Top coder

Algorithm Tutorials

part #0

Step by step guide for getting professional
or
How to get closer to World Finals?

Introduction

The first step towards an understanding of why the study and knowledge of algorithms are so important is to define exactly what we mean by an algorithm. According to the popular algorithms textbook Introduction to Algorithms (Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein), "an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output." In other words, algorithms are like road maps for accomplishing a given, well-defined task. So, a chunk of code that calculates the terms of the Fibonacci sequence is an implementation of a particular algorithm. Even a simple function for adding two numbers is an algorithm in a sense, albeit a simple one.

Some algorithms, like those that compute the Fibonacci sequences, are intuitive and may be innately embedded into our logical thinking and problem solving skills. However, for most of us, complex algorithms are best studied so we can use them as building blocks for more efficient logical problem solving in the future. In fact, you may be surprised to learn just how many complex algorithms people use every day when they check their e-mail or listen to music on their computers. This article will introduce some basic ideas related to the analysis of algorithms, and then put these into practice with a few examples illustrating why it is important to know about algorithms.

Runtime Analysis

One of the most important aspects of an algorithm is how fast it is. It is often easy to come up with an algorithm to solve a problem, but if the algorithm is too slow, it's back to the drawing board. Since the exact speed of an algorithm depends on where the algorithm is run, as well as the exact details of its implementation, computer scientists typically talk about the runtime relative to the size of the input. For example, if the input consists of N integers, an algorithm might have a runtime proportional to N^2, represented as $O(N^2)$. This means that if you were to run an implementation of the algorithm on your computer with an input of size N, it would take $C*N^2$ seconds, where C is some constant that doesn't change with the size of the input.

However, the execution time of many complex algorithms can vary due to factors other than the size of the input. For example, a sorting algorithm may run much faster when given a set of integers that are already sorted than it would when given the same set of integers in a random order. As a result, you often hear people talk about the worst-case runtime, or the average-case runtime. The worst-case runtime is how long it would take for the algorithm to run if it were given the most insidious of all possible inputs. The average-case runtime is the average of how long it would take the algorithm to run if it were given all possible inputs. Of the two, the worst-case is often easier to reason about, and therefore is more frequently used as a benchmark for a given algorithm. The process of determining the worst-case and average-case runtimes for a given algorithm can be tricky, since it is usually impossible to run an algorithm on all possible inputs. There are many good online resources that can help you in estimating these values.

Approximate completion time for algorithms, N = 100

O(Log(N))	10^{-7} seconds
O(N)	10^{-6} seconds
O(N*Log(N))	10^{-5} seconds
$O(N^2)$	10^{-4} seconds
$O(N^6)$	3 minutes
$O(2^N)$	10^{14} years.
O(N!)	10^{142} years.

Sorting

Sorting provides a good example of an algorithm that is very frequently used by computer scientists. The simplest way to sort a group of items is to start by removing the smallest item from the group, and put it first. Then remove the next smallest, and put it next and so on. Unfortunately, this algorithm is $O(N^2)$, meaning that the amount of time it takes is proportional to the number of items squared. If you had to sort a billion things, this algorithm would take around 10^{18} operations. To put this in perspective, a desktop PC can do a little bit over 10^9 operations per second, and would take years to finish sorting a billion things this way.

Luckily, there are a number of better algorithms (quicksort, heapsort and mergesort, for example) that have been devised over the years, many of which have a runtime of O(N * Log(N)). This brings the number of operations required to sort a billion items down to a reasonable number that even a cheap desktop could perform. Instead of a billion squared operations (10^{18}) these algorithms require only about 10 billion operations (10^{10}), a factor of 100 million faster.

Shortest Path

Algorithms for finding the shortest path from one point to another have been researched for years. Applications abound, but lets keep things simple by saying we want to find the shortest path from point A to point B in a city with just a few streets and intersections. There are quite a

few different algorithms that have been developed to solve such problems, all with different benefits and drawbacks. Before we delve into them though, lets consider how long a naive algorithm - one that tries every conceivable option - would take to run. If the algorithm considered every possible path from A to B (that didn't go in circles), it would not finish in our lifetimes, even if A and B were both in a small town. The runtime of this algorithm is exponential in the size of the input, meaning that it is $O(C^N)$ for some C. Even for small values of C, C^N becomes astronomical when N gets even moderately large.

One of the fastest algorithms for solving this problem has a runtime of O(E*V*Log(V)), where E is the number of road segments, and V is the number of intersections. To put this in perspective, the algorithm would take about 2 seconds to find the shortest path in a city with 10,000 intersections, and 20,000 road segments (there are usually about 2 road segments per intersection). The algorithm, known as Djikstra's Algorithm, is fairly complex, and requires the use of a data structure known as a priority queue. In some applications, however, even this runtime is too slow (consider finding the shortest path from New York City to San Francisco - there are millions of intersections in the US), and programmers try to do better by using what are known as heuristics. A heuristic is an approximation of something that is relevant to the problem, and is often computed by an algorithm of its own. In the shortest path problem, for example, it is useful to know approximately how far a point is from the destination. Knowing this allows for the development of faster algorithms (such as A*, an algorithm that can sometimes run significantly faster than Djikstra's algorithm) and so programmers come up with heuristics to approximate this value. Doing so does not always improve the runtime of the algorithm in the worst case, but it does make the algorithm faster in most real-world applications.

Approximate algorithms

Sometimes, however, even the most advanced algorithm, with the most advanced heuristics, on the fastest computers is too slow. In this case, sacrifices must be made that relate to the correctness of the result. Rather than trying to get the shortest path, a programmer might be satisfied to find a path that is at most 10% longer than the shortest path.

In fact, there are quite a few important problems for which the best-known algorithm that produces an optimal answer is insufficiently slow for most purposes. The most famous group of these problems is called NP, which stands for non-deterministic polynomial (don't worry about what that means). When a problem is said to be NP-complete or NP-hard, it mean no one knows a good way to solve them optimally. Furthermore, if someone did figure out an efficient algorithm for one NP-hard problem, that algorithm would be applicable to all NP-hard problems.

A good example of an NP-hard problem is the famous traveling salesman problem. A salesman wants to visit N cities, and he knows how long it takes to get from each city to each other city. The question is "how fast can he visit all of the cities?" Since the fastest known algorithm for solving this problem is too slow - and many believe this will always be true - programmers look for sufficiently fast algorithms that give good, but not optimal solutions.

Random Algorithms

Yet another approach to some problems is to randomize an algorithm is some way. While doing so does not improve the algorithm in the worst case, it often makes very good algorithms in the average case. Quicksort is a good example of an algorithm where randomization is often used. In the worst case, quicksort sorts a group of items in $O(N^2)$, where N is the number of items. If randomization is incorporated into the algorithm, however, the chances of the worst case actually occurring become diminishingly small, and on average, quicksort has a runtime of O(N*Log(N)). Other algorithms guarantee a runtime of O(N*Log(N)), even in the worst case, but they are slower in the average case. Even though both algorithms have a runtime proportional to N*Log(N), quicksort has a smaller constant factor - that is it requires C*N*Log(N) operations, while other algorithms require more like 2*C*N*Log(N) operations.

Another algorithm that uses random numbers finds the median of a group of numbers with an average runtime of O(N). This is a significant improvement over sorting the numbers and taking the middle one, which takes O(N*Log(N)). Furthermore, while deterministic (non-random) algorithms exist for finding the median with a runtime of O(N), the random algorithm is attractively simple, and often faster than the deterministic algorithms.

The basic idea of the median algorithm is to pick one of the numbers in the group at random, and count how many of the numbers in the group are less than it. Lets say there are N numbers, and K of them are less than or equal to the number we picked at random. If K is less than half of N, then we know that the median is the $(N/2-K)^{th}$ number that is greater than the random number we picked, so we discard the K numbers less than or equal to the random number. Now, we want to find the $(N/2-K)^{th}$ smallest number, instead of the median. The algorithm is the same though, and we simply pick another number at random, and repeat the above steps.

Compression

Another class of algorithm deals with situations such as data compression. This type of algorithm does not have an expected output (like a sorting algorithm), but instead tries to optimize some other criteria. In the case of data compression, the algorithm (LZW, for instance) tries to make the data use as few bytes as possible, in such a way that it can be decompressed to its original form. In some cases, this type of algorithm will use the same techniques as other algorithms, resulting in output that is good, but potentially sub-optimal. JPG and MP3 compression, for example, both compress data in a way that makes the final result somewhat lower quality than the original, but they create much smaller files. MP3 compression does not retain every feature of the original song file, but it attempts to maintain enough of the details to capture most of the quality, while at the same time ensuring the significantly reduced file size that we all know and love. The JPG image file format follows the same principle, but the details are significantly different since the goal is image rather than audio compression.

The Importance of Knowing Algorithms
As a computer scientist, it is important to understand all of these types of algorithms so that one can use them properly. If you are working on an important piece of software, you will likely need to be able to estimate how fast it is going to run. Such an estimate will be less accurate without an understanding of runtime analysis. Furthermore, you need to understand the details of the algorithms involved so that you'll be able to predict if there are special cases in which the software won't work quickly, or if it will produce unacceptable results.

Of course, there are often times when you'll run across a problem that has not been previously studied. In these cases, you have to come up with a new algorithm, or apply an old algorithm in a new way. The more you know about algorithms in this case, the better your chances are of finding a good way to solve the problem. In many cases, a new problem can be reduced to an old problem without too much effort, but you will need to have a fundamental understanding of the old problem in order to do this.

As an example of this, lets consider what a switch does on the Internet. A switch has N cables plugged into it, and receives packets of data coming in from the cables. The switch has to first analyze the packets, and then send them back out on the correct cables. A switch, like a computer, is run by a clock with discrete steps - the packets are send out at discrete intervals, rather than continuously. In a fast switch, we want to send out as many packets as possible during each interval so they don't stack up and get dropped. The goal of the algorithm we want to develop is to send out as many packets as possible during each interval, and also to send them out so that the ones that arrived earlier get sent out earlier. In this case it turns out that an algorithm for a problem that is known as "stable matching" is directly applicable to our problem, though at first glance this relationship seems unlikely. Only through pre-existing algorithmic knowledge and understanding can such a relationship be discovered.

More Real-world Examples
Other examples of real-world problems with solutions requiring advanced algorithms abound. Almost everything that you do with a computer relies in some way on an algorithm that someone has worked very hard to figure out. Even the simplest application on a modern computer would not be possible without algorithms being utilized behind the scenes to manage memory and load data from the hard drive.

There are dozens of applications of complicated algorithms, but I'm going to discuss two problems that require the same skills as some past TopCoder problems. The first is known as the maximum flow problem, and the second is related to dynamic programming, a technique that often solves seemingly impossible problems in blazing speed.

Maximum Flow
The maximum flow problem has to do with determining the best way to get some sort of stuff from one place to another, through a network of some sort. In more concrete terms, the problem first arose in relation to the rail networks of the Soviet Union, during the 1950's. The US wanted to know how quickly the Soviet Union could get supplies through its rail network to its satellite states in Eastern Europe.

In addition, the US wanted to know which rails it could destroy most easily to cut off the satellite states from the rest of the Soviet Union. It turned out that these two problems were closely related, and that solving the max flow problem also solves the min cut problem of figuring out the cheapest way to cut off the Soviet Union from its satellites.

The first efficient algorithm for finding the maximum flow was conceived by two Computer Scientists, named Ford and Fulkerson. The algorithm was subsequently named the Ford-Fulkerson algorithm, and is one of the more famous algorithms in computer science. In the last 50 years, a number of improvements have been made to the Ford-Fulkerson algorithm to make it faster, some of which are dauntingly complex.

Since the problem was first posed, many additional applications have been discovered. The algorithm has obvious relevance to the Internet, where getting as much data as possible from one point to another is important. It also comes up in many business settings, and is an important part of operations research. For example, if you have N employees and N jobs that need to be done, but not every employee can do every job, the max flow algorithm will tell you how to assign your N employees to jobs in such a way that every job gets done, provided that's possible. Graduation, from SRM 200, is a good example of a TopCoder problem that lends itself to a solution using max flow.

Sequence comparison
Many coders go their entire careers without ever having to implement an algorithm that uses dynamic programming. However, dynamic programming pops up in a number of important algorithms. One algorithm that most programmers have probably used, even though they may not have known it, finds differences between two sequences. More specifically, it calculates the minimum number of insertions, deletions, and edits required to transform sequence A into sequence B.

For example, lets consider two sequences of letters, "AABAA" and "AAAB". To transform the first sequence into the second, the simplest thing to do is delete the B in the middle, and change the final A into a B. This algorithm has many applications, including some DNA problems and plagiarism detection. However, the form in which many programmers use it is when comparing two versions of the same source code file. If the elements of the sequence are lines in the file, then this algorithm can tell a programmer which lines of code were removed, which ones were inserted, and which ones were modified to get from one version to the next.

Without dynamic programming, we would have to consider a - you guessed it - exponential number of transformations to get from one sequence to the other. As it is, however, dynamic programming makes for an algorithm with a runtime of only O(N*M), where N and M are the numbers of elements in the two sequences.

Conclusion

The different algorithms that people study are as varied as the problems that they solve. However, chances are good that the problem you are trying to solve is similar to another problem in some respects. By developing a good understanding of a large range of algorithms, you will be able to choose the right one for a problem and apply it properly. Furthermore, solving problems like those found in TopCoder's competitions will help you to hone your skills in this respect. Many of the problems, though they may not seem realistic, require the same set of algorithmic knowledge that comes up every day in the real world.

How many times has this happened to you: you register for the SRM, go into your assigned room when the system tells you to, and when the match starts, you open the 250... and find it incomprehensible.

Maybe it's never happened to you. You may be lucky, or you may already be extremely skilled. However, many experienced competitors (yes, reds too) might end up just staring at a problem for a long time. This is a pretty serious issue. How can you solve the problem if you have no idea what it's asking you to do?

Knowing your way around the various pieces will go a long way towards helping you understand what the problem is saying.

The Parts of a Problem Statement

Let's look at the composition of a typical TopCoder problem statement. First off is the introduction. Usually, a problem will be led off with a high-level description of a situation. This description may tie into real-life ideas and topics or it may just be a completely fictional story, serving only as some sort of context. For many problems the back-story itself is not particularly important in understanding the actual problem at hand.

Next comes the definition section. It gives you the skeleton of the solution you need to write: class name, method name, arguments, and return type, followed by the complete method signature. At minimum, you will need to declare a class with the given name, containing a method which conforms to the given method signature. The syntax given will always be correct for your chosen language.

Sometimes notes follow the method definition. They tend to be important reminders of things that you should pay attention to but might have missed, or they can also be things that are helpful background knowledge that you might not know beforehand. If the notes section appears, you should make sure to read it - usually the information contained within is extremely important.

The constraints section is arguably the most important. It lists specific constraints on the input variables. This lets you know crucial details such as how much memory to allocate or how efficient your algorithm will have to be.

Finally, a set of examples is provided. These give sample inputs against which you can test your program. The given parameters will be in the correct order, followed by the expected return value and, optionally, an explanation of the test case.

Introduction

The problem statement usually begins by motivating the problem. It gives a situation or context for the problem, before diving into the gory details. This is usually irrelevant to solving the problem, so ignore it if necessary. In some cases, the motivation can cause serious ambiguities if it is treated as binding - see MatchMaking (SRM 203 Div I Easy / Div II Medium). Also note that for some simple problems, the initial context may be left out.

The ordering of the rest of this section varies greatly from problem to problem, based on the writing style of the problem author.

There will be a description of what you need to do, in high-level terms. Take, for example, UserName (SRM 203, Div 2 easy). What the problem is asking for you to do is to find the first variant of a given username that is not already taken. Note that the problem has not yet said anything about variable names or types, or input formats.

There will also be a low-level description of the input. At the bare minimum, the types and variable names of the inputs will be given to you, as well as what they correspond to and what they mean. Sometimes much more information about input formats will be given; this typically occurs in more complicated problems.

Sometimes, even more detailed background information needs to be provided. That is also typically given here, or sometimes in the Notes section.

The Definition

This is a very barebones description of what TopCoder wants you to submit. It gives the class name, the method name to create inside that class, the parameters it should take, the return value, and a method signature. As mentioned before, the basic form of a submitted solution is to create a class containing a method with the required signature. Make sure that the class is declared public if not using C++, and make sure to declare the method public also.

Notes and Constraints

Notes don't always appear. If they do, READ THEM! Typically they will highlight issues that may have come up during testing, or they may provide background information that you may not have known beforehand. The constraints section gives a list of constraints on the input variables. These include constraints on sizes of strings and arrays, or allowed characters, or values of numbers. These will be checked automatically, so there is no need to worry about writing code to check for these cases.

Be careful of the constraints. Sometimes they may rule out certain algorithms, or make it possible for simpler but less efficient algorithms

to run in time. There can be a very big difference between an input of 50 numbers and an input of 5, both in terms of solutions that will end up passing, and in terms of ease of coding.

Examples

These are a list of sample test cases to test your program against. It gives the inputs (in the correct order) and then the expected return value, and sometimes an annotation below, to explain the case further if necessary.

It goes without saying that you should test your code against all of the examples, at the very least. There may be tricky cases, large cases, or corner cases that you have not considered when writing the solution; fixing issues before you submit is infinitely preferable to having your solution challenged or having it fail during system testing.

The examples are not always comprehensive! Be aware of this. For some problems, passing the examples is almost the same as passing every test case.

For others, however, they may intentionally (or not) leave out some test case that you should be aware of. If you are not completely sure that your code is correct, test extensively, and try to come up with your own test cases as well. You may even be able to use them in the challenge phase.

Solving a problem

Now we'll walk through a simple problem and dissect it, bit by bit.

Have a look at BettingMoney, the SRM 191 Division 2 Easy. First we identify the parts of this problem. In the statement itself, we first have the situation behind the problem - gambling. Then we have a little bit of background information about the betting itself. Then, we have a description of the input - data types, variable names, and what they represent. After this we have the task: to determine what the net gain is for the day and return the amount in cents.

Also note the two explanatory paragraphs at the end; the first provides an example of the input format and types, and the second gives a completely worked example, which should be extremely helpful to your understanding.

The definition section is uninteresting, but it is there for completeness' sake.

The notes for this problem are fairly comprehensive. In terms of background information, you might not know that there are 100 cents in a dollar. And in terms of clarification, there is explicit confirmation that the return value may in fact be negative, and that the margin of victory (the variable finalResult) is all that matters when deciding which payoff to make.

The constraints are fairly straightforward. The input arrays will contain the same number of elements, between 1 and 50, inclusive. (50 is a long-standing TopCoder tradition for input sizes). finalResult will be between 0 and that same size minus one (which means, if you give it a little thought, that someone will win their bet). Each element of each array will be between 0 and 5000, inclusive. This is most likely to make sure that integer arithmetic will do the job just fine.

Finally, there's the examples section. Often, the problem statement section will contain an annotated example case, which will become example case 0. Then there are a couple of other example cases, some with explanation and some without. Also note that one of the examples tests for negative return values, to supplement the notes.

A More Complicated Example

Now have a look at Poetry, the SRM 170 Div 2 Hard. In this case, you may not be able to actually solve this in the time allotted. That's ok - the emphasis should first be on understanding what the problem says, even if you can't code it in time.

The first section tells you immediately what you want to do - you'll be given a poem, and you will have to determine what its rhyme scheme is. The rest of the section clarifies what this actually means, in bottom-up fashion (from simpler concepts to more complicated ones). It defines what a legal word is and how to extract words from a poem, and then it defines what it means when two words rhyme - that their ending patterns are equal. The concept of ending pattern is then defined. After all this, we find out what it means to have two lines of the poem rhyme: their last words have to rhyme. Finally, (whew!) we are told how to actually construct the rhyme scheme and in what format to return it.

This is a problem where a lot of terms need to be defined to get to the heart of things, and so all the definitions deserve at least a couple of read-throughs, especially if you're not sure how they all fit together.

The next section is the problem definition section, just for reference. Then there is a single note that clarifies a point that may have been overlooked when it was stated in the problem statement itself: that blank lines will be labeled with a corresponding space in the rhyme scheme.

The constraints are fairly standard for TopCoder problems: there will be between 1 and 50 lines in the poem, and each line will contain between 0 and 50 characters. The only allowable characters in the poem will be spaces and letters, and there will be only legal words in poem.

Finally, there are a number of examples. Usually, problems which are trickier or which have more complex problem statements will have more examples, to clarify at least some of the finer points of the problem statement. Again, this doesn't mean that passing the example cases given is equivalent to having a completely correct solution, but there is a higher chance that you can catch any bugs or trivial mistakes if there are more examples that you know the answers to.

Try it Yourself

Listed below are a number of additional problems, grouped roughly by difficulty of comprehension. Try them for yourself in the TopCoder Arena Practice Rooms. Even if you can't solve them, at least work on figuring out what the problem wants by breaking it down in this manner.

Mentioned in this writeup:

SRM 203 Div 2 Easy - UserName
SRM 191 Div 2 Easy - BettingMoney
SRM 203 Div 1 Easy - MatchMaking
SRM 170 Div 2 Hard - Poetry

Similar tasks:

SRM 146 Div 2 Easy - Yahtzee
SRM 200 Div 2 Easy - NoOrderOfOperations
SRM 185 Div 2 Easy - PassingGrade
SRM 155 Div 2 Easy - Quipu
SRM 147 Div 2 Easy - CCipher
SRM 208 Div 1 Easy - TallPeople
SRM 173 Div 1 Easy - WordForm
SRM 162 Div 1 Easy - PaperFold

More challenging tasks:

SRM 197 Div 2 Hard - QuickSums
SRM 158 Div 1 Hard - Jumper
SRM 170 Div 1 Easy - RecurrenceRelation
SRM 177 Div 1 Easy - TickTick
SRM 169 Div 2 Hard - Twain
SRM 155 Div 1 Med - QuipuReader

Introduction

With many TopCoder problems, the solutions may be found instantly just by reading their descriptions. This is possible thanks to a collection of common traits that problems with similar solutions often have. These traits serve as excellent hints for experienced problem solvers that are able to observe them. The main focus of this article is to teach the reader to be able to observe them too.

Straight-forward problems that don't require any special technique (e.g. simulation, searching, sorting etc.)

In most cases, these problems will ask you to perform some step by step, straight-forward tasks. Their constraints are not high, and not too low. In most cases the first problems (the easiest ones) in TopCoder Single Rounds Matches are of this kind. They test mostly how fast and properly you code, and not necessarily your algorithmic skills.

Most simple problems of this type are those that ask you just to execute all steps described in the statement.

BusinessTasks - SRM 236 Div 1:
N tasks are written down in the form of a circular list, so the first task is adjacent to the last one. A number n is also given. Starting with the first task, move clockwise (from element 1 in the list to element 2 in the list and so on), counting from 1 to n. When your count reaches n, remove that task from the list and start counting from the next available task. Repeat this procedure until one task remains. Return it.

For N<=1000 this problem is just a matter of coding, no special algorithm is needed - do this operation step by step until one item is left. Usually these types of problems have a much smaller N, and so we'll not consider cases where N is very big and for which complicated solution may be needed. Remember that in TopCoder competitions even around 100 millions sets of simple operations (i.e. some multiplications, attributions or if statements) will run in allowed time.

This category of problems also includes those that need some simple searches.

TallPeople - SRM 208 Div 1:
A group of people stands before you arranged in rows and columns. Looking from above, they form an R by C rectangle of people. Your job is to return 2 specific heights - the first is computed by finding the shortest person in each row, and then finding the tallest person among them (the "tallest-of-the-shortest"); and the second is computed by finding the tallest person in each column, and then finding the shortest person among them (the "shortest-of-the-tallest").

As you see this is a really simple search problem. What you have to do is just to follow the steps described in the statement and find those 2 needed heights. Other TC problems may ask you to sort a collection of items by respecting certain given rules. These problems may be also included in this category, because they too are straight-forward - just sort the items respecting the rules! You can do that with a simple O(N^2) sorting algorithm, or use standard sorting algorithm that exist in your coding language. It's just a matter of coding.

Other example(s):
MedalTable - SRM 209 Div 1.

Breadth First Search (BFS)

Problems that use BFS usually ask to find the fewest number of steps (or the shortest path) needed to reach a certain end point (state) from the starting one. Besides this, certain ways of passing from one point to another are offered, all of them having the same cost of 1 (sometimes it may be equal to another number). Often there is given a N x M table (formed of N lines and M columns) where certain cells are passable and others are impassable, and the target of the problem is to find the shortest time/path needed to reach the end point from the start one. Such tables may represent mazes, maps, cities, and other similar things. These may be considered as classical BFS problems. Because BFS complexity is in most cases linear (sometimes quadratic, or N logN), constraints of N (or M) could be high - even up to 1 million.

SmartWordToy - SRM 233 Div 1:
A word composed of four Latin lowercase letters is given. With a single button click you can change any letter to the previous or next letter in alphabetical order (for example 'c' can be changed to 'b' or 'd'). The alphabet is circular, thus 'a' can become 'z', and 'z' can become 'a' with one click.

A collection of constraints is also given, each defining a set of forbidden words. A constraint is composed of 4 strings of letters. A word is forbidden if each of its characters is contained in corresponding string of a single constraint, i.e. first letter is contained in the first string, the second letter - in the second string, and so on. For example, the constraint "lf a tc e" defines the words "late", "fate", "lace" and "face".

You should find the minimum number of button presses required to reach the word finish from the word start without passing through forbidden words, or return -1 if this is not possible.

Problem hints:

- Words can be considered as states. There are at most 26^4 different words composed of 4 letters (thus a linear complexity will run in allowed time).
- There are some ways to pass from one state to another.
- The cost of passing from a state to another is always 1 (i.e. a single button click).
- You need to find the minimum number of steps required to reach the end state from start state.

Everything indicates us that it's a problem solved by the help of a BFS. Similar things can be found in any other BFS problems. Now let's see an interesting case of BFS problems.

CaptureThemAll - SRM 207 Div 2 (3rd problem):
Harry is playing a chess game. He has one knight, and his opponent Joe has a queen and a rook. Find the minimum number of steps that Harry's knight has to jump so that it captures both the queen and the rook.

Problem hints: At first sight this may seem like dynamic programming or backtracking. But as always, take a look into the text of the statement. After a while you should observe the following things:

- A table is given.
- The knight can jump from one cell to some of its neighbors.
- The cost of passing from a cell to another is always 1 (just one jump).
- You need to find the minimum number of steps (jumps).

Given this information we can see that the problem can be easily solved by the help of BFS. Don't get confused by the fact that connected points are represented by unconnected cells. Think of cells as points in a graph, or states (whatever you want) - and in order to pass from one point to another, the knight should be able to jump from the first to the second point.

Notice again that the most revealing hint about the BFS solution is the cost of 1 for any jump.

Train yourself in finding the hints of a BFS problem in following examples:

Other example(s):
RevolvingDoors - SRM 223 Div 1
WalkingHome - SRM 222 Div 1
TurntableService - SRM 219 Div 1

Flood Fill

Sometimes you may encounter problems that are solved by the help of Flood Fill, a technique that uses BFS to find all reachable points. The thing that makes them different from BFS problems described above is that a minimum path/cost is not needed.

For example, imagine a maze where 1 represents impassable cells and 0 passable cells. You need to find all cells that are reachable from the upper-left corner. The solution is very simple - take one-by-one a visited vertex, add its unvisited neighbors to the queue of visited vertices and proceed with the next one while the queue is still populated. Note that in most cases a DFS (Depth First Search) will not work for such problems due to stack overflows. Better use a BFS. For inexperienced users it may seem harder to implement, but after a little training it becomes a "piece of cake". A good example of such problem would be:

grafixMask - SRM 211 Div 1:
A 400 x 600 bitmap is given. A set of rectangles covers certain parts of this bitmap (the corners of rectangles have integer coordinates). You need to find all contiguous uncovered areas, including their sizes.

Problem hints: What do we have here?

- A map (table)
- Certain points are impassable (those covered by given rectangles)
- Contiguous areas need to be found

It is easy to understand that a problem with such a statement needs a Flood Fill. Usually problems using it are very easy to detect.

Brute Force and Backtracking

I have placed these 2 techniques in the same category because they are very similar. Both do the same thing - try all possible cases (situations) and choose the best one, or count only those that are needed (depending on the problem). Practically, Backtracking is just more advanced and optimized than Brute Force. It usually uses recursion and is applied to problems having low constraints (for example N<=20).

Brute Force
There are many problems that can be solved by the help of a simple brute force. Note that the limits must not be high. How does a brute force algorithm work? Actually, it tries all possible situations and selects the best one. It's simple to construct and usually simple to implement. If there is a problem that asks to enumerate or find all possible ways (situations) of doing a certain thing, and that doesn't have high limits - then it's most probably a brute force problem.

GeneralChess - SRM 197 Div 1:
You are given some knights (at most 8), with their positions on the table (-10000<=x, y<=10000). You need to find all positions to place another one, so that it threatens all given pieces.

Problem hints: Well, this is one of the easiest examples. So which are the hints of this statement?

- You need to find all possible situations (positions) that satisfy a certain rule (threatens all given pieces).
- The limits are very low - only 8 knights are at most given.

It's a common Brute Force problem's statement. Note that x and y limits are not relevant, because you need only try all positions that threaten one of the knights. For each of these positions see if the knight placed at that position threatens all others too.

Another interesting problem would be:
LargestCircle - SRM 212 Div 2 (3rd problem):
Given a regular square grid, with some number of squares marked, find the largest circle you can draw on the grid that does not pass through any of the marked squares. The circle must be centered on a grid point (the corner of a square) and the radius must be an integer. Return the radius of the circle.

The size of the grid is at most 50.

Problem hints: And again one of the most important hints is the low limit of the size of the grid - only 50. This problem is possible to be solved with the help of the Brute Force because for each cell you can try to find the circle whose center is situated in that cell and that respects the rules. Among all of these circles found, select the one that has the greatest radius.

Complexity analysis: there are at most 50x50 cells, a circle's radius is an integer and can be at most 25 units, and you need a linear time (depending on your implementation) for searching the cells situated on the border of the circle. Total complexity is low and thus you can apply a simple Brute Force here.

Other example(s):
Cafeteria - SRM 229 Div 1
WordFind - SRM 232 Div 1

Backtracking
This technique may be used in many types of problems. Just take a look at the limits (N, M and other main parameters). They serve as the main hint of a backtrack problem. If these are very small and you haven't found a solution that's easier to implement - then just don't waste your time on searching it and implement a straight-forward backtracking solution.

Usually problems of this kind ask you to find (similarly to Brute Force):

1. Every possible configuration (subset) of items. These configurations should respect some given rules.
2. The "best" configuration (subset) that respects some given rules.

BridgeCrossing - SRM 146 Div 2 (3rd problem):
A group of people is crossing an old bridge. The bridge cannot hold more than two people at once. It is dark, so they can't walk without a flashlight, and they only have one flashlight! Furthermore, the time needed to cross the bridge varies among the people in the group. When people walk together, they always walk at the speed of the slowest person. It is impossible to toss the flashlight across the bridge, so one person always has to go back with the flashlight to the others. What is the minimum amount of time needed to get all the people across the bridge?

There are at most 6 people.

Problem hints:

- First look at the constraints - there are at most ONLY 6 people! It's enough for generating all possible permutations, sets etc.
- There are different possible ways to pass the people from one side to another and you need to find the best one.

This is of course a problem solved with a backtracking: at the beginning choose any 2 people to pass the bridge first, and after that at each step try to pass any of those that have been left on the start side. From all these passages select the one that needs the smallest amount of time. Note that among persons that have passed over the bridge, the one having the greatest speed should return (it's better than returning one having a lower speed). This fact makes the code much easier to implement. After having realized these things - just code the solution. There may be others - but you will lose more time to find another than to code this one.
MNS - SRM 148 Div 1:
9 numbers need to be arranged in a magic number square. A magic number square is a square of numbers that is arranged such that every row and column has the same sum. You are given 9 numbers that range from 0 to 9 inclusive. Return the number of distinct ways that they can be arranged in a magic number square. Two magic number squares are distinct if they differ in value at one or more positions.

Problem hints: Only 9 numbers are given at most; and every distinct way (configuration) to arrange the numbers so that they form a magic number square should be found. These are the main properties of a Backtracking problem. If you have observed them - think about the code. You can generate all permutations of numbers and for each of them check if it forms a magic square. If so - add it to the answer. Note that it

must be unique. A possible way to do that - is to have a list of earlier found configurations, thus for each new magic square check if it exists in that list and if it doesn't - add it to the answer. There will not be many distinct magic squares, thus no additional problems will appear when applying this method.

Other example(s):
WeirdRooks - SRM 234 Div 1

Dynamic Programming

Quite a few problems are solved with the help of this technique. Knowing how to detect this type of problem can be very valuable. However in order to do so, one has to have some experience in dynamic programming. Usually a DP problem has some main integer variables (e.g. N) which are neither too small, nor too big - so that a usual DP complexity of N^2, N^3 etc. fits in time. Note that in the event that N is very small (for TC problems usually less than 30) - then it is likely the problem is not a DP one. Besides that there should exist states and one or more ways (rules) to reach one greater state from another lower one. In addition, greater states should depend only upon lower states. What is a so-called state? It's just a certain configuration or situation. To better understand dynamic programming, you may want to read this article.

Let's analyze a simple classic DP problem:

Given a list of N coins with their values (V1, V2, ... ,VN), and the total sum S. Find the minimum number of coins the sum of which is S (you can use as many coins of one type as you want), or report that it's not possible to select coins in such a way that they sum up to S.

Let N <= 1,000 and S <= 1,000.

Problem hints:

- Two main integer variables are given (N and S). These are neither too small, nor are they too big (i.e. a complexity of N*S fits in time).
- A state can be defined as the minimum number of coins needed to reach a certain sum.
- A sum (state) i depends only on lower sums (states) j (j<i).
- By adding a coin to a certain sum - another greater sum is reached. This is the way to pass from one state to another.

Thus all properties of a DP problem are uncovered in this statement. Let's see another (slightly harder) DP problem

ZigZag - 2003 TCCC Semifinals 3:
A sequence of numbers is called a zig-zag sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a zig-zag sequence. Given a sequence of integers, return the length of the longest subsequence that is a zig-zag sequence. A subsequence is obtained by deleting some number of elements (possibly zero) from the original sequence, leaving the remaining elements in their original order. Assume the sequence contains between 1 and 50 elements, inclusive.

Problem hints:

- There are N numbers given (1<=N<=50), thus N isn't too small, nor too big.
- A state (i,d) can be defined as the length of the longest zig-zag subsequence ending with the i-th number, for which the number before the last one is smaller than it for d=0, and bigger for d=1.
- A state i (i.e. a subsequence ending with the i-th number) depends only on lower states j (j<i).
- By adding a number to the end of a subsequence - another bigger (greater) subsequence is created. This is the way to pass from one state to another.

As you can see - this statement has almost the same traits (pattern) as in the previous problem. The most difficult part in identifying a DP problem statement is observing/seeing the states with the properties described above. Once you can do that, the next step is to construct the algorithm, which is out of the scope of this article.

Other example(s):
ChessMetric - 2003 TCCC Round 4
AvoidRoads - 2003 TCO Semifinals 4
FlowerGarden - 2004 TCCC Round 1
BadNeighbors - 2004 TCCC Round 4

Hard Drills:

Maximum Flow
In many cases it's hard to detect a problem whose solution uses maximum flow. Often you have to create/define graphs with capacities based on the problem statement.

Here are some signs of a Maximum Flow problem:

- Take a look at the constraints, they have to be appropriate for a O(N^3) or O(N^4) solution, i.e. N shouldn't be greater than 500 in extreme cases (usually it's less than 100).
- There should be a graph with edges having capacities given, or you should be able to define/create it from data given in the statement.
- You should be finding a maximum value of something.

Sample problem:

You are given a list of water pipes, each having a certain maximum water flow capacity. There are water pipes connected together at their extremities.

You have to find the maximum amount of water that can flow from start junction to end junction in a unit of time.

Let N<=100.

As you can see - it's a straight-forward maximum flow problem: water pipes represent edges of the graph, their junctions - vertices; and you have to find the maximum value of amount of water that can flow from start to end vertex in a unit of time.

Optimal Pair Matching:
These problems usually have a list of items (from a set A) for which other items (from a set B) should be assigned under some rules, so that all (or a maximum possible number of) items from set A have to each be assigned to a certain item from set B.

Mixed:
Some problems need other techniques in addition to network flows.

Parking - SRM 236 Div 1:
N cars and M parking lots are given. They are situated on a rectangular surface (represented by a table), where certain cells are impassable. You should find a way to assign each car to a parking lot, so that the greatest of the shortest distances from each car to its assigned parking lot is as small as possible. Each parking lot can have at most one car assigned to it.

Problem hints: By reading this problem one can simply understand the main idea of the solution - it should be something similar to optimal pair matching, because each car (point from a set A) should be assigned to a parking lot (point from a set B) so that all are assigned and that there is at most one car assigned to a parking lot. Additionally, there can be cars that can't reach certain parking lots, thus some pairs of points (one point from A and the other from B) are not connected. However a graph should be created for optimal pair matching. The way to make it is clear - an edge exists between a car and a parking lot if only there is a path between them, and its cost is equal to the shortest distance needed for the car to reach the parking lot. The next step of the solution is a binary search on the longest edge. Although it may be out of the scope of this article, I will provide a short explanation: At each step delete those edges of the initial graph that have costs greater than a certain value C (Note that you'll have to save the initial graph's state in order to repeat this step again for other C values). If it's possible in this case to assign all the cars to parking lots - then take a smaller C, and repeat the same operation. If not - take a greater C. After a complete binary search, the smallest C for which a complete assignment is possible will be found. This will be the answer.

Linear Programming (Simplex)

Most of the common traits of problems solved with the help of the linear programming technique are:

- You are given collection of items having different costs/weights. There is a certain quantity of each item that must be achieved.
- A list of sets is given. These sets are composed of some of the available items, having certain quantities of each of them. Each set has a certain cost.
- The goal of the problem is to find an optimal combination (the cheapest one) of these sets so that the sum of quantities of each of the items they have is exactly the one needed to achieve.

At first it may seem confusing, but let's see an example:

Mixture - SRM 231 Div 1):
A precise mixture of a number of different chemicals, each having a certain amount, is needed. Some mixtures of chemicals may be purchased at a certain price (the chemical components for the mixture might not be available in pure form). Each of them contains certain amounts of some of the chemicals. You need not purchase the available mixtures in integral amounts. Hence if you purchase a 1.5 of a mixture having a price of 3 and amounts of "2 0 1", you'll pay 4.5 and get "3 0 1.5" amounts of chemicals. Your task is to determine the lowest price that the desired mixture can be achieved.

Problem hints:

- A collection of items (chemicals).
- A list of sets (available mixtures), each containing certain amounts of each of the items, and having a certain cost.
- You need to find the lowest price of the desired collection of items achieved by the combination of the available sets. More than that - you can take also non-integral amounts of mixtures.

These are exactly the traits described above.

Conclusion

If you have found this article interesting and you have learned new things from it - train yourself on any of the problems in the TopCoder Algorithm Arena. Try hard to see the hints and determine the type of the solution by carefully reading through the problem statement. Remember, there are still many problems that may not be included properly in any of the categories described above and may need a different approach.

Planning an approach is a finicky art; it can stump the most seasoned coders as much as it stumps the newer ones, and it can be extremely hard to put into words. It can involve many calculations and backtracks, as well as foresight, intuition, creativity, and even dumb luck, and when these factors don't work in concert it can inject a feeling of helplessness in any coder. Sometimes it's this feeling of helplessness that discourages coders from even attempting the Div I Hard. There are even coders that stop competing because they abhor that mental enfeeblement that comes with some problems. However, if one stays diligent, the solution is never really out of the mind's reach. This tutorial will attempt to flesh out the concepts that will enable you to pick an approach to attack the problems with a solid plan.

Pattern Mining and the Wrong Mindset

It is easy to fall into the trap of looking at the algorithm competition as a collection of diverse yet classifiable story problems. For those that have done a lot of story problems, you know that there are a limited number of forms of problems (especially in classes where the professor tends to be repetitious), and when you read a problem in a certain form, your mind says, "Oh, this is an X problem, so I find the numbers that fit the problem and plug and chug." There are many times when this kind of pattern mining pays off; after a number of TopCoder Single Round Matches, most coders will recognize a set of common themes and practice against them, and this method of problem attack can be successful for many matches.

However, this approach is perilous. There are times when you skim the problem statement and assume it's of type Q, then start coding and discover that your code passes none of the examples. That's when you reread the problem and find out that this problem is unique to your experience. At that point, you are paralyzed by your practice; being unable to fit any of your problem types to the problem you are unable to proceed. You'll see this often when there's a really original problem that comes down the pipe, and a lot of seasoned coders fail the problem because they are blinded by their experience.

Pattern mining encourages this kind of mindset that all of the problem concepts have been exhausted, when in reality this is impossible. Only by unlearning what you have learned (to quote a certain wise old green midget) and by relearning the techniques of critical thought needed to plan an approach can your rating sustainably rise.

Coding Kata

Here's your first exercise: take any problem in the Practice Rooms that you haven't done. Fight through it, no matter how long it takes, and figure it out (use the editorial from the competition as a last resort). Get it to pass system tests, and then note how long you took to solve it. Next, clear your solution out, and try to type it in again (obviously cutting and pasting will ruin the effect). Again, get it to pass system tests. Note how long it took you to finish the second time. Then, clear it out and do the problem a third time, and again get it to pass system tests. Record this final time.

The time it takes for your first pass is how long it takes you when you have no expectations of the problem and no approach readily in mind. Your time on the second pass is usually the first time minus the amount of time it took you to understand the problem statement. (Don't be surprised at the number of bugs you'll repeat in the second pass.) That final recorded time is your potential, for you can solve it this fast in competition if you see the correct approach immediately after reading it. Let that number encourage you; it really is possible to solve some of these problems this quickly, even without super fast typing ability. But what you should also learn from the third pass is the feeling that you knew a working strategy, how the code would look, where you would tend to make the mistakes, and so on. That's what it feels like to have the right approach, and that feeling is your goal for future problems in competition.

In most martial arts, there's a practice called kata where the martial artist performs a scripted series of maneuvers in order, usually pretending to defend (or sometimes actually defending) against an onslaught of fighters, also scripted to come at the artist predictably. At first this type of practice didn't make any sense, because it didn't seem realistic to the chaotic nature of battle. Furthermore it seems to encourage the type of pattern mining mentioned in the previous section. Only after triple-coding many problems for a while can one comprehend the true benefit of this coding kata. The kata demonstrates to its practitioners the mental experience of having a plan, encouraging the type of discipline it takes to sit and think the problem through. This plan of attack is your approach, and it carries you through your coding, debugging, and submission.

Approach Tactics

Now that you know what an approach feels like and what its contents are, you'll realize that you know a lot of different types of these approaches. Do you give them names? "Oh, I used DP (dynamic programming) on that problem." "Really, I could have done that one greedy?" "Don't tell me that the brute-force solution would have passed in time." Really, the name you give an approach to a problem is a misnomer, because you can't classify every problem as a type like just greedy or just brute-force. There are an infinite number of problem types, even more solution types, and even within each solution type there are an infinite number of different variations. This name is only a very high level summary of the actual steps it takes to get to the solution.

In some of the better match editorials there is a detailed description of one approach to solving the code. The next time you look at a match summary, and there is a good write-up of a problem, look for the actual steps and formation of the approach. You start to notice that there is a granularity in the steps, which suggests a method of cogitation. These grains of insight are approach tactics, or ways to formulate your approach, transform it, redirect it, and solidify it into code that get you closer to the solution or at least point you away from the wrong solution. When planning your approach, the idea is that you will use whatever approach tactics are at your disposal to decide on your approach, the idea being that you are almost prewriting the code in your head before you proceed. It's almost as if you are convincing yourself that the code you are about to write will work.

Coders with a math background may recognize this method of thinking, because many of these approach tactics are similar to proof writing techniques. Chess players may identify it with the use of tactics to look many moves ahead of the current one. Application designers may already be acquainted with this method when working with design patterns. In many other problem solving domains there is a similar parallel to this kind of taxonomy.

To practice this type of critical thinking and to decide your preferences among approach tactics, it is very useful to record the solutions to your problems, and to write up a post-SRM analysis of your own performance. Detail in words how each of your solutions work so that others could understand and reproduce the approach if they wanted to just from your explanations. Not only will writing up your approaches help you to understand your own thoughts while coding, but this kind of practice also allows you to critique your own pitfalls and work on them in a constructive manner. Remember, it is difficult to improve that which you don't understand.

Breaking Down a Problem

Let's talk about one of the most common approach tactics: breaking down a problem. This is sometimes called top-down programming: the idea is that your code must execute a series of steps in order, and from simple decisions decide if other steps are necessary, so start by planning out what your main function needs before you think about how you'll do the subfunctions. This allows you to prototype the right functions on the fly (because you only code for what you need and no further), and also it takes your problem and fragments it into smaller, more doable parts.

A good example of where this approach is useful is in MatArith from Round 2 of the 2002 TopCoder Invitational. The problem requires you to evaluate an expression involving matrices. You know that in order to get to the numbers you'll need to parse them (because they're in String arrays) and pass those values into an evaluator, change it back into a String array and then you're done. So you'll need a print function, a parse function and a new calc function. Without thinking too hard, if you imaging having all three of these functions written already the problem could be solved in one line:

```
public String[] calculate(String[] A, String[] B, String[] C, String eval){

  return print(calc(parse(A),parse(B),parse(C),eval));

}
```

The beauty of this simplest approach tactic is the guidance of your thoughts into a functional hierarchy. You have now fragmented your work into three steps: making a parse function, a print function, and then a calc function, breaking a tough piece of code into smaller pieces. If you break down the code fine enough, you won't have to think hard about the simplest steps, because they'll become atomic (more on this below). In fact the rest of this particular problem will fall apart quickly by successive partitioning into functions that multiply and add the matrices, and one more that reads the eval statement correctly and applies the appropriate functions.

This tactic really works well against recursive problems. The entire idea behind recursive code is that you are breaking the problem into smaller pieces that look exactly like the original, and since you're writing the original, you're almost done. This approach tactic also plays into the hands of a method of thinking about programs called functional programming. There are several articles on the net and even a TopCoder article written by radeye that talk more about this concept in depth, but the concept is that if properly fragmented, the code will pass all variable information between functions, and no data needs to be stored between steps, which prevents the possibility of side-effects (unintended changes to state variables between steps in code) that are harder to debug.

Plan to Debug

Whenever you use an approach you should always have a plan to debug the code that your approach will create. This is the dark underbelly of every approach tactic. There is always a way that a solution may fail, and by thinking ahead to the many ways it can break, you can prevent the bugs in the code before you type them. Furthermore, if you don't pass examples, you know where to start looking for problems. Finally, by looking for the stress points in the code's foundation, it becomes easier to prove to yourself that the approach is a good one.

In the case of a top-down approach, breaking a problem down allows you to isolate sections of the code where there may be problems, and it will allow you to group tests that break your code into sections based on the subfunction they seem to exploit the most. There is also an advantage to breaking your code into functions when you fix a bug, because that bug is fixed in every spot where the code is used. The alternative to this is when a coder copy/pastes sections of code into every place it is needed, making it harder to propagate a fix and makes the fix more error prone. Also, when you look for bugs in a top-down approach, you should look for bugs inside the functions before you look between the calls to each function. These parts make up a debugging strategy: where to look first, how to test what you think is wrong, how to validate pieces and move on. Only after sufficient practice will a debugging strategy become more intuitive to your method of attack.

Atomic Code

If you arrive at a section of code that you cannot break down further this is atomic code. Hopefully you know how to code each of these sections, and these form the most common forms of atomic code. But, don't be discouraged when you hit a kernel of the problem that you don't know how to code; these hard-to-solve kernels are in fact what make the problem interesting, and sometimes being able to see these in advance can make the big difference between solving the problem early with the right approach and heading down the wrong path with the wrong approach, wasting a lot of time in the process.

The most common type of atomic code you'll write is in the form of primitives. I've always been a proponent of knowing the library of your language of choice. This is where that knowledge is of utmost importance. What better way to save yourself time is there in both planning your approach and coding your solution when you know that a possibly difficult section of your code is in fact atomic and solved using a library function or class?

The second type of atomic code you'll write are what I call language techniques. These are usually snippets of code committed to memory that perform a certain operation in the language, like locating the index of the first element in an array with the minimum value, or parsing a String into tokens separated by whitespace. These techniques are equally essential to planning an approach, because if you know how to do these fundamental operations intuitively, it makes more tasks in your search for a top-down approach atomic, thus making the search for the right approach shorter. In addition, it makes the segments of the code in these atomic segments less error prone. Furthermore, if you are asked to perform a task similar to one that you already know a language technique for, it makes it much easier to mutate the code to fit the situation (for example: searching for the index of the first maximal element in an array based on some heuristic is easy if you already know how to type up similar tasks). Looking for these common language techniques should become an element of your daily practice, and any atomic code should fly off your fingers as soon as you think about it.

As an aside, I must address the use of code libraries. I know that this is a contested topic, and many successful coders out there make use of a (sometimes encyclopedic) library as a pre-inserted segment of code before they start coding. This is totally legal (although changes to the rules after the 2004 TopCoder Open may affect their future legality), and there are obvious advantages to using a library, mainly through the ability to declare more parts of your top-down approach atomic, and by being able to more quickly construct bottom-up fragments of code (as discussed below). It is my opinion, however, that the disadvantages of using library code outweigh the advantages. On a small note, library code executed through functions can sometimes slow your coding, because you have to make the input match the prototype of the code you're trying to use. Library code is mostly non-mutatable, so if your library is asked to do something that isn't expressly defined, you find yourself fumbling over a language technique or algorithm that should already be internalized. It is also possible that your library code isn't bug-free, and debugging your library mid-competition is dangerous because you may have to propagate that change to code you've already submitted and also to the template before you open any more problems. Also, library use is not allowed in onsite competition. Finally, the use of library code (or macros for that manner) get you used to leaning on your library instead of your instincts of the language, making the use of normal primitives less intuitive and the understanding of other coder's solutions during challenge phase not as thorough. If used in moderation your library can be powerful, but it is not the ultimate weapon for all terrain.

There may be a point where you hit a piece of atomic code that you are unable to fragment. This is when you have to pull out the thinking cap and start analyzing your current approach. Should I have broken up the tasks differently? Should I store my intermediate values differently? Or maybe this is the key to the problem that makes the problem hard? All of these things must be considered before you pound the keys. Even at these points where you realize that you're stuck, there are ways to manipulate the problem at hand to come to an insight on how to proceed quickly, and these ways comprise the remaining approach tactics.

Bottom Up Programming

This technique is the antithesis to breaking down a program, and should be the first thing you start doing when you get stuck. Bottom-up programming is the process of building up primitive functions into more functional code until the solution becomes as trivial as one of the primitives. Sometimes you know that you'll need certain functions to form a solution and if these functions are atomic or easy to break down, you can start with these functions and build your solution upward instead of breaking it down.

In the case of MatArith, the procedure to add and multiply matrices was given in the problem statement making it easy to follow directions and get two functions to start with. From there you could make a smaller evalMult function that multiplied matrices together using a string evaluation and variable names, then a similar evalAdd that treats each term as a block and you have an approach to solve the problem.

In general, it's a very good strategy to code up any detailed procedure in the problem statement before tackling the actual problem. Examples of these are randomizer functions, any data structures you're asked to simulate, and any operations on mathematical objects like matrices and complex numbers. You'll find that by solving these smaller issues and then rereading the problem statement that you will understand what needs to be done much better. And sometimes, if you're really stuck, it doesn't hurt to write a couple atomic pieces of code that you know that you'll need in order to convince your mind to break down the problem towards those functions. As you can see, your path to the right approach

need not be linear as long as it follows your train of thought.

Also, in case of a hidden bug, keep in mind that any code that you write using this approach tactic should be scanned for bugs before your top-down code, because you tend to write this code first and thus at a stage where you understand the problem less than when you finish the code. This is a good rule of thumb to follow when looking for errors in your code; they usually sit in the older sections of the code, even if older is only decided by minutes or even seconds.

Brute Force

Any time the solution requires looking for an optimal configuration or a maximal number or any other choice of one of a finite set of objects, the simplest way to solve the problem is to try all configurations. Any time the solution requires calculating a massive calculation requiring many steps, the best way to solve it is to do every calculation as asked for in the problem. Any time the problem asks you to count the number of ways something can be done, the best way to solve it is to try every way and make a tally. In other words, the first approach to consider in any possibly time-intensive problem is the most obvious one, even if it is horribly inefficient.

This approach tactic, called brute force, is so called because there is no discerning thought about the method of calculation of the return value. Any time you run into this kind of an optimization problem the first thing you should do is try to figure in your head the worst possible test cases and if 8 seconds is enough time to solve each one. If so, brute force can be a very speedy and usually less error prone approach. In order to utilize brute force, you have to know enough about the programming environment to calculate an estimate how much time any calculation will take. But an estimate is just a guess, and any guess could be wrong. This is where your wisdom is forced to kick in and make a judgment call. And this particular judgment call has bitten many a coder that didn't think it could be done brute force and couldn't debug a fancier approach, and likewise those that didn't figure correctly the worst of the cases to be tested for time.

In general, if you can't think of a way to solve the problem otherwise, plan to use brute force. If it ends up that you are wrong, and there is a test case that takes too long, keep the brute force solution around, and while recoding the more elegant solution, use the brute-force solution to verify that your elegant code is correct in the smaller cases, knowing that its more direct approach is a good verification of these cases (being much less error-prone code).

A Place for Algorithms

Well-known and efficient algorithms exist for many standard problems, much like basic approaches exist for many standard word problems in math, just like standard responses exist for most common opening moves in chess. While in general it's a bad idea to lean heavily upon your knowledge of the standard algorithms (it leads down the path of pattern mining and leaves you vulnerable to more original problems), it's a very good idea to know the ones that come up often, especially if you can apply them to either an atomic section of code or to allow them to break your problem down.

This is not the place to discuss algorithms (there are big books to read and other tutorials in this series to follow that will show you the important stuff), but rather to discuss how algorithms should be used in determining an approach. It is not sufficient to know how to use an algorithm in the default sense; always strive to know any algorithms you have memorized inside and out. For example, you may run into a problem like CityLink (SRM 170 Div I Med), which uses a careful mutation of a basic graph algorithm to solve in time, whereby just coding the regular algorithm would not suffice. True understanding of how the algorithm works allows the insight needed to be able to even conceive of the right mutation.

So, when you study algorithms, you need to understand how the code works, how long it will take to run, what parts of the code can be changed and what effect any changes will have on the algorithm. It's also extremely important that you know how to code the algorithm by memory before you try to use it in an approach, because without the experience of implementing an algorithm, it becomes very hard to tell whether your bugs are being caused by a faulty implementation or faulty input into the implementation. It's also good to practice different ways to use the algorithms creatively to solve different problems, to see what works and what doesn't. Better for an experiment with code to fall flat on its face in practice than during a competition. This is why broad-based algorithmic techniques (like divide-and-conquer, dynamic programming, greedy algorithms) are better to study first before you study your more focused algorithms because the concepts are easier to manipulate and easier to implement once you understand the procedure involved.

Manipulating the Domain

This situation will become more and more familiar: you find yourself trudging through the planning stages of a problem because of the sheer amount of work involved in simulating the domain of the problem. This may be due to the inappropriateness of the presented domain, and there are times when manipulating the domain of the problem to something more convenient will create an easier or more recognizable problem. The classic example of this is the game of Fifteen (used as a problem in SRM 172). In the game of Fifteen, you have numbers from 1 to 9 of which you may claim one per turn, and if exactly three of your numbers add to 15 before exactly three of your opponent's numbers adds to 15, then you win. For this problem, you can manipulate the domain by placing the numbers in the configuration of a 3x3 magic square (where every row, column, and diagonal add up to the same sum, in this case 15). Instantly you realize that the game of Fifteen is just the game Tic-Tac-Toe in disguise, making the game easier to play and program a solution for, because the manipulation of the domain transformed the situation of a game where you have no prior knowledge into one where you have a lot more knowledge. Some mathematicians think of this as the ABA^{-1} approach, because the algebra suggests the proper process: first you transform the domain, then you perform your action, then (the A^{-1}) you reverse your transformation. This approach is very common in solving complex problems like diagonalizing matrices and solving the Rubik's Cube.

Most commonly this approach tactic is used to simplify basic calculations. A good example of this type of approach is HexagonIntersections from SRM 206. In this problem it was needed to find the number of tiled hexagons that touched a given line. The problem became much easier if you "slanted" the grid by transforming the numbers involved so that the hexagons involved had sides parallel to the x and y axis and the problem still had the same answer, thereby simplifying calculations.

Extreme care must be taken while debugging if you manipulate the domain. Remember that the correct procedure to domain manipulation is to first manipulate the domain, then solve the problem, and then correct the domain. When you test the code, remember that either the domain must be properly reversed by the transformation before the result is returned, or the reversal must not affect the answer. Also, when looking at values inside the domain manipulation, remember that these are transformed values and not the real ones. It's good to leave comment lines around your transformed section of code just to remind yourself of this fact.

Unwinding the Definitions

This approach tactic is an old mathematician's trick, relating to the incessant stacking of definitions upon definitions, and can be used to unravel a rather gnarly problem statement to get at the inner intended approach to the problem. The best way to do this is with code. When you read the definition of something you have never encountered before, try to think how you would code it. If the code asks you to find the simplest *grozmojt* in a set of integers, first figure out how your code would verify that something was a *grozmojt* and then figure out how to search for it, regardless if you even need to verify that something was a *grozmojt* in the solution. This is very similar to the bottom-up programming above, but taken at the definition level instead of the procedural one.

Simulation problems fall under similar tactics, and create one of those times when those predisposed to object oriented coding styles run up the scores. The best way to manage a simulation problem is to create a simulation object that can have actions performed on it from a main function. That way you don't worry if you passed enough state into a given function or not; since all of the information in the simulation is coming along with you, the approach becomes very convenient and reaches atomic code very quickly. This is also the correct approach to take if an algorithm needs to be simulated to count the steps needed in the algorithm (like MergeSort) or the number of objects deallocated in the execution of another algorithm (like ImmutableTrees). In these situations, the elegance of the code is usually sacrificed in the name of correctness and thoroughness, also making the approach easier to plan ahead for.

The Problem is Doable

An old geometry puzzle goes like this: you have a pair of concentric circles and the only length you are given is the length of a chord of the outer circle (call the chord length x) that is tangent to the inner circle, and you are asked for the area between the circles. You respond: "Well, if the problem is doable then the inner circle's radius is irrelevant to the calculation, so I'll declare it to be 0. Because the area of the inner circle is 0, or degenerates to the center of the outer circle, the chord of the outer circle passes through the center and is thus the diameter, and thus the area of the outer circle is $Pi(x/2)^2$." Note that a proper geometric proof of this fact is harder to do; the sheer fact that a solution exists actually makes the problem easier. Since the writer had to write a solution for the problem, you know it's always solvable, and this fact can be used to your advantage in an SRM.

This approach tactic broadens into the concept that the writer is looking for a particular type of solution, and sometimes through edits of the original problem statement this approach is given away (especially if the original problem is considered too hard for the level it's at). Look for lowered constraints like arrays of size 20 (which many a seasoned coder will tell you is almost a codeword that the writer is looking for a brute force solution), or integers limited to between 1 and 10000 (allowing safe multiplication in ints without overflow). By leaning on the constraints you are acting similarly to the situation above, by not allowing the complexities of the problem that were trimmed off by the constraints to complicate your approach.

Sometimes the level of the problem alone will give a hint to what solution is intended. For example, look at FanFailure (from SRM 195 Div I Easy). The problem used the language of subsets and maximal and minimal, so you start to think maybe attack with brute force, and then you see the constraints opened up to 50 for the array size. 2^{50} distinct subsets rules out brute force (better to find this out in the approach than in the code, right?) and you could look to fancier algorithms... but then you realize that this is a Div I Easy and probably isn't as hard as it looks so you think through the greedy algorithm and decide that it probably works. This choice wouldn't have been so obvious had it not been a Div I Easy.

Keep in mind that these invisible cues are not objective and can't be used to reason why an approach will work or not; they are there only to suggest what the writer's mind was thinking. Furthermore, if the writer is evil or particularly tricky, cues of this nature may be red herrings to throw these tactics astray. As long as you temper this approach tactic with solid analysis before you go on a wild goose chase, this "circular reasoning" can be used to great advantage.

Case Reduction

Sometimes the simplest problems to state are the ones that provide the most difficulty. With these types of problems it's not unusual that the solution requires that you break up the problem not into steps but into cases. By breaking a problem up into cases of different sets of inputs you can create subproblems that can be much easier to solve. Consider the problem TeamPhoto (SRM 167 Div I Medium). This problem is simple to state, but abhorrent to solve. If you break up the problem into a series of cases, you find that where the entire problem couldn't alone be solved by a greedy algorithm, each of the different cases could be, and you could take the best case from those optimal configurations to solve the problem.

The most common use of case reduction involves removing the boundary cases so that they don't mess up a naï¿½ve solution. A good example of this is BirthdayOdds (SRM 174 Div I Easy); many people hard coded if(daysInYear==1)return 2; to avoid the possible problems with the boundary case, even if their solution would have handled it correctly without that statement. By adding that level of security, it became easier to verify that the approach they chose was correct.

Plans Within Plans

As illustrated above, an approach isn't easily stated, and is usually glossed over if reduced to a one-word label. Furthermore, there are many times when there exist levels to a problem, each of which needs to be solved before the full solution comes to light. One clear example of this is the problem MagicianTour (SRM 191 Div I Hard). There are definitely two delineated steps to this problem: the first step requires a graph search to find all connected components and their 2-coloring, and the second step requires the DP knapsack algorithm. In cases like this, it's very helpful to remember that sometimes more than one approach tactic needs to be applied to the situation to get at the solution. Another great example is TopographicalImage (SRM 209 Div I Hard) which asks for the lowest angle that places a calculated value based on shortest path under a certain limit. To solve, note that looking for this lowest value can be approached by binary search, but there are plans within plans, and the inner plan is to apply Floyd-Warshall's All Pairs Shortest Paths algorithm to decide if the angle is satisfactory.

Remember also that an approach isn't just "Oh, I know how to break this down... Let's go!" The idea of planning your approach is to strategically think about: the steps in your code, how an algorithm is to be applied, how the values are to be stored and passed, how the solution will react to the worst case, where the most probable nesting places are for bugs. The idea is that if the solution is carefully planned, there is a lower chance of losing it to a challenge or during system tests. For each approach there are steps that contain plans for their steps.

Tactical Permutation

There is never a right approach for all coders, and there are usually at least two ways to do a problem. Let's look at an unsavory Division One Easy called OhamaLow (SRM 205 Div I Easy). One of the more popular ways to approach this problem is to try all combinations of hands, see if the hand combination is legal, sort the hand combination, and compare it to the best hand so far. This is a common brute-force search strategy. But it's not the entire approach. Remember that there are plans within plans. You have to choose an approach on how to form each hand (this could be done recursively or using multiple for-loops), how to store the hands (int arrays, Strings, or even a new class), and how to compare them. There are many different ways to do each of these steps, and most of the ways to proceed with the approach will work. As seen above as well as here, there are many times where more than one approach will do the job, and these approaches are considered permutable. In fact, one way to permute the top-level brute-force search strategy is to instead of considering all possible constructible hands and picking the best one, you can construct all possible final hands in best to worst order and stop when you have found one that's constructible. In other words you take all 5 character substrings of "87654321" in order and see if the shared hand and player hand can make the chosen hand, and if so return that hand. This approach also requires substeps (how to decide if the hand can be formed, how to walk the possible hands, and so on) but sometimes (and in this case it is better) you can break it down faster.

The only way you get to choose between two approaches is if you are able to come up with both of them. A very good way to practice looking for multiple approaches to problems is to try to solve as many of the problems in the previous SRM using two different approaches. By doing this, you stretch your mind into looking for these different solutions, increasing your chances of finding the more elegant solution, or the faster one to type, or even the one that looks easier to debug.

Backtracking from a Flawed Approach

As demonstrated in the previous section, it is very possible for there to exist more than one way to plan an approach to a problem. It may even hit you in the middle of coding your approach how to more elegantly solve the problem. One of the hardest disciplines to develop while competing in TopCoder Single Round Matches is the facility to stick to the approach you've chosen until you can prove without a shadow of a doubt that you made a mistake in the approach and the solution will not work. Remember that you are not awarded points for code elegance, or for cleverness, or for optimized code. You are granted points solely on the ability to quickly post correct code. If you come up with a more elegant solution than the one you're in the middle of typing up, you have to make the split second analysis of how much time you'll lose for changing your current approach, and in most cases it isn't worth it.

There is no easy answer to planning the right approach the first time. If you code up a solution and you know it is right but has bugs this is much easier to repair than the sudden realization that you just went the entirely wrong direction. If you get caught in this situation, whatever you do, don't erase your code! Relabel your main function and any subfunctions or data structures that may be affected by further changes. The reason is because while you may desire a clean slate, you must accept that some of your previous routines may be the same, and retracing your steps by retyping the same code can be counterproductive to your thinking anew. Furthermore, by keeping the old code you can test against it later looking for cases that will successfully challenge other coders using the same flawed approach.

Conclusion

Planning an approach is not a science, although there is a lot of rigor in the thought involved. Rather, it is mainly educated guesswork coupled with successful planning. By being creative, economical, and thorough about your thought process you can become more successful and confident in your solutions and the time you spend thinking the problem through will save you time later in the coding and debugging. This ability to plan your code before the fingers hit the keys only develops through lots of practice, but this diligence is rewarded with an increasing ability to solve problems and eventually a sustained rating increase.

Mentioned in this writeup:
TCI '02 Round 2 Div I Med - MatArith
SRM 170 Div I Med - CityLink
SRM 172 Div I Med - Fifteen
SRM 206 Div I Hard - HexagonIntersections
SRM 195 Div I Easy - FanFailure
SRM 167 Div I Med - TeamPhoto
SRM 174 Div I Easy - BirthdayOdds
SRM 191 Div I Hard - MagicianTour
SRM 210 Div II Hard - TopographicalImage
SRM 206 Div I Easy - OmahaLow

Introduction

I have seen a number of competitors complain that they are unfairly disadvantaged because many TopCoder problems are too mathematical. Personally, I love mathematics and thus I am biased in this issue. Nevertheless, I strongly believe that problems should contain at least some math, because mathematics and computer science often go hand in hand. It is hard to imagine a world where these two fields could exist without any interaction with each other. These days, a great deal of applied mathematics is performed on computers such as solving large systems of equations and approximating solutions to differential equations for which no closed formula exists. Mathematics is widely used in computer science research, as well as being heavily applied to graph algorithms and areas of computer vision.

This article discusses the theory and practical application to some of the more common mathematical constructs. The topics covered are: primes, GCD, basic geometry, bases, fractions and complex numbers.

Primes

A number is prime if it is only divisible by 1 and itself. So for example 2, 3, 5, 79, 311 and 1931 are all prime, while 21 is not prime because it is divisible by 3 and 7. To find if a number n is prime we could simply check if it divides any numbers below it. We can use the modulus (%) operator to check for divisibility:

```
for (int i=2; i<n; i++)
   if (n%i==0) return false;

return true;
```

We can make this code run faster by noticing that we only need to check divisibility for values of i that are less or equal to the square root of n (call this m). If n divides a number that is greater than m then the result of that division will be some number less than m and thus n will also divide a number less or equal to m. Another optimization is to realize that there are no even primes greater than 2. Once we've checked that n is not even we can safely increment the value of i by 2. We can now write the final method for checking whether a number is prime:

```
public boolean isPrime (int n)
{
   if (n<=1) return false;
   if (n==2) return true;
   if (n%2==0) return false;
   int m=Math.sqrt(n);

   for (int i=3; i<=m; i+=2)
      if (n%i==0)
         return false;

   return true;
}
```

Now suppose we wanted to find all the primes from 1 to 100000, then we would have to call the above method 100000 times. This would be very inefficient since we would be repeating the same calculations over and over again. In this situation it is best to use a method known as the Sieve of Eratosthenes. The Sieve of Eratosthenes will generate all the primes from 2 to a given number n. It begins by assuming that all numbers are prime. It then takes the first prime number and removes all of its multiples. It then applies the same method to the next prime number. This is continued until all numbers have been processed. For example, consider finding primes in the range 2 to 20. We begin by writing all the numbers down:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

2 is the first prime. We now cross out all of its multiples, ie every second number:

```
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~
```

The next non-crossed out number is 3 and thus it is the second prime. We now cross out all the multiples of 3, ie every third number from 3:

```
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~
```

All the remaining numbers are prime and we can safely terminate the algorithm. Below is the code for the sieve:

```
public boolean[] sieve(int n)
{
   boolean[] prime=new boolean[n+1];
   Arrays.fill(prime,true);
   prime[0]=false;
   prime[1]=false;
   int m=Math.sqrt(n);

   for (int i=2; i<=m; i++)
      if (prime[i])
```

```
         for (int k=i*i; k<=n; k+=i)
            prime[k]=false;

   return prime;
}
```

In the above method, we create a boolean array prime which stores the primality of each number less of equal than n. If prime[i] is true then number i is prime. The outer loop finds the next prime while the inner loop removes all the multiples of the current prime.

GCD

The greatest common divisor (GCD) of two numbers a and b is the greatest number that divides evenly into both a and b. Naively we could start from the smallest of the two numbers and work our way downwards until we find a number that divides into both of them:

```
for (int i=Math.min(a,b); i>=1; i--)
   if (a%i==0 && b%i==0)
      return i;
```

Although this method is fast enough for most applications, there is a faster method called Euclid's algorithm. Euclid's algorithm iterates over the two numbers until a remainder of 0 is found. For example, suppose we want to find the GCD of 2336 and 1314. We begin by expressing the larger number (2336) in terms of the smaller number (1314) plus a remainder:

```
2336 = 1314 x 1 + 1022
```

We now do the same with 1314 and 1022:

```
1314 = 1022 x 1 + 292
```

We continue this process until we reach a remainder of 0:

```
1022 = 292 x 3 + 146
292 = 146 x 2 + 0
```

The last non-zero remainder is the GCD. So the GCD of 2336 and 1314 is 146. This algorithm can be easily coded as a recursive function:

```
//assume that a and b cannot both be 0
public int GCD(int a, int b)
{
   if (b==0) return a;
   return GCD(b,a%b);
}
```

Using this algorithm we can find the lowest common multiple (LCM) of two numbers. For example the LCM of 6 and 9 is 18 since 18 is the smallest number that divides both 6 and 9. Here is the code for the LCM method:

```
public int LCM(int a, int b)
{
   return b*a/GCD(a,b);
}
```

As a final note, Euclid's algorithm can be used to solve linear Diophantine equations. These equations have integer coefficients and are of the form:

```
ax + by = c
```

Geometry

Occasionally problems ask us to find the intersection of rectangles. There are a number of ways to represent a rectangle. For the standard Cartesian plane, a common method is to store the coordinates of the bottom-left and top-right corners.

Suppose we have two rectangles R1 and R2. Let (x1, y1) be the location of the bottom-left corner of R1 and (x2, y2) be the location of its top-right corner. Similarly, let (x3, y3) and (x4, y4) be the respective corner locations for R2. The intersection of R1 and R2 will be a rectangle R3 whose bottom-left corner is at (max(x1, x3), max(y1, y3)) and top-right corner at (min(x2, x4), min(y2, y4)). If max(x1, x3) > min(x2, x4) or max(y1, y3) > min(y2, y4) then R3 does not exist, ie R1 and R2 do not intersect. This method can be extended to intersection in more than 2 dimensions as seen in CuboidJoin (SRM 191, Div 2 Hard).

Often we have to deal with polygons whose vertices have integer coordinates. Such polygons are called lattice polygons. In his tutorial on Geometry Concepts, lbackstrom presents a neat way for finding the area of a lattice polygon given its vertices. Now, suppose we do not know the exact position of the vertices and instead we are given two values:

```
B = number of lattice points on the boundary of the polygon

I = number of lattice points in the interior of the polygon
```

Amazingly, the area of this polygon is then given by:

```
Area = B/2 + I - 1
```

The above formula is called Pick's Theorem due to Georg Alexander Pick (1859 - 1943). In order to show that Pick's theorem holds for all lattice polygons we have to prove it in 4 separate parts. In the first part we show that the theorem holds for any lattice rectangle (with sides parallel to axis). Since a right-angled triangle is simply half of a rectangle it is not too difficult to show that the theorem also holds for any right-angled triangle (with sides parallel to axis). The next step is to consider a general triangle, which can be represented as a rectangle with some right-angled triangles cut out from its corners. Finally, we can show that if the theorem holds for any two lattice polygons sharing a common side then it will also hold for the lattice polygon, formed by removing the common side. Combining the previous result with the fact that every simple polygon is a union of triangles gives us the final version of Pick's Theorem. Pick's theorem is useful when we need to find the number of lattice points inside a large polygon.

Another formula worth remembering is Euler's Formula for polygonal nets. A polygonal net is a simple polygon divided into smaller polygons. The smaller polygons are called faces, the sides of the faces are called edges and the vertices of the faces are called vertices. Euler's Formula then states:

```
V - E + F = 2, where

V = number of vertices
E = number of edges
F = number of faces
```

For example, consider a square with both diagonals drawn. We have V = 5, E = 8 and F = 5 (the outside of the square is also a face) and so V - E + F = 2.

We can use induction to show that Euler's formula works. We must begin the induction with V = 2, since every vertex has to be on at least one edge. If V = 2 then there is only one type of polygonal net possible. It has two vertices connected by E number of edges. This polygonal net has E faces (E - 1 "in the middle" and 1 "outside"). So V - E + F = 2 - E + E = 2. We now assume that V - E + F = 2 is true for all 2<=V<=n. Let V = n + 1. Choose any vertex w at random. Now suppose w is joined to the rest of the net by G edges. If we remove w and all these edges, we have a net with n vertices, E - G edges and F - G + 1 faces. From our assumption, we have:

```
(n) - (E - G) + (F - G + 1) = 2
thus (n+1) - E + F = 2
```

Since V = n + 1, we have V - E + F = 2. Hence by the principal of mathematical induction we have proven Euler's formula.

Bases

A very common problem faced by TopCoder competitors during contests involves converting to and from binary and decimal representations (amongst others).

So what does the base of the number actually mean? We will begin by working in the standard (decimal) base. Consider the decimal number 4325. 4325 stands for 5 + 2 x 10 + 3 x 10 x 10 + 4 x 10 x 10 x 10. Notice that the "value" of each consequent digit increases by a factor of 10 as we go from right to left. Binary numbers work in a similar way. They are composed solely from 0 and 1 and the "value" of each digit increases by a factor of 2 as we go from right to left. For example, 1011 in binary stands for 1 + 1 x 2 + 0 x 2 x 2 + 1 x 2 x 2 x 2 = 1 + 2 + 8 = 11 in decimal. We have just converted a binary number to a decimal. The same applies to other bases. Here is code which converts a number n in base b (2<=b<=10) to a decimal number:

```
public int toDecimal(int n, int b)
{
   int result=0;
   int multiplier=1;

   while(n>0)
   {
      result+=n%10*multiplier;
      multiplier*=b;
      n/=10;
   }

   return result;
}
```

Java users will be happy to know that the above can be also written as:

```
return Integer.parseInt(""+n,b);
```

To convert from a decimal to a binary is just as easy. Suppose we wanted to convert 43 in decimal to binary. At each step of the method we divide 43 by 2 and memorize the remainder. The final list of remainders is the required binary representation:

```
43/2 = 21 + remainder 1
21/2 = 10 + remainder 1
10/2 = 5  + remainder 0
5/2  = 2  + remainder 1
2/2  = 1  + remainder 0
1/2  = 0  + remainder 1
```

So 43 in decimal is 101011 in binary. By swapping all occurrences of 10 with b in our previous method we create a function which converts from a decimal number n to a number in base b (2<=b<=10):

```
public int fromDecimal(int n, int b)
{
   int result=0;
   int multiplier=1;

   while(n>0)
   {
      result+=n%b*multiplier;
      multiplier*=10;
      n/=b;
   }

   return result;
}
```

If the base b is above 10 then we must use non-numeric characters to represent digits that have a value of 10 and more. We can let 'A' stand for 10, 'B' stand for 11 and so on. The following code will convert from a decimal to any base (up to base 20):

```
public String fromDecimal2(int n, int b)
{
   String chars="0123456789ABCDEFGHIJ";
   String result="";

   while(n>0)
   {
      result=chars.charAt(n%b) + result;
      n/=b;
   }

   return result;
}
```

In Java there are some useful shortcuts when converting from decimal to other common representations, such as binary (base 2), octal (base 8) and hexadecimal (base 16):

```
Integer.toBinaryString(n);
Integer.toOctalString(n);
Integer.toHexString(n);
```

Fractions and Complex Numbers

Fractional numbers can be seen in many problems. Perhaps the most difficult aspect of dealing with fractions is finding the right way of representing them. Although it is possible to create a fractions class containing the required attributes and methods, for most purposes it is sufficient to represent fractions as 2-element arrays (pairs). The idea is that we store the numerator in the first element and the denominator in the second element. We will begin with multiplication of two fractions a and b:

```
public int[] multiplyFractions(int[] a, int[] b)
{
   int[] c={a[0]*b[0], a[1]*b[1]};
   return c;
}
```

Adding fractions is slightly more complicated, since only fractions with the same denominator can be added together. First of all we must find the common denominator of the two fractions and then use multiplication to transform the fractions such that they both have the common denominator as their denominator. The common denominator is a number which can divide both denominators and is simply the LCM (defined

earlier) of the two denominators. For example lets add 4/9 and 1/6. LCM of 9 and 6 is 18. Thus to transform the first fraction we need to multiply it by 2/2 and multiply the second one by 3/3:

```
4/9 + 1/6 = (4*2)/(9 * 2) + (1 * 3)/(6 * 3) =  8/18 +  3/18
```

Once both fractions have the same denominator, we simply add the numerators to get the final answer of 11/18. Subtraction is very similar, except we subtract at the last step:

```
4/9 - 1/6 =  8/18 -  3/18 =  5/18
```

Here is the code to add two fractions:

```
public int[] addFractions(int[] a, int[] b)
{
   int denom=LCM(a[1],b[1]);
   int[] c={denom/a[1]*a[0] + denom/b[1]*b[0], denom};
   return c;
}
```

Finally it is useful to know how to reduce a fraction to its simplest form. The simplest form of a fraction occurs when the GCD of the numerator and denominator is equal to 1. We do this like so:

```
public void reduceFraction(int[] a)
{
   int b=GCD(a[0],a[1]);
   a[0]/=b;
   a[1]/=b;
}
```

Using a similar approach we can represent other special numbers, such as complex numbers. In general, a complex number is a number of the form a + ib, where a and b are reals and i is the square root of -1. For example, to add two complex numbers m = a + ib and n = c + id we simply group likewise terms:

```
m + n
= (a + ib) + (c + id)
= (a + c) + i(b + d)
```

Multiplying two complex numbers is the same as multiplying two real numbers, except we must use the fact that i^2 = -1:

```
m * n
= (a + ib) * (c + id)
= ac + iad + ibc + (i^2)bd
= (ac - bd) + i(ad + bc)
```

By storing the real part in the first element and the complex part in the second element of the 2-element array we can write code that performs the above multiplication:

```
public int[] multiplyComplex(int[] m, int[] n)
{
   int[] prod = {m[0]*n[0] - m[1]*n[1], m[0]*n[1] + m[1]*n[0]};
   return prod;
}
```

Conclusion

In conclusion I want to add that one cannot rise to the top of the TopCoder rankings without understanding the mathematical constructs and algorithms outlined in this article. Perhaps one of the most common topics in mathematical problems is the topic of primes. This is closely followed by the topic of bases, probably because computers operate in binary and thus one needs to know how to convert from binary to decimal. The concepts of GCD and LCM are common in both pure mathematics as well as geometrical problems. Finally, I have included the last topic not so much for its usefulness in TopCoder competitions, but more because it demonstrates a means of treating certain numbers.

Introduction

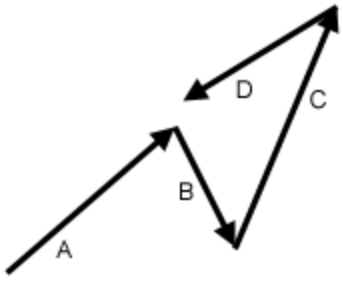
Many TopCoders seem to be mortally afraid of geometry problems. I think it's safe to say that the majority of them would be in favor of a ban on TopCoder geometry problems. However, geometry is a very important part of most graphics programs, especially computer games, and geometry problems are here to stay. In this article, I'll try to take a bit of the edge off of them, and introduce some concepts that should make geometry problems a little less frightening.

Vectors

Vectors are the basis of a lot of methods for solving geometry problems. Formally, a vector is defined by a direction and a magnitude. In the case of two-dimension geometry, a vector can be represented as pair of numbers, x and y, which gives both a direction and a magnitude. For example, the line segment from (1,3) to (5,1) can be represented by the vector (4,-2). It's important to understand, however, that the vector defines only the direction and magnitude of the segment in this case, and does not define the starting or ending locations of the vector.

Vector Addition

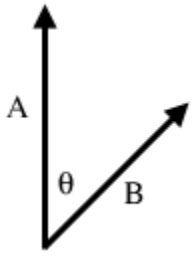
There are a number of mathematical operations that can be performed on vectors. The simplest of these is addition: you can add two vectors together and the result is a new vector. If you have two vectors (x_1, y_1) and (x_2, y_2), then the sum of the two vectors is simply (x_1+x_2, y_1+y_2). The image below shows the sum of four vectors. Note that it doesn't matter which order you add them up in - just like regular addition. Throughout these articles, we will use plus and minus signs to denote vector addition and subtraction, where each is simply the piecewise addition or subtraction of the components of the vector.



The sum of vectors A+B+C+D

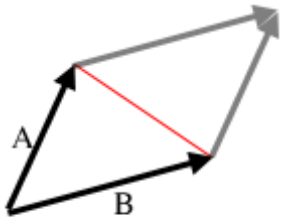
Dot Product

The addition of vectors is relatively intuitive; a couple of less obvious vector operations are dot and cross products. The dot product of two vectors is simply the sum of the products of the corresponding elements. For example, the dot product of (x_1, y_1) and (x_2, y_2) is $x_1*x_2 + y_1*y_2$. Note that this is not a vector, but is simply a single number (called a scalar). The reason this is useful is that the dot product, $A \cdot B = |A||B|Cos(\theta)$, where θ is the angle between the A and B. $|A|$ is called the norm of the vector, and in a 2-D geometry problem is simply the length of the vector, $sqrt(x^2+y^2)$. Therefore, we can calculate $Cos(\theta) = (A \cdot B)/(|A||B|)$. By using the acos function, we can then find θ. It is useful to recall that $Cos(90) = 0$ and $Cos(0) = 1$, as this tells you that a dot product of 0 indicates two perpendicular lines, and that the dot product is greatest when the lines are parallel. A final note about dot products is that they are not limited to 2-D geometry. We can take dot products of vectors with any number of elements, and the above equality still holds.



Cross Product

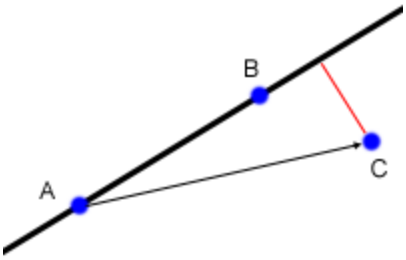
An even more useful operation is the cross product. The cross product of two 2-D vectors is $x_1*y_2 - y_1*x_2$ Technically, the cross product is actually a vector, and has the magnitude given above, and is directed in the +z direction. Since we're only working with 2-D geometry for now, we'll ignore this fact, and use it like a scalar. Similar to the dot product, $A \times B = |A||B|Sin(\theta)$. However, θ has a slightly different meaning in this case: $|\theta|$ is the angle between the two vectors, but θ is negative or positive based on the right-hand rule. In 2-D geometry this means that if A is less than 180 degrees clockwise from B, the value is positive. Another useful fact related to the cross product is that the absolute value of $|A||B|Sin(\theta)$ is equal to the area of the parallelogram with two of its sides formed by A and B. Furthermore, the triangle formed by A, B and the red line in the diagram has half of the area of the parallelogram, so we can calculate its area from the cross product also.



Parallelogram from A and B

Line-Point Distance

Finding the distance from a point to a line is something that comes up often in geometry problems. Lets say that you are given 3 points, A, B, and C, and you want to find the distance from the point C to the line defined by A and B (recall that a line extends infinitely in either direction). The first step is to find the two vectors from A to B (AB) and from A to C (AC). Now, take the cross product AB x AC, and divide by |AB|. This gives you the distance (denoted by the red line) as (AB x AC)/|AB|. The reason this works comes from some basic high school level geometry. The area of a triangle is found as base*height/2. Now, the area of the triangle formed by A, B and C is given by (AB x AC)/2. The base of the triangle is formed by AB, and the height of the triangle is the distance from the line to C. Therefore, what we have done is to find twice the area of the triangle using the cross product, and then divided by the length of the base. As always with cross products, the value may be negative, in which case the distance is the absolute value.



Things get a little bit trickier if we want to find the distance from a line segment to a point. In this case, the nearest point might be one of the endpoints of the segment, rather than the closest point on the line. In the diagram above, for example, the closest point to C on the line defined by A and B is not on the segment AB, so the point closest to C is B. While there are a few different ways to check for this special case, one way is to apply the dot product. First, check to see if the nearest point on the line AB is beyond B (as in the example above) by taking AB · BC. If this value is greater than 0, it means that the angle between AB and BC is between -90 and 90, exclusive, and therefore the nearest point on the segment AB will be B. Similarly, if BA · AC is greater than 0, the nearest point is A. If both dot products are negative, then the nearest point to C is somewhere along the segment.

```
    //Compute the dot product AB · BC
    int dot(int[] A, int[] B, int[] C){
        AB = new int[2];
        BC = new int[2];
        AB[0] = B[0]-A[0];
        AB[1] = B[1]-A[1];
        BC[0] = C[0]-B[0];
        BC[1] = C[1]-B[1];
        int dot = AB[0] * BC[0] + AB[1] * BC[1];
        return dot;
    }
    //Compute the cross product AB x AC
    int cross(int[] A, int[] B, int[] C){
        AB = new int[2];
        AC = new int[2];
        AB[0] = B[0]-A[0];
        AB[1] = B[1]-A[1];
        AC[0] = C[0]-A[0];
        AC[1] = C[1]-A[1];
        int cross = AB[0] * AC[1] - AB[1] * AC[0];
        return cross;
    }
    //Compute the distance from A to B
    double distance(int[] A, int[] B){
        int d1 = A[0] - B[0];
        int d2 = A[1] - B[1];
        return sqrt(d1*d1+d2*d2);
```

```
    }
```

```
    //Compute the distance from AB to C
    //if isSegment is true, AB is a segment, not a line.
    double linePointDist(int[] A, int[] B, int[] C, boolean isSegment){
        double dist = cross(A,B,C) / distance(A,B);
        if(isSegment){
            int dot1 = dot(A,B,C);
            if(dot1 > 0)return distance(B,C);
            int dot2 = dot(B,A,C);
            if(dot2 > 0)return distance(A,C);
        }
        return abs(dist);
    }
```

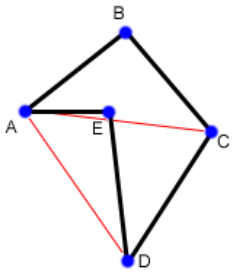
That probably seems like a lot of code, but lets see the same thing with a point class and some operator overloading in C++ or C#. The * operator is the dot product, while ^ is cross product, while + and - do what you would expect.

```
    //Compute the distance from AB to C
    //if isSegment is true, AB is a segment, not a line.
    double linePointDist(point A, point B, point C, bool isSegment){
        double dist = ((B-A)^(C-A)) / sqrt((B-A)*(B-A));
        if(isSegment){
            int dot1 = (C-B)*(B-A);
            if(dot1 > 0)return sqrt((B-C)*(B-C));
            int dot2 = (C-A)*(A-B);
            if(dot2 > 0)return sqrt((A-C)*(A-C));
        }
        return abs(dist);
    }
```

Operator overloading is beyond the scope of this article, but I suggest that you look up how to do it if you are a C# or C++ coder, and write your own 2-D point class with some handy operator overloading. It will make a lot of geometry problems a lot simpler.

Polygon Area

Another common task is to find the area of a polygon, given the points around its perimeter. Consider the non-convex polygon below, with 5 points. To find its area we are going to start by triangulating it. That is, we are going to divide it up into a number of triangles. In this polygon, the triangles are ABC, ACD, and ADE. But wait, you protest, not all of those triangles are part of the polygon! We are going to take advantage of the signed area given by the cross product, which will make everything work out nicely. First, we'll take the cross product of AB x AC to find the area of ABC. This will give us a negative value, because of the way in which A, B and C are oriented. However, we're still going to add this to our sum, as a negative number. Similarly, we will take the cross product AC x AD to find the area of triangle ACD, and we will again get a negative number. Finally, we will take the cross product AD x AE and since these three points are oriented in the opposite direction, we will get a positive number. Adding these three numbers (two negatives and a positive) we will end up with a negative number, so will take the absolute value, and that will be area of the polygon.



The reason this works is that the positive and negative number cancel each other out by exactly the right amount. The area of ABC and ACD ended up contributing positively to the final area, while the area of ADE contributed negatively. Looking at the original polygon, it is obvious that the area of the polygon is the area of ABCD (which is the same as ABC + ABD) minus the area of ADE. One final note, if the total area we end up with is negative, it means that the points in the polygon were given to us in clockwise order. Now, just to make this a little more concrete, lets write a little bit of code to find the area of a polygon, given the coordinates as a 2-D array, p.

```
int area = 0;
int N = lengthof(p);
//We will triangulate the polygon
//into triangles with points p[0],p[i],p[i+1]

for(int i = 1; i+1<N; i++){
```

```
    int x1 = p[i][0] - p[0][0];
    int y1 = p[i][1] - p[0][1];
    int x2 = p[i+1][0] - p[0][0];
    int y2 = p[i+1][1] - p[0][1];
    int cross = x1*y2 - x2*y1;
    area += cross;
}
return abs(cross/2.0);
```

Notice that if the coordinates are all integers, then the final area of the polygon is one half of an integer.

Line-Line Intersection

One of the most common tasks you will find in geometry problems is line intersection. Despite the fact that it is so common, a lot of coders still have trouble with it. The first question is, what form are we given our lines in, and what form would we like them in? Ideally, each of our lines will be in the form Ax+By=C, where A, B and C are the numbers which define the line. However, we are rarely given lines in this format, but we can easily generate such an equation from two points. Say we are given two different points, (x_1, y_1) and (x_2, y_2), and want to find A, B and C for the equation above. We can do so by setting

$A = y_2-y_1$

$B = x_1-x_2$

$C = A*x_1+B*y_1$

Regardless of how the lines are specified, you should be able to generate two different points along the line, and then generate A, B and C. Now, lets say that you have lines, given by the equations:

$A_1x + B_1y = C_1$

$A_2x + B_2y = C_2$

To find the point at which the two lines intersect, we simply need to solve the two equations for the two unknowns, x and y.

```
    double det = A1*B2 - A2*B1
    if(det == 0){
        //Lines are parallel
    }else{
        double x = (B2*C1 - B1*C2)/det
        double y = (A1*C2 - A2*C1)/det
    }
```

To see where this comes from, consider multiplying the top equation by B_2, and the bottom equation by B_1. This gives you

$A_1B_2x + B_1B_2y = B_2C_1$

$A_2B_1x + B_1B_2y = B_1C_2$

Now, subtract the bottom equation from the top equation to get

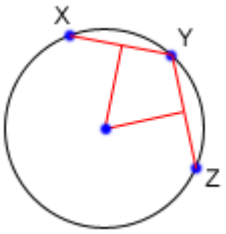
$A_1B_2x - A_2B_1x = B_2C_1 - B_1C_2$

Finally, divide both sides by $A_1B_2 - A_2B_1$, and you get the equation for x. The equation for y can be derived similarly.

This gives you the location of the intersection of two lines, but what if you have line segments, not lines. In this case, you need to make sure that the point you found is on both of the line segments. If your line segment goes from (x_1,y_1) to (x_2,y_2), then to check if (x,y) is on that segment, you just need to check that $\min(x_1,x_2) \le x \le \max(x_1,x_2)$, and do the same thing for y. You must be careful about double precision issues though. If your point is right on the edge of the segment, or if the segment is horizontal or vertical, a simple comparison might be problematic. In these cases, you can either do your comparisons with some tolerance, or else use a fraction class.

Finding a Circle From 3 Points

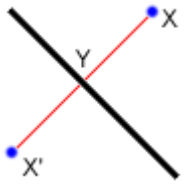
Given 3 points which are not colinear (all on the same line) those three points uniquely define a circle. But, how do you find the center and radius of that circle? This task turns out to be a simple application of line intersection. We want to find the perpendicular bisectors of XY and YZ, and then find the intersection of those two bisectors. This gives us the center of the circle.



To find the perpendicular bisector of XY, find the line from X to Y, in the form Ax+By=C. A line perpendicular to this line will be given by the equation -Bx+Ay=D, for some D. To find D for the particular line we are interested in, find the midpoint between X and Y by taking the midpoint of the x and y components independently. Then, substitute those values into the equation to find D. If we do the same thing for Y and Z, we end up with two equations for two lines, and we can find their intersections as described above.

Reflection

Reflecting a point across a line requires the same techniques as finding a circle from 3 points. First, notice that the distance from X to the line of reflection is the same as the distance from X' to the line of reflection. Also note that the line between X and X' is perpendicular to the line of reflection. Now, if the line of reflection is given as Ax+By=C, then we already know how to find a line perpendicular to it: -Bx+Ay=D. To find D, we simply plug in the coordinates for X. Now, we can find the intersection of the two lines at Y, and then find X' = Y - (X - Y).



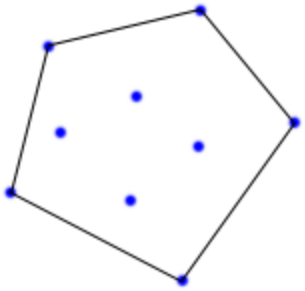
Rotation

Rotation doesn't really fit in with line intersection, but I felt that it would be good to group it with reflection. In fact, another way to find the reflected point is to rotate the original point 180 degrees about Y.

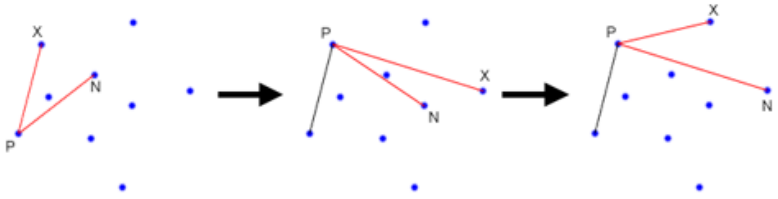
Imagine that we want to rotate one point around another, counterclockwise by θ degrees. For simplicity, lets assume that we are rotating about the origin. In this case, we can find that x' = x Cos(θ) - y Sin(θ) and y' = x Sin(θ) + y Cos(θ). If we are rotating about a point other than the origin, we can account for this by shifting our coordinate system so that the origin is at the point of rotation, doing the rotation with the above formulas, and then shifting the coordinate system back to where it started.

Convex Hull

A convex hull of a set of points is the smallest convex polygon that contains every one of the points. It is defined by a subset of all the points in the original set. One way to think about a convex hull is to imagine that each of the points is a peg sticking up out of a board. Take a rubber band and stretch it around all of the points. The polygon formed by the rubber band is a convex hull. There are many different algorithms that can be used to find the convex hull of a set of points. In this article, I'm just going to describe one of them, which is fast enough for most purposes, but is quite slow compared to some of the other algorithms.



First, loop through all of your points and find the leftmost point. If there is a tie, pick the highest point. You know for certain that this point will be on the convex hull, so we'll start with it. From here, we are going to move clockwise around the edge of the hull, picking the points on the hull, one at a time. Eventually, we will get back to the start point. In order to find the next point around the hull, we will make use of cross products. First, we will pick an unused point, and set the next point, N, to that point. Next, we will iterate through each unused points, X, and if (X-P) x (N-P) (where P is the previous point) is negative, we will set N to X. After we have iterated through each point, we will end up with the next point on the convex hull. See the diagram below for an illustration of how the algorithm works. We start with P as the leftmost point. Now, say that we have N and X as shown in the leftmost frame. In this case the cross product will be negative, so we will set N = X, and there will be no other unused points that make the cross product negative, and hence we will advance, setting P = N. Now, in the next frame, we will end up setting N = X again, since the cross product here will be negative. However, we aren't done yet because there is still another point that will make the cross product negative, as shown in the final frame.



The basic idea here is that we are using the cross product to find the point which is furthest counterclockwise from our current position at P. While this may seem fairly straightforward, it becomes a little bit tricky when dealing with colinear points. If you have no colinear points on the hull, then the code is very straightforward.

```
convexHull(point[] X){
    int N = lengthof(X);
    int p = 0;
    //First find the leftmost point
    for(int i = 1; i<N; i++){
        if(X[i] < X[p])
            p = i;
    }
    int start = p;
    do{
        int n = -1;
        for(int i = 0; i<N; i++){

            //Don't go back to the same point you came from
            if(i == p)continue;

            //If there is no N yet, set it to i
            if(n == -1)n = i;
            int cross = (X[i] - X[p]) x (X[n] - X[p]);

            if(cross < 0){
                //As described above, set N=X
                n = i;
            }
        }
        p = n;
    }while(start!=p);
}
```

Once we start to deal with colinear points, things get trickier. Right away we have to change our method signature to take a boolean specifying whether to include all of the colinear points, or only the necessary ones.

```
//If onEdge is true, use as many points as possible for
//the convex hull, otherwise as few as possible.
convexHull(point[] X, boolean onEdge){
    int N = lengthof(X);
    int p = 0;
    boolean[] used = new boolean[N];
    //First find the leftmost point
    for(int i = 1; i<N; i++){
        if(X[i] < X[p])
            p = i;
    }
    int start = p;
    do{
        int n = -1;
        int dist = onEdge?INF:0;
        for(int i = 0; i<N; i++){
            //X[i] is the X in the discussion

            //Don't go back to the same point you came from
            if(i==p)continue;

            //Don't go to a visited point
            if(used[i])continue;

            //If there is no N yet, set it to X
            if(n == -1)n = i;
            int cross = (X[i] - X[p]) x (X[n] - X[p]);

            //d is the distance from P to X
            int d = (X[i] - X[p]) · (X[i] - X[p]);
            if(cross < 0){
                //As described above, set N=X
                n = i;
                dist = d;
            }else if(cross == 0){
```

```
                    //In this case, both N and X are in the
                    //same direction.  If onEdge is true, pick the
                    //closest one, otherwise pick the farthest one.
                    if(onEdge && d < dist){
                        dist = d;
                        n = i;
                    }else if(!onEdge && d > dist){
                        dist = d;
                        n = i;
                    }
                }
            }
            p = n;
            used[p] = true;
        }while(start!=p);
    }
```

PointInPolygon (SRM 187)

Requires: Line-Line Intersection, Line-Point Distance

First off, we can use our Line-Point Distance code to test for the "BOUNDARY" case. If the distance from any segment to the test point is 0, then return "BOUNDARY". If you didn't have that code pre-written, however, it would probably be easier to just check and see if the test point is between the minimum and maximum x and y values of the segment. Since all of the segments are vertical or horizontal, this is sufficient, and the more general code is not necessary.

Next we have to check if a point is in the interior or the exterior. Imagine picking a point in the interior and then drawing a ray from that point out to infinity in some direction. Each time the ray crossed the boundary of the polygon, it would cross from the interior to the exterior, or vice versa. Therefore, the test point is on the interior if, and only if, the ray crosses the boundary an odd number of times. In practice, we do not have to draw a raw all the way to infinity. Instead, we can just use a very long line segment from the test point to a point that is sufficiently far away. If you pick the far away point poorly, you will end up having to deal with cases where the long segment touches the boundary of the polygon where two edges meet, or runs parallel to an edge of a polygon — both of which are tricky cases to deal with. The quick and dirty way around this is to pick two large random numbers for the endpoint of the segment. While this might not be the most elegant solution to the problem, it works very well in practice. The chance of this segment intersecting anything but the interior of an edge are so small that you are almost guaranteed to get the right answer. If you are really concerned, you could pick a few different random points, and take the most common answer.

```
String testPoint(verts, x, y){
    int N = lengthof(verts);
    int cnt = 0;
    double x2 = random()*1000+1000;
    double y2 = random()*1000+1000;
    for(int i = 0; i<N; i++){
        if(distPointToSegment(verts[i],verts[(i+1)%N],x,y) == 0)
            return "BOUNDARY";
        if(segmentsIntersect((verts[i],verts[(i+1)%N],{x,y},{x2,y2}))
            cnt++;
    }
    if(cnt%2 == 0)return "EXTERIOR";
    else return "INTERIOR";
}
```

TVTower(SRM 183)

Requires: Finding a Circle From 3 Points

In problems like this, the first thing to figure out is what sort of solutions might work. In this case, we want to know what sort of circles we should consider. If a circle only has two points on it, then, in most cases, we can make a slightly smaller circle, that still has those two points on it. The only exception to this is when the two points are exactly opposite each other on the circle. Three points, on the other hand, uniquely define a circle, so if there are three points on the edge of a circle, we cannot make it slightly smaller, and still have all three of them on the circle. Therefore, we want to consider two different types of circles: those with two points exactly opposite each other, and those with three points on the circle. Finding the center of the first type of circle is trivial — it is simply halfway between the two points. For the other case, we can use the method for Finding a Circle From 3 Points. Once we find the center of a potential circle, it is then trivial to find the minimum radius.

```
int[] x, y;
int N;
```

```
double best = 1e9;
void check(double cx, double cy){
    double max = 0;
    for(int i = 0; i< N; i++){
        max = max(max,dist(cx,cy,x[i],y[i]));
    }
    best = min(best,max);
}
double minRadius(int[] x, int[] y){
    this.x = x;
    this.y = y;
    N = lengthof(x);
    if(N==1)return 0;
    for(int i = 0; i<N; i++){
        for(int j = i+1; j<N; j++){
            double cx = (x[i]+x[j])/2.0;
            double cy = (y[i]+y[j])/2.0;
            check(cx,cy);
            for(int k = j+1; k<N; k++){
                //center gives the center of the circle with
                //(x[i],y[i]), (x[j],y[j]), and (x[k],y[k]) on
                //the edge of the circle.
                double[] c = center(i,j,k);
                check(c[0],c[1]);
            }
        }
    }
    return best;
}
```

Satellites (SRM 180)

Requires: Line-Point Distance

This problem actually requires an extension of the Line-Point Distance method discussed previously. It is the same basic principle, but the formula for the cross product is a bit different in three dimensions.

The first step here is to convert from spherical coordinates into (x,y,z) triples, where the center of the earth is at the origin.

```
    double x = sin(lng/180*PI)*cos(lat/180*PI)*alt;
    double y = cos(lng/180*PI)*cos(lat/180*PI)*alt;
    double z = sin(lat/180*PI)*alt;
```

Now, we want to take the cross product of two 3-D vectors. As I mentioned earlier, the cross product of two vectors is actually a vector, and this comes into play when working in three dimensions. Given vectors (x_1,y_1,z_1) and (x_2,y_2,z_2) the cross product is defined as the vector (i,j,k) where

```
    i = y1z2 - y2z1;
    j = x2z1 - x1z2;
    k = x1y2 - x2y1;
```

Notice that if $z_1 = z_2 = 0$, then i and j are 0, and k is equal to the cross product we used earlier. In three dimensions, the cross product is still related to the area of the parallelogram with two sides from the two vectors. In this case, the area of the parallelogram is the norm of the vector: sqrt(i*i+j*j+k*k).

Hence, as before, we can determine the distance from a point (the center of the earth) to a line (the line from a satellite to a rocket). However, the closest point on the line segment between a satellite and a rocket may be one of the end points of the segment, not the closest point on the line. As before, we can use the dot product to check this. However, there is another way which is somewhat simpler to code. Say that you have two vectors originating at the origin, S and R, going to the satellite and the rocket, and that $|X|$ represents the norm of a vector X. Then, the closest point to the origin is R if $|R|^2 + |R-S|^2 \le |S|^2$ and it is S if $|S|^2 + |R-S|^2 \le |R|^2$
Naturally, this trick works in two dimensions also.

Further Problems

Once you think you've got a handle on the three problems above, you can give these ones a shot. You should be able to solve all of them with the methods I've outlined, and a little bit of cleverness. I've arranged them in what I believe to ascending order of difficulty.

ConvexPolygon (SRM 166)

Requires: Polygon Area

Surveyor (TCCC '04 Qual 1)
Requires: Polygon Area

Travel (TCI '02)
Requires: Dot Product

Parachuter (TCI '01 Round 3)
Requires: Point In Polygon, Line-Line Intersection

PuckShot (SRM 186)
Requires: Point In Polygon, Line-Line Intersection

ElectronicScarecrows (SRM 173)
Requires: Convex Hull, Dynamic Programming

Mirrors (TCI '02 Finals)
Requires: Reflection, Line-Line Intersection

Symmetry (TCI '02 Round 4)
Requires: Reflection, Line-Line Intersection

Warehouse (SRM 177)
Requires: Line-Point Distance, Line-Line Intersection

The following problems all require geometry, and the topics discussed in this article will be useful. However, they all require some additional skills. If you got stuck on them, the editorials are a good place to look for a bit of help. If you are still stuck, there has yet to be a problem related question on the round tables that went unanswered.

DogWoods (SRM 201)
ShortCut (SRM 215)
SquarePoints (SRM 192)
Tether (TCCC '03 W/MW Regional)
TurfRoller (SRM 203)
Watchtower (SRM 176)

Introduction
Graphs are a fundamental data structure in the world of programming, and this is no less so on TopCoder. Usually appearing as the hard problem in Division 2, or the medium or hard problem in Division 1, there are many different forms solving a graph problem can take. They can range in difficulty from finding a path on a 2D grid from a start location to an end location, to something as hard as finding the maximum amount of water that you can route through a set of pipes, each of which has a maximum capacity (also known as the maximum-flow minimum-cut problem - which we will discuss later). Knowing the correct data structures to use with graph problems is critical. A problem that appears intractable may prove to be a few lines with the proper data structure, and luckily for us the standard libraries of the languages used by TopCoder help us a great deal here!

Recognizing a graph problem
The first key to solving a graph related problem is recognizing that it is a graph problem. This can be more difficult than it sounds, because the problem writers don't usually spell it out for you. Nearly all graph problems will somehow use a grid or network in the problem, but sometimes these will be well disguised. Secondly, if you are required to find a path of any sort, it is usually a graph problem as well. Some common keywords associated with graph problems are: vertices, nodes, edges, connections, connectivity, paths, cycles and direction. An example of a description of a simple problem that exhibits some of these characteristics is:

"Bob has become lost in his neighborhood. He needs to get from his current position back to his home. Bob's neighborhood is a 2 dimensional grid, that starts at (0, 0) and (width - 1, height - 1). There are empty spaces upon which bob can walk with no difficulty, and houses, which Bob cannot pass through. Bob may only move horizontally or vertically by one square at a time.

Bob's initial position will be represented by a 'B' and the house location will be represented by an 'H'. Empty squares on the grid are represented by '.' and houses are represented by 'X'. Find the minimum number of steps it takes Bob to get back home, but if it is not possible for Bob to return home, return -1.

An example of a neighborhood of width 7 and height 5:
...X..B
.X.X.XX
.H.....
...X...
.....X."
Once you have recognized that the problem is a graph problem it is time to start building up your representation of the graph in memory.

Representing a graph and key concepts
Graphs can represent many different types of systems, from a two-dimensional grid (as in the problem above) to a map of the internet that shows how long it takes data to move from computer A to computer B. We first need to define what components a graph consists of. In fact there are only two, nodes and edges. A node (or vertex) is a discrete position in the graph. An edge (or connection) is a link between two vertices that can be either directed or undirected and may have a cost associated with it. An undirected edge means that there is no restriction on the direction you can travel along the edge. So for example, if there were an undirected edge from A to B you could move from A to B or from B to A. A directed edge only allows travel in one direction, so if there were a directed edge from A to B you could travel from A to B, but not from B to A. An easy way to think about edges and vertices is that edges are a function of two vertices that returns a cost. We will see an example of this methodology in a second.

For those that are used to the mathematical description of graphs, a graph G = {V, E} is defined as a set of vertices, V, and a collection of edges (which is not necessarily a set), E. An edge can then be defined as (u, v) where u and v are elements of V. There are a few technical terms that it would be useful to discuss at this point as well:

Order - The number of vertices in a graph Size - The number of edges in a graph

Singly linked lists
An example of one of the simplest types of graphs is a singly linked list! Now we can start to see the power of the graph data structure, as it can represent very complicated relationships, but also something as simple as a list.

A singly linked list has one "head" node, and each node has a link to the next node. So the structure looks like this:

```
structure node
   [node] link;
   [data]
end

node head;
A simple example would be:
node B, C;
head.next = B;
B.next = C;
C.next = null;
```

This would be represented graphically as head -> B -> C -> null. I've used null here to represent the end of a list.

Getting back to the concept of a cost function, our cost function would look as follows:

```
cost(X, Y) := if (X.link = Y) return 1;
              else if (X = Y) return 0;
              else "Not possible"
```

This cost function represents the fact that we can only move directly to the link node from our current node. Get used to seeing cost functions because anytime that you encounter a graph problem you will be dealing with them in some form or another! A question that you may be asking at this point is "Wait a second, the cost from A to C would return not possible, but I can get to C from A by stepping through B!" This is a very valid point, but the cost function simply encodes the *direct* cost from a node to another. We will cover how to find distances in generic graphs later on.

Now that we have seen an example of the one of the simplest types of graphs, we will move to a more complicated example.

Trees

There will be a whole section written on trees. We are going to cover them very briefly as a stepping-stone along the way to a full-fledged graph. In our list example above we are somewhat limited in the type of data we can represent. For example, if you wanted to start a family tree (a hierarchal organization of children to parents, starting from one child) you would not be able to store more than one parent per child. So we obviously need a new type of data structure. Our new node structure will look something like this:

```
structure node
   [node] mother, father;
   [string] name
end

node originalChild;
With a cost function of:
cost(X, Y) := if ((X.mother = Y) or (X.father = Y)) return 1;
              else if (X = Y) return 0;
              else "Not possible"
```

Here we can see that every node has a mother and father. And since node is a recursive structure definition, every mother has mother and father, and every father has a mother and father, and so on. One of the problems here is that it might be possible to form a loop if you actually represented this data structure on a computer. And a tree clearly cannot have a loop. A little mind exercise will make this clear: a father of a child is also the son of that child? It's starting to make my head hurt already. So you have to be very careful when constructing a tree to make sure that it is truly a tree structure, and not a more general graph. A more formal definition of a tree is that it is a connected acyclic graph. This simply means that there are no cycles in the graph and every node is connected to at least one other node in the graph.

Another thing to note is that we could imagine a situation easily where the tree requires more than two node references, for example in an organizational hierarchy, you can have a manager who manages many people then the CEO manages many managers. Our example above was what is known as a binary tree, since it only has two node references. Next we will move onto constructing a data structure that can represent a general graph!

Graphs

A tree only allows a node to have children, and there cannot be any loops in the tree, with a more general graph we can represent many different situations. A very common example used is flight paths between cities. If there is a flight between city A and city B there is an edge between the cities. The cost of the edge can be the length of time that it takes for the flight, or perhaps the amount of fuel used.

The way that we will represent this is to have a concept of a node (or vertex) that contains links to other nodes, and the data associated with that node. So for our flight path example we might have the name of the airport as the node data, and for every flight leaving that city we have an element in neighbors that points to the destination.

```
structure node
   [list of nodes] neighbors
   [data]
end

cost(X, Y) := if (X.neighbors contains Y) return X.neighbors[Y];
              else "Not possible"

list nodes;
```

This is a very general way to represent a graph. It allows us to have multiple edges from one node to another and it is a very compact representation of a graph as well. However the downside is that it is usually more difficult to work with than other representations (such as the array method discussed below).

Array representation

Representing a graph as a list of nodes is a very flexible method. But usually on TopCoder we have limits on the problems that attempt to make life easier for us. Normally our graphs are relatively small, with a small number of nodes and edges. When this is the case we can use a different

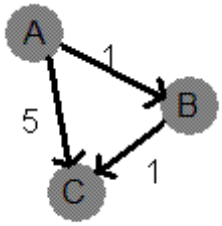
type of data structure that is easier to work with.

The basic concept is to have a 2 dimensional array of integers, where the element in row i, at column j represents the edge cost from node i to j. If the connection from i to j is not possible, we use some sort of sentinel value (usually a very large or small value, like -1 or the maximum integer). Another nice thing about this type of structure is that we can represent directed or undirected edges very easily.

So for example, the following connection matrix:

```
   A  B  C
A  0  1  5
B -1  0  1
C -1 -1  0
```

Would mean that node A has a 0 weight connection to itself, a 1 weight connection to node B and 5 weight connection to node C. Node B on the other hand has no connection to node A, a 0 weight connection to itself, and a 1 weight connection to C. Node C is connected to nobody. This graph would look like this if you were to draw it:



This representation is very convenient for graphs that do not have multiple edges between each node, and allows us to simplify working with the graph.

Basic methods for searching graphs

Introduction

So far we have learned how to represent our graph in memory, but now we need to start doing something with this information. There are two methods for searching graphs that are extremely prevalent, and will form the foundations for more advanced algorithms later on. These two methods are the Depth First Search and the Breadth First Search.

We will begin with the depth first search method, which will utilize a stack. This stack can either by represented explicitly (by a stack data-type in our language) or implicitly when using recursive functions.

Stack

A stack is one of the simplest data structures available. There are four main operations on a stack:

1. Push - Adds an element to the top of the stack
2. Pop - Removes the top element from the stack
3. Top - Returns the top element on the stack
4. Empty - Tests if the stack is empty or not

In C++, this is done with the STL class stack:

```
#include
stack myStack;
```

In Java, we use the Stack class:

```
import java.util.*;
Stack stack = new Stack();
```

In C#, we use Stack class:

```
using System.Collections;
Stack stack = new Stack();
```

Depth First Search

Now to solve an actual problem using our search! The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. A recent TopCoder problem was a classic application of the depth first search, the flood-fill. The flood-fill operation will be familiar to anyone who has used a graphic painting application. The concept is to fill a bounded region with a single color, without leaking outside the boundaries.

This concept maps extremely well to a Depth First search. The basic concept is to visit a node, then push all of the nodes to be visited onto the stack. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it, then unmarking it after we have finished our recursions. This action allows us to visit all the paths that exist in a graph; however for large graphs this is mostly infeasible so we sometimes omit the marking the node as not visited step to just find one valid path through the graph (which is good enough most of the time).

So the basic structure will look something like this:

```
dfs(node start) {
 stack s;
 s.push(start);
 while (s.empty() == false) {
  top = s.top();
  s.pop();
  mark top as visited;

  check for termination condition

  add all of top's unvisited neighbors to the stack.
  mark top as not visited;
 }
}
```

Alternatively we can define the function recursively as follows:

```
dfs(node current) {
 mark current as visited;
 visit all of current's unvisited neighbors by calling dfs(neighbor)
 mark current as not visited;
}
```

The problem we will be discussing is grafixMask, a Division 1 500 point problem from SRM 211. This problem essentially asks us to find the number of discrete regions in a grid that has been filled in with some values already. Dealing with grids as graphs is a very powerful technique, and in this case makes the problem quite easy.

We will define a graph where each node has 4 connections, one each to the node above, left, right and below. However, we can represent these connections implicitly within the grid, we need not build out any new data structures. The structure we will use to represent the grid in grafixMask is a two dimensional array of booleans, where regions that we have already determined to be filled in will be set to true, and regions that are unfilled are set to false.

To set up this array given the data from the problem is very simple, and looks something like this:

```
bool fill[600][400];
initialize fills to false;

foreach rectangle in Rectangles
  set from (rectangle.left, rectangle.top) to (rectangle.right, retangle.bottom) to true
```

Now we have an initialized connectivity grid. When we want to move from grid position (x, y) we can either move up, down, left or right. When we want to move up for example, we simply check the grid position in (x, y-1) to see if it is true or false. If the grid position is false, we can move there, if it is true, we cannot.

Now we need to determine the area of each region that is left. We don't want to count regions twice, or pixels twice either, so what we will do is set fill[x][y] to true when we visit the node at (x, y). This will allow us to perform a Depth-First search to visit all of the nodes in a connected region and never visit any node twice, which is exactly what the problem wants us to do! So our loop after setting everything up will be:

```
int[] result;

for x = 0 to 599
 for y = 0 to 399
  if (fill[x][y] == false)
   result.addToBack(doFill(x,y));
```

All this code does is check if we have not already filled in the position at (x, y) and then calls doFill() to fill in that region. At this point we have a choice, we can define doFill recursively (which is usually the quickest and easiest way to do a depth first search), or we can define it explicitly using the built in stack classes. I will cover the recursive method first, but we will soon see for this problem there are some serious issues with the recursive method.

We will now define doFill to return the size of the connected area and the start position of the area:

```
int doFill(int x, int y) {
// Check to ensure that we are within the bounds of the grid, if not, return 0
 if (x < 0 || x >= 600) return 0;
// Similar check for y
 if (y < 0 || y >= 400) return 0;
// Check that we haven't already visited this position, as we don't want to count it twice
 if (fill[x][y]) return 0;

// Record that we have visited this node
 fill[x][y] = true;
```

```
 // Now we know that we have at least one empty square, then we will recursively attempt to
 // visit every node adjacent to this node, and add those results together to return.
 return 1 + doFill(x - 1, y) + doFill(x + 1, y) + doFill(x, y + 1) + doFill(x, y - 1);
}
```

This solution should work fine, however there is a limitation due to the architecture of computer programs. Unfortunately, the memory for the implicit stack, which is what we are using for the recursion above is more limited than the general heap memory. In this instance, we will probably overflow the maximum size of our stack due to the way the recursion works, so we will next discuss the explicit method of solving this problem.

Sidenote:

Stack memory is used whenever you call a function; the variables to the function are pushed onto the stack by the compiler for you. When using a recursive function, the variables keep getting pushed on until the function returns. Also any variables the compiler needs to save between function calls must be pushed onto the stack as well. This makes it somewhat difficult to predict if you will run into stack difficulties. I recommend using the explicit Depth First search for every situation you are at least somewhat concerned about recursion depth.

In this problem we may recurse a maximum of 600 * 400 times (consider the empty grid initially, and what the depth first search will do, it will first visit 0,0 then 1,0, then 2,0, then 3,0 ... until 599, 0. Then it will go to 599, 1 then 598, 1, then 597, 1, etc. until it reaches 599, 399. This will push 600 * 400 * 2 integers onto the stack in the best case, but depending on what your compiler does it may in fact be more information. Since an integer takes up 4 bytes we will be pushing 1,920,000 bytes of memory onto the stack, which is a good sign we may run into trouble. We can use the same function definition, and the structure of the function will be quite similar, just we won't use any recursion any more:

```
class node { int x, y; }

int doFill(int x, int y) {
 int result = 0;

 // Declare our stack of nodes, and push our starting node onto the stack
 stack s;
 s.push(node(x, y));

 while (s.empty() == false) {
  node top = s.top();
  s.pop();

// Check to ensure that we are within the bounds of the grid, if not, continue
  if (top.x < 0 || top.x >= 600) continue;
// Similar check for y
  if (top.y < 0 || top.y >= 400) continue;
// Check that we haven't already visited this position, as we don't want to count it twice
  if (fill[top.x][top.y]) continue;

  fill[top.x][top.y] = true; // Record that we have visited this node

  // We have found this node to be empty, and part
  // of this connected area, so add 1 to the result
  result++;

  // Now we know that we have at least one empty square, then we will attempt to
  // visit every node adjacent to this node.
  s.push(node(top.x + 1, top.y));
  s.push(node(top.x - 1, top.y));
  s.push(node(top.x, top.y + 1));
  s.push(node(top.x, top.y - 1));
 }

 return result;
}
```

As you can see, this function has a bit more overhead to manage the stack structure explicitly, but the advantage is that we can use the entire memory space available to our program and in this case, it is necessary to use that much information. However, the structure is quite similar and if you compare the two implementations they are almost exactly equivalent.

Congratulations, we have solved our first question using a depth first search! Now we will move onto the depth-first searches close cousin the Breadth First search.

If you want to practice some DFS based problems, some good ones to look at are:

TCCC 03 Quarterfinals - Marketing - Div 1 500
TCCC 03 Semifinals Room 4 - Circuits - Div 1 275

Queue

A queue is a simple extension of the stack data type. Whereas the stack is a FILO (first-in last-out) data structure the queue is a FIFO (first-in first-out) data structure. What this means is the first thing that you add to a queue will be the first thing that you get when you perform a pop().

There are four main operations on a queue:

1. Push - Adds an element to the back of the queue
2. Pop - Removes the front element from the queue
3. Front - Returns the front element on the queue
4. Empty - Tests if the queue is empty or not

In C++, this is done with the STL class queue:

#include
queue myQueue;

In Java, we unfortunately don't have a Queue class, so we will approximate it with the LinkedList class. The operations on a linked list map well to a queue (and in fact, sometimes queues are implemented as linked lists), so this will not be too difficult.

The operations map to the LinkedList class as follows:

1. Push - boolean LinkedList.add(Object o)
2. Pop - Object LinkedList.removeFirst()
3. Front - Object LinkedList.getFirst()
4. Empty - int LinkedList.size()

import java.util.*;
LinkedList myQueue = new LinkedList();

In C#, we use Queue class:

The operations map to the Queue class as follows:

1. Push - void Queue.Enqueue(Object o)
2. Pop - Object Queue.Dequeue()
3. Front - Object Queue.Peek()
4. Empty - int Queue.Count

using System.Collections;
Queue myQueue = new Queue();

Breadth First Search

The Breadth First search is an extremely useful searching technique. It differs from the depth-first search in that it uses a queue to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property that if all of the edges in a graph are unweighted (or the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about what using a queue means to the search order. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the FIFO property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a Breadth First search is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source.

The basic structure of a breadth first search will look this:

```
void bfs(node start) {
 queue s;
 s.push(start);
 while (s.empty() == false) {
  top = s.front();
  s.pop();
  mark top as visited;
check for termination condition (have we reached the node we want to?) add all of top's unvisited
neighbors to the stack.
 }
}
```

Notice the similarities between this and a depth-first search, we only differ in the data structure used and we don't mark top as unvisited again.

The problem we will be discussing in relation to the Breadth First search is a bit harder than the previous example, as we are dealing with a slightly more complicated search space. The problem is the 1000 from Division 1 in SRM 156, Pathfinding. Once again we will be dealing in a grid-based problem, so we can represent the graph structure implicitly within the grid.

A quick summary of the problem is that we want to exchange the positions of two players on a grid. There are impassable spaces represented by 'X' and spaces that we can walk in represented by ' '. Since we have two players our node structure becomes a bit more complicated, we have to represent the positions of person A and person B. Also, we won't be able to simply use our array to represent visited positions any more, we will have an auxiliary data structure to do that. Also, we are allowed to make diagonal movements in this problem, so we now have 9 choices, we can move in one of 8 directions or simply stay in the same position. Another little trick that we have to watch for is that the players

can not just swap positions in a single turn, so we have to do a little bit of validity checking on the resulting state.

First, we set up the node structure and visited array:

```
class node {
 int player1X, player1Y, player2X, player2Y;
 int steps; // The current number of steps we have taken to reach this step
}

bool visited[20][20][20][20];
```

Here a node is represented as the (x,y) positions of player 1 and player 2. It also has the current steps that we have taken to reach the current state, we need this because the problem asks us what the minimum number of steps to switch the two players will be. We are guaranteed by the properties of the Breadth First search that the first time we visit the end node, it will be as quickly as possible (as all of our edge costs are 1).

The visited array is simply a direct representation of our node in array form, with the first dimension being player1X, second player1Y, etc. Note that we don't need to keep track of steps in the visited array.

Now that we have our basic structure set up, we can solve the problem (note that this code is not compilable):

```
int minTurns(String[] board) {
 int width = board[0].length;
 int height = board.length;

 node start;
 // Find the initial position of A and B, and save them in start.

 queue q;
 q.push(start);
 while (q.empty() == false) {
  node top = q.front();
  q.pop();

  // Check if player 1 or player 2 is out of bounds, or on an X square, if so continue
  // Check if player 1 or player 2 is on top of each other, if so continue

  // Make sure we haven't already visited this state before
  if (visited[top.player1X][top.player1Y][top.player2X][top.player2Y]) continue;
  // Mark this state as visited
  visited[top.player1X][top.player1Y][top.player2X][top.player2Y] = true;

  // Check if the current positions of A and B are the opposite of what they were in start.
  // If they are we have exchanged positions and are finished!
  if (top.player1X == start.player2X && top.player1Y == start.player2Y &&
      top.player2X == start.player1X && top.player2Y == start.player1Y)
      return top.steps;

  // Now we need to generate all of the transitions between nodes, we can do this quite easily
using some
  // nested for loops, one for each direction that it is possible for one player to move.  Since
we need
  // to generate the following deltas: (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1),
(1,0), (1,1)
  // we can use a for loop from -1 to 1 to do exactly that.
  for (int player1XDelta = -1; player1XDelta <= -1; player1XDelta++) {
   for (int player1YDelta = -1; player1YDelta <= -1; player1YDelta++) {
    for (int player2XDelta = -1; player2XDelta <= -1; player2XDelta++) {
     for (int player2YDelta = -1; player2YDelta <= -1; player2YDelta++) {
     // Careful though!  We have to make sure that player 1 and 2 did not swap positions on this
turn
      if (top.player1X == top.player2X + player2XDelta && top.player1Y == top.player2Y +
player2YDelta &&
          top.player2X == top.player1X + player1XDelta && top.player2Y == top.player1Y +
player1YDelta)
        continue;

     // Add the new node into the queue
      q.push(node(top.player1X + player1XDelta, top.player1Y + player1YDelta,
              top.player2X + player2XDelta, top.player2Y + player2YDelta,
              top.steps + 1));
```

```
      }
     }
    }
   }
  }

  // It is not possible to exchange positions, so
  // we return -1.  This is because we have explored
  // all the states possible from the starting state,
  // and haven't returned an answer yet.
  return -1;
}
```

This ended up being quite a bit more complicated than the basic Breadth First search implementation, but you can still see all of the basic elements in the code. Now, if you want to practice more problems where Breadth First search is applicable, try these:

Inviational 02 Semifinal Room 2 - Div 1 500 - Escape

Finding the best path through a graph

An extremely common problem on TopCoder is to find the shortest path from one position to another. There are a few different ways for going about this, each of which has different uses. We will be discussing two different methods, Dijkstra using a Heap and the Floyd Warshall method.

Dijkstra (Heap method)

Dijkstra using a Heap is one of the most powerful techniques to add to your TopCoder arsenal. It essentially allows you to write a Breadth First search, and instead of using a Queue you use a Priority Queue and define a sorting function on the nodes such that the node with the lowest cost is at the top of the Priority Queue. This allows us to find the best path through a graph in O(m * log(n)) time where n is the number of vertices and m is the number of edges in the graph.

Sidenote:

If you haven't seen big-O notation before then I recommend reading this.

First however, an introduction to the Priority Queue/Heap structure is in order. The Heap is a fundamental data structure and is extremely useful for a variety of tasks. The property we are most interested in though is that it is a semi-ordered data structure. What I mean by semi-ordered is that we define some ordering on elements that are inserted into the structure, then the structure keeps the smallest (or largest) element at the top. The Heap has the very nice property that inserting an element or removing the top element takes O(log n) time, where n is the number of elements in the heap. Simply getting the top value is an O(1) operation as well, so the Heap is perfectly suited for our needs.

The fundamental operations on a Heap are:

1. Add - Inserts an element into the heap, putting the element into the correct ordered location.
2. Pop - Pops the top element from the heap, the top element will either be the highest or lowest element, depending on implementation.
3. Top - Returns the top element on the heap.
4. Empty - Tests if the heap is empty or not.

Pretty close to the Queue or Stack, so it's only natural that we apply the same type of searching principle that we have used before, except substitute the Heap in place of the Queue or Stack. Our basic search routine (remember this one well!) will look something like this:

```
void dijkstra(node start) {
 priorityQueue s;
 s.add(start);
 while (s.empty() == false) {
  top = s.top();
  s.pop();
  mark top as visited;
check for termination condition (have we reached the target node?)
add all of top's unvisited neighbors to the stack.
 }
}
```

Unfortunately, not all of the default language libraries used in TopCoder have an easy to use priority queue structure.

C++ users are lucky to have an actual priority_queue<> structure in the STL, which is used as follows:

#include
using namespace std;
priority_queue pq;
1. Add - void pq.push(type)
2. Pop - void pq.pop()
3. Top - type pq.top()
4. Empty - bool pq.empty()

However, you have to be careful as the C++ priority_queue<> returns the *highest* element first, not the lowest. This has been the cause of many solutions that should be O(m * log(n)) instead ballooning in complexity, or just not working.

To define the ordering on a type, there are a few different methods. The way I find most useful is the following though:

```
Define your structure:
struct node {
 int cost;
 int at;
};
```
And we want to order by cost, so we define the less than operator for this structure as follows:
```
bool operator<(const node &leftNode, const node &rightNode) {
 if (leftNode.cost != rightNode.cost) return leftNode.cost < rightNode.cost;
 if (leftNode.at != rightNode.at) return leftNode.at < rightNode.at;
 return false;
}
```
Even though we don't need to order by the 'at' member of the structure, we still do otherwise elements with the same cost but different 'at' values may be coalesced into one value. The return false at the end is to ensure that if two duplicate elements are compared the less than operator will return false.

Java users unfortunately have to do a bit of makeshift work, as there is not a direct implementation of the Heap structure. We can approximate it with the TreeSet structure which will do full ordering of our dataset. It is less space efficient, but will serve our purposes fine.
```
import java.util.*;
TreeSet pq = new TreeSet();
```

1. Add - boolean add(Object o)
2. Pop - boolean remove(Object o)

In this case, we can remove anything we want, but pop should remove the first element, so we will always call it like this: pq.remove(pq.first());
3. Top - Object first()
4. Empty - int size()

To define the ordering we do something quite similar to what we use in C++:
```
class Node implements Comparable {
 public int cost, at;

 public int CompareTo(Object o) {
   Node right = (Node)o;
   if (cost < right.cost) return -1;
   if (cost > right.cost) return 1;
   if (at < right.at) return -1;
   if (at > right.at) return 1;
   return 0;
  }
}
```
C# users also have the same problem, so they need to approximate as well, unfortunately the closest thing to what we want that is currently available is the SortedList class, and it does not have the necessary speed (insertions and deletions are O(n) instead of O(log n)). Unfortunately there is no suitable built-in class for implementing heap based algorithms in C#, as the HashTable is not suitable either.

Getting back to the actual algorithm now, the beautiful part is that it applies as well to graphs with weighted edges as the Breadth First search does to graphs with un-weighted edges. So we can now solve much more difficult problems (and more common on TopCoder) than is possible with just the Breadth First search.

There are some extremely nice properties as well, since we are picking the node with the least total cost so far to explore first, the first time we visit a node is the best path to that node (unless there are negative weight edges in the graph). So we only have to visit each node once, and the really nice part is if we ever hit the target node, we know that we are done.

For the example here we will be using KiloManX, from SRM 181, the Div 1 1000. This is an excellent example of the application of the Heap Dijkstra problem to what appears to be a Dynamic Programming question initially. In this problem the edge weight between nodes changes based on what weapons we have picked up. So in our node we at least need to keep track of what weapons we have picked up, and the current amount of shots we have taken (which will be our cost). The really nice part is that the weapons that we have picked up corresponds to the bosses that we have defeated as well, so we can use that as a basis for our visited structure. If we represent each weapon as a bit in an integer, we will have to store a maximum of 32,768 values (2^15, as there is a maximum of 15 weapons). So we can make our visited array simply be an array of 32,768 booleans. Defining the ordering for our nodes is very easy in this case, we want to explore nodes that have lower amounts of shots taken first, so given this information we can define our basic structure to be as follows:
```
boolean visited[32768];

class node {
 int weapons;
 int shots;
 // Define a comparator that puts nodes with less shots on top appropriate to your language
```

```
};
Now we will apply the familiar structure to solve these types of problems.
int leastShots(String[] damageChart, int[] bossHealth) {
 priorityQueue pq;

 pq.push(node(0, 0));

 while (pq.empty() == false) {
  node top = pq.top();
  pq.pop();

  // Make sure we don't visit the same configuration twice
  if (visited[top.weapons]) continue;
  visited[top.weapons] = true;

  // A quick trick to check if we have all the weapons, meaning we defeated all the bosses.
  // We use the fact that (2^numWeapons - 1) will have all the numWeapons bits set to 1.
  if (top.weapons == (1 << numWeapons) - 1)
   return top.shots;

  for (int i = 0; i < damageChart.length; i++) {
   // Check if we've already visited this boss, then don't bother trying him again
   if ((top.weapons >> i) & 1) continue;

   // Now figure out what the best amount of time that we can destroy this boss is, given the
weapons we have.
   // We initialize this value to the boss's health, as that is our default (with our
KiloBuster).
   int best = bossHealth[i];
   for (int j = 0; j < damageChart.length; j++) {
    if (i == j) continue;
    if (((top.weapons >> j) & 1) && damageChart[j][i] != '0') {
     // We have this weapon, so try using it to defeat this boss
     int shotsNeeded = bossHealth[i] / (damageChart[j][i] - '0');
     if (bossHealth[i] % (damageChart[j][i] - '0') != 0) shotsNeeded++;
     best = min(best, shotsNeeded);
    }
   }

   // Add the new node to be searched, showing that we defeated boss i, and we used 'best' shots
to defeat him.
   pq.add(node(top.weapons | (1 << i), top.shots + best));
  }
 }
}
```

There are a huge number of these types of problems on TopCoder; here are some excellent ones to try out:
SRM 150 - Div 1 1000 - RoboCourier
SRM 194 - Div 1 1000 - IslandFerries
SRM 198 - Div 1 500 - DungeonEscape
TCCC '04 Round 4 - 500 - Bombman

Floyd-Warshall

Floyd-Warshall is a very powerful technique when the graph is represented by an adjacency matrix. It runs in O(n^3) time, where n is the number of vertices in the graph. However, in comparison to Dijkstra, which only gives us the shortest path from one source to the targets, Floyd-Warshall gives us the shortest paths from all source to all target nodes. There are other uses for Floyd-Warshall as well; it can be used to find connectivity in a graph (known as the Transitive Closure of a graph).

First, however we will discuss the Floyd Warshall All-Pairs Shortest Path algorithm, which is the most similar to Dijkstra. After running the algorithm on the adjacency matrix the element at adj[i][j] represents the length of the shortest path from node i to node j. The pseudo-code for the algorithm is given below:

```
for (k = 1 to n)
 for (i = 1 to n)
  for (j = 1 to n)
   adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
```

As you can see, this is extremely simple to remember and type. If the graph is small (less than 100 nodes) then this technique can be used to great effect for a quick submission.

An excellent problem to test this out on is the Division 2 1000 from SRM 184, TeamBuilder.

John Smith is in trouble! He is a TopCoder member and once he learned to master the "Force" of dynamic programming, he began solving problem after problem. But his once obedient computer acts quite unfriendly today. Following his usual morning ritual, John woke up at 10 AM, had a cup of coffee and went to solve a problem before breakfast. Something didn't seem right from the beginning, but based on his vast newly acquired experience, he wrote the algorithm in a flash. Tired of allocating matrices morning after morning, the computer complained: **"Segmentation fault!"**. Despite his empty stomach, John has a brilliant idea and gets rid of his beloved matrix by adding an extra "for cycle". But the computer cries again: **"Time limit exceeded!"**

Instead of going nuts, John makes a radical decision. Enough programming, he says! He decides to take a vacation as a reward for his hard work.

Being a very energetic guy, John wants to have the time of his life! With so many things to do, it is unfortunately impossible for him to enjoy them all. So, as soon as he eats his breakfast, he devises a "Fun Plan" in which he describes a schedule of his upcoming activities:

ID	Scheduled Activity	Time Span
1	Debug the room	Monday, 10:00 PM - Tuesday, 1:00 AM
2	Enjoy a trip to Hawaii	Tuesday, 6:00 AM - Saturday, 10:00 PM
3	Win the Chess Championship	Tuesday, 11:00 AM - Tuesday, 9:00 PM
4	Attend the Rock Concert	Tuesday, 7:00 PM - Tuesday, 11:00 PM
5	Win the Starcraft Tournament	Wednesday, 3:00 PM - Thursday, 3:00 PM
6	Have some paintball fun	Thursday, 10:00 AM - Thursday, 4:00 PM
7	Participate in the TopCoder Single Round Match	Saturday, 12:00 PM - Saturday, 2:00 PM
8	Take a shower	Saturday, 8:30 PM - Saturday 8:45 PM
9	Organize a Slumber Party	Saturday, 9:00 PM - Sunday, 6:00 AM
10	Participate in an "All you can eat" and "All you can drink" contest	Saturday, 9:01 PM - Saturday, 11:59 PM

He now wishes to take advantage of as many as he can. Such careful planning requires some cleverness, but his mind has gone on vacation too. This is John Smith's problem and he needs our help.

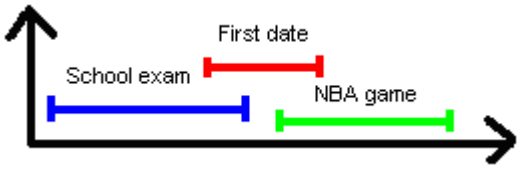
Could we help him have a nice holiday? Maybe we can! But let's make an assumption first. As John is a meticulous programmer, once he agrees on something, he sticks to the plan. So, individual activities may either be chosen or not. For each of the two choices regarding the first activity, we can make another two choices regarding the second. After a short analysis, we find out that we have 2 ^ N possible choices, in our case 1024. Then, we can check each one individually to see whether it abides the time restrictions or not. From these, finding the choice with the most activities selected should be trivial. There are quite a lot of alternatives, so John would need to enlist the help of his tired computer. But what happens if we have 50 activities? Even with the most powerful computer in the world, handling this situation would literally take years. So, this approach is clearly not feasible.

Let's simply the problem and trust our basic instinct for a moment. A good approach may be to take the chance as the first opportunity arises. That is, if we have two activities we can follow and they clash, we choose the one that starts earlier in order to save some time. In this case John will start his first evening by debugging his room. Early the next morning, he has a plane to catch. It is less than a day, and he has already started the second activity. This is great! Actually, **the best choice** for now. But what happens next? Spending 5 days in Hawaii is time consuming and by Saturday evening, he will still have only two activities performed. Think of all the activities he could have done during this five day span! Although very fast and simple, this approach is unfortunately not accurate.

We still don't want to check for every possible solution, so let's try another trick. Committing to such a time intensive activity like the exotic trip to Hawaii can simply be avoided by selecting first the activity which takes the least amount of time and then continuing this process for the remaining activities that are compatible with those already selected. According to the previous schedule, first of all we choose the shower. With only 15 minutes consumed, this is by far the **best local choice**. What we would like to know is whether we can still keep this **"local best"** as the other compatible activities are being selected. John's schedule will look like this:

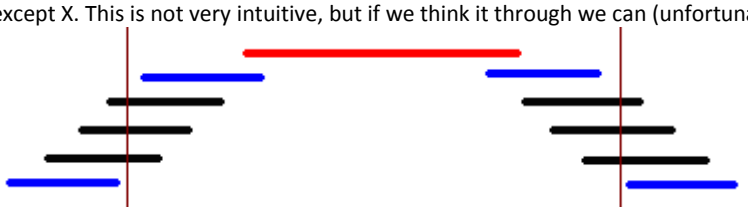
- Take a shower (15 minutes)
- Participate in the TopCoder Single Round Match (2 hours)
- Participate in an "All you can eat" and "All you can drink" contest (2 hours 58 minutes)
- Debug the room (3 hours)
- Attend the Rock Concert (4 hours)
- Have some paintball fun (6 hours)

Out of the 10 possible activities, we were able to select 6 (which is not so bad). We now run the slow but trustworthy algorithm to see if this is actually the best choice we can make. And the answer is indeed 6. John is very appreciative for our help, but once he returns from the holiday, confident in our ingenious approach, he may face a serious problem:



By going for the short date, he misses both the school exam and the match of his favorite team. Being the TopCoders that we are, we must get used to writing reliable programs. A single case which we cannot handle dooms this approach to failure.

What we generally have to do in situations like this is to analyze what might have caused the error in the first place and act accordingly to avoid it in the future. Let's look again at the previous scenario. The dating activity clashes with both the exam and the match, while the other two only clash with the date. So, the idea almost comes from itself. Why not always select the activity that produces the minimum amount of clashes with the remaining activities? Seems logical - it all makes sense now! We'll try to prove that this approach is indeed correct. Suppose we have already selected an activity X and try to check if we could have selected two activities A and B that clash with X instead. A and B should of course not clash, otherwise the final result will not improve. But now, we are back to the previous case (X has two clashes, while A and B have only one). If this is the case, A and B are selected from the beginning. The only way to disprove our assumption is to make A and B clash more, without affecting other activities except X. This is not very intuitive, but if we think it through we can (unfortunately) build such a case:



The activities represented by the blue lines are the optimal choice given the above schedule. But as the activity in red produces only 2 clashes, it will be chosen first. There are 4 compatible activities left before, but they all clash with each other, so we can only select one. The same happens for the activities scheduled after, leaving space for only one more choice. This only gives us 3 activities, while the optimum choice selects 4.

So far, every solution we came up with had a hidden flaw. It seems we have to deal with a devilish problem. Actually, this problem has quite an elegant and straightforward solution. If we study the figure above more carefully, we see that the blue activity on the bottom-left is the only one which finishes before the "timeline" indicated by the thin vertical bar. So, if we are to choose a single activity, choosing the one that ends first (at a time **t1**), will leave all the remaining time interval free for choosing other activities. If we choose any other activity instead, the remaining time interval will be shorter. This is obvious, because we will end up anyway with only one activity chosen, but at a time **t2 > t1**. In the first case we had available all the time span between **t1** and **finish** and that **included** the time between t2 and finish. Consequently, there is no disadvantage in choosing the activity that finishes earlier. The advantage may result in the situation when we are able to insert another activity that starts between **t1** and **t2** and ends up before the end of any activity that starts after time **t2**.

Known as the **"Activity Selection"**, this is a standard problem that can be solved by the **Greedy Method**. As a greedy man takes as much as he can as often as he can, in our case we are choosing at every step the activity that finishes first and do so every time there is no activity in progress. The truth is we all make greedy decisions at some point in our life. When we go shopping or when we drive a car, we make choices that seem best for the moment. Actually, there are two basic ingredients every greedy algorithm has in common:

- **Greedy Choice Property**: from a local optimum we can reach a global optimum, without having to reconsider the decisions already taken.
- **Optimal Substructure Property**: the optimal solution to a problem can be determined from the optimal solutions to its subproblems.

The following pseudo code describes the optimal activity selection given by the "greedy" algorithm proven earlier:
Let N denote the number of activities and
{I} the activity I (1 <= I <= N)
For each {I}, consider S[I] and F[I] its starting and finishing time
Sort the activities in the increasing order of their finishing time
- that is, for every I < J we must have F [I] <= F [J]

```
//  A denotes the set of the activities that will be selected
A = {1}
//  J denotes the last activity selected
J = 1
For I = 2  to N
// we can select activity 'I' only if the last activity
// selected has already been finished
      If S [I] >= F [J]
//  select activity 'I'
                A = A + {I}
// Activity 'I' now becomes the last activity selected
                J = I
      Endif
Endfor
Return A
```

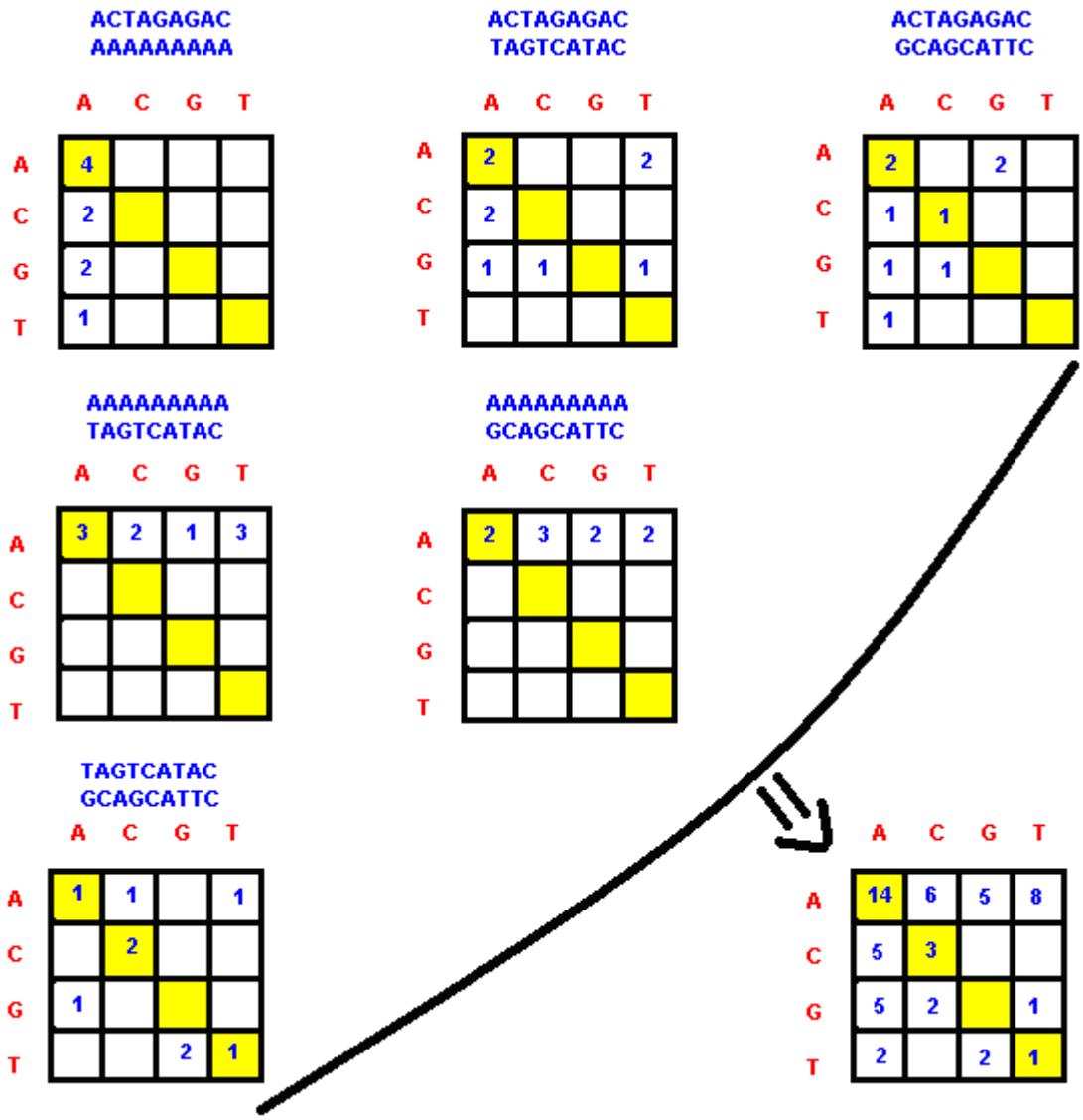
After applying the above algorithm, Johnny's "Fun Plan" would look like this:

- Eliminate all the bugs and take some time to rest
- Tuesday is for chess, prepare to beat them all
- A whole day of Starcraft follows, this should be fun
- The next two days are for recovery
- As for the final day, get a few rating points on TopCoder, take a shower and enjoy the versatile food and the good quality wine

The problem of John Smith is solved, but this is just one example of what Greedy can do. A few examples of real TopCoder problems will help you understand the concept better. But before moving on, you may wish to practice a little bit more what you have read so far on a problem similar with the Activity Selection, named Boxing.

BioScore
In this problem you are asked to maximize the average homology score for all the pairs in the set. As an optimal solution is required, this may be a valuable clue in determining the appropriate method we can use. Usually, this kind of problems can be solved by dynamic programming, but in many cases a Greedy strategy could also be employed.

The first thing we have to do here is to **build the frequency matrix**. This is an easy task as you just have to compare every pair of two sequences and count the occurrences of all the combinations of nucleic acids (AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, TT). Each of these combinations will be an element in the matrix and its value will represent the total number of occurrences. For example, let's take the set { "ACTAGAGAC", "AAAAAAAAA", "TAGTCATAC", "GCAGCATTC" } used in Example 2.



In the bottom-right part of the figure above, you can see the resulting frequency matrix. Let us denote it by **F** What we have to do from now is to find another matrix **S** such that the sum of the 16 corresponding products of the type **F[I,J] * S[I,J]** (1 <= I,J <= 4) is maximized.

Now, let's look at the matrix restrictions and analyze them one by one:

1) The sum of the 16 entries must be 0.

This is more like a commonsense condition. With all the elements in **F** positive, the final score tends to increase as we increase the elements in S. But because the sum must be kept at 0, in order to increase an element, we'll have to decrease others. The challenge of this problem resides in finding the optimal distribution.

2) All entries must be integers between -10 and 10 inclusive

Another commonsense condition! Our search space has been drastically reduced, but we are still left with a lot of alternatives.

3) It must be symmetric (score(x,y) = score(y,x))

Because of the symmetry, we must attribute the same homology score to combinations like "AC" and "CA". As a result, we can also count their occurrences together. For the previous example, we have the set of combinations with the following frequencies:

AA: 14	CC: 3	GG: 0	TT: 1
AC + CA: 11	AG + GA: 10	AT + TA: 10	

CG + GC: 2 CT + TC: 0

GT + TG: 3

An intuitive approach would be to assign a higher homology score to the combinations that appear more often. But as we must keep the score sum to 0, another problem arises. Combinations like AA, CC, GG and TT appear only once in the matrix. So, their homology score contribute less to the total sum.

4) Diagonal entries must be positive (score(x,x)>0)

This restriction differentiates the elements on the diagonal from the others even further. Basically, we have two groups: the four elements on the diagonal (which correspond to the combinations AA, CC, GG and TT) and the six elements not on the diagonal (which correspond to the combinations AC + CA, AG + GA, AT + TA, CG + GC, CT + TC and GT +TG). Each of these groups can have different states, depending on the value we assign to their elements.

To make things easier, **for each possible state in the first group we wish to find an optimal state for the second group**. As all the elements in the second group have the same property, we will try to find their optimal state by using a **Greedy approach**. But because the elements in the first group can take any values between 1 and 10, the sum we wish to obtain for the scores we choose in the second group has to be recalculated. It's easy to notice that the sum of the elements in the first group can range anywhere between 4 and 40. As a result, depending on the choice we make for the first group, we'll have to obtain a sum between -2 and -20 for the second (we shall not forget that the symmetrical elements in the matrix have been coupled together, thus they count twice in the score matrix).

Now, we have finally reached to the **problem core**. The solution to the entire problem depends on finding the optimal choice for the scores in the second group. If the problem has indeed the **greedy choice property** and the **optimal substructure property**, we'll be able to pick one element form the group, assign it the best scenario and proceed with the remaining elements in the same manner.

Claim: If we always give the highest possible score to the combination that has the most occurrences in the group, we'll obtain in the end the highest possible score for the entire group.

The first thing we have to do is to sort these six elements in matrix F. Then, we have to actually compute the corresponding score values in S. As the total score we should obtain is at least -20, one quick insight tells us that the first two elements could be given a score of 10 (if we assign -10 to all the remaining four elements, -20 can still be achieved). We know as well that the final score is less than 0. Because we want to maximize the scores for the first elements, the last three elements can only be -10 (in the best case the score sum of the elements is -2 and then, we assign scores in the following manner: [10, 10, 8, -10, -10, -10]). Finally, the value of the third element will depend on the choices we make for the first group. From the maximum of 10, we subtract half of the score sum of the elements in the first group (we should note here that the aforementioned sum must be even).

Now, we have to make sure that our approach is indeed correct. The proof is quite straightforward, as in order keep the sum in **S** constant we can only decrease from the score of a combination with more occurrences and increase to the score of a combination with fewer occurrences. Let **f1** and **f2** be the frequencies of the two combinations and **f1 >= f2**. We have f1 * s1 + f2 * s2 = X, where **X** is the sum we should maximize. By our **greedy assumption, s1 >= s2**. As s1 + s2 remains constant, the previous sum changes to: f1*(s1 - a) + f2*(s2 + a) = Y, where a is strictly greater than 0. We find out that Y - X = a * (f2 - f1). Because f1 >= f2, this difference will always be less than or equal to 0. It results that Y <= X. As Y was chosen arbitrarily, it can be concluded that the initial greedy choice always gives the maximum possible score.

We apply the algorithm described above for each state of the elements in the first group and save the best result.

Representation: Instead of using the matrices **F** and **S**, **we find it more convenient to use arrays for storing both the combination frequencies and their corresponding score**. The first 4 elements of **F** will denote the frequency of the combinations AA, CC, GG and TT. The next 6 elements will denote the other possible combinations and are sorted in the decreasing order of their frequency (F[5] >= F[6] >= F[7] >= F[8] >= F[9] >= F[10]). **S** will be an array of 10 elements such that S[I] is the score we attribute to the combination I.

The main algorithm is illustrated in the following pseudo code:

```
Best = -Infinity
For S [1] = 1 to 10
      For S [2] = 1 to 10
            For S [3] = 1 to 10
                  For S [4] = 1 to 10
                        If  (S [1] + S [2] + S [3] + S [4]) mod 2 = 0
                             S [5] = S[6] = 10
                             S [7] = 10 - (S [1] + S [2] + S [3] + S[4]) / 2
                             S [8] = S [9] = S [10] = -10
//  in Best we save the greatest average homology score
                             Best = max (Best , score (F,S))
```

```
//  obtained so far.
                        Endif
                    Endfor
                Endfor
            Endfor
        Endfor
Endfor
Return Best
```

Given the score matrix (in our case the array **S**), we compute the final result by just making the sum of the products of the form F[I] * S[I] (1 <= I <=10) and divide it by N * (N-1) / 2 in order to obtain the average homology score.

GoldMine

We are now going to see how a gold mine can be exploited to its fullest, by being greedy. Whenever we notice the maximum profit is involved, a greedy switch should activate. In this case, we must allocate all the miners to the available mines, such that the total profit is maximized. After a short analysis, we realize that we want to know how much money can be earned from a mine in all the possible cases. And there are not so many cases, as in each mine we can only have between 0 and 6 workers. The table below represents the possible earnings for the two mines described in the example 0 of the problem statement:

	0 workers	1 worker	2 workers	3 workers	4 workers	5 workers	6 workers
First mine	0	57	87	87	67	47	27
Second mine	0	52	66	75	75	66	48

As we are going to assign workers to different mines, we may be interested in the profit a certain worker can bring to the mine he was assigned. This can be easily determined, as we compute the difference between the earnings resulted from a mine with the worker and without. If we only had one worker, the **optimal choice** would have been to allocate him in the mine where he can bring the best profit. But as we have more workers, we want to check if assigning them in the same manner would bring the **best global profit**.

In our example we have 4 workers that must be assigned. The table below shows the profit obtained in the two mines for each additional worker.

	Initially	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	Worker 6
First mine	-	57	30	0	-20	-20	-20
Second mine	-	52	14	9	0	-9	-20

We notice that the first mine increases its profit by 57 if we add a worker, while the second by only 52. So, we allocate the first worker to the first mine.

	Initially	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	Worker 6
First mine	-	57	30	0	-20	-20	-20
Second mine	-	52	14	9	0	-9	-20

Now, an additional worker assigned to the first mine would only increase its profit by 30. We put him in the second, where the profit can be increased by 52.

	Initially	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	Worker 6
First mine	-	57	30	0	-20	-20	-20
Second mine	-	52	14	9	0	-9	-20

The third miner would be more useful to the first mine as he can bring a profit of 30.

	Initially	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	Worker 6
First mine	-	57	30	0	-20	-20	-20
Second mine	-	52	14	9	0	-9	-20

As for the last miner, we can either place him in the first mine (for a zero profit) or in the second (for a profit of 14). Obviously, we assign him to the second.

	Initially	Worker 1	Worker 2	Worker 3	Worker 4	Worker 5	Worker 6
First mine	-	57	30	0	-20	-20	-20
Second mine	-	52	14	9	0	-9	-20

In the end two of the workers have been allocated to the first mine and another two to the second. The example shows us that this is indeed the choice with the best total profit. But will our "greedy" approach always work?

Claim: We obtain the maximum total profit when we assign the workers one by one to the mine where they can bring the best immediate profit.

Proof: Let A and B be two mines and a1, a2, b1, b2 be defined as below:
a1 - the profit obtained when an additional worker is assigned to mine A
a1 + a2 - the profit obtained when two additional workers are assigned to mine A
b1 - the profit obtained when an additional worker is assigned to mine B
b1 + b2 - the profit obtained when two additional workers are assigned to mine B
Let us now consider that we have two workers to assign and a1 >= b1.

Our greedy algorithm will increase the profit by a1 for the first worker and by max (a2, b1) for the second worker. The total profit in this case is **a1+max(a2,b1)**. If we were to choose the profit b1 for the first worker instead, the alternatives for the second worker would be a profit of a1 or a profit of b2.

In the first case, the total profit would be b1+a1 <= a1 + max (a2,b1).
In the second case, the total profit would be b1+b2. We need to prove that b1+b2 <= a1+max(a2,b1). But b2 <= b1 as **the profit of allocating an extra worker to a mine is always higher or equal with the profit of allocating the next extra worker to that mine**.

Gold Mine Status	Profit from extra-worker 1	Profit from extra-worker 2
number of ores > number of workers + 2	60	60
number of ores = number of workers + 2	60	50
number of ores = number of workers + 1	50	-20
number of ores < number of workers + 1	-20	-20

As b1+b2 <= a1+b2 <= a1+b1 <= a1+max(a2,b1), the greedy choice is indeed the best .

Coding this is not difficult, but one has to take into account the problem constraints (all miners must be placed, there are at most six workers in a mine and if a worker can be optimally assigned to more than one mine, put him in the mine with the lowest index).

WorldPeace
The greedy algorithms we have seen so far work well in every possible situation as their correction has been proven. But there is another class of optimization problems where Greedy Algorithms have found their applicability. This category mostly includes NP-complete problems (like the Traveling Salesman Problem) and here, one may prefer to write an heuristic based on a greedy algorithm than to wait ... The solution is not always the best, but for most real purposes, it is good enough. While this problem is not NP, it is an excellent example of how a simple greedy algorithm can be adapted to fool not only the examples, but also the carefully designed system tests. Such an algorithm is not very hard to come with and after a short analysis we notice that in order to maximize the total number of groups **it is always optimal to form a group from**

the k countries that have the highest number of citizens. We apply this principle at every single step and then sort the sequence again to see which are the next k countries having the highest number of citizens. This idea is illustrated in the following pseudo code:

```
Groups = 0
Repeat
// sorts the array in decreasing order
        Sort (A)
        Min= A[K]
        If Min > 0 Groups = Groups + 1
         For I = 1 to K
                A[I] = A[I] - 1
         Endfor
Until  Min = 0
Return Groups
```

Unfortunately, a country can have up to a billion citizens, so we cannot afford to make only one group at a time. Theoretically, for a given set of k countries, we can make groups until all the citizens in one of these countries have been grouped. And this can be done in a single step:

```
Groups = 0
Repeat
// sorts the array in decreasing order
        Sort (A)
        Min= A[K]
        Groups = Groups + Min
         For I = 1 to K
                A[I] = A[I] - Min
         Endfor
Until  Min = 0
Return Groups
```

The execution time is no longer a problem, but it is the algorithm! As we check it on the example 0, our method returns 4 instead of 5. The result returned for the examples 1, 2 and 3 is correct. As for the last example, instead of making 3983180234 groups, we are able to make 3983180207. Taking into account the small difference, we may say that our solution is **pretty good**, so maybe we can refine it more on this direction.

So far, we have two algorithms:

- a first greedy algorithm that is accurate, but not fast enough
- a second greedy algorithm that is fast, but not very accurate.

What we want to do is to optimize accuracy as much as we can, without exceeding the execution time limit. Basically, we are looking for **a truce between speed and accuracy**. The only difference in the two algorithms described above is the number of groups we select at a given time. The compromise we will make is to select an arbitrarily large number of groups in the beginning, and as we approach the end to start being more cautious. When we are left with just a few ungrouped citizens in every country, it makes complete sense to use the safe brute force approach. In the variable **Allowance** defined in the algorithm below, we control the number of groups we want to make at a given moment.

```
Groups = 0
Repeat
// sorts the array in decreasing order
        Sort (A)
        Min= A[K]
        Allowance = (Min+999) / 1000
        Groups = Groups + Allowance
         For I = 1 to K
                A[I] = A[I] - Allowance
         Endfor
Until  Min = 0
Return Groups
```

If this approach is correct indeed, remains to be seen. Despite the fact it escaped both Tomek's keen eyes and system tests, it is very likely that the result is not optimal for all the set of possible test cases. This was just an example to show that a carefully chosen refinement on a simple (but obvious faulty) greedy approach can actually be the "right" way. For more accurate solutions to this problem, see the Match Editorial.

Conclusion

Greedy algorithms are usually easy to think of, easy to implement and run fast. Proving their correctness may require rigorous mathematical proofs and is sometimes insidious hard. In addition, greedy algorithms are infamous for being tricky. Missing even a very small detail can be fatal. But when you have nothing else at your disposal, they may be the only salvation. With backtracking or dynamic programming you are on a relatively safe ground. With greedy instead, it is more like walking on a mined field. Everything looks fine on the surface, but the hidden part may backfire on you when you least expect. While there are some standardized problems, most of the problems solvable by this method call for heuristics. There is no general template on how to apply the greedy method to a given problem, however the problem specification might give you a good insight. Advanced mathematical concepts such as matroids may give you a recipe for proving that a class of problems can be solved with greedy, but it ultimately comes down to the keen sense and experience of the programmer. In some cases there are a lot of greedy

assumptions one can make, but only few of them are correct (see the Activity Selection Problem). In other cases, a hard problem may hide an ingenious greedy shortcut, like there was the case in the last problem discussed, WorldPeace. And this is actually the whole beauty of greedy algorithms! Needless to say, they can provide excellent challenge opportunities...

A few final notes

- a problem that seems extremely complicated on the surface (see TCSocks) might signal a greedy approach.
- problems with a very large input size (such that a n^2 algorithm is not fast enough) are also more likely to be solved by greedy than by backtracking or dynamic programming.
- despite the rigor behind them, you should look to the greedy approaches through the eyes of a detective, not with the glasses of a mathematician.

- in addition, study some of the standard greedy algorithms to grasp the concept better (Fractional Knapsack Problem, Prim Algorithm, Kruskal Algorithm, Dijkstra Algorithm, Huffman Coding, Optimal Merging, Topological Sort).

Further Problems

Level 1

GroceryBagger - SRM 222
FanFailure - SRM 195
PlayGame - SRM 217
SchoolAssembly - TCO04 Round 2
RockStar - SRM 216
Apothecary - SRM 204
Boxing - TCO04 Round 3
Unblur - TCO04 Semifinal Room 3

Level 2

Crossroads - SRM 217
TCSocks - SRM 207
HeatDeath - TCO04 Round 4
BioScore - TCO04 Semifinal Room 1
Rationalization - SRM 224

Level 3

GoldMine - SRM 169
MLBRecord - TCO04 Round 2
RearrangeFurniture - SRM 220
WorldPeace - SRM 204

An important part of given problems can be solved with the help of dynamic programming (**DP** for short). Being able to tackle problems of this type would greatly increase your skill. I will try to help you in understanding how to solve problems using DP. The article is based on examples, because a raw theory is very hard to understand.

Note: If you're bored reading one section and you already know what's being discussed in it - skip it and go to the next one.

Introduction (Beginner)

What is a dynamic programming, how can it be described?

A **DP** is an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

Now let's see the base of DP with the help of an example:

Given a list of N coins, their values ($\mathbf{V_1}$, $\mathbf{V_2}$, ... , $\mathbf{V_N}$), and the total sum **S**. Find the minimum number of coins the sum of which is **S** (we can use as many coins of one type as we want), or report that it's not possible to select coins in such a way that they sum up to **S**.

Now let's start constructing a DP solution:

First of all we need to find a state for which an optimal solution is found and with the help of which we can find the optimal solution for the next state.

What does a "state" stand for?

It's a way to describe a situation, a sub-solution for the problem. For example a state would be the solution for sum **i**, where **i≤S**. A smaller state than state **i** would be the solution for any sum **j**, where **j<i**. For finding a **state i**, we need to first find all smaller states **j (j<i)** . Having found the minimum number of coins which sum up to **i**, we can easily find the next state - the solution for **i+1**.

How can we find it?

It is simple - for each coin **j,** $\mathbf{V_j \le i}$, look at the minimum number of coins found for the $\mathbf{i-V_j}$sum (we have already found it previously). Let this number be **m**. If **m+1** is less than the minimum number of coins already found for current sum **i**, then we write the new result for it.

For a better understanding let's take this example:
Given coins with values 1, 3, and 5.
And the sum **S** is set to be 11.

First of all we mark that for state 0 (sum 0) we have found a solution with a minimum number of 0 coins. We then go to sum 1. First, we mark that we haven't yet found a solution for this one (a value of Infinity would be fine). Then we see that only coin 1 is less than or equal to the current sum. Analyzing it, we see that for sum 1-$\mathbf{V_1}$= 0 we have a solution with 0 coins. Because we add one coin to this solution, we'll have a solution with 1 coin for sum 1. It's the only solution yet found for this sum. We write (save) it. Then we proceed to the next state - **sum 2**. We again see that the only coin which is less or equal to this sum is the first coin, having a value of 1. The optimal solution found for sum (2-1) = 1 is coin 1. This coin 1 plus the first coin will sum up to 2, and thus make a sum of 2 with the help of only 2 coins. This is the best and only solution for sum 2. Now we proceed to sum 3. We now have 2 coins which are to be analyzed - first and second one, having values of 1 and 3. Let's see the first one. There exists a solution for sum 2 (3 - 1) and therefore we can construct from it a solution for sum 3 by adding the first coin to it. Because the best solution for sum 2 that we found has 2 coins, the new solution for sum 3 will have 3 coins. Now let's take the second coin with value equal to 3. The sum for which this coin needs to be added to make 3 , is 0. We know that sum 0 is made up of 0 coins. Thus we can make a sum of 3 with only one coin - 3. We see that it's better than the previous found solution for sum 3 , which was composed of 3 coins. We update it and mark it as having only 1 coin. The same we do for sum 4, and get a solution of 2 coins - 1+3. And so on.

Pseudocode:

```
Set Min[i] equal to Infinity for all of i
Min[0]=0

For i = 1 to S
For j = 0 to N - 1
   If (Vj<=i AND Min[i-Vj]+1<Min[i])
Then Min[i]=Min[i-Vj]+1

Output Min[S]
```

Here are the solutions found for all sums:

Sum	Min. nr. of coins	Coin value added to a smaller sum to obtain this sum (it is displayed in brackets)
0	0	-
1	1	1 (0)
2	2	1 (1)
3	1	3 (0)
4	2	1 (3)
5	1	5 (0)
6	2	3 (3)
7	3	1 (6)
8	2	3 (5)
9	3	1 (8)
10	2	5 (5)
11	3	1 (10)

As a result we have found a solution of 3 coins which sum up to 11.

Additionally, by tracking data about how we got to a certain sum from a previous one, we can find what coins were used in building it. For example: to sum 11 we got by adding the coin with value 1 to a sum of 10. To sum 10 we got from 5. To 5 - from 0. This way we find the coins used: 1, 5 and 5.

Having understood the basic way a **DP** is used, we may now see a slightly different approach to it. It involves the change (update) of best solution yet found for a sum i, whenever a better solution for this sum was found. In this case the states aren't calculated consecutively. Let's consider the problem above. Start with having a solution of 0 coins for sum 0. Now let's try to add first coin (with value 1) to all sums already found. If the resulting sum **t** will be composed of fewer coins than the one previously found - we'll update the solution for it. Then we do the same thing for the second coin, third coin, and so on for the rest of them. For example, we first add coin 1 to sum 0 and get sum 1. Because we haven't yet found a possible way to make a sum of 1 - this is the best solution yet found, and we mark **S[1]=1**. By adding the same coin to sum 1, we'll get sum 2, thus making **S[2]=2**. And so on for the first coin. After the first coin is processed, take coin 2 (having a value of 3) and consecutively try to add it to each of the sums already found. Adding it to 0, a sum 3 made up of 1 coin will result. Till now, **S[3]** has been equal to 3, thus the new solution is better than the previously found one. We update it and mark **S[3]=1**. After adding the same coin to sum 1, we'll get a sum 4 composed of 2 coins. Previously we found a sum of 4 composed of 4 coins; having now found a better solution we update **S[4]** to 2. The same thing is done for next sums - each time a better solution is found, the results are updated.

Elementary

To this point, very simple examples have been discussed. Now let's see how to find a way for passing from one state to another, for harder problems. For that we will introduce a new term called recurrent relation, which makes a connection between a lower and a greater state.

Let's see how it works:

Given a sequence of N numbers - **A[1]** , **A[2]** , ..., **A[N]** . Find the length of the longest non-decreasing sequence.

As described above we must first find how to define a "state" which represents a sub-problem and thus we have to find a solution for it. Note that in most cases the states rely on lower states and are independent from greater states.

Let's define a state **i** as being the longest non-decreasing sequence which has its last number **A[i]** . This state carries only data about the length of this sequence. Note that for **i<j** the state **i** is independent from **j**, i.e. doesn't change when we calculate state **j**. Let's see now how these states are connected to each other. Having found the solutions for all states lower than **i**, we may now look for state **i**. At first we initialize it with a solution of 1, which consists only of the **i-th** number itself. Now for each **j<i** let's see if it's possible to pass from it to state i. This is possible only when **A[j]≤A[i]** , thus keeping (assuring) the sequence non-decreasing. So if **S[j]** (the solution found for state **j**) + **1** (number **A[i]** added to this sequence which ends with number **A[j]**) is better than a solution found for **i** (ie. **S[j]+1>S[i]**), we make **S[i]=S[j]+1**. This way we consecutively find the best solutions for each **i**, until last state N.

Let's see what happens for a randomly generated sequence: 5, 3, 4, 8, 6, 7:

I	The length of the longest non-decreasing sequence of first i numbers	The last sequence i from which we "arrived" to this one
1	1	1 (first number itself)
2	1	2 (second number itself)
3	2	2
4	3	3
5	3	3
6	4	4

Practice problem:
Given an undirected graph **G** having **N** (1<N<=1000) vertices and positive weights. Find the shortest path from vertex 1 to vertex N, or state that such path doesn't exist.

Hint: At each step, among the vertices which weren't yet checked and for which a path from vertex 1 was found, take the one which has the shortest path, from vertex 1 to it, yet found.

Try to solve the following problems from TopCoder competitions:

- ZigZag - 2003 TCCC Semifinals 3
- BadNeighbors - 2004 TCCC Round 4
- FlowerGarden - 2004 TCCC Round 1

Intermediate

Let's see now how to tackle bi-dimensional DP problems.

Problem:
A table composed of **N x M** cells, each having a certain quantity of apples, is given. You start from the upper-left corner. At each step you can go down or right one cell. Find the maximum number of apples you can collect.

This problem is solved in the same way as other DP problems; there is almost no difference.

First of all we have to find a state. The first thing that must be observed is that there are at most 2 ways we can come to a cell - from the left (if it's not situated on the first column) and from the top (if it's not situated on the most upper row). Thus to find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell.

From above, a recurrent relation can be easily obtained:
S[i][j]=A[i][j] + max(S[i-1][j], if i>0 ; S[i][j-1], if j>0) (where **i** represents the row and **j** the column of the table , its left-upper corner having coordinates {0,0} ; and **A[i][j]** being the number of apples situated in cell **i,j**).

S[i][j] must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.

Pseudocode:
```
For i = 0 to N - 1
   For j = 0 to M - 1
   S[i][j] = A[i][j] +
      max(S[i][j-1], if j>0 ; S[i-1][j], if i>0 ; 0)

Output S[n-1][m-1]
```
Here are a few problems, from TopCoder Competitions, for practicing:

- AvoidRoads - 2003 TCO Semifinals 4

- ChessMetric - 2003 TCCC Round 4

Upper-Intermediate

This section will discuss about dealing DP problems which have an additional condition besides the values that must be calculated.

As a good example would serve the following problem:

Given an undirected graph **G** having positive weights and **N** vertices.

You start with having a sum of **M** money. For passing through a vertex **i**, you must pay **S[i]** money. If you don't have enough money - you can't pass through that vertex. Find the shortest path from vertex 1 to vertex N, respecting the above conditions; or state that such path doesn't exist. If there exist more than one path having the same length, then output the cheapest one. Restrictions: 1<N<=100 ; 0<=M<=100 ; for each i, 0<=S[i]<=100. As we can see, this is the same as the classical Dijkstra problem (finding the shortest path between two vertices), with the exception that it has a condition. In the classical Dijkstra problem we would have used a uni-dimensional array **Min[i]** , which marks the length of the shortest path found to vertex **i**. However in this problem we should also keep information about the money we have. Thus it would be reasonable to extend the array to something like **Min[i][j]** , which represents the length of the shortest path found to vertex **i**, with **j** money being left. In this way the problem is reduced to the original path-finding algorithm. At each step we find the unmarked state (i,j) for which the shortest path was found. We mark it as visited (not to use it later), and for each of its neighbors we look if the shortest path to it may be improved. If so - then update it. We repeat this step until there will remain no unmarked state to which a path was found. The solution will be represented by **Min[N-1][j]** having the least value (and the greatest **j** possible among the states having the same value, i.e. the shortest paths to which has the same length).

Pseudocode:

```
Set states(i,j) as unvisited for all (i,j)
Set Min[i][j] to Infinity for all (i,j)

Min[0][M]=0

While(TRUE)

Among all unvisited states(i,j) find the one for which Min[i][j]
is the smallest. Let this state found be (k,l).

If there wasn't found any state (k,l) for which Min[k][l] is
less than Infinity - exit While loop.

Mark state(k,l) as visited

For All Neighbors p of Vertex k.
   If (l-S[p]>=0 AND
    Min[p][l-S[p]]>Min[k][l]+Dist[k][p])
      Then Min[p][l-S[p]]=Min[k][l]+Dist[k][p]
   i.e.
If for state(i,j) there are enough money left for
going to vertex p (l-S[p] represents the money that
will remain after passing to vertex p), and the
shortest path found for state(p,l-S[p]) is bigger
than [the shortest path found for
state(k,l)] + [distance from vertex k to vertex p)],
then set the shortest path for state(i,j) to be equal
to this sum.
End For

End While
```

Find the smallest number among Min[N-1][j] (for all j, 0<=j<=M);
if there are more than one such states, then take the one with greater
j. If there are no states(N-1,j) with value less than Infinity - then
such a path doesn't exist.
Here are a few TC problems for practicing:

- Jewelry - 2003 TCO Online Round 4
- StripePainter - SRM 150 Div 1
- QuickSums - SRM 197 Div 2

- ShortPalindromes - SRM 165 Div 2

Advanced

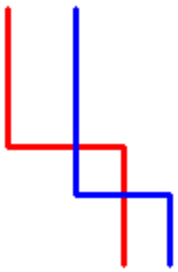
The following problems will need some good observations in order to reduce them to a dynamic solution.

Problem StarAdventure - SRM 208 Div 1:

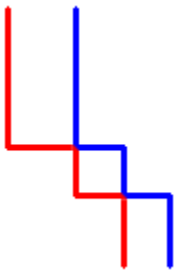
Given a matrix with **M** rows and **N** columns (**N x M**). In each cell there's a number of apples.
You start from the upper-left corner of the matrix. You can go down or right one cell. You need to arrive to the bottom-right corner. Then you need to go back to the upper-left cell by going each step one cell left or up. Having arrived at this upper-left cell, you need to go again back to the bottom-right cell.
Find the maximum number of apples you can collect.
When you pass through a cell - you collect all the apples left there.

Restrictions: 1 < **N**, **M** <= 50 ; each cell contains between 0 and 1000 apples inclusive.

First of all we observe that this problem resembles to the classical one (described in Section 3 of this article), in which you need to go only once from the top-left cell to the bottom-right one, collecting the maximum possible number of apples. It would be better to try to reduce the problem to this one. Take a good look into the statement of the problem - what can be reduced or modified in a certain way to make it possible to solve using DP? First observation is that we can consider the second path (going from bottom-right cell to the top-left cell) as a path which goes from top-left to bottom-right cell. It makes no difference, because a path passed from bottom to top, may be passed from top to bottom just in reverse order. In this way we get three paths going from top to bottom. This somehow decreases the difficulty of the problem. We can consider these 3 paths as left, middle and right. When 2 paths intersect (like in the figure below)



we may consider them as in the following picture, without affecting the result:



This way we'll get 3 paths, which we may consider as being one left, one middle and the other - right. More than that, we may see that for getting an optimal results they must not intersect (except in the leftmost upper corner and rightmost bottom corner). So for each row **y** (except first and last), the **x** coordinates of the lines (**x1[y]** , **x2[y]** and respectively **x3[y]**) will be : **x1[y]** < **x2[y]** < **x3[y]** . Having done that - the DP solution now becomes much clearer. Let's consider the row y. Now suppose that for any configuration of **x1[y-1]** , **x2[y-1]** and **x3[y-1]** we have already found the paths which collect the maximum number of apples. From them we can find the optimal solution for row **y**. We now have to find only the way for passing from one row to the next one. Let **Max[i][j][k]** represent the maximum number of apples collected till row **y-1** inclusive, with three paths finishing at column **i**, **j**, and respectively **k**. For the next row **y**, add to each **Max[i][j][k]** (obtained previously) the number of apples situated in cells (**y,i**) , (**y,j**) and (**y,k**). Thus we move down at each step. After we made such a move, we must consider that the paths may move in a row to the right. For keeping the paths out of an intersection, we must first consider the move to the right of the left path, after this of the middle path, and then of the right path. For a better understanding think about the move to the right of the left path - take every possible pair of, **k** (where **j<k**), and for each i (1 i<j) consider the move from position (**i-1,j,k**) to position (**i,j,k**). Having done this for

the left path, start processing the middle one, which is done similarly; and then process the right path.

TC problems for practicing:

- MiniPaint - SRM 178 Div 1

Additional Note:
When have read the description of a problem and started to solve it, first look at its restrictions. If a polynomial-time algorithm should be developed, then it's possible that the solution may be of DP type. In this case try to see if there exist such states (sub-solutions) with the help of which the next states (sub-solutions) may be found. Having found that - think about how to pass from one state to another. If it seems to be a DP problem, but you can't define such states, then try to reduce the problem to another one (like in the example above, from Section 5).

Mentioned in this writeup:
TCCC '03 Semifinals 3 Div I Easy - ZigZag
TCCC '04 Round 4 Div I Easy - BadNeighbors
TCCC '04 Round 1 Div I Med - FlowerGarden
TCO '03 Semifinals 4 Div I Easy - AvoidRoads
TCCC '03 Round 4 Div I Easy - ChessMetric
TCO '03 Round 4 Div I Med - Jewelry
SRM 150 Div I Med - StripePainter
SRM 197 Div II Hard - QuickSums
SRM 165 Div II Hard - ShortPalindromes
SRM 208 Div I Hard - StarAdventure
SRM 178 Div I Hard - MiniPaint

In this article I'll try to introduce you to the area of computation complexity. The article will be a bit long before we get to the actual formal definitions because I feel that the rationale behind these definitions needs to be explained as well - and that understanding the rationale is even more important than the definitions alone.

Why is it important?

Example 1. Suppose you were assigned to write a program to process some records your company receives from time to time. You implemented two different algorithms and tested them on several sets of test data. The processing times you obtained are in Table 1.

# of records	10	20	50	100	1000	5000
algorithm 1	0.00s	0.01s	0.05s	0.47s	23.92s	47min
algorithm 2	0.05s	0.05s	0.06s	0.11s	0.78s	14.22s

Table 1. Runtimes of two fictional algorithms.

In praxis, we probably could tell which of the two implementations is better for us (as we usually can estimate the amount of data we will have to process). For the company this solution may be fine. But from the programmer's point of view, it would be much better if he could estimate the values in Table 1 **before** writing the actual code - then he could only implement the better algorithm.

The same situation occurs during programming contests: The size of the input data is given in the problem statement. Suppose I found an algorithm. Questions I have to answer before I start to type should be: Is my algorithm worth implementing? Will it solve the largest test cases in time? If I know more algorithms solving the problem, which of them shall I implement?

This leads us to the question: How to compare algorithms? Before we answer this question in general, let's return to our simple example. If we extrapolate the data in Table 1, we may assume that if the number of processed records is larger than 1000, algorithm 2 will be substantially faster. In other words, if we consider all possible inputs, algorithm 2 will be better for almost all of them.

It turns out that this is almost always the case - given two algorithms, either one of them is almost always better, or they are approximately the same. Thus, this will be our definition of a better algorithm. Later, as we define everything formally, this will be the general idea behind the definitions.

A neat trick

If you thing about Example 1 for a while, it shouldn't be too difficult to see that there is an algorithm with runtimes similar to those in Table 2:

# of records	10	20	50	100	1000	5000
algorithm 3	0.00s	0.01s	0.05s	0.11s	0.78s	14.22s

Table 2. Runtimes of a new fictional algorithm.

The idea behind this algorithm: Check the number of records. If it is small enough, run algorithm 1, otherwise run algorithm 2.

Similar ideas are often used in praxis. As an example consider most of the sort() functions provided by various libraries. Often this function is an implementation of QuickSort with various improvements, such as:

- if the number of elements is too small, run InsertSort instead (as InsertSort is faster for small inputs)
- if the pivot choices lead to poor results, fall back to MergeSort

What is efficiency?

Example 2. Suppose you have a concrete implementation of some algorithm. (The example code presented below is actually an implementation of MinSort - a slow but simple sorting algorithm.)

```
for (int i=0; i<N; i++)
  for (int j=i+1; j<N; j++)
    if (A[i] > A[j])
      swap( A[i], A[j] );
```

If we are given an input to this algorithm (in our case, the array A and its size N), we can exactly compute the number of steps our algorithm does on this input. We could even count the processor instructions if we wanted to. However, there are too many possible inputs for this approach to be practical.

And we still need to answer one important question: What is it exactly we are interested in? Most usually it is the behavior of our program in the **worst possible case** - we need to look at the input data and to determine an upper bound on how long will it take if we run the program. But then, what is the worst possible case? Surely we can always make the program run longer simply by giving it a larger input. Some of the more important questions are: What is the worst input with 700 elements? **How fast** does the maximum runtime grow when we increase the input size?

Formal notes on the input size

What exactly is this "input size" we started to talk about? In the formal definitions this is the size of the input written in some fixed finite alphabet (with at least 2 "letters"). For our needs, we may consider this alphabet to be the numbers 0..255. Then the "input size" turns out to be exactly the size of the input file in bytes.

Usually a part of the input is a number (or several numbers) such that the size of the input is proportional to the number.

E.g. in Example 2 we are given an int N and an array containing N ints. The size of the input file will be roughly 5N (depending on the OS and architecture, but always linear in N).

In such cases, we may choose that this number will represent the size of the input. Thus when talking about problems on arrays/strings, the input size is the length of the array/string, when talking about graph problems, the input size depends both on the number of vertices (N) and the number of edges (M), etc.

We will adopt this approach and use N as the input size in the following parts of the article.
There is one tricky special case you sometimes need to be aware of. To write a (possibly large) number we need only logarithmic space. (E.g. to write 123456, we need only roughly $\log_{10}(123456)$ digits.) This is why the naive primality test does not run in polynomial time - its runtime is polynomial in the **size** of the number, but not in its **number of digits**! If you didn't understand the part about polynomial time, don't worry, we'll get there later.
How to measure efficiency?
We already mentioned that given an input we are able to count the number of steps an algorithm makes simply by simulating it. Suppose we do this for all inputs of size at most N and find the worst of these inputs (i.e. the one that causes the algorithm to do the most steps). Let $f(N)$ be this number of steps. We will call this function the time complexity, or shortly the runtime of our algorithm.
In other words, if we have any input of size N, solving it will require at most $f(N)$ steps.
Let's return to the algorithm from Example 2. What is the worst case of size N? In other words, what array with N elements will cause the algorithm to make the most steps? If we take a look at the algorithm, we can easily see that:

- the first step is executed exactly N times
- the second and third step are executed exactly $N(N - 1)/2$ times
- the fourth step is executed at most $N(N - 1)/2$ times

Clearly, if the elements in A are in descending order at the beginning, the fourth step will always be executed. Thus in this case the algorithm makes $3N(N - 1)/2 + N = 1.5N^2 - 0.5N$ steps. Therefore our algorithm has $f(N) = 1.5N^2 - 0.5N$.
As you can see, determining the exact function f for more complicated programs is painful. Moreover, it isn't even necessary. In our case, clearly the $-0.5N$ term can be neglected. It will usually be much smaller than the $1.5N^2$ term and it won't affect the runtime significantly. The result "$f(N)$ is roughly equal to $1.5N^2$" gives us all the information we need. As we will show now, if we want to compare this algorithm with some other algorithm solving the same problem, even the constant 1.5 is not that important.
Consider two algorithms, one with the runtime N^2, the other with the runtime $0.001N^3$. One can easily see that for N greater than 1 000 the first algorithm is faster - and soon this difference becomes apparent. While the first algorithm is able to solve inputs with N = 20 000 in a matter of seconds, the second one will already need several minutes on current machines.
Clearly this will occur always when one of the runtime functions grows **asymptotically faster** than the other (i.e. when N grows beyond all bounds the limit of their quotient is zero or infinity). Regardless of the constant factors, an algorithm with runtime proportional to N^2 will always be better than an algorithm with runtime proportional to N^3 **on almost all inputs**. And this observation is exactly what we base our formal definition on.
Finally, formal definitions
Let f, g be positive non-decreasing functions defined on positive integers. (Note that all runtime functions satisfy these conditions.) We say that $f(N)$ is $O(g(N))$ *(read: f is big-oh of g)* if for some c and N_0 the following condition holds:

$$\forall N > N_0; f(N) < c.g(N)$$

In human words, $f(N)$ is $O(g(N))$, if for some c almost the entire graph of the function f is below the graph of the function $c.g$. Note that this means that f grows at most as fast as $c.g$ does.
Instead of "$f(N)$ is $O(g(N))$" we usually write $f(N) = O(g(N))$. Note that this "equation" is **not symmetric** - the notion "$O(g(N)) = f(N)$" has no sense and "$g(N) = O(f(N))$" doesn't have to be true (as we will see later). (If you are not comfortable with this notation, imagine $O(g(N))$ to be a set of functions and imagine that there is a $\in$ instead of =.)
What we defined above is known as the big-oh notation and is conveniently used to specify upper bounds on function growth.
E.g. consider the function $f(N) = 3N(N - 1)/2 + N = 1.5N^2 - 0.5N$ from Example 2. We may say that $f(N) = O(N^2)$ (one possibility for the constants is $c = 2$ and $N_0 = 0$). This means that f doesn't grow (asymptotically) faster than N^2.
Note that even the exact runtime function f doesn't give an exact answer to the question "How long will the program run on my machine?" But the important observation in the example case is that the runtime function is quadratic. If we double the input size, the runtime will increase approximately to four times the current runtime, no matter how fast our computer is.
The $f(N) = O(N^2)$ upper bound gives us almost the same - it guarantees that the growth of the runtime function is at most quadratic.
Thus, we will use the O-notation to describe the time (and sometimes also memory) complexity of algorithms. For the algorithm from Example 2 we would say "The time complexity of this algorithm is $O(N^2)$" or shortly "This algorithm is $O(N^2)$".

In a similar way we defined O we may define Ω and Θ.
We say that $f(N)$ is $\Omega(g(N))$ if $g(N) = O(f(N))$, in other words if f grows at least as fast as g.
We say that $f(N) = \Theta(g(N))$ if $f(N) = O(g(N))$ and $g(N) = O(f(N))$, in other words if both functions have approximately the same rate of growth.
As it should be obvious, Ω is used to specify lower bounds and Θ is used to give a tight asymptotic bound on a function. There are other similar bounds, but these are the ones you'll encounter most of the time.
Some examples of using the notation

- $1.5N^2 - 0.5N = O(N^2)$.
- $47N \log N = O(N^2)$.
- $N \log N + 1\,000\,047N = \Theta(N \log N)$.
- All polynomials of order k are $O(N^k)$.
- The time complexity of the algorithm in Example 2 is $\Theta(N^2)$.
- If an algorithm is $O(N^2)$, it is also $O(N^5)$.
- Each comparision-based sorting algorithm is $\Omega(N \log N)$.

- MergeSort run on an array with N elements does roughly $N \log N$ comparisions. Thus the time complexity of MergeSort is $\Theta(N \log N)$. If we trust the previous statement, this means that MergeSort is an asymptotically optimal general sorting algorithm.
- The algorithm in Example 2 uses $\Theta(N)$ bytes of memory.
- The function giving my number of teeth in time is $O(1)$.
- A naive backtracking algorithm trying to solve chess is $O(1)$ as the tre of positions it will examine is finite. (But of course in this case the constant hidden behind the $O(1)$ is unbelievably large.)
- The statement "Time complexity of this algorithm is at least $O(N^2)$" is meaningless. (It says: "Time complexity of this algorithm is at least at most roughly quadratic." The speaker probably wanted to say: "Time complexity of this algorithm is $\Omega(N^2)$.")

When speaking about the time/memory complexity of an algorithm, instead of using the formal $\Theta(f(n))$-notation we may simply state the class of functions f belongs to. E.g. if $f(N) = \Theta(N)$, we call the algorithm *linear*. More examples:

- $f(N) = \Theta(\log N)$: logarithmic
- $f(N) = \Theta(N^2)$: quadratic
- $f(N) = \Theta(N^3)$: cubic
- $f(N) = O(N^k)$ for some k: polynomial
- $f(N) = \Omega(2^N)$: exponential

For graph problems, the complexity $\Theta(N + M)$ is known as "linear in the graph size".

Determining execution time from an asymptotic bound

For most algorithms you may encounter in praxis, the constant hidden behind the O (or Θ) is usually relatively small. If an algorithm is $\Theta(N^2)$, you may expect that the exact time complexity is something like $10N^2$, not 10^7N^2.

The same observation in other words: if the constant is large, it is usually somehow related to some constant in the problem statement. In this case it is good practice to give this constant a name and to include it in the asymptotic notation.

An example: The problem is to count occurences of each letter in a string of N letters. A naive algorithm passes through the whole string once for each possible letter. The size of alphabet is fixed (e.g. at most 255 in C), thus the algorithm is linear in N. Still, it is better to write that its time complexity is $\Theta(|S|.N)$, where S is the alphabet used. (Note that there is a better algorithm solving this problem in $\Theta(|S| + N)$.)

In a TopCoder contest, an algorithm doing 1 000 000 000 multiplications runs barely in time. This fact together with the above observation and some experience with TopCoder problems can help us fill the following table:

complexity	maximum N
$\Theta(N)$	100 000 000
$\Theta(N \log N)$	40 000 000
$\Theta(N^2)$	10 000
$\Theta(N^3)$	500
$\Theta(N^4)$	90
$\Theta(2^N)$	20
$\Theta(N!)$	11

Table 3. Approximate maximum problem size solvable in 8 seconds.

A note on algorithm analysis

Usually if we present an algorithm, the best way to present its time complexity is to give a Θ-bound. However, it is common practice to only give an O-bound - the other bound is usually trivial, O is much easier to type and better known. Still, don't forget that O represents only an upper bound. Usually we try to find an O-bound that's as good as possible.

Example 3. Given is a sorted array A. Determine whether it contains two elements with the difference D. Consider the following code solving this problem:

```
int j=0;
for (int i=0; i<N; i++) {
  while ( (j<N-1) && (A[i]-A[j] > D) )
    j++;
  if (A[i]-A[j] == D) return 1;
}
```

It is easy to give an $O(N^2)$ bound for the time complexity of this algorithm - the inner while-cycle is called N times, each time we increase j at most N times. But a more careful analysis shows that in fact we can give an $O(N)$ bound on the time complexity of this algorithm - it is sufficient to realize that during the **whole execution** of the algorithm the command "j++;" is executed no more than N times.

If we said "this algorithm is $O(N^2)$", we would have been right. But by saying "this algorithm is $O(N)$" we give more information about the algorithm.

Conclusion

We have shown how to write bounds on the time complexity of algorithms. We have also demonstrated why this way of characterizing algorithms is natural and (usually more-or-less) sufficient.

The next logical step is to show how to estimate the time complexity of a given algorithm. As we have already seen in Example 3, sometimes this can be messy. It gets really messy when recursion is involved. We will address these issues in the second part of this article.

In this part of the article we will focus on estimating the time complexity for recursive programs. In essence, this will lead to finding the order of growth for solutions of recurrence equations. Don't worry if you don't understand what exactly is a recurrence solution, we will explain it in the right place at the right time. But first we will consider a simpler case - programs without recursion.

Nested loops

First of all let's consider simple programs that contain no function calls. The rule of thumb to find an upper bound on the time complexity of such a program is:

- estimate the maximum number of times each loop can be executed,
- **add** these bounds for cycles following each other.
- **multiply** these bounds for nested cycles/parts of code,

Example 1. Estimating the time complexity of a random piece of code.

```
int result=0;                              //  1
for (int i=0; i<N; i++)                    //  2
  for (int j=i; j<N; j++) {                //  3
    for (int k=0; k<M; k++) {              //  4
      int x=0;                             //  5
      while (x<N) { result++; x+=3; }      //  6
    }                                      //  7
    for (int k=0; k<2*M; k++)              //  8
      if (k%7 == 4) result++;              //  9
  }                                        // 10
```

The time complexity of the while-cycle in line 6 is clearly $O(N)$ - it is executed no more than $N/3 + 1$ times.

Now consider the for-cycle in lines 4-7. The variable k is clearly incremented $O(M)$ times. Each time the whole while-cycle in line 6 is executed. Thus the total time complexity of the lines 4-7 can be bounded by $O(MN)$.

The time complexity of the for-cycle in lines 8-9 is $O(M)$. Thus the execution time of lines 4-9 is $O(MN + M) = O(MN)$.

This inner part is executed $O(N^2)$ times - once for each possible combination of i and j. (Note that there are only $N(N + 1)/2$ possible values for $[i, j]$. Still, $O(N^2)$ is a correct upper bound.)

From the facts above follows that the total time complexity of the algorithm in Example 1 is $O(N^2.MN) = O(MN^3)$.

From now on we will assume that the reader is able to estimate the time complexity of simple parts of code using the method demonstrated above. We will now consider programs using recursion (i.e. a function occasionally calling itself with different parameters) and try to analyze the impact of these recursive calls on their time complexity.

Using recursion to generate combinatorial objects

One common use of recursion is to implement a *backtracking* algorithm to generate all possible solutions of a problem. The general idea is to generate the solution incrementally and to step back and try another way once all solutions for the current branch have been exhausted.

This approach is not absolutely universal, there may be problems where it is impossible to generate the solution incrementally. However, very often the set of all possible solutions of a problem corresponds to the set of all combinatorial objects of some kind. Most often it is the set of all permutations (of a given size), but other objects (combinations, partitions, etc.) can be seen from time to time.

As a side note, it is always possible to generate all strings of zeroes and ones, check each of them (i.e. check whether it corresponds to a valid solution) and keep the best found so far. If we can find an upper bound on the size of the best solution, this approach is finite. However, this approach is everything but fast. Don't use it if there is **any** other way.

Example 2. A trivial algorithm to generate all permutations of numbers 0 to $N - 1$.

```
vector<int> permutation(N);
vector<int> used(N,0);

void try(int which, int what) {
  // try taking the number "what" as the "which"-th element
  permutation[which] = what;
  used[what] = 1;

  if (which == N-1)
    outputPermutation();
  else
    // try all possibilities for the next element
    for (int next=0; next<N; next++)
      if (!used[next])
        try(which+1, next);

  used[what] = 0;
}
```

```
int main() {
  // try all possibilities for the first element
  for (int first=0; first<N; first++)
    try(0,first);
}
```

In this case a trivial **lower** bound on the time complexity is the number of possible solutions. Backtracking algorithms are usually used to solve hard problems - i.e. such that we don't know whether a significantly more efficient solution exists. Usually the solution space is quite large and uniform and the algorithm can be implemented so that its time complexity is close to the theoretical lower bound. To get an upper bound it should be enough to check how much additional (i.e. unnecessary) work the algorithm does.
The number of possible solutions, and thus the time complexity of such algorithms, is usually exponential - or worse.
Divide&conquer using recursion
From the previous example we could get the feeling that recursion is evil and leads to horribly slow programs. The contrary is true. Recursion can be a very powerful tool in the design of effective algorithms. The usual way to create an effective recursive algorithm is to apply the *divide&conquer paradigm* - try to split the problem into several parts, solve each part separately and in the end combine the results to obtain the result for the original problem. Needless to say, the "solve each part separately" is usually implemented using recursion - and thus applying the same method again and again, until the problem is sufficiently small to be solved by brute force.
Example 3. The sorting algorithm MergeSort described in pseudocode.

```
MergeSort(sequence S) {
  if (size of S <= 1) return S;
  split S into S_1 and S_2 of roughly the same size;
  MergeSort(S_1);
  MergeSort(S_2);
  combine sorted S_1 and sorted S_2 to obtain sorted S;
  return sorted S;
}
```

Clearly $O(N)$ time is enough to split a sequence with N elements into two parts. (Depending on the implementation this may be even possible in constant time.) Combining the shorter sorted sequences can be done in $\Theta(N)$: Start with an empty S. At each moment the smallest element not yet in S is either at the beginning of S_1 or at the beginning of S_2. Move this element to the end of S and continue.
Thus the total time to MergeSort a sequence with N elements is $\Theta(N)$ plus the time needed to make the two recursive calls.
Let $f(N)$ be the time complexity of MergeSort as defined in the previous part of our article. The discussion above leads us to the following equation:

$$f(N) = f(\lfloor N/2 \rfloor) + f(\lceil N/2 \rceil) + p(N)$$

where p is a linear function representing the amount of work spent on splitting the sequence and merging the results.
Basically, this is just a *recurrence equation*. If you don't know this term, please don't be afraid. The word "recurrence" stems from the latin phrase for "to run back". Thus the name just says that the next values of f are defined using the previous (i.e. smaller) values of f.
Well, to be really formal, for the equation to be complete we should specify some initial values - in this case, $f(1)$. This (and knowing the implementation-specific function p) would enable us to compute the exact values of f.
But as you hopefully understand by now, this is not necessarily our goal. While it is theoretically possible to compute a closed-form formula for $f(N)$, this formula would most probably be really ugly... and we don't really need it. We only want to find a Θ-bound (and sometimes only an O-bound) on the growth of f. Luckily, this can often be done quite easily, if you know some tricks of the trade.
As a consequence, we won't be interested in the exact form of p, all we need to know is that $p(N) = \Theta(N)$. Also, we don't need to specify the initial values for the equation. We simply assume that all problem instances with small N can be solved in constant time.
The rationale behind the last simplification: While changing the initial values does change the solution to the recurrence equation, it usually doesn't change its asymptotic order of growth. (If your intuition fails you here, try playing with the equation above. For example fix p and try to compute $f(8)$, $f(16)$ and $f(32)$ for different values of $f(1)$.)
If this would be a formal textbook, at this point we would probably have to develop some theory that would allow us to deal with the floor and ceiling functions in our equations. Instead we will simply neglect them from now on. (E.g. we can assume that each division will be integer division, rounded down.)
A reader skilled in math is encouraged to prove that if p is a polynomial (with non-negative values on N) and $q(n) = p(n + 1)$ then $q(n) = \Theta(p(n))$. Using this observation we may formally prove that (assuming the f we seek is polynomially-bounded) the right side of each such equation remains asymptotically the same if we replace each ceiling function by a floor function.
The observations we made allow us to rewrite our example equation in a more simple way:

$$f(N) = 2\, f(N/2) + \Theta(N) \tag{1}$$

Note that this is not an equation in the classical sense. As in the examples in the first part of this article, the equals sign now reads "is asymptotically equal to". Usually there are lots of different functions that satisfy such an equation. But usually all of them will have the same order of growth - and this is exactly what we want to determine. Or, more generally, we want to find the smallest upper bound on the growth of **all possible** functions that satisfy the given equation.
In the last sections of this article we will discuss various methods of solving these "equations". But before we can do that, we need to know a bit more about logarithms.
Notes on logarithms

By now, you may have already asked one of the following questions: If the author writes that some complexity is e.g. $O(N \log N)$, what is the base of the logarithm? In some cases, wouldn't $O(N \log_2 N)$ be a better bound?
The answer: The base of the logarithm does not matter, all logarithmic functions (with base > 1) are asymptotically equal. This is due to the well-known equation:

$$\log_a N = \frac{\log_b N}{\log_b a} \tag{2}$$

Note that given two bases a, b, the number $1/\log_b a$ is just a constant, and thus the function $\log_a N$ is just a constant multiple of $\log_b N$.
To obtain more clean and readable expressions, we always use the notation log N inside big-Oh expressions, even if logarithms with a different base were used in the computation of the bound.
By the way, sadly the meaning of log N differs from country to country. To avoid ambiguity where it may occur: I use log N to denote the decadic (i.e. base-10) logarithm, ln N for the natural (i.e. base-e) logarithm, lg N for the binary logarithm and $\log_b N$ for the general case.

Now we will show some useful tricks involving logarithms, we will need them later. Suppose a, b are given constants such that a, $b > 1$.
From (2) we get:

$$\log_a b = \frac{\log_b b}{\log_b a} = \frac{1}{\log_b a}$$

Using this knowledge, we can simplify the term $a^{\log_b N}$ as follows:

$$a^{\log_b N} = a^{\log_a N / \log_a b} = \left(a^{\log_a N}\right)^{1/\log_a b} = \left(a^{\log_a N}\right)^{\log_b a} = N^{\log_b a} \tag{3}$$

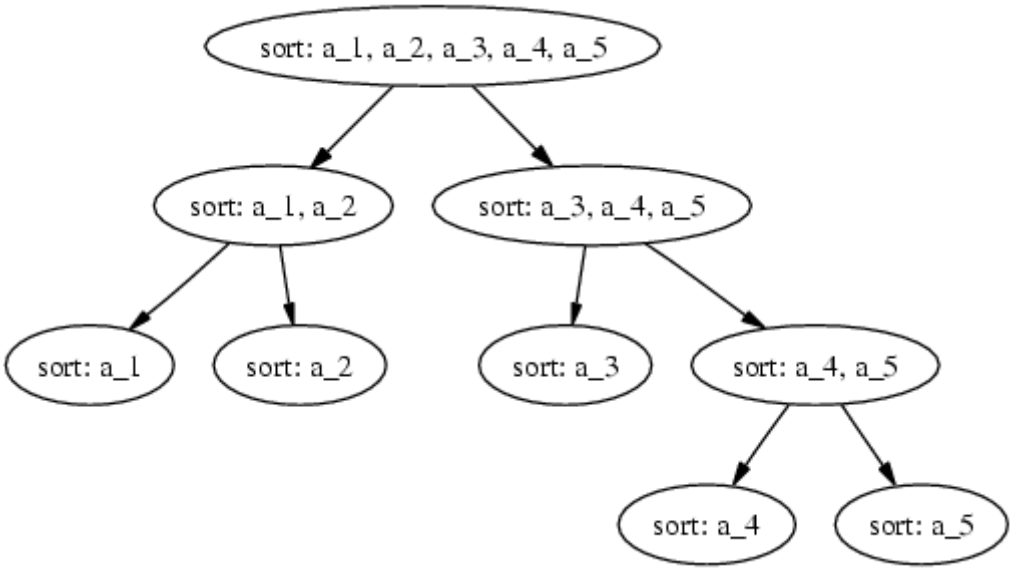
The substitution method
This method can be summarized in one sentence: Guess an asymptotic upper bound on f and (try to) prove it by induction.
As an example, we will prove that if f satisfies the equation (1) then $f(N) = O(N \log N)$.
From (1) we know that

$$\forall N;\ f(N) \le 2f(N/2) + cN$$

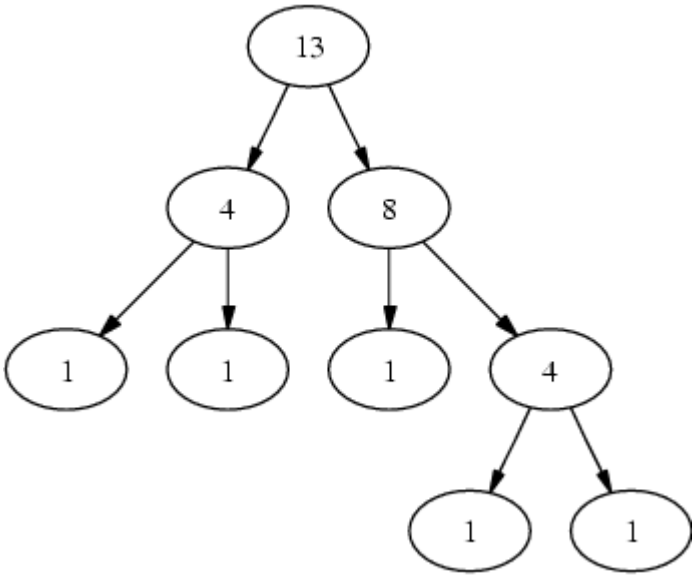
for some c. Now we will prove that if we take a large enough (but constant) d then for almost all N we have $f(N) \le dN \lg N$. We will start by proving the induction step.
Assume that $f(N/2) \le d\,(N/2)\lg(N/2)$. Then

$$\begin{aligned} f(N) &\le 2f(N/2) + cN \\ &\le 2d(N/2)\lg(N/2) + cN \\ &= dN(\lg N - \lg 2) + cN \\ &= dN \lg N - dN + cN \end{aligned}$$

In other words, the induction step will hold as long as $d > c$. We are always able to choose such d.
We are only left with proving the inequality for some initial value N. This gets quite ugly when done formally. The general idea is that if the d we found so far is not large enough, we can always increase it to cover the initial cases.
Note that for our example equation we won't be able to prove it for $N = 1$, because lg 1 = 0. However, by taking $d > 2(f(1) + f(2) + f(3) + c)$ we can easily prove the inequality for $N = 2$ and $N = 3$, which is more than enough.
Please note what exactly did we prove. Our result is that if f satisfies the equation (1) then for almost all N we have $f(N) \le dN \lg N$, where d is some fixed constant. Conclusion: from (1) it follows that $f(N) = O(N \lg N)$.
The recursion tree
To a beginner, the previous method won't be very useful. To use it successfully we need to make a good guess - and to make a good guess we need some insight. The question is, how to gain this insight? Let's take a closer look at what's happening, when we try to evaluate the recurrence (or equivalently, when we run the corresponding recursive program).
We may describe the execution of a recursive program on a given input by a rooted tree. Each node will correspond to some instance of the problem the program solves. Consider an arbitrary vertex in our tree. If solving its instance requires recursive calls, this vertex will have children corresponding to the smaller subproblems we solve recursively. The root node of the tree is the input of the program, leaves represent small problems that are solved by brute force.
Now suppose we label each vertex by the amount of work spent solving the corresponding problem (excluding the recursive calls). Clearly the runtime is exactly the sum of all labels.
As always, we only want an asymptotic bound. To achieve this, we may "round" the labels to make the summation easier. Again, we will demonstrate this method on examples.
Example 4. The recursion tree for MergeSort on 5 elements.

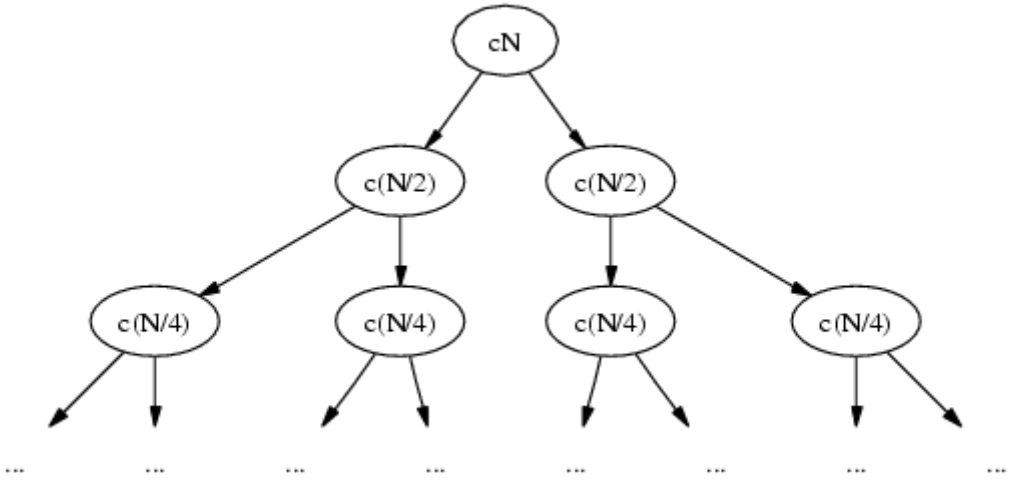


The recursion tree for the corresponding recurrence equation. This time, the number inside each vertex represents the number of steps the algorithm makes there.



Note that in a similar way we may sketch the general form of the recursion tree for any recurrence. Consider our old friend, the equation (1). Here we know that there is a number *c* such that the number of operations in each node can be bound by (*c* times the current value of *N*). Thus the tree in the example below is indeed the worst possible case.

Example 5. A worst-case tree for the general case of the recurrence equation (1).



Now, the classical trick from combinatorics is to sum the elements in an order different from the order in which they were created. In this case, consider an arbitrary level of the tree (i.e. a set of vertices with the same depth). It is not hard to see that the total work on each of the levels is *cN*.

Now comes the second question: What is the number of levels? Clearly, the leaves correspond to the trivial cases of the algorithm. Note that the size of the problem is halved in each step. Clearly after lg *N* steps we are left with a trivial problem of size 1, thus the number of levels is Θ (log *N*).

Combining both observations we get the final result: The total amount of work done here is Θ (*cN* x log *N*) = Θ (*N* log *N*).

A side note. If the reader doesn't trust the simplifications we made when using this method, he is invited to treat this method as a "way of making a good guess" and then to prove the result using the substitution method. However, with a little effort the application of this method could also be upgraded to a full formal proof.

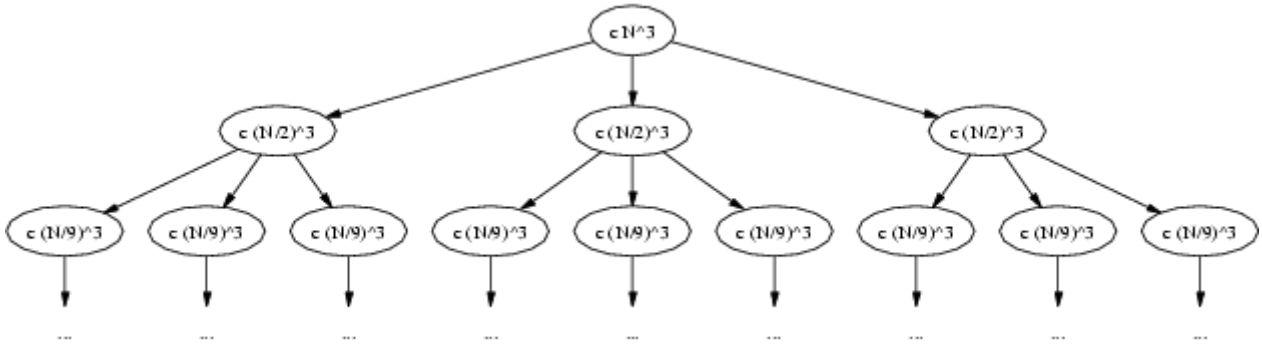
More recursion trees

By now you should be asking: Was it really only a coincidence that the total amount of work on each of the levels in Example 5 was the same? The answer: No and yes. No, there's a simple reason why this happened, we'll discover it later. Yes, because this is not always the case - as we'll see in the following two examples.

Example 6. Let's try to apply our new "recursion tree" method to solve the following recurrence equation:

$$f(N) = 3f(N/2) + \Theta(N^3)$$

The recursion tree will look as follows:



Let's try computing the total work for each of the first few levels. Our results:

level	1	2	3	...
work	cN^3	$\frac{3}{8}cN^3$	$\frac{3^2}{8^2}cN^3$	...

Clearly as we go deeper in the tree, the total amount of work on the current level decreases. The question is, how fast does it decrease? As we move one level lower, there will be three times that many subproblems. However, their size gets divided by 2, and thus the time to process each of them decreases to one eighth of the original time. Thus the amount of work is decreased by the factor 3/8.

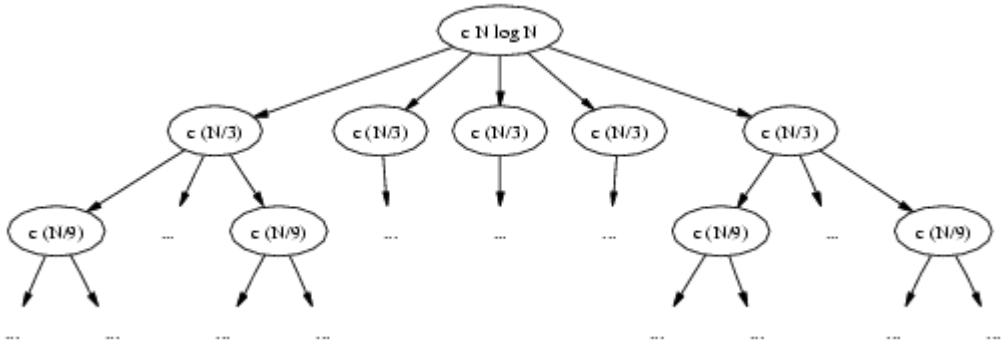
But this means that the entries in the table above form a geometric progression. For a while assume that this progression is infinite. Then its sum would be

$$S = \frac{cN^3}{1 - \frac{3}{8}} = \frac{8}{5}cN^3 = \Theta(N^3)$$

Thus the total amount of work in our tree is $\Omega(N^3)$ (summing the infinite sequence gives us an upper bound). But already the first element of our progression is $\Theta(N^3)$. It follows that the total amount of work in our tree is $\Theta(N^3)$ and we are done.
The important generalization of this example: If the amounts of work at subsequent levels of the recursion tree form a **decreasing geometric progression**, the total amount of work is asymptotically the same as the amount of work done in the root node.
From this result we can deduce an interesting fact about the (hypothetical) algorithm behind this recurrence equation: The recursive calls didn't take much time in this case, the most time consuming part was preparing the recursive calls and/or processing the results. (I.e. this is the part that should be improved if we need a faster algorithm.)
Example 7. Now let's try to apply our new "recursion tree" method to solve the following recurrence equation:

$$f(N) = 5f(N/3) + \Theta(N)$$

The recursion tree will look as follows:



Again, let's try computing the total work for each of the first few levels. We get:

level	1	2	3	...
work	cN	$\frac{5}{3}cN$	$\frac{5^2}{3^2}cN$	...

This time we have the opposite situation: As we go deeper in the tree, the total amount of work on the current level increases. As we move one level lower, there will be five times that many subproblems, each of them one third of the previous size, the processing time is linear in problem size. Thus the amount of work increased by the factor 5/3.
Again, we want to compute the total amount of work. This time it won't be that easy, because the most work is done on the lowest level of the tree. We need to know its depth.
The lowest level corresponds to problems of size 1. The size of a problem on level k is $N/3^k$. Solving the equation $1 = N/3^k$ we get $k = \log_3 N$. Note that this time we explicitly state the base of the logarithm, as this time it will be important.
Our recursion tree has $\log_3 N$ levels. Each of the levels has five times more vertices than the previous one, thus the last level has $5^{\log_3 N}$ levels.
The total work done on this level is then $c5^{\log_3 N}$.
Note that using the trick (3) we may rewrite this as $cN^{\log_3 5}$.
Now we want to sum the work done on all levels of the tree. Again, this is a geometric progression. But instead of explicitly computing the sum, we now **reverse** it. Now we have a **decreasing** geometric progression...and we are already in the same situation as in the previous example. Using the same reasoning we can show that the sum is asymptotically equal to the largest element.
It follows that the total amount of work in our tree is $\Theta(N^{\log_3 5}) \approx \Theta(N^{1.465})$ and we are done.
Note that the base-3 logarithm ends in the exponent, that's why the base is important. If the base was different, also the result would be asymptotically different.
The Master Theorem
We already started to see a pattern here. Given a recurrence equation, take the corresponding recurrence tree and compute the amounts of work done on each level of the tree. You will get a geometric sequence. If it decreases, the total work is proportional to work done in the root node. If it increases, the total work is proportional to the number of leaves. If it remains the same, the total work is (the work done on one level) times (the number of levels).
Actually, there are a few ugly cases, but almost often one of these three cases occurs. Moreover, it is possible to prove the statements from the previous paragraph formally. The formal version of this theorem is known under the name Master Theorem.
For reference, we give the full formal statement of this theorem. (Note that knowing the formal proof is not necessary to **apply** this theorem on a given recurrence equation.)
Let $a \geq 1$ and $b > 1$ be integer constants. Let p be a non-negative non-decreasing function. Let f be any solution of the recurrence equation

$$f(N) = af(N/b) + p(N)$$

Then:

1. If $p(N) = O(N^{(\log_b a)-\varepsilon})$ for some $\varepsilon > 0$ then $f(N) = \Theta(N^{\log_b a})$

2. If $p(N) = \Theta(N^{log_b a})$, then $f(N) = \Theta(p(N) \log N)$.
3. If $p(N) = \Omega(N^{(log_b a)+\varepsilon})$ for some $\varepsilon > 0$, and if $ap(N/b) \leq cp(N)$ for some $c < 1$ and for almost all N, then $f(N) = \Theta(p(N))$.

Case 1 corresponds to our Example 7. Most of the time is spent making the recursive calls and it's the number of these calls that counts.
Case 2 corresponds to our Example 5. The time spent making the calls is roughly equal to the time to prepare the calls and process the results. On all levels of the recursion tree we do roughly the same amount of work, the depth of the tree is always logarithmic.
Case 3 corresponds to our Example 6. Most of the time is spent on preparing the recursive calls and processing the results. Usually the result will be asymptotically equal to the time spent in the root node.
Note the word "usually" and the extra condition in Case 3. For this result to hold we need p to be somehow "regular" - in the sense that for each node in the recursion tree the time spent in the node must be greater than the time spent in its chidren (excluding further recursive calls). This is nothing to worry about too much, most probably all functions p you will encounter in practice will satisfy this condition (if they satisfy the first condition of Case 3).

Example 8. Let $f(N)$ be the time Strassen's fast matrix multiplication algorithm needs to multiply two N x N square matrices. This is a recursive algorithm, that makes 7 recursive calls, each time multiplying two (N/2) x (N/2) square matrices, and then computes the answer in $\Theta(N^2)$ time. This leads us to the following recurrence equation:

$$f(N) = 7f(N/2) + \Theta(N^2)$$

Using the Master Theorem, we see that Case 1 applies. Thus the time complexity of Strassen's algorithm is $\Theta(N^{\log_2 7}) \approx \Theta(N^{2.807})$.
Note that by implementing the definition of matrix multiplication we get only a $\Theta(N^3)$ algorithm.

Example 9. Occasionally we may encounter the situation when the problems in the recursive calls are not of the same size. An example may be the "median-of-five" algorithm to find the k-th element of an array. It can be shown that its time complexity satisfies the recurrence equation

$$f(N) = f(N/5) + f(7N/10 + 6) + \Theta(N)$$

How to solve it? Can the recursion tree be applied also in such asymmetric cases? Is there a more general version of Master Theorem that handles also these cases? And what should I do with the recurrence $f(N) = 4f(N/4) + \Theta(N \log N)$, where the Master Theorem doesn't apply? We won't answer these questions here. This article doesn't claim to be the one and only reference to computational complexity. If you are already asking these questions, you understand the basics you need for programming contests - and if you are interested in knowing more, there are good books around that can help you.
Thanks for reading this far. If you have any questions, comments, bug reports or any other feedback, please use the Round tables. I'll do my best to answer.

Introduction

A regular expression is a special string that describes a search pattern. Many of you have surely seen and used them already when typing expressions like ls(or dir) *.txt , to get a list of all the files with the extension txt. Regular expressions are very useful not only for pattern matching, but also for manipulating text. In SRMs regular expressions can be extremely handy. Many problems that require some coding can be written using regular expressions on a few lines, making your life much easier.

(Not so) Formal Description of Regular Expressions

A regular expression(regex) is one or more non-empty branches, separated by '|'. It matches anything that matches one of the branches. The following regular expression will match any of the three words "the","top","coder"(quotes for clarity).

REGEX is : the|top|coder
INPUT is : Marius is one of the topcoders.
Found the text "the" starting at index 17 and ending at index 20.
Found the text "top" starting at index 21 and ending at index 24.
Found the text "coder" starting at index 24 and ending at index 29.

A branch is one or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by a '*', '+', '?', or bound. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by `+' matches a sequence of 1 or more matches of the atom. An atom followed by `?' matches a sequence of 0 or 1 matches of the atom.

The following regular expression matches any successive occurrence of the words 'top' and 'coder'.

REGEX is: (top|coder)+
INPUT is: This regex matches topcoder and also codertop.
Found "topcoder" starting at index 19 and ending at index 27.
Found "codertop" starting at index 37 and ending at index 45.

A bound is '{' followed by an unsigned decimal integer, possibly followed by ',' possibly followed by another unsigned decimal integer, always followed by '}'.If there are two integers, the first may not exceed the second. An atom followed by a bound containing one integer i and no comma matches a sequence of exactly i matches of the atom. An atom followed by a bound containing one integer i and a comma matches a sequence of i or more matches of the atom. An atom followed by a bound containing two integers i and j matches a sequence of i through j (inclusive) matches of the atom. The following regular expression matches any sequence made of '1's having length 2,3 or 4 .

REGEX is: 1{2,4}
INPUT is: 101 + 10 = 111 , 11111 = 10000 + 1111
Found the text "111" starting at index 11 and ending at index 14.
Found the text "1111" starting at index 17 and ending at index 21.
Found the text "1111" starting at index 33 and ending at index 37.

One should observe that, greedily, the longest possible sequence is being matched and that different matches do not overlap. An atom is a regular expression enclosed in '()' (matching a match for the regular expression), a bracket expression (see below), '.' (matching any single character), '^' (matching the null string at the beginning of a line), '$' (matching the null string at the end of a line), a `\' followed by one of the characters `^.[$()|*+?{\' (matching that character taken as an ordinary character) or a single character with no other significance (matching that character). There is one more type of atom, the back reference: `\' followed by a non-zero decimal digit d matches the same sequence of characters matched by the d-th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) `([bc])\1' matches `bb' or `cc' but not `bc'.

The following regular expression matches a string composed of two lowercase words separated by any character.

Current REGEX is: ([a-z]+).\1
Current INPUT is: top-topcoder|coder
I found the text "top-top" starting at index 0 and ending at index 7.
I found the text "coder|coder" starting at index 7 and ending at index 18.

A bracket expression is a list of characters enclosed in '[]'. It normally matches any single character from the list. If the list begins with '^', it matches any single character not from the rest of the list. If two characters in the list are separated by `-', this is shorthand for the full range of characters between those two inclusive (e.g. '[0-9]' matches any decimal digit). With the exception of ']','^','-' all other special characters, including `\', lose their special significance within a bracket expression.

The following regular expression matches any 3 character words not starting with 'b','c','d' and ending in 'at'.

Current REGEX is: [^b-d]at
Current INPUT is: bat
No match found.

Current REGEX is: [^b-d]at
Current INPUT is: hat
I found the text "hat" starting at index 0 and ending at index 3.

This example combines most concepts presented above. The regex matches a set of open/close pair of html tags.

REGEX is: <([a-zA-Z][a-zA-Z0-9]*)((| [^>]*)>(.*)</\1>
INPUT is: TopCoder is the best

Found "TopCoder is the" starting at index 0 and ending at index 37.
Found "best" starting at index 38 and ending at index 49.
([a-zA-Z][a-zA-Z0-9]*) will match any word that starts with a letter and continues with an arbitrary number of letters or digits. (()| [^>]*) will match either the empty string or any string which does not contain '>' . \1 will be replaced using backreferencing with the word matched be ([a-zA-Z][a-zA-Z0-9]*)

The above description is a brief one covering the basics of regular expressions. A regex written following the above rules should work in both Java(>=1.4) and C++(POSIX EXTENDED). For a more in depth view of the extensions provided by different languages you can see the links given in the References section.

Using regular expressions

In java
In java(1.4 and above) there is a package "java.util.regex" which allows usage of regular expressions.

This package contains three classes : Pattern, Matcher and PatternSyntaxException.

- A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must call one of its public static compile methods, both of which will return a Pattern object.
- A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by calling the public matcher method on a Pattern object.
- A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Example(adapted from[4]):

```
Pattern pattern;
Matcher matcher;
pattern = Pattern.compile(<REGEX>);
matcher = pattern.matcher(<INPUT>);
boolean found;
while(matcher.find()) {
  System.out.println("Found the text \"" + matcher.group() +  "\" starting at index " +
matcher.start() +
       " and ending at index " + matcher.end() + ".");
  found = true;
}

if(!found){
  System.out.println("No match found.");
}
```

Java also offers the following methods in the String class:

- boolean matches(String regex) (returns if the current string matches the regular expression regex)
- String replaceAll(String regex, String replacement) (Replaces each substring of this string that matches the given regular expression with the given replacement.)
- String replaceFirst(String regex, String replacement) (Replaces the first substring of this string that matches the given regular expression with the given replacement.)
- String[] split(String regex) (Splits this string around matches of the given regular expression)

In C++
Many TopCoders believe that regular expressions are one of Java's main strengths over C++ in the arena. C++ programmers don't despair, regular expressions can be used in C++ too.

There are several regular expression parsing libraries available for C++, unfortunately they are not very compatible with each other. Fortunately as a TopCoder in the arena one does not have to cope with all this variety of "not so compatible with one another" libraries. If you plan to use regular expressions in the arena you have to choose between two flavors of regex APIs: POSIX_regex and GNU_regex. To use these APIs the header file "regex.h" must be included. Both of these work in two steps - first there is a function call that compiles the regular expression, and then there is a function call that uses that compiled regular expression to search or match a string.

Here is a short description of both of these APIs, leaving it up to the coder to choose the one that he likes the most.

POSIX_regex
Includes support for two different regular expression syntaxes, basic and extended. Basic regular expressions are similar to those in ed, while extended regular expressions are more like those in egrep, adding the '|', '+' and '?' operators and not requiring backslashes on parenthesized subexpressions or curly-bracketed bounds. Basic is the default, but extended is prefered.

With POSIX, you can only search for a given regular expression; you can't match it. To do this, you must first compile it in a pattern buffer, using `regcomp'. Once you have compiled the regular expression into a pattern buffer you can search in a null terminated string using 'regexec'. If either of the 'regcomp' or 'regexec' function fail they return an error code. To get an error string corresponding to these codes you must use

'regerror'. To free the allocated fields of a pattern buffer you must use 'regfree'.

For an in depth description of how to use these functions please consult [2] or [3] in the References section.

Example:
Here is a small piece of code showing how these functions can be used:

```
regex_t reg;

string pattern = "[^tpr]{2,}";
string str = "topcoder";

regmatch_t matches[1];

regcomp(&reg,pattern.c_str(),REG_EXTENDED|REG_ICASE);

if (regexec(&reg,str.c_str(),1,matches,0)==0) {
  cout << "Match "
  cout << str.substr(matches[0].rm_so,matches[0].rm_eo-matches[0].rm_so)
  cout << " found starting at: "
  cout << matches[0].rm_so
  cout << " and ending at "
  cout << matches[0].rm_eo
  cout << endl;
} else {
  cout << "Match not found."
  cout << endl;
}
regfree(&reg);
```

GNU_regex
The GNU_regex API has a richer set of functions. With GNU regex functions you can both match a string with a pattern and search a pattern in a string. The usage of these functions is somehow similar with the usage of the POSIX functions: a pattern must first be compiled with 're_compile_pattern', and the pattern buffer obtained is used to search and match. The functions used for searching and matching are 're_search' and 're_match'. In case of searching a fastmap can be used in order to optimize search. Without a fastmap the search algorithm tries to match the pattern at consecutive positions in the string. The fastmap tells the algorithm what the characters are from which a match can start. The fastmap is constructed by calling the 're_compile_fastmap'. The GNU_regex also provides the functions 're_search2' and 're_match2' for searching and matching with split data. To free the allocated fields of a pattern buffer you must use 'regfree'.

For an in-depth description of how to use these functions please consult [3].

Example:
```
string pattern = "([a-z]+).\\1";
string str = "top-topcoder|coder";

re_pattern_buffer buffer;
  char map[256];

buffer.translate = 0;
buffer.fastmap = map;
buffer.buffer = 0;
buffer.allocated = 0;

re_set_syntax(RE_SYNTAX_POSIX_EXTENDED);
const char* status = re_compile_pattern(pattern.c_str(),pattern.size(),&buffer);
if (status) {
    cout << "Error: " << status << endl;
}
re_compile_fastmap(&buffer);

struct re_registers regs;
int ofs = 0;
if (re_search(&buffer,str.c_str(),str.size(),0,str.size(),&regs)!=-1) {
  cout << "Match "
  cout << str.substr(regs.start[0],regs.end[0]-regs.start[0])
  cout << " found starting at: "
  cout << regs.start[0]
```

```
  cout << " and ending at "
  cout << regs.end[0]
  cout << endl;
} else {
  cout << "Match not found."
  cout << endl;
}
regfree(&buffer);
```

Real SRMs Examples

The following examples are all written in Java for the sake of clarity. A C++ user can use the POSIX or the GNU regex APIs to construct functions similar to those available in Java(replace_all, split, matches).

CyberLine (SRM 187 div 1, level 1)

```
import java.util.*;
public class Cyberline
{
  public String lastCyberword(String cyberline)
  {
    String[]w=cyberline.replaceAll("-","")
          .replaceAll("[^a-zA-Z0-9]"," ")
          .split(" ") ;
    return w[w.length-1];
  }
}
```

UnLinker (SRM 203 div 2, level 3)

```
import java.util.*;
public class UnLinker
{
  public String clean(String text)
  {
    String []m = text.split("((http://)?www[.]|http://)[a-zA-Z0-9.]+[.](com|org|edu|info|tv)",-
1);
    String s = m[0] ;
    for (int i = 1 ; i < m.length ; ++i)
      s = s + "OMIT" + i + m[i] ;
    return s ;
  }
}
```

CheatCode (SRM 154 div 1, level 1)

```
import java.util.*;
public class CheatCode {
  public int[] matches(String keyPresses, String[] codes) {
    boolean []map = new boolean[codes.length] ;
    int count = 0 ;
    for (int i=0;i<codes.length; ++i)
    {
      String regex = ".*" ;
      for (int j=0; j<codes[i].length(); ) {
        int k = 1;
        while ((j+k)<codes[i].length() && codes[i].charAt(j+k)==codes[i].charAt(j)) k++;
        regex = regex + codes[i].charAt(j) + "{"+k+",}";
        j+=k;
      }

      regex = regex + ".*" ;
      if (keyPresses.matches(regex))
      {
        map[i] = true ;
        count++ ;
      }
    }
    int []res = new int[count] ;
    int j=0;
    for (int i= 0 ; i < codes.length; ++i)
      if(map[i] == true)
        res[j++]=i ;
    return res ;
```

```
    }
}
```

References

1. The regex(7) linux manual page
2. The regex(3) linux manual page
3. http://docs.freebsd.org/info/regex/regex.info.Programming_with_Regex.html
4. http://www.regular-expressions.info/
5. http://java.sun.com/docs/books/tutorial/extra/regex/

It has been said that life is a school of probability. A major effect of probability theory on everyday life is in risk assessment. Let's suppose you have an exam and you are not so well prepared. There are 20 possible subjects, but you only had time to prepare for 15. If two subjects are given, what chances do you have to be familiar with both? This is an example of a simple question inspired by the world in which we live today. Life is a very complex chain of **events** and almost everything can be imagined in terms of probabilities.

Gambling has become part of our lives and it is an area in which probability theory is obviously involved. Although gambling had existed since time immemorial, it was not until the seventeenth century that the mathematical foundations finally became established. It all started with a simple question directed to Blaise Pascal by Chevalier de Mï¿½rï¿½, a nobleman that gambled frequently to increase his wealth. The question was whether a double six could be obtained on twenty-four rolls of two dice.

As far as TopCoder problems are concerned, they're inspired by reality. You are presented with many situations, and you are explained the rules of many games. While it's easy to recognize a problem that deals with probability computations, the solution may not be obvious at all. This is partly because probabilities are often overlooked for not being a common theme in programming contests. But it is not true and TopCoder has plenty of them! Knowing how to approach such problems is a big advantage in TopCoder competitions and this article is to help you prepare for this topic.

Before applying the necessary algorithms to solve these problems, you first need some mathematical understanding. The next chapter presents the basic principles of probability. If you already have some experience in this area, you might want to skip this part and go to the following chapter: Step by Step Probability Computation. After that it follows a short discussion on Randomized Algorithms and in the end there is a list with the available problems on TopCoder. This last part is probably the most important. Practice is the key!

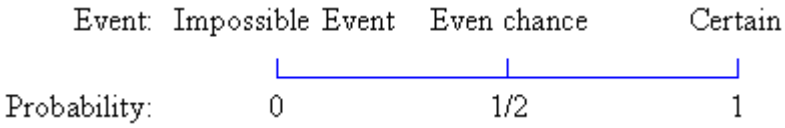
Basics

Working with probabilities is much like conducting an experiment. An **outcome** is the result of an experiment or other situation involving uncertainty. The set of all possible outcomes of a probability experiment is called a **sample space**. Each possible result of such a study is represented by one and only one point in the sample space, which is usually denoted by S. Let's consider the following experiments:

Rolling a die once
  Sample space S = {1, 2, 3, 4, 5, 6}
Tossing two coins
  Sample space S = {(Heads, Heads), (Heads, Tails), (Tails, Heads), (Tails, Tails)}

We define an **event** as any collection of outcomes of an experiment. Thus, an event is a subset of the sample space S. If we denote an event by **E**, we could say that **E⊆S**. If an event consists of a single outcome in the sample space, it is called a simple event. Events which consist of more than one outcome are called compound events.

What we are actually interested in is the probability of a certain event to occur, or **P(E)**. By definition, **P(E)** is a real number between 0 and 1, where 0 denotes the impossible event and 1 denotes the certain event (or the whole sample space).



As stated earlier, each possible outcome is represented by exactly one point in the sample space. This leads us to the following formula:

$$P(E) = \frac{|E|}{|S|}$$

That is, the probability of an event to occur is calculated by dividing the number of **favorable outcomes** (according to the event E) by the **total number of outcomes** (according to the sample space S). In order to represent the relationships among events, you can apply the known principles of set theory. Consider the experiment of rolling a die once. As we have seen previously, the sample space is S = {1, 2, 3, 4, 5, 6}. Let's now consider the following events:

Event A = 'score > 3' = {4, 5, 6}
Event B = 'score is odd' = {1, 3, 5}
Event C = 'score is 7' = Ø
AUB ='the score is > 3 or odd or both' = {1, 3, 4, 5, 6}
A∩B ='the score is > 3 and odd' = {5}
A' = 'event A does not occur' = {1, 2, 3}

We have:

P(A∪B) = 5/6
P(A∩B) = 1/6
P(A') = 1 - P(A) = 1 - 1/2 = 1/2
P(C) = 0

The first step when trying to solve a probability problem is to be able to recognize the sample space. After that, you basically have to determine the number of favorable outcomes. This is the classical approach, but the way we implement it may vary from problem to problem. Let's take a look at QuizShow (SRM 223, Div 1 - Easy). The key to solving this problem is to take into account all the possibilities, which are not too many. After a short analysis, we determine the sample space to be the following:

S = { (**wager 1** is wrong, **wager 2** is wrong, **you** are wrong),
(**wager 1** is wrong, **wager 2** is wrong, **you** are right),
(**wager 1** is wrong, **wager 2** is right, **you** are wrong),
(**wager 1** is wrong, **wager 2** is right, **you** are right),
(**wager 1** is right, **wager 2** is wrong, **you** are wrong),
(**wager 1** is right, **wager 2** is wrong, **you** are right),
(**wager 1** is right, **wager 2** is right, **you** are wrong),
(**wager 1** is right, **wager 2** is right, **you** are right) }

The problem asks you to find a wager that maximizes the number of favorable outcomes. In order to compute the number of favorable outcomes for a certain wager, we need to determine how many points the three players end with for each of the 8 possible outcomes. The idea is illustrated in the following program:

```
int wager (vector  scores, int wager1, int wager2)
{
 int best, bet, odds, wage, I, J, K;
 best = 0; bet = 0;

 for (wage = 0; wage ≤ scores[0]; wage++)
 {
  odds = 0;
  //  in 'odds' we keep the number of favorable outcomes
  for (I = -1; I ≤ 1; I = I + 2)
   for (J = -1; J ≤ 1; J = J + 2)
    for (K = -1; K ≤ 1; K = K + 2)
     if (scores[0] + I * wage > scores[1] + J * wager1  &&
      scores[0] + I * wage > scores[2] + K * wager2)    { odds++; }
  if (odds > best)  { bet = wage ; best = odds; }
  //  a better wager has been found
 }
 return bet;
}
```

Another good problem to start with is PipeCuts (SRM 233, Div 1 - Easy). This can be solved in a similar manner. There is a finite number of outcomes and all you need to do is to consider them one by one.

Let's now consider a series of n independent events: E1, E2, ... , En. Two surprisingly common questions that may appear (and many of you have already encountered) are the following:

1. What is the probability that all events will occur?
2. What is the probability that at least one event will occur?

To answer the first question, we relate to the occurrence of the first event (call it E1). If E1 does not occur, the hypothesis can no longer be fulfilled. Thus, it must be inferred that E1 occurs with a probability of P(E1). This means there is a P(E1) chance we need to check for the occurrence of the next event (call it E2). The event E2 occurs with a probability of P(E2) and we can continue this process in the same manner. Because probability is by definition a real number between 0 and 1, we can synthesize the probability that all events will occur in the following formula:

$$P(\text{all events}) = P(E1) * P(E2) * \ldots * P(En)$$

The best way to answer the second question is to first determine the probability that no event will occur and then, take the complement. We have:

P(at least one event) = 1 - P(E1') * P(E2') * ... * P(En')

These formulae are very useful and you should try to understand them well before you move.

BirthdayOdds

A good example to illustrate the probability concepts discussed earlier is the classical "Birthday Paradox". It has been shown that if there are at least 23 people in a room, there is a more than 50% chance that at least two of them will share the same birthday. While this is not a paradox in the real sense of the word, it is a mathematical truth that contradicts common intuition. The TopCoder problem asks you to find the minimum number of people in order to be more than minOdds% sure that at least two of them have the same birthday. One of the first things to notice about this problem is that it is much easier to solve the complementary problem: "What is the probability that N randomly selected people have all different birthdays?". The strategy is to start with an empty room and put people in the room one by one, comparing their birthdays with those of them already in the room:

```
int minPeople (int minOdds, int days)
{
 int nr;
 double target, p;

 target = 1 - (double) minOdds / 100;
 nr = 1;
 p = 1;

 while (p > target)
 {
  p = p * ( (double) 1 - (double) nr / days);
  nr ++;
 }

 return nr;
}
```

This so called "Birthday Paradox'' has many real world applications and one of them is described in the TopCoder problem called Collision (SRM 153, Div 1 - Medium). The algorithm is practically the same, but one has to be careful about the events that may alter the sample space.

Sometimes a probability problem can be quite tricky. As we have seen before, the 'Birthday Paradox' tends to contradict our common sense. But the formulas prove to us that the answer is indeed correct. Formulas can help, but to become a master of probabilities you need one more ingredient: "number sense" . This is partly innate ability and partly learned ability acquired through practice. Take this quiz to assess your number sense and to also become familiar with some of the common probability misconceptions.

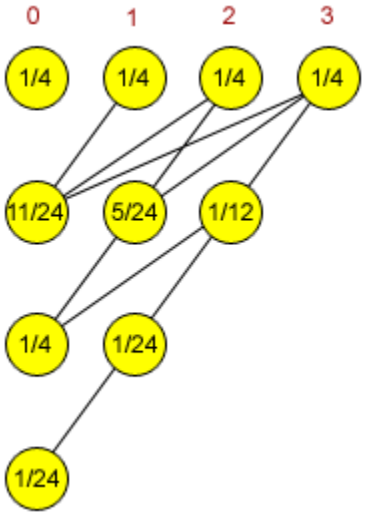
Step by Step Probability Computation

In this chapter we will discuss some real TopCoder problems in which the occurrence of an event is influenced by occurrences of previous events. We can think of it as a graph in which the nodes are events and the edges are dependencies between them. This is a somewhat forced analogy, but the way we compute the probabilities for different events is similar to the way we traverse the nodes of a graph. We start from the root, which is the initial state and has a probability of 1. Then, as we consider different scenarios, the probability is distributed accordingly.

NestedRandomness

This problem looked daunting to some people, but for those who figured it out, it was just a matter of a few lines. For the first step, it is clear what do we have to do: the function random(N) is called and it returns a random integer uniformly distributed in the range 0 to N-1. Thus, every integer in this interval has a probability of 1/N to occur. If we consider all these outcomes as input for the next step, we can determine all the outcomes of the random(random(N)) call. To understand this better, let's work out the case when N = 4.

- After the **first nesting** all integers have the same probability to occur, which is 1 / 4.
- For the **second nesting** there is a 1/4 chance for each of the following functions to be called: random(0), random(1), random(2) and random(3). Random(0) produces an error, random(1) returns 0, random (2) returns 0 or 1 (each with a probability of 1/2) and random(3) returns 0, 1 or 2.
- As a result, for the **third nesting**, random(0) has a probability of 1/4 + 1/8 + 1/12 of being called, random(1) has a probability of 1/8 + 1/12 of being called and random(2) has a probability of 1/12 of being called.
- Analogously, for the **fourth nesting**, the function random(0) has a probability of 1/4 of being called, while random(1) has a probability of 1/24.
- As for the **fifth nesting**, we can only call random(0), which produces an error. The whole process is described in the picture to the right.



NestedRandomness for N = 4

The source code for this problem is given below:

```
double probability (int N, int nestings, int target)
{
 int I, J, K;
 double A[1001], B[2001];
 // A[I] represents the probability of number I  to appear

 for (I = 0; I < N ; I++)  A[I] = (double) 1 / N;
 for (K = 2; K ≤ nestings; K++)
 {
  for (I = 0; I < N; I++)  B[I] = 0;
  // for each I between 0 and N-1 we call the function "random(I)"
  // as described in the problem statement
  for (I = 0; I < N; I++)
   for (J = 0; J < I; J++)
    B[J] +=  (double) A[I] / I;
    for (I = 0; I < N; I++)  A[I] = B[I];
 }
  return A[target];
}
```

If you got the taste for this problem, here are another five you may want to try:

ChessKnight - assign each square a probability and for every move check the squares one by one to compute the probabilities for the next move.
DiceThrows - determine the probability of each possible outcome for both players and then compare the results.
RockSkipping - the same approach, just make sure you got the lake pattern correctly.
PointSystem - represent the event space as a matrix of possible scores (x, y).
VolleyBall - similar to PointSystem, but the scores may go up pretty high.

Let's now take a look at another TopCoder problem, GeneticCrossover, which deals with **conditional probability**. Here, you are asked to predict the quality of an animal, based on the genes it inherits from its parents. Considering the problem description, there are two situations that may occur: a gene does not depend on another gene, or a gene is dependent.

For the first case, consider p the probability that the gene is to be expressed dominantly. There are only 4 cases to consider:

- at least one parent has two dominant genes. (**p** = 1)
- each parent has exactly one dominant gene. (**p** = 0.5)
- one parent has one dominant gene and the other has only recessive genes (**p** = 0.25)
- both parents have two recessive genes (**p** = 0)

Now let's take the case when a gene is dependent on another. This make things a bit trickier as the "parent" gene may also depend on another and so on ... To determine the probability that a dependent gene is dominant, we take the events that each gene in the chain (starting with the current gene) is dominant. In order for the current gene to be expressed dominantly, we need all these events to occur. To do this, we take the product of probabilities for each individual event in the chain. The algorithm works recursively. Here is the complete source code for this problem:

```
int n, d[200];
double power[200];

// here we determine the characteristic for each gene (in power[I]
// we keep the probability of gene I to be expressed dominantly)
double detchr (string p1a, string p1b, string p2a, string p2b, int nr)
{
 double p, p1, p2;
 p = p1 = p2 = 1.0;
 if (p1a[nr] ≤ 'Z')  p1 = p1 - 0.5;
 //  is a dominant gene
 if (p1b[nr] ≤ 'Z')  p1 = p1 - 0.5;
 if (p2a[nr] ≤ 'Z')  p2 = p2 - 0.5;
 if (p2b[nr] ≤ 'Z')  p2 = p2 - 0.5;
 p = 1 - p1 * p2;

 if (d[nr] != 1) power[nr] = p * detchr (p1a, p1b, p2a, p2b, d[nr]);
 // gene 'nr' is dependent on gene d[nr]
    else power[nr] = p;
 return power[nr];
}

double cross (string p1a, string p1b, string p2a, string p2b,
 vector  dom, vector  rec, vector  dependencies)
{
  int I;
 double fitness = 0.0;

 n = rec.size();
 for (I = 0; I < n; i++) d[i] = dependencies[i];
 for (I = 0 ;I < n; I++) power[i] = -1.0;
 for (I = 0; I < n; i++)
  if (power[I] == -1.0) detchr (p1a, p1b, p2a, p2b, i);
  // we check if the dominant character of gene I has
  // not already been computed
 for (I = 0; I ≤ n; I++)
  fitness=fitness+(double) power[i]*dom[i]-(double) (1-power[i])*rec[i];
  // we compute the expected 'quality' of an animal based on the
  // probabilities of each gene to be expressed dominantly

 return fitness;
}
```

See also ProbabilityTree.

Randomized Algorithms

We call randomized algorithms those algorithms that use random numbers to make decisions during their execution. Unlike deterministic algorithms that for a fixed input always give the same output and the same running-time, a randomized algorithm behaves differently from execution to execution. Basically, we distinguish two kind of randomized algorithms:

1. Monte Carlo algorithms: may sometimes produce an incorrect solution - we bound the probability of failure.
2. Las Vegas algorithms: always give the correct solution, the only variation is the running time - we study the distribution of the running time.

Read these lecture notes from the College of Engineering at UIUC for an example of how these algorithms work.

The main goal of randomized algorithms is to build faster, and perhaps simpler solutions. Being able to tackle "harder" problems is also a benefit of randomized algorithms. As a result, these algorithms have become a research topic of major interest and have already been utilized to more easily solve many different problems.

An interesting question is whether such an algorithm may become useful in TopCoder competitions. Some problems have many possible solutions, where a number of which are also optimal. The classical approach is to check them one by one, in an established order. But it cannot

be guaranteed that the optima are uniformly distributed in the solution domain. Thus, a deterministic algorithm may not find you an optimum quickly enough. The advantage of a randomized algorithm is that there are actually no rules to set about the order in which the solutions are checked and for the cases when the optima are clustered together, it usually performs much better. See QueenInterference for a TopCoder example.

Randomized algorithms are particularly useful when faced with malicious attackers who deliberately try to feed a bad input to the algorithm. Such algorithms are widely used in cryptography, but it sometimes makes sense to also use them in TopCoder competitions. It may happen that you have an efficient algorithm, but there are a few degenerate cases for which its running time is significantly slower. Assuming the algorithm is correct, it has to run fast enough for all inputs. Otherwise, all the points you earned for submitting that particular problem are lost. This is why here, on TopCoder, we are interested in **worst case execution time**.

To challenge or not to challenge?
Another fierce coding contest is now over and you have 15 minutes to look for other coders' bugs. The random call in a competitor's submission is likely to draw your attention. This will most likely fall into one of two scenarios:

1. the submission was just a desperate attempt and will most likely fail on many inputs.
2. the algorithm was tested rather thoroughly and the probability to fail (or time out) is virtually null.

The first thing you have to do is to ensure it was not already unsuccessfully challenged (check the coder's history). If it wasn't, it may deserve a closer look. Otherwise, you should ensure that you understand what's going on before even considering a challenge. Also take into account other factors such as coder rating, coder submission accuracy, submission time, number of resubmissions or impact on your ranking.

Will "random" really work?
In most optimizing problems, the ratio between the number of optimal solutions and the total number of solutions is not so obvious. An easy, but not so clever solution, is to simply try generating different samples and see how the algorithm behaves. Running such a simulation is usually pretty quick and may also give you some extra clues in how to actually solve the problem.

```
Max = 1000000; attempt = 0;
while (attempt < Max)
{
 answer = solve_random (...);
 if (better (answer, optimum))
 // we found a better solution
 {
  optimum = answer;
  cout << "Solution " << answer << " found on step " << attempt << "\n";
 }
 attempt ++;
}
```

Practice Problems
Level 1
PipeCuts - SRM 233
BirthdayOdds - SRM 174
BenfordsLaw - SRM 155
QuizShow - SRM 223

Level 2
Collision - SRM 153
ChessKnight - TCCC05 Round 1
ChipRace - SRM 199
DiceThrows - SRM 242
TopFive - SRM 243
ProbabilityTree - SRM 174
OneArmedBandit - SRM 226
RangeGame - SRM 174
YahtzeeRoll - SRM 222
BagOfDevouring - SRM 184
VolleyBall - TCO04 Round 3
RandomFA - SRM 178
PackageShipping - TCCC05 Round 3
QueenInterference - SRM 208
BaseballLineup - TCO '03 Finals

Level 3
GeneticCrossover - TCO04 Qual 3

NestedRandomness - TCCC05 Qual 5
RockSkipping - TCCC '04 Round 1
PointSystem - SRM 174
AntiMatter - SRM 179
TestScores - SRM 226
Hangman42 - SRM 229
KingOfTheCourt - SRM 222
WinningProbability - SRM 218
Disaster - TCCC05 Semi 1

Even though computers can perform literally millions of mathematical computations per second, when a problem gets large and complicated, performance can nonetheless be an important consideration. One of the most crucial aspects to how quickly a problem can be solved is how the data is stored in memory.

To illustrate this point, consider going to the local library to find a book about a specific subject matter. Most likely, you will be able to use some kind of electronic reference or, in the worst case, a card catalog, to determine the title and author of the book you want. Since the books are typically shelved by category, and within each category sorted by author's name, it is a fairly straightforward and painless process to then physically select your book from the shelves.

Now, suppose instead you came to the library in search of a particular book, but instead of organized shelves, were greeted with large garbage bags lining both sides of the room, each arbitrarily filled with books that may or may not have anything to do with one another. It would take hours, or even days, to find the book you needed, a comparative eternity. This is how software runs when data is not stored in an efficient format appropriate to the application.

Simple Data Structures
The simplest data structures are primitive variables. They hold a single value, and beyond that, are of limited use. When many related values need to be stored, an array is used. It is assumed that the reader of this article has a solid understanding of variables and arrays.

A somewhat more difficult concept, though equally primitive, are pointers. Pointers, instead of holding an actual value, simply hold a memory address that, in theory, contains some useful piece of data. Most seasoned C++ coders have a solid understanding of how to use pointers, and many of the caveats, while fledgling programmers may find themselves a bit spoiled by more modern "managed" languages which, for better or worse, handle pointers implicitly. Either way, it should suffice to know that pointers "point" somewhere in memory, and do not actually store data themselves.

A less abstract way to think about pointers is in how the human mind remembers (or cannot remember) certain things. Many times, a good engineer may not necessarily know a particular formula/constant/equation, but when asked, they could tell you exactly which reference to check.

Arrays
Arrays are a very simple data structure, and may be thought of as a list of a fixed length. Arrays are nice because of their simplicity, and are well suited for situations where the number of data items is known (or can be programmatically determined). Suppose you need a piece of code to calculate the average of several numbers. An array is a perfect data structure to hold the individual values, since they have no specific order, and the required computations do not require any special handling other than to iterate through all of the values. The other big strength of arrays is that they can be accessed randomly, by index. For instance, if you have an array containing a list of names of students seated in a classroom, where each seat is numbered 1 through n, then studentName[i] is a trivial way to read or store the name of the student in seat i.

An array might also be thought of as a pre-bound pad of paper. It has a fixed number of pages, each page holds information, and is in a predefined location that never changes.

Linked Lists
A linked list is a data structure that can hold an arbitrary number of data items, and can easily change size to add or remove items. A linked list, at its simplest, is a pointer to a data node. Each data node is then composed of data (possibly a record with several data values), and a pointer to the next node. At the end of the list, the pointer is set to null.

By nature of its design, a linked list is great for storing data when the number of items is either unknown, or subject to change. However, it provides no way to access an arbitrary item from the list, short of starting at the beginning and traversing through every node until you reach the one you want. The same is true if you want to insert a new node at a specific location. It is not difficult to see the problem of inefficiency.

A typical linked list implementation would have code that defines a node, and looks something like this:

```
class ListNode {
   String data;
   ListNode nextNode;
}
ListNode firstNode;
You could then write a method to add new nodes by inserting them at the beginning of the list:
ListNode newNode = new ListNode();
NewNode.nextNode = firstNode;
firstNode = newNode;
```

Iterating through all of the items in the list is a simple task:

```
ListNode curNode = firstNode;
while (curNode != null) {
```

```
    ProcessData(curNode);
    curNode = curNode.nextNode;
}
```

A related data structure, the doubly linked list, helps this problem somewhat. The difference from a typical linked list is that the root data structure stores a pointer to both the first and last nodes. Each individual node then has a link to both the previous and next node in the list. This creates a more flexible structure that allows travel in both directions. Even still, however, this is rather limited.

Queues

A queue is a data structure that is best described as "first in, first out". A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

Perhaps the most common use of a queue within a TopCoder problem is to implement a Breadth First Search (BFS). BFS means to first explore all states that can be reached in one step, then all states that can be reached in two steps, etc. A queue assists in implementing this solution because it stores a list of all state spaces that have been visited.

A common type of problem might be the shortest path through a maze. Starting with the point of origin, determine all possible locations that can be reached in a single step, and add them to the queue. Then, dequeue a position, and find all locations that can be reached in one more step, and enqueue those new positions. Continue this process until either a path is found, or the queue is empty (in which case there is no path). Whenever a "shortest path" or "least number of moves" is requested, there is a good chance that a BFS, using a queue, will lead to a successful solution.

Most standard libraries, such the Java API, and the .NET framework, provide a Queue class that provides these two basic interfaces for adding and removing items from a queue.

BFS type problems appear frequently on contests; on some problems, successful identification of BFS is simple and immediately, other times it is not so obvious.

A queue implementation may be as simple as an array, and a pointer to the current position within the array. For instance, if you know that you are trying to get from point A to point B on a 50x50 grid, and have determined that the direction you are facing (or any other details) are not relevant, then you know that there are no more than 2,500 "states" to visit. Thus, your queue is programmed like so:

```
class StateNode {
    int xPos;
    int yPos;
    int moveCount;
}

class MyQueue {
    StateNode[] queueData = new StateNode[2500];
    int queueFront = 0;
    int queueBack = 0;

    void Enqueue(StateNode node) {
        queueData[queueBack] = node;
        queueBack++;
    }

    StateNode Dequeue() {
        StateNode returnValue = null;
        if (queueBack > queueFront) {
        returnValue = queueData[queueFront];
        QueueFront++;
    }
    return returnValue;
    }

    boolean isNotEmpty() {
        return (queueBack > queueFront);
    }
}
```

Then, the main code of your solution looks something like this. (Note that if our queue runs out of possible states, and we still haven't reached our destination, then it must be impossible to get there, hence we return the typical "-1" value.)

```
MyQueue queue = new MyQueue();
queue.Enqueue(initialState);
while (queue.isNotEmpty()) {
    StateNode curState = queue.Dequeue();
    if (curState == destState)
```

```
return curState.moveCount;
   for (int dir = 0; dir < 3; dir++) {
      if (CanMove(curState, dir))
         queue.Enqueue(MoveState(curState, dir));
   }
}
```

Stacks

Stacks are, in a sense, the opposite of queues, in that they are described as "last in, first out". The classic example is the pile of plates at the local buffet. The workers can continue to add clean plates to the stack indefinitely, but every time, a visitor will remove from the stack the top plate, which is the last one that was added.

While it may seem that stacks are rarely implemented explicitly, a solid understanding of how they work, and how they are used implicitly, is worthwhile education. Those who have been programming for a while are intimately familiar with the way the stack is used every time a subroutine is called from within a program. Any parameters, and usually any local variables, are allocated out of space on the stack. Then, after the subroutine has finished, the local variables are removed, and the return address is "popped" from the stack, so that program execution can continue where it left off before calling the subroutine.

An understanding of what this implies becomes more important as functions call other functions, which in turn call other functions. Each function call increases the "nesting level" (the depth of function calls, if you will) of the execution, and uses increasingly more space on the stack. Of paramount importance is the case of a recursive function. When a recursive function continually calls itself, stack space is quickly used as the depth of recursion increases. Nearly every seasoned programmer has made the mistake of writing a recursive function that never properly returns, and calls itself until the system throws up an "out of stack space" type of error.

Nevertheless, all of this talk about the depth of recursion is important, because stacks, even when not used explicitly, are at the heart of a depth first search. A depth first search is typical when traversing through a tree, for instance looking for a particular node in an XML document. The stack is responsible for maintaining, in a sense, a trail of what path was taken to get to the current node, so that the program can "backtrack" (e.g. return from a recursive function call without having found the desired node) and proceed to the next adjacent node.

Soma (SRM 198) is an excellent example of a problem solved with this type of approach.

Trees

Trees are a data structure consisting of one or more data nodes. The first node is called the "root", and each node has zero or more "child nodes". The maximum number of children of a single node, and the maximum depth of children are limited in some cases by the exact type of data represented by the tree.

One of the most common examples of a tree is an XML document. The top-level document element is the root node, and each tag found within that is a child. Each of those tags may have children, and so on. At each node, the type of tag, and any attributes, constitutes the data for that node. In such a tree, the hierarchy and order of the nodes is well defined, and an important part of the data itself. Another good example of a tree is a written outline. The entire outline itself is a root node containing each of the top-level bullet points, each of which may contain one or more sub-bullets, and so on. The file storage system on most disks is also a tree structure.

Corporate structures also lend themselves well to trees. In a classical management hierarchy, a President may have one or more vice presidents, each of whom is in charge of several managers, each of whom presides over several employees.

PermissionTree (SRM 218) provides an unusual problem on a common file system.

bloggoDocStructure (SRM 214) is another good example of a problem using trees.

Binary Trees

A special type of tree is a binary tree. A binary tree also happens to be one of the most efficient ways to store and read a set of records that can be indexed by a key value in some way. The idea behind a binary tree is that each node has, at most, two children.

In the most typical implementations, the key value of the left node is less than that of its parent, and the key value of the right node is greater than that of its parent. Thus, the data stored in a binary tree is always indexed by a key value. When traversing a binary tree, it is simple to determine which child node to traverse when looking for a given key value.

One might ask why a binary tree is preferable to an array of values that has been sorted. In either case, finding a given key value (by traversing

a binary tree, or by performing a binary search on a sorted array) carries a time complexity of O(log n). However, adding a new item to a binary tree is an equally simple operation. In contrast, adding an arbitrary item to a sorted array requires some time-consuming reorganization of the existing data in order to maintain the desired ordering.

If you have ever used a field guide to attempt to identify a leaf that you find in the wild, then this is a good way to understand how data is found in a binary tree. To use a field guide, you start at the beginning, and answer a series of questions like "is the leaf jagged, or smooth?" that have only two possible answers. Based upon your answer, you are directed to another page, which asks another question, and so on. After several questions have sufficiently narrowed down the details, you are presented with the name, and perhaps some further information about your leaf. If one were the editor of such a field guide, newly cataloged species could be added to field guide in much the same manner, by traversing through the questions, and finally at the end, inserting a new question that differentiates the new leaf from any other similar leaves. In the case of a computer, the question asked at each node is simply "are you less than or greater than X?"

Priority Queues

In a typical breadth first search (BFS) algorithm, a simple queue works great for keeping track of what states have been visited. Since each new state is one more operational step than the current state, adding new locations to the end of the queue is sufficient to insure that the quickest path is found first. However, the assumption here is that each operation from one state to the next is a single step.

Let us consider another example where you are driving a car, and wish to get to your destination as quickly as possible. A typical problem statement might say that you can move one block up/down/left/right in one minute. In such a case, a simple queue-based BFS works perfectly, and is guaranteed to provide a correct result.

But what happens if we say that the car can move forward one block in two minute, but requires three minutes to make a turn and then move one block (in a direction different from how the car was originally facing)? Depending on what type of move operation we attempt, a new state is not simply one "step" from the current state, and the "in order" nature of a simple queue is lost.

This is where priority queues come in. Simply put, a priority queue accepts states, and internally stores them in a method such that it can quickly pull out the state that has the least cost. (Since, by the nature of a "shortest time/path" type of problem, we always want to explore the states of least cost first.)

A real world example of a priority queue might be waiting to board an airplane. Individuals arriving at their gate earlier will tend to sit closest to the door, so that they can get in line as soon as they are called. However, those individuals with a "gold card", or who travel first class, will always be called first, regardless of when they actually arrived.

One very simple implementation of a priority queue is just an array that searches (one by one) for the lowest cost state contained within, and appends new elements to the end. Such an implementation has a trivial time-complexity for insertions, but is painfully slow to pull objects out again.

A special type of binary tree called a heap is typically used for priority queues. In a heap, the root node is always less than (or greater than, depending on how your value of "priority" is implemented) either of its children. Furthermore, this tree is a "complete tree" from the left. A very simple definition of a complete tree is one where no branch is n + 1 levels deep until all other branches are n levels deep. Furthermore, it is always the leftmost node(s) that are filled first.

To extract a value from a heap, the root node (with the lowest cost or highest priority) is pulled. The deepest, rightmost leaf then becomes the new root node. If the new root node is larger than its left child, then the root is swapped with its left child, in order to maintain the property that the root is always less than its children. This continues as far as necessary down the left branch. Adding a value to the heap is the reverse. The new value is added as the next leaf, and swapped upward as many times as necessary to maintain the heap property.

A convenient property of trees that are complete from the left is that they can be stored very efficiently in a flat array. In general, element 0 of the array is the root, and elements k + 1 and k + 2 are the children of element k. The effect here is that adding the next leaf simply means appending to the array.

Hash Tables

Hash tables are a unique data structure, and are typically used to implement a "dictionary" interface, whereby a set of keys each has an associated value. The key is used as an index to locate the associated values. This is not unlike a classical dictionary, where someone can find a definition (value) of a given word (key).

Unfortunately, not every type of data is quite as easy to sort as a simple dictionary word, and this is where the "hash" comes into play. Hashing is the process of generating a key value (in this case, typically a 32 or 64 bit integer) from a piece of data. This hash value then becomes a basis for organizing and sorting the data. The hash value might be the first n bits of data, the last n bits of data, a modulus of the value, or in some cases, a more complicated function. Using the hash value, different "hash buckets" can be set up to store data. If the hash values are distributed evenly (which is the case for an ideal hash algorithm), then the buckets will tend to fill up evenly, and in many cases, most buckets will have no more than one or only a few objects in them. This makes the search even faster.

A hash bucket containing more than one value is known as a "collision". The exact nature of collision handling is implementation specific, and is crucial to the performance of the hash table. One of the simplest methods is to implement a structure like a linked list at the hash bucket level, so that elements with the same hash value can be chained together at the proper location. Other, more complicated schemes may involve utilizing adjacent, unused locations in the table, or re-hashing the hash value to obtain a new value. As always, there are good and bad performance considerations (regarding time, size, and complexity) with any approach.

Another good example of a hash table is the Dewey decimal system, used in many libraries. Every book is assigned a number, based upon its subject matterï¿½ the 500's are all science books, the 700's are all the arts, etc. Much like a real hash table, the speed at which a person could find a given book is based upon how well the hash buckets are evenly dividedï¿½ It will take longer to find a book about frogs in a library with many science materials than in a library consisting mostly of classical literature.

In applications development, hash tables are a convenient place to store reference data, like state abbreviations that link to full state names. In problem solving, hash tables are useful for implementing a divide-and-conquer approach to knapsack-type problems. In LongPipes, we are asked to find the minimum number of pipes needed to construct a single pipe of a given length, and we have up to 38 pieces of pipe. By dividing this into two sets of 19, and calculating all possible lengths from each set, we create hash tables linking the length of the pipe to the fewest number of segments used. Then, for each constructed pipe in one set, we can easily look up, whether or not we constructed a pipe of corresponding length in the other set, such that the two join to form a complete pipe of the desired length.

Conclusion

The larger picture to be seen from all of this is that data structures are just another set of tools that should be in the kit of a seasoned programmer. Comprehensive libraries and frameworks available with most languages nowadays preempt the need for a full understanding of how to implement each of these tools. The result is that developers are able to quickly produce quality solutions that take advantage of powerful ideas. The challenge lies in knowing which one to select.

Nonetheless, knowing a little about how these tools work should help to make the choices easier. And, when the need arises, perhaps leave the programmer better equipped to think up a new solution to a new problemï¿½ if not while on the job doing work for a client, then perhaps while contemplating the 1000 point problem 45 minutes into the coding phase of the next SRM.

Introduction

This new release of Java brings many significant improvements, not only in the APIs, but also in the language itself. Old code can still run with Java 1.5; but when writing code for this new version you must profit of the new great features that will make your code more robust, powerful and clearer. This feature intends to explain those new features so you can quickly start taking advantage of them.

Auto Boxing and Auto Unboxing
May be something like this happened to you before: you wanted to insert numbers in a list or a map, so you wrote:

```
List l = new ArrayList();
l.add(new Integer (26));
```

You need to use the Integer wrapper because the add method expects an object, and a literal integer is an int, which is not an object. Something similar happens if you have the value stored in a variable.

Then, for retrieving an element, let's say the first one, you can do:

```
int x = ((Integer) l.get(0)).intValue();
System.out.println (x);
```

Beautiful! Actually not very much, that code is relatively hard to read for what it does. This kind of situation arises very often, because primitive types are mainly used to manipulate data but they can't be stored in a collection or passed as a reference. Conversions from primitives to their respective wrappers and vice versa are very common in many Java applications. Since Java 1.5, those conversions are handled automatically by the compiler. That means, if an int value is passed where an Integer or an Object is required; the int is automatically boxed into an Integer. If an Integer is passed where an int is expected, the object is unboxed to an int.

For example, in Java 1.5 this is perfectly valid:

```
Integer value = 26;
```

And has the same effect as writing:

```
Integer value = new Integer (26);
```

For the reverse conversion (unboxing), just use the following:

```
int x = value;
```

That is equivalent to:

```
int x = value.intValue();
```

As you can see, autoboxing and autounboxing makes code look clearer, and it makes code more robust, because the compiler will never box or unbox to a wrong type.

If you box and unbox manually, this could happen:

```
Integer x = new Integer (300);
-
byte y = x.byteValue(); // wrong value: x doesn't fit in a byte
System.out.println(y);
```

This compiles but will give wrong values when x is not in the byte range, and this can be quite hard to debug. If you use Java 1.5 and you let the compiler do its work, you would just write:

```
Integer x = 300;
-
byte y = x;
System.out.println(y);
This code is still wrong, but now it doesn't even compile and you'll immediately notice the
problem. Even if the compiler cares for those conversions, you must understand that this is being
implicitly done in order to avoid doing things like:
Integer x=1;
Integer y=2;
Integer z=3;
Integer a;
a= x+y+z;
```

Even if this works the compiler will box the int values to put them in the Integer variables, then it will unbox them to sum its values, and box again to store the Integer variable a, performing much worse than if you would have used int variables. The rule of thumb is that wrappers should be used just when an object representation of a primitive type is needed.

Generics

Probably you've already used the collection classes from Java, which provides classes the means for storing data and applying algorithms on it. For example, the ArrayList class makes easy to use an array that automatically grows as you need. But you might need an array of integers, strings or a Person class you've defined. The algorithms for working on any of those kinds of data are identical, so it won't make sense to rewrite the code for each data type. In previous versions of Java, they solved this by making the collection classes take Object as their elements; and because any object inherits from that class, any instance could be passed to the collections.

For example:

```
List list = new ArrayList();
```

```
list.add("hello");

-

String s;
s = (String) list.get(0);
```
Note than when retrieving elements, very often a cast must be used because the returned type is an Object, and the stored object is probably from another class.

The main problem with this approach is that you're "loosing" the type of your objects and then doing a cast because you know beforehand what type they're. But this is very error prone, because if you do the wrong cast, the program will run but give a runtime error. Java 1.5 introduced Generics, which make this kind of code much clearer and type safe.

The above code will be written as:
```
List<String> list = new ArrayList<String>();
list.add("hello");

String s;
s = list.get(0);
```
The class List is now generic, that means that it takes parameterized types; in that case the type is String. With List<String> you're saying to the List class that this instance will work with the type String. Then, the constructor is also called with the parameter type in order to make the instance be of that type. Now the cast is removed, because the get method returns a String, as a consequence of the list declaration. That way, you work with your instance as if it were specifically designed to work with Strings, making you save casts that are risky.

Also it's possible, and sometimes very useful, to have more than one type as parameter; for example the maps needs two types, one for the key and one for the values:
```
Map<String, Integer> map = new HashMap<String, Integer>();

map.put("Bill", 40);
map.put("Jack", 35);

System.out.println(map.get("Jack"));
```
This forces the keys to be Strings and the values to be Integers.

Now, let's see how we can declare a generic class, doing a class that holds any kind of object:
```
public class Holder<T> {
    private T value;

    void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```
The parameterized types are right after the class name, enclosed between angle brackets. If using more than one, they must be comma separated, for example "<K,V>" is used for Map class. Then, we can use T almost (later we'll see the differences) as if it were a class, even if we don't know which one is it. This class must be specified when declaring instances, for example we could use the Holder class as follows:
```
Holder<String> h = new Holder<String>();
h.set("hello!");
System.out.println(h);
```
In the first line we are declaring h as an instance of Holder<String>, so we can think of T as being a String and read the Holder class doing a mentally search and replace. As you can see, the set method takes an argument of type T (in this case String), so if another class is used instead, it will generate a compilation error. The method get returns an object of type T (again String), so we don't need to cast to String when calling it. The way the compiler does that is through erasure; that means that it actually compiles generic classes as if they work with Object instances and then it applies casts and do checks for type safety. This is different as how C++ works (it generates one copy of the class for each different type instantiated) and has some consequences on its use. I said before that T should be treated *almost* as if it were a class, because there are things that you can't do, due to the erasure procedure.

For example,
```
value = new T();
```
will not compile even if it seems to be right and quite useful. The problem is that the compiler doesn't have the information of which type is T at compile time, so it can't create the instance. For the same reason, you can't create an array of T's:
```
value = new T[5];
```
will also give a compiler error. Let's make a class for handling complex numbers using generics, so it can work with different numeric types:

```
public class Complex<T> {

    private T re;
    private T im;

    public Complex(T re, T im) {
        this.re = re;
        this.im = im;
    }

    public T getReal() {
        return re;
    }

    public T getImage() {
        return im;
    }

    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

The constructor will take two variables of type T, the real and imaginary parts.

We can use that class as follows:

```
Complex<Integer> c= new Complex<Integer>(3,4);
System.out.println (c);
Getting as output: "(3, 4)"
```

Notice that Integer is used because only classes can be parameters; primitive types are not allowed. However, thanks to autoboxing we can pass int parameters to the constructor to make life easier; if not we should do:
Complex<Integer> c= new Complex<Integer>(new Integer(3), new Integer(4));
We could do some other things with the Complex class, for example:
Complex<String> c= new Complex<String>("hello","world");
But hey, this is not the idea of a complex number! We wanted to use generics so it could hold different types of numbers. Let's leave that aside for a brief moment to add a method to calculate the modulus:

```
public double getModulus() {
       return Math.sqrt(Math.pow(re, 2) + Math.pow(im, 2));
}
```

But this doesn't even compile! However this behavior seems reasonable; if it would have compiled, how would it have solved the modulus of the last complex instantiated?

We need to get a numeric value from re and im, and this could be done using doubleValue() method:

```
return Math.sqrt(Math.pow(re.doubleValue(), 2) +
Math.pow(im.doubleValue(), 2));
```

We're nearer now, but the compiler still doesn't like it-how can it know that re and im have the doubleValue method? If we pass a String, how would it solve that? The answer to all those questions is to promise the compiler that re and im will be numbers; that is, they'll be instances of classes that inherit from Number. To do that, you just have to modify the declaration of the class to be:

```
public class Complex<T extends Number> {
```

That way, you're saying that T must be a Number or a subclass. This is called a bounded type. Now, you won't be able to compile Complex<String> because String is not a Number subclass. Because Number class defines the method doubleValue() - and thus it is defined for all its subclasses - you can use this method on re and im variables, as well as you can call any method defined in Number or its superclasses.

Let's go further and define a method to compare the vector length (i.e. its modulus) with other vector:

```
public boolean isLarger(Complex<T> c) {
  return getModulus() > c.getModulus();
}
```

This can be used as follows:

```
Complex<Integer> x= new Complex<Integer>(3,4);
Complex<Integer> y = new Complex<Integer>(4,5);

System.out.println (x.isLarger(y));
```

And it works as expected. However, we might also want to do:

```
Complex<Integer> x= new Complex<Integer>(3,4);
Complex<Double> y = new Complex<Double>(4.2,5.8);

System.out.println (x.isLarger(y));
```

And this doesn't even compile, because x is a vector of Integer and it expects the same type in the isLarger method. However, we know that this is a valid operation, because even if they real and imaginary parts are of different types, we still can compare their modulus. What we need now is to use wildcards to tell the isLarger method that we really don't care for the type of complex we receive, and this is very simple to do:

```
public boolean isLarger(Complex<?> c) {
```

The interrogation sign stands for the wildcard, meaning any type. Because the Complex type is already bounded to Number, actually the type can't be other than a Number. Wildcards can also be bounded in the same way we bounded T to be a Number.

You may have noticed that at the beginning of the section I used the ArrayList class as follows:

```
new ArrayList<String>();
```

But ArrayList is an old class, at least older than Java 1.5, so how does it comes that now it takes a type? Is it in another package? Is the same class and they've broken backwards compatibility? The answer is that is the same class as in previous versions of Java; but this new version wouldn't be well received if it wouldn't be backwards compatible, so even if it can used as a generic class, omitting the type will result in a raw class, which works over the Object class as the previous Java versions. That way, the old code won't break, but new code should use the parameterized form to be type safe. When using the raw type, the Java compiler issues a warning about that. Generic classes can be inherited or can inherit from other generic or non generic classes; all the possibilities are valid.

For example, you could have another complex class that has some extra functionality and declare it as:

```
public class BetterComplex<T extends Number> extends Complex<T> {
```

Note that T should be declared at least to be bounded to Number, otherwise it won't compile because Complex is not guaranteed to have its parameter bounded to Number. It could also be bounded to a subclass of Number, but not to anything else. The parameter list could have more than the T parameter in BetterComplex; however you should always give Complex exactly one type. You can also make a non generic class inheriting from a generic class.

For example if you define some kind of Complex that you know their values are always of type Double:

```
class BetterComplex extends Complex<Double> {
```

That way, you don't even need to specify a type (actually you can't) to BetterComplex type.

The For-Each loop

A task that is done often is to iterate through a collection, and do something with each element. For example, this shows all the elements in an array:

```
String fruits[]= {"apple", "orange", "strawberry"};

      for (int i = 0; i < fruits.length; i++) {
            System.out.println (fruits[i]);
      }
We could also think of a method to show all the elements in a list:
public void Show(List l) {
     for (int i = 0; i < l.size(); i++) {
          System.out.println (l.get(i));
     }
}
```

This code seems right; however it could be very inefficient. Surprised? Try to guess why - And if not, just run the following:

```
List l = new LinkedList();
for (int i = 0; i < 10000; i++) {
      l.add("number " + i);
}
Show(l);
```

You'll have some time to think why it is inefficient while you wait for it to finish running. The reason is that a linked list doesn't have random access to its elements, so when you ask for an element, the linked list will sequentially search your element. For getting the 10000th element, the entire list will be iterated. So, the problem with that approach is that depending on the list implementation you could get an algorithm of order O(n^2) whereas an algorithm O(n) is easily obtained.

One way of making this algorithm work in O(n) independently of the list implementation is using iterators. This is the new version of the show method:

```
public static void ShowFast(List l) {
     for (Iterator it = l.iterator(); it.hasNext();) {
          System.out.println (it.next());
     }
}
```

When you ask for an iterator of the collection using the method iterator(), a reference to the beginning of the list is retrieved. Then, the hasNext() method returns whether there are still more elements to come, and it.next() does two things: it advances the reference to the next element in the collection and retrieves the current element. This gives the LinkedList the opportunity to iterate through the list in O(n). For the moment, all that can be done in earlier versions of Java. Even if the above code works, it's not nice and quite error prone. For example, if in the hurry you call again to it.next() in the for block in order to retrieve again the element, you'll get half the list in one place and half in another. Of course that this can be solved storing the value of it.next() in a local variable, but Java 1.5 brings a nicer and safer way to do the same: the for-each loop.

The method is written this way:
```
public static void ShowFastAndNice(List l) {
     for (Object o : l) {
          System.out.println (o);
     }
}
```
You can read the for sentence as "for each Object o in l". As you can see, the for keyword is still used, but now it takes the syntax:
for(type iteration_var : iterable_Object) code
The break keyword can be used in the same way as in the regular for. Observe that the type is mandatory; for example this won't compile:
```
Object o;
   for (o : l) {
         System.out.println (o);
   }
```
With the compiler forcing the iteration variable to be in the iteration block, a little flexibility is lost, because that variable can't be used after exiting the for, a practice that is common, for example, if the for is broken when something is found. However these kinds of practices are not very clear and robust, so the compiler is making you to write better code, even if it might require an additional flag. The iterable_object is any object that implements the Iterable interface, which only defines the method Iterator<T> iterator(). The type must be compatible with the type returned by the next() method of the iterator.

Arrays can also be used, so the loop of the first example in this section could be written as:
```
for (String fruit : fruits) {
    System.out.println (fruit);
}
```
To make your own types compatibles with the for-each loop, you must implement that interface and return the appropriate iterator, that often is the object itself, who is also implementing Iterator. You should use this form of for as much as you can; however there are some cases where it is not possible or straightforward:

- When you will remove some elements in the collection; in that case you need to have the iterator to call the remove method.
- When you're iterating through more than one collection at the same time, for example you have an array of names and one of values and you know that value(i) belongs to name(i)

Variable-Length Arguments
Imagine that you're programming an application that repeatedly needs to know the maximum of some integer variables. You probably will want to put the code in a method.

For example, you might do:
```
int x0, x1;
…
int z = max(x0, x1);
```
And define a method max that takes 2 arguments and returns the bigger of the two. So, you code that method and happily continue programming until you find that sometimes you need to calculate the maximum between x0, x1 and x2, and sometimes between 10 variables. One approach would be to overload the max method, defining it with different quantities of parameters, which is laborious and ugly.

A better approach is to receive an array, so your method max could be:
```
public int max(int []x) {
  int maxValue = x[0];
  for (int value : x)  {
      if (value >  maxValue) {
          maxValue = value;
      }
  }
  return maxValue;
}
```
The method uses the for-each to get the maximum element in the array. Note that if an empty array is passed, an ArrayIndexOutOfBoundsException will be thrown; you could check that case and throw another exception to explain that it can't be called with an empty array. Now that we have a nice method that takes any number of arguments, let's use it:
```
int x0, x1, x2, x3;
…
int y = max(new int[] {x0, x1});
int z = max(new int[] {x0, x1, x2, x3});
```
Even if it works, it makes the code very unclear, but with Java 1.5, you can have your cake and eat it, too! With variable-length arguments you can make the method take any number of arguments and call it in a straightforward way. The variable-length arguments are specified using three dots before the variable name.
br> For example, the above method will be written now as:

```
public int max(int ...x) {
  int maxValue = x[0];
  for (int value : x)  {
      if (value >  maxValue) {
          maxValue = value;
      }
  }
  return maxValue;
}
```

The only difference is in the signature; observe that x is treated as a normal array. The advantage is when you have to call that method. Now you can use:

```
int y = max(x0, x1);
int z = max(x0, x1, x2, x3);
int error = max(); // it will throw an exception
```

The compiler will internally do something very similar as we did before: create an array with all the values and pass it to the method that will receive them as an array.

You're still able to call the method with an array of integers, which can be useful if you need to build it programmatically, or if you want to upgrade your methods to variable arguments without breaking compatibility.

When using variable-length arguments, zero arguments are permitted, which sometimes is an advantage, and others, as in that case, is something that you don't desire.

You might be tempted to write the method declaration in the following way:

```
public int max(int y, int ...x) {
// get the maximum between y and the elements of the array x
}
```

This will force the user to pass at least one parameter. Doing this brings a complication: if you want to pass an array, you have to pass its first element as the first parameter and the rest as the second parameter, and that code will look really bad. Normal and variable-length parameters can be combined, but not any combination is valid. The variable-length argument must be the last parameter, which implies that it's not possible to have more than one variable-length argument in a method declaration. Sometimes there can be ambiguity between overloaded methods that take variable-length arguments.

For example, consider the following declarations:

```
public int method(int ...x) {
// do something
}

public int method(String ...x) {
// do something
}
```

I'm sure you can deduct which method will be called in those statements:

```
method(3);
method("3");
method(1, 2, 3);
method("hello", "world");
```

But, you can't know which method will be called if I do:

```
method();
```

Both method declarations match, so you can't know, and neither the compiler does, so the above statement will not compile. You'll have an error message saying: "reference to method is ambiguous,". The ambiguity also happens if you declare:

```
public int method(int ...x) {
// do something
}

public int method(int y, int...x) {
// do something
}
```

And try to call the method with one or more arguments; those declarations will probably be useless. Variable-length arguments make code look clearer in many common situations. However, there are situations where they can't help: for example a method that takes names and values for the names, could be written as:

```
public void multiple(String names[], int values[]) {
// do something with names and values.
}
```

But it can't be translated to variable-length arguments, because this won't compile:

```
public void multiple(String -names, int -values) {
   /-
}
```

There's no way to use variable-length arguments in this example, but don't worry; just use the arrays in the old way.

Enumerations

Sometimes you need to manage information that doesn't fit well in the existent types. Imagine for example a method that needs to receive which programming language a coder uses. The method could be:

```
public void doSomething(String handle, String language) {
   if (language.equals("Java")) {
       System.out.println ("Hope you already know 1.5!");
   }
   -
}
```

And then, it will be called using, for example

```
doSomething("abc","Java")
```

This has lots of problems: you could misspell the name, you might not know all the available names, you could pass an inexistent language and so on. A better approach is to use constants, for example you could write the following code:

```
public class Languages {
    public static final int JAVA = 1;
    public static final int CPP = 2;
    public static final int CSHARP = 3;
    public static final int VBNET = 4;
}
```

Then, your method doSomething will be:

```
public void doSomething(String handle, int language) {
   if (language == Languages.Java) {
       System.out.println ("Hope you already know 1.5!");
   }
   -
}
```

And your call will be:

```
doSomething("abc",Languages.JAVA)
```

This is better; at least you're sure that any misspelling is caught at compile time; however if you do:

```
doSomething("abc", 6)
```

You're giving the method a code for a language that doesn't exist. Java 1.5 introduced enumerations, that is something that exist in many older languages; however Java 1.5 has enumerations far more powerful than the ones found on those languages. Let's first see how the above example will be done.

We define the enumeration type as following:

```
enum Language { JAVA, CPP, CHSARP, VBNET }
```

Now we have a new type called Language and we can define variables of that type, and assign any of the declared values:

Language lang;

```
lang = Language.JAVA;
```

Note that Lanuage.JAVA is like an instance of Language, which is similar to a class, but can't be instantiated. The method will be coded now as:

```
public void doSomething(String handle, Language language) {
    if (language== Language.JAVA) {
        System.out.println ("Hope you already know 1.5!");
    }
}
```

And the call will be:

```
doSomething("abc", Language.JAVA)
```

Now, it's impossible to pass anything else than the defined values, making the method safer. But Java doesn't stop here. While in some languages the enumerations are a bit more than a collection of integer constants, in Java many nice additional features are provided.

Try this one:

```
System.out.println (Language.JAVA);
```

This will print "JAVA", so that you don't have to do a map from the enumeration values to its names; Java already stores the name of the constant. You can also know all the available values for a type using values() method, that returns an array of the enumeration type, and can be used as follows:

```
System.out.println ("Available Languages:");
for(Language l : Language.values())
   System.out.println(l);
```

This will print the list of the four available languages. Now, imagine that the user is asked for the language he's willing to use, this could be done using a "combo box" where the values are retrieved as above, so that if new languages are added, they automatically appear in the combo box as soon as you add the constant in the enumeration. If you want to call the method doSomething, you'll notice that a Language constant is needed, but we have a string constant from the combo box (or may be some free text input). To translate it, you have a very easy way: the method valueOf(String str) that returns the enumeration constant corresponding to that name, or throws an IllegalArgumentException if not found. You could do:

```
doSomething("abc", Language.valueOf(selectedLang));
```
where selectedLang is the String variable that holds the user selection. Now, imagine that our method doSomething needs to know the extension of the file for saving it. We could do:
```
String extension;
switch (language) {
  case JAVA: extension = ".java"; break;
  case CPP: extension = ".cpp"; break;
  case CSHARP: extension = ".cs"; break;
  case VBNET: extension = ".vb"; break;
  default: throw new IllegalArgumentException
             ("don't know that language extension");
}
// do something using extension variable
```
This example shows that the switch sentence can be used with enumerations. Notice that the languages in the switch are referred without specifying the enumeration type. If you try to compile the above code without the default in the switch, you'll got an error saying that extension might not be initialized. But you're sure it is!! language variable is bounded to be one of the four defined constants, so there's no way to escape! You might be tempted to just initialize the extension to an empty string or null to shout the compiler's mouth, and even if it will perfectly work for the moment, is not the best for being extensible. If other language is added and you forget to add the case statement, the extension variable will be null or empty string, and the error might be much more complicated to find than if you throw an exception as we did. But this solution is still not the best; as we've been talking, you could easily forget to declare the case statement, and you might have many "maps" as the above in different parts of code, making it hard to maintain. Because the extension is linked to the language itself, why not store the extension itself on the constant? An enumeration is actually a special kind of class, and it can have methods and attributes, so you can do:
```
enum Language {
        JAVA   (".java"),
        CPP    (".cpp"),
        CSHARP (".cs"),
        VBNET  (".vb");

        private String extension;

        Language(String ext) {
            extension = ext;
        }

        public String getExtension() {
            return extension;
        }
}
```
Now the constant definition must match the only available constructor. The constructor can't be declared as public, you'll get a compile error because enums can't be instantiated. You could overload the constructor to provide many ways to initialize the constants, in the same way you would do in a normal class. Now, our method doSomething will simply do:
```
String extension = language.getExtension();
// do something using extension variable,
// that actually is not even needed now.
```
The code is more robust now, because when adding a new language is impossible to forget to initialize the extension; you won't be able to compile if you don't specify it. And as a plus, you are declaring the extension in the same place as the language, so you don't need to search all the code for switch'es to add the extension.

In conclusion, Java provides a new kind of enumerations that is very powerful because the constants are more like a singleton instance than an integer variable, so more information can be stored in the constant; and even singletons won't be so good to use as constants because enums provide extra functionality like listing all the constants for a type and mapping from a constant to its string representation and vice versa.

Annotations

Annotations, or metadata, introduce a new dimension in your code. They enable you to embed information in your source code. That information does nothing by itself; the idea is that another program can use it. For example, when writing unit tests, the JUnit framework needs to know which methods are you willing to test; and it does that using reflection to get all the methods whose name starts with "test", as well as to look up for methods "setUp", "tearDown" and so on. Annotations can help here, marking that a method is a test method, that an exception is expected and so on. Actually, it's quite probable that in a near future JUnit will implement this, because NUnit is already using the analogous idea in .Net platform. Now, a method that tests if a constructor throws NullPointerException looks like this:
```
public void testNullConstructor () {
   try {
       new TestedClass(null);
   } catch (NullPointerException e) {
   }
}
```
Using annotations, it could look like:

```
@TestMethod
@ExpectedException(NullPointerException.class)
   public void nullConstructor () {
   new TestedClass();
}
```

As you can see, there are two annotations before the method that are recognized by their starting "@". The name of the first annotation is TestMethod and it doesn't take any parameters. The second annotation (ExpectedException) takes one parameter. Now thay you understand what they can be used for, let's see how they're declared and accessed.

An annotation is based on an interface, but it can just have method declarations that take no parameters and the return type must be one of the primitive types, String, Class, an enum type, an annotation type, or an array of one of the preceding types. For example, this is an annotation declaration:

```
@interface Note {
   String author();
   String note();
   int version();
}
```

This annotation will be called, for example, in the following way:
@Note(author="TCSDEVELOPER", note="Working fine", version=3)
You might want some of the fields to take default values, so you don't have to specify them each time. This can be easily done using:

```
@interface Note {
   String author() default "TCSDEVELOPER";
   String note();
   int version();
}
```

And then, if you don't specify the author, the default will be used:
@Note(note="Working fine", version=1) // "TCSDEVELOPER" is used
If your annotation just takes one parameter, you can omit its name when using it, however for this purpouse, the name of the field must be "value":

```
@interface ExpectedException {
    Class value() default String.class;
}
```

With this declaration, the annotation can be used as shown in a previous example, saving some typing:
@ExpectedException(NullPointerException.class)
Also annotations without parameters can be declared, and they're very useful to mark something, for example that a method must be tested. The annotations we've defined can be used in fields, methods, classes, etc. Often you want the annotation to be used just in some of those targets. This is done using the @Target annotation on our annotation:

```
@Target(ElementType.METHOD)
@interface Note {
   String author() default "TCSDEVELOPER";
   String note();
   int version();
}
```

Now, the Note annotation can be used just for methods. You can also specify more than one element type using the following syntax:
@Target({ElementType.TYPE, ElementType.METHOD})
Annotations can have different Retention Policies, which specify when the annotation is discarded. The SOURCE retention policy makes the annotation just available in the source code, discarting it on the compilation. This could be used in a similar way as javadocs are used: a tool that extracts information from source code to do something. The CLASS retention police, used by default, stores the annotation in the .class file but it won't be available in the JVM. The last retention policy is RUNTIME, that makes the annotation to be available also in the JVM. The retention policy is specified using the @Retention annotation.

For example:
@Retention(RetentionPolicy.RUNTIME)
We've learned to declare and use annotations; however they do nothing by themselves. Now, we need to access them to do something, and this is done using reflection. The AnnotatedElement interface represents an element that can be annoted, and thus it declares the methods we need to retrieve the annotations. The reflection classes (Class, Constructor, Field, Method, Package) implement this interface.

For example, this is a sample program that gets the Note annotations for the AnnotDemo class and shows them:

```
@Note(note="Finish this class!", version=1)
public class AnnotDemo {
    public static void main (String args[]) {
      Note note = AnnotDemo.class.getAnnotation(Note.class);
      if (note != null) {
         System.out.println("Author=" + note.author());
         System.out.println("Note=" + note.note());
         System.out.println("Version=" + note.version());
```

```
      } else {
         System.out.println("Note not found.");
      }
   }
}
```

If you run this and it says that the note was not found don't panic. Try to discover what the problem is. Did you find it? If the Note annotation doesn't have a retention policy specified, it uses CLASS by default, so the annotations are not available in the JVM. Just specify the RUNTIME retention police and it will work. The important line in the above code is:
Note note = AnnotDemo.class.getAnnotation(Note.class);
With AnnotDemo.class we get a Class object representing that class, and the getAnnotation method asks for an annotation of the type specified in the argument. That method retrieves the note if it exists or null if there are no notes of such type. The AnnotatedElement interface provides more methods in order to retrieve all the annotations for an element (without specifying the type) and to know if an annotation is present. For more information refer to the API documentation of this interface.

- Java defines seven built in annotations.
- In java.lang.annotation: @Retention, @Documented, @Target, @Inherited
- In java.lang: @Override, @Deprecated, @SuppressWarnings

Some of them were already explained on this feature. For the rest, please see the API documentation.

If you want to go deeper in this interesting subject, I recommend you to start by doing a mini-Junit class with annotations. You have to look for a @Test annotation in each method of the class, and if found, execute that method. Then, you can improve it by using the @ExpectedException as we defined above.

Static Import
Let's do some math:

```
x = Math.cos(d * Math.PI) * w + Math.sin(Math.sqrt(Math.abs(d))) * h;
```

This looks horrible! Not only is the formula hard to understand, but also the "Math." takes a lot of space and doesn't help to understand. It will be much clearer if you could do:
x = cos(d * PI) * w + sin (sqrt(abs(d))) * h;
Well, with Java 1.5 this is possible. You just have to do a static import over the members you're using, so you must add at the beginning of the file:

```
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static java.lang.Math.abs;
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

Those lines make the static methods and constants visible without the need of the class name. You could also do:

```
import static java.lang.Math.*;
```

But this is not recommended; almost sure you're importing a lot of things that you don't need and you don't event know that you're importing. Also, when reading the code, if you used ".*" in many static import's, you won't know from which one an attribute or method is coming. Also this could result in ambiguities when there is a static member with the same name in two different places. You should use static imports with care, just when they'll make your code look clearer because you're repeatedly accessing some static members of a class. Abusing of the import static recourse will make the code less clear.

Updates in the API
A lot of updates were done in the API, adding methods, classes and packages. The new API definition is backwards compatible, so your previous code must compile without any modifications. We'll briefly see some of the most important changes. If you want to know the full list of updates, please see http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#base_libs

Formatted I/O
The new class java.util.Formatter enables you to use formatted strings in the old and loved C style. For example, you can do:
Formatter f = new Formatter ();

```
f.format("Decimal: %d, hexa: %x, character %c, double %1.4f.",
50,50,'A', 3.1415926);
System.out.println (f.toString());
```

The constructor of Formatter can be used in different ways so that the buffer for the formatted output is somewhere else, for example in a file. If you want to show the input on the screen, instead of the above form, you can do it directly with printf:

```
System.out.printf("Decimal: %d, hexa: %x, character %c,
double %1.4f.", 50,50,'A', 3.1415926);
```

Although this is similar to the formatting strings in C, there are some differences and improvements, so you should have a look at the Formatter class documentation for more detail. On the other side, you can read formatted input and convert to its binary form using the java.util.Scanner. For example, if you have any strings containing an unknown number of integers, you can sum them using:

```
int x=0;
```

```
Scanner scan = new Scanner("10 5 3 1");
while (scan.hasNextInt()) {
   x+=scan.nextInt();
}
```

When you create the Scanner object, you must specify a string to read from, or some other source like an InputStream. Then, you have methods to determinate if you have an element of a specified type using hasNext*, and you can read those elements with the methods next*. Of course that if you know beforehand the number of each type, you don't need the hasNext* methods. Let's see another example, where we know that we have a string and two doubles, separated either by ";" or "&":

```
Scanner scan = new Scanner("Hello World;15.4&8.4");
scan.useLocale(Locale.US);
scan.useDelimiter("[;&]");
String s;
double x,y;
s = scan.next();
x = scan.nextDouble();
y = scan.nextDouble();
```

Here, we set the locale to be US, so that the decimal separator is the dot. You have to be very careful with locales, because this might work perfect in your computer but in someone else's it might throw an exception because his locale is set to use the comma as decimal separator. Then, we set the delimiter with the method useDelimiter, that takes a regular expression pattern. As we know that we'll have a string and two doubles, we just read them from the scanner.

Collections

The Collections package has changed from its roots, by supporting generic types. The old code will compile and work, but it will produce compiler warnings. New code should be written using generics.

For example, here we create a list of Integers and then show it:

```
List<Integer> l = new ArrayList<Integer>();
l.add(10);
l.add(20);

for (int value : l) {
   System.out.println(value);
}
```

Notice that we can add ints directly thanks to autoboxing, but the type of the collection is always a class. Some other collection classes, like HashMap, take two types. See the generics section for an example. Also, new class collections are defined: enumMap and enumSet to make working with maps keys and sets in enums more efficient; AbstractQueue that provides some basic common functionality of the queues, and PriorityQueue, which can be very useful in SRMs.

Let's see an example of a PriorityQueue:

```
PriorityQueue<Double> pq = new PriorityQueue<Double>();
pq.add(4.5);
pq.add(9.3);
pq.add(1.7);

while(true) {
   Double d = pq.poll();
   if (d == null) break;
   System.out.println(d);
}
```

Adding elements it's done - not very surprisingly - with the add method. Then, the poll method retrieves the next element in the priority queue or null if it's empty, and removes it. You can also use the peek method, which works like poll but doesn't remove the element. There are also new methods in the Collections utility class. Two of them can be particularly useful in SRM's: the frequency method that counts the number of times that an element appears in a collection, and the disjoint method that returns whether two collections have no common elements. The other two added methods are addAll, which enables you to add all the elements in an array to a collection, and reverseOrder, that returns a Comparator representing the reverse order of the comparator provided as parameter.

Introduction

Any number of practical applications in computing require things to be in order. Even before we start computing, the importance of sorting is drilled into us. From group pictures that require the tallest people to stand in the back, to the highest grossing salesman getting the largest Christmas bonus, the need to put things smallest to largest or first to last cannot be underestimated.

When we query a database, and append an ORDER BY clause, we are sorting. When we look for an entry in the phone book, we are dealing with a list that has already been sorted. (And imagine if it weren't!) If you need to search through an array efficiently using a binary search, it is necessary to first sort the array. When a problem statement dictates that in the case of a tie we should return the lexicographically first result, well… you get the idea.

General Considerations

Imagine taking a group of people, giving them each a deck of cards that has been shuffled, and requesting that they sort the cards in ascending rank order. Some people might start making piles, others might spread the cards all over a table, and still others might juggle the cards around in their hands. For some, the exercise might take a matter of seconds, for others several minutes or longer. Some might end up with a deck of cards where spades always appear before hearts, in other cases it might be less organized. Fundamentally, these are all the big bullet points that lead algorithmists to debate the pros and cons of various sorting algorithms.

When comparing various sorting algorithms, there are several things to consider. The first is usually runtime. When dealing with increasingly large sets of data, inefficient sorting algorithms can become too slow for practical use within an application.

A second consideration is memory space. Faster algorithms that require recursive calls typically involve creating copies of the data to be sorted. In some environments where memory space may be at a premium (such as an embedded system) certain algorithms may be impractical. In other cases, it may be possible to modify the algorithm to work "in place", without creating copies of the data. However, this modification may also come at the cost of some of the performance advantage.

A third consideration is stability. Stability, simply defined, is what happens to elements that are comparatively the same. In a stable sort, those elements whose comparison key is the same will remain in the same relative order after sorting as they were before sorting. In an unstable sort, no guarantee is made as to the relative output order of those elements whose sort key is the same.

Bubble Sort

One of the first sorting algorithms that is taught to students is bubble sort. While it is not fast enough in practice for all but the smallest data sets, it does serve the purpose of showing how a sorting algorithm works. Typically, it looks something like this:

```
for (int i = 0; i < data.Length; i++)
   for (int j = 0; j < data.Length - 1; j++)
      if (data[j] > data[j + 1])
      {
         tmp = data[j];
         data[j] = data[j + 1];
         data[j + 1] = tmp;
      }
```

The idea is to pass through the data from one end to the other, and swap two adjacent elements whenever the first is greater than the last. Thus, the smallest elements will "bubble" to the surface. This is $O(n^2)$ runtime, and hence is very slow for large data sets. The single best advantage of a bubble sort, however, is that it is very simple to understand and code from memory. Additionally, it is a stable sort that requires no additional memory, since all swaps are made in place.

Insertion Sort

Insertion sort is an algorithm that seeks to sort a list one element at a time. With each iteration, it takes the next element waiting to be sorted, and adds it, in proper location, to those elements that have already been sorted.

```
for (int i = 0; i <= data.Length; i++) {
   int j = i;
   while (j > 0 && data[i] < data[j - 1])
      j--;
   int tmp = data[i];
   for (int k = i; k > j; k--)
      data[k] = data[k - 1];
   data[j] = tmp;
}
```

The data, as it is processed on each run of the outer loop, might look like this:

{**18**, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{**6**, 18, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, **9**, 18, 1, 4, 15, 12, 5, 6, 7, 11}
{**1**, 6, 9, 18, 4, 15, 12, 5, 6, 7, 11}
{1, **4**, 6, 9, 18, 15, 12, 5, 6, 7, 11}
{1, 4, 6, 9, **15**, 18, 12, 5, 6, 7, 11}

{1, 4, 6, 9, **12**, 15, 18, 5, 6, 7, 11}
{1, 4, **5**, 6, 9, 12, 15, 18, 6, 7, 11}
{1, 4, 5, 6, **6**, 9, 12, 15, 18, 7, 11}
{1, 4, 5, 6, 6, **7**, 9, 12, 15, 18, 11}
{1, 4, 5, 6, 6, 7, 9, **11**, 12, 15, 18}
One of the principal advantages of the insertion sort is that it works very efficiently for lists that are nearly sorted initially. Furthermore, it can also work on data sets that are constantly being added to. For instance, if one wanted to maintain a sorted list of the highest scores achieved in a game, an insertion sort would work well, since new elements would be added to the data as the game was played.

Merge Sort
A merge sort works recursively. First it divides a data set in half, and sorts each half separately. Next, the first elements from each of the two lists are compared. The lesser element is then removed from its list and added to the final result list.

```
int[] mergeSort (int[] data) {
   if (data.Length == 1)
      return data;
   int middle = data.Length / 2;
   int[] left = mergeSort(subArray(data, 0, middle - 1));
   int[] right = mergeSort(subArray(data, middle, data.Length - 1));
   int[] result = new int[data.Length];
   int dPtr = 0;
   int lPtr = 0;
   int rPtr = 0;
   while (dPtr < data.Length) {
      if (lPtr == left.Length) {
         result[dPtr] = right[rPtr];
         rPtr++;
      } else if (rPtr == right.Length) {
         result[dPtr] = left[lPtr];
         lPtr++;
      } else if (left[lPtr] < right[rPtr]) {
         result[dPtr] = left[lPtr];
         lPtr++;
      } else {
         result[dPtr] = right[rPtr];
         rPtr++;
      }
      dPtr++;
   }
   return result;
}
```

Each recursive call has O(*n*) runtime, and a total of O(log *n*) recursions are required, thus the runtime of this algorithm is O(*n* * log *n*). A merge sort can also be modified for performance on lists that are nearly sorted to begin with. After sorting each half of the data, if the highest element in one list is less than the lowest element in the other half, then the merge step is unnecessary. (The Java API implements this particular optimization, for instance.) The data, as the process is called recursively, might look like this:
{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{18, 6, 9, 1, 4} {15, 12, 5, 6, 7, 11}
{18, 6} {9, 1, 4} {15, 12, 5} {6, 7, 11}
{18} {6} {9} {1, 4} {15} {12, 5} {6} {7, 11}
{18} {6} {9} {1} {4} {15} {12} {5} {6} {7} {11}
{18} {6} {9} {1, 4} {15} {5, 12} {6} {7, 11}
{6, 18} {1, 4, 9} {5, 12, 15} {6, 7, 11}
{1, 4, 6, 9, 18} {5, 6, 7, 11, 12, 15}
{1, 4, 5, 6, 6, 7, 9, 11, 12, 15, 18}
Apart from being fairly efficient, a merge sort has the advantage that it can be used to solve other problems, such as determining how "unsorted" a given list is.

Heap Sort
In a heap sort, we create a heap data structure. A heap is a data structure that stores data in a tree such that the smallest (or largest) element is always the root node. (Heaps, also known as priority queues, were discussed in more detail in Data Structures.) To perform a heap sort, all data from a list is inserted into a heap, and then the root element is repeatedly removed and stored back into the list. Since the root element is always the smallest element, the result is a sorted list. If you already have a Heap implementation available or you utilize the Java PriorityQueue (newly available in version 1.5), performing a heap sort is fairly short to code:

```
Heap h = new Heap();
for (int i = 0; i < data.Length; i++)
   h.Add(data[i]);
int[] result = new int[data.Length];
```

```
for (int i = 0; i < data.Length; i++)
   data[i] = h.RemoveLowest();
```

The runtime of a heap sort has an upper bound of O(*n* * log *n*). Additionally, its requirement for storage space is only that of storing the heap; this size is linear in proportion to the size of the list. Heap sort has the disadvantage of not being stable, and is somewhat more complicated to understand beyond just the basic implementation.

Quick Sort

A quick sort, as the name implies, is intended to be an efficient sorting algorithm. The theory behind it is to sort a list in a way very similar to how a human might do it. First, divide the data into two groups of "high" values and "low" values. Then, recursively process the two halves. Finally, reassemble the now sorted list.

```
Array quickSort(Array data) {
   if (Array.Length <= 1)
      return;
   middle = Array[Array.Length / 2];
   Array left = new Array();
   Array right = new Array();
   for (int i = 0; i < Array.Length; i++)
      if (i != Array.Length / 2) {
         if (Array[i] <= middle)
            left.Add(Array[i]);
         else
            right.Add(Array[i]);
      }
   return concatenate(quickSort(left), middle, quickSort(right));
}
```

The challenge of a quick sort is to determine a reasonable "midpoint" value for dividing the data into two groups. The efficiency of the algorithm is entirely dependent upon how successfully an accurate midpoint value is selected. In a best case, the runtime is O(*n* * log *n*). In the worst case-where one of the two groups always has only a single element-the runtime drops to O(n^2). The actual sorting of the elements might work out to look something like this:

{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, 9, 1, 4, 12, 5, 6, 7, 11} {15} {18}
{6, 9, 1, 4, 5, 6, 7, 11} {12} {15} {18}
{1, 4} {5} {6, 9, 6, 7, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {9, 7, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {7} {9, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {7} {9} {11} {12} {15} {18}

If it is known that the data to be sorted all fit within a given range, or fit a certain distribution model, this knowledge can be used to improve the efficiency of the algorithm by choosing midpoint values that are likely to divide the data in half as close to evenly as possible. A generic algorithm that is designed to work without respect to data types or value ranges may simply select a value from the unsorted list, or use some random method to determine a midpoint.

Radix Sort

The radix sort was designed originally to sort data without having to directly compare elements to each other. A radix sort first takes the least-significant digit (or several digits, or bits), and places the values into buckets. If we took 4 bits at a time, we would need 16 buckets. We then put the list back together, and have a resulting list that is sorted by the least significant radix. We then do the same process, this time using the second-least significant radix. We lather, rinse, and repeat, until we get to the most significant radix, at which point the final result is a properly sorted list.

For example, let's look at a list of numbers and do a radix sort using a 1-bit radix. Notice that it takes us 4 steps to get our final result, and that on each step we setup exactly two buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, 4, 12, 6} {9, 1, 15, 5, 7, 11}
{4, 12, 9, 1, 5} {6, 6, 15, 7, 11}
{9, 1, 11} {4, 12, 5, 6, 6, 15, 7}
{1, 4, 5, 6, 6, 7} {9, 11, 12, 15}

Let's do the same thing, but now using a 2-bit radix. Notice that it will only take us two steps to get our result, but each step requires setting up 4 buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{4, 12} {9, 1, 5} {6, 6} {15, 7, 11}
{1} {4, 5, 6, 6, 7} {9, 11} {12, 15}

Given the relatively small scope of our example, we could use a 4-bit radix and sort our list in a single step with 16 buckets:

{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{1} {} {} {4} {5} {6, 6} {7} {} {9} {} {11} {12} {} {} {15}

Notice, however, in the last example, that we have several empty buckets. This illustrates the point that, on a much larger scale, there is an obvious ceiling to how much we can increase the size of our radix before we start to push the limits of available memory. The processing time

to reassemble a large number of buckets back into a single list would also become an important consideration at some point.

Because radix sort is qualitatively different than comparison sorting, it is able to perform at greater efficiency in many cases. The runtime is O(*n* * *k*), where *k* is the size of the key. (32-bit integers, taken 4 bits at a time, would have *k* = 8.) The primary disadvantage is that some types of data may use very long keys (strings, for instance), or may not easily lend itself to a representation that can be processed from least significant to most-significant. (Negative floating-point values are the most commonly cited example.)

Sorting Libraries

Nowadays, most programming platforms include runtime libraries that provide a number of useful and reusable functions for us. The .NET framework, Java API, and C++ STL all provide some built-in sorting capabilities. Even better, the basic premise behind how they work is similar from one language to the next.

For standard data types such as scalars, floats, and strings, everything needed to sort an array is already present in the standard libraries. But what if we have custom data types that require more complicated comparison logic? Fortunately, object-oriented programming provides the ability for the standard libraries to solve this as well.

In both Java and C# (and VB for that matter), there is an interface called Comparable (IComparable in .NET). By implementing the IComparable interface on a user-defined class, you add a method int CompareTo (object other), which returns a negative value if less than, 0 if equal to, or a positive value if greater than the parameter. The library sort functions will then work on arrays of your new data type.

Additionally, there is another interface called Comparator (IComparer in .NET), which defines a single method int Compare (object obj1, object obj2), which returns a value indicating the results of comparing the two parameters.

The greatest joy of using the sorting functions provided by the libraries is that it saves a lot of coding time, and requires a lot less thought and effort. However, even with the heavy lifting already completed, it is still nice to know how things work under the hood.

Introduction

This article covers a problem that often arises in real life situations and, as expected, in programming contests, with Top Coder being no exception. It is addressed mostly to coders who are not familiar with the subject, but it may prove useful to the more experienced as well. Lots of papers have been written, and there are many algorithms known to solve this problem. While they are not the fastest, the algorithms presented here have the advantage of being simple and efficient, and because of this they are usually preferred during a contest setup. The reader is advised to read the article on graph theory first, as the concepts presented there are needed to understand those presented here.

The Standard Maximum Flow Problem

So, what are we being asked for in a max-flow problem? The simplest form that the statement could take would be something along the lines of: "A list of pipes is given, with different flow-capacities. These pipes are connected at their endpoints. What is the maximum amount of water that you can route from a given starting point to a given ending point?" or equivalently "A company owns a factory located in city X where products are manufactured that need to be transported to the distribution center in city Y. You are given the one-way roads that connect pairs of cities in the country, and the maximum number of trucks that can drive along each road. What is the maximum number of trucks that the company can send to the distribution center?"

A first observation is that it makes no sense to send a truck to any other city than Y, so every truck that enters a city other than Y must also leave it. A second thing to notice is that, because of this, the number of trucks leaving X is equal to the number of trucks arriving in Y.

Rephrasing the statement in terms of graph theory, we are given a network - a directed graph, in which every edge has a certain capacity c associated with it, a starting vertex (the source, X in the example above), and an ending vertex (the sink). We are asked to associate another value f satisfying $f \leq c$ for each edge such that for every vertex other than the source and the sink, the sum of the values associated to the edges that enter it must equal the sum of the values associated to the edges that leave it. We will call f the flow along that edge. Furthermore, we are asked to maximize the sum of the values associated to the arcs leaving the source, which is the total flow in the network.

The image below shows the optimal solution to an instance of this problem, each edge being labeled with the values f/c associated to it.

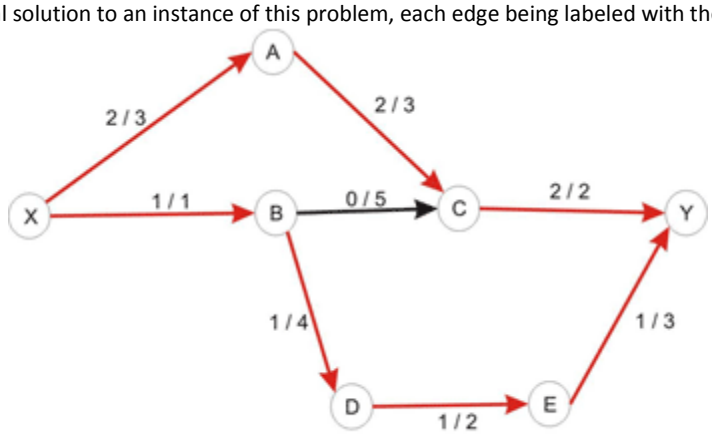


Figure 1a - Maximum Flow in a network

How to Solve It

Now how do we actually solve the problem? First, let us define two basic concepts for understanding flow networks: residual networks and augmenting paths. Consider an arbitrary flow in a network. The residual network has the same vertices as the original network, and one or two edges for each edge in the original. More specifically, if the flow along the edge x-y is less than the capacity there is a forward edge x-y with a capacity equal to the difference between the capacity and the flow (this is called the residual capacity), and if the flow is positive there is a backward edge y-x with a capacity equal to the flow on x-y. An augmenting path is simply a path from the source to the sink in the residual network, whose purpose is to increase the flow in the original one. It is important to understand that the edges in this path can point the "wrong way" according to the original network. The path capacity of a path is the minimum capacity of an edge along that path. Let's take the following example:

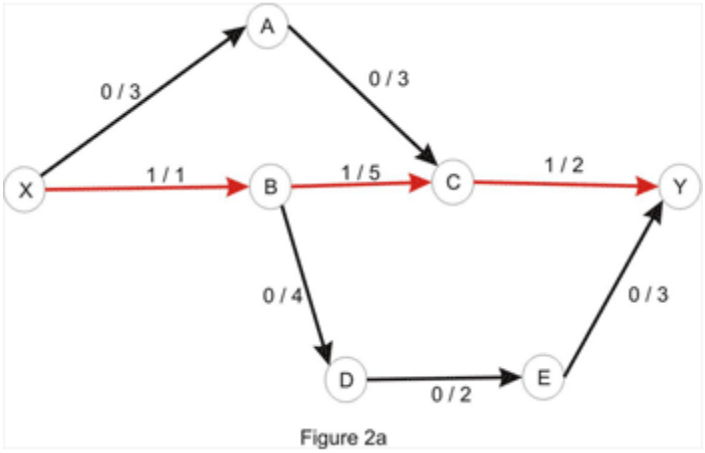


Figure 2a

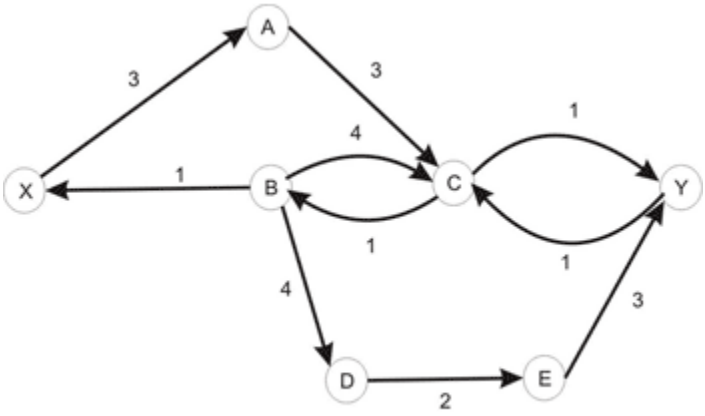


Figure 2b - The residual network of the network in 2a

By considering the path X_A_C_Y, we can increase the flow by 1 - the edges X_A and A_C have capacity of 3, as in the original network, but the edge C_Y has capacity 1, and we take the minimum of these values to get the path capacity. Increasing the flow along this path with 1 yields the flow below:

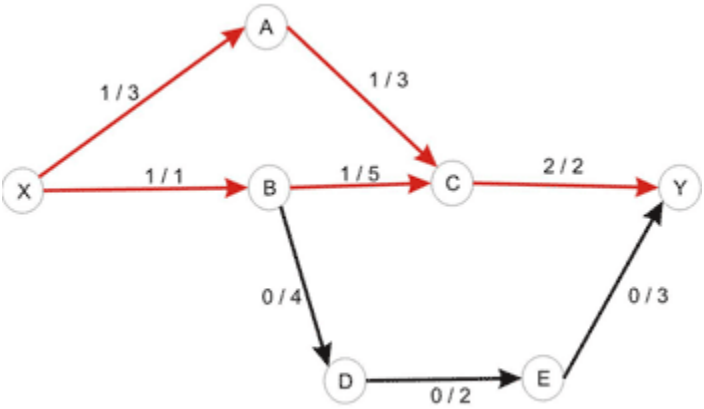


Figure 3a

The value of the current flow is now 2, and as shown in Figure 1, we could do better. So, let's try to increase the flow. Clearly, there is no point in considering the directed paths X_A_C_Y or X_B_D_E_Y as the edges C_Y and X_B, respectively, are filled to capacity. As a matter of fact, there is no directed path in the network shown above, due to the edges mentioned above being filled to capacity. At this point, the question that naturally comes to mind is: is it possible to increase the flow in this case? And the answer is yes, it is. Let's take a look at the residual network:

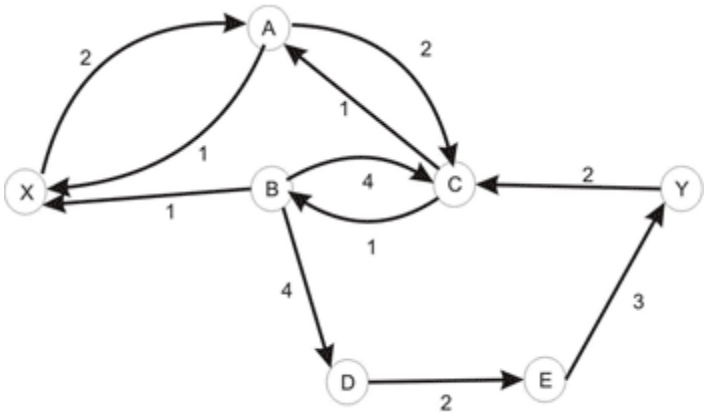


Figure 3b - The residual network of the network in 3a

Let's consider the only path from X to Y here: X_A_C_B_D_E_Y. Note that this is not a path in the directed graph, because C_B is walked in the opposite way. We'll use this path in order to increase the total flow in the original network. We'll "push" flow on each of the edges, except for C_B which we will use in order to "cancel" flow on B_C. The amount by which this operation can be performed is limited by the capacities of all edges along the path (as shown in Figure 3b). Once again we take the minimum, to conclude that this path also has capacity 1. Updating the path in the way described here yields the flow shown in Figure 1a. We are left with the following residual network where a path between the source and the sink doesn't exist:

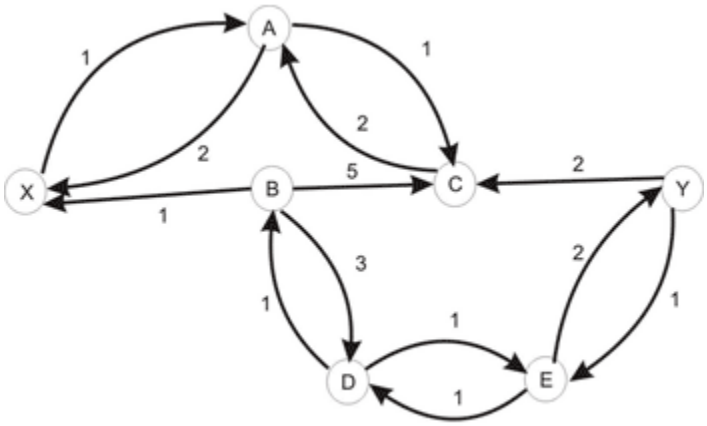
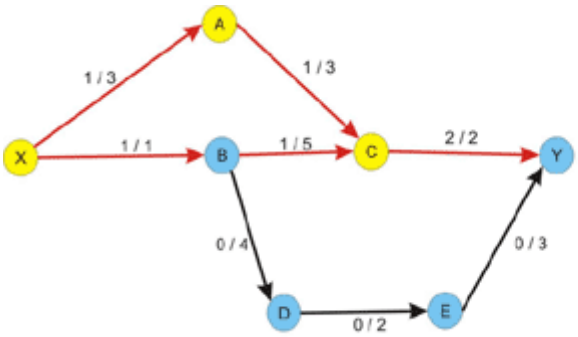


Figure 1b - The residual network of the network in 1a

This example suggests the following algorithm: start with no flow everywhere and increase the total flow in the network while there is an augmenting path from the source to the sink with no full forward edges or empty backward edges - a path in the residual network. The algorithm (known as the Ford-Fulkerson method) is guaranteed to terminate: due to the capacities and flows of the edges being integers and the path-capacity being positive, at each step we get a new flow that is closer to the maximum. As a side note, the algorithm isn't guaranteed to even terminate if the capacities are irrationals.

What about the correctness of this algorithm? It is obvious that in a network in which a maximum flow has been found there is no augmenting path, otherwise we would be able to increase the maximum value of the flow, contradicting our initial assumption. If the converse of this affirmation is true, so that when there is no augmenting path, the value of the flow has reached its maximum, we can breathe a sigh of relief, our algo is correct and computes the maximum flow in a network. This is known as the max-flow min-cut theorem and we shall justify its correctness in a few moments.

A cut in a flow network is simply a partition of the vertices in two sets, let's call them A and B, in such a way that the source vertex is in A and the sink is in B. The capacity of a cut is the sum of the capacities of the edges that go from a vertex in A to a vertex in B. The flow of the cut is the difference of the flows that go from A to B (the sum of the flows along the edges that have the starting point in A and the ending point in B), respectively from B to A, which is exactly the value of the flow in the network, due to the entering flow equals leaving flow - property, which is true for every vertex other than the source and the sink.

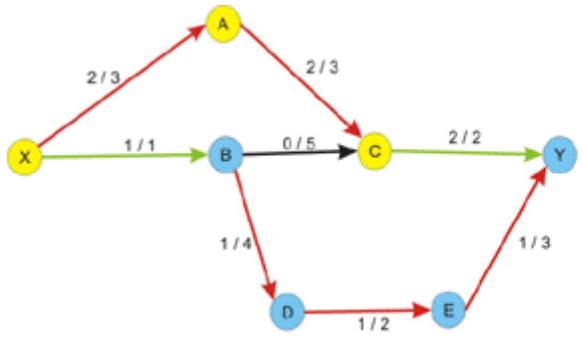


The yellow vertices are in set A and those in blue are in set B.
The capacity of the cut is the sum of the capacities of the edges XB and CY: 1 + 2 = 3.
The flow of the cut is the sum of the flow along the edges XB and CY minus the flow on BC: 1 + 2 - 1 = 2, and is equal to the value of the flow.

Figure 4 - A cut in a network

Notice that the flow of the cut is less or equal to the capacity of the cut due to the constraint of the flow being less or equal to the capacity of every edge. This implies that the maximum flow is less or equal to every cut of the network. This is where the max-flow min-cut theorem comes in and states that the value of the maximum flow through the network is exactly the value of the minimum cut of the network. Let's give an intuitive argument for this fact. We will assume that we are in the situation in which no augmenting path in the network has been found. Let's color in yellow, like in the figure above, every vertex that is reachable by a path that starts from the source and consists of non-full forward edges and of non-empty backward edges. Clearly the sink will be colored in blue, since there is no augmenting path from the source to the sink. Now take every edge that has a yellow starting point and a blue ending point. This edge will have the flow equal to the capacity, otherwise we could have added this edge to the path we had at that point and color the ending point in yellow. Note that if we remove these edges there will be no directed path from the source to the sink in the graph. Now consider every edge that has a blue starting point and a yellow ending point. The flow on this edge must be 0 since otherwise we could have added this edge as a backward edge on the current path and color the starting point in yellow. Thus, the value of the flow must equal the value of the cut, and since every flow is less or equal to every cut, this must be a maximum flow, and the cut is a minimum cut as well.

In fact, we have solved another problem that at first glance would appear to have nothing to do with maximum flow in a network, ie. given a weighted directed graph, remove a minimum-weighted set of edges in such a way that a given node is unreachable from another given node. The result is, according to the max-flow min-cut theorem, the maximum flow in the graph, with capacities being the weights given. We are also able to find this set of edges in the way described above: we take every edge with the starting point marked as reachable in the last traversal of the graph and with an unmarked ending point. This edge is a member of the minimum cut.



The minimum cut of this network is the sum of the capacities of XB and CY and equals 3, which is also the value of the maximum flow.

Figure 5 - A minimum cut in a network

Augmenting-Path Algorithms

The neat part of the Ford-Fulkerson algorithm described above is that it gets the correct result no matter how we solve (correctly!!) the sub-problem of finding an augmenting path. However, every new path may increase the flow by only 1, hence the number of iterations of the algorithm could be very large if we carelessly choose the augmenting path algorithm to use. The function *max_flow* will look like this, regardless of the actual method we use for finding augmenting paths:

```
int max_flow()
result = 0
  while (true)
// the function find_path returns the path capacity of the augmenting path found
    path_capacity = find_path()
    // no augmenting path found
    if (d = 0) exit while
    else result += path_capacity
  end while
  return result
```

To keep it simple, we will use a 2-dimensional array for storing the capacities of the residual network that we are left with after each step in the algorithm. Initially the residual network is just the original network. We will not store the flows along the edges explicitly, but it's easy to figure out how to find them upon the termination of the algorithm: for each edge x-y in the original network the flow is given by the capacity of the backward edge y-x in the residual network. Be careful though; if the reversed arc y-x also exists in the original network, this will fail, and it is

recommended that the initial capacity of each arc be stored somewhere, and then the flow along the edge is the difference between the initial and the residual capacity.

We now require an implementation for the function *find_path*. The first approach that comes to mind is to use a depth-first search (DFS), as it probably is the easiest to implement. Unfortunately, its performance is very poor on some networks, and normally is less preferred to the ones discussed next.

The next best thing in the matter of simplicity is a breadth-first search (BFS). Recall that this search usually yields the shortest path in an un-weighted graph. Indeed, this also applies here to get the shortest augmenting path from the source to the sink. In the following pseudocode we will basically: find a shortest path from the source to the sink and compute the minimum capacity of an edge (that could be a forward or a backward edge) along the path - the path capacity. Then, for each edge along the path we reduce its capacity and increase the capacity of the reversed edge with the path capacity.

```
int bfs()
  queue Q
  push source to Q
  mark source as visited
  keep an array from with the semnification: from[x] is the
previous vertex visited in the shortest path from the source to x;
initialize from with -1 (or any other sentinel value)
  while Q is not empty
    where = pop from Q
    for each vertex next adjacent to where
      if next is not visited and capacity[where][next] > 0
        push next to Q
        mark next as visited
        from[next] = where
        if next = sink
          exit while loop
    end for
  end while
  // we compute the path capacity
  where = sink, path_cap = infinity
  while from[where] > -1
    prev = from[where] // the previous vertex
    path_cap = min(path_cap, capacity[prev][where])
    where = prev
  end while
  // we update the residual network; if no path is found the while loop will not be entered
  where = sink
  while from[where] > -1
    prev = from[where]
    capacity[prev][where] -= path_capacity
        capacity[where][prev] += path_capacity
    where = prev
  end while
  // if no path is found, path_cap is infinity
  if path_cap = infinity
    return 0
  else return path_cap
```

As we can see, this is pretty easy to implement. As for its performance, it is guaranteed that this takes at most $N * M/2$ steps, where N is the number of vertices and M is the number of edges in the network. This number may seem very large, but it is over-estimated for most networks. For example, in the network we considered 3 augmenting paths are needed which is significantly less than the upper bound of 28. Due to the $O(M)$ running time of BFS (implemented with adjacency lists) the worst-case running time of the shortest-augmenting path max-flow algorithm is $O(N * M^2)$, but usually the algorithm performs much better than this.

Next we will consider an approach that uses a priority-first search (PFS), that is very similar to the Dijkstra heap method explained here. In this method the augmenting path with a maximum path capacity is preferred. Intuitively this would lead to a faster algorithm, since at each step we increase the flow with the maximum possible amount. However, things are not always so, and the BFS implementation has better running times on some networks. We assign as a priority to each vertex the minimum capacity of a path (in the residual network) from the source to that vertex. We process vertices in a greedy manner, as in Dijkstra's algorithm, in decreasing order of priorities. When we get to the sink, we are done, since a path with a maximum capacity is found. We would like to implement this with a data structure that allows us to efficiently find the vertex with the highest priority and increase the priority of a vertex (when a new better path is found) - this suggests the use of a heap which has a space complexity proportional to the number of vertices. In TopCoder matches we may find it faster and easier to implement this with a priority queue or some other data structure that approximates one, even though the space required might grow to being proportional with the number of edges. This is how the following pseudocode is implemented. We also define a structure node that has the members vertex and priority with the above significance. Another field from is needed to store the previous vertex on the path.

```
int pfs()
  priority queue PQ
  push node(source, infinity, -1) to PQ
  keep the array from as in bfs()
  // if no augmenting path is found, path_cap will remain 0
  path_cap = 0
  while PQ is not empty
    node aux = pop from PQ
    where = aux.vertex, cost = aux.priority
    if we already visited where continue
    from[where] = aux.from
    if where = sink
      path_cap = cost
      exit while loop
    mark where as visited
    for each vertex next adjacent to where
      if capacity[where][next] > 0
        new_cost = min(cost, capacity[where][next])
        push node(next, new_cost, where) to PQ
    end for
  end while
  // update the residual network
  where = sink
  while from[where] > -1
    prev = from[where]
    capacity[prev][where] -= path_cap
    capacity[where][prev] += path_cap
    where = prev
  end while
  return path_cap
```

The analysis of its performance is pretty complicated, but it may prove worthwhile to remember that with PFS at most $2M1gU$ steps are required, where U is the maximum capacity of an edge in the network. As with BFS, this number is a lot larger than the actual number of steps for most networks. Combine this with the $O(M$ 1g $M)$ complexity of the search to get the worst-case running time of this algorithm.

Now that we know what these methods are all about, which of them do we choose when we are confronted with a max-flow problem? The PFS approach seems to have a better worst-case performance, but in practice their performance is pretty much the same. So, the method that one is more familiar with may prove more adequate. Personally, I prefer the shortest-path method, as I find it easier to implement during a contest and less error prone.

Max-Flow/Min-Cut Related Problems

How to recognize max-flow problems? Often they are hard to detect and usually boil down to maximizing the movement of something from a location to another. We need to look at the constraints when we think we have a working solution based on maximum flow - they should suggest at least an $O(N^3)$ approach. If the number of locations is large, another algorithm (such as dynamic programming or greedy), is more appropriate.

The problem description might suggest multiple sources and/or sinks. For example, in the sample statement in the beginning of this article, the company might own more than one factory and multiple distribution centers. How can we deal with this? We should try to convert this to a network that has a unique source and sink. In order to accomplish this we will add two "dummy" vertices to our original network - we will refer to them as super-source and super-sink. In addition to this we will add an edge from the super-source to every ordinary source (a factory). As we don't have restrictions on the number of trucks that each factory can send, we should assign to each edge an infinite capacity. Note that if we had such restrictions, we should have assigned to each edge a capacity equal to the number of trucks each factory could send. Likewise, we add an edge from every ordinary sink (distribution centers) to the super-sink with infinite capacity. A maximum flow in this new-built network is the solution to the problem - the sources now become ordinary vertices, and they are subject to the entering-flow equals leaving-flow property. You may want to keep this in your bag of tricks, as it may prove useful to most problems.

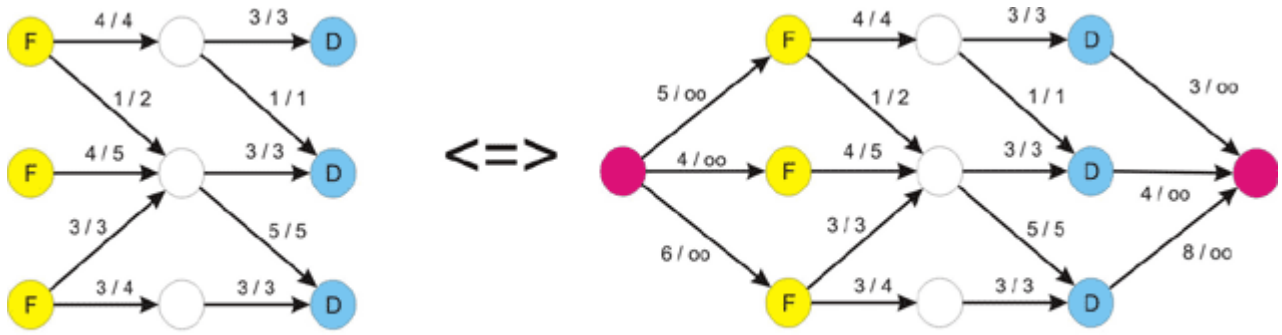


Figure 6 - Reduction of a multiple-source / multiple-sink max-flow problem

What if we are also given the maximum number of trucks that can drive through each of the cities in the country (other than the cities where the factory and the distribution center are located)? In other words we have to deal with vertex-capacities too. Intuitively, we should be able to reduce this to maximum-flow, but we must find a way to take the capacities from vertices and put them back on edges, where they belong. Another nice trick comes into play. We will build a network that has two times more vertices than the initial one. For each vertex we will have two nodes: an in-vertex and an out-vertex, and we will direct each edge x-y from the out-vertex of x to the in-vertex of y. We can assign them the capacities from the problem statement. Additionally we can add an edge for each vertex from the in to the out-vertex. The capacity this edge will be assigned is obviously the vertex-capacity. Now we just run max-flow on this network and compute the result.

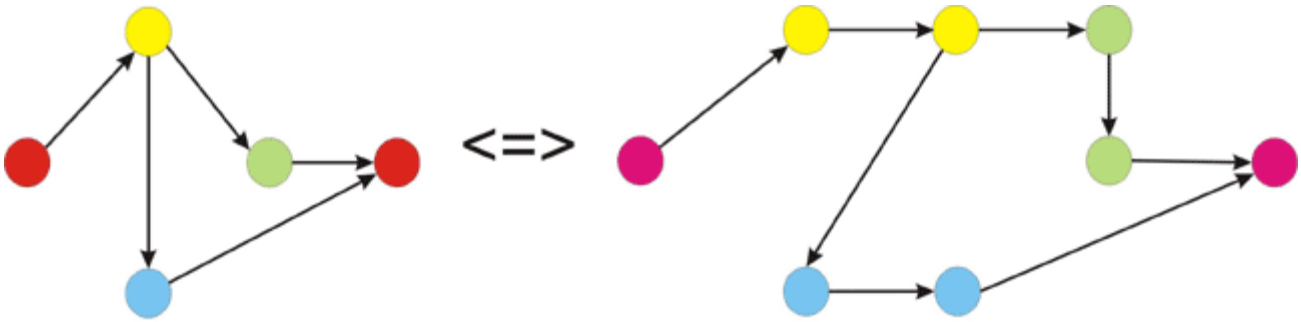


Figure 7 - Eliminating vertex-capacities

Maximum flow problems may appear out of nowhere. Let's take this problem for instance: "You are given the in and out degrees of the vertices of a directed graph. Your task is to find the edges (assuming that no edge can appear more than once)." First, notice that we can perform this simple test at the beginning. We can compute the number M of edges by summing the out-degrees or the in-degrees of the vertices. If these numbers are not equal, clearly there is no graph that could be built. This doesn't solve our problem, though. There are some greedy approaches that come to mind, but none of them work. We will combine the tricks discussed above to give a max-flow algorithm that solves this problem. First, build a network that has 2 (in/out) vertices for each initial vertex. Now draw an edge from every out vertex to every in vertex. Next, add a super-source and draw an edge from it to every out-vertex. Add a super-sink and draw an edge from every in vertex to it. We now need some capacities for this to be a flow network. It should be pretty obvious what the intent with this approach is, so we will assign the following capacities: for each edge drawn from the super-source we assign a capacity equal to the out-degree of the vertex it points to. As there may be only one arc from a vertex to another, we assign a 1 capacity to each of the edges that go from the outs to the ins. As you can guess, the capacities of the edges that enter the super-sink will be equal to the in-degrees of the vertices. If the maximum flow in this network equals M - the number of edges, we have a solution, and for each edge between the out and in vertices that has a flow along it (which is maximum 1, as the capacity is 1) we can draw an edge between corresponding vertices in our graph. Note that both x-y and y-x edges may appear in the solution. This is very similar to the maximum matching in a bipartite graph that we will discuss later. An example is given below where the out-degrees are (2, 1, 1, 1) and the in-degrees (1, 2, 1, 1).

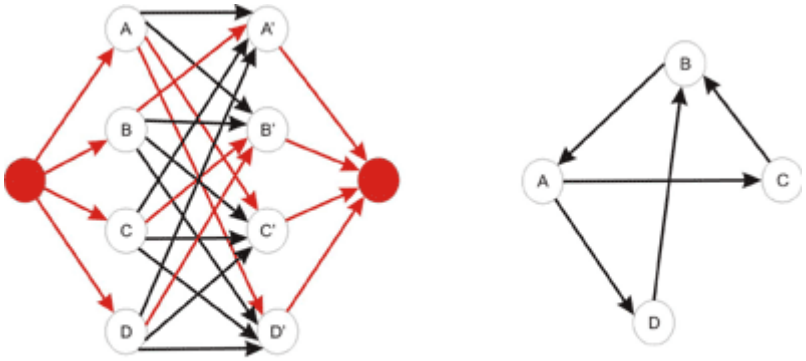


Figure 8

Some other problems may ask to separate two locations minimally. Some of these problems usually can be reduced to minimum-cut in a network. Two examples will be discussed here, but first let's take the standard min-cut problem and make it sound more like a TopCoder problem. We learned earlier how to find the value of the min-cut and how to find an arbitrary min-cut. In addition to this we will now like to have a minimum-cut with the minimum number of edges. An idea would be to try to modify the original network in such a way that the minimum cut here is the minimum cut with the minimum edges in the original one. Notice what happens if we multiply each edge capacity with a constant T. Clearly, the value of the maximum flow is multiplied by T, thus the value of the minimum cut is T times bigger than the original. A minimum cut in the original network is a minimum cut in the modified one as well. Now suppose we add 1 to the capacity of each edge. Is a minimum cut in the original network a minimum cut in this one? The answer is no, as we can see in Figure 8 shown below, if we take T = 2.

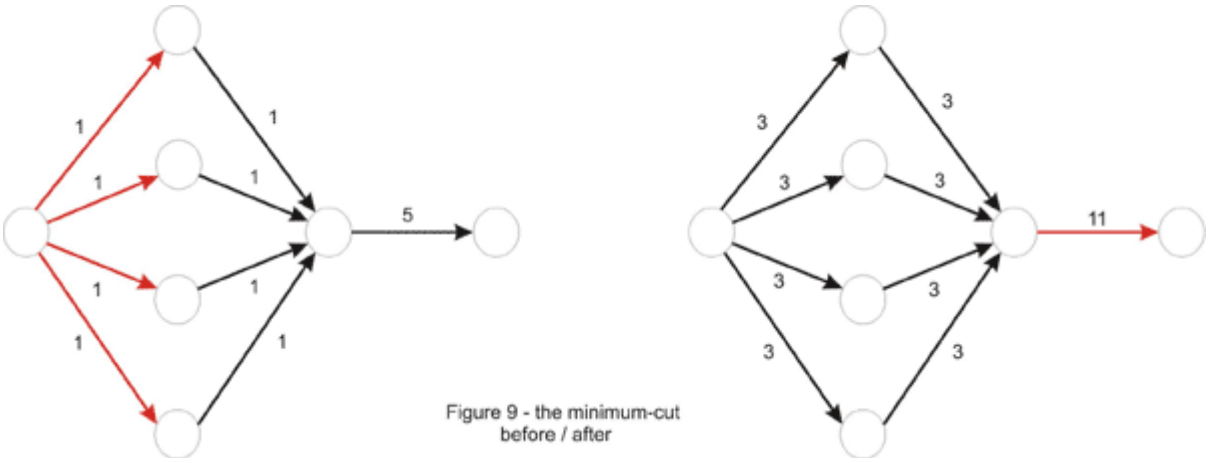


Figure 9 - the minimum-cut before / after

Why did this happen? Take an arbitrary cut. The value of the cut will be T times the original value of the cut, plus the number of edges in it. Thus, a non-minimum cut in the first place could become minimum if it contains just a few edges. This is because the constant might not have been chosen properly in the beginning, as is the case in the example above. We can fix this by choosing T large enough to neutralize the difference in the number of edges between cuts in the network. In the above example T = 4 would be enough, but to generalize, we take T = 10, one more than the number of edges in the original network, and one more than the number of edges that could possibly be in a minimum-cut. It is now true that a minimum-cut in the new network is minimum in the original network as well. However the converse is not true, and it is to our advantage. Notice how the difference between minimum cuts is now made by the number of edges in the cut. So we just find the min-cut in this new network to solve the problem correctly.

Let's illustrate some more the min-cut pattern: "An undirected graph is given. What is the minimum number of edges that should be removed in order to disconnect the graph?" In other words the problem asks us to remove some edges in order for two nodes to be separated. This should ring a bell - a minimum cut approach might work. So far we have only seen maximum flow in directed graphs, but now we are facing an undirected one. This should not be a very big problem though, as we can direct the graph by replacing every (undirected) edge x-y with two arcs: x-y and y-x. In this case the value of the min-cut is the number of edges in it, so we assign a 1 capacity to each of them. We are not asked to separate two given vertices, but rather to disconnect optimally any two vertices, so we must take every pair of vertices and treat them as the source and the sink and keep the best one from these minimum-cuts. An improvement can be made, however. Take one vertex, let's say vertex numbered 1. Because the graph should be disconnected, there must be another vertex unreachable from it. So it suffices to treat vertex 1 as the source and iterate through every other vertex and treat it as the sink.

What if instead of edges we now have to remove a minimum number of vertices to disconnect the graph? Now we are asked for a different min-cut, composed of vertices. We must somehow convert the vertices to edges though. Recall the problem above where we converted vertex-capacities to edge-capacities. The same trick works here. First "un-direct" the graph as in the previous example. Next double the number of

vertices and deal edges the same way: an edge x-y is directed from the out-x vertex to in-y. Then convert the vertex to an edge by adding a 1-capacity arc from the in-vertex to the out-vertex. Now for each two vertices we must solve the sub-problem of minimally separating them. So, just like before take each pair of vertices and treat the out-vertex of one of them as the source and the in-vertex of the other one as the sink (this is because the only arc leaving the in-vertex is the one that goes to the out-vertex) and take the lowest value of the maximum flow. This time we can't improve in the quadratic number of steps needed, because the first vertex may be in an optimum solution and by always considering it as the source we lose such a case.

Maximum Bipartite Matching

This is one of the most important applications of maximum flow, and a lot of problems can be reduced to it. A matching in a graph is a set of edges such that no vertex is touched by more than one edge. Obviously, a matching with a maximum cardinality is a maximum matching. For a general graph, this is a hard problem to deal with.

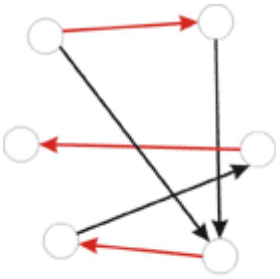


Figure 10

Let's direct our attention towards the case where the graph is bipartite - its vertices can be split into two sets such that there is no edge connecting vertices from the same set. In this case, it may sound like this: "Each of your employees can handle a given set of jobs. Assign a job to as many of them as you can."

A bipartite graph can be built in this case: the first set consists of your employees while the second one contains the jobs to be done. There is an edge from an employee to each of the jobs he could be assigned. An example is given below:

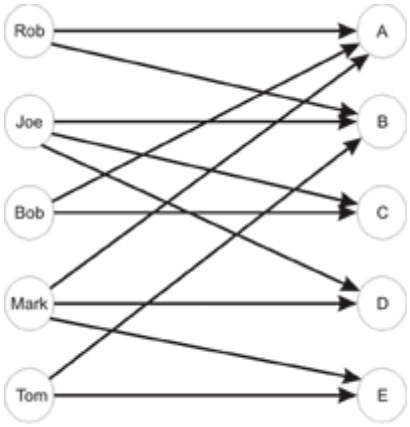


Figure 11

So, Joe can do jobs B, C and D while Mark wouldn't mind being assigned jobs A, D or E. This is a happy case in which each of your employees is assigned a job:

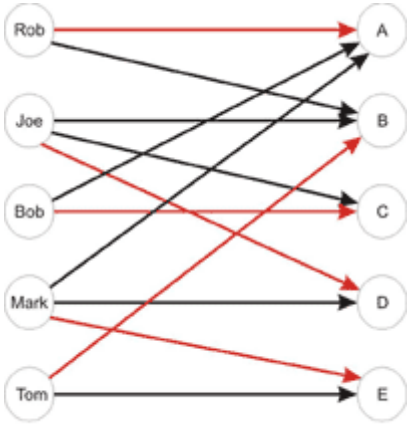


Figure 12

In order to solve the problem we first need to build a flow network. Just as we did in the multiple-source multiple-sink problem we will add two "dummy" vertices: a super-source and a super-sink, and we will draw an edge from the super-source to each of the vertices in set A (employees in the example above) and from each vertex in set B to the super-sink. In the end, each unit of flow will be equivalent to a match between an employee and a job, so each edge will be assigned a capacity of 1. If we would have assigned a capacity larger than 1 to an edge from the super-source, we could have assigned more than one job to an employee. Likewise, if we would have assigned a capacity larger than 1 to an edge going to the super-sink, we could have assigned the same job to more than one employee. The maximum flow in this network will give us the cardinality of the maximum matching. It is easy to find out whether a vertex in set B is matched with a vertex x in set A as well. We look at each edge connecting x to a vertex in set B, and if the flow is positive along one of them, there exists a match. As for the running time, the number of augmenting paths is limited by $\min(|A|,|B|)$, where by $|X|$ is denoted the cardinality of set X, making the running time $O(N \cdot M)$, where N is the number of vertices, and M the number of edges in the graph.

An implementation point of view is in place. We could implement the maximum bipartite matching just like in the pseudocode given earlier. Usually though, we might want to consider the particularities of the problem before getting to the implementation part, as they can save time or space. In this case, we could drop the 2-dimensional array that stored the residual network and replace it with two one-dimensional arrays: one of them stores the match in set B (or a sentinel value if it doesn't exist) for each element of set A, while the other is the other way around. Also, notice that each augmenting path has capacity 1, as it contributes with just a unit of flow. Each element of set A can be the first (well, the second, after the super-source) in an augmenting path at most once, so we can just iterate through each of them and try to find a match in set B. If an augmenting path exists, we follow it. This might lead to de-matching other elements along the way, but because we are following an augmenting path, no element will eventually remain unmatched in the process.

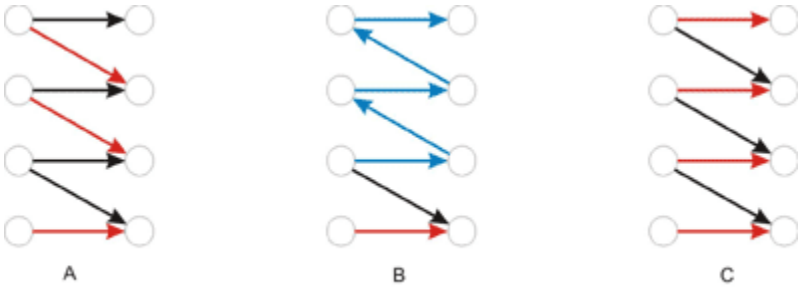


Figure 13 A) before B) an augmenting path C) after

Now let's solve some TopCoder problems!

RookAttack
Problem Statement

This problem asks us to place a maximum number of rooks on a *rows x cols* chessboard with some squares cut out. The idea behind this might be a little hard to spot, but once this is done, we get into a standard maximum bipartite-matching problem.

Notice that at most one rook can be placed on each row or column. In other words, each row corresponds at most to one column where a rook can be placed. This suggests a bipartite matching where set A is composed of elements corresponding to every row of the board, while set B consists of the columns. For each row add edges to every column if the corresponding square is not cut out of the board. Now we can just run maximum bipartite-matching in this network and compute the result. Since there are at most *rows * cols* edges, the time complexity of the algorithm is: $O(rows^2 * cols)$

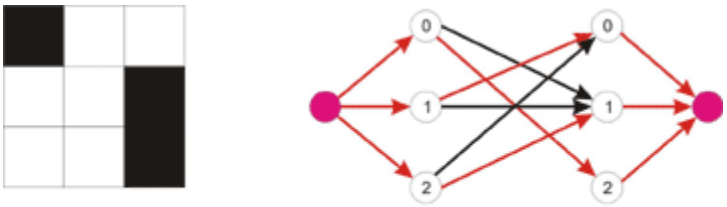


Figure 14 - Example 3 and the corresponding network

In the C++ code below BFS is used for finding an augmenting-path:

```
class RookAttack
 {     // a list of the non-empty squares for each row
   vector lst[300];
   // in this arrays we keep matches found to every row and column
   int row_match[300], col_match[300];
   // we search for an augmenting path starting with row source
   bool find_match(int source) {
     // from[x] = the row-vertex that precedes x in the path
     int from[300], where, match;
     memset(from, -1, sizeof(from));
     from[source] = source;
     deque q;
     q.push_back(source);
     bool found_path = false;
     while (!found_path && !q.empty()) {
       // where = current row-vertex we are in
       where = q.front(); q.pop_front();
       // we take every uncut square in the current row
       for (int i = 0; i < lst[where].size(); ++ i) {
         match = lst[where][i];
         // next = the row matched with column match
         int next = col_match[match];
         if (where != next) {
           // no row matched with column match thus  we found an augmenting path
           if (next == -1) {
             found_path = true;
             break;
           }
           // a check whether we already visited the row-vertex next
           if (from[next] == -1) {
             q.push_back(next);
             from[next] = where;
           }
         }
       }
     }
     if (!found_path)
       return false;
     while (from[where] != where) {
       // we de-match where from its current match (aux) and match it with match
       int aux = row_match[where];
       row_match[where] = match;
       col_match[match] = where;
       where = from[where];
       match = aux;
     }
     // at this point where = source
     row_match[where] = match;
     col_match[match] = where;
     return true;
   }

   public:
   int howMany(int rows, int cols, vector  cutouts)
   {  // build lst from cutouts; column j should appear in row's i list if square (i, j) is
present on the board
```

```
      int ret = 0;
      memset(row_match, -1, sizeof(row_match));
      memset(col_match, -1, sizeof(col_match));
      // we try to find a match for each row
      for (int i = 0; i < rows; ++ i)
        ret += find_match(i);
      return ret;
    }
  };
```

Let's take a look at the DFS version, too. We can implement the *find_match* function like this: for each non-empty square in the current row try to match the row with its corresponding column and call *find_match* recursively to attempt to find a new match for the current match (if the current match exists - if not, an augmenting path is found) of this column. If one is found, we can perform the desired match. Note that to make this run in time we must not visit the same column (or row) twice. Notice the C++ code below is extremely short:

```
 bool find_match(int where) {
     // the previous column was not matched
     if (where == -1)
       return true;
     for (int i = 0; i < lst[where].size(); ++ i) {
       int match = lst[where][i];
       if (visited[match] == false) {
         visited[match] = true;
         if (find_match(col_match[match])) {
           col_match[match] = where;
           return true;
         }
       }
     }
     return false;
   }
```

This runs in time because the number of augmenting paths is the same for both versions. The only difference is that BFS finds the shortest augmenting-path while DFS finds a longer one. As implementation speed is an important factor in TopCoder matches, in this case it would be a good deal to use the slower, but easier DFS version.

The following version of the problem is left as an exercise for the reader: to try and place as many rooks as possible on the board in such a way that the number of rooks on each row is equal to the number of rooks on each column (it is allowed for two rooks to attack each other).

Graduation
Problem Statement

In this problem we are given a set of requirements, each stating that a number of classes should be taken from a given set of classes. Each class may be taken once and fulfills a single requirement. Actually, the last condition is what makes the problem harder, and excludes the idea of a greedy algorithm. We are also given a set of classes already taken. If it weren't for this, to ensure the minimality of the return, the size of the returned string would have been (if a solution existed) the sum of the number of classes for each requirement. Now as many classes as possible must be used from this set.

At first glance, this would have been a typical bipartite-matching problem if every requirement had been fulfilled by taking just a single class. Set A would have consisted of the classes available (all characters with ASCII code in the range 33-126, except for the numeric characters '0'-'9'), while the set of requirements would have played the role of set B. This can be taken care of easily. Each requirement will contribute to set B with a number of elements equal to the number of classes that must be taken in order to fulfill it - in other words, split each requirement into several requirements. At this point, a bipartite-matching algorithm can be used, but care should be allotted to the order in which we iterate through the set of classes and match a class with a requirement.

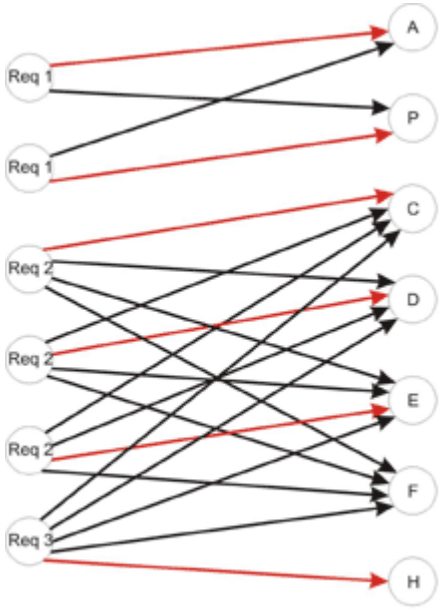


Figure 15 - Example 4 from the problem statement

It is important to understand that any order to iterate through set A can be considered when solving the standard bipartite-matching problem. For example, it doesn't matter what element from set A we choose to be the first one to be matched. Consider the solution found by the algorithm containing this element x from A, matched with an element y from B. Also, we should consider any optimal solution. Clearly, in the optimal, y must be matched with an element z from A, otherwise we can add the pair x-y to the matching, contradicting the fact that the solution is optimal. Then, we can just exchange z with x to come with a solution of the same cardinality, which completes the proof.

That being said, to gain as much as possible from the classes already taken we first must match each of these with a requirement. If, after completing this step, all requirements are fulfilled, we just need to return the empty string, as there is no need for taking more classes. Now we have to deal with the requirement that the return must be the first in lexicographic order. It should be obvious now that the other classes must be considered in increasing order. If a match is found for a class, that class is added to the return value. In the end, if not every requirement is fulfilled, we don't have a solution. The implementation is left as an exercise for the reader.

As a final note, it is possible to speed things up a bit. To achieve this, we will drop the idea of splitting each requirement. Instead we will modify the capacities of the edges connecting those with the super-sink. They will now be equal to the number of classes to be taken for each requirement. Then we can just go on with the same approach as above.

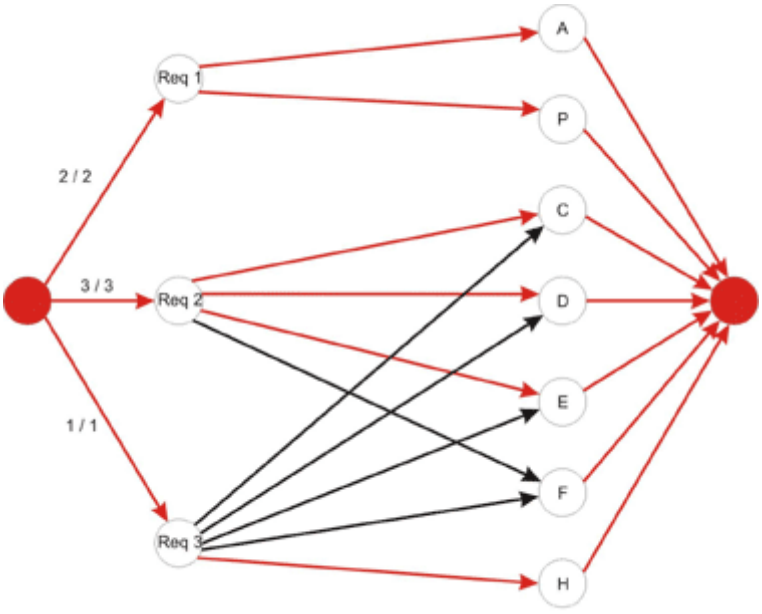


Figure 16 - Example 4 from the problem statement

Parking
Problem Statement

In this problem we have to match each of the cars with a parking spot. Additionally the time it takes for all cars to find a parking spot must be minimized. Once again we build a bipartite graph: set A is the set that consists of the cars and set B contains the parking spots. Each edge connecting elements from different sets has as the cost (and not the capacity!) the time required for the car to get to the parking spot. If the spot is unreachable, we can assign it an infinite cost (or remove it). These costs are determined by running breadth-first search.

For solving it, assume that the expected result is less than or equal to a constant D. Then, there exists a matching in which each edge connecting a car and a parking spot has the cost less than or equal to D. Thus, removing edges with cost greater than D will have no effect on the solution. This suggests a binary search on D, removing all edges with cost greater than D, and then performing a maximum bipartite-matching algorithm. If a matching exists in which every car can drive to a parking spot, we can decrease D otherwise we must increase it.

However, there is a faster and more elegant solution using a priority-first search. Instead of keeping D fixed as above, we could try to successively increase D whenever we find that it is too low. We will start with D = 0. Then we iterate through each of the cars and try to find an augmenting path in which no edge has a cost larger than D. If none exists, we increase D until one path exists. Obviously, we will increase it with the smallest possible amount. In order to achieve this, we will search for the augmenting path with the smallest cost - the cost of the path is the maximum cost of an edge on that path. This can be done with a priority-first search similar to the PFS augmenting-path algorithm presented in the first section of the article. C++ code follows:

```
struct node {
  int where, cost, from;
  node(int _where, int _cost, int _from): where(_where),
cost(_cost), from(_from) {};
};
bool operator < (node a, node b) {
  return a.cost > b.cost;
}

int minTime(vector  park)
{
  // build a cost matrix cost[i][j] = cost of getting from car i to
parking spot j, by doing a BFS
  // vertices 0, 1, ..., N - 1 will represent the cars, and vertices N, N + 1, ..., N + M - 1
will represent
//the parking spots; N + M will be the super-sink
  int D = 0, sink = N + M;
  int car_match[105], park_match[105];
  memset(car_match, -1, sizeof(car_match));
  memset(park_match, -1, sizeof(park_match));

  for (int source = 0; source < N; ++ source) {
    bool visited[210];
    memset(visited, false, sizeof(visited));
    int from[210];
    memset(from, -1, sizeof(from));
    priority_queue pq;
    pq.push(node(source, 0, -1));
    while (!pq.empty()) {
      int cst = pq.top().cost, where = pq.top().where,_from = pq.top().from;
      pq.pop();
      if (visited[where]) continue;
      visited[where] = true;
      from[where] = _from;
      // if where is a car try all M parking spots
      if (where < N) {
        for (int i = 0; i < M; ++ i) {
          // if the edge doesn't exist or this car is already matched with this parking spot
          if (cost[where][i] == infinity || car_match[where] == i) continue;
          int ncst = max(cst, cost[where][i]);
          // the i-th parking spot is N + i
          pq.push(node(N + i, ncst, where));
        }
      }
      else {
        // if this parking spot is unmatched we found
the augmenting path with minimum cost
        if (park_match[where - N] == -1) {
```

```
          from[sink] = where;
          // if D needs to be increased, increase it
          D = max(D, cst);
          break;
        }
        // otherwise we follow the backward edge
        int next = park_match[where - N];
        int ncst = max(cst, cost[next][where]);
        pq.push(node(next, ncst, where));
      }
    }

    int where = from[sink];
    // if no augmenting path is found we have no solution
    if (where == -1)
      return -1;
    // follow the augmenting path
    while (from[where] > -1) {
      int prev = from[where];
      // if where is a parking spot the edge (prev, where)
is a forward edge and the match must be updated
      if (where >= N) {
        car_match[prev] = where;
        park_match[where - N] = prev;
      }
      where = prev;
    }
  }

  return D;
}
```

Here are some problems to practice:

PlayingCubes - for this one ignore the low constraints and try to find a max-flow algorithm

DataFilter - be warned this is the hard problem from the TCCC 2004 Finals and is tough indeed!

Some other problems from http://acm.uva.es/p/: 563, 753, 820, 10122, 10330, 10511, 10735.

Choosing the correct data type for your variables can often be the only difference between a faulty solution and a correct one. Especially when there's some geometry around, precision problems often cause solutions to fail. To make matters even worse, there are many (often incorrect) rumors about the reasons of these problems and ways how to solve them.

To be able to avoid these problems, one has to know a bit about how things work inside the computer. In this article we will take a look at the necessary facts and disprove some false rumors. After reading and understanding it, you should be able to avoid the problems mentioned above.

This article is in **no way** intended to be a complete reference, nor to be 100% accurate. Several times, presented things will be a bit simplified. As the readers of this article are TopCoder (TC) members, we will concentrate on the x86 architecture used by the machines TC uses to evaluate solutions. For example, we will assume that on our computers a byte consists of 8 bits and that the machines use 32-bit integer registers.

While most of this article is general and can be applied on all programming languages used at TC, the article is slightly biased towards C++ and on some occasions special notes on g++ are included.

We will start by presenting a (somewhat simplified) table of integer data types available in the g++ compiler. You can find this table in any g++ reference. All of the other compilers used at TC have similar data types and similar tables in their references, look one up if you don't know it by heart yet. Below we will explain that all we need to know is the storage size of each of the types, the range of integers it is able to store can be derived easily.

Table 1: Integer data types in g++.

name	size in bits	representable range
char	8	-2^7 to $2^7 - 1$
unsigned char	8	0 to $2^8 - 1$
short	16	-2^{15} to $2^{15} - 1$
unsigned short	16	0 to $2^{16} - 1$
int	32	-2^{31} to $2^{31} - 1$
unsigned int	32	0 to $2^{32} - 1$
long	32	-2^{31} to $2^{31} - 1$
unsigned long	32	0 to $2^{32} - 1$
long long	64	-2^{63} to $2^{63} - 1$
unsigned long long	64	0 to $2^{64} - 1$

Notes:

- The storage size of an int and an unsigned int is platform dependent. E.g., on machines using 64-bit registers, ints in g++ will have 64 bits. The old Borland C compiler used 16-bit ints. It is guaranteed that an int will always have at least 16 bits. Similarly, it is guaranteed that on any system a long will have at least 32 bits.
- The type long long is a g++ extension, it is not a part of any C++ standard (yet?). Many other C++ compilers miss this data type or call it differently. E.g., MSVC++ has __int64 instead.

Rumor: *Signed integers are stored using a sign bit and "digit" bits.*

Validity: *Only partially true.*

Most of the current computers, including those used at TC, store the integers in a so-called *two's complement form*. It is true that for non-

negative integers the most significant bit is zero and for negative integers it is one. But this is not exactly a sign bit, we can't produce a "negative zero" by flipping it. Negative numbers are stored in a somewhat different way. The negative number -n is stored as a bitwise negation of the non-negative number (n-1).

In Table 2 we present the bit patterns that arise when some small integers are stored in a (signed) char variable. The rightmost bit is the least significant one.

Table 2: Two's complement bit patterns for some integers.

value	two's complement form
0	00000000
1	00000001
2	00000010
46	00101110
47	00101111
127	01111111
-1	11111111
-2	11111110
-3	11111101
-47	11010001
-127	10000001
-128	10000000

Note that due to the way negative numbers are stored the set of representable numbers is not placed symmetrically around zero. The largest representable integer in *b* bits is $2^{b-1} - 1$, the smallest (i.e., most negative) one is -2^{b-1}.

A neat way of looking at the two's complement form is that the bits correspond to digits in base 2 with the exception that the largest power of two is negative. E.g., the bit pattern 11010001 corresponds to 1 x (- 128) + 1 x 64 + 0 x 32 + 1 x 16 + 0 x 8 + 0 x 4 + 0 x 2 + 1 x 1 = - 128 + 81 = - 47

Rumor: *Unsigned integers are just stored as binary digits of the number.*

Validity: *True.*

In general, the bit pattern consists of base 2 digits of the represented number. E.g., the bit pattern 11010001 corresponds to 1 x 128 + 1 x 64 + 0 x 32 + 1 x 16 + 0 x 8 + 0 x 4 + 0 x 2 + 1 x 1 = 209.

Thus, in a *b*-bit unsigned integer variable, the smallest representable number is zero and the largest is $2^b - 1$ (corresponding to an all-ones pattern).

Note that if the leftmost (most significant) bit is zero, the pattern corresponds to the same value regardless of whether the variable is signed or unsigned. If we have a *b*-bit pattern with the leftmost bit set to one, and the represented unsigned integer is *x*, the same pattern in a signed variable represents the value $x - 2^b$.

In our previous examples, the pattern 11010001 can represent either 209 (in an unsigned variable) or -47 (in a signed variable).

Rumor: *In C++, the code "int A[1000]; memset(A,x,sizeof(A));" stores 1000 copies of x into A.*

Validity: *False.*

The memset() function fills a part of the memory with chars, not ints. Thus for most values of x you would get unexpected results.

However, this does work (and is often used) for two special values of x: 0 and -1. The first case is straightforward. By filling the entire array with zeroes, all the bits in each of the ints will be zero, thus representing the number 0. Actually, the second case is the same story: -1 stored in a char is 1111111, thus we fill the entire array with ones, getting an array containing -1s.

(Note that most processors have a special set of instructions to fill a part of memory with a given value. Thus the memset() operation is usually much faster than filling the array in a cycle.)

When you know what you are doing, memset() can be used to fill the array A with sufficiently large/small values, you just have to supply a suitable bit pattern as the second argument. E.g., use *x* = 63 to get really large values (1, 061, 109, 567) in A.

Rumor: *Bitwise operations can be useful.*

Validity: *True.*

First, they are fast. Second, many useful tricks can be done using just a few bitwise operations.

As an easy example, x is a power of 2 if and only if (x & (x-1) == 0). (Why? Think how does the bit pattern of a power of 2 look like.) Note that x=x & (x-1) clears the least significant set bit. By repeatedly doing this operation (until we get zero) we can easily count the number of ones in the binary representation of x.

If you are interested in many more such tricks, download the free second chapter of the book Hacker's Delight and read The Aggregate Magic Algorithms.

One important trick: unsigned ints can be used to encode subsets of {0, 1,..., 31} in a straightforward way - the *i*-th bit of a variable will be one if and only if the represented set contains the number *i*. For example, the number 18 (binary 10010 = $2^4 + 2^1$) represents the set {1, 4}.

When manipulating the sets, bitwise "and" corresponds to their intersection, bitwise "or" gives their union.

In C++, we may explicitly set the *i*-th bit of *x* using the command x |= (1<<i), clear it using x &= ~(1<<i) and check whether it is set using ((x & (1<<i)) != 0). Note that bitset and vector<bool> offer a similar functionality with arbitrarily large sets.

This trick can be used when your program has to compute the answer for all subsets of a given set of things. This concept is quite often used in SRM problems. We won't go into more details here, the best way of getting it right is looking at an actual implementation (try looking at the best solutions for the problems below) and then trying to solve a few such problems on your own.

- BorelSets (a simple exercise in set manipulation, generate sets until no new sets appear)
- TableSeating
- CompanyMessages
- ChessMatch (for each subset of your players find the best assignment)
- RevolvingDoors (encode your position and the states of all the doors into one integer)

Rumor: *Real numbers are represented using a floating point representation.*

Validity: *True.*

The most common way to represent "real" numbers in computers is the *floating point* representation defined by the IEEE Standard 754. We will

give a brief overview of this representation.

Basically, the words "floating point" mean that the position of the decimal (or more exactly, binary) point is not fixed. This will allow us to store a large range of numbers than fixed point formats allow.

The numbers will be represented in scientific notation, using a normalized number and an exponent. For example, in base 10 the number 123.456 could be represented as 1.23456×10^2. As a shorthand, we sometimes use the letter E to denote the phrase "times 10 to the power of". E.g., the previous expression can be rewritten as 1.23456e2.

Of course, in computers we use binary numbers, thus the number 5.125 (binary 101.001) will be represented as 1.01001×2^2, and the number -0.125 (binary -0.001) will be represented as -1×2^{-3}.

Note that any (non-zero) real number x can be written in the form $(-1)^s \times m \times 2^e$, where $s \in \{0, 1\}$ represents the sign, $m \in [1, 2)$ is the normalized number and e is the (integer) exponent. This is the general form we are going to use to store real numbers.

What exactly do we need to store? The base is fixed, so the three things to store are the sign bit s, the normalized number (known as the *mantissa*) m and the exponent e.

The IEEE Standard 754 defines four types of precision when storing floating point numbers. The two most commonly used are *single* and *double precision*. In most programming languages these are also the names of corresponding data types. You may encounter other data types (such as *float*) that are platform dependent and usually map to one of these types. If not sure, stick to these two types.

Single precision floating point numbers use 32 bits (4 bytes) of storage, double precision numbers use 64 bits (8 bytes). These bits are used as shown in Table 3:

Table 3: Organization of memory in singles and doubles.

	sign	exponent	mantissa
single precision	1	8	23
double precision	1	11	52

(The bits are given in order. I.e., the sign bit is the most significant bit, 8 or 11 exponent bits and then 23 or 52 mantissa bits follow.)

The sign bit
The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Inverting this bit changes the sign of the number.

The exponent
The exponent field needs to represent both positive and negative exponents. To be able to do this, a *bias* is added to the actual exponent e. This bias is 127 for single precision and 1023 for double precision. The result is stored as an unsigned integer. (E.g., if e = - 13 and we use single precision, the actual value stored in memory will be -13 + 127 = 114.)

This would imply that the range of available exponents is -127 to 128 for single and -1023 to 1024 for double precision. This is almost true. For reasons discussed later, both boundaries are reserved for special numbers. The actual range is then -126 to 127, and -1022 to 1023, respectively.

The mantissa
The mantissa represents the precision bits of the number. If we write the number in binary, these will be the first few digits, regardless of the position of the binary point. (Note that the position of the binary point is specified by the exponent.)

The fact that we use base 2 allows us to do a simple optimization: We know that for any (non-zero) number the first digit is surely 1. Thus we don't have to store this digit. As a result, a b-bit mantissa can actually store the $b + 1$ most significant bits of a number.

Rumor: *Floating point variables can store not only numbers but also some strange values.*

Validity: *True.*

As stated in the previous answer, the standard reserves both the smallest and the largest possible value of the exponent to store special

numbers. (Note that in memory these values of the exponent are stored as "all zeroes" and "all ones", respectively.)

Zero
When talking about the sign-mantissa-exponent representation we noted that any **non-zero** number can be represented in this way. Zero is not directly representable in this way. To represent zero we will use a special value denoted with both the exponent field and the mantissa containing all zeroes. Note that -0 and +0 are distinct values, though they both compare as equal.

It is worth noting that if memset() is used to fill an array of floating point variables with zero bytes, the value of the stored numbers will be zero. Also, global variables in C++ are initialized to a zero bit pattern, thus global floating point variables will be initialized to zero.

Also, note that negative zero is sometimes printed as "-0" or "-0.0". In some programming contests (with inexperienced problemsetters) this may cause your otherwise correct solution to fail.

There are quite a few subtle pitfalls concerning the negative zero. For example, the expressions "0.0 - x" and "-x" are not equivalent - if $x = 0.0$, the value of the first expression is 0.0, the second one evaluates to -0.0.

My favorite quote on this topic: *Negative zeros can "create the opportunity for an educational experience" when they are printed as they are often printed as "-0" or "-0.0" (the "educational experience" is the time and effort that you spend learning why you're getting these strange values).*

Infinities
The values +infinity and -infinity correspond to an exponent of all ones and a mantissa of all zeroes. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations.

Not a Number
The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaNs are represented by a bit pattern with an exponent of all ones and a non-zero mantissa. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN).

A QNaN is a NaN with the most significant bit of the mantissa set. QNaNs propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. (For example, 3*sqrt(-1.0) is a QNaN.)

An SNaN is a NaN with the most significant bit of the mantissa clear. It is used to signal an exception when used in operations. SNaNs can be handy to assign to uninitialized variables to trap premature usage.

If a return value is a QNaN, it means that it is impossible to determine the result of the operation, a SNaN means that the operation is invalid.

Subnormal numbers
We still didn't use the case when the exponent is all zeroes and the mantissa is non-zero. We will use these values to store numbers very close to zero.

These numbers are called *subnormal*, as they are smaller than the normally representable values. Here we don't assume we have a leading 1 before the binary point. If the sign bit is s, the exponent is all zeroes and the mantissa is m, the value of the stored number is $(-1)^s \times 0.m \times 2^{-q}$, where q is 126 for single and 1022 for double precision.

(Note that zero is just a special case of a subnormal number. Still, we wanted to present it separately.)

Summary of all possible values
In the following table, b is the bias used when storing the exponent, i.e., 127 for single and 1023 for double precision.

sign s	**exponent** e	**mantissa** m	**represented number**
0	00...00	00...00	+0.0
0	00...00	00...01 to 11...11	$0.m \times 2^{-b+1}$
0	00...01 to 11...10	anything	$1.m \times 2^{e-b}$
0	11...11	00...00	+Infinity
0	11...11	00...01 to 01...11	SNaN
0	11...11	10...00 to 11...11	QNaN

1	00...00	00...00	-0.0
1	00...00	00...01 to 11...11	$-0.m \times 2^{-b+1}$
1	00...01 to 11...10	anything	$-1.m \times 2^{e-b}$
1	11...11	00...00	-Infinity
1	11...11	00...01 to 01...11	SNaN
1	11...11	10...00 to 11.11	QNaN

Operations with all the special numbers
All operations with the special numbers presented above are well-defined. This means that your program won't crash just because one of the computed values exceeded the representable range. Still, this is usually an unwanted situation and if it may occur, you should check it in your program and handle the cases when it occurs.

The operations are defined in the probably most intuitive way. Any operation with a NaN yields a NaN as a result. Some other operations are presented in the table below. (In the table, *r* is a positive representable number, ∞ is Infinity, ÷ is normal floating point division.) A complete list can be found in the standard or in your compiler's documentation. Note that even comparison operators are defined for these values. This topic exceeds the scope of this article, if interested, browse through the references presented at the end of the article.

operation	**result**
$0 \div \pm\infty$	0
$\pm r \div \pm\infty$	0
$(-1)^s\infty \times (-1)^t\infty$	$(-1)^{st}\infty$
$\infty + \infty$	∞
$\pm r \div 0$	$\pm\infty$
$0 \div 0$	NaN
$\infty - \infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Rumor: *Floating point numbers can be compared by comparing the bit patterns in memory.*

Validity: *True.*

Note that we have to handle sign comparison separately. If one of the numbers is negative and the other is positive, the result is clear. If both numbers are negative, we may compare them by flipping their signs, comparing and returning the opposite answer. From now on consider non-negative numbers only.

When comparing the two bit patterns, the first few bits form the exponent. The larger the exponent is, the further is the bit pattern in lexicographic order. Similarly, patterns with the same exponent are compared according to their mantissa.

Another way of looking at the same thing: when comparing two non-negative real numbers stored in the form described above, the result of the comparison is always the same as when comparing integers with the same bit pattern. (Note that this makes the comparison pretty fast.)

Rumor: *Comparing floating point numbers for equality is usually a bad idea.*

Validity: *True.*

Consider the following code:
for (double r=0.0; r!=1.0; r+=0.1) printf("*");
How many stars is it going to print? Ten? Run it and be surprised. The code just keeps on printing the stars until we break it.

Where's the problem? As we already know, doubles are not infinitely precise. The problem we encountered here is the following: In binary, the representation of 0.1 is not finite (as it is in base 10). Decimal 0.1 is equivalent to binary 0.0(0011), where the part in the parentheses is repeated forever. When 0.1 is stored in a double variable, it gets rounded to the closest representable value. Thus if we add it 10 times the result is not exactly equal to one.

The most common advice is to use some tolerance (usually denoted ε) when comparing two doubles. E.g., you may sometimes hear the following hint: consider the doubles a and b equal, if fabs(a-b)<1e-7. Note that while this is an improvement, it is not the best possible way. We will show a better way later on.

Rumor: *Floating point numbers are not exact, they are rounded.*

Validity: *Partially true.*

Yes, if a number can't be represented exactly, it has to be rounded. But sometimes an even more important fact is that lots of important numbers (like zero, the powers of two, etc.) can be stored exactly. And it gets even better. Note that the mantissa of doubles contains more than 32 bits. Thus all the binary digits of an int fit into the mantissa and the stored value is exact.

This can still be improved. If we note that $2^{54} > 10^{16}$, it should be clear that any integer with up to 15 decimal digits has at most 54 binary digits, and thus it can be stored in a double without rounding. This observation can even be extended to non-integer values: double is able to store 15 most significant decimal digits of any number (in the normally representable range).

A similar computation for singles shows that they are able to store only 7 most significant decimal digits. This is way too little for almost any practical applications, and what's even more important, it is less than the precision required by TC when a floating point value shall be returned. The moral of this story is pretty clear: **Never use singles!** Seriously. Don't even think about it. There's plenty of available memory nowadays.

As a side note, once rounding errors occur in your computation, they are propagated into further computations. Thus even if the final result shall be an integer, its representation in a floating point variable may not be exact. As an example consider the star-printing cycle above.

Rumor: *I've heard that long doubles will give me more precision.*

Validity: *Platform dependent.*

One less common precision type defined in the IEEE-754 standard is *extended double precision*, which requires at least 79 bits of storage space. Some compilers have a data type with this precision, some don't. E.g., in g++ on the x86 architecture we have a data type long double that uses 10 bytes (80 bits) of memory. (In MSVC++ the type long double is present, but it is mapped to a double.)

The 80-bit extended double precision format is used internally by the Intel 80x87 floating-point math co-processor in order to be able to shift operands back and forth without any loss of precision in the IEEE-754 64-bit (and 32-bit) format. When optimization in g++ is set to a non-zero value, g++ may even generate code that uses long doubles internally instead of doubles and singles. This format is able to store 19 most significant decimal digits.

If even more precision is required, you may either implement your own arithmetic, or use the BigInteger and BigDecimal classes from Java's math library.

Rumor: *In practice, there's no real difference between using different values of ε when comparing floating point numbers.*

Validity: *False.*

Often if you visit the Round Tables after a SRM that involved a floating point task you can see people posting messages like "after I changed the precision from 1e-12 to 1e-7 it passed all systests in the practice room"

Examples of such discussions: here, here, here, here and here. (They are worth reading, it is always less painful to learn on the mistakes of other people made than to learn on your own mistakes.)

We will start our answer by presenting another simple example.
for (double r=0.0; r<1e22; r+=1.0) printf(".");
How many dots will this program print? This time it's clear, isn't it? The terminating condition doesn't use equality testing. The cycle has to stop after 10^{22} iterations. Or... has it?

Bad luck, this is again an infinite cycle. Why is it so? Because when the value of *r* becomes large, the precision of the variable isn't large enough to store all decimal digits of *r*. The last ones become lost. And when we add 1 to such a large number, the result is simply rounded back to the original number.

Exercise: Try to estimate the largest value of *r* our cycle will reach. Verify your answer. If your estimate was wrong, find out why.

After making this observation, we will show why the expression fabs(a-b)<epsilon (with a fixed value of epsilon, usually recommended between 1e-7 and 1e-9) is not ideal for comparing doubles.

Consider the values 123456123456.1234588623046875 and 123456123456.1234741210937500. There's nothing that special about them. These are just two values that can be stored in a double without rounding. Their difference is approximately 2e-5.

Now take a look at the bit patterns of these two values:

first: 01000010 00111100 10111110 10001110 11110010 01000000 00011111 10011011
second: 01000010 00111100 10111110 10001110 11110010 01000000 00011111 10011100

Yes, right. These are two consecutive values that can be stored in a double. Almost any rounding error can change one of them onto the other one (or even further). And still, they are quite far apart, thus our original test for "equality" fails.

What we really want is to tolerate small precision errors. As we already saw, doubles are able to store approximately 15 most significant decimal digits. By accumulating precision errors that arise due to rounding, the last few of these digits may become corrupt. But how exactly shall we implement tolerating such errors?

We won't use a constant value of ε, but a value relative to the magnitude of the compared numbers. More precisely, if *x* is a double, then x*1e-10 is a number that's 10 degrees of magnitude smaller than *x*. Its most significant digit corresponds to *x*'s eleventh most significant digit. This makes it a perfect ε for our needs.

In other words, a better way to compare doubles a and b for "equality" is to check whether a lies between b*(1-1e-10) and b*(1+1e-10). (Be careful, if b is negative, the first of these two numbers is larger!)

See any problems with doing the comparison this way? Try comparing 1e-1072 and -1e-1072. Both numbers are almost equal to zero and to each other, but our test fails to handle this properly. This is why we have to use **both** the first test (known as testing for an absolute error) and the second test (known as testing for a relative error).

This is the way TC uses to check whether your return value is correct. Now you know why.

There are even better comparison functions (see one of the references), but it is important to know that in practice you can often get away with using only the absolute error test. Why? Because the numbers involved in computation come from a limited range. For example, if the largest number you will ever compare is 9947, you know that a double will be able to store another 11 digits after the decimal point correctly. Thus if we use epsilon=1e-8 when doing the absolute error test, we allow the last three significant digits to become corrupt.

The advantage this approach gives you is clear: checking for an absolute error is much simpler than the advanced tests presented above.

- Elections (a Div2 easy with a success rate of only 57.58%)
- Archimedes
- SortEstimate (the binary search is quite tricky to get right if you don't understand precision issues)
- PerforatedSheet (beware, huge rounding errors possible)
- WatchTower
- PackingShapes

Rumor: *Computations using floating point variables are as exact as possible.*

Validity: *True.*

Most of the standards require this. To be even more exact: For any arithmetical operation the returned value has to be that representable value that's closest to the exact result. Moreover, in C++ the default rounding mode says that if two values are tied for being the closest, the one that's more even (i.e., its least significant bit of the mantissa is 0) is returned. (Other standards may have different rules for this tie breaking.)

As a useful example, note that if an integer n is a square (i.e., $n = k^2$ for some integer k), then sqrt(double(n)) will return the exact value k. And as we know that k can be stored in a variable of the same type as n, the code int k = int(sqrt(double(n))) is safe, there will be no rounding errors.

Rumor: *If I do the same computation twice, the results I get will be equal.*

Validity: *Partially true.*

Wait, only partially true? Doesn't this contradict the previous answer? Well, it doesn't.

In C++ this rumor isn't always true. The problem is that according to the standard a C++ compiler can sometimes do the internal calculations using a larger data type. And indeed, g++ sometimes internally uses long doubles instead of doubles to achieve larger precision. The value stored is only typecast to double when necessary. If the compiler decides that in one instance of your computation long doubles will be used and in the other just doubles are used internally, the different roundings will influence the results and thus the final results may differ.

This is one of THE bugs that are almost impossible to find and also one of the most confusing ones. Imagine that you add debug outputs after each step of the computations. What you unintentionally cause is that after each step each of the intermediate results is cast to double and output. In other words, you just pushed the compiler to only use doubles internally and suddenly everything works. Needless to say, after you remove the debug outputs, the program will start to misbehave again.

A workaround is to write your code using long doubles only.

Sadly, this only cures one of the possible problems. The other is that when optimizing your code the compiler is allowed to rearrange the order in which operations are computed. On different occasions, it may rewrite two identical pieces of C++ code into two different sets of instructions. And all the precision problems are back.

As an example, the expression $x + y - z$ may once be evaluated as $x + (y - z)$ and the other time as $(x + y) - z$. Try substituting the values $x = 1.0$ and $y = z = 10^{30}$.

Thus even if you have two identical pieces of code, you can't be sure that they will produce exactly the same result. If you want this guarantee, wrap the code into a function and call the same function on both occasions.

Binary search is one of the fundamental algorithms in computer science. In order to explore it, we'll first build up a theoretical backbone, then use that to implement the algorithm properly and avoid those nasty off-by-one errors everyone's been talking about.

Finding a value in a sorted sequence

In its simplest form, binary search is used to quickly find a value in a sorted sequence (consider a sequence an ordinary array for now). We'll call the sought value the *target* value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

0 5 13 19 22 41 55 68 72 81 98

We are interested in the location of the target value in the sequence so we will represent the search space as indices into the sequence. Initially, the search space contains indices 1 through 11. Since the search space is really an interval, it suffices to store just two numbers, the low and high indices. As described above, we now choose the median value, which is the value at index 6 (the midpoint between 1 and 11): this value is 41 and it is smaller than the target value. From this we conclude not only that the element at index 6 is not the target value, but also that no element at indices between 1 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 7 through 11:

55 68 72 81 98

Proceeding in a similar fashion, we chop off the second half of the search space and are left with:

55 68

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element. Either way, we conclude that the index where the target value is located is 7.

If the target value was not present in the sequence, binary search would empty the search space entirely. This condition is easy to check and handle. Here is some code to go with the description:

```
binary_search(A, target):
   lo = 1, hi = size(A)
   while lo <= hi:
      mid = lo + (hi-lo)/2
      if A[mid] == target:
         return mid
      else if A[mid] < target:
         lo = mid+1
      else:
         hi = mid-1

   // target was not found
```

Complexity

Since each comparison binary search uses halves the search space, we can assert and easily prove that binary search will never use more than (in big-oh notation) $O(\log N)$ comparisons to find the target value.

The logarithm is an awfully slowly growing function. In case you're not aware of just how efficient binary search is, consider looking up a name in a phone book containing a million names. Binary search lets you systematically find any given name using at most 21 comparisons. If you could manage a list containing all the people in the world sorted by name, you could find any person in less than 35 steps. This may not seem feasible or useful at the moment, but we'll soon fix that.

Note that this assumes that we have random access to the sequence. Trying to use binary search on a container such as a linked list makes little sense and it is better use a plain linear search instead.

Binary search in standard libraries
C++'s Standard Template Library implements binary search in algorithms lower_bound, upper_bound, binary_search and equal_range, depending exactly on what you need to do. Java has a built-in Arrays.binary_search method for arrays and the .NET Framework has Array.BinarySearch.

You're best off using library functions whenever possible, since, as you'll see, implementing binary search on your own can be tricky.

Beyond arrays: the discrete binary search
This is where we start to abstract binary search. A sequence (array) is really just a function which associates integers (indices) with the corresponding values. However, there is no reason to restrict our usage of binary search to tangible sequences. In fact, we can use the same algorithm described above on any monotonic function *f* whose domain is the set of integers. The only difference is that we replace an array lookup with a function evaluation: we are now looking for some x such that f(x) is equal to the target value. The search space is now more formally a subinterval of the domain of the function, while the target value is an element of the codomain. The power of binary search begins to show now: not only do we need at most $O(\log N)$ comparisons to find the target value, but we also do not need to evaluate the function more than that many times. Additionally, in this case we aren't restricted by practical quantities such as available memory, as was the case with arrays.

Taking it further: the main theorem
When you encounter a problem which you think could be solved by applying binary search, you need some way of proving it will work. I will now present another level of abstraction which will allow us to solve more problems, make proving binary search solutions very easy and also help implement them. This part is a tad formal, but don't get discouraged, it's not that bad.

Consider a predicate *p* defined over some ordered set *S* (the search space). The search space consists of candidate solutions to the problem. In this article, a predicate is a function which returns a boolean value, true or false (we'll also use *yes* and *no* as boolean values). We use the predicate to verify if a candidate solution is legal (does not violate some constraint) according to the definition of the problem.

What we can call the *main theorem* states that **binary search can be used if and only if for all x in *S*, *p*(x) implies *p*(y) for all y > x**. This property is what we use when we discard the second half of the search space. It is equivalent to saying that $\neg p(x)$ implies $\neg p(y)$ for all y < x (the symbol ¬ denotes the logical not operator), which is what we use when we discard the first half of the search space. The theorem can easily be proven, although I'll omit the proof here to reduce clutter.

Behind the cryptic mathematics I am really stating that if you had a yes or no question (the predicate), getting a *yes* answer for some potential solution x means that you'd also get a *yes* answer for any element after x. Similarly, if you got a *no* answer, you'd get a *no* answer for any element before x. As a consequence, if you were to ask the question for each element in the search space (in order), you would get a series of *no* answers followed by a series of *yes* answers.

Careful readers may note that binary search can also be used when a predicate yields a series of *yes* answers followed by a series of *no* answers. This is true and complementing that predicate will satisfy the original condition. For simplicity we'll deal only with predicates described in the theorem.

If the condition in the main theorem is satisfied, we can use binary search to find the smallest legal solution, i.e. the smallest x for which *p*(x) is true. The first part of devising a solution based on binary search is designing a predicate which can be evaluated and for which it makes sense to use binary search: we need to choose what the algorithm should find. We can have it find either the *first x for which p(x) is true* or the *last x for which p(x) is false*. The difference between the two is only slight, as you will see, but it is necessary to settle on one. For starters, let us seek the first *yes* answer (first option).

The second part is proving that binary search can be applied to the predicate. This is where we use the main theorem, verifying that the conditions laid out in the theorem are satisfied. The proof doesn't need to be overly mathematical, you just need to convince yourself that *p*(x) implies *p*(y) for all y > x or that $\neg p(x)$ implies $\neg p(y)$ for all y < x. This can often be done by applying common sense in a sentence or two.

When the domain of the predicate are the integers, it suffices to prove that *p*(x) implies *p*(x+1) or that $\neg p(x)$ implies $\neg p(x-1)$, the rest then follows by induction.

These two parts are most often interleaved: when we think a problem can be solved by binary search, we aim to design the predicate so that it satisfies the condition in the main theorem.

One might wonder why we choose to use this abstraction rather than the simpler-looking algorithm we've used so far. This is because many problems can't be modeled as searching for a particular value, but it's possible to define and evaluate a predicate such as *"Is there an assignment which costs x or less?"*, when we're looking for some sort of assignment with the lowest cost. For example, the usual traveling salesman problem (TSP) looks for the *cheapest round-trip which visits every city exactly once*. Here, the *target value* is not defined as such, but we can define a predicate *"Is there a round-trip which costs x or less?"* and then apply binary search to find the smallest x which satisfies the

predicate. This is called *reducing* the original problem to a decision (yes/no) problem. Unfortunately, we know of no way of efficiently evaluating this particular predicate and so the TSP problem isn't easily solved by binary search, but many optimization problems are.

Let us now convert the simple binary search on sorted arrays described in the introduction to this abstract definition. First, let's rephrase the problem as: *"Given an array A and a target value, return the index of the first element in A equal to or greater than the target value."* Incidentally, this is more or less how lower_bound behaves in C++.

We want to find the index of the target value, thus any index into the array is a candidate solution. The search space *S* is the set of all candidate solutions, thus an interval containing all indices. Consider the predicate *"Is A[x] greater than or equal to the target value?"*. If we were to find the first x for which the predicate says *yes*, we'd get exactly what decided we were looking for in the previous paragraph.

The condition in the main theorem is satisfied because the array is sorted in ascending order: if A[x] is greater than or equal to the target value, all elements after it are surely also greater than or equal to the target value.

If we take the sample sequence from before:

0	5	13	19	22	41	55	68	72	81	98

With the search space (indices):

1	2	3	4	5	6	7	8	9	10	11

And apply our predicate (with a target value of 55) to it we get:

no	no	no	no	no	no	yes	yes	yes	yes	yes

This is a series of *no* answers followed by a series of *yes* answers, as we were expecting. Notice how index 7 (where the target value is located) is the first for which the predicate yields *yes*, so this is what our binary search will find.

Implementing the discrete algorithm

One important thing to remember before beginning to code is to settle on what the two numbers you maintain (lower and upper bound) mean. A likely answer is *a closed interval which surely contains the first x for which p(x) is true*. All of your code should then be directed at maintaining this invariant: it tells you how to properly move the bounds, which is where a bug can easily find its way in your code, if you're not careful.

Another thing you need to be careful with is how high to set the bounds. By "high" I really mean "wide" since there are two bounds to worry about. Every so often it happens that a coder concludes during coding that the bounds he or she set are wide enough, only to find a counterexample during intermission (when it's too late). Unfortunately, little helpful advice can be given here other than to always double- and triple-check your bounds! Also, since execution time increases logarithmically with the bounds, you can always set them higher, as long as it doesn't break the evaluation of the predicate. Keep your eye out for overflow errors all around, especially in calculating the median.

Now we finally get to the code which implements binary search as described in this and the previous section:

```
binary_search(lo, hi, p):
   while lo < hi:
      mid = lo + (hi-lo)/2
      if p(mid) == true:
         hi = mid
      else:
         lo = mid+1

   if p(lo) == false:
      complain                // p(x) is false for all x in S!

   return lo          // lo is the least x for which p(x) is true
```

The two crucial lines are hi = mid and lo = mid+1. When *p*(mid) is true, we can discard the second half of the search space, since the predicate is true for all elements in it (by the main theorem). However, we can not discard mid itself, since it may well be the first element for which *p* is true. This is why moving the upper bound to mid is as aggressive as we can do without introducing bugs.

In a similar vein, if *p*(mid) is false, we can discard the first half of the search space, but this time including mid. *p*(mid) is false so we don't need it

in our search space. This effectively means we can move the lower bound to mid+1.

If we wanted to find the last x for which *p*(x) is false, we would devise (using a similar rationale as above) something like:

```
// warning: there is a nasty bug in this snippet!
binary_search(lo, hi, p):
   while lo < hi:
      mid = lo + (hi-lo)/2    // note: division truncates
      if p(mid) == true:
         hi = mid-1
      else:
         lo = mid

   if p(lo) == true:
      complain                // p(x) is true for all x in S!

   return lo          // lo is the greatest x for which p(x) is false
```

You can verify that this satisfies our condition that the element we're looking for always be present in the interval (lo, hi). However, there is another problem. Consider what happens when you run this code on some search space for which the predicate gives:

no yes

The code will get stuck in a loop. It will always select the first element as mid, but then will not move the lower bound because it wants to keep the *no* in its search space. The solution is to change mid = lo + (hi-lo)/2 to mid = lo + (hi-lo+1)/2, i.e. so that it rounds up instead of down. There are other ways of getting around the problem, but this one is possibly the cleanest. Just remember to **always** test your code on a two-element set where the predicate is false for the first element and true for the second.

You may also wonder as to why mid is calculated using mid = lo + (hi-lo)/2 instead of the usual mid = (lo+hi)/2. This is to avoid another potential rounding bug: in the first case, we want the division to always round down, towards the lower bound. But division truncates, so when lo+hi would be negative, it would start rounding towards the higher bound. Coding the calculation this way ensures that the number divided is always positive and hence always rounds as we want it to. Although the bug doesn't surface when the search space consists only of positive integers or real numbers, I've decided to code it this way throughout the article for consistency.

Real numbers

Binary search can also be used on monotonic functions whose domain is the set of real numbers. Implementing binary search on reals is usually easier than on integers, because you don't need to watch out for how to move bounds:

```
binary_search(lo, hi, p):
   while we choose not to terminate:
      mid = lo + (hi-lo)/2
      if p(mid) == true:
         hi = mid
      else:
         lo = mid

   return lo // lo is close to the border between no and yes
```

Since the set of real numbers is dense, it should be clear that we usually won't be able to find the exact target value. However, we can quickly find some x such that f(x) is within some tolerance of the border between *no* and *yes*. We have two ways of deciding when to terminate: terminate when the search space gets smaller than some predetermined bound (say 10^{-12}) or do a fixed number of iterations. On TopCoder, your best bet is to just use a few hundred iterations, this will give you the best possible precision without too much thinking. 100 iterations will reduce the search space to approximately 10^{-30} of its initial size, which should be enough for most (if not all) problems.

If you need to do as few iterations as possible, you can terminate when the interval gets small, but try to do a relative comparison of the bounds, not just an absolute one. The reason for this is that doubles can never give you more than 15 decimal digits of precision so if the search space contains large numbers (say on the order of billions), you can never get an absolute difference of less than 10^{-7}.

Example

At this point I will show how all this talk can be used to solve a TopCoder problem. For this I have chosen a moderately difficult problem, FairWorkload, which was the division 1 level 2 problem in SRM 169.

In the problem, a number of workers need to examine a number of filing cabinets. The cabinets are not all of the same size and we are told for each cabinet how many folders it contains. We are asked to find an assignment such that each worker gets a sequential series of cabinets to go through and that it minimizes the maximum amount of folders that a worker would have to look through.

After getting familiar with the problem, a touch of creativity is required. Imagine that we have an unlimited number of workers at our disposal.

The crucial observation is that, for some number *MAX*, we can calculate the minimum number of workers needed so that each worker has to examine no more than *MAX* folders (if this is possible). Let's see how we'd do that. Some worker needs to examine the first cabinet so we assign any worker to it. But, since the cabinets must be assigned in sequential order (a worker cannot examine cabinets 1 and 3 without examining 2 as well), it's always optimal to assign him to the second cabinet as well, if this does not take him over the limit we introduced (*MAX*). If it would take him over the limit, we conclude that his work is done and assign a new worker to the second cabinet. We proceed in a similar manner until all the cabinets have been assigned and assert that we've used the minimum number of workers possible, with the artificial limit we introduced. Note here that the number of workers is inversely proportional to *MAX*: the higher we set our limit, the fewer workers we will need.

Now, if you go back and carefully examine what we're asked for in the problem statement, you can see that we are really asked for the smallest *MAX* such that the number of workers required is less than or equal to the number of workers available. With that in mind, we're almost done, we just need to connect the dots and see how all of this fits in the frame we've laid out for solving problems using binary search.

With the problem rephrased to fit our needs better, we can now examine the predicate *Can the workload be spread so that each worker has to examine no more than x folders, with the limited number of workers available?* We can use the described greedy algorithm to efficiently evaluate this predicate for any x. This concludes the first part of building a binary search solution, we now just have to prove that the condition in the main theorem is satisfied. But observe that increasing x actually relaxes the limit on the maximum workload, so we can only need the same number of workers or fewer, not more. Thus, if the predicate says *yes* for some x, it will also say *yes* for all larger x.

To wrap it up, here's an STL-driven snippet which solves the problem:

```
int getMostWork( vector  folders, int workers ) {
   int n = folders.size();
   int lo = *max_element( folders.begin(), folders.end() );
   int hi = accumulate( folders.begin(), folders.end(), 0 );

   while ( lo < hi ) {
      int x = lo + (hi-lo)/2;

      int required = 1, current_load = 0;
      for ( int i=0; i<n; ++i ) {
         if ( current_load + folders[i] <= x ) {
            // the current worker can handle it
            current_load += folders[i];
         }
         else {
            // assign next worker
            ++required;
            current_load = folders[i];
         }
      }

      if ( required <= workers )
         hi = x;
      else
         lo = x+1;
   }

   return lo;
}
```

Note the carefully chosen lower and upper bounds: you could replace the upper bound with any sufficiently large integer, but the lower bound must not to be less than the largest cabinet to avoid the situation where a single cabinet would be too large for any worker, a case which would not be correctly handled by the predicate. An alternative would be to set the lower bound to zero, then handle too small x's as a special case in the predicate.

To verify that the solution doesn't lock up, I used a small no/yes example with folders={1,1} and workers=1.

The overall complexity of the solution is $O(n \log SIZE)$, where *SIZE* is the size of the search space. This is very fast.

As you see, we used a greedy algorithm to evaluate the predicate. In other problems, evaluating the predicate can come down to anything from a simple math expression to finding a maximum cardinality matching in a bipartite graph.

Conclusion

If you've gotten this far without giving up, you should be ready to solve anything that can be solved with binary search. Try to keep a few things in mind:

- Design a predicate which can be efficiently evaluated and so that binary search can be applied

- Decide on what you're looking for and code so that the search space always contains that (if it exists)
- If the search space consists only of integers, test your algorithm on a two-element set to be sure it doesn't lock up
- Verify that the lower and upper bounds are not overly constrained: it's usually better to relax them as long as it doesn't break the predicate

Here are a few problems that can be solved using binary search:

Simple
AutoLoan - SRM 258
SortEstimate - SRM 230

Moderate
UnionOfIntervals - SRM 277
Mortgage - SRM 189
FairWorkload - SRM 169
HairCuts - SRM 261

Harder
PackingShapes - SRM 270
RemoteRover - SRM 235
NegativePhotoresist - SRM 210
WorldPeace - SRM 204
UnitsMoving - SRM 278
Parking - SRM 236
SquareFree - SRM 190
Flags - SRM 147

Introduction

Most of the optimizations that go into TopCoder contests are high-level; that is, they affect the algorithm rather than the implementation. However, one of the most useful and effective low-level optimizations is bit manipulation, or using the bits of an integer to represent a set. Not only does it produce an order-of-magnitude improvement in both speed and size, it can often simplify code at the same time.

I'll start by briefly recapping the basics, before going on to cover more advanced techniques.

The basics

At the heart of bit manipulation are the bit-wise operators & (and), | (or), ~ (not) and ^ (xor). The first three you should already be familiar with in their boolean forms (&&, || and !). As a reminder, here are the truth tables:

A	B	!A	A && B	A || B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

The bit-wise versions of the operations are the same, except that instead of interpreting their arguments as true or false, they operate on each bit of the arguments. Thus, if A is 1010 and B is 1100, then

- A & B = 1000
- A | B = 1110
- A ^ B = 0110
- ~A = 11110101 (the number of 1's depends on the type of A).

The other two operators we will need are the shift operators a << b and a >> b. The former shifts all the bits in a to the left by b positions; the latter does the same but shifts right. For non-negative values (which are the only ones we're interested in), the newly exposed bits are filled with zeros. You can think of left-shifting by b as multiplication by 2^b and right-shifting as integer division by 2^b. The most common use for shifting is to access a particular bit, for example, 1 << x is a binary number with bit x set and the others clear (bits are almost always counted from the right-most/least-significant bit, which is numbered 0).

In general, we will use an integer to represent a set on a domain of up to 32 values (or 64, using a 64-bit integer), with a 1 bit representing a member that is present and a 0 bit one that is absent. Then the following operations are quite straightforward, where ALL_BITS is a number with 1's for all bits corresponding to the elements of the domain:

Set union
```
A | B
```
Set intersection
```
A & B
```
Set subtraction
```
A & ~B
```
Set negation
```
ALL_BITS ^ A
```
Set bit
```
A |= 1 << bit
```
Clear bit
```
A &= ~(1 << bit)
```
Test bit
```
(A & 1 << bit) != 0
```

Extracting every last bit

In this section I'll consider the problems of finding the highest and lowest 1 bit in a number. These are basic operations for splitting a set into its elements.

Finding the lowest set bit turns out to be surprisingly easy, with the right combination of bitwise and arithmetic operators. Suppose we wish to find the lowest set bit of x (which is known to be non-zero). If we subtract 1 from x then this bit is cleared, but all the other one bits in x remain set. Thus, x & ~(x - 1) consists of only the lowest set bit of x. However, this only tells us the bit value, not the index of the bit.

If we want the index of the highest or lowest bit, the obvious approach is simply to loop through the bits (upwards or downwards) until we find

one that is set. At first glance this sounds slow, since it does not take advantage of the bit-packing at all. However, if all 2^N subsets of the N-element domain are equally likely, then the loop will take only two iterations on average, and this is actually the fastest method.

The 386 introduced CPU instructions for bit scanning: BSF (bit scan forward) and BSR (bit scan reverse). GCC exposes these instructions through the built-in functions __builtin_ctz (count trailing zeros) and __builtin_clz (count leading zeros). These are the most convenient way to find bit indices for C++ programmers in TopCoder. Be warned though: the return value is *undefined* for an argument of zero.

Finally, there is a portable method that performs well in cases where the looping solution would require many iterations. Use each byte of the 4- or 8-byte integer to index a precomputed 256-entry table that stores the index of the highest (lowest) set bit in that byte. The highest (lowest) bit of the integer is then the maximum (minimum) of the table entries. This method is only mentioned for completeness, and the performance gain is unlikely to justify its use in a TopCoder match.

Counting out the bits

One can easily check if a number is a power of 2: clear the lowest 1 bit (see above) and check if the result is 0. However, sometimes it is necessary to know how many bits are set, and this is more difficult.

GCC has a function called __builtin_popcount which does precisely this. However, unlike __builtin_ctz, it does not translate into a hardware instruction (at least on x86). Instead, it uses a table-based method similar to the one described above for bit searches. It is nevertheless quite efficient and also extremely convenient.

Users of other languages do not have this option (although they could re-implement it). If a number is expected to have very few 1 bits, an alternative is to repeatedly extract the lowest 1 bit and clear it.

All the subsets

A big advantage of bit manipulation is that it is trivial to iterate over all the subsets of an N-element set: every N-bit value represents some subset. Even better, if A is a subset of B then the number representing A is less than that representing B, which is convenient for some dynamic programming solutions.

It is also possible to iterate over all the subsets of a particular subset (represented by a bit pattern), provided that you don't mind visiting them in reverse order (if this is problematic, put them in a list as they're generated, then walk the list backwards). The trick is similar to that for finding the lowest bit in a number. If we subtract 1 from a subset, then the lowest set element is cleared, and every lower element is set. However, we only want to set those lower elements that are in the superset. So the iteration step is just i = (i - 1) & superset.

Even a bit wrong scores zero

There are a few mistakes that are very easy to make when performing bit manipulations. Watch out for them in your code.

1. When executing shift instructions for a << b, the x86 architecture uses only the bottom 5 bits of b (6 for 64-bit integers). This means that shifting left (or right) by 32 does nothing, rather than clearing all the bits. This behaviour is also specified by the Java and C# language standards; C99 says that shifting by at least the size of the value gives an undefined result. Historical trivia: the 8086 used the full shift register, and the change in behaviour was often used to detect newer processors.
2. The & and | operators have lower precedence than comparison operators. That means that x & 3 == 1 is interpreted as x & (3 == 1), which is probably not what you want.
3. If you want to write completely portable C/C++ code, be sure to use unsigned types, particularly if you plan to use the top-most bit. C99 says that shift operations on negative values are undefined. Java only has signed types: >> will sign-extend values (which is probably *not* what you want), but the Java-specific operator >>> will shift in zeros.

Cute tricks

There are a few other tricks that can be done with bit manipulation. They're good for amazing your friends, but generally not worth the effect to use in practice.

Reversing the bits in an integer

```
x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);
```

As an exercise, see if you can adapt this to count the number of bits in a word.

Iterate through all k-element subsets of {0, 1, … N-1}

```
int s = (1 << k) - 1;
while (!(s & 1 << N))
{
    // do stuff with s
    int lo = s & ~(s - 1);        // lowest one bit
    int lz = (s + lo) & ~s;       // lowest zero bit above lo
    s |= lz;                      // add lz to the set
```

```
        s &= ~(lz - 1);              // reset bits below lz
        s |= (lz / lo / 2) - 1;      // put back right number of bits at end
    }
```

In C, the last line can be written as s |= (lz >> ffs(lo)) - 1 to avoid the division.

Evaluate x ? y : -y, where x is 0 or 1

(-x ^ y) + x

This works on a twos-complement architecture (which is almost any machine you find today), where negation is done by inverting all the bits then adding 1. Note that on i686 and above, the original expression can be evaluated just as efficiently (i.e., without branches) due to the CMOVE (conditional move) instruction.

Sample problems

TCCC 2006, Round 1B Medium

For each city, keep a bit-set of the neighbouring cities. Once the part-building factories have been chosen (recursively), ANDing together these bit-sets will give a bit-set which describes the possible locations of the part-assembly factories. If this bit-set has k bits, then there are ${}^{k}C_{m}$ ways to allocate the part-assembly factories.

TCO 2006, Round 1 Easy

The small number of nodes strongly suggests that this is done by considering all possible subsets. For every possible subset we consider two possibilities: either the smallest-numbered node does not communicate at all, in which case we refer back to the subset that excludes it, or it communicates with some node, in which case we refer back to the subset that excludes both of these nodes. The resulting code is extremely short:

```
static int dp[1 << 18];

int SeparateConnections::howMany(vector <string> mat)
{
    int N = mat.size();
    int N2 = 1 << N;
    dp[0] = 0;
    for (int i = 1; i < N2; i++)
    {
        int bot = i & ~(i - 1);
        int use = __builtin_ctz(bot);
        dp[i] = dp[i ^ bot];
        for (int j = use + 1; j < N; j++)
            if ((i & (1 << j)) && mat[use][j] == 'Y')
                dp[i] = max(dp[i], dp[i ^ bot ^ (1 << j)] + 2);
    }
    return dp[N2 - 1];
}
```

SRM 308, Division 1 Medium

The board contains 36 squares and the draughts are indistinguishable, so the possible positions can be encoded into 64-bit integers. The first step is to enumerate all the legal moves. Any legal move can be encoded using three bit-fields: a *before* state, an *after* state and a *mask*, which defines which parts of the before state are significant. The move can be made from the current state if (current & mask) == before; if it is made, the new state is (current & ~mask) | after.

SRM 320, Division 1 Hard

The constraints tell us that there are at most 8 columns (if there are more, we can swap rows and columns), so it is feasible to consider every possible way to lay out a row. Once we have this information, we can solve the remainder of the problem (refer to the match editorial for details). We thus need a list of all n-bit integers which do not have two adjacent 1 bits, and we also need to know how many 1 bits there are in each such row. Here is my code for this:

```
for (int i = 0; i < (1 << n); i++)
{
    if (i & (i << 1)) continue;
    pg.push_back(i);
    pgb.push_back(__builtin_popcount(i));
}
```

Introduction

The problem of finding the Lowest Common Ancestor (LCA) of a pair of nodes in a rooted tree has been studied more carefully in the second part of the 20th century and now is fairly basic in algorithmic graph theory. This problem is interesting not only for the tricky algorithms that can be used to solve it, but for its numerous applications in string processing and computational biology, for example, where LCA is used with suffix trees or other tree-like structures. Harel and Tarjan were the first to study this problem more attentively and they showed that after linear preprocessing of the input tree LCA, queries can be answered in constant time. Their work has since been extended, and this tutorial will present many interesting approaches that can be used in other kinds of problems as well.

Let's consider a less abstract example of LCA: the tree of life. It's a well-known fact that the current habitants of Earth evolved from other species. This evolving structure can be represented as a tree, in which nodes represent species, and the sons of some node represent the directly evolved species. Now species with similar characteristics are divided into groups. By finding the LCA of some nodes in this tree we can actually find the common parent of two species, and we can determine that the similar characteristics they share are inherited from that parent.

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. We will see later that the LCA problem can be reduced to a restricted version of an RMQ problem, in which consecutive array elements differ by exactly 1.

However, RMQs are not only used with LCA. They have an important role in string preprocessing, where they are used with suffix arrays (a new data structure that supports string searches almost as fast as suffix trees, but uses less memory and less coding effort).

In this tutorial we will first talk about RMQ. We will present many approaches that solve the problem -- some slower but easier to code, and others faster. In the second part we will talk about the strong relation between LCA and RMQ. First we will review two easy approaches for LCA that don't use RMQ; then show that the RMQ and LCA problems are equivalent; and, at the end, we'll look at how the RMQ problem can be reduced to its restricted version, as well as show a fast algorithm for this particular case.

Notations

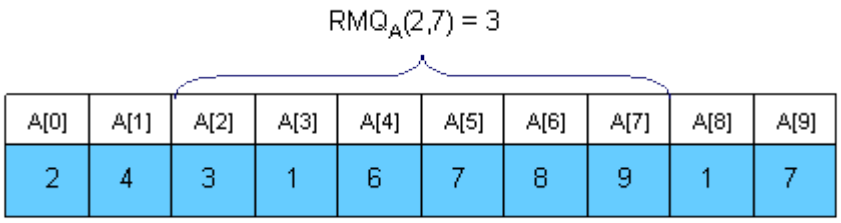
Suppose that an algorithm has preprocessing time **f(n)** and query time **g(n)**. The notation for the overall complexity for the algorithm is **<f(n), g(n)>**.

We will note the position of the element with the minimum value in some array **A** between indices **i** and **j** with $\mathbf{RMQ_A(i, j)}$.

The furthest node from the root that is an ancestor of both **u** and **v** in some rooted tree **T** is $\mathbf{LCA_T(u, v)}$.

Range Minimum Query(RMQ)

Given an array **A[0, N-1]** find the position of the element with the minimum value between two given indices.



Trivial algorithms for RMQ

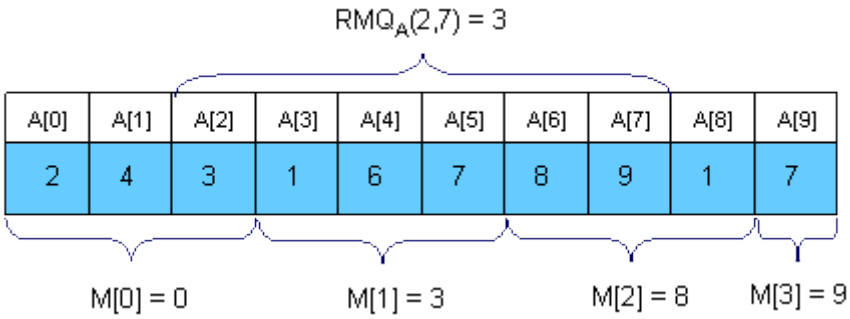
For every pair of indices (i, j) store the value of $RMQ_A(i, j)$ in a table M[0, N-1][0, N-1]. Trivial computation will lead us to an <$O(N^3)$, O(1)> complexity. However, by using an easy dynamic programming approach we can reduce the complexity to <$O(N^2)$, O(1)>. The preprocessing function will look something like this:

```
void process1(int M[MAXN][MAXN], int A[MAXN], int N)
{
    int i, j;
    for (i =0; i < N; i++)
        M[i][i] = i;
    for (i = 0; i < N; i++)
        for (j = i + 1; j < N; j++)
            if (A[M[i][j - 1]] < A[j])
                M[i][j] = M[i][j - 1];
            else
                M[i][j] = j;
}
```

This trivial algorithm is quite slow and uses **$O(N^2)$** memory, so it won't work for large cases.

An <O(N), O(sqrt(N))> solution

An interesting idea is to split the vector in **sqrt(N)** pieces. We will keep in a vector **M[0, sqrt(N)-1]** the position for the minimum value for each section. **M** can be easily preprocessed in **O(N)**. Here is an example:

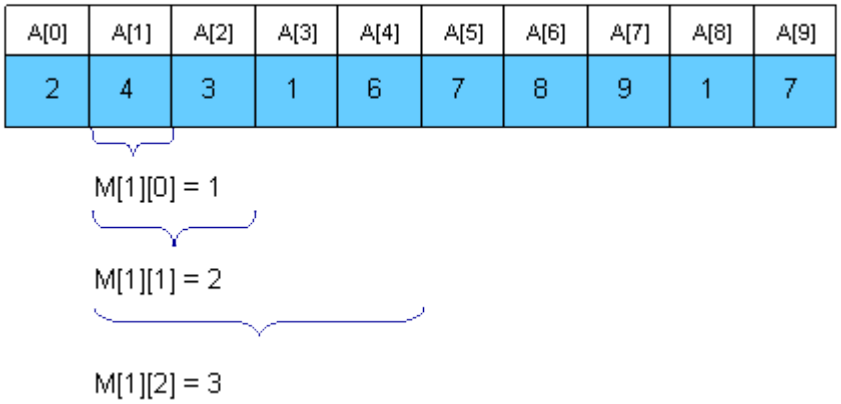


Now let's see how can we compute **RMQ$_A$(i, j)**. The idea is to get the overall minimum from the **sqrt(N)** sections that lie inside the interval, and from the end and the beginning of the first and the last sections that intersect the bounds of the interval. To get **RMQ$_A$(2,7)** in the above example we should compare **A[2]**, **A[M[1]]**, **A[6]** and **A[7]** and get the position of the minimum value. It's easy to see that this algorithm doesn't make more than **3 * sqrt(N)** operations per query.

The main advantages of this approach are that is to quick to code (a plus for TopCoder-style competitions) and that you can adapt it to the dynamic version of the problem (where you can change the elements of the array between queries).

Sparse Table (ST) algorithm

A better approach is to preprocess **RMQ** for sub arrays of length **2^k** using dynamic programming. We will keep an array **M[0, N-1][0, logN]** where **M[i][j]** is the index of the minimum value in the sub array starting at **i** having length **2^j**. Here is an example:



For computing **M[i][j]** we must search for the minimum value in the first and second half of the interval. It's obvious that the small pieces have **2^{j-1}** length, so the recurrence is:

$$M[i][j] = \begin{cases} M[i][j-1], & A[M[i][j-1]] <= A[M[i+2^{j-1}-1][j-1]] \\ M[i+2^{j-1}-1][j-1], & otherwise \end{cases}$$

The preprocessing function will look something like this:

```
void process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)
{
    int i, j;

//initialize M for the intervals with length 1
    for (i = 0; i < N; i++)
        M[i][0] = i;
//compute values from smaller to bigger intervals
    for (j = 1; 1 << j <= N; j++)
        for (i = 0; i + (1 << j) - 1 < N; i++)
            if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
                M[i][j] = M[i][j - 1];
            else
                M[i][j] = M[i + (1 << (j - 1))][j - 1];
```

```
}
```

Once we have these values preprocessed, let's show how we can use them to calculate **$RMQ_A(i, j)$**. The idea is to select two blocks that entirely cover the interval **[i..j]** and find the minimum between them. Let **k = [log(j - i + 1)]**. For computing **$RMQ_A(i, j)$** we can use the following formula:

$$RMQ_A(i,j) = \begin{cases} M[i][k], & A[M[i][k]] <= A[M[j-2^k+1][k]] \\ M[j-2^k+1][k], & otherwise \end{cases}$$

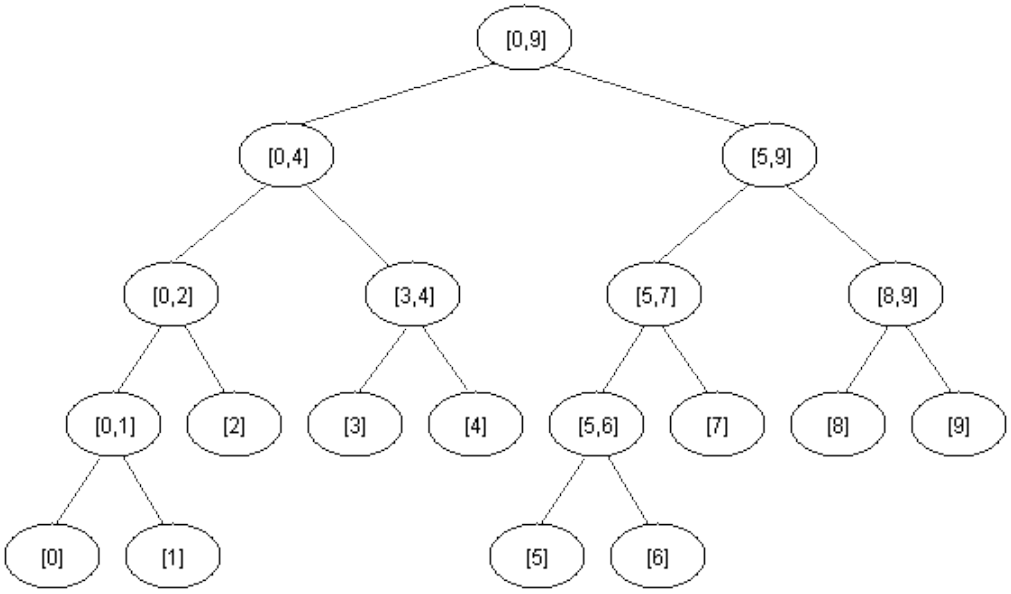
So, the overall complexity of the algorithm is **<O(N logN), O(1)>**.

Segment trees

For solving the RMQ problem we can also use segment trees. A segment tree is a heap-like data structure that can be used for making update/query operations upon array intervals in logarithmical time. We define the segment tree for the interval **[i, j]** in the following recursive manner:

- the first node will hold the information for the interval **[i, j]**
- if i<j the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

Notice that the height of a segment tree for an interval with **N** elements is **[logN] + 1**. Here is how a segment tree for the interval **[0, 9]** would look like:



The segment tree has the same structure as a heap, so if we have a node numbered **x** that is not a leaf the left son of **x** is **2*x** and the right son **2*x+1**.

For solving the RMQ problem using segment trees we should use an array **$M[1, 2 * 2^{[logN]+1}]$** where **M[i]** holds the minimum value position in the interval assigned to node **i**. At the beginning all elements in **M** should be **-1**. The tree should be initialized with the following function (**b** and **e** are the bounds of the current interval):

```
  void initialize(intnode, int b, int e, int M[MAXIND], int A[MAXN], int N)
  {
      if (b == e)
          M[node] = b;
      else
        {
  //compute the values in the left and right subtrees
          initialize(2 * node, b, (b + e) / 2, M, A, N);
          initialize(2 * node + 1, (b + e) / 2 + 1, e, M, A, N);
  //search for the minimum value in the first and
  //second half of the interval
          if (A[M[2 * node]] <= A[M[2 * node + 1]])
              M[node] = M[2 * node];
          else
              M[node] = M[2 * node + 1];
      }
  }
```

The function above reflects the way the tree is constructed. When calculating the minimum position for some interval we should look at the values of the sons. You should call the function with **node = 1**, **b = 0** and **e = N-1**.

We can now start making queries. If we want to find the position of the minimum value in some interval **[i, j]** we should use the next easy function:

```
int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int j)
  {
      int p1, p2;


  //if the current interval doesn't intersect
  //the query interval return -1
      if (i > e || j < b)
          return -1;

  //if the current interval is included in
  //the query interval return M[node]
      if (b >= i && e <= j)
          return M[node];

  //compute the minimum position in the
  //left and right part of the interval
      p1 = query(2 * node, b, (b + e) / 2, M, A, i, j);
      p2 = query(2 * node + 1, (b + e) / 2 + 1, e, M, A, i, j);

  //return the position where the overall
  //minimum is
      if (p1 == -1)
          return M[node] = p2;
      if (p2 == -1)
          return M[node] = p1;
      if (A[p1] <= A[p2])
          return M[node] = p1;
      return M[node] = p2;

  }
```

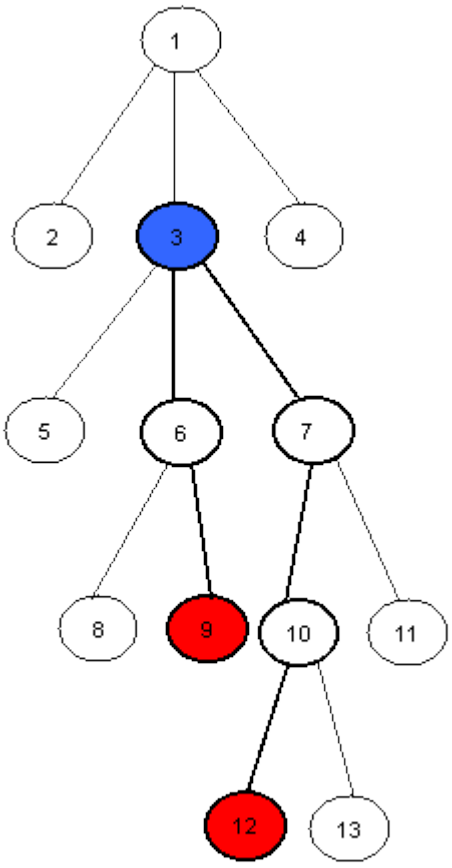
You should call this function with **node = 1**, **b = 0** and **e = N - 1**, because the interval assigned to the first node is **[0, N-1]**.

It's easy to see that any query is done in **O(log N)**. Notice that we stop when we reach completely in/out intervals, so our path in the tree should split only one time.

Using segment trees we get an **<O(N), O(logN)>** algorithm. Segment trees are very powerful, not only because they can be used for RMQ. They are a very flexible data structure, can solve even the dynamic version of RMQ problem, and have numerous applications in range searching problems.

Lowest Common Ancestor (LCA)

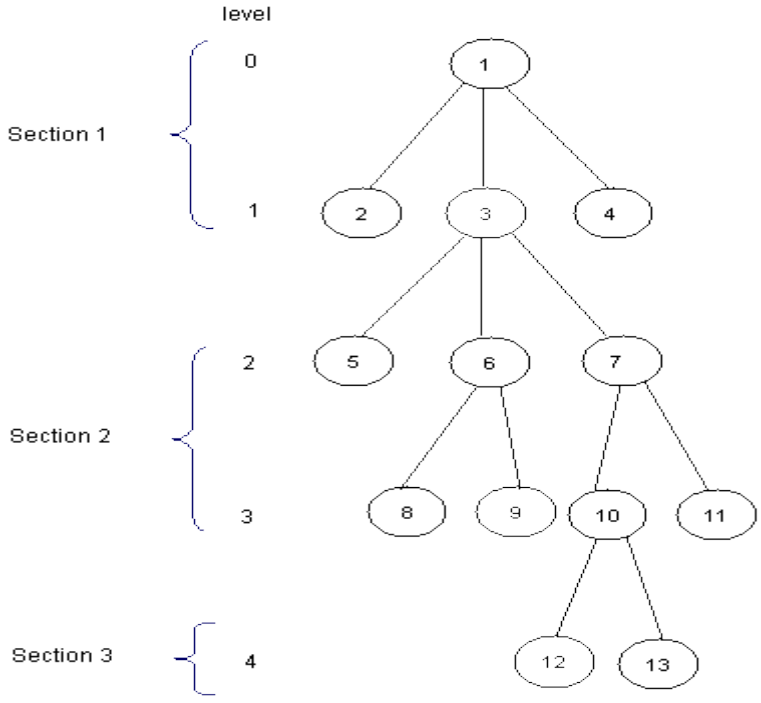
Given a rooted tree **T** and two nodes **u** and **v,** find the furthest node from the root that is an ancestor for both **u** and **v**. Here is an example (the root of the tree will be node 1 for all examples in this editorial):



$LCA_T(9, 12) = 3$

An <O(N), O(sqrt(N))> solution

Dividing our input into equal-sized parts proves to be an interesting way to solve the RMQ problem. This method can be adapted for the LCA problem as well. The idea is to split the tree in **sqrt(H)** parts, were **H** is the height of the tree. Thus, the first section will contain the levels numbered from **0 to sqrt(H) - 1**, the second will contain the levels numbered from **sqrt(H) to 2 * sqrt(H) - 1**, and so on. Here is how the tree in the example should be divided:



Now, for each node, we should know the ancestor that is situated on the last level of the upper next section. We will preprocess this values in an array **P[1, MAXN]**. Here is how **P** should look like for the tree in the example (for simplity, for every node **i** in the first section let **P[i] = 1**):

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]	P[9]	P[10]	P[11]	P[12]	P[13]
1	1	1	1	3	3	3	3	3	3	3	10	10

Notice that for the nodes situated on the levels that are the first ones in some sections, **P[i] = T[i]**. We can preprocess **P** using a depth first search (**T[i]** is the father of node **i** in the tree, **nr** is **[sqrt(H)]** and **L[i]** is the level of the node **i**):

```
 void dfs(int node, int T[MAXN], int N, int P[MAXN], int L[MAXN], int nr)  {
      int k;

  //if node is situated in the first
  //section then P[node] = 1
  //if node is situated at the beginning
  //of some section then P[node] = T[node]
  //if none of those two cases occurs, then
  //P[node] = P[T[node]]
      if (L[node] < nr)
          P[node] = 1;
      else
          if(!(L[node] % nr))
              P[node] = T[node];
          else
              P[node] = P[T[node]];

     for each son k of node
         dfs(k, T, N, P, L, nr);
  }
```

Now, we can easily make queries. For finding **LCA(x, y)** we we will first find in what section it lays, and then trivially compute it. Here is the code:

```
  int LCA(int T[MAXN], int P[MAXN], int L[MAXN], int x, int y)
  {
  //as long as the node in the next section of
  //x and y is not one common ancestor
  //we get the node situated on the smaller
  //lever closer
      while (P[x] != P[y])
          if (L[x] > L[y])
             x = P[x];
          else
                y = P[y];

  //now they are in the same section, so we trivially compute the LCA
      while (x != y)
          if (L[x] > L[y])
             x = T[x];
          else
             y = T[y];
      return x;
  }
```

This function makes at most **2 * sqrt(H)** operations. Using this approach we get an **<O(N), O(sqrt(H))>** algorithm, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity is **<O(N), O(sqrt(N))>**. The main advantage of this algorithm is quick coding (an average Division 1 coder shouldn't need more than 15 minutes to code it).

Another easy solution in <O(N logN, O(logN)>

If we need a faster solution for this problem we could use dynamic programming. First, let's compute a table P[1,N][1,logN] where P[i][j] is the 2^j-th ancestor of i. For computing this value we may use the following recursion:

$$P[i][j] = \begin{cases} T[i], & j = 0 \\ P[P[i][j-1]][j-1], & j > 0 \end{cases}$$

The preprocessing function should look like this:

```
void process3(int N, int T[MAXN], int P[MAXN][LOGMAXN])
 {
     int i, j;

 //we initialize every element in P with -1
     for (i = 0; i < N; i++)
         for (j = 0; 1 << j < N; j++)
             P[i][j] = -1;

 //the first ancestor of every node i is T[i]
     for (i = 0; i < N; i++)
         P[i][0] = T[i];

 //bottom up dynamic programing
     for (j = 1; 1 << j < N; j++)
        for (i = 0; i < N; i++)
            if (P[i][j - 1] != -1)
                P[i][j] = P[P[i][j - 1]][j - 1];
 }
```

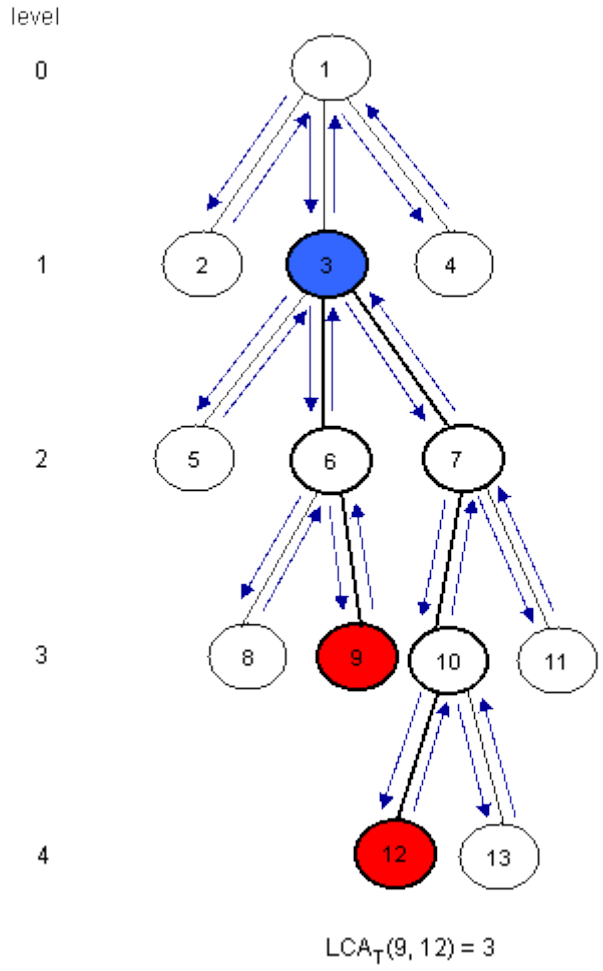
This takes **O(N logN)** time and space. Now let's see how we can make queries. Let **L[i]** be the level of node **i** in the tree. We must observe that if **p** and **q** are on the same level in the tree we can compute **LCA(p, q)** using a meta-binary search. So, for every power **j** of **2 (**between **log(L[p])** and **0,** in descending order), if **P[p][j] != P[q][j]** then we know that **LCA(p, q)** is on a higher level and we will continue searching for **LCA(p = P[p][j], q = P[q][j])**. At the end, both **p** and **q** will have the same father, so return **T[p]**. Let's see what happens if **L[p] != L[q]**. Assume, without loss of generality, that **L[p] < L[q]**. We can use the same meta-binary search for finding the ancestor of **p** situated on the same level with **q**, and then we can compute the **LCA** as described below. Here is how the query function should look:

```
int query(int N, int P[MAXN][LOGMAXN], int T[MAXN],
 int L[MAXN], int p, int q)
 {
     int tmp, log, i;

 //if p is situated on a higher level than q then we swap them
     if (L[p] < L[q])
         tmp = p, p = q, q = tmp;

 //we compute the value of [log(L[p)]
     for (log = 1; 1 << log <= L[p]; log++);
     log--;

 //we find the ancestor of node p situated on the same level
 //with q using the values in P
     for (i = log; i >= 0; i--)
         if (L[p] - (1 << i) >= L[q])
             p = P[p][i];

     if (p == q)
         return p;

 //we compute LCA(p, q) using the values in P
     for (i = log; i >= 0; i--)
         if (P[p][i] != -1 && P[p][i] != P[q][i])
             p = P[p][i], q = P[q][i];

     return T[p];
 }
```

Now, we can see that this function makes at most **2 * log(H)** operations, where **H** is the height of the tree. In the worst case **H = N**, so the overall complexity of this algorithm is **<O(N logN), O(logN)>**. This solution is easy to code too, and it's faster than the previous one.

Reduction from LCA to RMQ

Now, let's show how we can use RMQ for computing LCA queries. Actually, we will reduce the LCA problem to RMQ in linear time, so every algorithm that solves the RMQ problem will solve the LCA problem too. Let's show how this reduction can be done using an example:

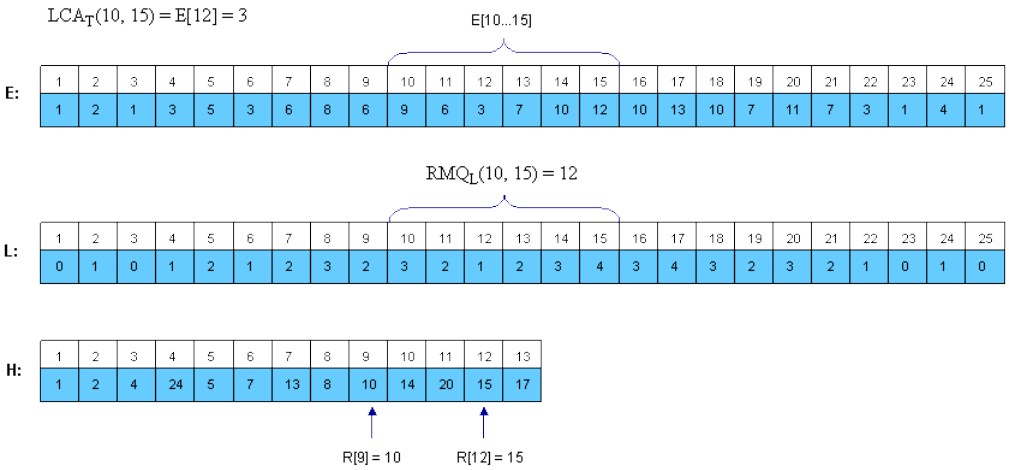


click to enlarge image

Notice that **$LCA_T(u, v)$** is the closest node from the root encountered between the visits of **u** and **v** during a depth first search of **T**. So, we can consider all nodes between any two indices of **u** and **v** in the Euler Tour of the tree and then find the node situated on the smallest level between them. For this, we must build three arrays:

- **E[1, 2*N-1]** - the nodes visited in an Euler Tour of **T**; **E[i]** is the label of **i-th** visited node in the tour
- **L[1, 2*N-1]** - the levels of the nodes visited in the Euler Tour; **L[i]** is the level of node **E[i]**
- **H[1, N]** - **H[i]** is the index of the first occurrence of node **i** in **E** (any occurrence would be good, so it's not bad if we consider the first one)

Assume that **H[u] < H[v]** (otherwise you must swap **u** and **v**). It's easy to see that the nodes between the first occurrence of **u** and the first occurrence of **v** are **E[H[u]...H[v]]**. Now, we must find the node situated on the smallest level. For this, we can use **RMQ**. So, **$LCA_T(u, v) = E[RMQ_L(H[u], H[v])]$** (remember that RMQ returns the index). Here is how **E**, **L** and **H** should look for the example:



click to enlarge image

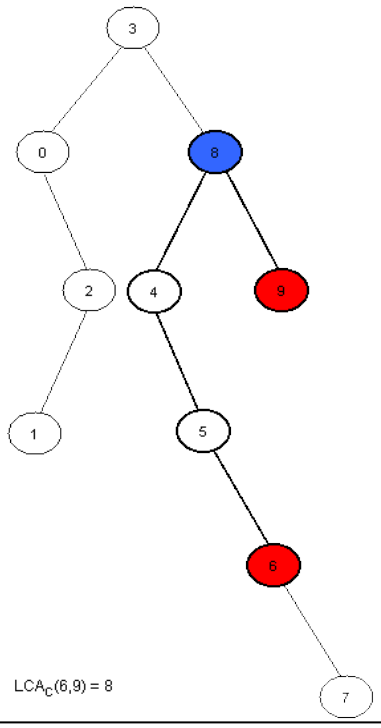
Notice that consecutive elements in L differ by exactly 1.

From RMQ to LCA

We have shown that the LCA problem can be reduced to RMQ in linear time. Here we will show how we can reduce the RMQ problem to LCA. This means that we actually can reduce the general RMQ to the restricted version of the problem (where consecutive elements in the array differ by exactly 1). For this we should use cartesian trees.

A Cartesian Tree of an array **A[0, N - 1]** is a binary tree **C(A)** whose root is a minimum element of **A**, labeled with the position **i** of this minimum. The left child of the root is the Cartesian Tree of **A[0, i - 1]** if **i > 0**, otherwise there's no child. The right child is defined similary for **A[i + 1, N - 1]**. Note that the Cartesian Tree is not necessarily unique if **A** contains equal elements. In this tutorial the first appearance of the minimum value will be used, thus the Cartesian Tree will be unique. It's easy to see now that **$RMQ_A(i, j) = LCA_C(i, j)$**.

Here is an example:



Now we only have to compute **C(A)** in linear time. This can be done using a stack. At the beginning the stack is empty. We will then insert the elements of **A** in the stack. At the **i-th** step **A[i]** will be added next to the last element in the stack that has a smaller or equal value to **A[i]**, and all the greater elements will be removed. The element that was in the stack on the position of **A[i]** before the insertion was done will become the left son of **i**, and **A[i]** will become the right son of the smaller element behind him. At every step the first element in the stack is the root of the cartesian tree. It's easier to build the tree if the stack will hold the indexes of the elements, and not their value.

Here is how the stack will look at each step for the example above:

Step	Stack	Modifications made in the tree
0	0	0 is the only node in the tree.
1	0 1	1 is added at the end of the stack. Now, 1 is the right son of 0.
2	0 2	2 is added next to 0, and 1 is removed (A[2] < A[1]). Now, 2 is the right son of 0 and the left son of 2 is 1.
3	3	A[3] is the smallest element in the vector so far, so all elements in the stack will be removed and 3 will become the root of the tree. The left child of 3 is 0.
4	3 4	4 is added next to 3, and the right son of 3 is 4.
5	3 4 5	5 is added next to 4, and the right son of 4 is 5.
6	3 4 5 6	6 is added next to 5, and the right son of 5 is 6.
7	3 4 5 6 7	7 is added next to 6, and the right son of 6 is 7.
8	3 8	8 is added next to 3, and all greater elements are removed. 8 is now the right child of 3 and the left child of 8 is 4.
9	3 8 9	9 is added next to 8, and the right son of 8 is 9.

Note that every element in **A** is only added once and removed at most once, so the complexity of this algorithm is **O(N)**. Here is how the tree-processing function will look:

```
void computeTree(int A[MAXN], int N, int T[MAXN])
{
    int st[MAXN], i, k, top = -1;

//we start with an empty stack
//at step i we insert A[i] in the stack
    for (i = 0; i < N; i++)
    {
//compute the position of the first element that is
//equal or smaller than A[i]
        k = top;
        while (k >= 0 && A[st[k]] > A[i])
            k--;
//we modify the tree as explained above
       if (k != -1)
            T[i] = st[k];
       if (k < top)
            T[st[k + 1]] = i;
//we insert A[i] in the stack and remove
//any bigger elements
        st[++k] = i;
        top = k;
    }
//the first element in the stack is the root of
//the tree, so it has no father
    T[st[0]] = -1;
}
```

An<O(N), O(1)> algorithm for the restricted RMQ

Now we know that the general RMQ problem can be reduced to the restricted version using LCA. Here, consecutive elements in the array differ by exactly 1. We can use this and give a fast **<O(N), O(1)>** algorithm. From now we will solve the RMQ problem for an array **A[0, N - 1]** where **|A[i] - A[i + 1]| = 1**, **i = [1, N - 1]**. We transform **A** in a binary array with **N-1** elements, where **A[i] = A[i] - A[i + 1]**. It's obvious that elements in **A** can be just **+1** or **-1**. Notice that the old value of **A[i]** is now the sum of **A[1]**, **A[2]** .. **A[i]** plus the old **A[0]**. However, we won't need the old values from now on.

To solve this restricted version of the problem we need to partition **A** into blocks of size **l = [(log N) / 2]**. Let **A'[i]** be the minimum value for the **i-th** block in **A** and **B[i]** be the position of this minimum value in **A**. Both **A** and **B** are **N/l** long. Now, we preprocess **A'** using the ST algorithm described in Section1. This will take **O(N/l * log(N/l)) = O(N)** time and space. After this preprocessing we can make queries that span over several blocks in **O(1)**. It remains now to show how the in-block queries can be made. Note that the length of a block is **l = [(log N) / 2]**, which is quite small. Also, note that **A** is a binary array. The total number of binary arrays of size **l** is 2^l**=sqrt(N)**. So, for each binary block of size **l** we need to lock up in a table **P** the value for RMQ between every pair of indices. This can be trivially computed in **O(sqrt(N)***l^2**)=O(N)** time and space. To index table **P**, preprocess the type of each block in **A** and store it in array **T[1, N/l]**. The block type is a binary number obtained by replacing -1 with 0 and +1 with 1.

Now, to answer $\mathbf{RMQ_A(i, j)}$ we have two cases:

- **i** and **j** are in the same block, so we use the value computed in **P** and **T**
- **i** and **j** are in different blocks, so we compute three values: the minimum from **i** to the end of **i's** block using **P** and **T**, the minimum of all blocks between **i's** and j**'s** block using precomputed queries on **A'** and the minimum from the begining of **j's** block to **j**, again using **T** and **P**; finally return the position where the overall minimum is using the three values you just computed.

Conclusion

RMQ and LCA are strongly related problems that can be reduced one to another. Many algorithms can be used to solve them, and they can be adapted to other kind of problems as well.

Here are some training problems for segment trees, LCA and RMQ:

SRM 310 -> Floating Median
http://acm.pku.edu.cn/JudgeOnline/problem?id=1986
http://acm.pku.edu.cn/JudgeOnline/problem?id=2374
http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=2045
http://acm.pku.edu.cn/JudgeOnline/problem?id=2763
http://www.spoj.pl/problems/QTREE2/
http://acm.uva.es/p/v109/10938.html
http://acm.sgu.ru/problem.php?contest=0&problem=155

References

- "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE" [PDF] by Johannes Fischer and Volker Heunn
- "The LCA Problem Revisited" [PPT] by Michael A.Bender and Martin Farach-Colton - a very good presentation, ideal for quick learning of some LCA and RMQ aproaches
- "Faster algorithms for finding lowest common ancestors in directed acyclic graphs" [PDF] by Artur Czumaj, Miroslav Kowaluk and Andrzej Lingas