Programming Contest Guide

Preparing for the ACM/ICPC World Finals

Andy Martin

Saturday, April 26, 2003

Programming Contest Guide

Preparing for the ACM/ICPC World Finals

Introduction

Programming contests are a fun way to learn more about quickly solving typical and atypical computer programming problems. There are many different formats that contests may take, but this document will focus on getting to and doing well in the arguably most famous contest of all: the Association of Computing Machinery (ACM) International Collegiate Programming Contest (ICPC) World Finals.

I am a student at the University of Kentucky pursuing a B.S. in Computer Science and a B.S. in Electrical Engineering. I am a two time participant in the ACM/ICPC world finals, and a three time participant in the mid-central regional contest. Throughout the document, I will draw upon my personal experience in these programming contests, upon the experience of teammates, upon the experience of our coach, and upon the wealth of information available online.

This document will first address the various means of preparing for such a contest. After a thorough discussion of preparation, a section will be devoted to participating in the contest. At the end, there will be a section on the various valuable references that there are both online and at the bookstore. And finally, there will be a brief discussion on the contents of the rather bulky code anthology.

It is suggested that you first review the rules of the ACM/ICPC World Finals and qualifying regional contests to familiarize yourself with the format of the contest. Also, it is recommended that you read a few of the problem statements from one of these contests (some are included in the code anthology) in order to understand the types of problems given.

Preparation

Study

A preliminary, but necessary, activity is study. The individual who wishes to participate and excel in the ICPC contest environment will need to have a good command of many different types of data structures and algorithms, and their respected implementations in C, C++ or Java. This study can sometimes be integrated into practice, but there is a certain amount of knowledge which is prerequisite in order to solve these types of computer programming problems. Recommended books for study are listed in the Reference section along with some online sources. A good preparation course in this respect would be your university's data structures class and its algorithms class. Also, many different types of math courses would be useful, such as graph theory, number theory, numerical methods, discrete math, linear algebra and operations research. It has been my experience that having at least one person on the team with an extensive background in mathematics helps tremendously in recognizing, dissecting and solving the really tough problems. Also, it is useful to have at least one person on the team who excels at the implementation of these problems in the contest languages. It is especially useful to have a firm command of the Standard Template Library (STL) in C++ and the various Java classes which are useful to the contest (such as BigInteger and BigReal).

Practice

Practice is by far the most important preparation tool. Teams with the best strategies, the best luck, the most knowledge and a library of reference material will still perform poorly on contest day if they have not practiced. The biggest issue here is time, not talent. To do well, one must devote the time to practice, and to practice hard.

Individual Practice

Practicing individually can be a useful tool. It can help you identify from a problem statement what the difficulty of the problem is, what algorithms will be useful in solving the problem, and how long the problem will take to solve and code.

The biggest temptation to avoid is sitting in front of the computer to flesh out your approach to solving the problem. It is strongly suggested that for every problem you attempt to solve, to flesh out the solution on paper. This will become a very handy skill in the contest when there is only one computer for three people. Also, it is much easier to verify and to code a pre-designed approach than one which is hacked out on the fly on the computer. Furthermore, drawing pictures and diagrams helps out tremendously in disassembling a problem into manageable chunks and in recognizing the key aspects of each problem.

The problems themselves can come from many sources. They may be posed by your coach, you may find them in a textbook, or perhaps come up with your own problems. By far the best source is the Internet. Many of the problems on the Internet have sample solutions and some even have the judge's input and output, or better yet, an online judge. A list of good sources for problems may be found in the Reference section and in the Code Anthology.

Learning how to test your problem for correctness is a vital skill. In the contest, you will not know if your program will be accepted unless you have thoroughly tested it. Good test cases are usually a combination of the sample test case (since you know what the correct output is supposed to be), and boundary cases. Also, stress testing the code is important since there is always a time limit associated with each problem. One thing that can be done is to use a scripting program, such as Python or Perl, to generate large test cases (or, you could use C/C++ or Java) instead of trying to type them out in a text editor.

Another practical matter is to use an environment as close to the contest environment as possible. The regional contest environment may be vastly different from the world finals environment, so be sure to choose an appropriate environment for the contest you are preparing for. I have also found that becoming familiar with a powerful text editor such as VIM or EMACS makes coding a lot faster. These editors are almost always part of a contest environment.

Avoid using a runtime debugger when practicing. In the contest, you will not have time on the computer to debug your code in that way. The best way to do debugging, is get your code to the point where it can generate output, run it for some test cases and print the source code and the buggy output and trace down the problem by hand. Printing is a very useful technique in the contest to optimize the amount of time on the computer.

Top Coder (http://www.topcoder.com/) is an excellent way to prepare for the personal challenge of beating the buzzer. It helps train you to think and code under pressure. Also, the challenge phase helps you to quickly spot problems with other's solutions to a problem which may help you debug a teammate's code during the contest (I have helped many teammates in the past spot an error they didn't see – this can be a very useful skill to develop). Furthermore, the challenge phase helps to develop a sense for the breaking cases – the test cases which may break a particular solution to a problem which is a vital skill for writing bug-free code the first time. Nothing is worse than designing a solution which works for some of the test cases, but falls far short for the boundary cases (such is the case with many problems where the search space can be very large – a trivial solution works on small cases, but takes too much time to run on large cases so a more clever solution must be found).

Team Practice

Team practice is far more important than personal practice. Team practice should be held in contest conditions; five to six hours working on a previously unseen set of problems as a team. Team practice will help you figure out each other's strengths and weaknesses. It will build the particular team dynamic. It will force you to figure problems out on paper (since there will only be one computer). It will help to build team endurance and rhythm, all of which are very important to contest performance.

Probably the most effective way to motivate and facilitate team practices is to offer a university course which encapsulates the practices. This way, students get credit for the work they put into the contest, and they will be better equipped to meet the challenge of the contest. Such courses already exist at some of the benchmark contest institutions (see Reference for more details). This course could teach students various algorithms, but the emphasis should be on becoming a cohesive team with a solid strategy and on personal practice which yields faster solving times of the problems.

Also, a useful aspect of such a course could be to compile an anthology of all problems successfully completed organized by topic in a fashion similar to the Code Anthology found at the end of this document. Since printed material is allowed at the contest, such known good code is an invaluable resource. Many problems resemble past problems, so entire functions may be lifted and reused saving precious time at the contest. For instance, if you have solved a problem last year which had to find connected components of a graph, and another problem arises at this year's contest which also requires finding connected components, much of the first code can be reused to speed up the development of the second.

Team Strategy

Team strategy can be a very complicated issue. In the two ICPC world finals I participated in, each team was unique and had a slightly different strategy. In 2001 we didn't perform well as a team. This was due to many factors; the main two were lack of practice as a team and lack of a cohesive strategy. At the contest, the computer became a bottleneck because one team member was having problems getting his two problems to work, and spent much time debugging on the computer (as noted in the Individual Practice section, this is a big no-no). Nothing is more irritating to your teammates than not being able to use the computer because the other is squandering precious moments tracing and debugging code on the computer instead of by hand.

In 2003 we vastly improved on the team level. We worked well together (and we had practiced together more than the previous year). Our team dynamic at the regional contest was excellent; we solved all seven problems by making good use of our team. At the finals, we let the pressure and the problem difficulty get to us a little, but we did better than the previous year, solving three problems instead of only two. We heavily used the print-and-switch technique, which is print a buggy program and switch so another can work on typing in his solution.

An important part of the team strategy should be to setup a rhythm where a problem is solved, the team switches who is at the computer, another problem is solved, switch, etc. This rhythm builds confidence and efficiency. The rhythm is broken when either an incorrect solution is submitted, or no one is ready to code on the computer. Either of these scenarios is bad for the rhythm, and does not optimize the available resources.

To establish a rhythm, much practice is needed. In every team, there will be one person who is fastest at the trivial programs. That person should be the first to code on the easiest problem. However, before coding begins, it is best for the team to look over the problem set and find an easy problem which they all agree is easy. This eliminates an individual from underestimating the amount of work a problem will take. Once the team has decided on a problem, the quick programmer should briefly scratch out an outline of a solution on paper, and then begin typing.

Now, while the quick programmer is solving the first problem (both on paper and on the computer), the other two team members should carefully read each problem statement and outline a solution. In this way, each problem can be broken down quickly at the beginning of the contest and classified according to its algorithmic difficulty and implementation difficulty. Sometimes, the most straightforward problem algorithmically will present many implementation difficulties. A good example from my experience was the problem I solved in the 2003 finals, Problem G (all the problems from that finals year are in the Code Anthology). I was the 'quick programmer' who started work on this problem, but I went a little too quickly, because there were many unanticipated kinks in the implementation. In actuality, problem J should have been the first to be solved (it is the easiest algorithmically and the easiest to implement).

After the problem set has been digested, the two team members who were reading switch to solving the next two easiest problems in detail on paper. Once the person on the computer is finished, he will switch with a teammate who has a solution ready to go. He will then begin work on the next problem, and so on, until the contest is about an hour or so from finishing. At this point, it is probably most advantageous to help debug if there are buggy programs, or to help solve the last problem as a team of two on paper. At any point in the contest, one of the non-computer members should be ready and willing to listen to another teammate explain the problem he is having with his program. The process of explaining almost always finds the bug; if not the extra pair of eyes should help find it.

Of course, team strategy will depend on the individual composition of the team. I have read of teams where they only used one person as an implementer, and the other two solved the problems algorithmically. This may fit some teams better than the above switching strategy. No matter what the strategy, it needs to be reinforced by much practice.

Luck

Luck is the last element needed for success. Unfortunately, this one is difficult, if not impossible, to control. The best teams sometimes have bad luck, and the worst teams sometimes have great luck.

An example of good 'luck' was the 2002 ACM mid-central regional I participated in. The problem set was manageable for the team members we had, and we established the rhythm as discussed above by solving most problems without a hitch. When a bug was encountered, we had no trouble printing and tracing by hand. When it was down to the final two problems, one teammate was dedicated to helping the other two debug and solve the last hard problems. And, to top it off, the last problem was submitted successfully with about 5 seconds to spare!

An example of bad 'luck' was the 2003 ICPC world finals. I was the quick programmer to start off the coding. I had a solution I had tested and believed was correct, so after getting approval from the team, I submitted it and printed it for reference in case it was rejected. After about 10 minutes, it came back as rejected, so I stopped working on the new problem I had picked up, and began working on finding the problem. I found a couple of suspect statements, traced them out by hand and discovered the problem. I switched out quickly and corrected the bug, tested the program, and resubmitted. It was rejected again about 10 minutes later. I printed the code, looked over it, and found some things I thought might be wrong but couldn't see how they would affect the solution. I corrected these maybe mistakes and resubmitted again. Again, it was rejected. Flustered, I poured over every line of code about three times, finding no mistakes. I finally just gave up. About five minutes later, my teammate said 'Yes!' The last two runs had been re-judged *as being correct*! This is an unheard of event in the world finals – that the judges would make such an over ruling. It turns out that the judges had not tested each problem, and that they had inadvertently matched the wrong input and output for the particular problem I was working on. Had we chosen problem J to start with, we would have avoided this problem as long as another team had submitted it first. This is extremely bad luck, because it destroyed our rhythm and flustered me for the whole contest. We are always trained to trust the judge; the judge is always right. Well, in this case the judge was somehow wrong.

Participation

Before the Contest

Don't use the few days before the contest to cram, but rather, use them to relax and to conserve energy for the big day. Approach the contest day like you would a big exam, such as the GRE. Make certain that you will be awake and operating at peak efficiency during the scheduled contest time, which is generally in the afternoon. If this is the world finals, don't stress out before hand. Relax and enjoy the trip to whatever exotic location they are having the finals at that year. If it is the regional contest, make sure you don't stay up late the night before.

During the Contest

Stay focused and don't let the pressure get to you. During the finals, the environment will be very distracting, with sixty or seventy teams discussing there problems, typing, etc. You may want to bring earplugs (they are allowed), but that will probably annoy your teammates when they try to talk to you. I suggest getting used to working in less than ideal surroundings. Maybe you could try practicing in your cafeteria during lunch or in some other noisy place. Don't compete directly with the other teams, just compete with the problems. Do the best you can given the circumstances and be happy with it. Between problems, take a small break to the bathroom and the food table to clear your mind. Sometimes, walking to the bathroom and back will help you think of a creative solution to the next problem, or maybe where your error was if you are debugging.

After the Contest

Spend the days after the contest debriefing each other on what when right, what when wrong, etc. Try to document your solutions, and to solve all the remaining problems. This will help refine your skills and your team.

Reference

Internet

The ICPC official website is http://icpc.baylor.edu/. It has all the previous world finals problems and links to each of the regional pages. The official page for the ACM mid-central region (the region U. of K. is in) is http://cs.smsu.edu/~mcpc/. The University of Waterloo also is a good resource for practice problems. The official site is http://plg.uwaterloo.ca/~acm00/. Other problem set pages include: http://www.acm.inf.ethz.ch/ProblemSetArchive.html, http://www.inf.bme.hu/contests/tasks/, http://www.karrels.org/Ed/ACM/, and http://www.informatik.uni-ulm.de/acm/index-en.html.

Top Coder is an individual contest in which you may be able to win money. The problem set is very similar to the ACM contest, while the rules and interface are quite different. The official Top Coder website is http://www.topcoder.com/.

La Unversidad de Valladolid has an excellent website with tons of problems with an online judge which will judge your submissions (note that for a few of the problems the judge does not work). The official page is http://acm.uva.es/.

A new book is being published about preparing for various programming contests. It is *Programming Challenges* by Steven S. Skiena and Miguel Revilla. The official website for the book is http://www.programming-challenges.com/pg.php?page=index.

Some web sites of university courses which prepare for the ACM contest (as of spring 2003):
http://www.cs.unr.edu/~westphal/cs491_spring_2003/
http://www.cs.sunysb.edu/~skiena/392/
http://www.cs.berkeley.edu/~hilfingr/csx98/
http://www.csee.wvu.edu/~callahan/cs191c/
http://www.cs.hmc.edu/ACM/

An invaluable online resource for any mathematical formulae or theorems is Eric Weissteins's World of Mathematics (this is closely akin to his *CRC Concise Encyclopedia of Mathematics*). The official site is http://mathworld.wolfram.com/.

The Java API can be found at http://java.sun.com/j2se/1.4.1/docs/api/. The official SGI Standard Template Library (STL) documentation can be found at http://www.sgi.com/tech/stl/.

Textbooks

You are allowed to take whatever printed material you want with you to the contests, so good reference material can help give you an edge on the harder problems in a problem set. One of the most useful books is the *CRC Concise Encyclopedia of Mathematics*, by Eric W. Weisstein. This book has just about every formula you could possibly need, and has many useful articles on famous numbers, series, etc. It is very helpful for geometry and number theory type problems (really any problem which involves some finer math). The other must have book is the so-called algorithm's bible: the *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein. This book has algorithms for many of the types of problems one will encounter in the programming contest. It is concise and has pseudo-code for all the algorithms, as well as asymptotic analysis. Another good reference for algorithms is *Algorithms in C++* by Robert Sedgewick. Also, it is a good idea to round out the collection with a book on numerical mathematics algorithms, such as *Numerical Mathematics and Computing* by Ward Cheney and David Kincaid and perhaps a reference on computation geometry (no recommendation yet on which book). Finally, syntax and API reference books can be very handy. A good C/C++ resource I recommend is the *C/C++ Programmer's Reference* by Herbert Schildt, and for Java I recommend *Java in a Nutshell* by David Flanagan (also, make sure you have access to the Java API which can be found online).

Guide to the Code Anthology

The code anthology contains printouts of many of the online resources, and it also contains sample problems by category along with the author's solution. Also, a CDROM will be packaged with this document with a soft copy of all problems, code, and sample input for some of the problems.

The Reference section of the code anthology contains various documents printed from the Internet which may be useful in the contest. For instance, it has a page about triangles. Also, the Java BigInteger section has the entire BigInteger API for Java.

The rest of the sections are categories of problems which have been solved. The author of the solutions is for the most part labeled in a comment at the top of each implementation file. (They were mostly coded by me, but there are a couple of solutions from other team members. If a file is not labeled, it was written by me). The source of all these problems is Top Coder, Valladolid, the regionals, or the world finals. Again, the source code is usually documented somewhere with the source of the problem. Some of the problems don't fit neatly in any category, but are placed in the closest category that fits the particular problem.

Index of the Code Anthology

12. Text Formatting
    12.1. Telephone Tangles. Valladolid problem 139.
    12.2. A Linking Loader. Problem G from the ACM/ICPC World Finals.

```
set shiftwidth=2
set tabstop=2
set expandtab
set autoindent
set cindent
```

```
// Andy’s simple template
#include <iostream>
#include <fstream>

using namespace std;

int n;

int main()
{
        // variables used in loops
        int i,j,k;
        char * fname = new char[sizeof(__FILE__)-4];
        char * fnamet = new char[sizeof(__FILE__)];

        ifstream infile;
        ofstream outfile;

        // construct filenames from __FILE__
        strncpy(fname,__FILE__,sizeof(__FILE__)-5);

        infile.open( strcat(strcpy(fnamet,fname),".in") );
        outfile.open( strcat(strcpy(fnamet,fname),".out") );

        // test file validity
        if( !infile )
        {
                cerr << "Bad input file." << endl;
                return -1;
        }
        if( !outfile )
        {
                cerr << "Bad output file." << endl;
                return -2;
        }

        infile >> n;
        // loop on each input
        while( n )
        {
                // do processing on each input
                infile >> n;
        }

        return 0;
}
```

ACM Crossroads Student Magazine
The ACM's First Electronic Publication

Crossroads Home
Join the ACM!
Search Crossroads
crossroads@acm.org

About Crossroads
Participate!
Submit Article
Subscribe
Link to us!

Index:
Back Issues
Articles By Topic
Columns
Reviews
Student Resources

Press Room
Privacy Statement

Crossroads en Español

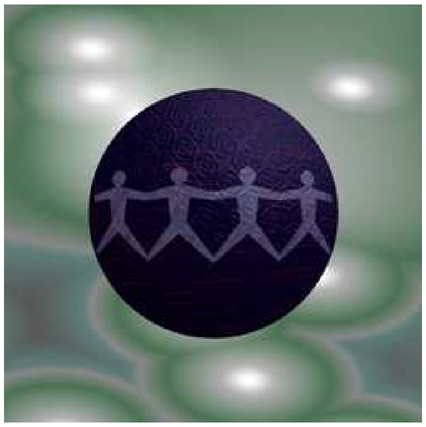
ACM / Crossroads / Xrds3-2 / Teamwork in Programming Contests: 3 * 1 = 4

Links2Go Key Resource
Programming Contests Topic
Awarded July 10, 2000 -- Press Release

Teamwork in Programming Contests: 3 * 1 = 4

by *Fabian Ernst*
Jeroen Moelands, and *Seppo Pieterse*

Introduction



Every year since 1977, the ACM has organized the ACM International Collegiate Programming Contest. This contest, which consists of a regional qualifying contest and the Finals, provides college students with the opportunity to demonstrate and sharpen their programming skills. During this contest, teams consisting of three students and one computer are to solve as many of the given problems as possible within 5 hours. The team with the most problems solved wins, where ``solved'' means producing the right outputs for a set of (secret) test inputs. Though the individual skills of the team members are important, in order to be a top team it is necessary to make use of synergy within the team. As participants in the 1995 Contest Finals (two of us also participated in the 1994 Finals), we have given a lot of thought to strategy and teamwork.

In this article we summarize our observations from various contests, and we hope that if you ever participate in this contest (or any other) that this information will be valuable to you.

The Basics: Practice, Practice, Practice!

Because of the fact that only one computer is available for three people, good teamwork is essential. However, to make full use of a strategy, it is also important that your individual skills are as honed as possible. You do not have to be a genius as practicing can take you quite far. In our philosophy, there are three factors crucial for being a good programming team:

- Knowledge of standard algorithms and the ability to find an appropriate algorithm for

every problem in the set;
- Ability to code an algorithm into a working program; and
- Having a strategy of cooperation with your teammates.

Team strategy will be the core discussion of this article. Nevertheless, there are some important considerations for improving your individual skills.

After analyzing previous contest programming problems, we noticed that the same kind of problems occurred over and over again. They can be classified into five main categories:

1. Search problems. These involve checking a large number of situations in order to find the best way or the number of ways in which something can be done. The difficulty is often the imposed execution time limit, so you should pay attention to the complexity of your algorithm.
2. Graph problems. The problems have a special structure so they can be represented as a graph-theoretical problem for which standard algorithms are available.
3. Geometrical problems. These involve geometrical shapes, lines, and angles.
4. Trivial problems. The choice of appropriate algorithm is clear, but these usually take quite a long time to program carefully.
5. Non-standard problems.

For the first three categories, standard algorithms are well documented in the literature, and you should program these algorithms beforehand and take the listings with you to the contest. In this way, you can avoid making the same (small) mistakes repeatedly and you can concentrate on the difficult aspects of the problem.

Another angle of practice is efficient programming. This does *not* mean type as fast as you can and subsequently spend a lot of time debugging. Instead, think carefully about the problem and all the cases which might occur. Then program your algorithm, and take the time to ensure that you get it right the first time with a minimum amount of debugging, since debugging usually takes a lot of valuable time.

To become a team, it is important that you play a lot of training contests under circumstances which are as close to the real contest as possible: Five hours, one computer, a new set of problems each time, and a jury to judge your programs.

Team Strategy: The Theory

When your algorithmic and programming skills have reached a level which you cannot improve any further, refining your team strategy will give you that extra edge you need to reach the top. We practiced programming contests with different team members and strategies for many years, and saw a lot of other teams do so too. From this we developed a theory about how an optimal team should behave during a contest. However, a refined strategy is not a must: The World Champions of 1995, Freiburg University, were a rookie team, and the winners of the 1994 Northwestern European Contest, Warsaw University, met only two weeks before that contest.

Why is team strategy important? There is only one computer, so it has to be shared. The problems have to be distributed in some way. Why not use the synergy that is always present within a team?

``Specialization" is a good way of using the inherent synergy. If each team member is an expert for a certain category of problem, they will program this problem more robustly, and maybe more quickly than the other two team members. Specialization in another sense is also possible. Maybe one of the team is a great programmer but has poor analytical skills, while another member can choose and create algorithms but cannot write bug-free programs. Combining these skills will lead to bug-free solutions for difficult problems!

Another way to use synergy is to have two people analyze the problem set. Four eyes see more than two, so it is harder for a single person to misjudge the difficulty of a problem. Forming a think-tank in the early stages of a contest might help to choose the best problems from the set and find correct algorithms for them. However, once the algorithm is clear, more than one member working on a single program should be avoided.

It is our experience that the most efficient way to write a program is to write it alone. In that way you avoid communication overhead and the confusion caused by differing programming styles. These differences are unavoidable, though you should try to use the same style standards for function and variable names. In this way you can really make 3*1 equal to four!

Other Considerations

Since the contest final standings are based on the number of problems correctly solved, and (in the case of ties) on the sum of elapsed time for each problem, a team should adopt a strategy that maximizes the number of solved problems at the end of the five hours, and view the total elapsed time as a secondary objective. In every contest there are some teams in the ``top six'' after three hours, that are not even in the ``top ten'' after the total five hours. The reverse also occurs. A long term strategy is therefore important: Try to optimize the 15 man hours and 5 hours of computer time, and do not worry about your total time or how quickly you solve the first two problems.

To optimize this scarce time, try to finish all problems that you start. A 99% solved problem gives you no points. Analyze the problem set carefully at the beginning (for example, by using a ``think-tank'' approach) to avoid spending more time than absolutely necessary on a problem that you will not finish anyway, and to avoid misjudging an easy problem as being too difficult. You need a good notion about the true difficulty of the various problems as this is the only way to be sure that you pick exactly those which you can finish within five hours.

Since you never have perfect information, you have to take risks. If you follow a risky strategy by choosing to tackle a large number of problems, you might end up in the bottom half of the score list when each is only 90% solved, or you might be the winner in the end. On the other hand, choosing a smaller number of problems has the risk that you have solved them correctly after four and a half hours, but the remaining time is too short to start and finish a new problem, thus wasting ten percent of the valuable contest time.

Time management should play a role in your strategy. If you are going to work on a large problem, start with it immediately or you will not finish it. Although this sounds trivial, there are a lot of teams which start out with the small problems, finish them quickly, and end up with only three problems solved because they did not finish the larger ones. In our opinion, debugging should have the highest priority at the terminal after 3.5 hours. When you start with a new problem that late in a contest, the terminal will become a bottleneck for the rest of the contest.

Of course terminal management is crucial. Though most programs are quite small (usually not exceeding one hundred lines of code), the terminal is often a bottleneck: Everyone wants to use it at the same time. How can this be avoided? The first thing to remember is: Use the chair in front of the terminal only for typing, not for thinking. Write your program on paper, down to the last semicolon. In this way you usually have a much better overview, and you have the time to consider all possible exceptions without someone breathing down your neck, longing for the terminal. Once you have finished writing, typing will take no more than 15 minutes. Though you should avoid debugging (this IS possible if you plan the program carefully on paper), when you really have to do it you should do it in a similar way:

Collect as much data as possible from your program, print it out and analyze it on paper together with your code listing. Real- time tracing is *THE ULTIMATE SIN*.

Some Example Strategies

1. The Simple Strategy

This strategy is quite useful for novice teams, or those who do not want to get into a lot of practice and strategy tuning, and, therefore, is in no way optimal. The basic idea is to work as individually as possible to try to minimize overhead. Everyone reads a couple of problems, takes the one he likes most and starts working on it. When a problem is finished, a new one is picked in the same way and so on.

Advantages are that little practice is needed. Total elapsed time will be minimal, since the easiest problems are solved first. However, there are also severe disadvantages: Since the easiest problems usually have the same level of difficulty, everyone will finish their problem at about the same time. Thus the terminal will not be used for the first hour, since everyone is working on a problem on paper, and remains a bottleneck thereafter. Furthermore, only the easy problems will be solved, because no time will be left for the hard ones. The conclusion is that, provided your programming skills are adequate, you will solve about three or four problems as a team. This will bring you, with a good total elapsed time, into the top ten, but probably not into the top three.

2. Terminal Man

In the terminal man (TM) strategy, only one of the team members, the T, uses the computer. The other two team members analyze the problem set, write down the algorithms and the (key parts) of the code, while the T makes the necessary I/O-routines. After an algorithm is finished, the T starts typing and, if necessary, does some simple debugging. If the bug is difficult to find, the original author of the algorithm helps the T to find it.

Advantages of this strategy are that the terminal is not a bottleneck anymore, and the task of solving a problem is split over people who specialized in the different parts of the problem solving process. A disadvantage is that no optimal use is made of the capacities of the T, who is mainly a kind of secretary. If you only one of you is familiar with the programming environment, this might be a good strategy. You can write a lot of programs in the first part of the contest when your brain is still fresh, since the typing and debugging is done by someone else. It depends strongly on the composition of your team if this strategy is suitable for you.

3. Think Tank

The strategy we followed during the preparation and playing of the Contest Finals of 1995 made use of the above-mentioned ``think tank'' (TT). We felt that choosing and analyzing the problems was such a crucial task in the early stages of a contest that it should not be left to a single person. The two team members who are the best problem analyzers form the TT and start reading the problems. Meanwhile the third member, the ``programmer'', will type in some useful standard subroutines and all the test data, which are checked carefully. After 15 minutes, the TT discusses the problems briefly and picks the one most suitable for the third team member. After explaining the key idea to the programmer, they can start working on it. Then the TT discusses all problems thoroughly, and puts the main ideas of the algorithm down on paper. We found out that two people examining hard problems often lead to creative solutions. After one hour the TT had a good overview over the problem set, and all algorithms were found. The next decision is how many problems you want to solve.

The easiest or shortest problems are handled by the programmer, while the TT divides the other ones among themselves.

The terminal is only used for typing in code from paper or gathering information from a buggy program. If a program is rejected by the jury and no bug can be found, it is put aside until the last hour of the contest. In principle, after three and a half hours no more new code is typed. The team will briefly discuss the situation, and a plan is made for how to solve the problems which have yet to be debugged.

Some advantages of this approach are that you will almost always tackle the programs which have a reasonable chance of being solved correctly, and the hard problems can be solved because the TT will start working on them in an early stage of the contest. A clear disadvantage is that you will have a relatively slow start and your total time is not optimal. So to win, you need to solve one problem more than the other teams. We feel that for a team consisting of partners with about equal skills, this strategy will help you solve as many problems as possible.

Some Other Tips

You can practice a lot for a programming contest, and your strategy can be very good. However, luck always has its part in the contest and you have to live with that. Do not be disturbed by it (or the lack of it). Play your own contest. Never look at other team's standing, except to see if some teams solved a problem rather quickly that you thought to be too hard. If a program gets rejected by the jury, don't panic. Try to find the bug, there always is one. Consider especially the limits of your program, and ask yourself under which circumstances these limits will be exceeded. You do not have to submit a correct program. It only has to produce the right output for the jury input. Therefore you should program robustly and cleanly, and not write the shortest or fastest code possible. And always remember: Programming contests are great fun!

Concluding Remarks

In this article we have recorded some of our experiences with programming contest strategies. Though these strategies are very dependent on the individual qualities of the team members, the concepts apply equally to all teams. We hope that the information contained in this article will be useful to you, should you ever want to participate in the ACM Programming Contest (we definitely recommend it!). More information about the contest can be found on http://www.acm.org/~contest. A report of our experiences at the Contest Finals, including more considerations on team strategy, can be found at http://www.cs.vu.nl/~acmteam/ .

About the authors:

Fabian Ernst is a 24-year old PhD-student at Delft University of Technology, the Netherlands, in the field of Mathematical Physics, and also studies Administration Science in Delft. He competed once in the Contest Finals (for VU Amsterdam) in 1995. He can be reached at

ernst@math.tudelft.nl

Jeroen Moelands is a 22-year old computer science and business computer science student at Vrije Universiteit Amsterdam, the Netherlands. He competed in the Contest Finals both in 1994 and 1995. His email-address is:

jeroenm@cs.vu.nl

Seppo Pieterse is a 23-year old computer science and econometrics student at Vrije Universiteit Amsterdam, the Netherlands. He competed in the Contest Finals in 1994 and 1995, finishing 5th in 1994. He can be reached at

spieters@cs.vu.nl

Want more Crossroads articles about Programming Languages? Go to the index, or the next one or to the previous one.

Last Modified: Monday, 16-Jul-01 17:06:00
Location: www.acm.org/crossroads/xrds3-2/progcon.html

© Copyright 2000-2002 by ACM, Inc.

ACM Crossroads Readers Ring
[Join Now | Ring Hub | Random | **<< Prev** | **Next >>**]

24560 hits since February 13, 2000

Coach of Programming Contest World Champions Shares Strategy

The 26th Annual ACM International Collegiate Programming Contest (ICPC) was held in Honolulu, Hawaii on March 23rd and was sponsored by IBM. The title of 2002 World Champions went to the team from Shanghai Jiao Tong University in China, which beat out 64 world finalist teams to win. As contest Executive Director Bill Poucher noted in a recent interview, "We've got to shine the spotlight on the next generation of leaders." View winning team.

MemberNet interviewed Yu Yong, the coach, to learn the secrets of their success.

MN: Why does your team work well together?

YY: The most important reason is that we all have the same goal and we all work toward that goal. The team members all have a clear understanding of each other's strengths and weaknesses, and they completely trust each other. We also have some discipline and rules that every team member must abide by. Training together for a long time also plays an important role in reinforcing the tight cooperation between the team members. During the training, the team members also develop friendship and tacit understanding.

MN: What did your team do differently from the other teams?

YY: We can carry out our strategy and tactics thoroughly and have a tight control over the timing and speed. The fast speed of problem solving is one of our advantages. In the World Finals, we solved the first problem in 17 minutes and it built a solid confidence among the team members. We are also good at solving difficult problems. The time saved from [solving] the easy problems provided us the opportunity to solve more difficult problems.

MN: How did you prepare for the competition?

YY: We have a long-term training program which includes both personal training and team training. Personal training mainly consists of algorithm reading and understanding, and program writing. Team training is actually a simulated contest and is carried out in a contest environment. Team training starts with trying out and using various algorithms so that the basic problem- solving skills can be established. After that, every team will try to find the right tactics for the team. The tactics are continuously practiced, evaluated and adjusted during the team training. They are also tested in the regional contests. After the regional contests, the final tactics and strategies for the World Finals are determined. The training before the World Finals further sharpens every team member's skills according to the tactics and strategies.

MN: Was this your first time at the Finals?

YY: It is the first time for one of our team members. The other two team members have had the experience of being at the Finals.

MN: Which problem was the hardest, and why?

YY: The last one is the hardest. It is very complex and has many details that need substantial time to deal with. The large number of details also leads easily to errors in programming.

Problem I: Merrily, We Roll Along!

One method used to measure the length of a path is to roll a wheel (similar to a bicycle wheel) along the path. If we know the radius of the wheel and the number of revolutions it makes as it travels along the path, the length of the path can be computed.

This method works well if the path is smooth. But when there are curbs or other abrupt elevations changes in the path, the path distance may not be accurately determined, because the wheel may rotate around a point (like the edge of a curb), or the wheel may roll along a vertical surface. In this problem you are to determine the distance moved by the center of such a wheel as it travels along a path that includes only horizontal and vertical surfaces.

To measure a path, the wheel is placed with its center directly above the origin of the path. The wheel is then moved forward over the path as far as possible, always remaining in contact with the surface, ending with its center directly above the end of the path.

--excerpt from Problem I (in a total of 9 problems, A through I). For the complete problem set from this year's ICPC competition, go to http://acm.baylor.edu/icpc/ and click on the "problem set" link.

Coach Yu Yong (far right) and the Shanghai team.

[MemberNet homepage] [Top of Page]

© 2002 ACM, Inc. All rights reserved. ACM Privacy Policy.

Next: 5.2 Quadrilaterals
Up: 5 Polygons
Previous: 5 Polygons

5.1 Triangles

Because the angles of a triangle add up to 180°, at least two of them must be acute (less than 90°). In an **acute triangle** all angles are acute. A **right triangle** has one right angle, and an **obtuse triangle** has one obtuse angle.

The **altitude** corresponding to a side is the perpendicular dropped to the line containing that side from the opposite vertex. The **bisector** of a vertex is the line that divides the angle at that vertex into two equal parts. The **median** is the segment joining a vertex to the midpoint of the opposite side. See Figure 1.

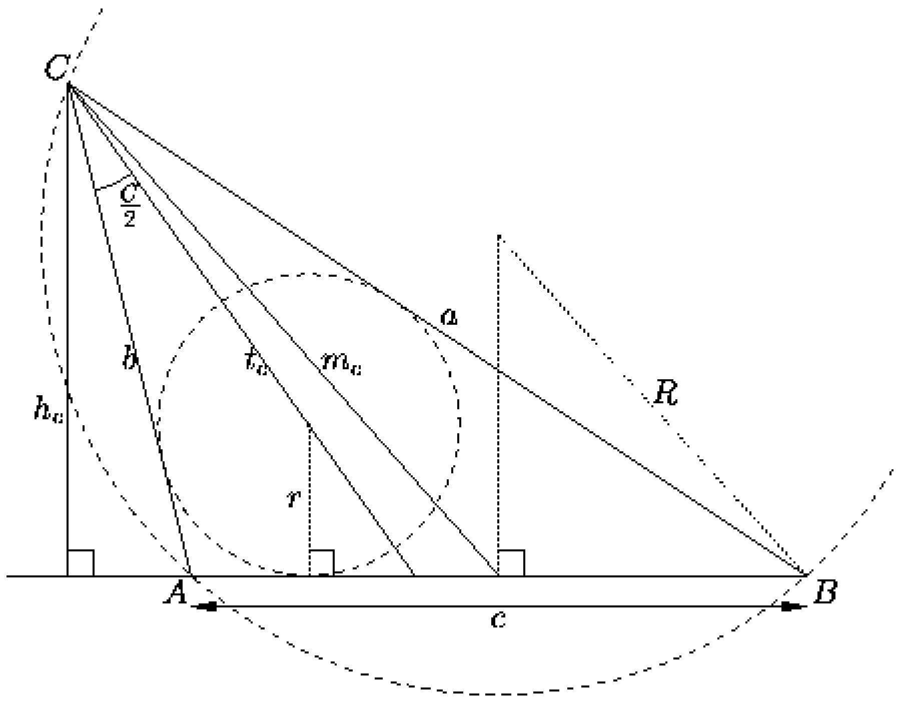


Figure 1: Notations for an arbitrary triangle of sides *a*, *b*, *c* and vertices *A*, *B*, *C*. The altitude corresponding to *C* is h_c, the median is m_c, the bisector is t_c. The radius of the circumscribed circle is *R*, that of the inscribed circle is *r*.

Every triangle also has an **inscribed circle** tangent to its sides and interior to the triangle (in other words, any three nonconcurrent lines determine a circle). The center of this circle is the point of intersection of the bisectors. We denote the radius of the inscribed circle by *r*.

Every triangle has a **circumscribed circle** going through its vertices; in other words, any three noncollinear points determine a circle. The point of intersection of the medians is the center of mass of the triangle (considered as an area in the plane). We denote the radius of the circumscribed circle by *R*.

Introduce the following notations for an **arbitrary triangle** of vertices *A*, *B*, *C* and sides *a*, *b*, *c* (see Figure 1). Let h_c, t_c and m_c be the lengths of the altitude, bisector and median originating in vertex *C*, let *r* and *R* be as usual the radii of the inscribed and circumscribed circles, and let *s*=½(*a*+*b*+*c*). Then:

$$\begin{aligned}
&A+B+C=180°,\\
&c^2=a^2+b^2-2ab\cos C \quad \textbf{(law of cosines)},\\
&a=b\cos C+c\cos B,\\
&\frac{a}{\sin A}=\frac{b}{\sin B}=\frac{c}{\sin C} \quad \textbf{(law of sines)},\\
&\text{area}=\tfrac12 h_c c=\tfrac12 ab\sin C=\frac{c^2\sin A\sin B}{2\sin C}=rs=\frac{abc}{4R}\\
&\quad=\sqrt{s(s-a)(s-b)(s-c)} \quad \textbf{(Heron)},\\
&r=c\sin\tfrac12 A\sin\tfrac12 B\sec\tfrac12 C=\frac{ab\sin C}{2s}=(s-c)\tan\tfrac12 C\\
&\quad=\left(\frac{1}{h_a}+\frac{1}{h_b}+\frac{1}{h_c}\right)^{-1},\\
&R=\frac{c}{2\sin C}=\frac{abc}{4\,\text{area}},\\
&h_c=a\sin B=b\sin A=\frac{2\,\text{area}}{c},\\
&t_c=\frac{2ab}{a+b}\cos\tfrac12 C=\sqrt{ab\left(1-\frac{c^2}{(a+b)^2}\right)},\\
&m_c=\sqrt{\tfrac12 a^2+\tfrac12 b^2-\tfrac14 c^2}.
\end{aligned}$$

A triangle is **equilateral** if all its sides have the same length, or, equivalently, if all its angles are the same (and equal to 60°). It is **isosceles** if two sides are the same, or, equivalently, if two angles are the same. Otherwise it is **scalene**.

For an **equilateral triangle** of side *a* we have:

$$\text{area}=\tfrac14 a^2\sqrt{3},$$
$$r=\tfrac16 a\sqrt{3},$$
$$R=\tfrac13 a\sqrt{3},$$
$$h=\tfrac12 a\sqrt{3},$$

where *h* is any altitude. The altitude, the bisector and the median for each vertex coincide.

For an **isosceles triangle**, the altitude for the unequal side is also the corresponding bisector and median, but this is not true for the other two altitudes. Many formulas for an isosceles triangle of sides *a*, *a*, *c* can be immediately derived from those for a right triangle of legs *a*, ½*c* (see Figure 2, left).

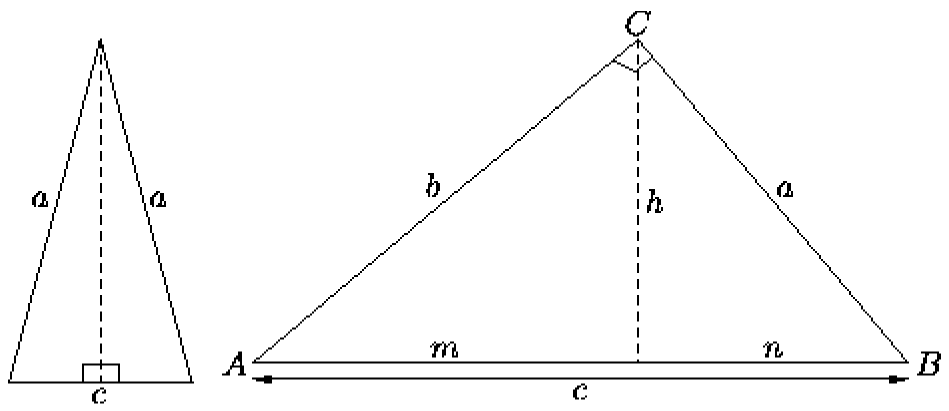


Figure 2: Left: An isosceles triangle can be divided into two congruent right triangles. Right: notations for a right triangle.

For a **right triangle** the **hypothenuse** is the longest side opposite the right angle; the **legs** are the two shorter sides, adjacent to the right angle. The altitude for each leg equals the other leg. In Figure 2 (right), *h* denotes the altitude for the hypothenuse, while *m* and *n* denote the segments into which this altitude divides the hypothenuse.

The following formulas apply for a right triangle:

$$A+B=90^\circ$$
$$r=ab/(a+b+c)$$
$$a=c\sin A = c\cos B$$
$$mc=b^2$$
$$\text{area}=\tfrac{1}{2}ab$$
$$c^2=a^2+b^2 \text{ (Pythagoras)}$$
$$R=\tfrac{1}{2}c$$
$$b=c\sin B = c\cos A$$
$$nc=a^2$$
$$hc=ab$$

The hypothenuse is a diameter of the circumscribed circle. The median joining the midpoint of the hypothenuse (the center of the circumscribed circle) to the right angle makes angles 2*A* and 2*B* with the hypothenuse.

Additional facts about triangles:

- In any triangle, the longest side is opposite the largest angle, and the shortest side is opposite the smallest angle. This follows from the law of sines.
- (**Ceva's Theorem**: see Figure 3, left.) In a triangle *ABC*, let *D*, *E* and *F* be points on the lines *BC*, *CA* and *AB*, respectively. Then the lines *AD*, *BE* and *CF* are concurrent if and only if the signed distances *BD*, *CE*, ... satisfy

$$BD \cdot CE \cdot AF = DC \cdot EA \cdot FB.$$

  This is so in three important particular cases: when the three lines are the medians, when they are the bisectors, and when they are the altitudes.

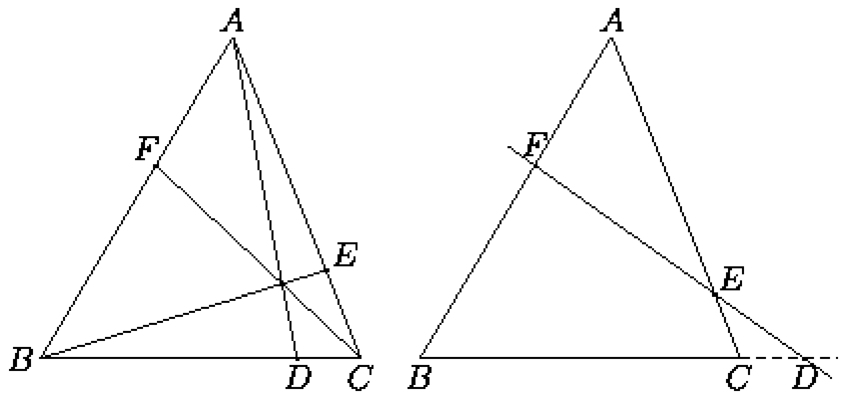


Figure 3: Left: Ceva's Theorem. Right: Menelaus's Theorem.

- (**Menelaus's Theorem**: see Figure 3, right.) In a triangle *ABC*, let *D*, *E* and *F* be points on the lines *BC*, *CA* and *AB*, respectively. Then *D*, *E* and *F* are collinear if and only if the signed distances *BD*, *CE*, ... satisfy

$$BD \cdot CE \cdot AF = -DC \cdot EA \cdot FB.$$

- Each side of a triangle is less than the sum of the other two. For any three lengths such that each is less than the sum of the other two, there is a triangle with these side lengths.

Next: 5.2 Quadrilaterals
Up: 5 Polygons
Previous: 5 Polygons

The Geometry Center Home Page

Silvio Levy
Wed Oct 4 16:41:25 PDT 1995

This document is excerpted from the 30th Edition of the CRC Standard Mathematical Tables and Formulas (CRC Press). Unauthorized duplication is forbidden.

Area of Triangles and Polygons (2D & 3D)

by Dan Sunday

Computing the area of a planar polygon is a basic geometry calculation and can be found in many introductory texts. However, there are several different methods for computing planar areas depending on the information available.

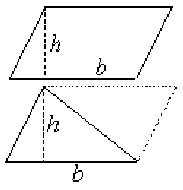
Triangles

Ancient Triangles

Before Pythagoras, the area of the parallelogram (including the rectangle and the square) has been known to equal the product of its base times its height. Further, two copies of the same triangle paste together to form a parallelogram, and thus the area of a triangle is half of its base *b* times its height *h*. So, for these simple but commonly occurring cases, we have:

Parallelogram: $A(\square) = bh$

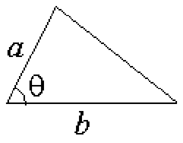
Triangle: $A(\Delta) = \frac{1}{2}bh$



However, except in special situations, finding the height of a triangle at an arbitrary orientation usually requires also computing the perpendicular distance of the top vertex from the base.

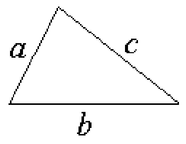
For example, if one knows the lengths of two sides, *a* and *b*, of a triangle and also the angle q between them, then Euclid says this is enough to determine the triangle and its area. Using trigonometry, the height of the triangle over the base *b* is given by *h* = *a* sin q, and thus the area is:

$$A(\Delta) = \frac{1}{2}ab\sin\theta$$



Another frequently used computation is derived from the fact that triangles with equal sides are congruent, and thus have the same area. This observation from Euclid (~300 BC) culminated in Heron's formula (~50 AD) for area as a function of the lengths of its three sides [Note: some historians attribute this result to Archimedes (~250 BC)]; namely:

$$A(\Delta) = \sqrt{s(s-a)(s-b)(s-c)}$$
$$\text{where } s = \tfrac{1}{2}(a+b+c)$$



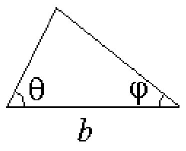
where *a*,*b*,*c* are the lengths of the sides, and *s* is the semiperimeter. There are interesting algebraic variations of this formula; such as:

$$A(\Delta) = \tfrac{1}{4}\sqrt{4a^2b^2 - \left(a^2+b^2-c^2\right)^2}$$

which avoids calculating the 3 square roots to explicitly get the lengths *a*,*b*,*c* from the triangle's vertex coordinates. Other variations on Heron's formula can be found at Kevin Brown's Heron's Formula page and Eric Weisstein's Triangle page.

The remaining classical triangle congruence is when two angles and one side are known. Knowing two angles gives all three, so we can assume the angles q and j are both adjacent to the known base *b*. Then the formula for area is:

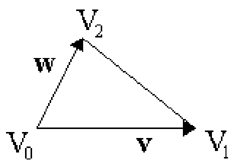
$$A(\Delta) = \frac{b^2}{2(\cot\theta + \cot\varphi)}$$



Modern Triangles

More recently, starting in the 17-th century with Descartes and Fermat, linear algebra produced new simple formulas for area. In *3 dimensional space* (3D), the area of a parallelogram and triangle can be expressed as the magnitude of the cross-product of two edge vectors, since |**v**×**w**| = |**v**||**w**||sin q| where q is the angle between the two vectors **v** and **w**. Thus for a 3D triangle with vertices $V_0V_1V_2$ putting **v**=V_1-V_0 and **w**=V_2-V_0, one gets:

$$A(\Delta) = \tfrac{1}{2}|\mathbf{v}\times\mathbf{w}|$$
$$= \tfrac{1}{2}\left|(V_1-V_0)\times(V_2-V_0)\right|$$

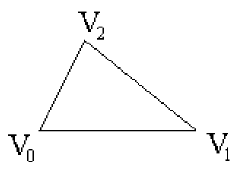


In *2 dimensional space* (2D), a vector can be viewed as embedded in 3D by adding a third component which is set = 0. This lets one take the cross-product of 2D vectors, and use it to compute area. Given a triangle with vertices $V_i=(x_i,y_i)=(x_i,y_i,0)$ for *i*=0,2, we can compute that:

$$(V_1-V_0)\times(V_2-V_0) = \left(0, 0, \begin{vmatrix}(x_1-x_0) & (x_2-x_0)\\(y_1-y_0) & (y_2-y_0)\end{vmatrix}\right)$$

And the absolute value of the third z-component is twice the absolute area of the triangle. However, it is useful to not take the absolute value here, and instead let the area be a signed quantity.

$$2A(\Delta) = \begin{vmatrix}(x_1-x_0) & (x_2-x_0)\\(y_1-y_0) & (y_2-y_0)\end{vmatrix} = \begin{vmatrix}x_0 & y_0 & 1\\x_1 & y_1 & 1\\x_2 & y_2 & 1\end{vmatrix}$$
$$= (x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0)$$
$$\text{where } V_i = (x_i, y_i)$$



This formula for area is a very efficient computation with no roots or trigonometric functions involved - just 2 multiplications and 5 additions, and possibly 1 division by 2 (which can sometimes be avoided).

Note that the signed area will be *positive* if $V_0V_1V_2$ are oriented counterclockwise around the triangle, and will be *negative* if the triangle is oriented clockwise; and so, this area computation can be used to test for a triangle's orientation. The signed area can also be used to test whether the point V_2 is to the left (positive) or the right (negative) of the directed line segment V_0V_1 . So this is a very useful primitive, and it's great to have such an efficient formula for it.

Quadrilaterals

The Greeks singled out certain quadrilaterals (also called quadrangles) for special treatment, including the square, the rectangle, the parallelogram, and the trapezium. Then, given an arbitrary quadrilateral, they showed how to construct a parallelogram [Euclid, Book I, Prop 45] or square [Euclid, Book II, Prop 14] with an equal area. And the area of a parallelogram was equal to its base times its height. But there was no general formula for the quadrilateral's area.

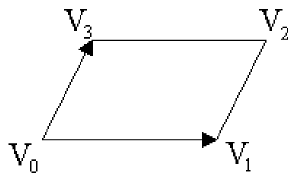
An extension of Heron's triangle area formula to quadrilaterals was discovered by the Hindu geometer Brahmagupta (~620 AD) [Coxeter,1967, Section 3.2]. However, it only works for ***cyclic quadrilaterals*** where all four vertices lie on the same circle. For a cyclic quadrilateral **Q**, let the lengths of the four sides be a, b, c, d, and the semiperimeter be s=(a+b+c+d)/2. Then, the area of **Q** is given by:

$$A(\Theta) = \sqrt{(s-a)(s-b)(s-c)(s-d)}$$

which is an amazing symmetric formula. If one side is zero length, say d=0, then we have a triangle (which is always cyclic) and this formula reduces to Heron's one.

In modern linear algebra, as already noted, the area of a ***planar parallelogram*** is the magnitude of the cross product of two adjacent edge vectors. So, for any 3D planar parallelogram $V_0V_1V_2V_3$, we have:

$$A(V_0V_1V_2V_3) = \left|(V_1 - V_0) \times (V_3 - V_0)\right|$$

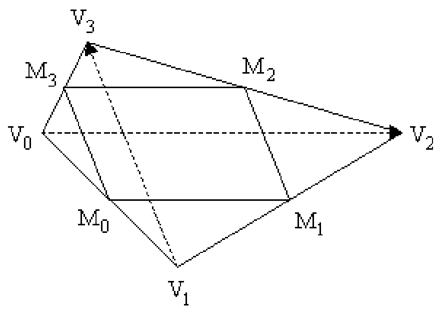


In 2D, with vertices $V_i=(x_i,y_i)=(x_i,y_i,0)$ for i=0,3, this becomes:

$$A(\square) = \begin{vmatrix} (x_1-x_0) & (y_1-y_0) \\ (x_3-x_0) & (y_3-y_0) \end{vmatrix}$$
$$= (x_1-x_0)(y_3-y_0)-(x_3-x_0)(y_1-y_0)$$

which is again a *signed* area just as we had for triangles.

Next, for an ***arbitrary quadrilateral***, one can compute its area using a parallelogram discovered by Pierre Varignon (first published in 1731). It is amazing that the Greeks missed Varignon's simple result which was discovered 2000 years after Euclid! Given any quadrilateral, one can take the midpoints of its 4 edges to get 4 vertices which form a new quadrilateral. It is then easy to show that this midpoint quadrilateral is always a parallelogram, called the "Varignon parallelogram", and that its area is exactly one-half the area of the original quadrilateral [Coxeter,1967, Section 3.1]. So, for a quadrilateral **Q**=$V_0V_1V_2V_3$, let this parallelogram have midpoint vertices $M_0M_1M_2M_3$ as shown in the diagram:



From elementary geometry, we know that in triangle $V_0V_1V_2$ the midpoint line M_0M_1 is parallel to the base V_0V_2. In triangle $V_0V_1V_2V_3$, the line M_3M_2 is parallel to that same base V_0V_2. Thus, M_0M_1 and M_3M_2 are parallel to each other. Similarly, M_0M_3 and M_1M_2 are parallel, which shows that $M_0M_1M_2M_3$ is a parallelogram. The area relation is also easy to demonstrate, and we can compute the quadrilateral's area as:

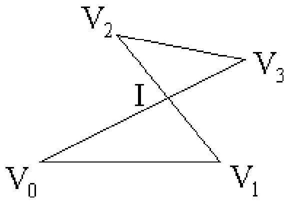
$$= 2\left|\left(\frac{}{2}-\frac{}{2}\right)\times\left(\frac{}{2}-\frac{}{2}\right)\right|$$
$$= \tfrac{1}{2}\left|\left(V_2 - V_0\right)\times\left(V_3 - V_1\right)\right|$$

which equals one-half the magnitude of the cross-product of the two diagonals of the quadrilateral. This result was noted by [Van Gelder,1995] who used a different proof.. This formula holds for any 3D planar quadrilateral. When restricted to 2D with $V_i=(x_i,y_i)$, it becomes a formula for a *signed* area:

$$2A(\Theta) = \begin{vmatrix} (x_2 - x_0) & (y_2 - y_0) \\ (x_3 - x_1) & (y_3 - y_1) \end{vmatrix}$$
$$= (x_2 - x_0)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_0)$$

This formula for an arbitrary quadrilateral is just as efficient as the one for an arbitrary triangle, using only 2 multiplications and 5 additions. For simple quadrilaterals, the area is positive when the vertices are oriented counterclockwise, and negative when they are clockwise. However, it also works for nonsimple quadrilaterals and is equal to the difference in area of the two regions the quadrilateral bounds. For example, in the following diagram where I is the self-intersection point of a nonsimple quadrilateral **Q**=$V_0V_1V_2V_3$, we have:

$$\begin{aligned} A(\Theta) &= A(\triangle V_0V_1I) + A(\triangle IV_2V_3) \\ &= A(\triangle V_0V_1I) - A(\triangle IV_3V_2) \end{aligned}$$

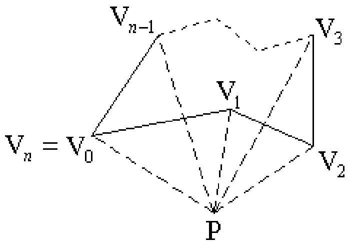


Polygons

2D Polygons

A 2D polygon can be decomposed into triangles. For computing area, there is a very easy decomposition method for ***simple polygons*** (i.e. ones without self intersections). Let a polygon **W** be defined by its vertices $V_i=(x_i,y_i)$ for i=0,n with $V_n=V_0$. Also, let P be any point; and for each edge V_iV_{i+1} of **W**, form the triangle $\mathbf{D}_i$=**D**PV_iV_{i+1}. Then, the area of **W** is equal to the sum of the *signed* areas of all the triangles $\mathbf{D}_i$ for i=0,n-1; and we have:

$$A(\Omega) = \sum_{i=0}^{n-1} A(\Delta_i)$$
$$\text{where } \Delta_i = \Delta PV_i\,V_{i+1}$$



Notice that, for a counterclockwise oriented polygon, when the point P is on the "inside" left side of an edge V_iV_{i+1}, then the area of $\mathbf{D}_i$ is positive; whereas, when P is on the "outside" right side of an edge V_iV_{i+1}, then $\mathbf{D}_i$ has a negative area. If instead the polygon is oriented clockwise, then the signs are reversed, and inside triangles become negative.

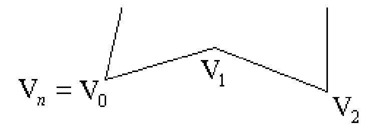
For example, in the above diagram, the triangles $\mathbf{D}_2$=**D**PV_2V_3 and $\mathbf{D}_{n-1}$=**D**P$V_{n-1}V_n$ have positive area, and contribute positively to the total area of polygon **W**. However, as one easily observes, only part of $\mathbf{D}_2$ and $\mathbf{D}_{n-1}$ are actually inside **W** and there is a part of each triangle that is also exterior. On the other hand, the triangles $\mathbf{D}_0$ and $\mathbf{D}_1$ have negative area and this cancels out the exterior excesses of positive area triangles. In the final analysis, exterior areas all get cancelled, and one is left with exactly the area of the polygon **W**.

One can make the formula more explicit by picking a specific point P and expanding the terms. By selecting P=(0,0), the area of each triangle reduces to $2A(\mathbf{D}_i) = (x_iy_{i+1}-x_{i+1}y_i)$. This yields:

$$= \sum_{i=0} (x_i + x_{i+1})(y_{i+1} - y_i)$$

$$= \sum_{i=1}^{n} x_i \left(y_{i+1} - y_{i-1}\right)$$

$$\textbf{where} \quad \mathrm{V}_i = \left(x_i, y_i\right), \ \textbf{with } i \left(\textbf{mod } n\right)$$



A little algebra shows that the second and third summations are equal to the first. For a polygon with *n* vertices, the first summation uses 2*n* multiplications and (2*n*-1) additions; the second uses *n* multiplications and (3*n*-1) additions; and the third uses only *n* multiplications and (2*n*-1) additions. So, the third is preferred for efficiency, but to avoid any overhead from computing the index *i*(mod *n*), one must extend the polygon array up to $V_{n+1}=V_1$.

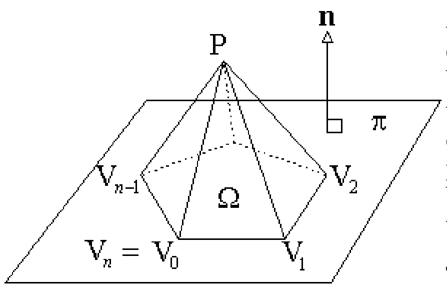
This computation gives a *signed* area for a polygon; and, similar to the signed area of a triangle, is *positive* when the vertices are oriented counterclockwise around the polygon, and *negative* when oriented clockwise. So, this computation can be used to test for a polygon's global orientation. However, there are other more efficient algorithms for determining polygon orientation. The easiest is to find the rightmost lowest vertex of the polygon, and then test the orientation of the entering and leaving edges at this vertex. This test can be made by checking if the end vertex of the leaving edge is to the left of the entering edge, which means that the orientation is counterclockwise.

3D Planar Polygons

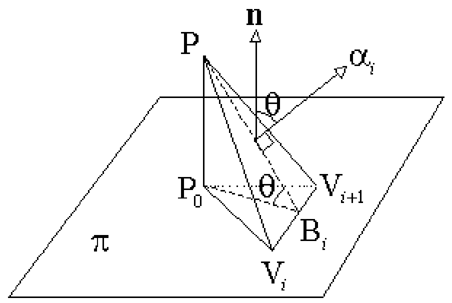
An important generalization is for planar polygons embedded in 3D space [Goldman, 1994]. We have already shown that the area of a 3D triangle **D**=$V_0V_1V_2$ is given by half the magnitude of the cross product of two of its edge vectors; namely, ½ $|V_0V_1 \times V_0V_2|$.

The Standard Formula

There is a classic standard formula for the area of a 3D polygon [Goldman, 1994] that extends the cross-product formula for a triangle. It can be derived from Stokes Theorem. However, we show here how to derive it from a 3D triangular decomposition that is geometrically more intuitive.



A general 3D planar polygon **W** has vertices $V_i=(x_i,y_i,z_i)$ for *i*=0,*n*, with $V_n=V_0$. All the vertices lie on the same 3D plane **p** which has a ***unit normal*** vector **n**. Now, the same as in the 2D case, let P be any 3D point (not generally on the plane **p**); and for each edge $\mathbf{e}_i=V_iV_{i+1}$ of **W**, form the 3D triangle $\mathbf{D}_i=\mathbf{D}PV_iV_{i+1}$. We would like to relate the sum of the areas of all these triangles to the area of the polygon **W** in the plane **p**. But what we have is a pyramidal cone with P as an apex over the polygon **W** as a base. We need to project the triangular sides of this cone onto the plane **p** of the base polygon, and compute signed areas of the projected triangles. If we can do this, then the sum of the projected areas will equal the total area of the planar polygon.



To achieve this, start by associating to each triangle $\mathbf{D}_i$ an area vector $\mathbf{a}_i$=½($PV_i \times PV_{i+1}$), perpendicular to $\mathbf{D}_i$, whose magnitude we know is equal to that triangle's area. Next, drop a perpendicular from P to a point P_0 on **p**, and consider the projected triangle $\mathbf{T}_i=\mathbf{D}P_0V_iV_{i+1}$. Drop a perpendicular P_0B_i from P_0 to B_i on the edge $\mathbf{e}_i=V_iV_{i+1}$. Since PP_0 is also perpendicular to $\mathbf{e}_i$, the three points PP_0B_i define a plane normal to $\mathbf{e}_i$, and PB_i is a perpendicular from P to the edge $\mathbf{e}_i$. Thus, $|PB_i|$ is the height of $\mathbf{D}_i$, and $|P_0B_i|$ is the height of $\mathbf{T}_i$. Further, the angle between these two altitudes = q = the angle between **n** and $\mathbf{a}_i$ since a 90° rotation results in congruence. This gives:

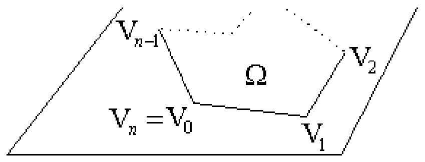
$$A\left(\mathrm{T}_i\right) = \tfrac{1}{2}\left|\mathrm{V}_i\mathrm{V}_{i+1}\right|\left|\mathrm{P}_0\mathrm{B}_i\right| = \tfrac{1}{2}\left|\mathrm{V}_i\mathrm{V}_{i+1}\right|\left|\mathrm{PB}_i\right|\cos\theta = A\left(\Delta_i\right)\cos\theta = \left|\mathbf{n}\right|\left|\boldsymbol{\alpha}_i\right|\cos\theta = \mathbf{n}\cdot\boldsymbol{\alpha}_i$$

This *signed* area computation is positive if the vertices of $\mathbf{T}_i$ are oriented counterclockwise when we look at the plane **p** from the side pointed to by **n**. The same as in the 2D case, we can now add together the signed areas of all the triangles $\mathbf{T}_i$ to get the area of the polygon **W**. Writing this down, we have:

$$A(\Omega) = \sum_{i=0}^{n-1} A\left(\mathrm{T}_i\right) = \sum_{i=0}^{n-1} \mathbf{n}\cdot\boldsymbol{\alpha}_i = \frac{\mathbf{n}}{2}\cdot\sum_{i=0}^{n-1}\left(\mathrm{PV}_i \times \mathrm{PV}_{i+1}\right)$$

Finally, by selecting P=(0,0,0), we have $PV_i=V_i$ and this yields the simple formula:

$$2A(\Omega) = \mathbf{n}\cdot\sum_{i=0}^{n-1}\left(\mathrm{V}_i \times \mathrm{V}_{i+1}\right)$$



which uses 6n+3 multiplications and 4n+2 additions

Similar to the 2D case, this is a *signed* area which is positive when the vertices are oriented counterclockwise around the polygon when viewed from the side of **p** pointed to by **n**.

Quadrilateral Decomposition

[Van Gelder, 1995] has shown how to significantly speed up this computation by using a decomposition into quadrilaterals instead of triangles. He first noted that the area of a 3D planar quadrilateral **Q**=$V_0V_1V_2V_3$ can be computed in terms of the cross-product of its diagonals; that is:

$$2A(\Theta) = \mathbf{n} \cdot \left[(V_2 - V_0) \times (V_3 - V_1) \right]$$

which reduces four expensive cross-product computations to just one!

Then, any large polygon **W** with n>4 vertices can be decomposed into quadrilaterals formed by V_0 and three other sequential vertices V_{2i-1}, V_{2i}, and V_{2i+1} for i=1,h where h= the greatest integer £(n-1)/2. For n odd, the decomposition ends with a triangle. This gives:

$$2A(\Omega) = \mathbf{n} \cdot \left(\sum_{i=1}^{h-1} (V_{2i} - V_0) \times (V_{2i+1} - V_{2i-1}) + (V_{2h} - V_0) \times (V_k - V_{2h-1}) \right)$$

where k=0 for n odd, and k=n-1 for n even. This formula reduces the number of expensive cross-products by a factor of two (replacing them with vector subtractions). In total there are 3n+3 multiplications and 5n+1 additions making this formula roughly twice as fast as the classic standard one.

[Van Gelder, 1995] also states that this method can be applied to 2D polygons, but he does not write down the details. Working this out produces a formula that uses n multiplications and 3n-1 additions, which is not as fast as the prior formulas we have given. We simply note this here, and do not pursue it further.

Projection to 2D

One can speed up the computation of 3D planar polygon area by projecting the polygon onto a 2D plane [Snyder & Barr, 1987]. Then, the area can be computed in 2D using our fastest formula, and the 3D area is recovered using an area scaling factor. This method is implemented by projecting onto an axis-aligned plane by ignoring one of the three coordinates. To avoid degeneracy and optimize robustness, we look at the plane's normal vector **n** (see the April 2001 Algorithm About Planes), and choose the component with the greatest absolute as the one to ignore. Let $\text{Proj}_c()$ be the projection that ignores the coordinate $c = x$, y, or z. Then, the ratio of areas for the projected polygon $\text{Proj}_c(\mathbf{W})$ and original planar polygon **W** with normal **n** = (n_x, n_y, n_z) is:

$$\frac{A(\text{Proj}_c(\Omega))}{A(\Omega)} = \frac{|n_c|}{|\mathbf{n}|} \quad \text{where } c = x, y, \text{ or } z$$

Thus, the 3D planar area can be recovered by a single extra multiplication, and in total this algorithm uses n+5 multiplications, 2n+1 additions, 1 square root (when **n** is not a unit normal), plus a small overhead choosing the coordinate to ignore. This is a very significant improvement of the classic standard formula, achieving almost a 6 times speed-up. We give an efficient implementation below in the routine `area3D_Polygon()`.

Implementations

Here are some sample "C++" implementations of these formulas as algorithms. We just give the 2D case with integer coordinates, and use the simplest structures for a point, a triangle, and a polygon which may differ in your application. We represent a polygon as an array of points, but it is often more convenient to have it as a linked list of vertices (to allow insertion or deletion during drawing operations), and the polygon routines can be easily modified to scan through the linked list (see [O'Rourke, 1998] for an example of this approach).

```
// Copyright 2000, softSurfer (www.softsurfer.com)
// This code may be freely used and modified for any purpose
// providing that this copyright notice is included with it.
// SoftSurfer makes no warranty for this code, and cannot be held
// liable for any real or imagined damage resulting from its use.
// Users of this code must verify correctness for their application.

// a Point (or vector) is defined by its coordinates
typedef struct {int x, y, z;} Point;    // exclude z for 2D
// a Triangle is given by three points: Point V0, V1, V2
// a Polygon is given by:
//        int n = number of vertex points
//        Point* V[] = an array of points with V[n]=V[0], V[n+1]=V[1]

// Note: for efficiency low-level functions are declared to be inline.

// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2 on the line
//            <0 for P2 right of the line
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x - P0.x) * (P1.y - P0.y) );
}
//===================================================================

// orientation2D_Triangle(): test the orientation of a triangle
//    Input:  three vertex points V0, V1, V2
//    Return: >0 for counterclockwise
//            =0 for none (degenerate)
//            <0 for clockwise
inline int
orientation2D_Triangle( Point V0, Point V1, Point V2 )
{
    return isLeft(V0, V1, V2);
}
//===================================================================

// area2D_Triangle(): compute the area of a triangle
//    Input:  three vertex points V0, V1, V2
//    Return: the (float) area of T
inline float
area2D_Triangle( Point V0, Point V1, Point V2 )
{
    return (float)isLeft(V0, V1, V2) / 2.0;
}
//===================================================================

// orientation2D_Polygon(): tests the orientation of a simple polygon
//    Input:  int n = the number of vertices in the polygon
//            Point* V = an array of n+1 vertices with V[n]=V[0]
//    Return: >0 for counterclockwise
//            =0 for none (degenerate)
//            <0 for clockwise
//    Note: this algorithm is faster than computing the signed area.
int
orientation2D_Polygon( int n, Point* V )
{
    // first find rightmost lowest vertex of the polygon
    int rmin = 0;
    int xmin = V[0].x;
    int ymin = V[0].y;

    for (int i=1; i<n; i++) {
        if (V[i].y > ymin)
            continue;
        if (V[i].y == ymin) {    // just as low
            if (V[i].x < xmin)   // and to left
                continue;
        }
        rmin = i;          // a new rightmost lowest vertex
        xmin = V[i].x;
        ymin = V[i].y;
    }

    // test orientation at this rmin vertex
    // ccw <=> the edge leaving is left of the entering edge
    if (rmin == 0)
        return isLeft( V[n-1], V[0], V[1] );
    else
        return isLeft( V[rmin-1], V[rmin], V[rmin+1] );
}
//===================================================================

// area2D_Polygon(): computes the area of a 2D polygon
//    Input:  int n = the number of vertices in the polygon
//            Point* V = an array of n+2 vertices
//                       with V[n]=V[0] and V[n+1]=V[1]
//    Return: the (float) area of the polygon
float
area2D_Polygon( int n, Point* V )
{
    float area = 0;
    int   i, j, k;     // indices

    for (i=1, j=2, k=0; i<=n; i++, j++, k++) {
        area += V[i].x * (V[j].y - V[k].y);
    }
    return area / 2.0;
}
//===================================================================

// area3D_Polygon(): computes the area of a 3D planar polygon
//    Input:  int n = the number of vertices in the polygon
//            Point* V = an array of n+2 vertices in a plane
//                       with V[n]=V[0] and V[n+1]=V[1]
//            Point N = unit normal vector of the polygon's plane
//    Return: the (float) area of the polygon
float
area3D_Polygon( int n, Point* V, Point N )
{
    float area = 0;
    float an, ax, ay, az;  // abs value of normal and its coords
    int   coord;           // coord to ignore: 1=x, 2=y, 3=z
    int   i, j, k;         // loop indices

    // select largest abs coordinate to ignore for projection
    ax = (N.x>0 ? N.x : -N.x);     // abs x-coord
    ay = (N.y>0 ? N.y : -N.y);     // abs y-coord
    az = (N.z>0 ? N.z : -N.z);     // abs z-coord

    coord = 3;                     // ignore z-coord
    if (ax > ay) {
        if (ax > az) coord = 1;    // ignore x-coord
    }
    else if (ay > az) coord = 2;   // ignore y-coord

    // compute area of the 2D projection
    for (i=1, j=2, k=0; i<=n; i++, j++, k++)
        switch (coord) {
        case 1:
            area += (V[i].y * (V[j].z - V[k].z));
            continue;
        case 2:
            area += (V[i].x * (V[j].z - V[k].z));
            continue;
        case 3:
            area += (V[i].x * (V[j].y - V[k].y));
            continue;
        }

    // scale to get area before projection
    an = sqrt( ax*ax + ay*ay + az*az);  // length of normal vector
    switch (coord) {
    case 1:
        area *= (an / (2*ax));
        break;
    case 2:
        area *= (an / (2*ay));
        break;
    case 3:
        area *= (an / (2*az));
    }
    return area;
}
//===================================================================
```

References

Kevin Brown, MathPages : Heron's Formula

Donald Coxeter & Samuel Greitzer, Geometry Revisited (1967)

Ronald Goldman, "Area of Planar Polygons and Volume of Polyhedra" in Graphics Gems II (1994)

Joseph O'Rourke, Computational Geometry in C (2nd Edition), Section 1.3 "Area of a Polygon" (1998)

J.P. Snyder and A.H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations", ACM Comp Graphics 21, (1987)

Allen Van Gelder, "Efficient Computation of Polygon Area and Polyhedron Volume" in Graphics Gems V (1995)

Eric Weisstein, MathWorld Site: Triangle or in The CRC Concise Encyclopedia of Mathematics (1998)

Copyright © 2001-2003 softSurfer. All rights reserved.
Email comments and suggestions to feedback@softsurfer.com

An Algorithm for Generating Highly Composite Numbers

D. B. Siano and J. D. Siano

Oct. 7, 1994

Introduction

All positive integers can be written in only one way as a product of powers of primes:

$$n = 2^{a_2} 3^{a_3} 5^{a_5} \dots . p^{a_p} . \qquad (1)$$

The number of distinct divisors of n is, since each power of a prime factor can include zero,

$$d(n) = (a_2 + 1)(a_3 + 1)(a_5 + 1)\dots(a_p + 1). \qquad (2)$$

When n=12, for example, eqs. 1 and 2 yield 12 = 2^2 3^1 so d(n)= (2+1) (1+1) = 6. The divisors of 12 can be readily enumerated as {1,2,3,4,6,12}.

The divisor function d(n) has been of great interest to number theorists for a long time. It fluctuates wildly from one integer to the next, and one might think it would be quite unpredictable. However, it is actually possible to derive some simple rules about its average behavior. One result, for example, is that the average of the number of divisors of n from 1 to N is approximately ln[N].

Prime numbers, of course, have the minimum number of divisors possible: 2 (1 and the prime number itself). It is natural to examine the numbers at the other end of the range--the numbers that have the highest possible number of divisors. These numbers were first studied by S. Ramanujan and a number of interesting results were demonstrated.

Highly composite numbers

The definition of a highly composite number (integer) is that it is a number that has a larger number of divisors than any number less than itself. For example, 12 is a highly composite number, because it has 6 divisors while every number less than 12 has a smaller number of divisors: e.g. 10, 8 and 6 have 4 divisors, 9 has

only 3, etc. The sequence of highly composite numbers starts out as 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, 1260, etc. The 100th number published by Ramanujan is 3212537328000 which has 8192 divisors. while the average number of divisors of all the numbers up to this one is about 29.

It is easy to show that composite numbers have some interesting simple properties. For example, for n to be highly composite, none of the prime factors in eq. 1 can be omitted. Thus,

$$k = 2^4 3^6 7^3$$

cannot be highly composite (5 is omitted) because

$$k' = 2^4 3^6 5^3$$

has the same number of factors as k, by virtue of eq. 2, but is smaller than k. Also, by similar reasoning, k' cannot be highly composite either, because

$$k'' = 2^6 3^4 5^3$$

is obviously smaller than k' and has the same number of factors as k'.

Clearly, for n to be highly composite, we must have

$$a_2 \geq a_3 \geq a_5 \geq a_7 \ldots \geq 1.$$

Ramanujan gave a list of the first one hundred of these numbers (with one error of omission), which he found "by trial". There are several outstanding conjectures about the properties of the sequence of highly composite numbers and it would be of some interest to have a method of generating them automatically. A very simple algorithm, in the form of a sieve, is

```
n:=2;  nd:=2;
label1:  n:=n+1;
if divisors(n)≤ nd  then  goto label1;
else nd:=divisors(n), print n, goto 1;
```

will in principle print out all of the highly composite numbers. However, this is much too slow for even moderately large n and it

quickly runs out of steam. Highly composite numbers are relatively rare: there are (roughly) only 10 per decade. Therefore speeding the sieve up by an order of magnitude will therefore result in only an additional 10 highly composite numbers. To calculate the thousandth highly composite number by the brute force sieve using a computer that tested one number per picosecond would still take many times (10^45 or so times!) the age of the universe.

The challenge is to discover a much faster algorithm that can yield highly composite numbers far larger than those already tabulated.

Some preliminary observations

It is instructive to examine a short section of the known highly composite numbers:

n	pwrs of primes	hc(n)/hc(n-1)
85	63221111	28/23
86	731111111	46/35
87	54221111	105/92
88	532111111	23/21
89	442111111	3/2
90	64221111	28/23
91	632111111	23/21
92	542111111	3/2
93	732111111	4/3
94	642111111	3/2
95	532211111	7/6
96	633111111	10/7
97	442211111	21/20

They are listed here in a compact way--only the powers of the successive primes are shown, e.g., for n=85, the highly composite number is

$$2^6 3^3 5^2 7^2 11^1 13^1 17^1 19^1 = 97772875200.$$

There are several things worth noting. The prime having the highest power is 2, and the power of the highest prime is unity. Ramanujan proved that this is nearly always true,with only two exceptions: 4 and 36. Unfortunately, the highest prime factor does not increase monotonically with n. The ratios of successive highly composite numbers are always rational numbers greater than one. The ratios are also easily seen to be less than or equal to two: multiplying any highly composite number by two always gives a larger number with a larger number of divisors. Thus, given a highly composite number n, we are guaranteed to always have at least one in the range between n and 2n.

The algorithm

A method for calculating successive highly composite numbers can be devised from these observations. The essential part is to get hc(n+1) from the previous one, hc(n) by multiplying it by a suitable ordered (increasing) list of rational numbers {r}, where each member of {r} is greater than one and less than or equal to two. The number

of divisors of each of the products is calculated and compared to the number of divisors of hc(n); the first one with a greater number of divisors is hc(n+1).

The interesting part is to calculate a suitable list {r}. We consider first the case where the highest prime factor, p, does not change from hc(n) to hc(n+1). It is useful to factor hc(n) into two parts: a highly variable one, which we denote v, and a part that is the same for both hc(n) and hc(n+1), which is defined as

$$s = \prod_{i=1}^{m} p_i ,$$

where m is the ordinal number of the highest prime factor. The other, more variable part, is given by

$$\mathrm{v} = \mathrm{hc(n)/s.}$$

For example for n = 85

$$s(85) = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$$

and

$$\mathrm{v}(85) = 2^5 3^2 5^1 7^1$$

If a maximum largest power (of 2) is assumed to be g (say 8, for example), then it is easy to generate a list of numbers like v with the constraint that the successive powers are non-increasing:

```
do i=g to 1 step -1
do j=i to 1 step -1
do k=j to 1 step -1
do l=k to 1 step -1
v(i,j,k,l)= 2^i 3^j 5^k 7^l
end do
```

When g is not too large, nor the number of prime factors under consideration, f, (f=four in the case here) too many, the list of v's is not so large as to be unmanageable. It is not hard to show that the number of terms in v is

$$n = \frac{\prod_{i=1}^{f}(i+g)}{f!}.$$

Following the example along, when g=8 and f=4, this list has 495 terms in it. It can be narrowed to just those that are larger than v(85), and smaller than or equal to two times v(85). Multiplying each member of the list of v's by s, then ordering it gives a $_0$list, V of numbers that are candidates for hc(n+1). In the example, this gives a list of only 11 elements.

Sometimes the largest prime factor in hc(n+1) is larger than that in hc(n). To cover this possibility a small modification of the procedure is required. The same list of v's is used, but the search is narrowed to just those that are between rp and 2 rp where rp = v(85)/pp, and pp is the next prime after p. Multiplying the list of v's by s pp will give a list Vp, which in the example given, has 7 members.

Similarly, when the largest prime factor in hc(n+1) is smaller than in hc(n), the search is narrowed to just those that a are between rm and 2 rm, where rm = v(85) pm, and pm is the prime before p. Multiplying the list of v's by s/pm will give a list Vm. For the example given it has 17 members.

Each of these three lists are then combined (Union) and has only 35 members for the example) into one list, ordered and searched for the first one having a larger number of divisors than hc(n) to give, finally hc(n+1). For the example, the correct result is found on the 13th try. Narrowing the search in this way minimizes the time and memory requirements, but is still clearly exhaustive of all of the possibilities, provided g and f are chosen to be large enough. Proper limits on these are discussed in the appendix.

The above process can then be repeated to give hc(n+2). In principle a new list of v's, with increased values for g and/or f may be assumed to get it. In practice it is better to make the list of v's large enough in the beginning to encompass the range of interest, and use the same list over and over again until done, or a larger list must be calculated. Thus f=4 and g=6 gives a list sufficiently large to calculate all of the one hundred highly composite numbers given by Ramanujan. On a PowerMac 7100/66, using the Mathematica program language, the calculation of the first 100 takes only about 10 s. The complete listing of the algorithm is given in Appendix II.

Properties of the calculated highly composite numbers

The final results of the calculation of 1000 terms in the series of highly complex numbers is shown in Fig. 1. The complete table is too long to show here, but we may note briefly that the l000th highly composite number is found to have 76 digits: it is

50739595324912050170305529996630230464563024879813409718878962795046826080000,

and it has 109586090557440 divisors. This can be compared to the average number of divisors up to this number, which is approximately ln[hc(1000)) = 177.

The powers of the successive primes of this number is also interesting. It can be written in a compact form:

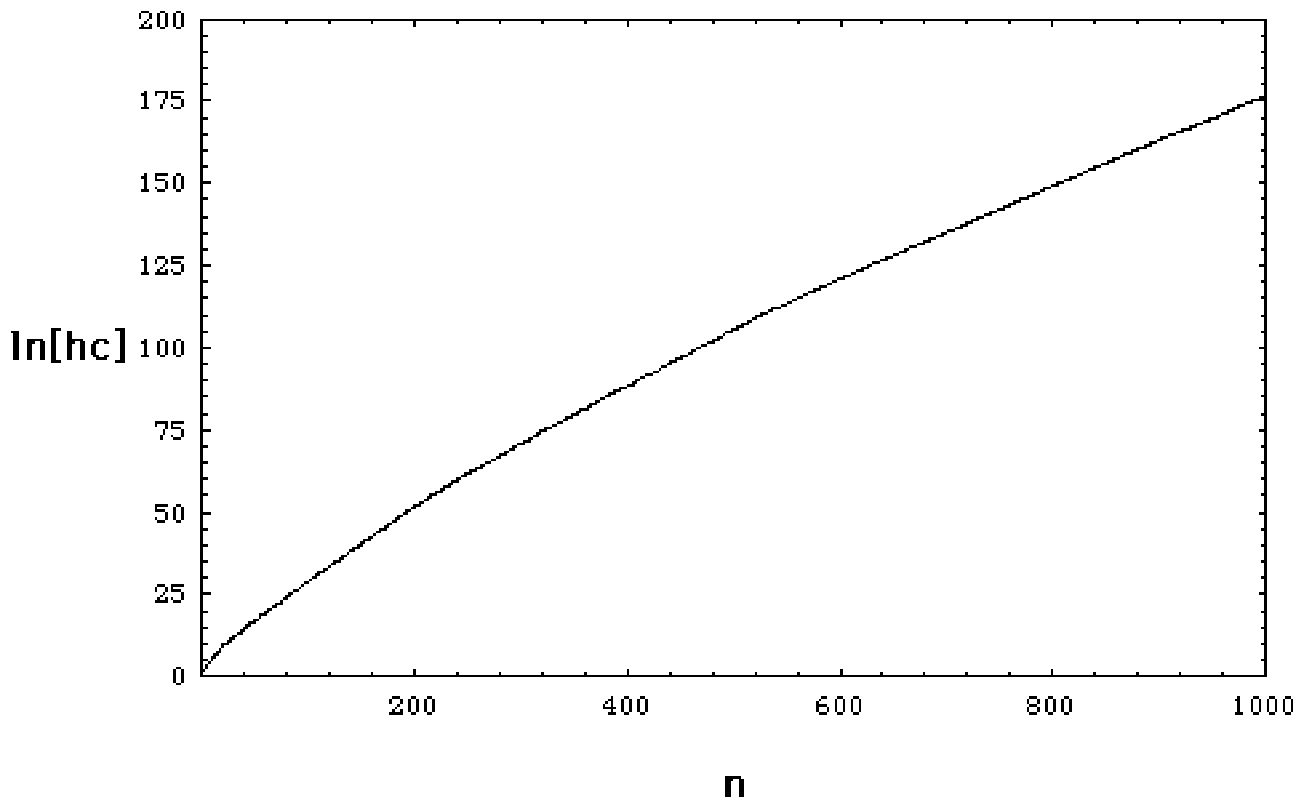
86432222111111111111111111111111111111;

That is, $2^8\ 3^6\ 5^4\ 7^3$ .... 157 163.

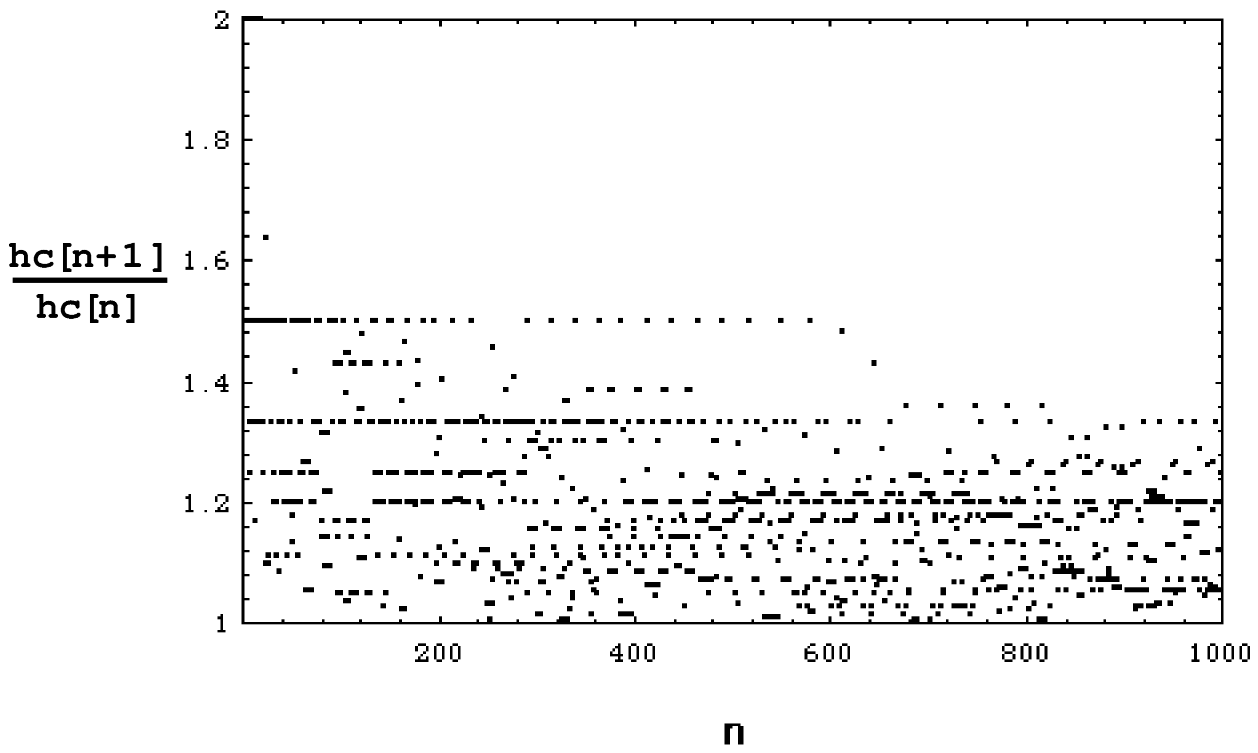
Examination of the numbers in the entire range computed shows that the first three exponents (of 2, 3 and 5) in all of the numbers larger than the 517th one are strictly decreasing. Ramanujan showed that in very large highly composite numbers, we would have $a_2 > a_3 > a_5 \ldots > a_\lambda$ (strictly decreasing exponents) when $\ln(p) > 8\lambda^2 (\ln(\lambda))^3$. For $\lambda = 5,$ this would make p huge. Evidently the strictly decreasing exponents occurs far earlier than predicted by this inequality.

We can make use of this fact to decrease the size of the list of v's required for still larger numbers, and speed up the algorithm too.

The calculation of the 1000 highly composite numbers took only about 2000 seconds on the PowerMac, but took a fair amount of memory--24 M of RAM with 45 M of virtual memory. The list of v's required for the last 100 terms had over 48,000 entries.



Although the hc(n) appear on this scale to be a smooth function of n, the ratio of successive numbers, shown in Fig. 2 illustrates the variation more clearly. The ratios, of course generally decrease with increasing n, as may be expected from the decreasing slope in Fig. 1.



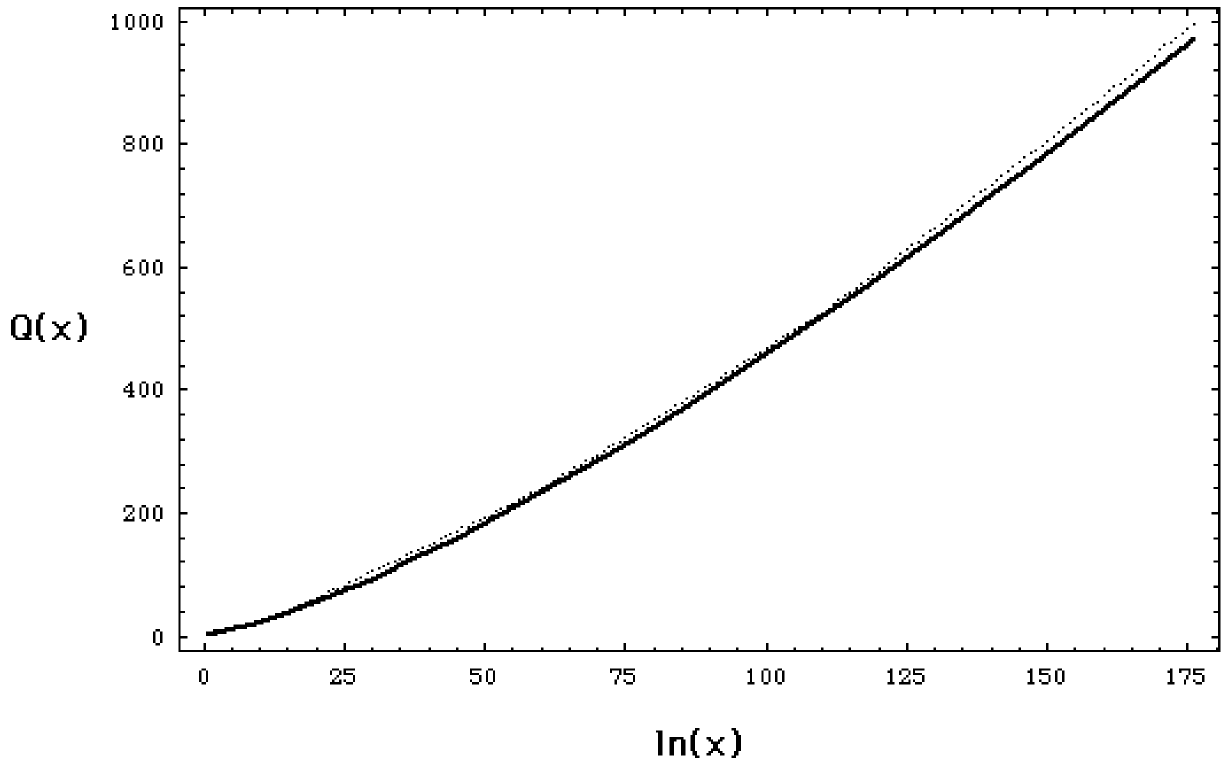
The density of the highly composite numbers is also of some interest. Here we define the number of highly composite numbers less than x to be Q[x]. For example, for z=100, Q[x]=8, and for z=1000, Q[x] = 14. Fig.1 shows the density as a function of the natural logarithm of x as discrete points. For comparison, the solid line shows the function

$$Q(x) = \ln(x)^{1+c},$$

which was first shown by Erdos to give a lower bound on Q(x). Here c is an undetermined constant greater than zero. The solid line is drawn with c= 1/3.

Summary

We have shown an algorithm to quickly calculate highly composite numbers and have used it to extend the list of known highly composite numbers from the 100, previously published, up to 1000.


Q(x)
ln(x)
0
200
400
600
800
1000
0
25
50
75
100
125
150
175

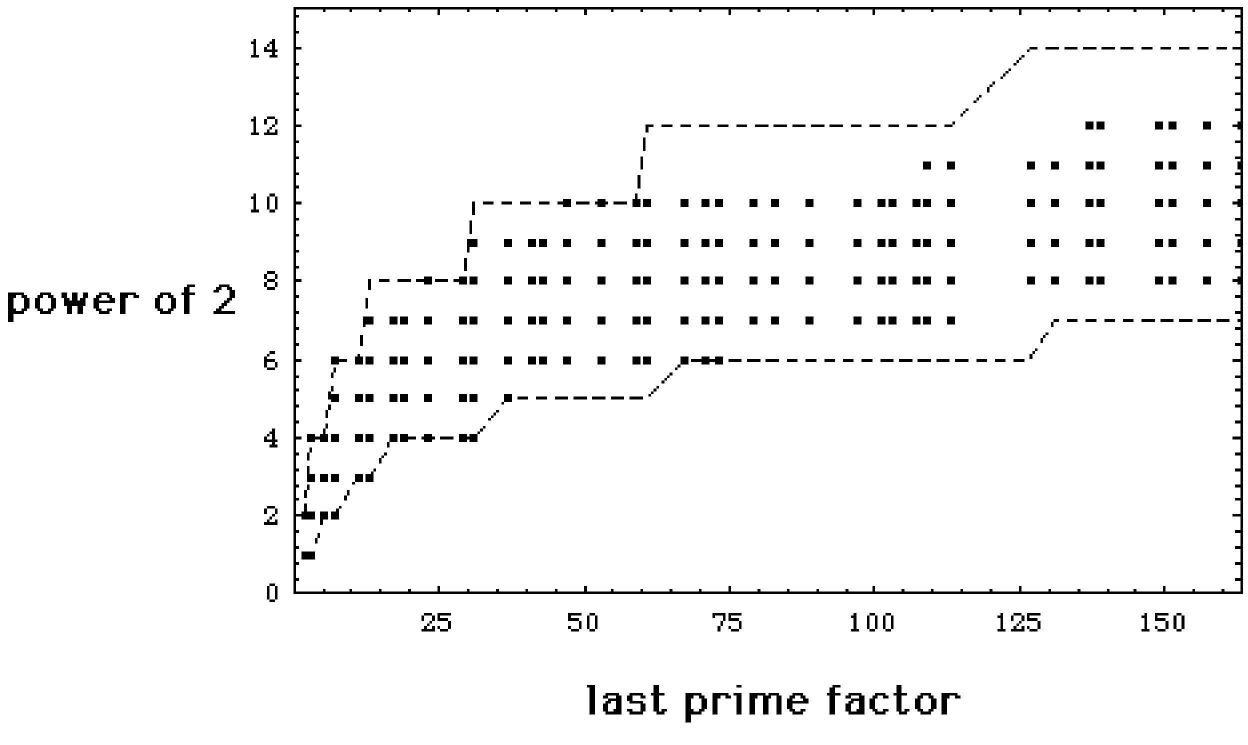
Appendix I.

Correct bounds f and g

The values of f and g to be used for a larger range of desired highly composite numbers can determined as follows. Ramanujan has shown that there is a connection between the power of two, a2 and the largest prime factor, p in a highly composite number:

$$\left[\frac{\log(p)}{\log(2)}\right] \le a_2 \le 2\left[\frac{\log(p_+)}{\log(2)}\right],$$

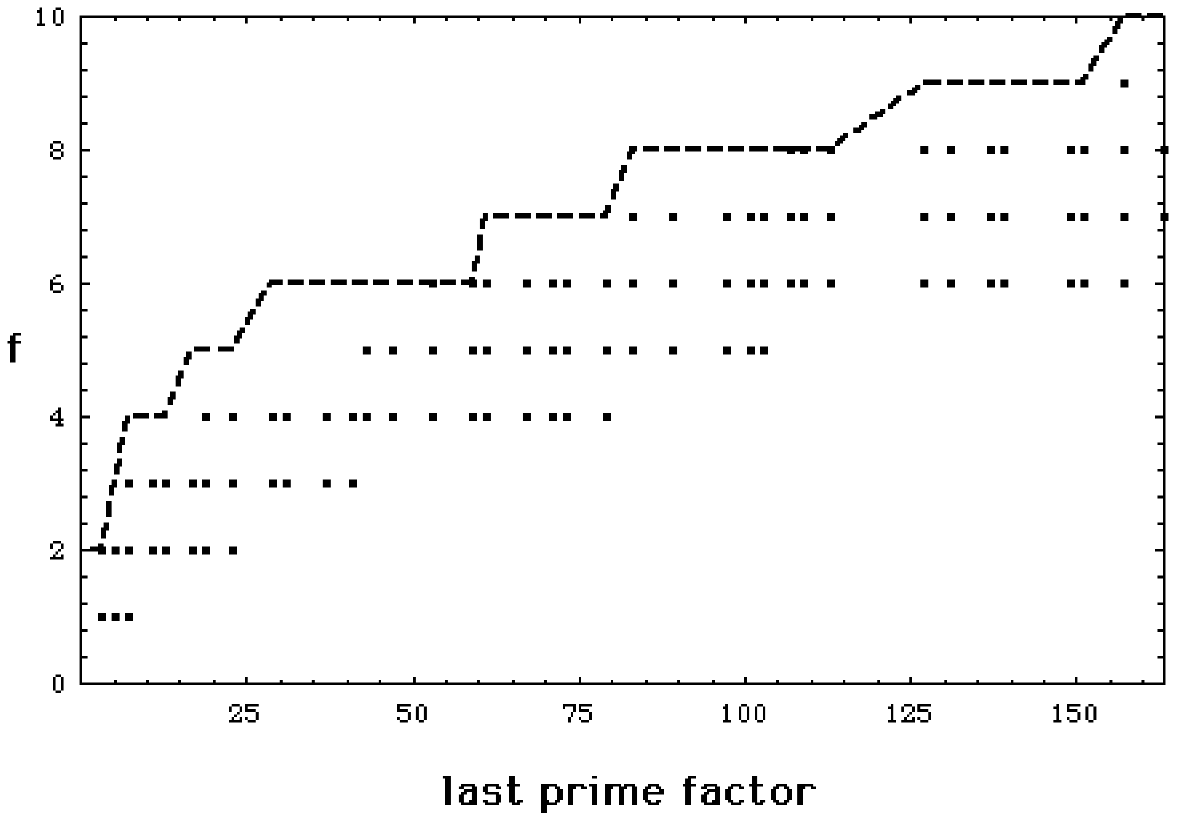
where [] stands for the integral part. Thus, for our example of the 85th highly composite number, p=19 and pp=23 gives a2 between 4 and 8, compared to the actual value of 6. When p=23 and pp=29 the formula also gives a2 between 4 and 8. Therefore g = 8 is good enough to certainly get the next higher highly composite number.



The number, f, of factors in v that is large enough to cover all of the possibilities involves a somewhat more obscure function.

$$\Lambda(\lambda, p) = [1/(2^{\log\lambda/\log p} - 1)]$$

Here λ is a prime less than p, and, as before, p is the largest prime factor of hc(n). For values of λ greater than some critical value Λ equals unity. The critical value of l therefore determines the size of f, the number of prime exponents that is not one. This is shown in fig. 1 as dashed line. The number actually found in the sequence of the first 1000 highly composite numbers is shown in Fig. 1.



References

1. "Collected Papers of S. Ramanujan", Ed. by G.H. Hardy, PV. Seshu Aiyar and B. M. Wilson, Chelsea Publishing Co. , NY, NY.
2. P. Erdos, J. London Math. Soc. 19, 130-133 (1944).

java.math

Class BigInteger

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.math.BigInteger
```

All Implemented Interfaces:

Comparable, Serializable

---

public class **BigInteger**
extends Number
implements Comparable

Immutable arbitrary-precision integers. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

Semantics of arithmetic operations exactly mimic those of Java's integer arithmetic operators, as defined in *The Java Language Specification*. For example, division by zero throws an `ArithmeticException`, and division of a negative by a positive yields a negative (or zero) remainder. All of the details in the Spec concerning overflow are ignored, as BigIntegers are made as large as necessary to accommodate the results of an operation.

Semantics of shift operations extend those of Java's shift operators to allow for negative shift distances. A right-shift with a negative shift distance results in a left shift, and vice-versa. The unsigned right shift operator (>>>) is omitted, as this operation makes little sense in combination with the "infinite word size" abstraction provided by this class.

Semantics of bitwise logical operations exactly mimic those of Java's bitwise integer operators. The binary operators (`and`, `or`, `xor`) implicitly perform sign extension on the shorter of the two operands prior to performing the operation.

Comparison operations perform signed integer comparisons, analogous to those performed by Java's relational and equality operators.

Modular arithmetic operations are provided to compute residues, perform exponentiation, and compute multiplicative inverses. These methods always return a non-negative result, between `0` and `(modulus - 1)`, inclusive.

Bit operations operate on a single bit of the two's-complement representation of their operand. If necessary, the operand is sign- extended so that it contains the designated bit. None of the single-bit operations can produce a BigInteger with a different sign from the BigInteger being operated on, as they affect only a single bit, and the "infinite word size" abstraction provided by this class ensures that there are infinitely many "virtual sign bits" preceding each BigInteger.

For the sake of brevity and clarity, pseudo-code is used throughout the descriptions of BigInteger methods. The pseudo-code expression `(i + j)` is shorthand for "a BigInteger whose value is that of the BigInteger `i` plus that of the BigInteger `j`." The pseudo-code expression `(i == j)` is shorthand for "`true` if and only if the BigInteger `i` represents the same value as the the BigInteger `j`." Other pseudo-code expressions are interpreted similarly.

All methods and constructors in this class throw `NullPointerException` when passed a null object reference for any input parameter.

Since:
JDK1.1
See Also:
`BigDecimal`, Serialized Form

Field Summary

`static BigInteger`	**`ONE`** The BigInteger constant one.
`static BigInteger`	**`ZERO`** The BigInteger constant zero.

Constructor Summary

`BigInteger``(byte[] val)` Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger.
`BigInteger``(int signum, byte[] magnitude)` Translates the sign-magnitude representation of a BigInteger into a BigInteger.
`BigInteger``(int bitLength, int certainty, Random rnd)` Constructs a randomly generated positive BigInteger that is probably prime, with the specified bitLength.
`BigInteger``(int numBits, Random rnd)` Constructs a randomly generated BigInteger, uniformly distributed over the range `0` to $(2^{numBits} - 1)$, inclusive.
`BigInteger``(String val)` Translates the decimal String representation of a BigInteger into a BigInteger.
`BigInteger``(String val, int radix)` Translates the String representation of a BigInteger in the specified radix into a BigInteger.

Method Summary

`BigInteger`	**`abs()`** Returns a BigInteger whose value is the absolute value of this BigInteger.
`BigInteger`	**`add(BigInteger val)`** Returns a BigInteger whose value is `(this + val)`.
`BigInteger`	**`and(BigInteger val)`** Returns a BigInteger whose value is `(this & val)`.
`BigInteger`	**`andNot(BigInteger val)`** Returns a BigInteger whose value is `(this & ~val)`.
`int`	**`bitCount()`** Returns the number of bits in the two's complement representation of this BigInteger that differ from its sign bit.
`int`	**`bitLength()`** Returns the number of bits in the minimal two's-complement representation of this BigInteger, *excluding* a sign bit.
`BigInteger`	**`clearBit(int n)`** Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit cleared.
`int`	**`compareTo(BigInteger val)`** Compares this BigInteger with the specified BigInteger.
`int`	**`compareTo(Object o)`** Compares this BigInteger with the specified Object.
`BigInteger`	**`divide(BigInteger val)`** Returns a BigInteger whose value is `(this / val)`.
`BigInteger[]`	**`divideAndRemainder(BigInteger val)`** Returns an array of two BigIntegers containing `(this / val)` followed by `(this % val)`.
`double`	**`doubleValue()`** Converts this BigInteger to a `double`.
`boolean`	**`equals(Object x)`** Compares this BigInteger with the specified Object for equality.
`BigInteger`	**`flipBit(int n)`** Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped.
`float`	**`floatValue()`** Converts this BigInteger to a `float`.
`BigInteger`	**`gcd(BigInteger val)`** Returns a BigInteger whose value is the greatest common divisor of `abs(this)` and `abs(val)`.
`int`	**`getLowestSetBit()`** Returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost one bit).

`int`	**`hashCode()`** Returns the hash code for this BigInteger.
`int`	**`intValue()`** Converts this BigInteger to an `int`.
`boolean`	**`isProbablePrime(int certainty)`** Returns `true` if this BigInteger is probably prime, `false` if it's definitely composite.
`long`	**`longValue()`** Converts this BigInteger to a `long`.
`BigInteger`	**`max(BigInteger val)`** Returns the maximum of this BigInteger and `val`.
`BigInteger`	**`min(BigInteger val)`** Returns the minimum of this BigInteger and `val`.
`BigInteger`	**`mod(BigInteger m)`** Returns a BigInteger whose value is `(this mod m)`.
`BigInteger`	**`modInverse(BigInteger m)`** Returns a BigInteger whose value is $(\mathtt{this}^{-1} \bmod m)$.
`BigInteger`	**`modPow(BigInteger exponent, BigInteger m)`** Returns a BigInteger whose value is $(\mathtt{this}^{\mathtt{exponent}} \bmod m)$.
`BigInteger`	**`multiply(BigInteger val)`** Returns a BigInteger whose value is `(this * val)`.
`BigInteger`	**`negate()`** Returns a BigInteger whose value is `(-this)`.
`BigInteger`	**`not()`** Returns a BigInteger whose value is `(~this)`.
`BigInteger`	**`or(BigInteger val)`** Returns a BigInteger whose value is `(this | val)`.
`BigInteger`	**`pow(int exponent)`** Returns a BigInteger whose value is $(\mathtt{this}^{\mathtt{exponent}})$.
`static BigInteger`	**`probablePrime(int bitLength, Random rnd)`** Returns a positive BigInteger that is probably prime, with the specified bitLength.
`BigInteger`	**`remainder(BigInteger val)`** Returns a BigInteger whose value is `(this % val)`.
`BigInteger`	**`setBit(int n)`** Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set.
`BigInteger`	**`shiftLeft(int n)`** Returns a BigInteger whose value is `(this << n)`.
`BigInteger`	**`shiftRight(int n)`** Returns a BigInteger whose value is `(this >> n)`.
`int`	**`signum()`** Returns the signum function of this BigInteger.

`BigInteger`	**`subtract`**`(BigInteger val)` Returns a BigInteger whose value is `(this - val)`.
`boolean`	**`testBit`**`(int n)` Returns `true` if and only if the designated bit is set.
`byte[]`	**`toByteArray`**`()` Returns a byte array containing the two's-complement representation of this BigInteger.
`String`	**`toString`**`()` Returns the decimal String representation of this BigInteger.
`String`	**`toString`**`(int radix)` Returns the String representation of this BigInteger in the given radix.
`static BigInteger`	**`valueOf`**`(long val)` Returns a BigInteger whose value is equal to that of the specified `long`.
`BigInteger`	**`xor`**`(BigInteger val)` Returns a BigInteger whose value is `(this ^ val)`.

Methods inherited from class java.lang.Number
`byteValue`, `shortValue`

Methods inherited from class java.lang.Object
`clone`, `finalize`, `getClass`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Field Detail

ZERO

`public static final BigInteger` **`ZERO`**

The BigInteger constant zero.

Since:
1.2

ONE

`public static final BigInteger` **`ONE`**

The BigInteger constant one.

Since:
1.2

Constructor Detail

BigInteger

```
public BigInteger(byte[] val)
```

Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger. The input array is assumed to be in *big-endian* byte-order: the most significant byte is in the zeroth element.

Parameters:

`val` - big-endian two's-complement binary representation of BigInteger.

Throws:

`NumberFormatException` - `val` is zero bytes long.

BigInteger

```
public BigInteger(int signum,
                  byte[] magnitude)
```

Translates the sign-magnitude representation of a BigInteger into a BigInteger. The sign is represented as an integer signum value: -1 for negative, 0 for zero, or 1 for positive. The magnitude is a byte array in *big-endian* byte-order: the most significant byte is in the zeroth element. A zero-length magnitude array is permissible, and will result in in a BigInteger value of 0, whether signum is -1, 0 or 1.

Parameters:

`signum` - signum of the number (-1 for negative, 0 for zero, 1 for positive).

`magnitude` - big-endian binary representation of the magnitude of the number.

Throws:

`NumberFormatException` - `signum` is not one of the three legal values (-1, 0, and 1), or `signum` is 0 and `magnitude` contains one or more non-zero bytes.

BigInteger

```
public BigInteger(String val,
                  int radix)
```

Translates the String representation of a BigInteger in the specified radix into a BigInteger. The String representation consists of an optional minus sign followed by a sequence of one or more digits in the specified radix. The character-to-digit mapping is provided by `Character.digit`. The String may not contain any extraneous characters (whitespace, for example).

Parameters:

`val` - String representation of BigInteger.

`radix` - radix to be used in interpreting `val`.

Throws:

`NumberFormatException` - `val` is not a valid representation of a BigInteger in the specified radix, or `radix` is outside the range from `Character.MIN_RADIX` to `Character.MAX_RADIX`, inclusive.

See Also:

`Character.digit(char, int)`

BigInteger

```
public BigInteger(String val)
```

Translates the decimal String representation of a BigInteger into a BigInteger. The String representation consists of an optional minus sign followed by a sequence of one or more decimal digits. The character-to-digit mapping is provided by `Character.digit`. The String may not contain any extraneous characters (whitespace, for example).

Parameters:

`val` - decimal String representation of BigInteger.

Throws:

`NumberFormatException` - `val` is not a valid representation of a BigInteger.

See Also:

`Character.digit(char, int)`

BigInteger

```
public BigInteger(int numBits,
                  Random rnd)
```

Constructs a randomly generated BigInteger, uniformly distributed over the range `0` to $(2^{numBits} - 1)$, inclusive. The uniformity of the distribution assumes that a fair source of random bits is provided in `rnd`. Note that this constructor always constructs a non-negative BigInteger.

Parameters:

`numBits` - maximum bitLength of the new BigInteger.

`rnd` - source of randomness to be used in computing the new BigInteger.

Throws:

`IllegalArgumentException` - `numBits` is negative.

See Also:

`bitLength()`

BigInteger

```
public BigInteger(int bitLength,
                  int certainty,
                  Random rnd)
```

Constructs a randomly generated positive BigInteger that is probably prime, with the specified bitLength.

It is recommended that the `probablePrime` method be used in preference to this constructor unless there is a compelling need to specify a certainty.

Parameters:
`bitLength` - bitLength of the returned BigInteger.
`certainty` - a measure of the uncertainty that the caller is willing to tolerate. The probability that the new BigInteger represents a prime number will exceed $(1 - 1/2^{certainty})$. The execution time of this constructor is proportional to the value of this parameter.
`rnd` - source of random bits used to select candidates to be tested for primality.
Throws:
`ArithmeticException` - `bitLength < 2`.
See Also:
`bitLength()`

Method Detail

probablePrime

```
public static BigInteger probablePrime(int bitLength,
                                       Random rnd)
```

Returns a positive BigInteger that is probably prime, with the specified bitLength. The probability that a BigInteger returned by this method is composite does not exceed 2^{-100}.

Parameters:
`bitLength` - bitLength of the returned BigInteger.
`rnd` - source of random bits used to select candidates to be tested for primality.
Returns:
a BigInteger of `bitLength` bits that is probably prime
Throws:
`ArithmeticException` - `bitLength < 2`.
See Also:
`bitLength()`

valueOf

```
public static BigInteger valueOf(long val)
```

Returns a BigInteger whose value is equal to that of the specified `long`. This "static factory method" is provided in preference to a (`long`) constructor because it allows for reuse of frequently used BigIntegers.

Parameters:
`val` - value of the BigInteger to return.
Returns:
a BigInteger with the specified value.

---

add

```
public BigInteger add(BigInteger val)
```

Returns a BigInteger whose value is `(this + val)`.

Parameters:
`val` - value to be added to this BigInteger.
Returns:
`this + val`

---

subtract

```
public BigInteger subtract(BigInteger val)
```

Returns a BigInteger whose value is `(this - val)`.

Parameters:
`val` - value to be subtracted from this BigInteger.
Returns:
`this - val`

---

multiply

```
public BigInteger multiply(BigInteger val)
```

Returns a BigInteger whose value is `(this * val)`.

Parameters:
`val` - value to be multiplied by this BigInteger.
Returns:
`this * val`

---

divide

```
public BigInteger divide(BigInteger val)
```

Returns a BigInteger whose value is `(this / val)`.

Parameters:
`val` - value by which this BigInteger is to be divided.
Returns:
`this / val`
Throws:
`ArithmeticException` - `val==0`

---

divideAndRemainder

```
public BigInteger[] divideAndRemainder(BigInteger val)
```

Returns an array of two BigIntegers containing `(this / val)` followed by `(this % val)`.

Parameters:
`val` - value by which this BigInteger is to be divided, and the remainder computed.
Returns:
an array of two BigIntegers: the quotient `(this / val)` is the initial element, and the remainder `(this % val)` is the final element.
Throws:
`ArithmeticException` - `val==0`

---

remainder

```
public BigInteger remainder(BigInteger val)
```

Returns a BigInteger whose value is `(this % val)`.

Parameters:
`val` - value by which this BigInteger is to be divided, and the remainder computed.
Returns:
`this % val`
Throws:
`ArithmeticException` - `val==0`

---

pow

```
public BigInteger pow(int exponent)
```

Returns a BigInteger whose value is $(\mathtt{this}^{\mathtt{exponent}})$. Note that `exponent` is an integer rather than a BigInteger.

Parameters:

`exponent` - exponent to which this BigInteger is to be raised.

Returns:

`this`$^{\texttt{exponent}}$

Throws:

`ArithmeticException` - `exponent` is negative. (This would cause the operation to yield a non-integer value.)

---

gcd

```
public BigInteger gcd(BigInteger val)
```

Returns a BigInteger whose value is the greatest common divisor of `abs(this)` and `abs(val)`. Returns 0 if `this==0 && val==0`.

Parameters:

`val` - value with with the GCD is to be computed.

Returns:

`GCD(abs(this), abs(val))`

---

abs

```
public BigInteger abs()
```

Returns a BigInteger whose value is the absolute value of this BigInteger.

Returns:

`abs(this)`

---

negate

```
public BigInteger negate()
```

Returns a BigInteger whose value is `(-this)`.

Returns:

`-this`

---

signum

```
public int signum()
```

Returns the signum function of this BigInteger.

Returns:
-1, 0 or 1 as the value of this BigInteger is negative, zero or positive.

---

mod

```
public BigInteger mod(BigInteger m)
```

Returns a BigInteger whose value is `(this mod m`). This method differs from `remainder` in that it always returns a *non-negative* BigInteger.

Parameters:
`m` - the modulus.

Returns:
`this mod m`

Throws:
`ArithmeticException` - `m <= 0`

See Also:
`remainder(java.math.BigInteger)`

---

modPow

```
public BigInteger modPow(BigInteger exponent,
                         BigInteger m)
```

Returns a BigInteger whose value is `(this`$^{\texttt{exponent}}$ `mod m)`. (Unlike `pow`, this method permits negative exponents.)

Parameters:
`exponent` - the exponent.
`m` - the modulus.

Returns:
`this`$^{\texttt{exponent}}$ `mod m`

Throws:
`ArithmeticException` - `m <= 0`

See Also:
`modInverse(java.math.BigInteger)`

---

modInverse

```
public BigInteger modInverse(BigInteger m)
```

Returns a BigInteger whose value is `(this`$^{-1}$ `mod m)`.

Parameters:

`m` - the modulus.

Returns:

`this`$^{-1}$ `mod m`.

Throws:

`ArithmeticException` - `m <= 0`, or this BigInteger has no multiplicative inverse mod m (that is, this BigInteger is not *relatively prime* to m).

---

shiftLeft

```
public BigInteger shiftLeft(int n)
```

Returns a BigInteger whose value is `(this << n)`. The shift distance, `n`, may be negative, in which case this method performs a right shift. (Computes `floor(this * 2`^{n}`)`.)

Parameters:

`n` - shift distance, in bits.

Returns:

`this << n`

See Also:

`shiftRight(int)`

---

shiftRight

```
public BigInteger shiftRight(int n)
```

Returns a BigInteger whose value is `(this >> n)`. Sign extension is performed. The shift distance, `n`, may be negative, in which case this method performs a left shift. (Computes `floor(this / 2`^{n}`)`.)

Parameters:

`n` - shift distance, in bits.

Returns:

`this >> n`

See Also:

`shiftLeft(int)`

---

and

```
public BigInteger and(BigInteger val)
```

Returns a BigInteger whose value is `(this & val)`. (This method returns a negative BigInteger if and only if this and val are both negative.)

Parameters:

`val` - value to be AND'ed with this BigInteger.

Returns:

`this & val`

---

or

```
public BigInteger or(BigInteger val)
```

Returns a BigInteger whose value is `(this | val)`. (This method returns a negative BigInteger if and only if either this or val is negative.)

Parameters:

`val` - value to be OR'ed with this BigInteger.

Returns:

`this | val`

---

xor

```
public BigInteger xor(BigInteger val)
```

Returns a BigInteger whose value is `(this ^ val)`. (This method returns a negative BigInteger if and only if exactly one of this and val are negative.)

Parameters:

`val` - value to be XOR'ed with this BigInteger.

Returns:

`this ^ val`

---

not

```
public BigInteger not()
```

Returns a BigInteger whose value is `(~this)`. (This method returns a negative value if and only if this BigInteger is non-negative.)

Returns:

`~this`

---

andNot

```
public BigInteger andNot(BigInteger val)
```

Returns a BigInteger whose value is `(this & ~val)`. This method, which is equivalent to `and(val.not())`, is provided as a convenience for masking operations. (This method returns a negative BigInteger if and only if `this` is negative and `val` is positive.)

Parameters:
`val` - value to be complemented and AND'ed with this BigInteger.
Returns:
`this & ~val`

---

testBit

`public boolean` **`testBit`**`(int n)`

Returns `true` if and only if the designated bit is set. (Computes `((this & (1<<n)) != 0)`.)

Parameters:
`n` - index of bit to test.
Returns:
`true` if and only if the designated bit is set.
Throws:
`ArithmeticException` - `n` is negative.

---

setBit

`public BigInteger` **`setBit`**`(int n)`

Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set. (Computes `(this | (1<<n))`.)

Parameters:
`n` - index of bit to set.
Returns:
`this | (1<<n)`
Throws:
`ArithmeticException` - `n` is negative.

---

clearBit

`public BigInteger` **`clearBit`**`(int n)`

Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit cleared. (Computes `(this & ~(1<<n))`.)

Parameters:

`n` - index of bit to clear.

Returns:

`this & ~(1<<n)`

Throws:

`ArithmeticException` - `n` is negative.

---

flipBit

```
public BigInteger flipBit(int n)
```

Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped. (Computes `(this ^ (1<<n))`.)

Parameters:

`n` - index of bit to flip.

Returns:

`this ^ (1<<n)`

Throws:

`ArithmeticException` - `n` is negative.

---

getLowestSetBit

```
public int getLowestSetBit()
```

Returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost one bit). Returns -1 if this BigInteger contains no one bits. (Computes (this==0? -1 : $\log_2$(this & -this)).)

Returns:

index of the rightmost one bit in this BigInteger.

---

bitLength

```
public int bitLength()
```

Returns the number of bits in the minimal two's-complement representation of this BigInteger, *excluding* a sign bit. For positive BigIntegers, this is equivalent to the number of bits in the ordinary binary representation. (Computes (ceil($\log_2$(this < 0 ? -this : this+1))).)

Returns:

number of bits in the minimal two's-complement representation of this BigInteger, *excluding* a sign bit.

---

bitCount

`public int **bitCount**()`

Returns the number of bits in the two's complement representation of this BigInteger that differ from its sign bit. This method is useful when implementing bit-vector style sets atop BigIntegers.

Returns:
number of bits in the two's complement representation of this BigInteger that differ from its sign bit.

---

isProbablePrime

`public boolean **isProbablePrime**(int certainty)`

Returns `true` if this BigInteger is probably prime, `false` if it's definitely composite. If `certainty` is `<= 0`, `true` is returned.

Parameters:
`certainty` - a measure of the uncertainty that the caller is willing to tolerate: if the call returns `true` the probability that this BigInteger is prime exceeds $(1 - 1/2^{certainty})$. The execution time of this method is proportional to the value of this parameter.
Returns:
`true` if this BigInteger is probably prime, `false` if it's definitely composite.

---

compareTo

`public int **compareTo**(BigInteger val)`

Compares this BigInteger with the specified BigInteger. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: `(x.compareTo(y)` <*op*> `0)`, where <*op*> is one of the six comparison operators.

Parameters:
`val` - BigInteger to which this BigInteger is to be compared.
Returns:
-1, 0 or 1 as this BigInteger is numerically less than, equal to, or greater than `val`.

---

compareTo

`public int **compareTo**(Object o)`

Compares this BigInteger with the specified Object. If the Object is a BigInteger, this method behaves like `compareTo(BigInteger)`. Otherwise, it throws a `ClassCastException` (as BigIntegers are comparable only to other BigIntegers).

Specified by:
: `compareTo` in interface `Comparable`

Parameters:
: `o` - Object to which this BigInteger is to be compared.

Returns:
: a negative number, zero, or a positive number as this BigInteger is numerically less than, equal to, or greater than `o`, which must be a BigInteger.

Throws:
: `ClassCastException` - `o` is not a BigInteger.

Since:
: 1.2

See Also:
: `compareTo(java.math.BigInteger)`, `Comparable`

---

equals

```
public boolean equals(Object x)
```

Compares this BigInteger with the specified Object for equality.

Overrides:
: `equals` in class `Object`

Parameters:
: `x` - Object to which this BigInteger is to be compared.

Returns:
: `true` if and only if the specified Object is a BigInteger whose value is numerically equal to this BigInteger.

See Also:
: `Object.hashCode()`, `Hashtable`

---

min

```
public BigInteger min(BigInteger val)
```

Returns the minimum of this BigInteger and `val`.

Parameters:
: `val` - value with with the minimum is to be computed.

Returns:
: the BigInteger whose value is the lesser of this BigInteger and `val`. If they are equal, either may be returned.

max

`public BigInteger max(BigInteger val)`

Returns the maximum of this BigInteger and `val`.

Parameters:
`val` - value with with the maximum is to be computed.
Returns:
the BigInteger whose value is the greater of this and `val`. If they are equal, either may be returned.

hashCode

`public int hashCode()`

Returns the hash code for this BigInteger.

Overrides:
`hashCode` in class `Object`
Returns:
hash code for this BigInteger.
See Also:
`Object.equals(java.lang.Object)`, `Hashtable`

toString

`public String toString(int radix)`

Returns the String representation of this BigInteger in the given radix. If the radix is outside the range from `Character.MIN_RADIX` to `Character.MAX_RADIX` inclusive, it will default to 10 (as is the case for `Integer.toString`). The digit-to-character mapping provided by `Character.forDigit` is used, and a minus sign is prepended if appropriate. (This representation is compatible with the `(String, int)` constructor.)

Parameters:
`radix` - radix of the String representation.
Returns:
String representation of this BigInteger in the given radix.
See Also:
`Integer.toString(int, int)`, `Character.forDigit(int, int)`, `BigInteger(java.lang.String, int)`

toString

`public String **toString**()`

Returns the decimal String representation of this BigInteger. The digit-to-character mapping provided by `Character.forDigit` is used, and a minus sign is prepended if appropriate. (This representation is compatible with the `(String)` constructor, and allows for String concatenation with Java's + operator.)

Overrides:
: `toString` in class `Object`

Returns:
: decimal String representation of this BigInteger.

See Also:
: `Character.forDigit(int, int)`, `BigInteger(java.lang.String)`

---

toByteArray

`public byte[] **toByteArray**()`

Returns a byte array containing the two's-complement representation of this BigInteger. The byte array will be in *big-endian* byte-order: the most significant byte is in the zeroth element. The array will contain the minimum number of bytes required to represent this BigInteger, including at least one sign bit, which is `(ceil((this.bitLength() + 1)/8))`. (This representation is compatible with the `(byte[])` constructor.)

Returns:
: a byte array containing the two's-complement representation of this BigInteger.

See Also:
: `BigInteger(byte[])`

---

intValue

`public int **intValue**()`

Converts this BigInteger to an `int`. This conversion is analogous to a *narrowing primitive conversion* from `long` to `int` as defined in the Java Language Specification: if this BigInteger is too big to fit in an `int`, only the low-order 32 bits are returned. Note that this conversion can lose information about the overall magnitude of the BigInteger value as well as return a result with the opposite sign.

Specified by:
: `intValue` in class `Number`

Returns:
: this BigInteger converted to an `int`.

---

longValue

```
public long longValue()
```

Converts this BigInteger to a `long`. This conversion is analogous to a *narrowing primitive conversion* from `long` to `int` as defined in the Java Language Specification: if this BigInteger is too big to fit in a `long`, only the low-order 64 bits are returned. Note that this conversion can lose information about the overall magnitude of the BigInteger value as well as return a result with the opposite sign.

Specified by:
`longValue` in class `Number`

Returns:
this BigInteger converted to a `long`.

---

floatValue

```
public float floatValue()
```

Converts this BigInteger to a `float`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` defined in the Java Language Specification: if this BigInteger has too great a magnitude to represent as a `float`, it will be converted to `Float.NEGATIVE_INFINITY` or `Float.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigInteger value.

Specified by:
`floatValue` in class `Number`

Returns:
this BigInteger converted to a `float`.

---

doubleValue

```
public double doubleValue()
```

Converts this BigInteger to a `double`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` defined in the Java Language Specification: if this BigInteger has too great a magnitude to represent as a `double`, it will be converted to `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigInteger value.

Specified by:
`doubleValue` in class `Number`

Returns:

this BigInteger converted to a `double`.

Submit a bug or feature

For further API reference and developer documentation, see Java 2 SDK SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

java.math

Class BigDecimal

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.math.BigDecimal
```

All Implemented Interfaces:

Comparable, Serializable

---

public class **BigDecimal**
extends Number
implements Comparable

Immutable, arbitrary-precision signed decimal numbers. A BigDecimal consists of an arbitrary precision integer *unscaled value* and a non-negative 32-bit integer *scale*, which represents the number of digits to the right of the decimal point. The number represented by the BigDecimal is $(\texttt{unscaledValue}/10^{\texttt{scale}})$. BigDecimal provides operations for basic arithmetic, scale manipulation, comparison, hashing, and format conversion.

The BigDecimal class gives its user complete control over rounding behavior, forcing the user to explicitly specify a rounding behavior for operations capable of discarding precision (`divide(BigDecimal, int)`, `divide(BigDecimal, int, int)`, and `setScale(int, int)`). Eight *rounding modes* are provided for this purpose.

Two types of operations are provided for manipulating the scale of a BigDecimal: scaling/rounding operations and decimal point motion operations. Scaling/rounding operations (`setScale`) return a BigDecimal whose value is approximately (or exactly) equal to that of the operand, but whose scale is the specified value; that is, they increase or decrease the precision of the number with minimal effect on its value. Decimal point motion operations (`movePointLeft(int)` and `movePointRight(int)`) return a BigDecimal created from the operand by moving the decimal point a specified distance in the specified direction; that is, they change a number's value without affecting its precision.

For the sake of brevity and clarity, pseudo-code is used throughout the descriptions of BigDecimal methods. The pseudo-code expression `(i + j)` is shorthand for "a BigDecimal whose value is that of the BigDecimal `i` plus that of the BigDecimal `j`." The pseudo-code expression `(i == j)` is shorthand for "`true` if and only if the BigDecimal `i` represents the same value as the the BigDecimal `j`." Other pseudo-code expressions are interpreted similarly.

Note: care should be exercised if BigDecimals are to be used as keys in a `SortedMap` or elements in a `SortedSet`, as BigDecimal's *natural ordering* is *inconsistent with equals*. See `Comparable`, `SortedMap` or `SortedSet` for more information.

All methods and constructors for this class throw `NullPointerException` when passed a null object reference for any input parameter.

See Also:

`BigInteger`, `SortedMap`, `SortedSet`, Serialized Form

Field Summary

`static int`	**`ROUND_CEILING`** Rounding mode to round towards positive infinity.
`static int`	**`ROUND_DOWN`** Rounding mode to round towards zero.
`static int`	**`ROUND_FLOOR`** Rounding mode to round towards negative infinity.
`static int`	**`ROUND_HALF_DOWN`** Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
`static int`	**`ROUND_HALF_EVEN`** Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor.
`static int`	**`ROUND_HALF_UP`** Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.
`static int`	**`ROUND_UNNECESSARY`** Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary.
`static int`	**`ROUND_UP`** Rounding mode to round away from zero.

Constructor Summary

`BigDecimal``(BigInteger val)` Translates a BigInteger into a BigDecimal.
`BigDecimal``(BigInteger unscaledVal, int scale)` Translates a BigInteger unscaled value and an `int` scale into a BigDecimal.
`BigDecimal``(double val)` Translates a `double` into a BigDecimal.
`BigDecimal``(String val)` Translates the String representation of a BigDecimal into a BigDecimal.

Method Summary

`BigDecimal`	**`abs`**`()` Returns a BigDecimal whose value is the absolute value of this BigDecimal, and whose scale is `this.scale()`.
`BigDecimal`	**`add`**`(BigDecimal val)` Returns a BigDecimal whose value is `(this + val)`, and whose scale is `max(this.scale(), val.scale())`.
`int`	**`compareTo`**`(BigDecimal val)` Compares this BigDecimal with the specified BigDecimal.
`int`	**`compareTo`**`(Object o)` Compares this BigDecimal with the specified Object.
`BigDecimal`	**`divide`**`(BigDecimal val, int roundingMode)` Returns a BigDecimal whose value is `(this / val)`, and whose scale is `this.scale()`.
`BigDecimal`	**`divide`**`(BigDecimal val, int scale, int roundingMode)` Returns a BigDecimal whose value is `(this / val)`, and whose scale is as specified.
`double`	**`doubleValue`**`()` Converts this BigDecimal to a `double`.
`boolean`	**`equals`**`(Object x)` Compares this BigDecimal with the specified Object for equality.
`float`	**`floatValue`**`()` Converts this BigDecimal to a `float`.
`int`	**`hashCode`**`()` Returns the hash code for this BigDecimal.
`int`	**`intValue`**`()` Converts this BigDecimal to an `int`.
`long`	**`longValue`**`()` Converts this BigDecimal to a `long`.
`BigDecimal`	**`max`**`(BigDecimal val)` Returns the maximum of this BigDecimal and `val`.
`BigDecimal`	**`min`**`(BigDecimal val)` Returns the minimum of this BigDecimal and `val`.
`BigDecimal`	**`movePointLeft`**`(int n)` Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the left.
`BigDecimal`	**`movePointRight`**`(int n)` Moves the decimal point the specified number of places to the right.
`BigDecimal`	**`multiply`**`(BigDecimal val)` Returns a BigDecimal whose value is `(this * val)`, and whose scale is `(this.scale() + val.scale())`.
`BigDecimal`	**`negate`**`()` Returns a BigDecimal whose value is `(-this)`, and whose scale is `this.scale()`.

`int`	**`scale`**`()` Returns the *scale* of this BigDecimal.
`BigDecimal`	**`setScale`**`(int scale)` Returns a BigDecimal whose scale is the specified value, and whose value is numerically equal to this BigDecimal's.
`BigDecimal`	**`setScale`**`(int scale, int roundingMode)` Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
`int`	**`signum`**`()` Returns the signum function of this BigDecimal.
`BigDecimal`	**`subtract`**`(BigDecimal val)` Returns a BigDecimal whose value is `(this - val)`, and whose scale is `max(this.scale(), val.scale())`.
`BigInteger`	**`toBigInteger`**`()` Converts this BigDecimal to a BigInteger.
`String`	**`toString`**`()` Returns the string representation of this BigDecimal.
`BigInteger`	**`unscaledValue`**`()` Returns a BigInteger whose value is the *unscaled value* of this BigDecimal.
`static BigDecimal`	**`valueOf`**`(long val)` Translates a `long` value into a BigDecimal with a scale of zero.
`static BigDecimal`	**`valueOf`**`(long unscaledVal, int scale)` Translates a `long` unscaled value and an `int` scale into a BigDecimal.

Methods inherited from class java.lang.Number
`byteValue, shortValue`

Methods inherited from class java.lang.Object
`clone, finalize, getClass, notify, notifyAll, wait, wait, wait`

Field Detail

ROUND_UP

```
public static final int ROUND_UP
```

Rounding mode to round away from zero. Always increments the digit prior to a non-zero discarded fraction. Note that this rounding mode never decreases the magnitude of the calculated value.

See Also:
Constant Field Values

ROUND_DOWN

```
public static final int ROUND_DOWN
```

Rounding mode to round towards zero. Never increments the digit prior to a discarded fraction (i.e., truncates). Note that this rounding mode never increases the magnitude of the calculated value.

See Also:
Constant Field Values

ROUND_CEILING

```
public static final int ROUND_CEILING
```

Rounding mode to round towards positive infinity. If the BigDecimal is positive, behaves as for `ROUND_UP`; if negative, behaves as for `ROUND_DOWN`. Note that this rounding mode never decreases the calculated value.

See Also:
Constant Field Values

ROUND_FLOOR

```
public static final int ROUND_FLOOR
```

Rounding mode to round towards negative infinity. If the BigDecimal is positive, behave as for `ROUND_DOWN`; if negative, behave as for `ROUND_UP`. Note that this rounding mode never increases the calculated value.

See Also:
Constant Field Values

ROUND_HALF_UP

```
public static final int ROUND_HALF_UP
```

Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up. Behaves as for `ROUND_UP` if the discarded fraction is >= .5; otherwise, behaves as for `ROUND_DOWN`. Note that this is the rounding mode that most of us were taught in grade school.

See Also:
Constant Field Values

ROUND_HALF_DOWN

`public static final int` **`ROUND_HALF_DOWN`**

Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down. Behaves as for `ROUND_UP` if the discarded fraction is > .5; otherwise, behaves as for `ROUND_DOWN`.

See Also:
Constant Field Values

ROUND_HALF_EVEN

`public static final int` **`ROUND_HALF_EVEN`**

Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor. Behaves as for ROUND_HALF_UP if the digit to the left of the discarded fraction is odd; behaves as for ROUND_HALF_DOWN if it's even. Note that this is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations.

See Also:
Constant Field Values

ROUND_UNNECESSARY

`public static final int` **`ROUND_UNNECESSARY`**

Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary. If this rounding mode is specified on an operation that yields an inexact result, an `ArithmeticException` is thrown.

See Also:
Constant Field Values

Constructor Detail

BigDecimal

`public` **`BigDecimal`**`(String val)`

Translates the String representation of a BigDecimal into a BigDecimal. The String representation consists of an optional sign, `'+'` (`'\u002B'`) or `'-'` (`'\u002D'`), followed by a sequence of zero or more decimal digits ("the integer"), optionally followed by a fraction, optionally followed by an exponent.

The fraction consists of of a decimal point followed by zero or more decimal digits. The string must contain at least one digit in either the integer or the fraction. The number formed by the sign, the integer and the fraction is referred to as the *significand*.

The exponent consists of the character `'e'` (`'\u0075'`) or `'E'` (`'\u0045'`) followed by one or more decimal digits. The value of the exponent must lie between -`Integer.MAX_VALUE` (`Integer.MIN_VALUE`+1) and `Integer.MAX_VALUE`, inclusive.

More formally, the strings this constructor accepts are described by the following grammar:

BigDecimalString:
  Sign$_{opt}$ Significand Exponent$_{opt}$

Sign:
  `+`
  `-`

Significand:
  IntegerPart `.` *FractionPart$_{opt}$*
  `.` *FractionPart*
  IntegerPart

IntegerPart:
  Digits

FractionPart:
  Digits

Exponent:
  ExponentIndicator SignedInteger

ExponentIndicator:
  `e`
  `E`

SignedInteger:
  Sign$_{opt}$ Digits

Digits:
  Digit
  Digits Digit

Digit:
  any character for which `Character.isDigit(char)` returns `true`, including 0, 1, 2 ...

The scale of the returned BigDecimal will be the number of digits in the fraction, or zero if the string contains no decimal point, subject to adjustment for any exponent: If the string contains an exponent, the exponent is subtracted from the scale. If the resulting scale is negative, the scale of the returned BigDecimal is zero and the unscaled value is multiplied by the appropriate power of ten so that, in every case, the resulting BigDecimal is equal to $significand \times 10^{exponent}$. (If in the future this specification is amended to permit negative scales, the final step of zeroing the scale and adjusting the unscaled value will be eliminated.)

The character-to-digit mapping is provided by `Character.digit(char, int)` set to convert to radix 10. The String may not contain any extraneous characters (whitespace, for example).

Note: For values other `float` and `double` NaN and ±Infinity, this constructor is compatible with the values returned by `Float.toString(float)` and `Double.toString(double)`. This is generally the preferred way to convert a `float` or `double` into a BigDecimal, as it doesn't suffer from the unpredictability of the `BigDecimal(double)` constructor.

Note: the optional leading plus sign and trailing exponent were added in release 1.3.

Parameters:
`val` - String representation of BigDecimal.

Throws:
`NumberFormatException` - `val` is not a valid representation of a BigDecimal.

BigDecimal

```
public BigDecimal(double val)
```

Translates a `double` into a BigDecimal. The scale of the BigDecimal is the smallest value such that $(10^{scale} * val)$ is an integer.

Note: the results of this constructor can be somewhat unpredictable. One might assume that `new BigDecimal(.1)` is exactly equal to .1, but it is actually equal to .1000000000000000055511151231257827021181583404541015625. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed *in* to the constructor is not exactly equal to .1, appearances notwithstanding.

The (String) constructor, on the other hand, is perfectly predictable: `new BigDecimal(".1")` is *exactly* equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.

Parameters:
`val` - `double` value to be converted to BigDecimal.

Throws:
`NumberFormatException` - `val` if `val` is infinite or NaN.

BigDecimal

```
public BigDecimal(BigInteger val)
```

Translates a BigInteger into a BigDecimal. The scale of the BigDecimal is zero.

Parameters:
`val` - BigInteger value to be converted to BigDecimal.

BigDecimal

```
public BigDecimal(BigInteger unscaledVal,
                  int scale)
```

Translates a BigInteger unscaled value and an `int` scale into a BigDecimal. The value of the BigDecimal is `(unscaledVal/10`$^{\texttt{scale}}$`)`.

Parameters:
`unscaledVal` - unscaled value of the BigDecimal.
`scale` - scale of the BigDecimal.
Throws:
`NumberFormatException` - scale is negative

Method Detail

valueOf

```
public static BigDecimal valueOf(long unscaledVal,
                                 int scale)
```

Translates a `long` unscaled value and an `int` scale into a BigDecimal. This "static factory method" is provided in preference to a (`long`, `int`) constructor because it allows for reuse of frequently used BigDecimals.

Parameters:
`unscaledVal` - unscaled value of the BigDecimal.
`scale` - scale of the BigDecimal.
Returns:
a BigDecimal whose value is `(unscaledVal/10`$^{\texttt{scale}}$`)`.

---

valueOf

```
public static BigDecimal valueOf(long val)
```

Translates a `long` value into a BigDecimal with a scale of zero. This "static factory method" is provided in preference to a (`long`) constructor because it allows for reuse of frequently used BigDecimals.

Parameters:
`val` - value of the BigDecimal.
Returns:
a BigDecimal whose value is `val`.

---

add

```
public BigDecimal add(BigDecimal val)
```

Returns a BigDecimal whose value is `(this + val)`, and whose scale is `max(this.scale(), val.scale())`.

Parameters:
`val` - value to be added to this BigDecimal.
Returns:
`this + val`

---

subtract

```
public BigDecimal subtract(BigDecimal val)
```

Returns a BigDecimal whose value is `(this - val)`, and whose scale is `max(this.scale(), val.scale())`.

Parameters:
`val` - value to be subtracted from this BigDecimal.
Returns:
`this - val`

---

multiply

```
public BigDecimal multiply(BigDecimal val)
```

Returns a BigDecimal whose value is `(this * val)`, and whose scale is `(this.scale() + val.scale())`.

Parameters:
`val` - value to be multiplied by this BigDecimal.
Returns:
`this * val`

---

divide

```
public BigDecimal divide(BigDecimal val,
                         int scale,
                         int roundingMode)
```

Returns a BigDecimal whose value is `(this / val)`, and whose scale is as specified. If rounding must be performed to generate a result with the specified scale, the specified rounding mode is applied.

Parameters:
`val` - value by which this BigDecimal is to be divided.
`scale` - scale of the BigDecimal quotient to be returned.
`roundingMode` - rounding mode to apply.

Returns:
`this / val`

Throws:
`ArithmeticException` - `val` is zero, `scale` is negative, or `roundingMode==ROUND_UNNECESSARY` and the specified scale is insufficient to represent the result of the division exactly.
`IllegalArgumentException` - `roundingMode` does not represent a valid rounding mode.

See Also:
`ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_UNNECESSARY`

---

divide

```
public BigDecimal divide(BigDecimal val,
                         int roundingMode)
```

Returns a BigDecimal whose value is `(this / val)`, and whose scale is `this.scale()`. If rounding must be performed to generate a result with the given scale, the specified rounding mode is applied.

Parameters:
`val` - value by which this BigDecimal is to be divided.
`roundingMode` - rounding mode to apply.

Returns:
`this / val`

Throws:
`ArithmeticException` - `val==0`, or `roundingMode==ROUND_UNNECESSARY` and `this.scale()` is insufficient to represent the result of the division exactly.
`IllegalArgumentException` - `roundingMode` does not represent a valid rounding mode.

See Also:
`ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_UNNECESSARY`

abs

`public BigDecimal abs()`

Returns a BigDecimal whose value is the absolute value of this BigDecimal, and whose scale is `this.scale()`.

Returns:
`abs(this)`

negate

`public BigDecimal negate()`

Returns a BigDecimal whose value is `(-this)`, and whose scale is `this.scale()`.

Returns:
`-this`

signum

`public int signum()`

Returns the signum function of this BigDecimal.

Returns:
-1, 0 or 1 as the value of this BigDecimal is negative, zero or positive.

scale

`public int scale()`

Returns the *scale* of this BigDecimal. (The scale is the number of digits to the right of the decimal point.)

Returns:
the scale of this BigDecimal.

unscaledValue

```
public BigInteger unscaledValue()
```

Returns a BigInteger whose value is the *unscaled value* of this BigDecimal. (Computes `(this * 10`$^{\texttt{this.scale()}}$`)`.)

Returns:
the unscaled value of this BigDecimal.

Since:
1.2

---

setScale

```
public BigDecimal setScale(int scale,
                           int roundingMode)
```

Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value. If the scale is reduced by the operation, the unscaled value must be divided (rather than multiplied), and the value may be changed; in this case, the specified rounding mode is applied to the division.

Note that since BigDecimal objects are immutable, calls of this method do *not* result in the original object being modified, contrary to the usual convention of having methods named `set`*`X`* mutate field *`X`*. Instead, `setScale` returns an object with the proper scale; the returned object may or may not be newly allocated.

Parameters:
`scale` - scale of the BigDecimal value to be returned.
`roundingMode` - The rounding mode to apply.

Returns:
a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.

Throws:
`ArithmeticException` - `scale` is negative, or `roundingMode==ROUND_UNNECESSARY` and the specified scaling operation would require rounding.
`IllegalArgumentException` - `roundingMode` does not represent a valid rounding mode.

See Also:
`ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_UNNECESSARY`

---

setScale

```
public BigDecimal setScale(int scale)
```

Returns a BigDecimal whose scale is the specified value, and whose value is numerically equal to this BigDecimal's. Throws an ArithmeticException if this is not possible. This call is typically used to increase the scale, in which case it is guaranteed that there exists a BigDecimal of the specified scale and the correct value. The call can also be used to reduce the scale if the caller knows that the BigDecimal has sufficiently many zeros at the end of its fractional part (i.e., factors of ten in its integer value) to allow for the rescaling without loss of precision.

This method returns the same result as the two argument version of setScale, but saves the caller the trouble of specifying a rounding mode in cases where it is irrelevant.

Note that since BigDecimal objects are immutable, calls of this method do *not* result in the original object being modified, contrary to the usual convention of having methods named `set`*`X`* mutate field *`X`*. Instead, `setScale` returns an object with the proper scale; the returned object may or may not be newly allocated.

Parameters:
`scale` - scale of the BigDecimal value to be returned.
Returns:
a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
Throws:
`ArithmeticException` - `scale` is negative, or the specified scaling operation would require rounding.
See Also:
`setScale(int, int)`

---

movePointLeft

`public BigDecimal movePointLeft(int n)`

Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the left. If n is non-negative, the call merely adds n to the scale. If n is negative, the call is equivalent to movePointRight(-n). (The BigDecimal returned by this call has value `(this * 10`$^{-n}$`)` and scale `max(this.scale()+n, 0)`.)

Parameters:
`n` - number of places to move the decimal point to the left.
Returns:
a BigDecimal which is equivalent to this one with the decimal point moved `n` places to the left.

---

movePointRight

`public BigDecimal movePointRight(int n)`

Moves the decimal point the specified number of places to the right. If this BigDecimal's scale is >=

`n`, the call merely subtracts `n` from the scale; otherwise, it sets the scale to zero, and multiplies the integer value by `10`$^{\texttt{(n - this.scale)}}$. If `n` is negative, the call is equivalent to `movePointLeft(-n)`. (The BigDecimal returned by this call has value `(this * 10`$^{\texttt{n}}$`)` and scale `max(this.scale()-n, 0)`.)

Parameters:
`n` - number of places to move the decimal point to the right.
Returns:
a BigDecimal which is equivalent to this one with the decimal point moved `n` places to the right.

---

compareTo

`public int **compareTo**(BigDecimal val)`

Compares this BigDecimal with the specified BigDecimal. Two BigDecimals that are equal in value but have a different scale (like 2.0 and 2.00) are considered equal by this method. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: `(x.compareTo(y)` <*op*> `0)`, where <*op*> is one of the six comparison operators.

Parameters:
`val` - BigDecimal to which this BigDecimal is to be compared.
Returns:
-1, 0 or 1 as this BigDecimal is numerically less than, equal to, or greater than `val`.

---

compareTo

`public int **compareTo**(Object o)`

Compares this BigDecimal with the specified Object. If the Object is a BigDecimal, this method behaves like `compareTo`. Otherwise, it throws a `ClassCastException` (as BigDecimals are comparable only to other BigDecimals).

Specified by:
`compareTo` in interface `Comparable`
Parameters:
`o` - Object to which this BigDecimal is to be compared.
Returns:
a negative number, zero, or a positive number as this BigDecimal is numerically less than, equal to, or greater than `o`, which must be a BigDecimal.
Throws:
`ClassCastException` - `o` is not a BigDecimal.
Since:
1.2

See Also:
`compareTo(java.math.BigDecimal)`, `Comparable`

---

equals

`public boolean **equals**(Object x)`

Compares this BigDecimal with the specified Object for equality. Unlike `compareTo`, this method considers two BigDecimals equal only if they are equal in value and scale (thus 2.0 is not equal to 2.00 when compared by this method).

Overrides:
`equals` in class `Object`

Parameters:
`x` - Object to which this BigDecimal is to be compared.

Returns:
`true` if and only if the specified Object is a BigDecimal whose value and scale are equal to this BigDecimal's.

See Also:
`compareTo(java.math.BigDecimal)`

---

min

`public BigDecimal **min**(BigDecimal val)`

Returns the minimum of this BigDecimal and `val`.

Parameters:
`val` - value with which the minimum is to be computed.

Returns:
the BigDecimal whose value is the lesser of this BigDecimal and `val`. If they are equal, as defined by the `compareTo` method, either may be returned.

See Also:
`compareTo(java.math.BigDecimal)`

---

max

`public BigDecimal **max**(BigDecimal val)`

Returns the maximum of this BigDecimal and `val`.

Parameters:
`val` - value with which the maximum is to be computed.

Returns:

the BigDecimal whose value is the greater of this BigDecimal and `val`. If they are equal, as defined by the `compareTo` method, either may be returned.

See Also:

`compareTo(java.math.BigDecimal)`

---

hashCode

```
public int hashCode()
```

Returns the hash code for this BigDecimal. Note that two BigDecimals that are numerically equal but differ in scale (like 2.0 and 2.00) will generally *not* have the same hash code.

Overrides:

`hashCode` in class `Object`

Returns:

hash code for this BigDecimal.

See Also:

`Object.equals(java.lang.Object)`, `Hashtable`

---

toString

```
public String toString()
```

Returns the string representation of this BigDecimal. The digit-to- character mapping provided by `Character.forDigit(int, int)` is used. A leading minus sign is used to indicate sign, and the number of digits to the right of the decimal point is used to indicate scale. (This representation is compatible with the (String) constructor.)

Overrides:

`toString` in class `Object`

Returns:

String representation of this BigDecimal.

See Also:

`Character.forDigit(int, int)`, `BigDecimal(java.lang.String)`

---

toBigInteger

```
public BigInteger toBigInteger()
```

Converts this BigDecimal to a BigInteger. This conversion is analogous to a *narrowing primitive conversion* from `double` to `long` as defined in the Java Language Specification: any fractional part of this BigDecimal will be discarded. Note that this conversion can lose information about the precision of the BigDecimal value.

Returns:
this BigDecimal converted to a BigInteger.

---

intValue

```
public int intValue()
```

Converts this BigDecimal to an `int`. This conversion is analogous to a *narrowing primitive conversion* from `double` to `short` as defined in the Java Language Specification: any fractional part of this BigDecimal will be discarded, and if the resulting "BigInteger" is too big to fit in an `int`, only the low-order 32 bits are returned. Note that this conversion can lose information about the overall magnitude and precision of the BigDecimal value as well as return a result with the opposite sign.

Specified by:
`intValue` in class `Number`
Returns:
this BigDecimal converted to an `int`.

---

longValue

```
public long longValue()
```

Converts this BigDecimal to a `long`. This conversion is analogous to a *narrowing primitive conversion* from `double` to `short` as defined in the Java Language Specification: any fractional part of this BigDecimal will be discarded, and if the resulting "BigInteger" is too big to fit in a `long`, only the low-order 64 bits are returned. Note that this conversion can lose information about the overall magnitude and precision of the BigDecimal value as well as return a result with the opposite sign.

Specified by:
`longValue` in class `Number`
Returns:
this BigDecimal converted to an `long`.

---

floatValue

```
public float floatValue()
```

Converts this BigDecimal to a `float`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` defined in the Java Language Specification: if this BigDecimal has too great a magnitude to represent as a `float`, it will be converted to `Float.NEGATIVE_INFINITY` or `Float.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigDecimal value.

Specified by:

`floatValue` in class `Number`

Returns:

this BigDecimal converted to a `float`.

---

doubleValue

```
public double doubleValue()
```

Converts this BigDecimal to a `double`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` as defined in the Java Language Specification: if this BigDecimal has too great a magnitude represent as a `double`, it will be converted to `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigDecimal value.

Specified by:

`doubleValue` in class `Number`

Returns:

this BigDecimal converted to a `double`.

Submit a bug or feature

For further API reference and developer documentation, see Java 2 SDK SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Problem Statement

Cribbage is a card game played with a standard 52-card deck. A hand in cribbage contains five cards, one of which is designated the *starter card*. A hand earns points as follows:

- **Fifteens:** Every distinct combination of cards that sums to fifteen contributes two points. For the purposes of summing, aces are worth one and face cards (jacks, queens, and kings) are each worth ten. All other cards are worth their numeric value.
- **Pairs:** Every distinct pair of cards of the same rank contributes two points.
- **Runs:** Every distinct run of at least three cards of consecutive ranks contributes a number of points equal to the length of the run. Aces are ranked below twos, and jacks, queens, and kings are ranked above tens, in that order. Kings and aces are not consecutive, so runs that wrap around are not permitted. Do not count a run if a longer run including the same cards is possible.
- **Flush:** A five-card flush (all five cards the same suit) contributes five points. A four-card flush (four cards of one suit and the fifth card a different suit) contributes four points if and only if the card with the non-matching suit is the starter card.
- **His Nobs:** A jack in the same suit as the starter card contributes one point, and is called "his nobs". Do not count the point if the jack *is* the starter card.

Note that a single card may be part of several different fifteens, pairs, and/or runs. For example, in a hand containing the 5 of clubs and all four tens, the 5 of clubs would participate in four distinct fifteens (once with each ten) and the ten of spades would participate in three distinct pairs of tens (once with each of the other tens).

A hand is written as a string in the following format. A single card is written as a rank ('A','2'-'9','T','J','Q','K') followed by a suit ('C','D','H','S'). A hand is five cards separated by commas. The first card listed is the starter card.

For example, the hand `"4H,6D,5S,JH,6C"` (quotes for clarity only) is worth 15 points.

- Three fifteens: `4H+5S+6D`, `4H+5S+6C`, and `5S+JH`.
- One pair: `6D+6C`.
- Two runs: `4H+5S+6D` and `4H+5S+6C`.
- His nobs: `JH` (starter card is `4H`).

Create a class **Cribbage** containing a method **score** that takes a string **hand** and calculates its score.

Definition

Class:	Cribbage
Method:	score
Parameters:	string
Returns:	int
Method signature:	int score(string hand)
(be sure your method is public)	

Notes

- The letter 'A' means "ace", 'T' means "ten", 'J' means "jack", 'Q' means "queen", and 'K' means "king".

Constraints

- **hand** contains exactly fourteen characters, and has the format `"<R1><S1>,...,<R5><S5>"`, where each `<Ri>` is one of the characters `{'A','2','3','4','5','6','7','8','9','T','J','Q','K'}` and each `<Si>` is one of the characters `{'C','D','H','S'}`.
- No card appears twice in the same hand.

Examples

0)

```
"4H,6D,5S,JH,6C"
Returns: 15
```

The example above.

1)

```
"5H,5S,JH,5C,5D"
Returns: 29
```

8 fifteens, 6 pairs, and his nobs for 16+12+1 = 29 points.

2)

```
"JS,TH,JH,9H,QH"
Returns: 14
```

1 pair, 2 four-card runs, and a four-card flush for 2+8+4 = 14 points.

3)

```
"7S,6H,7D,8C,8S"
Returns: 24
```

4 fifteens, 2 pairs, and 4 three-card runs for 8+4+12 = 24 points.

4)

```
"AH,2H,6C,QH,KH"
Returns: 0
```

Runs may not wrap around and four-card flushes may not include the starter card.

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

class card
{
  public:
    void set( char r, char s )
    {
      rank_char=r;
      switch( r )
      {
        case 'A': rank=1; break;
        case '2': rank=2; break;
        case '3': rank=3; break;
        case '4': rank=4; break;
        case '5': rank=5; break;
        case '6': rank=6; break;
        case '7': rank=7; break;
        case '8': rank=8; break;
        case '9': rank=9; break;
        case 'T': rank=10; break;
        case 'J': rank=11; break;
        case 'Q': rank=12; break;
        case 'K': rank=13; break;
      }
      value = (rank>10?10:rank);
      suit=s;
    }
    bool operator<(const card& lv)const{return rank<lv.rank;}
    friend ostream& operator<<(ostream& os, const card& lv);
    char rank_char;
    int rank;
    int value;
    char suit;
};

ostream& operator<<(ostream& os, const card& lv)
{
  os << lv.rank_char << lv.suit;
}

class Cribbage
{
        public:
                int score(string str)
                {
      int rv=0,i,j,k,l;
      vector<card> hand;
      card hp;
      for( i=0;i<5;i++ )
      {
        hp.set( str[3*i], str[3*i+1] );
        hand.push_back(hp);
      }

      cout << "Scoring hand:";
      for( i=0;i<5;i++ ) cout << " " << hand[i];
      cout << endl;

      // check for his nobs
      for( i=1;i<5;i++ )
      {
        if( hand[i].rank == 11 && hand[i].suit == hand[0].suit )
        {
          cout << hand[i] << " his nobs for 1" << endl;
          rv+=1;
```

```
        }
      }

      // check for flush
      if( hand[1].suit == hand[2].suit && hand[2].suit == hand[3].suit
          && hand[3].suit == hand[4].suit )
      {
        if( hand[0].suit == hand[1].suit )
        {
          cout << hand[0] << "," << hand[1] << "," << hand[2] << ","
            << hand[3] << "," << hand[4] << " flush for 5" << endl;
          rv+=5;
        }
        else
        {
          cout << hand[1] << "," << hand[2] << "," << hand[3] << ","
            << hand[4] << " flush for 4" << endl;
          rv+=4;
        }
      }

      // sorting makes run checking easy
      sort(hand.begin(),hand.end());

      // check for fifteens and pairs
      for( i=0;i<4;i++ )
      {
        for( j=i+1;j<5;j++ )
        {
          if( hand[i].value+hand[j].value == 15 )
          {
            cout << hand[i] << "+" << hand[j] << "=15 for 2" << endl;
            rv+=2;
          }
          if( hand[i].rank == hand[j].rank )
          {
            cout << hand[i] << "," << hand[j] << " is a pair for 2" << endl;
            rv+=2;
          }
        }
      }

      for( i=0;i<3;i++ )
      {
        for( j=i+1;j<4;j++ )
        {
          for( k=j+1;k<5;k++ )
          {
            if( hand[i].value+hand[j].value+hand[k].value == 15 )
            {
              cout << hand[i] << "+" << hand[j] << "+" << hand[k]
                << "=15 for 2" << endl;
              rv+=2;
            }
          }
        }
      }
      for( i=0;i<2;i++ )
      {
        for( j=i+1;j<3;j++ )
        {
          for( k=j+1;k<4;k++ )
          {
            for( l=k+1;l<5;l++ )
            {
              if( hand[i].value+hand[j].value+hand[k].value+
                  hand[l].value == 15 )
              {
                cout << hand[i] << "+" << hand[j] << "+" << hand[k] << "+"
```

```
                  << hand[l] << "=15 for 2" << endl;
                rv+=2;
              }
            }
          }
        }
      }
      if( hand[0].value+hand[1].value+hand[2].value+hand[3].value+
          hand[4].value == 15 )
      {
        cout << hand[0] << "+" << hand[1] << "+" << hand[2] << "+"
          << hand[3] << "+" << hand[4] << "=15 for 2" << endl;
        rv+=2;
      }


      // check for runs of 5
      if( hand[0].rank+1 == hand[1].rank &&
          hand[1].rank+1 == hand[2].rank &&
          hand[2].rank+1 == hand[3].rank &&
          hand[3].rank+1 == hand[4].rank )
      {
        cout << hand[0] << ","<< hand[1] << "," << hand[2] << ","
          << hand[3] << "," << hand[4] << " run for 5" << endl;
        rv+=5;
      }
      // of course, don't check for more runs if there was run of 5!
      else
      {
        bool foundrun=false;
        // now check for runs of 4
        for( i=0;i<2;i++ )
        {
          for( j=i+1;j<3;j++ )
          {
            for( k=j+1;k<4;k++ )
            {
              for( l=k+1;l<5;l++ )
              {
                if( hand[i].rank+1 == hand[j].rank &&
                    hand[j].rank+1 == hand[k].rank &&
                    hand[k].rank+1 == hand[l].rank )
                {
                  cout << hand[i] << "," << hand[j] << "," << hand[k] << ","
                    << hand[l] << " a run for 4" << endl;
                  rv+=4,foundrun=true;
                }
              }
            }
          }
        }
        if( !foundrun )
        {
          // now for runs of 3
          for( i=0;i<3;i++ )
          {
            for( j=i+1;j<4;j++ )
            {
              for( k=j+1;k<5;k++ )
              {
                if( hand[i].rank+1 == hand[j].rank &&
                    hand[j].rank+1 == hand[k].rank )
                {
                  cout << hand[i] << "," << hand[j] << "," << hand[k]
                    << " a run for 3" << endl;
                  rv+=3;
                }
              }
            }
```

```
          }
        }
      }

      return rv;
                }
};
```

Problem Statement

Recently all cars in NJ had a specific pattern of digits and letters for their license plate numbers. For example, the license plate number "UG830H" follows the pattern "LLDDDL", where 'L' stands for letter, and 'D' stands for digit. 17,576,000 distinct license plate numbers can be generated using this format. As the population and the number of cars grows, new formats appear. For example, the new format in NJ is "LLLDDL" (as in "MEL87H"). The new format will be able to number 45,697,600 cars. Assume that plates are assigned in lexicographic order (the order they would appear in a dictionary. i.e. "AAA" comes before "AAB" and "AA1" comes before "AA7"). For practical purposes it is important to know how many plates are available for further assignment at each moment in time.

Your task is, given a string representing a license plate number, return the total number of possible license plates following the same pattern as the input that can be assigned after the given plate number.

For example, given the string "D43", we must count all of the plate numbers that have one letter, followed by two digits, and come after "D43". Any sequence that starts with a letter after 'D' works (there are 100*22 of these), as does any sequence that starts with the letter 'D' and is followed by a two digit number greater than "43" (there are 56 of these).

Definition

Class:	Plates
Method:	numLeft
Parameters:	string
Returns:	long long
Method signature:	long long numLeft(string plateNum)

(be sure your method is public)

Notes

- We assume that only upper-case letters and digits can be used in plate numbers.
- Digit 0 (zero) should be treated like any other digit. That is, the plate number can start with 0, if it starts with a digit.

Constraints

- **plateNum** contains between 3 and 8 characters inclusive
- each character of **plateNum** is a digit ('0'-'9') or an upper-case letter ('A'-'Z')

Examples

0)

```
"222"
Returns: 777
```

plates are made of three digits starting from "000" and ending with "999".

1)

```
 "TA4KA"
Returns: 1227355
```

2)

```
 "KAPETA"
Returns: 189835177
```

3)

```
 "MAPA3M"
Returns: 63875149
```

4)

```
 "ZZZZZ"
Returns: 0
```

5)

```
 "D43"
Returns: 2256
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
// from SRM 135 DIV I 250
// by ~Andy~
#include <string>
#include <vector>
using namespace std;
class Plates {
        public:
        long long numLeft(string plateNum) {
    // my solution failed because I used long int instead of long long!!!
                long long total=1;
                long long rep=0;
                for( int i=plateNum.length()-1; i>=0; i-- )
                {
                        if( plateNum[i] >= '0' && plateNum[i] <= '9' )
                        {
                                rep+=(plateNum[i]-'0')*total;
                                total*=10;
                        }
                        else
                        {
                                rep+=(plateNum[i]-'A')*total;
                                total*=26;
                        }
                }
                return total-rep-1;
        }
};

// Powered by PopsEdit
```

```
/* This problem solves the Fibonacci string problem (also called the golden
 * string). The problem is:
 * given two strings a and b, find the number of occurences of string c at
 * the nth entry in the following sequence:
 * 0: a
 * 1: b
 * 2: ba
 * 3: bab
 * 4: babba
 * 5: babbabab
 * 6: babbababbabba
 * 7: ...
 * this is done by taking c(n) = c(n−1) + c(n−2) where + is concatenation
 *
 * This problem was posed by Dr. J to Andy Martin for practice
 */

/* Solution coded by: C. Andy Martin */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

#include "bigint.h"
typedef bigint mynum_t;
//in case you don’t have bigints, use long long
//typedef long long mynum_t;

int count_occ( string x, string f )
{
  int i,rv=0;
  /* BOY what a SNEAKY problem!!!
   * x.length()−f.length(), what is it when x="" and f="x" ???
   * it is, of course 0xffffffff, BUT they are UNSIGNED ints,
   * so this is a positive 4.2 billion something!!!!
   * so add an if to fix this */
  if( x.length() >= f.length() )
  {
    for( i=0;i<=x.length()−f.length();i++ )
    {
      if( x.substr(i,f.length()) == f )
      {
        rv++;
      }
    }
  }
  return rv;
}

int main()
{
  string xl, xr, yl, yr, zl, zr, match;
  mynum_t xc, yc, zc;
  int n,i,j;
  int state;

  ifstream infile( "fibstring.in" );
  ofstream outfile( "fibstring.out" );

  if( !infile ) { cerr << "Could not open input file" << endl; return 1; }
  if( !outfile ) { cerr << "Could not open outfile file" << endl; return 2; }

  // input spec (using worldfinals input conventions):
  // first string is string x ( |x| <= 10 )
  // second string is string y ( |y| <= 10 )
  // third string is the string to match ( |match| <= 70 )
```

```
  // fourth number is number of iterations of concatenation ( n <= max_n )

  infile >> xl >> yl >> match >> n;
  xc=yc=zc=0;
  zl=zr=yr=xr="";
  state=0;
  while( infile )
  {
    // run n concat’s of z=y+x
//    cout << xl << endl << yl << endl;
    for( i=0;i<n;i++ )
    {
      if( state == 2 )
      {
//        cout << yl << yc << yr << xl << xc << xr << ":";
        zc = xc + yc + count_occ(yr+xl,match);
        zl=yl;
        zr=xr;
      }
      else if( state==1 )
      {
        state++;
//        cout << yl << yc << yr << xl << ":";
        zc = yc + count_occ(yr+xl,match);
        zl=yl;
        zr=xl.substr(xl.length()−match.length()+1,match.length()−1);
      }
      else
      {
        zl = yl + xl;
//        cout << zl <<":";
        if( zl.length() > 2*(match.length()−1) )
        {
          state++;
          // split string now
          zc = count_occ( zl, match );
          zr = zl.substr(zl.length()−match.length()+1,match.length()−1);
          zl = zl.substr(0,match.length()−1);
        }
      }
      xr=yr;
      xl=yl;
      yr=zr;
      yl=zl;
      xc=yc;
      yc=zc;
//      cout << zl << zc << zr << endl;
    }

    // if n was so small we never got to count ...
    if( state == 0 )
    {
      zc = count_occ( zl, match );
    }

    outfile << zc << endl;
    infile >> xl >> yl >> match >> n;
    xc=yc=zc=0;
    state=0;
    zl=zr=yr=xr="";
  }

  return 0;
}
```

Problem Statement

Suppose you had an *n* by *n* chess board and a super piece called a kingknight. Using only one move the kingknight denoted 'K' below can reach any of the spaces denoted 'X' or 'L' below:

```
.......
..L.L..
.LXXXL.
..XKX..
.LXXXL.
..L.L..
.......
```

In other words, the kingknight can move either one space in any direction (vertical, horizontal or diagonally) or can make an 'L' shaped move. An 'L' shaped move involves moving 2 spaces horizontally then 1 space vertically or 2 spaces vertically then 1 space horizontally. In the drawing above, the 'L' shaped moves are marked with 'L's whereas the one space moves are marked with 'X's. In addition, a kingknight may never jump off the board.

Given the **size** of the board, the **start** position of the kingknight and the **end** position of the kingknight, your method will return how many possible ways there are of getting from **start** to **end** in exactly **numMoves** moves. **start** and **finish** are vector <int>s each containing 2 elements. The first element will be the (0-based) row position and the second will be the (0-based) column position. Rows and columns will increment down and to the right respectively. The board itself will have rows and columns ranging from 0 to **size**-1 inclusive.

Note, two ways of getting from **start** to **end** are distinct if their respective move sequences differ in any way. In addition, you *are* allowed to use spaces on the board (including **start** and **finish**) repeatedly during a particular path from **start** to **finish**. We will ensure that the total number of paths is less than or equal to 2^63-1 (the upper bound for a long long).

Definition

Class:	ChessMetric
Method:	howMany
Parameters:	int, vector <int>, vector <int>, int
Returns:	long long
Method signature:	long long howMany(int size, vector <int> start, vector <int> end, int numMoves)
(be sure your method is public)	

Notes

- For C++ users: long long is a 64 bit datatype and is specific to the GCC compiler.

Constraints

- **size** will be between 3 and 100 inclusive

- **start** will contain exactly 2 elements
- **finish** will contain exactly 2 elements
- Each element of **start** and **finish** will be between 1 and **size**-1 inclusive
- **numMoves** will be between 1 and 50 inclusive
- The total number of paths will be at most 2^63-1.

Examples

0)

```
3
{0,0}
{1,0}
1
Returns: 1
```

Only 1 way to get to an adjacent square in 1 move

1)

```
3
{0,0}
{1,2}
1
Returns: 1
```

A single L-shaped move is the only way

2)

```
3
{0,0}
{2,2}
1
Returns: 0
```

Too far for a single move

3)

```
3
{0,0}
{0,0}
2
Returns: 5
```

Must count all the ways of leaving and then returning to the corner

4)

```
100
{0,0}
{0,99}
50
Returns: 243097320072600
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

long long atsq( int x, int y, const vector < vector < long long > > & z )
{
  if( x < 0 || y < 0 || x >= z.size() || y >= z[0].size() ) return 0;
  return z[x][y];
}

class ChessMetric
{
        public:
                long long howMany(int size, vector <int> start, vector <int> end
, int numMoves)
                {
      vector < vector < long long > > b, ob;
      vector < long long > t;
      int i,j,k;
      for( i=0;i<size;i++ )
      {
        for( j=0;j<size;j++ )
          t.push_back(0);
        b.push_back( t );
      }
      b[start[0]][start[1]]=1;
      for( i=0;i<numMoves;i++ )
      {
//        cout << "At step " << i+1 << endl;
        ob=b;
        for( j=0;j<size;j++ )
        {
          for( k=0;k<size;k++ )
          {
            b[j][k] = atsq(j-1,k-1,ob) +
              atsq(j-1,k,ob) +
              atsq(j-1,k+1,ob) +
              atsq(j,k-1,ob) +
              atsq(j,k+1,ob) +
              atsq(j+1,k-1,ob) +
              atsq(j+1,k,ob) +
              atsq(j+1,k+1,ob) +
              atsq(j+1,k-2,ob) +
              atsq(j+1,k+2,ob) +
              atsq(j-1,k-2,ob) +
              atsq(j-1,k+2,ob) +
              atsq(j+2,k-1,ob) +
              atsq(j+2,k+1,ob) +
              atsq(j-2,k-1,ob) +
              atsq(j-2,k+1,ob);
//            cout << b[j][k] << " ";
          }
//          cout << endl;
        }
      }
      return b[end[0]][end[1]];
                }
};


// Powered by FileEdit
```

Problem B: Ferry Loading II

Before bridges were common, ferries were used to transport cars across rivers. River ferries, unlike their larger cousins, run on a guide line and are powered by the river's current. Cars drive onto the ferry from one end, the ferry crosses the river, and the cars exit from the other end of the ferry.

There is a ferry across the river that can take *n* cars across the river in *t* minutes and return in *t* minutes. *m* cars arrive at the ferry terminal by a given schedule. What is the earliest time that all the cars can be transported across the river? What is the minimum number of trips that the operator must make to deliver all cars by that time?

The first line of input contains *c*, the number of test cases. Each test case begins with *n*, *t*, *m*. *m* lines follow, each giving the arrival time for a car (in minutes since the beginning of the day). The operator can run the ferry whenever he or she wishes, but can take only the cars that have arrived up to that time. For each test case, output a single line with two integers: the time, in minutes since the beginning of the day, when the last car is delivered to the other side of the river, and the minimum number of trips made by the ferry to carry the cars within that time.

You may assume that 0 < n, t, m < 1440. The arrival times for each test case are in non-decreasing order.

Sample input

```
2
2 10 10
0
10
20
30
40
50
60
70
80
90
2 10 3
10
30
40
```

Output for sample input

```
100 5
50 2
```

```c
/* @JUDGE_ID: 27516PA B C "dynamic" */
#include <stdio.h>
#include <string.h>

int n,t,m;
int car[1440];
//int mem[1440];

int read_states(void){
  int a;
  scanf("%d %d %d\n",&n, &t, &m);
  for (a=0; a<m; a++) {
    scanf("%d\n",&car[a]);
    //mem[a] = 0;
  }
}

typedef struct {
  int time;
  int trips;
} info;

info ans;
//info memo[1440];
inline info rec(int taken, int Time) {
  int c, w;
  info cse;
  info min;
  min.time = 99999999;

  for (c=1; c<=n && (c+taken)<=m; c++) {

    // if the next car can fit, wait for him!

    if (car[taken+c] < Time && c<n && (c+taken) <m)
      continue;

    // calculate the wait time

    w = 0;
    if (car[taken+c-1]>Time)
      w = car[taken+c-1]-Time;

    // calculate the total time

    if (taken + c < m) {
      cse = rec(taken+c, Time + w + 2*t);
      cse.time += w + 2*t;
      cse.trips += 1;
      //printf("%d %d %d %d %d < m\n", c, Time, cse.time, cse.trips , taken + c)
;
    } else {
      cse.trips = 1;
      cse.time = w + t; // no return trip!
      //printf("%d %d %d %d %d == m\n", c, Time, cse.time, cse.trips , taken + c
);
    }

    // if better, use it!

    if (cse.time < min.time)
      min = cse;
    else if (cse.time == min.time)
      if (cse.trips < min.trips)
        min.trips = cse.trips;
  }
  return min;
}
```

```c
int evolve_states(void) {
  ans = rec(0, 0);
}

int output_states(void) {
  printf("%d %d\n", ans.time, ans.trips);
}

void main(void) {
  int tests;
  scanf("%d\n", &tests);
  do {
    tests--;
    read_states();
    evolve_states();
    output_states();
  } while (tests > 0);
}
```

Problem E: Rock, Scissors, Paper

Bart's sister Lisa has created a new civilization on a two-dimensional grid. At the outset each grid location may be occupied by one of three life forms: *Rocks*, *Scissors*, or *Papers*. Each day, differing life forms occupying horizontally or vertically adjacent grid locations wage war. In each war, Rocks always defeat Scissors, Scissors always defeat Papers, and Papers always defeat Rocks. At the end of the day, the victor expands its territory to include the loser's grid position. The loser vacates the position.

Your job is to determine the territory occupied by each life form after *n* days. The first line of input contains *t*, the number of test cases. Each test case begins with three integers not greater than 100: *r* and *c*, the number of rows and columns in the grid, and *n*. The grid is represented by the *r* lines that follow, each with *c* characters. Each character in the grid is R, S, or P, indicating that it is occupied by Rocks, Scissors, or Papers respectively.

For each test case, print the grid as it appears at the end of the *n*th day. Leave an empty line between the output for successive test cases.

Sample Input

```
2
3 3 1
RRR
RSR
RRR
3 4 2
RSPR
SPRS
PRSP
```

Output for Sample Input

```
RRR
RRR
RRR

RRRS
RRSP
RSPR
```

```
/* @JUDGE_ID: 27516PA E C "Celluar Automata" */
#include <stdio.h>
#include <string.h>

char state[2][103][103];
int cur;
int x,y,n;
FILE* input;

int read_states(void){
  int a,b;
  cur = 0;
  for (a=0; a<103; a++)
    for (b=0; b<103; b++) {
      state[0][a][b] = 0;
      state[1][a][b] = 0;
    }
  scanf("%d %d %d\n", &y, &x, &n);
  for (a=1; a<=y; a++)
  {
    for (b=1; b<=x; b++)
      scanf("%c", &state[cur][a][b]);
    scanf("\n", &state[cur][a][x+1]);
    state[cur][a][x+1]=0;
  }
}

int evolve_states(void) {
  int a,b;
  int next=!cur;
  for (a=1; a<=y; a++)
    for (b=1; b<=x; b++)
      switch (state[cur][a][b]) {
        case 'R':
          if ((state[cur][a+1][b] == 'P') || (state[cur][a-1][b] == 'P') ||
             (state[cur][a][b+1] == 'P') || (state[cur][a][b-1] == 'P'))
            state[next][a][b] = 'P';
          else
            state[next][a][b] = 'R';
          break;
        case 'P':
          if ((state[cur][a+1][b] == 'S') || (state[cur][a-1][b] == 'S') ||
             (state[cur][a][b+1] == 'S') || (state[cur][a][b-1] == 'S'))
            state[next][a][b] = 'S';
          else
            state[next][a][b] = 'P';
          break;
        case 'S':
          if ((state[cur][a+1][b] == 'R') || (state[cur][a-1][b] == 'R') ||
             (state[cur][a][b+1] == 'R') || (state[cur][a][b-1] == 'R'))
            state[next][a][b] = 'R';
          else
            state[next][a][b] = 'S';
          break;
      }
  cur = next;
}

int output_states(void) {
  int a,b;
  for (a=1; a<=y; a++) {
    for (b=1; b<=x; b++)
      printf("%c", state[cur][a][b]);
    printf("\n");
  }
}

void main(void) {
  int tests;
```

```
  scanf("%d\n", &tests);
  do {
    tests--;
    read_states();
    while (n>0) {
      evolve_states();
      n--;
    }
    output_states();
    if (tests)
      printf("\n");
  } while (tests > 0);
}
```

The 3*n* + 1 problem

Background

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

The Problem

Consider the following algorithm:

```
1.    input $n$
2.    print $n$
3.    if $n$ = 1 then STOP
4.        if $n$ is odd then $n \longleftarrow 3n+1$
5.        else $n \longleftarrow n/2$
6.    GOTO 2
```

Given the input 22, the following sequence of numbers will be printed 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1{,}000{,}000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed (including the 1). For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between i and j.

The Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including *i* and *j*.

The Output

For each pair of input integers *i* and *j* you should output *i*, *j*, and the maximum cycle length for integers between and including *i* and *j*. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers *i* and *j* must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

```
/* @judge_id: 27516PA 100 C++ "Memoization" */

#include <iostream>
#include <math.h>

#define MAX_N 1000000

unsigned short int * m;

unsigned short cyc_len( double n )
{
  long int intn;
  bool isodd = fmod( n, 2 );
  if( n < MAX_N )
  {
    intn = (long int)(n+.5);
  }
  else
  {
    intn = -1;
  }
  if( intn > 0 && m[intn] ) return m[intn];
  double next_n = ((isodd)?(3*n+1):(n/2));
  unsigned short cl = cyc_len( next_n )+1;
  if( intn > 0 ) m[intn]=cl;
  return cl;
}

void init( void )
{
  m = new unsigned short[MAX_N];

  // seed the table
  m[1]=1;
  // fill the table
  for( double i=MAX_N-1; i>0; i-- )
  {
    cyc_len( i );
  }
}

int main()
{
  init();

  long int i,j,k;
  unsigned short max;
  cin >> i >> j;
  while( cin )
  {
    max=0;
    if( i >= MAX_N || j >= MAX_N || i <= 0 || j <=0 )
    {
      cerr << "input out of range" << endl;
    }
    else
    {
      if( j > i )
      {
        for(k=i;k<=j;k++)
        {
          if( m[k] > max ) max=m[k];
        }
      }
      else
      {
        for(k=j;k<=i;k++)
        {
          if( m[k] > max ) max=m[k];
```

```
        }
      }
      cout << i << " " << j << " " << max << endl;
    }
    cin >> i >> j;
  }
}
```

The 2003 27th Annual **acm** International Collegiate

Programming Contest World Finals

sponsored by IBM

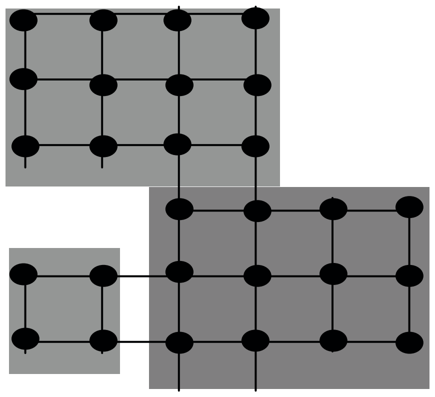
Problem D
Eurodiffusion
Input File: euro.in

On January 1, 2002, twelve European countries abandoned their national currency for a new currency, the euro. No more francs, marks, lires, guldens, kroner,... only euros, all over the eurozone. The same banknotes are used in all countries. And the same coins? Well, not quite. Each country has limited freedom to create its own euro coins:

> "Every euro coin carries a common European face. On the obverse, member states decorate the coins with their own motif. No matter which motif is on the coin, it can be used anywhere in the 12 Member States. For example, a French citizen is able to buy a hot dog in Berlin using a euro coin with the imprint of the King of Spain." (source: http://europa.eu.int/euro/html/entry.html)

On January 1, 2002, the only euro coins available in Paris were French coins. Soon the first non-French coins appeared in Paris. Eventually, one may expect all types of coins to be evenly distributed over the twelve participating countries. (Actually this will not be true. All countries continue minting and distributing coins with their own motifs. So even in a stable situation, there should be an excess of German coins in Berlin.) So, how long will it be before the first Finnish or Irish coins are in circulation in the south of Italy? How long will it be before coins of each motif are available everywhere?

You must write a program to simulate the dissemination of euro coins throughout Europe, using a highly simplified model. Restrict your attention to a single euro denomination. Represent European cities as points in a rectangular grid. Each city may have up to 4 neighbors (one to the north, east, south and west). Each city belongs to a country, and a country is a rectangular part of the plane. The figure below shows a map with 3 countries and 28 cities. The graph of countries is connected, but countries may border holes that represent seas, or non-euro countries such as Switzerland or Denmark. Initially, each city has one million (1000000) coins in its country's motif. Every day a representative portion of coins, based on the city's beginning day balance, is transported to each neighbor of the city. A representative portion is defined as one coin for every full 1000 coins of a motif.



A city is *complete* when at least one coin of each motif is present in that city. A country is *complete* when all of its cities are complete. Your program must determine the time required for each country to become complete.

Input

The input consists of several test cases. The first line of each test case is the number of countries ($1 \leq c \leq 20$). The next c lines describe each country. The country description has the format: *name xl yl xh yh*, where *name* is a single word with at most 25 characters; *xl*, *yl* are the lower left city coordinates of that country (most southwestward city) and *xh, yh* are the upper right city coordinates of that country (most northeastward city). $1 \leq xl \leq xh \leq 10$ and $1 \leq yl \leq yh \leq 10$.

The last case in the input is followed by a single zero.

Output

For each test case, print a line indicating the case number, followed by a line for each country with the country name and number of days for that country to become complete. Order the countries by days to completion. If two countries have identical days to completion, order them alphabetically by name.

Use the output format shown in the example.

Sample Input

```
3
France  1 4 4 6
Spain          3 1 6 3
Portugal    1 1 2 2
1
Luxembourg  1 1 1 1
2
Netherlands 1 3 2 4
Belgium     1 1 2 2
0
```

Output for the Sample Input

```
Case Number 1
   Spain   382
   Portugal   416
   France   1325
Case Number 2
   Luxembourg   0
Case Number 3
   Belgium   2
   Netherlands   2
```

```
/* Problem D of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

struct result
{
  string country;
  int ftime;
  bool operator<( const result & rv ) const
  {
    return ftime < rv.ftime || ftime == rv.ftime && country < rv.country;
  }
  bool operator==( const result & rv ) const
  {
    return country == rv.country;
  }
};

struct country
{
  string name;
  int xl, yl, xh, yh;
  int count; // counts the number of currency types in all cities
  int target_count; // the value count will have when country is complete
  int my_cur_count; // counts number of cites with my currency
};

int main()
{
  ifstream infile( "euro.in" );
  int cnt=0;

  int i,j,k,q;
  // array representing europe and each countries currency
  int europe[12][12][20];
  int num_cities;
  int last_europe[12][12][20];
  int cmask[12][12];
  int time;
  int nc;
  country ctmp;
  result rtmp;
  vector <result> results;
  vector <country> countries;
  bool doprint=false;

  if( !infile ) { cerr << "File problems." << endl; return 1; }

  infile >> nc;
  while( nc )
  {
    time=0;
    // clear europe (its zero padded to remove bounds checks)
    memset( last_europe, 0, 12*12*20*sizeof(int) );
    memset( europe, 0, 12*12*20*sizeof(int) );
    memset( cmask, −1, 12*12*sizeof(int) );
    // clear results
    results.clear();
    // clear countries
    countries.clear();
    num_cities=0;
```

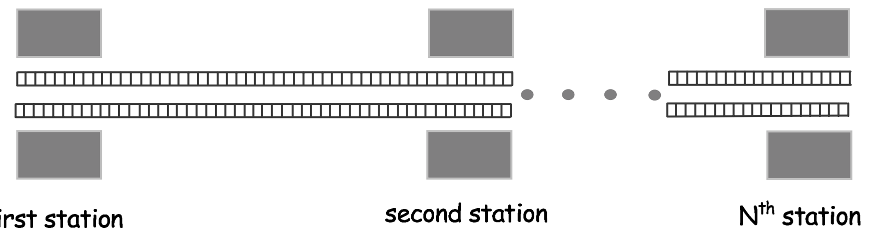
```
    // read in countries
    for( k=0;k<nc;k++ )
    {
      infile >> ctmp.name;
      infile >> ctmp.xl >> ctmp.yl >> ctmp.xh >> ctmp.yh;
      ctmp.count = (ctmp.xh−ctmp.xl+1)*(ctmp.yh−ctmp.yl+1);
      ctmp.my_cur_count=ctmp.count;
      num_cities+=ctmp.count;
      ctmp.target_count = ctmp.count * nc;
      countries.push_back( ctmp );
      for( i=ctmp.xl; i<=ctmp.xh; i++ )
      {
        for( j=ctmp.yl; j<=ctmp.yh; j++ )
        {
          cmask[i][j]=k;
          last_europe[i][j][k]=europe[i][j][k]=1000000;
        }
      }
    }
    /*
    for( i=0;i<12;i++ )
    {
      for( j=0;j<12;j++ )
      {
        cout << setw(3) << cmask[i][j];
      }
      cout << endl;
    }
    */
    if( nc == 1 )
    {
      rtmp.country = countries[0].name;
      rtmp.ftime = 0;
      results.push_back( rtmp );
    }
    else
    {
      while( results.size() != nc )
      {
        // propegate money
        time++;
        for( q=0;q<nc;q++ )
        {
          for( i=countries[q].xl;i<=countries[q].xh;i++ )
          {
            for( j=countries[q].yl;j<=countries[q].yh;j++ )
            {
              for( k=0;k<nc;k++ )
              {
                // don’t bother propegating money for countries which have
                // money in every city
                if( countries[k].my_cur_count == num_cities ) continue;
                if( last_europe[i][j][k]/1000 > 0 )
                {
#define my_macro(x,y) \
                  if( cmask[x][y] != −1 ) \
                  { \
                    if( europe[x][y][k] == 0 ) \
                    { \
                      countries[k].my_cur_count++; \
                      if( ++(countries[cmask[x][y]].count) ==  \
                          countries[cmask[x][y]].target_count ) \
                      { \
                        rtmp.country = countries[cmask[x][y]].name; \
                        rtmp.ftime = time; \
                        results.push_back( rtmp ); \
                        doprint = true; \
                      } \
                    } \
                    europe[x][y][k]+=last_europe[i][j][k]/1000; \
                    europe[i][j][k]-=last_europe[i][j][k]/1000; \
                  }
                  my_macro(i-1,j);
                  my_macro(i,j-1);
                  my_macro(i+1,j);
                  my_macro(i,j+1);
                }
              }
            }
          }
        }
        /*
        if( doprint )
        {
          doprint=false;
          cout << "Time: " << time << endl;
          for( k=0;k<nc;k++ )
          {
            cout << countries[k].name << "'s money:" << endl;
            if( countries[k].my_cur_count == num_cities )
            {
              cout << " has been distributed to all cities globally" << endl;
            }
            else
            {
              for( i=0;i<12;i++ )
              {
                for( j=0;j<12;j++ )
                {
                  cout << setw(10) << europe[i][j][k];
                }
                cout << endl;
              }
            }
          }
        }
        */
        // save the new europe
        memcpy( last_europe, europe, 12*12*20*sizeof(int) );
      }
    }

    sort( results.begin(), results.end() );
    cout << "Case Number " << ++cnt << endl;
    for( i=0; i<results.size(); i++ )
    {
      cout << " " << results[i].country << " " << results[i].ftime << endl;
    }
    infile >> nc;
  }
  return 0;
}
```

Problem H
A Spy in the Metro
Input File: metro.in

Secret agent Maria was sent to Algorithms City to carry out an especially dangerous mission. After several thrilling events we find her in the first station of Algorithms City Metro, examining the time table. The Algorithms City Metro consists of a single line with trains running both ways, so its time table is not complicated.

Maria has an appointment with a local spy at the last station of Algorithms City Metro. Maria knows that a powerful organization is after her. She also knows that while waiting at a station, she is at great risk of being caught. To hide in a running train is much safer, so she decides to stay in running trains as much as possible, even if this means traveling backward and forward. Maria needs to know a schedule with minimal waiting time at the stations that gets her to the last station in time for her appointment. You must write a program that finds the total waiting time in a best schedule for Maria.

The Algorithms City Metro system has N stations, consecutively numbered from 1 to N. Trains move in both directions: from the first station to the last station and from the last station back to the first station. The time required for a train to travel between two consecutive stations is fixed since all trains move at the same speed. Trains make a very short stop at each station, which you can ignore for simplicity. Since she is a very fast agent, Maria can always change trains at a station even if the trains involved stop in that station at the same time.



Input
The input file contains several test cases. Each test case consists of seven lines with information as follows.

Line 1. The integer N ($2 \le N \le 50$), which is the number of stations.
Line 2. The integer T ($0 \le T \le 200$), which is the time of the appointment.
Line 3. N-1 integers: t_1, t_2, ... t_{N-1} ($1 \le t_i \le 20$), representing the travel times for the trains between two consecutive stations: t_1 represents the travel time between the first two stations, t_2 the time between the second and the third station, and so on.
Line 4. The integer $M1$ ($1 \le M1 \le 50$), representing the number of trains departing from the first station.
Line 5. $M1$ integers: d_1, d_2, ... d_{M1} ($0 \le d_i \le 250$ and $d_i < d_{i+1}$), representing the times at which trains depart from the first station.
Line 6. The integer $M2$ ($1 \le M2 \le 50$), representing the number of trains departing from the N^{th} station.
Line 7. $M2$ integers: e_1, e_2, ... e_{M2} ($0 \le e_i \le 250$ and $e_i < e_{i+1}$) representing the times at which trains depart from the N^{th} station.

The last case is followed by a line containing a single zero.

Output
For each test case, print a line containing the case number (starting with 1) and an integer representing the total waiting time in the stations for a best schedule, or the word “impossible” in case Maria is unable to make the appointment. Use the format of the sample output.

Sample Input

```
4
55
5 10 15
4
0 5 10 20
4
0 5 10 15
4
18
1 2 3
5
0 3 6 10 12
6
0 3 5 7 12 15
2
30
20
1
20
7
1 3 5 7 11 13 17
0
```

Output for the Sample Input

```
Case Number 1: 5
Case Number 2: 0
Case Number 3: impossible
```

```
/* Problem H of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */
/* Solution notes:
 * This is a dynamic solution to problem H of the ACM 2003 world finals. we
 * build a set of piecewise functions which represent the minium amount of
 * waiting time needed versus arrival time at each of the stations. these
 * functions are represented as lists of pairs of numbers:
 * (3,1),(5,1),(6,0),(9,1)
 * (3,1) would mean that you can get to the station at time 3 and only have to
 * wait for one unit of time, (5,1) would mean that you can get to the
 * station at time 5 and wait 1 unit of time, (6,0), get there at 6, wait
 * zero, etc. The gaps are filled in by realizing that you can just sit at
 * the station to get (4,2) from (3,1), or (7,1) and (8,2) from (6,0). We
 * only add entries if they give us better waiting times than just sitting
 * at the station. Then, we notice that for each entry in the piecewise
 * list, we can take trains from the opposite direction we arrived from and
 * arrive at other stations with a possibly minimal arrival time. We must
 * check off each entry as being processed and remember the arrival
 * direction. We don’t add new entries if there are better ones already, but
 * we can add entries if it is a better entry. At the end of the process
 * when no more entries can be added (because all the trains have past) then
 * we have a complete picture of the minimal waiting times versus the
 * arrival time for the destination station. We then find the minimal
 * arrival time for our particular destination time (appointment time is the
 * terminology used in the problem
 */
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

class minArrive
{
  public:
    minArrive() { fromLeft=false; checked=false; arrivalTime=0; waitTime=0; }
    bool fromLeft;
    bool checked;
    int arrivalTime;
    int waitTime;
    // inserts *this into the vector, possibly removing an old value if the
    // closest or equal arrivalTime in v (v must be sorted by the operator
    // < defined in this function) results in more waiting time
    // return true if a value is inserted
    bool insertIfBetterThan( vector <minArrive> & v )
    {
      // if the vector is empty, we know we are better than anything in the
      // vector by default
      if( v.size() == 0 )
      {
        v.push_back( *this );
        return true;
      }
      vector <minArrive>::iterator it = upper_bound( v.begin(), v.end(), *this )
;
      // if we are less than all elements, than we have to be better
      // otherwise, if our wait time is such that if we waited at the
      // station from the last element we would have less wait time, we will
      // need to consider this another optimal point, so it is better.
      if( it == v.begin() || ((*(it−1)).waitTime + (arrivalTime − (*(it−1)).arri
valTime) > waitTime) )
      {
        if( it != v.begin() && (*(it−1)).arrivalTime == arrivalTime )
        {
          (*(it−1))=*this;
        }
        else
```

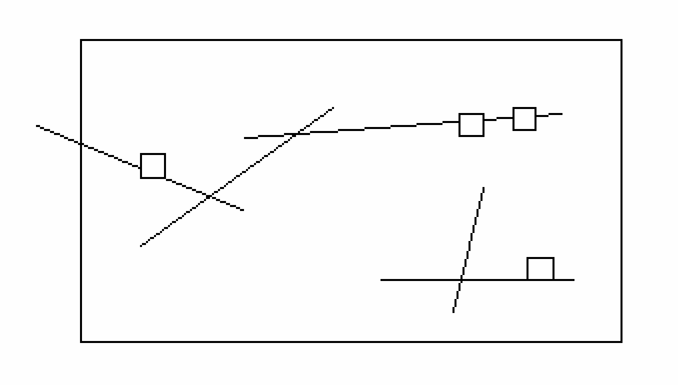
```
        {
          it = v.insert( it, *this );
          ++it;
        }
        while( it != v.end() )
        {
          if( waitTime + ((*it).arrivalTime − arrivalTime) <= (*it).waitTime )
          {
            it = v.erase(it);
          }
          else
          {
            ++it;
          }
        }
        return true;
      }
      return false;
    }
    bool operator<( const minArrive & r ) const
    {
      return arrivalTime < r.arrivalTime;
    }
};

int main()
{
  ifstream infile( "metro.in" );
  int cnt=0;
  vector <int> forwardTrains, reverseTrains, stationPositions;
  vector < vector<minArrive> > stationTimes;
  int i,j,k, cpos, n, appointmentTime, m1, m2, t;
  bool flg;

  if( !infile ) { cerr << "File problems." << endl; return 1; }

  infile >> n;
  while( n )
  {
    infile >> appointmentTime;
    stationTimes.resize(n);
    stationTimes[0].clear();
    stationPositions.clear();
    forwardTrains.clear();
    reverseTrains.clear();
    stationPositions.push_back(0);
    for( i=1,cpos=0;i<n;i++ )
    {
      stationTimes[i].clear();
      infile >> t;
      cpos+=t;
      stationPositions.push_back( cpos );
    }
    infile >> m1;
    for( i=0;i<m1;i++ )
    {
      infile >> t;
      forwardTrains.push_back(t);
    }
    infile >> m2;
    for( i=0;i<m2;i++ )
    {
      infile >> t;
      reverseTrains.push_back(t);
    }
    flg=true;
    minArrive wrk;
    // start at first station with zero time and no waiting time
    stationTimes[0].push_back(wrk);
```

```
begin:
    for(i=0;i<n;i++)
    {
      cout << "Station " << i+1 << " at " << stationPositions[i];
      for(j=0;j<stationTimes[i].size();j++)
      {
        cout << " ";
        if( stationTimes[i][j].fromLeft ) cout << "->";
        else cout << "<-";
        cout << "(" << stationTimes[i][j].arrivalTime << "," << stationTimes[i][
j].waitTime << ")";
        if( stationTimes[i][j].checked ) cout << "*";
      }
      cout << endl;
    }
    cout << endl;

    for(i=0;i<n;i++)
    {
      for(j=0;j<stationTimes[i].size();j++)
      {
        if( !stationTimes[i][j].checked )
        {
          vector <int>::iterator it;
          // if from left is true, we must take reverse trains to optimize
          // (taking trains in the same direction is equilvalent to just
          // not getting off a train and waiting at the later station)
          if( stationTimes[i][j].fromLeft )
          {
            // find the train which will get here earliest
            it = lower_bound( reverseTrains.begin(),reverseTrains.end(),stationT
imes[i][j].arrivalTime - (stationPositions[n-1]-stationPositions[i]));
            if( it != reverseTrains.end() )
            {
              wrk.fromLeft=false;
              wrk.checked=false;
              for( k=i-1;k>=0;k-- )
              {
                wrk.arrivalTime=(*it)+stationPositions[n-1]-stationPositions[k];
                wrk.waitTime=stationTimes[i][j].waitTime+(*it)+stationPositions[
n-1]-stationPositions[i]-stationTimes[i][j].arrivalTime;
                wrk.insertIfBetterThan( stationTimes[k] );
              }
            }
          }
          else
          {
            // find the train which will get here earliest
            it = lower_bound( forwardTrains.begin(),forwardTrains.end(),stationT
imes[i][j].arrivalTime - stationPositions[i]);
            if( it != forwardTrains.end() )
            {
              wrk.fromLeft=true;
              wrk.checked=false;
              for( k=i+1;k<n;k++ )
              {
                wrk.arrivalTime=(*it)+stationPositions[k];
                wrk.waitTime=stationTimes[i][j].waitTime+(*it)+stationPositions[
i]-stationTimes[i][j].arrivalTime;
                wrk.insertIfBetterThan( stationTimes[k] );
              }
            }
          }

          stationTimes[i][j].checked=true;
          goto begin;
        }
      }
    }
```

```
    cout << "Case Number " << ++cnt << ": ";
    wrk.fromLeft=false; wrk.checked=false;
    wrk.arrivalTime=appointmentTime; wrk.waitTime=0;
    vector <minArrive>::iterator min = upper_bound( stationTimes[n-1].begin(), s
tationTimes[n-1].end(), wrk );
    if( min == stationTimes[n-1].begin() )
    {
      cout << "impossible";
    }
    else
    {
      cout << (*(min-1)).waitTime + appointmentTime - (*(min-1)).arrivalTime;
    }
    cout << endl;

    infile >> n;
  }

  return 0;
}
```

Walking on Sticks

On a rectangle R we drop n sticks. On each stick we place one or more red points. A red point can move along sticks and move from one stick to another at their intersection. Your task is to report how many red points can be moved outside the rectangle.



In the above example, we can move three squares (red points) outside of the rectangle.

Input:

J - # input sets 1<=J<=10

X1 y1 x2 y2 – integers coordinates of the opposite corners of the rectangle.

N - # sticks -0<=N<=1000

N quadruples sx1 sy1 sx2 sy2 – integer coordinates of the endpoints of sticks.

M - # red points, 1<=M<=10000

M lines listing the stick number where the consecutive red points are placed.

Example input:

```
1
0 0 10 10
3
0 0 1 2
1 0 -1 2
3 3 5 5
5
1
3
3
2
1
```

Output:

```
3
```

(Note: Original source unknown – ask Dr. Jaromczyk for more information?)

```
// solved at a practice contest given by Dr. J. on 10/28/2002.
// solved by Andy Martin
#include <iostream>
#include <fstream>
#include <deque>
#include <string.h>

using namespace std;

struct point { int x; int y; };
struct seg { point p1; point p2; int nrp; };

#define _NORM(a) if( a != 0 ) a=((a>0)?1:-1);

int pt_seg ( seg s1, point p )
{
  int rv=(s1.p2.x-s1.p1.x)*(p.y-s1.p1.y) - (s1.p2.y-s1.p1.y)*(p.x-s1.p1.x);
  _NORM(rv);
  return rv;
}

bool seg_isect( seg s1, seg s2 )
{
  bool retval;
  // this is now handled by the colinear line segments case -- two points are
  // always 'colinear'
  /*
  if( s1.p1.x == s1.p2.x && s1.p1.y == s1.p2.y
   && s2.p1.x == s2.p2.x && s2.p1.y == s2.p2.y )
  {
    retval = (s1.p1.x == s2.p1.x && s1.p1.y == s2.p1.y);
  }
  else
  {
  */
    int cp1=pt_seg(s1,s2.p1);
    int cp2=pt_seg(s1,s2.p2);
    int cp3=pt_seg(s2,s1.p1);
    int cp4=pt_seg(s2,s1.p2);
    // if all the cross products are zero, we have two totally colinear
    // line segments, and we must check a different way to see if they
    // intersect or not
    if( cp1 == 0 && cp2 == 0 && cp3 == 0 && cp4 == 0 )
    {
      // check to see if the segments line up vertically
      if( s1.p1.x == s1.p2.x && s1.p1.x == s2.p1.x && s1.p1.x == s2.p2.x )
      {
        // we have to use y coordinates since vertical
        // if either segment has either endpoint in the other segment,
        // the two semgents intersect, otherwise they do not intersect
        retval=( ( s1.p1.y <= s2.p1.y && s2.p1.y <= s1.p2.y )
              || ( s1.p1.y <= s2.p2.y && s2.p2.y <= s1.p2.y )
              || ( s2.p1.y <= s1.p1.y && s1.p1.y <= s2.p2.y )
              || ( s2.p1.y <= s1.p2.y && s1.p2.y <= s2.p2.y ) );
      }
      else
      {
        // if either segment has either endpoint in the other segment,
        // the two semgents intersect, otherwise they do not intersect
        retval=( ( s1.p1.x <= s2.p1.x && s2.p1.x <= s1.p2.x )
              || ( s1.p1.x <= s2.p2.x && s2.p2.x <= s1.p2.x )
              || ( s2.p1.x <= s1.p1.x && s1.p1.x <= s2.p2.x )
              || ( s2.p1.x <= s1.p2.x && s1.p2.x <= s2.p2.x ) );
      }
    }
    else
    {
      retval=( cp1*cp2 <= 0 && cp3*cp4 <= 0);
    }
```

```
  /*
  }
  */

    /*
  cout << "cp1: " << cp1;
  cout << "cp2: " << cp2;
  cout << "cp3: " << cp3;
  cout << "cp4: " << cp4 << endl;
  cout << "segment ("<<s1.p1.x<<","<<s1.p1.y<<")-("<<s1.p2.x<<","<<s1.p2.y<<") "
;
  cout << "and segment ("<<s2.p1.x<<","<<s2.p1.y<<")-("<<s2.p2.x<<","<<s2.p2.y<<
") ";
  if( retval )
  {
    cout << "intersect" << endl;
  }
  else
  {
    cout << "do not intersect" << endl;
  }
  */
  return retval;

}

int main()
{
  int J;
  int x1,y1,x2,y2;
  seg rs1,rs2,rs3,rs4;
  int N;
  seg segs[1000];
  int M;
  deque<seg*> escape_segs;
  deque<seg*> next_escape_segs;
  deque<seg*> test_segs;
  int total_red;

  ifstream infile( "sticks.in" );
  ofstream outfile( "sticks.out" );

  if( !infile || !outfile )
  {
    cout << "bad files!" << endl;
    return 1;
  }

  infile >> J;

  for( int i=0;i<J;i++ )
  {
    escape_segs.clear();
    test_segs.clear();
    memset(segs,0,sizeof(segs));
    total_red=0;
    infile >> x1 >> y1 >> x2 >> y2;
    rs1.p2.y=rs1.p1.y=y1;
    rs1.p1.x=x1;
    rs1.p2.x=x2;
    rs2.p2.x=rs2.p1.x=x2;
    rs2.p1.y=y1;
    rs2.p2.y=y2;
    rs3.p2.y=rs3.p1.y=y2;
    rs3.p1.x=x1;
    rs3.p2.x=x2;
    rs4.p2.x=rs4.p1.x=x1;
    rs4.p1.y=y1;
    rs4.p2.y=y2;
    escape_segs.push_back(&rs1);
    escape_segs.push_back(&rs2);
    escape_segs.push_back(&rs3);
    escape_segs.push_back(&rs4);
    infile >> N;
    for( int j=0;j<N;j++ )
    {
      infile >> segs[j].p1.x >> segs[j].p1.y >> segs[j].p2.x >> segs[j].p2.y;
      test_segs.push_back(segs+j);
    }
    infile >> M;
    int sn;
    for( int j=0;j<M;j++ )
    {
      infile >> sn;
      segs[sn-1].nrp++;
    }

    while( escape_segs.size() > 0 )
    {
      next_escape_segs.clear();
      for( deque<seg*>::iterator esc_it=escape_segs.begin();
          esc_it<escape_segs.end();esc_it++ )
      {
        for( deque<seg*>::iterator test_it=test_segs.begin();
            test_it<test_segs.end(); )
        {
          if( seg_isect( **esc_it, **test_it ) )
          {
            next_escape_segs.push_back( *test_it );
            total_red+=(*test_it)->nrp;
            test_it=test_segs.erase(test_it);
          }
          else
          {
            test_it++;
          }
        }
      }
      escape_segs.clear();
      if( next_escape_segs.size() > 0 )
      {
        escape_segs.assign(next_escape_segs.begin(),next_escape_segs.end());
      }
    }

    outfile << total_red << endl;

  }

  return 0;
}
```

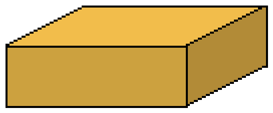
Problem Statement

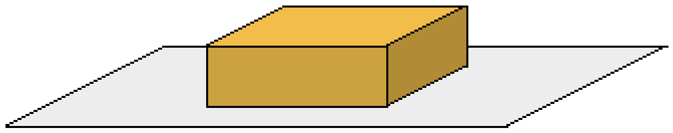
NOTE: for those using plugins to read the problem statement, there is an informative image in the html version of the problem statement, please view it in the normal applet window.

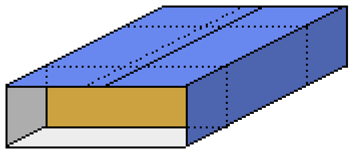
When wrapping your gifts this holiday season, you realize that you are wasting too much money on wrapping paper. To help you reduce the cost, you want to write a program that minimizes the wrapping paper used.

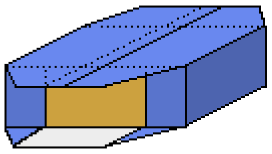
Your program will focus on one gift, which is a 3-dimensional box. The gift has 3 lengths which comprise its dimensions. In order to wrap it, you need to have a piece of paper that is long enough to wrap around the entire box in one direction, and that has enough hanging over the edges to cover the sides of the box. Therefore, there will be three seams, two on the sides and one on the top. In each seam, you want the overlap to be at least 1 inch (that is, at least one inch of paper is covered by another part of the paper).

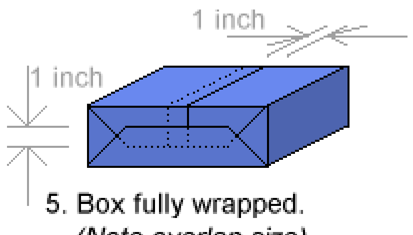
The following image illustrates how the wrapping paper is used to wrap the box. The dotted lines are where edges of paper or the box are obscured by the paper. Pay close attention to where the seams are:


1. Box without paper.


2. Box on top of paper.


3. Box partially wrapped.


4. Sides partially folded.



5. Box fully wrapped.
(Note overlap size)

Create a class Giftwrap that contains the method minSize, which takes 2 arguments:

`dimensions`: a vector <int> which contains the three dimensions of the box, measured in inches.

`paper`: a vector <int> which contains the two dimensions of the paper you have to wrap the box with, measured in inches.

The method should return a vector <int> which contains the two dimensions of the paper you can cut from the given piece that will satisfactorily wrap the box and has the smallest area. If the two dimensions are not the same, the larger dimension should be first in the resulting vector <int>. If two different cuts produce the same area, return the one with the smallest first dimension. If no cut can satisfy the requirements, return {-1, -1}

Definition

Class:	Giftwrap
Method:	minSize
Parameters:	vector <int>, vector <int>
Returns:	vector <int>
Method signature:	vector <int> minSize(vector <int> dimensions, vector <int> paper)

(be sure your method is public)

Notes

- The box does not need to be wrapped in a specific orientation, but the paper must be aligned with the edges of the box. i.e. you cannot wrap the box diagonally.
- You must use only one piece of paper to wrap the box, i.e. you cannot tape multiple pieces together.
- If you can cover the box in a certain wrapping configuration, but there is not a 1 inch overlap at all seams, then you cannot use this configuration to wrap the box.

Constraints

- **`dimensions`** will have exactly 3 elements.
- Each element of **`dimensions`** will be between 5 and 50, inclusive.
- **`paper`** will have exactly 2 elements.
- Each element of **`paper`** will be between 5 and 250, inclusive.

Examples

0)

```
{5,6,7}
{25,25}
Returns: { 23,  13 }
```

There are three ways to wrap this gift, which require 23x13, 25x12, or 27x12 inch paper sheets. The optimal way is with 23x13, and this is accomplished by positioning the box such that the faces that the paper covers last are 5x6. In reference to the diagram above, you can see one of these faces partially in steps 3 and 4.

1)

```
{5,7,6}
{25,12}
Returns: { 25,  12 }
```

Same example, except now the 23x13 cannot be accomplished because it cannot be cut from the paper given (even though the paper has more total area than 23x13).

2)

```
{7,5,6}
{22,22}
```

```
Returns: { -1,  -1 }
```

Even though the paper is 185 square inches larger than the optimal wrapping size, the gift cannot be wrapped because neither dimension will satisfy the minimal length of 23.

3)

```
{5,5,15}
{21,21}
Returns: { 21,  21 }
```

Two possible sizes exist, 21x21 and 41x11.

4)

```
{5,5,15}
{10,50}
Returns: { -1,  -1 }
```

Again, even though the area of the paper will more than cover the area of the box with 1 inch seams in any orientation, the box cannot be wrapped because 10 inches is too short for any orientation.

5)

```
{50,50,50}
{201,101}
Returns: { 201,  101 }
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

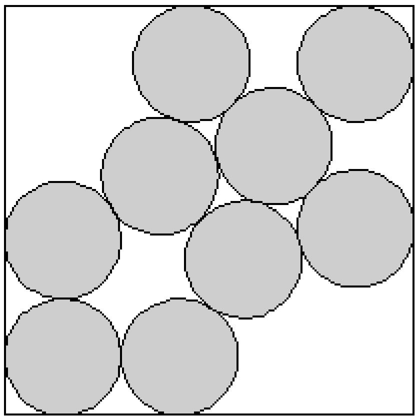
```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct sq
{
  int x,y;
  bool operator<(const sq & lv) const
  {
    if( x*y < lv.x*lv.y ) return true;
    if( x*y == lv.x*lv.y && x < lv.x ) return true;
    return false;
  }
};

sq func( int a, int b, int c )
{
  sq rv; int t;
  rv.x=2*a+2*b+1;
  rv.y=a+c+1;
  if( rv.x < rv.y ) { t=rv.x; rv.x=rv.y; rv.y=t; }
  return rv;
}

class Giftwrap
{
  public:
    vector <int> minSize(vector <int> dimensions, vector <int> paper)
    {
      vector <sq> poss;
      vector <int> ans;

      poss.push_back(func(dimensions[0],dimensions[1],dimensions[2]));
      poss.push_back(func(dimensions[0],dimensions[2],dimensions[1]));
      poss.push_back(func(dimensions[1],dimensions[0],dimensions[2]));
      poss.push_back(func(dimensions[1],dimensions[2],dimensions[0]));
      poss.push_back(func(dimensions[2],dimensions[0],dimensions[1]));
      poss.push_back(func(dimensions[2],dimensions[1],dimensions[0]));
      sort(poss.begin(),poss.end());

      for( int i=0;i<6;i++ )
      {
        if( poss[i].x <= paper[0] && poss[i].y <= paper[1]
         || poss[i].x <= paper[1] && poss[i].y <= paper[0] )
        {
          ans.push_back(poss[i].x);
          ans.push_back(poss[i].y);
          return ans;
        }
      }
      ans.push_back(-1);
      ans.push_back(-1);
      return ans;
    }
};
```

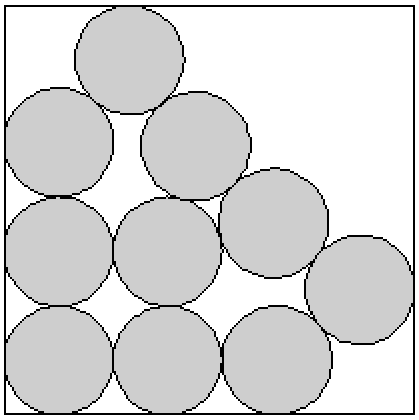
Return of the Aztecs

Problem F: Rigid Circle Packing

Time Limit: 4 seconds
Memory Limit: 32 MB

Aztec kings were very rich and they were proud of their wealth. Once an Aztec king ordered some decoration items to decorate the palace. The decoration items were big cirlces (actually spheres, but here we will consider them to be circles) of glasses. All the circles had the same radius. But it was risky to bring the fragile circles to the palace, because the circles could break easily. The minister of the king suggested that the circles should be packed in the smallest box possible so that they cannot move inside the box. But the king was too proud to do so. He orderd that the boxes should be as large as possible ensuring that the circles won't be able to move inside the boxes, and of course they must be of square-size. So the royal mathematician had a job in his hands, and he seeks your help. Each box can have 9 or 10 circles. See the pictures below.


Box with 9 circles


Box with 10 circles

Input

Input will consist of several test cases. In each test case, there will be a real number r (0 < r < 10000000) denoting the radius of the circles. Input ends with EOF.

Output

For each test case print the length of one side of the box with 9 circles, then a space and then the size of one side of the box with 10 circles, both upto 5-decimal places. A special judge will be used to check your solution. So you need not worry about small precision errors.

Sample Input

```
0.00001
0.00002
```

```
0.00003
```

Sample Output

```
0.00007 0.00008
0.00014 0.00015
0.00021 0.00023
```

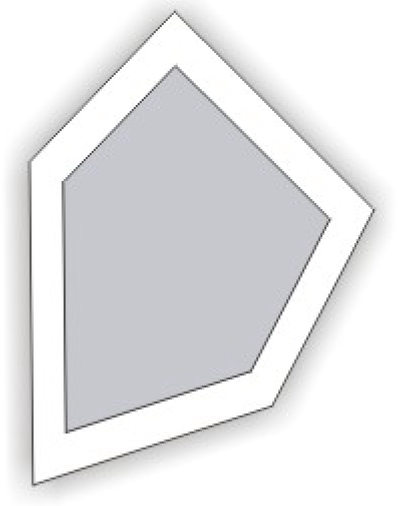
Problem setter: Sadrul Habib Chowdhury (Adil)

See the happy moron,
He doesn't give a damn,
I wish I were a moron,
My God! Perhaps I am!
-- *Anonymous*

```
/* @judge_id: 27516PA 10468 C++ */

/*
 * Valladolid problem 10468 (From Return of the Aztecs contest, F)
 * Rigid Circle Packing
 *
 * Coded by: Andy Martin
 *
 */

#include <iostream>
#include <stdio.h>
#include <math.h>
#include <iomanip>

using namespace std;

#define sqr(x) ((x)*(x))

int main()
{
  double m9=7.05, m10=7.5;
  double l,r,last=1,d,x3;
  double pi=2*acos(0);
  int i;

  // binary search for solution to equations
  l=7.0;
  r=7.1;
  for( i=0;i<250;i++ )
  {
    d=sqr((m9-1-2*sqrt(2)-2*cos(75*pi/180)-1)) + sqr((m9-1-2*sin(75*pi/180)-3));
    if( d > 4 )
    {
      r=m9;
    }
    else if( d < 4 )
    {
      l=m9;
    }
    else
    {
      break;
    }
    m9=(l+r)/2;
  }

  l=7.25;
  r=7.75;
  // just run 250 iterations
  for( i=0;i<250;i++ )
  {
    x3=1+sqrt((8-m10)*(m10-4));
    d=sqr((x3-3-2*cos(75*pi/180))) + sqr((m10-4-2*sin(75*pi/180)));
    if( d > 4 )
    {
      r=m10;
    }
    else if( d < 4 )
    {
      l=m10;
    }
    else
    {
      break;
    }
    m10=(l+r)/2;
  }
```

```
  cin >> r;
  while( cin )
  {
    printf( "%.5lf %.5lf\n",m9*r,m10*r);
    cin >> r;
  }
  return 0;
}
```

Problem D: Cutting tabletops



Bever Lumber hires beavers to cut wood. The company has recently received a shippment of tabletops. Each tabletop is a convex polygon. However, in this hard economic times of cutting costs the company has ordered the tabletops from a not very respectable but cheap supplier. Some of the tabletops have the right shape but they are slightly too big. The beavers have to chomp of a strip of wood of a fixed width from each edge of the tabletop such that they get a tabletop of a similar shape but smaller. Your task is to find the area of the tabletop after beavers are done.

Input consists of a number of cases each presented on a separate line. Each line consists of a sequence of numbers. The first number is **d** the width of the strip of wood to be cut off of each edge of the tabletop in centimeters. The next number **n** is an integer giving the number of vertices of the polygon. The next **n** pairs of numbers present $\mathbf{x}_i$ and $\mathbf{y}_i$ coordinates of polygon vertices for 1 <= i <= **n** given in clockwise order. A line containing only two zeroes terminate the input.

d is much smaller than any of the sides of the polygon. The beavers cut the edges one after another and after each cut the number of vertices of the tabletop is the same.

For each line of input produce one line of output containing one number to three decimal digits in the fraction giving the area of the tabletop after cutting.

Sample input

```
2 4 0 0 0 5 5 5 5 0
1 3 0 0 0 5 5 0
1 3 0 0 3 5.1961524 6 0
3 4 0 -10 -10 0 0 10 10 0
0 0
```

Output for sample input

```
1.000
1.257
2.785
66.294
```

Problem Setter: Piotr Rudnicki

```
/* @judge_id: 27516PA 10406 C++ */

/* judged correct by the online judge */
/* Written by: C. Andy Martin */
/* Solves: Problem D, Cutting Tabletops */

#include <iostream>
#include <string>
#include <vector>
#include <math.h>
#include <stdio.h>

using namespace std;

// distance between 2 points
#define dist(a,b) sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y))

struct pt {double x; double y;};

// simple way to get accurate pi
static const double pi = 2*acos(0);

int mymod( int a, int b )
{
  int rv = a%b;
  if( rv<0 ) rv+=b;
  return rv;
}

// finds the area of a convex polygon represented by clockwize points
double area( vector<pt> pts )
{
  double s,a,b,c,rv=0;
  int i;

  // just split polygon into triangles between pt 0, pt i, and pt i+1
  b=dist(pts[0],pts[1]);
  for(i=1;i<pts.size()-1;i++)
  {
    // use heron's formula for triangle area
    a=b;
    b=dist(pts[0],pts[i+1]);
    c=dist(pts[i],pts[i+1]);
    s=(a+b+c)/2.0;
    rv+=sqrt(s*(s-a)*(s-b)*(s-c));
  }
  return rv;
}

int main()
{
  int n, i;
  double x, y, ang, elev, d;
  pt tp, v1, v2;
  vector <pt> pts, npts;

  cin >> d >> n;
  while( cin && (d || n) )
  {
    pts.clear();
    npts.clear();
    for( i=0; i<n; i++ )
    {
      cin >> tp.x >> tp.y;
      pts.push_back(tp);
    }
    for( i=0;i<n;i++ )
    {
      // shift polygon in by d units
```

```
      // (the shift is always along the angular bisector, so we calculate
      // the elevation of the vector from this point to the previous point
      // and the angle of the bisector at the current vertex using the dot
      // product and then use trig. to find the new x,y coordinates in the
      // trimmed down polygon. We are told in the problem that the new
      // polygon always has the same number of verticies as the original,
      // which gets rid of a special case).
      v1.x=pts[mymod(i-1,n)].x-pts[i].x;
      v1.y=pts[mymod(i-1,n)].y-pts[i].y;
      v2.x=pts[mymod(i+1,n)].x-pts[i].x;
      v2.y=pts[mymod(i+1,n)].y-pts[i].y;
      ang = acos(
(v1.x*v2.x+v1.y*v2.y)/(dist(pts[mymod(i-1,n)],pts[i])*dist(pts[mymod(i+1,n)],pts
[i]))
);
      elev = atan(v1.y/v1.x);
      if( v1.x < 0 ) elev+=pi;
      tp.x=pts[i].x+(d/(sin(ang/2)))*cos(ang/2+elev);
      tp.y=pts[i].y+(d/(sin(ang/2)))*sin(ang/2+elev);
      npts.push_back(tp);
    }

    printf("%.3lf\n",area(npts));
    cin >> d >> n;
  }
  return 0;
}
```

Problem statement originally from: http://www.oi.edu.pl/blue/oi8/kop.html
Translated by Dr. Jaromczyk, retyped by C. Andy Martin.

Gold rush

Let S be a set of ***n*** points in the plane. Find anisothetic rectangle of size ***s*** time ***w*** that contains the largest number of points (including the boundary).

Input

The first row of `kop.in` contains two integers *s* and *w* separated with one space ($1 \leq s, w \leq 10\,000$), denoting the width and height of the rectangle. In the second row there is a positive *n* ($1 \leq n \leq 15\,000$), the number of points. In the remaining *n* rows there are integer coordinates of the points. Each row has two numbers *x* and *y* ($-30\,000 \leq x, y \leq 30\,000$), separated with one space, denoting *x* and *y* coordinates.

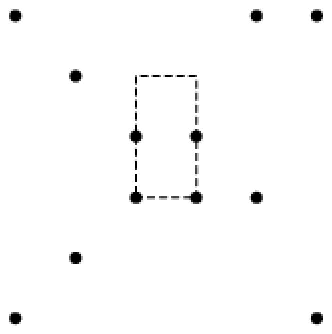
Output

Output `kop.out` contains one number: the maximal number of points of S in the optimal rectangle.

Example:

`kop.in`:

```
1 2
 12
0 0
1 1
2 2
3 3
4 5
5 5
4 2
1 4
0 5
5 0
2 3
3 2
```



kop.out:

```
4
```

```
// solved by Ryan Gabbard
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

struct pt {
  int x, y;
};

class comp {
public:
  bool operator()(pt a, pt b) const {
    if (a.x<b.x)
      return true;
    else if (a.x==b.x)
      return (a.y<b.y);
    return false;
  }
};

int main() {
  int s, w, n;
  vector<pt > points;
  vector<pt > sorted;
  pt tmp;

  cin >> w >> s >> n;

  for (int i=0; i<n; i++) {
    cin >> tmp.x >> tmp.y;
    points.push_back(tmp);
  }

  sorted=points;

  sort(sorted.begin(), sorted.end(), comp());

  int left_x, bottom_y, right_x, top_y;

  int max=0, count=0, start;

  for (int i=0; i<n; i++) {
    left_x=points[i].x, right_x=points[i].x+w;
    bottom_y=points[i].y, top_y=points[i].y+s;

    count=0;

    for (int j=0; j<n; j++) {
      if (sorted[j].x>=left_x) {
        start=-1;

        for (start=j; (start<n) && (sorted[start].x<=right_x) && (sorted[start].
y<bottom_y); start++);

        for (count=start; (count<n) && (sorted[count].x<=right_x) && (sorted[cou
nt].y<=top_y); count++);

        count-=start;

        break;
      }

    }
    if (count>max)
      max=count;
```

```
  }



  cout << max << endl;


  return 0;
}
```

The 2003 27th Annual **acm** International Collegiate

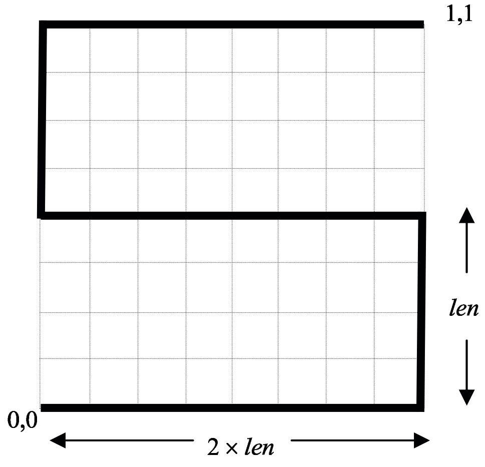
Programming Contest World Finals

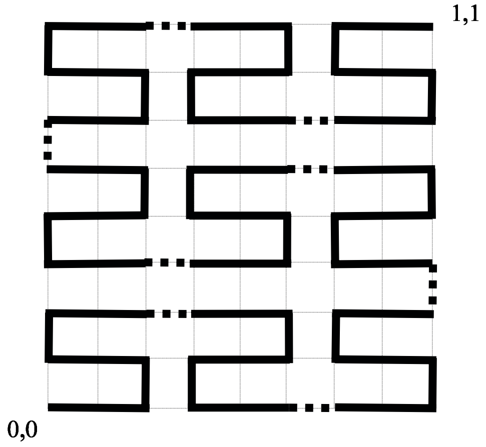
sponsored by IBM

Problem C
Riding the Bus
Input File: bus.in

The latest research in reconfigurable multiprocessor chips focuses on the use of a single bus that winds around the chip. Processor components, which can be anywhere on the chip, are attached to *connecting points* on the bus so that they can communicate with each other.

Some research involves bus layout that uses recursively-defined "SZ" curves, also known as "S-shaped Peano curves." Two examples of these curves are shown below. Each curve is drawn on the unit square. The order-1 curve, shown on the left, approximates the letter "S" and consists of line segments connecting the points (0,0), (1,0), (1,0.5), (0,0.5), (0,1), and (1,1) in order. Each horizontal line in an "S" or "Z" curve is twice as long as each vertical line. For the order-1 curve, the length of a vertical line, *len*, is 0.5.





The order-2 curve, shown on the right, contains 9 smaller copies of the order-1 curve (4 of which are reversed left to right to yield "Z" curves). These copies are connected by line segments of length *len*, shown as dotted lines. Since the width and height of the order-2 curve is 8 × *len*, and the curve is drawn on the unit square, *len* = 0.125 for the order-2 curve.

The order-3 curve contains 9 smaller copies of the order-2 curve (with 4 reversed left to right), connected by line segments, as described for the order-2 curve. Higher order curves are drawn in a similar manner. The *connecting points* to which processor components attach are evenly spaced every *len* units along the bus. The first connecting point is at (0,0) and the last is at (1,1). There are 9^k connecting points along the order-*k* curve, and the total bus length is $(9^k–1) \times len$ units.

You must write a program to determine the total distance that signals must travel between two processor components. Each component's coordinates are given as an *x*, *y* pair, $0 \le x \le 1$ and $0 \le y \le 1$, where *x* is the distance from the left side of the chip, and *y* is the distance from the lower edge of the chip. Each component is attached to the closest connecting point by a straight line. If multiple connecting points are equidistant from a component, the one with the smallest *x* coordinate and smallest *y* coordinate is used. The total distance a signal must travel between two components is the sum of the length of the lines connecting the components to the bus, and the length of the bus between the two connecting points. For example, the distance between components located at (0.5, 0.25) and (1.0, 0.875) on a chip using the order-1 curve is 3.8750 units.

Input

The input contains several cases. For each case, the input consists of an integer that gives the order of the SZ curve used as the bus (no larger than 8), and then four real numbers x_1, y_1, x_2, y_2 that give the coordinates of the processor components to be connected. While each processor component should actually be in a unique location not on the bus, your program must correctly handle all possible locations.

The last case in the input is followed by a single zero.

Output

For each case, display the case number (starting with 1 for the first case) and the distance between the processor components when they are connected as described. Display the distance with 4 digits to the right of the decimal point.

Use the same format as that shown in the sample output shown below. Leave a blank line between the output lines for consecutive cases.

Sample Input

```
1 0.5 .25 1 .875
1 0 0 1 1
2 .3 .3 .7 .7
2 0 0 1 1
0
```

Output for the Sample Input

```
Case 1.  Distance is 3.8750

Case 2.  Distance is 4.0000

Case 3.  Distance is 8.1414

Case 4.  Distance is 10.0000
```

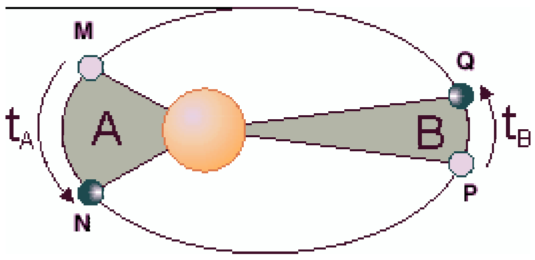
```
/* Problem C of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */
#include <fstream>
#include <cstdio>
#include <cmath>

using namespace std;

const double tol = 1e-8;

unsigned long mypow( unsigned long a, unsigned long b )
{
  unsigned long c=1;
  while( b )
  {
    c*=a;
    b--;
  }
  return c;
}

class point
{
 public:
  double x, y;
  point( void ) { x=0.0; y=0.0; }
  point( double nx, double ny ) { x=nx; y=ny; }
  point( double n[2] ) { x=n[0]; y=n[1]; }
  double distanceTo( const point & p ) const
  {
    return sqrt( (x-p.x)*(x-p.x) + (y-p.y)*(y-p.y) );
  }
  bool operator==( const point & r ) const
  {
    return ( fabs( r.x-x ) < tol && fabs( r.y-y ) < tol );
  }
};

const int novant_map[3][3] = { { 0, 1, 2 },
                               { 5, 4, 3 },
                               { 6, 7, 8 } };

// the length of the smallest unit in each order k curve. define len[0] to
// be one. the total length of an order k curve is then just 9^k - 1 times
// the length of the smallest unit in each order k curve.
long len[9];

// find distance between points on an order k Peano curve
// we represent the curve differently than the problem. we scale the upper
// right corner by 1 / len (what we call len). So the return value is the
// number of minimal length segments. To get the actual length, divide the
// answer by len[k] (don't forget to cast to double).
unsigned long dist( unsigned long x1, unsigned long y1, unsigned long x2, unsign
ed long y2, int k )
{
  unsigned long novant1, novant2;
//  printf( " p1: (%ld,%ld) p2: (%ld,%ld)\n", x1,y2,x2,y2 );
  if( x1 == x2 && y1 == y2 ) return 0;
  unsigned long xndx1 = (3*x1)/len[k];
  if( xndx1 == 3 ) xndx1=2;
  unsigned long yndx1 = (3*y1)/len[k];
  if( yndx1 == 3 ) yndx1=2;
  unsigned long xndx2 = (3*x2)/len[k];
  if( xndx2 == 3 ) xndx2=2;
  unsigned long yndx2 = (3*y2)/len[k];
  if( yndx2 == 3 ) yndx2=2;
  novant1 = novant_map[yndx1][xndx1];
  novant2 = novant_map[yndx2][xndx2];
```

```
  if( novant1 > novant2 )
  {
    swap<unsigned long>(x1,x2);
    swap<unsigned long>(y1,y2);
    swap<unsigned long>(novant1,novant2);
  }
  if( x1 == 0 && y1 == 0 && x2 == len[k] && y2 == len[k] )
  {
    return mypow(9,k)-1;
  }
//  printf( " novant1: %d novant2: %d\n", novant1, novant2 );
  if( k == 1 )
  {
    return novant2-novant1;
  }
  // map coordinates to new values
  x1 -= xndx1*(len[k-1]+1);
  y1 -= yndx1*(len[k-1]+1);
  x2 -= xndx2*(len[k-1]+1);
  y2 -= yndx2*(len[k-1]+1);
  unsigned long x1a = len[k-1];
  unsigned long y1a = len[k-1];
  unsigned long x2a = 0;
  unsigned long y2a = 0;
  if( novant1 & 0x1 )
  {
    x1 = len[k-1]-x1;
    y1 = len[k-1]-y1;
    x1a = len[k-1]-x1a;
  }
  if( novant2 & 0x1 )
  {
    x2 = len[k-1]-x2;
    y2 = len[k-1]-y2;
    x2a = len[k-1]-x2a;
  }
  if( novant1 == novant2 )
  {
    return dist(x1,y1,x2,y2,k-1);
  }
  // add up the length of the novants in between, the connecting lengths and
  // the lengths of the points on the novants to the end of the curve
  return (dist(x1,y1,x1a,y1a,k-1) + dist(x2a,y2a,x2,y2,k-1) +
    (novant2-novant1-1)*mypow(9,(k-1)) + 1);
}

unsigned long nearest_int( double x )
{
  unsigned long r = (unsigned long)x;
  if( x-r <= (r+1)-x )
  {
//    printf( "nearest int to %.10lf is %d\n", x, r );
    return r;
  }
//  printf( "nearest int to %.10lf is %d\n", x, r+1 );
  return r+1;
}

int main()
{
  ifstream infile( "bus.in" );
  point p1, p2, comp1, comp2;
  unsigned long nx1,nx2,ny1,ny2;
  int k, cnt=0;

  len[0]=1;
  len[1]=2;
//  printf( "side of order %d is %d\n", 1, 2 );
  for( k=2;k<9;k++ )
  {
    len[k] = 3*len[k-1]+2;
//    printf( "side of order %d is %d\n", k, len[k] );
  }

  if( !infile ) { fprintf(stderr,"File problems.\n"); return 1; }

  infile >> k;
  while( k )
  {
    infile >> p1.x >> p1.y >> p2.x >> p2.y;

    if( p1 == p2 )
    {
      printf( "Case %d.  Distance is %.4lf\n", ++cnt, 0.0 );
    }
    else
    {
      nx1 = nearest_int(p1.x*len[k]);
      ny1 = nearest_int(p1.y*len[k]);
      nx2 = nearest_int(p2.x*len[k]);
      ny2 = nearest_int(p2.y*len[k]);
      comp1.x = ((double)nx1)/len[k];
      comp1.y = ((double)ny1)/len[k];
      comp2.x = ((double)nx2)/len[k];
      comp2.y = ((double)ny2)/len[k];

      printf( "Case %d.  Distance is %.4lf\n", ++cnt,
          ((double)dist(nx1,ny1,nx2,ny2,k))/len[k] +
          comp1.distanceTo(p1) + comp2.distanceTo(p2) );
    }

    infile >> k;
    if( k ) printf("\n");
  }

  return 0;
}
```

Problem I
The Solar System
Input File: solar.in

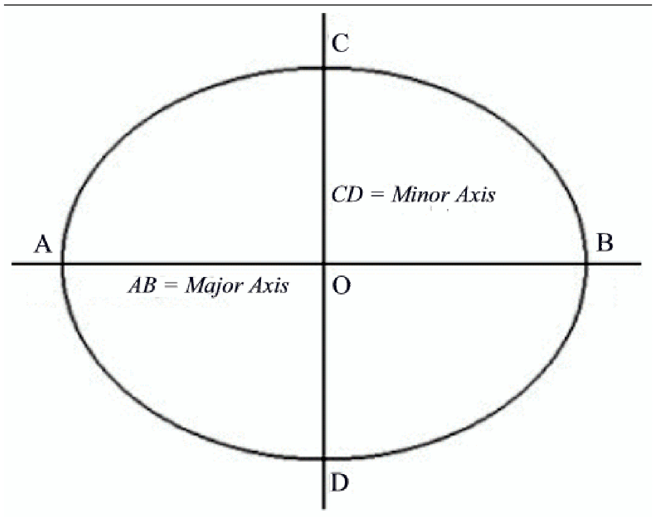
It is common knowledge that the Solar System consists of the sun at its center and nine planets moving around the sun on elliptical orbits. Less well known is the fact that the planets' orbits are not at all arbitrary. In fact, the orbits obey three laws discovered by Johannes Kepler. These laws, also called "The Laws of Planetary Motion," are the following.

1. The orbits of the planets are ellipses, with the sun at one focus of the ellipse. (Recall that the two foci of an ellipse are such that the sum of the distances to them is the same for all points on the ellipse.)
2. The line joining a planet to the sun sweeps over equal areas during equal time intervals as the planet travels around the ellipse.
3. The ratio of the squares of the revolutionary periods of two planets is equal to the ratio of the cubes of their semi major axes.



By Kepler's first law, the path of the planet shown in the figure on the left is an ellipse. According to Kepler's second law, if the planet goes from M to N in time t_A and from P to Q in time t_B and if $t_A = t_B$, then area A equals area B. Kepler's third law is illustrated next.

Consider an ellipse whose center is at the origin O and that is symmetric with respect to the two coordinate axes. The x-axis intersects the ellipse at points A and B and the y-axis intersects the ellipse at points C and D. Set $a=½|AB|$ and $b=½|CD|$. Then the ellipse is defined by the equation $x^2/a^2+y^2/b^2=1$. If $a \geq b$, AB is called the **major axis**, CD the **minor axis**, and OA (with length a) is called the **semi major axis**. When two planets are revolving around the sun in times t_1 and t_2 respectively, and the semi major axes of their orbits have lengths a_1 and a_2, then according to Kepler's third law $(t_1/t_2)^2 = (a_1/a_2)^3$.



In this problem, you are to compute the location of a planet using Kepler's laws. You are given the description of one planet in the Solar System (i.e., the length of its semi-major axis, semi-minor axis, and its revolution time) and the description of a second planet (its semi-major axis and semi-minor axis). Assume that the second planet's orbit is aligned with the coordinate axes (as in the above figure), that it moves in counter clockwise direction, and that the sun is located at the focal point with non-negative x-coordinate. You are to compute the position of the second planet a specified amount of time after it starts at the point with maximal x-coordinate on its orbit (point B in the above figure).

Input
The input file contains several descriptions of pairs of planets. Each line contains six integers a_1, b_1, t_1, a_2, b_2, t. The first five integers are positive, and describe two planets as follows:

a_1 = semi major axis of the first planet's orbit
b_1 = semi minor axis of the first planet's orbit

t_1 = period of revolution of the first planet (in days)
a_2 = semi major axis of the second planet's orbit
b_2 = semi minor axis of the second planet's orbit

The non-negative integer t is the time (in days) at which you have to determine the position of the second planet, assuming that the planet starts in position (a_2,0).

The last description is followed by a line containing six zeros.

Output

For each pair of planets described in the input, produce one line of output. For each line, print the number of the test case. Then print the x- and y-coordinates of the position of the second planet after t days. These values must be exact to three digits to the right of the decimal point. Follow the format of the sample output provided below.

Sample Input

```
10 5 10 10 5 10
10 5 10 20 10 10
0 0 0 0 0 0
```

Output for the Sample Input

```
Solar System 1: 10.000 0.000
Solar System 2: -17.525 4.819
```

```
/* Problem I of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */

/* Use numerical integration and a binary search method to find the solution
 * to the area of the ellipse which yields the right time in orbit. Some
 * useful equations were (from http://mathworld.wolfram.com/ on ellipse):
 * a - major axis
 * b - minor axis
 * c - focus distance from center
 * e - eccentricity
 *
 * (1) c = sqrt( a^2 - b^2 ) = a*e
 *
 * equations for ellipse with r and theta defined from a focus:
 * (2) x = c + r*cos(theta)
 * (3) y = r*sin(theta)
 * (4) r = a*(1-e^2)/(1+e*cos(theta))
 *
 * And from the problem statement we know:
 * dA/dtheta is proportional to dt/dtheta
 * The constant is, of course:
 * (5) K*dA/dtheta = dt/dtheta
 * (6) K = T/pi*a*b
 * by solving (5) at theta = 2*pi. (the area is pi*a*b, and the
 * time is the period of orbit, T).
 *
 * Also,
 * (7) dA = r^2*dtheta/2
 * So,
 * (8) A = integral(a,b,r^2*dtheta/2)
 * where r is given by (4). We then do a binary search on theta over the range
 * [0,2*pi) looking for the value which corresponds to the desired orbit time
 * passed in (modulo the orbit period since we only want the position in the
 * orbit). We perform the integration using an adaptive simpson scheme which
 * gives us the desired error. We then find x and y using (2) and (3).
 */
#include <fstream>
#include <stdio.h>
#include <math.h>

using namespace std;

#define EPSILON 1e-4

#define f dAdtheta
double dAdtheta( double theta );
#define level_max 10

// Numerical integration using Adaptive Simpson Procedure to calculate an
// integral efficiently to a desired accuracy. The other viable altertnative
// which may be faster in time complexity but greater in memory complexity
// is the Romberg Algorithm. This algorithm is lifted directly from
// pseudocode found on page 226 of the Fourth Edition of Numerical
// Mathematics and Computing by Ward Cheney and David Kincaid (c) 1999 by
// Brooks/Cole Publishing Company. The only modification to the psuedocode
// is to explicitly restrict the recursion level. We will use the implicit
// restriction of the function call stack size to limit the recursive depth.
// Also, the code is condensed (variables are elimanted so the call stack
// doesn't grow so big)
double simpson( double a, double b, double epsilon, int level=0 )
{
#define one_simpson ((b-a)*(f(a)+4*f((a+b)/2)+f(b))/6)
#define two_simpson ((b-a)*(f(a)+4*f((a+(a+b)/2)/2)+2*f((a+b)/2)+4*f(((a+b)/2+b)
/2)+f(b))/12)
  if( level == level_max )
  {
    return two_simpson;
  }
```

```
  else if( fabs(two_simpson-one_simpson) < 15*epsilon )
  {
    return 16*two_simpson/15-one_simpson/15;
  }
  else
  {
    return simpson(a,(a+b)/2,epsilon/2,level+1) + simpson((a+b)/2,b,epsilon/2,le
vel+1);
  }
}

// define these before calling dAdtheta
static double a,e;
// the derivative function of area versus theta - we integrate this to
// obtain the area over a definate period using a numerical method
double dAdtheta( double theta )
{
  double r = a*(1-e*e)/(1+e*cos(theta));
  return r*r/2;
}

int main()
{
  int cnt=0;
  double a1, b1, t1, a2, b2, t;
  double t2, c, x, y, theta, left, right, r, t_guess;
  double area, left_area;
  ifstream infile( "solar.in" );
  const double pi = 2*acos(0);

  if( !infile ) { fprintf( stderr, "File problems.\n" ); return 1; }

  infile >> a1 >> b1 >> t1 >> a2 >> b2 >> t;
  while( a1 )
  {
    a = a2;
    e = sqrt( 1 - b2/a2*b2/a2 );
    c = a*e;
    t2 = sqrt(t1/a1*t1/a1*a2/a1*a2*a2);
    t = fmod( t, t2 );
    left = 0;
    right = 2*pi;
    theta = (left+right)/2;
    t_guess = t2*10;
    left_area=0;
    while( fabs(left-right) > 1e-15 )
    {
      area = simpson(left,theta,EPSILON);
      t_guess = (left_area+area)*t2/(pi*a2*b2);
//      printf( "right:%f theta:%f left:%f t:%f t_guess:%f\n",right, theta, left
, t, t_guess );
      // we are finding t by finding the area and multiplying by the
      // constant scaling factor (we are given time is proportional to area,
      // and since one orbit takes one elipse area, we know the time)
      if( t_guess < t )
      {
        left = theta;
        left_area+=area;
      }
      else
      {
        right = theta;
      }
      theta = (left+right)/2;
    }

    r = a*(1-e*e)/(1+e*cos(theta));
    x = c+r*cos(theta);
    y = r*sin(theta);
//    printf( "e:%f t2:%f theta:%f r:%f x:%f y:%f\n",e,t2,theta,r,x,y);
    if( x < 0 && x > -.0005 ) x = 0;
    if( y < 0 && y > -.0005 ) y = 0;
    printf( "Solar System %d: %.3f %.3f\n", ++cnt, x, y );
    infile >> a1 >> b1 >> t1 >> a2 >> b2 >> t;
  }

  return 0;
}
```

Problem E
Covering Whole Holes
Input File: holes.in

Can you cover a round hole with a square cover? You can, as long as the square cover is big enough. It obviously will not be an exact fit, but it is still possible to cover the hole completely.

The Association of Cover Manufacturers (ACM) is a group of companies that produce covers for all kinds of holes — manholes, holes on streets, wells, ditches, cave entrances, holes in backyards dug by dogs to bury bones, to name only a few. ACM wants a program that determines whether a given cover can be used to completely cover a specified hole. At this time, they are interested only in covers and holes that are rectangular polygons (that is, polygons with interior angles of only 90 or 270 degrees). Moreover, both cover and hole are aligned along the same coordinate axes, and are not supposed to be rotated against each other — just translated relative to each other.

Input
The input consists of several descriptions of covers and holes. The first line of each description contains two integers h and c ($4 \leq h \leq 50$ and $4 \leq c \leq 50$), the number of points of the polygon describing the hole and the cover respectively. Each of the following h lines contains two integers x and y, which are the vertices of the hole's polygon in the order they would be visited in a trip around the polygon. The next c lines give a corresponding description of the cover. Both polygons are rectangular, and the sides of the polygons are aligned with the coordinate axes. The polygons have positive area and do not intersect themselves.

The last description is followed by a line containing two zeros.

Output
For each problem description, print its number in the sequence of descriptions. If the hole can be completely covered by moving the cover (without rotating it), print "Yes" otherwise print "No". Recall that the cover may extend beyond the boundaries of the hole as long as no part of the hole is uncovered. Follow the output format in the example given below.

Sample Input	Output for the Sample Input

```
4 4
0 0
0 10
10 10
10 0
0 0
0 20
20 20
20 0
4 6
0 0
0 10
10 10
10 0
0 0
0 10
10 10
10 1
9 1
9 0
0 0
```
	```
	Hole 1: Yes
	Hole 2: No
	```

About problem E, I think I have it figured out (for Dr. J., this is the problem with a cover and a hole polygon. The polygons may be concave, but not self-intersecting or degenerate, and all interior angles must be either 90 or 270 degrees. The polygons are aligned on the coordinates axes, have integer x and y values, and for the problem they can not be rotated, but the may be translated. The integers have no bounds given, so I assume that they fit into normal C++ int's (I also assume they may be negative) but other than that there is no other restriction on their size (so beware of overflow/underflow with translations). Also, the polygons have between 4 and 50 vertices (the number of vertices has to be even because of the restrictions on polygon shape). The problem is: given polygon C does it cover polygon H? (translations allowed, polygons given to above restrictions)):

Polygon C is the cover, H is the hole

First, test the areas. If A(C) < A(H) the cover won't cover the hole (we can actually skip this test, because the next one is faster, easier to compute, and will catch many of the same cases as this one will

Next find the bounding rectangles of the polygons (min x value, min y value, max x value, max y value). If the bounding rectangle of H doesn't fit into C than the cover won't cover the hole.

Ok, now that we have ruled out easy test cases, next comes the more grueling computation:

Next make a list of all the possible X values of the vertices of C. Make a list of all possible Y vertices of C. Make a set of test points which is all possible combinations of these lists. (At most 25*25=625 because there are up to 50 vertices, and because of the nature of these polygons every two vertices can contribute up to one unique x and y value, making up to 25 unique x and y values). Try putting the hole into the cover at each of these points in four different ways, once with the upper left corner of the bounding box of the hole on the point, once with the upper right, once with the lower left, once with the lower right. Then at each orientation test for polygon inclusion. This will make at most 25*25*4 (2500) tests. Passing the test of course means the polygon fits. Failing all the tests means there is no way to place the hole in the cover such that two of the sides of the cover touch two of the sides of the hole (this should be enough to prove that the cover can not cover the hole). I think the fastest way to do the polygon inclusion test will be a scan line algorithm. Find the critical lines to check both vertically and horizontally. When a point on the cover polygon is reached, remember you are in the cover. Then, if a point on the hole is encountered outside of the cover we know that the hole is not inside the cover. Since we test in both dimensions, we don't need to worry about some of the bizarre polygon shapes, because in one of the dimensions one of the lines will be outside of the cover. The testing points for the sweep can be the set of all the x and y values for both the cover and the hole (although, I think you may be able to just do the hole, since we are interested mainly in it). There may be a few cases I have not thought through, but I think these are the key things to notice. I think each test is at worst 50*50*2 (all x's, all y's, two dimensions). This means we can run processing on 25*25*4*50*50*2 points which is 12.5 million points. This should take on the order of 10 to 50 ms on a 1GHz machine. Assuming 50 ms, we could run 20 test cases a second, which should be enough for the contest.

Problem D: Basic

The programming language Ada has integer constants that look like this: 123, 8#123#, 16#abc#. These constants represent the integers 123, 83 (123 base 8) and 2739 (abc base 16). More precisely, an integer may be a decimal integer given as a sequence of one or more digits less than 10, or it may be an integer to some specific base, given as the base followed by a sequence of one or more digits less than the base enclosed by # symbols. Lower case letters from a through f are used as the digits representing 10 through 15. In Ada, the base, if specified, must be a sequence of decimal digits. For this problem, however, the base may be of any form described above so long as it represents an integer between 2 and 16 inclusive.

The first line of input contains a positive integer *n*. *n* lines follow. For each line of input, output a line "yes" if it is a valid integer constant according to the above rules; otherwise output a line containing "no". Input lines contain no spaces and are between 1 and 80 characters in length.

Sample Input

```
5
2#101#
2#101##123#
17#abc#
16#123456789abcdef#
16#123456789abcdef#123456789abcdef#
```

Output for Sample Input

```
yes
yes
no
yes
no
```

```
/* @JUDGE_ID: 27516PA 10442 C++ */

/* solution to problem D */
/* for the Valladolid practice contest on saturday 1/25/03
 * (actually this a Waterloo local contest) */

#include <iostream>
#include <fstream>
#include <math.h>
#include <string.h>
#include <stdlib.h>

using namespace std;

/* at one point, while I was debugging, I suspected that strtok() did
not work
 * as I had anticpated. So I made asttok (andy string tokenizer), which is
 * really a misnomer. This function has two 'modes' like strtok(). If
you pass
 * in NULL, it uses the last pointer value it remembered and continues
 * processing that string. If you pass in a valid pointer, it will
process that
 * string. The processing just consists of finding the first '#' mark and
 * changing it to a '\0' and remembering the pointer right after the
null. It
 * then returns the pointer to the newly termnated section. This should
be the
 * behavior of strtok( s, "#" ). */
char * asttok( char *s )
{
  static char *o;
  if( s ) o=s;
  else s=o;
  while( *o!=0 )
  {
    /* yes, *o++ is right, we want to store 0 at o, and have o be
incremented
     * one for the next time asttok(NULL) is called */
    if( *o == '#' ) { *o++=0; return s; }
    o++;
  }
  /* so we return NULL if no '#' was incountered before a '\0' - this is the
   * behavior the rest of the program expects */
  return NULL;
}

/* this simply takes a charcter in the regex [0-9a-f] and turns it into a
 * number in the range [0,15], or reports -1 on error */
int tonum( char c )
{
  /* CAUTION:
   * make sure these are CHARACTER constants and not integer constants! */
  if( c >= '0' && c <= '9' )
  {
    return c-'0';
  }
  if( c >= 'a' && c <= 'f' )
  {
    /* we want c-'a'+10, because this gives us when c=='a', return
'a'-'a'+10
     * which is 10, which is what we want */
    return c-'a'+10;
  }
  return -1;
}

/* this will check to see if the parameter st contains a valid decimal
number */
bool checval( char * st )
```

```
{
  for(int i=0;i<strlen(st);i++ )
  {
    if( st[i] > '9' || st[i] < '0' ) return false;
  }
  return true;
}

/* this function takes the base, represented as an integer, and processes
 * the string num to make sure it is a valid number of base 'base'. The
 * function returns -1 if num is not a valid number, or 0,1 or 17 if the
number
 * is valid, but would make an invalid base for the next number (i.e.,
too low
 * or too high), or it returns the base the number would represent in
[2,16] */
int calcnum( int base, char * num )
{
  int sl = strlen( num );
  int i, c, m=1, rv=0, dig;

  /* we must check for things like '###' by testing the string length */
  if( sl < 1 ) return -1;
  if( base > 16 || base < 2 ) return -1;

  /* don't forget '>=' instead of '>' so we test the last digit! */
  for( i=sl-1;i>=0;i-- )
  {
    /* first make this digit a number */
    dig=tonum(num[i]);
    /* if the digit is invalid, the number is invalid */
    if( dig < 0 ) return -1;
    if( dig >= base ) return -1;
    /* if our return value is still below the maximum, continue to
update the
     * base. CAUTION: this was the trickiest part to get right. There
are some
     * nasty inputs, like 100000002 which cause the multiplier m to roll
over
     * into a valid range (or even, as in the case above, make it 0!) */
    if( rv <= 16 )
    {
      // don't bother updating number for too-big bases
      if( m > 16 && dig > 0 ) rv=17;
      else rv+=dig*m;
    }
    /* don't bother updating multipliers for too-big bases (we just care
     * that the base is too big, not what its exact decimal represntation
     * is) */
    if( m <= 16 )
    {
      m*=base;
    }
  }

  // make all the 'too big base' numbers be exactly 17 (up to this point, I
  // think they can fall in the range [17,1024) )
  if( rv > 16 ) rv=17;

  return rv;
}

/* this checks one line of the file and returns true if this is a valid
 * representation of a number and false if it is not
 */
bool docase( char *buf )
{
  char *base;
  int b;
  char *num;
  char *ll;
  // first get base string
  base = asttok( buf );
  if( base == NULL )
  {
    // if the base is NULL, we should have just a plain base 10 number
    return checval(buf);
  }
  // must be base 10
  if( !checval(base) ) return false;
  // convert to an integer
  // (note, a range check is done when we pass this into calcnum())
  b=atoi(base);
  // ll is a 'lookahead' pointer. we use it to see if there is more to this
  // number.
  ll=base+strlen(base)+1;
  // this means there is an extra '#' at the beginning
  if( *ll == '#' ) return false;
  num = asttok( NULL );
  // at this point, if num is NULL, the string was badly formated (the
  // numbers are encased in '#')
  if( !num ) return false;
  // loop and do each consecutive number until we run out of characters
  for(;;)
  {
    b=calcnum(b,num);
    // note: not-good--for-base (b>16, 0<=b<2) is OK unless we have more
    // number, so only check right now if the number was invalid (b<0)
    if( b<0 ) return false;
    ll=num+strlen(num)+1;
    // if the lookahead is a null, we are done
    if( *ll == '\0' ) return true;
    // if the lookahead wasn't a null nor a '#' this is invalid
    if( *ll != '#' ) return false;
    // if the lookahead+1 IS a '#' we have too many '#'s
    if( *(ll+1) == '#' ) return false;
    // chew up the empty part inbetween ##
    asttok( NULL );
    // get num as a string
    num = asttok( NULL );
    // again, a NULL means it wasn't encased in '#'s
    if( !num ) return false;
    // we don't have to check the base, because the next calcnum() will do
    // this for us
  }
}

// just go line by line calling docase() until there is no more to do
int main()
{
  int n;
  int i;
  char buf[255];
  cin >> n;
  for( i=0;i<n;i++ )
  {
    cin >> buf;
    if( docase(buf) )
    {
      cout << "yes" << endl;
    }
    else
    {
      cout << "no" << endl;
    }
  }
  return 0;
}
```

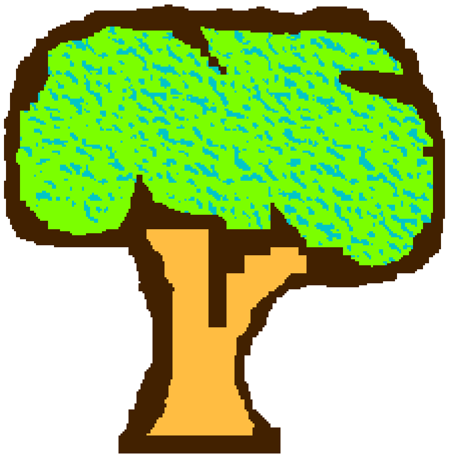
Return of the Aztecs

Problem E: Parse Tree

Time Limit: 2 seconds
Memory Limit: 32 MB

Given an expression and some rules, a corresponding parse-tree can be drawn. If the leaves of the trees are traversed from left to right, then the expression from which the tree was generated can be found. The leaves contain the terminal symbols, and the internal nodes contain the non-terminal symbols. For this problem, terminals consist of **{i, +, -, *, /, (,)}** and the non-terminals are **{E, T, F}**. The rules for this specific problem are:

- E -> E + T
- E -> E - T
- E -> T
- T -> T * F
- T -> T / F
- T -> F
- F -> i
- F -> (E)


This is NOT a parse tree

Input

Input consists of several test cases. Each case is in a line by itself. Each line contains a non-empty expression which doesn't have any blank space in it. There will be no invalid and ambiguous input, i.e., there will always be a unique parse-tree for the input. Input is terminated by EOF.

Output

For each test case, print the parse- tree. The leftmost leaf should be at column 1, the leaf next to that should be at column 4, the next leaf should be at column 7, and so on. Each non-terminal should have a '|' on the same column in the following line and a non- terminal in the next line on the same column. A string of '=' should spread on both sides of '|' just enough to cover its immediate children. There should be no blank line within a parse-tree. Print a single blank line between test cases. No line should have blank spaces at the end. You can safely assume that a single parse tree won't need more than 200 lines to be drawn and no line will need more than 200 characters. Look at the sample outputs for clearer idea.

Sample Input

```
i+i*i
i+i*(i+i)
```

Sample Output

```
   E
===|======
E  +     T
|     ===|===
T     T  *  F
|     |     |
F     F     i
|     |
i     i

   E
===|======
E  +     T
|     ===|=========
T     T  *        F
|     |     ======|======
F     F     (     E     )
|     |        ===|===
i     i        E  +  T
               |     |
               T     F
               |     |
               F     i
               |
               i
```

Problem setter: Sadrul Habib Chowdhury (Adil)

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
-- *Sherlock Holmes*

```
/* @judge_id: 27516PA 10467 C++ */

/*
 * Valladolid problem 10467 (From Return of the Aztecs contest, E)
 * Parse Tree
 *
 * Coded by: Andy Martin
 *
 */

#include <iostream>
#include <string>
#include <vector>

using namespace std;

/* find leftmost nonterminal of {c1,c2} not enclosed in parenthesis. this is
 * a fancy lookahead so we can actually parse the grammar the way it is
 * presented in the problem */
int findpos( string s, char c1, char c2 )
{
  int pc=0;
  for( int i=s.size()−1; i>=0; i−− )
  {
    if( s[i] == ’)’ ) pc++;
    else if( s[i] == ’(’ ) pc−−;
    else if( (s[i] == c1 || s[i] == c2) && pc==0 )
    {
      return i;
    }
  }
  return −1;
}

/* concatenates a child branch with a parent non terminal formatted with
 * the correct formating for the problem statement. We use a couple clever
 * STL tricks, we use find_first_not_of to get the position of the head
 * of the child tree, and assign() to fill a string with spaces */
vector <string> unary_branch_concat( vector <string> l, char non_terminal )
{
  vector <string> rv;
  string str;
  int s=l[0].size();
  int head=l[0].find_first_not_of(" ");
  str.assign(s,’ ’); str[head]=non_terminal; rv.push_back( str );
  str[head]=’|’; rv.push_back( str );
  for( unsigned int i=0;i<l.size();i++ )
  {
    rv.push_back(l[i]);
  }
  return rv;
}

/* similar to above, but using two children, a right and a left */
vector <string> binary_branch_concat( vector <string> l, vector <string> r, char
 op, char non_terminal )
{
  vector <string> rv,mid;
  string str;
  int op_pos,l_head,r_head,s;
  unsigned int i;

  op_pos=l[0].size()+2;
  s=op_pos+3+r[0].size();
  l_head=l[0].find_first_not_of(" ");
  r_head=op_pos+3+r[0].find_first_not_of(" ");
  str.assign(s,’ ’); str[op_pos]=non_terminal; rv.push_back( str );
  str.erase(); str.append(l_head,’ ’); str.append(r_head−l_head+1,’=’);
  str.append(s−r_head−1,’ ’); str[op_pos]=’|’; rv.push_back( str );
```

```
  // first put together bottom of tree
  str.assign(l[0].size(),’ ’);
  while(l.size()<r.size()) l.push_back(str);
  str.assign(r[0].size(),’ ’);
  while(r.size()<l.size()) r.push_back(str);
  str="  "; str[2]=op; mid.push_back( str );
  while(mid.size()<l.size()) mid.push_back("  ");
  for( i=0;i<l.size();i++ )
  {
    rv.push_back(l[i]+mid[i]+r[i]);
  }

  return rv;
}

/* very similar to the unary_branch_concat, but puts parenthesis around
 * the child branch */
vector <string> paren_barnch_concat( vector <string> l, char non_terminal )
{
  vector <string> rv;
  string str;
  int s=l[0].size()+6;
  int head=3+l[0].find_first_not_of(" ");
  str.assign(s,’ ’); str[head]=non_terminal; rv.push_back( str );
  str.assign(s,’=’); str[head]=’|’; rv.push_back( str );
  str="( "+l[0]+" )";
  rv.push_back(str);
  for( unsigned int i=1;i<l.size();i++ )
  {
    str="  "+l[i]+"  ";
    rv.push_back(str);
  }
  return rv;
}

/* must predefine because of circular dependencies */
vector <string> expr( string e );
vector <string> term( string t );
vector <string> fact( string f );

vector <string> expr( string e )
{
  vector <string> rv;
  int p;
  if( (p=findpos(e,’+’,’−’)) != −1 )
  {
    // E −> E + T
    // E −> E − T
    rv=binary_branch_concat(expr(e.substr(0,p)),term(e.substr(p+1)),e[p],’E’);
  }
  else
  {
    // E −> T
    rv=unary_branch_concat(term(e),’E’);
  }
  return rv;
}

vector <string> term( string t )
{
  vector <string> rv;
  int p;
  if( (p=findpos(t,’*’,’/’)) != −1 )
  {
    // T −> T * F
    // T −> T / F
    rv=binary_branch_concat(term(t.substr(0,p)),fact(t.substr(p+1)),t[p],’T’);
  }
  else
  {
    // T -> F
    rv=unary_branch_concat(fact(t),'T');
  }
  return rv;
}

vector <string> fact( string f )
{
  vector <string> rv;
  if( f[0] == '(' )
  {
    rv=paren_barnch_concat(expr(f.substr(1,f.size()-2)),'F');
  }
  else
  {
    // make the terminal string array
    rv.push_back("F");
    rv.push_back("|");
    rv.push_back("i");
  }
  return rv;
}

int main()
{
  string s;
  vector <string> v;
  unsigned int i;
  bool ftime=true;

  cin >> s;
  while( cin )
  {
    v=expr(s);

    if( ftime )
    {
      ftime=false;
    }
    else
    {
      cout << endl;
    }

    // ah, we must remove all the whitespace (problem says so!) and we
    // needed keep things in nice neat arrays for the way I solved the
    // problem, so the easiest way is to manually remove it!
    for(i=0;i<v.size();i++)
    {
      cout << v[i].erase(v[i].find_last_not_of(" ")+1) << endl;
    }
    cin >> s;
  }

  return 0;
}
```

Problem Statement

When reading data as a string of characters, it is often useful to convert the data into a different datatype, which is easier to work with than a string of characters. For example, many operations are made much easier if the string of characters "123" is converted from a string of characters into a different datatype, such as an int or a float. We want to determine the best datatype to use for some data, given a string of characters. For the purposes of this problem, we will consider converting a string of characters into one of 4 datatypes:

- "INTEGER" - if the string consists only of the digits '0' - '9'.
- "BOOLEAN" - if the string is "true" or "false" (ignoring case)
- "DECIMAL" - if the string contains only digits '0' - '9', and exactly one decimal point ('.'). "123.12", ".123", and "124." are all of type "DECIMAL". Note that by this definition the string "." is classified as a "DECIMAL"
- "STRING" - if the string is none of the other datatypes.

Your task is to write a class Datatype with a method getType that takes a string, **var**, and classifies it into one of the above four types, and returns the datatype as a string.

Definition

Class:	Datatype
Method:	getType
Parameters:	string
Returns:	string
Method signature:	string getType(string var)

(be sure your method is public)

Constraints

- **var** will contain between 1 and 50 characters, inclusive.
- Each character in **var** will be a letter ('a'-'z' or 'A'-'Z'), a digit ('0'-'9'), a decimal point ('.'), a space (' '), or one of the following characters: ,/<>?;':"[]{}\|`~!@#$%^*()_+-=&

Examples

0)

```
"123"
Returns: "INTEGER"
```

1)

```
"324.1"
Returns: "DECIMAL"
```

2)

```
".12"
Returns: "DECIMAL"
```

3)

```
 "453."
Returns: "DECIMAL"
```

4)

```
 "770.555.1212"
Returns: "STRING"
```

5)

```
 "TrUe"
Returns: "BOOLEAN"
```

6)

```
 "this is just a string"
Returns: "STRING"
```

7)

```
 "453 ducks flew 4739.45 miles."
Returns: "STRING"
```

8)

```
 "."
Returns: "DECIMAL"
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>
#include <ctype.h>

using namespace std;

class Datatype
{
        public:
                string getType(string var)
                {
      string lowervar;
      lowervar=var;
      int numdots=0;
      int numnums=0;
      for( int i=0;i<lowervar.length();i++ )
      {
        lowervar[i]=tolower(lowervar[i]);
        if( var[i] <= '9' && var[i] >= '0' ) numnums++;
        if( var[i] == '.' ) numdots++;
      }
      if( lowervar == "true" || lowervar == "false" )
      {
        return "BOOLEAN";
      }
      if( numnums == var.length() ) return "INTEGER";
      if( numnums == var.length()-1 && numdots == 1 ) return "DECIMAL";

      return "STRING";
                }
};


// Powered by FileEdit


// Powered by FileEdit

// Powered by PopsEdit
```

1997/98 ACM International Collegiate Programming Contest
University of Ulm Local Contest

Problem H

Tree Recovery

Source file: tree.(c|C|pas)
Input file: tree.in

Little Valentine liked playing with binary trees very much. Her favorite game was constructing randomly looking binary trees with capital letters in the nodes.
This is an example of one of her creations:

```
      D
     / \
    /   \
   B     E
  / \     \
 /   \     \
A     C     G
           /
          /
         F
```

To record her trees for future generations, she wrote down two strings for each tree: a preorder traversal (root, left subtree, right subtree) and an inorder traversal (left subtree, root, right subtree). For the tree drawn above the preorder traversal is DBACEGF and the inorder traversal is ABCDEFG.
She thought that such a pair of strings would give enough information to reconstruct the tree later (but she never tried it).

Now, years later, looking again at the strings, she realized that reconstructing the trees was indeed possible, but only because she never had used the same letter twice in the same tree.
However, doing the reconstruction by hand, soon turned out to be tedious.
So now she asks *you* to write a program that does the job for her!

Input Specification

The input file will contain one or more test cases.
Each test case consists of one line containing two strings *preord* and *inord*, representing the preorder traversal and inorder traversal of a binary tree. Both strings consist of unique capital letters. (Thus they are not longer than 26 characters.)
Input is terminated by end of file.

Output Specification

For each test case, recover Valentine's binary tree and print one line containing the tree's postorder traversal (left subtree, right subtree, root).

Sample Input

```
DBACEGF ABCDEFG
BCAD CBAD
```

Sample Output

```
ACBFGED
CDAB
```

```
#include <iostream>
#include <fstream>

using namespace std;

char inord[27], preord[27];

ofstream outfile( "tree.out" );

// mem in-place alg. using offsets
void p_postord( int inl, int inr, int prl, int prr )
{
  cout << "Considering tree: ";

  for( int i=prl;i<=prr;i++ )
  {
    cout << preord[i];
  }
  cout << " ";
  for( int i=inl;i<=inr;i++ )
  {
    cout << inord[i];
  }
  cout << endl;

  if( inl==inr || prl==prr )
  {
    if( inl==inr && prl==prr && inord[inl] == preord[prl])
    {
      // print the leaf
      outfile << inord[inl];
    }
    else
    {
      cerr << " error!!" << endl;
    }
  }
  else
  {
    int i,j;
    // find root in in-order print
    for( i=inl;i<=inr;i++ )
    {
      if( preord[prl] == inord[i] ) break;
    }
    if( i > inr )
    {
      cerr << "error !!!" <<endl;
    }
    // if there is a left child, print it
    if( i!=inl )
    {
      p_postord( inl,i-1,prl+1,prl+i-inl );
      // if there is a right child too, print it
      if( i!=inr )
      {
        p_postord( i+1,inr,prl+i-inl+1,prr );
      }
    }
    else
    {
      p_postord( inl+1,inr,prl+1,prr );
    }
    // now print root
    outfile << preord[prl];
  }
}

int main()
```

```
{
  ifstream infile( "tree.in" );

  if( !infile || !outfile )
  {
    cerr << "bad files" << endl;
    return 1;
  }

  infile >> preord >> inord;
  while( infile )
  {
    p_postord( 0, strlen(inord)-1, 0, strlen(preord)-1 );
    outfile << endl;

    infile >> preord >> inord;
  }

  return 0;
}
```

Problem Statement

The World Wide Web (WWW), as we all know, is a collection of pages, which are linked together to form a very large graph, where each page represents a node in the graph, and each link represents a directed edge. Since there are billions of pages on the WWW, it is important to be able to search through them, and have good methods to determine which ones are relevant.

One statistic about a page that might be of interest to a search engine is its *clustering coefficient*. To find the clustering coefficient of a page, *p*, first we find all of the pages that *p* links directly to. Then, we count the total number of links between all of those pages and divide by the total number of possible links between those pages (for our purposes, a pages may not be linked multiple times to the same page). If *p* links to zero or one pages, then its clustering coefficient is undefined. Note that clustering coefficients are usually used in conjunction with undirected graphs, but that we are expanding them here to be used on directed graphs (since the links in the WWW are directed).

Your task is, given a list of pages, and how they are linked together, determine the pages with the maximal (defined) clustering coefficient. You will be given a vector <string>, **links**, where each element is a single-space delimited list of the names of the pages. The first term represents the page that the links are on, and the remaining terms represent the pages that it links to. So, "A B C D" would mean that page "A" has a link to pages "B", "C", and "D". You are to return a vector <string> that contains all of the pages which have the maximal clustering coefficient sorted in lexicographic order. If all of the pages have an undefined clustering coefficient, return an empty vector <string>.

Definition

Class:	Clusters
Method:	mostClustered
Parameters:	vector <string>
Returns:	vector <string>
Method signature:	vector <string> mostClustered(vector <string> links)
(be sure your method is public)	

Notes

- Assume that the same name always describes the same page.

Constraints

- **links** has between 1 and 20 elements inclusive
- each element of **links** will contain between 1 and 50 characters inclusive.
- each element of **links** consists only of upper-case letters ('A'-'Z') and spaces
- each element of **links** doesn't contain leading/trailing spaces
- each element of **links** consists of single-space delimited names
- no element of **links** contains duplicate name
- no two elements of **links** start with the same name
- no page links to itself

Examples

0)

```
{"A B C D", "B A E D"}
Returns: { "A",  "B" }
```

"A" is linked to three pages, "B", "C", and "D". There are 6 different possible links between these three pages (B->C, B->D, C->B, C->D, D->B, D->C). Only one of these links (B->D) actually exists, so the clustering coefficient of "A" is 1/6.

"B" is also linked to three pages, which have 6 possible links between them. Only one of the possible links exists though (A->D) so its clustering coefficient is also 1/6.

All of the other pages have undefined clustering coefficients, since they do not link to any pages. Thus, both "A" and "B" have the maximal clustering coefficient of 1/6, and we return then in sorted order.

1)

```
{"A", "B"}
Returns: { }
```

A and B are not linked to any other pages.

2)

```
{"SEARCH SCHOOL NEWS", "NEWS FINANCE WORLD", "FINANCE WORLD"}
Returns: { "NEWS" }
```

3)

```
{"A B C D E F",
"B A C D E F",
"C A B D E F",
"D A B C E F",
"E A B C D F",
"F A B C D E"}
Returns: { "A",  "B",  "C",  "D",  "E",  "F" }
```

Every page links to every other page here.

4)

```
{"Z Y X"}
Returns: { "Z" }
```

5)

```
{"A", "B C"}
Returns: { }
```

6)

```
{"A"}
Returns: { }
```

7)

```
{
"ABRA CA DABRA",
"DABRA CA",
"CA DABRA",
"D A B",
"A B C"
}
Returns: { "ABRA" }
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
// practice
// from SRM 135 DIV I 450
// by ~Andy~
#include <string>
#include <vector>
#include <iostream>
#include <map>
#include <string.h>
using namespace std;
struct rat{ int n; int d; };
class Clusters {
  public:
  int g[500][500];
  int nem;
  rat clus(int nd)
  {
    rat ret;
    int i,j;
    int t=0;
    int n=0;
    for( i=0;i<nem;i++ )
    {
      if( g[nd][i] )
      {
        n++;
        for( j=0;j<nem;j++ )
        {
          if( g[i][j] && g[nd][j] )
          {
            t++;
          }
        }
      }
    }
    if( n<2 )
    {
      ret.n=-1;
      ret.d=-1;
    }
    else
    {
      ret.n=t;
      ret.d=n*(n-1);
    }
    return ret;
  }

  vector <string> mostClustered(vector <string> links) {
    vector <string> rvs;
    map <string,int> m;
    map <int,string> rm;
    int sn;
    char par[51];
    char *str;
    rat mx={0,1};
    string t;
    // init graph
    nem=0;
    for( int i=0; i<500; i++ )
    {
      for( int j=0; j<500; j++ )
      {
        g[i][j]=0;
      }
    }
    for( int i=0;i<links.size();i++ )
    {
      strcpy( par, links[i].c_str() );
      t=strtok( par, " " );
```

```
      if( m.count(t) == 0 )
      {
        m[t]=nem;
        rm[nem]=t;
        nem++;
      }
      sn=m[t];
      while( (str=strtok( NULL, " " )) )
      {
        t=str;
//        cout << " parse string: " << t << endl;
        if( m.count(t) == 0 )
        {
          m[t]=nem;
          rm[nem]=t;
          nem++;
        }
        g[sn][m[t]]=1;
      }
    }
    rat rv;
    for( int i=0;i<nem;i++ )
    {
      rv=clus(i);
//      cout << "clus " << rv.n << "/" << rv.d << endl;
      if( rv.n < 0 ) continue;
      if( rv.n*mx.d > mx.n*rv.d )
      {
//        cout << "new max cluster at node " << rm[i] << " with cluster of " <<
rv.n << "/" << rv.d << endl;
        rvs.clear();
        mx=rv;
      }
      if( rv.n*mx.d == mx.n*rv.d )
      {
//        cout << "adding another tie at node " << rm[i] << endl;
        rvs.push_back( rm[i] );
      }
    }
    sort(rvs.begin(),rvs.end());
    return rvs;
  }
};

// Powered by PopsEdit
```

```
/* A solution to the strongly connected components problem. Solution was
 * helped by Introduction to Algorithms by Cormen, Leiserson, Rivest and
 * Stein. */

/* input is read from strconn.in in the following form:
 * <number_of_verticies>
 * <from_edge_1> m1 [to_edge_1 [to_edge_2 [...[to_edge_m1] ]]]
 * <from_edge_2> m2 [to_edge_1 [to_edge_2 [...[to_edge_m2] ]]]
 * ...
 * <from_edge_n> mn [to_edge_1 [to_edge_2 [...[to_edge_mn] ]]]
 *
 * there can be multiple cases per file. end of input is indicated by a zero
 * number of verticies. The edges are specified as strings on the alphabet
 * set [a-zA-Z0-9]. Output is the stronly connected components, sorted
 * printed with a label before each case.
 *
 * Example strconn.in:
 * 3
 * A 1 B
 * B 2 A C
 * C 0
 * 5
 * Bill 1 Ryan
 * Andy 3 Jesse Andy Neal
 * Jesse 3 Andy Neal Ryan
 * Neal 2 Andy Ryan
 * Ryan 1 Bill
 * 0
 *
 * Example strconn.out:
 * Graph 1:
 * A B
 * C
 * Graph 2:
 * Andy Jesse Neal
 * Bill Ryan
 *
 * Problem statement written by C. Andy Martin */

/* Solution coded by: C. Andy Martin */

#include <iostream>
#include <fstream>
#include <map>
#include <vector>
#include <string>
#include <algorithm>

#include <stdio.h>

using namespace std;

const char color_names[3][8]={"white","gray","black"};
enum coloring { white, gray, black };

struct vertex_info
{
  coloring color;
  int discoveryTime;
  int finishingTime;
  string pred;
};

// return list of nodes visited
vector<string> dfs_visit( map< string, vector<string> > &adj_list, map<string,ve
rtex_info> &info, string &u, int &time )
{
  vector<string> rv, tt;
  vector<string>::iterator v;
```

```
  info[u].color=gray;
  info[u].discoveryTime= ++time;
  for( v=adj_list[u].begin(); v!=adj_list[u].end(); v++ )
  {
    if( info[*v].color == white )
    {
      info[*v].pred=u;
      tt=dfs_visit(adj_list,info,*v,time);
      // remember old list of nodes
      rv.insert(rv.end(),tt.begin(),tt.end());
    }
  }
  info[u].color=black;
  info[u].finishingTime= ++time;
  // add the visited node to the list of visited nodes
  rv.push_back(u);
  return rv;
}

// return a list of the different trees visited
vector< vector<string> > dfs( vector<string> &vertex_order, map< string, vector<
string> > &adj_list, map<string,vertex_info> &info )
{
  int time;
  vector<string>::iterator i;
  vector< vector<string> > rv;
  for( i=vertex_order.begin();i!=vertex_order.end();i++ )
  {
    info[*i].color=white;
    info[*i].pred.resize(0);
  }
  time=0;
  for( i=vertex_order.begin();i!=vertex_order.end();i++ )
  {
    if( info[*i].color == white )
    {
      rv.push_back(dfs_visit(adj_list,info,*i,time));
    }
  }
  return rv;
}

map<string,vertex_info> nfo;
bool finish_cmpr(const string &lv, const string &rv)
{
  return nfo[lv].finishingTime > nfo[rv].finishingTime;
}

int main()
{
  map< string, vector<string> > adj, radj;
  vector< string > vorder;
  vector< vector<string> > comps;
  vector<string>outp;
  int i, j, n, m, cnt=0, vcnt;
  string u, v;
  ifstream infile( "strconn.in" );
  ofstream outfile( "strconn.out" );

  if( !infile || !outfile ) { cerr << "File problems!" << endl; return 1; }
  infile >> n;
  while( n )
  {
    adj.clear();
    radj.clear();
    nfo.clear();
    vorder.clear();

    for( i=0; i<n; i++ )
    {
      infile >> u >> m;
      vorder.push_back(u);
      for( j=0; j<m; j++ )
      {
        infile >> v;
        adj[u].push_back(v);
        radj[v].push_back(u);
      }
    }

    dfs(vorder,adj,nfo);
    sort(vorder.begin(),vorder.end(),finish_cmpr);
    comps=dfs(vorder,radj,nfo);

    outp.clear();
    for(i=0;i<comps.size();i++)
    {
      string wk;
      sort(comps[i].begin(),comps[i].end());
      wk=comps[i][0];
      for(j=1;j<comps[i].size();j++)
      {
        wk=wk+" "+comps[i][j];
      }
      outp.push_back(wk);
    }
    sort(outp.begin(),outp.end());


    outfile << "Graph" << ++cnt << ":" << endl;
    for(i=0;i<outp.size();i++)
    {
      outfile << outp[i] << endl;
    }

    infile >> n;
  }
  return 0;
}
```

From: http://www.lo3.gdynia.pl/infokolej/oi/oi3/agenci.html
Translated by Dr. Jaromczyk. Edited by Andy Martin

III Olimpiada Informatyczna 1995/96

Zadanie: AGE Autor: Marcin Kubica
Agents

We say that agent A corrupted agent B, if A has documents on B. Some agents are willing to sell all the documents that they have, if the price is right. After arresting an agent, we collect all of his documents. Thus bribing a set of agents can launch a series of arrests and can lead to dismantling the network.

We have information about all the active agents n <= 3000 and the price of each of them

Task

Write a program that given the information finds the least expensive way to arrest all the agents, if possible. Otherwise it outputs the agent (one of them) that cannot be arrested.

Input

- The first row of AGE.IN has the number of all the agents 1 <= n <= 3000.
- The second row has the number p, 1 <= p <= n, of all the agents willing to sell documents.
- In the next p rows there is info about the agents that are willing to sell documents: the agent number, space, the amount <= 20,000.
- Next row contains one number r, 1 <= r <= 8000 denoting the number of pairs (A,B), such that A has documents on B.
- The next rows list those pairs: the number corresponding to A, space, the number corresponding to B.

Output

- In the first row of AGE.OUT write: TAK – if there is a way to arrest all the agents, NIE otherwise.
- In the second row write the minimal cost or the number of the agent that cannot be arrested.

Example
AGE.IN:
3

```
2
1 10
2 100
2
1 3
2 3
```

AGE.OUT:

```
TAK
110
```

AGE.IN:

```
4
2
1 100
4 200
2
1 2
3 4
```

AGE.OUT:

```
NIE
3
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <fstream>

#define oo 1000000

using namespace std;

const char color_names[3][8]={"white","gray","black"};
enum coloring {white,gray,black};

struct vertex_info
{
        coloring color;
  int discoveryTime;
  int finishingTime;
  int pred;
};

vector <int> dfs_visit( map< int, vector<int> > &adj_list, map <int, vertex_info
> & info, int &u, int &time )
{
  vector <int> rv, tt;
  vector <int>::iterator v;
  info[u].color=gray;
  info[u].discoveryTime= ++time;
  for( v=adj_list[u].begin(); v!=adj_list[u].end(); v++ )
  {
    if( info[*v].color == white )
    {
      info[*v].pred=u;
      tt=dfs_visit(adj_list,info,*v,time);
      rv.insert(rv.end(),tt.begin(),tt.end());
    }
  }
  info[u].color=black;
  info[u].finishingTime=++time;
  rv.push_back(u);
  return rv;
}

vector< vector<int> > dfs( vector<int> &vertex_order, map<int,vector<int> > &adj
_list, map<int,vertex_info> &info )
{
  int time;
  vector<int>::iterator i;
  vector< vector<int> > rv;
  for( i=vertex_order.begin();i!=vertex_order.end();i++ )
  {
    info[*i].color=white;
    info[*i].pred=-1;
  }
  time=0;
  for( i=vertex_order.begin();i!=vertex_order.end();i++ )
  {
    if( info[*i].color == white )
    {
      rv.push_back(dfs_visit(adj_list,info,*i,time));
    }
  }
  return rv;
}

map<int,vertex_info>nfo;
bool finish_cmpr(const int &lv, const int &rv)
{
  return nfo[lv].finishingTime > nfo[rv].finishingTime;
```

```
}

int main()
{
  // adjacency list of documents
  map< int, vector<int> > adj, radj;
  vector< int > vorder;
  vector< vector<int> > comps;
  ifstream infile( "age.in" );
  ofstream outfile( "age.out" );

  if( !infile || !outfile ) { cerr << "File problems!" << endl; return 1; }
  int n,p;
  infile >> n >> p;
  // agents prices
  map <int,int> prices;
  int agent;
  int price,i;
  for( i=1;i<=n;i++ )
  {
    prices[i]=oo;
  }
  for( i=0;i<p;i++ )
  {
    infile >> agent >> price;
    prices[agent]=price;
  }
  int nedges;
  infile >> nedges;
  int corrupted;
  for( i=1;i<=n;i++ )
  {
    // a list of all node names (agent #'s)
    vorder.push_back(i);
    // every one 'implicates' himself for his price
    adj[i].push_back(i);
    radj[i].push_back(i);
  }
  // read in edges
  for( i=0;i<nedges;i++ )
  {
    infile >> agent >> corrupted;
    adj[agent].push_back(corrupted);
    radj[corrupted].push_back(agent);
  }

  dfs(vorder,adj,nfo);
  sort(vorder.begin(),vorder.end(),finish_cmpr);
  // comps is going to have stronly connected components
  comps=dfs(vorder,radj,nfo);

//  map<int,vector<int> > comp_graph;
  vector<int> comp_indeg;
  vector<int> min_price;

  for(i=0;i<comps.size();i++)
  {
    comp_indeg.push_back(0);
    min_price.push_back(oo);
  }

  int j,k,l;
  for(i=0;i<comps.size();i++)
  {
    for(j=0;j<comps[i].size();j++)
    {
      if( prices[comps[i][j]] < min_price[i] )
      {
        min_price[i]=prices[comps[i][j]];
      }
      for(k=0;k<adj[comps[i][j]].size();k++)
      {
        for(l=0;l<comps.size();l++)
        {
          if( l==i ) continue;
          vector<int>::iterator it=find(comps[l].begin(),comps[l].end(),adj[comp
s[i][j]][k]);
          if( it != comps[l].end() )
          {
            // don't actually need this for this problem
//            comp_graph[i].push_back(l);
            comp_indeg[l]++;
          }
        }
      }
    }
  }

  map<int,vertex_info> comp_info;
  vector<int> visited;
  int total_price=0;
  for(i=0;i<comps.size();i++)
  {
    if( comp_indeg[i] == 0 )
    {
      if( min_price[i] == oo )
      {
        cout << "NIE" << endl;
        // since we can print any, just print first in comp
        cout << comps[i][0] << endl;
        return 0;
      }
      else
      {
        total_price+=min_price[i];
      }
    }
  }

  cout << "TAK" << endl;
  cout << total_price << endl;
  return 0;
}
```

The 2003 27th Annual **acm** International Collegiate

Programming Contest World Finals

sponsored by IBM

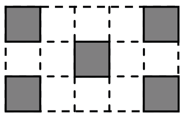
Problem A
Building Bridges
Input File: bridges.in

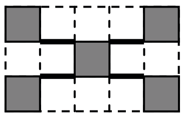
The City Council of New Altonville plans to build a system of bridges connecting all of its downtown buildings together so people can walk from one building to another without going outside. You must write a program to help determine an optimal bridge configuration.

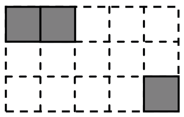
New Altonville is laid out as a grid of squares. Each building occupies a connected set of one or more squares. Two occupied squares whose corners touch are considered to be a single building and do not need a bridge. Bridges may be built only on the grid lines that form the edges of the squares. Each bridge must be built in a straight line and must connect exactly two buildings.

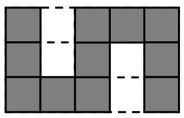
For a given set of buildings, you must find the minimum number of bridges needed to connect all the buildings. If this is impossible, find a solution that minimizes the number of disconnected groups of buildings. Among possible solutions with the same number of bridges, choose the one that minimizes the sum of the lengths of the bridges, measured in multiples of the grid size. Two bridges may cross, but in this case they are considered to be on separate levels and do not provide a connection from one bridge to the other.

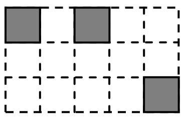
The figure below illustrates four possible city configurations. City 1 consists of five buildings that can be connected by four bridges with a total length of 4. In City 2, no bridges are possible, since no buildings share a common grid line. In City 3, no bridges are needed because there is only one building. In City 4, the best solution uses a single bridge of length 1 to connect two buildings, leaving two disconnected groups (one containing two buildings and one containing a single building).

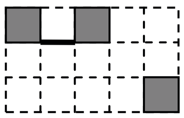

City 1


City 1
with bridges


City 2
No bridges are possible


City 3
No bridges are needed


City 4


City 4
with bridges

Input
The input data set describes several rectangular cities. Each city description begins with a line containing two integers *r* and *c,* representing the size of the city on the north-south and east-west axes measured in grid lengths ($1 \leq r \leq 50$ and $1 \leq c \leq 50$). These numbers are followed by exactly *r* lines, each consisting of *c* hash ("#") and dot (".") characters. Each character corresponds to one square of the grid. A hash character corresponds to a square that is occupied by a building, and a dot character corresponds to a square that is not occupied by a building.

The input data for the last city will be followed by a line containing two zeros.

Output

For each city description, print two or three lines of output as shown below. The first line consists of the city number. If the city has fewer than two buildings, the second line is the sentence "No bridges are needed." If the city has two or more buildings but none of them can be connected by bridges, the second line is the sentence "No bridges are possible." Otherwise, the second line is "*N* bridges of total length *L*" where *N* is the number of bridges and *L* is the sum of the lengths of the bridges of the best solution. (If *N* is 1, use the word "bridge" rather than "bridges.") If the solution leaves two or more disconnected groups of buildings, print a third line containing the number of disconnected groups.

Print a blank line between cases. Use the output format shown in the example.

Sample Input

```
3 5
#...#
..#..
#...#
3 5
##...
.....
....#
3 5
#.###
#.#.#
###.#
3 5
#.#..
.....
....#
0 0
```

Output for the Sample Input

```
City 1
4 bridges of total length 4

City 2
No bridges are possible.
2 disconnected groups

City 3
No bridges are needed.

City 4
1 bridge of total length 1
2 disconnected groups
```

```cpp
/* Problem A of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */
#include <iostream>
#include <fstream>
#include <list>
#include <vector>
#include <algorithm>

// infinte edge just needs to be greater than 48 for this problem
#define oo 50
using namespace std;

// get rid of bounds checking by surounding with -1's
int city[52][52];
char row[51];

// max number of distinct buildings is 625
struct edge
{
  int from; int to; int dist;
  bool operator==( const edge & rv ) const
  {
    return from==rv.from && to==rv.to;
  }
};

bool lt_by_dist( const edge& lv, const edge& rv )
{
  return lv.dist < rv.dist;
}

// adjacency list used for the graph reprsenting the possible bridges
list<edge> adjl;

// adds edge if it doesn't exist, otherwise it changes the value of dist iff
// the new value is less than the old
void update_edge( int x, int y, int d )
{
  if( x > y ) { int t=x;x=y;y=t; }
  edge conv;
  conv.from=x;
  conv.to=y;
  conv.dist=d;
  list<edge>::iterator it = find( adjl.begin(), adjl.end(), conv );
  if( it != adjl.end() )
  {
    if( it->dist > d ) it->dist=d;
  }
  else adjl.push_back(conv);
}

// label a building with the passed in label. if the square isn't labelable
// (it has already been labelled, or isn't part of a building) return false.
// otherwise return true.
bool label_building( int i, int j, int l )
{
  if( city[i][j] != -2 ) return false;
  city[i][j]=l;
  label_building( i-1,j-1,l );
  label_building( i,j-1,l );
  label_building( i+1,j-1,l );
  label_building( i-1,j,l );
  label_building( i+1,j,l );
  label_building( i-1,j+1,l );
  label_building( i,j+1,l );
  label_building( i+1,j+1,l );
  return true;
}
```

```cpp
int main()
{
  ifstream infile( "bridges.in" );
  int r,c;
  int i,j;
  int num_buildings;
  int from_building,to_building,from_pos;
  int city_number=0;

  if( !infile ) { cerr << "File problems." << endl; return 1; }
  infile >> r >> c;
  while( r && c )
  {
    // bound the city grid with empty space. this eliminates bounds checking
    // and makes the algorithm more general and easier to read. the only
    // down side is remembering the city grid starts at position 1,1 instead
    // of 0,0
    for( i=0;i<=r+1;i++ ) city[i][0]=city[i][c+1]=-1;
    for( i=0;i<=c+1;i++ ) city[0][i]=city[r+1][i]=-1;

    // read in the file one row at a time
    for( i=1;i<=r;i++ )
    {
      infile >> row;
      // process the row. if a '.' is encountered, put -1 in the city grid.
      // this means an unusable space (not part of a building) - or another
      // way of looking at it is space a bridge may occupy. otherwise put a
      // -2 in the space, which means part of a building is here.
      for( j=1;j<=c;j++ )
      {
        if( row[j-1] == '.' ) city[i][j] = -1;
        else city[i][j] = -2;
      }
    }

    // now process the buildings, and label each distinct building with
    // increasing numbers
    num_buildings=0;
    for( i=1;i<=r;i++ )
    {
      for( j=1;j<=c;j++ )
      {
        if( label_building(i,j,num_buildings) )
        {
          num_buildings++;
        }
      }
    }

    // first take care of trival case of 1 or 0 buildings
    cout << "City " << ++city_number << endl;
    if( num_buildings <= 1 )
    {
      cout << "No bridges are needed." << endl;
    }
    else
    {
      // we construct a graph representing all the shortest possible bridges
      // between buildings (that is, if more than one bridge exists between
      // buildings, eventually the shortest bridge between them will be
      // represented by the edge weight in the graph). this is not yet a
      // minimal spanning tree - but we just run an MST algorithm on this
      // graph to get the tree, and hence the answer.

      // initialize adjacency list
      adjl.clear();

      // finding the bridges is done by scanning each line in between the
      // city grid (the lines *between* the buildings). First we scan
      // horizontally, and then vertically. when a building piece is
      // encountered on either the row above the scan line or the row below
      // (or column, if vertical), remember its number. if the old remembered
      // number is different than this remembered number, than a bridge may
      // be built between those buildings. call the update_edge() function
      // which will add the edge if it is a better edge than one that
      // previouslly linked these two cities (graph nodes).

      // label horizontal bridges
      for( i=0;i<=r;i++ )
      {
        from_building=-1;
        from_pos=-1;
        for( j=1;j<=c;j++ )
        {
          to_building=-1;
          if( city[i][j] != -1 ) to_building = city[i][j];
          else if( city[i+1][j] != -1 ) to_building = city[i+1][j];
          if( to_building != -1 )
          {
            if( from_building != -1 && from_building != to_building )
            {
              update_edge(from_building,to_building,j-from_pos-1);
            }
            from_building=to_building;
            from_pos=j;
          }
        }
      }

      // label vertical bridges
      for( j=0;j<=c;j++ )
      {
        from_building=-1;
        from_pos=-1;
        for( i=1;i<=r;i++ )
        {
          to_building=-1;
          if( city[i][j] != -1 ) to_building = city[i][j];
          else if( city[i][j+1] != -1 ) to_building = city[i][j+1];
          if( to_building != -1 )
          {
            if( from_building != -1 && from_building != to_building )
            {
              update_edge(from_building,to_building,i-from_pos-1);
            }
            from_building=to_building;
            from_pos=i;
          }
        }
      }

      // sort the adjacency list by distances. this speeds up kruskal's MST
      // algorithm.
      adjl.sort( lt_by_dist );

      // setup the node labels for the MST algorithm. These labels indicate
      // which connected group the nodes are in. Initially, each node starts
      // in its own connected group. To update the groups, we just do an
      // O(number of nodes) search and replace because our number of nodes
      // is small (max 625).
      vector <int> node_group;
      for( i=0; i<num_buildings; i++ )
      {
        node_group.push_back( i );
      }
      // now use kruskal's algorithm to 'construct' the minimal spanning tree
      // we don't actually make the tree, but make node groups which are in
```

```
      // a MST and total up the number of bridges, total length, and
      // disconnected components (which is just the number of buildings
      // minus the total number of bridges). This is more convinient for the
      // output format used, espicially since we don't need the information
      // gained by constructing an actual MST.
      int total_bridges=0;
      int total_bridge_length=0;
      int disconnected_groups=node_group.size();
      // start with smallest edge weights. this is kruskal's MST approach
      // (technically, we are abusing the term MST, since we may construct a
      // minimal spanning forest, since our graph is not neccessarily
      // connected. the algorithm still works correctly in the case of
      // unconnected graphs).
      for( list<edge>::iterator it=adjl.begin();it!=adjl.end();it++ )
      {
        // if they are in different connected components
        if( node_group[it->from] != node_group[it->to] )
        {
          // then join the components
          int old_group = node_group[it->to];
          for( i=0; i<node_group.size(); i++ )
          {
            if( node_group[i] == old_group )
            {
              node_group[i] = node_group[it->from];
            }
          }
          // add the bridge to the total
          total_bridges++;
          disconnected_groups--;
          total_bridge_length+=it->dist;
        }
      }
      // now output the specified information in the specified format
      if( total_bridges == 0 )
      {
        cout << "No bridges are possible." << endl;
      }
      else
      {
        cout << total_bridges << " bridge" << ((total_bridges==1)?"":"s")
          << " of total length " << total_bridge_length << endl;
      }
      if( disconnected_groups > 1 )
      {
        cout << disconnected_groups << " disconnected groups" << endl;
      }
    }

    // get next case
    infile >> r >> c;
    // if there is a next case, preface it with an extra line.
    if( r && c ) cout << endl;
  }

  return 0;
}
```

Problem F
Combining Images
Input File: images.in

As the exchange of images over computer networks becomes more common, the problem of image compression takes on increasing importance. Image compression algorithms are used to represent images using a relatively small number of bits.

One image compression algorithm is based on an encoding called a "Quad Tree." An image has a Quad Tree encoding if it is a square array of binary pixels (the value of each pixel is 0 or 1, called the "color" of the pixel), and the number of pixels on the side of the square is a power of two.

If an image is homogeneous (all its pixels are of the same color), the Quad Tree encoding of the image is 1 followed by the color of the pixels. For example, the Quad Tree encoding of an image that contains pixels of color 1 only is 11, regardless of the size of the image.

If an image is heterogeneous (it contains pixels of both colors), the Quad Tree encoding of the image is 0 followed by the Quad Tree encodings of its upper-left quadrant, its upper-right quadrant, its lower-left quadrant, and its lower-right quadrant, in order.

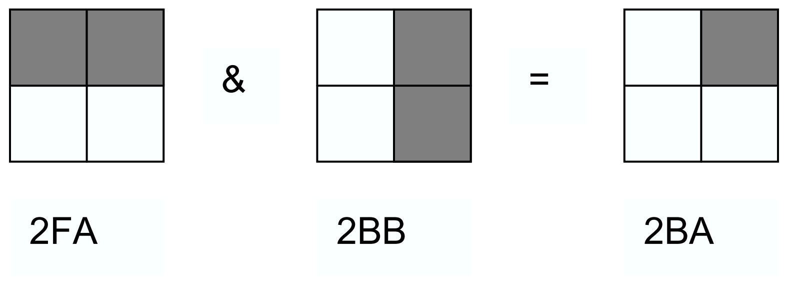
The Quad Tree encoding of an image is a string of binary digits. For easier printing, a Quad Tree encoding can be converted to a Hex Quad Tree encoding by the following steps:

a. Prepend a 1 digit as a delimiter on the left of the Quad Tree encoding.
b. Prepend 0 digits on the left as necessary until the number of digits is a multiple of four.
c. Convert each sequence of four binary digits into a hexadecimal digit, using the digits 0 to 9 and capital A through F to represent binary patterns from 0000 to 1111.

For example, the Hex Quad Tree encoding of an image that contains pixels of color 1 only is 7, which corresponds to the binary string 0111.

You must write a program that reads the Hex Quad Tree encoding of two images, computes a new image that is the intersection of those two images, and prints its Hex Quad Tree encoding. Assume that both input images are square and contain the same number of pixels (although the lengths of their encodings may differ). If two images A and B have the same size and shape, their intersection (written as A & B) also has the same size and shape. By definition, a pixel of A & B is equal to 1 if and only if the corresponding pixels of image A and image B are both equal to 1.

The following figure illustrates two input images and their intersection, together with the Hex Quad Tree encodings of each image. In the illustration, shaded squares represent pixels of color 1.



Input
The input data set contains a sequence of test cases, each of which is represented by two lines of input. In each test case, the first input line contains the Hex Quad Tree encoding of the first image and the second line contains the Hex Quad Tree encoding of the second image. For each input image, the number of hexadecimal digits in its Hex Quad Tree encoding will not exceed 100.

The last test case is followed by two input lines, each containing a single zero.

Output

For each test case, print “Image” followed by its sequence number. On the next line, print the Hex Quad Tree encoding of the intersection of the two images for that test case. Separate the output for consecutive test cases with a blank line.

Sample Input	Output for the Sample Input

```
2FA
2BB
2FB
2EF
7
2FA
0
0
```
	```
	Image 1:
	2BA

	Image 2:
	2EB

	Image 3:
	2FA
	```

```java
// Problem F of the ACM/ICPC 2003 World Finals
// Solved by: C. Andy Martin
//
// The key to this problem is noticing that the images do not need to be
// turned into rasters, we can work on them in tree form. Actually, we don’t
// even need to construct a tree, we can directly construct the output
// string from the input strings. We do this by examining the substructure
// of each case. If the A quadTree starts with a 1, then if the next digit
// is also a 1, we need to copy the corresponding part of B quadTree into
// the result. If its a zero, we need to copy in a 10 to the result, and the
// same if B starts with 1. Otherwise, they both start with zero, and we
// have to merge the subtrees. This is the only tricky case. We have to
// extract the quadrants from the strings (we do this with a different
// function) and merge them recursively.
//
import java.io.*;
import java.math.*;

public class Images
{
  private static class Result
  {
    public Result( BigInteger a, int la )
    {
      image=a;
      bitlen=la;
    }
    public BigInteger image;
    public int bitlen;
  }
  public static Result getQuad( BigInteger a, int la )
  {
    Result rv;
    if( a.testBit(la−1) )
    {
      rv = new Result( a.shiftRight( la−2 ).and( new BigInteger( "11", 2 )), 2 )
;
    }
    else
    {
      Result ul = getQuad( a, la−1 );
      Result ur = getQuad( a, la−1−ul.bitlen );
      Result ll = getQuad( a, la−1−ul.bitlen−ur.bitlen );
      Result lr = getQuad( a, la−1−ul.bitlen−ur.bitlen−ll.bitlen );
      rv = new Result( ul.image.shiftLeft( ur.bitlen+ll.bitlen+lr.bitlen )
          .or( ur.image.shiftLeft( ll.bitlen+lr.bitlen )
            .or( ll.image.shiftLeft( lr.bitlen )
              .or( lr.image )
            )
          ), ul.bitlen+ur.bitlen+ll.bitlen+lr.bitlen+1 );
      // plus one at end for the zero that belongs at the beginning of the
      // bitstring
    }
//    System.out.println( "getQuad( " + a.toString(2) + ", " + Integer.toString(
la) + ") returns: ( " + rv.image.toString(2) + ", " + Integer.toString( rv.bitle
n ) + ")" );
    return rv;
  }

  // returns quad tree in BigInteger that is an and of the two passed in
  // quad trees (leading one must be stripped from HexQuadTree)
  public static Result process( BigInteger a, BigInteger b, int la, int lb )
  {
    Result rv;
    if( a.testBit( la−1 ) )
    {
      if( a.testBit( la−2 ) )
      {
        rv = new Result(b,lb);
```

```java
      }
      else
      {
        rv = new Result(a,la);
      }
    }
    else if( b.testBit( lb−1 ) )
    {
      if( b.testBit( lb−2 ) )
      {
        rv = new Result(a,la);
      }
      else
      {
        rv = new Result(b,lb);
      }
    }
    else
    {
      // now, both start bits must be zero, so we have to merge the subtrees
      // of each of the quadrants of the image
      Result aul = getQuad( a, la−1 );
      Result bul = getQuad( b, lb−1 );
      Result ul = process( aul.image, bul.image, aul.bitlen, bul.bitlen );
      Result aur = getQuad( a, la−1−aul.bitlen );
      Result bur = getQuad( b, lb−1−bul.bitlen );
      Result ur = process( aur.image, bur.image, aur.bitlen, bur.bitlen );
      Result all = getQuad( a, la−1−aul.bitlen−aur.bitlen );
      Result bll = getQuad( b, lb−1−bul.bitlen−bur.bitlen );
      Result ll = process( all.image, bll.image, all.bitlen, bll.bitlen );
      Result alr = getQuad( a, la−1−aul.bitlen−aur.bitlen−all.bitlen );
      Result blr = getQuad( b, lb−1−bul.bitlen−bur.bitlen−bll.bitlen );
      Result lr = process( alr.image, blr.image, alr.bitlen, blr.bitlen );
      rv = new Result( ul.image.shiftLeft( ur.bitlen+ll.bitlen+lr.bitlen )
        .or( ur.image.shiftLeft( ll.bitlen+lr.bitlen )
          .or( ll.image.shiftLeft( lr.bitlen )
            .or( lr.image )
          )
        ), ul.bitlen+ur.bitlen+ll.bitlen+lr.bitlen+1 );
    }
//    System.out.println( "process( " + a.toString(2) + ", " + b.toString(2) + "
, " + Integer.toString(la) + ", " + Integer.toString(lb) + ") returns: ( " + rv.
image.toString(2) + ", " + Integer.toString( rv.bitlen ) + ")" );
    return rv;
  }
  public static void main( String args[] )
  {
    try
    {
      FileReader inFile = new FileReader( "images.in" );
      StreamTokenizer inTok = new StreamTokenizer( inFile );
      inTok.resetSyntax();
      inTok.wordChars( ’0’, ’9’ );
      inTok.wordChars( ’A’, ’F’ );
      inTok.wordChars( ’a’, ’f’ );
      inTok.whitespaceChars( ’\t’, ’\t’ );
      inTok.whitespaceChars( ’ ’, ’ ’ );
      inTok.whitespaceChars( ’\n’, ’\n’ );
      inTok.whitespaceChars( ’\r’, ’\r’ );
      inTok.eolIsSignificant( false );
      BigInteger a,b;
      Result res;
      int la,lb;
      int cnt = 0;
      inTok.nextToken();
      a = new BigInteger( inTok.sval, 16 );
      inTok.nextToken();
      b = new BigInteger( inTok.sval, 16 );
      while( a.compareTo( BigInteger.ZERO ) != 0 || b.compareTo( BigInteger.ZERO
 ) != 0 )
      {
        la=a.bitLength()-1;
        lb=b.bitLength()-1;
        // strip off leading one bit
        a = a.clearBit( la );
        b = b.clearBit( lb );
        res = process( a, b, la, lb );
        res.image = res.image.setBit( res.bitlen );
        System.out.println( "Image " + Integer.toString( ++cnt ) + ":" );
        System.out.println( res.image.toString( 16 ).toUpperCase() );
        inTok.nextToken();
        a = new BigInteger( inTok.sval, 16 );
        inTok.nextToken();
        b = new BigInteger( inTok.sval, 16 );
        if( a.compareTo( BigInteger.ZERO ) != 0 || b.compareTo( BigInteger.ZERO
) != 0 )
          System.out.println( "" );
      }
    }
    catch( Exception e )
    {
      System.out.println( "Caught exception: " + e.toString() );
      e.printStackTrace( System.out );
    }
  }
}
```

Problem J
Toll
Input File: toll.in

Sindbad the Sailor sold 66 silver spoons to the Sultan of Samarkand. The selling was quite easy; but delivering was complicated. The items were transported over land, passing through several towns and villages. Each town and village demanded an entry toll. There were no tolls for leaving. The toll for entering a *village* was simply one item. The toll for entering a *town* was one piece per 20 items carried. For example, to enter a town carrying 70 items, you had to pay 4 items as toll. The towns and villages were situated strategically between rocks, swamps and rivers, so you could not avoid them.

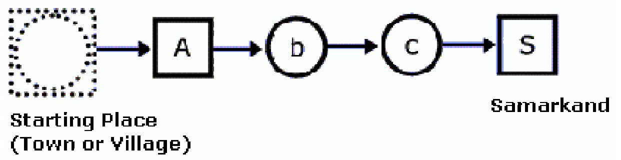


Figure 1: To reach Samarkand with 66 spoons, traveling through a town followed by two villages, you must start with 76 spoons.

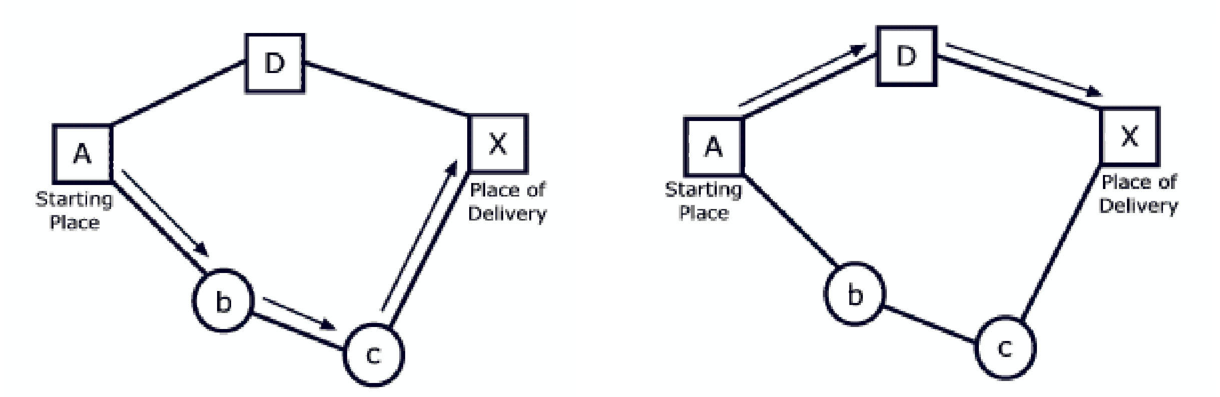


Figure 2: The best route to reach X with 39 spoons, starting from A, is A→b→c→X, shown with arrows in the figure on the left. The best route to reach X with 10 spoons is A→D→X, shown in the figure on the right. The figures display towns as squares and villages as circles.

Predicting the tolls charged in each village or town is quite simple, but finding the best route (the cheapest route) is a real challenge. The best route depends upon the number of items carried. For numbers up to 20, villages and towns charge the same. For large numbers of items, it makes sense to avoid towns and travel through more villages, as illustrated in Figure 2.

You must write a program to solve Sindbad's problem. Given the number of items to be delivered to a certain town or village and a road map, your program must determine the total number of items required at the beginning of the journey that uses a cheapest route.

Input

The input consists of several test cases. Each test case consists of two parts: the roadmap followed by the delivery details.

The first line of the roadmap contains an integer *n*, which is the number of roads in the map ($0 \leq n$). Each of the next *n* lines contains exactly two letters representing the two endpoints of a road. A capital letter represents a town; a lower case letter represents a village. Roads can be traveled in either direction.

Following the roadmap is a single line for the delivery details. This line consists of three things: an integer *p* ($0 < p \leq 1000$) for the number of items that must be delivered, a letter for the starting place, and a letter for the place of delivery. The roadmap is always such that the items can be delivered.

The last test case is followed by a line containing the number -1.

Output

The output consists of a single line for each test case. Each line displays the case number and the number of items required at the beginning of the journey. Follow the output format in the example given below.

Sample Input	Output for the Sample Input
1 a Z 19 a Z 5 A D D X A b b c c X 39 A X -1	Case 1: 20 Case 2: 44

```
/* Problem J of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */

/* Basically, we do a single source shortest path from the DESTINATION using
 * the dynamic edge weights for towns as specified in the problem. This will
 * give us the smallest amount we can start with and get to the finish.
 */
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

#define oo 1000000000

int main()
{
  ifstream infile( "toll.in" );
  int cnt=0;

  int i,j,k;
  int n,p;
  int w;
  // start and finish
  char s,f;
  // adjacency matrix - doesn't have edge weights since they are dynamic
  int adj[52][52];
  // shortest path guesses (updated by relaxing edges)
  int spath[52];

  if( !infile ) { cerr << "File problems." << endl; return 1; }

  infile >> n;
  while( n != -1 )
  {
    memset(adj,0,52*52*sizeof(int));
    for( i=0;i<52;i++ )
    {
      spath[i]=oo;
    }
    for( i=0;i<n;i++ )
    {
      infile >> s >> f;
      // map the towns to [0,25] and the villages to [26,51]
      if( s > 'Z' ) s-='a'-26; else s-='A';
      if( f > 'Z' ) f-='a'-26; else f-='A';
      adj[s][f]=1;
      adj[f][s]=1;
    }
    infile >> p >> s >> f;
    if( s > 'Z' ) s-='a'-26; else s-='A';
    if( f > 'Z' ) f-='a'-26; else f-='A';

    spath[f]=p;

    // do a modified bellman-ford shortest path from the END to the BEGIN.
    // this algorithm basically modifies our guess for the shortest path at
    // each step. we know we can stop after |V| which is at most 52 (26*2).
    // we aren't really finding the shortest path, but the path which costs
    // the least given our final desired cost p (we are working our way back
    // to the start)
    for( i=0; i<52; i++ )
    {
      for( j=0; j<52; j++ )
      {
        for( k=0; k<52; k++ )
        {
          if( adj[j][k] && spath[j] != oo )
```

```
          {
            // edge weight is 1 for villiage...
            if( j >= 26 ) w=1;
            else
            {
              // for a town it can be found by the following:
              // we know that if z is the amount we have getting to j, and
              // y is the amount we have leaving k then ceil(y/20)=y-z. if
              // we assume 20 divides y, then z=20*y/19. if 20 does not
              // divide y, then z=20*y/19+1. so, assume 20 divides y,
              // calculate z then check if y indeed is divisible by 20. if
              // not, then add one to z.
              // so the edge weight is z-y:
              w = (20*spath[j])/19;
              if( w % 20 != 0 ) w+=1;
              w -= spath[j];
            }
            if( spath[k] > spath[j] + w )
            {
              spath[k] = spath[j] + w;
            }
          }
        }
      }
    }

    /*
    for( i=0;i<52;i++ )
    {
      for( j=0;j<52;j++ )
      {
        cout << adj[i][j] << " ";
      }
      cout << endl;
    }
    for( j=0;j<52;j++ )
    {
      cout << spath[j] << endl;
    }
    */
    cout << "Case " << ++cnt << ":" << spath[s] << endl;
    infile >> n;
  }
  return 0;
}
```

Problem A
Balloons in a Box
Input: balloon.in

You must write a program that simulates placing spherical balloons into a rectangular box.

The simulation scenario is as follows. Imagine that you are given a rectangular box and a set of points. Each point represents a position where you might place a balloon. To place a balloon at a point, center it at the point and inflate the balloon until it touches a side of the box or a previously placed balloon. You may not use a point that is outside the box or inside a previously placed balloon. However, you may use the points in any order you like, and you need not use every point. Your objective is to place balloons in the box in an order that maximizes the total volume occupied by the balloons.

You are required to calculate the volume within the box that is not enclosed by the balloons.

Input
The input consists of several test cases. The first line of each test case contains a single integer *n* that indicates the number of points in the set ($1 \le n \le 6$). The second line contains three integers that represent the (x, y, z) integer coordinates of a corner of the box, and the third line contains the (x, y, z) integer coordinates of the opposite corner of the box. The next *n* lines of the test case contain three integers each, representing the (x, y, z) coordinates of the points in the set. The box has non-zero length in each dimension and its sides are parallel to the coordinate axes.

The input is terminated by the number zero on a line by itself.

Output
For each test case print one line of output consisting of the test case number followed by the volume of the box not occupied by balloons. Round the volume to the nearest integer. Follow the format in the sample output given below.

Place a blank line after the output of each test case.

Sample Input

```
2
0 0 0
10 10 10
3 3 3
7 7 7
0
```

Output for the Sample Input

```
Box 1: 774
```

```
/* Solution (hopefully - not tested against a judging program or stringent
 * test cases) to problem A of 2002 ACM/ICPC World Finals */

/* Solved by: C. Andy Martin
 *
 * Combination of greedy/brute force. It works well because of the limit on
 * the number of balloons. We can be greedy because for a given ordering of
 * balloon blowings, we can blow the first until it hits a wall or blown
 * balloon, the second, the third, etc. It will always give us a smaller
 * area to blow less (i.e., leave space for another blown balloon) OR it
 * will give us a case we will catch with one of the other perumtations of
 * greedy blowings. We then find the max of all the volumes generated by the
 * greedy blowings (actually, find min of area not in balloons, which is
 * equivalent, i'm not sure why I did it this way - I am adding these
 * comments about 8 months after doing this problem */

#include <iostream>
#include <fstream>
#include <math.h>
#include <string.h>
#include <algorithm>

using namespace std;

class pt
{
  public:
    double x,y,z;
};

int main()
{
  ifstream infile;
  ofstream outfile;
  int n,i,j,k;
  pt bl, tr;
  pt pts[6];
  int permute[6];
  const double pi=acos(0)*2;
  double min, bv;
  double min_vol,t;
  int count=0;

  infile.open( "balloon.in" );
  outfile.open( "balloon.out" );
  if( !infile ) { cerr << "Could not open infile." << endl; return -1; }
  if( !outfile ) { cerr << "Could not open outfile." << endl; return -2; }

  infile >> n;
  while( n && infile )
  {
    double best_radii[6];
    infile >> bl.x >> bl.y >> bl.z;
    infile >> tr.x >> tr.y >> tr.z;
    if( bl.x > tr.x ) { t=bl.x;bl.x=tr.x;tr.x=t; }
    if( bl.y > tr.y ) { t=bl.y;bl.y=tr.y;tr.y=t; }
    if( bl.z > tr.z ) { t=bl.z;bl.z=tr.z;tr.z=t; }
    min_vol = bv = (tr.x-bl.x)*(tr.y-bl.y)*(tr.z-bl.z);
    for(i=0;i<n;i++)
    {
      infile >> pts[i].x >> pts[i].y >> pts[i].z;
      permute[i]=i;
      best_radii[i]=0;
    }
    // brute 6! = 720 so won't be too slow
    do
    {
      double radius[6];
      double volume;
```

```
      volume=bv;
      for(i=0;i<n;i++)
      {
        radius[i]=0;
      }
      for(i=0;i<n;i++)
      {
        //find the closest obstruction
        double mint=tr.x-pts[permute[i]].x;
        if( pts[permute[i]].x-bl.x < mint )
          mint=pts[permute[i]].x-bl.x;
        if( tr.y-pts[permute[i]].y < mint )
          mint=tr.y-pts[permute[i]].y;
        if( pts[permute[i]].y-bl.y < mint )
          mint=pts[permute[i]].y-bl.y;
        if( tr.z-pts[permute[i]].z < mint )
          mint=tr.z-pts[permute[i]].z;
        if( pts[permute[i]].z-bl.z < mint )
          mint=pts[permute[i]].z-bl.z;
        for(j=0;j<i;j++)
        {
          // must have nonzero radius
          // to present and obstruciton
          if(radius[permute[j]])
          {
            double tt=sqrt( ((pts[permute[j]].x-pts[permute[i]].x)*(pts[permute[
j]].x-pts[permute[i]].x)+(pts[permute[j]].y-pts[permute[i]].y)*(pts[permute[j]].
y-pts[permute[i]].y)+(pts[permute[j]].z-pts[permute[i]].z)*(pts[permute[j]].z-pt
s[permute[i]].z)))-radius[permute[j]];
            if( tt<mint )
              mint=tt;
          }
        }
        if( mint > 0 )
        {
          radius[permute[i]]=mint;
          volume -= 4.0*pi*radius[permute[i]]*radius[permute[i]]*radius[permute[
i]]/3.0;
        }
        else
        {
          radius[permute[i]]=0;
        }
      }

      if( volume < min_vol )
      {
        memcpy(best_radii,radius,sizeof(radius));
        min_vol=volume;
      }

    }
    while( next_permutation(permute,permute+n) );

    outfile << "Box " << ++count << ": " << (int)(min_vol+.5) << endl;
    infile >> n;
  }
  return 0;
}
```

Problem Statement

You have been given a sealed box full of objects. Each object in the box is either red or blue; in addition, each object is either a cube or a sphere. You will be given some information about the contents of the box. Among the things you might be told are:

`known[0]`: The total number of objects in the box.

`known[1]`: The number of red objects in the box.

`known[2]`: The number of blue objects in the box.

`known[3]`: The number of cubes in the box.

`known[4]`: The number of spheres in the box.

`known[5]`: The number of red cubes in the box.

`known[6]`: The number of red spheres in the box.

`known[7]`: The number of blue cubes in the box.

`known[8]`: The number of blue spheres in the box.

You are given a vector <int>, **`known`**, containing the information you have been told. Each element in **`known`** will either be a nonnegative integer (the count of the object types indicated in the above key), or -1 (meaning that you haven't been told this number). For example, say `known = {50, -1, -1, -1, 30, -1, -1, -1, -1}`. This means that you only know two things about the box: it contains 50 objects, total; and 30 of those objects are spheres.

You will also be given an integer, **`target`**, indicating the information you would like to find out. Using the above example, if `target = 3`, it would mean that you would like to know how many cubes are in the box. You would return 20, since every object is either a cube or a sphere.

Given **`known`** and **`target`**, return an integer representing the number of objects of the type indicated by **`target`**, where **target** has the same meaning as the indices of **known**. There will always be one and exactly one possible value for the target which is consistent with the information in **known**. In other words, you will be able to uniquely determine the target value, given **known**. Furthermore, you know that the total number of objects in the box cannot exceed 100, and the number of any particular type of object (red cube, red sphere, blue cube, or blue sphere) cannot exceed 25.

Definition

Class:	ObjectCounter
Method:	getCount
Parameters:	vector <int>, int
Returns:	int
Method signature:	int getCount(vector <int> known, int target)
(be sure your method is public)	

Notes

- There will be at most 25 red cubes, 25 red spheres, 25 blue cubes, and 25 blue spheres in the box. Usually, you would try to figure out some set of equations relating to the different pieces of data. However, given that you have a computer handy, there are probably simpler ways to do it.

Constraints

- There will be exactly one possible object count for the target which is consistent with **known**.
- **`known`** will contain exactly 9 elements.
- Element 0 of **`known`** will be between -1 and 100, inclusive.
- Elements 1, 2, 3, and 4 of **`known`** will each be between -1 and 50, inclusive.
- Elements 5, 6, 7, and 8 of **`known`** will each be between -1 and 25, inclusive.
- **`target`** will be between 0 and 8, inclusive.
- The data in **`known`** will not contradict itself, given the restrictions imposed above.

Examples

0)

```
{50,-1,-1,-1,30,-1,-1,-1,-1}
3
Returns: 20
```

This is the example from above. You know that there are 50 objects in the box, 30 of which are spheres. You want to know how many cubes there are. The only possible value that will work for the number of cubes (target = 3) is 20. Any other choice c will cause 30 + c to not equal 50.

1)

```
{-1,25,-1,35,-1,15,-1,-1,0}
0
Returns: 45
```

You know that there are 25 red objects, 35 cubes, 15 red cubes, and 0 blue spheres in the box. You want to know the total number of objects in the box. Since there are 25 red objects, and 15 of those are cubes, the other 10 must be spheres. In addition, since there are 35 cubes, 15 of which are red, the other 20 must be blue. We now know that there are 15 red cubes, 10 red spheres, 20 blue cubes, and 0 blue spheres in the box. Therefore, there are 15+10+20+0=45 total objects in the box.

2)

```
{-1,-1,-1,-1,-1,-1,-1,17,-1}
7
Returns: 17
```

The only thing you know is that there are 17 blue cubes in the box. However, the number of blue cubes is exactly what you want to know.

3)

```
{0,-1,-1,-1,-1,-1,-1,-1,-1}
2
Returns: 0
```

If there are no objects in the box, then there are no blue objects in the box, either.

4)

```
{23,-1,11,3,20,0,-1,-1,-1}
6
Returns: 12
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
using namespace std;
class ObjectCounter {
  public:
  int getCount(vector <int> known, int target) {
    for(int i=0;i<=25;i++)
    {
      for(int j=0;j<=25;j++)
      {
        for(int k=0;k<=25;k++)
        {
          for(int l=0;l<=25;l++)
          {
            if( (known[5] < 0 || known[5] == i)
             && (known[6] < 0 || known[6] == j)
             && (known[7] < 0 || known[7] == k)
             && (known[8] < 0 || known[8] == l)
             && (known[1] < 0 || known[1] == i+j)
             && (known[2] < 0 || known[2] == k+l)
             && (known[3] < 0 || known[3] == i+k)
             && (known[4] < 0 || known[4] == j+l)
             && (known[0] < 0 || known[0] == i+j+k+l) )
            {
              known[0]=i+j+k+l;
              known[1]=i+j;
              known[2]=k+l;
              known[3]=i+k;
              known[4]=j+l;
              known[5]=i;
              known[6]=j;
              known[7]=k;
              known[8]=l;
              break;
            }
          }
        }
      }
    }
    return known[target];
  }
};

// Powered by PopsEdit
```

The 2003 27th Annual **acm** International Collegiate

Programming Contest World Finals

sponsored by IBM

Problem B
Light Bulbs
Input File: bulbs.in

Hollywood's newest theater, the Atheneum of Culture and Movies, has a huge computer-operated marquee composed of thousands of light bulbs. Each row of bulbs is operated by a set of switches that are electronically controlled by a computer program. Unfortunately, the electrician installed the wrong kind of switches, and tonight is the ACM's opening night. You must write a program to make the switches perform correctly.

A row of the marquee contains n light bulbs controlled by n switches. Bulbs and switches are numbered from 1 to n, left to right. Each bulb can either be ON or OFF. Each input case will contain the initial state and the desired final state for a single row of bulbs.

The original lighting plan was to have each switch control a single bulb. However the electrician's error caused each switch to control two or three consecutive bulbs, as shown in Figure 1. The leftmost switch ($i = 1$) toggles the states of the two leftmost bulbs (1 and 2); the rightmost switch ($i = n$) toggles the states of the two rightmost bulbs ($n - 1$ and n). Each remaining switch ($1 < i < n$) toggles the states of the three bulbs with indices $i - 1$, i, and $i + 1$. (In the special case where there is a single bulb and a single switch, the switch simply toggles the state of that bulb.) Thus, if bulb 1 is ON and bulb 2 is OFF, flipping switch 1 will turn bulb 1 OFF and bulb 2 ON. The minimum cost of changing a row of bulbs from an initial configuration to a final configuration is the minimum number of switches that must be flipped to achieve the change.

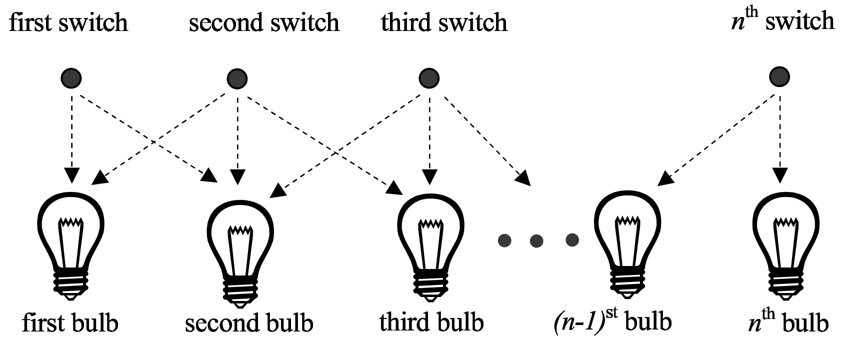


Figure 1

You can represent the state of a row of bulbs in binary, where 0 means the bulb is OFF and 1 means the bulb is ON. For instance, 01100 represents a row of five bulbs in which the second and third bulbs are both ON. You could transform this state into 10000 by flipping switches 1, 4, and 5, but it would be less costly to simply flip switch 2.

You must write a program that determines the switches that must be flipped to change a row of light bulbs from its initial state to its desired final state with minimal cost. Some combinations of initial and final states may not be feasible. For compactness of representation, decimal integers are used instead of binary for the bulb configurations. Thus, 01100 and 10000 are represented by the decimal integers 12 and 16.

Input
The input file contains several test cases. Each test case consists of one line. The line contains two non-negative decimal integers, at least one of which is positive and each of which contains at most 100 digits. The first integer represents the initial state of the row of bulbs and the second integer represents the final state of the row. The binary equivalent of these integers represents the initial and final states of the bulbs, where 1 means ON and 0 means OFF.

To avoid problems with leading zeros, assume that the first bulb in either the initial or the final configuration (or both) is ON. There are no leading or trailing blanks in the input lines, no leading zeros in the two decimal integers, and the initial and final states are separated by a single blank.

The last test case is followed by a line containing two zeros.

Output

For each test case, print a line containing the case number and a decimal integer representing a minimum-cost set of switches that need to be flipped to convert the row of bulbs from initial state to final state. In the binary equivalent of this integer, the rightmost (least significant) bit represents the *n*th switch, 1 indicates that a switch has been flipped, and 0 indicates that the switch has not been flipped. If there is no solution, print "impossible". If there is more than one solution, print the one with the smallest decimal equivalent.

Print a blank line between cases. Use the output format shown in the example.

Sample Input

```
12 16
1 1
3 0
30 5
7038312 7427958190
4253404109 657546225
0 0
```

Output for the Sample Input

```
Case Number 1: 8

Case Number 2: 0

Case Number 3: 1

Case Number 4: 10

Case Number 5: 2805591535

Case Number 6: impossible
```

```
// Problem B of the ACM/ICPC 2003 World Finals
// Solved by: C. Andy Martin
//
// Solution method:
// Add an extra bulb to the front and end which are also controlled by the
// leftmost and rightmost switches. This generalizes the problem. We solve
// each of the four possible strings created by adding the possible values
// on either end. We then take the minimum solution of these strings. (This
// is the same as trying to flip and not to flip the first switch, but it
// makes more sense this way). We then take the XOR of the starting string
// and the finishing string. This new string is the bulbs which must be
// toggled from their starting state. Now we notice for the first bulb in this
// new string that if it needs toggling, we MUST flip the first switch,
// since it is the only switch that controls this bulb. We simulate the
// switch by toggling the first three bits. Now, if the second bit is set,
// the second switch must be flipped, if not it must not, etc. At the end,
// if the last two bits match the desired state, we have a possible
// solution. Remember its value. After collecting the possible values, pick
// the one with minimum representation. Done.
//
import java.math.*;
import java.io.*;

public class Bulbs
{
  public static BigInteger test( BigInteger t, int n )
  {
    BigInteger rv = new BigInteger( "0" );
    for( int i=0; i<n; i++ )
    {
      if( t.testBit(i) )
      {
        rv = rv.setBit(i);
        t = t.flipBit(i);
        t = t.flipBit(i+1);
        t = t.flipBit(i+2);
      }
    }
    if( t.equals( BigInteger.ZERO ) )
    {
      return rv;
    }
    return null;
  }

  public static void main( String args[] )
  {
    try
    {
      FileReader inFile = new FileReader( "bulbs.in" );
      StreamTokenizer inTok = new StreamTokenizer( inFile );
      inTok.resetSyntax();
      inTok.wordChars( ’0’, ’9’ );
      inTok.whitespaceChars( ’\t’, ’\t’ );
      inTok.whitespaceChars( ’ ’, ’ ’ );
      inTok.whitespaceChars( ’\n’, ’\n’ );
      inTok.whitespaceChars( ’\r’, ’\r’ );
      inTok.eolIsSignificant( false );
      BigInteger a,b;
      int cnt = 0;
      int n = 0;
      inTok.nextToken();
      a = new BigInteger( inTok.sval );
      inTok.nextToken();
      b = new BigInteger( inTok.sval );
      while( a.compareTo( BigInteger.ZERO ) != 0 || b.compareTo( BigInteger.ZERO
 ) != 0 )
      {
        BigInteger x = a.xor(b);
```

```
        n = x.bitLength();
        x = x.shiftLeft(1);
        x = x.clearBit(0);
        BigInteger min = test( x, n );
        x = x.setBit(0);
        BigInteger t = test( x, n );
        if( min == null )
          min = t;
        else if( t != null && (t.bitCount() < min.bitCount() ||
              t.bitCount() == min.bitCount() && t.compareTo(min) < 0) )
          min = t;
        x = x.clearBit(0);
        x = x.setBit(x.bitLength());
        t = test( x, n );
        if( min == null )
          min = t;
        else if( t != null && (t.bitCount() < min.bitCount() ||
              t.bitCount() == min.bitCount() && t.compareTo(min) < 0) )
          min = t;
        x = x.setBit(0);
        t = test( x, n );
        if( min == null )
          min = t;
        else if( t != null && (t.bitCount() < min.bitCount() ||
              t.bitCount() == min.bitCount() && t.compareTo(min) < 0) )
          min = t;

        cnt++;
        System.out.print( "Case number " + Integer.toString( cnt ) + ": " );
        if( min == null )
        {
          System.out.println( "impossible" );
        }
        else
        {
          System.out.println( min.toString() );
        }

        inTok.nextToken();
        a = new BigInteger( inTok.sval );
        inTok.nextToken();
        b = new BigInteger( inTok.sval );
        if( a.compareTo( BigInteger.ZERO ) != 0 || b.compareTo( BigInteger.ZERO
) != 0 )
          System.out.println( "" );
      }
    }
    catch( Exception e )
    {
      System.out.println( "Caught exception: " + e.toString() );
      e.printStackTrace( System.out );
    }
  }
}
```

```
#include <iostream>
#include <math.h>
#include <deque>
#include <ctype.h>

using namespace std;

// bigints are stored as vectors of digits (which are in order of increasing
// significance)

class bigint
{
  public:
    bigint( void );
    bigint( int iv );
    bigint( const char * iv );
    bigint( const bigint & iv );
    const bigint & operator=( const bigint & lv );
    ostream & print( ostream & out ) const;
    istream & parse( istream & in );
    void bigint::swap( bigint & rv );
    void add( const bigint * rv, bigint * result ) const;
    void mult( const bigint * rv, bigint * result ) const;
    void div( const bigint * rv, bigint * q, bigint * r ) const;
    int compare( const bigint * rv, bool ignoreSign ) const;
    const bigint & operator+( const bigint & rv ) const;
    const bigint & operator-( void ) const;
    const bigint & operator-( const bigint & rv ) const;
    const bigint & operator+=( const bigint & rv );
    const bigint & operator-=( const bigint & rv );
    const bigint & operator++( void );
    const bigint & operator--( void );
    const bigint & operator++( int notused );
    const bigint & operator--( int notused );
    const bigint & operator*( const bigint & rv ) const;
    const bigint & operator*=( const bigint & rv );
    bool operator==( const bigint & rv ) const;
    bool operator!=( const bigint & rv ) const;
    bool operator<( const bigint & rv ) const;
    bool operator>( const bigint & rv ) const;
    bool operator<=( const bigint & rv ) const;
    bool operator>=( const bigint & rv ) const;
  private:
    deque<char> digits;
    bool ephemeral;
    bool negative;
};

ostream & operator<<( ostream & stream, bigint & rv );
istream & operator>>( istream & stream, bigint & rv );

ostream & operator<<( ostream & stream, bigint & rv )
{
  return rv.print(stream);
}

istream & operator>>( istream & stream, bigint & rv )
{
  return rv.parse(stream);
}

bigint::bigint( void )
{
  ephemeral=false;
  negative=false;
  digits.clear();
}

bigint::bigint( int iv )
```

```
{
  if( iv < 0 )
  {
    negative=true;
    iv=-iv;
  }
  else
  {
    negative=false;
  }
  ephemeral=false;
  while( iv > 0 )
  {
    digits.push_back( iv%10 );
    iv/=10;
  }
}

bigint::bigint( const char * iv )
{
  bool start=true;
  digits.clear();
  negative=false;
  ephemeral=false;
  if( *iv == '-' )
  {
    iv++;
    negative=true;
  }
  while( isdigit(*iv) )
  {
    if( !(start && *iv == '0') )
    {
      start=false;
      digits.push_front( *iv-'0' );
    }
    iv++;
  }
}

bigint::bigint( const bigint & iv )
{
  ephemeral=false;
  negative=iv.negative;
  digits.assign( iv.digits.begin(), iv.digits.end() );
}

const bigint & bigint::operator=( const bigint & lv )
{
  ephemeral=false;
  negative=lv.negative;
  digits.assign( lv.digits.begin(), lv.digits.end() );
  return *this;
}

ostream & bigint::print( ostream & out ) const
{
  int i=0;
  int n=digits.size();
  if( n==0 )
  {
    return out << "0";
  }
  if( negative )
  {
    n++;
  }
  char temp[n+1];
  deque<char>::const_iterator itr=digits.end();
```

```cpp
  if( negative && !(n==1 && digits[0] == 0))
  {
    temp[i++]='-';
  }
  for( ; i<n; i++ )
  {
    temp[i]=*--itr+'0';
  }
  temp[i]='\0';
//      if( ephemeral ) delete this;
  return out << temp;
}

istream & bigint::parse( istream & in )
{
  char tk;
  bool start=true;
  digits.clear();
  negative=false;
  ephemeral=false;
  // read one character at a time until we get a non-digit
  in.get(tk);
  if( tk == '-' )
  {
    negative=true;
    in.get(tk);
  }
  while( isdigit(tk) )
  {
    if( !(start && tk == '0') )
    {
      start=false;
      // its OK to parse n^2
      digits.push_front( tk-'0' );
    }
    in.get(tk);
  }
}

// swaps data with rv
void bigint::swap( bigint & rv )
{
  bool t;
  t=ephemeral;
  ephemeral=rv.ephemeral;
  rv.ephemeral=t;
  t=negative;
  negative=rv.negative;
  rv.negative=t;
  digits.swap( rv.digits );
}

void bigint::add( const bigint * rv, bigint * result ) const
{
  const bigint * lv=this;

  if( result != this && result != rv )
  {
    result->digits.clear();
  }

  // put bigger number on top, ignoring sign
  if( compare(rv,true) == -1 )
  {
    const bigint *t=lv;
    lv=rv;
    rv=t;
  }
```

```cpp
  // now add each digit with carry
  int carry = 0;
  int sum;
  int dgt;
  for( dgt=0;dgt<lv->digits.size();dgt++ )
  {
    sum=carry+lv->digits[dgt];
    if( dgt<rv->digits.size() )
    {
      // if exactly one is negative do a subtraction
      if( lv->negative && !rv->negative || !lv->negative && rv->negative )
      {
        sum-=rv->digits[dgt];
      }
      else
      {
        sum+=rv->digits[dgt];
      }
    }
    if( sum>=10 )
    {
      sum-=10;
      carry=1;
    }
    else if( sum<0 )
    {
      sum+=10;
      carry=-1;
    }
    else
    {
      carry=0;
    }
    if( result->digits.size() == dgt )
    {
      result->digits.push_back(sum);
    }
    else
    {
      result->digits[dgt]=sum;
    }
  }
  // if there was an extra carry, add a 1 to the result
  if( carry )
  {
    result->digits.push_back(1);
  }
  // the result is negative if the top number was negative
  if( lv->negative )
  {
    result->negative=true;
  }
  else
  {
    result->negative=false;
  }
  // and, remove any zeroes that are left over
  while( result->digits.size() > 0 && result->digits.back() == 0 )
  {
    result->digits.pop_back();
  }
//      if( lv.ephemeral ) delete &lv;
//      if( ephemeral ) delete this;
}

// return 0 for equal, -1 if this<rv 1 if this>rv
// if ignoreSign is true, the comparsion is done on absolute value of
// the numbers
int bigint::compare( const bigint * rv, bool ignoreSign=false ) const
```

```
{
  if( rv->digits.size() == digits.size() )
  {
    if( !ignoreSign && rv->negative && !negative )
    {
      return 1;
    }
    else if( !ignoreSign && !rv->negative && negative )
    {
      return -1;
    }
    // take care of trivial case (the following loop will fail on this)
    if( digits.size() == 0 )
    {
      return 0;
    }
    // start with most significant digit
    deque<char>::const_iterator lvi = digits.end();
    deque<char>::const_iterator rvi = rv->digits.end();
    do
    {
      rvi--;
      lvi--;
      if( *rvi < *lvi )
      {
        if( !ignoreSign && negative )
        {
          return -1;
        }
        else
        {
          return 1;
        }
      }
      else if( *rvi > *lvi )
      {
        if( !ignoreSign && negative )
        {
          return 1;
        }
        else
        {
          return -1;
        }
      }
    }
    while( lvi != digits.begin() );
    // else equal
    return 0;
  }
  else if( rv->digits.size() < digits.size() )
  {
    if( !ignoreSign && negative )
    {
      return -1;
    }
    else
    {
      return 1;
    }
  }
  else
  {
    if( !ignoreSign && negative )
    {
      return 1;
    }
    else
    {
```

```
      return -1;
    }
  }
}

void bigint::mult( const bigint * rv, bigint * result ) const
{
  const bigint * lv=this;
  bigint work;

  work.negative=false;
  work.digits.clear();

  // put bigger number on top, ignoring sign
  if( compare(rv,true) == -1 )
  {
    const bigint *t=lv;
    lv=rv;
    rv=t;
  }

  // now multiply numbers
  //
  // XXXXXX <- j progresses on this number
  //  YYYYY <- i progresses on this number

  int i,j;
  int carry,mres;
  bigint line;
  for( i=0;i<rv->digits.size();i++ )
  {
    carry=0;
    line.digits.clear();
    for( j=0;j<i;j++ )
    {
      line.digits.push_back(0);
    }
    for( j=0;j<lv->digits.size();j++ )
    {
      mres=lv->digits[j]*rv->digits[i]+carry;
      line.digits.push_back(mres%10);
      carry=mres/10;
    }
    if( carry>0 )
    {
      line.digits.push_back(carry);
    }
    work.add(&line,&work);
  }

  // put proper sign on result
  if( lv->negative && rv->negative || !lv->negative && !rv->negative )
  {
    work.negative=false;
  }
  else
  {
    work.negative=true;
  }
  // now copy in result
  *result = work;
//      if( lv.ephemeral ) delete &lv;
//      if( ephemeral ) delete this;
}

void bigint::div( const bigint * rv, bigint * q, bigint * r ) const
{

}
```

```
// convinience operator functions.
// right now, these waste memory (ephemeral doesn't work)
const bigint & bigint::operator+( const bigint & rv ) const
{
  bigint * result = new bigint;
  result->ephemeral = true;
  add( &rv, result );
  return *result;
}

const bigint & bigint::operator-( void ) const
{
  bigint * result = new bigint;
  result->ephemeral = true;
  *result = *this;
  result->negative=!result->negative;
  return *result;
}

const bigint & bigint::operator-( const bigint & rv ) const
{
  return *this+-rv;
}

const bigint & bigint::operator+=( const bigint & rv )
{
  add( &rv, this );
  return *this;
}

const bigint & bigint::operator-=( const bigint & rv )
{
  add( &-rv, this );
  return *this;
}

const bigint & bigint::operator++( void )
{
  static const bigint one(1);
  add( &one, this );
  return *this;
}

const bigint & bigint::operator--( void )
{
  static const bigint neg_one(-1);
  add( &neg_one, this );
  return *this;
}

const bigint & bigint::operator++( int notused )
{
  bigint *ret = new bigint( *this );
  ret->ephemeral=true;
  ++*this;
  return *ret;
}

const bigint & bigint::operator--( int notused )
{
  bigint *ret = new bigint( *this );
  ret->ephemeral=true;
  --*this;
  return *ret;
}

const bigint & bigint::operator*( const bigint & rv ) const
{
```

```
  bigint *ret = new bigint();
  ret->ephemeral = true;
  mult( &rv, ret );
  return *ret;
}

const bigint & bigint::operator*=( const bigint & rv )
{
  mult( &rv, this );
  return *this;
}

bool bigint::operator==( const bigint & rv ) const
{
  return compare( &rv ) == 0;
}

bool bigint::operator!=( const bigint & rv ) const
{
  return compare( &rv ) != 0;
}

bool bigint::operator<( const bigint & rv ) const
{
  return compare( &rv ) < 0;
}

bool bigint::operator>( const bigint & rv ) const
{
  return compare( &rv ) > 0;
}

bool bigint::operator<=( const bigint & rv ) const
{
  return compare( &rv ) <= 0;
}

bool bigint::operator>=( const bigint & rv ) const
{
  return compare( &rv ) >= 0;
}

int main()
{
  bigint A, B, C, D;
  ifstream infile( "bigint.in" );

  if( infile )
  {
    infile >> A;
  }
  while( infile )
  {
    B=A;
    infile >> A;
    if( !infile ) break;
    cout << B << "*" << A << "=";
    B=B*A;
    cout << B << endl;
  }

  /* fib test
  cout << "Input starting number 1: ";
  cin >> A;
  cout << "Input starting number 2: ";
  cin >> B;
  cout << "Input count: ";
  cin >> C;
  C-=2;
  cout << A << endl;
  cout << B << endl;
  while( C>0 )
  {
    D=A+B;
    A=B;
    B=D;
    cout << B << endl;
    C--;
  }
  */

  /* fact test
  cout << "Input factorial to calculate: ";
  cin >> A;

  for( B=1,C=1; B<=A; B++ )
  {
    C*=B;
  }
  cout << C << endl;
      */

  return 0;
}
```

The Cat in the Hat

Background

(An homage to Theodore Seuss Geisel)

The Cat in the Hat is a nasty creature,
But the striped hat he is wearing has a rather nifty feature.

With one flick of his wrist he pops his top off.

Do you know what's inside that Cat's hat?
A bunch of small cats, each with its own striped hat.

Each little cat does the same as line three,
All except the littlest ones, who just say ``Why me?''

Because the littlest cats have to clean all the grime,
And they're tired of doing it time after time!

The Problem

A clever cat walks into a messy room which he needs to clean. Instead of doing the work alone, it decides to have its helper cats do the work. It keeps its (smaller) helper cats inside its hat. Each helper cat also has helper cats in its own hat, and so on. Eventually, the cats reach a smallest size. These smallest cats have no additional cats in their hats. These unfortunate smallest cats have to do the cleaning.

The number of cats inside each (non-smallest) cat's hat is a constant, *N*. The height of these cats-in-a-hat is $\frac{1}{N+1}$ times the height of the cat whose hat they are in.

> The smallest cats are of height one;
> these are the cats that get the work done.

All heights are positive integers.

Given the height of the initial cat and the number of worker cats (of height one), find the number of cats that are not doing any work (cats of height greater than one) and also determine the sum of all the cats' heights (the height of a stack of all cats standing one on top of another).

The Input

The input consists of a sequence of cat-in-hat specifications. Each specification is a single line consisting of two positive integers, separated by white space. The first integer is the height of the initial cat, and the second integer is the number of worker cats.

A pair of 0's on a line indicates the end of input.

The Output

For each input line (cat-in-hat specification), print the number of cats that are not working, followed by a space, followed by the height of the stack of cats. There should be one output line for each input line other than the ``0 0'' that terminates input.

Sample Input

```
216 125
5764801 1679616
0 0
```

Sample Output

```
31 671
335923 30275911
```

```
/* @judge_id: 27516PA 107 C++ */

/*
 * Valladolid problem 107
 * The Cat in the Hat
 *
 * Coded by: Andy Martin
 * Assisted by: Bill Wyatt
 *
 * Solved by searching the height of the first cat for prime factors.
 * Once we find a prime factor, we know that that prime factor is N (the
 * number of cats in a hat).
 */

#include <iostream>
#include <string.h>
#include <memory.h>
#include <math.h>

using namespace std;

int main()
{
  // given numbers
  // H -- height of starting cat
  // nw -- number of worker cats (cats w/ height 1)
  unsigned int H,nw;
  // calculated numbers
  // N -- number of cats per hat
  unsigned int N;
  // current cat height
  unsigned int h;
  // total cat height
  unsigned int th;
  // number of cats at this stage
  unsigned int nc;
  // total number of non-workers
  unsigned int tnw;
  // depth of 'cat replication' tree
  unsigned int d;
  // iterator
  unsigned int i;

  cin >> H >> nw;
  while( H )
  {
    // find N and d
    // special case: nw==1
    if( nw == 1 )
    {
      N=1;
      d=(int)(.5+log(H)/log(2));
    }
    // special case: H==1
    else if( H == 1 )
    {
      N=1;
      d=0;
    }
    // otherwise, find N and d such that:
    // N = e^(log(nw)/d)  and  N+1 = e^(log(H)/d)
    // this is derived from:
    // H*(1/(N+1))^i == 1  and  N^i == nw
    // we find N and d by searching for integer solutions by incrementing d
    // (max d is small, at worst it is <=32, assuming 32-bit integer
    // solutions)
    else
    {
      for(d=1;d<=32;d++)
```

```
      {
        // use one formula to approximate N, and then check the solution
        // with the other formula

        N = (int)(.5+exp(log(nw)/d));
        if( H == pow( N+1, d ) )
        {
          break;
        }
      }
      if( d==32 )
      {
        cerr << "Panic! We didn't find a solution!!!";
        return 1;
      }
    }

    h=H;
    th=0;
    nc=1;
    tnw=0;
    i=0;
    while( h!=1 )
    {
      th+=nc*h;
      tnw+=nc;
      h=(h/(N+1));
      nc=nc*N;
      i++;
    }

    // now check to make sure the number of workers equals the last value
    // of number of cats
    if( nc != nw || i!=d)
    {
      cerr << "Panic!" << endl;
      cerr << "N = " << N << endl;
      cerr << "H = " << H << endl;
      cerr << "nw = " << nw << endl;
      cerr << "nc = " << nc << endl;
      cerr << "tnw = " << tnw << endl;
      cerr << "th = " << th << endl;
      cerr << "d = " << d << endl;
      cerr << "i = " << i << endl;
      return 1;
    }

    // add in the final heights of the workers
    th+=nw;

    cout << tnw << " " << th << endl;
    cin >> H >> nw;
  }

  return 0;
}
```

Return of the Aztecs

Problem H: Shifted Coefficient Number System

Time Limit: 5 seconds
Memory Limit: 32 MB

Let us define the Shifted Coefficient Number System as a number system that has B base and uses the coefficients L, L+1, L+2,... , L+B-1. As you'd soon find out if certain constraints are not met it would not be possible to express all integers in the SCNS. For this problem you'd have to find the SCNS representation of integers given in Decimal Number System, and when the representation is not possible you'd state so.

Input

There can be multiple test cases. Each test case consists of 3 integers: B, **1 < B < 17** the base for the number system, L, **|L| < 10** the least valued coefficient in the number system and N, **|N| < 2^{15}-1** a number given in Decimal Number System.

Output

For each of test cases, print two lines. The first line would be of format: "**CASE# x:**" where x is the test case number (starting at 1). In the next line print a SCNS representation of the number in the following format: **c_n*B^^{n} c_{n-1}*B^$^{n-1}$ ... c_0*B^^{0} = N**. *If there are multiple solutions print the one that uses the least number of coefficients. For two numbers in SCNS P_a and P_b, P_a gets preference over P_b when C_{ai} < C_{bi}, and for all i=n..i-1 C_{ai} = C_{bi}.* In other words between two representation of a number with same number of coefficients choose the one that has the smallest most significant digit (if they are equal then the smallest next significant digit and so on...). **None of the representations should use more than 15 coefficients.** If it is not possible to represent the number in SCNS with 15 or less coefficients, then print the words **NOT REPRESENTABLE**.

Sample Input

```
2 0 5
2 1 5
2 2 5
```

Sample Output

```
CASE# 1:
+1*2^2+0*2^1+1*2^0 = 5
CASE# 2:
+2*2^1+1*2^0 = 5
CASE# 3:
NOT REPRESENTABLE
```

Problem Setter: Monirul Hasan (Tomal), CSE Dept, Southeast University, Bangladesh

Job Interviews are so cool... the interviewers get to ask all the question that they know and the interviewee is not supposed to know. For the interviewee the experience is even cooler... they get to anticipate questions which are never asked. Those are the questions that they should have answers to, but the interviewer has no clue of.

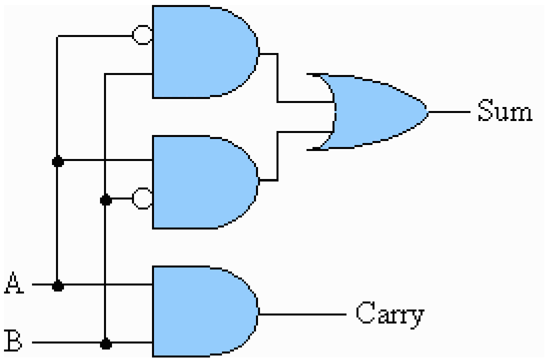
```
/* @judge_id: 27516PA 10470 C++ */

/* judged correct by the online judge */

/*
 * Valladolid problem 10470 (From Return of the Aztecs contest, H)
 * Shifted Coefficient Number System
 *
 * Coded by: Andy Martin
 *
 */

#include <iostream>
#include <stdio.h>
#include <vector>

using namespace std;

/* useful to have a true modulus function. For C, a%b returns numbers in range
 * [-b,b) instead of [0,b). So fix by, adding a b in if negative */
int mymod(int a, int b)
{
  int rv=a%b;
  if( rv<0 ) rv+=b;
  return rv;
}

/* quick integer powers by multiplication */
int mypow(int a, int b)
{
  int rv=1;
  for( int c=0; c<b; c++ )
  {
    rv*=a;
  }
  return rv;
}

int main()
{
  int b, s, n,on, d, ac;
  vector <int> c;
  int cnt=0;
  int i;

  cin >> b >> s >> on;
  n=on;
  while( cin )
  {
    printf("CASE# %d:\n",++cnt);
    c.clear();
    for( i=0;i<15;i++ )
    {
      /* this is so simple once the math is worked out. by looking at the
       * right end of the number:
       * n = (ci+s)*b^i ... (c1+s)*b + c0+s
       * we notice that at the right we just have c0+s. To get c0 from n,
       * c0 = (n-s) mod b
       * then, we just repeat this process by 'shifting' n:
       * n = (n-d)/b
       * and this works for all shifts, both positive and negative and all
       * bases !!! Best of all, this algorithm is O(#digits^2) */
      d=(mymod(n-s,b))+s;
      c.push_back( d );
      n=(n-d)/b;
      if( n == 0 ) { break; }
    }
    if( n != 0 )
    {
```

```
      printf("NOT REPRESENTABLE\n");
    }
    else
    {
      while( c.size() > 1 && c[c.size()-1] == 0 ) c.pop_back();
      for( int k=c.size()-1;k>=0;k-- )
      {
        printf("%+d*%d^%d",c[k],b,k);
      }
      printf(" = %d\n",on);
    }
    cin >> b >> s >> on;
    n=on;
  }

  return 0;
}
```

Return of the Aztecs

Problem G: To Carry or not to Carry

Time Limit: 1 second
Memory Limit: 32 MB



6+9=15 seems okay. But how come 4+6=2?

You see, Mofiz had worked hard throughout his digital logic course, but when he was asked to implement a 32 bit adder for the laboratory exam, he did some mistake in the design part. After tracing the design for half an hour, he found his flaw!! He was doing bitwise addition but his carry bit always had zero output. Thus,

```
 4 = 00000000 00000000 00000000 00000100
+6 = 00000000 00000000 00000000 00000110
----------------------------------------
 2 = 00000000 00000000 00000000 00000010
```

Its a good thing that he finally found his mistake, but it was too late. Considering his effort throughout the course, the instructor gave him one more chance. Mofiz has to write an efficient program that would take **2 unsigned 32 bit decimal numbers** as input, and produce an **unsigned 32 bit decimal number** as the output adding in the same was as his circuit does.

Input

In each line of input there will be a pair of integer separated by a single space. Input ends at EOF.

Output

For each line of input, output one line -- the value after adding the two numbers in the "Mofiz way".

Sample Input

```
4 6
6 9
```

Sample Output

```
2
15
```

Problem setter: Monirul Hasan (Tomal), CSE Dept, Southeast University, Bangladesh

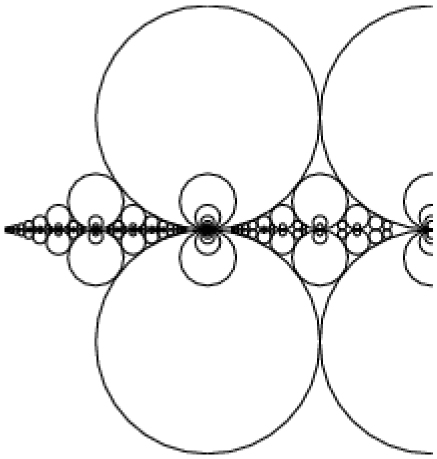
"that you used cut and paste there is really nerdy." -- goose
"really? i use cut and paste all the time." -- trip

Collected from: nerd quotes

```
/* @judge_id: 27516PA 10469 C++ */

/*
 * Valladolid problem 10469 (From Return of the Aztecs contest, G)
 * To Carry or not to Carry
 *
 * Coded by: Andy Martin
 *
 * This is a remarkably easy problem!!! It’s just XOR!!!
 * Shows that sometimes, easy problems are super easy!
 */

#include <iostream>

using namespace std;

int main()
{
  unsigned int x,y;

  cin >> x >> y;
  while( cin )
  {
    cout << (x^y) << endl;
    cin >> x >> y;
  }

  return 0;
}
```

Problem F: Farey sequences

A fraction **h**/**k** is called a proper fraction if it lies between 0 and 1 and if **h** and **k** have no common factors. For any natural number **n** >= 1, the Farey sequence of order **n**, $\mathbf{F_n}$, is the sequence of all proper fractions with denominators which do not exceed **n** together with the "fraction" 1/1, arranged in increasing order. So, for example, $\mathbf{F_5}$ is the sequence:



```
1  1  1  2  1  3  2  3  4  1
-, -, -, -, -, -, -, -, -, -.
5  4  3  5  2  5  3  4  5  1
```

It is not clear exactly who first thought of looking at such sequences. The first to have proved genuine mathematical results about them seems to be Haros, in 1802. Farey stated one of Haros' results without a proof in an article written in 1816, and when Cauchy subsequently saw the article he discoverd a proof of the result and ascribed the concept to Farey, thereby giving rise to the name *Farey sequence*. Hardy in his *A mathematician's apology* writes:

> *... Farey is immortal because he failed to understand a theorem which Haros had proved perfectly fourteen years before ...*

Surprisingly, certain simple looking claim about Farey sequences is equivalent to the Riemann hypothesis, the single most important unsolved problem in all of mathematics. Ford circles, see picture, provide a method of visualizing the Farey sequence.

But your task is much simpler than this.

For a given **n**, you are to find the **k**-th fraction in the sequence $\mathbf{F_n}$. Input consists of a sequence of lines containing two natural numbers **n** and **k**, 1 <= **n** <= 1000 and **k** sufficiently small such that there is the **k**-th term in $\mathbf{F_n}$. (The length of $\mathbf{F_n}$ is approximately $0.3039635\mathbf{n}^2$). For each line of input print one line giving the **k**-th element of $\mathbf{F_n}$ in the format as below.

Sample input

```
5 5
5 1
5 9
5 10
117 348
288 10000
```

Output for sample input

```
1/2
1/5
4/5
1/1
```

```
9/109
78/197
```

Problem Setter: Piotr Rudnicki

```
/* @judge_id: 27516PA 10408 C++ */

/* judged correct by the online judge */
/* Written by: C. Andy Martin */
/* Solves: Problem F, Farey sequences */

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std; //introduces namespace std

/* structure to represent a fraction */
struct frac
{
  int n; int d;
  bool operator<( const frac & lv ) const { return n*lv.d < lv.n*d; }
  const frac & operator=( const frac & lv ) { n=lv.n; d=lv.d; return *this; }
};

/* Euclid's GCD algorithm */
int gcd( int a, int b )
{
  if( b > a ) return gcd (b,a);
  if( b == 0 ) return a;
  return gcd( b, a%b );
}

int main ( void )
{
  vector <frac> s;
  frac tt;
  int n,k,i,j;
  int ci,mci;
  frac cur;

  cin >> n >> k;

  while( cin )
  {
    s.clear();
    /* our plan of attack is to have a list of the next n possible fractions
     * with each of the different possible numerators. we pick the smallest
     * of each of the possiblities (we rule on non simple fractions with GCD).
     * we then decrement d to the next simple fraction for that numerator,
     * in this fashion we can generate the list of farey numbers of order n
     * one at a time until we get to the kth one. A better program would
     * have an explicit formula for the kth farey number */
    for( i=1;i<=n;i++ )
    {
      tt.d=n;
      tt.n=i;
      while( gcd(tt.d,tt.n) != 1 ) tt.d--;
      s.push_back( tt );
    }
    sort(s.begin(),s.end());
    for( i=0;i<k;i++ )
    {
      mci=0;
      cur=s[mci];
      s[mci].d--;
      while( gcd(s[mci].d,s[mci].n) != 1 ) s[mci].d--;
      tt=s[mci];
      while( mci+1 < n && !(tt < s[mci+1]) )
      {
        s[mci]=s[mci+1];
        mci++;
      }
      s[mci]=tt;
```

```
    }
    cout << cur.n << "/" << cur.d << endl;

    cin >> n >> k;
  }
  return 0;
}
```

Problem Statement

The Siberian Highway starts at Samovar Village and goes for 1000 kilometers (km) in a circle. It is a one lane highway and the driving is allowed only one way: in the clockwise direction. Your car has enough tank capacity for exactly 500 km. There are ten gas stations on this highway, and they are running out of gas. You know that, at the beginning, the total amount of gas in all gas stations and in your tank is exactly enough for 1000 km. You will be given the distances from Samovar village to each of the gas stations in the clockwise direction in millimeters. (1 km = 1000 meters (m), 1 meter = 1000 millimeters (mm)). You will be given the amount of gas available at each gas station and the amount of gas in your tank before you start, also in millimeters (meaning the number of millimeters you can drive with it).

Your task is to determine the closest point to Samovar Village in the clockwise direction from which you can start the car and drive a full circle around this Highway. You will be given a vector <int> distances and a vector <int> gas, where element *i* of **distances** is the clockwise distance from the village to gas station *i* in millimeters, and element *i* of gas is the number of millimeters worth of gas available at gas station *i*. You will also be given an int, yourGas, representing the number of millimeters of gas you already have in your tank. Since there is exactly enough gas to drive 1000 km, you will have to stop at every gas station you pass and take all of the gas available if you hope to drive the full 1000 km. However, since your tank will only hold 500 km worth of gas, this will not always be possible. If it is impossible to drive around the full circle from any starting point, your method should return -1.

Definition

Class:	CircleHighway
Method:	closest
Parameters:	vector <int>, vector <int>, int
Returns:	int
Method signature:	int closest(vector <int> distances, vector <int> gas, int yourGas)

(be sure your method is public)

Notes

- You are not allowed to take more gas with you than you can fit in your tank.
- The gas stations all have zero effective width, so they can be jammed together only a millimeter apart.

Constraints

- **distances** and **gas** have exactly ten elements each
- each element of **distances** is between 0 and 999,999,999 inclusive.
- elements of **distances** are distinct and given in increasing order
- each element of **gas** is between 0 and 1,000,000,000 inclusive
- **yourGas** is between 0 and 500,000,000 inclusive
- the total sum of all elements of **gas** and **yourGas** is 1,000,000,000

Examples

0)

```
{1,5,100,101,1000,2000,3000,4000,5000,6000}
{0,1000000000,0,0,0,0,0,0,0,0}
0
Returns: -1
```

The only place where you could possibly start driving is the second station. But you can't fit all the gas into your tank, so the gas would be enough for you to drive only half the circle. Return -1

1)

```
{1,5,100,101,1000,2000,3000,4000,5000,6000}
{0,0,500000000,0,0,0,0,500000000,0,0}
0
Returns: -1
```

If you start driving at the third station then when you reach the eighth station you would not be able to fit all the gas from that station into your tank. If you start at the eighth station you wouldn't have enough gas to reach the third one. Return -1.

2)

```
{100,200,1000,2000,3000,4000,5000,500001000,600000000,699999999}
{0,0,500000000,0,0,0,0,500000000,0,0}
0
Returns: 1000
```

We can start from the third or from the eighth gas station and drive the full circle. The third station is closer clockwise. Return 1000.

3)

```
{100,200,1000,2000,3000,4000,5000,500001000,600000000,699999999}
{0,0,500000000,0,0,0,0,499999975,0,0}
25
Returns: 975
```

4)

```
{1,100000001,200000001,300000001,400000001,
500000001,600000001,700000001,800000001,900000001}
{100000000,100000000,100000000,100000000,100000000,100000000,
100000000,100000000,100000000,100000000}
0
Returns: 1
```

5)

```
{0,1,2,3,499999999,500000000,600000000,700000000,800000000,900000000}
{0,0,0,0,500000000,0,0,0,0,0}
500000000
Returns: 999999999
```

6)

```
{14001234,25790093,29205405,133628031,216203501,
241968487,335747855,358411989,409099026,436935534}

{41068575,102880168,22811826,60528045,52885843,
54609765,161190371,98070914,112850585,185668492}
107435416
Returns: 906565818
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
// from SRM 135 DIV I 950
// by ~Andy~
#include <string>
#include <vector>
#include <iostream>
using namespace std;
#define LL 1000000000
class CircleHighway {
  public:
  int m( int p )
  {
    if( p < 0 ) p+=LL;
    if( p >= LL ) p-=LL;
    return p;
  }
  int testpos( vector <int> d, vector <int> g, int yg, int yp )
  {
    vector <int> ig;
    ig.assign(g.begin(), g.end());
    int sp=yp, sg=yg, sst, st;
    int gc=0, gcr=0;
    for( sst=0; sst<10 && d[sst]<yp; sst++ )
      ;
    if( sst==10 )
      sst=0;
    st=sst;
    cout << "Testing position " << yp << endl;
    if( m(d[st]-yp) < (yg-(LL/2-g[st])) ) return -1;

    while( 1 )
    {
      cout << " At postion " << yp << " station " << st << " with " << yg << " gas " << end
l;
      if( yg < m(d[st]-yp) )
      {
        gc+=yg;
        yp=m(yp+yg);
        yg=0;
        cout << " Ran out of gas at postion " << yp << endl;
        break;
      }
      gcr=m(d[st]-yp);
      gc+=gcr;
      yg = yg - gcr + g[st];
      if( yg > LL/2 ) yg = LL/2;
      g[st]=0;
      yp = d[st++];
      if( st == 10 ) st=0;
    }

    if( gc == LL ) return sp;
    // if we had some gas to start with and came up short, try starting closer t
o the the station
    if( sg && m(d[sst]-sp) > m(d[st]-yp) )
      return testpos( d, ig, sg, m(m(d[st]-yp)+sp) );
    return -1;
  }
  int closest(vector <int> distances, vector <int> gas, int yourGas)
  {
    int min=LL,p;
    // 0 is a special case, we can't be greedy when our ranges wrap the zero poi
nt - so test it first
    p=testpos( distances, gas, yourGas, 0 );
    if( p >= 0 && p < min ) min=p;
    for( int st=0;st<10;st++ )
    {
      p=testpos( distances, gas, yourGas, distances[st] );
      if( p >= 0 && p < min ) min=p;
    }
```

```
    if( yourGas > 0 )
    {
      // try burning all your gas
      for( int st=0;st<10;st++ )
      {
        p=testpos( distances, gas, yourGas, m(distances[st]-yourGas) );
        if( p >= 0 && p < min ) min=p;
      }

      // try getting to the station with enough room to pick up gas
      for( int st=0;st<10;st++ )
      {
        p=testpos( distances, gas, yourGas, m(distances[st]-(yourGas-(LL/2-gas[s
t]))) );
        if( p >= 0 && p < min ) min=p;
      }

    }
    if( min < LL ) return min;
    return -1;
  }
};

// Powered by PopsEdit
```

Problem Statement

You are writing software that calculates a bowler's score based on the number of pins knocked down over a sequence of attempts. Because you are only building a prototype, you may assume the bowling game only has one player and that the game is completed.

Scoring in bowling works as follows. A game consists of 10 frames. For each frame, a bowler, who we'll assume is male, has up to two attempts to try and knock down a total of 10 pins. If he knocks down 10 pins on the first attempt of a frame, he gets a STRIKE and that frame is over. If he knocks down 10 pins after two attempts, he gets a SPARE. The pins do not reset between attempts in the same frame, but they do reset back to 10 at the end of each frame.

- If a bowler does not get a STRIKE or SPARE, the score for that frame is equal to the total number of pins knocked down.
- If a bowler gets a SPARE, the score for that frame is 10 plus the number of pins knocked down on the next attempt
- If a bowler gets a STRIKE, the score for that frame is 10 plus the total number of pins knocked down on the next two attempts

If a bowler gets a SPARE on the last frame, then he gets to take another attempt on a fresh set of 10 pins solely for the purpose of calculating the value of the SPARE (see examples). If a bowler gets a STRIKE on the last frame, then he gets to take two more attempts on a fresh set of 10 pins solely for the purpose of calculating the value of the STRIKE. If on the first of those two "bonus" attempts he knocks down all 10, then the second of the two bonus attempts will be on a fresh set of 10 pins (see examples).

You are given a vector <int> representing the number of pins knocked down for each attempt of a one player game. Return the final score of the game after the last attempt has been made.

Definition

Class:	BowlSim
Method:	calcScore
Parameters:	vector <int>
Returns:	int
Method signature:	int calcScore(vector <int> pinsDown)

(be sure your method is public)

Notes

- The highest possible score is a 300 from getting 10 STRIKES and 10 pins knocked down on each bonus atempt on the last frame.
- A frame is over after getting a STRIKE (unless it is the 10th frame, in which case you try for bonuses).

Constraints

- Each element of **pinsDown** is between 0 and 10, inclusive.
- **pinsDown** contains between 12 and 20 elements.

- **pinsDown** will represent a valid, complete game. That is:
  - Each element will be between 0 and 10, inclusive
  - Each frame will have between 0 and 10 pins (inclusive) knocked down, total.
  - The number of elements in **pinsDown** will be consistent with the rules of the game.

Examples

0)

```
{10,5,4,3,7,10,10,10,5,4,3,7,10,10,4,3}
Returns: 192
```

Here are the scores for each frame:

Frame 1: 10 points for knocking down 10 pins + 9 bonus points for the STRIKE

Frame 2: 9 pins for a total of 9 points

Frame 3: 10 pins + 10 bonus points for the SPARE

Frame 4: 10 pins + 20 bonus points for the STRIKE

Frame 5: 10 pins + 15 bonus points for the STRIKE

Frame 6: 10 pins + 9 bonus points for the STRIKE

Frame 7: 9 pins for a total of 9 points

Frame 8: 10 points + 10 bonus points for a SPARE

Frame 9: 10 pins + 14 bonus points for a STRIKE

Frame 10: 10 pins plus 7 bonus points for a STRIKE

The last two elements of **pinsDown** were bonus attempts from the STRIKE on frame 10

1)

```
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
Returns: 0
```

A terrible bowler has just had a huge streak of gutter balls (the ball rolled into the gutter and knocked down zero pins).

2)

```
{10,10,10,10,10,10,10,10,10,10,10,10}
Returns: 300
```

The perfect bowling score

3)

```
{0,10,5,4,0,10,5,4,0,10,5,4,0,10,5,4,0,10,5,4}
Returns: 120
```

Note that if the first attempt of a frame is a 0, and the second is a 10, this counts as a SPARE, not a STRIKE.

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

class BowlSim
{
  public:
    int calcScore(vector <int> pinsDown)
    {
      int score=0;
      for( int i=0;i<pinsDown.size();i++ )
      {
        score+=pinsDown[i];
        if( pinsDown[i] == 10 )
        {
          score+=pinsDown[i+1]+pinsDown[i+2];
          if( pinsDown.size()-3 == i )break;
        }
        else
        {
          i++;
          score+=pinsDown[i];
          if( pinsDown[i] + pinsDown[i-1] == 10 )
          {
            score+=pinsDown[i+1];
            if( pinsDown.size()-2 == i )break;
          }
        }
      }
      return score;
    }
};


// Powered by FileEdit


// Powered by FileEdit

// Powered by PopsEdit
```

Problem G: Die Game

Life is not easy. Sometimes it is beyond your control. Now, as contestants of ACM ICPC, you might be just tasting the bitter of life. But don't worry! Do not look only on the dark side of life, but look also on the bright side. Life may be an enjoyable game of chance, like throwing dice. Do or die! Then, at last, you might be able to find the route to victory.

This problem comes from a game using a die. By the way, do you know a die? It has nothing to do with "death." A die is a cubic object with six faces, each of which represents a different number from one to six and is marked with the corresponding number of spots. Since it is usually used in pair, "a die" is a rarely used word. You might have heard a famous phrase "the die is cast," though.

When a game starts, a die stands still on a flat table. During the game, the die is tumbled in all directions by the dealer. You will win the game if you can predict the number seen on the top face at the time when the die stops tumbling.

Now you are requested to write a program that simulates the rolling of a die. For simplicity, we assume that the die neither slips nor jumps but just rolls on the table in four directions, that is, north, east, south, and west. At the beginning of every game, the dealer puts the die at the center of the table and adjusts its direction so that the numbers one, two, and three are seen on the top, north, and west faces, respectively. For the other three faces, we do not explicitly specify anything but tell you the golden rule: the sum of the numbers on any pair of opposite faces is always seven.

Your program should accept a sequence of commands, each of which is either "north", "east", "south", or "west". A "north" command tumbles the die down to north, that is, the top face becomes the new north, the north becomes the new bottom, and so on. More precisely, the die is rotated around its north bottom edge to the north direction and the rotation angle is 90 degrees. Other commands also tumble the die accordingly to their own directions. Your program should calculate the number finally shown on the top after performing the commands in the sequence. Note that the table is sufficiently large and the die never falls off during the game.

Input

The input consists of one or more command sequences, each of which corresponds to a single game. The first line of a command sequence contains a positive integer, representing the number of the following command lines in the sequence. You may assume that this number is less than or equal to 1024. A line containing a zero indicates the end of the input. Each command line includes a command that is one of `north`, `east`, `south`, and `west`. You may assume that no white space occurs in any lines.

Output

For each command sequence, output one line containing solely the number on the top face at the time when the game is finished.

Sample Input

```
1
north
3
north
east
south
0
```

Output for the Sample Input

```
5
1
```

Problem Source: Kyoto'99

```
/* @judge_id: 27516PA 10409 C++ */

/* judged correct by the online judge */

/*
* Valladolid problem 10409
* Die Game
*
* Coded by: Andy Martin
*
*/
#include <iostream>
#include <string>

using namespace std; //introduces namespace std

/* solve the problem by simple simulation */
int main ( void )
{
  int nmove, i;
  string s;
  /* FACES:
     2
   6 1 5
     4
     3
   */
  int face[7],t;

  cin >> nmove;
  while( cin && nmove )
  {
    face[1]=1;
    face[2]=2;
    face[6]=3;
    face[3]=6;
    face[4]=5;
    face[5]=4;
    for( i=0; i<nmove; i++ )
    {
      cin >> s;
      if( s == "south" )
      {
        t=face[1];
        face[1]=face[2];
        face[2]=face[3];
        face[3]=face[4];
        face[4]=t;
      }
      else if( s == "north" )
      {
        t=face[1];
        face[1]=face[4];
        face[4]=face[3];
        face[3]=face[2];
        face[2]=t;
      }
      else if( s == "west" )
      {
        t=face[1];
        face[1]=face[5];
        face[5]=face[3];
        face[3]=face[6];
        face[6]=t;
      }
      else if( s == "east" )
      {
        t=face[1];
        face[1]=face[6];
```

```
        face[6]=face[3];
        face[3]=face[5];
        face[5]=t;
      }
    }
    cout << face[1] << endl;
    cin >> nmove;
  }
  return 0;
}
```

Problem Statement

You just got a phone with a speed-dial feature, and you want to determine which numbers to store in the speed-dial memory. To do this, you look at last month's bill, and decide to use the numbers that appear most frequently on the bill.

Create a class NewPhone that contains the method bestNumbers, which takes three arguments:

`numbers`: A vector <string> representing all the numbers you called last month. Each string has the form `"NNN-NNN-NNNN"`, where N is a numerical digit ('0'-'9').

`frequencies`: A vector <int> indicating how many times each number was called last month. Element i of this vector <int> maps to element i of **`numbers`**.

`spaces`: The number of memory spaces available to use.

The method should return a vector <int> of the numbers to program into the memory spaces available, starting with the most frequently called number, and going in decreasing frequency order. If two or more numbers are called the same number of times, they should be returned in the order they were in the original input. The size of the return value should be equal to **`spaces`**, discarding any numbers that do not fit into the result. If two or more numbers have the same frequency, but not all of them will fit in the result, add the ones that were first in the original input (See example 3).

Definition

Class:	NewPhone
Method:	bestNumbers
Parameters:	vector <string>, vector <int>, int
Returns:	vector <string>
Method signature:	vector <string> bestNumbers(vector <string> numbers, vector <int> frequencies, int spaces)

(be sure your method is public)

Constraints

- **`numbers`** will have between 1 and 50 elements, inclusive.
- Each element in **`numbers`** will have the form `"NNN-NNN-NNNN"`, where each N is a digit ('0'-'9').
- There will be no repeated elements in **`numbers`**.
- **`frequencies`** will have the same number of elements as **`numbers`**.
- Each element of **`frequencies`** will be between 1 and 100, inclusive.
- **`spaces`** will be between 1 and length of **`numbers`**, inclusive.

Examples

0)

```
{"123-456-7890", "012-333-4455", "800-825-6699"}
{3,2,1}
3
```

```
Returns: { "123-456-7890",  "012-333-4455",  "800-825-6699" }
```

You made 6 calls last month, three of them were 123-456-7890, two of them were 012-333-4455, and the last was 800-825-6699. Given the frequency of the numbers, they are already in the order of most frequent to least frequent, and there are enough spaces to hold all the numbers.

1)

```
{"123-456-7890", "012-333-4455", "800-825-6699"}
{3,1,2}
3
```

```
Returns: { "123-456-7890",  "800-825-6699",  "012-333-4455" }
```

Now, the second and third numbers are out of order.

2)

```
{"123-456-7890", "012-333-4455", "800-825-6699"}
{3,1,2}
2
```

```
Returns: { "123-456-7890",  "800-825-6699" }
```

Now, there are only two spaces available for speed-dial numbers.

3)

```
{"123-456-7890", "012-333-4455", "800-825-6699", "333-333-3333"}
{3,1,3,3}
2
```

```
Returns: { "123-456-7890",  "800-825-6699" }
```

Three numbers were called with a frequency of 3. Since the first and third elements come before the fourth, they are programmed first, and there is not enough room for the fourth element in the memory spaces.

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

class NewPhone
{
  public:
    vector <string> bestNumbers(vector <string> numbers, vector <int> frequencie
s, int spaces)
    {
      vector <string> rv;

      int i,j;
      for( i=0;i<spaces;i++ )
      {
        int max=0;
        int ndx=0;
        for( j=0;j<frequencies.size();j++ )
        {
          if( frequencies[j] > max ) { max=frequencies[j]; ndx=j; }
        }
        rv.push_back( numbers[ndx] );
        numbers.erase( numbers.begin()+ndx );
        frequencies.erase( frequencies.begin()+ndx );
      }

      return rv;
    }
};


// Powered by FileEdit
```

Problem Statement

A health club chain allows its members to visit any of its many health club locations an unlimited number of times per day. The only constraining rule is, a customer can only visit one health club location per day, even though he or she may return to that location an unlimited number of times for the rest of that day.

Although the honor system has always worked quite well, the club wants to run some tests to see how many people really follow the rules. You are to write a program that takes the entrance log files from three different clubs (all logging the same day) and return a sorted list of the people who are not honest and went to more than one health club location in the same day.

The log files are represented as vector <string>'s where each element is the member name of a customer who entered that day. For example, if a customer showed up three times to one of the club locations that day, the member's name would appear three times in the corresponding vector <string>.

Definition

Class:	MemberCheck
Method:	whosDishonest
Parameters:	vector <string>, vector <string>, vector <string>
Returns:	vector <string>
Method signature:	vector <string> whosDishonest(vector <string> club1, vector <string> club2, vector <string> club3)

(be sure your method is public)

Notes

- **club1**, **club2**, and **club3** may contain a different number of elements.
- The same member name can appear multiple times in a single log file.
- The elements of the returned vector <string> should be sorted in lexicographic order (the order they would appear in a dictionary).
- Assume that two people with the same name are in fact the same person.

Constraints

- **club1**, **club2**, and **club3** each have between 1 and 50 elements, inclusive.
- Each element of **club1**, **club2**, and **club3** contains between 1 and 50 characters, inclusive..
- Each element of **club1**, **club2**, and **club3** consists only of uppercase letters ('A'-'Z').

Examples

0)

```
{"JOHN","JOHN","FRED","PEG"}
{"PEG","GEORGE"}
{"GEORGE","DAVID"}
Returns: { "GEORGE",  "PEG" }
```

"PEG" went to **club1** and **club2**, and "GEORGE" went to **club2** and **club3**.

1)

```
{"DOUG","DOUG","DOUG","DOUG","DOUG"}
{"BOBBY","BOBBY"}
{"JAMES"}
Returns: { }
```

Here, no one went to more than one club location.

2)

```
{"BOBBY"}
{"BOB","BOBBY"}
{"BOB"}
Returns: { "BOB",  "BOBBY" }
```

Note that "BOB" is sorted before "BOBBY"

3)

```
{"BOBBY","HUGH","LIZ","GEORGE"}
{"ELIZABETH","WILL"}
{"BOB","BOBBY","BOBBY","PAM","LIZ","BOBBY","BOBBY","WILL"}
Returns: { "BOBBY",  "LIZ",  "WILL" }
```

4)

```
{"JAMES","HUGH","HUGH","GEORGE","ELIZABETH","ELIZABETH","HUGH",
"DAVID","ROBERT","DAVID","BOB","BOBBY","PAM","JAMES","JAMES"}
{"BOBBY","ROBERT","GEORGE","JAMES","PEG","JAMES","DAVID","JOHN","LIZ",
"SANDRA","GEORGE","JOHN","GEORGE","ELIZABETH","LIZ","JAMES"}
{"ROBERT","ROBERT","ROBERT","SANDRA","PAM","BOB","LIZ","GEORGE"}
Returns:
{ "BOB",
 "BOBBY",
 "DAVID",
 "ELIZABETH",
 "GEORGE",
 "JAMES",
 "LIZ",
 "PAM",
 "ROBERT",
 "SANDRA" }
```

5)

```
{"LIZ","WILL","JAMES"}
{"JOHN","ROBERT","GEORGE","LIZ","PEG","HUGH","BOB","BOBBY","ROBERT","ELIZABETH","DAVID"}
{"PAM","DAVID","SANDRA","GEORGE","JOHN","ROBERT","SANDRA","GEORGE"}
Returns: { "DAVID",  "GEORGE",  "JOHN",  "LIZ",  "ROBERT" }
```

6)

```
{"PEG","ROBERT","PAM","JOHN","DAVID","JOHN","ROBERT",
"GEORGE","HUGH","WILL","JAMES","JAMES","BOBBY","BOBBY","SANDRA"}
{"SANDRA","BOB","PAM","JAMES","WILL","DAVID","BOBBY","GEORGE",
"WILL","LIZ","BOBBY","ROBERT","WILL","BOB","BOBBY","ELIZABETH","HUGH"}
{"WILL","PEG","ELIZABETH","DAVID","HUGH","BOBBY","JOHN","SANDRA","ELIZABETH",
"ELIZABETH","SANDRA","GEORGE","PAM","ELIZABETH","BOBBY","DAVID","PAM"}
Returns:
{ "BOBBY",
 "DAVID",
 "ELIZABETH",
 "GEORGE",
 "HUGH",
 "JAMES",
 "JOHN",
 "PAM",
 "PEG",
 "ROBERT",
 "SANDRA",
 "WILL" }
```

7)

```
{"AHHOZY","AHHAPLL","ASNV"}
{"AHDLTOE","AHUKPJ","AHDENCTPP","AHDENCJ","AHDLNZC","AHDLTOGG","AHHAPMBG",
"ALE","AHBHA","AHUKP","AHDQMILLP","AHDENEDY","AHDENEE","AHHOHVCX","AHISK",
"AHW","AQDB","AHUP","AQDBNPU","AGWZUV","AHHOSUW","AHXS","AHDENCP","AHDQM",
"AHDLTURV","AHBHVV","AHDQMILL","AHDQMD","AHH","AHDLTU","AHISFNO","AHURF",
"AH","AHHAPNQ","AQPL","AHDXL","AHDLTUGX","AHDLT","AHUKRC","AHDLTUGX",
"AQDTXYX","AGWZS"}
{"AHHAPMFF","AHURA","AHHOZ","AHISKH","AHUPR","AHHAPM","AHUKRHIN","AHHAP",
"AHDLTMO","AHDLTUJ","AHDQY","AHUK","AHDENEDY","AHWK","AHHOZGJJ","AHXS",
"AHDLTUREL","AHHOZQNL","AHHOSUWOS"}
Returns: { "AHDENEDY",  "AHXS" }
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

class MemberCheck
{
  public:
    vector <string> whosDishonest(vector <string> club1, vector <string> club2,
vector <string> club3)
    {
      vector <string> rv;
      int i,j;
      rv.clear();
      for( i=0;i<club1.size();i++ )
      {
        if( find( rv.begin(),rv.end(),club1[i] ) == rv.end() )
        {
          for( j=0;j<club2.size();j++ )
          {
            if( club1[i]==club2[j] )
            {
              rv.push_back(club1[i]);
              break;
            }
          }
          if( j==club2.size() )
          {
            for( j=0;j<club3.size();j++ )
            {
              if( club1[i]==club3[j] )
              {
                rv.push_back(club1[i]);
                break;
              }
            }
          }
        }
      }
      for( i=0;i<club2.size();i++ )
      {
        if( find( rv.begin(),rv.end(),club2[i] ) == rv.end() )
        {
          for( j=0;j<club3.size();j++ )
          {
            if( club2[i]==club3[j] )
            {
              rv.push_back(club2[i]);
              break;
            }
          }
        }
      }
      sort(rv.begin(),rv.end());
      return rv;
    }
};

// Powered by FileEdit
```

Problem Statement

A square *n* by *n* tile has *n*^2 one by one cells, some of which are colored. A number of identical tiles will be laid in a rectangular format, with the corner of each tile coinciding with the corners of three other tiles. The tiles will not be laid upside down.

```
 Legal Format      Illegal Format

_|___|___|___|_    _|   |___|   |_
 |   |   |   |      |___|   |___|
_|___|___|___|_    _|   |___|   |_
 |   |   |   |      |___|   |___|
_|___|___|___|_    _|   |___|   |_
 |   |   |   |      |   |   |   |
```

We say that a tile has the "edge-crossing" property if, no matter how the tiles are rotated when they are laid, every colored cell on the edge of one tile is adjacent (orthogonally or diagonally) to at least one colored cell on another tile. This is a desirable property since it helps to hide tile edges.

This picture shows 3 by 3 tiles, with the colored cells indicated by 'X' and the uncolored cells by ' '. Note that all the tiles have the same coloring pattern but that each tile can be laid in any of four possible rotations.

```
 |   |   |   |
-|---|---|---|-
 |XX | X | XX|
 |XXX|XX | XX|
 |   |XX | X |
-|---|---|---|-
 |   |   |   |
```

Create a class TileMatch that contains the method uncolor that takes a vector <string> giving the original **pattern** of colored cells on a tile as input and returns the minimum number of cells that we need to uncolor to give the tile the "edge-crossing" property.

pattern is a square collection of '-', denoting an uncolored cell, and 'X', denoting a colored cell. Each element of **pattern** gives the coloring of one row.

Definition

Class:	TileMatch
Method:	uncolor
Parameters:	vector <string>
Returns:	int
Method signature:	int uncolor(vector <string> pattern)
(be sure your method is public)	

Notes

- A colored cell on the edge of a tile must be uncolored if there is some way to rotate and arrange tiles so that the colored cell is not adjacent to any colored cells on an *adjacent* tile (vertically, horizontally, or diagonally).

- Some cells, which need not be uncolored at first, may have to be uncolored after some other cell is uncolored. See example 4.

Constraints

- **pattern** contains between 2 and 50 elements, inclusive
- the length of each string in **pattern** is the same as the number of strings in **pattern**
- each character in each element of **pattern** is either '-' or 'X'

Examples

0)

```
{"XXX",
 "XXX",
 "---"}
```

```
Returns: 5
```

Since the adjacent tiles may all be oriented so the adjacent edges have no color, you must uncolor all the edge cells. Leave the center cell colored -- you can always leave a cell colored when it is not on the edge of a tile.

1)

```
{"XXX",
 "XXX",
 "-X-"}
```

```
Returns: 0
```

Every adjacent tile will have a colored cell in the middle of each edge, and this will be adjacent to every cell in the edge.

2)

```
{"XXXX",
 "XXXX",
 "-X--",
 "XX--"}
```

```
Returns: 3
```

It is necessary to remove the color from the 3 colored corner cells. This gives the following tile with the "edge-crossing" property:

```
-XX-
XXXX
-X--
-X--
```

3)

```
{"XX--",
 "---X",
 "X---",
 "--X-"}
```

```
Returns: 0
```

4)

```
{"-XX-",
 "XXXX",
 "XXXX",
  "---X"}
Returns: 7
```

We end up having to uncolor all of the colored cells on the edge.

5)

```
 {"---X--",
  "-----X",
  "------",
  "------",
  "X-----",
  "----X-"}
Returns: 4
```

6)

```
 {"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXX-",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XX--X-X-XX-XX-XXXXX-X-XXX-X-X--XXXXXXXXXXXXXXXXXXX",
  "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"}
Returns: 170
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

bool cmpr( string a, string b, int p )
{
  int n=a.size();
  // manually reverse second string
  return (a[p] == 'X' && b[n-1-p] == '-' &&  b[n-1-(p-1)] == '-'
      && b[n-1-(p+1)] == '-');
}

bool corn_cmpr( string a, string b1, string b2, bool corndash )
{
  int n=a.size();
  return (a[0] == 'X' && b1[n-1] == '-' && b1[n-2] == '-' && b2[0] == '-'
      && b2[1] == '-' && corndash );
}

class TileMatch
{
  public:
    int uncolor(vector <string> pattern)
    {
      // top_string....r
      // .              i
      // r              g
      // t   PATTERN    h
      // s              t
      // _              _
      // t              s
      // f              t
      // e              r
      // l.gnirts_mottob
      // edges is in orientation of above
      // to compare, just consider reverse of edge you're looking at.
      // the internals of the pattern aren't even needed in this problem
      string edges[4];
      int i,j,k,l;
      bool cornerdash;
      const int n=pattern.size();

      for( i=0;i<n;i++ )
      {
        edges[0].push_back(pattern[0][i]);
        edges[1].push_back(pattern[i][n-1]);
        edges[2].push_back(pattern[n-1][n-1-i]);
        edges[3].push_back(pattern[n-1-i][0]);
      }
      cornerdash = (edges[0][0] == '-' || edges[1][0] == '-' ||
          edges[2][0] == '-' || edges[3][0] == '-' );
      for(i=0;i<4*(n-1); )
      {
        // do corners
        for(j=0;j<4;j++)
        {
          for(k=0;k<4;k++)
          {
            for(l=0;l<4;l++)
            {
              if( corn_cmpr( edges[j], edges[k], edges[l], cornerdash ) )
              {
                // make sure to set the corners in each string to dash
                edges[j][0]='-';
                edges[(j-1<0)?3:j-1][n-1]='-';
                // now that a corner is a dash, cornerdash is always true
                cornerdash=true;
```

```
                goto nextloop;
              }
            }
          }
        }
        // now just try internal points
        for(j=0;j<4;j++)
        {
          for(k=0;k<4;k++)
          {
            for(l=1;l<n-1;l++)
            {
              if( cmpr( edges[j], edges[k], l ) )
              {
                edges[j][l]='-';
                goto nextloop;
              }
            }
          }
        }
        break;
nextloop: i++;
      }
      return i;
    }
};


// Powered by FileEdit
```

1997/98 ACM International Collegiate Programming Contest
University of Ulm Local Contest

Problem C

Compromise

Source file: compromise.(c|C|pas)
Input file: compromise.in

In a few months the European Currency Union will become a reality. However, to join the club, the Maastricht criteria must be fulfilled, and this is not a trivial task for the countries (maybe except for Luxembourg). To enforce that Germany will fulfill the criteria, our government has so many wonderful options (raise taxes, sell stocks, revalue the gold reserves,...) that it is really hard to choose what to do.

Therefore the German government requires a program for the following task:
Two politicians each enter their proposal of what to do. The computer then outputs the longest common subsequence of words that occurs in both proposals. As you can see, this is a totally fair compromise (after all, a common sequence of words is something what both people have in mind).

Your country needs this program, so your job is to write it for us.

Input Specification

The input file will contain several test cases.
Each test case consists of two texts. Each text is given as a sequence of lower-case words, separated by whitespace, but with no punctuation. Words will be less than 30 characters long. Both texts will contain less than 100 words and will be terminated by a line containing a single '#'.
Input is terminated by end of file.

Output Specification

For each test case, print the longest common subsequence of words occuring in the two texts. If there is more than one such sequence, any one is acceptable. Separate the words by one blank. After the last word, output a newline character.

Sample Input

```
die einkommen der landwirte
sind fuer die abgeordneten ein buch mit sieben siegeln
um dem abzuhelfen
muessen dringend alle subventionsgesetze verbessert werden
#
die steuern auf vermoegen und einkommen
sollten nach meinung der abgeordneten
nachdruecklich erhoben werden
dazu muessen die kontrollbefugnisse der finanzbehoerden
```

```
dringend verbessert werden
#
```

Sample Output

```
die einkommen der abgeordneten muessen dringend verbessert werden
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <deque>

using namespace std;

enum { N, W, NW } b[100][100];
int c[100][100];

deque<string> x,y;

// prints an LCS given the precomputed b and c tables (see 515 book)
void print_lcs( ofstream & outfile, int i, int j )
{
  if( 0 == i || 0 == j ) return;
  if( NW == b[i][j] )
  {
    print_lcs(outfile,i-1,j-1);
    cout << x[i-1] << endl;
    outfile << x[i-1];
    if( c[i][j] == c[x.size()][y.size()] )
    {
      outfile << endl;
    }
    else
    {
      outfile << " ";
    }
  }
  else if( N == b[i][j] )
  {
    print_lcs( outfile, i-1, j );
  }
  else
  {
    print_lcs( outfile, i, j-1 );
  }
}


int main()
{
  ifstream infile( "compromise.in" );
  ofstream outfile( "compromise.out" );

  if( !infile || !outfile )
  {
    cerr << "file problems" << endl;
    return 1;
  }

  string inp;
  while( infile )
  {
    x.clear();
    y.clear();

    infile >> inp;
    if( !infile ) break;
    while( inp != "#" )
    {
      x.push_back(inp);
      infile >> inp;
    }
    infile >> inp;
    while( inp != "#" )
    {
      y.push_back(inp);
```

```
      infile >> inp;
    }

    if( x.size() >= 100 || y.size() >= 100 )
    {
      cerr << "input too large" << endl;
    }
    else
    {
      //setup tables for dynamic O(n+m) largest common subseqence algorithm
      int i,j;
      for( i=1;i<=x.size();i++ )
      {
        c[i][0] = 0;
      }
      for( j=0;j<=y.size();j++ )
      {
        cout << "0 ";
        c[0][j] = 0;
      }
      cout << endl;
      for( i=1;i<=x.size();i++ )
      {
        cout << "0 ";
        for( j=1;j<=y.size();j++ )
        {
          if( x[i-1] == y[j-1] )
          {
            c[i][j] = c[i-1][j-1]+1;
            cout << c[i][j] << "\\";
            b[i][j] = NW;
          }
          else if( c[i-1][j] >= c[i][j-1] )
          {
            c[i][j] = c[i-1][j];
            cout << c[i][j] << "|";
            b[i][j] = N;
          }
          else
          {
            c[i][j] = c[i][j-1];
            cout << c[i][j] << "-";
            b[i][j] = W;
          }
        }
        cout << endl;
      }
      print_lcs( outfile, x.size(), y.size() );
    }
  }
  return 0;
}
```

Problem C: Longest Common Subsequence

Sequence 1:

Sequence 2:

Given two sequences of characters, print the length of the longest common subsequence of both sequences. For example, the longest common subsequence of the following two sequences:

```
abcdgh
aedfhr
```

is `adh` of length 3.

Input consists of pairs of lines. The first line of a pair contains the first string and the second line contains the second string. Each string is on a separate line and consists of at most 1,000 characters

For each subsequent pair of input lines, output a line containing one integer number which satisfies the criteria stated above.

Sample input

```
a1b2c3d4e
zz1yy2xx3ww4vv
abcdgh
aedfhr
abcdefghijklmnopqrstuvwxyz
a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0y0z0
abcdefghijklmnzyxwvutsrqpo
opqrstuvwxyzabcdefghijklmn
```

Output for the sample input

```
4
3
26
14
```

Problem Setter: Piotr Rudnicki

```
/* @judge_id: 27516PA 10405 C++ */

/* judged correct by the online judge */
/* Written by: Ryan Gabbard */

#include <iostream>
#include <string>
#include <vector>

using namespace std; //introduces namespace std

long lcs(string x, string y);

long c[1000][1000];

// use the well-known quick LCS algorithim
long lcs(string x, string y) {
  long m=x.length(), n=y.length();

  for (int i=0; i<=m; i++) {
    c[i][0]=0;
  }

  for (int i=0; i<=n; i++) {
    c[0][i]=0;
  }

  for (int i=1; i<=m; i++) {
    for (int j=1; j<=n; j++) {
      if (x[i-1]==y[j-1]) {
        c[i][j]=c[i-1][j-1]+1;
        //b[i,j]=UP_LEFT;
      }
      else if (c[i-1][j]>=c[i][j-1]) {
        c[i][j]=c[i-1][j];
        //b[i][j]=UP;
      }
      else {
        c[i][j]=c[i][j-1];
        //b[i][j]=LEFT:
      }
    }
  }

  return c[m][n];
}

int main ( void )
{
  string a, b;

  // always read the problem statment well!! They allowed BLANK lines, which
  // represented empty strings, so use getline!
  getline(cin,a,'\n');
  getline(cin,b,'\n');

  while (!cin.eof()) {
    cout << lcs(a,b) << endl;

    getline(cin,a,'\n');
    getline(cin,b,'\n');
  }

  return 0;
}
```

Problem Statement

Many times, certain data file types can consist of large amounts of repeated data. For instance, images can have large runs of the same color. This can be easily compressed using a technique called run length encoding. With run length encoding, large amounts of repeated data are stored as the repeated data and the number of times to repeat it.

Create a class RunLengthEncode that contains the method encode which takes one argument:

`input`: a string to be encoded as described below

The return value should be a string which has been encoded with the following algorithm:

If any character is repeated more than 4 times, the entire set of repeated characters should be replaced with a slash '/', followed by a 2-digit number which is the length of the set of characters, and the character. For example, `"aaaaa"` would be encoded as `"/05a"`. Runs of 4 or less characters should not be replaced since performing the encoding would not decrease the length of the string.

Definition

Class:	RunLengthEncode
Method:	encode
Parameters:	string
Returns:	string
Method signature:	string encode(string input)

(be sure your method is public)

Notes

- Letters are case sensitive. For example `"AaAaAa"` cannot be encoded.
- You may only encode repeats of a single character, repeats of multiple characters cannot be encoded. For example `"ababababab"` cannot be encoded as `"/05ab"`.

Constraints

- **`input`** will have between 0 and 50 characters, inclusive.
- **`input`** will consist only of letters 'a' - 'z' and 'A' - 'Z', digits '0' - '9', the space character, and the characters in the following string: `"{}[]():;'+=.,"`. (quotes are for clarity only and cannot be in the input string)

Examples

0)

```
"aaaaa"
Returns: "/05a"
```

The example stated above

1)

```
"aaaa"
Returns: "aaaa"
```

Remember not to encode runs of length 4 or less.

2)

```
"abcabcabcabcabc"
Returns: "abcabcabcabcabc"
```

Do not encode repeated segments of more than one character

3)

```
"if(a){if(b){if(c){if(d){if(e){5 deeeeeeep}}}}}"
Returns: "if(a){if(b){if(c){if(d){if(e){5 d/07ep/05}"
```

4)

```
""
Returns: ""
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
#include <string>
#include <stdio.h>
#include <vector>
#include <iostream>

using namespace std;

class RunLengthEncode
{
  public:
    string encode(string input)
    {
      string rv;

      int rl=1;
      char last=input[0];
      int i;
      char t[8];
      if( input.size() <= 4 ) return input;
      for( i=0;i<input.size();i++ )
      {
        if( input[i] == last && i!=0 )
        {
          rl++;
        }
        if( input[i] != last )
        {
          if( rl > 4 )
          {
            sprintf(t,"/%02d%c",rl,last);
            rv=rv+t;
          }
          else
          {
            for( int j=0;j<rl;j++ )
            {
              rv.push_back(last);
            }
          }
          rl=1;
        }
        last=input[i];
      }
      if( rl > 4 )
      {
        sprintf(t,"/%02d%c",rl,last);
        rv=rv+t;
      }
      else
      {
        for( int j=0;j<rl;j++ )
        {
          rv.push_back(last);
        }
      }
      return rv;
    }
};


// Powered by FileEdit
```

Problem Statement

When new files are written to a hard disk, oftentimes they are broken up into fragments and stored in different physical locations on the drive. "Defragmentation" is the process of reordering these fragments such that they are all adjacent on the hard disk. Not only does this increase performance of the computer, but it helps prevent new files from being fragmented in the future.

Given a string **`FAT`**, representing the hard disk's file allocation table, return a int indicating the number of clusters which will be moved during the defragmentation process. A typical file allocation table might look like the following:

```
"C.RC.C.CC..C....RCC.."
```

A `'C'` represents a cluster on the disk, an `'R'` represents a read-only cluster (one that cannot be moved), and a period (`'.'`) represents a position on the disk which does not correspond to a file. The defragmentation algorithm reads the FAT from right to left, looking for a cluster to move. Once it finds one, it starts searching from left to right (starting at position 0) for an empty space. In the above example, the rightmost 'C' is at position 18 in the string, and the leftmost period is at position 1. The defragmenter then moves the cluster to the empty space, resulting in the new FAT below:

```
"CCRC.C.CC..C....RC..."
```

Now, the rightmost cluster is at position 17, and the leftmost empty space is at position 4. After moving this cluster, the FAT is arranged as follows:

```
"CCRCCC.CC..C....R...."
```

The next cluster is at position 16, but it is read-only and cannot be moved. Therefore, the defragmenter continues past it to the cluster at position 11, and moves it to the empty space at position 6.

```
"CCRCCCCCC.......R...."
```

When the defragmenter now tries to move the cluster at position 8, it discovers that there is no empty space to the left of 8 in which to place it. This means that defragmentation is complete. Three clusters were moved, so for this example your method would return 3.

Definition

Class:	Defragment
Method:	clustersMoved
Parameters:	string
Returns:	int
Method signature:	int clustersMoved(string FAT)

(be sure your method is public)

Constraints

- **`FAT`** will contain between 1 and 50 characters, inclusive.
- **`FAT`** will contain only periods ('.') and the characters 'C' and 'R'.

Examples

0)

```
"C.RC.C.CC..C....RCC.."
Returns: 3
```

This is the example from above.

1)

```
"CCRCCCCCC.......R...."
Returns: 0
```

At this stage, the hard disk is already defragmented as much as possible. There are no clusters that need to be moved.

2)

```
"..........CCCCCC"
Returns: 6
```

Each of the six clusters on the right will need to be moved to the left.

3)

```
".RR...RR.RR....RR....R......R.R.RRRR.RR...RRR....."
Returns: 0
```

The disk is highly fragmented, but each cluster is marked as read-only, so no move operations can be performed.

4)

```
"RR.R..C.CC..CRRRRRC.C.C.RCC..CR....RC.C.."
Returns: 7
```

This problem statement is the exclusive and proprietary property of TopCoder, Inc. Any unauthorized use or reproduction of this information without the prior written consent of TopCoder, Inc. is strictly prohibited. (c)2002, TopCoder, Inc. All rights reserved.

```
// solution to SRM 134 DIV 2 (250)
#include <string>
#include <vector>
#include <iostream>

using namespace std;

class Defragment
{
  public:
    int clustersMoved(string FAT)
    {
      int rv=0;
      int l=FAT.size();
      int c=FAT.find_last_of('C');
      int s=FAT.find_first_of('.');
      while( c >= 0 && c < l && s >= 0 && s < l && s < c )
      {
        rv++;
        FAT[s]='C';
        FAT[c]=' ';
        c=FAT.find_last_of('C');
        s=FAT.find_first_of('.');
      }
      return rv;
    }
};

// Powered by FileEdit
```

Telephone Tangles

A large company wishes to monitor the cost of phone calls made by its personnel. To achieve this the PABX logs, for each call, the number called (a string of up to 15 digits) and the duration in minutes. Write a program to process this data and produce a report specifying each call and its cost, based on standard Telecom charges.

International (IDD) numbers start with two zeroes (00) followed by a country code (1-3 digits) followed by a subscriber's number (4-10 digits). National (STD) calls start with one zero (0) followed by an area code (1-5 digits) followed by the subscriber's number (4-7 digits). The price of a call is determined by its destination and its duration. Local calls start with any digit other than 0 and are free.

Input

Input will be in two parts. The first part will be a table of IDD and STD codes, localities and prices as follows:

> Code △ Locality name$price in cents per minute

where △ represents a space. Locality names are 25 characters or less. This section is terminated by a line containing 6 zeroes (000000).

The second part contains the log and will consist of a series of lines, one for each call, containing the number dialled and the duration. The file will be terminated a line containing a single `#`. The numbers will not necessarily be tabulated, although there will be at least one space between them. Telephone numbers will not be ambiguous.

Output

Output will consist of the called number, the country or area called, the subscriber's number, the duration, the cost per minute and the total cost of the call, as shown below. Local calls are costed at zero. If the number has an invalid code, list the area as ``Unknown'' and the cost as -1.00.

Sample input

```
088925 Broadwood$81
03 Arrowtown$38
0061 Australia$140
000000
031526        22
0061853279  3
0889256287213   122
779760 1
002832769 5
#
```

Sample output

```
1               17                               51   56    62     69
031526          Arrowtown                      1526   22  0.38   8.36
0061853279      Australia                    853279    3  1.40   4.20
0889256287213   Broadwood                   6287213  122  0.81  98.82
779760          Local                        779760    1  0.00   0.00
002832769       Unknown                                5         -1.00
```

```c
/* @judge_id: 27516PA 139 C */

/* solution to problem 139, telephone tangles
 *
 * Coded by: Andy Martin
 * Assited by: Bill Wyatt
 *
 */

#include <stdio.h>
#include <string.h>

typedef struct loc
{
  int cl;
  char code[7];
  char name[26];
  float cents;
} loc_t;

loc_t locs[4096];

int main()
{
  int nloc=0,t,numlen;
  char num[16];
  int min, i, sl;

  scanf( "%s", locs[nloc].code );
  while( strcmp( "000000", locs[nloc].code ) )
  {
    scanf( " %25[^$]$%d\n",locs[nloc].name,&t);
    locs[nloc].cl=strlen(locs[nloc].code);
    locs[nloc].cents = t/100.0;
    scanf( "%s", locs[++nloc].code );
  }

  scanf( "%s", num );
  /*
  printf("12345678901234567890123456789012345678901234567890123456789012345678901234567890
\n");
   */
/*
  printf( "1               17                               51   56    62     69
\n");
 */
  while( *num != '#' )
  {
    numlen=strlen(num);
    scanf( "%d\n", &min );
    for( i=0;i<nloc;i++ )
    {
      if( strncmp( locs[i].code, num, locs[i].cl ) == 0 )
      {
        if( locs[i].code[1] == '0' )
        {
          if( numlen - locs[i].cl >= 4 && numlen - locs[i].cl <= 10 )
          {
            break;
          }
        }
        else
        {
          if( numlen - locs[i].cl >= 4 && numlen - locs[i].cl <= 7 )
          {
            break;
          }
        }
      }
```

```c
    }

    if( i < nloc )
    {
      printf( "%-16s%-25s%11s%5d%6.2f%7.2f\n",
          num, locs[i].name, num+locs[i].cl, min,
          locs[i].cents, min*locs[i].cents );
    }
    else if( *num == '0' )
    {
      printf( "%-16s%-25s%11s%5d%6s%7.2f\n",
          num, "Unknown", "", min, "", -1.0 );
    }
    else
    {
      printf( "%-16s%-25s%11s%5d%6.2f%7.2f\n",
          num, "Local", num, min, 0.0, 0.0 );
    }

    scanf( "%s", num );
  }
  return 0;
}
```

Problem G
A Linking Loader
Input File: linker.in

An *object module* is produced by a compiler as a result of processing a source program. A *linking loader* (or just a *linker*) is used to combine the multiple object modules used when a program contains several separately compiled modules. Two of its primary tasks are to relocate the code and data in each object module (since the compiler does not know where in memory a module will be placed), and to resolve symbolic references from one module to another. For example, a main program may reference a square root function called *sqrt*, and that function may be defined in a separate source module. The linker will then minimally have to assign addresses to the code and data in each module, and put the address of the *sqrt* function in the appropriate location(s) in the main module's code.

An object module contains (in order) zero or more *external symbol definitions*, zero or more *external symbol references*, zero or more bytes of code and data (that may include references to the values of external symbols), and an end of module marker. In this problem, an object module is represented as a sequence of text lines, each beginning with a single uppercase character that characterizes the remainder of the line. The format of each of these lines is as follows. Whitespace (one or more blanks and/or tab characters) will appear between the fields in these lines. Additional whitespace may follow the last field in each line.

- A line of the form "D *symbol offset*" is an external symbol definition. It defines *symbol* as having the address *offset* bytes greater than the address where the first byte of code and data for the current object module is located by the linker. A *symbol* is a string of no more than eight upper case alphabetic characters. The *offset* is a hexadecimal number with no more than four digits (using only upper case alphabetic characters for the digits A through F). For example, in a module that is loaded starting at the address 100_{16}, the line "D START 5C" indicates that the symbol START is defined as being associated with the address $15C_{16}$. The number of D lines in a test case is at most 100.
- A line of the form "E *symbol*" is an external symbol reference, and indicates that the value of *symbol* (presumably defined in another object module) may be referenced as part of the code and data for the current module. For example, the line "E START" indicates that the value of the symbol START (that is, the address defined for it) may be used as part of the code and data for the module. Each of the "E" lines for each module is numbered sequentially, starting with 0, so they can be referenced in the "C" lines.
- A line of the form "C *n* $byte_1$ $byte_2$ … $byte_n$" specifies the first or next *n* bytes of code and data for the current module. The value *n* is specified as a one or two digit hexadecimal number, and will be no larger than 10 hexadecimal. Each *byte* is either a one or two digit hexadecimal number, or a dollar sign. The first byte following a dollar sign (always on the same line) gives the 0-origin index of an external symbol reference for this module, and identifies the symbol which is to have its 16-bit value inserted at the current point in the linked program (that is, in the location indicated by the dollar sign and the following byte). The high-order byte is placed in the location indicated by the dollar sign. The values specified for the other bytes (those not following a dollar sign) are loaded into sequential memory locations, starting with the first (lowest) unused memory location. For example, the line "C 4 25 $ 0 37" would cause the values 25_{16} 01_{16} $5C_{16}$ and 37_{16} to be placed in the next four unused memory locations, assuming the first "E" line for the current module specified a symbol defined as having the address $15C_{16}$. If the 0-origin index of the external symbol reference is an undefined symbol, the 16-bit value inserted at the current point in the linked program is 0000_{16}.
- A line of the form "Z" marks the end of an object module.

You may assume that no address requires more than four hexadecimal digits. Lines are always given in the order shown above. There are no syntax errors in the input.

Input

This problem has multiple input cases. The input for each case is one or more object modules, in sequence, that are to be linked, followed by a line beginning with a dollar sign. The first address at which code is to be loaded in each case is 100_{16}.

The last case will be followed by a line containing only a dollar sign.

Output

For each case, print the case number (starting with 1), the 16-bit checksum of the loaded bytes (as described below), and the load map showing the address of each externally defined or referenced symbol, in ascending order of symbol name. For undefined symbols, print the value as four question marks, but use zero as the symbol's value when it is referenced in "C" lines. If a symbol is defined more than once, print "M" following the address shown in the load map, and use the value from the first definition encountered in any object module to satisfy external references. Format the output exactly as shown in the samples.

The 16-bit checksum is computed by first setting it to zero. Then, for each byte assigned to a memory location by the loader, in increasing address order, circularly left shift the checksum by one bit, and add the byte from the memory location, discarding any carry out of the low-order 16 bits.

Sample Input

```
D MAIN 0
D END 5
C 03 01 02 03
C 03 04 05 06
Z
$
D ENTRY 4
E SUBX
E SUBY
C 10 1 2 3 4 5 $ 0 6 7 8 9 A B C D E
C 8 10 20 30 40 50 60 70 80
C 8 90 A0 B0 C0 D0 E0 $ 1
C 5 $ 0 FF EE DD
Z
D SUBX 01
C 06 A B C D E F
Z
D SUBX 05
C 06 51 52 53 54 55 56
Z
$
$
```

Output for the Sample Input

```
Case 1: checksum = 0078
 SYMBOL   ADDR
--------  ----
END       0105
MAIN      0100

Case 2: checksum = 548C
 SYMBOL   ADDR
--------  ----
ENTRY     0104
SUBX      0126 M
SUBY      ????
```

```
/* Problem G of the ACM/ICPC 2003 World Finals
 * Solved by: C. Andy Martin
 */
/* This is the actual solution that I submitted last and was eventually
 * judged correct on. It is not annotated as the algorithm is fairly
 * straightforward. This is mainly a text processing/data structures
 * problem. */
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cstdio>
#include <iomanip>
#include <algorithm>

using namespace std;

class symbol
{
 public:
  int times_defined;
  int offset;
  string name;
  symbol() { times_defined=0; offset=0; name=""; }
  bool operator<( const symbol &r ) const { return name < r.name; }
  bool operator==( const symbol &r ) const { return name==r.name; }
};

class object
{
 public:
  vector <string> external;
  // −1 means insert symbol
  vector <int> code;
};

int main()
{
  ifstream infile( "linker.in" );
  char next;
  int cnt=0;

  infile >> next;
  while( next != '$' )
  {
    vector <symbol> table;
    vector <object> objs;
    int num_objs=0;
    int cur_add=0x100;
    object tnewobj;
    objs.push_back( tnewobj );
    while( next != '$' )
    {
      if( next == 'D' )
      {
        symbol newsym;
        char temp[8];
        infile >> newsym.name;
        infile >> temp;
        newsym.times_defined=1;
        sscanf( temp, "%x", &newsym.offset );
        newsym.offset+=cur_add;
        vector <symbol>::iterator it=find( table.begin(), table.end(), newsym );
        if( it == table.end() )
        {
          table.push_back(newsym);
        }
        else
        {
```

```
          if( (*it).times_defined == 0 )
          {
            (*it).offset=newsym.offset;
          }
          (*it).times_defined++;
        }
      }
      else if( next == 'E' )
      {
        string symname;
        infile >> symname;
        objs[num_objs].external.push_back(symname);
        symbol tsym;
        tsym.name = symname;
        tsym.times_defined=0;
        tsym.offset=0;
        vector <symbol>::iterator it=find( table.begin(), table.end(), tsym );
        if( it == table.end() )
        {
          table.push_back( tsym );
        }
      }
      else if( next == 'C' )
      {
        int n;
        char temp2[8];
        infile >> temp2;
        sscanf( temp2, "%x", &n );
        for( int i=0;i<n;i++ )
        {
          char temp[8];
          int cpie;
          infile >> temp;
          if( temp[0] == '$' )
          {
            cpie=−1;
          }
          else
          {
            sscanf( temp, "%x", &cpie );
          }
          objs[num_objs].code.push_back(cpie);
        }
      }
      else if( next == 'Z' )
      {
        cur_add+=objs[num_objs].code.size();
        object newobj;
        num_objs++;
        objs.push_back( newobj );
      }
      infile >> next;
    }
    int checksum=0;
    for( int i=0;i<num_objs;i++ )
    {
      for( int j=0;j<objs[i].code.size();j++ )
      {
        if( objs[i].code[j] != −1 )
        {
          checksum = (checksum<<1)%0x10000 + (checksum<<1)/0x10000;
          checksum += objs[i].code[j];
          checksum = checksum & 0xFFFF;
        }
        else
        {
          int i1, i2;
          symbol tsymbol;
          tsymbol.name=objs[i].external[objs[i].code[j+1]];
          vector <symbol>::iterator it = find(table.begin(),table.end(),tsymbol)
;
          if( it == table.end() || (*it).times_defined == 0 )
          {
            i1=i2=0;
          }
          else
          {
            i1 = (*it).offset >> 8;
            i2 = (*it).offset & 0xFF;
          }
          checksum = (checksum<<1)%0x10000 + (checksum<<1)/0x10000;
          checksum += i1;
          checksum = checksum & 0xFFFF;
          checksum = (checksum<<1)%0x10000 + (checksum<<1)/0x10000;
          checksum += i2;
          checksum = checksum & 0xFFFF;
          j++;
        }
      }
    }
    char tttt[20];
    sprintf(tttt,"%04X",checksum);
    cout << "Case " << ++cnt << ": checksum = " << tttt << endl;
    sort(table.begin(),table.end());
    cout << " SYMBOL  ADDR" << endl;
    cout << "-------- ----" << endl;
    for( int i=0;i<table.size();i++ )
    {
      char temp22[16];
      sprintf(temp22,"%-8s ",table[i].name.c_str());
      cout << temp22;
      if( table[i].times_defined == 0 )
      {
        cout << "????";
      }
      else
      {
        char temper[8];
        sprintf(temper,"%04X",table[i].offset);
        cout << temper;
        if( table[i].times_defined > 1 )
        {
          cout << " M";
        }
      }
      cout << endl;
    }

    infile >> next;
    if( next != '$' )
    {
      cout << endl;
    }
  }

  return 0;
}
```