Contents

1 Oinarrizkoak

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> VI;
typedef vector<vector<int>> VVI;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;

typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;

typedef vector<bool> VB;

```

```
typedef pair<int, int> PII;
typedef vector<PII> VPII;

ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

2 Math

2.1 EulerFunction.cpp

```
// a^phi(N) = 1 (mod N) if gcd(a, N) = 1
long long euler_totient2(long long n, long long ps) {
    for (long long i = ps; i * i <= n; i++) {
        if (n % i == 0) {
            long long p = 1;
            while (n % i == 0) {
                n /= i;
                p *= i;
            }
            return (p - p / i) * euler_totient2(n, i + 1);
        }
        if (i > 2) i++;
    }
    return n - 1;
}
long long euler_totient(long long n) {
    return euler_totient2(n, 2);
}
```

2.2 NthPermutation.c

```
// e The number of entries
// n The index of the permutation
int fact[MAX];  // MAX >= n
int perm[MAX];  // MAX >= n
void nthPermutation(const int e, int n) {
   int j, k = 0;

   // compute factorial numbers
   fact[k] = 1;
   while (++k < e)
      fact[k] = fact[k - 1] * k;

   // compute factorial code
   for (k = 0; k < n; ++k) {
      perm[k] = i / fact[n - 1 - k];
      n = n % fact[e - 1 - k];
   }

   // readjust values to obtain the permutation
   // start from the end and check if preceding values are lower
   for (k = e - 1; k > 0; --k)
      for (j = k - 1; j >= 0; --j)
         if (perm[j] <= perm[k])
            perm[k]++;

   // perm[0..e] contains the nth permutation
}
```

2.3 GaussJordan.cpp

```
// Gauss-Jordan elimination with full pivoting.
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
// Running time: O(n^3)
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
// OUTPUT:   X       = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]
const double EPS = 1e-10;

double GaussJordan(VVD &a, VVD &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  double det = 1;

  for (int i = 0; i < n; i++) {
```

```
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
    if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    //if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    double c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}
```

2.4 NthPermutationRepetitions.cpp

```
typedef map<char,int> mci;
L factorial(L i) {
    if (i <= 1) return 1;

    L totala = 2;
    for (L j = 3; j <= i; ++j)
        totala *= j;
    return totala;
}
string nthPermutationRepetitions(string input, L n) {
    mci mapa;
    int input_len = input.length();
    for (int i = 0; i < input_len; ++i) {
        if (mapa.find(input[i]) == mapa.end())
            mapa[input[i]] = 1;
        else
            mapa[input[i]]++;
    }

    string buffer;
    buffer.resize(input_len);
    L totala = 0;
    for (int i = 0; i < input_len; ++i) {
        for (mci::iterator elem = mapa.begin(); elem != mapa.end(); ++elem) {
            if (elem->second > 0) {
                elem->second--;
                L perm = factorial(input_len - i -1);
                for (mci::iterator c = mapa.begin(); c != mapa.end(); ++c)
                    perm /= factorial(c->second);

                if (n < totala + perm) {
                    buffer[i] = elem->first;
                    break;
                }
                totala += perm;
                elem->second++;
            }
        }
    }
    return buffer;
}
```

2.5 NumberTheory.cpp

```
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.
```

```

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a/gcd(a,b)*b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]).  Return (z,M).  On
// failure, M = -1.  Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
```

```
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

2.6 Simplex.cpp

```
// Two-phase simplex algorithm for solving linear programs of the form
//      maximize      c^T x
//      subject to    Ax <= b
//                    x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
// OUTPUT: value of the optimal solution (infinity if unbounded
//          above, nan if infeasible)
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
    D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
    if (phase == 2 && N[j] == -1) continue;
    if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
    if (D[i][s] <= 0) continue;
    if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
        D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
    int s = -1;
    for (int j = 0; j <= n; j++)
      if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
    Pivot(i, s);
      }
```

```
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};
```

2.7 Factors.cpp

```
// COMPLEXITY: O(sqrt(n))
//Returns a list of all the factors of n
//Example: n = 12 -> result = [2, 2, 3]
// n > 1
VI factors(int n)  {
    int z = 2;
    VI result;
    while (z * z <= n) {
        if (n % z == 0) {
            result.push_back(z);
            n /= z;
        } else z++;
    }
    if (n > 1) result.push_back(n);
    return result;
}
```

2.8 BrentCycleDetection.cpp

```
template<typename T, typename F>
class BrentCycleDetection {
    F f;
public:
    PII findCycle(T x0) {
        // 1st part: search successive powers of two
        int power = 1, lambda = 1; T tortoise = x0, hare = f(x0);
        while (tortoise != hare) {
            if (power == lambda) {
                tortoise = hare;
                power *= 2;
                lambda = 0;
            }
            hare = f(hare);
            lambda++;
        }
        // 2nd part: Find the position of the first repetition of length lambda
        int mu = 0;
        tortoise = hare = x0;
        for (int i = 0; i < lambda; i++) hare = f(hare);
        // 3rd part: the hare and tortoise move at same speed till they agree
        while (tortoise != hare) {
            tortoise = f(tortoise);
            hare = f(hare);
            mu++;
        }
        return PII(mu, lambda);
    }
};
```

2.9 ModularExponentiation.cpp

```
//Complexity: O(log b)
//Returns (a^b)%c
int modular_pow(int a, int b, int c) {
    int result = 1;
    while (b > 0) {
        if (b % 2 == 1)
            result = (result * a) % c;
        b = b >> 1;
        a = (a * a) % c;
    }
    return result;
}
```

3 StringProcessing

3.1 KMP.cpp

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], n, m; // b = back table, n = length of T, m = length of P
void kmpPreprocess() {
```

```
    int i = 0, j = -1; b[0] = -1;
    while (i < m) {
        while (j >= 0 && P[i] != P[j]) j=b[j];
        i++; j++;
        b[i] = j; // observe i = 8, 9, 10,
} }

void kmpSearch() {
    int i = 0, j = 0;
    while (i < n) {
        while (j >= 0 && T[i] != P[j]) j = b[j];
        i++; j++;
        if (j == m) {
            printf("P␣is␣found␣at␣index␣%d␣in␣T\n", i - j);
            j = b[j]; // prepare j for the next possible match
} } }
```

3.2 EditDistance.java

```
public class EditDistance {

    public static final EditDistance LEVENSHTEIN_DISTANCE = new EditDistance(2, -1, -1, -1);
    public static final EditDistance LONGEST_COMMON_SUBSEQUENCE = new EditDistance(1, -1000000000, 0, 0
        );

    public final int matchScore;     // Character a[i] and b[i] match and we do nothing
    public final int mismatchScore;  // Character a[i] and b[i] mismatch and we replace a[i] with b[i]
    public final int insertScore;    // We insert a space in A[i]
    public final int deleteScore;    // We delete a letter from A[i]

    public EditDistance(int matchScore, int mismatchScore, int insertScore, int deleteScore) {
        this.matchScore = matchScore;
        this.mismatchScore = mismatchScore;
        this.insertScore = insertScore;
        this.deleteScore = deleteScore;
    }

    public int getMaxScore(String a, String b) { // Needleman-Wunsch's algorithm
        final int[][] table = new int[a.length()+1][b.length()+1];
        for (int i = 1; i <= a.length(); i++) table[i][0] = i * deleteScore;
        for (int j = 1; j <= b.length(); j++) table[0][j] = j * insertScore;
        for (int i = 1; i <= a.length(); i++) {
            for (int j = 1; j <= b.length(); j++) {
                table[i][j] = table[i-1][j-1] + (a.charAt(i-1) == b.charAt(j-1) ? matchScore :
                    mismatchScore);
                table[i][j] = Math.max(table[i][j], table[i-1][j] + deleteScore);
                table[i][j] = Math.max(table[i][j], table[i][j-1] + insertScore);
            }
        }
        return table[a.length()][b.length()];
    }

}
```

3.3 SuffixArray.cpp

```
#define MAX_N 100010
//O(nlogn)
char T[MAX_N]; // the input string, up to 100K characters
int n; // the length of input string
int RA[MAX_N], tempRA[MAX_N]; // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N]; // suffix array and temporary suffix array
int c[MAX_N]; // for counting/radix sort

void countingSort(int k) { // O(n)
    int i, sum, maxi = max(300, n); // up to 255 ASCII chars or length of n
    memset(c, 0, sizeof c); // clear frequency table
    for (i = 0; i < n; i++) // count the frequency of each integer rank
        c[i + k < n ? RA[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++) // shuffle the suffix array if necessary
        tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
    for (i = 0; i < n; i++) // update the suffix array SA
        SA[i] = tempSA[i];
}

void constructSA() {  // this version can go up to 100000 characters
    int i, k, r;
    for (i = 0; i < n; i++) RA[i] = T[i]; // initial rankings
```

```
    for (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
    for (k = 1; k < n; k <<= 1) { // repeat sorting process log n times
        countingSort(k); // actually radix sort: sort based on the second item
        countingSort(0); // then (stable) sort based on the first item
        tempRA[SA[0]] = r = 0; // re-ranking; start from rank r = 0
        for (i = 1; i < n; i++) // compare adjacent suffixes
            tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
            (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
        for (i = 0; i < n; i++) // update the rank array RA
            RA[i] = tempRA[i];
        if (RA[SA[n-1]] == n-1) break; // nice optimization trick
} }

int main() {
    n = (int)strlen(gets(T)); // input T as per normal, without the    $
    T[n++] =    $    ; // add terminating character
    constructSA();
    for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;


ii stringMatching() { // string matching in O(m log n)
    int lo = 0, hi = n-1, mid = lo; // valid matching = [0..n-1]
    while (lo < hi) { // find lower bound
        mid = (lo + hi) / 2; // this is round down
        int res = strncmp(T + SA[mid], P, m); // try to find P in suffix    mid
        if (res >= 0) hi = mid; // prune upper half (notice the >= sign)
        else           lo = mid + 1; // prune lower half including mid
    }                   // observe    =     in "res >= 0" above
    if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1); // if not found
    ii ans; ans.first = lo;
    lo = 0; hi = n - 1; mid = lo;
    while (lo < hi) { // if lower bound is found, find upper bound
        mid = (lo + hi) / 2;
        int res = strncmp(T + SA[mid], P, m);
        if (res > 0) hi = mid; // prune upper half
        else          lo = mid + 1; // prune lower half including mid
    }               // (notice the selected branch when res == 0)
    if (strncmp(T + SA[hi], P, m) != 0) hi--; // special case
    ans.second = hi;
    return ans;
} // return lower/upperbound as first/second item of the pair, respectively



void computeLCP() {
    int i, L;
    Phi[SA[0]] = -1; // default value
    for (i = 1; i < n; i++) // compute Phi in O(n)
        Phi[SA[i]] = SA[i-1]; // remember which suffix is behind this suffix
    for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
        if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case
        while (T[i + L] == T[Phi[i] + L]) L++; // L increased max n times
        PLCP[i] = L;
        L = max(L-1, 0); // L decreased max n times
    }
    for (i = 0; i < n; i++) // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct position
}
```

4 JavaFastIO

4.1 MyScanner.java

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

    //-----------PrintWriter for faster output---------------------------------
    public static PrintWriter out;

    //-----------MyScanner class for faster input----------
    public static class MyScanner {
        BufferedReader br;
        StringTokenizer st;

        public MyScanner() {
            br = new BufferedReader(new InputStreamReader(System.in));
        }

        String next() {
```

```
            while (st == null || !st.hasMoreElements()) {
                try {
                    st = new StringTokenizer(br.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt() { return Integer.parseInt(next()); }
        long nextLong() { return Long.parseLong(next()); }
        double nextDouble() { return Double.parseDouble(next()); }

        String nextLine(){
            String str = "";
            try {
                str = br.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return str;
        }
    }
    //--------------------------------------------------------
```

5 Misc

5.1 LongestIncreasingSubsequence.cpp

```
inline VI longestIncreasingSubsequence(const vector<int> &a) { // O(n log k)
    int n = a.size(), lsize = 0;
    VI lval(n), lind(n), rec(n);
    for (int i = 0; i < n; i++) {
        int pos = lower_bound(lval.begin(), lval.begin() + lsize, a[i]) - lval.begin();
        lval[pos] = a[i]; lind[pos] = i;
        rec[i] = pos == 0 ? -1 : lind[pos-1];
        if (pos == lsize) lsize++;
    }
    // Recover the solution (return lsize and remove lind and rec if you only need its length)
    VI res(lsize);
    for (int i = lind[lsize-1]; i != -1; i = rec[i]) res[--lsize] = a[i];
    return res;
}
```

6 DataStructures

6.1 SparseTableRMQ.java

```
public class SparseTableRMQ {

    private final int a[], logTable[], rmq[][];

    public SparseTableRMQ(int[] a) {
        final int n = a.length;
        this.a = a;
        this.logTable = new int[n+1];
        for (int i = 2; i <= n; i++) logTable[i] = logTable[i>>1] + 1;
        this.rmq = new int[logTable[n]+1][n];
        for (int i = 0; i < n; i++) rmq[0][i] = i;
        for (int k = 1; (1<<k) < n; k++) {
            for (int i = 0; i+(1<<k) <= n; i++) {
                final int x = rmq[k-1][i], y = rmq[k-1][i+(1<<k-1)];
                rmq[k][i] = a[x] <= a[y] ? x : y;
            }
        }
    }

    public int minPos(int i, int j) { // Both inclusive
        final int k = logTable[j-i], x = rmq[k][i], y = rmq[k][j-(1<<k)+1];
        return a[x] <= a[y] ? x : y;
    }

    public int minVal(int i, int j) { // Both inclusive
        final int k = logTable[j-i];
        return Math.min(a[rmq[k][i]], a[rmq[k][j-(1<<k)+1]]);
    }

}
```

6.2 SegmentTreeRangeUpdate.java

```java
public class SegmentTreeRangeUpdate {
    public long[] leaf;
    public long[] update;
    public int origSize;
    public SegmentTreeRangeUpdate(int[] list)   {
        origSize = list.length;
        leaf = new long[4*list.length];
        update = new long[4*list.length];
        build(1,0,list.length-1,list);
    }
    public void build(int curr, int begin, int end, int[] list) {
        if(begin == end)
            leaf[curr] = list[begin];
        else {
            int mid = (begin+end)/2;
            build(2 * curr, begin, mid, list);
            build(2 * curr + 1, mid+1, end, list);
            leaf[curr] = leaf[2*curr] + leaf[2*curr+1];
        }
    }
    public void update(int begin, int end, int val) {
        update(1,0,origSize-1,begin,end,val);
    }
    public void update(int curr,  int tBegin, int tEnd, int begin, int end, int val)    {
        if(tBegin >= begin && tEnd <= end)
            update[curr] += val;
        else    {
            leaf[curr] += (Math.min(end,tEnd)-Math.max(begin,tBegin)+1) * val;
            int mid = (tBegin+tEnd)/2;
            if(mid >= begin && tBegin <= end)
                update(2*curr, tBegin, mid, begin, end, val);
            if(tEnd >= begin && mid+1 <= end)
                update(2*curr+1, mid+1, tEnd, begin, end, val);
        }
    }
    public long query(int begin, int end)   {
        return query(1,0,origSize-1,begin,end);
    }
    public long query(int curr, int tBegin, int tEnd, int begin, int end)   {
        if(tBegin >= begin && tEnd <= end)  {
            if(update[curr] != 0)   {
                leaf[curr] += (tEnd-tBegin+1) * update[curr];
                if(2*curr < update.length){
                    update[2*curr] += update[curr];
                    update[2*curr+1] += update[curr];
                }
                update[curr] = 0;
            }
            return leaf[curr];
        }
        else    {
            leaf[curr] += (tEnd-tBegin+1) * update[curr];
            if(2*curr < update.length){
                update[2*curr] += update[curr];
                update[2*curr+1] += update[curr];
            }
            update[curr] = 0;
            int mid = (tBegin+tEnd)/2;
            long ret = 0;
            if(mid >= begin && tBegin <= end)
                ret += query(2*curr, tBegin, mid, begin, end);
            if(tEnd >= begin && mid+1 <= end)
                ret += query(2*curr+1, mid+1, tEnd, begin, end);
            return ret;
        }
    }
}
```

6.3 SegmentTree.cpp

```cpp
template<typename T, typename Op>
class SegmentTree {
    Op op; vector<T> a, st; size_t n;
    T update(int p, int l, int r, int lo, int hi) {
        return l == r ? a[l] : hi < l || lo > r ? st[p] : st[p] = op(update(2*p, l, (l+r)/2, lo, hi),
            update(2*p+1, (l+r)/2+1, r, lo, hi));
    }
    T query(int p, int l, int r, int lo, int hi) {
```

```
        if (l == r) return a[l];
        if (lo <= l && hi >= r) return st[p];
        if (!(hi < l || lo > (l+r)/2)) {
            T left = query(2*p, l, (l+r)/2, lo, hi);
            return !(hi < (l+r)/2+1 || lo > r) ? op(left, query(2*p+1, (l+r)/2+1, r, lo, hi)) : left;
        } else {
            return query(2*p+1, (l+r)/2+1, r, lo, hi);
        }
    }
public:
    SegmentTree() : n(0), st(0) {}
    template <typename InputIterator>
    SegmentTree(InputIterator first, InputIterator last, const Op &op = Op()) : n(last-first), a(first,
         last), st(2*(last-first)), op(op) {
        update(0, n-1);
    }
    T get(int i) { return a[i]; }
    T query(int lo, int hi) { return query(1, 0, n-1, lo, hi); } // Both inclusive
    void set(int i, int v) { a[i] = v; update(i); }
    void update(int i) { update(i, i); }
    void update(int lo, int hi) { update(1, 0, n-1, lo, hi); }  // Both inclusive
};

template<typename T, typename Compare>
class MinimumIndexSegmentTree {
    struct Op {
        Compare cmp; vector<T> *a;
        int operator()(int i, int j) { return cmp((*a)[i], (*a)[j]) ? i : j; }
    }; vector<T> a; SegmentTree<size_t, Op> st;
public:
    template <typename InputIterator>
    MinimumIndexSegmentTree(InputIterator first, InputIterator last) : a(first, last) {
        VI aux(last-first);
        for (int i = 0; i < aux.size(); i++) aux[i] = i;
        Op op; op.a = &a;
        st = SegmentTree<size_t, Op>(aux.begin(), aux.end(), op);
    }
    T get(int i) { return a[i]; }
    int query(int lo, int hi) { return st.query(lo, hi); } // Both inclusive
    void set(int i, int v) { a[i] = v; st.update(i); }
};
typedef SegmentTree<int, plus<int> > RSQ;
typedef MinimumIndexSegmentTree<int, less<int> > RMQ;
```

6.4 SparseTable.cpp

```
struct RMQ {
    VI A, logtable;
    VVI spt; // SpT[i][j] = RMQ of range starting at i and length (2^j)
    RMQ(int N, VI data) : A(data), logtable(N + 1) {
        for (int i = 2; i <= N; i++)
            logtable[i] = logtable[i >> 1] + 1;
        spt = VVI(logtable[N] + 1, VI(N));
        for (int i = 0; i < N; i++)
            spt[0][i] = i;
        for (int j = 1; (1 << j) <= N; j++)
            for (int i = 0; i + (1 << j) - 1 < N; i++)
                if (A[spt[j - 1][i]] < A[spt[j - 1][i + (1 << (j - 1))]])
                    spt[j][i] = spt[j - 1][i];
                else
                    spt[j][i] = spt[j - 1][i + (1 << (j - 1))];
    }
    int query(int i, int j) {
        int k = logtable[j-i+1]; // 2^k <= (j-i+1)
        if (A[spt[k][i]] <= A[spt[k][j - (1 << k) + 1]])
            return spt[k][i];
        else
            return spt[k][j - (1 << k) + 1];
    }
};
```

6.5 UnionFindDisjointSet.cpp

```
class UnionFindDisjointSet {
    VI p, setSize; int numSets;
public:
    explicit UnionFindDisjointSet(int n) : p(n,-1), setSize(n,1), numSets(n) {}
    int findSet(int i) { return (p[i] < 0) ? i : (p[i] = findSet(p[i])); }
    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
    int numDisjointSets() { return numSets; }
    int sizeOfSet(int i) { return setSize[findSet(i)]; }
```

```cpp
    void unionSet(int i, int j) {
        if (!isSameSet(i, j)) {
            numSets--;
            int x = findSet(i), y = findSet(j);
            if (p[x] < p[y]) { // rank[x] > rank[y]
                p[y] = x;
                setSize[x] += setSize[y];
            } else {
                p[x] = y;
                setSize[y] += setSize[x];
                if (p[x] == p[y]) p[y]--;
            }
        }
    }
};
```

6.6 BinaryIndexedTree.cpp

```cpp
#define LSOne(S) (S & (-S))

class FenwickTree { // Queries for dynamic RSQ in O(log n), elements numbered from 1 to n
private:
  VI ft;
public:
  FenwickTree(int n) : ft(n+1, 0) {} // initialization: n + 1 zeroes, ignore index 0
  int rsq(int b) { int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b]; return sum; } // RSQ(1, b)
  int rsq(int a, int b) { return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); } // RSQ(a, b)
  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) { for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
```

7 Graphs

7.1 MinCostMaxFlow.cpp

```cpp
// Implementation of min cost max flow algorithm using adjacency
// matrix.  This implementation keeps track of forward and reverse
// edges separately (so you can set cap[i][j] != cap[j][i]).
// For a regular max flow, set all edge costs to 0.
// Note that negative cost values are not allowed.
// INPUT: - graph, constructed using AddEdge()
//        - source
//        - sink
// OUTPUT: - (maximum flow value, minimum cost value)
//         - To obtain the actual flow, look at positive values only.
#include <cmath>

const L INF = 1LL << 60;

struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;

    MinCostMaxFlow(int N) :
        N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
        found(N), dist(N), pi(N), width(N), dad(N) {}

    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;
```

```

        while (s != -1) {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }

        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }

    pair<L, L> GetMaxFlow(int s, int t) {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t)) {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first) {
                if (dad[x].second == 1) {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                } else {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
        }
        return make_pair(totflow, totcost);
    }
};
```

7.2 StronglyConnectedComponents.cpp

```
VII adj, scc;
VI num, low;
VI S;
VB visited, currentSCC;
int nodeCount, numSCC;

/* num[i] = orden en el que se visita por primera vez el nodo i */
/* low[i] = minimo num alcanzable desde el nodo i y desde sus hijos en la busqueda */
/* currentSCC[i] <=> el nodo i forma parte del SCC que se est  explorando */
/* S = Pila que guarda los nodos seg n el orden en que se exploran */
/* Los nodos que forman cada componente quedan en scc */
/* Inicializar 'nodeCount' y 'numSCC' a 0 antes de llamar a la funci n */
void dfs(int u) {
    num[u] = low[u] = nodeCount++;
    S.push_back(u);
    visited[u] = currentSCC[u] = true;
    int v;
    for (int i = 0; i < (int)adj[u].size(); ++i) {
        v = adj[u][i];
        if (!visited[v])
            dfs(v);
        if (currentSCC[v]) /* si es parte de la misma componente que u */
            low[u] = min(low[u], low[v]); /* desde u alcanzo lo mismo que desde v */
    }
    if (low[u] == num[u]) { /* si u es ra z de una SCC */
        scc.push_back(VI());
        do { /* El SCC lo forman los nodos en la pila hasta alcanzar u */
            v = S.back(); S.pop_back(); currentSCC[v] = 0;
            scc[numSCC].push_back(v);
        } while (u != v);
        numSCC++; /* Si solo se desea el n mero de SCCs, el vector scc sobra */
    }
}
/* Ejemplo de main, donde N es el n mero de nodos*/
// adj = VII(N, VI()); scc = VII(); num = VI(N); low = VI(N);
// S = VI(); visited = VB(N);
// currentSCC = VB(N); nodeCount = numSCC = 0;
/* Rellenar la lista de adyacencia */
// for (int i = 0; i < N; ++i)
//   if (!visited[i])
//       dfs(i);
```

7.3 MaxBipartiteMatching.cpp

```cpp
// Maximum Cardinality Bipartite Matching
//   -- Representaci n con matriz de adyacencia
//   -- Pasar a la constructora el  tama o  de las dos particiones (izquierda y derecha)
//   -- A adir ejes con addEdge(origen, destino)
// La funci n getMatching devuelve la cardinalidad del matching, y rellena los vectores mr y mc:
//    mr[i] = nodo asignado al nodo izquierdo i
//    mc[j] = nodo asignado al nodo derecho j
// Tambi n es  til para:
//    Maximum Independent Set = |V| - MCBM
//    Minimum Vertex Cover = MCBM
class MCBM {
    int nLeft, nRight;
    VVI mat_adj;
    VI mr, mc;
    bool FindMatch(int i, const VVI &mat_adj, VI &seen) {
        for (int j = 0; j < nRight; j++) {
            if (mat_adj[i][j] && !seen[j]) {
                seen[j] = true;
                if (mc[j] < 0 || FindMatch(mc[j], mat_adj, seen)) {
                    mr[i] = j; mc[j] = i; return true;
                }
            }
        }
        return false;
    }
public:
    MCBM(int NLeft, int NRight) : nLeft(NLeft), nRight(NRight), mr(NLeft, -1), mc(NRight, -1) {
        mat_adj = VVI(NLeft, VI(NRight));
    }
    void addEdge(int u, int v) { mat_adj[u][v] = 1; }
    int getMatching() {
        int ct = 0;
        for (int i = 0; i < nLeft; i++) {
            VI seen(nRight);
            if (FindMatch(i, mat_adj, seen)) ct++;
        }
        return ct;
    }
    VI getLeftMatches() { return mr; };
    VI getRightMatches() { return mc; };
};
```

7.4 ArticulationPoints.cpp

```cpp
VVI adj;
VI num, low;
VI parent;
vb visited;
vb artPoint;
int nodeCount, root, rootChildren;

/* num[i] = orden en el que se visita por primera vez el nodo i */
/* low[i] = m nimo num alcanzable desde el nodo i y desde sus hijos en la b squeda */
/* Establecer 'root' al nodo ra z de la b squeda, y 'rootChildren' y 'nodeCount' a 0 antes de llamar a dfs(root) */
void dfs(int u) {
    num[u] = low[u] = nodeCount++;
    visited[u] = true;
    for (int i = 0; i < (int)adj[u].size(); ++i) {
        int v = adj[u][i];
        if (!visited[v]) {
            parent[v] = u;
            if (u == root) rootChildren++;
            dfs(v);
            if (low[v] >= num[u]) /* si desde v no puedo alcanzar nada m s arriba de u */
                artPoint[u] = true;
            if (low[v] > num[u]) /* si desde v no puedo alcanzar ni u */
                cout << "(u,v) is a bridge" << endl;
            low[u] = min(low[u], low[v]); /* desde u alcanzo lo mismo que desde v */
        } else if (v != parent[u])
            low[u] = min(low[u], num[v]); /* desde u alcanzo v */
    }
}

/* Despu s de la llamada: 'root' es un punto de articulaci n sii 'rootChildren' > 1 */
artPoint[root] = (rootChildren > 1);
```

7.5 MinCostBipartiteMatching.cpp

```cpp
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs. In practice, it solves 1000x1000 problems in around 1
// second. Note that both partitions must be of equal size!!
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
#include <cmath>

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());
    // construct dual feasible solution
    VD u(n);
    VD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++)
            u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++)
            v[j] = min(v[j], cost[i][j] - u[i]);
    }
    // construct primal solution satisfying complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }
        }
    }

    VD dist(n);
    VI dad(n);
    VI seen(n);
    // repeat until primal solution is feasible
    while (mated < n) {
        // find an unmatched left node
        int s = 0;
        while (Lmate[s] != -1) s++;
        // initialize Dijkstra
        fill(dad.begin(), dad.end(), -1);
        fill(seen.begin(), seen.end(), 0);
        for (int k = 0; k < n; k++)
            dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        while (true) {
            // find closest
            j = -1;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;
            // termination condition
            if (Rmate[j] == -1) break;
            // relax neighbors
            const int i = Rmate[j];
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
```

```
                    }
                }
            }
            // update dual variables
            for (int k = 0; k < n; k++) {
                if (k == j || !seen[k]) continue;
                const int i = Rmate[k];
                v[k] += dist[k] - dist[j];
                u[i] -= dist[k] - dist[j];
            }
            u[s] += dist[j];
            // augment along path
            while (dad[j] >= 0) {
                const int d = dad[j];
                Rmate[j] = Rmate[d];
                Lmate[Rmate[j]] = j;
                j = d;
            }
            Rmate[j] = s;
            Lmate[s] = j;
            mated++;
        }

        double value = 0;
        for (int i = 0; i < n; i++)
            value += cost[i][Lmate[i]];

        return value;
}
```

7.6 LowestCommonAncestor.cpp

```
// Lowest Common Ancestor with adjacency list
// Requires an RMQ implementation
//  Par[i] = parent of node i in the DFS, root is its own parent
//  E[i] = i-th node visited in the DFS (Euler tour)
//  L[i] = levels of the i-th node visited in the DFS (Euler tour)
//  H[i] = index of the first occurrence of node i in E
struct LCA {
    int idx;
    VVI adj;
    VI Par, E, L, H;
    RMQ * rmq;

    LCA(int N, VVI adjlist) :
      idx(0), adj(adjlist), Par(N, -1), E(2*N-1), L(2*N-1), H(N, -1) {
        dfs(0, 0, 0); // We fix the root at index 0
        rmq = new RMQ(2*N-1, L);
    }

    void dfs(int cur, int depth, int parent) {
        Par[cur] = parent;
        H[cur] = idx;
        E[idx] = cur;
        L[idx++] = depth;
        for (int i = 0; i < (int) adj[cur].size(); i++) {
            if (Par[adj[cur][i]] == -1) {
                dfs(adj[cur][i], depth + 1, cur);
                E[idx] = cur;
                L[idx++] = depth;
            }
        }
    }

    int depth(int u) { return L[H[u]]; } // Depth of u
    int parent(int u) { return Par[u]; } // Parent of u
    int find(int u, int v) { // LCA(u, v)
        if (H[u] > H[v]) swap(u, v);
        return E[rmq->query(H[u], H[v])];
    }
};
```

7.7 MaxFlow.cpp

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
// INPUT: - graph, constructed using AddEdge()
//         - source
//         - sink
// OUTPUT: - maximum flow value
//          - To obtain the actual flow values, look at all edges with
```

```cpp
//             capacity > 0 (zero capacity edges are residual edges).
#include <cmath>

const int INF = 2000000000;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
    int N;
    vector<vector<Edge> > G;
    vector<Edge *> dad;
    VI Q;

    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    long long BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;

        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++) {
                Edge &e = G[x][i];
                if (!dad[e.to] && e.cap - e.flow > 0) {
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;
                }
            }
        }
        if (!dad[t]) return 0;

        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++) {
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                if (!e) { amt = 0; break; }
                amt = min(amt, e->cap - e->flow);
            }
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;
            }
        totflow += amt;
        }
        return totflow;
    }

    long long GetMaxFlow(int s, int t) {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t))
            totflow += flow;
        return totflow;
    }
};
```

7.8 SPFA.cpp

```cpp
// Shortest Path Faster Algorithm
// SSSP adjacency-list implementation that handles negative weight cycles.
// The function returns true if such a cycle is detected (i.e., it can be reached from s).
// If not, dist[i] = distance from source node s to node i.
// Worst-case complexity: O(VE), in practice better than Bellman-Ford, but not than Dijkstra.
#define INF 1 << 30

bool spfa(int s, const vector<VPII>& adj, VI& dist) {
    int N = adj.size(), u, i;
```

```
    queue<int> cola;
    VI encolado(N), veces(N);

    dist = VI(N, INF);
    dist[s] = 0;

    cola.push(s);
    encolado[s] = veces[s] = 1;
    while (!cola.empty()) {
        u = cola.front();
        cola.pop();
        encolado[u] = 0;
        for (i = 0; i < (int) adj[u].size(); ++i) {
            PII p = adj[u][i];
            if (dist[u] + p.second < dist[p.first]) {
                dist[p.first] = dist[u] + p.second;
                if (!encolado[p.first]) {
                    cola.push(p.first);
                    encolado[p.first] = 1;
                    veces[p.first]++;
                    // Tratar ciclo negativo si se desea
                    if (veces[p.first] == N) return true;
                }
            }
        }
    }
    return false;
}
```

7.9 Kruskal.cpp

```
// COMPLEXITY: O(E log E)
#include "UnionFindDisjointSet.cpp"
//Returns the cost of the msp
int Kruskal(vector<pair<int, PII>> edgeList, int graphSize) {
    sort(edgeList.begin(), edgeList.end());

    int mst_cost = 0;
    UnionFindDisjointSet UF(graphSize);

    for (int i = 0; i < edgeList.size(); ++i) {
        pair<int, PII> front = edgeList[i];
        if (!UF.isSameSet(front.second.first, front.second.second)) {
            mst_cost += front.first;
            UF.unionSet(front.second.first, front.second.second);
        }
    }
    return mst_cost;
}
```

7.10 Dijkstra.cpp

```
// COMPLEXITY: O((V+E)log V) (V,E < 300K)
const int INF = 1e9; //Use long long if something bigger is needed
VVI graph, weight;
VI dist;
void dijkstra(int source) {
    dist = VI(graph.size(), INF);
    dist[source] = 0;
    priority_queue<pii, vector<pii>, greater<pii> > pq;
    pq.push(pii(0, source));
    while(!pq.empty()) {
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();
        if (d > dist[u])    continue;
        for (int i = 0; i < (int)graph[u].size(); ++i) {
            int v = graph[u][i];
            int w = weight[u][i];
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w ;
                pq.push(pii(dist[v], v));
            }
        }
    }
}
```

7.11 FloydWarshall.cpp

```
// COMPLEXITY: O(V^3) (V < 400)
// adj_mat = matriz de adyacencia del grafo
// adj_mat[i][j] = INF si no hay arista
```

```
// adj_mat[i][i] = 0
// V = cantidad de nodos
// Si despues de todo la diagonal tiene un valor menor que cero, tiene ciclos negativos
void FloydWarshall (VVI &adj_mat) {
    int V = adj_mat.size();
    for (int k = 0; k < V; ++k)
        for (int i = 0; i < V; ++i)
            for (int j = 0; j < V; ++j)
                adj_mat[i][j] = min(adj_mat[i][j], adj_mat[i][k] + adj_mat[k][j]);
}
```

8 Geometry

8.1 ConvexHull.cpp

```
// Running time: O(n log n)
// INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull, counterclockwise, starting
//            with bottommost/leftmost point
#include <cstdio>
#include <cmath>

// #define REMOVE_REDUNDANT // To eliminate redundant points from hull

typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}

int main() {
    PT val[] = {PT(0, 0), PT(1, 1), PT(2, 2), PT(-1, 0)};
    vector<PT> puntuak;
    for (int i = 0; i < 4; i++)
        puntuak.push_back(val[i]);
    ConvexHull(puntuak);
    for (int i = 0; i < puntuak.size(); i++)
```

```
        cout << puntuak[i].x << "␣" << puntuak[i].y << endl;
    return 0;
}
```

8.2 GeometryMiscellaneous.cpp

```
#include <cmath>

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}
// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
```

```
}
// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
```

```
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}
double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}
PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}
// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```