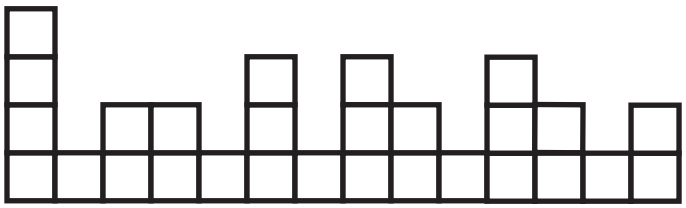
Problem Solving in Computer Science: 1

Ivor Page, The University of Texas at Dallas

June 1, 2016

Contents

Quiz 1:
Your little brother made a wall from bricks stacked into columns. Here is an example:



What's the minimum number of bricks that must be moved to make this wall of level height?

What algorithm did you use?

In this version of the wall, the bricks are extremely heavy. The only operation available is to slide a brick to the left or right from the top of a column with height h to a neighboring column with height less than or equal to h.

Can the wall always be leveled if the total number of bricks is a multiple of the number of columns?

1 Big "O" notation

If an algorithm's *worst case* runtime is given by a function $T(n)$, where n is the *size* of the input to the algorithm, then we say that its runtime is $O(f(n))$ if there is a constant c and an integer n_0 such that

$$\forall_{n>n_0} |T(n)| \leq c|f(n)|$$

We say that the runtime is *bounded above* by $f(n)$. Big "O" notation expresses the *growth rate* of the runtime function $T(n)$.

It is also called *asymptotic notation* since it is used to compare the runtimes of algorithms as $n \to \infty$.

$T(n)$	Big "O"
$1000 + 10^6 n^2$	$O(n^2)$
$n^3 + 10^6 n^2$	$O(n^3)$
$n^3 + n^2 + 10^6 n$	$O(n^3)$
$20n^2$	$O(n^3)$
$20n^2$	$O(n^2)$
$5log_{10}\, n + 10\sqrt{n}$	
$nlog_2\, n + 10\sqrt{n} + n$	

Some Examples		
Linear Search	$O(N)$	N elements in an array
Binary Search	$O(log\, N)$	
Insertion Sort	$O(N^2)$	
Selection Sort	$O(N^2)$	
Merge Sort	$O(N\, Log\, N)$	Used by Java, optimal runtime
Radix Sort	$O(N\, log_r M)$	Radix r, N elements, range size M
Quick Sort	$O(N\, log\, N)$	With high probability

Quiz 2:
Given an array of `int` or `String` or objects containing keys, determine if a certain value is present in the array.

How do we do this fast?

If we sort the values first, what is the best search strategy and what is its worst case runtime for each search operation?

Quiz 3:
A function takes in an integer array `a[ ]`as its only argument. It returns an integer array of the same size, but does not alter `a[ ]`.

```
int[] shuffle(int [] a);
```

The values in the returned array are the same as those in a[ ], but they have been randomly "shuffled" or permuted.

Describe how the function should work. It should complete in the shortest possible time.

You can use `int N = a.length` to determine how many values are in the array and you can use the `Java Random` class's `nextInt(N)` method to generate random integers.

Quiz 4:
Repeat the exercise using an *in-place* algorithm. The function mutates the input array. void shuffle(int [] a);

The naïve O(N) algorithm is biased. It does not produce a randomized shuffle because there is a higher chance of some values appearing early in the output than others.

Below is an unbiased in place algorithm originally by Fisher and Yates.

This is a simple version by Durstenfeld:

```
// To shuffle an array a of n elements (indices 0..n-1):

  for(int i=0; i<n-1; i++) {
     int j = random integer such that 0 <= j < n-i
     exchange a[i] and a[i+j]
  }
```

Quiz 5:
One steps through integer points on the straight line. The length of a step must be nonnegative and can be one bigger than, equal to, or one smaller than the length of the previous step. What is the minimum number of steps in order to get from x to y? The length of the first and the last step must be 1.

Input and Output
Input consists of a line containing n, the number of test cases. For each test case, a line follows with two integers: $0 \leq x \leq y < 231$. For each test case, print a line giving the minimum number of steps to get from x to y.

Sample Input	**Output for Sample Input**
3 45 48 45 49 45 50	3 3 4

Quiz 6:
You have a pile of square tiles.

With 4 tiles you can make two rectangles, 1×4 and 2×2.

With 12 tiles you can make 3 rectangles, 1×12, 2×6, 3×4.

What is the minimum number of tiles needed to make N different rectangles? You have to use all the tiles for each rectangle.

Number of Rectangles	Number of Tiles
2	4
3	12
4	
5	
6	
7	
8	

Number of Rectangles	Number of Tiles	Number of Rectangles	Number of Tiles
1	1	11	576
2	4	12	360
3	12	13	1296
4	24	14	900
5	36	15	720
6	60	16	840
7	192	17	9216
8	120	18	1260
9	180	19	786432
10	240	20	1680

Quiz 7:
A train leaves LA traveling east at x mph.

A second train leaves Chicago traveling west on the same line at y mph.

The distance between LA and Chicago on this track is d miles.

A bird flies back and forth between the trains at z mph, $z > max(x, y)$

How far will the bird fly before the trains cross?

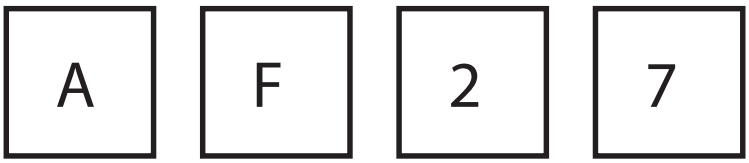
What are the steps necessary to solve this problem?

Quiz 8:
Elmer Fudd is aiming his gun at Buggs, who sits in the branch of a 50 ft tall tree that is about 50 ft away. Just as Elma is about to fire, Buggs drops from the branch and is in free fall.

Should Elmer aim below the wabbit, or at the wabbit?

Quiz 9:
Rule: *If there is a vowel on one face of a card, there is an even number on the opposite face.*

Here are four cards:



Which card(s) must I turn over to prove or disprove the rule?

Quiz 10:
Someone has a basket full of pencils. He tosses them into the air so that they are randomly oriented. Each pencil is equally likely to have any orientation.

Are there more pencils that are nearly horizontal, or more that are nearly vertical, or are the numbers about the same?

Quiz 11:
A man buys 20 pencils for 20 cents and gets three kinds of pencils in return. Some of the pencils cost 4 cents each, some are two for a penny and the rest are four for a penny. How many pencils of each type does the man get? The solution must include at least one pencil of each type.

Citation: The program 'Sunday Week-end Edition' on the US National Public Radio (NPR) network has a 'Sunday Puzzle' segment. The show that aired on Sunday, June 29, 2008 presented the puzzle above. It was taken from a nineteenth century trade card advertising Bassett's Horehound Troches, a remedy for coughs and colds

2 Programming for the UVa online Judge

- Your submission must comprise one source file called Main.java
- The outer class must be called Main
- Do not use the "package" construct
- Input must be from System.in (no files, no command line arguments etc.)
- Output must be to System.out
- Be very careful to follow the specification for input and output data format including spacing, blank lines, periods and other punctuation.

2.1 UVa Problem Specifications

UVa 100: The 3n+1 Problem
Background
Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of a problem whose classification is not known for all possible inputs.

The Problem
Consider the following algorithm: 1. input n 2. print n 3. if n = 1 then STOP 4. if n is odd then n = n*3 + 1 5. else n = n/2 6. GOTO 2

Given the input 22, the following sequence of numbers will be printed

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

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

Given an input n, it is possible to determine the number of numbers printed (including the 1).

For a given n this is called the cycle-length of n. In the example above, the cycle length of 22 is 16.

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

The Input
The input will consist of a series of pairs of integers i and j, one pair of integers per line.

All integers will be less than 1,000,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j. You can assume that no operation overflows a 32-bit integer.

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

Sample Input	Sample Output
1 10	1 10 20
100 200	100 200 125
201 210	201 210 89
900 1000	900 1000 174

2.2 Discussion of the problem specification

The first thing to notice is that the specification is long, typically beginning with some background information that you may not need, but read it anyway.

Next, be careful of words like "between". Does "between 1 and 10" mean $[1, 10]$ or $(1, 10)$? The latter excludes the 1 and the 10. You might have to determine the exact meaning by studying the sample input and output. If in doubt during a contest you should ask the judges for a clarification. In this problem, the exact meaning of "between" is clear.

But there is an even bigger problem here. What if the input data contains the line 200 100? In fact, the UVa judges' test data does contain such a line. Many UVa rejections are because students fail to follow the (unwritten) rules listed at the beginning of this note, OR they fail to spot something ambiguous, misleading, or missing in the language of the problem.

The 3n+1 problem is interesting from a theory standpoint. It is known as the Collatz Conjecture (see the Wiki page). The sequence of numbers produced by the algorithm for a certain starting value of n is called the **Hailstone Sequence.**

```
import java.util.*;
/**
 * Problem UVa 100: The 3n + 1 problem
 */
public class Main {
  public static int cycleLength(long n) {
    int count = 1;
    while (n > 1) {
      count++;
      if((n&1)==0)  // bitwise AND
          n = n/2;  // n even
      else
          n = 3*n+1;  // n odd
    }
    return count;
  }
```

```
  public static void main(String[] args)  {
    Scanner cin = new Scanner(System.in);
    while (cin.hasNextInt()) {
      int a = cin.nextInt();
      int b = cin.nextInt();
      int maxCycleLength = 0;
      int startIndex = Math.min(a, b);
      int endIndex = Math.max(a, b);
      for (int i = startIndex; i <= endIndex; i++) {
         maxCycleLength = Math.max(maxCycleLength,
                                   cycleLength(i));
      }
      System.out.println(a + " " + b + " " + maxCycleLength);
    }
  }
}
```

Challenge 1:
Your first programming challenge is to write a version of this program that has no input.

For $n \in [1, 100, 000, 000]$, it prints n and the largest value reached in the hailstone sequence for n **only if** that largest value exceeds n^2.

The first four lines of output should be:
3 16
7 52
27 9232
31 9232

2.3 Measuring Runtime

You can measure the runtime crudely by adding the following lines to main():

```
    long startTime = System.currentTimeMillis();

      \* do calculations *\

    long endTime = System.currentTimeMillis();
    System.out.println("Runtime = " + (endTime - startTime) +
                                     " milliseconds");
```

A modern PC or Mac can do about 80,000,000 simple steps per second.

Here are measured instruction execution times on a 1.86 GHz MAC Book Air and on the UTD 2.93 GHzLab Machines.

	1.86 GHz Mac Book Air		UTD 2.93 GHz Lab PCs	
INST'N	Time	Factor	Time	Factor
int add	122 ps	1	60pS	1
long add	122 ps	1	60pS	1
int mult	122 ps	1	60pS	1
long multi	133 ps	1.09	60pS	1
int div	350 ps	3	330pS	5.5
long div	1600 ps	14	330pS	5.5

2.4 Root Finding by Binary Search

Consider the problem of finding the integer part of the square root of a given integer. We need to solve for x in the equation

$$x^2 - N = 0$$

for a given value of N. We could use the `Math.sqrt()` function and cast the result to an `int`, but let's assume that the problem was a little more complex and no library function was available for the function. $f(x) - N = 0$.

To simplify the mathematics, let's assume that we know that $f(x)$ increases monotonically with the value of x and that we want the integer part of the solution.
We will use the square root function as an example. We pick a value of x that is too large, $x = x_{max}$, a value that is too small, $x = x_{min}$. We'll call $[x_{min}, x_{max}]$ the current range. Then we compute $x_{mid} = (x_{max}+x_{min})/2$, $f(x_{max})$, $f(x_{min})$, and $f(x_{mid})$. From the three results we can tell which half range contains the result. We use that range to narrow the search further until the desired result is found:

Solution for N = 30					
x_{min}	x_{mid}	x_{max}	x^2_{min}	x^2_{mid}	x^2_{max}
0	15	30	0	225	900
0	7	15	0	49	225
0	3	7	0	9	49
3	5	7	9	15	49
5	6	7	25	36	49

At this point the computation would stop and report the result 5 since the next range would be [5,6] and there is a difference of only 1 between the limits of the range.

Let's try a perfect square:

Solution for N = 64					
x_{min}	x_{mid}	x_{max}	x^2_{min}	x^2_{mid}	x^2_{max}
0	32	64	0	1024	4096
0	16	32	0	256	1024
0	8	16	0	64	256
8	12	16	64	144	256

At this point the computation would stop since $x^2_{min} = 64$.

Quiz Question:

On paper, use this technique to find the integer part of the solution to

$$x^3 - 750 = 0$$

Start with range $x \in [0, 128]$.

3 Sorting Algorithms

In these notes we concentrate on sorting algorithms for an arbitrary sequence of integers, usually presented in an array. These algorithms also work on an array of Strings and of any comparable objects. They are all comparison based, comparing two keys at a time.

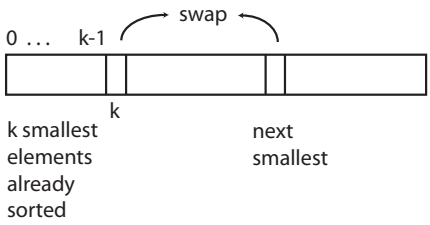
3.1 Selection Sort

Assume that, at some point in the execution of the algorithm, the first k values in the array, A[0..k-1], have already been sorted and are the k smallest values of the entire array. A[k...N-1] remains to be sorted. Initially k=1.

The algorithm searches the A[k...N-1] region for the next smallest value and swaps it with A[k]. Then it increases k by one:

The algorithm has runtime $O(N^2)$.

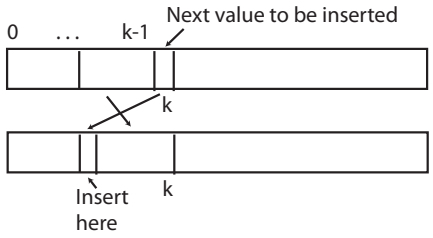
```
selectionSort(int A[]) {
    int N = A.length;
    int k = 0;
    while(k<N-1) {
        int m = posnOfSmallest(A,k,N-1);
            // returns the index of the
            // smallest value in A[k...N-1]
        swap(A,k,m);
            // swaps values at posn's k and m
        k++;
    }
}
```



3.2 Insertion Sort

This algorithm also maintains a growing region of the array, A[0...k-1] that is in order, but those values may not be the k smallest in A. Initially, k=1.

At each stage, the array picks the element in position k and "inserts it" into the sorted region, making the region grow to size k+1.



You can think of the insertion process as
(1) moving the element at position k to a temporary
(2) moving values in A[0...k-1] that are larger than the temporary one place to the right
(3) inserting the temporary value in the space created by the moves

OR, int p = k; while(A[p-1]¿A[p] swap(A,p,p-1); // swaps the values in positions p and p-1 p–; k++;

Both processes yield an algorithm with runtime $O(N^2)$.

3.2.1 Inversions

An inversion is a pair of values in the array (A[i], A[j]), i<j such that A[i]>A[j].

$$\text{In A[ ]} = \{3,1,9,5,4,2\};$$

there are 8 inversions (3,1), (3,2), (9,5), (9,4), (9,2), (5,4), (5,2), (4,2).

If we use insertion sort with the swapping strategy, there will be 8 swaps necessary in total.

```
[3]  1  9  5  4  2   // initially, k = 1, 1 swap needed
[1  3]  9  5  4  2   // k = 2, no swaps needed
[1  3  9]  5  4  2   // k = 3, 1 swap needed
[1  3  5  9]  4  2   // k = 4, 2 swaps needed
[1  3  4  5  9]  2   // k = 5, 4 swaps needed
[1  2  3  4  5  9]   // done. A total of 8 swaps
```

UVa 299: Train Swapping

The train swapper needs to use insertion sort. You only have to count the number of inversions.

UVa 10152: Shell Sort
In this problem we have a stack of turtles and they must rearrange themselves into a specific order. The problem is easier to think about if we use a column of integers:

```
5     Assume that the turtles must be sorted so that the
3     smallest integer is at the top. The only operation
8     available is for a turtle to slip out of the stack
2     and crawl to the top. We must use a minimum number of
1     these operations. The 8 need not be moved since turtles
6     beneath it must crawl over it on their way to the top.
```

The 6 moves up first, then the 5, etc.

What is the algorithm?

3.3 Radix Sort

Consider the radix sort, with radix 10, of the following list for r=10:

$$A[] = \{20, 121, 53, 86, 99, 10, 81, 89, 66, 100\};$$

Place all the values on a linked list

$$L = 20, 121, 53, 86, 99, 10, 81, 89, 66, 100$$

Create r=10 temporary linked lists, d0, d1, d2, ..., d9, all initially empty.

In the first pass go through the original list, copying values into the linked lists, based on their least significant digits. Values may be inserted into the heads or tails of the linked lists.

Then concatenate the lists, d0 + d1 + d2 + $\cdots$ + d9, into a single list and restart the process, selecting values based on their second digits, and so on.

Here are the results.

	After pass 1:	After pass 2	After pass 3
d0 =	20, 10, 100	100	10, 20, 53, 66, 81, 86, 89, 99
d1 =	121, 81	10	100, 121
d2 =		20, 121	
d3 =	53		
d4 =			
d5 =		53	
d6 =	86, 66	66	
d7 =			
d8 =		81, 86, 89	
d9 =	99, 89	99	

The sorted list is the concatenation of d0, d1, ..., d9.

We can use radix sort with any radix, r. The algorithm is particularly suited to the case where the range of the input values is known.

For example, if there are N values to be sorted and we know that $\forall_{0 \leq i < N},\ A[i] \in [0, M]$ then using radix r would result in $\lceil log_r\, M \rceil$ passes, each one taking $O(N)$ time.

The overall run time would be $O(N\, log_r M)$.

Radix sort is faster that other methods if $M < N$ but it requires extra space for the lists.

For $N = 1000$ elements, all in the range 0 to $2^{12} - 1$, with $r = 16$, we would need 16 temporary lists (d0 to d15), and the run time would be

$$const \times 1000 \times \lceil log_{16}\, 2^{12} \rceil = const \times 3000$$

If we change r to 64, we would need 64 lists and and only two passes. The run time would be

$$const \times 1000 \times \lceil log_{64} 2^{12} \rceil = const \times 2000$$

If we set $r = M$, where there are N values to be sorted and the range of input values is $[0, M - 1]$, only one pass is needed. If there are no repeated values and only integers are being sorted, a boolean array can used in place of the N linked lists, and the run time is $O(N)$.

If there are repeated values, then an integer array, count[ ], is used. At the end of the algorithm, count[i] contains the number of times that value i occurred in the input.

But note that, if we have multiple objects sharing the same key, the integer array idea will not work. We would then have to use an array of lists instead, one array entry for each possible key value.

3.4 Merge Sort

Merge sort is often described initially as a recursive function: mergeSort(A[0...N]) = merge(mergeSort(A[0...N/2-1]), mergeSort(A[N/2...N-1)) : N¿1

The merge operation takes two regions of the array that have already been sorted and merges them. Here is an example of the recursive version in operation:

```
initially A = 5 3 7 2 9 8 4 1
mergeSort(5,3,7,2,9,8,4,1) =
merge(mergeSort(5,3,7,2), mergeSort(9,8,4,1)) =
merge(merge(mergeSort(5,3),mergeSort(7,2)),
            merge(mergeSort(9,8),mergeSort(4,1)))
```

In the next level, mergeSort(5,3) becomes
merge(mergeSort(5), mergeSort(3)) = merge(5,3) = (3,5)
and so on.

Recall the input array was A[] = {5 3 7 2 9 8 4 1}.
The lines above, going from bottom to top, become:

```
merge(merge(5,3),merge(7,2)),merge(merge(9,8),merge(4,1)) =
merge((3,5),(2,7)),merge((8,9),merge(4,1)) =
merge(2,3,5,7),(1,4,8,9)) =
(1,2,3,4,5,7,8,9)
```

The runtime is $O(N log\, N)$, which is optimal, but the algorithm uses twice the storage space of an *in place* algorithm.

Merge Sort is a **Stable Sorting Algorithm** meaning that if there are duplicate keys in the input, those keys will remain in their same relative positions in the output.

According to Java 7 API docs, Arrays#Sort() for object arrays now uses TimSort, which is a hybrid of MergeSort and InsertionSort.
But, Arrays#sort() for primitive arrays now uses Dual-Pivot QuickSort.

The algorithm is usually written using iteration and it follows the process above, merging pairs, then quads of elements. The results of one level of merges on array A will be stored in array B. Then we will merge regions from B into A, and so on:

```
   square brackets show the regions to be merged
level 1: A = [(5),(3)] [(7),(2)] [(9),(8)] [(4),(1)]
level 2: B = [(3,5),(2,7)], [(8,9),(1,4)]
level 3: A = [(2,3,5,7),(1,4,8,9)]
finally: B = [1,2,3,4,5,7,8,9]
```

Merge sort works just as well when N is not an integral power of 2. It requires 2N space.

3.5 Quicksort

Quicksort is an excellent in-place algorithm (it requires almost no extra space). Its runtime is $O(N^2)$ but the probability of such a poor result can be reduced to a negligible value. Its average runtime is $O(N log\, N)$ which is optimal. The algorithm is recursive and is almost always implemented that way:

quickSort(A[0...N]) = concatenate(quickSort(A[0...p-1],A[p],quickSort(A[p+1...N-1]))

The concatenation process is vacuous, since the three regions of the array are non-overlapping.

```
quickSort(A,p,q) { // sort the region A[p...q]
    int pivotPosition = choosePivot(A,p,q);
    int pivot = A[pivotPosition];
    swap(A,pivotPosition,q);  // swap pivot with last value
    int i = p;
    int j = q-1; // pos'n of penultimate value in the region
    for(;;) {
        while(A[i]<pivot)
            i++;
        while(A[j]>pivot)
            j--;
```

```
            if(i<j)
                swap(A,i,j);  // swap A[i] and A[j]
            else
                break;
        }
        swap(A,i,q); // put pivot in its correct place
        quickSort(A,p,i-1);
        quickSort(A,i+1,q);
    }
```

```
A = 5   3   7   2   9   8   4   1
pick the pivot 2 and swap it with the rightmost value:
A = [5   3   7   1   9   8   4   2]
     i                       j       // i, j initially refer to the
                                     // first and penultimate elements
     i                       j       // i doesn't move b/c a[i]>pivot
     i           j                   // j moves left until a[j]<pivot
A = [1   3   7   5   9   8   4   2]  // swap A[i] and A[j]
         i                           // i moves right one place
    j                                // j moves to the far left.
                                     // i>j so swap A[i] with pivot
A = [1] [2] [7   5   9   8   4   3]  // This pass of the algorithm
                                     // has completed with 3 partitions,
                                     // [1], [2], [7,5,9,8,4,3]
                                     // the pivot is in the right place
                                     // quickSort is called on the
                                     // regions [1] and [7,5,9,8,4,3]
                                     // quickSort on one element does nothing.

                                     // quickSort is called on [7,5,9,8,4,3]
                                     // and 7 is picked as the pivot
A =         [3   5   9   8   4   7]
             i               j       // i and j are positioned on the first
                                     // and penultimate values
                     i               // i moves right, under the 9
                 j                   // j moves to the pos'n of the 5
                 j                   // i>j so swap the pivot with A[i]
A =         [3   5] [7] [8   4   9]  // now, the region is
                                     // [3,5], [7], [8,4,9] and the pivot 7 is
                                     // in the correct place.
                                     // Calls follow to sort [3,5] and [8,4,9]
```

3.5.1 QuickSort Pivot Selection

On each recursive call the algorithm partitions the current region into three: values that are smaller than the pivot, the pivot, and values that are larger than the pivot. The pivot selection method is extremely important for minimizing the run time of quickSort. An arbitrary selection, such as the first, last, or middle value, leads to $O(N^2)$ run time with reasonably high probability. The *median of three* method has been shown to reduce the probability of the worst case runtime to an acceptably low level. In this method, the median of the first, middle, and last elements of the region is used.

3.5.2 Runtime of QuickSort

Practical implementations of quickSort stop the recursion process when the region is smaller than 20 to 30 elements and call a simple sorting algorithm, such as insertion sort instead. The overhead of the recursion tends to dominate quickSort's performance for small regions.

Notice that i and j both stop if they encounter a value equal to the pivot. In an array of N identical values, there would be many unnecessary swaps with this strategy, but the alternative is to have i and j track to the ends of the region on each recursive call. The result would be the poorest partitioning with sizes m-1, 1, 0 for a region of size m, and a run time for N identical values of $O(N^2)$.

Theorem: The best case run time is $O(N log\, N)$

Theorem: The average run time is $O(N log\, N)$

In Class Exercises: Sorting Problems

1. Given an array A of size N+M containing N sorted values followed by M unsorted values, state the fastest algorithm for sorting the array A if
   (a) $M <<$N
   (b) $M \approx log\, N$
   (c) $M \approx \sqrt{N}$
   (d) $M \approx N$
2. Array A contains N values that are either 0 and 1. Give a fast algorithm to sort it into order, putting all the zero values first.
3. Jo has a bag of N nuts and a bag of N bolts. He knows that there is exactly one nut for each bolt. Joe wants to pair the nuts and bolts.

   Assume that Joe has a large table, but is unable to compare the relative sizes of nut to nut or bolt to bolt. The ONLY comparison method available is to take a nut and a bolt and to try to fit them. The result of such a test is that the bolt is too small, too large, or the right size for the nut.

   Describe in English an efficient algorithm for matching all the bolts and nuts.

UVa 123: Searching Quickly

UVa 102: Ecological Bin Packing
In this problem, it is instructive to compute the number of possible answers. The possible answers are the permutations of the three characters B,C,G:

BCG, BGC, CBG, CGB, GCB, GBC

In general, there are $n!$ permutations of n things, $3! = 6$.
Trying all six permutations will take only a tiny amount of time, so this method is simplest and best.

A modern PC or Mac can do about 80,000,000 simple steps per second.

4 Number Systems

In all number systems an integer has a value and a representation. In decimal notation we use the digit set $d_i \in \{0, 1, 2, \cdots, 9\}$ and the value is expressed as a summation,

For the k digit decimal number $D = d_{k-1}d_{k-2}\cdots d_1 d_0$

$$value(D) = \sum_{i=0}^{k-1} d_i \times 10^i =$$

$$d_{k-1} \times 10^{k-1} + d_{k-2} \times 10^{k-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

Decimal, binary, hexadecimal, octal, and most of the number systems that we use, are **positional number systems**. In positional number systems, the contribution of a digit d_i in position i in the number is given by $d_i \times r^i$ where the radix or base of the number system is r. The value of a number D is then:

$$value(D) = \sum_{i=0}^{k-1} d_i \times r^i$$

We evaluate this summation in the radix of the required answer. That would usually be decimal for most pencil and paper problems.

Roman Numerals do not constitute a positional number system.

What is 1900 in Roman Numerals?
What is MCMLXIV in decimals?
In Hexadecimal (radix 16) the digit set is $d_i \in \{0, 1, 2, \cdots, 9, A, B, C, D, E, F\}$, where A has decimal value 10, B is 11, $\cdots$, F is 15. The value of a HEX number is then:

$$value(D) = \sum_{i=0}^{k-1} d_i \times 16^i$$

For example, $14ADF23C_{16} = 346944060_{10}$.

We mostly use Hex and Octal in Computer Science as a way to represent bit patterns.

There is a simple relationship between binary and hexadecimal:

$$0100\ 0111\ 1010\ 1110\ 0101\ 0110\ 0010_2 = 47AE562_{16}$$

To convert a binary number to Hex we group the binary bit pattern into blocks of 4, starting at the binary point (the right hand end), and convert each group to a HEX digit.

Octal has digit set $d_i \in \{0, 1, 2, \cdots, 7\}$ and radix 8. The value of a k bit octal number is

$$value(D) = \sum_{i=0}^{k-1} d_i \times 8^i$$

Conversion from binary is by grouping into blocks of three:

$$0\ 100\ 011\ 110\ 101\ 110\ 010\ 101\ 100\ 010_2 = 0436562542_8$$

How many decimal digits are necessary to represent a 10 bit signed binary integer?
How many decimal digits are necessary to represent a 20 bit signed binary integer?
How many binary digits are necessary in a 2's complement integer to represent a 5 digit decimal integer?
How many binary digits are necessary in a 2's complement integer to represent a 10 digit decimal integer?

4.1 Reading numbers in strange bases

Say we want to read a number that is in radix 7 and we have already read the number into the String variable `sevenNumber`.

```
int binaryResult = 0;
for(int i=0; i<sevenNumber.length(); i++)
    binaryResult = binaryResult*7 +
                (sevenNumber.charAt(i)-'0');
```

I subtract the ascii code for the character 0 from each character in the String and add the difference to `binaryResult`.
I multiply `binaryResult` by seven on each iteration except the last. I'm using Horner's rule.

Here is an example for a 4 digit radix 7 number:

$$d_3 * 7^3 + d_2 * 7^2 + d_1 * 7^1 + d_0 * 7^0 = ((d_3 * 7 + d_2) * 7 + d_1) * 7 + d_0$$

where each d_i means the value of [`number.charAt(i) - '0'`].

Reading a number that is in Hex notation would require a little more effort since the characters corresponding to the digit set $\{0, 1, 2, \cdots, 9, A, B, C, D, E, F\}$ do not have contiguous codes.

We would have to subtract ('A'-10) from each of the characters {'A', 'B', $\cdots$, 'F'}.

Hex notation also allows lower case characters, making our task even worse.

4.2 Printing numbers in strange bases

To print a positive integer in radix 7, the reverse of the input process is used. We will create the result in the String `outString`. The number to be printed is in the int variable `binaryNumber`.

```
String outString = "";
while(binaryNumber>0) {
    outString = (char)(binaryNumber%7 + '0') + outString;
    binaryNumber = binaryNumber/7;
}
```

4.3 The Remainder Operator, %

I've used the remainder operator % here and I've used String concatenation. The characters of the answer are produced in reverse order, least significant digit first, and prepended to `outstring`

Notice that the character '0' is equally happy being an `int` or a `char`. In the first code fragment I subtract it from a `char`, and in the second I add it to an `int`. `char` is an **integral type**.

If I write `z = x/y;` for integer (or long) values `x, y, z`, what answer should I expect?

The question seems simple enough, but what if one, or both, of the arguments is negative? For example, take a look at the following table:

Dividend	Divisor	Quotient	Remainder
x	y	x/y	x%y
13	7	1	6
-13	7	-1	-6
13	-7	-1	6
-13	-7	1	-6
13	12	1	1
-13	12	-1	-1
13	-12	-1	1
-13	-12	1	-1

The remainder always has the same sign as the dividend, x.
It's best not to think of the % operator as *mod*. Its value is fixed by the rules for division.

4.4 The Java Bitwise Operators

x = `0xF0F0F0F0`, y = `0xFF00FF00`, n = 3			
Symbol	Meaning	Example	Result
~	Complements all bits	y = ~x;	`0F0F0F0F`
&	Bitwise AND	z = x & y	`F000F000`
|	Bitwise OR	z = x |y	`FFF0FFF0`
^	Bitwise Ex-Or	z = x ^y	`0FF00FF0`
<<	Signed left shift	z = x << n	`87878780`
>>	Signed right shift	z = x >> n	`FE1E1E1E`
>>>	Unsigned right shift	z = x >>> n	`1E1E1E1E`

The signed left shift operation "<< n" shifts a bit pattern to the left n places, inserting zeros at the right. It is equivalent to a multiply by 2^n, unless overflow occurs.

The signed right shift operation ">> n" shifts a bit pattern to the right n places, propagating the sign bit to preserve the sign. It is equivalent to an integer division by 2^n.

The unsigned right shift operation ">>> n" shifts a bit pattern to the right n places, inserting zeros at the left.

What is the least significant decimal digit of $1234 \times 5678 \times 9753 \times 4319$?

What is the least significant decimal digit of 7!?

What is the least significant decimal digit of 100!?

Most modern computers use the 2's complement binary representation for integers. It enables integers of positive and negative value to be represented and added, subtracted, etc. with simple logic circuits.

4.5 Java Numbers

A k-bit binary field has 2^k different states. In 2's complement we spread those states almost evenly over negative and positive ranges.

A k-bit value represents all values in the range $[-2^{k-1}, 2^{k-1}-1]$. For example, if $k = 4$, the range would be $[-2^7, 2^7 - 1] = [-8, +7]$.

The positive range is always one smaller than the negative range. This is a consequence of having an even total number of states and the need to use one of them to represent zero.

A Java int is a 32 bit 2's complement number. It's range $[-2^{31}, 2^{31}-1]$. In binary that is [1000 0000 0000 0000, 0111 1111 1111
The most negative value is one larger in magnitude than the most positive value.

Decimal	Binary	HEX	Comment
$2^{31} - 1$	0111 1111 1111 1111 1111 1111 1111 1111	`7FFFFFFF`	has 31 ones.
127	0000 0000 0000 0000 0000 0000 0111 1111	`0000007F`	
7	0000 0000 0000 0000 0000 0000 0000 0111	`00000007`	
1	0000 0000 0000 0000 0000 0000 0000 0001	`00000001`	
0	0000 0000 0000 0000 0000 0000 0000 0000	`00000000`	
-1	1111 1111 1111 1111 1111 1111 1111 1111	`FFFFFFFF`	has 32 ones
-2	1111 1111 1111 1111 1111 1111 1111 1110	`FFFFFFFE`	
-7	1111 1111 1111 1111 1111 1111 1111 1001	`FFFFFFF9`	
-127	1111 1111 1111 1111 1111 1111 1000 0001	`FFFFFF81`	
-2^{31}	1000 0000 0000 0000 0000 0000 0000 0000	80000000	has 31 zeros

The sign bit is the most significant bit. All negative values have sign bits equal to 1. All non-negative values have sign bits of 0.

If we add one to the most positive number, what is the result?
If we subtract one from the most negative number, what is the result?
What is the largest value of an `int` that will not cause overflow if we square it?
Convert 1023_{10} to binary.
How many binary bits would be needed to store the result of $1023 * 1023 + 1023$?

4.5.1 Negation

To negate a value (positive or negative) complement all the bits and add 1.
This is the same as the mathematical formulation:

$$y = 2^{32} - (\text{ the unsigned value of } x)$$

where the unsigned value of x is the value of its bit pattern if we treat the sign bit as having value $+2^{31}$:

$$\text{Unsigned value of x } = \sum_{i=0}^{k-1} x_i 2^i$$

$$\text{Signed value of x } = -x_{k-1}2^{k-1} + \sum_{i=0}^{k-2} x_i 2^i$$

Notice that the assignment $y = -x$ will give the wrong answer (it will overflow) if x has the most negative value, $x = -2^{31}$.

Integers have limited range, as seen above.

Integer overflow has occurred if the result of any integer arithmetic will not fit into the range $[-2^{31},\ 2^{31}-1]$.

Here is what the Java spec' says:

> The built-in integer operators do not indicate overflow or underflow in any way. $\cdots$ the only integer operators that can throw an exception are the integer divide operator / and the integer remainder operator %, which throw an ArithmeticException if the right-hand operand is zero, $\ldots$

We cannot test for overflow, except by doing extra work. Here are some rules:

- If we add two positive integers, overflow will cause a negative result.
- If we add two negative integers, overflow will cause a positive result.
- Adding two integers of differing signs or subtracting two integers of the same sign cannot cause overflow.

In general, if you suspect that intermediate or final values will exceed the range of integers, use the type **long**, or in the extreme, use the **BigInteger** package (see later in these notes).

The range of an int is $[-2^{31}, 2^{31} - 1] \approx \pm 2 \times 10^9$
The range of a long is $[-2^{63}, 2^{63} - 1] \approx \pm 9 \times 10^{18}$
There are no unsigned integral types in Java.

Type	description	Denotation or Range
char	16 bit unicode	'a' or '\u0108' = 'C'
byte	8 bit signed 2's complement	[-128,127]
short	16 bit signed 2's complement	[-32768,32767]
int	32 bit signed 2's complement	$[-2^{31}, 2^{31} - 1]$
long	64 bit signed 2's complement	6654L
float	32 bit IEEE single precision floating point	0.6f
double	64 bit IEEE double precision floating point	0.3d
boolean	length undefined but currently 8-bits	[true, false]

You can put underline characters in numeric literals: 12_345_678.
For Hex numbers the denotation is: `long x = 0xCAFE_BABE;`
For binary numbers we write: `int y = 0b1111_0011_1010_0011;`

UVa 136: Ugly Numbers
In this problem we need to accumulate multiples of 2, 3, and 5 that have no other prime factors and, without repeating any values, print the 1500th such value.

We want only values of form:

$$2^a 3^b 5^c$$

for integer $a, b, c \geq 0$.

Java's PriorityQueue class is ideally suited to this problem. It will keep the values in increasing order, even if you don't insert them in order. But the PriorityQueue allows repeated values to be entered. `add()` and `poll()` each run in time $O(log\, n)$ for n elements in the priority queue.

A TreeSet is ideal for accumulating the values and quickly checking to avoid repeated values. `add()` and `contains()` each take $O(log\, n)$ time.

Say our priority queue is called pq and the set is called set. We start by putting 1 onto pq. Then,

```
until the set contains 1500 values{
   x = (long)pq.poll(); // remove smallest value
   If x is not already in set {
      set.add(x);
      pq.add(x*2);
      pq.add(x*3);
      pq.add(x*5);
   }
}
```

The poll() function removes pq elements in increasing order.
Here are the set and pq elements after 5 iterations. The parens are there to illustrate the groups of three values inserted on each iteration.

set: $1, 2, 3, 4, 5$
pq: $\underline{1}, (\underline{2}, \underline{3}, \underline{5}), (\underline{4}, 6, 10), (6, 9, 15), (8, 12, 20), (10, 15, 25)$

The underlined values are no longer in pq at this point.

4.6 Generating Primes

Jack needs to write a program to print the first 1000 prime numbers. He doesn't know how large the 1000'th prime is, but he is pretty sure it's less than 1,000,000. He concludes that Java type `int` will suffice. The amount of space needed during the computation isn't clear yet.

Here is his first attempt:

```
int testNumber = 1;
int numberOfPrimes = 0;
while(numberOfPrimes<=1000) {
    if(isPrime(testNumber)) {
        numberOfPrimes++;
        System.out.println(testNumber);
    }
    testNumber++;
}
```

Jack knows that there is some debate about whether one is prime, but he chooses to print it anyway.

Like all good (lazy) programmers, Jack leaves the tough problem for last. He puts the logic into a function.

Here is Jack's primality testing function:

```
boolean isPrime(int n) {
    if(n<3)
        return true;
   int divisor = 2;
   while(n%divisor != 0)
       divisor++;
   if(divisor==n)
       return true;
   return false;
}
```

Jack's solution requires minimal storage space, but does it work? His professor says that its runtime will be excessive. Jack comes up with a couple of ideas that might help.

- Instead of testing all integers $1, 2, 3, \cdots$ he decides to test only odd numbers.
- For each testNumber, he will only divide by odd numbers up to the square root of the testNumber.

Here is Jack's second attempt:

```
    System.out.println(1); // why not cheat a little?
    System.out.println(2);
    int testNumber = 3;
    int numberOfPrimes = 2;
    while(numberOfPrimes<=1000) {
        if(isPrime(testNumber)) {
            numberOfPrimes++;
            System.out.println(testNumber);
        }
        testNumber+=2;
    }
    boolean isPrime(int n) {
       int divisor = 3;
       while(divisor <= (int)Math.sqrt(n)) {
           if(n%divisor==0)
              return false;
           divisor+=2;
    }
       return true;
}
```

How can we improve Jack's code?

4.6.1 The Sieve of Eratosthenes

The simplest form of this algorithm uses an array of booleans of size equal to the highest number to be tested for primality. For the millionth prime we would need a boolean array:

```
boolean sieve[ ] = new boolean[1 000 001];
```

The algorithm starts with `sieve[0] = sieve[1] = false` and all other sieve elements `= true`. It then sets all multiples to 2 to false, but not 2 itself. It repeats the process for multiples of 3, and for all multiples of the next value in the array that is still true. Here are the first 20 array elements after multiples of 2. 3, 5 have been sieved out:

0	1	2	3	4	5	6	7	8	9	10
F	F	T	T	F	T	F	T	F	F	F

11	12	13	14	15	16	17	18	19
T	F	T	F	F	F	T	F	T

Multiples of 7 will be sieved out next.

UVa 160: Factors and Factorials

5 Computing with Factorials

Factorial n is defined by the recurrence relation:

$$n! = \begin{cases} 1 & : \quad n = 1 \\ n \times (n-1)! & : \quad n > 1 \end{cases}$$

Recurrence relations are simple to convert to recursive functions:

```
int fact(int n) {
    if(n==0)  // added for completeness
        return 0;
    if(n==1)
        return 1;
    return n*fact(n-1);
}
```

Every recursive function must have boundary conditions or stopping rules. Here we ensure that the function doesn't call itself in an infinite loop by adding the two `if` statements before the recursive call.

The problem is that the factorial function grows very rapidly. For example $20! = 2.4 \times 10^{18}$. That value occupies 62 binary bits and just fits into a Java `long`. 100! has 158 decimal digits and 525 binary bits.

The factorial function turns up in lots of places: The number of ways to choose m things from a set of n things is

$${}^nC_m = \binom{n}{m} = \frac{n!}{m! \times (n-m)!}$$

How many different 5 card hands can you deal from a standard deck of 52 cards?

$$\binom{52}{5} = \frac{52!}{5!(52-5)!} = \frac{52!}{5!47!}$$

If we calculate the three factorials separately the answers will exceed the range of a long ($52! > 8 \times 10^{67}$). We could use doubles, but the answer is an integer and, if we simplify the equation, the computation will not overflow:

$$\frac{52!}{5!47!} = \frac{48 \times 49 \times 50 \times 51 \times 52}{2 \times 3 \times 4 \times 5}$$

There are still problems. For example, consider the number of ways to select 26 cards from a standard deck of 52 cards:

$$\binom{52}{26} = \frac{52!}{26!\,26!} = \frac{27 \times 28 \times \cdots \times 52}{26!} \approx \frac{2 \times 10^{41}}{4 \times 10^{26}}$$

Both dividend and divisor are larger than the range of a long.

5.1 Binomial Coefficients

The same equation is used to compute Binomial Coefficients.

There are plenty of applications for these numbers. Given $(a+b)^n$ we need the coefficients of all the terms of the equations:

$$\begin{aligned}
n = 1 &: a + b \\
n = 2 &: a^2 + 2ab + b^2 \\
n = 3 &: a^3 + 3a^2b + 3ab^2 + b^2 \\
n = 4 &: a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4
\end{aligned}$$

The coefficients are described by Pascal's Triangle:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
\end{array}$$

If the rows are numbered $0, 1, 2, \cdots$ from the top and the entries within a row are numbered $0, 1, 2, \cdots$ from the left, then the $k'th$ coefficient in the $n'th$ row is given by the closed form expression:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The following recurrence relation shows how to compute the elements of a Pascal's Triangle:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{0} = 1, \qquad \binom{n}{n} = 1$$

Using this approach we can build Pascal's Triangle to any required extent. The algorithm uses **Dynamic Programming** (see later).

Here is one possible implementation:

```
long binomialCoefficient(int n, int k) { // compute n choose k
    long bc[100][100];                   // for n, k <= 100
    for(int i=0; i<=n; i++)
        bc[i][0] = 1;
    for(int j=0; j<=n; j++)
        bc[j][j] = 1;
    for(int i=1; i<=n; i++)
        for(int j=1; j<i; j++)
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
    return bc[n][k];
}
```

The equation:

$$\binom{n+m}{n}$$

gives the number of paths across a grid of $(n+1) \times (m+1)$ cells from the upper left cell to the lower right cell where the moves are restricted to *downward one cell* and *to the right one cell*.

We can see that there are $n+m$ total moves in every path. The paths are distinguished by how we mix the horizontal and vertical moves. We choose where to place n horizontal moves among $n+m$ total moves. The result we need is

$$\binom{n+m}{n}$$

5.2 Computing Fibonacci Numbers

Lots of problems concern number sequences defined by recurrence relations. The Fibonacci Number sequence is a classic example:

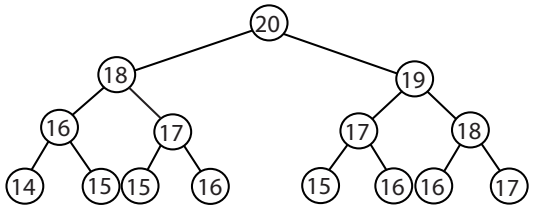
$$Fib(n) = Fib(n-1) + Fib(n-2), \quad Fib(1) = Fib(2) = 1$$

Recurrence relations and recursive functions go hand-in-hand, but we need to be careful of excessive runtime and stack overflow if we use recursion.

Consider the naïve recursive implementation:

```
int fib(int n) {
    if(n<1)
        return -1;
    if(n<3)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

If we use this approach to compute Fib(100) the program will run for years.



The recursive function combines the results of subproblems. Solutions to the same subproblems are recomputed many times.

The recursive version causes $O(2^n)$ calls to `fib(n)`.

Consider a call to `fib(100)`. $2^{100} \approx 10^{30}$ is a huge number. The fastest computer of today, China's Tianhe-2, computes 30×10^{15} steps per second (as of May 2016), it would take that computer 26×10^{12} seconds to find `fib(100)`, which is about 83 million years.

5.3 Dynamic Programming

When we have a recursive solution to a problem that combines solutions to subproblems, **Dynamic Programming** is suggested. There are two forms of DP. Pseudo DP, or **memoization**, is the easiest to program. We add an array to the recursive function to remember solutions to subproblems to avoid recomputing them:

```
int a[] = new int[someLargeNumber]; // added

int fib(int n) {
    if(n<1)
        return -1;
    if(n<3)
        return 1;
    if(a[n]>0)                      // added
        return a[n];                // added
   int result = fib(n-1) + fib(n-2);
   a[n] = result;                   // added
   return result;
}
```

True DP uses an array that holds solutions to subproblems too. The solutions are computed to all subproblems lower in order than the required solution.

For Fibonacci Numbers we only to store the solutions to the two previous subproblems. An array is not necessary. We can have two variables called `previous` and `penultimate`, say, to hold $fib(n-1)$ and $fib(n-2)$ respectively.

A true DP solution does not use recursion. Instead an iterative loop fills table entries until the desired result is obtained.

Here is an `int` version of the Fibonacci function:

```
int fib(int n) {
     if(n<1)
          return -1;
    if(n<3)
          return 1;
    int previous = 1;
    int penultimate = 1;
    int current = 0;
    for(int i=2;i<n;i++) {
        current = previous + penultimate;
        penultimate = previous;
        previous = current;
    }
    return current;
}
```

Clearly this function has linear runtime, $O(n)$.

The integer version has very limited utility since the value of $fib(n)$ grows very rapidly with n.

We can use Benet's formula for $fib(n)$ to determine the largest values of n for which $fib(n)$ fits into an int:

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

For large n, the second term becomes insignificant.

$$fib(n) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$$

Taking logs of both sides and rearranging:

$$n \approx \frac{log(\sqrt{5} \times fib(n))}{log 1.618}$$

This enables us to estimate the value of n for any Fibonacci number $fib(n)$. Substituting $fib(n) = 2^{31} - 1$, the largest positive value of an int, gives $n \approx 46$.

Therefore our `int` function can only compute Fibonacci numbers for $n < 47$.

If we replace `int` by `long`, the result isn't much better. Fibonacci numbers can only be calculated for $n < 93$.

UVa 10183: How Many Fibs
To solve this problem, the Java BigInteger class must be used. The problem requires us to work with values up to 10^{100}. Using the approximation above, $fib(478) \approx 10^{100}$.

```
import java.util.*;
import java.math.BigInteger;
public class Fibs {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        BigInteger low =  BigInteger.ONE;   // low end of range
        BigInteger high =  BigInteger.ONE;  // high end of range
        BigInteger zero =  BigInteger.ZERO; // have to initialize these
        int answer;
        while(low.compareTo(zero)>0 || high.compareTo(zero)>0) {
            low = in.nextBigInteger();
            high = in.nextBigInteger();
            answer = 0;
            if(low.compareTo(zero)>0 || high.compareTo(zero)>0) {
                BigInteger last = BigInteger.ONE;
                BigInteger penult = BigInteger.ONE;
                BigInteger current = BigInteger.ONE;
                while(current.compareTo(low)<0) {
                    current = last.add(penult);
                    penult = last;
                    last = current;
                }


                if(current.compareTo(high)<=0)
                    answer = 1;
                       // add a loop to continue computing
                       // Fibs until current equals or
                       // exceeds high, incrementing answer
                       // as you go.
            }
            System.out.println(answer);
        }
    }
}
```

Replace the comment lines with your code and submit it to the online judge.

6 Optimal change

Given a set of coin denominations and an amount A find a way to make up A using a minimal number of coins.

For example, if we represent the denominations by an array of ints d = [1,5,10,25] and A = 87, the best way to give change would be 3 quarters, one dime, and two cents. That's 6 coins.

We used the largest denomination as many times as possible, then we did the same with the next lowest denomination, and so on. This is a **greedy algorithm**. At each stage it makes a simple step that seems sensible at that point and it never reconsiders any previous step.

For our coin denominations and any **intuitive set of denominations** the greedy algorithm works. But if we introduce a twelve cent coin, the amount 15c would best be made up of 10c + 5c, but the greedy algorithm will always use the largest possible denomination, the 12c, first, giving the wrong answer of a twelve cent coin plus three one cent coins.

6.1 Robot Motion

Many contest problems require a program that simulates the progress of a game, the running of a program, or the motion of an object.

In this problem you are to trace the path of a robot in a 2-D grid. The grid is $p \times q$ in size with *position* going from 1 to $p \times q$. Here is a 3×4 example grid:

1	2	3	4
5	6	7	8
9	10	11	12

The robot's initial position is given as i and a sequence of move commands are given using compass directions, NESW.

In the 3×4 grid:
N reduces the robot's position by 4, unless it is already in the top row, in which case the robot doesn't move.
E increases the robot's position by 1, unless it is already in the east-most column, in which case the robot doesn't move.
S increases the robot's position by 4, unless it is already in the bottom row, in which case the robot doesn't move.
W reduces the robot's position by 1, unless it is already in the west-most column, in which case the robot doesn't move.
Your job is to find the endpoint of the robot's journey after a given sequence of commands.

Input
The input begins with a line containing a single integer N, indicating the number of test cases to follow. The data for N test cases follow. The data for each test case are given as a single line of space-separated parameters: p q i String
where integers p and q give the dimensions if the grid, integer i indicates the robot's start position and String is a string of characters selected from the set $\{N, E, S, W\}$. `String` will not he longer than 80 characters.

Output
For each test case, output a single integer on a line by itself giving the robot's final position.

Sample Input	Sample output
2 3 3 5 NNNNNNN 4 4 10 NESWWSEN	2 10

UVa 10116 Robot Motion

UVa 10041: Vito's Family

Background

The world-known gangster Vito Deadstone is moving to New York. He has a very big family there, all of them living in Lamafia Avenue. Since he will visit all his relatives very often, he is trying to find a house close to them.

Problem

Vito wants to minimize the total distance to all of them and has blackmailed you to write a program that solves his problem.

Input

The input consists of several test cases. The first line contains the number of test cases. For each test case you will be given the integer number of relatives r ($0 < r < 500$) and the street numbers (also integers) $s_1, s_2, s_3, \cdots s_i, \cdots, s_r$, where they live ($0 < s_i < 30000$). Note that several relatives could live at the same street number.

Output

For each test case your program must write the minimal sum of distances from the optimal Vito's house to each one of his relatives. The distance between two street numbers s_i and s_j is $d_{ij} = |s_i - s_j|$.

Sample Input	Sample output
2 2 2 4 3 2 4 6	2 4

It is always worth studying the problem using a simple, but instructive example. For example, assume Vito's relatives live at addresses 0,1,2,3,20. Then build a simple table showing the total distance Vito has to travel to visit all his relatives if he lives at 0,1, 2, 3, . . .

Vito's address	0	1	2	3	4	5	6	7	8	9
Total distance to all relatives	26									

I've left spaces in the table for you to fill in. Once you have done so, determine the best location for Vito to live. Then change the highest address from 20 to 100 and rebuild the table.

Try adding another relative at number 5, so that the relatives now live at 0,1,2,3,5,20.

How would you characterize the solution to this problem?

UVa 10110 Light More Light

The Problem

There is man named "mabu" who switches on and off the lights in our University. He switches on-off the lights in a corridor. Every bulb has its own toggle switch. That is, if it is pressed then the bulb turns on. Another press will turn it off.

To save power consumption (or maybe he is mad) he does a peculiar thing. If in a corridor there are n bulbs, he walks along the corridor back and forth n times and in the i^{th} walk he toggles only the switches whose serial number is divisible by i.

He does not press any switch when coming back to his initial position. An i^{th} walk is defined as going down the corridor (while doing the peculiar thing) and coming back again.

Now you have to determine what is the final condition of the last bulb. Is it on or off? Assume that all bulbs begin in the off state.

The Input
The input will be an integer indicating n, the number of bulbs in a corridor, which is less then or equals $2^{32} - 1$. A zero indicates the end of input. You should not process this input.

The Output
Output "yes" if the light is on otherwise "no", on a single line.

Sample Input	Sample Output
3	no
6241	yes
8191	no
0	

I've added to the problem description to make it readable.

Notice that the value of n is bounded above by $2^{32} - 1$. The range of an int is $[-2^{31}, 2^{31} - 1]$, so a `long` will be needed for n.

7 Flood Filling, Backtracking

You are given an $N \times M$ grid of characters representing a maze. Walls are donated by '*' characters and passageways are donated by space characters. Here is an example:

```
   Input              Output

********** *       **********~*
*  * * *** *       *  * *~***~*
** * *   * *       ** * *~~~*~*
*  * * * * *       *  * *~*~*~*
**   * *   *       **   *~*~~~*
************       ************
```

Assume that two '*' characters that are immediate neighbors in the same row or column are joined. As shown in the example, the perimeter of the maze is a continuous wall, except for one entrance cell on the top row. Now assume that water floods into the maze through that entrance. Your job is to output the maze with '~'characters inserted to show which cells the water floods into, as shown in the figure on the right.

The simplest approach is to employ the *flood fill* algorithm:

```
void floodFill(int i, int j) {
    if(i<0 || i>=M || j<0 || j>=N ||
            maze[i][j]=='*' || maze[i][j]=='~')
```

```
        return;
    maze[i][j] = '~';
    floodFill(i,j+1);  // east
    floodFill(i+1,j);  // south
    floodFill(i,j-1);  // west
    floodFill(i-1,j);  // north
    return;
}
```

This recursive function should be easy to understand. For each cell visited by the function, if that cell is a valid cell and it isn't part of a wall and isn't already flooded, the function floods it. It then calls itself for each of its four neighbors and then returns to its previous invocation.
Here is a table showing the order that the cells are visited, a, b, c, ...

```
 Flood Order

**********a*
*  * *n***b*
** * *kji*c*
*  * *l*h*d*
**   *m*gfe*
***********
```

The algorithm can be used to fill a bounded region, given a starting point that is within that region. The wall cells and the marking of cells that have already been flooded prevent the algorithm from infinitely wandering around the region. Once the fourth recursive call backs out to the invocation of the function at the entrance, all reachable cells have been marked as flooded.

Each cell can be entered by a recursive call from each of its four neighbors. We could reduce the number of recursive calls by checking before each call if the target cell is valid, not a part of a wall, and not already flooded.

The the total number of recursive calls would then be equal to the number of cells that get flooded.

UVa 572: Oil Deposits

7.1 Maze Running

You are given a square grid representing a maze:

7	3	5
2	-2	0
-5	2	6

The objective is to enter at the top-left square and move in NESW directions, square at a time, never revisiting the same square, until you reach the bottom right square.
You cannot enter a square containing a zero.
You must find the route, if one exists, that has the largest sum of the integers in the squares that you visit.

The solution will be based on a recursive function `traverse()`. It calls itself to make progress from cell to cell. Two of its arguments are the row and column of the cell that it is visiting. I also pass in the direction from which the call was made (0=E, 1=S, 2=W, 3=N) because I don't want the function to waste time reentering the cell from which it came. I don't want the function to revisit a cell that it has already visited on the current route. For that reason I add a class variable, a boolean array, `visited[ ][ ]`. I also need to know the best score obtained on any route that ends at the exit square. I use a class variable, an `int bestScore`, that is updated if I reach the exit square with a better score than on any previous route. A boolean class variable `found` is set to `true` when the exit square is reached.

Now let's look at an example. We will assume that the top left corner of the maze corresponds with cell 0,0 of the `maze[ ][ ]` array.

The `traverse()` function is initially called from the north (say) with coordinates 0,0. `visited[0][0]` is set to `true`.
The function then calls itself with coordinates 0,1, and then 0,2 (going east along the top row). Here are the values in the arrays corresponding to this partial route:

Maze			Visited		
7	3	5	T	T	T
2	-2	0	F	F	F
-5	2	6	F	F	F

The function cannot proceed further east, nor can it turn south, so it has to backtrack to cell 0,1. It unsets visited[0,2] as it returns to its previous invocation, at cell 0,1.

Let's assume that the `for` loop in the `traverse()` function attempts to exit a cell first to the east, then to the south, then to the west, and finally to the north.

The next recursive call takes the function south to cell 1,1.

It cannot go east, so it heads south again to cell 2,1. Here are the array contents at that point:

Maze			Visited		
7	3	5	T	T	F
2	-2	0	F	T	F
-5	2	6	F	T	F

The next call is to the east, to the exit cell, 2,2. 16 is recored in the class variable `bestScore` and visited[2][2] is set to `true`.

The function unsets `visited[2][2]` and backtracks to 2,1. The function has exited that cell to the east already and cannot exit to the south, so it exits to the west next. At cell 2,0 the only available (unvisited) direction is north, and that is a dead end. Here are the values in the arrays at that time:

Maze			Visited		
7	3	5	T	T	F
2	-2	0	T	T	F
-5	2	6	T	T	F

The function backtracks all the way to 1,1, where it heads west, then south, then east twice to the exit cell.

This process continues until all valid routes have been explored. The function has backtracked to its initial caller at that point.

```
void traverse(int row, int col,
           int entryDirection, int bestScoreOnThisRoute) {
    if(row==(n-1) && col==(n-1)) {   // at the goal state
        if(bestScoreOnThisRoute>bestScore)
            bestScore = bestScoreOnThisRoute;
        found = true;
        return;
    }
    if(visited[row][col])
        return;   // we have been here before on this route
    visited[row][col] = true;
    for(int direction = 0; direction<4; direction++) {
        switch(direction) {
        case 0: // east
            if(col<(n-1) // are we at the east boundary?
                     && maze[row][col+1]!=0 // can't go there
                     && entryDirection!=2)  // don't go back
                traverse(row, col+1, 0, bestScoreOnThisRoute
                       + maze[row][col+1]);
            break ;

        case 1: // south
            if(row<(n-1)  // are we at the south boundary?
                     && maze[row+1][col]!=0 // can't go there
                     && entryDirection!=3)  // don't go back
                traverse(row+1, col, 1, bestScoreOnThisRoute
                       + maze[row+1][col]);
            break ;
        case 2: // west
                    // code for heading west
            break ;
        case 3: // north
                    // code for heading north
            break ;
        } // end of switch
    } // end of for
    visited[row][col] = false;
} // end of traverse()
```

UVa 929: Number Maze

UVa 11569: Lovely Hint

8 Geometry

A **line** is of infinite length and can uniquely be described by two points $(x_1, y_1), (x_2, y_2)$ where $x1 \neq x2$ or $y_1 \neq y_2$.

A Line can also be described by the familiar equation $y = mx + b$ where the slope is m and the y intercept is b. Horizontal lines are expressed by $y = b_y$ and vertical lines are expressed by $x = b_x$ for constants b_y and b_x.

Another common form is $ax+by+c = 0$ which permits both horizontal and vertical lines within one equation. Since one line can have an infinite number of representations in this form, we standardize by insisting that $b = 1$ if it is not zero and $a = 1$ otherwise.

8.1 Line Segment

A **Line Segment** is the section of a line between two **endpoints** $(x_1, y_1), (x_2, y_2)$.

In Computational Geometry we frequently use parametric form to describe line segments:

$$\begin{aligned} x &= x_1 + (x_2 - x_1)s \\ y &= y_1 + (y_2 - y_1)s \end{aligned}$$

At (x_1, y_1), $s = 0$ and at (x_2, y_2), $s = 1$.

8.1.1 Line Intersection

Two lines intersect unless they are parallel. If they are parallel, they have the same slope.

Using the standardized form, $a_1x + b_1y + c_1 = 0$ and $a_2x + b_2y + c_2 = 0$ for the two lines, they are parallel if:

$$(abs(a_1 - a_2) \leq \epsilon)) \wedge (abs(b_1 - b_2) \leq \epsilon)$$

The intersection point, if $m_1 \neq m_2$, is found by using the formulation $y = m_1x + b_1$, $y = m_2x + b_2$:

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \qquad y = m_1x + b_1$$

8.1.2 The Angle of Intersection

The **Angle of Intersection** is the angle between two lines that intersect. It is given by:

$$tan\,\theta = \frac{a_1b_2 - a_2b_1}{a_1a_2 - b_1b_2} = \frac{m_2 - m_1}{m_1m_2 + 1}$$

Note that the tangent goes to ∞ for perpendicular lines and to 0 for parallel lines.

8.1.3 The Closest Point on a Line to a Given Point

The **closest point on a line to a given point** is often needed.

Given a line L that passes through $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ and a point $P_3 = (x_3, y_3)$ find the coordinates of a point $P_c = (x, y)$ on L that is closest to P_3.

We first compute the parameter u:

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Then

$$\begin{aligned} x_c &= x_1 + u(x_2 - x_1) \\ y_c &= y_1 + u(x_y - y_1) \end{aligned}$$

8.1.4 The minimum distance between a line and a given point

The **minimum distance between a line and a point** is then the length of the segment from P_c to P_3:

$$\sqrt{(x_c - x_3)^2 + (y_c - y_3)^2}$$

8.2 Triangles

The law of sines relates angles and sides. For angles A, B, C and opposite sides a, b, c:

$$\frac{a}{sin\, A} = \frac{b}{sin\, B} = \frac{c}{sin\, C}$$

The law of cosines generalizes Pythagoras' theorem for all triangles. For angles A, B, C and opposite sides a, b, c:

$$a^2 = b^2 + c^2 - 2ab\, cos A$$

8.2.1 The Area of a Triangle

The area of a triangle with coordinates $(a_x, a_y), (b_x, b_y), (c_x, c_y)$ is given by the determinant:

$$\frac{1}{2} \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \frac{1}{2} |a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y|$$

See the Shoelace Theorem later.

8.2.2 The Perimeter of a Triangle

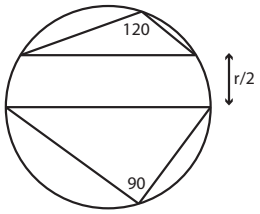
The perimeter of a triangle with sides a, b, c is given by:

$$P = 2s, \qquad s = \frac{a+b+c}{2}$$

where s is called the **semi perimeter**. The area of the triangle in terms of s is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

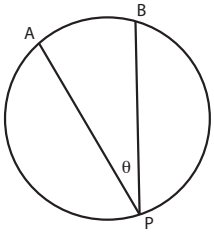
8.3 Circles



The area and perimeter of a circle are πr^2 and $2\pi r$ respectively.

The triangle inscribed in a semicircle has angle 90 degrees.

The triangle inscribed between a chord of a circle that is $r/2$ from the center and the smaller arc is 120 degrees.

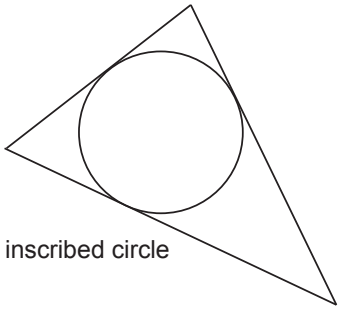


The inscribed angle θ in a circle between two lines PA and PB, where P, A, and B are points on the circumference, is:

$$\theta = \frac{90L}{\pi r} \text{ degrees} = \frac{L}{4r} \text{ Radians}$$

where L is the length of the (shortest) arc between points A and B.

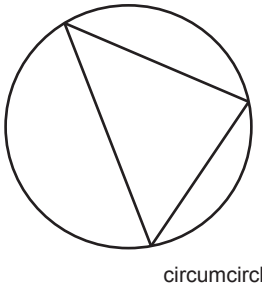
8.3.1 Inscribed Circle



The inscribed circle in a triangle has center at the point where the angle bisectors meet. Its radius is given by:

$$r = \frac{2A}{P} = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$$

8.3.2 Circumcircle



The diameter of the circumcircle is given by the length of any side divided by the sine of the opposite angle. There are multiple alternative ways to express the diameter:

$$2r = \frac{abc}{2 \times Area} = \frac{abc}{2\sqrt{s(s-a)(s-b)(s-c)}} =$$

$$\frac{2abc}{\sqrt{(a+b+c)(-a+b_c)(a-b+c)(a+b-c)}}$$

8.4 Shoelace Theorem

Given any simple polygon (no holes, edges do not cross each other) traverse its perimeter in clockwise or counter clockwise order placing the vertex coordinates into a column matrix. Repeat the start point at the

end:

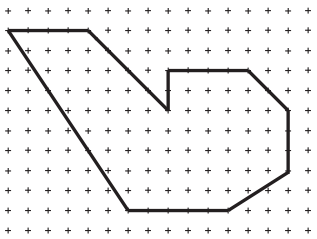
$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \dots & \\ x_n & y_n \\ x_1 & y_1 \end{bmatrix}$$

Then form products and sum them as follows:

$$\text{Area } = \frac{1}{2}\left[(x_1y_2 + x_2y_3 + ... + x_ny_1) - (y_1x_2 + y_2x_3 + \cdots + y_nx_1)\right]$$

If you connect the elements that are multiplied together a shoelace pattern results.

8.5 Pick's Theorem



Given a simple polygon (with no holes or edges that intersect) where all vertices have integer coordinates the area is given by:

$$A = i + \frac{b}{2} - 1$$

where i is the number of internal grid points and b is the number of boundary grid points.

For the above polygon, $i = 61$ and $b = 28$, so the area is $61 + 28/2 - 1 = 74$. Is b always even?

9 Combinations

It is sometimes necessary to generate all combinations of M things from N. For example, given the String ABCDE, all combinations of 3 things from 5 gives:

CDE, BDE, BCE, BCD, ADE, ACE, ACD, ABE, ABD, ABC

corresponding to the bit patterns:

00111, 01011, 01101, 01110, 10011, 10101, 10110, 11001, 11010, 11100.

We can easily generate this list of binary numbers with 3 nested for-loops:

```
for(int i=4;i<32;i=i*2)
     for(int j=2;j<i;j=j*2)
          for(int k=1; k<j;k=k*2) {
               int sum = i+j+k;
               f(sum);
          // f() is the function that uses the bit patterns
          }
```

Or, to generate all subsequences of length 3 from the string ABCDE:

```
String s = "ABCDE";
for(int i=0;i<3;i++)
     for(int j=i+1;j<4;j++)
          for(int k=j+1;k<5;k++) {
               String result = s.substring(i,i+1) +
                  s.substring(j,j+1) + s.substring(k,k+1);
               f(result);
                   // f() uses the Strings
          }
```

In general we need a function,

String combinations(String s, int m)

that generates all the length m subsequences of s.

We need to simulate the s.length - m + 1 nested for-loops.

This can easily be done using recursion. Each call implements another level of for-loop.

Here is a sketch:

```
   // print all subsequences of length m
void combinations(String s, int m) {
   combinations(s,"",0,s.length()-m-1);
}

void combinations(String s, String prefix, int m, int n) {
   if(n==s.length())
      System.out.println(prefix); // or call a f'n that uses them
   else {
      for(int i=m;i<=n;i++)     // next level of for-loop
         combinations(s,prefix+s.substring(i,i+1),i+1,n+1);
   }
}
```

Write a program to input integers n and m and print all bit patterns of length m that contain exactly n ones using this recursive technique. You can use strings to represent the numbers.

Input:
The input will comprise a single line of text, containing two space-separated integers, n and m, $n < m$.

Output:
All bit patterns of length m bits that contain n ones. Print the values in numerical order, one per line:

Sample Input:	Sample Output:
2 5	00011 00101 00110 01001 . . . 10100 11000

10 Permutations

Like the last problem, it is sometimes necessary to generate all permutations, or orderings, of N things. For example, given the string ABC, all permutations of the string are: ABC, ACB, BAC, BCA, CAB, CBA.

The breakthrough is to see that we need each character, or thing, to take up the first place, and then we need to concatenate all permutations of the remaining things. i.e.

```
perm("ABCD") =
"A" + perm("BCD")
   = "AB" + perm("CD"), "AC"+perm("BD"), "AD"+perm("BC"),
   =       "ABC" + perm("D"), "ABD" + perm("C"),
                  "ACB" + perm("D"), "ACD" + perm("B"),
                  "ADB" + perm("C"), "ADC" + perm("B")
   = "ABCD", "ABDC", "ACBD", "ACDB", "ADBC", "ADCB"
```

We repeat the process with "B", then "C", then "D" at the front.
We pass a prefix String and a suffix String into a recursive function that generates the rest. Here is a sketch. Replace the comments with code:

```
  public static void permute(String s) {
     permute("",s);
  }
  private static void permute(String prefix, String suffix) {
     if(suffix.length()>0) {
        if(suffix.length()==1)
           System.out.println(prefix + suffix);   // or pass it to the application
        for(int i=0;i<suffix.length();i++) {
                // partition suffix into three parts: String before = the string before index i
                // char at = suffix.charAt(i), String after = the string after index i
                // then make the following recursive call
             permute(prefix+at,before+after);
        }
     }
  }
```

Write a program that takes in a String from System.in and uses the above function to print all permutations of the characters of that String.

Input:
The input is a single line containing a string of up to 8 characters.

Output:
The output comprises multiple lines. Each one is a different permutation of the input string. These permutations may be listed in any order.

11 Dynamic Programming

In this section we will examine several dynamic programming solutions to problems.

Recall that, if there is a recursive solution to a problem in which solutions to subproblems are computed many times and the overall solution is computed from solutions to subproblems, then dynamic programming is possible.

11.1 The Desert Traversal Problem

You are given an $n \times m$ grid with positive integers in the squares:

1	6	3	4	2
8	5	2	1	3
4	6	1	9	4
1	5	1	3	6

Find the least cost path from any position above the top row to any position below the bottom row. At each step, we can move vertically down, or diagonally down one square. From the cell containing 3 in the top row, we can move to one of the cells containing 5, 2, or 1 in the second row.

The cost of a step is equal to the positive difference between the cell weights. Assume that the areas above and below the grid have weight zero. So, if the chosen path is vertically down the left edge, where the cell weights are 1, 8, 4, 1, the cost of that path is $|0-1|+|1-8|+|8-4|+|4-1|+|1-0| = 1+7+4+3+1 = 16$.

To solve this problem we use Dynamic Programming. We construct an array C of the same width, and one more row than the above grid, where $C_{i,j}$ is the minimum cost of getting from above the grid to cell (i, j). We won't worry about storing the path yet. Obviously, the first row of C is equal to the first row of G. The second row of C is added as follows:

1	**6**	**3**	**4**	2
8	5	**4**	3	3

Consider the middle entry of the 2nd row. There are 3 ways to get to that cell from its neighbors in the first row, from the 6, the 3, and the 4. The costs of those paths are $6+4$, $3+1$, and $4+2$. The middle path is the best, so the corresponding entry in C is 4.

Each row of the array is developed in the same way:

Input Grid				
1	6	3	4	2
8	5	2	1	3
4	6	1	9	4
1	5	1	3	6

DP solution				
1	6	3	4	2
8	5	4	3	3
6	6	3	9	4
7	7	3	5	6
8	12	4	8	12

How do we use the table to construct an optimal path? Clearly the minimum cost path in this problem ends in the middle square of the bottom row. The previous square on the path must be one of its three neighbors in the row above. It must be the middle one, holding 3. Before that square we have the square holding 3, above and to the right, and then the square holding 2 in the top right-hand corner.

The best path is through squares 0, 2, 1, 1, 1, 0 in the input grid with total cost 4.

11.2 Largest Contiguous Subsequence Sum

Given a sequence of integers: $[3, 5, -6, 8, 4, -20, 5, 12, -4, -3, 6]$ find the largest contiguous subsequence sum.

A subsequence is obtained by deleting zero or more elements from a sequence while maintaining the order of the elements that remain.

A contiguous subsequence is a subsequence of elements that are immediate neighbors in the original sequence.

Some subsequences of S = [2 3 1 5 7 9 4 1 7 6 9] are [7], [2,1,9], [5,4,6]. But [6,9,4] and [9,2] are not subsequences of S.

We will assume that at least one value in the sequence is non-negative.

```
/**
 * Cubic-time maximum contiguous subsequence sum algorithm
 */
      public static int maxSubSum1( int [ ] a ) {
          int maxSum = 0;

          for( int i = 0; i < a.length; i++ )
              for( int j = i; j < a.length; j++ ) {
                  int thisSum = 0;
                  for( int k = i; k <= j; k++ )
                      thisSum += a[ k ];
                  if( thisSum > maxSum )
                      maxSum   = thisSum;
              }
          return maxSum;
      }
```

The innermost loop computes the sum of elements from index i to index j inclusive. Then it throws this intermediate result away to compute the sum from index i to index j+1.

```
/**
 * Quadratic-time maximum contiguous subsequence sum algorithm
 */

      public static int maxSubSum2( int [ ] a ) {
          int maxSum = 0;
          for( int i = 0; i < a.length; i++ ) {
              int thisSum = 0;
              for( int j = i; j < a.length; j++ ) {
                  thisSum += a[ j ];
                  if( thisSum > maxSum )
                      maxSum = thisSum;
              }
          }
          return maxSum;
      }


/**
 * Linear-time maximum contiguous subsequence sum algorithm
 */
      public static int maxSubSum4( int [ ] a ) {
          int maxSum = 0, thisSum = 0;
          for( int j = 0; j < a.length; j++ ) {
              thisSum += a[ j ];
              if( thisSum > maxSum )
                  maxSum = thisSum;
              else if( thisSum < 0 )
                  thisSum = 0;
          }
          return maxSum;
      }
```

Study the following dry-run:

index	1	2	3	4	5	6	7	8	9	10	11
value	3	5	-6	8	4	-20	5	12	-4	-3	6
thisSum	3	8	2	10	14	0	5	17	13	10	16
maxSum	3	8	8	10	14	14	14	17	17	17	17

We do not need to add an array. The two values, `thisSum` and `maxSum` are all that we need to compute the next sum value.

This is a Dynamic Programming solution.

11.3 Maximum Contiguous Rectangular Region Sum

UVa 108: Maximum Sum
Given a square array of integers, find the sub rectangle that has the largest sum:

Input			
0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

Max part	
9	2
-4	1
-1	8

Sum = 15

The max sum rectangle could be part of any single row, or part of any two neighboring rows, or part of any neighboring set of rows.

We call the max contiguous subsequence sum function on the sums of every set of neighboring rows.

First we call that function on each row in the input matrix, recording the max sum obtained in a class variable `maxSum`.

Then we construct the following *prefix sum* matrix:

row 1	0	-2	-7	0
row 1 + row 2	9	0	-13	2
row 1 + row 2 + row 3	5	1	-17	3
row 1 + row + row 3 + row 4	4	9	-17	1

From this matrix we can subtract any two rows to get the sum of any contiguous set of rows of the original matrix. There is no need to subtract neighboring rows of the prefix sum matrix since those differences are equal to rows of the input matrix.

We need to compute (row 4 - row 2), (row 4 - row 1), and (row 3 - row 1) and call the max contiguous subsequence sum function on each of these differences, updating `maxSum` each time we find a larger sum.

Here is the input matrix and the max Sum results

Input			
0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

Rows	Max Sum
1	0
2	11
3	1
4	8
1+2	9
2+3	8
3+4	9
1+2+3	6
2+3+4	**15**
1+2+3+4	13

11.4 Longest Increasing Subsequence

In this problem we are asked to find the length of the longest subsequence of increasing values. Given the eleven element sequence:

$$S_{[0,10]} = [2\ \ 3\ \ 1\ \ 5\ \ 7\ \ 9\ \ 4\ \ 1\ \ 7\ \ 6\ \ 9]$$

The length of the longest increasing subsequence appears to be 5 for the subsequence [2,3,5,7,9].

A recursive solution is possible:

$$maxLength(j) = 1 + max(maxLength(i)\ such\ that\ a[i] < a[j]\ and\ 0 \leq i < j)$$

We would begin the recursion at $i = 10$ for the sequence S above. To calculate maxLength[10] we would need to calculate maxLength(j) for all indices $j \in \{0...9\}$. But to calculate maxLength(9) we would need to calculate maxLength(j) for all indices $j \in \{0...8\}$.

There are many recalculations of solutions to the same subproblems. The solution to the entire problem is made up of solutions to subproblems. Therefore, Dynamic Programming is suggested.

We add an array `int maxLength[ ]` to store the solutions to subproblems. Values are filled in from left to right:

index i	1	2	3	4	5	6	7	8	9	10	11
value S[i]	2	3	1	5	7	9	4	1	7	6	9
maxLength[i]	1	2	1	3	4	5	3	1	4	4	5

To calculate maxLength[j], we must examine all S[i] and maxLength[i], for i<j.

The algorithm runs in $O(n^2)$ time.

11.5 Number of Ways to Make Change

This problem requires us to compute the number of ways that a certain dollar amount can be given, using only the set of monetary denominations stated in the problem. For example, we might restrict the denominations to the set $\{1, 5, 10, 25, 50\}$ and ask how many ways can we use any number of coins of each type such that the sum equals 100c.

Once again, a recursive solution is suggested in which solutions to subproblems can be combined to produce the solution to the main problem. An array, $d[\,]$ contains the denominations in increasing order.

$$ways(amount, d_{0\cdots k-1}) =$$

$$ways(amount, d_{0\cdots k-2}) + ways(amount - d_k, d_{0\cdots k-1})$$

For example the number of ways to make up 10 cents using the set 1, 5 is given by:

$$ways(10, [1, 5]) = ways(10, [1]) + ways(5, [1, 5]) = 1 + 2$$

A recursive method can easily be developed from this recurrence, but the runtime is exponential for large values. Once again the problem yields to dynamic programming techniques. A straight recursive approach solves each sub-problems many times.

Dynamic programming replaces the recursion by iteration. We build an array W to hold solutions to subproblems. The array will have a number of columns equal to the number of denominations and a number of rows equal to the amount that we wish to change.
To solve the problem $ways(15, [1, 2, 4, 5])$, we build an array of size 15×4. Row $i = 0$ can be filled in with ones, corresponding to the ways to make zero cents. Column $j = 0$ can also be filled in with ones, corresponding to the ways to make any amount with just one cent coins. This column can be eliminated if the denominations always contain a one cent coin.
Entries in the row $j = 1$ then follow. These entries all equal 1, corresponding to the number of ways to make up 1 cent.
Using these results, we compute entries in the row 2,

$$\begin{aligned} ways(2, [1]) &= ways(2, [\,]) + ways(1, [1]) = 0 + W[1][0] = 1 \\ ways(2, [1, 2]) &= ways(2, [1]) + ways(0, [1, 2]) = W[2][0] + W[0][1] = 2 \\ ways(2, [1, 2, 4]) &= ways(2, [1, 2]) + ways(-2, [1, 2, 4]) = 2 \\ ways(2, [1, 2, 4, 5]) &= ways(2, [1, 2, 4]) + ways(-3, [1, 2, 4, 5]) = 2 \end{aligned}$$

Using the array notation, and asserting that for $i < 0$ or $j < 0$, W[i][j]= 0:

$$\begin{aligned} W[2][0] &= W[2][-1] + W[1][0] = 0 + 1 = 1 \\ W[2][1] &= W[2][0] + W[0][1] = 1 + 1 = 2 \\ W[2][2] &= W[2][1] + W[-2][2] = 2 + 0 = 2 \\ W[2][3] &= W[2][2] + W[-3][3] = 2 + 0 = 2 \end{aligned}$$

Here is the array W:

Amount i	Denoms [1] $j = 0$	[1,2] $j = 1$	[1,2,4] $j = 2$	[1,2,4,5] $j = 3$
0	1	1	1	1
1	1	1	1	1
2	1	2	2	2
3	1	2	2	2
4	1	3	4	4
5	1	3	4	4
6	1	4	6	7
7	1	4	6	8
8	1	5	**9**	11
9	1	5	9	13
10	1	6	12	17
11	1	6	12	19
12	1	**7**	**16**	24
13	1	7	16	27
14	1	8	20	33
15	1	8	20	37

We fill in the array row at a time from top to bottom and left to right until the required entry is reached.

A new entry in row k column j is computed from the entry to its left and an entry in the same column, denom[k] positions above it.

To compute W[12][2] we add W[12][1] and W[12-demon[2]][2]. The result is W[12][1] + W[8][2] = 7 + 9 = 16.

Similarly W[12][3] = W[12][2] + W[12-denom[3]][2] = W[12][2] + W[7][2] = 16 + 8 = 24.

11.6 Optimal Change with Unintuitive Denomination, Recursive Solution

Earlier we computed the minimum number of coins to make up a given amount using a greedy method. That method only works when the denominations set is "intuitive."

Consider the denominations set [1,5,10,12,25]. Fifteen cents can be made up with three coins (3*5) but the greedy method will give 4 coins (1*12 + 3*1).

How do we solve the more general change problem with arbitrary denominations?

We compute the ways to make change trying just one member of the denomination set at each stage:

```
BestChange(15) = min (
                            1 + BestChange(14),         // one 1c
                            1 + BestChange(10),         // one 5c
                            1 + BestChange(5),          // one 10c
                            1 + BestChange(3),          // one 12c
                                     . . .              )
```

Since the next denomination, 25, is greater than the target amount of 15c, we need go no further than the 12c denomination.

We develop the above into a recursive function that uses a class array denoms[ ].

```
int minCoinsToChange(int amount) {
   int minCoins = amount;  // worst case using all 1c pieces
   if(amount<=0)
      return 0;

   for(int i=0;i<demoninations.length;i++) { // recursive splits
      if(amount>denomonations[i] {
          int bestWithThisCoin = 1 +
                minCoinsToChange(amount - denominations[i]);
          if(bestWithThisCoin<minCoins)
              minCoins = bestWithThisCoin;
      }
   }
   return minCoins;
}
```

Write a program that uses this function.

Input:
The input will comprise a number of test cases. First, an integer n on a line by itself will specify the number of test cases to follow. Each test case will comprise a single line of text. The first value will be the target amount, the amount that must be changed. Then comes N, the number of denominations, which is followed by N denomination values, given in increasing order.

Output:
For each test case, output a single integer on a line by itself, giving the minimum number of coins to make up that amount of change.

Sample Input:	Sample Output:
4	3
23 5 1 5 10 12 25	1
12 5 1 5 10 12 25	5
53 5 1 5 10 12 25	2
60 15 1 3 5 7 11 13 17 19 23 29 31 37 41 43 47	

11.7 Optimal Change with Unintuitive Denomination, DP Solution

We solve the same problem using dynamic programming. It's easy to see that the last solution had long runtime because it recomputed solutions to the same subproblems over and over. We add an array that holds solutions to all subproblems and fill in the array from the beginning.

For example, if we have already computed the minimum number of coins to change amounts 0, 1, 2, $\cdots$, 20 with denominations 1, 5, 12, 25, we calculate the best way to change 21 cents by considering the best ways to make change for amounts 21-1, 21-5, and 21-12. i.e. we simulate cases where the last coin added is each of the denominations. We exclude cases where the denomination is greater than the amount being changed.

Here are the array contents before we compute the value for amount 21. Recall the denominations are [1, 5, 12, 25]

`i`	`minCoins[i]`
0	0
1	1
2	2
3	3
4	4
. . .	
9	5
. . .	
16	4
. . .	
20	4
21	?

We examine entries for i = 21-1, i = 21-5, and i = 21-12, select the smallest entry, and add one.

minCoins[20] = 4, minCoins[16] = 4, and minCoins[9] = 5, so we insert the value 5 for minCoins[21].

Here is a sketch of the code:

```
int minCoinsToChange2(int amount) {
   int minCoins[] = new int[amount+1]; // array to hold soln's
   minCoins[0] = 0;
          // work up to amount, filling minCoins from 1
   for(int cents = 1; cents <= amount; cents++) {
      int bestCoins = cents;   // all 1c coins is worst case
         // try all denominations to see if
         // 0 <= (cents - denoms[j] + 1) < bestCoins
         // if so, bestCoins = cents - denoms[j] + 1
      coinsUsed[cents] = bestCoins;
   }
   return minCoins[amount];
}
```

Replace the three comment lines with a for loop. Write a program to test your function.

The input and output specifications will be the same as for the last problem.

Sample Input	Sample Output
6	3
39 5 1 5 10 13 25	4
40 8 1 3 5 7 11 13 17 19	4
50 8 1 3 5 7 11 13 17 19	3
40 8 1 2 3 5 8 13 21 34	3
50 8 1 2 3 5 8 13 21 34	4
51 8 1 2 3 5 8 13 21 34	

11.8 Strings, Edit Distance = Levenshtein Distance

Java has an excellent String class with many very powerful functions. Let's examine some common problems and their solutions.

The *Levenshtein distance* is a metric for measuring the amount of difference between two sequences. The term *edit distance* is synonymous with Levenshtein distance. The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being **insertion, deletion**, or **substitution** of a single character. It is named after Vladimir Levenshtein.

The DP solution uses an array `d[ ][ ]` to record the edit distances as they are computed. For strings `s` and `t`, `d[i][j]` holds the edit distance between the first `i` characters of `s` and the first `j` characters of `t`.
Here is the essential recurrence relation:

```
if( s.charAt(i) == t.charAt(j) )
      d[i][j] = d[i-1][j-1];
else
      d[i][j] = min(d[i][j-1],d[i-1][j],d[i-1][j-1]) + 1;
```

Here is the code:

```
   int editDistance(String s, String t) {

      int m = s.length(); // chars in a String begin at index 0
      int n = t.length();

      // d[i][j] will hold the Levenshtein distance
      // between the first i characters of s and
      // the first j characters of t;
      int d[ ][ ] = new int[m+1][n+1];

      for(int i=1; i<=m; i++) // set first column to i
         d[i][0] = i;  // Equivalent to an empty second String

      for(int j = 1; j<=n; j++) // set first row to j
         d[0][j] = j;  // Equivalent to an empty first String

      for(int j=1; j<=n; j++)
      {
         for(int i=1; i<=m; i++)
         {
            if(s.charAt(i-1) == t.charAt(j-1))
               d[i][j] = d[i-1][j-1];
            else
               d[i][j] = Math.min(
                    d[i-1][j] + 1,          // a deletion
                    Math.min(d[i][j-1] + 1, // an insertion
                         d[i-1][j-1] + 1    // a substitution
                    )
               );
         }
      }
   }
   return d[m][n];
}
```

Here is the array after procession the call with s = “start” and t = “trap”:

		t	r	a	p
	0	1	2	3	4
s	1	1	2	3	4
t	2	1	2	3	4
a	3	2	2	2	3
r	4	**3**	**2**	3	3
t	5	**4**	3	3	4

The first row, under the word “trap” corresponds to an empty first string. In column i we see that i changes would be necessary to change the empty String into “trap”.

Next consider the row next to the character ‘a‘ in “start”. The numberers are 3 2 2 2 3.

Three changes would be necessary to make “sta” equal to the empty string.
Two changes would be necessary to make “sta” equal to “t”: delete the ‘s’ and the ‘a’.
Two changes would be necessary to make “sta” equal to “tr”: delete ’s’ and substitute ‘r‘ for ’a’.
Two changes would be necessary to make “sta” equal to “tra’: substitute “tr” for “st”.
Three changes would be necessary to make “sta” equal to “trap’: substitute “tr” for “st” and insert ‘p’.

Now consider the bolded region in the array containing the three cells that were used to compute cell 5,2:

		t	r
.	.	.	.
r	.	3	2
t	.	4	

Cell 4,2 (the 2) corresponds to the number of edits to make “star” equal to “tr”.
Cell 5,1 (the 4) corresponds to the number of edits to make “start” equal to “t”.
Cell 4,1 (the 3) corresponds to the number of edits to make “star” equal to “t”.
Since s[5]$\neq t$[2], the algorithm chooses the smallest of these and adds 1 to compute the value (3) for cell 5,2.

11.9 Longest Common Substring

Do not confuse this problem with the **Longest Common Subsequence** problem. We will examine that problem next.

A substring of a string S is a contiguous sequence of characters in S. If S = “ABACAD” then “ABAC”, “BACA”, and “C” are substrings of S, but “BCD” is not.
Our problem is, given two strings $S_{0\cdots p-1}$ and $T_{0\cdots q-1}$, we must find the longest common substring.
To refine our problem further, we define $LCS(S,T)$ to be the length of the longest common substring between S and T.
The insight into the Dynamic Programming Algorithm comes from noticing that, if characters u of S and v of T are the right-most pair of equal characters, then

$$LCS(S_{0\cdots u}, T_{0\cdots v}) = 1 + LCS(S_{0\cdots u-1}, T_{0\cdots v-1})$$

The DP algorithm uses an iterative process to build a 2-D array L. Here is an example with S = “ABAB” and T = “BABA”:

	j=	0	1	2	3
i=		B	A	B	A
0	A	0	1	0	1
1	B	1	0	2	0
2	A	0	2	0	3
3	B	1	0	3	0

The array is filled in row at a time, beginning with row i=0. For each character in T that equals S_0 = ’A’ a one is inserted. A zero is inserted otherwise.

In the remaining rows, if S[i] = T[j], L[i][j] = 1+ L[i-1][j-1], L[i][j]=0 otherwise. The largest value(s) in the array marks the end(s) of the longest substring(s).
The two 3's in the array mark the endpoints of the longest common substrings in S and T.
Their positions give the endpoints and the 3s give the lengths of the two equally good common substring in S, "ABA" and "BAB".
Substring "ABA" ends at S_3 and T_4, while "BAB" ends at S_4 and T_3.

	c	r	i	c	k	e	t
r	0	**1**	0	0	0	0	0
i	0	0	**2**	0	0	0	0
c	1	0	0	**3**	0	0	0
k	0	0	0	0	**4**	0	0
e	0	0	0	0	0	**5**	0
t	0	0	0	0	0	0	**6**
s	0	0	0	0	0	0	0

The Longest Common Substring ends at L[5][6] and has length 6 = "ricket"

```
/*
    Update rule:\\
    if(S[i]==T[j])
       L[i][j] = L[i-1][j-1] + 1;
    else
       L[i][j] = 0;
*/

 Set<String> LCSubstr(String S, String T) {
    int p = S.length();
    int q = T.length();
    int L[][] = new int[p][q];
    int z = 0; // length of the longest common substring so far
    Set<String> ret = new HashSet<String>(); // the LCS Strings
    for( int i=0; i<p; i++ )
        for( int j=0; j<q; j++) {
            if( S.charAt(i) == T.charAt(j) ) {
                if( i=0 || j=0 )
                    L[i][j] = 1;
                else              // i>0, j>0
                    L[i][j] = L[i-1][j-1] + 1;
                if( L[i][j] > z ) {
                    z = L[i][j];
                    ret.clear(); // empty the set
                }
                else
                   if( L[i][j] == z )
                      ret.add(SS.substring(i-z+1,i));
            } // end if
        } // end for
        return ret;
  }
```

The set `ret` accumulates the Longest Common Substrings.

This algorithm runs in $O(pq)$ time where the lengths of the two strings are p and q. But this result neglects the time necessary to copy Strings into ret. A better approach would be to use a Set of integer pairs, $(endIndex, length)$ that store the end index in S and length of each LCS.

It is simple to recode the algorithm to use only one row instead of a a 2-D array.

11.10 Longest Common Subsequence

Recall that we obtain a subsequence of a sequence S by deleting zero or more elements of S but retaining the order of the remaining elements. Given two Strings S = "ABCACAD" and T = "ABRA" the longest common subsequence is "ABA". The algorithm is similar to the one for Longest Common Substring, but the update part is different. A 2-D array C keeps track of the largest common subsequences. C[i][j] contains the length of the longest common subsequence in $S_{0\cdots i}$ and $T_{0\cdots j}$.

```
if( S[i] == T[j])
    C[i][j] = 1 + C[i-1][j-1];  // diagonal element
else
    C[i][j] = Math.max(C[i-1][j], C[i][j-1]);  // max of elements to the left and above
```

Here is an example using S = [C,E,B,F] and T = [E,F,B,F].

	j=	0	1	2	3
i=		E	F	B	F
0	C	0	0	0	0
1	E	1	1	1	1
2	B	1	1	2	2
3	F	1	2	2	3

```
int LCSubseq(String S, String T) { // returns the length
   int p = S.length();             // of the LCS
   int q = T.length();
   if( p==0 || q==0)
      return 0;
   int C[] = new int[p][q];
   for( int i=0; i<p; i++ )
      for( int j = 0; j<q; j++) {
         if( S.charAt(i) == T.charAt(j) ) { // a match
            if( i==0 || j==0 ) C[i][j] = 1;
            else C[i][j] = C[i-1][j-1] + 1;
         }
         else  { // S[i] != T[j]
            if( i==0 ) {
               if( j==0 ) C[i][j] = 0;
               else C[i][j] = C[i][j-1];
         }
```

```
            else { // i>0
               if( j==0 ) C[i][j] = C[i-1][j];
               else C[i][j] = Math.max(C[i][j-1], C[i-1][j])
            }
         } // end for j
         return C[p][q];
   } // end LCSubseq
```

This table shows the common subsequences corresponding to the lengths given in the table above.

	j=	0	1	2	3
i=		E	F	B	F
0	C	ϕ	ϕ	ϕ	ϕ
1	E	(E)	(E)	(E)	(E)
2	B	(E)	(E)	(EB)	(EB)
3	F	(E)	(EF)	(EF)	(EBF)

A simple backtracking procedure, beginning at C[p][q] is used to extract the common subsequences.

	A	C	T	O	R
C	0	1	1	1	1
H	0	1	1	1	1
A	1	1	1	1	1
R	1	1	1	1	2
A	**1**	1	1	1	2
C	1	**2**	2	2	2
T	1	2	**3**	**3**	3
E	1	2	3	**3**	3
R	1	2	3	3	**4**

The bold characters show the trajectory of the extraction algorithm. It starts at C[8][4]. The LCS string is given by those characters along this path where S[i]=T[j], as indicated by the underlined entries.

The Longest Common Subsequence has length 4. It is "ACTR".

Here is an algorithm to recover **one** of the longest subsequences:

```
String backTrack(int C[][], String S, String T, int i, int j) {
          // top level call has i=p, j=q
    if( i == 0 || j == 0 )
        return "";
    if( S.charAt(i) == T.charAt(j))
        return backTrack(C, S, T, i-1, j-1) + S.charAt(i);
    else
        if( C[i][j-1] > C[i-1][j])
            return backTrack(C, S, T, i, j-1)
        else
            return backTrack(C, S, T, i-1, j)
}
```

There are improvements given in the literature to the Longest Common Subsequence Algorithm that reduce the space requirements from pq to $1 + min(p, q)$. The runtimes of these algorithms is $O(pq)$.

11.11 UVa 10003: Cutting Sticks

You have to cut a wooden stick into pieces. The cost of a cut is equal to the length of the stick being cut. Only one cut can be made at a time. Making the cuts in different orders can lead to different prices. For example, consider a stick of length 10 meters that has to be cut at 2, 4 and 7 meters from one end. If we cut first at 2, then at 4, then at 7, the price will be $10 + 8 + 6 = 24$ because the first stick was of 10 meters, resulting in 8 and the last one of 6. Another choice could be cutting at 4, then at 2, then at 7. This would lead to a price of $10 + 4 + 6 = 20$, which is a better price. You have to find the minimum cost for cutting a given stick.

Input:
The input will consist of several cases. The first line of each test case will contain a positive number L that represents the length of the stick to be cut, (L $<$ 1000). The next line will contain the number n (n $<$ 50) of cuts to be made.
The next line will consist of n positive numbers c_i $(0 < c_i < L)$ representing the places where the cuts have to be made, given in strictly increasing order.
An input case with L = 0 will represent the end of the input.

Output:
You have to print the cost of the optimal solution of the cutting problem, that is the minimum cost of cutting the given stick. Format the output as shown below.

Sample Input	Sample Output
100 3 25 50 75 10 4 4 5 7 8 0	The minimum cutting is 200. The minimum cutting is 22.

Here is a sketch of the code:

```
// cuts[] contains $[0, c_1, c_2, c_3, . . ., c_n, L] $
// where the stick has length L
// start and fin are the indices into cuts[] for this call.
// Top level call: int answer = cut(cuts,0,n+1) for n cuts

  int cut(int cuts[], int start, int fin) {
     if(costTable[start][fin]>0)        // pseudo DP
        return costTable[start][fin];   // pseudo DP
     if((fin-start)<2)
        return 0;
     int bestCost = 10000000;
     for(int i=start+1;i<=fin-1;i++) { // try every cut in the
                                       // range as the next cut
        int cost1 = cut(cuts,start,i);
```

```
        int cost2 = cut(cuts,i,fin);
        int cost =  cuts[fin] - cuts[start] + cost1 + cost2;
        if(cost<bestCost)
           bestCost = cost;
     }
     costTable[start][fin] = bestCost;  // pseudo DP
     return bestCost;
  }
```

Without the pseudo DP lines the above function would recompute answers to subproblems and would run in time $O(2^L)$.
A true DP solution exists but it's too complicated for this course.

11.12 UVa 562: Dividing Coins

Given a bag with a maximum of 100 coins, determine the most fair division between two persons. This means that the difference between the amount each person obtains should be minimized. The value of a coin varies from 1 cent to 500 cents. It's not allowed to split a single coin.

Input:
A line with the number of problems n, followed by n times:
a line with a non negative integer m ($m \leq 100$) indicating the number of coins in the bag
a line with m numbers separated by one space, each number indicates the value of a coin.

Output:
The output consists of n lines. Each line contains the minimal positive difference between the amount the two persons obtain when they divide the coins from the corresponding bag.

Sample Input	Sample Output
2	0
3	1
2 3 5	
4	
1 2 4 6	

You have a list of coin values, from 0 to 100 coins, each of the values is in [1,500]. There can and will be repeated values. You have to divide the set of coins into 2 non-overlapping subsets such that the value difference is minimized.

For example, given the set {1, 1, 2, 3, 100}, the best partition is 1,1,2,3 100, giving a value difference of 100 - 7 = 93. We will use dynamic programming to solve this problem.

Let's consider another problem, where the coins have values {3,5,6,8}.

The sum is 22, so we want a partition where each subset has value as close to 11 as possible. In general, for N coins, the two subsets could have sizes anywhere from (1,N-1) to (N/2,N/2).

We use an int array `subTotals[N-1][X]` and initialize it to all zeros. We will consider the value of X later.

Then we fill in the top row with all the values you can get with just the first coin. i.e. 3.

In the second row, we list all the values that can be got using any combination of the first two coins{3,5}. Here is the array after these 2 iterations:

3	.	.	.	.				
[3]	[5, 8]	.	.	.	=	3	5	8

The 2nd row is easily constructed from the first. Let's add the third row, incorporating the 6 coin:

3	.	.	.	.	.	.	.	.		
3	5	8	.	.	.	.	.	.		
[3, 5, 8],	[6, 9, 11, 14]	=	3	5	6	8	9	11	14	

Once we reach or exceed the target value of 11 there is no need to generate further values. I've used the brackets to show how the calculation is done. The first bracket is just the previous row.

The second bracket contains the new number (6) followed by a list of values made up of that number added to all the values in the first bracket. You could do the calculations that way, storing the two brackets into separate lists (vectors) and then merging them back into the array.

Finally (in this example we don't need this step, but I'm including it to illustrate the process) we add the 8 coin.

3	.	.	.	.	.	.	.	.	.		
3	5	8	.	.	.	.	.	.	.		
3	5	6	8	9	11	14					
[3, 5, 6, 8, 9, 11, 14]	[8,11,13,16,14,17,19,22]	=	3	5	6	8	9	11			

Values larger than 11 have been excluded. Using (2 of) the first 3 coins, we can get a partitioning where the two sums are exactly 11. We do not need to generate the fourth row, since an 11 occurs in the 3rd row.

In this problem, we only have to print the positive difference between the total values of the 2 subsets, so we print 0.

What if you had to print the coin values in each subset?

If you retained the bracketed notation and ordering of values above, it would be easy to backtrack to the individual coins. Then, instead of the 2-D array, only one vector would be needed, plus an int array to remember where the brackets were:

[3][5,8][6,9,11,14][8,11,13,16,14,17,19,22] would be stored in 2 vectors as:

3,5,8,6,9,11,14,8,11,13,16,14,17,19,22
1,3,7

The 2nd vector indicates where the brackets go.

A search from left to right in the first vector yields an 11 in the third bracket. that bracket begins with a 6, so a 6 is needed. The position of the 11 (and/or its value) indicates that the 11 is made up of 6+5. We look

back to the previous bracket to find the 5. It is the first number in that bracket, so no further searching is needed. One of the subsets is 6+5. the other is made up of the remaining values, 3 and 8.

The length, X, of the first vector, grows exponentially with the number of rows. For 100 coins, the space requirement would be prohibitive. But we do not need to store subtotals that are larger than the target amount. Indeed, the process should stop as soon as the target value is reached or exceeded. The largest possible target value is 100*500/2, so X would need to be that large if a simple array is used. A vector or an Arraylist would grow as needed until the target value was reached.

But in this problem we only need to compute the minimal difference between the sums of coin values in the two partitions.

We modify the algorithm to use a boolean array, B[], such that B[i] = true if and only if the value i can be made up from the coins.

We first sort the given coin values into increasing order and consider the coins in that order. We sum the coins and divide by two to get the target value and we stop as soon as we reach or exceed the target value.

Note that in Java the length of a boolean is not specified and is, at present, one byte.

11.13 Buying Pencils

A man buys 20 pencils for 20 cents and gets three kinds of pencils in return. Some of the pencils cost 4 cents each, some are two for a penny and the rest are four for a penny. How many pencils of each type does the man get?

A clarification provided to the problem indicated that correct solutions would contain at least one of each pencil type.

This is an enhancement of the problem that originally aired on the NPR show. Rather than just considering 20 pencils for 20 cents, consider the case of N pencils for N cents.

Given a value of N , determine if a solution is possible, and if so, determine all possible solutions.

Input
Each input line, except the last, contains a value of N in the range 1 to 256 for which the problem is to be solved. The last input line contains the integer 0.

Output
For each value of N in the input, display the case number (1, 2, ...) and the phrase 'N pencils for N cents' as shown in the sample output below. If there are no solutions for a particular value of N , then display the line 'No solution found.' If there are solutions, display three lines for each one, separating the groups of three lines for each solution by a blank line. Order these solutions by increasing numbers of four-cent pencils. Display a blank line after the output for each case.

Sample Input	Sample Output
10	Case 1: 10 pencils for 10 cents
20	No solution found.
40	
0	Case 2: 20 pencils for 20 cents
	3 at four cents each
	15 at two for a penny
	2 at four for a penny
	Case 3: 40 pencils for 40 cents
	6 at four cents each
	30 at two for a penny
	4 at four for a penny
	7 at four cents each
	15 at two for a penny
	18 at four for a penny

12 Graphs

Graphs are fundamental structures in Computer Science. We will study many types:

Undirected	The edges are all bidirectional
Directed	All edges are directed
Unweighted	All edges have weight 1
Weighted	all edges have weights

Vertices contain keys or ID numbers and edges represent connections or relationships. Street maps, prerequisite charts, and electrical circuits are simple examples of graphs.

Formally, $G = (V, E)$ where $V = \{v_i$ such that v_i is a vertex $\}$,
$E = \{(i, j)$ such that $i \in V$ and $i \in V\}$.

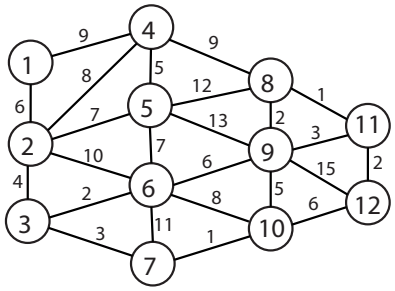
$|V|$ is the number of vertices.
$|E|$ is the number of edges.

A graph may be **completely connected**, in which every vertex is connected to every other vertex and $|E| = |V|(|V| - 1)/2$.

A dense graph has $O(|V|^2)$ edges. A sparse graph has $O(|V|)$ edges.

A graph may be cyclic or non-cyclic. A directed or undirected path that visits any node more than once constitutes a cycle.

Here is an undirected weighted graph:



The graph above may be represented by an array of ArrayLists, one ArrayList for each vertex, containing the neighbors of that vertex. A similar structure contains the corresponding edge weights:

12.1 Adjacency List Structure

Vertex	Neighbor List	Corresponding Edge Weights
1	2, 4	6, 9
2	1, 4, 5, 6, 3	6, 8, 7, 10, 4
3	2, 6, 7	4, 2, 3
4	1, 2, 5, 8	9, 8, 5, 9
5	2, 4, 8, 6, 9	7, 6, 12, 7, 13
6	2, 5, 9, 10, 7, 3	10, 7, 6, 8, 11, 2
7	3, 6, 10	3, 11, 1
8	4, 5, 9, 11	9, 12, 2, 1
9	8, 11, 12, 10, 6, 5	2, 3, 15, 5, 6, 13
10	9, 12, 7, 6	5, 6, 1, 8
11	8, 9, 12	1, 3, 2
12	11, 9, 10	2, 15, 6

The adjacency list structure is most suited to sparse graphs.

12.2 Adjacency Matrix

The adjacency matrix is a 2 dimensional array or size $|V|$ by $|V|$. The elements of the array contain the edge weights. Here is the Adjacency Matrix for the graph above:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	6	0	9	0	0	0	0	0	0	0	0
2	6	0	4	8	7	10	0	0	0	0	0	0
3	0	4	0	0	0	2	3	0	0	0	0	0
4	9	8	0	0	5	0	0	9	0	0	0	0
5	0	7	0	5	0	7	0	12	13	0	0	0
6	0	10	2	0	7	0	11	0	6	8	0	0
7	0	0	3	0	0	11	0	0	0	1	0	0
8	0	0	0	9	12	0	0	0	2	0	1	0
9	0	0	0	0	13	6	0	2	0	5	3	15
10	0	0	0	0	0	8	1	0	5	0	0	6
11	0	0	0	0	0	0	0	1	3	0	0	2
12	0	0	0	0	0	0	0	0	15	6	2	0

The adjacency matrix is best suited to dense graphs, but some algorithms require this structure for any graph.

This structure implies at least an $O(|V|^2)$ time complexity for any algorithm that employs it.

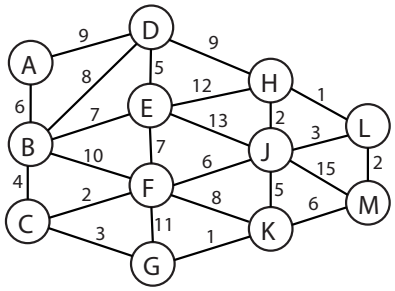
The cells of the array contain weights in a weighted graph.

12.3 Single Source Shortest Paths Problem

We often need to compute minimal paths from a distinguished vertex to all others. This is called the **Single Source Shortest Path Problem**.

The graph could be directed or not. The best algorithm *for an unweighted graph* is **Breadth First Search**.

Consider the shortest paths from source vertex s = A.



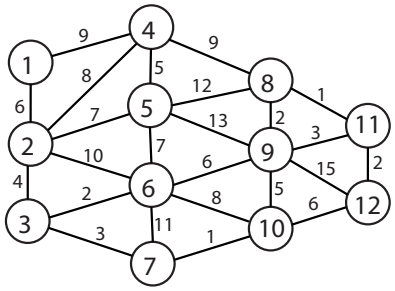
Here is the BFS algorithm, with start vertex s:

```
.....
    Queue que = new Queue()
    for each vertex u ∈ G.V − {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    que.enque(s)
    while(que.notEmpty())
        u = que.dequeue()
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d = v.d + 1
                v.π = u
                que.enqueue(v)
        u.color = BLACK
```

Each vertex v has class elements $v.color \in \{WHITE, GRAY, BLACK\}$, $v.d$ that records the shortest distance from s, and $v.\pi$ that records the predecessor vertex on the path from s. WHITE implies that the vertex has not been visited, GRAY implies that it is on the queue, and BLACK implies that it has been assigned a distance from s.
BFS for the undirected, unweighted graph above with start vertex A reveals the following:
Distance 1 from A: B, D
Distance 2 from A: C, E, F, H
Distance 3 from A: G, J, K, L
Distance 4 from A: M
The running time of BFS is $O(|E| + |V|)$ if an adjacency list structure is used. The algorithm visits every vertex once and considers every edge twice (once from each end).

12.4 Dijkstra's Single Source Shortest Path Algorithm

For weighted, directed or undirected graphs, Dijkstra's algorithm is used (unless there are negative edge weights). Again, the algorithm operates from a distinguished start vertex and computes shortest paths to all other vertices. For a sparse graph the adjacency matrix is used.
See the following weighted undirected graph:



The algorithm maintains a set of "known" vertices, S, for which the shortest path has already been computed. One vertex is added to S on each iteration. Once a vertex is added to the S, the shortest path to that vertex will not change in any subsequent iteration. The algorithm is "greedy" in this sense. At each iteration, the algorithm adds the vertex to S that is a neighbor of a vertex in S, and has the current shortest distance from the start vertex among all vertices not currently in S.

12.4.1 Dijkstra's Algorithm Using Arrays

The algorithm can be written using arrays. We will consider that approach first.
An `int` array `d` is used. `d[i]` contains the current best distance from the start vertex `s` to vertex `i`.
A boolean array `S` is used. `S[i]` is true if vertex `i` is a member of the set S.
Finally, `int` array `p` is used. `p[i]` records the predecessor vertex along the shortest path from the start vertex to vertex `i`.

```
 // class variables
 // results are put in d[] and p[]
int d[];     // d[i] = current best distance to vtx i
int p[];     // p[i] = index of prev' vtx on best path to i

void Dijkstra(ArrayList[] al, ArrayList[] aw, int startV){
         //   al = adjacency list of ints = nbrs
         //   aw = adjacency list of ints = edge wts
         //   startV is the start vertex
   int M = al.length();          // number of vertices
         //   vertices are numbered 1,2,3, ...,M
   boolean S[] = new boolean[M+1]; // the known set
   for(int i=1;i<=M;i++) {
      S[i] = false;
      d[i] = Integer.MAX_VALUE;
      p[i] = -1;
   }
   S[startV] = true;
   d[startV] = 0;         // distance to startV is 0
   p[startV] = -1;        // previous of startV = -1

   int nmbrOfNbrs = al[startV].size(); // nbrs of startV
   for(int i=0;i<nmbrOfNbrs;i++) {  //
      int nbr = al[startV].get(i); // get next nbr of startV
```

```
      d[nbr] = aw[startV].get(i);  // update dist' from startV
      p[nbr] = startV;             // update previous of nbr
   }

int newV;
   for(int i=1;i<M;i++) {  // find M-1 routes from startVertex
      int minDist = Integer.MAX_VALUE;
      newV = -1;
      for(int k=1;k<=M;k++)  // find vtx not in S w/ smallest wt
         if(!S[k] && d[k]<minDist) {
            minDist = d[k];
            newV = k;
      }
      S[newV] = true;          // add newV to the known set
      nmbrOfNbrs = al[newV].size();
      for(int j=0;j<nmbrOfNbrs;j++) {// update nbrs of newV
         nbr = al[newV].get(j);
         int w = aw[newV].get(j); // dist' from newV to nbr
         if(d[nbr]>(d[newV] + w)) {
            d[nbr] = d[newV] + w;
            p[nbr] = newV;
         }
      }
   }
}  // end of Dijkstra
```

Initially												
v	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$
1	F	∞	ϕ									
2	F	∞	ϕ									
3	F	∞	ϕ									
4	F	∞	ϕ									
5	F	∞	ϕ									
6	F	∞	ϕ									
7	F	∞	ϕ									
8	F	∞	ϕ									
9	F	∞	ϕ									
10	F	∞	ϕ									
11	F	∞	ϕ									
12	F	∞	ϕ									

After adding vertex 1 to S												
v	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$
1	F	∞	ϕ	T	0	-						
2	F	∞	ϕ	F	6	1						
3	F	∞	ϕ	F	∞	ϕ						
4	F	∞	ϕ	F	9	1						
5	F	∞	ϕ	F	∞	ϕ						
6	F	∞	ϕ	F	∞	ϕ						
7	F	∞	ϕ	F	∞	ϕ						
8	F	∞	ϕ	F	∞	ϕ						
9	F	∞	ϕ	F	∞	ϕ						
10	F	∞	ϕ	F	∞	ϕ						
11	F	∞	ϕ	F	∞	ϕ						
12	F	∞	ϕ	F	∞	ϕ						

After adding vertex 2 to S												
v	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$
1	F	∞	ϕ	T	0	-	T	0	-			
2	F	∞	ϕ	F	6	1	T	6	1			
3	F	∞	ϕ	F	∞	ϕ	F	10	2			
4	F	∞	ϕ	F	9	1	F	9	1			
5	F	∞	ϕ	F	∞	ϕ	F	13	2			
6	F	∞	ϕ	F	∞	ϕ	F	16	2			
7	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			
8	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			
9	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			
10	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			
11	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			
12	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ			

After adding vertex 4 to S												
v	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$	$S[]$	$d[]$	$p[]$
1	F	∞	ϕ	T	0	-	T	0	-	T	0	-
2	F	∞	ϕ	F	6	1	T	6	1	T	6	1
3	F	∞	ϕ	F	∞	ϕ	F	10	2	F	10	2
4	F	∞	ϕ	F	9	1	F	9	1	T	9	1
5	F	∞	ϕ	F	∞	ϕ	F	13	2	F	13	2
6	F	∞	ϕ	F	∞	ϕ	F	16	2	F	16	2
7	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
8	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	18	4
9	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
10	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
11	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
12	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ

The final result is below:

v	$S[]$	$d[]$	$p[]$	Shortest Path
1	T	0	1	1
2	T	6	1	1 2
3	T	10	2	1 2 3
4	T	9	1	1 4
5	T	13	2	1 2 5
6	T	12	3	1 2 3 6
7	T	13	3	1 2 3 7
8	T	18	4	1 4 8
9	T	18	6	1 2 3 6 9
10	T	14	7	1 2 3 7 10
11	T	19	8	1 4 8 11
12	T	20	10	1 2 3 7 10 12

The $d[]$ values give the shortest path costs.

The shortest paths are discovered by backtracking using the $p[]$ values.

12.4.2 Dijkstra's Algorithm Using a Priority Queue

Here is Dijkstra's algorithm using a priority queue for d[i] values and a set for S:

```
.....
    Dijkstra(G)
    S = φ
    PriorityQueue pque = new PriorityQueue(V)
    While (pque.notEmpty())
        u = pque.deleteMin()
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v)
```

deleteMin() returns the vertex ID of the vertex in the priority queue that has the current smallest distance value.

For non-neighbors of s, a distance of ∞ is initially inserted. These distance values change as edges are relaxed.

For the above weighted graph with start vertex $s = 1$, once vertex 1 has been added to S, the distances would be 1.d = 0, 2.d = 6, 4.d = 9. $v.d = \infty$ for all the other vertices.

On each iteration the vertex u is added to the known set. It is the nearest unknown vertex to s at that time, based on the $vertex.d$ values.

The function RELAX(u, v) is as follows:

.....
RELAX(u, v)
if $v.d > u.d + w(u, v)$
$v.d = u.d + w(u, v)$
$v.\pi = u$

The function $w(u, v)$ returns the weight of the edge (u, v). Do not assume that w implies an adjacency matrix structure. It is usually implemented by iterating along the adjacency list for vertex u.

Recall that when RELAX is called the algorithm has just added u to the known set S.

The **for** loop extracts all the unknown neighbors of u. At this time, the distances recorded for these vertices are via paths that only include known vertices, but excluding u (since we have only just added u).

For each v in this subset the algorithm checks to see if the current path to v can be improved by going via u.

> Note that the RELAX function requires a method, `decreaseKey()` in the Priority Queue class to modify the key of an entry. Java's Priority Queue class does not have this capability. It is, however, simple to avoid the need for `decreaseKey()` by re-enqueueing nodes as their distances are decreased. The algorithm terminates not when the priority queue is empty, but when all nodes have been added to the `Known Set, S`.

Initializing the priority queue takes $O(|V|)$ time by a call to buildHeap().

There are $|V|$ iterations. In each iteration we:

1. call x = deleteMin() on the priority queue
2. relax all the neighbors of x

If a binary heap is used as the priority queue, deleteMin() takes $O(log|V|)$ time.

If an adjacency list is used to store the edges of G, the algorithm can step through the list of neighbors of x during relaxation. For each neighbor y of x, a relaxation that changes $y.d$ will require a call to decreaseKey(y). Each decreaseKey() takes $O(log|V|)$. For each new vertex x added to S, relaxation takes $O(x.degree\ log|V|)$ where $x.degree$ is the number of neighbors of x.

Overall, the algorithm takes:

$$\sum_{v \in G} (1 + v.degree) log|V|$$

which is

$$O(|V| + |E|) log|V|$$

For a dense graph, $|E| = O(|V|^2$ and the running time is $O(|V|^2 log|V|)$.

For sparse graphs, $|E| = O(|V|)$ and the running time is $O(|V| log|V|)$.

As we saw earlier, an alternative implementation of the algorithm makes use of an array instead of a priority queue to store the $v.d$ values.

The equivalent of deleteMin() in an array requires a linear search to find the minimum value, taking $O(|V|)$ time. If an adjacency list is used, finding each neighbor of a vertex takes constant time. Relaxing vertex v also takes constant time (we simply adjust the value of $v.d$ in the distance array).

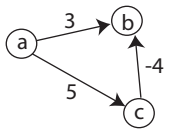
Overall the algorithm takes

$$\sum_{v \in G} (|V| + degree(v)) = O(|V|^2 + |E|)$$

For a sparse graph or a dense graph the running time is $O(|V|^2)$. For a dense graph, this algorithm is optimal since it runs in a time linear in the number of edges (it beats the implementation above using a binary heap).

A Fibonacci Heap provides amortized constant time for decreaseKey(), giving $O(|E| + |V|log|V|)$ runtime for Dijkstra's algorithm for sparse graphs.

Consider the following directed, weighted graph:



Beginning with the start vertex a, Dijkstra's algorithm first adds a and then b to S. After b is added to S, its shortest path distance from a is fixed at 3. But after c is added, the negative edge (c, b) would provide an opportunity to update $a's$ distance to 1. This shows that Dijkstra's greedy algorithm should never be used with a graph containing negative edges.

12.5 Bellman-Ford Single Source Shortest Path Algorithm

The **Bellman Ford Algorithm** is used when there are negative edge weights. It reports the presence of a negative cost cycle, the presence of which makes it impossible to compute shortest paths.

12.6 Floyd-Warshall All Pairs Shortest Path Algorithm

The algorithm computes the shortest path between every pair of vertices in $\theta(|V|^3)$ time. It computes a $|V| \times |V|$ array d of shortest path distances and a $|V| \times |V|$ array $path$ containing the predecessor vertices on the shortest paths.

```
.....
    FloydWarshall(W)
    n = W.rows
//Initialize d and path
    for i = 0 to n − 1
        for j = 0 to n − 1
            d_{i,j} = W_{i,j} // W_{i,j} is assumed to be 0
```

```
        path_{i,j} = φ
for k = 0 to n − 1
    for i = 0 to n − 1
        for j = 0 to n − 1
            if d_{i,k} + d_{k,j} < d_{i,j}
                d_{i,j} = d_{i,k} + d_{k,j}
                path_{i,j} = k
```

Although the algorithm takes $O(|V|^3)$ time, the loops are tight and, for small dense graphs, Floyd Warshall is often used in place of Dijkstra's single source shortest path algorithm.

The algorithm proceeds by considering each edge (i, j) and checking if its shortest path might be improved by going via an intermediate vertex k: $d_{i,k} + d_{k,j} < d_{i,j}$. This process is repeated for all $k = 0, 1, 2, \cdots, |V| - 1$.

The largest value in the array d gives the **Diameter** of the graph. This is the largest of the shortest distances.

12.7 Prim's Minimal Spanning Tree Algorithm

A minimal spanning tree is defined for an undirected, weighted graph $G(V, E)$. It is a set of $|V| - 1$ edges from E that induces a tree on the vertices in V.

The total weight of all the tree edges is minimal. When the edge weights of G are not unique there may be several equally good minimal spanning trees.

Prim's algorithm is very similar to Dijkstra's single source shortest path algorithm. The relaxation method is different. Here is the algorithm with start vertex s:

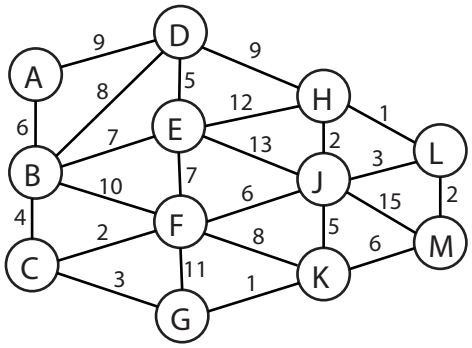
```
.....................
    Prim(G,s)
    Set S = new Set()
    S = φ
    PriorityQueue pque = new PriorityQueue(V)
    for each u ∈ G.V
        u.d = ∞
        u.π = NIL
    r.d = 0
    while pque.notEmpty()
        u = pque.deleteMin()
        S = S ∪ u
        for each v ∈ G.Adj[u]
            if (v ∉ S) and w(u, v) < v.d
                v.π = u
                v.d = w(u, v)
```

In the relaxation sequence (the last 4 lines) only vertices that are neighbors of u and not members of S are relaxed. Note that all vertices are either in S or in the priority queue. If the priority queue has a "contains" function then the set S can be eliminated.

Once a vertex u is selected by a call to deleteMin(), its distance $u.d$ will never change again. In that sense the algorithm is greedy. Selected vertices are members of the growing spanning tree.

Careful inspection will show that when vertex u is added to S, the distance $u.d$ is the shortest distance between any vertex not in S and any member of S at that time.

Consider the following undirected graph:



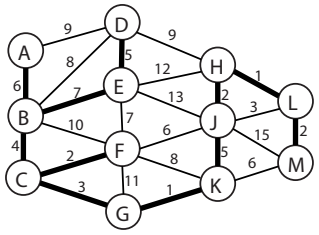
Here is a dry-run of the first three iterations of Prim's algorithm on the undirected graph above with $s = A$ as the start vertex.

v	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$
A	F	∞	ϕ	T	0	-	T	0	-	T	0	-
B	F	∞	ϕ	F	6	A	T	6	A	T	6	A
C	F	∞	ϕ	F	∞	ϕ	F	4	B	T	4	B
D	F	∞	ϕ	F	9	A	F	8	A	F	8	B
E	F	∞	ϕ	F	∞	ϕ	F	7	B	F	7	B
F	F	∞	ϕ	F	∞	ϕ	F	10	B	F	2	C
G	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	3	C
H	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
J	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
K	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
L	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
M	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
	S empty			A added to S			B added to S			C added to S		

The final result is below:

v	S	$v.d$	$v.\pi$	edges
A	T	0	A	(A,B)
B	T	6	A	(B,C)
C	T	4	B	(B,E)
D	T	5	E	(C,F)
E	T	7	B	(C,G)
F	T	2	C	(D,E)
G	T	3	C	(G,K)
H	T	2	J	(H,J)
J	T	5	K	(H,L)
K	T	1	G	(J,K)
L	T	1	H	(L,M)
M	T	2	L	

Here is the resulting minimal spanning tree:



The running time of Prim's algorithm is identical to that of Dijkstra. An adjacency list structure should be used. Then the running time is $O(|V|^2)$ if an array is used to store the $v.d$ distance values. This is optimal for dense graphs.

If a binary heap is used to store the $v.d$ values, then the running time is $O(|E|log|V|)$, which is a better result for sparse graphs than when an array is used.

If a Fibonacci Heap is used for storing $v.d$ values, the running time of Prim's algorithm is $O(|E|+|V|log|V|)$, which is optimal for sparse graphs.