct an instance of TSP as follows. Let the locations be $\{c_1, c_2, \ldots, c_n\}$. Define the distances thus

$$\forall c_i, c_j : \text{dist}(c_i, c_j) = \begin{cases} 0, & \text{if } i = j, \\ 1, & \text{if } i \neq j \text{ and } (v_i, v_j) \in E \\ 2, & \text{else} \end{cases}$$

Let the bound B be n.

Assume G has a Hamiltonian cycle. Then in the TSP instance there is a permutation of the locations $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)}$ such that $\sum_{i=1}^{n} \text{dist}(c_{\pi(i)}, c_{\pi(i+1)}) = n - 1$, therefore $\left(\sum_{i=1}^{n} \text{dist}(c_{\pi(i)}, c_{\pi(i+1)})\right) + \text{dist}(c_{\pi(n)}, c_{\pi(1)}) = n$. In the other direction, consider any permutation π such that $\left(\sum_{i=1}^{n} \text{dist}(c_{\pi(i)}, c_{\pi(i+1)})\right) + \text{dist}(c_{\pi(n)}, c_{\pi(1)}) \leq n$. Since the distances are either 1 or 2, by Dirichlet's principle it must be the case that all weights are ones, therefore in G there is a simple cycle containing all vertices that corresponds to the said permutation. But that is a Hamiltonian cycle. □

7.2.8 Partition $\propto$ Knapsack

Suppose $A = \{a_1, a_2, \ldots, a_n\}$ is a set and $w : A \to \mathbb{Z}^+$ is a weight function on it. Let the instance of Knapsack have the same set and weight function, and let the value function be $v(a) = w(a), \forall a \in A$. Let size constraint B and the value goal K be $B = K = \frac{1}{2} \sum_{a \in A} w(a)$.

Assume the instance of Partition is an yes-instance. Then $\exists A' \subset A$ such that $\sum_{a \in A'} = \frac{1}{2} \sum_{a \in A} w(a)$. It is a trivial observation that then the size and value constraints are satisfied. In the other direction, if the said size and value constraints are satisfied, which means the instance is an yes-instance of Knapsack, then the same subset of A over which those constraints are satisfied, induces a partition such that the sum of its weights equals the sum of the weights of its complement. □

7.2.9 3SAT $\propto$ Clique

Clique and VC are very closely related. Indeed, a minimum vertex cover induces a maximum independent set which in its turn induces a maximum clique in the complement graph, so the proof that 3SAT $\propto$ VC implies almost immediately that 3SAT $\propto$ IS and 3SAT $\propto$ Clique. However, we show a direct reduction from 3SAT to Clique.

Consider an arbitrary instance of 3SAT: a set $X = \{x_1, x_2, \ldots, x_n\}$ of boolean variables and CNF $Q = \{q_1, q_2, \ldots, q_m\}$, each clause q_j being a disjunction of precisely 3 literals over X. We are going to construct an instance of Clique: a graph $G = (V, E)$ and a number k such that Q is satisfiable iff G has clique of size $\geq k$. V consists of $3m$ vertices. The vertices of G are partitioned into triples where triple number j corresponds to q_j, for $1 \leq j \leq m$. For brevity we use the name of the literals from Q to name the vertices; clearly we can get vertices with the same name in this way.

We put an edge between vertices u and v iff

- u and v are from different triples, and
- it is not the case that u and v are named by "opposite" literals of the same variable.

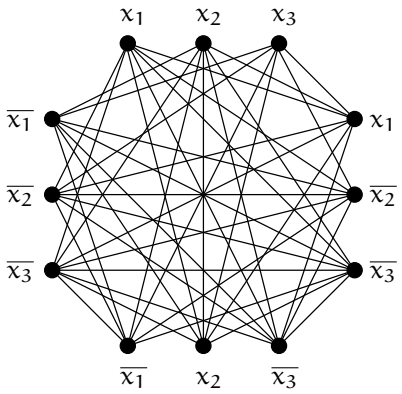


Figure 7.16: The graph that corresponds to $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$ in the reduction 3SAT $\propto$ CLIQUE.

The number k is m. We claim the graph has an m-clique whenever the CNF is satisfiable. Indeed, let Q be satisfiable. Then in every clause there is a literal that is assigned TRUE. Imagine the corresponding vertices in the graph and note that every two of them are connected by an edge because they

- belong to distinct triples, and
- they cannot possibly be named by opposite literals of the same variable – that would mean the truth assignment assigns both TRUE and FALSE to their variable.

In the opposite direction, assume H is a clique of size n in G. Assign TRUE to all its literals[†]. But that means that every clause of Q has at least one literal that is TRUE so the whole clause is satisfied.
For example, let

$$Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

The graph that corresponds to Q is shown on Figure 7.16. A satisfying assignment is, for instance, $t(x_1) = t(x_2) = 1$, $(x_3) = 0$. Under it, x_1 from the first clause, x_1 from the second clause, x_2 from the third clause and $\overline{x_3}$ from the fourth clause are satisfied. The desired 4-clique is shown on Figure 7.17. □

7.2.10 3SAT $\propto$ EDP

Consider an arbitrary instance of 3SAT: a set $X = \{x_1, x_2, \ldots, x_n\}$ of boolean variables and CNF $Q = \{q_1, q_2, \ldots, q_m\}$, each clause q_j being a disjunction of precisely 3 literals over X. We are going to construct an instance of EDP such that it is an yes-instance iff Q is satisfiable.

For each variable x_i we construct a vertex tuple (s_i^v, t_i^v) and two directed vertex-disjoint—except for the common endpoints—paths $p_i^0 = s_i, \ldots, t_i$ and $p_i^1 = s_i, \ldots, t_i$ of length $m + 2$ each, *i.e.* with m internal vertices. Furthermore, p_j^a is vertex-disjoint with

[†]That does not mean to assign TRUE to the variables: those variables whose literals are negated will be assigned FALSE.

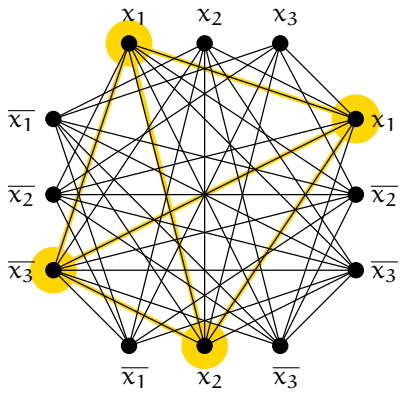


Figure 7.17: A 4-clique in the graph that corresponds to $Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

p_s^b, for $a, b \in \{0, 1\}$ and $1 \leq j < s \leq m$. p_i^0 models the assignment of 0 to x_i and p_i^1 models the assignment of 1 to x_i. The internal vertices of p_i^0 are named $u_i^1, \ldots, u_i^{m+1}$, in that order away from s_i^v, and the internal vertices of p_i^1 are named $v_i^1, \ldots, v_i^{m+1}$, in that order away from s_i^v. The edge (u^j, u_i^{j+1}) models the appearance of x_i is q_j without negation and the edge (v^j, v_i^{j+1}) models the appearance of x_i is q_j with negation.

For example, let the CNF be

$$Q = (x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

Our construction at this stage has resulted in the graph shown on Figure 7.18. Then for each clause $q_j = (y_p, y_q, y_r)$ we place another vertex tuple (s_j^c, t_j^c) and six edges:

- either (s_j^c, u_p^j) or (s_j^c, v_p^j) depending on whether y_p is negated or not, respectively,
- either (s_j^c, u_q^j) or (s_j^c, v_q^j) depending on whether y_q is negated or not, respectively,
- either (s_j^c, u_r^j) or (s_j^c, v_r^j) depending on whether y_r is negated or not, respectively,
- either (u_p^{j+1}, t_j^c) or (v_p^{j+1}, t_j^c) depending on whether y_p is negated or not, respectively,
- either (u_q^{j+1}, t_j^c) or (v_q^{j+1}, t_j^c) depending on whether y_q is negated or not, respectively,
- either (u_r^{j+1}, t_j^c) or (v_r^{j+1}, t_j^c) depending on whether y_r is negated or not, respectively.

Figure 7.19 shows the result of the those edges' additions to our example. Figure 7.20 shows that in our example there are indeed 7 edge-disjoint paths, each with one endpoint an s vertex amd the other endpoint, a t vertex. Furthermore, those paths are constructed according to the satisfying truth assignment $t(x_1) = t(x_2) = 1$ and $t(x_3) = 0$. Recall that we adoptded the convention that the u intermediate vertices model the literals that are not negated and the v vertices model the negated literals. Some clauses may be satisfied by more than one literal but the paths we construct represent exactly one satisfying literal per clause. Under the currently considered truth assignment, we choose literal x_1 for $(x_1 \vee x_2 \vee x_3)$, literal x_1 for $(x_1 \vee \overline{x_2} \vee \overline{x_3})$, literal x_2 for $(\overline{x_1} \vee x_2 \vee \overline{x_3})$, and literal $\overline{x_3}$ for $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

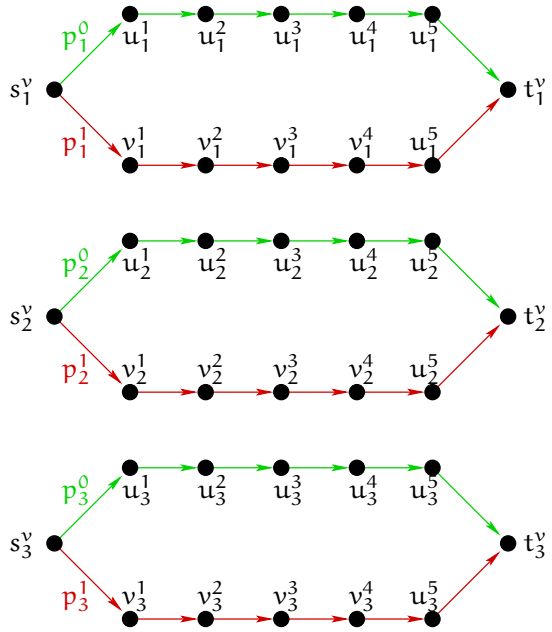


Figure 7.18: The reduction 3SAT $\propto$ EDP: the first stage of the construction.

The four green paths on Figure 7.20 represent those choices for satisfying literals: the dark green path on top corresponds to the choice of x_1 for the first clause (the vertices s_1^c and t_1^c model the first clause), the bright green path on top corresponds to the choice of x_1 for the second clause, *etc.* Having placed the green paths in an edge-disjoint manner, we place the three red paths that model the truth assignments: the red path on top models the assignment of TRUE to x_1, *etc.* The red paths can be placed in this manner iff for every component (see Figure 7.18 that depicts only those components), the green paths leave the either the u vertices or the v vertices "free". In other words, if we have already satisfied the clauses in a consistent way.

Now we prove that Q is satisfiable iff the instance of EDP is an yes-instance.

$\Longrightarrow$ Suppose there is a satisfying truth assignment t for Q. Choose a satisfying literal for every clause q_j and call the variable of that literal, *the satisfying variable of* q_j. For each (s_i^v, t_i^v) tuple use the p_i^1 path if $t(x_i) =$ TRUE and the p_i^0 path if if $t(x_i) =$ FALSE. And for each (s_j^c, t_j^c) tuple, if x_k is its satisfying variable, use either the three-edge path $s_j^c, u_k^j, u_{k+1}^j, t_j^c$ in case x_k is not negated in q_j, or the three-edge path $s_j^c, v_k^j, v_{k+1}^j, t_j^c$, in case x_k is negated in q_j. It is obvious that no path connecting any (s_i^v, t_i^v) can have a common edge with any path connecting any (s_j^c, t_j^c). Furthermore, every two paths connecting (s_j^c, t_j^c) and $(s_{j'}^c, t_{j'}^c)$ that are used, are edge-disjoint for $1 \leq j < j' \leq m$:

- if we use different satisfying variables for q_j and $q_{j'}$ the claim is obvious,
- if we use the same satisfying variable x_k for q_j and $q_{j'}$, indeed the middle edge in both of them is from p_k^0 or p_k^1, whichever one is applicable, but they do not use the same edge from that path.

Finally, it is obvious the path connecting (s_i^v, t_i^v) is disjoint with the path connecting $(s_{i'}^v, t_{i'}^v)$, for $1 \leq i < i' \leq n$.

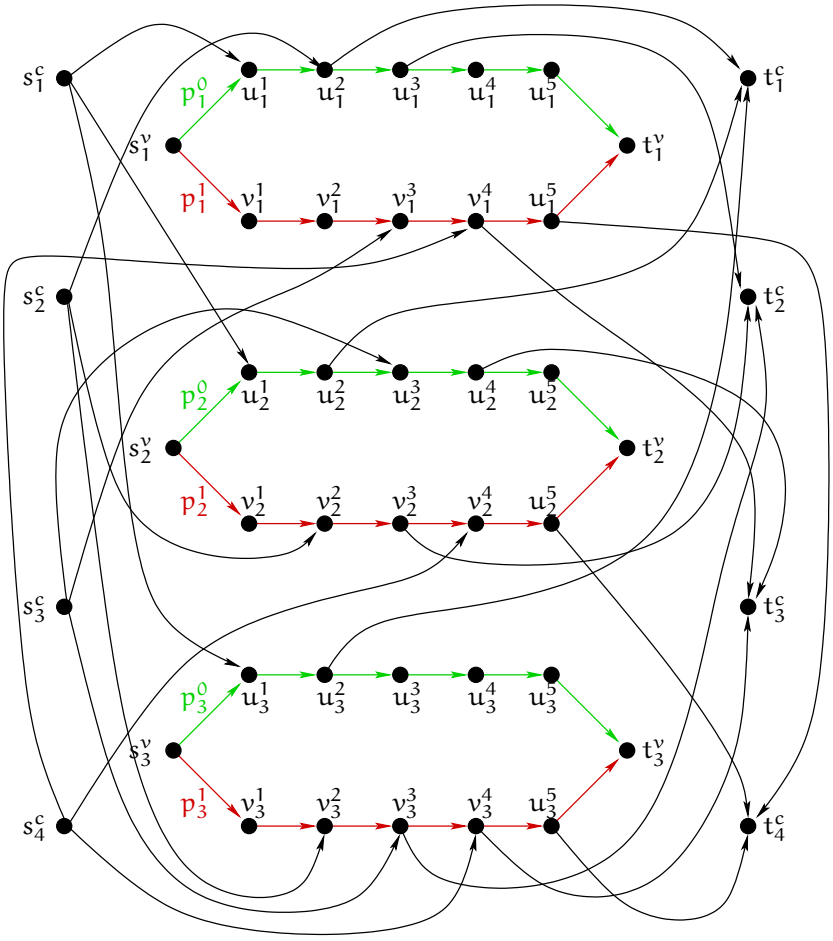


Figure 7.19: The reduction 3SAT $\propto$ EDP: the second stage of the construction. The CNF is $(x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

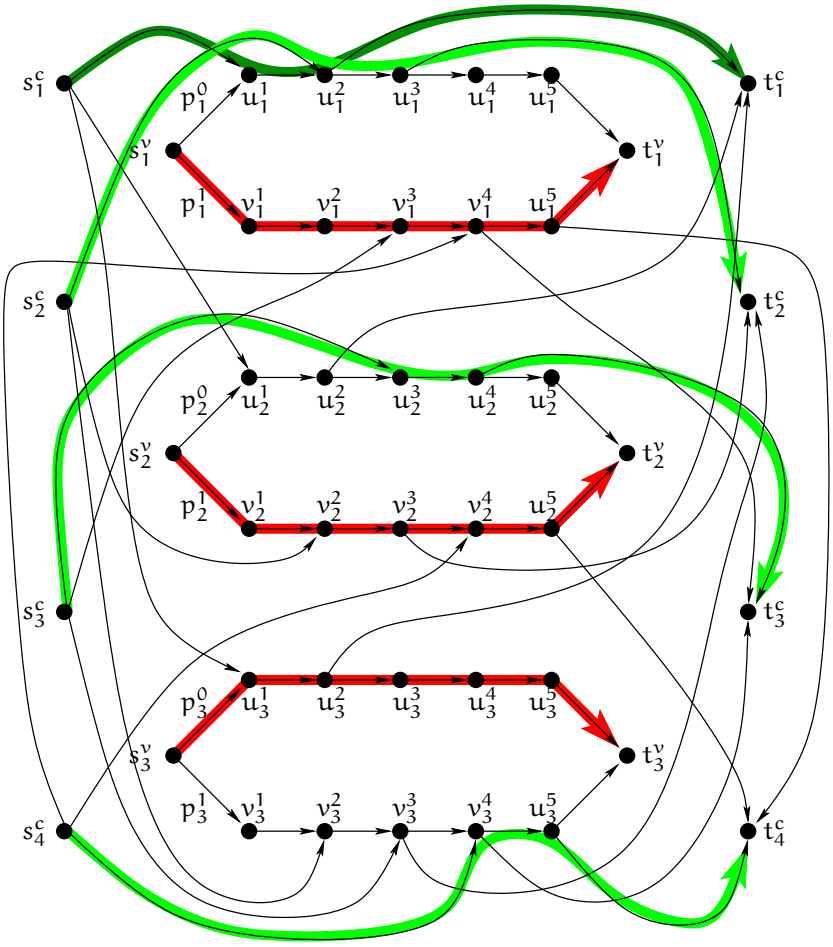


Figure 7.20: The reduction 3SAT $\propto$ EDP. The CNF is $(x_1 \vee x_2 \vee x_3)(x_1 \vee \overline{x_2} \vee \overline{x_3})(\overline{x_1} \vee x_2 \vee \overline{x_3})(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$. There are 7 edge-disjoint paths, each with one endpoint an s vertex amd the other endpoint, a t vertex.

$\Longleftarrow$ Suppose the instance of EDP is an yes-instance. Then for every i, $1 \leq i \leq n$, the tuple (s_i^v, t_i^v) is connected and so at least one of the paths p_i^0 or p_i^1 is used to that end. If any tuple (s_j^c, t_j^c) is connected via an edge from p_i^0 or p_i^1, it is clear that precisaly one of p_i^0 or p_i^1 is used for connecting (s_i^v, t_i^v) and edges from the other one are used for connecting one or more tuples (s_j^c, t_j^c). That implies unambiguously a truth assignment for the variables corresponding to the (s_i^v, t_i^v), such that edges either from p_i^0 or from p_i^1 are used for connecting tuples (s_j^c, t_j^c). On the other hand, for every tuple (s_i^v, t_i^v) such that no edge either from p_i^0 or from p_i^1 are used for connecting tuples (s_j^c, t_j^c), we have the freedom to choose the truth assignment for the corresponding x_i. In any event, a satisfying truth assignment exists.

The fact that the construction can be performed in polynomial time is obvious.

Problem 140. *It is well-known that the version of the disjoint-paths problem in which* $s_1 = s_2 = \ldots = s_k$ *and* $t_1 = t_2 = \ldots t_k$ *is solvable in polynomial time, for instance with max-flow algorithms [CLR00]. As pointed out in [Pap95, pp. 204], only* $s_1 = s_2 = \ldots = s_k$ *implies the problem is in* $\mathcal{P}$.

Let us call the version of EDP in which $s_1 = s_2 = \ldots = s_k$ *and* $t_1 = t_2 = \ldots t_k$, RESTRICTED EDP *or shortly* REDP. *Explain what is wrong with the following "reduction"* EDP $\propto$ REDP. *Starting with any instance of EDP, add two new vertices* α *and* ω *to the graph and add the edges* (α, s_1), (α, s_2), ..., (α, s_k), (t_1, ω), (t_2, ω), ..., (t_k, ω). *Clearly, there are* k *edge-disjoint paths from* α *to* ω *in the new graph iff there are* k *edge disjoint paths in the original graph, each path connecting an* s_i *vertex to a distinct* t_j *vertex.*

Solution:
The EDP problem requires specifically that the edge-disjoint paths connect s_1 to t_1, s_2 to t_2, ..., s_k to t_k. The incorrect "reduction" constructs paths that are edge-disjoint and connect each s_i vertex to a distinct t_j but that does not suffice. Each s_i must be connected to t_i in order to have an yes-instance of EDP, the associations between the s and t vertices being defined by the tuples of the instance of EDP. □

7.2.11 VDP $\propto$ EDP

Suppose we are given an instance of VDP: a directed graph $G = (V, E)$ and a list of tuples (s_i, t_i), (s_2, t_2), ..., (s_k, t_k). For every vertex $u \in V$ do the following. Let $A = \{v \in V \,|\, (v, u) \in E\}$ and $B = \{v \in V \,|\, (u, v) \in E\}$. Delete u from G, add two new vertices z' and z'' to G, add (z', z'') to E, and add $\{(v, z') \,|\, v \in A\}$ and $\{(z'', v) \,|\, v \in B\}$ to E. Once added, those new vertices are never deleted. Let us call z', *the in-substitute of* u and z'', *the out-substitute of* u

The instance of EDP is the obtained graph G' together with the k tuples of vertices defined as follows. For any tuples (s_i, t_i) of the original graph (the VDP instance), add the tuples (x_i, y_i) to the EDP instance, where x_i is the in-substitute of s_i and y_i is the out-substitute of t_i.

Suppose the instance of VDP is an yes-instance. Let p_i be a path in G from s_i to t_i, for $1 \leq i \leq k$, such that all p_i are pairwise internally vertex-disjoint. The images of p_i, for $1 \leq i \leq k$, in G' are clearly edge-disjoint.

On the other hand, for every pair of paths among p_i, for $1 \leq i \leq k$, that share an internal vertex in G, it is the case that their images share an edge in G'. Figure 7.21 demonstrates

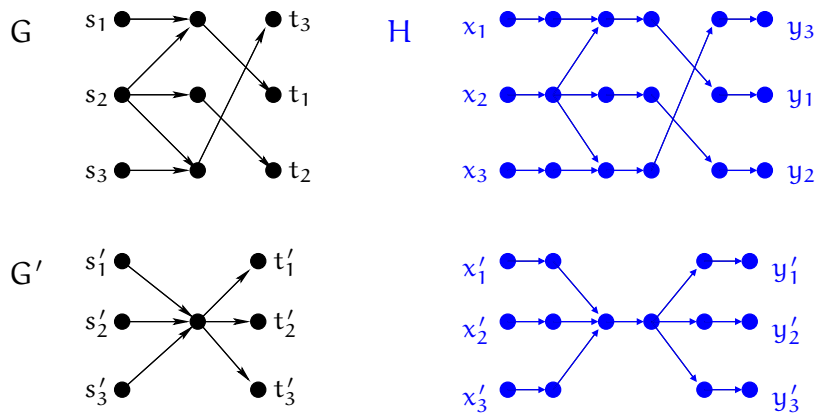


Figure 7.21: A yes-instance $\langle G, \langle (s_1, t_1), (s_2, t_2), (s_3, t_3) \rangle \rangle$ of VDP is mapped on a yes-instance $\langle H, \langle (x_1, y_1), (x_2, y_2), (x_3, y_3) \rangle \rangle$ of EDP. A no-instance $\langle G', \langle (s'_1, t'_1), (s'_2, t'_2), (s'_3, t'_3) \rangle \rangle$ of VDP is mapped on a no-instance $\langle H', \langle (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3) \rangle \rangle$ of EDP.

the construction from an yes-instance to an yes-instance and from a no-instance to a no-instance.

7.2.12 EDP $\propto$ VDP

Definition 11. *Suppose* $G = (V, E)$ *is a graph.* The line graph of G *is the graph* $G' = (V', E')$ *where* V' *is a vertex set such that there exists a bijection* $f : E \to V'$ *and* E' *is defined as follows:*

$$u, v \in V' : (u, v) \in E' \text{ iff } f^{-1}(u) \text{ and } f^{-1}(v) \text{ are coincident.}$$

□

Definition 12. *Suppose* $G = (V, E)$ *is a directed graph.* The line graph of the directed graph G *is the directed graph* $G' = (V', E')$ *where* V' *is a vertex set such that there exists a bijection* $f : E \to V'$ *and* E' *is defined as follows:*

$$u, v \in V' : (u, v) \in E' \text{ iff } f^{-1}(u) = (x, y) \text{ and } f^{-1}(v) = (y, z) \text{ for the same vertex } y.$$

□

Suppose we are given an instance of EDP: a directed graph $G = (V, E)$ and a list of tuples (s_i, t_i), (s_2, t_2), ..., (s_k, t_k). Construct the following instance of VDP: the line graph of G, call it G', together with the list of tuples (x_i, y_i), $1 \leq i \leq k$, where x_i is the vertex of G' corresponding to the edge of G whose initial vertex (in G) is s_i and y_i is the vertex of G' corresponding to the edge of G whose ending vertex (in G) is t_i.

Assume the instance of EDP is an yes-instance. Let p_i, for $1 \leq i \leq k$, be a set edge-disjoint paths certifying that is indeed a yes-instance. Consider any two of them. If they are internally vertex-disjoint then their images in G' are vertex-disjoint, too, for obvious

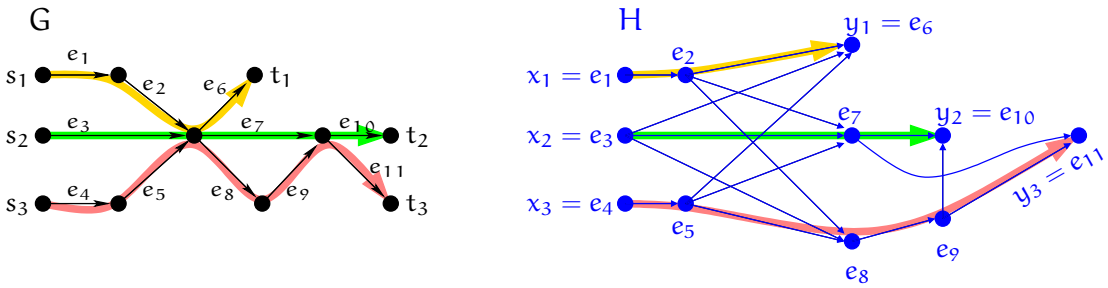


Figure 7.22: G is a yes-instance of EDP. Its line graph H is a yes-instance of VDP. The edge-disjoint paths in G are mapped on vertex-disjoint paths in H.

reasons. If they share vertices but do not share edges then their images are vertex-disjoint because for each shared internal vertex z, the predecessors of z and the successors of z in the paths in G are distinct—otherwise the paths would have a common edge—therefore the images of the paths in G' each have an edge connecting the images of the respective predecessor and successor edges, and those are distinct. Figure 7.22 shows the construction for an yes-instance.

Assume the instance of EDP is a no-instance. Then all sets of k paths, each one connecting the vertices from exactly one tuple, are such that there are always two paths that share an edge. It is a trivial observation that their images in G' share a vertex.

Chapter 8

Appendix

Problem 141. *Prove that*

$$\sum_{k=1}^{n} \lg k \asymp n \lg n$$

Solution:
One way to solve it is to see that the sum is $\sum_{k=1}^{n} \lg k = \lg(n!)$ and then use Problem 1.48 on page 11. There is another way. Let $m = \lfloor \frac{n}{2} \rfloor$, $A = \sum_{k=1}^{m-1} \lg k$, and $B = \sum_{k=m}^{n} \lg k$. First we prove that $B \asymp n \lg n$.

$$\sum_{k=m}^{n} \lg m \leq \sum_{k=m}^{n} \lg k \leq \sum_{k=m}^{n} \lg n \quad \Leftrightarrow$$

$$(\lg m) \sum_{k=m}^{n} 1 \leq B \leq (\lg n) \sum_{k=m}^{n} 1 \quad \Leftrightarrow$$

$$\underbrace{\left(\lg \left\lfloor \frac{n}{2} \right\rfloor\right)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{C} \leq B \leq \underbrace{(\lg n)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{D}$$

Clearly, $C \asymp n \lg n$ and $D \asymp n \lg n$. It must be the case that $B \asymp n \lg n$. Now we prove that $A \preceq B$. A has $\frac{n-2}{2}$ terms, in case n is even, and $\frac{n-3}{2}$ terms, in case n is odd. In any event, A has less terms than B. Furthermore, every term of A is smaller than any term of B. It follows $A \preceq B$. Since $\sum_{k=1}^{n} \lg k = A + B$, $A \preceq B$, and $B \asymp n \lg n$, it must be the case that $\sum_{k=1}^{n} \lg k \asymp n \lg n$. □

Problem 142.

$$\sum_{k=1}^{n} k \lg k \asymp n^2 \lg n$$

Solution:

Let $m = \lfloor \frac{n}{2} \rfloor$, $A = \sum_{k=1}^{m-1} k \lg k$, and $B = \sum_{k=m}^{n} k \lg k$. First we prove that $B \asymp n^2 \lg n$.

$$\sum_{k=m}^{n} m \lg m \le \sum_{k=m}^{n} k \lg k \le \sum_{k=m}^{n} n \lg n \quad \Leftrightarrow$$

$$(m \lg m) \sum_{k=m}^{n} 1 \le B \le (n \lg n) \sum_{k=m}^{n} 1 \quad \Leftrightarrow$$

$$\underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor\right)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{C} \le B \le \underbrace{(n \lg n)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{D}$$

Clearly, $C \asymp n^2 \lg n$ and $D \asymp n^2 \lg n$. It must be the case that $B \asymp n^2 \lg n$. Now we prove that $A \preceq B$. A has $\frac{n-2}{2}$ terms, in case n is even, and $\frac{n-3}{2}$ terms, in case n is odd. In any event, A has less terms than B. Furthermore, every term of A is smaller than any term of B. It follows $A \preceq B$. Since $\sum_{k=1}^{n} k \lg k = A + B$, it must be the case that $\sum_{k=1}^{n} k \lg k \asymp n \lg n$. □

The following derivation of Stirling's approximation is based on [Sch].

Problem 143. *Prove that for some constant* c,

$$c\sqrt{n}\left(\frac{n}{e}\right)^n \le n! \le c\sqrt{n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

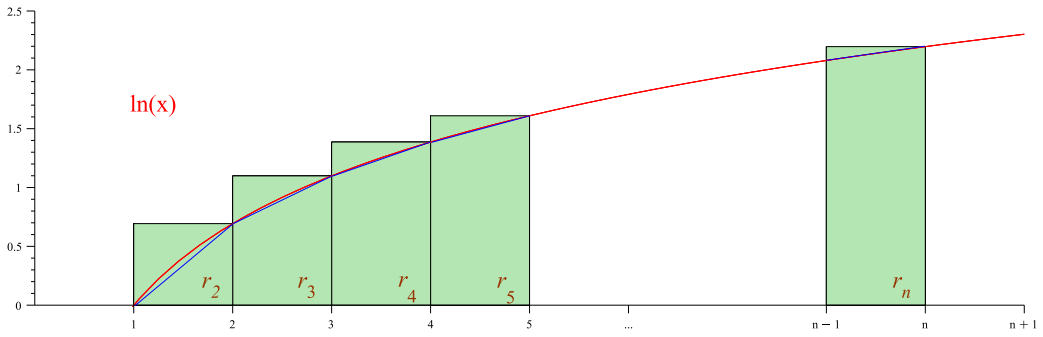
Solution:
Consider the natural log of the factorial:

$$\ln n! = \sum_{i=1}^{n} \ln i$$

Let each summand be multiplied by 1:

$$\sum_{i=1}^{n} \ln i = \sum_{i=1}^{n} ((\ln i) \times 1)$$

That sum is the area of the n rectangles r_1, r_2, r_3, ..., r_n shown on the following figure†, r_1 being a degenerate rectangle with area 0:



†The figure was made by using Maple(TM).

Let a_i denote the area of rectangle r_i, for $2 \leq i \leq n$. On the one hand, $a_i = (\ln i) \times 1 = \ln i$. On the other hand,

$$a_i = y_i + t_i - z_i \tag{8.1}$$

where

- y_i is the area of the region R_i within r_i that lies below the curve $\ln x$,
- t_i is the area of the triangle T_i with vertices $(i-1, \ln(i-1))$, $(i-1, \ln i)$, $(i, \ln i)$.
- and z_i is the area of the overlap of R_i and T_i.

Those overlaps grow smaller and smaller very rapidly, so on the figure above only the first overlap is clearly seen: it is the region within r_2 bounded by the red curved segment and the blue straight segment. Clearly,

$$\begin{aligned} y_i &= \int_{i-1}^{i} \ln x \, dx \\ t_i &= \frac{\ln i - \ln(i-1)}{2} \\ z_i &= y_i - u_i - t_i' \end{aligned}$$

where u_i is the area of the rectangle with vertices $(i-1, 0)$, $(i-1, \ln(i-1))$, $(i, \ln(i-1))$, and $(i, 0)$, and t_i' is the area of the triangle with vertices $(i-1, \ln(i-1))$, $(i, \ln i)$, and $(i, \ln(i-1))$. Clearly,

$$\begin{aligned} u_i &= \ln(i-1) \\ t_i' &= t_i = \frac{\ln i - \ln(i-1)}{2} \end{aligned}$$

Since

$$\int_{i-1}^{i} \ln x \, dx = \ln(i-1) - i\ln(i-1) - 1 + i\ln i,$$

$$\begin{aligned} z_i &= \ln(i-1) - i\ln(i-1) - 1 + i\ln i - \ln(i-1) - \frac{\ln i - \ln(i-1)}{2} \\ &= \left(1 - i + -1 + \frac{1}{2}\right)\ln(i-1) + \left(i - \frac{1}{2}\right)\ln i - 1 \\ &= \left(i - \frac{1}{2}\right)\ln i - \left(i - \frac{1}{2}\right)\ln(i-1) - 1 \\ &= \left(i - \frac{1}{2}\right)\ln\left(\frac{i}{i-1}\right) - 1 \end{aligned} \tag{8.2}$$

Now we show that the series $\sum_{i=2}^{\infty} z_i$ converges. Consider the series expansion

$$\ln(1+x) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} x^k$$

Then

$$\ln(1-x) = \ln(1+(-x)) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k}(-x)^k = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k}(-1)^k x^k$$
$$= \sum_{k=1}^{\infty} \frac{(-1)^{2k+1}}{k} x^k = -\sum_{k=1}^{\infty} \frac{x^k}{k}$$

It follows that

$$\ln(1+x) - \ln(1-x) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} x^k + \sum_{k=1}^{\infty} \frac{x^k}{k} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}+1}{k} x^k$$
$$= 2x + \frac{2x^3}{3} + \frac{2x^5}{5} + \frac{2x^7}{7} + \ldots$$

We derived

$$\frac{1}{2}\ln\left(\frac{1+x}{1-x}\right) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots$$

Substitute x by $\frac{1}{2i-1}$ to obtain

$$\frac{1}{2}\ln\left(\frac{1+\frac{1}{2i-1}}{1-\frac{1}{2i-1}}\right) = \frac{1}{2i-1} + \frac{1}{3(2i-1)^3} + \frac{1}{5(2i-1)^5} + \frac{1}{7(2i-1)^7} + \ldots \quad \Leftrightarrow$$
$$\frac{2i-1}{2}\ln\left(\frac{\frac{2i+2}{2i-1}}{\frac{2i}{2i-1}}\right) = 1 + \frac{1}{3(2i-1)^2} + \frac{1}{5(2i-1)^4} + \frac{1}{7(2i-1)^6} + \ldots \quad \Leftrightarrow$$
$$\left(i-\frac{1}{2}\right)\ln\left(\frac{i}{i-1}\right) - 1 = \frac{1}{3(2i-1)^2} + \frac{1}{5(2i-1)^4} + \frac{1}{7(2i-1)^6} + \ldots$$

Therefore,

$$\left(i-\frac{1}{2}\right)\ln\left(\frac{i}{i-1}\right) - 1 \leq \frac{1}{3}\left(\frac{1}{(2i-1)^2} + \frac{1}{(2i-1)^4} + \frac{1}{(2i-1)^6} + \ldots\right)$$

But $\frac{1}{(2i-1)^2} + \frac{1}{(2i-1)^4} + \frac{1}{(2i-1)^6} + \ldots$ is a convergent geometric series for $i > 1$ with sum

$$\frac{\frac{1}{(2i-1)^2}}{1-\frac{1}{(2i-1)^2}} = \frac{\frac{1}{(2i-1)^2}}{\frac{(2i-1)^2-1}{(2i-1)^2}} = \frac{1}{4i^2-4i+1-1} = \frac{1}{4}\left(\frac{1}{i(i-1)}\right) = \frac{1}{4}\left(\frac{1}{i-1} - \frac{1}{i}\right)$$

It follows that

$$\left(i-\frac{1}{2}\right)\ln\left(\frac{i}{i-1}\right) - 1 \leq \frac{1}{3}\left(\frac{1}{4}\left(\frac{1}{i-1} - \frac{1}{i}\right)\right) = \frac{1}{12}\left(\frac{1}{i-1} - \frac{1}{i}\right) \tag{8.3}$$

According to (8.2), the left-hand side of equation (8.3) is z_i. For any $m > 1$ and $k > 0$,

$$z_m \leq \frac{1}{12(m-1)} - \frac{1}{12m}$$
$$z_{m+1} \leq \frac{1}{12m} - \frac{1}{12(m+1)}$$
$$\ldots$$
$$z_{m+k} \leq \frac{1}{12(m+k-1)} - \frac{1}{12(m+k)}$$

thus

$$z_m + z_{m+1} + \ldots + z_{m+k} \leq \frac{1}{12(m-1)} - \frac{1}{12(m+k)}$$

So,

$$\sum_{i=m}^{\infty} z_i \leq \frac{1}{12(m-1)} \tag{8.4}$$

and in particular

$$\sum_{i=2}^{\infty} z_i \leq \frac{1}{12} \tag{8.5}$$

Now we get back to equation (8.1). Sum for $2 \leq i \leq n$ to obtain:

$$\underbrace{\sum_{i=2}^{n} a_i}_{\ln n!} = \underbrace{\sum_{i=2}^{n} \int_{i-1}^{i} \ln x \, dx}_{\int_1^n \ln x \, dx = 1 + n \ln n - n} + \underbrace{\sum_{i=2}^{n} \frac{\ln i - \ln (i-1)}{2}}_{\frac{\ln n}{2}} + \sum_{i=2}^{n} z_i \Leftrightarrow$$

$$\ln n! = 1 + n \ln n - n + \frac{\ln n}{2} + \left(\sum_{i=2}^{\infty} z_i - \sum_{i=n+1}^{\infty} z_i \right)$$

Take the exponent to base e of both sides to obtain

$$\begin{aligned} n! &= e^1 \times e^{n \ln n} \times e^{-n} \times e^{\frac{\ln n}{2}} \times e^{\sum_{i=2}^{\infty} z_i} \times e^{\sum_{i=n+1}^{\infty} z_i} \\ &= e \left(\frac{n}{e} \right)^n \sqrt{n} \times e^{\sum_{i=2}^{\infty} z_i} \times e^{\sum_{i=n+1}^{\infty} z_i} \end{aligned}$$

By (8.5), $e^{\sum_{i=2}^{\infty} z_i}$ does not exceed $\frac{1}{12}$, *i.e.* a constant, as n grows infinitely. By (8.4), $e^{\sum_{i=n+1}^{\infty} z_i} \leq e^{\frac{1}{12n}}$. It has to be the case that for some constant c,

$$c \left(\frac{n}{e} \right)^n \sqrt{n} \leq n! \leq c \left(\frac{n}{e} \right)^n \sqrt{n} e^{\frac{1}{12n}}$$

□

Problem 144. *Find a closed formula for*

$$\sum_{k=0}^{n} 2^k k$$

Solution:
Let

$$S_n = \sum_{k=0}^{n} 2^k k$$

Then

$$S_n + (n+1)2^{n+1} = \sum_{k=0}^{n} 2^k k + (n+1)2^{n+1} = \sum_{k=0}^{n} 2^{k+1}(k+1) = 2\sum_{k=0}^{n} 2^k k + 2\sum_{k=0}^{n} 2^k$$

Since $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$,

$$S_n + (n+1)2^{n+1} = \underbrace{2\sum_{k=0}^{n} 2^k k}_{2S_n} + 2(2^{n+1} - 1) = 2S_n + 2.2^{n+1} - 2$$

Then

$$S_n = n2^{n+1} + 2^{n+1} - 2.2^{n+1} + 2 = n2^{n+1} - 2^{n+1} + 2$$

So,

$$S_n = (n-1)2^{n+1} + 2 \tag{8.6}$$

□

Problem 145. *Find a closed formula for*

$$\sum_{k=0}^{n} 2^k k^2$$

Solution:
Let

$$S_n = \sum_{k=0}^{n} 2^k k^2$$

Then

$$\begin{aligned} S_n + 2^{n+1}(n+1)^2 &= \sum_{k=0}^{n} 2^k k^2 + 2^{n+1}(n+1)^2 = \sum_{k=0}^{n} 2^{k+1}(k+1)^2 \\ &= 2\sum_{k=0}^{n} 2^k(k^2 + 2k + 1) \\ &= \underbrace{2\sum_{k=0}^{n} 2^k k^2}_{2S_n} + \underbrace{4\sum_{k=0}^{n} 2^k k}_{4(n-1)2^{n+1}+8} + \underbrace{2\sum_{k=0}^{n} 2^k}_{2.2^{n+1}-2} \end{aligned}$$

Then

$$S_n + n^2 2^{n+1} + 2n2^{n+1} + 2.2^{n+1} = 2S_n + 4n2^{n+1} - 4.2^{n+1} + 8 + 2.2^{n+1} - 2$$

So,

$$S_n = n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 \tag{8.7}$$

□

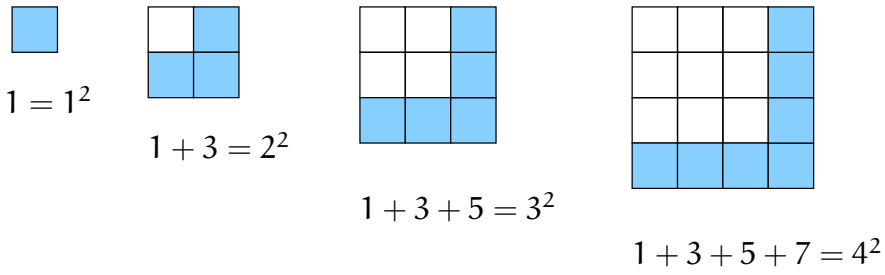


Figure 8.1: A geometric proof that the sum of the first n odd numbers is the n^{th} square n^2.

Problem 146. *Find a closed formula for the sum of the first n odd numbers*

$$S_n = 1 + 3 + 5 + \ldots + 2n - 1$$

Solution:
It is trivial to prove by induction on n that $S_n = n^2$.
Basis: $S_1 = 1^2$.
Induction hypothesis: assume $S_n = n^2$.
Induction step:

$$\begin{aligned} S_{n+1} &= 1 + 3 + 5 + \ldots + 2n - 1 + 2n + 1 \\ &= S_n + 2n + 1 \qquad \text{by definition} \\ &= n^2 + 2n + 1 \qquad \text{by the induction hypothesis} \\ &= (n+1)^2 \end{aligned}$$

Indeed,

$$S_n = n^2 \tag{8.8}$$

There is a geometric proof of the same fact, illustrated on Figure 8.1. □

Problem 147. *Find a closed formula for*

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor$$

Solution:
To gain some intuition, let us write down the sum explicitly, *i.e.* all the terms, for some small n, say $n = 17$. For clarity put boxes around the terms whose positions are perfect squares, *i.e.* around the first, fourth, ninth, and sixtieth term.

$$\sum_{i=1}^{17} \left\lfloor \sqrt{i} \right\rfloor = \underbrace{\boxed{1} + 1 + 1}_{\text{run 1}} + \underbrace{\boxed{2} + 2 + 2 + 2 + 2}_{\text{run 2}} + \underbrace{\boxed{3} + 3 + 3 + 3 + 3 + 3 + 3}_{\text{run 3}} + \underbrace{\boxed{4} + 4}_{\text{run 4}}$$

The pattern is clear: the sum is the first n, in this case $n = 17$, terms of a series whose terms are the consecutive positive integers grouped in *runs*, run j being the sum of $2j + 1$

in number j's. Naturally, each run starts at a term whose position in the series is a perfect square: run 1 starts at position 1, run 2 starts at position 4, run 3 starts at position 9, *etc.* Problem 146 explains why the runs, except possibly for the last run, have lengths that are the consecutive odd numbers—since the first j odd numbers sum precisely to a perfect square, *viz.* j^2, it follows the difference between the two consecutive perfect squares $(j+1)^2 - j^2$ is an odd number, *viz.* $2j+1$.

The run with the largest number can be incomplete, as is the case when $n = 17$—run number 4 has only two terms. Let us call the number of complete runs, *i.e.* the ones that have all the terms, k_n. For instance, $k_{17} = 3$. We claim that

$$k_n = \lfloor \sqrt{n+1} \rfloor - 1$$

To see why, imagine that n decreases one by one and think of the moment when k_n decreases. That is not when n becomes a perfect square minus one but when n becomes a perfect square minus two. For instance, $k_{15} = 3$ but $k_{14} = 2$. Hence we have $\sqrt{n+1}$, not $\sqrt{n}$.

Having all that in mind we break the desired sum down into two sums:

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run. $S_2 = 0$ if and only if n is a perfect square minus one. More precisely, if we denote the number of terms in S_2 by l_n,

$$l_n = n - \lfloor \sqrt{n+1} \rfloor^2 + 1$$

For instance, $l_{17} = 2$ as seen above and indeed $17 - \lfloor \sqrt{17+1} \rfloor^2 + 1 = 17 - 4^2 + 1 = 2$; $l_{15} = 0$ as seen above and indeed $15 - \lfloor \sqrt{15+1} \rfloor^2 + 1 = 15 - 4^2 + 1 = 0$.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.3 + 2.5 + 3.7 + 4.9 + 5.11 + \ldots + k(n)(2k(n)+1) \\
&= \sum_{i=1}^{k(n)} i(2i+1) \\
&= 2\sum_{i=1}^{k(n)} i^2 + \sum_{i=1}^{k(n)} i \\
&= 2\frac{k(n).(k(n)+1).(2k(n)+1)}{6} + \frac{k(n).(k(n)+1)}{2} \quad \text{by (8.26) and (8.27)} \\
&= k(n).(k(n)+1)\left(\frac{4k(n)+2}{6} + \frac{3}{6}\right) \\
&= \frac{1}{6}k(n).(k(n)+1).(4k(n)+5) \\
&= \frac{1}{6}(\lfloor \sqrt{n+1} \rfloor - 1)(\lfloor \sqrt{n+1} \rfloor - 1 + 1)(4\lfloor \sqrt{n+1} \rfloor - 4 + 5) \\
&= \frac{1}{6}(\lfloor \sqrt{n+1} \rfloor - 1)\lfloor \sqrt{n+1} \rfloor (4\lfloor \sqrt{n+1} \rfloor + 1)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{3}{2}}\right)$. S_2 is easier to compute, it has $l(n)$ terms, each term being $k(n)+1$.

$$\begin{aligned} S_2 &= l_n(k_n+1) \\ &= (n - \lfloor\sqrt{n+1}\rfloor^2 + 1)(\lfloor\sqrt{n+1}\rfloor - 1 + 1) \\ &= (n - \lfloor\sqrt{n+1}\rfloor^2 + 1)\lfloor\sqrt{n+1}\rfloor \end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{3}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{3}{2}}\right) + O\left(n^{\frac{3}{2}}\right) = \Theta\left(n^{\frac{3}{2}}\right)$.

Let us denote $\lfloor\sqrt{n+1}\rfloor$ by $\widetilde{n}$. It follows that

$$\begin{aligned} S_1 &= \frac{\widetilde{n}(\widetilde{n}-1)(4\widetilde{n}+1)}{6} \\ S_2 &= (n - \widetilde{n}^2 + 1)\widetilde{n} \\ \sum_{i=1}^{n} \left\lfloor\sqrt{i}\right\rfloor &= \widetilde{n}\left(\frac{(\widetilde{n}-1)(4\widetilde{n}+1)}{6} + (n - \widetilde{n}^2 + 1)\right) \end{aligned} \tag{8.9}$$

and

$$\sum_{i=1}^{n} \left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{3}{2}}\right) \tag{8.10}$$

□

Problem 148. *Find a closed formula for*

$$\sum_{i=1}^{n} \left\lceil\sqrt{i}\right\rceil$$

Solution:
Let us start with a small example as in Problem 147, say for $n = 17$. For clarity put boxes around the terms whose positions are perfect squares, *i.e.* around the first, fourth, ninth, and sixtieth term.

$$\sum_{i=1}^{17} \left\lceil\sqrt{i}\right\rceil = \underbrace{\boxed{1}}_{\text{run 1}} + \underbrace{2+2+\boxed{2}}_{\text{run 2}} + \underbrace{3+3+3+3+\boxed{3}}_{\text{run 3}} + \underbrace{4+4+4+4+4+4+\boxed{4}}_{\text{run 4}} + \underbrace{5}_{\text{run5}}$$

The pattern is quite similar to the one in Problem 147. We sum the first n terms of a series whose terms are the consecutive positive integers grouped in runs, run j being the sum of $2j-1$ in number j's.

The run with the largest number can be incomplete. For instance, if $n = 17$ then run number 5 has only one term. Let us call the number of complete runs, *i.e.* the ones that have all the terms, s_n. For instance, $s_{17} = 4$. It is obvious that

$$s_n = \lfloor\sqrt{n}\rfloor$$

We break the desired sum down into two sums:

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run. $S_2 = 0$ if and only if n is a perfect square. We denote the number of terms in S_2 by t_n.

$$t_n = n - \lfloor \sqrt{n} \rfloor^2$$

For instance, $t_{17} = 1$ as seen above and indeed $17 - \lfloor \sqrt{17} \rfloor^2 = 17 - 4^2 = 1$; $t_{16} = 0$ as seen above and indeed $16 - \lfloor \sqrt{16} \rfloor^2 = 16 - 4^2 = 0$.

Let us compute S_1.

$$\begin{aligned}
S_1 &= 1.1 + 2.3 + 3.5 + 4.7 + 5.9 + \ldots + s_n(2s_n - 1) \\
&= \sum_{i=1}^{s_n} i(2i-1) \\
&= 2\sum_{i=1}^{s_n} i^2 - \sum_{i=1}^{s_n} i \\
&= 2\frac{s_n.(s_n+1).(2s_n+1)}{6} - \frac{s_n.(s_n+1)}{2} \quad \text{by (8.26) and (8.27)} \\
&= s_n.(s_n+1)\left(\frac{4s_n+2}{6} - \frac{3}{6}\right) \\
&= \frac{1}{6}s_n.(s_n+1).(4s_n-1) \\
&= \frac{1}{6}(\lfloor \sqrt{n} \rfloor)(\lfloor \sqrt{n} \rfloor + 1)(4\lfloor \sqrt{n} \rfloor - 1)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{3}{2}}\right)$. Now we compute S_2. It has t_n terms, each term being $s_n + 1$.

$$\begin{aligned}
S_2 &= t_n.(s_n+1) \\
&= (n - \lfloor \sqrt{n} \rfloor^2)(\lfloor \sqrt{n} \rfloor + 1)
\end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{3}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{3}{2}}\right) + O\left(n^{\frac{3}{2}}\right) = \Theta\left(n^{\frac{3}{2}}\right)$.

It follows that

$$\sum_{i=1}^{n} \left\lceil \sqrt{i} \right\rceil = (\lfloor \sqrt{n} \rfloor + 1)\left(\frac{\lfloor \sqrt{n} \rfloor(4\lfloor \sqrt{n} \rfloor - 1)}{6} + n - \lfloor \sqrt{n} \rfloor^2\right) \tag{8.11}$$

and

$$\sum_{i=1}^{n} \left\lceil \sqrt{i} \right\rceil = \Theta\left(n^{\frac{3}{2}}\right) \tag{8.12}$$

□

Problem 149. *Find a closed formula for*

$$\sum_{i=1}^{n} i \left\lfloor \sqrt{i} \right\rfloor$$

Solution:
The line of reasoning is very similar to the one in Problem 147. We sum the first n terms of a series, the series being the one mentioned in the solution of Problem 147 with each term multiplied by its position. Consider for example $n = 17$. The terms whose positions are perfect squares are boxed.

$$\sum_{i=1}^{17} i \left\lfloor \sqrt{i} \right\rfloor = \underbrace{\boxed{1} + 2 + 3}_{\text{run 1}} + \underbrace{\boxed{8} + 10 + 12 + 14 + 16}_{\text{run 2}} + \underbrace{\boxed{27} + 30 + 33 + 36 + 39 + 42 + 45}_{\text{run 3}} + \underbrace{\boxed{64} + 68}_{\text{run 4}}$$

Unlike Problem 147, now the runs consist of those consecutive terms whose differences are equal (and equal to the number of the run). Just as in Problem 147, all the runs but the last one are complete, the last run being either complete or incomplete. We denote the number of the complete runs with k_n and the number of terms in the incomplete run by l_n. It is the case that

$$k_n = \lfloor \sqrt{n+1} \rfloor - 1$$
$$l_n = n - \lfloor \sqrt{n+1} \rfloor^2 + 1$$

the reasoning being exactly the same as in Problem 147. We break the desired sum down into two sums:

$$\sum_{i=1}^{n} i \left\lfloor \sqrt{i} \right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.(1+2+3)+2.(4+5+6+7+8)+3.(9+10+11+12+13+14+15)\\
&\quad +4.(16+17+18+19+20+21+22+23+24)\\
&\quad +5.(25+26+27+28+29+30+31+32+33+34+35)\\
&\quad +\ldots\\
&\quad +k_n\big(k_n^2+(k_n^2+1)+(k_n^2+2)+\ldots+\underbrace{((k_n+1)^2-1)}_{k_n^2+2k_n}\big)\\
&=\sum_{i=1}^{k_n} i\sum_{j=i^2}^{i^2+2i} j\\
&=\sum_{i=1}^{k_n} i\left(\sum_{j=1}^{i^2+2i} j-\sum_{j=1}^{i^2-1} j\right)\\
&=\sum_{i=1}^{k_n} i\left(\frac{(i^2+2i)(i^2+2i+1)}{2}-\frac{(i^2-1)i^2}{2}\right)\\
&=\frac{1}{2}\sum_{i=1}^{k_n} i\left(i^4+2i^3+i^2+2i^3+4i^2+2i-i^4+i^2\right)\\
&=\frac{1}{2}\sum_{i=1}^{k_n} i\left(4i^3+6i^2+2i\right)\\
&=2\sum_{i=1}^{k_n} i^4+3\sum_{i=1}^{k_n} i^3+\sum_{i=1}^{k_n} i^2 \quad \text{apply (8.27), (8.28), and (8.29)}\\
&=2\frac{k_n(k_n+1)(2k_n+1)(3k_n^2+3k_n-1)}{30}+3\frac{k_n^2(k_n+1)^2}{4}+\frac{k_n(k_n+1)(2k_n+1)}{6}\\
&=\frac{k_n(k_n+1)}{2}\left(\frac{(4k_n+2)(3k_n^2+3k_n-1)}{15}+\frac{3k_n(k_n+1)}{2}+\frac{2k_n+1}{3}\right)\\
&=\frac{k_n(k_n+1)}{60}\left((8k_n+4)(3k_n^2+3k_n-1)+45k_n(k_n+1)+20k_n+10\right)\\
&=\frac{k_n(k_n+1)}{60}\left(24k_n^3+24k_n^2-8k_n+12k_n^2+12k_n-4+45k_n^2+45k_n+20k_n+10\right)\\
&=\frac{k_n(k_n+1)}{60}\left(24k_n^3+81k_n^2+69k_n+6\right)\\
&=\frac{k_n(k_n+1)(8k_n^3+27k_n^2+23k_n+2)}{20} \qquad (8.13)
\end{aligned}$$

Substitute k_n with $\lfloor\sqrt{n+1}\rfloor - 1$ in (8.13) to obtain

$$\begin{aligned}
S_1 &= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8(\lfloor\sqrt{n+1}\rfloor - 1)^3 + \\
&\qquad 27(\lfloor\sqrt{n+1}\rfloor - 1)^2 + 23(\lfloor\sqrt{n+1}\rfloor - 1) + 2\big) \\
&= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8\lfloor\sqrt{n+1}\rfloor^3 - 24\lfloor\sqrt{n+1}\rfloor^2 + 24\lfloor\sqrt{n+1}\rfloor - 8 \\
&\qquad 27\lfloor\sqrt{n+1}\rfloor^2 - 54\lfloor\sqrt{n+1}\rfloor + 27 + 23\lfloor\sqrt{n+1}\rfloor - 23 + 2\big) \\
&= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8\lfloor\sqrt{n+1}\rfloor^3 + 3\lfloor\sqrt{n+1}\rfloor^2 - 7\lfloor\sqrt{n+1}\rfloor - 2\big)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{5}{2}}\right)$. Now we compute S_2. It has l_n terms, the first term is $(k_n+1)^3$, and the difference between every two consecutive terms is (k_n+1).

$$\begin{aligned}
S_2 &= \sum_{i=1}^{l_n}(k_n+1)^3 + (i-1)(k_n+1) \\
&= (k_n+1)^3\sum_{i=1}^{l_n} 1 + (k_n+1)\sum_{i=1}^{l_n}(i-1) \\
&= (k_n+1)^3 l_n + \frac{(k_n+1)(l_n-1)l_n}{2} \\
&= \lfloor\sqrt{n+1}\rfloor^3(n - \lfloor\sqrt{n+1}\rfloor^2 + 1) + \frac{\lfloor\sqrt{n+1}\rfloor(n - \lfloor\sqrt{n+1}\rfloor^2)(n - \lfloor\sqrt{n+1}\rfloor^2 + 1)}{2}
\end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{5}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{5}{2}}\right) + O\left(n^{\frac{5}{2}}\right) = \Theta\left(n^{\frac{5}{2}}\right)$.

Let us denote $\lfloor\sqrt{n+1}\rfloor$ by $\widetilde{n}$. It follows that

$$\begin{aligned}
S_1 &= \frac{\widetilde{n}(\widetilde{n}-1)(8\widetilde{n}^3 + 3\widetilde{n}^2 - 7\widetilde{n} - 2)}{20} \\
S_2 &= \widetilde{n}^3(n - \widetilde{n}^2 + 1) + \frac{\widetilde{n}(n-\widetilde{n}^2)(n-\widetilde{n}^2+1)}{2}
\end{aligned}$$

and

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \frac{\widetilde{n}(\widetilde{n}-1)(8\widetilde{n}^3 + 3\widetilde{n}^2 - 7\widetilde{n} - 2)}{20} + \widetilde{n}^3(n - \widetilde{n}^2 + 1) + \frac{\widetilde{n}(n-\widetilde{n}^2)(n-\widetilde{n}^2+1)}{2} \tag{8.14}$$

and

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{5}{2}}\right) \tag{8.15}$$

□

Problem 150. *Find a closed formula for*

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil$$

Solution:
The solution of this problem is quite similar to the solution of Problem 148. We sum the first n terms of a series, the series being the one mentioned in the solution of Problem 148 with each term multiplied by its position. Consider for example $n = 17$. The terms whose positions are perfect squares are boxed.

$$\sum_{i=1}^{17} i\left\lfloor\sqrt{i}\right\rfloor = \underbrace{\boxed{1}}_{\text{run 1}} + \underbrace{4+6+\boxed{8}}_{\text{run 2}} + \underbrace{15+18+21+24+\boxed{27}}_{\text{run 3}} + \underbrace{40+44+48+52+56+60+\boxed{64}}_{\text{run 4}} + \underbrace{85}_{\text{run5}}$$

Unlike Problem 148, now the runs consist of those consecutive terms whose differences are equal (and equal to the number of the run). Just as in Problem 148, all the runs but the last one are complete, the last run being either complete or incomplete. We denote the number of the complete runs with $s(n)$ and

$$s(n) = \lfloor\sqrt{n}\rfloor$$

the reasoning being exactly the same as in Problem 148. The number of terms in the incomplete run is

$$t(n) = n - \lfloor\sqrt{n}\rfloor^2$$

We break the desired sum down into two sums:

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.1 + 2.(2+3+4) + 3.(5+6+7+8+9) \\
&\quad + 4.(10+11+12+13+14+15+16) \\
&\quad + 5.(17+18+19+20+21+22+23+24+25) \\
&\quad + \ldots \\
&\quad + s_n\big(((s_n-1)^2+1) + ((s_n-1)^2+2) + \ldots + \underbrace{s_n^2}_{(s_n-1)^2+2s_n-1}\big) \\
&= \sum_{i=1}^{s_n} i \sum_{j=1}^{2i-1} (i-1)^2 + j \qquad (8.16) \\
&= \sum_{i=1}^{s_n} i \left(\sum_{j=1}^{2i-1} (i-1)^2 + \sum_{j=1}^{2i-1} j \right) \\
&= \sum_{i=1}^{s_n} i \left((i-1)^2(2i-1) + \frac{(2i-1)2i}{2} \right) \\
&= \sum_{i=1}^{s_n} i \Big((i^2-2i+1)(2i-1) + 2i^2 - i \Big) \\
&= \sum_{i=1}^{s_n} i(2i^3 - i^2 - 4i^2 + 2i + 2i - 1 + 2i^2 - i) \\
&= \sum_{i=1}^{s_n} i(2i^3 - 3i^2 + 3i - 1) \\
&= 2\sum_{i=1}^{s_n} i^4 - 3\sum_{i=1}^{s_n} i^3 + 3\sum_{i=1}^{s_n} i^2 - \sum_{i=1}^{s_n} i \quad \text{apply (8.26), (8.27), (8.28), and (8.29)} \\
&= 2\frac{s_n(s_n+1)(2s_n+1)(3s_n^2+3s_n-1)}{30} - 3\frac{s_n^2(s_n+1)^2}{4} + \\
&\qquad 3\frac{s_n(s_n+1)(2s_n+1)}{6} - \frac{s_n(s_n+1)}{2} \\
&= \frac{s_n(s_n+1)}{2}\left(\frac{2(2s_n+1)(3s_n^2+3s_n-1)}{15} - \frac{3s_n(s_n+1)}{2} + \frac{6s_n+3}{3} - 1 \right) \qquad (8.17)
\end{aligned}$$

Simplify (8.17) to obtain

$$\frac{s_n(s_n+1)}{2}\left(\frac{12s_n^3+12s_n^2-4s_n+6s_n^2+6s_n-2}{15}-\frac{3s_n^2+3s_n}{2}+\frac{6s_n+3}{3}-1\right)=$$
$$\frac{s_n(s_n+1)}{2}\left(\frac{24s_n^3+36s_n^2+4s_n-4}{30}-\frac{45s_n^2+45s_n}{30}+\frac{60s_n+30}{30}-\frac{30}{30}\right)=$$
$$\frac{s_n(s_n+1)}{60}(24s_n^3+36s_n^2+4s_n-4-45s_n^2-45s_n+60s_n+30-30)=$$
$$\frac{s_n(s_n+1)(24s_n^3-9s_n^2+19s_n-4)}{60}=$$
$$\frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+1)(24\lfloor\sqrt{n}\rfloor^3-9\lfloor\sqrt{n}\rfloor^2+19\lfloor\sqrt{n}\rfloor-4)}{60}$$

Clearly, $S_1=\Theta\left(n^{\frac{5}{2}}\right)$. Now we compute S_2. It has t_n terms, the first term is $(s_n^2+1)(s_n+1)$, and the difference between every two consecutive terms is (s_n+1).

$$\begin{aligned}
S_2 &= \sum_{i=1}^{t_n}(s_n^2+1)(s_n+1)+(i-1)(s_n+1)= \\
&= (s_n^2+1)(s_n+1)\sum_{i=1}^{t_n}1+(s_n+1)\sum_{i=1}^{t_n}(i-1) \\
&= t_n(s_n^2+1)(s_n+1)+\frac{(s_n+1)(t_n-1)t_n}{2}= \\
&= \frac{t_n(s_n+1)}{2}\left(2s_n^2+2+t_n-1\right)= \\
&= \frac{t_n(s_n+1)(2s_n^2+t_n+1)}{2} \\
&= \frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(2\lfloor\sqrt{n}\rfloor^2+n-\lfloor\sqrt{n}\rfloor^2+1)}{2} \\
&= \frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(n+\lfloor\sqrt{n}\rfloor^2+1)}{2}
\end{aligned}$$

Clearly, $S_2=O\left(n^{\frac{5}{2}}\right)$, therefore $S_1+S_2=\Theta\left(n^{\frac{5}{2}}\right)+O\left(n^{\frac{5}{2}}\right)=\Theta\left(n^{\frac{5}{2}}\right)$. It follows that

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor=\frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+1)(24\lfloor\sqrt{n}\rfloor^3-9\lfloor\sqrt{n}\rfloor^2+19\lfloor\sqrt{n}\rfloor-4)}{60}+\frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(n+\lfloor\sqrt{n}\rfloor^2+1)}{2} \tag{8.18}$$

and

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil=\Theta\left(n^{\frac{5}{2}}\right) \tag{8.19}$$

□

Fact: The Fibonacci numbers are the natural numbers defined by the recurrence relation

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \text{ for all } n > 1 \end{aligned}$$

The first several elements of the sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The asymptotic growth rate of F_n is determined by the following equality [GKP94, pp. 300]

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor = \frac{\phi^n}{\sqrt{5}}, \text{ rounded to the nearest integer}$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the so called "golden ratio", the positive root of $\phi^2 = \phi + 1$. Clearly, for any positive constant c,

$$c^{F_n} = \Theta\left(c^{\frac{\phi^n}{\sqrt{5}}}\right) = \Theta\left(k^{\phi^n}\right), \text{ where } k = c^{\frac{1}{\sqrt{5}}} \tag{8.20}$$

□

Fact: The harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots = \sum_{i=1}^{\infty} \frac{1}{i}$$

is divergent. Its n^{th} partial sum is denoted by H_n.

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} \tag{8.21}$$

It is known that

$$H_n = \Theta(\lg n) \tag{8.22}$$

Furthermore, $\ln n < H_n < \ln n + 1$ for $n > 1$. For details, see [GKP94, pp. 272–278]. □

Fact: The sum of the first k^{th} powers for some integer constant $k \geq 1$ is

$$1^k + 2^k + \ldots + n^k = \sum_{i=0}^{n} i^k \tag{8.23}$$

It is well known that

$$\sum_{i=0}^{n} i^k = \frac{1}{k+1} \sum_{j=0}^{k} \binom{k+1}{j} B_j (n+1)^{k+1-j} \tag{8.24}$$

where B_j is the j^{th} *Bernoulli number*. The Bernolli numbers are defined with the recurrence

$$B_0 = 1$$
$$B_m = -\frac{1}{m}\sum_{j=0}^{m-1}\binom{m+1}{j}B_j, \text{ for } m \in \mathbb{N}^+$$

For details on the summation formula (8.24) and plenty of information on the Bernoulli numbers, see [GKP94, pp. 283–290]. Just keep in mind that Knuth *et al.* denote the sum by $S_k(n)$ and define it as

$$S_k(n) = 0^k + 1^k + 2^k + \ldots + (n-1)^k$$

For our purposes in this manual it is sufficient to know that

$$1^k + 2^k + \ldots + n^k = \Theta(n^{k+1}) \tag{8.25}$$

which fact follows easily from (8.24). In fact, (8.24) is a polynomial of degree $k+1$ of n because the $\binom{k+1}{j}$ factor and the Bernoulli numbers are just constants and clearly the highest degree of n is $k+1$. Strictly speaking, we have not proved here formally that (8.24) is a degree $k+1$ polynomial of n because we have not shown that the coefficient before n^{k+1} is not zero. But that is indeed the case—see for instance [GKP94, (6.98), pp. 288].

⁘ NB ⁘ *Be careful to avoid the error of thinking that*

$$1^k + 2^k + \ldots + n^k$$

is a degree k *polynomial of* n *and thus erroneosly concluding that its order of growth is* $\Theta(n^k)$. *It is* not *a polynomial of* n *because a polynomial has an* a priori *fixed number of terms, while the above sum has* n *terms where* n *is the variable.*

Using (8.24), we can easily derive

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} \tag{8.26}$$
$$1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6} \tag{8.27}$$
$$1^3 + 2^3 + \ldots + n^3 = \frac{n^2(n+1)^2}{4} \tag{8.28}$$
$$1^4 + 2^4 + \ldots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \tag{8.29}$$

□

Lemma 33.

$$\sum_{k=1}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

Proof:

$$\sum_{k=1}^{n} k(k+1) = 1\times(1+1) + 2\times(2+1) + \ldots + n(n+1) =$$

$$1\times 1 + 1 + 2\times 2 + 2 + \ldots n\times n + n =$$

$$1\times 1 + 2\times 2 + \ldots + n\times n + 1 + 2 + \ldots + n = \quad \text{by (8.27) and (8.28)}$$

$$\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} =$$

$$\frac{n(n+1)}{2}\left(\frac{2n+1}{3} + 1\right) =$$

$$\frac{n(n+1)(n+2)}{3}$$

□

Problem 151. *Let* T *be a binary heap of height* h *vertices. Find the minimum and maixmum number of vertices in* T*.*

Solution:
The vertices of any binary tree are partitioned into *levels*, the vertices from level number i being the ones that are at distance i from the root. By definition, every level i in T, except possibly for level h, is complete in the sense it has all the 2^i vertices possible. The last level (number h) can have anywhere between 1 and 2^h vertices inclusive. If n denotes the number of vertices in the heap, it is the case that

$$\underbrace{2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}}_{\text{the number of vertices in the complete levels}} + 1 \le n \le \underbrace{2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}}_{\text{the number of vertices in the complete levels}} + 2^h$$

Since $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = \frac{2^h-1}{2-1} = 2^h - 1$, it follows that

$$2^h - 1 + 1 \le n \le 2^h - 1 + 2^h$$

$$2^h \le n \le 2^{h+1} - 1 \tag{8.30}$$

□

Problem 152. *Let* T *be a binary heap with* n *vertices. Find the height* h *of* T*.*

Solution:

$$2^h \le n \le 2^{h+1} - 1 \quad \text{see Problem 151, (8.30)}$$

$$2^h \le n < 2^{h+1}$$

$$h \le \lg n < h + 1 \quad \text{take lg of both sides}$$

Clearly,

$$h = \lfloor \lg n \rfloor \tag{8.31}$$

□

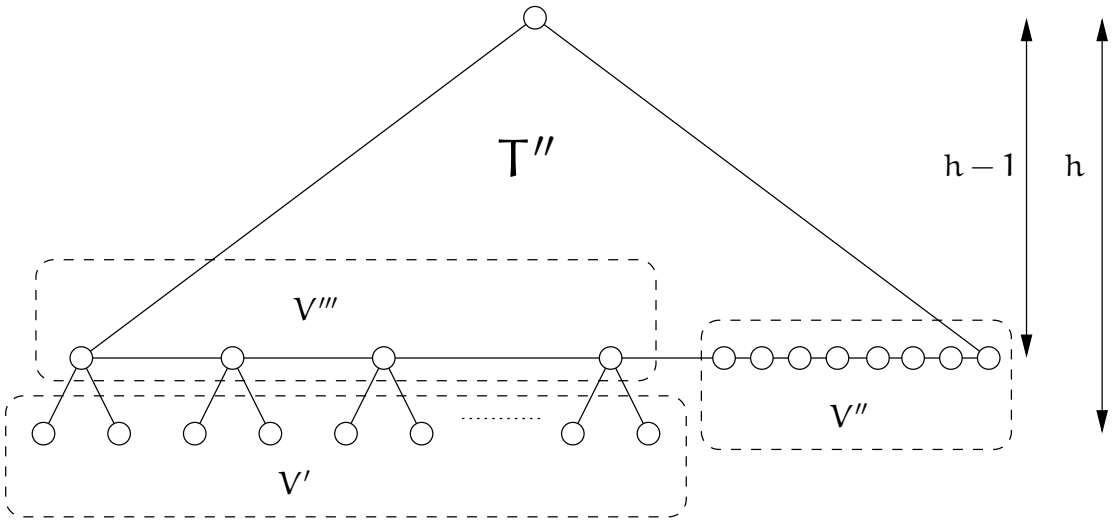


Figure 8.2: The heap in Problem 153.

Problem 153. *Let* T *be a binary heap with* n *vertices. Prove that the number of leaves of* T *is* $\left\lceil \frac{n}{2} \right\rceil$ *and the number of internal vertices in* T *is* $\left\lfloor \frac{n}{2} \right\rfloor$.

Solution:
Let h be the height of T. We know (8.31) that $h = \lfloor \lg n \rfloor$. Let V' be the vertices of T at level h. Let T'' be obtained from T by deleting V' (see Figure 8.2). Clearly, T'' is a complete binary tree of height $h - 1 = \lfloor \lg n \rfloor - 1$. The number of its vertices is

$$2^{\lfloor \lg n \rfloor - 1 + 1} - 1 = 2^{\lfloor \lg n \rfloor} - 1 \tag{8.32}$$

It follows

$$|V'| = n - (2^{\lfloor \lg n \rfloor} - 1) = n + 1 - 2^{\lfloor \lg n \rfloor} \tag{8.33}$$

The vertices at level $h-1$ are $2^{h-1} = 2^{\lfloor \lg n \rfloor - 1}$. Those vertices are partitioned into V'', the vertices that have no children, and V''', the vertices that have a child or two children (see Figure 8.2). So,

$$|V''| + |V'''| = 2^{\lfloor \lg n \rfloor - 1} \tag{8.34}$$

Note that $|V'''| = \left\lceil \frac{|V'|}{2} \right\rceil$. Having in mind (8.33), it follows that

$$|V'''| = \left\lceil \frac{n + 1 - 2^{\lfloor \lg n \rfloor}}{2} \right\rceil = \left\lceil \frac{n+1}{2} - \frac{2^{\lfloor \lg n \rfloor}}{2} \right\rceil = \left\lceil \frac{n+1}{2} - 2^{\lfloor \lg n \rfloor - 1} \right\rceil =$$
$$\left\lceil \frac{n+1}{2} \right\rceil - 2^{\lfloor \lg n \rfloor - 1} \qquad \text{since } 2^{\lfloor \lg n \rfloor - 1} \text{ is integer} \tag{8.35}$$

Use (8.34) and (8.35) to conclude that

$$\begin{aligned}|V''| &= 2^{\lfloor \lg n \rfloor - 1} - \left(\left\lceil \frac{n+1}{2} \right\rceil - 2^{\lfloor \lg n \rfloor - 1} \right) \\ &= 2^{\lfloor \lg n \rfloor - 1} - \left\lceil \frac{n+1}{2} \right\rceil + 2^{\lfloor \lg n \rfloor - 1} \\ &= 2.2^{\lfloor \lg n \rfloor - 1} - \left\lceil \frac{n+1}{2} \right\rceil \\ &= 2^{\lfloor \lg n \rfloor} - \left\lceil \frac{n+1}{2} \right\rceil \end{aligned} \tag{8.36}$$

It is obvious the leaves of T are $V' \cup V''$. Use (8.33) and (8.36) to conclude that

$$\begin{aligned}|V'| + |V''| &= n + 1 - 2^{\lfloor \lg n \rfloor} + 2^{\lfloor \lg n \rfloor} - \left\lceil \frac{n+1}{2} \right\rceil \\ &= n + 1 - \left\lceil \frac{n+1}{2} \right\rceil = n + 1 + \left\lfloor -\frac{n+1}{2} \right\rfloor \\ &= \left\lfloor n + 1 - \frac{n+1}{2} \right\rfloor \qquad \text{since } n+1 \text{ is integer} \\ &= \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil \end{aligned} \tag{8.37}$$

Then the internal vertices of T must be $\lfloor \frac{n}{2} \rfloor$ since $m = \lfloor \frac{m}{2} \rfloor + \lceil \frac{m}{2} \rceil$ for any natural number m. □

Lemma 34 ([GKP94], pp. 71)**.** *Let* $f(x)$ *be any continuous, monotonically increasing function with the property that*

$f(x)$ *is integer* $\Rightarrow$ x *is integer*

Then,

$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \quad \textit{and} \quad \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

□

Corollary 5.

$$\forall x \in \mathbb{R}^+ \; \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \quad \textit{and} \quad \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil \right)$$

Proof:
Apply Lemma 34 with $f(x) = \frac{x}{b}$. □

The equalities in Corollary 6 are presented in [CLR00] but without any proof.

Corollary 6.

$$\forall x \in \mathbb{R}^+ \; \forall a \in \mathbb{N}^+ \; \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \quad \textit{and} \quad \left\lceil \frac{\lceil \frac{x}{a} \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \right)$$

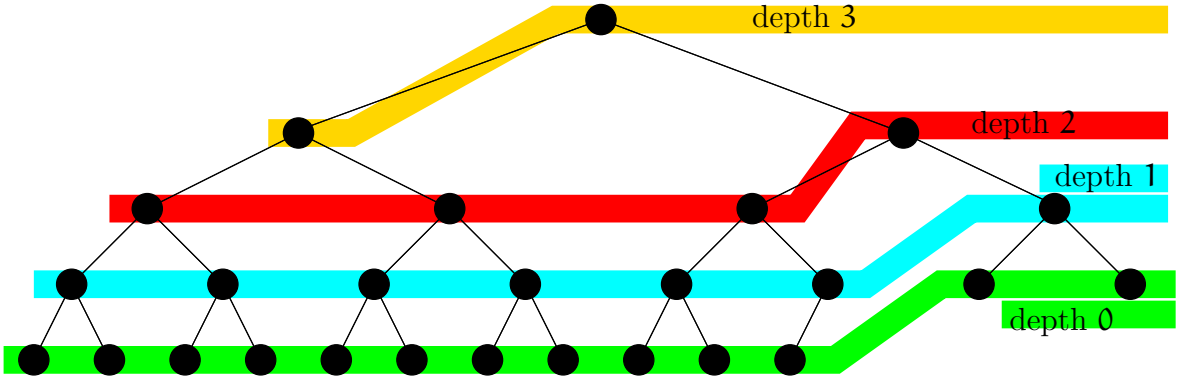


Figure 8.3: Depths of vertices in a complete binary tree T.

Proof:
Apply Corollary 5 with $\frac{x}{a}$ instead of x. □

Lemma 35. *In any binary heap T of n vertices, there are precisely $\left\lfloor \frac{n}{2^d} \right\rfloor$ vertices of depth $\geq d$.*

Proof:
We remind the reader *depth* is defined as follows: any vertex u has depth d if the longest path p—that does not contain the parent of u if one exists—between u and any leaf is of length d (see also Definition 6 on page 132). For instance, see Figure 8.3. The proof is by induction on d.

Basis. $d = 0$. The vertices of depth ≥ 0 are precisely the vertices of T. But there are $n = \left\lfloor \frac{n}{2^0} \right\rfloor$ vertices altogether in T. ✓

Induction Hypothesis. Assume the claim holds for some depth d that is not the maximum.

Induction Step. Delete all vertices of depth $< d$ from T. Let us call the obtained tree T' and let n' be the number of its vertices. The leaves of T', *i.e.* the vertices of depth 0 in T', are precisely the vertices of depth d in T. All vertices of T' are precisely the vertices of depth $\geq d$ in T. By the inductive hypothesis, $n' = \left\lfloor \frac{n}{2^d} \right\rfloor$. For example, consider T from Figure 8.3; let $d = 1$ and after deleteting all vertices of depth < 1, we obtain the tree T' from Figure 8.4.

We know (see Problem 153 on page 251) there are $\left\lfloor \frac{n'}{2} \right\rfloor$ internal vertices in T'. But the internal vertices in T' are precisely the vertices of depth $\geq d+1$ in T. It follows there are $\left\lfloor \frac{\left\lfloor \frac{n}{2^d} \right\rfloor}{2} \right\rfloor$ vertices of depth $\geq d+1$ in T. By Corollary 6, $\left\lfloor \frac{\left\lfloor \frac{n}{2^d} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2 \times 2^d} \right\rfloor$, and certainly $\left\lfloor \frac{n}{2 \times 2^d} \right\rfloor = \left\lfloor \frac{n}{2^{d+1}} \right\rfloor$. □

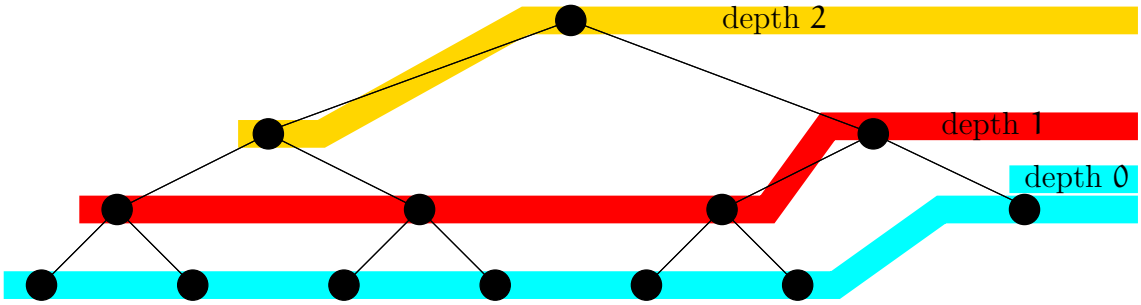


Figure 8.4: The complete tree T' obtained from T by deleting the vertices of depth < 1, *i.e.* of depth 0. That vertex deletion decrements by one the depths of the remaining vertices.

Corollary 7. *In any binary heap* T *of* n *vertices, there are precisely* $\left\lceil \frac{\lfloor \frac{n}{2^d} \rfloor}{2} \right\rceil$ *vertices of depth* d.

Proof:
By Lemma 35, there are $\left\lfloor \frac{n}{2^d} \right\rfloor$ vertices of depth $\geq d$. The vertices of depth d are precisely the leaves of the subtree of T induced by the vertices of depth $\geq d$. By Problem 153 on page 251, those leaves are $\left\lceil \frac{\lfloor \frac{n}{2^d} \rfloor}{2} \right\rceil$. □

Problem 154. *Find a closed formula for*

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil$$

Solution:

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil =$$
$$\left\lfloor \frac{0-1}{2} \right\rfloor \left\lceil \frac{0-1}{2} \right\rceil + \left\lfloor \frac{1-1}{2} \right\rfloor \left\lceil \frac{1-1}{2} \right\rceil + \left\lfloor \frac{2-1}{2} \right\rfloor \left\lceil \frac{2-1}{2} \right\rceil + \left\lfloor \frac{3-1}{2} \right\rfloor \left\lceil \frac{3-1}{2} \right\rceil +$$
$$\ldots + \left\lfloor \frac{(n-1)-1}{2} \right\rfloor \left\lceil \frac{(n-1)-1}{2} \right\rceil + \left\lfloor \frac{n-1}{2} \right\rfloor \left\lceil \frac{n-1}{2} \right\rceil =$$
$$(-1) \times 0 + 0 \times 0 + 0 \times 1 + 1 \times 1 + 1 \times 2 +$$
$$\ldots + \left\lfloor \frac{(n-1)-1}{2} \right\rfloor \left\lceil \frac{(n-1)-1}{2} \right\rceil + \left\lfloor \frac{n-1}{2} \right\rfloor \left\lceil \frac{n-1}{2} \right\rceil$$

Suppose n is odd, *i.e.* $n = 2t+1$ for some $t \in \mathbb{N}$. We have to evaluate the sum

$$1 \times 1 + 1 \times 2 + 2 \times 2 + 2 \times 3 + \ldots + (t-1) \times t + t \times t =$$
$$\underbrace{1 \times 1 + 2 \times 2 + \ldots + (t-1) \times (t-1) + t \times t}_{A} + \underbrace{1 \times 2 + 2 \times 3 + \ldots + (t-2) \times (t-1) + (t-1) \times t}_{B}$$

By (8.28) on page 249, $A = \frac{t(t+1)(2t+1)}{6}$, and by Lemma 33 on page 249, $B = \frac{(t-1)t(t+1)}{3}$. So,

$$A + B = \frac{t(t+1)(2t+1)}{6} + \frac{(t-1)t(t+1)}{3} = \frac{t(t+1)}{3}\left(\frac{2t+1}{2} + (t-1)\right) = \frac{t(t+1)}{3}\left(\frac{2t+1+2t-2}{2}\right) = \frac{t(t+1)(4t-1)}{6}$$

Now suppose n is even, *i.e.* $n = 2t$ for some $t \in \mathbb{N}$. We have to evaluate the sum

$$1 \times 1 + 1 \times 2 + 2 \times 2 + 2 \times 3 + \ldots + (t-1) \times (t-1) + (t-1) \times t = \underbrace{1 \times 1 + 2 \times 2 + \ldots + (t-1) \times (t-1)}_{A} + \underbrace{1 \times 2 + 2 \times 3 + \ldots + (t-2) \times (t-1) + (t-1) \times t}_{B}$$

By (8.28) on page 249, $A = \frac{(t-1)t(2t-1)}{6}$, and by Lemma 33 on page 249, $B = \frac{(t-1)t(t+1)}{3}$. So,

$$A + B = \frac{(t-1)t(2t-1)}{6} + \frac{(t-1)t(t+1)}{3} = \frac{t(t-1)}{3}\left(\frac{2t-1}{2} + (t+1)\right) = \frac{t(t-1)}{3}\left(\frac{2t-1+2t+2}{2}\right) = \frac{t(t-1)(4t+1)}{6}$$

Overall,

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil = \begin{cases} \frac{(n-1)(n+1)(2n-3)}{24}, & n \text{ odd} \\ \frac{(n-2)n(2n+1)}{24}, & n \text{ even} \end{cases} \tag{8.38}$$

□

Computational Problem HALTING PROBLEM
Generic Instance: $\langle \mathfrak{P}, \mathfrak{I} \rangle$ where $\mathfrak{P}$ is a computer program and $\mathfrak{I}$ its input
Question: Does $\mathfrak{P}(\mathfrak{I})$ halt? □

The following is a simplistic version of the proof of the famous undecidability result. For a more thorough treatment see, for instance, [Sip06].

Theorem 5. *The* HALTING PROBLEM *is algorithmically unsolvable.*

Proof:
Assume the opposite. Then there exists an program $\mathfrak{Q}$ with input an ordered pair $\langle \mathfrak{P}, \mathfrak{I} \rangle$ of (the encoding of) a computer program $\mathfrak{P}$ and its input $\mathfrak{I}$ (*i.e.*, $\mathfrak{I}$ is input to $\mathfrak{P}$), such that $\mathfrak{Q}(\mathfrak{P}, \mathfrak{I})$ returns TRUE if $\mathfrak{P}(\mathfrak{I})$ halts, and FALSE otherwise. Define program $\mathfrak{S}(\mathfrak{P})$ as $\mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$. That is, $\mathfrak{S}(\mathfrak{P})$ consists of the single line

return $\mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$

Define yet another program $\mathfrak{T}(\mathfrak{P})$ as follows:

```
if 𝔖(𝔓) then loop forever
else return TRUE
```

Analyse $\mathfrak{T}(\mathfrak{T})$. If it goes into infinite loop, it must be the case that $\mathfrak{S}(\mathfrak{T})$ returns TRUE. Then it must be the case that $\mathfrak{Q}(\mathfrak{T},\mathfrak{T})$ returns TRUE. Then it must be the case that $\mathfrak{T}$ halts with input $\mathfrak{T}$. This is a contradiction.

If $\mathfrak{T}$ halts, it returns TRUE. Then it must be the case that $\mathfrak{S}(\mathfrak{T})$ returns FALSE. Then it must be the case that $\mathfrak{Q}(\mathfrak{T},\mathfrak{T})$ returns FALSE. Then it must be the case that $\mathfrak{T}$ does not halt with input $\mathfrak{T}$. This is a contradiction, too. □

Definition 13. A Turing machine M *is an abstract device consisting of:*

- *a two-ways infinite tape, each cell of which contains exactly one symbol out of finitely many possible tape symbols,*
- *a read-write head that is positioned initially at cell number* 1 *and can then move in discrete steps from a cell to any of its two adjacent cells,*
- *a control unit that in any moment of the machine's work is in precisely one of finitely many* states.

The machine operates in discrete steps. At each step it considers the symbol that is currently read by the head and the state it is currently in. There are fixed rules (transition table) that specify a new symbol to be written in that cell, a new state to go into, and a new cell for the head, left or right. Formally,

$$M = \langle \Sigma, \Gamma, Q, q_0, q_Y, q_N, \delta \rangle$$

where Σ *is a finite alphabet called* the input alphabet, Γ *is another finite alphabet that is a strict superset of* Σ *and such that there is at least one symbol* $\lrcorner \in \Gamma \setminus \Sigma$ *called* blank, Q *is a finite set of states that necessarily contains those three states* q_0, q_Y, *and* q_N *where* q_0 *is the* start state *and* q_Y *and* q_N *collectively are known as the* halting states, *and finally* δ *is* the transition function

$$\delta : \big(Q \setminus \{q_Y, q_N\}\big) \times \Gamma \to Q \times \Gamma \times \Delta$$

where $\Delta = \{\leftarrow, \rightarrow\}$. Σ *is the alphabet used to encode the input and* Γ *is the overall alphabet of the machine. Initially, the input* $x = \sigma_1, \sigma_2, \ldots, \sigma_n$ *is written into cells number* $1, 2, \ldots, n$, *respectively, and all other cells of the tape are filled with* $\lrcorner$. *The two arrows* $\leftarrow$, $\rightarrow$ *encode the movement of the head in the obvious way. Initially, the machine is in the starting state and then, in accordance with the tape content and the transition table, it changes states, changes the tape content and moves the head. Whenever, if ever, the machine reaches any of the halting states, it stops working, or halts. It it halts in* q_Y *we way it* accepths *input* x *and if it halts in* q_N *we say it* rejects *input* x. □

Recall that a palindrome is a string that coincides with its reverse permutation, *e.g.* 010111010.

Problem 155. *Construct a Turing machine that recognises palindromes over* $\Sigma = \{0, 1\}$.

Solution:
A Turing machine can be defined by just defining its input alphabet Σ and its transition function δ, for instance by a table. From the table we can infer the set of states and the tape alphabet (the input alphabet is given already).

	0	1	␣
q_0	$\langle r_0, \text{␣}, \rightarrow\rangle$	$\langle r_1, \text{␣}, \rightarrow\rangle$	$\langle q_Y, \text{␣}, \leftarrow\rangle$
r_0	$\langle r'_0, 0, \rightarrow\rangle$	$\langle r'_0, 1, \rightarrow\rangle$	$\langle q_y, \text{␣}, \leftarrow\rangle$
r'_0	$\langle r'_0, 0, \rightarrow\rangle$	$\langle r'_0, 1, \rightarrow\rangle$	$\langle s_0, \text{␣}, \leftarrow\rangle$
s_0	$\langle t, \text{␣}, \leftarrow\rangle$	$\langle q_N, \text{␣}, \leftarrow\rangle$	$\langle s_0, \text{␣}, \leftarrow\rangle$
t	$\langle t, 0, \leftarrow\rangle$	$\langle t, 1, \leftarrow\rangle$	$\langle t', \text{␣}, \rightarrow\rangle$
t'	$\langle r_0, \text{␣}, \rightarrow\rangle$	$\langle r_1, \text{␣}, \rightarrow\rangle$	$\langle t', \text{␣}, \leftarrow\rangle$
r_1	$\langle r'_1, 0, \rightarrow\rangle$	$\langle r'_1, 1, \rightarrow\rangle$	$\langle q_Y, \text{␣}, \leftarrow\rangle$
r'_1	$\langle r'_1, 0, \rightarrow\rangle$	$\langle r'_1, 1, \rightarrow\rangle$	$\langle s_1, \text{␣}, \leftarrow\rangle$
s_1	$\langle q_N, \text{␣}, \leftarrow\rangle$	$\langle t, \text{␣}, \leftarrow\rangle$	$\langle s_1, \text{␣}, \leftarrow\rangle$

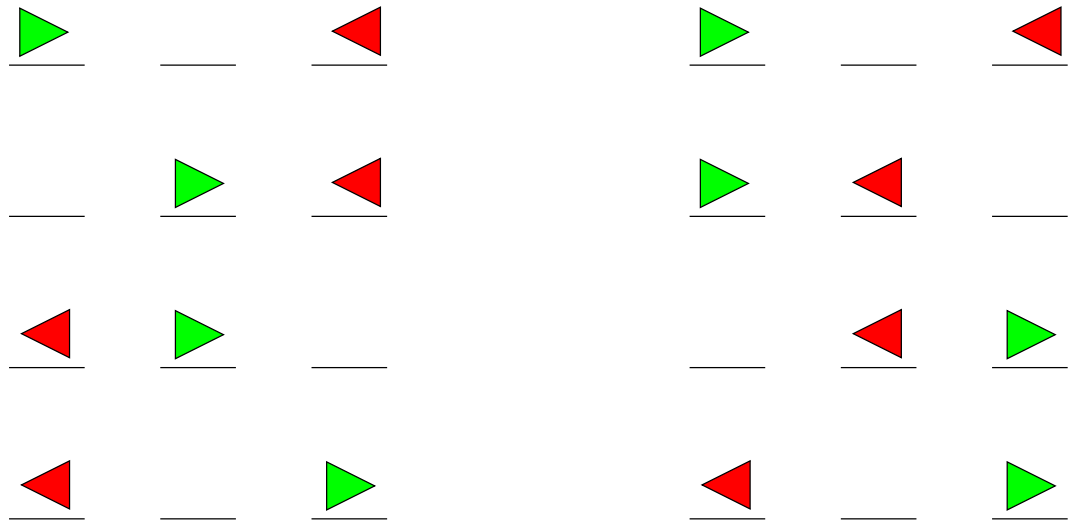
Table element $T[i, j]$ is the ordered triple $\langle x, y, z\rangle$ where x is the new state, y is the new symbol, and z is the movement of the head that correspond to state i and symbol j. For instance (row 1), if the machine is in q_0 and the symbol is 0 then it goes into state r_0, writes blank, and moves the head rightwards. The three coloured cells correspond to unreachable (impossible) combinations of state and symbol so it really does not matter what it is in them. □

The n *frogs leap frog puzzle* is defined as follows. There are $2n + 1$ lily pads arranged in a line and numbered $1, \ldots, 2n + 1$. There are $2n$ frogs, n green and n red, on the pads. Each pad can have at most one frog on it. The initial arrangement is, the green frogs are on pads $1, \ldots, n$ and the red ones, on pads $n + 2, \ldots, 2n + 1$. Thus initially the middle lily pad has no frog. The frogs can jump in discrete subsequent moments. At each moment at most one frog jumps. Furthermore, it can jump from pad k only to pad $k - 2$ or $k - 1$ or $k + 1$ or $k + 2$, provided the number is in $\{1, 2, \ldots, 2n + 1\}$ and that the target pad has no frog on it already. Furthermore, the green frogs jump only rightwards and the red frogs, only leftwards. The goal is to move all green frogs to pads $n + 2, \ldots, 2n + 1$ and all red frogs to pads $1, \ldots, n$, that is, to exhange the greens and the reds, with as few subsequent jumps as possible. Solutions for $n = 3$ are commonly found on the Internet.

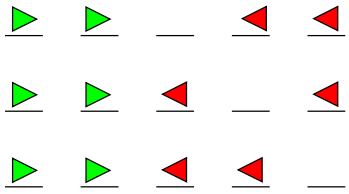
Problem 156. *What is the minimum number of jumps that solves the* n *frogs leap frog puzzle? Give a precise answer.*

Solution:
Let us investigate small instances. Let $n = 1$. Let the frogs be drawn by coloured triangles. Consider the initial arrangement:

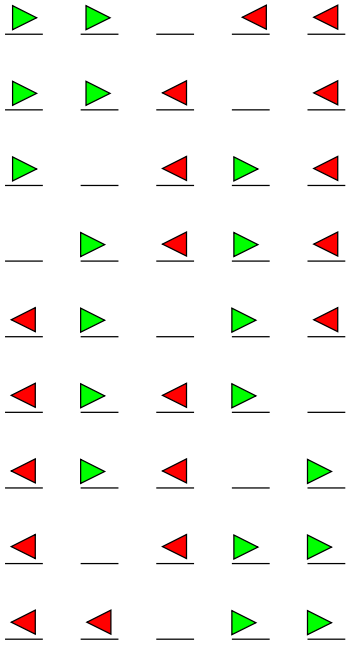
There are precisely two solutions, both using 3 jumps:



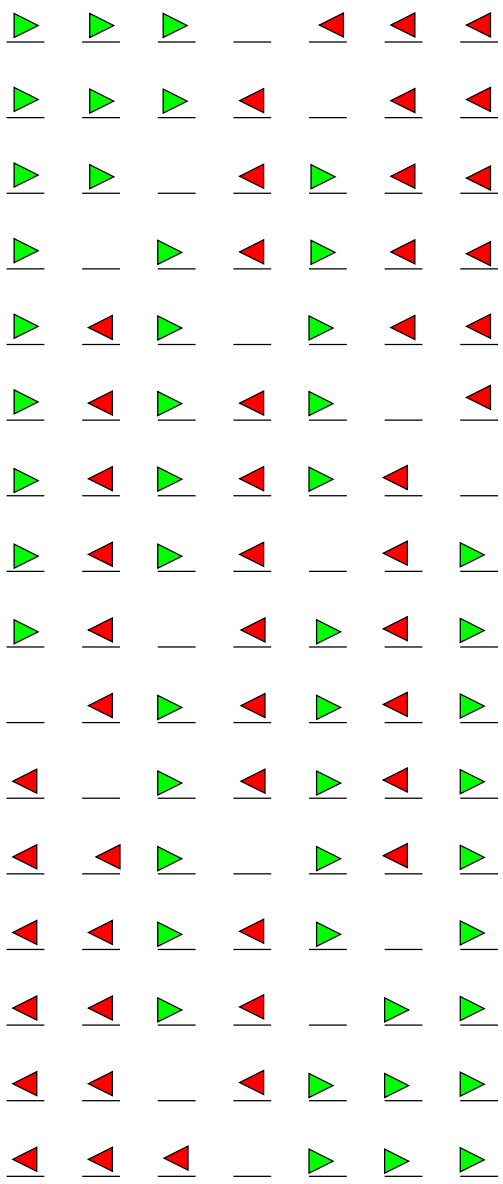
Once we choose which frog to move initially, the solution follows. There is no way to get into a dead-end position when $n = 1$. However, when $n = 2$ it is possible to get into a dead-end position:



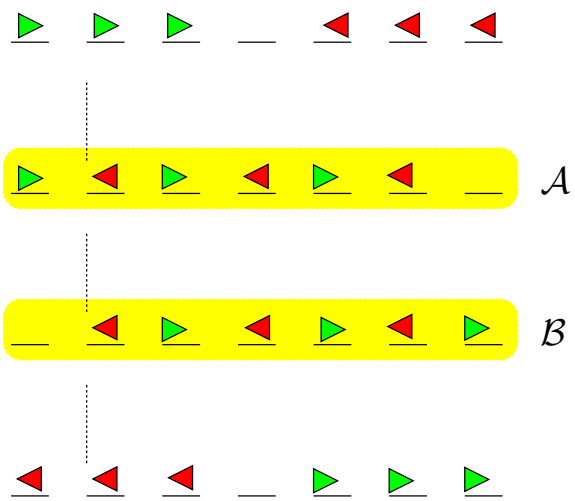
Here is a possible solution for $n = 2$ that uses 8 jumps:



And a solution for $n = 3$ that uses 15 jumps:



The last example is big enough to generalise. There are two key intermediate arrangements $\mathcal{A}$ and $\mathcal{B}$ (outlined in yellow):



Arrangement $\mathcal{A}$ is right after the rightmost red frog has jumped for the first time. Let us define *the disbalance* of some arrangement as the absolute value of the difference between the number of the green frogs and the number of the red frogs on either side of the empty pad; clearly, the disbalance of both sides is equal in any arrangement. Initially the disbalance is $\mathfrak{n} = 3$, then it gets monotonusly down to zero, stays zero for a while, and then gets monotonously up to $\mathfrak{n} = 3$ in the final arrangement. Arrangement $\mathcal{A}$ is the first one in which the disbalance is zero. Arrangement $\mathcal{B}$ is the last arrangement in which the disbalance is zero. To get from $\mathcal{A}$ to $\mathcal{B}$ takes precisely $\mathfrak{n} = 3$ jumps. To get from the initial arrangement to $\mathcal{A}$ takes the same number of jumps as the number of jumps from $\mathcal{B}$ to the final arrangment because those two sequences of jumps mirror each other. An arrangement is *interleaved* iff the colours of the frogs alternate along the lily pads sequence.

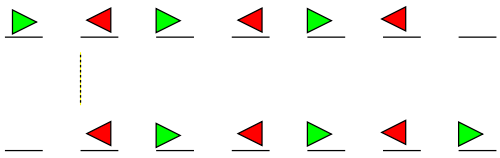
Let us generalise this idea. The sequence of jumps consists of three stages.

Stage i: From the initial arrangement to the first interleaved arrangement $\mathcal{A}$. The empty pad is at either the right end (as in the above example):

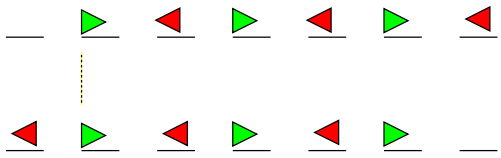
or the left end (there is nothing special about the red frogs):

In either case, the interleaved frogs start with a green frog on the left (and finish with a red frog on the right). Let us say it takes $\mathsf{P}(\mathfrak{n})$ jumps to accomplish **Stage i**.

Stage ii: From the first interleaved arrangement to the last interleaved arrangement. Either the green frogs "slide between" the red frogs as in:



or the red frogs slide between the green frogs as in:



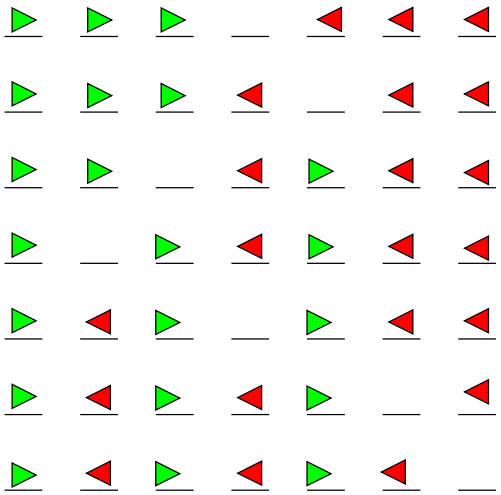
Stage ii takes precisely n jumps.

Stage iii: From the last interleaved arrangement to the final arrangement. This stage clearly mirrors **Stage i** and thus it takes $P(n)$ jumps.

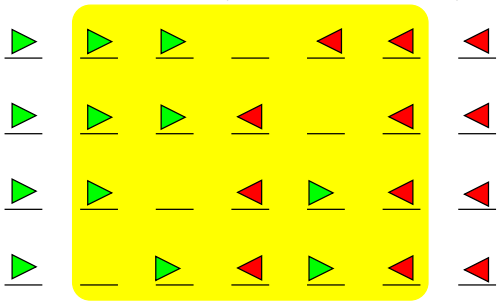
Let $T(n)$ be the number of jumps *for this particular scheme*. There is no natural recursive expression for $T(n)$ because the solution for n frogs does not contain the solution for $n-1$ frogs in any obvious way. However, we just derived:

$$T(n) = 2P(n) + n$$

The initial condition is $T(1) = 3$. In order to figure out what $P(n)$ is, consider **Stage i** of the solution for $n = 3$:



It clearly contains a solution for $n = 2$ (outlined in yellow):



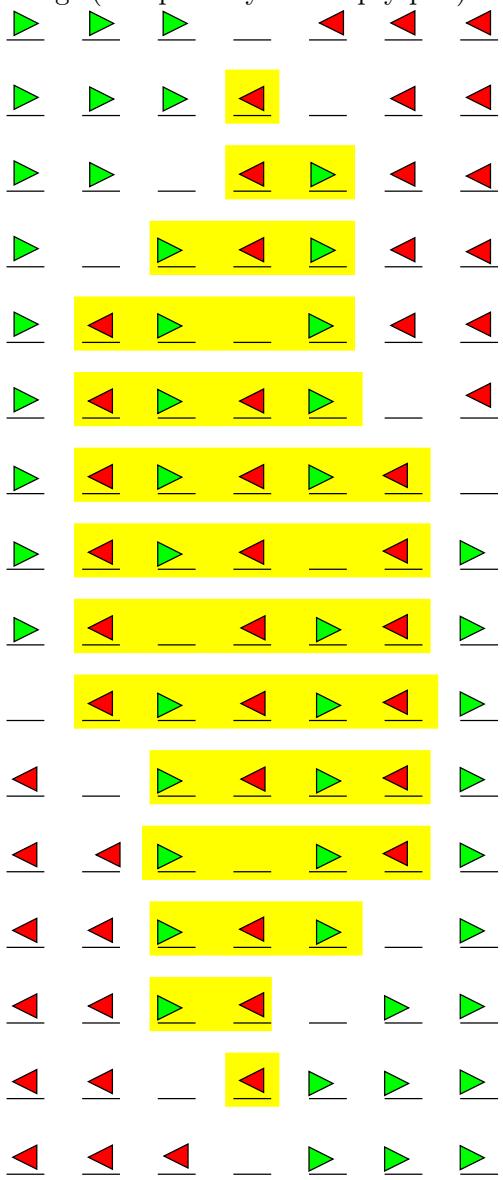
Let us generalise. **Stage i** for instance of size $n > 1$ consists of a **Stage i** for instance of size $n-1$, followed by n more jumps. Having in mind that $P(1) = 1$, it is the case that

$$P(1) = 1$$
$$P(n) = P(n-1) + n$$

The solution is $P(n) = \frac{1}{2}n(n+1)$, which the reader can easily verify by induction. Therefore,

$$T(n) = n(n+1) + n = n^2 + 2n \tag{8.39}$$

n^2+2n is a lower bound. So far we have demonstrated semi-rigorously that the scheme we introduced solves the problem in n^2+2n jumps for any n, thus a solution always exists. Now we argue that is a lower bound. Let us first define that *the inactive frogs* in any arrangement are the frogs that have either not jumped at all, or are already in their final positions. All the other forgs are *the active frogs*. See the following figure (the solution for $n=3$) in which the active frogs (and possibly the empty pad) are outlined in yellow:



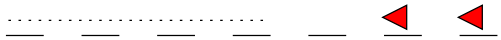
In each arrangement, the sub-arrangement of the active frogs is interleaved. This is no coincidence. The following fact is immediately obvious.

Observation 3. *Under the current names and assumptions, if during a sequence of jumps there comes an arrangement in which there are two adjacent green or red frogs, the desired final arrangement is unreachable.* □

Now we prove that **Stage i** is unavoidable. Without loss of generality, let the first frog among the two extreme frogs (leftmost and rightmost) that jumps be the red frog from pad

$2n+1$, as in the example above. We argue that this frog jumps to pad $2n$. By the rules of the games, it may potentially jump to pad $2n-1$, provided it is empty. Now we refute that possibility. Assume there is a successful sequence of jumps (*i.e.* it ends with the desired final arrangement) such that the first jump of the rightmost red frog is on pad $2n-1$. Then there is a frog on pad $2n$.

- If the frog on pad $2n$ is red:

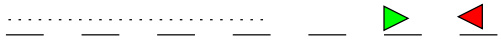
  

  then after the jump

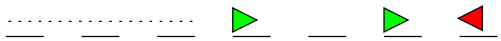
  

  there are two adjacent active red frogs and by Observation 3, the game is lost.

- If the frog on pad $2n$ is green:

  then the frog on pad $2n-2$ must be green:

  otherwise after the jump we will end up with two adjacent red frogs. But then, in order to avoid adjacent pair of same-colour frogs, it follows that between pads 1 and $2n-2$ inclusive the frog colours interchange like that:

  It is easy to see that in order to have that arrangement, at some prior moment the arrangement must have been:

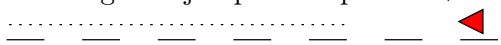
  

  and immediately prior to it:

  But then the first among the two leftmost and rightmost frogs that has jumped has been the green one on the left, contradicting the earlier assumption.

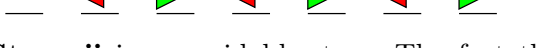
We proved that the rightmost frog first jumps from pad $2n+1$ to pad $2n$:



The sub-arrangement from position 1 to position $2n-1$ inclusive has disbalance 1, with one green frog more, and its must look like:

The jump of the rightmost frog yields the arrangement that we called $\mathcal{A}$ above (it is obvious how to generalise it for input of size n):

It follows that **Stage i** is indeed unavoidable. It is trivial to show that from arrangement $\mathcal{A}$ we have to get into arrangement $\mathcal{A}$ unless we lose the game:



In other words, **Stage ii** is unavoidable, too. The fact that **Stage iii** follows from considerations of symmetry with **Stage i**. It follows that n^2+2n is indeed a lower bound for the number of jumps. □

Problem 157. *Let* a *and* b *be integers such that* $a < b$. *Let* P *be a predicate defined over* $\{i \in \mathbb{N} \mid i \geq a\}$ *and* Q *be a predicate over* $\{i \in \mathbb{N} \mid i \geq b\}$. *Show that any proof by induction of the statement*

$$\forall n_{\,n \geq a}(P(n) \wedge (n \geq b \rightarrow Q(n)))$$

requires two bases because single basis for $n = a$ *does not suffice.*

Solution:
Consider a concrete example. It is well-known that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Therefore, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} + 9$ is false. Consider the statement

$$\forall n_{\,n \geq -15}\left(n = (n+1) - 1 \wedge \left(n \geq 0 \rightarrow \sum_{i=0}^{n} i = \frac{n(n+1)}{2} + 9\right)\right)$$

The statement is false. Consider the following invalid "proof by induction" for it.

Basis. $n = -15$. The substatement $n = (n+1) - 1$ is always true and so it is true for $n = -15$. The implication $n \geq 0 \rightarrow \sum_{i=0}^{n} i = \frac{n(n+1)}{2} + 9$ is true because its antecedent is false: -15 is not greater than or equal to 0. The conjunction of those two is then true, and the basis holds.

Inductive hypothesis. Assume the claim holds for some value n of the argument.

Inductive step. Consider the claim for value $n+1$ of the argument. Of course, $n+1 = (n+1+1) - 1$ holds. Furthermore,

$$\sum_{i=0}^{n+1} i = \left(\sum_{i=0}^{n} i\right) + n + 1 = \qquad \text{(* by the inductive hypothesis *)}$$

$$\frac{n(n+1)}{2} + 9 + n + 1 = \frac{n^2 + n + 2n + 2}{2} + 9 = \frac{(n+1)(n+2)}{2} + 9$$

And that concludes the proof.

What is wrong with this "proof" is that in the inductive step it substitutes the implication with the consequent of the implication and that is not right. The basis is surely true and the whole statement remains true for $n = -15, -14, \ldots, -1$ because the second substatement—the implication—remains true, its antecedent being false. However, for $n = 0$ the proof breaks because the antecedent is no longer false and the consequent $\sum_{i=0}^{0} i = \frac{0(0+1)}{2} + 9$ is false. Thus the statement for $n = -1$ does not imply it for $n = 0$. So, the proof is bogus. A proof by induction is valid whenever the statement for n implies the statement for $n+1$ over the whole range of values.

That example makes it clear that if we insist two prove

$$\forall n_{\,n \geq a}(P(n) \wedge (n \geq b \rightarrow Q(n)))$$

by induction we have to prove the following two statements separately by induction:

$$\forall n_{\,a \leq n < b}(P(n))$$
$$\forall n_{\,n \geq b}(P(n) \wedge Q(n))$$

Of course, if we have valid proofs for both statements we have a valid proof for the original one. But that invloves considering separate bases for $n = a$ and $n = b$. □

Chapter 9

Acknowledgements

I express my gratitude to:

> **Zornitsa Kostadinova**, **Iskren Chernev**, **Stoyan Dimitrov**, **Martin Toshev**, **Georgi Georgiev**, **Yordan Stefanov**, **Mariya Zhelezova**, and **Nikolina Eftimova**

for all the errors and typos they discovered and corrected in this manual. In addition, I thank **Georgi Georgiev** for suggesting solutions and improvement to several problems.

Bibliography

[AB98] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[Bal91] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1991. Available online at `http://books.google.com/books?id=pOBXUoVZ9EEC&printsec=frontcover&dq=Introductory+discrete+mathematics++By+V.+K.+Balakrishnan&source=bl&ots=1lYLvMpVfY&sig=Jwklfma4Zf3EIvNC0UH-fmI5JPA&hl=en&ei=1MCkTf2II4_1sgaR0ISBBw&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBcQ6AEwAA#v=onepage&q&f=false`.

[Buz99] Kevin Buzzard. Review of *Modular forms and Fermat's Last Theorem, by G. Cornell, J. H. Silverman, and G. Stevens, Springer-Verlag, New York, 1997, xix + 582 pp., $49.95, ISBN 0-387-94609-8*. Bulletin (New Series) of the American Mathematical Society, Volume 36, Number 2, Pages 261–266, 1999.

[CEPS08] Mark Cieliebak, Stephan Eidenbenz, Aris T. Pagourtzis, and Konrad Schlude. On the complexity of variations of equal sum subsets. *Nordic Journal of Computing*, 14(3):151–172, September 2008. Available online at `http://users.softlab.ece.ntua.gr/~pagour/papers/NJC09.pdf`.

[CLR00] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, first edition, 2000.

[CSS97] G. Cornell, J.H. Silverman, and G. Stevens. *Modular forms and Fermat's last theorem*. Springer, 1997. Available online at `http://books.google.com/books?id=Va-quzVwtMsC` .

[FHW80] Steven Fortune, John E. Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[GGJK78] M. R. Garey, F R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization, 1978. Available online at `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=741746BEDFA0899CA962D9C49034028D?doi=10.1.1.133.6943&rep=rep1&type=pdf`.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, second edition, 1973.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[LD05] Charles Leiserson and Erik Demaine. Assignments with solutions to free online course "Introduction to Algorithms", 2005. Available online at `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/assignments/`.

[Lei96] Leighton. Note on Better Master Theorems for Divide-and-Conquer Recurrences, 1996. Available online at `http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf`.

[MAMc] Minko Markov, Mugurel Ionuţ Andreica, Krassimir Manev, and Nicolae Ţăpuş. A linear time algorithm for computing longest paths in cactus graphs. Submitted for publication in Journal of Discrete Algorithms.

[Man05] Krasimir Manev. *Uvod v Diskretnata Matematika*. KLMN – Krasimir Manev, fourth edition, 2005.

[NIS] tree (data structure). National Institute of Standards and Technology's web site. URL `http://xlinux.nist.gov/dads//HTML/tree.html` .

[Pap95] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[Par95] Ian Parberry. *Problems on Algorithms*. Prentice Hall PTR, 1995. A second edition is available online with a special license `http://larc.unt.edu/ian/books/free/license.html`.

[RS95] Neil Robertson and Paul D. Seymour. Graph minors XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[Sch] Byron Schmuland. Shouting factorials! Available online at `www.math.ualberta.ca/pi/issue7/page10-12.pdf`.

[Sip06] Michael Sipser. *Introduction to the theory of computation: second edition*. PWS Pub., Boston, 2 edition, 2006.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[Slo] N. J. Sloane. The on-line encyclopedia of integer sequences. maintained by N. J. A. Sloane `njas@research.att.com`, available at `http://www.research.att.com/~njas/sequences/`.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley Professional, 2011.