University of Miami
ACM Codebook

October 28, 2014

Contents

1 Mathematics **3**

1.1 Common Sums . . . 3
1.2 Factorials . . . 3
1.3 Derangements . . . 5
1.4 Combinations . . . 5
1.5 Euclidean Algorithm . . . 6
1.6 Extended Euclidean Algorithm . . . 6
1.7 Catalan Numbers . . . 7
1.8 Sieve of Eratosthenes . . . 7
1.9 Efficient Modular Exponentiation . . . 8
1.10 Prime Factorization . . . 8
1.11 Primality Testing . . . 9
1.12 Totient Function . . . 9
1.13 Chinese Remainder Theorem . . . 10
1.14 Perfect Power Test . . . 11
1.15 Fraction-Decimal Converter . . . 12
1.16 Fast Pythagorean Triplet Generator . . . 13
1.17 Matrix Determinant . . . 14
1.18 Inverse of a Matrix . . . 14

2 Graph Algorithms **16**

2.1 Breadth-First Search . . . 16
2.2 Depth-First Search . . . 18
2.3 Dijkstra's Algorithm . . . 20
2.4 Floyd-Warshall Algorithm . . . 21
2.5 Prim's Algorithm . . . 21
2.6 Bellman-Ford Algorithm . . . 22
2.7 Bridge Detection . . . 23
2.8 Articulation Point Detection . . . 24
2.9 Edmond's Karp Algorithm (Maximum-Flow) . . . 25
2.10 Minimum-Cost Maximum-Flow Algorithm . . . 26

3 Geometry **28**

3.1 Line Intersection . . . 28
3.2 Point/Vector Class . . . 28
3.3 Circle Class . . . 29
3.4 Minimum Enclosing Circle Algorithm . . . 30
3.5 Andrew's Algorithm (Convex Hull) . . . 31
3.6 Longitude and Latitude . . . 32

4 Dynamic Programming **34**

4.1 Unbounded Knapsack . . . 34
4.2 Bounded (0/1) Knapsack . . . 34
4.3 Longest Increasing Subsequence . . . 35
4.4 Longest Common Subsequence . . . 36
4.5 Longest Common Substring . . . 37
4.6 Maximum Contiguous Sum . . . 37
4.7 Maximum Rectangular Sum . . . 38
4.8 Levenshtein Distance (Edit Distance) . . . 38

5 Tree Data Structures **39**

5.1 Fenwick Tree . . . 39
5.2 Trie . . . 41

6 Miscellaneous **42**

6.1 Kirchoff's Matrix . . . 42
6.2 Josephus Problem . . . 42
6.3 Poker Class . . . 42
6.4 Decimal to Roman Numeral Converter . . . 44
6.5 Expression Parsing . . . 46
6.6 String Matching with KMP . . . 48

7 Language Specific References **50**

7.1 C++ Startup Code . . . 50
7.2 Java Start-Up Code . . . 50
7.3 Java BigInteger Reference . . . 51

1 Mathematics

1.1 Common Sums

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{n} 2k-1 = n^2$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} (2k-1)^2 = \frac{n(4n^2-1)}{3}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{k=1}^{n} (2k-1)^3 = n^2(2n^2-1)$$

$$\sum_{k=1}^{n} k^4 = \frac{n(n+1)(n+2)(n+3)}{30}$$

$$\sum_{k=1}^{n} k^5 = \frac{n^2(n+1)^2(2n^2+2n-1)^2}{12}$$

$$\sum_{k=1}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

$$\sum_{k=1}^{n} k(k+1)(k+2) = \frac{n(n+1)(n+2)(n+3)}{4}$$

$$\sum_{k=1}^{n} k(k+1)(k+2)(k+3) = \frac{n(n+1)(n+2)(n+3)(n+4)}{5}$$

1.2 Factorials

Most of the time you'll have to compute factorials more than once in a program. Hence in order to save time, it is beneficial to compute all of them at once and store them in an array for future use.

Listing 1.1: Calculating a factorial (C++)

```
long long* calcFactorial(int n)
{
  long long * f = new long long[n+1];
  f[0] = 1;
  for (int i=1; i<=n; i++)
    f[i] = i*f[i-1];

  return f;
}
```

Note the usage of the **long long** type, a 64-bit type. However, it can only accurately compute factorials until 20!. After that it will overflow the **long long** type and be horribly wrong. In Java you can use the **long** type to achieve similar results, or you can cheat a bit and use the built-in **BigInteger** type like so.

Listing 1.2: Calculating a factorial with the BigInteger class (Java)

```
import java.math.BigInteger;
BigInteger[] calcFactorial(int n)
{
    BigInteger[] f = new BigInteger[n+1];
    f[0] = BigInteger.ONE;
    for (int i=1; i<=n; i++)
        f[i] = new BigInteger(i+"").multiply(f[i-1]);

    return f;
}
```

Problem : How many digits are in the factorial of n?
Solution : Simply add up the logs of the numbers less than or equal to n starting at 1.

Listing 1.3: Counting the digits in the factorial of a number (Java)

```
double countDigits(long n)
{
    double sum;
    if(n==0) return 1.0;
    else
    {
        sum = 0.0;
        for(long i = 1; i<=n; i++)
            sum+=Math.log10(i);
    }
    return Math.floor(sum)+1;
}
```

Problem : How many trailing zeroes does the number $n!$ have?

Listing 1.4: Counting the number of trailing zeroes in the factorial of a number (Java)

```
1 double countZeroes(long n)
2 {
3     int count = 0;
4     while(n!=0)
5     {
6         count+=n/5;
7         n/=5;
8     }
9     return count;
10 }
```

1.3 Derangements

Number of permutations with no fixed points.

$!0 = 1$ and $!1 = 1$

$!n = (n-1)(!(n-1)+!(n-2))$ for $n \geq 2$

1.4 Combinations

Written as ${}_nC_r$, this represents the number of ways of selecting r objects from n where order is irrelevant.

$$ {}_nC_r = \binom{n}{r} = \frac{n!}{(n-r)!r!} \qquad \binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \qquad \binom{n}{r} = \binom{n}{n-r} $$

$$ \binom{n}{r} = \binom{n}{r-1}\frac{n-r+1}{r} \qquad \binom{n}{r} = \binom{n-1}{r}\frac{n}{n-r} \qquad \binom{n}{r} = \binom{n-1}{r-1}\frac{n}{r} $$

Listing 1.5: Calculating combinations using the BigInteger class (Java)

```
1 BigInteger calcCombinations (int n, int k)
2 {
3    k = Math.min(k, n-k);
4    BigInteger ret = BigInteger.ONE ;
5    for ( int i=n-k+1; i<=n; i++)
6       ret = ret.multiply(BigInteger.valueOf(i));
7    for ( int i=2; i<=k; i++)
8       ret = ret.divide(BigInteger.valueOf(i));
9    return ret ;
10 }
```

However, it is safer to use the second formula in a tabular fashion since it only involves addition. It runs in $O(n^2)$ but most of the time n is rather small, so it won't affect the overall performance.

Listing 1.6: Calculating combinations in a tabular fashion (C++)

```
1 const int MAX_S = 52;
2 long long C[MAX_S][MAX_S];
```

```
 3
 4 void calcCombinations()
 5 {
 6    for (int i=0; i<MAX_S; i++)
 7    {
 8       C[i][0] = 1;
 9       for (int j=1; j<=i; j++)
10          C[i][j] = C[i-1][j-1] + C[i-1][j];
11    }
12 }
```

1.5 Euclidean Algorithm

This is an algorithm used to find the greatest common divisor of two numbers. Runs in $O(max(\log A, \log B))$ time.

Listing 1.7: Euclidean Algorithm (Java)

```
1 int gcd(int a, int b)
2 { return (b==0 ? a : gcd(b, a%b)); }
```

Note: The LCM (least common multiple) is

$$\frac{ab}{gcd(a,b)}$$

1.6 Extended Euclidean Algorithm

Given numbers a and b, this number returns the $gcd(a,b)$ and it will set the values of x and y such that $ax + by = gcd(a,b)$ is satisfied.

Uses : finding the inverse of a number under a certain modulo.

Note : When a and b are relatively prime, i.e. $gcd(a,b) = 1$, then

$$x = a^{-1} \pmod{b}$$

$$y = b^{-1} \pmod{a}$$

Listing 1.8: Extended Euclidean Algorithm (C++)

```
1 int x=0, y=0;
2 int ext_gcd(int a, int b)
3 {
4   if (b == 0)
5   {
6     x = 1; y = 0;
7     return a;
8   }
9
10  int ret = ext_gcd(b, a%b);
11  int t = x; x = y; y = t-a/b*y;
12
```

```
13   return ret;
14 }
```

1.7 Catalan Numbers

The Catalan numbers are a sequence of numbers that occur in many counting problems, particularly those that involve recursively defined objects.

The $n^(th)$ Catalan number is given by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n}\frac{n+k}{k} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\sum_{i=0}^{n}\binom{n}{i}^2$$

Asymptotically, the Catalan numbers grow as

$$C_n \sim \frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}}$$

Listing 1.9: Catalan Numbers (C++)

```
1 //1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, ...
2 int Cat[51];
3 fill(Cat, Cat + 51, 0);
4 Cat[0] = 1;
5 for(int i = 1; i < 51; i++)
6   for(int j = 0; j < i; j++)
7     Cat[i] += Cat[j] * Cat[ i - j - 1];
```

1.8 Sieve of Eratosthenes

An efficient way to find all primes that are less than a certain number. It runs in $O(\sqrt{n}\log n \log\log n)$ time.

Listing 1.10: Sieve of Eratosthenes (Java)

```
1 int MAX_S = 1000000;
2 boolean p[] = new boolean[MAX_S];
3 void primeSieve(boolean p[])
4 {
5    Arrays.fill(p, true);
6    p[0] = p[1] = false;
7    for (int i=2; i*i<p.length; i++)
8       if (p[i])
9          for (int j=i*i; j<p.length; j+=i)
10            p[j] = false;
11 }
```

1.9 Efficient Modular Exponentiation

Computes $a^b \pmod{c}$ quickly using the method of repeated squaring

Careful : 0^0 will return 1. It's up to you to catch this!
Complexity : $O(\log b)$

Listing 1.11: Efficient Modular Exponentiation (Java)

```
long modPow(long a, long b, long c) //a^b (mod c)
{
    long p = 1;
    a %= c; //just in case a is already large
    while (b > 0)
    {
        if (b%2==1 p = (p*a)%c;
        a = (a*a)%c;  //square, careful not to overflow
        b >>= 1;  //move on to next bit
    }
    return p;
}
```

1.10 Prime Factorization

Given an **int**, this function returns a vector composed of its prime factors (repetition included) in ascending order.

Complexity : $O(\sqrt{n})$

Listing 1.12: Prime Factorization (C++)

```
vector<int> prime_factorization(int n)
{
   vector<int> factors;
   while (n%2 == 0) //get rid of all even numbers
   {
      factors.push_back(2);
      n >>= 1;
   }
   for (int i=3; i*i<=n; i+=2)
      while (n%i == 0)
      {
         factors.push_back(i);
         n /= i;
      }
   if (n != 1) factors.push_back(n); //catch prime
   return factors;
}
```

1.11 Primality Testing

Complexity : $O(\sqrt{n})$

Listing 1.13: Primality Testing (C++)

```
bool isPrime(int n)
{
   if (n==2 || n==3) return true;
   if (n%2==0 || n%3==0 || n<2) return false;
   int stop = (int)sqrt(1.*n) + 2;
   for (int i=6; i<=stop; i+=6)
      if (n%(i+1) == 0 || n%(i-1) == 0) return false
   return true;
}
```

1.12 Totient Function

Denoted as $\phi(n)$, is the number of positive integers that are less than or equal to n. Mathematically, it it

$$\prod_{p_i|n} n\left(1-\frac{1}{p_i}\right)$$

for all distinct primes p that divide n. Also, there is a slight optimization that takes advantage of the fact that other than 2 and 3, all primes are either 1 (mod 6) or 5 (mod 6).

Listing 1.14: Totient Function (C++)

```
int phi(int n)
{
   int ret = n;
   if (n%2==0)  //remove all 2s
   {
      ret /= 2;
      do { n /= 2; } while (n%2==0);
   }
   if (n%3==0) //remove all 3s
   {
      ret = ret/3*2;
      do { n /= 3; } while (n%3==0);
   }
   int stop = (int)sqrt(1.*n)+2;
   for (int i=6; i<=stop; i+=6) //all other primes are 6m+1 or 6m-1
   {
      if (n%(i-1)==0) //6m-1
      {
         ret = ret/(i-1)*(i-2);
         while ( n%(i-1) == 0) n /= i-1; //removed repeats
         stop = (int)sqrt(1.*n)+2; //update stop
      }
      if (n%(i+1)==0) //ditto for 6m+1
```

```
24        {
25           ret = ret/(i+1)*i;
26           while ( n%(i+1) == 0) n /= i+1;
27           stop = (int)sqrt(1.*n)+2;
28        }
29     }
30     if (n!=1) //n might be a prime, if so include it
31        ret = ret/n*(n-1);
32     return ret;
33  }
```

1.13 Chinese Remainder Theorem

Given two arrays a, n where

$$x = a[0] \pmod{n[0]}$$
$$x = a[1] \pmod{n[1]}$$
$$.$$
$$.$$
$$.$$

this returns an array z, the solution of this system of equations where $x = z[0] \pmod{z[1]}$.

Listing 1.15: Chinese Remainder Theorem (Java)

```
1  public static final BigInteger ZERO = BigInteger.ZERO;
2  public static final BigInteger ONE = new BigInteger("1");
3  public static final BigInteger nONE = new BigInteger("-1");
4
5    public static class triplet
6    {
7          BigInteger d, x, y;
8           public triplet(BigInteger a, BigInteger b, BigInteger c)
9           {
10             d = a ;
11             x = b;
12             y = c;
13          }
14   }
15
16   static triplet Euclid(BigInteger a, BigInteger b)
17   {
18          if (b.compareTo(ZERO) == 0)
19             return new triplet(a, ONE, ZERO);
20        triplet tp = Euclid(b, a.mod(b));
21          return new triplet( tp.d, tp.y                    tp.x.subtract((a
                .divide(b)).multiply(tp.y)));
22   }
23
24
25   final static int SZ= 1000;
26   static int T;
```

```
27  public static final BigInteger ZERO = BigInteger.ZERO;
28  public static final BigInteger ONE = new BigInteger("1");
29     public static final BigInteger nONE = new BigInteger("-1");
30  public static class triplet
31  {
32        BigInteger d, x, y;
33         public triplet(BigInteger a, BigInteger b, BigInteger c)
34         {
35            d = a ;
36            x = b;
37            y = c;
38         }
39  }
40
41  static triplet Euclid(BigInteger a, BigInteger b)
42  {
43         if (b.compareTo(ZERO) == 0)
44            return new triplet(a, ONE, ZERO);
45        triplet tp = Euclid(b, a.mod(b));
46         return new triplet( tp.d, tp.y,
47                     tp.x.subtract((a.divide(b)).multiply(tp.y)));
48  }
49
50  public static BigInteger inverseMod(BigInteger a,
51                                                      BigInteger n)
52  {
53        triplet t = Euclid(a, n);
54        BigInteger m = t.x.mod(n);
55         return m.add(n).mod(n);
56  }
57
58  public static BigInteger[] CRT(BigInteger[] a, BigInteger[] n)
59  {
60        BigInteger N = ONE;
61         for(int i = 0 ; i < T; i++)
62            N = N.multiply(n[i]);
63        BigInteger Z = ZERO;
64        for(int i = 0 ; i < T; i++)
65        {
66            BigInteger ni = N.divide(n[i]);
67            BigInteger ni_1 = inverseMod(ni, n[i]);
68            Z = Z.add(a[i].multiply(ni).multiply(ni_1)).mod(N);
69        }
70         return new BigInteger[]{Z, N};
71  }
```

1.14 Perfect Power Test

Efficient method of testing if a number can be written in the form a^b.

Listing 1.16: Perfect Power Test (Java)

```
public static final BigInteger ZERO = BigInteger.ZERO;
public static final BigInteger ONE = new BigInteger("1");
public static final BigInteger nONE = new BigInteger("-1");
public static final BigInteger TWO = new BigInteger("2");
  public static BigInteger[] test(BigInteger n)
  {
          int k = n.bitLength();
          for(int e = 2; e <= k + 1; e++)
          {
               int b1 = (k-1)/e;
               int b2 = (k+e-1)/e;
               BigInteger u  = TWO.pow(b1);
               BigInteger v  = TWO.pow(b2);
               do
               {
                    BigInteger w = u.add(v).divide(TWO);
                    BigInteger z = w.pow(e);
                    if(z.compareTo(n) == 0)
                        return new BigInteger[]{w,
                                                new BigInteger(e+"")};
                    if(z.compareTo(n) < 0 )
                        u = w.add(ONE);
                    else
                        v = w;
                }
                while(u.compareTo(v) < 0);
            }
            return new BigInteger[0];
    }
```

1.15 Fraction-Decimal Converter

When dealing with fractions, its sometimes required to print them to a certain precision. You definitely dont want to convert it to a double and print that out since there might be errors after 10 or so digits. Another way to approach this problem is to repeatedly multiply and mod while maintaining the number in fractional form. Note, this method is similar to how people divide by hand. This runs in $O(k)$ where k is the number of digits of precision.

Listing 1.17: Fraction-Decimal Converter (C++)

```
string D2S(int nume, int deno, int precision)
{
   stringstream ss;
   ss << nume/deno;
   nume %= deno;
   if (precision != 0) ss << ".";
   while (precision-- > 0)
   {
      nume *= 10;
      ss << nume/deno;
      nume %= deno;
   }
```

```
13     return ss.str();
14 }
```

1.16 Fast Pythagorean Triplet Generator

This algorithm uses a BFS on the initial triple $(3, 4, 5)$ to generate all other triplets. Given an integer **MAX_N**, this function returns a vector of all Pythagorean triplets with sides no bigger than **MAX_N**.

Complexity : $O(k)$ - This algorithm is output sensitive (k is the number of answers).
Note : All triplets returned are unique. In each triplet (a, b, c) a is not necessarily less than or equal to b. The list includes non-primitive triplets. (This can easily be removed if not needed)

Listing 1.18: Fast Pythagorean Triplet Generator (C++)

```
1 struct Triplet
2 {
3    int x, y, z;
4    Triplet() {}
5    Triplet(int a, int b, int c) { x=a; y=b; z=c; }
6 };
7 vector<Triplet> generate_triplets(int MAX_N)
8 {
9    set<Triplet> s;
10   vector<Triplet> ans;
11   queue<Triplet> q;
12   Triplet st(3,4,5); //initial value
13   q.push(st);
14   while (!q.empty()) //BFS
15   {
16      Triplet C = q.front();
17      q.pop();
18      int a = C.x, b = C.y, c = C.z;
19      ans.push_back(Triplet(a, b, c)); //Add primitive triplet
20      //Add all multiples of primitive triplets
21      //remove this if you only need primitive triplets
22      for (int i=2; c*i<=MAX_N; i++)
23         ans.push_back(Triplet(a*i, b*i, c*i));
24      if (2*a - 2*b + 3*c <= MAX_N)
25         q.push(Triplet(a-2*b+2*c,2*a-b+2*c,2*a-2*b+3*c));
26      if (2*a + 2*b + 3*c <= MAX_N)
27         q.push(Triplet(a+2*b+2*c,2*a+b+2*c,2*a+2*b + 3*c));
28      if (-2*a + 2*b + 3*c <= MAX_N)
29         q.push(Triplet(-a+2*b+2*c,-2*a+b+2*c,-2*a+ 2*b+3*c));
30   }
31   return ans;
32 }
```

1.17 Matrix Determinant

This uses partial Gaussian elimination to obtain an upper triangular matrix, then the product of the main diagonal is the determinant.
Assume that a Matrix class has already been created that contains:
int n, m; (the dimensions)
double mat[][]; (the entries)
has already been declared. Along with some basic constructors and helper methods. (i.e. **swapRow**)

Listing 1.19: Matrix Determinant (Java)

```
1 double determinant()
2 {
3    assert(n == m); //must be a square matrix
4    Matrix L = new Matrix(n, n);
5    for (int i=0; i<n; i++)
6       for (int j=0; j<n; j++)
7          L.mat[i][j] = mat[i][j];
8    double ret = 1.;
9    for (int i=0; i<n; i++)
10   {
11      if (abs(L.mat[i][i]) < 1e-9) //a zero along the pricipal diagonal
12      {
13         for (int j=i+1; j<n; j++)
14         if (abs(L.mat[j][i]) > 1e-9) //swap row j with row i
15         {
16            L.swapRow(j, i);
17            ret *= -1;
18               break;
19         }
20      }
21      if (abs(L.mat[i][i]) < 1e-9) return 0.;
22      for (int j=i+1; j<n; j++)
23         if (abs(L.mat[j][i]) > 1e-9)  //found non-zero element
24         {
25            double f = L.mat[j][i]/L.mat[i][i];
26            for (int jj=0; jj<n; jj++) //subtract scaled row
27            L.mat[j][jj] -= f*L.mat[i][jj];
28         }
29      ret *= L.mat[i][i];
30   }
31   return ret;
32 }
```

1.18 Inverse of a Matrix

This uses Gaussian elimination

Note : Doubles are not very precise.

Listing 1.20: Matrix Inverse (Java)

```
1 Matrix inverse() //gaussian elimination
2 {
3    Matrix L = new Matrix(n, n);
4    for (int i=0; i<n; i++)
5       for (int j=0; j<n; j++)
6          L.mat[i][j] = mat[i][j];
7    Matrix R = new Matrix(n, n);
8    for (int i=0; i<n; i++) R.mat[i][i] = 1;
9    for (int i=0; i<n; i++)
10   {
11      if (abs(L.mat[i][i]) < 1e-14) //a zero along the pricipal diagonal
12         for (int j=i+1; j<n; j++)
13            if (abs(L.mat[j][i]) > 1e-14)
14            {
15               L.swapRow(j, i);  //swap row j with row i
16          R.swapRow(j, i);
17          break;
18            }
19      if (abs(L.mat[i][i]-1.0) > 1e-14) //if it's not 1.0, scale down
           entire row
20      {
21         double f = L.mat[i][i];
22    for (int j=0; j<n; j++)
23    { L.mat[i][j] /= f; R.mat[i][j] /= f; }
24      }
25      for (int j=0; j<n; j++)
26         if (i!=j && abs(L.mat[j][i]) > 1e-14)  //found non-zero element
              underneath
27    {
28       double f = L.mat[j][i];
29       for (int jj=0; jj<n; jj++) //subtract scaled row
30       { L.mat[j][jj] -= f*L.mat[i][jj]; R.mat[j][jj] -= f*R.mat[i][jj];
            }
31    }
32   }
33   return R;
34 }
```

2 Graph Algorithms

2.1 Breadth-First Search

Breadth-First Search (BFS) is a single source shortest path algorithm that runs in linear time on the number of edges and vertices of a graph ($O(E+V)$ runtime). It is implemented using a queue (first-in-first-out data structure). Starting from a given source, the algorithm will process vertices in order by the minimum number of edges from the source. This allows the algorithm to break early if looking for a single vertex. However, the BFS will only allow you to find the shortest path between vertices in an unweighted graph.

Note : Using an adjacency matrix for a graph will cause BFS to run in $O(V^2)$ time since you will be searching through V vertices for each vertex in the graph. An easy way to think about this is that a $V \times V$ matrix contains V^2 edges. To achieve $O(E+V)$ time, use an adjacency list.
Examples : An implementation of a BFS is to find the minimum number of moves to get a knight in chess from one spot to another on a board. Here is an example of how to code this.

Listing 2.1: Knight BFS (C++)

```
1 //Directional arrays that specify the possible moves a Knight can do
2 int di[] = {1, 1, -1, -1, 2, 2, -2, -2};
3 int dj[] = {2, -2, 2, -2, 1, -1, 1, -1};
4
5 //State describes the position and number of moves to get to that position
6 struct state
7 {
8   int i, j, d;
9   state(int xx, int yy, int cc)
10  {
11    i = xx;
12    j = yy;
13    d = cc;
14  }
15  bool valid()
16  {
17    return i>=0 && i<8 && j>=0 && j<8;
18  }
19  state(){}
20 };
21
22 //Seen array to avoid repetitions of positions on a chess board
23 bool seen[8][8];
24
25 int bfs(int si, int sj, int ei, int ej)
26 {
```

```
27   for(int i=0;i<8;i++)
28     for(int j=0;j<8;j++)
29       seen[i][j]=false;
30
31   seen[si][sj]=true;
32   state start= state(si, sj,0);
33
34   queue<state> Q;
35   Q.push( start);
36      while(!Q.empty())
37   {
38     state curS = Q.front();
39     Q.pop();
40
41     int curi = curS.i;
42     int curj = curS.j;
43     int curd = curS.d;
44
45     if(curi==ei && curj==ej)
46       return curd;
47
48     for(int d =0; d<8; d++)
49     {
50       int newi = curi+di[d];
51       int newj = curj+dj[d];
52       int newd = curd+1;
53       state ns = state(newi, newj, newd);
54       if(ns.valid() && !seen[newi][newj])
55       {
56         seen[newi][newj] = true;
57         Q.push(ns);
58       }
59     }
60   }
61 }
```

A similar problem is finding the minimum number of steps required to go from a starting location to a target moving only up, down, left, or right.

In the graph, **X** means an obstacle and a ***** is the target.

Listing 2.2: Steps BFS (Java)

```
1 import java.util.*;
2 import java.io.*;
3
4 public class BreadthFirstSearch
5 {
6    //Testing code. Can be removed.
7    public static void main(String[] args){
8
9    }
10
```

```
11    static int bfs(char[][] g, int si, int sj)
12    {
13     int[] di = {1, -1, 0,  0};  //directional arrays
14     int[] dj = {0,  0, 1, -1};
15
16     Queue<State> q = new LinkedList<State>();
17
18     q.offer(new State(si, sj, 0));
19     g[si][sj] = 'X';   //don't forget to mark it as seen!
20
21     while (!q.isEmpty())
22     {
23     State c = q.poll();
24
25      for (int k=0; k<4; k++)
26      {
27       int ni = c.i+di[k];
28       int nj = c.j+dj[k];
29
30       if (ni>=0 && ni<g.length  && nj>=0 && nj<g[0].length && g[ni][nj]!='
            X')
31       {
32          if (g[ni][nj]=='*') return c.dist+1;
33
34          q.offer(new State(ni, nj, c.dist+1));
35          g[ni][nj] = 'X';
36       }
37      }
38     }
39
40     return -1;
41    }
42 }
43
44 class State  //helper class
45 {
46    int i, j, dist;
47    State(int a, int b, int c)
48    {
49     i = a;
50     j = b;
51     dist = c;
52    }
53 }
```

2.2 Depth-First Search

Depth-First Search (DFS) is an algorithm for traversing or searching through a graph. It runs in linear time ($O(E+V)$ runtime) if searching without repeating vertices. An exhaustive search with repetitions will push the algorithm to run in factorial time (since you will try all possible permutations with this method). The algorithm functions by starting at a root or beginning and searching down an entire path until it reaches a point with no more unseen connections. It then backtracks along the path until it finds an unseen path from

the current vertex. This search may not lead to a shortest path but can be used to determine if a node is reachable from another node and various other things. It can be done recursively using the sustem stack or using an explicit stack.

Note: Think of this as exploring a cave. You go as far down a cave as possible, planting torches to keep track of where you have been. When you hit a dead end you backtrack until you find another corridor you have not seen yet.
Examples : Finding the area of connected components in a grid can be done with a DFS. This is called the "Flood-Fill Algorithm". Here is a problem that returns a vector of areas of shapes in order.

Listing 2.3: Flood-Fill Algorithm (C++)

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

using namespace std;

bool seen[400][600];

int dfs(int i, int j)
{
  if( i>=400 || i<0 || j>=600 || j<0 || seen[i][j]) return 0;
  seen[i][j] = true;
  int A=1;
  A+=dfs(i-1, j);
  A+=dfs(i+1, j);
  A+=dfs(i, j-1);
  A+=dfs(i, j+1);
  return A;
}

vector <int> sortedAreas(vector <string> recs) {
  vector<int> areas;
  for(int i=0;i<recs.size();i++)
    {
      int i1, i2, j1, j2;
    stringstream ss(recs[i]);
      ss>>i1>>j1>>i2>>j2;

    for(int ii=i1;ii<=i2;ii++)
      for(int jj=j1;jj<=j2;jj++)
          seen[ii][jj]=true;
     }

     for(int i=0;i<400;i++)
      for(int j=0;j<600;j++)
           if(!seen[i][j])
              areas.push_back(dfs(i, j));

  sort(areas.begin(), areas.end());
```

```
42      return areas;
43  }
```

2.3 Dijkstra's Algorithm

Dijkstra's algorithm is a single source shortest path algorithm for a weighted graph (with non-negative weights). It runs in $O(E \log V)$ time using a priority queue. It is very similar to a BFS except it searches for the next closest vertex in terms of actual distance rather than number of edges away. Note that it will not work with negative weights!

Warning : If you are going to be calling the method several times, make sure you set all the elements in bool seen[] to false after every run, otherwise you might get incorrect results in all runs except the first.
Complexity : $O(E \log V)$

Listing 2.4: Dijkstra's Algorithm (C++)

```
1  struct Edge
2  {
3     int x, w;
4     Edge() {}
5     Edge(int a, int b) { x = a; w = b; }
6     bool operator < (const Edge& b) const
7     { return w > b.w; } //Since the priority queue sorts in descending
          order
8  };
9
10 #define MAX_NODES 10005
11 bool seen[MAX_NODES];
12 vector<Edge> graph[MAX_NODES];  //Adjacency List
13
14 void dijkstra(int start, int dist[], int n)  //From one node to all others
15 {
16    fill(dist, dist+n, -1); // Initial distances
17    priority_queue<Edge> pq;
18    pq.push(Edge(start,0)); //initial edge
19    while (!pq.empty())
20    {
21       Edge c = pq.top();
22       pq.pop();
23       if (seen[c.x]) continue;
24       seen[c.x] = true;  //mark as seen
25       dist[c.x] = c.w;   //remember the shortest path to c.x
26       for (int i=0; i<graph[c.x].size(); i++)
27          if ( !seen[graph[c.x][i].x] )  //if not seen yet, add it
28             pq.push( Edge( graph[c.x][i].x, c.w + graph[c.x][i].w ) );
29    }
30 }
31
32 NOTE FOR JAVA - to avoid generic creation
33 static ArrayList<Edge> edges[] = new ArrayList[10000];
```

2.4 Floyd-Warshall Algorithm

Computes the All-Pairs Shortest Path of a graph in $O(N^3)$. This algorithm also computes the shortest paths for a graph with negative weights. If there is a negative cycle, then there should be a path from a vertex to itself with negative weight.

Listing 2.5: Floyd-Warshall Algorithm (C++)

```
//Assume N is the number of vertices in our graph.
const int sentinel=1000000000;
int dist[N][N];
int next[N][N]; //next[i][j] is the next vertex from i to j.

void Floyd_Warshall()
{
   for(int i=0; i<N; i++)
      for(int j=0; j<N; j++)
         if(dist[i][j]!=sentinel)
            next[i][j]=j;
   for(int k=0; k<N; k++)
      for(int i=0; i<N; i++)
         if(dist[i][k]!=sentinel)
            for(int j=0; j<N; j++)
               if(dist[k][j]!=sentinel && dist[i][j]>dist[i][k]+dist[k][j
                  ])
               {
                  dist[i][j]=dist[i][k]+dist[k][j];
                  next[i][j]=next[i][k];
               }
}

vector <int> build_path(int start, int end)
{
   vector <int> path;
   path.push_back(start);
   int cur = start;
   while(next[cur][end]!=end)
   {
      cur=next[cur][end];
      path.push_back(cur);
   }
   path.push_back(end);
   return path;
}
```

2.5 Prim's Algorithm

Prim's algorithm is used to find the minimum spanning tree (MST) for a weighted undirected graph. It runs in $O(E \log V)$ time.

Note : It does not matter which vertex you start off from since all vertices will be included in the end. The

graph must also be undirected for this to work.

Listing 2.6: Prim's Algorithm (C++)

```
struct Edge
{
   int x, w;
   Edge(int a, int b) { x=a, w=b; }
   bool operator < (const Edge & b) const
   { return w > b.w; } //Since the priority queue sorts in descending
       order
};

vector<Edge> graph[10004]; //adjacency list
bool seen[10004]; //keeps track of visited nodes

long long prim()
{
   priority_queue<Edge> pq;
   seen[0] = true; //initial case
   for (int j=0; j<graph[0].size(); j++)
      pq.push(graph[0][j]);
   long long ret = 0;
   while (!pq.empty())
   {
      Edge c = pq.top();
      pq.pop();
      if (seen[c.x]) continue;
      else seen[c.x] = true; //mark
      ret += c.w; //add weight
      for (int j=0; j<graph[c.x].size(); j++)
         if (!seen[ graph[c.x][j].x ])
            pq.push(graph[c.x][j]);
   }
   return ret;
}
```

2.6 Bellman-Ford Algorithm

Bellman-Ford algorithm is a single source shortest path algorithm for weighted graphs. It is primarily used for graphs with negative edge weights. If there are no negative edge weights, then use Dijkstra's algorithm for faster runtime. Bellman-Ford's runtime is $O(EV)$ since it makes V passes through E edges. It can detect negative cycles in a graph by running one more time to see if it finds a shorter path to a vertex from the source. If no negative cycles are found, then Bellman-Ford will find the shortest path from a single source to all other vertices.

The predecessor array (pi) is filled up so you can retrieve the shortest path if you desire by tracing back the indices. If you don't need to remember the actual path, you are only interested in the value of the shortest path and thus should remove the 4 lines that use the array pi.

Listing 2.7: Bellman-Ford Algorithm (C++)

```
struct Edge //denotes to an edge of weight w that goes from x to y
{
   int u, v, w;
   Edge() {}
   Edge(int a, int b, int c)
   { u = a; v = b; w = c; }
};
#define MAX_E 2048
#define MAX_V 1024
Edge graph[ MAX_E ];
int d[ MAX_V ];
int pi[ MAX_V ];
bool bellman_ford(int n, int m, int source)
{
   fill(d, d+n, 1000000000); //fill with +inf
   fill(pi, pi+n, -1000000000); //predecessor array
   //initial conditions
   d[source] = 0;
   pi[source] = -1; //root
   for (int i=0; i < n-1; i++)
   {
      bool relaxed = false; //this is a little optimization I added
      for (int j=0; j<m; j++)
      if (d[ graph[j].v ] > d[ graph[j].u ] + graph[j].w) //relaxation
          step
      {
         d[ graph[j].v ] = d[ graph[j].u ] + graph[j].w; //found a better
             path
         pi[ graph[j].v ] = graph[j].u;
         relaxed = true;
      }
      if (!relaxed) break; //will break early if it finishes early
   }
   for (int i=0; i<m; i++) //check for negative cycle
      if (d[ graph[i].v ] > d[ graph[i].u ] + graph[i].w)
         return false;
   return true;
}
```

2.7 Bridge Detection

Bridges are edges in a graph that if removed will disconnect the graph. It can be found in a linear search with a DFS.

Listing 2.8: Bridge Detection (C++)

```
vector <int> list[1001];
int pre[1001], low[1001], parent[1001];

void dfs(int v, int & cnt)
{
   pre[v]=cnt++;
   low[v]=pre[v];
```

```
 8    for(int i=0; i<list[v].size(); i++)
 9    {
10       int w=list[v][i];
11       if(pre[w]==-1) //an unvisisted edge from v to w
12       {
13          parent[w]=v;
14          printf("%d %d\n", v, w);
15          dfs(w, cnt);
16          if(low[w]<low[v])
17             low[v]=low[w];
18          if(low[w]==pre[w]) //must be a bridge
19             printf("%d %d\n", w, v);
20       }
21       else if(parent[v]!=w && pre[w]<pre[v]) //a back link from v to w
22       {
23          printf("%d %d\n", v, w);
24          if(low[w]<low[v])
25             low[v]=low[w];
26       }
27    }
28 }
```

2.8 Articulation Point Detection

Articulation points are vertices in a graph that if removed will disconnect the graph. It can be found in a simple linear search with a DFS.

Listing 2.9: Articulation Point Detection (C++)

```
 1 int grid[101][101];
 2 int pre[101], low[101], parent[101];
 3
 4 void dfs(int v, int & cnt, int & crit, int n)
 5 {
 6    pre[v]=low[v]=cnt++;
 7    int child=0;
 8    bool critical=false;
 9    for(int w=1; w<=n; w++) //This code uses 1-base. Change to 0-base as
          necessary.
10       if(grid[v][w])
11          if(pre[w]==-1)
12          {
13             child++;
14             parent[w]=v;
15             dfs(w, cnt, crit, n);
16             if(low[w] < low[v])
17                low[v]=low[w];
18        //if any child's lowest preorder number referenced by a backedge is
19        //greater than or equal to parent's preorder number, then that
             parent is
20        //an articulation point
21             if(low[w] >= pre[v] && !critical && pre[v]!=0)
```

```
22                crit++, critical=true;
23          }
24          else if(parent[w]!=v && pre[w] < pre[v]) //A backedge
25             if(pre[w] < low[v])
26                low[v]=pre[w];
27    if(pre[v]==0 && child>1)
28         crit++;
29 }
```

2.9 Edmond's Karp Algorithm (Maximum-Flow)

The Edmond's Karp Maximum-Flow algorithm computes the feasible flow through a single-source, single-sink flow network that is maximum. This version runs in $O(VE^2)$ time.

Listing 2.10: Edmond's Karp Algorithm (C++)

```
1 int n;
2 int cap[250][250];//cap[u][v]=0 if edge e doesn t exist cap(e)=capacity(
      e)-flow(e)
3 int prev[250];
4 int maxFlow(int s, int t)
5 {
6    int ans = 0;
7    while(true)
8    {
9       fill(prev, prev + n, -1);
10      queue<int> q;
11      q.push(s);
12      while(!q.empty() && prev[t] == -1)
13      {
14         int u = q.front(); q.pop();
15         for(int v = 0; v < n; v++)
16         {
17            if( v!=s && prev[v] == -1 && cap[u][v] > 0)
18            {
19               prev[v] = u;
20               q.push(v);
21            }
22         }
23      }
24      if(prev[t] == -1)
25         break;
26      int bottle_neck = inf;
27      for(int v = t, u = prev[v]; u!=-1; v = u, u = prev[v])
28         bottle_neck = min(bottle_neck, cap[u][v]);
29      for(int v = t, u = prev[v]; u!=-1; v = u, u = prev[v])
30      {
31          cap[u][v] -= bottle_neck;
32          cap[v][u] += bottle_neck;
33      }
34      ans += bottle_neck;
35   }
```

```
36     return ans;
37 }
```

2.10 Minimum-Cost Maximum-Flow Algorithm

Used to find the max-flow of a network. Returns the min-cost through the reference parameter.

Listing 2.11: Min-Cost Max Flow (C++)

```
1 #define MAXN 100
2 #define inf 1000000000
3 int min_cost_max_flow(int n, int mat[][MAXN], int cost[][MAXN], int source
      , int sink, int flow[][MAXN], int& netcost)
4 {
5     int pre[MAXN], min[MAXN], d[MAXN], i, j, t, tag;
6     if (source == sink) return inf;
7     for (i=0; i<n; i++)
8         for (j=0; j<n; flow[i][j++]=0);
9     for(netcost=0; ; )
10    {
11        for (i=0; i<n; i++)
12            pre[i] = 0, min[i] = inf;
13        for (pre[source] = source+1, min[source]=0, d[source]=inf, tag=1;
              tag != 0; )
14            for (tag=t=0;t<n; t++)
15                if (d[t])
16                    for(i=0;i<n; i++)
17                        if (j=mat[t][i]-flow[t][i] && min[t]+cost[t][i] <
                              min[i])
18                        {
19                            tag = 1;
20                            min[i]=min[t]+cost[t][i];
21                            pre[i] = t+1;
22                            d[i] = d[t] < j ? d[t]:j;
23                        }
24                        else if(j=flow[i][t] && min[t] < inf && min[t]-
                              cost[i][t] < min[i])
25                        {
26                            tag = 1;
27                            min[i] = min[t]-cost[i][t];
28                            pre[i] = -t-1;
29                            d[i] = d[t] < j ? d[t] : j;
30                        }
31
32        if (!pre[sink]) break;
33        for (netcost += min[sink] * d[i=sink]; i!= source;)
34            if (pre[i] > 0)
35            {
36                flow[ pre[i]-1][i] += d[sink];
37                i = pre[i]-1;
38            }
39            else
```

```
40              {
41                  flow[i][-pre[i]-1] -= d[sink];
42                  i = -pre[i]-1;
43              }
44      }
45
46      for (j=i=0; i<n; j+=flow[source][i++]);
47
48      return j;
49  }
```

3 Geometry

3.1 Line Intersection

The following programs are used to find the intersection between two lines using parametric equations. The time solved through the system of parametric equations is important depending on the lines. Some things to note are the following:

Line Segment : $t \geq 0$ and $t \leq 1$
Infinite Line : $t \in \Re$
Ray : $t \geq 0$

Listing 3.1: Line Intersection (C++)

```
1 //solve the parametric equation for t given 2 pairs of points for each
      line
2 //the t solved is for the line represented by points a1 and a2
3 double parametric_solver(pair <int,int> a1, pair<int,int> a2, pair<int,int
      > b1, pair<int,int> b2)
4 {
5    int num=(a1.first-b1.first)*(b2.second-b1.second)-(a1.second-b1.second)
         * (b2.first-b1.first);
6    int den=(a2.second-a1.second)*(b2.first-b1.first)-(a2.first-a1.first) *
         (b2.second-b1.second);
7   //when denominator is 0, it means that the two lines are parallel
8    return den==0 ? inf : 1.*num/den;
9 }
10
11 //plug in the t from parametric_solver in to the parametric equations for
      line a
12 //to get the intersection point
13 pair<double,double> intersection(double t, pair <int,int> a1, pair<int,int
      > a2)
14 {
15    double x=a1.first+t*(a2.first-a1.first);
16    double y=a1.second+t*(a2.second-a1.second);
17    return make_pair(x,y);
18 }
```

3.2 Point/Vector Class

Listing 3.2: Point/Vector Class (Java)

```java
class Point implements Comparable<Point> //Point/Vector Class
{
   double x, y;
   Point(double a, double b) { x = a; y = b; }

   Point  plus (Point b) { return new Point(x + b.x, y + b.y); }
   Point  minus (Point b) { return new Point(x - b.x, y - b.y); }
   Point  times (double s) { return new Point(x * s, y * s); }
   Point  divide (double s) { return new Point(x / s, y / s); }
   double dot (Point b) { return this.x * b.x + y * b.y; }
   double cross (Point b) { return this.x * b.y - y * b.x; }
   double norm () { return this.dot(this); }
   double length () { return Math.sqrt(norm()); }
   Point  unit () { return this.divide(length()); }
   Point  rot90 () { return new Point(-y, x); }
   double dist (Point b) { return b.minus(this).length(); }
   Point  midPoint(Point a, Point b) { return a.plus(b).divide(2); }

   public String toString() { return String.format("(%f, %f)", x, y); }

   public int compareTo(Point b)
   {
      if (x < b.x) return -1;
      if (x > b.x) return 1;
      if (y < b.y) return -1;
      if (y > b.y) return 1;
      return 0;
   }
}
```

3.3 Circle Class

Listing 3.3: Circle Class (Java)

```java
class Circle
{
   Point c;
   double r;
   ArrayList<Point> inside;

   Circle(double x, double y, double z)
   {
      c = new Point(x, y);
      r = z;
      inside = new ArrayList<Point>();
   }

   Circle(Point a, double b)
   {
      c = a;
```

```
17        r = b;
18        inside = new ArrayList<Point>();
19        inside.add(a);
20     }
21
22     Circle(Point a, Point b)
23     {
24        c = a.plus(b).divide(2);
25        r = c.dist(a)+.1;
26        inside = new ArrayList<Point>();
27     }
28
29     Circle(Point x, Point y, Point z)
30     {
31        Point m = x.plus(y).divide(2);
32        Point n = y.minus(x).rot90().unit();
33        Point xm = m.minus(x);
34        Point zm = m.minus(z);
35        double t = -(xm.dot(xm) - zm.dot(zm)) / (2 * (xm.dot(n) - zm.dot(n))
             );
36        c = m.plus(n.times(t));
37        r = x.dist(c)+.1
38        inside = new ArrayList<Point>();
39     }
40
41     void add(Point a)
42     { inside.add(a); }
43
44     boolean contains(Point b)
45     { return c.minus(b).norm()<=r*r; }
46
47     double area()
48     { return Math.PI*r*r; }
49
50     public String toString()
51     { return c+" r = "+r; }
52  }
```

3.4 Minimum Enclosing Circle Algorithm

Listing 3.4: Minimum Enclosing Circle (Java)

```
1  public class MEC
2  {
3     static Circle MEC(ArrayList<Point> p)
4     {
5        ArrayList<Point> inside = new ArrayList<Point>();
6        Circle c;
7        if (p.size()>=2)
8           c = new Circle(p.get(0), p.get(1));
9        else return new Circle(p.get(0).x, p.get(0).y, .13);
```

```
10        inside.add(p.get(0));
11        if (p.size()>=2) inside.add(p.get(1))
12
13        for (int i=2; i<p.size(); i++)
14        {
15           if (!c.contains(p.get(i))) c = MEC1(p.get(i), inside);
16        inside.add(p.get(i));
17        }
18        c.r += .1;
19        c.inside.clear();
20        c.inside.addAll(inside);
21        return c;
22     }
23
24     static Circle MEC1(Point a, ArrayList<Point> p) //helper
25     {
26        Circle c = new Circle(a, p.get(0));
27        ArrayList<Point> inside = new ArrayList<Point>();
28        inside.add(p.get(0));
29
30        for (int i=1; i<p.size(); i++)
31        {
32           if (!c.contains(p.get(i))) c = MEC2(a, p.get(i), inside);
33           side.add(p.get(i));
34        }
35        return c;
36     }
37
38     static Circle MEC2(Point a, Point b, ArrayList<Point> p) //helper
39     {
40        Circle c = new Circle(a, b);
41        for (int i=0; i<p.size(); i++)
42        {
43           if (!c.contains(p.get(i)))
44              c = new Circle(a, b, p.get(i));
45        }
46        return c;
47     }
48  }
```

3.5 Andrew's Algorithm (Convex Hull)

Listing 3.5: Convex Hull (Java)

```
1  struct Point
2  {
3      int x, y;
4      Point() {}
5      Point(int a, int b)
6      { x = a; y = b; }
7
```

```
8     bool operator < (const Point & a) const
9     {
10        if (x==a.x) return y<a.y;
11        else return x<a.x;
12    }
13 };
14
15 #define MAX_P 1000
16 Point set_p[MAX_P];   //The points should be stored here
17 //left turn: AB X AC>0, right turn: AB X AC < 0. Inclusive will detect
       colinear points
18 bool left_turn(Point a, Point b, Point c)
19 { return (b.x-a.x)*(c.y-a.y)-(c.x-a.x)*(b.y-a.y) >= 0; }
20
21 //Computes the convex hull of a sorted set of n points (set_p)
22 vector<Point> convexHull(int n)
23 {
24    sort(set_p, set_p+n); //Points must be sorted for this to work.
25    vector<Point> upper, lower; //upper & lower hulls
26
27    //initial states
28    upper.push_back(set_p[0]);
29    upper.push_back(set_p[1]);
30    lower.push_back(set_p[n-1]);
31    lower.push_back(set_p[n-2]);
32
33    for (int i=2; i<n; i++)
34    {
35        upper.push_back(set_p[i]);
36        lower.push_back(set_p[n-1-i]);
37        int s1 = upper.size(), s2 = lower.size();
38        while (s1>=3 && left_turn(upper[s1-3], upper[s1-2], upper[s1-1]))
39            upper.erase(upper.begin() + s1-- -2); //update size
40        while (s2>=3 && left_turn(lower[s2-3], lower[s2-2], lower[s2-1]))
41            lower.erase(lower.begin() + s2-- -2); //update size
42    }
43    for (int i=1; i+1<lower.size(); i++) //combine hulls, without overlap
44        upper.push_back(lower[i]);
45    return upper;
46 }
```

3.6 Longitude and Latitude

Problem : What is the spherical/geographical distance between two cities p and q on earth with radius r, denoted by (p_lat, p_long) to (q_lat, q_long). All coordinates are in radians. (i.e. convert $[-180..180]$ range of longitude and $[-90..90]$ range of latitudes to $[-\pi..\pi]$ respectively.

Listing 3.6: Latitude and Longitude Conversion (Java)

```
1 double spherical_distance(p_lat,p_long,q_lat,q_long)
2 {
```

```
3     acos(sin(p_lat)*sin(q_lat)+cos(p_lat)*cos(q_lat)*cos(p_long - q_long))
          *r;
4 }
5
6 since cos(a-b) = cos(a)*cos(b) + sin(a)*sin(b), we can simplify the above
      method to:
7
8 double spherical_distance(p_lat,p_long,q_lat,q_long)
9 {
10    acos(sin(p_lat)*sin(q_lat)+cos(p_lat)*cos(q_lat)*cos(p_long)*cos(
          q_long)+
11                    cos(p_lat)*cos(q_lat)*sin(p_long)*sin(q_long))*r;
12 }
```

4 Dynamic Programming

4.1 Unbounded Knapsack

Given an infinite amount certain kinds of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

Parameters : An array of weights, an array of values, and an integer representing the size of the knapsack.
Returns : An array of length size+1, the value at index 'i' is the optimal value obtained with weight i. A value of -1 means that weight i is not possible to obtain.
Complexity : $O(NT)$ (N = number of items, T = size of knapsack)

Listing 4.1: Unbounded Knapsack (Java)

```
int[] unboundedKnapsack(int[] weight, int[] value, int size)
{
   int[] dp = new int[size+1];
   Arrays.fill(dp, -1);
   dp[0] = 0;
   int n = weight.length;
   for (int i=0; i<=size; i++)
      if (dp[i]!=-1)
         for (int j=0; j<n; j++)
            if (i+weight[j]<=size) //if this fits
               dp[ i+weight[j] ] = Math.max(dp[ i+weight[j] ], dp[i]+value
                  [j]);
   return dp;
}
```

4.2 Bounded (0/1) Knapsack

Given certain kinds of items, each with a weight and a value, determine the largest total value you can take without exceeding your weight limit.

Note : You can't take more than one of each item.
Parameters : An array in weights, an array of values, and an integer representing the size of the knapsack.
Returns : A 2D array of size $size + 1 \times knapsack\_size + 1$, were the value at index (i, j) is optimal value of a subset of the first i elements that weight a total of j. A value of -1 means that its impossible to find.
Complexity : $O(NM)$ (N = number of items, T = Size of knapsack)

Listing 4.2: Bounded Knpasack (Java)

```java
int[][] boundedKnapsack(int[] weight, int[] value, int size)
{
   int n = weight.length; //number of items
   int[][] dp = new int[n][size+1];
   for (int[] a : dp) Arrays.fill(a, -1);
   dp[0][0] = 0; //initial condition 1
   if (weight[0] <= size) dp[0][weight[0]] = value[0]; //initial condition
       2
   for (int i=1; i<n; i++)
      for (int j=0; j<=size; j++)
      {
         dp[i][j] = dp[i-1][j];
         if (j-weight[i]>=0 && dp[i][j] < dp[i-1][j-weight[i]] + value[i])
            dp[i][j] = dp[i-1][j-weight[i]] + value[i];
      }
  return dp;
}
```

4.3 Longest Increasing Subsequence

Complexity : $O(N \log N)$ The idea behind this version is: Imagine you are playing a game where you have to place a sequence of numbers into stacks. The first number always creates the first stack, then after, you place the rest of the sequence following these rules: 1. You consider the cards in the same order you received them. 2. To insert a card in a stack, it must be smaller than all other cards in that stack. Equivalently, it must be smaller than the top of the stack 3. If there is no stack that can accept a certain card, make a new stack to the right of the last stack. 4. If there exist multiple stacks which can accept a card, pick the leftmost stack. 5. Whenever you insert a card into a stack that is not the first, save a pointer from that card to the top of the previous stack.

I claim that the number of stacks is the length of the LIS and if you follow the pointers from the very end you can build the LIS.

Note : It easy to see that the top of all of the stacks will be in increasing order from left to right. (proof via contradiction). Therefore you can find which stack to insert a card into in log(k) if you use a binary search, where k is the number of stack.

Listing 4.3: Longest Increasing Subsequence (Java)

```java
int longestIncreasingSubsequence(int[] n)
{
   /*
    n is the sequence
    p represents the predecessor array
    m represents the stack of cards
   */
   //initial conditions
   m[0] = 0;
   p[0] = -1;
   int len_m = 1;
   for (int i=1; i<len; i++)
   {
      int  lo = 0, hi = len_m-1, ave;
```

```
15        while (lo <= hi)
16        {
17           ave = (lo+hi)/2;
18           if (n[ m[ave] ] == n[i]) break; //repeated element, place onto
                pile
19           else if (n[ m[ave] ] < n[i]) lo = ave + 1;
20           else if (hi != ave) hi = ave;
21           else break; //didn't find it
22        }
23        if (lo<=hi) //if binary search was successful
24        {
25            m[ave] = i; //place n[i] on top of the correct "stack"
26            if (ave!=0) p[i] = m[ave-1]; //set the back pointer
27            else p[i] == -1; //in first stack, so it has no predecessor
28        }
29        else  //must create a new stack
30        {
31            m[len_m] = i;
32            p[i] = m[len_m-1];
33            len_m++; //update number of stacks
34        }
35     }
36  }
```

4.4 Longest Common Subsequence

This algorithm will find the longest common sub-sequence between two strings, a and b. M and N are the lengths of a and b, respectively. Runs in $O(n^2)$.

Listing 4.4: Longest Common Subsequence (Java)

```
1  int lcs(string a, string b) {
2     int dp[1024][1024];
3     for (int i=0; i<a.length(); i++)
4     fill(dp[i], dp[i]+b.length(), 0);
5
6     dp[0][0] = (a[0]==b[0]); //for Java, use a.charAt() and b.charAt()
7
8     //initial conditions (first column and row are filled in)
9     for (int i=1; i<a.length(); i++)
10      dp[i][0] = max(dp[i-1][0], (int)(a[i]==b[0]));
11
12    for (int j=1; j<b.length(); j++)
13      dp[0][j] = max(dp[0][j-1], (int)(a[0]==b[j]));
14
15    for (int i=1; i<a.length(); i++) //actual algorithm
16      for (int j=1; j<b.length(); j++)
17       if (a[i] == b[j]) dp[i][j] = dp[i-1][j-1]+1;
18         else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
19
20    return dp[a.length()-1][b.length()-1]; //the answer is stored here
21 }
```

4.5 Longest Common Substring

Complexity : $O(n^2)$

Listing 4.5: Longest Common Substring (Java)

```
int longest_common_substr(string a, string b)
{
  //set initial values
  for (int i=0; i<=a.length(); i++) dp[i][b.length()] = 0;
   for (int i=0; i<=b.length(); i++) dp[a.length()][i] = 0;

   int best = 0;
   for (int i=a.length()-1; i>=0; i--)
     for (int j=b.length()-1; j>=0; j--)
       if( a[i] == b[j] ) //if its a match, you can extend both prefixes
           by one
       {
         dp[i][j] = 1 + dp[i+1][j+1];
         if (best < dp[i][j]) //remember best so far
         best = dp[i][j];
       }
       else dp[i][j] = 0; //else you have to start over
  return best;
}
```

4.6 Maximum Contiguous Sum

Problem Statement : Find the maximum sum of contiguous elements in an array
Parameters : An array of integers
Returns : Max sum found
Complexity : $O(N)$ where N is the length of the array

Listing 4.6: Maximum Continuous Sum (Java)

```
static int maxSum(int[] num)
{
  int n = num.length;
  int[] dp = new int[n];

  dp[0] = num[0];
  for (int i=1; i<n; i++)
    dp[i]  = (dp[i-1]>0) ? dp[i-1]+num[i] : num[i];

  int ret = dp[0];
  for (int x : dp)
    ret = Math.max(ret, x);

  return ret;
}
```

4.7 Maximum Rectangular Sum

Problem Statement : Finds the maximum sum of all subrectangles in an NxN matrix.

Use a partial sums array on the columns in order to be able to compute the sum of the elements from $g[a][j]$ through $g[b][j]$, for any a,b ($a < b$) in $O(1)$

Then we traverse through all possible horizontal boundaries $y = a$ and $y = b$ and build a 1D array where $array[i] = \sum_{k=a}^{b} g[k][i]$. Then run the maximum contiguous 1D sum algorithm on that array.

4.8 Levenshtein Distance (Edit Distance)

The Levenshtein (Edit) Distance is the minimum number of changes in spelling required to change one word into another.

Listing 4.7: Edit Distance (Java)

```
int LevenshteinDistance(char s[1..m], char t[1..n])
{
  //for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t;
  // note that d has (m+1)x(n+1) values
  declare int d[0..m, 0..n]

  for i from 0 to m
    d[i, 0] := i
    // the distance of any first string to an empty second string
  for j from 0 to n
    d[0, j] := j
    // the distance of any second string to an empty first string

  for j from 1 to n
  {
    for i from 1 to m
    {
      if s[i] = t[j] then
        d[i, j] := d[i-1, j-1]       // no operation required
      else
        d[i, j] := minimum
                   (
                     d[i-1, j] + 1,  // a deletion
                     d[i, j-1] + 1,  // an insertion
                     d[i-1, j-1] + 1 // a substitution
                   )
    }
  }

  return d[m,n]
}
```

5 Tree Data Structures

5.1 Fenwick Tree

Listing 5.1: Fenwick Tree 1D (Java)

```
1 public class FenwickTree1D
2 {
3    int[] tree;
4    FenwickTree1D(int max_size)
5    { tree = new int[max_size+2]; }
6    int get(int x)
7    {
8       int ret = 0;
9       while (x>0)
10      {
11         ret += tree[x];
12         x -= (x & -x);
13      }
14      return ret;
15   }
16   void set(int x, int val)
17   {
18      val -= get(x);
19      update(x, val);
20   }
21   void update(int x, int val)
22   {
23      while (x<tree.length)
24      {
25         tree[x] += val;
26         x += (x & -x);
27      }
28  }
29 //a faster way of getting just one value.
30  int getSingle(int x)
31  {
32     int ret = tree[x];
33         if (x > 0)
34         {
35             int z = x - (x & -x);
36             x--;
37             while (x != z)
38             {
```

```
39                ret -= tree[x];
40                x -= (x & -x);
41              }
42          }
43          return ret;
44       }
45 }
```

Listing 5.2: Fenwick Tree 2D (Java)

```
1  public class FenwickTree2D
2  {
3     int[][] tree;
4     FenwickTree2D(int max_x, int max_y)
5     {
6        tree = new int[max_x+2][max_y+2];
7     }
8     int get(int x, int y)
9     {
10       int ret = 0;
11       while (x > 0)
12       {
13          int ty = y;
14          while (ty > 0)
15          {
16             ret += tree[x][ty];
17             ty -= (ty & -ty);
18          }
19          x -= (x & -x);
20       }
21       return ret;
22    }
23
24    void set(int x, int y, int val)
25    {
26       val -= get(x, y);
27       update(x, y, val);
28    }
29
30    void update(int x, int y, int val)
31    {
32       while (x<tree.length)
33       {
34          int ty = y;
35          while (ty<tree[0].length)
36          {
37             tree[x][y] += val;
38             ty += (ty & -ty);
39          }
40          x += (x & -x);
41       }
42    }
43 }
```

5.2 Trie

Listing 5.3: Trie (C++)

```
1 struct node
2 {
3    char key;
4    node * next ,*children;
5    node(char chr, node * a, node * c)
6    {
7       key = chr;
8       next = a;
9       children = c;
10   }
11 };
12
13 struct Trie
14 {
15   node * root;
16   int size;
17   Trie(){size = 0; root = NULL;}
18   void add(string & s, int x = 0)
19   {
20      add(s, root, x);
21   }
22   void add(string & s, node *& N, int x)
23   {
24      if(x == s.length())return;
25      if(N == NULL)
26      {
27         size++;
28         N =  new node(s[x], N, NULL);
29         add(s, N->children, x+1);
30         return;
31      }
32      if(N->key == s[x])
33      {
34         add(s, N->children, x+1);
35         return;
36      }
37      add(s, N->next, x);
38   }
39 };
```

6 Miscellaneous

6.1 Kirchoff's Matrix

Kirchoff's matrix allows one to calculate the number of spanning trees in a graph as follows:

1. Compute the Laplacian Matrix as follows
   $A[i][i]$ = degree of vertex i
   $A[i][j]$ = -1 if vertex i and vertex j are connected, 0 otherwise
2. The number of spanning trees is equal to the determinant of any cofactor matrix. (The original matrix with the first row and column removed is a valid cofactor)

6.2 Josephus Problem

Of the first n numbers, if you pick and remove the kth number, determine the last one standing.

Listing 6.1: Josephus Problem (Java)

```
1 int joseph (int n, int k)
2 {
3    int res = 0 ;
4    for ( int i = 1 ; i <= n ; i++ )
5       res = ( res + k ) % i ;
6    return res + 1;
7 }
```

6.3 Poker Class

Listing 6.2: Poker Class (Java)

```
1 class PokerHand implements Comparable < PokerHand >
2 {
3     int [] rank = new int [13];
4     int [] suit = new int [4];
5     int [] cmp ;
6     PokerHand ( String [] hand ) {
7         for ( String s: hand ) {
8             switch (s. charAt (0)) {
9                 case 'T': ++ rank [8]; break ;
10                case 'J': ++ rank [9]; break ;
11                case 'Q': ++ rank [10]; break ;
12                case 'K': ++ rank [11]; break ;
```

```
13                  case 'A': ++ rank [12]; break ;
14                  default : ++ rank [s. charAt (0) - '2'];
15                       break;
16              }
17
18              switch (s. charAt (1)) {
19                  case 'C': ++ suit [0]; break ;
20                  case 'D': ++ suit [1]; break ;
21                  case 'H': ++ suit [2]; break ;
22                  case 'S': ++ suit [3]; break ;
23                  default : throw new RuntimeException ();
24              }
25          }
26
27          cmp = new int [] { straightFlush (), fourOfAKind (),
28          fullHouse (), flush (), straight (),
29          threeOfAKind (), twoPairs (), pair (),
30          highCard ()
31      };
32  }
33
34  int flush () {
35      for ( int s: suit )
36          if(s == 5) return highCard ();
37      return -1;
38  }
39
40  int straight () {
41      for ( int i = 12; i >= 4; --i) {
42          boolean good = true ;
43          for ( int j = 0; j <= 4; ++j)
44              if( rank [i - j] != 1) {
45                  good = false ;
46                  break ;
47              }
48          if( good ) return i;
49      }
50      return -1;
51  }
52
53  int highCard() {
54      int ret = 0;
55      for (int i = 12; i >= 0; --i) if(rank[i] == 1)
56          ret = ret * 13 + i;
57          return ret ;
58  }
59
60  int straightFlush () {
61      return flush () == -1 ? -1 : straight ();
62  }
63
64  int fourOfAKind () {
65      for (int i = 12; i >= 0; --i)
66          if(rank [i] == 4) return i;
```

```
67      return -1;
68  }
69
70  int threeOfAKind () {
71      for ( int i = 12; i >= 0; --i)
72          if( rank [i] == 3) return i;
73      return -1;
74  }
75
76
77  int pair() {
78      int ret = p (1);
79      return ret==-1 ? -1 : ret*13*13*13+highCard();
80  }
81
82  int twoPairs() {
83      int ret = p (2);
84      return ret == -1 ? -1 : ret * 13 + highCard ();
85  }
86
87  int fullHouse() {
88      int ret = pair ();
89      return ret == -1 ? -1 : threeOfAKind ();
90  }
91
92  int p(int goal) {
93      int pc = 0;
94      int ret = 0;
95      for ( int i = 12; i >= 0; --i)
96          if(rank [i]== 2) {
97              ret = ret * 13 + i;
98              ++pc;
99          }
100     return pc == goal ? ret : -1;
101 }
102
103 public int compareTo (PokerHand h){
104     int index = 0;
105     while(index<cmp.length&&cmp[index]==h.cmp[ index ])
106         ++index;
107         return index==cmp. length ? 0 : cmp[index]-h.cmp[index];
108     }
109 }
```

6.4 Decimal to Roman Numeral Converter

Listing 6.3: Decimal to Roman Numeral Converter (C++)

```
1  #include<stdio.h>
2  //Convert number 1 to 10
3  void unit(int n){
```

```
4       switch(m)
5       {
6       case 3: cout <<    I   ;
7       case 2: cout <<    I   ;
8       case 1: cout <<    I   ; break;
9       case 4: cout <<    I   ;
10      case 5: cout <<    V   ; break;
11      case 6: cout <<    V I   ; break;
12      case 7: cout <<    V I I   ; break;
13      case 8: cout <<    V I I I   ; break;
14      case 9: cout <<    I X   ; break;
15      }
16  }
17
18  //Convert number 10 to 100
19  void ten(int n)
20  {
21      switch(n)
22      {
23      case 3: cout <<    X   ;
24      case 2: cout <<    X   ;
25      case 1: cout <<    X   ; break;
26      case 4: cout <<    X   ;
27      case 5: cout <<    L   ; break;
28      case 6: cout <<    L X   ; break;
29      case 7: cout <<    L X X   ; break;
30      case 8: cout <<    L X X X   ; break;
31      case 9: cout <<    X C   ; break;
32      }
33  }
34
35  //Convert number 100-500
36  void hundred(int n)
37  {
38      switch(n)
39      {
40      case 3: cout <<    C   ;
41      case 2: cout <<    C   ;
42      case 1: cout <<    C   ; break;
43      case 4: cout <<    C   ;
44      case 5: cout <<    <   ; break;
45      }
46  }
47
48  void roman(int n)
49  {
50      int a, i;
51      if(n>=500)
52      {
53          a = n/500;
54          for(i = 1; i<=a; i++)
55              cout <<    M   ;
56      }
57      n%=500;
```

```
58      hundred(n/100);
59      n%=100;
60      ten(n/10);
61      unit(n%10);
62  }
```

6.5 Expression Parsing

Listing 6.4: Expression Parsing (C++)

```
1  //Crazy Calculator
2  #include <string>
3  #include <stack>
4  #include <cstdio>
5  #include <algorithm>
6  #include <sstream>
7  #include <iostream>
8  #include <map>
9  #define max_n 1025
10 using namespace std;
11 char in[max_n+1];
12 struct op
13 {
14   int p;
15   char c;
16   bool leftA;
17   op( char cc, int pp, bool la)
18   {
19   c = cc;
20   p = pp;
21   leftA = la;
22   }
23   op()
24   {
25   }
26 };
27 map<char, op> ops;
28 bool isD(char c){return c <= '9' && c>= '0';}
29 int operate(char c, int a, int b)
30 {
31   switch(c)
32   {
33   case '-': return a-b;
34   case '+': return a+b;
35   case '*': return a*b;
36   case '/': return a/b;
37   default: return -1;
38   }
39 }
40 int eval(string s)
41 {
```

```
42   stack<int> numstk;
43   stack<op> opstk;
44   for(int i = 0 ; i < s.length(); i++)
45   {
46     if( isD( s[i] ) )
47       {
48         int n = 0;
49         while( isD( s[i] ) && i < s.length())
50           {
51               n = 10* n + (s[i] - '0');
52               i++;
53             }
54         i--;
55           numstk.push(n);
56       }
57     else{
58       op curop = ops[ s[i] ];
59       if(curop.leftA)
60         while(!opstk.empty() && curop.p <= opstk.top().p)
61             {
62               char stkop = opstk.top().c;
63               opstk.pop();
64               int b = numstk.top();
65               numstk.pop();
66               int a = numstk.top();
67               numstk.pop();
68               int c = operate(stkop, a, b);
69               numstk.push(c);
70             }
71       else
72         while(!opstk.empty() && (curop.p < opstk.top().p || (curop.p ==
              opstk.top().p && curop.c != opstk.top().c) ))
73             {
74               char stkop = opstk.top().c;
75               opstk.pop();
76               int b = numstk.top();
77               numstk.pop();
78               int a = numstk.top();
79               numstk.pop();
80               int c = operate(stkop, a, b);
81               numstk.push(c);
82             }
83         opstk.push(curop);
84    }
85   }
86   return numstk.top();
87
88 }
89
90 int main()
91 {
92   int T;
93   scanf("%d", &T);
94   int c= 0;
```

```
95   while(T--)
96   {
97     if( c > 0)
98         printf("\n");
99     c++;
100    ops.clear();
101    char opchr[5];
102    map<char, char> rep;
103    for(int i = 0 ; i < 4; i++)
104      {
105        scanf("%s\n", opchr);
106          rep[opchr[1]] = opchr[0];
107          ops[opchr[0]] = op(opchr[0], opchr[2] - '0', opchr[3] == 'L');
108      }
109    string s;
110    char c;
111    while( (c = getchar()) != '\n' && (c != EOF))
112      {
113        while( true ){
114          if( rep.find(c) != rep.end() )
115              s += rep[c];
116          else
117               s+= c;
118          if ( (c = getchar()) == '\n' || (c == EOF) )
119              break;
120        }
121        string cpy = s;
122       s+='E';
123        ops['E'] = op('E',-1, true);
124        int ans = eval(s);
125        printf("%s = %d\n", cpy.c_str(), ans);
126        // cout<<s<<endl;
127            s = "";
128      }
129
130  }
131 }
```

6.6 String Matching with KMP

Returns true if text matches into some substring of p.

Listing 6.5: KMP String Matching (C++)

```
1 int v[2002];
2
3 void table(string p)
4 {
5     v[0]= -1;
6
7     int cur = 0;
8     for(int j = 1;j < p.size();j++)
```

```
 9      {
10      cur = v[j-1];
11          while(cur >=0 && p[cur]!=p[j-1])
12            cur=v[cur];
13          v[j]=cur;
14      }
15  }
16
17  bool kmp(string p,string text)
18  {
19      table(text);
20    int i = 0;
21    int k = 0;
22    while(i < p.length())
23    {
24      if(k == -1)
25      {
26        i++;
27        k = 0;
28      }
29      else if(p[i] == text[k])
30      {
31        i++;
32        k++;
33        if(k == text.length())
34          return true;
35      }
36      else
37        k = v[k];
38    }
39      return false;
40  }
```

7 Language Specific References

7.1 C++ Startup Code

Listing 7.1: C++ Startup Code (C++)

```cpp
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <queue>
#include <stack>
#include <bitset>
#include <algorithm>
#include <functional>
#include <numeric>
#include <utility>
#include <sstream>
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <cctype>
#include <string>
#include <cstring>

using namespace std;

int main()
{
  return 0;
}
```

7.2 Java Start-Up Code

Listing 7.2: Java Startup Code (Java)

```java
import java.io.*;
```

```
2 import java.util.regex.*;
3 import java.math.*;
4 import java.util.*;
5 import static java.util.Arrays.*;
6 import static java.lang.Integer.*;
7 import static java.lang.Math.*;
8 import static java.math.BigInteger.*;
9 import static java.util.Collections.*;
10
11 public class ClassName
12 {
13   public static void main(String[] args) throws Exception
14   {
15     BufferedReader r = new BufferedReader(
16                          new InputStreamReader(System.in));
17     //or
18     Scanner in = new Scanner(System.in);
19               //this is much slower than BufferedReader but simpler to use
20   }
21 }
```

7.3 Java BigInteger Reference

Listing 7.3: Java BigInteger Code (Java)

```
1 import java.math.BigInteger;
2 public class FactorialBigInteger {
3
4   public static void main(String[] args) {
5     //Calculate up to 100!
6     BigInteger[] fact = new BigInteger[101];
7     fact[0] = BigInteger.ONE;
8     fact[1] = BigInteger.ONE;
9     for(int i = 2; i < 101; i++)
10    {
11      fact[i] = fact[i-1].multiply(BigInteger.valueOf(i));
12    }
13
14    //Now use this to calculate 100C50
15    BigInteger res = fact[100].divide(fact[50].multiply(fact[50]));
16
17    //We can add this
18    BigInteger res2 = res.add(res);
19
20    //We can also take maxes or mins
21    //max will have the same value as res2, min same as res
22    BigInteger max = res2.max(res);
23    BigInteger min = res2.min(res);
24
25    //If we want to compare values
26    //This should print "res < res2"
```

```
27      if(res.compareTo(res2) == -1)
28        System.out.println("res < res2");
29      else if(res.compareTo(res2) == 0)
30        System.out.println("res = res2");
31      else if(res.compareTo(res2) == 1)
32        System.out.println("res > res2");
33
34      //We can exponentiate
35      BigInteger factsquare = fact[100].pow(2);
36
37      //We can do modular arithmetic
38      BigInteger divs[] = fact[100].divideAndRemainder(BigInteger.valueOf
            (29));
39      BigInteger quotient = divs[0];
40      BigInteger remainder = divs[1];
41      //Alternatively, this will always return nonegative
42      BigInteger remainder2 = fact[100].mod(BigInteger.valueOf(29));
43      //In the same vein we can also do this,
44      //Which returns a number equivalent to (100!)^2 mod 29
45      //You can use negative exponents in this method
46      BigInteger remainder3 = fact[100].modPow(BigInteger.valueOf(2),
            BigInteger.valueOf(29));
47    }
48
49  }
```