Mathematics for TopCoders

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.

Dynamic Programming: From novice to advanced

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 **i-**$\mathbf{V_j}$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 (V[j] i AND Min[i-V[j]]+1<Min[i])
Then Min[i]=Min[i-V[j]]+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

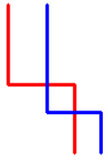
The following problems will need some good observations in order to reduce them to a dynamic solution.

Problem StarAdventure - SRM 208 Div 1:

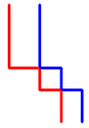
Given a matrix with **M** rows and **N** columns (**N x M**). In each cell there's a number of apples.
You start from the upper-left corner of the matrix. You can go down or right one cell. You need to arrive to the bottom-right corner. Then you need to go back to the upper-left cell by going each step 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).

Greedy is Good

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.

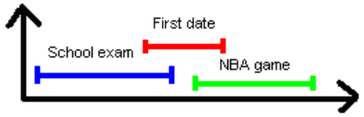
Could we help him have a nice holiday? Maybe we can! But let's make an assumption first. As John is a meticulous programmer, once he agrees on something, he sticks to the plan. So, individual activities may either be chosen or not. For each of the two choices regarding the first activity, we can make 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:

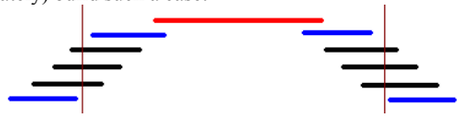
- Take a shower (15 minutes)
- Participate in the TopCoder Single Round Match (2 hours)
- Participate in an "All you can eat" and "All you can drink" contest (2 hours 58 minutes)
- Debug the room (3 hours)
- Attend the Rock Concert (4 hours)
- Have some paintball fun (6 hours)

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:

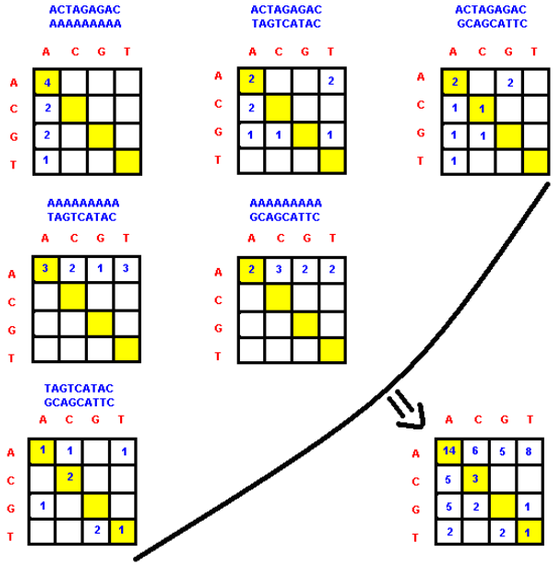
- Eliminate all the bugs and take some time to rest
- Tuesday is for chess, prepare to beat them all
- A whole day of Starcraft follows, this should be fun
- The next two days are for recovery

- 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
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.

A good detective | **Greedy and lucky** | **Greedy and not so lucky**

- 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).

A bit of fun: fun with bits

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));
}
```

Power up C++ with the Standard Template Library: Part I

Perhaps you are already using C++ as your main programming language to solve TopCoder problems. This means that you have already used STL in a simple way, because arrays and strings are passed to your function as STL objects. You may have noticed, though, that many coders manage to write their code much more quickly and concisely than you.

Or perhaps you are not a C++ programmer, but want to become one because of the great functionality of this language and its libraries (and, maybe, because of the very short solutions you've read in TopCoder practice rooms and competitions).

Regardless of where you're coming from, this article can help. In it, we will review some of the powerful features of the Standard Template Library (STL) – a great tool that, sometimes, can save you a lot of time in an algorithm competition.

The simplest way to get familiar with STL is to begin from its containers.

Containers

Any time you need to operate with many elements you require some kind of container. In native C (not C++) there was only one type of container: the array.

The problem is not that arrays are limited (though, for example, it's impossible to determine the size of array at runtime). Instead, the main problem is that many problems require a container with greater functionality.

For example, we may need one or more of the following operations:

- Add some string to a container.
- Remove a string from a container.
- Determine whether a string is present in the container.
- Return a number of distinct elements in a container.
- Iterate through a container and get a list of added strings in some order.

Of course, one can implement this functionality in an ordinal array. But the trivial implementation would be very inefficient. You can create the tree- of hash- structure to solve it in a faster way, but think a bit: does the implementation of such a container depend on elements we are going to store? Do we have to re-implement the module to make it functional, for example, for points on a plane but not strings?

If not, we can develop the interface for such a container once, and then use everywhere for data of any type. That, in short, is the idea of STL containers.

Before we begin

When the program is using STL, it should #include the appropriate standard headers. For most containers the title of standard header matches the name of the container, and no extension is required. For example, if you are going to use stack, just add the following line at the beginning of your program:

```
#include <stack>
```

Container types (and algorithms, functors and all STL as well) are defined not in global namespace, but in special namespace called "std." Add the following line after your includes and before the code begin:

```
using namespace std;
```

Another important thing to remember is that the type of a container is the template parameter. Template parameters are specified with the '<'/'>' "brackets" in code. For example:

```
vector<int> N;
```

When making nested constructions, make sure that the "brackets" are not directly following one another – leave a blank between them.

```
vector< vector<int> > CorrectDefinition;
vector<vector<int>> WrongDefinition; // Wrong: compiler may be
confused by 'operator >>'
```

Vector

The simplest STL container is vector. Vector is just an array with extended functionality. By the way, vector is the only container that is backward-compatible to native C code – this means that vector actually IS the array, but with some additional features.

```
vector<int> v(10);
for(int i = 0; i < 10; i++) {
    v[i] = (i+1)*(i+1);
}
for(int i = 9; i > 0; i--) {
    v[i] -= v[i-1];
}
```

Actually, when you type

```
vector<int> v;
```

the empty vector is created. Be careful with constructions like this:

```
vector<int> v[10];
```

Here we declare 'v' as an array of 10 vector<int>'s, which are initially empty. In most cases, this is not that we want. Use parentheses instead of brackets here.

The most frequently used feature of vector is that it can report its size.

```
int elements_count = v.size();
```

Two remarks: first, size() is unsigned, which may sometimes cause problems. Accordingly, I usually define macros, something like sz(C) that returns size of C as ordinal signed int. Second, it's not a good practice to compare v.size() to zero if you want to know whether the container is empty. You're better off using empty() function:

```
bool is_nonempty_notgood = (v.size() >= 0); // Try to avoid this
bool is_nonempty_ok = !v.empty();
```

This is because not all the containers can report their size in O(1), and you definitely should not require counting all elements in a double-linked list just to ensure that it contains at least one.

Another very popular function to use in vector is push_back. Push_back adds an element to the end of vector, increasing its size by one. Consider the following example:

```
vector<int> v;
for(int i = 1; i < 1000000; i *= 2) {
     v.push_back(i);
}
int elements_count = v.size();
```

Don't worry about memory allocation -- vector will not allocate just one element each time. Instead, vector allocates more memory then it actually needs when adding new elements with push_back. The only thing you should worry about is memory usage, but at TopCoder this may not matter. (More on vector's memory policy later.)

When you need to resize vector, use the resize() function:

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
     v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
     v[i] = i*2;
}
```

The resize() function makes vector contain the required number of elements. If you require less elements than vector already contain, the last ones will be deleted. If you ask vector to grow, it will enlarge its size and fill the newly created elements with zeroes.

Note that if you use push_back() after resize(), it will add elements AFTER the newly allocated size, but not INTO it. In the example above the size of the resulting vector is 25, while if we use push_back() in a second loop, it would be 30.

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
     v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
     v.push_back(i*2); // Writes to elements with indices
[25..30), not [20..25) ! <
}
```

To clear a vector use clear() member function. This function makes vector to contain 0 elements. It does not make elements zeroes -- watch out -- it completely erases the container.

There are many ways to initialize vector. You may create vector from another vector:

```
vector<int> v1;
// ...
vector<int> v2 = v1;
vector<int> v3(v1);
```

The initialization of v2 and v3 in the example above are exactly the same.

If you want to create a vector of specific size, use the following constructor:

```
vector<int> Data(1000);
```

In the example above, the data will contain 1,000 zeroes after creation. Remember to use parentheses, not brackets. If you want vector to be initialized with something else, write it in such manner:

```
vector<string> names(20, "Unknown");
```

Remember that you can create vectors of any type.

Multidimensional arrays are very important. The simplest way to create the two-dimensional array via vector is to create a vector of vectors.

```
vector< vector<int> > Matrix;
```

It should be clear to you now how to create the two-dimensional vector of given size:

```
int N, N;
// ...
vector< vector<int> > Matrix(N, vector<int>(M, -1));
```

Here we create a matrix of size N*M and fill it with -1.

The simplest way to add data to vector is to use push_back(). But what if we want to add data somewhere other than the end? There is the insert() member function for this purpose. And there is also the erase() member function to erase elements, as well. But first we need to say a few words about iterators.

You should remember one more very important thing: When vector is passed as a parameter to some function, a copy of vector is actually created. It may take a lot of time and memory to create new vectors when they are not really needed. Actually, it's hard to find a task where the copying of vector is REALLY needed when passing it as a parameter. So, you should never write:

```
void some_function(vector<int> v) { // Never do it unless you're
sure what you do!
     // ...
}
```

Instead, use the following construction:

```
void some_function(const vector<int>& v) { // OK
     // ...
}
```

If you are going to change the contents of vector in the function, just omit the 'const' modifier.

```
int modify_vector(vector<int>& v) { // Correct
     V[0]++;
}
```

Pairs

Before we come to iterators, let me say a few words about pairs. Pairs are widely used in STL. Simple problems, like TopCoder SRM 250 and easy 500-point problems, usually require some simple data structure that fits well with pair. STL std::pair is just a pair of elements. The simplest form would be the following:

```
template<typename T1, typename T2> struct pair {
     T1 first;
     T2 second;
};
```

In general pair<int,int> is a pair of integer values. At a more complex level, pair<string, pair<int, int> > is a pair of string and two integers. In the second case, the usage may be like this:

```
pair<string, pair<int,int> > P;
string s = P.first; // extract string
int x = P.second.first; // extract first int
int y = P.second.second; // extract second int
```

The great advantage of pairs is that they have built-in operations to compare themselves. Pairs are compared first-to-second element. If the first elements are not equal, the result will be based on the comparison of the first elements only; the second elements will be compared only if the first ones are equal. The array (or vector) of pairs can easily be sorted by STL internal functions.

For example, if you want to sort the array of integer points so that they form a polygon, it's a good idea to put them to the vector< pair<double, pair<int,int> >, where each element of vector is { polar angle, { x, y } }. One call to the STL sorting function will give you the desired order of points.

Pairs are also widely used in associative containers, which we will speak about later in this article.

Iterators

What are iterators? In STL iterators are the most general way to access data in containers. Consider the simple problem: Reverse the array A of N int's. Let's begin from a C-like solution:

```
void reverse_array_simple(int *A, int N) {
     int first = 0, last = N-1; // First and last indices of
elements to be swapped
     While(first < last) { // Loop while there is something to
swap
          swap(A[first], A[last]); // swap(a,b) is the standard
STL function
          first++; // Move first index forward
          last--; // Move last index back
     }
}
```

This code should be clear to you. It's pretty easy to rewrite it in terms of pointers:

```
void reverse_array(int *A, int N) {
     int *first = A, *last = A+N-1;
     while(first < last) {
          Swap(*first, *last);
          first++;
          last--;
     }
}
```

Look at this code, at its main loop. It uses only four distinct operations on pointers 'first' and 'last':

- compare pointers (first < last),
- get value by pointer (*first, *last),
- increment pointer, and
- decrement pointer

Now imagine that you are facing the second problem: Reverse the contents of a double-linked list, or a part of it. The first code, which uses indexing, will definitely not work. At least, it will not work in time, because it's impossible to get element by index in a double-linked list in O(1), only in O(N), so the whole algorithm will work in O(N^2). Errr...

But look: the second code can work for ANY pointer-like object. The only restriction is that that object can perform the operations described above: take value (unary *), comparison (<), and increment/decrement (++/--). Objects with these properties that are associated with containers are called iterators. Any STL container may be traversed by means of an iterator. Although not often needed for vector, it's very important for other container types.

So, what do we have? An object with syntax very much like a pointer. The following operations are defined for iterators:

- get value of an iterator, int x = *it;
- increment and decrement iterators it1++, it2--;
- compare iterators by '!=' and by '<'
- add an immediate to iterator it += 20; <=> shift 20 elements forward
- get the distance between iterators, int n = it2-it1;

But instead of pointers, iterators provide much greater functionality. Not only can they operate on any container, they may also perform, for example, range checking and profiling of container usage.

And the main advantage of iterators, of course, is that they greatly increase the reuse of code: your own algorithms, based on iterators, will work on a wide range of containers, and your own containers, which provide iterators, may be passed to a wide range of standard functions.

Not all types of iterators provide all the potential functionality. In fact, there are so-called "normal iterators" and "random access iterators". Simply put, normal iterators may be compared with '==' and '!=', and they may also be incremented and decremented. They may not be subtracted and we can not add a value to the normal iterator. Basically, it's impossible to implement the described operations in O(1) for all container types. In spite of this, the function that reverses array should look like this:

```
template<typename T> void reverse_array(T *first, T *last) {
     if(first != last) {
          while(true) {
               swap(*first, *last);
               first++;
               if(first == last) {
                    break;
               }
               last--;
               if(first == last) {
                    break;
               }
          }
     }
}
```

The main difference between this code and the previous one is that we don't use the "<" comparison on iterators, just the "==" one. Again, don't panic if you are surprised by the function prototype: template is just a way to declare a function, which works on any appropriate parameter types. This function should work perfectly on pointers to any object types and with all normal iterators.

Let's return to the STL. STL algorithms always use two iterators, called "begin" and "end." The end iterator is pointing not to the last object, however, but to the first invalid object, or the object directly following the last one. It's often very convenient.

Each STL container has member functions begin() and end() that return the begin and end iterators for that container.

Based on these principles, c.begin() == c.end() if and only if c is empty, and c.end() – c.begin() will always be equal to c.size(). (The last sentence is valid in cases when iterators can be subtracted, i.e. begin() and end() return random access iterators, which is not true for all kinds of containers. See the prior example of the double-linked list.)

The STL-compliant reverse function should be written as follows:

```
template<typename T> void reverse_array_stl_compliant(T *begin, T
*end) {
      // We should at first decrement 'end'
      // But only for non-empty range
      if(begin != end)
      {
           end--;
           if(begin != end) {
                while(true) {
                     swap(*begin, *end);
                     begin++;
                     If(begin == end) {
                          break;
                     }
                     end--;
                     if(begin == end) {
                          break;
                     }
                }
           }
      }
}
```

Note that this function does the same thing as the standard function std::reverse(T begin, T end) that can be found in algorithms module (#include <algorithm>).

In addition, any object with enough functionality can be passed as an iterator to STL algorithms and functions. That is where the power of templates comes in! See the following examples:

```
vector<int> v;
// ...
vector<int> v2(v);
vector<int> v3(v.begin(), v.end()); // v3 equals to v2

int data[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 };
vector<int> primes(data, data+(sizeof(data) / sizeof(data[0])));
```

The last line performs a construction of vector from an ordinal C array. The term 'data' without index is treated as a pointer to the beginning of the array. The term 'data + N' points to N-th element, so, when N if the size of array, 'data + N' points to first element not in array, so 'data + length of data' can be treated as end iterator for array 'data'. The expression 'sizeof(data)/sizeof(data[0])' returns the size of the array data, but only in a few cases, so don't use it anywhere except in such constructions. (C programmers will agree with me!)

Furthermore, we can even use the following constructions:

```
vector<int> v;
// ...
vector<int> v2(v.begin(), v.begin() + (v.size()/2));
```

It creates the vector v2 that is equal to the first half of vector v.

Here is an example of reverse() function:

```
int data[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
reverse(data+2, data+6); // the range { 5, 7, 9, 11 } is now { 11,
9, 7, 5 };
```

Each container also has the rbegin()/rend() functions, which return reverse iterators. Reverse iterators are used to traverse the container in backward order. Thus:

```
vector<int> v;
vector<int> v2(v.rbegin()+(v.size()/2), v.rend());
```

will create v2 with first half of v, ordered back-to-front.

To create an iterator object, we must specify its type. The type of iterator can be constructed by a type of container by appending "::iterator", "::const_iterator", "::reverse_iterator" or "::const_reverse_iterator" to it. Thus, vector can be traversed in the following way:

```
vector<int> v;

// ...

// Traverse all container, from begin() to end()
for(vector<int>::iterator it = v.begin(); it != v.end(); it++) {
     *it++; // Increment the value iterator is pointing to
}
```

I recommend you use '!=' instead of '<', and 'empty()' instead of 'size() != 0' -- for some container types, it's just very inefficient to determine which of the iterators precedes another.

Now you know of STL algorithm reverse(). Many STL algorithms are declared in the same way: they get a pair of iterators – the beginning and end of a range – and return an iterator.

The find() algorithm looks for appropriate elements in an interval. If the element is found, the iterator pointing to the first occurrence of the element is returned. Otherwise, the return value equals the end of interval. See the code:

```
vector<int> v;
for(int i = 1; i < 100; i++) {
     v.push_back(i*i);
}

if(find(v.begin(), v.end(), 49) != v.end()) {
     // ...
}
```

To get the index of element found, one should subtract the beginning iterator from the result of find():

```
int i = (find(v.begin(), v.end(), 49) - v.begin();
if(i < v.size()) {
     // ...
}
```

Remember to #include <algorithm> in your source when using STL algorithms.

The min_element and max_element algorithms return an iterator to the respective element. To get the value of min/max element, like in find(), use *min_element(...) or *max_element(...), to get index in array subtract the begin iterator of a container or range:

```
int data[5] = { 1, 5, 2, 4, 3 };
vector<int> X(data, data+5);
int v1 = *max_element(X.begin(), X.end()); // Returns value of max element in vector
int i1 = min_element(X.begin(), X.end()) - X.begin; // Returns index of min element in vector

int v2 = *max_element(data, data+5); // Returns value of max element in array
int i3 = min_element(data, data+5) - data; // Returns index of min element in array
```

Now you may see that the useful macros would be:

```
#define all(c) c.begin(), c.end()
```

Don't put the whole right-hand side of these macros into parentheses -- that would be wrong!

Another good algorithm is sort(). It's very easy to use. Consider the following examples:

```
vector<int> X;

// ...

sort(X.begin(), X.end()); // Sort array in ascending order
sort(all(X)); // Sort array in ascending order, use our #define
sort(X.rbegin(), X.rend()); // Sort array in descending order using with reverse iterators
```

Compiling STL Programs

One thing worth pointing out here is STL error messages. As the STL is distributed in sources, and it becomes necessary for compilers to build efficient executables, one of STL's habits is unreadable error messages.

For example, if you pass a vector<int> as a const reference parameter (as you should do) to some function:

```
void f(const vector<int>& v) {
     for(
          vector<int>::iterator it = v.begin(); // hm... where's the error?..
          // ...
     // ...
}
```

The error here is that you are trying to create the non-const iterator from a const object with the begin() member function (though identifying that error can be harder than actually correcting it). The right code looks like this:

```
void f(const vector<int>& v) {
     int r = 0;
     // Traverse the vector using const_iterator
     for(vector<int>::const_iterator it = v.begin(); it != v.end(); it++) {
          r += (*it)*(*it);
     }
     return r;
}
```

In spite of this, let me tell about very important feature of GNU C++ called 'typeof'. This operator is replaced to the type of an expression during the compilation. Consider the following example:

```
typeof(a+b) x = (a+b);
```

This will create the variable x of type matching the type of (a+b) expression. Beware that typeof(v.size()) is unsigned for any STL container type. But the most important application of typeof for TopCoder is traversing a container. Consider the following macros:

```
#define tr(container, it) \
     for(typeof(container.begin()) it = container.begin(); it != container.end(); it++)
```

By using these macros we can traverse every kind of container, not only vector. This will produce const_iterator for const object and normal iterator for non-const object, and you will never get an error here.

```
void f(const vector<int>& v) {
     int r = 0;
     tr(v, it) {
          r += (*it)*(*it);
     }
     return r;
}
```

Note: I did not put additional parentheses on the #define line in order to improve its readability. See this article below for more correct #define statements that you can experiment with in practice rooms.

Traversing macros is not really necessary for vectors, but it's very convenient for more complex data types, where indexing is not supported and iterators are the only way to access data. We will speak about this later in this article.

Data manipulation in vector

One can insert an element to vector by using the insert() function:

```
vector<int> v;
// ...
v.insert(1, 42); // Insert value 42 after the first
```

All elements from second (index 1) to the last will be shifted right one element to leave a place for a new element. If you are planning to add many elements, it's not good to do many shifts – you're better off calling insert() one time. So, insert() has an interval form:

```
vector<int> v;
vector<int> v2;

// ..

// Shift all elements from second to last to the appropriate number of elements.
// Then copy the contents of v2 into v.
v.insert(1, all(v2));
```

Vector also has a member function erase, which has two forms. Guess what they are:

```
erase(iterator);
erase(begin iterator, end iterator);
```

At first case, single element of vector is deleted. At second case, the interval, specified by two iterators, is erased from vector.

The insert/erase technique is common, but not identical for all STL containers.

String

There is a special container to manipulate with strings. The string container has a few differences from vector<char>. Most of the differences come down to string manipulation functions and memory management policy.

String has a substring function without iterators, just indices:

```
string s = "hello";
string
     s1 = s.substr(0, 3), // "hel"
     s2 = s.substr(1, 3), // "ell"
     s3 = s.substr(0, s.length()-1), "hell"
     s4 = s.substr(1); // "ello"
```

Beware of (s.length()-1) on empty string because s.length() is unsigned and unsigned(0) – 1 is definitely not what you are expecting!

Set

It's always hard to decide which kind of container to describe first – set or map. My opinion is that, if the reader has a basic knowledge of algorithms, beginning from 'set' should be easier to understand.

Consider we need a container with the following features:

- add an element, but do not allow duples [duplicates?]
- remove elements
- get count of elements (distinct elements)
- check whether elements are present in set

This is quite a frequently used task. STL provides the special container for it – set. Set can add, remove and check the presence of particular element in O(log N), where N is the count of objects in the set. While adding elements to set, the duples [duplicates?] are discarded. A count of the elements in the set, N, is returned in O(1). We will speak of the algorithmic implementation of set and map later -- for now, let's investigate its interface:

```
set<int> s;

for(int i = 1; i <= 100; i++) {
     s.insert(i); // Insert 100 elements, [1..100]
}

s.insert(42); // does nothing, 42 already exists in set

for(int i = 2; i <= 100; i += 2) {
     s.erase(i); // Erase even values
}

int n = int(s.size()); // n will be 50
```

The push_back() member may not be used with set. It make sense: since the order of elements in set does not matter, push_back() is not applicable here.

Since set is not a linear container, it's impossible to take the element in set by index. Therefore, the only way to traverse the elements of set is to use iterators.

```
// Calculate the sum of elements in set
```

```
set<int> S;
// ...
int r = 0;
for(set<int>::const_iterator it = S.begin(); it != S.end(); it++)
{
      r += *it;
}
```

It's more elegant to use traversing macros here. Why? Imagine you have a set< pair<string, pair< int, vector<int> > >. How to traverse it? Write down the iterator type name? Oh, no. Use our traverse macros instead.

```
set< pair<string, pair< int, vector<int> > > SS;
int total = 0;
tr(SS, it) {
      total += it->second.first;
}
```

Notice the 'it->second.first' syntax. Since 'it' is an iterator, we need to take an object from 'it' before operating. So, the correct syntax would be '(*it).second.first'. However, it's easier to write 'something->' than '(*something)'. The full explanation will be quite long –just remember that, for iterators, both syntaxes are allowed.

To determine whether some element is present in set use 'find()' member function. Don't be confused, though: there are several 'find()' 's in STL. There is a global algorithm 'find()', which takes two iterators, element, and works for O(N). It is possible to use it for searching for element in set, but why use an O(N) algorithm while there exists an O(log N) one? While searching in set and map (and also in multiset/multimap, hash_map/hash_set, etc.) do not use global find – instead, use member function 'set::find()'. As 'ordinal' find, set::find will return an iterator, either to the element found, or to 'end()'. So, the element presence check looks like this:

```
set<int> s;
// ...
if(s.find(42) != s.end()) {
      // 42 presents in set
}
else {
      // 42 not presents in set
}
```

Another algorithm that works for O(log N) while called as member function is count. Some people think that

```
if(s.count(42) != 0) {
      // …
}
```

or even

```
if(s.count(42)) {
      // …
}
```

is easier to write. Personally, I don't think so. Using count() in set/map is nonsense: the element either presents or not. As for me, I prefer to use the following two macros:

```
#define present(container, element) (container.find(element) !=
container.end())
#define cpresent(container, element) (find(all(container),element)
!= container.end())
```

(Remember that all(c) stands for "c.begin(), c.end()")

Here, 'present()' returns whether the element presents in the container with member function 'find()' (i.e. set/map, etc.) while 'cpresent' is for vector.

To erase an element from set use the erase() function.

```
set<int> s;
// …
s.insert(54);
s.erase(29);
```

The erase() function also has the interval form:

```
set<int> s;
// ..

set<int>::iterator it1, it2;
it1 = s.find(10);
it2 = s.find(100);
// Will work if it1 and it2 are valid iterators, i.e. values 10
and 100 present in set.
s.erase(it1, it2); // Note that 10 will be deleted, but 100 will
remain in the container
```

Set has an interval constructor:

```
int data[5] = { 5, 1, 4, 2, 3 };
set<int> S(data, data+5);
```

It gives us a simple way to get rid of duplicates in vector, and sort it:

```
vector<int> v;
// …
set<int> s(all(v));
vector<int> v2(all(s));
```

Here 'v2' will contain the same elements as 'v' but sorted in ascending order and with duplicates removed.

Any comparable elements can be stored in set. This will be described later.

Map

There are two explanation of map. The simple explanation is the following:

```
map<string, int> M;
M["Top"] = 1;
M["Coder"] = 2;
M["SRM"] = 10;

int x = M["Top"] + M["Coder"];

if(M.find("SRM") != M.end()) {
     M.erase(M.find("SRM")); // or even M.erase("SRM")
}
```

Very simple, isn't it?

Actually map is very much like set, except it contains not just values but pairs <key, value>. Map ensures that at most one pair with specific key exists. Another quite pleasant thing is that map has operator [] defined.

Traversing map is easy with our 'tr()' macros. Notice that iterator will be an std::pair of key and value. So, to get the value use it->second. The example follows:

```
map<string, int> M;
// …
int r = 0;
tr(M, it) {
      r += it->second;
}
```

Don't change the key of map element by iterator, because it may break the integrity of map internal data structure (see below).

There is one important difference between map::find() and map::operator []. While map::find() will never change the contents of map, operator [] will create an element if it does not exist. In some cases this could be very convenient, but it's definitly a bad idea to use operator [] many times in a loop, when you do not want to add new elements. That's why operator [] may not be used if map is passed as a const reference parameter to some function:

```
void f(const map<string, int>& M) {
      if(M["the meaning"] == 42) { // Error! Cannot use [] on const
map objects!
      }
      if(M.find("the meaning") != M.end() && M.find("the meaning")-
>second == 42) { // Correct
            cout << "Don't Panic!" << endl;
      }
}
```

Notice on Map and Set

Internally map and set are almost always stored as red-black trees. We do not need to worry about the internal structure, the thing to remember is that the elements of map and set are always sorted in ascending order while traversing these containers. And that's why it's strongly not recommended to change the key value while traversing map or set: If you make the modification that breaks the order, it will lead to improper functionality of container's algorithms, at least.

But the fact that the elements of map and set are always ordered can be practically used while solving TopCoder problems.

Another important thing is that operators ++ and -- are defined on iterators in map and set. Thus, if the value 42 presents in set, and it's not the first and the last one, than the following code will work:

```
set<int> S;
// ...
set<int>::iterator it = S.find(42);
set<int>::iterator it1 = it, it2 = it;
it1--;
it2++;
int a = *it1, b = *it2;
```

Here 'a' will contain the first neighbor of 42 to the left and 'b' the first one to the right.

More on algorithms

It's time to speak about algorithms a bit more deeply. Most algorithms are declared in the #include <algorithm> standard header. At first, STL provides three very simple algorithms: min(a,b), max(a,b), swap(a,b). Here min(a,b) and max(a,b) returns the minimum and maximum of two elements, while swap(a,b) swaps two elements.

Algorithm sort() is also widely used. The call to sort(begin, end) sorts an interval in ascending order. Notice that sort() requires random access iterators, so it will not work on all containers. However, you probably won't ever call sort() on set, which is already ordered.

You've already heard of algorithm find(). The call to find(begin, end, element) returns the iterator where 'element' first occurs, or end if the element is not found. Instead of find(...), count(begin, end, element) returns the number of

occurrences of an element in a container or a part of a container. Remember that set and map have the member functions find() and count(), which works in O(log N), while std::find() and std::count() take O(N).

Other useful algorithms are next_permutation() and prev_permutation(). Let's speak about next_permutation. The call to next_permutation(begin, end) makes the interval [begin, end) hold the next permutation of the same elements, or returns false if the current permutation is the last one. Accordingly, next_permutation makes many tasks quite easy. If you want to check all permutations, just write:

```
vector<int> v;

for(int i = 0; i < 10; i++) {
     v.push_back(i);
}

do {
     Solve(..., v);
} while(next_permutation(all(v));
```

Don't forget to ensure that the elements in a container are sorted before your first call to next_permutation(...). Their initial state should form the very first permutation; otherwise, some permutations will not be checked.

String Streams

You often need to do some string processing/input/output. C++ provides two interesting objects for it: 'istringstream' and 'ostringstream'. They are both declared in #include <sstream>.

Object istringstream allows you to read from a string like you do from a standard input. It's better to view source:

```
void f(const string& s) {

     // Construct an object to parse strings
     istringstream is(s);

     // Vector to store data
     vector<int> v;

     // Read integer while possible and add it to the vector
     int tmp;
     while(is >> tmp) {
          v.push_back(tmp);
     }
}
```

The ostringstream object is used to do formatting output. Here is the code:

```
string f(const vector<int>& v) {

     // Constucvt an object to do formatted output
     ostringstream os;

     // Copy all elements from vector<int> to string stream as
text
     tr(v, it) {
          os << ' ' << *it;
     }

     // Get string from string stream
     string s = os.str();

     // Remove first space character
     if(!s.empty()) { // Beware of empty string here
          s = s.substr(1);
     }

     return s;
}
```

Summary

To go on with STL, I would like to summarize the list of templates to be used. This will simplify the reading of code samples and, I hope, improve your TopCoder skills. The short list of templates and macros follows:

```
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> ii;
#define sz(a) int((a).size())
#define pb push_back
#defile all(c) (c).begin(),(c).end()
#define tr(c,i) for(typeof((c).begin() i = (c).begin(); i !=
(c).end(); i++)
#define present(c,x) ((c).find(x) != (c).end())
#define cpresent(c,x) (find(all(c),x) != (c).end())
```

The container vector<int> is here because it's really very popular. Actually, I found it convenient to have short aliases to many containers (especially for vector<string>, vector<ii>, vector< pair<double, ii> >). But this list only includes the macros that are required to understand the following text.

Another note to keep in mind: When a token from the left-hand side of #define appears in the right-hand side, it should be placed in braces to avoid many nontrivial problems.

Power up C++ with the Standard Template Library: Part II: Advanced Uses

In this tutorial we will use some macros and typedefs from Part I of the tutorial.

Creating Vector from Map

As you already know, map actually contains pairs of element. So you can write it in like this:

```
map<string, int> M;
// ...
vector< pair<string, int> > V(all(M)); // remember all(c) stands
for
 (c).begin(),(c).end()
```

Now vector will contain the same elements as map. Of course, vector will be sorted, as is map. This feature may be useful if you are not planning to change elements in map any more but want to use indices of elements in a way that is impossible in map.

Copying data between containers

Let's take a look at the copy(...) algorithm. The prototype is the following:

```
copy(from_begin, from_end, to_begin);
```

This algorithm copies elements from the first interval to the second one. The second interval should have enough space available. See the following code:

```
vector<int> v1;
vector<int> v2;

// ...

// Now copy v2 to the end of v1
v1.resize(v1.size() + v2.size());
// Ensure v1 have enough space
copy(all(v2), v1.end() - v2.size());
// Copy v2 elements right after v1 ones
```

Another good feature to use in conjunction with copy is inserters. I will not describe it here due to limited space but look at the code:

```
vector<int> v;
// ...
set<int> s;
// add some elements to set
copy(all(v), inserter(s));
```

The last line means:

```
tr(v, it) {
// remember traversing macros from Part I
     s.insert(*it);
}
```

But why use our own macros (which work only in gcc) when there is a standard function? It's a good STL practice to use standard algorithms like copy, because it will be easy to others to understand your code.

To insert elemements to vector with push_back use back_inserter, or front_inserter is available for deque container. And in some cases it is useful to remember that the first two arguments for 'copy' may be not only begin/end, but also rbegin/rend, which copy data in reverse order.

Merging lists

Another common task is to operate with sorted lists of elements. Imagine you have two lists of elements -- A and B, both ordered. You want to get a new list from these two. There are four common operations here:

- 'union' the lists, R = A+B
- intersect the lists, R = A*B
- set difference, R = A*(~B) or R = A-B
- set symmetric difference, R = A XOR B

STL provides four algorithms for these tasks: set_union(...), set_intersection(...), set_difference(...) and set_symmetric_difference(...). They all have the same calling conventions, so let's look at set_intersection. A free-styled prototype would look like this:

```
end_result = set_intersection(begin1, end1, begin2, end2,
begin_result);
```

Here [begin1,end1) and [begin2,end2) are the input lists. The 'begin_result' is the iterator from where the result will be written. But the size of the result is unknown, so this function returns the end iterator of output (which determines how many elements are in the result). See the example for usage details:

```
int data1[] = { 1, 2, 5, 6, 8, 9, 10 };
int data2[] = { 0, 2, 3, 4, 7, 8, 10 };

vector<int> v1(data1, data1+sizeof(data1)/sizeof(data1[0]));
vector<int> v2(data2, data2+sizeof(data2)/sizeof(data2[0]));

vector<int> tmp(max(v1.size(), v2.size());
```

```
vector<int> res = vector<int> (tmp.begin(),
set_intersection(all(v1), all(v2), tmp.begin());
```
Look at the last line. We construct a new vector named 'res'. It is constructed via interval constructor, and the beginning of the interval will be the beginning of tmp. The end of the interval is the result of the set_intersection algorithm. This algorithm will intersect v1 and v2 and write the result to the output iterator, starting from 'tmp.begin()'. Its return value will actually be the end of the interval that forms the resulting dataset.

One comment that might help you understand it better: If you would like to just get the number of elements in set intersection, use int cnt = set_intersection(all(v1), all(v2), tmp.begin()) – tmp.begin();

Actually, I would never use a construction like ' vector<int> tmp'. I don't think it's a good idea to allocate memory for each set_*** algorithm invoking. Instead, I define the global or static variable of appropriate type and enough size. See below:
```
set<int> s1, s2;
for(int i = 0; i < 500; i++) {
        s1.insert(i*(i+1) % 1000);
        s2.insert(i*i*i % 1000);
}

static int temp[5000]; // greater than we need

vector<int> res = vi(temp, set_symmetric_difference(all(s1),
all(s2), temp));
int cnt = set_symmetric_difference(all(s1), all(s2), temp) - temp;
```
Here 'res' will contain the symmetric difference of the input datasets.

Remember, input datasets need to be sorted to use these algorithms. So, another important thing to remember is that, because sets are always ordered, we can use set-s (and even map-s, if you are not scared by pairs) as parameters for these algorithms.

These algorithms work in single pass, in O(N1+N2), when N1 and N2 are sizes of input datasets.

Calculating Algorithms
Yet another interesting algorithm is accumulate(...). If called for a vector of int-s and third parameter zero, accumulate(...) will return the sum of elements in vector:
```
vector<int> v;
// ...
int sum = accumulate(all(v), 0);
```
The result of accumulate() call always has the type of its third argument. So, if you are not sure that the sum fits in integer, specify the third parameter's type directly:
```
vector<int> v;
// ...
long long sum = accumulate(all(v), (long long)0);
```
Accumulate can even calculate the product of values. The fourth parameter holds the predicate to use in calculations. So, if you want the product:
```
vector<int> v;
// ...
double product = accumulate(all(v), double(1),
multiplies<double>());
// don't forget to start with 1 !
```
Another interesting algorithm is inner_product(...). It calculates the scalar product of two intervals. For example:
```
vector<int> v1;
vector<int> v2;
for(int i = 0; i < 3; i++) {
      v1.push_back(10-i);
      v2.push_back(i+1);
}
int r = inner_product(all(v1), v2.begin(), 0);
```
'r' will hold (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2]), or (10*1+9*2+8*3), which is 52.

As for 'accumulate' the type of return value for inner_product is defined by the last parameter. The last parameter is the initial value for the result. So, you may use inner_product for the hyperplane object in multidimensional space: just write inner_product(all(normal), point.begin(), -shift).

It should be clear to you now that inner_product requires only increment operation from iterators, so queues and sets can also be used as parameters. Convolution filter, for calculating the nontrivial median value, could look like this:
```
set<int> values_ordered_data(all(data));
int n = sz(data); // int n = int(data.size());
vector<int> convolution_kernel(n);
for(int i = 0; i < n; i++) {
     convolution_kernel[i] = (i+1)*(n-i);
}
double result = double(inner_product(all(ordered_data),
convolution_kernel.begin(), 0)) /
accumulate(all(convolution_kernel), 0);
```
Of course, this code is just an example -- practically speaking, it would be faster to copy values to another vector and sort it.

It's also possible to write a construction like this:
```
vector<int> v;
// ...
int r = inner_product(all(v), v.rbegin(), 0);
```
This will evaluate V[0]*V[N-1] + V[1]+V[N-2] + ... + V[N-1]*V[0] where N is the number of elements in 'v'.

Nontrivial Sorting
Actually, sort(...) uses the same technique as all STL:

- all comparison is based on 'operator <'

This means that you only need to override 'operator <'. Sample code follows:
```
struct fraction {
      int n, d; // (n/d)
      // ...
      bool operator < (const fraction& f) const {
           if(false) {
                return (double(n)/d) < (double(f.n)/f.d);
                // Try to avoid this, you're the TopCoder!
           }
           else {
                return n*f.d < f.n*d;
           }
      }
};

// ...

vector<fraction> v;

// ...

sort(all(v));
```
In cases of nontrivial fields, your object should have default and copy constructor (and, maybe, assignment operator -- but this comment is not for TopCoders).

Remember the prototype of 'operator <' : return type bool, const modifier, parameter const reference.

Another possibility is to create the comparison functor. Special comparison predicate may be passed to the sort(...) algorithm as a third parameter. Example: sort points (that are pair<double,double>) by polar angle.
```
typedef pair<double, double> dd;

const double epsilon = 1e-6;

struct sort_by_polar_angle {
       dd center;
       // Constuctor of any type
       // Just find and store the center
       template<typename T> sort_by_polar_angle(T b, T e) {
            int count = 0;
            center = dd(0,0);
            while(b != e) {
                        center.first += b->first;
                        center.second += b->second;
                  b++;
                count++;
            }
                  double k = count ? (1.0/count) : 0;
            center.first *= k;
                  center.second *= k;
       }
 // Compare two points, return true if the first one is earlier
 // than the second one looking by polar angle
 // Remember, that when writing comparator, you should
 // override not 'operator <' but 'operator ()'
       bool operator () (const dd& a, const dd& b) const {
            double p1 = atan2(a.second-center.second, a.first-
center.first);
            double p2 = atan2(b.second-center.second, b.first-
center.first);
            return p1 + epsilon < p2;
       }
 };

 // ...

 vector<dd> points;

 // ...
```

```
        sort(all(points), sort_by_polar_angle(all(points)));
```

This code example is complex enough, but it does demonstrate the abilities of STL. I should point out that, in this sample, all code will be inlined during compilation, so it's actually really fast.

Also remember that 'operator <' should always return false for equal objects. It's very important – for the reason why, see the next section.

Using your own objects in Maps and Sets

Elements in set and map are ordered. It's the general rule. So, if you want to enable using of your objects in set or map you should make them comparable. You already know the rule of comparisons in STL:

| * all comparison is based on 'operator <'

Again, you should understand it in this way: "I only need to implement operator < for objects to be stored in set/map."

Imagine you are going to make the 'struct point' (or 'class point'). We want to intersect some line segments and make a set of intersection points (sound familiar?). Due to finite computer precision, some points will be the same while their coordinates differ a bit. That's what you should write:

```
const double epsilon = 1e-7;


struct point {
        double x, y;

        // ...


        // Declare operator < taking precision into account
        bool operator < (const point& p) const {
             if(x < p.x - epsilon) return true;
             if(x > p.x + epsilon) return false;
             if(y < p.y - epsilon) return true;
             if(y > p.y + epsilon) return false;
             return false;
        }
 };
```

Now you can use set<point> or map<point, string>, for example, to look up whether some point is already present in the list of intersections. An even more advanced approach: use map<point, vector<int> > and list the list of indices of segments that intersect at this point.

It's an interesting concept that for STL 'equal' does not mean 'the same', but we will not delve into it here.

Memory management in Vectors

As has been said, vector does not reallocate memory on each push_back(). Indeed, when push_back() is invoked, vector really allocates more memory than is needed for one additional element. Most STL implementations of vector double in size when push_back() is invoked and memory is not allocated. This may not be good in practical purposes, because your program may eat up twice as much memory as you need. There are two easy ways to deal with it, and one complex way to solve it.

The first approach is to use the reserve() member function of vector. This function orders vector to allocate additional memory. Vector will not enlarge on push_back() operations until the size specified by reserve() will be reached.

Consider the following example. You have a vector of 1,000 elements and its allocated size is 1024. You are going to add 50 elements to it. If you call push_back() 50 times, the allocated size of vector will be 2048 after this operation. But if you write

```
v.reserve(1050);
```

before the series of push_back(), vector will have an allocated size of exactly 1050 elements.

If you are a rapid user of push_back(), then reserve() is your friend.

By the way, it's a good pattern to use v.reserve() followed by copy(…, back_inserter(v)) for vectors.

Another situation: after some manipulations with vector you have decided that no more adding will occur to it. How do you get rid of the potential allocation of additional memory? The solution follows:

```
vector<int> v;
// ...
vector<int>(all(v)).swap(v);
```

This construction means the following: create a temporary vector with the same content as v, and then swap this temporary vector with 'v'. After the swap the original oversized v will be disposed. But, most likely, you won't need this during SRMs.

The proper and complex solution is to develop your own allocator for the vector, but that's definitely not a topic for a TopCoder STL tutorial.

Implementing real algorithms with STL

Armed with STL, let's go on to the most interesting part of this tutorial: how to implement real algorithms efficiently.

Depth-first search (DFS)

I will not explain the theory of DFS here – instead, read this section of gladius's Introduction to Graphs and Data Structures tutorial – but I will show you how STL can help.

At first, imagine we have an undirected graph. The simplest way to store a graph in STL is to use the lists of vertices adjacent to each vertex. This leads to the vector< vector<int> > W structure, where W[i] is a list of vertices adjacent to i. Let's verify our graph is connected via DFS:

```
/*
Reminder from Part 1:
typedef vector<int> vi;
typedef vector<vi> vvi;
*/

 int N; // number of vertices
 vvi W; // graph
 vi V; // V is a visited flag

 void dfs(int i) {
       if(!V[i]) {
            V[i] = true;
            for_each(all(W[i]), dfs);
       }
 }

 bool check_graph_connected_dfs() {
       int start_vertex = 0;
       V = vi(N, false);
       dfs(start_vertex);
       return (find(all(V), 0) == V.end());
 }
```

That's all. STL algorithm 'for_each' calls the specified function, 'dfs', for each element in range. In check_graph_connected() function we first make the Visited array (of correct size and filled with zeroes). After DFS we have either visited all vertices, or not – this is easy to determine by searching for at least one zero in V, by means of a single call to find().

Notice on for_each: the last argument of this algorithm can be almost anything that "can be called like a function". It may be not only global function, but also adapters, standard algorithms, and even member functions. In the last case, you will need mem_fun or mem_fun_ref adapters, but we will not touch on those now.

One note on this code: I don't recommend the use of vector<bool>. Although in this particular case it's quite safe, you're better off not to use it. Use the predefined 'vi' (vector<int>). It's quite OK to assign true and false to int's in vi. Of course, it requires 8*sizeof(int)=8*4=32 times more memory, but it works well in most cases and is quite fast on TopCoder.

A word on other container types and their usage

Vector is so popular because it's the simplest array container. In most cases you only require the functionality of an array from vector – but, sometimes, you may need a more advanced container.

It is not good practice to begin investigating the full functionality of some STL container during the heat of a Single Round Match. If you are not familiar with the container you are about to use, you'd be better off using vector or map/set. For example, stack can always be implemented via vector, and it's much faster to act this way if you don't remember the syntax of stack container.

STL provides the following containers: list, stack, queue, deque, priority_queue. I've found list and deque quite useless in SRMs (except, probably, for very special tasks based on these containers). But queue and priority_queue are worth saying a few words about.

Queue

Queue is a data type that has three operations, all in O(1) amortized: add an element to front (to "head") remove an element from back (from "tail") get the first unfetched element ("tail") In other words, queue is the FIFO buffer.

Breadth-first search (BFS)

Again, if you are not familiar with the BFS algorithm, please refer back to this

TopCoder tutorial first. Queue is very convenient to use in BFS, as shown below:

```
/*
Graph is considered to be stored as adjacent vertices list.
Also we considered graph undirected.

vvi is vector< vector<int> >
W[v] is the list of vertices adjacent to v
*/

 int N; // number of vertices
 vvi W; // lists of adjacent vertices


 bool check_graph_connected_bfs() {
      int start_vertex = 0;
      vi V(N, false);
      queue<int> Q;
      Q.push(start_vertex);
      V[start_vertex] = true;
      while(!Q.empty()) {
           int i = Q.front();
           // get the tail element from queue
           Q.pop();
           tr(W[i], it) {
                if(!V[*it]) {
                     V[*it] = true;
                     Q.push(*it);
                }
           }
      }
      return (find(all(V), 0) == V.end());
 }
```

More precisely, queue supports front(), back(), push() (== push_back()), pop (== pop_front()). If you also need push_front() and pop_back(), use deque. Deque provides the listed operations in O(1) amortized.

There is an interesting application of queue and map when implementing a shortest path search via BFS in a complex graph. Imagine that we have the graph, vertices of which are referenced by some complex object, like:

```
pair< pair<int,int>, pair< string, vector< pair<int, int> > > >
```

```
 (this case is quite usual: complex data structure may define the
position in
 some game, Rubik's cube situation, etc…)
```

Consider we know that the path we are looking for is quite short, and the total number of positions is also small. If all edges of this graph have the same length of 1, we could use BFS to find a way in this graph. A section of pseudo-code follows:

```
// Some very hard data structure

typedef pair< pair<int,int>, pair< string, vector< pair<int, int> >
> > POS;

// ...

int find_shortest_path_length(POS start, POS finish) {

     map<POS, int> D;
     // shortest path length to this position
     queue<POS> Q;
     D[start] = 0; // start from here
     Q.push(start);

     while(!Q.empty()) {
          POS current = Q.front();
          // Peek the front element
          Q.pop(); // remove it from queue

          int current_length = D[current];

          if(current == finish) {
               return D[current];
               // shortest path is found, return its length
          }

          tr(all possible paths from 'current', it) {
               if(!D.count(*it)) {
               // same as if(D.find(*it) == D.end), see Part I
                    // This location was not visited yet
                    D[*it] = current_length + 1;
               }
          }
     }

     // Path was not found
     return -1;
}

// ...
```

If the edges have different lengths, however, BFS will not work. We should use Dijkstra instead. It's possible to implement such a Dijkstra via priority_queue -- see below.

Priority_Queue

Priority queue is the binary heap. It's the data structure, that can perform three operations:

- push any element (push)
- view top element (top)
- pop top element (pop)

For the application of STL's priority_queue see the TrainRobber problem from SRM 307.

Dijkstra

In the last part of this tutorial I'll describe how to efficiently implement Dijktra's algorithm in sparse graph using STL containers. Please look through this tutorial for information on Dijkstra's algoritm.

Consider we have a weighted directed graph that is stored as vector< vector< pair<int,int> > > G, where

- G.size() is the number of vertices in our graph
- G[i].size() is the number of vertices directly reachable from vertex with index i
- G[i][j].first is the index of j-th vertex reachable from vertex i
- G[i][j].second is the length of the edge heading from vertex i to vertex G[i][j].first

We assume this, as defined in the following two code snippets:

```
typedef pair<int,int> ii;
typedef vector<ii> vii;
typedef vector<vii> vvii;
```

Dijstra via priority_queue

Many thanks to misof for spending the time to explain to me why the complexity of this algorithm is good despite not removing deprecated entries from the queue.

```
     vi D(N, 987654321);
     // distance from start vertex to each vertex

     priority_queue<ii,vector<ii>, greater<ii> > Q;
     // priority_queue with reverse comparison operator,
     // so top() will return the least distance
     // initialize the start vertex, suppose it's zero
     D[0] = 0;
     Q.push(ii(0,0));

     // iterate while queue is not empty
     while(!Q.empty()) {

          // fetch the nearest element
          ii top = Q.top();
          Q.pop();

          // v is vertex index, d is the distance
          int v = top.second, d = top.first;

          // this check is very important
          // we analyze each vertex only once
          // the other occurrences of it on queue (added earlier)
          // will have greater distance
          if(d <= D[v]) {
               // iterate through all outcoming edges from v
               tr(G[v], it) {
                    int v2 = it->first, cost = it->second;
                    if(D[v2] > D[v] + cost) {
                         // update distance if possible
                         D[v2] = D[v] + cost;
                         // add the vertex to queue
                         Q.push(ii(D[v2], v2));

                    }
               }
          }
     }
```

I will not comment on the algorithm itself in this tutorial, but you should notice the priority_queue object definition. Normally, priority_queue<ii> will work, but the top() member function will return the largest element, not the smallest. Yes, one of the easy solutions I often use is just to store not distance but (-distance) in the first element of a pair. But if you want to implement it in the "proper" way, you need to reverse the comparison operation of priority_queue to reverse one. Comparison function is the third template parameter of priority_queue while the second paramerer is the storage type for container. So, you should write priority_queue<ii, vector<ii>, greater<ii> >.

Dijkstra via set
Petr gave me this idea when I asked him about efficient Dijkstra implementation in C#. While implementing Dijkstra we use the priority_queue to add elements to the "vertices being analyzed" queue in O(logN) and fetch in O(log N). But there is a container besides priority_queue that can provide us with this functionality -- it's 'set'! I've experimented a lot and found that the performance of Dijkstra based on priority_queue and set is the same.

So, here's the code:

```
vi D(N, 987654321);

// start vertex
set<ii> Q;
D[0] = 0;
Q.insert(ii(0,0));

while(!Q.empty()) {

    // again, fetch the closest to start element
    // from "queue" organized via set
    ii top = *Q.begin();
    Q.erase(Q.begin());
    int v = top.second, d = top.first;

    // here we do not need to check whether the distance
    // is perfect, because new vertices will always
    // add up in proper way in this implementation

    tr(G[v], it) {
        int v2 = it->first, cost = it->second;
        if(D[v2] > D[v] + cost) {
            // this operation can not be done with
priority_queue,
            // because it does not support DECREASE_KEY
            if(D[v2] != 987654321) {
                Q.erase(Q.find(ii(D[v2],v2)));
            }
            D[v2] = D[v] + cost;
            Q.insert(ii(D[v2], v2));
        }
    }
}
```

One more important thing: STL's priority_queue does not support the DECREASE_KEY operation. If you will need this operation, 'set' may be your best bet.

I've spent a lot of time to understand why the code that removes elements from queue (with set) works as fast as the first one.

These two implementations have the same complexity and work in the same time. Also, I've set up practical experiments and the performance is exactly the same (the difference is about ~%0.1 of time).

As for me, I prefer to implement Dijkstra via 'set' because with 'set' the logic is simpler to understand, and we don't need to remember about 'greater<int>' predicate overriding.

What is not included in STL
If you have made it this far in the tutorial, I hope you have seen that STL is a very powerful tool, especially for TopCoder SRMs. But before you embrace STL wholeheartedly, keep in mind what is NOT included in it.

First, STL does not have BigInteger-s. If a task in an SRM calls for huge calculations, especially multiplication and division, you have three options:

- use a pre-written template
- use Java, if you know it well
- say "Well, it was definitely not my SRM!"

I would recommend option number one.

Nearly the same issue arises with the geometry library. STL does not have geometry support, so you have those same three options again.

The last thing – and sometimes a very annoying thing – is that STL does not have a built-in string splitting function. This is especially annoying, given that this function is included in the default template for C++ in the ExampleBuilder plugin! But actually I've found that the use of istringstream(s) in trivial cases and sscanf(s.c_str(), …) in complex cases is sufficient.

Those caveats aside, though, I hope you have found this tutorial useful, and I hope you find the STL a useful addition to your use of C++. Best of luck to you in the Arena!

Note from the author: In both parts of this tutorial I recommend the use of some templates to minimize the time required to implement something. I must say that this suggestion should always be up to the coder. Aside from whether templates are a good or bad tactic for SRMs, in everyday life they can become annoying for other people who are trying to understand your code. While I did rely on them for some time, ultimately I reached the decision to stop. I encourage you to weigh the pros and cons of templates and to consider this decision for yourself.

Introduction to graphs and their data structures: Section 1

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

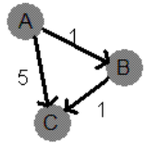
Representing a graph as a list of nodes is a very flexible method. But usually on TopCoder we have limits on the problems that attempt to make life easier for us. Normally our graphs are relatively small, with a small number of nodes and edges. When this is the case we can use a different type of data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

Introduction to graphs and their data structures: Section 2

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

Introduction to graphs and their data structures: Section 3

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.