Ho Chi Minh City International University
ACM Team Cheatsheets (2017)

Template 1
Maths 1
a. Prime Sieve 1
b. Fast Exponential 2
c. Prime Modulus 2
d. Extened GCD 2
e. Inverse Multiplicative 2
f. LCM 2
g. Combination 2
h. Catalan numbers 2
i. Matrix Multiplication and Exponential 2
Dynamic Programming 2
a. Maximum Contiguous Subarray 2
b. Longest Common Subsequence 3
c. Longest Increasing Subsequence 3
d. Classic Subset Sum 3
e. Knapsack Problems 3
i. Classic Knapsack (Unbounded Knapsack) 3
ii. Exactly V value with minimum item. 4
iii. Largest weight of all items that <= W 4
iv. Exactly W weight 4
Geometry 4
a. Convex hull 4
b. Area and centroid 5
Graph 5
a. Dijktra 5
b. Bellman-Ford 5
c. Floyd 6
d. Euler Path/Tour 6
e. Hamiltonian Path 6
f. SCC 6
v. Korasaju 6
vi. Tarjan 7
g. Maximum Flow (Dinic) 7
h. Minimum Spanning Tree (Kruskal) 8
i. Aticulation Point and Bridge 8
j. Topological Sort 8
String Processing 9
a. KMP 9
b. Trie 9
c. Hash 9
d. Z – Algorithm 9
e. Lexicographically Minimal String Rotation 9
Data Structures 10
a. Segment Tree 10
i. Classical 10
ii. Update an interval to a new value 10
iii. Merge-sort tree 10
f. Binary Indexed Tree 11
g. Disjoint Set Union 11
h. Lowest Common Ancestor 11
i. LCA – Binary lifting 11
j. Ordered set/multiset 11
Miscellaneous 11
a. Custom Comparing Class 11
k. Useful Calculus 12
l. Useful Geometry 12
m. Iterate through all permutation 12
n. Generate all subsets 12
o. Josephus problem 12
p. Java theme and syntax 12
q. Geometry Template 12
i. Points And Lines 12
ii. Circles 14
iii. Triangles 15
iv. Polygons 16

Template

```
#include <bits/stdc++.h>
#define bug(x) cout << #x << " = " << x << endl;
#define fr(a) freopen(a,"r",stdin);
#define fw(a) freopen(a,"w",stdout);
#define tc() int tc;cin >> tc; for(int
_tc=1;_tc<=tc;_tc++)
#define up(i,l,r) for (int i=l;i<=r;i++)
#define down(i,r,l) for (int i=r;i>=l;i--)
#define rep(i,l,r) for (int i=l;i<r;i++)
#define all(a) a.begin(),a.end()
#define reset(a) memset(a,0,sizeof(a))
#define pb push_back
#define mp make_pair
#define ins insert
#define fi first
#define se second
using namespace std;
typedef long long int ll;
typedef pair<int,int> pii;
typedef vector<int> vi;
/*******************************************/
void printArr(int a[], int l, int r) {
    up(i,l,r) cout << a[i] << " \n"[i==r];
}

void printMtx(int a[][1010],int n, int m) {
    up(i,1,n) up(j,1,m) cout << a[i][j] << "
\n"[j==m];
}
/*********************************************/

int main() {
    ios_base::sync_with_stdio(0);
}
```

Maths

a. Prime Sieve

Usage: Create list of primes smaller than n.

Complexity: O(n).

In this implementation:

- prime: list of primes

- if is_composite[i] = 0 → i is prime

```
std::vector <int> prime;
bool is_composite[MAXN];

void sieve (int n) {
    std::fill (is_composite, is_composite + n,
false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) prime.push_back (i);
        for (int j = 0; j < prime.size () && i *
prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) break;
        }
    }
}
```

b. Fast Exponential

```
ll expMod(ll b,ll p,ll m) {
    if (p==0) return 1;
    ll t = expMod(b,p/2,m);
    if (p%2==0) return (t*t)%m;
    return (((b%m)*t)%m)*t)%m;
}
Fast Multiplication
function mulmod(A, B, MOD) {
    RES = 0;
    while (B > 0) {
        if (B is odd) RES = (RES + A) % MOD;
        A = (A * 2) % MOD;
        B = B / 2;
    }
    return RES;
}
```

c. Prime Modulus

(A / B) % MOD = ((A % MOD) * (B^{MOD-2} % MOD)) % MOD.

Conditions: B and MOD are coprimes, MOD is a prime number.

d. Extened GCD

```
int exgcd(int x, int y, int &a, int &b) {
    /// extended gcd, ax + by = g
    int a0 = 1, a1 = 0, b0 = 0, b1 = 1;
    while (y != 0) {
        a0 -= x / y * a1;
        swap(a0, a1);
        b0 -= x / y * b1;
        swap(b0, b1);
        x %= y;
        swap(x, y);
    }
    if (x < 0) a0 = -a0, b0 = -b0, x = -x;
    a = a0, b = b0;
    return x;
}
```

e. Inverse Multiplicative

```
int inverse(int x, int mod) {
    /// multiplicative inverse
    int a = 0, b = 0;
    if (exgcd(x, mod, a, b) != 1) return -1;
    /// case 1: x & mod are co-prime
    return (a % mod + mod) % mod;
    /// case 2: mod is prime
    return fastPow(x, mod - 2, mod);
}

void inverse_all(int mod) {
    /// O(n), mod is prime
    inv[0] = inv[1] = 1;
    for (int i = 2; i < n; ++i) {
        inv[i] = 1ll * inv[mod % i] * (mod - mod /
i) % mod;
        /// overflows?
    }
}
```

f. LCM

LCM(A, B) = (A * B) / GCD(A, B).

g. Combination

```
void Divbygcd(ll& a,ll& b) {
    ll g=__gcd(a,b);
    a/=g;
    b/=g;
}
ll combination(ll n, ll k) {
    ll
numerator=1,denominator=1,toMul,toDiv,i;
    if (k>n/2) k=n-k; /* use smaller k */
    for (i=k; i; i--) {
        toMul=n-k+i;
        toDiv=i;
        Divbygcd(toMul,toDiv); /* always divide
before multiply */
        Divbygcd(numerator,toDiv);
        Divbygcd(toMul,denominator);
        numerator*=toMul;
        denominator*=toDiv;
    }
    return numerator/denominator;
}
```

h. Catalan numbers

```
ll catalanDP(int n) {
    ll catalan[n + 1];
    catalan[0] = catalan[1] = 1;
    up(i,2,n) {
        catalan[i] = 0;
        rep(j,0,i)
        catalan[i] += catalan[j] * catalan[i
- j - 1];
    }
    return catalan[n];
}
```

i. Matrix Multiplication and Exponential

$$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix} * \begin{pmatrix}F_i\\F_{i-1}\end{pmatrix} = \begin{pmatrix}F_{i+1}\\F_i\end{pmatrix} \to \begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}^T * \begin{pmatrix}F_1\\F_0\end{pmatrix} = \begin{pmatrix}F_{T+1}\\F_T\end{pmatrix}$$

Given a resursive relation: $a_n = \alpha * a_{n-1} + \beta * a_{n-2}$

We'll have a matrix A = $\begin{pmatrix}\alpha & \beta\\1 & 0\end{pmatrix}$ such that $A * \begin{pmatrix}a_n\\a_{n-1}\end{pmatrix} = \begin{pmatrix}a_{n+1}\\a_n\end{pmatrix}$

```
MatrixMul:
    for i:=1 to M do
    for j:=1 to P do
        begin
        C[i,j]:=0;
        for k:=1 to N do
            C[i,j]:=C[i,j]+A[i,k] * B[k,j];
        end;

void powMatrix(ll p) { // p is  power
    if (p <= 1) return;
    powMatrix(p/2);
    MulMatrix(a,a);  // a is matrix
    if (p%2 == 0) return;
    else MulMatrix(a,ori);
// ori is the original matrix
}
```

Dynamic Programming

a. Maximum Contiguous Subarray

Usage: Find the contiguous subarray which have maximum sum. Works with negative values.

Complexity: O(n).

In this implementation:

- max_so_far: the result sum.
- start, end: start, end index of the result subarray.
- The array is 0-base indexed

```
int max_so_far = INT_MIN, max_ending_here = 0,
start =0, end = 0, s=0;
int maxSubArraySum(int a[], int size) {
    for (int i=0; i< size; i++ ) {
        max_ending_here += a[i];
        if (max_so_far < max_ending_here) {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }
        if (max_ending_here < 0) {
            max_ending_here = 0;
            s = i+1;
        }
    }
}
```

b. Longest Common Subsequence

Complexity: O(n*m)

In this implementation:

- Call traceback: Track_Back(0,0,s1.size(),s2.size());

```
int m[2][10000];
string s1, s2;

int LCS() {
    int i, j, M, N, ii;
    M = s1.length();
    N = s2.length();

    for (i = M; i >= 0; i--) {
        ii = i & 1;
        for (j = N; j >= 0; j--) {
            if (i == M || j == N) {
                m[ii][j] = 0;
                continue;
            }
            if (s1[i] == s2[j]) {
                m[ii][j] = 1 + m[1 - ii][j + 1];
            } else {
                m[ii][j] = max(m[ii][j+1], m[1-
ii][j]);
            }
        }
    }
    return m[0][0];
}
void Trace_Back(int i, int j, int M, int N) {
    if (i == M || j == N) {
        return;
    } else if (t[i][j] == 1) {
        cout << s1[i] << " ";
        Trace_Back(i + 1, j + 1);
    } else if (t[i][j] == 3) {
        Trace_Back(i + 1, j);
    } else {
        Trace_Back(i, j + 1);
    }
}
```

c. Longest Increasing Subsequence

Complexity: O(nlog(n))

```
int a[30000];
set<int> st;
set<int>::iterator it;
int main () {
    // Input
    int N, t;
    cin >> N;
    for (int i = 0; i < N; ++i) {
        cin >> t;
        a[i] = t;
    }
    //Thuat toan
    st.clear();
    for(int i=0; i<N; i++) {
        it = st.lower_bound(a[i]);
        if (it != st.end()) st.erase(it);
        st.insert(a[i]);
    }
    //In ket qua
    cout<<st.size()<<endl;
    for (it = st.begin(); it != st.end(); ++it) {
        cout << *it;
    }
    cout << endl;
    return 0;
}
```

d. Classic Subset Sum

Usage: Cho 1 chuỗi n số. Kiểm tra xem nếu có subset có tổng bằng S cho trước.

➔ Partition Problem: Determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same. → Check for S = sum/2

In this implementation:

- N is number of elements
- M is maximum value the set can get
- S is the specific value for counting subset

```
int a[1000]; // store value of set
int m[1000]; // store number of subset
int SubsetSum(int N, int M, int S) {
    int i, j;
    for (i = 0; i < M + 10; i++) m[i] = 0;
    m[0] = 1;

    for (i = 0; i < N; i++)
        for (j = M; j >= a[i]; j--)
            m[j] += m[j - a[i]];
}
int main() {
    int N, t, M = 0, S = 0;
    cin >> N;
    for (int i = 0; i < N; ++i) {
        cin >> t;
        a[i] = t;
        M += t;
    }
    cin >> S;
    SubsetSum(N, M, S);

    for (int i = 0; i < M; ++i) {
        cout << m[i] << " ";
    }
    cout << endl;

    cout << m[S] << endl;
    return 0;
}
```

e. Knapsack Problems

i. Classic Knapsack (Unbounded Knapsack)

Có N món hàng. Mỗi món hàng có w khối lượng và v giá trị. Có 1 túi xách đựng được tối đa **W** khối lượng. Tìm cách lấy nhiều món hàng nhất có thể với tổng giá trị lớn nhất.

```
int w[100]; // weight
int v[100]; // value
int m[100][10000]; // dynacmic table arr
int t[100][10000]; // trace array
```

```
void Trace_Back(int i, int j) {
    if (i == 0 || j == 0) return;
    else if (t[i][j] == 1) {
        Trace_Back(i - 1, j - w[i]);
        result << i << " ";
    } else {
        Trace_Back(i - 1, j);
    }
}
void Knapsack_Algorithm(int N, int W) {
  int i, j;
  for (i = 1; i <= W; ++i) m[0][i] = t[0][i] = 0;

  for (i = 1; i <= N; ++i) {
    for (j = 1; j <= W; ++j) {
       if (j >= w[i]) {
         m[i][j] = max(m[i-1][j],
                       m[i-1][j-w[i]] + v[i]);
         t[i][j] = (m[i][j]==m[i-1][j]) ? 0:1;
       } else {
          m[i][j] = m[i - 1][j];
          t[i][j] = 0;
       }
    }
  }
  Trace_Back(N, W);
}
```

ii. **Exactly V value with minimum item.**

Có N tờ tiền, mỗi tờ tiền có v giá trị. Tìm cách lấy số tờ tiền ít nhất với tổng giá trị = V cho trước.

Note: Bài này dùng Trace_Back riêng để xuất ra giá trị của biến được chọn

```
void Trace_Back(int i, int j) {
    if (i == 0 || j == 0) return;
    else if (t[i][j] == 1) {
        Trace_Back(i - 1, j - v[i]);
        cout << v[i] << " ";
    } else {
        Trace_Back(i - 1, j);
    }
}
void Knapsack_Algorithm(int N, int V) {
  int i, j, temp = 0;
  sort(v, v+N);
  for (i = 1; i <= N; ++i) temp += v[i];
  if (temp < V) {
      cout << "No Solution"; return;
  }
  for (i = 1; i <= V; ++i) m[0][i] =t[0][i]= 0;

  for (i = 1; i <= N; ++i) {
      for (j = 1; j <= V; ++j) {
          if (j >= v[i]) {
            m[i][j] = max(m[i-1][j],
                          m[i-1][j-v[i]] + v[i]);
            t[i][j] = m[i-1][j] == m[i][j] ? 0 : 1
;
            if(m[i][j] == j) t[i][j] = 1;
          } else m[i][j] = m[i-1][j];
      }
  }

  if (m[N][V] == V) Trace_Back(N, V);
  else cout << "No Solution";
}
```

iii. **Largest weight of all items that <= W**

Có N cục đá, mỗi cục có w khối lượng.

Tìm cách lấy sao cho số lượng đá là nhiều nhất.

```
int Trace_Back(int i, int j, int count) {
    if (i == 0 || j == 0) return count;
    else if (t[i][j] == 1) {
        count ++;
        Trace_Back(i - 1, j - v[i], count);
            //cout << v[i] << " ";
    } else {
        Trace_Back(i - 1, j, count);
    }
}
int Knapsack_Algorithm(int N, int W) {
  int i, j, temp = 0;
  for (i = 1; i <= N; ++i) temp += w[i];
  if (temp < W) {
      return 0;
  }
  for (i = 1; i <= W; ++i) m[0][i] =t[0][i]= 0;

  for (i = 1; i <= N; ++i) {
      for (j = 1; j <= W; ++j) {
          if (j >= w[i]) {
            m[i][j] = max(m[i-1][j],
                          m[i-1][j-w[i]] + w[i]);
        t[i][j] = m[i-1][j] == m[i][j] ? 0 : 1 ;
          } else m[i][j] = m[i-1][j];
      }
  }

  if (m[N][W] == W) { return Trace_Back(N, W, 0);
}
  else return 0;
}
```

iv. **Exactly W weight**

Kiểm tra xem có thể bỏ chính xác maxWeight khối lượng cục đá vào túi ko?

W ở đây là total weight của N cục đá

```
int Knapsack_Algorithm(int N, int W) {
  int i, j;
  for (i = 1; i <= W; ++i) m[0][i] = INF;

  for (i = 1; i <= N; ++i) {
    for (j = 1; j <= W; ++j) {
        if (j >= w[i]) {
            m[i][j] = min(m[i-1][j],
                          m[i-1][j-w[i]] + 1);
        } else {
            m[i][j] = m[i - 1][j];
        }
     }
  }
  if (m[N][maxWeight] == INF) return 0;
  return m[N][maxWeight];
}
```

Geometry

a. Convex hull

Usage: Compute the 2D convex hull of a set of points using the monotone chain algorithm.

Complexity: O(n log n).

In this implementation:

- If REMOVE_REDUNDANT is #define-ed, redundant points from the hull is eliminated.
- Input: a vector of points under the form of a vector of pair<double, double>.
- Output: a vector of points in the convex hull, counterclockwise.

```
#define REMOVE_REDUNDANT

typedef double T;
typedef pair<T,T> PT;
typedef vector<PT> VPT;
const double EPS = 1e-7;

T det (const PT &a, const PT &b) {
```

```
    return a.first * b.second - a.second *
b.first;
}
T area2 (const PT &a, const PT &b, const PT
&c) {
    return det(a,b) + det(b,c) + det(c,a);
}

#ifdef REMOVE_REDUNDANT
// return true if point b is between points a and
c
bool between (const PT &a, const PT &b, const
PT &c) {
    return (fabs(area2(a,b,c)) < EPS &&
            (a.first - b.first) * (c.first -
b.first) <= 0 &&
            (a.second - b.second) * (c.second
- b.second) <= 0);
}
#endif

void convex_hull (VPT &pts) {
    sort (pts.begin(), pts.end());
    pts.erase (unique (pts.begin(),
pts.end()), pts.end());
    VPT 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

}
```

b. Area and centroid

Usage: Compute the area or centroid of a polygon, assuming the coordinates are listed in a clockwise or counterclockwise fashion.

(Centroid: "center of gravity" or "center of mass".)

Complexity: O(n).

In this implementation:

- Input: list of x[] and y[] coordinates.
- Output: (signed) area or centroid.

```
typedef pair<double,double> PD;
double ComputeSignedArea (const VD &x, const VD
&y) {
    double area = 0;
    for (int i = 0; i < x.size(); i++) {
        int j = (i+1) % x.size();
        area += x[i]*y[j] - x[j]*y[i];
    }
    return area / 2.0;
}
double ComputeArea (const VD &x, const VD &y) {
    return fabs (ComputeSignedArea (x, y));
}

PD ComputeCentroid (const VD &x, const VD &y) {
    double cx = 0, cy = 0;
    double scale = 6.0 * ComputeSignedArea (x, y);
    for (int i = 0; i < x.size(); i++) {
        int j = (i+1) % x.size();
        cx += (x[i]+x[j])*(x[i]*y[j]-x[j]*y[i]);
        cy += (y[i]+y[j])*(x[i]*y[j]-x[j]*y[i]);
    }
    return make_pair (cx/scale, cy/scale);
}
```

Graph

a. Dijktra

Usage: Find shortest cost from node START to all others

Complexity: O(nlog(n))

In this implementation:

- Nodes are 1-base indexed
- a[u]: pair array contains linking node with node u in form of {cost,index}
- d[i]: result array, min distance from node START

```
vector <ii> a[2309];
int n, m;
int d[2309];

void dijkstra(int START) {
    priority_queue <pii, vector<pii>, greater<pii>
> pq;
    int i, u, v, du, uv;

    up(i,1,n) d[i] = INT_MAX;
    d[START] = 0;
    pq.push({0, START});

    while (pq.size()) {
        u=pq.top().se;
        du=pq.top().fi;
        pq.pop();
        if (du!=d[u]) continue;

        rep(i,0,a[u].size()) {
            v=a[u][i].se;
            uv=a[u][i].fi;
            if (d[v]>du+uv) {
                d[v]=du+uv;
                pq.push({d[v], v});
            }
        }
    }
}
```

b. Bellman-Ford

Usage: Find shortest cost from node START to all others, can handle negative edges and cycles

Complexity: O(V*E)

In this implementation:

- Nodes are 1-base indexed
- edges: array contains all edges in the graph under the form of Edge struct
- distance[i]: result array, min distance from node START

```
struct Edge { int u, v, w; };

up(i,1,n) distance[i] = INT_MAX;
distance[START] = 0;
rep(i,1,n) {
    for (Edges e : edges) {
        int a = e.u, b=e.v, w=e.w;
        distance[b] = min(distance[b],
distance[a]+w);
    }
}
```

c. Floyd

Usage: Find shortest cost from all nodes all others

Complexity: O(n^3)

In this implementation:

- a[][]: result array - adjacent matrix - with a[i][j] holds min distance from node i to j

```
int a[239][239];
int n, m;
main() {
    int i,j,k, p,q,w;
    cin >> n >> m;
    ///INIT
    up(i,1,n) up(j,1,n) a[i][j] = INT_MAX;
    up(i,1,n) a[i][i] = 0;
    ///INPUT
    up(i,1,m) {
        cin >> p >> q >> w;
        a[p][q] = a[q][p] = w;
    }

    up(k,1,n) up(i,1,n) up(j,1,n)
    a[i][j] = min(a[i][j], a[i][k]+a[k][j]);
}
```

d. Euler Path/Tour

Usage: An Euler path is defined as a path in a graph which visits each edge of the graph exactly once. Similarly, an Euler tour/cycle is an Euler path which starts and ends on the same vertex.

Technique:

- Euler tour: check if all its vertices have even degrees
- Euler path: an undirected graph has an Euler path if all except two vertices have even degrees. This Euler path will start from one of these odd degree vertices and end in the other

```
list<int> cyc; /// we need list for fast
insertion in the middle
void EulerTour(list<int>::iterator i, int u)
{
    for (int j = 0; j <
(int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (v.second) { /// if this edge can
still be used/not removed
            v.second = 0; /// make the weight
of this edge to be 0 ('removed')
            for (int k = 0; k <
(int)AdjList[v.first].size(); k++) {
                ii uu = AdjList[v.first][k];
// remove bi-directional edge
                if (uu.first == u &&
uu.second) {
                    uu.second = 0;
                    break;
                }
            }
            EulerTour(cyc.insert(i, u),
v.first);
        }
    }
}
/// inside int main()
cyc.clear();
EulerTour(cyc.begin(), A); /// cyc contains
an Euler tour starting at A
for (list<int>::iterator it = cyc.begin(); it
!= cyc.end(); it++)
    printf("%d\n", *it); /// the Euler tour
```

e. Hamiltonian Path

Usage: The travelling salesman problem. Finds a path in the graph that visits each edge exactly once.

```
int i, j, TC, xsize, ysize, n, x[11], y[11],
dist[11][11], memo[11][1 << 11]; /// Karel + max
10 beepers
int tsp(int pos, int bitmask) { /// bitmask stores
the visited coordinates
    if (bitmask == (1 << (n + 1)) - 1)
        return dist[pos][0]; /// return trip to
close the loop
    if (memo[pos][bitmask] != -1)
        return memo[pos][bitmask];

    int ans = 2000000000;
    for (int nxt = 0; nxt <= n; nxt++) /// O(n)
here
        if (nxt != pos && !(bitmask & (1 << nxt)))
/// if coordinate nxt is not visited yet
            ans = min(ans, dist[pos][nxt] +
tsp(nxt, bitmask | (1 << nxt)));
    return memo[pos][bitmask] = ans;
}

int main() {
    tc() {
        scanf("%d %d", &xsize, &ysize); /// these
two values are not used
        scanf("%d %d", &x[0], &y[0]);
        scanf("%d", &n);
        for (i = 1; i <= n; i++) /// karel's
position is at index 0
            scanf("%d %d", &x[i], &y[i]);

        for (i = 0; i <= n; i++) /// build
distance table
            for (j = 0; j <= n; j++)
                dist[i][j] = abs(x[i] - x[j]) +
abs(y[i] - y[j]); /// Manhattan distance

        memset(memo, -1, sizeof memo);
        printf("The shortest path has length
%d\n", tsp(0, 1)); /// DP-TSP
    }
    return 0;
}
```

f. SCC

v. Korasaju

Usage: Create a list of strong connected components in topological order indexed from 1 to counter.

Complexity: O(V+E)

In this implementation:

- adj[]: adjacent list, input
- scc: output

```
void dfs(int x) {
    lastVis[x]=1;
    for(int i=0; i<adj[x].size(); i++) {
        if(!lastVis[adj[x][i]])
            dfs(adj[x][i]);
    }
    sortedList.push_back(x); // lúc xài nhớ
reverse
}
void kosaraju(int x,int group) {
    scc[group].push_back(x);
    lastVis[x]=1;
    for(int i=0; i<revAdj[x].size(); i++) {
        if(!lastVis[revAdj[x][i]]) {
            kosaraju(revAdj[x][i],group);
        }
    }
}

int main() {
    reverse(sortedList.begin(),sortedList.end());
    for(int i=0; i<sortedList.size(); i++)
        if(!lastVis[sortedList[i]])
kosaraju(sortedList[i],++counter); //counter là số
đếm group

    for(int i=1; i<=counter; i++) {
        for(int j=0; j<scc[i].size(); j++) {
            component[scc[i][j]]=i;
        }
    }
}
```

vi. Tarjan

Complexity: O(n).

```
vi dfs_num, dfs_low, S, visited; // global
variables
void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
// dfs_low[u] <= dfs_num[u]
    S.push_back(u); // stores u in a vector based
on order of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size();
j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED)
            tarjanSCC(v.first);
        if (visited[v.first]) // condition for
update
            dfs_low[u] = min(dfs_low[u],
dfs_low[v.first]);
    }
    if (dfs_low[u] == dfs_num[u]) { // if this is
a root (start) of an SCC
        printf("SCC %d:", ++numSCC); // this part
is done after recursion
        while (1) {
            int v = S.back();
            S.pop_back();
            visited[v] = 0;
            printf(" %d", v);
            if (u == v) break;
        }
        printf("\n");
    }
}
int main() {
    dfs_num.assign(V, UNVISITED);
    dfs_low.assign(V, 0);
    visited.assign(V, 0);
    dfsNumberCounter = numSCC = 0;
    for (int i = 0; i < V; i++)
        if (dfs_num[i] == UNVISITED)
            tarjanSCC(i);
}
```

g. Maximum Flow (Dinic)

```
struct edge {
    int a, b, f, c;
};

int n, m;
vector <edge> e;
int pt[MAXN]; // very important performance trick
vector <int> g[MAXN];
long long flow = 0;
queue <int> q;
int d[MAXN];
int lim;
int s, t;


void add_edge(int a, int b, int c) {
    edge ed;
    //keep edges in vector: e[ind] - direct edge,
e[ind ^ 1] - back edge
    ed.a = a;
    ed.b = b;
    ed.f = 0;
    ed.c = c;
    g[a].push_back(e.size());
    e.push_back(ed);

    ed.a = b;
    ed.b = a;
    ed.f = c;
    ed.c = c;
    g[b].push_back(e.size());
    e.push_back(ed);
}
bool bfs() {
    for(int i = s; i <= t; i++) d[i] = inf;
    d[s] = 0;
    q.push(s);
    while (!q.empty() && d[t] == inf) {
        int cur = q.front();
        q.pop();
        for (size_t i = 0; i < g[cur].size(); i++)
{
            int id = g[cur][i];
            int to = e[id].b;
            if (d[to] == inf && e[id].c - e[id].f
>= lim) {
                d[to] = d[cur] + 1;
                q.push(to);
            }
        }
    }
    while (!q.empty()) q.pop();
    return d[t] != inf;
}
bool dfs(int v, int flow) {
    if (flow == 0) return false;
    if (v == t) return true;
    for (; pt[v] < g[v].size(); pt[v]++) {
        int id = g[v][pt[v]];
        int to = e[id].b;
        if (d[to] == d[v] + 1 && e[id].c - e[id].f
>= flow) {
            int pushed = dfs(to, flow);
            if (pushed) {
                e[id].f += flow;
                e[id ^ 1].f -= flow;
                return true;
```

```
            }
        }
    }
    return false;
}


void dinic() {
    for (lim = (1 << 30); lim >= 1;) {
        if (!bfs()) {
            lim >>= 1;
            continue;
        }
        for (int i = s; i <= t; i++)
            pt[i] = 0;
        int pushed;
        while (pushed = dfs(s, lim)) {
            flow = flow + lim;
        }
        //cout << flow << endl;
    }
}
int main() {
    scanf("%d %d", &n, &m);
    s = 1;
    t = n;
    for (int i = 1; i <= m; i++) {
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        add_edge(a, b, c);
    }
    dinic();
    cout << flow << endl;
}
```

h. Minimum Spanning Tree (Kruskal)

Usage: Calculate the cost of a minimum spanning tree of a graph using DSU.

Complexity: O(Elog(E)).

In this implementation:

- The result is stored in mst_cost.
- The number of disjoint sets must eventually be 1 for a valid MST
- EdgeList: Edge list stores each edge as {weight, two vertices}

```
vector< pair<int, ii> > EdgeList;
int mst_cost = 0;
int main() {
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &u, &v, &w); // read the
triple: (u, v, w)
        EdgeList.push_back({w, {u, v}});
    }
    sort(EdgeList.begin(), EdgeList.end()); //
sort by edge weight O(E log E)
    UnionFind UF(V); // all V are disjoint sets
initially
    for (int i = 0; i < E; i++) {
        pair<int, ii> front = EdgeList[i];
        if (!UF.isSameSet(front.se.fi,
front.se.se)) { // check
            mst_cost += front.fi; // add the
weight of e to MST
            UF.unionSet(front.se.fi, front.se.se);
// link them
        }
    }
    printf("MST cost = %d (Kruskalâ€™s)\n",
mst_cost);
}
```

i. Aticulation Point and Bridge

Complexity: O(n).

```
void articulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
// dfs_low[u] <= dfs_num[u]
    for (int j = 0; j < (int)AdjList[u].size();
j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED) { // a
tree edge
            dfs_parent[v.first] = u;
            if (u == dfsRoot) rootChildren++; //
special case if u is a root
            articulationPointAndBridge(v.first);
            if (dfs_low[v.first] >= dfs_num[u]) //
for articulation point
                articulation_vertex[u] = true; //
store this information first
            if (dfs_low[v.first] > dfs_num[u]) //
for bridge
                printf(" Edge (%d, %d) is a
bridge\n", u, v.first);
            dfs_low[u] = min(dfs_low[u],
dfs_low[v.first]); // update dfs_low[u]
        } else if (v.first != dfs_parent[u]) // a
back edge and not direct cycle
            dfs_low[u] = min(dfs_low[u],
dfs_num[v.first]); // update dfs_low[u]
    }
}
// inside int main()
dfsNumberCounter = 0;
dfs_num.assign(V, UNVISITED);
dfs_low.assign(V, 0);
dfs_parent.assign(V, 0);
articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) {
        dfsRoot = i;
        rootChildren = 0;
        articulationPointAndBridge(i);
        articulation_vertex[dfsRoot] =
(rootChildren > 1);
    } // special case
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
        printf(" Vertex %d\n", i);
```

j. Topological Sort

Usage: Sort an DAG into a topological order.

In this implementation:

- n = # of V, m = # of E
- a = adjacent list
- res = result vector, if res.size() < n → no topological order.

```
int n, m;
vi a[N+1];
bool vis[N+1] = {};
int deg[N+1] = {};
vi res;
int main() {
    cin >> n >> m;
    queue<int> qu; ///Replace by set to have
smallest dictionary order.
    ///Input
    up(i,1,m) {
        int x, y;
        cin >> x >> y;
        a[x].pb(y);
        deg[y]++;
    }
    up(i,1,n) if(!deg[i]) {
        qu.insert(i);
        vis[i] = 1;
    }
```

```
    while(!qu.empty()) {
        int cur = qu.front();
        qu.pop();
        vis[cur] = 1;
        res.pb(cur);
        for(int i:a[cur]) {
            if(!vis[i]) {
                deg[i]--;
                if(!deg[i]) {
                    qu.insert(i);
                    vis[i] = 1;
                }
            }
        }
    }
    for(int i:res) cout << i << " ";
}
```

String Processing

a. KMP

Usage: Search pattern P in text T.

Complexity: O(n).

In this implementation:

- T = text, P = pattern
- b = back table
- n = length of T, m = length of P

```
char T[MAX_N], P[MAX_N];
int b[MAX_N], n, m;
void kmpPreprocess() { // call this before calling
kmpSearch()
    int i = 0, j = -1;
    b[0] = -1; // starting values
    while (i < m) { // pre-process the pattern
string P
        while (j >= 0 && P[i] != P[j]) j = b[j];
// different, reset j using b
        i++;
        j++; // if same, advance both pointers
        b[i] = j; // observe i = 8, 9, 10, 11, 12,
13 with j = 0, 1, 2, 3, 4, 5
    }
} // in the example of P = "SEVENTY SEVEN" above
void kmpSearch() { // this is similar as
kmpPreprocess(), but on string T
    int i = 0, j = 0; // starting values
    while (i < n) { // search through string T
        while (j >= 0 && T[i] != P[j]) j = b[j];
// different, reset j using b
        i++;
        j++; // if same, advance both pointers
        if (j == m) { // a match found when j == m
            printf("P is found at index %d in
T\n", i - j);
            j = b[j]; // prepare j for the next
possible match
        }
    }
}
```

b. Trie

Usage: Data structure to manage set of strings

Complexity:

- Build Trie: O(total length of all strings)
- Find : O(length of string)

```
void build(string s) {
    int v = 0, tmp;
    rep(i,0,s.length()) {
        tmp = int(s[i]) - int('a');
        if (trie[v][tmp] == 0) {
            nxt++;
            trie[v][tmp] = nxt;
            g[v].pb(nxt); // Use if build a graph
to reduce complexity when searching
        }
        v = trie[v][tmp];
    }
    trie[v][26] = INT_MAX; // terminating
character at the end of the string
}
```

c. Hash

Complexity: O(n)

Useful primes: 31, 33, 10^9+3, 10^9+7, 26^4-1

```
ll POW[maxn], hashT[maxn];

ll getHashT(int i,int j) {
    return (hashT[j] - hashT[i - 1] * POW[j - i +
1] + MOD * MOD) % MOD;
}

int main() {
    /// Input
    string T, P;
    cin >> T >> P;
    /// Initialize
    int m=T.size(), n=P.size();
    T = " " + T;
    P = " " + P;
    POW[0] = 1;
    /// Precalculate 26^i
    for(i = 1; i <= m; i++)
        POW[i] = (POW[i - 1] * 26) % MOD;
    /// Calculate hash value of T[1..i]
    for(i = 1; i <= m; i++)
        hashT[i] = (hashT[i - 1] * 26 + T[i] -
'a') % MOD;
}
```

d. Z – Algorithm

Usage: Z[i] is the length of the longest substring starting from S[i] which is also a prefix of S.

Complexity: O(n).

```
int L = 0, R = 0;
Z[0] = n;
for (int i = 1; i < n; i++)
    if (i > R) {
        L = R = i;
        while (R < n && S[R] == S[R - L]) R++;
        Z[i] = R - L;
        R--;
    } else {
        int k = i - L;
        if (Z[k] < R - i + 1) Z[i] = Z[k];
        else {
            L = i;
            while(R<n && S[R] == S[R - L]) R++;
            Z[i] = R - L;
            R--;
        }
    }
```

e. Lexicographically Minimal String Rotation

```
int minmove(string s) {
    int n = s.length();
    int x, y, i, j, u, v;
/// x is the smallest string before string y
    for (x = 0, y = 1; y < n; ++ y) {
        i = u = x;
        j = v = y;
        while (s[i] == s[j]) {
            ++ u;
            ++ v;
            if (++ i == n) i = 0;
            if (++ j == n) j = 0;
            if (i == x) break;
/// All strings are equal
        }
```

```
        if (s[i] <= s[j]) y = v;
        else {
            x = y;
            if (u > y) y = u;
        }
    }
    return x;
}
```

Data Structures

a. Segment Tree

i. Classical

```
void build(int id,int l,int r) {
    if(l==r) {
        tree[id]=a[l];
        return;
    } else {
        int mid=(l+r)/2;
        build(id*2,l,mid);
        build(id*2+1,mid+1,r);
        tree[id]=min(tree[id*2],tree[id*2+1]);
    }
}
void modify(int idx,int x,int id,int l,int r) {
    if(l==r) {
        tree[id]=x;
        a[idx]=x;
        return;
    }
    int mid=(l+r)/2;
    if(idx<=mid) modify(idx,x,id*2,l,mid);
    else modify(idx,x,id*2+1,mid+1,r);
    tree[id]=min(tree[id*2],tree[id*2+1]);
}
int query(int x,int y,int l,int r,int id) {
    if(x>r||y<l) return 2e9;
    else if(x<=l&&r<=y) return tree[id];
    else return
min(query(x,y,l,(l+r)/2,id*2),query(x,y,(l+r)/2+1,
r,id*2+1));
}
```

ii. Update an interval to a new value

```
void build(int id,int l,int r) {
    if(l==r) {
        tree[id].F=a[l];
        tree[id].S=0;
        return;
    }
    int mid=(l+r)/2;
    build(id*2,l,mid);
    build(id*2+1,mid+1,r);
    tree[id]=max(tree[id*2],tree[id*2+1]);
}
void upd(int id,ll x) {
    tree[id].F=max(tree[id].F,x);
    tree[id].S=max(tree[id].S,x);
}
void shiftDown(int id,int l,int r) {
    int mid=(l+r)/2;
    if(l<=mid) upd(id*2,tree[id].S);
    if(mid+1<=r) upd(id*2+1,tree[id].S);
    tree[id].S=0;
}
void update(int id,int l,int r,int x,int y,ll val)
{
    if(x>r||y<l) return;
    else if(x<=l&&r<=y) {
        upd(id,val);
        return;
    }
    shiftDown(id,l,r);
    int mid=(l+r)/2;
    update(id*2,l,mid,x,y,val);
    update(id*2+1,mid+1,r,x,y,val);
    tree[id]=max(tree[id*2],tree[id*2+1]);
}
ll queryMax(int id,int l,int r,int x,int y) {
    if(x>r||y<l) return 0;
    else if(x<=l&&r<=y) return tree[id].F;
    shiftDown(id,l,r);
    int mid=(l+r)/2;
    return
max(queryMax(id*2,l,mid,x,y),queryMax(id*2+1,mid+1
,r,x,y));
}
// Cộng đoạn l-r them giá trị v
void upd(int id,int l,int r,int x) {
    tree[id].lazy+=x;
    tree[id].val+=(r-l+1)*x;
}
void shiftDown(int id,int l,int r) {
    int mid=(l+r)/2;
    upd(id*2,l,mid,tree[id].lazy);
    upd(id*2+1,mid+1,r,tree[id].lazy);
    tree[id].lazy=0;
}
void increaseQuery(int id,int x,int y,int v,int
l,int r) {
    if(x>r||y<r) return;
    if(x<=l&&r<=y) {
        upd(id,l,r,v);
        return;
    }
    shiftDown(id,l,r);
    int mid=(l+r)/2;
    increaseQuery(id*2,x,y,v,l,mid);
    increaseQuery(id*2+1,x,y,v,mid+1,r);
    tree[id].val=tree[id*2].val+tree[id*2+1].val;
}
int sum(int id,int x,int y,int l,int r) {
    if(x>l||y<r) return 0;
    if(x<=l&&r<=y) return tree[id].val;
    shiftDown(id,l,r);
    return
sum(2*id,x,y,l,(l+r)/2)+sum(id*2+1,x,y,(l+r)/2+1,r
);
}
```

iii. Merge-sort tree

```
vi tree[800005],adj[100005];
int
n,a[100005],idx=0,Ti[100005],temp,dfsOrder[200005]
;
pii euler[100005];
bool vis[100005]= {0};
void dfs(int x) {
    euler[x].first=++idx;
    dfsOrder[idx]=x;
    vis[x]=1;
    for(auto i:adj[x]) {
        if(!vis[i]) {
            dfs(i);
        }
    }
    euler[x].second=++idx;
    dfsOrder[idx]=x;
}
void build(int id,int l,int r) {
    if(l==r) {
        tree[id].pb(a[dfsOrder[l]]);
        return;
    }
    int mid=(l+r)/2;
    build(id*2,l,mid);
    build(id*2+1,mid+1,r);

tree[id].resize(tree[id*2].size()+tree[id*2+1].siz
e());

merge(all(tree[id*2]),all(tree[id*2+1]),tree[id].b
egin());
```

```
}
int query(int x,int y,int id,int l,int r,int val)
{
    if(x>r||y<l) return 0;
    else if(x<=l&&r<=y) {
        return tree[id].end()-
upper_bound(all(tree[id]),val);
    }
    int mid=(l+r)/2;
    return
query(x,y,id*2,l,mid,val)+query(x,y,id*2+1,mid+1,r
,val);
}
```

f. Binary Indexed Tree

Complexity: O(log(n))

```
int BIT[N+1]= {0};

void update(int x,int idx) {
    while(idx<=N) {
        BIT[idx]+=x;
        idx+=(idx&(-idx));
    }
}
int query(int idx) {
    int sum=0;
    while(idx>0) {
        sum+=BIT[idx];
        idx-=(idx&(-idx));
    }
    return sum;
}
```

g. Disjoint Set Union

Complexity: O(α(n))

Nếu par[i] < 0 thì viên sỏi i nằm trong hộp i, và -par[i] chính là số sỏi trong hộp đó.

```
par[N+1];

void init() { up(i,0,N) par[i]=-1; }

int root(int v) {
    return (par[v] < 0 ? v : (par[v] =
root(par[v])))
}

void uni(int x, int y) {
    if ((x = root(x)) == (y = root(y)) return;
    if (par[y] < par[x]) swap(x, y);
    par[x] += par[y];
    par[y] = x;
}
```

h. Lowest Common Ancestor

Complexity:

- Build: O(nlog(n))
- Query: O(logn)

```
void process3(int N, int T[MAXN], int
P[MAXN][LOGMAXN]) {
    int i, j;
    //we initialize every element in P with -1
    for (i = 0; i < N; i++)
        for (j = 0; 1 << j < N; j++)
            P[i][j] = -1;
    //the first ancestor of every node i is T[i]
    for (i = 0; i < N; i++)
        P[i][0] = T[i];
    //bottom up dynamic programing
    for (j = 1; 1 << j < N; j++)
        for (i = 0; i < N; i++)
            if (P[i][j - 1] != -1)
                P[i][j] = P[P[i][j - 1]][j - 1];
}
```

```
int query(int N, int P[MAXN][LOGMAXN], int
T[MAXN],
          int L[MAXN], int p, int q) {
    int tmp, log, i;
    //if p is situated on a higher level than q
then we swap them
    if (L[p] < L[q])
        tmp = p, p = q, q = tmp;
    //we compute the value of [log(L[p)]
    for (log = 1; 1 << log <= L[p]; log++);
    log--;
    //we find the ancestor of node p situated on
the same level
    //with q using the values in P
    for (i = log; i >= 0; i--)
        if (L[p] - (1 << i) >= L[q])
            p = P[p][i];
    if (p == q)
        return p;
    //we compute LCA(p, q) using the values in P
    for (i = log; i >= 0; i--)
        if (P[p][i] != -1 && P[p][i] != P[q][i])
            p = P[p][i], q = P[q][i];
    return T[p];
}
```

i. LCA – Binary lifting

```
int walk(int s,int p){
    for(int i=0;i<=19;i++){
        if(p&(1<<i)) s=parent[s][i];
    }
    return s;
}
```

j. Ordered set/multiset

```
#include <ext/pb_ds/assoc_container.hpp> // Common
file
#include <ext/pb_ds/tree_policy.hpp> // Including
tree_order_statistics_node_update
#include <bits/stdc++.h>
using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>,
rb_tree_tag, tree_order_statistics_node_update>
ordered_set;
typedef tree<pii, null_type, less<pii>,
rb_tree_tag, tree_order_statistics_node_update>
orderedMulti_set;
int main () {
    ordered_set test;
    ordered_set X;
    X.insert(1);
    X.insert(2);
    X.insert(4);
    X.insert(8);
    X.insert(16);

    cout<<*X.find_by_order(1)<<endl; // 2
    cout<<*X.find_by_order(2)<<endl; // 4
    cout<<*X.find_by_order(4)<<endl; // 16
    cout<<X.order_of_key(-5)<<endl;  // 0
    cout<<X.order_of_key(1)<<endl;   // 0
    cout<<X.order_of_key(3)<<endl;   // 2
    cout<<X.order_of_key(4)<<endl;   // 2
    cout<<X.order_of_key(400)<<endl; // 5
    cout<<X.order_of_key(16)<<endl;
    cout<<X.size()<<endl;
    return 0;
}
```

Miscellaneous

a. Custom Comparing Class

```
class Compare {
public:
    bool operator() (Foo, Foo) {
```

```
        return true;
    }
};
```

Usage:

- `priority_queue<int, vector<int> ,Compare> pq;`
- `sort(v.begin(),v.end(),Compare());`
- `map<char,int,Compare> ma;`
- `set<int,Compare> se;`
- `multiset<int,Compare> se;`
- `multimap<char,int,Compare> ma;`
- `binary_search(v.begin(), v.end(), val, Compare());`

k. Useful Calculus

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

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

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

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$$

$$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2}-(c+1)c^{n+1}+c}{(c-1)^2}, c \neq 1$$

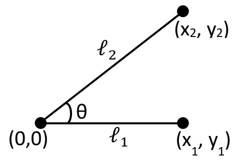
$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

l. Useful Geometry

Area of triangle (x_0,y_0), (x_1,y_1), (x_2,y_2):

$$S = \frac{1}{2}\begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix}$$

Angle formed by 3 points:



$$\cos\theta = \frac{(x_1, y_1) \cdot (x_2, y_2)}{\ell_1 \ell_2}$$

m. Iterate through all permutation

```
vi v;
rep(i,1,n) v.pb(a[i]);
sort(v.begin(),v.end());
do {
/// process
} while (next_permutation(v.begin(),v.end()));
```

n. Generate all subsets

```
for (int b = 0; b < (1<<n); b++) {
    vi subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) subset.pb(i);
    }
}
```

o. Josephus problem

```
def josephus (n , k): # 1.. n
    r , i = 0 , 2
    while i <= n:
        r , i = (r + k) % i , i + 1
    return r + 1
```

p. Java theme and syntax

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;

public class Main {

    public static void main(String[] args) throws
IOException {
        BufferedReader bufferedReader = new
BufferedReader( new InputStreamReader(System.in));
// fast input
        StringTokenizer st = new
StringTokenizer(bufferedReader.readLine()); //
fast input

        BigInteger a = new
BigInteger(st.nextToken()); // init
        BigInteger b = new
BigInteger(st.nextToken()); // init
        BigInteger result;
        result = a.add(b); // add
        result = a.multiply(b); // mul
        result = a.abs(); // abs
        result = b.divide(a); // result is like
b/a (long long)
        result = a.gcd(b);//gcd
        result = a.max(b); // max
        result = a.min(b); // min
        int c = a.compareTo(b); // 0  = , 1 > ,-1
<;
        result = a.mod(new BigInteger("8")); //
mod
        result = a.modPow(new
BigInteger("100"),new BigInteger("1000000007"));
        result = a.negate(); // -a;
        result = a.nextProbablePrime(); // next
possible prime > a;
        result = a.subtract(b); // a-b
        result = a.pow(10);
    }
}
```

q. Geometry Template

i. Points And Lines

```
#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) /// important constant; alternative #define
PI (2.0 * acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0; }
double RAD_to_DEG(double r) { return r * 180.0 / PI; }

/// struct point_i { int x, y; }; basic raw form, minimalist mode
struct point_i {
    int x, y; /// whenever possible, work with point_i
    point_i() {
        x = y = 0; /// default constructor
    }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};

struct point {
    double x, y; /// only used if more precision is needed
    point() {
        x = y = 0.0; /// default constructor
    }
    point(double _x, double _y) : x(_x), y(_y) {} /// user-
defined
    bool operator < (point other) const { /// override less than
operator
        if (fabs(x - other.x) > EPS) /// useful for sorting
            return x < other.x; /// first criteria , by x-
coordinate
        return y < other.y; /// second criteria, by y-coordinate
    }
    /// use EPS (1e-9) when testing equality of two floating
points
    bool operator == (point other) const {
        return (fabs(x - other.x) < EPS && (fabs(y - other.y) <
EPS));
    }
};

double dist(point p1, point p2) { /// Euclidean distance
```

```
    /// hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

/// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
    double rad = DEG_to_RAD(theta); /// multiply theta with PI /
180.0
    return point(p.x * cos(rad) - p.y * sin(rad),
                 p.x * sin(rad) + p.y * cos(rad));
}

struct line {
    double a, b, c;
}; /// a way to represent a line

/// the answer is stored in the third parameter (pass by
reference)
void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x - p2.x) < EPS) { /// vertical line is fine
        l.a = 1.0;
        l.b = 0.0;
        l.c = -p1.x; /// default values
    } else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0; /// IMPORTANT: we fix the value of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}

/// not needed since we will use the more robust form: ax + by +
c = 0 (see above)
struct line2 {
    double m, c;
}; /// another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
    if (abs(p1.x - p2.x) < EPS) { /// special case: vertical line
        l.m = INF; /// l contains m = INF and c = x_value
        l.c = p1.x; /// to denote vertical line x = x_value
        return 0; /// we need this return variable to
differentiate result
    } else {
        l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
        l.c = p1.y - l.m * p1.x;
        return 1; /// l contains m and c of the line equation y =
mx + c
    }
}

bool areParallel(line l1, line l2) { /// check coefficients a & b
    return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}

bool areSame(line l1, line l2) { /// also check coefficient c
    return areParallel(l1,l2) && (fabs(l1.c - l2.c) < EPS);
}

/// returns true (+ intersection point) if two lines are
intersect
bool areIntersect(line l1, line l2, point &p) {
    if (areParallel(l1, l2)) return false; /// no intersection
    /// solve system of 2 linear algebraic equations with 2
unknowns
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a *
l2.b);
    /// special case: test for vertical line to avoid division by
zero
    if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
    else                  p.y = -(l2.a * p.x + l2.c);
    return true;
}

struct vec {
    double x, y;  /// name: `vec' is different from STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) { /// convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}

vec scale(vec v, double s) { /// nonnegative s = [<1 .. 1 .. >1]
    return vec(v.x * s, v.y * s);
} /// shorter.same.longer

point translate(point p, vec v) { /// translate p according to v
    return point(p.x + v.x, p.y + v.y);
}

/// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
    l.a = -m; /// always -m
    l.b = 1; /// always 1
    l.c = -((l.a * p.x) + (l.b * p.y));
} /// compute this

void closestPoint(line l, point p, point &ans) {
    line perpendicular; /// perpendicular to l and pass through p
    if (fabs(l.b) < EPS) { /// special case 1: vertical line
        ans.x = -(l.c);
        ans.y = p.y;
        return;
    }

    if (fabs(l.a) < EPS) { /// special case 2: horizontal line
        ans.x = p.x;
        ans.y = -(l.c);
        return;
    }

    pointSlopeToLine(p, 1 / l.a, perpendicular); /// normal line
    /// intersect line l with this perpendicular line
    /// the intersection point is the closest point
    areIntersect(l, perpendicular, ans);
}

/// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
    point b;
    closestPoint(l, p, b); /// similar to distToLine
    vec v = toVec(p, b); /// create a vector
    ans = translate(translate(p, v), v);
}       /// translate p twice

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

/// returns the distance from p to the line defined by two points
a and b (a and b must be different) the closest point is stored
in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
    /// formula: c = a + u * ab
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u)); /// translate a to c
    return dist(p, c);
} /// Euclidean distance between p and c

/// returns the distance from p to the line segment ab defined by
two points a and b (still OK if a == b) the closest point is
stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) {
        c = point(a.x, a.y); /// closer to a
        return dist(p, a);
    } /// Euclidean distance between p and a
    if (u > 1.0) {
        c = point(b.x, b.y); /// closer to b
        return dist(p, b);
    }          /// Euclidean distance between p and b
    return distToLine(p, a, b, c);
}         /// run distToLine as above

double angle(point a, point o, point b) {  /// returns angle aob
in rad
    vec oa = toVec(o, a), ob = toVec(o, b);
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

/// another variant
///int area2(point p, point q, point r) { /// returns 'twice' the
area of this triangle A-B-c
///  return p.x * q.y - p.y * q.x +
///         q.x * r.y - q.y * r.x +
///         r.x * p.y - r.y * p.x;
///}

/// note: to accept collinear points, we have to change the `> 0'
/// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) { return cross(toVec(p, q),
toVec(p, r)) > 0; }

/// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
    return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}

int main() {
    point P1, P2, P3(0, 1); /// note that both P1 and P2 are
(0.00, 0.00)
    printf("%d\n", P1 == P2); /// true
    printf("%d\n", P1 == P3); /// false

    vector<point> P;
    P.push_back(point(2, 2));
    P.push_back(point(4, 3));
    P.push_back(point(2, 4));
    P.push_back(point(6, 6));
    P.push_back(point(2, 6));
    P.push_back(point(6, 5));
```

```
    /// sorting points demo
    sort(P.begin(), P.end());
    for (int i = 0; i < (int)P.size(); i++)
        printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

    /// rearrange the points as shown in the diagram below
    P.clear();
    P.push_back(point(2, 2));
    P.push_back(point(4, 3));
    P.push_back(point(2, 4));
    P.push_back(point(6, 6));
    P.push_back(point(2, 6));
    P.push_back(point(6, 5));
    P.push_back(point(8, 6));

    /**
    /// the positions of these 7 points (0-based indexing)
    6   P4      P3  P6
    5           P5
    4   P2
    3       P1
    2   P0
    1
    0 1 2 3 4 5 6 7 8
    */

    double d = dist(P[0], P[5]);
    printf("Euclidean distance between P[0] and P[5] = %.2lf\n",
d); /// should be 5.000

    /// line equations
    line l1, l2, l3, l4;
    pointsToLine(P[0], P[1], l1);
    printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l1.a, l1.b,
l1.c); /// should be -0.50 * x + 1.00 * y - 1.00 = 0.00

    pointsToLine(P[0], P[2], l2); /// a vertical line, not a
problem in "ax + by + c = 0" representation
    printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l2.a, l2.b,
l2.c); /// should be 1.00 * x + 0.00 * y - 2.00 = 0.00

    /// parallel, same, and line intersection tests
    pointsToLine(P[2], P[3], l3);
    printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2));
/// no
    printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3));
/// yes, l1 (P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

    pointsToLine(P[2], P[4], l4);
    printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); /// no
    printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); ///
yes, l2 (P[0]-P[2]) and l4 (P[2]-P[4]) are the same line (note,
they are two different line segments, but same line)

    point p12;
    bool res = areIntersect(l1, l2, p12); /// yes, l1 (P[0]-P[1])
and l2 (P[0]-P[2]) are intersect at (2.0, 2.0)
    printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res,
p12.x, p12.y);

    /// other distances
    point ans;
    d = distToLine(P[0], P[2], P[3], ans);
    printf("Closest point from P[0] to line         (P[2]-P[3]):
(%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d);
    closestPoint(l3, P[0], ans);
    printf("Closest point from P[0] to line V2      (P[2]-P[3]):
(%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

    d = distToLineSegment(P[0], P[2], P[3], ans);
    printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]):
(%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); /// closer to
A (or P[2]) = (2.00, 4.00)
    d = distToLineSegment(P[1], P[2], P[3], ans);
    printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]):
(%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); /// closer to
midway between AB = (3.20, 4.60)
    d = distToLineSegment(P[6], P[2], P[3], ans);
    printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]):
(%.2lf, %.2lf), dist = %.2lf\n", ans.x, ans.y, d); /// closer to
B (or P[3]) = (6.00, 6.00)

    reflectionPoint(l4, P[1], ans);
    printf("Reflection point from P[1] to line      (P[2]-P[4]):
(%.2lf, %.2lf)\n", ans.x, ans.y); /// should be (0.00, 3.00)

    printf("Angle P[0]-P[4]-P[3] = %.2lf\n",
RAD_to_DEG(angle(P[0], P[4], P[3]))); /// 90 degrees
    printf("Angle P[0]-P[2]-P[1] = %.2lf\n",
RAD_to_DEG(angle(P[0], P[2], P[1]))); /// 63.43 degrees
    printf("Angle P[4]-P[3]-P[6] = %.2lf\n",
RAD_to_DEG(angle(P[4], P[3], P[6]))); /// 180 degrees

    printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0],
P[2], P[3])); /// no
    printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0],
P[3], P[2])); /// yes

    printf("P[0], P[2], P[3] are collinear? %d\n",
collinear(P[0], P[2], P[3])); /// no
    printf("P[0], P[2], P[4] are collinear? %d\n",
collinear(P[0], P[2], P[4])); /// yes

    point p(3, 7), q(11, 13), r(35, 30); /// collinear if r(35,
31)
    printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left"
: "right"); /// right

    /**
     the positions of these 6 points
       E<--  4
             3       B D<--
             2   A C
             1
    -4-3-2-1 0 1 2 3 4 5 6
            -1
            -2
     F<--   -3
    */

    /// translation
    point A(2.0, 2.0);
    point B(4.0, 3.0);
    vec v = toVec(A, B); /// imagine there is an arrow from A to
B (see the diagram above)
    point C(3.0, 2.0);
    point D = translate(C, v); /// D will be located in
coordinate (3.0 + 2.0, 2.0 + 1.0) = (5.0, 3.0)
    printf("D = (%.2lf, %.2lf)\n", D.x, D.y);
    point E = translate(C, scale(v, 0.5)); /// E will be located
in coordinate (3.0 + 1/2 * 2.0, 2.0 + 1/2 * 1.0) = (4.0, 2.5)
    printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

    /// rotation
    printf("B = (%.2lf, %.2lf)\n", B.x, B.y); /// B = (4.0, 3.0)
    point F = rotate(B, 90); /// rotate B by 90 degrees COUNTER
clockwise, F = (-3.0, 4.0)
    printf("F = (%.2lf, %.2lf)\n", F.x, F.y);
    point G = rotate(B, 180); /// rotate B by 180 degrees COUNTER
clockwise, G = (-4.0, -3.0)
    printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

    return 0;
}
```

ii. Circles

```
#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }
double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i {
    int x, y;     /// whenever possible, work with point_i
    point_i() {
        x = y = 0;    /// default constructor
    }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};          /// constructor

struct point {
    double x, y; /// only used if more precision is needed
    point() {
        x = y = 0.0; /// default constructor
    }
    point(double _x, double _y) : x(_x), y(_y) {}
};      /// constructor

int insideCircle(point_i p, point_i c, int r) {
/// all integer version
    int dx = p.x - c.x, dy = p.y - c.y;
    int Euc = dx * dx + dy * dy, rSq = r * r; /// all integer
    return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
} ///inside/border/outside

bool circle2PtsRad(point p1, point p2, double r, point &c) {
    double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
                (p1.y - p2.y) * (p1.y - p2.y);
    double det = r * r / d2 - 0.25;
    if (det < 0.0) return false;
    double h = sqrt(det);
    c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
    c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
    return true;
}   /// to get the other center, reverse p1 and p2

int main() {
    /// circle equation, inside, border, outside
    point_i pt(2, 2);
    int r = 7;
    point_i inside(8, 2);
    printf("%d\n", insideCircle(inside, pt, r)); /// 0-inside
    point_i border(9, 2);
    printf("%d\n", insideCircle(border, pt, r)); /// 1-at border
```

```
    point_i outside(10, 2);
    printf("%d\n", insideCircle(outside, pt, r));/// 2-outside

    double d = 2 * r;
    printf("Diameter = %.2lf\n", d);
    double c = PI * d;
    printf("Circumference (Perimeter) = %.2lf\n", c);
    double A = PI * r * r;
    printf("Area of circle = %.2lf\n", A);

    printf("Length of arc    (central angle = 60 degrees) =
%.2lf\n", 60.0 / 360.0 * c);
    printf("Length of chord  (central angle = 60 degrees) =
%.2lf\n", sqrt((2 * r * r) * (1 - cos(DEG_to_RAD(60.0)))));
    printf("Area of sector   (central angle = 60 degrees) =
%.2lf\n", 60.0 / 360.0 * A);

    point p1;
    point p2(0.0, -1.0);
    point ans;
    circle2PtsRad(p1, p2, 2.0, ans);
    printf("One of the center is (%.2lf, %.2lf)\n", ans.x,
ans.y);
    circle2PtsRad(p2, p1, 2.0, ans);
/// we simply reverse p1 with p2
    printf("The other center  is (%.2lf, %.2lf)\n", ans.x,
ans.y);

    return 0;
}
```

iii. Triangles

```
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }
double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i {
    int x, y; /// whenever possible, work with point_i
    point_i() {
        x = y = 0;    /// default constructor
    }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};          /// constructor

struct point {
    double x, y; /// only used if more precision is needed
    point() {
        x = y = 0.0;    /// default constructor
    }
    point(double _x, double _y) : x(_x), y(_y) {}
};      /// constructor

double dist(point p1, point p2) { return hypot(p1.x - p2.x, p1.y
- p2.y); }

double perimeter(double ab, double bc, double ca) { return ab +
bc + ca; }

double perimeter(point a, point b, point c) { return dist(a, b) +
dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {
    /// Heron's formula, split sqrt(a * b) into sqrt(a) *
sqrt(b); in implementation
    double s = 0.5 * perimeter(ab, bc, ca);
    return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca);
}

double area(point a, point b, point c) { return area(dist(a, b),
dist(b, c), dist(c, a)); }

///=================================================================
/// from ch7_01_points_lines
struct line {
    double a, b, c;
}; /// a way to represent a line

/// the answer is stored in the third parameter (pass by
reference)
void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x - p2.x) < EPS) { /// vertical line is fine
        l.a = 1.0;
        l.b = 0.0;
        l.c = -p1.x; /// default values
    } else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0; /// IMPORTANT: we fix the value of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}

bool areParallel(line l1, line l2) { /// check coefficient a + b
    return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}

/// returns true (+ intersection point) if two lines are
intersect
bool areIntersect(line l1, line l2, point &p) {
    if (areParallel(l1, l2)) return false; /// no intersection
    /// solve system of 2 linear algebraic equations with 2
unknowns
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a *
l2.b);
    /// special case: test for vertical line to avoid division by
zero
    if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
    else                  p.y = -(l2.a * p.x + l2.c);
    return true;
}

struct vec {
    double x, y; /// name: `vec' is different from STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) { /// convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}

vec scale(vec v, double s) { /// nonnegative s = [<1 .. 1 .. >1]
    return vec(v.x * s, v.y * s);
} /// shorter.same.longer

point translate(point p, vec v) { /// translate p according to v
    return point(p.x + v.x, p.y + v.y);
}
///=================================================================
======

double rInCircle(double ab, double bc, double ca) { return
area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) { return
rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

/// assumption: the required points/lines functions have been
written
/// returns 1 if there is an inCircle center, returns 0 otherwise
/// if this function returns 1, ctr will be the inCircle center
/// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r)
{
    r = rInCircle(p1, p2, p3);
    if (fabs(r) < EPS) return 0; /// no inCircle center

    line l1, l2; /// compute these two angle bisectors
    double ratio = dist(p1, p2) / dist(p1, p3);
    point p = translate(p2, scale(toVec(p2, p3), ratio / (1 +
ratio)));
    pointsToLine(p1, p, l1);

    ratio = dist(p2, p1) / dist(p2, p3);
    p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
    pointsToLine(p2, p, l2);

    areIntersect(l1, l2, ctr); /// get their intersection point
    return 1;
}

double rCircumCircle(double ab, double bc, double ca) { return ab
* bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) { return
rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

/// assumption: the required points/lines functions have been
written
/// returns 1 if there is a circumCenter center, returns 0
otherwise
/// if this function returns 1, ctr will be the circumCircle
center
/// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double
&r) {
    double a = p2.x - p1.x, b = p2.y - p1.y;
    double c = p3.x - p1.x, d = p3.y - p1.y;
    double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
    double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
    double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
    if (fabs(g) < EPS) return 0;

    ctr.x = (d*e - b*f) / g;
    ctr.y = (a*f - c*e) / g;
    r = dist(p1, ctr);
/// r = distance from center to 1 of the 3 points
    return 1;
}

/// returns true if point d is inside the circumCircle defined by
a,b,c
int inCircumCircle(point a, point b, point c, point d) {
    return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x)
+ (c.y - d.y) * (c.y - d.y)) +
           (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y)
```

```
* (b.y - d.y)) * (c.x - d.x) +
           ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y -
d.y)) * (b.x - d.x) * (c.y - d.y) -
           ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y -
d.y)) * (b.y - d.y) * (c.x - d.x) -
           (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x)
+ (c.y - d.y) * (c.y - d.y)) -
           (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y)
* (b.y - d.y)) * (c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
    return (a + b > c) && (a + c > b) && (b + c > a);
}

int main() {
    double base = 4.0, h = 3.0;
    double A = 0.5 * base * h;
    printf("Area = %.2lf\n", A);

    point a; /// a right triangle
    point b(4.0, 0.0);
    point c(4.0, 3.0);

    double p = perimeter(a, b, c);
    double s = 0.5 * p;
    A = area(a, b, c);
    printf("Area = %.2lf\n", A); /// must be the same as above

    double r = rInCircle(a, b, c);
    printf("R1 (radius of incircle) = %.2lf\n", r); /// 1.00
    point ctr;
    int res = inCircle(a, b, c, ctr, r);
    printf("R1 (radius of incircle) = %.2lf\n", r); /// same,
1.00
    printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y); /// (3.00,
1.00)

    printf("R2 (radius of circumcircle) = %.2lf\n",
rCircumCircle(a, b, c)); /// 2.50
    res = circumCircle(a, b, c, ctr, r);
    printf("R2 (radius of circumcircle) = %.2lf\n", r);   ///
same, 2.50
    printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   ///
(2.00, 1.50)

    point d(2.0, 1.0); /// inside triangle and circumCircle
    printf("d inside circumCircle (a, b, c) ? %d\n",
inCircumCircle(a, b, c, d));
    point e(2.0, 3.9);
/// outside the triangle but inside circumCircle
    printf("e inside circumCircle (a, b, c) ? %d\n",
inCircumCircle(a, b, c, e));
    point f(2.0, -1.1);  /// slightly outside
    printf("f inside circumCircle (a, b, c) ? %d\n",
inCircumCircle(a, b, c, f));

    /// Law of Cosines
    double ab = dist(a, b);
    double bc = dist(b, c);
    double ca = dist(c, a);
    double alpha = RAD_to_DEG(acos((ca * ca + ab * ab - bc * bc)
/ (2.0 * ca * ab)));
    printf("alpha = %.2lf\n", alpha);
    double beta  = RAD_to_DEG(acos((ab * ab + bc * bc - ca * ca)
/ (2.0 * ab * bc)));
    printf("beta  = %.2lf\n", beta);
    double gamma = RAD_to_DEG(acos((bc * bc + ca * ca - ab * ab)
/ (2.0 * bc * ca)));
    printf("gamma = %.2lf\n", gamma);

    /// Law of Sines
    printf("%.2lf == %.2lf == %.2lf\n", bc /
sin(DEG_to_RAD(alpha)), ca / sin(DEG_to_RAD(beta)), ab /
sin(DEG_to_RAD(gamma)));

    /// Phytagorean Theorem
    printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

    /// Triangle Inequality
    printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5,
canFormTriangle(3, 4, 5)); /// yes
    printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7,
canFormTriangle(3, 4, 7)); /// no, actually straight line
    printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8,
canFormTriangle(3, 4, 8)); /// no

    return 0;
}
```

iv. Polygons

```
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }
double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point {
    double x, y;   /// only used if more precision is needed
    point() {
        x = y = 0.0;    /// default constructor
    }
    point(double _x, double _y) : x(_x), y(_y) {} /// user-
defined
    bool operator == (point other) const {
        return (fabs(x - other.x) < EPS && (fabs(y - other.y) <
EPS));
    }
};

struct vec {
    double x, y;  /// name: `vec' is different from STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) { /// convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}

double dist(point p1, point p2) { /// Euclidean distance
    return hypot(p1.x - p2.x, p1.y - p2.y);
} /// return double

/// returns the perimeter, which is the sum of Euclidian
distances
/// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
    double result = 0.0;
    for (int i = 0; i < (int)P.size()-1; i++)
/// remember that P[0] = P[n-1]
        result += dist(P[i], P[i+1]);
    return result;
}

/// returns the area, which is half the determinant
double area(const vector<point> &P) {
    double result = 0.0, x1, y1, x2, y2;
    for (int i = 0; i < (int)P.size()-1; i++) {
        x1 = P[i].x;
        x2 = P[i+1].x;
        y1 = P[i].y;
        y2 = P[i+1].y;
        result += (x1 * y2 - x2 * y1);
    }
    return fabs(result) / 2.0;
}

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

double angle(point a, point o, point b) {
/// returns angle aob in rad
    vec oa = toVec(o, a), ob = toVec(o, b);
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

/// note: to accept collinear points, we have to change the `> 0'
/// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) { return cross(toVec(p, q),
toVec(p, r)) > 0;}

/// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) { return
fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

/// returns true if we always make the same turn while examining
all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
    int sz = (int)P.size();
    if (sz <= 3) return false;
/// a point/sz=2 or a line/sz=3 is not convex
    bool isLeft = ccw(P[0], P[1], P[2]);/// remember one result
    for (int i = 1; i < sz-1; i++)/// compare with the others
        if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) !=
isLeft)
    return false; /// different sign -> this polygon is concave
    return true;
}            /// this polygon is convex

/// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
    if ((int)P.size() == 0) return false;
    double sum = 0;
/// assume the first vertex is equal to the last vertex
    for (int i = 0; i < (int)P.size()-1; i++) {
        if (ccw(pt, P[i], P[i+1]))
            sum += angle(P[i], pt, P[i+1]);  /// left turn/ccw
        else sum -= angle(P[i], pt, P[i+1]);
    }    /// right turn/cw
    return fabs(fabs(sum) - 2*PI) < EPS;
}

/// line segment p-q intersect with line A-B.
```

```
point lineIntersectSeg(point p, point q, point A, point B) {
    double a = B.y - A.y;
    double b = A.x - B.x;
    double c = B.x * A.y - A.x * B.y;
    double u = fabs(a * p.x + b * p.y + c);
    double v = fabs(a * q.x + b * q.y + c);
    return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u)
/ (u+v));
}

/// cuts polygon Q along the line formed by point a -> point b
/// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point>
&Q) {
    vector<point> P;
    for (int i = 0; i < (int)Q.size(); i++) {
        double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2
= 0;
        if (i != (int)Q.size()-1) left2 = cross(toVec(a, b),
toVec(a, Q[i+1]));
        if (left1 > -EPS) P.push_back(Q[i]);
/// Q[i] is on the left of ab
        if (left1 * left2 < -EPS)
/// edge (Q[i], Q[i+1]) crosses line ab
            P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
    }
    if (!P.empty() && !(P.back() == P.front()))
        P.push_back(P.front());
/// make P's first point = P's last point
    return P;
}

point pivot;
bool angleCmp(point a, point b) { /// angle-sorting function
    if (collinear(pivot, a, b)) /// special case
        return dist(pivot, a) < dist(pivot, b); /// check which
one is closer
    double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
    double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
    return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}   /// compare two angles

vector<point> CH(vector<point> P) {
/// the content of P may be reshuffled
    int i, j, n = (int)P.size();
    if (n <= 3) {
        if (!(P[0] == P[n-1])) P.push_back(P[0]); /// safeguard
from corner case
        return P;  /// special case, the CH is P itself
    }

    /// first, find P0 = point with lowest Y and if tie:
rightmost X
    int P0 = 0;
    for (i = 1; i < n; i++)
        if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x >
P[P0].x))
            P0 = i;

    point temp = P[0];
    P[0] = P[P0];
    P[P0] = temp;    /// swap P[P0] with P[0]

    /// second, sort points by angle w.r.t. pivot P0
    pivot = P[0]; /// use this global variable as reference
    sort(++P.begin(), P.end(), angleCmp); /// we do not sort P[0]

    /// third, the ccw tests
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]);   /// initial S
    i = 2;  /// then, we check the rest
    while (i < n) {
/// note: N must be >= 3 for this method to work
        j = (int)S.size()-1;
        if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  ///
left turn, accept
        else S.pop_back();
    }   /// or pop the top of S until we have a left turn
    return S;
}  /// return the result

int main() {
    /// 6 points, entered in counter clockwise order, 0-based
indexing
    vector<point> P;
    P.push_back(point(1, 1));
    P.push_back(point(3, 3));
    P.push_back(point(9, 1));
    P.push_back(point(12, 4));
    P.push_back(point(9, 7));
    P.push_back(point(1, 7));
    P.push_back(P[0]); /// loop back

    printf("Perimeter of polygon = %.2lf\n", perimeter(P)); ///
31.64
    printf("Area of polygon = %.2lf\n", area(P)); /// 49.00
    printf("Is convex = %d\n", isConvex(P)); /// false (P1 is the
culprit)

    ////// the positions of P6 and P7 w.r.t the polygon
    ///7 P5--------------P4
    ///6 |                  \
    ///5 |                   \
    ///4 |   P7               P3
    ///3 |   P1___           /
    ///2 | / P6    \ ___   /
    ///1 P0              P2
    ///0 1 2 3 4 5 6 7 8 9 101112

    point P6(3, 2); /// outside this (concave) polygon
    printf("Point P6 is inside this polygon = %d\n",
inPolygon(P6, P)); /// false
    point P7(3, 4); /// inside this (concave) polygon
    printf("Point P7 is inside this polygon = %d\n",
inPolygon(P7, P)); /// true

    /// cutting the original polygon based on line P[2] -> P[4]
(get the left side)
    ///7 P5--------------P4
    ///6 |               |  \
    ///5 |               |   \
    ///4 |               |    P3
    ///3 |   P1___       |   /
    ///2 | /       \ ___ |  /
    ///1 P0              P2
    ///0 1 2 3 4 5 6 7 8 9 101112
    /// new polygon (notice the index are different now):
    ///7 P4--------------P3
    ///6 |               |
    ///5 |               |
    ///4 |               |
    ///3 |   P1___       |
    ///2 | /       \ ___ |
    ///1 P0              P2
    ///0 1 2 3 4 5 6 7 8 9

    P = cutPolygon(P[2], P[4], P);
    printf("Perimeter of polygon = %.2lf\n", perimeter(P)); ///
smaller now 29.15
    printf("Area of polygon = %.2lf\n", area(P)); /// 40.00

    /// running convex hull of the resulting polygon (index
changes again)
    ///7 P3--------------P2
    ///6 |               |
    ///5 |               |
    ///4 |   P7          |
    ///3 |               |
    ///2 |               |
    ///1 P0--------------P1
    ///0 1 2 3 4 5 6 7 8 9

    P = CH(P); /// now this is a rectangle
    printf("Perimeter of polygon = %.2lf\n", perimeter(P)); ///
precisely 28.00
    printf("Area of polygon = %.2lf\n", area(P)); /// precisely
48.00
    printf("Is convex = %d\n", isConvex(P)); /// true
    printf("Point P6 is inside this polygon = %d\n",
inPolygon(P6, P)); /// true
    printf("Point P7 is inside this polygon = %d\n",
inPolygon(P7, P)); /// true

    return 0;
}
```