Personal Notebook (2023)

Contents

1 Contest 1
- 1.1 Template . . . 1
- 1.2 .Bashrc . . . 2
- 1.3 Troubleshoot . . . 2
- 1.4 Technical Note . . . 2

2 Data Structures 2
- 2.1 DSU . . . 2
- 2.2 Fenwick Tree . . . 2
- 2.3 Lazy Segment Tree . . . 3
- 2.4 Trie . . . 3
- 2.5 Sparse Table . . . 4
- 2.6 Small To Large Merging . . . 4

3 Graph 4
- 3.1 Dijkstra . . . 4
- 3.2 BellmanFord . . . 4
- 3.3 Floyd Warshall . . . 5
- 3.4 BFS . . . 5
- 3.5 BFS 0-1 . . . 6
- 3.6 Kruskal . . . 6
- 3.7 Prim . . . 6
- 3.8 MST variants . . . 6
- 3.9 Flood Fill . . . 7
- 3.10 Bipartiteness Check . . . 7
- 3.11 Shortest Cycle . . . 7
- 3.12 Kahn Algorithm for TopoSort . . . 7
- 3.13 TopoSort using DFS . . . 7
- 3.14 DP on DAG . . . 8
- 3.15 Cycle Check . . . 8

4 Tree Algorithm 9
- 4.1 General about tree problems . . . 9
- 4.2 Tree Diameter . . . 9
- 4.3 Finding LCA . . . 9
- 4.4 Euler Tour . . . 10
- 4.5 Binary Lifting . . . 10

5 Geometry 10
- 5.1 Point Struct . . . 10
- 5.2 Line Struct . . . 11
- 5.3 Vector Struct . . . 11
- 5.4 Complex number . . . 12
- 5.5 Circle . . . 12
- 5.6 Quadrilaterals . . . 13
- 5.7 Triangle . . . 13
- 5.8 Polygon . . . 13
- 5.9 Convex hull . . . 14

6 Bitwise 14
- 6.1 Bit Manipulation . . . 14
- 6.2 Optimization using Bit . . . 15
- 6.3 Other trick . . . 15

7 STL 15
- 7.1 Vector . . . 15
- 7.2 Unordered Set . . . 16
- 7.3 Tuple . . . 16
- 7.4 String . . . 16
- 7.5 Stack . . . 16
- 7.6 Set . . . 16
- 7.7 Queue . . . 16
- 7.8 Heap (Prority Queue) . . . 16
- 7.9 Order Statistic Tree . . . 17
- 7.10 Multiset . . . 17
- 7.11 Map . . . 17
- 7.12 Deque . . . 17
- 7.13 Custom Comparators . . . 17

8 Number Theory 18
- 8.1 ModInt template . . . 18
- 8.2 Modular Inverse . . . 19
- 8.3 Modular exponentiation . . . 19
- 8.4 Prime . . . 19
- 8.5 GCD, LCM . . . 20
- 8.6 Euler's Totient Function . . . 20

9 Combinatorics 20
- 9.1 Binomial Coefficients . . . 20

10 Other 20
- 10.1 Coordinate Compression . . . 20
- 10.2 Backtracking . . . 20

11 Dynamic Programming 21
- 11.1 Coin Exchange . . . 21
- 11.2 Edit Distance . . . 21
- 11.3 Knapsack . . . 21
- 11.4 LIS . . . 21

12 Divide and Conquer 21
- 12.1 Binary Search . . . 21

1 Contest

1.1 Template

```
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
using pi = pair<int,int>;
using pl = pair<ll,ll>;
using vi = vector<int>;
using vll = vector<ll>;
using iii = tuple<int, int, int>;

#define PB push_back
#define MP make_pair
#define LSOne(S) ((S) & -(S))
#define sz(x) int((x).size())
#define all(x) begin(x), end(x)
#define tcT template<class T

#define FOR(i,a,b) for(int i=(a),_b=(b); i<=_b; i++)
#define FORD(i,a,b) for(int i=(a),_b=(b); i>=_b; i--)
#define REP(i,a) for(int i=0,_a=(a); i<_a; i++)
#define DEBUG(x) { cout << #x << " = "; cout << (x) << endl; }
#define PR(a,n) { cout << #a << " = "; FOR(_,1,n) cout << a[_] << ' '; cout << endl; }
#define PR0(a,n) { cout << #a << " = "; REP(_,n) cout << a[_] << ' '; cout << endl; }

// const int dx[4]{1,0,-1,0}, dy[4]{0,1,0,-1}; // for every grid problem!!

tcT> using pqg = priority_queue<T,vector<T>,greater<T>>; // minheap

tcT> bool ckmin(T& a, const T& b) {
        return b < a ? a = b, 1 : 0; } // set a = min(a,b)
tcT> bool ckmax(T& a, const T& b) {
        return a < b ? a = b, 1 : 0; } // set a = max(a,b)

tcT> T gcd(T a, T b){ T r; while (b != 0) { r = a % b; a = b; b = r; } return a;}
tcT> T lcm(T a, T b) { return a / gcd(a, b) * b; }

ll cdiv(ll a, ll b) { return a/b+((a^b)>0&&a%b); } // divide a by b rounded up
ll fdiv(ll a, ll b) { return a/b-((a^b)<0&&a%b); } // divide a by b rounded down

tcT> T gcd(T a, T b){ T r; while (b != 0) { r = a % b; a = b; b = r; } return a;}
tcT> T lcm(T a, T b) { return a / gcd(a, b) * b; }
```

```cpp
// bitwise ops
constexpr int pct(int x) { return __builtin_popcount(x); } // # of bits set
constexpr int bits(int x) { // assert(x >= 0);
        return x == 0 ? 0 : 31-__builtin_clz(x); } // floor(log2(x))
constexpr int p2(int x) { return 1<<x; }
constexpr int msk2(int x) { return p2(x)-1; }

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);


    return 0;
}
```

1.2 .Bashrc

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
        -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>

g++ -std=c++17 -O2 name.cpp -o name -Wall

open ~/.zshrc and copy:
co() { g++ -std=c++17 -O2 -o "${1%.*}" $1 -Wall; }
run() { co $1 && ./${1%.*} & fg; }

run name.cpp // compile and run
```

1.3 Troubleshoot

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

1.4 Technical Note

```
Submission :
All program source code files and/or test data files which you create must be located in or
beneath your home directory. Your home directory will normally be named /home/team
but this may vary between sites so check with your site coordinator. You may create
subdirectories beneath your home directory.

If your program exits with a non-zero exit code, it will be judged as a Run Time Error.
-----------
Compiling :
1. For C++:
compileg++ progname.cpp (compiling)
To execute a C/C++ program after compiling it as above, type the command : ./a.out

2. For Java:
compilejava Progname.java
runjava Progname

3. For python3
compilepython3 progname.py
runpython3 progname.py
----------
IDE: They can be accessed using the Applications Programming menu.
Other editors: They can be accessed using the Applications Accessories menu.
```

2 Data Structures

2.1 DSU

```cpp
/**
 * Description: Disjoint-set data structure.
 * Time: O(alpha(N)), almost constant
 */

struct DSU {
    vi lab;
    DSU(int n) { lab = vi(n, -1); }

    // get representive component (uses path compression)
    int get(int x) { return lab[x] < 0 ? x : lab[x] = get(lab[x]); }

    bool same_set(int a, int b) { return get(a) == get(b); }

    bool unite(int x, int y) {  // union by size
                x = get(x), y = get(y);
                if (x == y) return false;
                if (lab[x] > lab[y]) swap(x, y);
                lab[x] += lab[y];
                lab[y] = x;
                return true;
        }
};

int main() {
    DSU dsu(node_num);
    return 0;
}
```

2.2 Fenwick Tree

```cpp
/**
 * Description: Computes partial sums a[0] + a[1] + ... + a[pos - 1],
 * and updates single elements a[i],
 * taking the difference between the old and new value.
 * Time: Both operations are $O(log N)$.
 */

// 0 based index
template<
    typename T
> struct FT {
    FT(int _n) : n(_n), f(_n + 1) {}

    // a[u] += val
    void update(int u, T val) {
        assert(0 <= u && u < n);
```

```
        ++u;
        for (; u <= n; u += u & -u) {
            f[u] += val;
        }
    }

    // return a[0] + .. + a[u-1]
    T get(int u) const {
        assert(0 <= u && u <= n);
        T res = 0;
        for (; u > 0; u -= u & -u) {
            res += f[u];
        }
        return res;
    }

    // return a[l] + .. + a[r-1]
    T get(int l, int r) const {
        assert(0 <= l && l <= r && r <= n);
        if (l == r) return 0;  // empty
        return get(r) - get(l);
    }

    void reset() {
        std::fill(f.begin(), f.end(), T(0));
    }

    int lower_bound(T sum) {// min pos st sum of [0, pos] >= sum
                // Returns n if no sum is >= sum, or -1 if empty sum is.
                if (sum <= 0) return -1;
                int pos = 0;
                for (int pw = 1 << 25; pw; pw >>= 1) {
                        if (pos + pw <= sz(f) && f[pos + pw-1] < sum)
                                pos += pw, sum -= f[pos-1];
                }
                return pos;
        }

    int n;
    vector<T> f;
};

int main () {
    FT<ll> ft(n);
}
```

2.3 Lazy Segment Tree

```
/**
 * Description: Segment Tree with lazy propagation
 */

// everything should be 1-based index

// 1 x y val: Increase all elements in [x, y] by val
// 2 x y: max value in [x, y]

struct SegTree {
    int n;  // size of A
    vi A, st, lazy;

    SegTree(int sz) : n(sz), st(4 * n), lazy(4 * n, 0) {}

    SegTree(const vi& initialA, int a_size): SegTree(a_size) {
        A = initialA;
        build(1, 1, n);
    }

    void build(int id, int l, int r) { // O(n)
        if (l == r) {
            st[id] = A[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(2 * id, l, mid);
        build(2 * id + 1, mid + 1, r);
        st[id] = max(st[2 * id], st[2 * id + 1]);
    }

    // propagate
    void fix(int id, int l, int r) {
        if (!lazy[id]) return;
        st[id] += lazy[id]; // careful about how to update info from lazy to node

        if (l != r){ // if not a leaf then propagate downwards
            lazy[2 * id] += lazy[id];
            lazy[2 * id + 1] += lazy[id];
        }
        else {
            A[l] += lazy[id];
        }

        lazy[id] = 0; // erase lazy flag
    }

    void update(int id, int l, int r, int u, int v, int val) { // O(log n)
        fix(id, l, r);
        if (l > v || r < u) return;
        if (l >= u && r <= v) {
            lazy[id] += val;
            fix(id, l, r);
            return;
        }
        int mid = (l + r) >> 1;
        update(2 * id, l, mid, u, v, val);
        update(2 * id + 1, mid + 1, r, u, v, val);
        st[id] = max(st[2 * id], st[2 * id + 1]);
    }

    long long get(int id, int l, int r, int u, int v) { // O(log n)
        fix(id, l, r);
        if (l >  v || r <  u) return -inf;
        if (l >= u && r <= v) return st[id];

        int mid = (l + r) >> 1;
        long long get1 = get(2 * id, l, mid, u, v);
        long long get2 = get(2 * id + 1, mid + 1, r, u, v);
        return max(get1, get2);
    }

    void update(int u, int v, int val) { update(1, 1, n, u, v, val); }
    long long get(int u, int v) { return get(1, 1, n, u, v); }
};

int main()
{
    cin >> n >> m;
    vll A(n + 1, 0);
    SegTree st(A, n);

    while (m--) {
        int t; cin >> t;
        if (t == 0) {
            int u, v, val;
            cin >> u >> v >> val;
            st.update(u, v, val);
        }
        else {
            int x, y;
            cin >> x >> y;
            cout << st.get(x, y) << endl;
        }
    }

    return 0;
}
```

2.4 Trie

```
/**
 * Description: Prefix tree
 * Time: Both operations are O(string length).
 */

struct Trie {
    struct Node {
        Node* child[26];
        int exist, cnt;

        Node() {
            for (int i = 0; i < 26; i++) child[i] = NULL;
            exist = cnt = 0;
        }
    };

    int cur;
    Node* root;
    Trie() : cur(0) {
        root = new Node();
    };

    void add_string(string s) {
        Node* p = root;
        for (auto f : s) {
            int c = f - 'a';
```

```
            if (p->child[c] == NULL) p->child[c] = new Node();
            p = p->child[c];
            p->cnt++;
        }
        p->exist++;
    }

    bool delete_string_recursive(Node* p, string& s, int i) {
        if (i != (int)s.size()) {
            int c = s[i] - 'a';
            bool isChildDeleted = delete_string_recursive(p->child[c], s, i + 1);
            if (isChildDeleted) p->child[c] = NULL;
        }
        else p->exist--;

        if (p != root) {
            p->cnt--;
            if (p->cnt == 0) {
                delete(p);
                return true;
            }
        }
        return false;
    }

    void delete_string(string s) {
        if (find_string(s) == false) return;

        delete_string_recursive(root, s, 0);
    }

    bool find_string(string s) {
        Node* p = root;
        for (auto f : s) {
            int c = f - 'a';
            if (p->child[c] == NULL) return false;
            p = p->child[c];
        }
        return (p->exist != 0);
    }

    void dfs(Node* p, string& cur_string, vector<string>& res) {
        for (int i = 0; i < 26; i++) {
            if (p->child[i] != NULL) {
                cur_string += char(i + 'a');
                dfs(p->child[i], cur_string, res);
                cur_string.pop_back();
            }
        }
    }

    vector<string> sorted_strings() {
        vector<string> res;
        string current_string = "";
        dfs(0, current_string, res);
        return res;
    }
};
```

2.5 Sparse Table

```
/**
 * Description: Sparse Table. LG la so lon nhat thoa 2^LG < N.
 * vi du: N = 10^5 thi LG = 16 vi 2^16 = 65536
 */

// a is 1-based index
int a[N], st[LG + 1][N];
void preprocess() {
    for (int i = 1; i <= n; ++i) st[0][i] = a[i];
    for (int j = 1; j <= LG; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            st[j][i] = min(st[j - 1][i], st[j - 1][i + (1 << (j - 1))]);
}
int queryMin(int l, int r) {
    int k = __lg(r - l + 1);
    return min(st[k][l], st[k][r - (1 << k) + 1]);
}

// Preprocessing: O(NlogN)
// query: O(1)
// __lg(x) = floor(log_2(x)) while log2(x) return double
```

2.6 Small To Large Merging

```
/**
 * Description: Small-to-large merging technique to speed up the process of merging to data structure
 */

/*
Note that swap exchanges two sets in O(1) time. Thus, merging a smaller set of size m into
the larger one of size n takes O(mlogn) time. We can also merge other standard library data structures
such as std::map or std:unordered_map in the same way. However, std::swap does not always
run in O(1) time. For example, swapping std::array s takes time linear in the sum of the sizes
of the arrays, and the same goes for GCC policy-based data structures such as
__gnu_pbds::tree or __gnu_pbds::gp_hash_table. To swap two policy-based data structures a and b
in O(1), use a.swap(b) instead. Note that for standard library data structures, swap(a,b)
is equivalent to a.swap(b) .
*/

// Always merge smaller set to larger set
if (a.size() < b.size()) swap(a, b);
for (int x : b) a.insert(x);
```

3 Graph

3.1 Dijkstra

```
/**
 * Description: Dijkstra algorithm for finding SSSP
 * Time: O(Mlog N)
 */

void dijkstra(int n, int s, vector<vector<Edge>> &E, vector<long long> &D, vector<int> &trace) {
    D.resize(n, INF);
    trace.resize(n, -1);
    vector<bool> P(n, 0);

    D[s] = 0;
    priority_queue<pi, vector<pi>, greater<pi>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        int u = pq.top().second;
        int du = pq.top().first;
        pq.pop();
        if (du != D[u]) continue; // node is already processed or equivalently processed[u] = true
        for (auto e : E[u]) {
            int v = e.v;
            long long w = e.w;

            if (D[v] > D[u] + w) {
                D[v] = D[u] + w;
                pq.push({v, D[v]});
                trace[v] = u;
            }
        }
    }
}

/*
Trick: To find all shortest paths of all nodes to a node A. Simply reverse
the orginal graph and then run dijkstra from node A. Useful when finding shortest path
given the condition that an edge is fixed.
*/
```

3.2 BellmanFord

```
/**
 * Description: Bellman-Frod
 * Time: O(MN)
 */

struct Edge {
    int u, v, w;
    Edge(int u = 0, int v = 0, int w = 0) {
        this->u = u;
        this->v = v;
        this->w = w;
    }
};

vi d(maxn, INF);
```

```
vi path(maxn, -1);
vector<Edge> g;
int n, m;

bool BellmanFord(int s) {
    int u, v, w;
    d[s] = 0;
    for (int i = 1; i <= n - 1; i++) {
        for (int j = 0; j < m; j++) {
            u = g[j].u;
            v = g[j].v;
            w = g[j].w;
            if (d[u] != INF && d[u] + w < d[v]) {
                d[v] = d[u] + w;
                path[v] = u;
            }
        }
    }

    // check if graph contains a negative cycle
    for (int i = 0; i < m; i++) {
        u = g[i].u; v = g[i].v; w = g[i].w;
        if (d[u] != INF && d[u] + w < d[v]) {
            return false;
        }
    }
    return true;
}

vector<int> trace_path(int s, int t) {
    if (t != s && path[t] == -1) return vi(0);

    vi res;
    while (t != -1) {
        res.push_back(t);
        t = path[t];
    }
    reverse(res.begin(), res.end());
    return res;
}

bool findNegativeCycle(vector<int> &negCycle) {
    // after running Bellman-ford n - 1 times
    int negStart = -1;
    for (auto E: g) {
        int u = E.u;
        int v = E.v;
        int w = E.w;
        if (d[u] != INF && d[v] > d[u] + w) {
            d[v] = -INF;
            path[v] = u;
            negStart = v;
        }
    }

    if (negStart == -1) return false; // no negative cycle

    int u = negStart;
    // To get the vertices that are guaranteed to lie in a negative cycle
    for (int i = 0; i < n; i++) {
        u = path[u];
    }

    negCycle = vector<int>(1, u);
    for (int v = path[u]; v != u; v = path[u]) {
        negCycle.push_back(v); // truy vet mot dong
    }
    reverse(negCycle.begin(), negCycle.end());

    return true;
}

int main() {
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.push_back(Edge(u, v, w));
    }

    int s = 0;
    bool res = BellmanFord(s);

    if (!res) {
        // if there is a negative cycle, identify all nodes that does not exist shortest path
        for (int i = 0; i < n; i++) {
            for (auto E : g) {
                int u = E.u;
                int v = E.v;
                int w = E.w;
                if (d[u] != INF && d[v] > d[u] + w) {
                    d[v] = -INF; path[v] = u;
                }
            }
        }
    }

    return 0;
}
```

3.3 Floyd Warshall

```
/**
 * Description: Floyd warshall
 * Time: O(N^3)
 */

void init_trace(vector<vector<int>> &trace) {
    int n = trace.size();
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            trace[u][v] = u;
        }
    }
}

bool floydWarshall(int n, vector<vector<long long>> &w, vector<vector<long long>> &D, vector<vector<
    int>> &trace) {
    D = w; // if d[i][i] = 0, d[i][j] = INF if no edge
    init_trace(trace); // neu can do duong di

    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                // if the graph has negative weight edges, it is better to write the Floyd-Warshall
                    algorithm in the following way, so that it does not perform transitions using
                    paths that don't exist.
                if (D[i][k] < INF && D[k][j] < INF) {
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
                    trace[i][j] = trace[k][j];

                    // make sure that D[i][j] wont be overflow when graph contains negative cycles
                    D[i][j] = max(D[i][j], -INF);
                }
            }
        }
    }

    // check if graph contain negative cycle
    for (int i = 0; i < n; i++) {
        if (D[i][i] < 0) return false;
    }
    return true;
}

vector<int> trace_path(vector<vector<int>> &trace, int u, int v) {
    vector<int> path;
    while (v != u) { // truy vet nguoc tu v ve u
        path.push_back(v);
        v = trace[u][v];
    }
    path.push_back(u);

    reverse(path.begin(), path.end()); // can reverse vi duong di tu v nguoc ve u

    return path;
}
```

3.4 BFS

```
/**
 * Description: BFS
 * Time: O(V + E)
 */

void BFS(int s) {
    queue<int> q;
    visited[s] = true;
    q.push(s);

    while (!q.empty()){
        int u = q.front();
        q.pop();
        for (auto v : g[u]){
            if (!visited[v]){
```

```
                visited[v] = true;
                d[v] = d[u] + 1;
                path[v] = u;
                q.push(v);
            }
        }
    }
}

// Tracing
void printPath(int u) {
    if (!visited[u]) cout << "No path!";
    else {
        vi path;
        for (int v = u; v != -1; v = path[v]){
            path.push_back(v);
        }
        reverse(path.begin(), path.end());
        cout << "Path: ";
        for (auto v: path) cout << v << " ";
    }
}
```

3.5 BFS 0-1

```
/**
 * Description: 0-1 BFS, find shortest path in 0-1 weighted graph.
 * Time: better than Dijkstra
 */

// 0-1 BFS could be use to find the minimum of edges that is needed to
// be reversed in direction to make the path 1->N possible

int n, m;
const int inf = 1e9;
const int maxn = 1e5 + 7;
vector<pi> g[maxn];
int d[maxn];

void sssp(int s) {
    fill_n(d, n + 1, inf);
    deque<int> q;
    q.push_back(s);
    d[s] = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop_front();

        for (auto e: g[u]) {
            int v = e.second;
            int uv = e.first;
            if (d[v] > d[u] + uv) {
                d[v] = d[u] + uv;
                if (uv == 1) {
                    q.push_back(v);
                }
                else {
                    q.push_front(v);
                }
            }
        }
    }
}
```

3.6 Kruskal

```
/**
 * Description: Kruskal Algo
 * Time: if the graph is densed, use Prim for better performance
 */

struct DSU {
    vi lab;
    DSU(int n) { lab = vi(n, -1); }

    int get(int x) { return lab[x] < 0 ? x : lab[x] = get(lab[x]); }

    bool same_set(int a, int b) { return get(a) == get(b); }

    bool unite(int x, int y) {
                x = get(x), y = get(y);
                if (x == y) return false;
                if (lab[x] > lab[y]) swap(x, y);
                lab[x] += lab[y];
                lab[y] = x;
                return true;
        }
};

ll kruskal(int n, vector<pair<ll, pi>> Edges) {
    DSU d(n);
    ll res = 0; int taken = 0;

    sort(Edges.begin(), Edges.end());

    for (auto edge: Edges) {
        int u = edge.second.first;
        int v = edge.second.second;
        if (d.unite(u, v)) {
            res += edge.first;
            taken++;
        }
        if (taken == n - 1) break;
    }
    if (taken != n - 1) return -1;
    return res;
}
```

3.7 Prim

```
/**
 * Description: Prim Algo
 * Time: O(ElogV)
 * Status:
 */

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

vector<vii> AL;                                 // the graph stored in AL
vi taken;                                       // to avoid cycle
priority_queue<ii> pq;                          // to select shorter edges
// C++ STL priority_queue is a max heap, we use -ve sign to reverse order

void process(int u) { // set u as taken and enqueue neighbors of u
  taken[u] = 1;
  for (auto &[v, w] : AL[u])
    if (!taken[v])
      pq.push({-w, -v});                        // sort by non-dec weight
}                                               // then by inc id

int main() {
    int V, E; scanf("%d %d", &V, &E);
    AL.assign(V, vii());
    for (int i = 0; i < E; ++i) {
        int u, v, w; scanf("%d %d %d", &u, &v, &w);  // read as (u, v, w)
        AL[u].emplace_back(v, w);
        AL[v].emplace_back(u, w);
    }
    taken.assign(V, 0);                           // no vertex is taken
    process(0);                                   // take+process vertex 0
    int mst_cost = 0, num_taken = 0;              // no edge has been taken
    while (!pq.empty()) {                         // up to O(E)
        auto [w, u] = pq.top(); pq.pop();           // C++17 style
        w = -w; u = -u;                           // negate to reverse order
        if (taken[u]) continue;                   // already taken, skipped
        mst_cost += w;                            // add w of this edge
        process(u);                               // take+process vertex u
        ++num_taken;                              // 1 more edge is taken
        if (num_taken == V-1) break;              // optimization
    }
    printf("MST cost = %d (Prim's)\n", mst_cost);

}
```

3.8 MST variants

```
/**
 * Description: Some variants of MST problems
 */
```

Minimum Spanning Subgraph of MST problem. Some edges in the given graph have already been fixed and must be taken as part of the solution. For Kruskal's algorithm, we first take into account all the fixed edges and their costs. Then, we continue running Kruskal's algorithm on the remaining free edges until we have a spanning subgraph (or spanning tree).

For Prim's algorithm, we give higher priorities to these fixed edges so that we will always take them and their costs.

Second-Best Spanning Tree of MST problem, We can see that the second best ST is actually the MST with just two edges difference. One edge is taken out from the MST and another chord edge is added into the MST. Next, for each edge in the MST (there are at most V-1 edges in the MST), temporarily flag it so that it cannot be chosen, then try to find the MST again in O(E) but now excluding that flagged edge. Note that we do not have to re-sort the edges at this point. The best spanning tree found after this process is the second best ST.

3.9 Flood Fill

```
/**
 * Description: FloodFill
 */

// const int dx[8]{1,0,-1,0,-1,1,-1,1}, dy[8]{0,1,0,-1,-1,1,1,-1};
// const int dx[4]{1,0,-1,0}, dy[8]{0,1,0,-1};
bool isValid(int r, int c){
    return r >= 0 && r < n && c >= 0 && c < m;
}

int bfs(pi c){
    int size = 1;
    visited[c.f][c.s] = true;
    queue<pi> q;
    q.push(c);
    while (!q.empty()) {
        int x = q.front().first;
        int y = q.front().second;
        q.pop();

        for (int i = 0; i < 4; i++){
            int u = x + dx[i];
            int v = y + dy[i];
            if (isValid(u, v) && a[u][v] == 1 && !visited[u][v]){
                q.push({u, v});
                size++;
                visited[u][v] = true;
            }
        }
    }
    return size;
}
```

3.10 Bipartiteness Check

```
/**
 * Description: Bipartiteness
 * Time: O(V + E)
 */

int n, l;
const int maxn = 210;
vector<vi> g(maxn);
int color[maxn];

// A Bipartite Graph has no odd-length cycle

bool checkBipartite() {
    fill_n(color, n + 1, -1);

    queue<int> q;
    q.push(0);
    color[0] = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : g[u]) {
            if (color[v] == color[u]) return false;
            if (color[v] == -1) {
                color[v] = !color[u];
                q.push(v);
            }
        }
    }
    return true;
}
```

3.11 Shortest Cycle

```
/**
 * Description: Find shortest cycle starting from a source vertex
 */


int bfs(int s) {
    fill_n(d, n + 1, 0);
    fill_n(visit, n + 1, false);

    queue <int> q;
    q.push(s);
    visit[s] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v : g[u]) {

            // If get back to source, return cycle length and end BFS
            if (v == s) return d[u] + 1;

            if (!visit[v]) {
                d[v] = d[u] + 1;
                visit[v] = true;
                q.push(v);
            }
        }
    }
    return 0;
}
```

3.12 Kahn Algorithm for TopoSort

```
/**
 * Description: Kahn Algorithm for Topo sort
 */

vi toposort() {
    vi order;
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        if (indeg[i] == 0) q.push(i);
    }
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        order.push_back(cur);
        for (auto v : g[cur]) {
            indeg[v]--;
            if (indeg[v] == 0) q.push(v);
        }
    }
    return order;
}

// if there isn't a valid topological sorting
if (order.size() != n) {

}

/*
We can also use Kahn's algorithm to extract the lexicographically minimum
topological sort by breaking ties lexographically. Simply replace the queue
with a priority_queue to implement this extension.
*/
```

3.13 TopoSort using DFS

```
/**
 * Description: using DFS to find one valid Topo Sort of a give DAG
 */

/*
Every DAG has at least one (a Singly Linked List-like DAG),
possibly more than one topological sorts, and up to n! topological sorts
(a DAG with n vertices and 0 edge). There is no possible topological ordering of a non DAG.
*/
int n, m;
const int maxn = 1e5 + 7;
vector<vi> g(maxn);
vector<bool> visited(maxn, false);
vi res;
```

```
void dfs(int s) {
    visited[s] = true;
    for (auto u : g[s]) {
        if (!visited[u]) dfs(u);
    }
    res.PB(s);
}

bool toposort() {
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) dfs(i);
    }
    reverse(res.begin(), res.end());

    // check if the graph is actually a DAG
    vi ind(n + 3);
    for (int i = 0; i < n; i++) ind[res[i]] = i + 1;
    for (int i = 1; i <= n; i++) {
        for (auto u : g[i]) {
            if (ind[u] < ind[i]) return false;
        }
    }
    return true;
}

// return true if graph is a DAG, valid topo order is stored in res
```

3.14 DP on DAG

```
/**
 * Description: DP on DAG
 */

// Extending Dijkstra algorithm
/*
A by product of Dijkstra algorithm is a directed, acyclic graph that indicates
for each node of the original graph the possible ways to reach the node using a
shortest path from the starting node. Dynamic programming can be applied to
that graph. For example: calculate the number of shortest paths from node 1
to node 5 using dynamic programming
*/

/*
 If we process the states in topological order, it is guaranteed that dp[u]
 will already have been computed before computing dp[v]
*/

// Example: ordering of node to be processed

vi order = toposort();
dp[1] = 1;
for (auto city : order) {
    for (auto prev : rev_g[city]) {
        ckmax(dp[city], dp[prev] + 1);
    }
}
cout << dp[n] << endl;
```

3.15 Cycle Check

```
/**
 * Description: Cycle Check
 */


/*
We will run a series of DFS in the graph. Initially all vertices are colored white (0).
From each unvisited (white) vertex, start the DFS, mark it gray (1) while entering
and mark it black (2) on exit. If DFS moves to a gray vertex, then we have found a cycle
(if the graph is undirected, the edge to parent is not considered).
The cycle itself can be reconstructed using parent array.
*/

// For directed graph
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u))
                return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}

void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))
            break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle)
            cout << v << " ";
        cout << endl;
    }
}

/*
In undirected graph, another way is to simply calculate the number of nodes and edges in every
component. If a component contains c nodes and no cycle, it must contain exactly c-1 edges
(so it has to be a tree). If there are c or more edges, the component surely contains a cycle.
*/

/*
Here is an implementation for undirected graph. Note that in the undirected version, if a vertex
v gets colored black, it will never be visited again by the DFS. This is because we already
explored all connected edges of v when we first visited it. The connected component containing v
(after removing the edge between v and its parent) must be a tree, if the DFS has completed
processing v without finding a cycle. So we don't even need to distinguish between gray and black
states. Thus we can turn the char vector color into a boolean vector visited.
*/

// undirected graph
int n;
vector<vector<int>> adj;
vector<bool> visited;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v, int par) { // passing vertex and its parent vertex
    visited[v] = true;
    for (int u : adj[v]) {
        if(u == par) continue; // skipping edge to parent vertex
        if (visited[u]) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
        parent[u] = v;
        if (dfs(u, parent[u]))
            return true;
    }
    return false;
}

void find_cycle() {
    visited.assign(n, false);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (!visited[v] && dfs(v, parent[v]))
            break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
```

```
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);

        cout << "Cycle found: ";
        for (int v : cycle)
            cout << v << " ";
        cout << endl;
    }
}
```

4 Tree Algorithm

4.1 General about tree problems

```
/**
 * Description: General about tree
 */

/*
Tree traversal is easier to implement than a general graph, There are no cycles in the
tree and it is not possible to reach a node from multiple directions.

A general way to approach many tree problems is to first root the tree arbitrarily.
After this, we can try to solve the problem separately for each subtree.

An important observation is that every path in a rooted tree has a highest
point: the highest node that belongs to the path.
*/

void dfs(int s, int e) {
    res[s] = 1; // calculate some information during a tree traversal
    for (auto u : g[s]) {
        if (u == e) continue;
        dfs(u, s);
        res[s] += res[u];
    }
}

dfs(x, 0); // first node
```

4.2 Tree Diameter

```
/**
 * Description: Tree Diameter
 */

/*
Run a DFS from any node p. Let a be a node whose distance from node p is maximized.
Run another DFS from node a. Let b be a node whose distance from node a is maximized.
a -> b is a diameter.

the height of each component with root in the left half of the diameter
(i.e., dist(a,d)<dist(d,b)) is at most the distance of the root of the component from
the left end of the diameter. Same statement for the right half of the diameter

For each node i, let's find a node j such that dist(i,j) is maximum.
Claim: j = a or j = b always works.

Most of the times, spamming "the farthest node from each node is one end of the diameter"
and "the height of each component is smaller than the distance to the closest end of the diameter"
is enough to reduce the problem to something simpler. you may need to consider any path of the
tree. There are two cases: the path intersects or doesn't intersect the diameter.
*/



pi dfs(int cur, int par) {
    pi res = {0, cur};
    for (auto u : g[cur]) {
        if (u == par) continue;
        pi tmp = dfs(u, cur);
        tmp.first++;
        ckmax(res, tmp);
    }
    return res;
}
int a = dfs(1, 0).second;
int b = dfs(a, 0).first;
```

4.3 Finding LCA

```
/**
 * Description: Different methods for finding LCA in a tree
 */

// Method 1: Euler tour + RMQ (SegTree / Sparse Table). Node should be numbered from 0
// O(N) preprocessing, O(logN) queries: segtree
// O(NlogN) preprocessing, O(1) queries: Sparse Table (static data)
struct LCA {
    vi height, euler, first, st;
    int n;

    LCA(int _n, int root = 0) {
        n = _n;
        height.resize(n);
        first.resize(n);
        euler.reserve(2 * n);
        dfs(root, -1);
        int m = sz(euler);
        st.resize(4 * m);
        build(1, 0, m - 1);
    }

    // build euler path
    void dfs(int cur, int par, int h = 0) {
        height[cur] = h;
        first[cur] = sz(euler);
        euler.push_back(cur);
        for (auto next : g[cur]) {
            if (next == par) continue;
            dfs(next, cur, h + 1);
            euler.push_back(cur);
        }
    }

    // Segtree
    void build(int id, int l, int r) { // O(n)
        if (l == r) {
            st[id] = euler[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(2 * id, l, mid);
        build(2 * id + 1, mid + 1, r);
        int lchild = st[2 * id], rchild = st[2 * id  + 1];
        st[id] = (height[lchild] < height[rchild]) ? lchild : rchild;
    }

    int get(int id, int l, int r, int u, int v) { // O(log n)
        if (l >  v || r <  u) return -inf;
        if (l >= u && r <= v) return st[id];

        int mid = (l + r) >> 1;
        int get1 = get(2 * id, l, mid, u, v);
        int get2 = get(2 * id + 1, mid + 1, r, u, v);
        if (get1 == -inf) return get2;
        if (get2 == -inf) return get1;
        return height[get1] < height[get2] ? get1 : get2;
    }


    int lca(int u, int v) {
        int left = first[u], right = first[v];
        if (left > right) swap(left, right);
        return get(1, 0, sz(euler) - 1, left, right);
    }
};

// Method 2: Binary Lifting
// init up
void dfs(int u) {
    for (int v : g[u]) {
        if (v == up[u][0]) continue;
        h[v] = h[u] + 1;

        up[v][0] = u;
        for (int j = 1; j < 20; ++j)
            up[v][j] = up[up[v][j - 1]][j - 1];

        dfs(v);
    }
}
```

```cpp
int h[N], up[N][20];
int lca(int u, int v) {
    if (h[u] != h[v]) {
        if (h[u] < h[v]) swap(u, v);

        // find ancestor u' of u in which h(u') = h(v)
        int k = h[u] - h[v];
        for (int j = 0; (1 << j) <= k; ++j)
            if (k >> j & 1) // if jth bit of k is 1
                u = up[u][j];
    }
    if (u == v) return u;

    // Find LCA(u,v)
    int k = __lg(h[u]);
    for (int j = k; j >= 0; --j)
        if (up[u][j] != up[v][j]) // if (1<<j)th ancestors of u and v are different
            u = up[u][j], v = up[v][j];
    return up[u][0];
}

// Properties
/*
1. lca(v1, v2) lies on a shortest path from v1 and v2
2. dist(a, b) = depth(a) + depth(b) - 2depth(c) where c = lca(a, b)
3. if v1 is ancestor of v2 => lca(v1,v2) = v1
*/
```

4.4 Euler Tour

```cpp
/**
 * Description: Euler tour. Using dfs traversal to flatten out tree in an array
 * where each subtree is represented by a range. Use with BIT or segtree for subtree
 * queries
 */

vector<vector<int>> g(maxn);
vector<int> start;
vector<int> en;
int timer = 0;

// tour or tree traversal array will be 0-based (which will be used to construct FT or segtree)
// Node: tree traversal array is just a record of the first occurence of a node in an euler tour/path
void euler_tour(int at, int prev) {
        start[at] = timer++;
        for (int n : g[at]) {
                if (n != prev) { euler_tour(n, at); }
        }
        en[at] = timer;
}

start.resize(node_num);
en.resize(node_num);
euler_tour(0, -1);

/*
[start[i], en[i] - 1] is the subtree root i. en[i] - start[i] is the subtree size
*/

// Generally when doing euler tour, we should use 0-based node
// Complete Euler path
```

4.5 Binary Lifting

```cpp
/**
 * Description: Binary Lifting, find kth ancestor of a node in a tree
 */

int par[N], up[N][17];
void preprocess() {
    for (int u = 1; u <= n; ++u) up[u][0] = par[u];
    for (int j = 1; j < 17; ++j)
        for (int u = 1; u <= n; ++u)
            up[u][j] = up[up[u][j - 1]][j - 1];
}

int ancestor_k(int u, int k) {
    for (int j = 0; (1 << j) <= k; ++j)
        if (k >> j & 1) u = up[u][j];
    return u;
}
```

```cpp
/**
 * Description: find the furthest ancestor of a node in which dist <= x
 * Time: O(N/logN + Q\log^{2}N)
 */

// Algo 1
int dist[N][17];
int calc_dist(int u, int k) {
    int sum = 0;
    for (int j = 0; (1 << j) <= k; ++j)
        if (k >> j & 1) {
            sum += dist[u][j];
            u = up[u][j];
        }
    return sum;
}

    // binary search to find ans
int solve(int u, int x) {
    int lo = 0, hi = h[u], mid, ans = 0;
    while (lo <= hi) {
        mid = (lo + hi) / 2;
        if (calc_dist(u, mid) <= x) {
            ans = mid;
            lo = mid + 1;
        }
        else hi = mid - 1;
    }
    return ancestor_k(u, ans);
}

// Algo 2 (Efficient)
int dist[N][17];
int solve(int u, int x) {
    int now_dist = 0, k = 0;
    for (int j = __lg(h[u]); j >= 0; --j) {
        if (h[u] >= (1 << j) && now_dist + dist[u][j] <= x) {
            now_dist += dist[u][j];
            k |= 1 << j;
            u = up[u][j];
        }
    }
    return u;
}
```

5 Geometry

5.1 Point Struct

```cpp
/**
 * Description: Point Struct
 */

const double INF = 1e9;
const double EPS = 1e-9;

// const double PI = acos(-1.0) or 2.0 * acos(0.0) for alternative to M_PI
double DEG_to_RAD(double d) { return d*M_PI/180.0; }

double RAD_to_DEG(double r) { return r*180.0/M_PI; }

struct point_i {
    int x, y;                                   // use this if possible
    point_i() { x = y = 0; }                    // default constructor
    point_i(int _x, int _y) : x(_x), y(_y) {}   // constructor
};

// check the required precision and set EPS appropriately.
struct point {
    double x, y;                                // if need more precision
    point() { x = y = 0.0; }                    // default constructor
    point(double _x, double _y) : x(_x), y(_y) {}  // constructor
    bool operator < (point other) const {       // override < operator
        if (fabs(x-other.x) > EPS)              // useful for sorting
            return x < other.x;                 // by x-coordinate
        return y < other.y;                     // if tie, by y-coordinate
    }
    // use EPS (1e-9) when testing equality of two floating points
    bool operator == (const 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);            // return double
}

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {              // theta in degrees
    double rad = DEG_to_RAD(theta);                // convert to radian
    return point(p.x*cos(rad) - p.y*sin(rad),
                 p.x*sin(rad) + p.y*cos(rad));
}
```

5.2 Line Struct

```
/**
 * Description: Line Struct
 */

/*
A line in 2D Euclidean space is the set of points whose coordinates satisfy
a given linear equation  ax + by + c = 0. This linear equation has b = 1 for
non-vertical lines and b = 0 for vertical lines unless otherwise stated.
*/

struct line { double a, b, c; };                   // most versatile

// We can compute the line equation if we are given at least two points on that line via the following
      function.
// 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 = {1.0, 0.0, -p1.x};                       // default values
    else {
        double a = -(double)(p1.y-p2.y) / (p1.x-p2.x);
        l = {a, 1.0, -(double)(a*p1.x) - p1.y};      // NOTE: b always 1.0
    }
}

// convert point and gradient/slope to line, not for vertical line
void pointSlopeToLine(point p, double m, line &l) { // m < Inf
    l.a = -m;                                      // always -m
    l.b = 1.0;                                     // always 1.0
    l.c = -((l.a * p.x) + (l.b * p.y));            // compute this
}

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

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

// returns true (+ intersection point p) 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;
}
```

5.3 Vector Struct

```
/**
 * Description: Vector
 */

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

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

vec scale(const vec &v, double s) {
  return vec(v.x*s, v.y*s);
}

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

// return a new point

// returns the dot product of two vectors a and b
double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }

// returns the squared value of the normalized vector
double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

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

// 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) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  // formula: c = a + u*ab
  c = translate(a, scale(ab, u));                  // translate a to c
  return dist(p, c);                               // Euclidean distance
}

// returns the distance from p to the line segment ab defined by
// two points a and b (technically, a has to be different than 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) {                                   // closer to a
    c = point(a.x, a.y);
    return dist(p, a);                             // dist p to a
  }
  if (u > 1.0) {                                   // closer to b
    c = point(b.x, b.y);
    return dist(p, b);                             // dist p to b
  }
  return distToLine(p, a, b, c);                   // use distToLine
}

// returns the cross product of two vectors a and b
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)) > -EPS;
}

// 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;
}

// 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) {
  // this line is perpendicular to l and pass through p
  line perpendicular;
  if (fabs(l.b) < EPS) {                           // vertical line
    ans.x = -(l.c);
    ans.y = p.y;
    return;
  }
  if (fabs(l.a) < EPS) {                           // 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
}
```

5.4 Complex number

```
/**
 * Description: use complex number to represent points and vector
 */

// define x, y as real(), imag()
typedef complex<double> point;
#define x real()
#define y imag()
const double pi = acos(-1.0);
const double EPS = 1e-9;

#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)
#define C(p0, r) P(p0), double r
#define PP(pp) pair<point,point> &pp

// Note: std::complex has issues with integral data types. The library will work
// for simple arithmetic like vector addition and such, but not for polar or abs.
// It will compile but there will be some errors in correctness! The tip then is
// to rely on the library only if you're using floating point data all throughout.

double dot(P(a), P(b)) {
    return real(conj(a) * b);
}

// x1y2-x2y1
// The cross product tells us whether b turns left (positive value),
// does not turn (zero) or turns right (negative value) when it is placed
// directly after a
double cross(P(a), P(b)) {
    return imag(conj(a) * b);
}

point rotate(P(p), double radians = pi / 2, P(about) = point(0,0)) {
    return (p - about) * exp(point(0, radians)) + about;
}
point proj(P(u), P(v)) {
    return dot(u, v) / dot(u, u) * u;
}
point normalize(P(p), double k = 1.0) {
    return abs(p) == 0 ? point(0,0) : p / abs(p) * k;
}
bool parallel(L(a, b), L(p, q)) {
    return abs(cross(b - a, q - p)) < EPS;
}
double ccw(P(a), P(b), P(c)) {
    return cross(b - a, c - b);
}
bool collinear(P(a), P(b), P(c)) { return abs(ccw(a, b, c)) < EPS; }
double angle(P(a), P(b), P(c)) {
    return acos(dot(b - a, c - b) / abs(b - a) / abs(c - b));
}
bool intersect(L(a, b), L(p, q), point &res, bool segment = false) {
    // NOTE: check for parallel/collinear lines before calling this function
    point r = b - a, s = q - p;
    double c = cross(r, s), t = cross(p - a, s) / c, u = cross(p - a, r) / c;
    if (segment && (t < 0-EPS || t > 1+EPS || u < 0-EPS || u > 1+EPS))
        return false;
    res = a + t * r;
    return true;
}
point closest_point(L(a, b), P(c), bool segment = false) {
    if (segment) {
        if (dot(b - a, c - b) > 0) return b;
        if (dot(a - b, c - a) > 0) return a;
    }
    double t = dot(c - a, b - a) / norm(b - a);
    return a + t * (b - a);
}

typedef vector<point> polygon;
#define MAXN 1000
point hull[MAXN];
bool cmp(const point &a, const point &b) {
    return abs(real(a) - real(b)) > EPS ?
        real(a) < real(b) : imag(a) < imag(b); }
int convex_hull(vector<point> p) {
    int n = p.size(), l = 0;
    sort(p.begin(), p.end(), cmp);
    for (int i = 0; i < n; i++) {
        if (i > 0 && p[i] == p[i - 1])
            continue;
        while (l >= 2 && ccw(hull[l - 2], hull[l - 1], p[i]) >= 0)
            l--;
        hull[l++] = p[i];
    }
    int r = l;
    for (int i = n - 2; i >= 0; i--) {
        if (p[i] == p[i + 1])
            continue;
        while (r - l >= 1 && ccw(hull[r - 2], hull[r - 1], p[i]) >= 0)
            r--;
        hull[r++] = p[i];
    }
    return l == 1 ? 1 : r - 1;
}

int main() {
        point a = 2;
        point b(3, 7);
        cout << a << ' ' << b << endl; // (2, 0) (3, 7)
        cout << a + b << endl;         // (5, 7)

        point v = {3,1};
        point u = {2,2};
        point s = v+u;
        cout << s.x << " " << s.y << "\n"; // 5 3

        point a = {4,2};
        point b = {3,-1};
        cout << abs(b-a) << "\n"; // distance between points

        // A vector can be rotated by an angle a by multiplying it by a vector with length 1 and angle
            a.
        P v = {4,2};
        cout << arg(v) << "\n"; // 0.463648
        v *= polar(1.0,0.5);
        cout << arg(v) << "\n"; // 0.963648
}

// If the input coordinates are integer,
// most computations can be done in integers,
// but the result may be real (for example, in line intersection, as we'll see later).
// So, it's useful to have different point types in one program. And here's the first advice:
// do computations in integers as long as possible, for two obvious reasons:
// they are faster; they avoid precision issues.
```

5.5 Circle

```
/**
 * Description: Circle
 */

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 ? 1 : Euc == rSq ? 0 : -1;    // inside/border/outside
}

// constant pi
// M_PI in C++ <cmath> library. or use PI = arccos(-1.0) or PI = 2 * arccos(0.0).
// Length of arc: alpha*c/360.0 (c is circum)
// Length of chord: sqrt(2*r^2*(1-cos(alpha))) or 2*r*sin(alpha/2)

/*
Sector of a circle is defined as a region of the circle enclosed by two radii and an arc
lying between the two radii. A circle with area A and a central angle alpha (in degrees)
has the corresponding sector area alpha*A/360.0
*/

/*
Given 2 points on the circle (p1 and p2) and radius r of the corresponding circle, we can determine
the location of the centers (c1 and c2) of the two possible circles
*/
bool circle2PtsRad(point p1, point p2, double r, point &c) {
  // to get the other center, reverse p1 and p2
  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 check if three line segments of length a, b and c can form a triangle, we can simply
check these triangle inequalities: (a+b>c)&&(a+c>b)&&(b+c>a).
If the three lengths are sorted, with a being the smallest and c the largest, then we c
an simplify the check to just (a + b > c).
*/
```

5.6 Quadrilaterals

```
/**
 * Description: Quadrilaterals
 */

// Given a rectangle described with its bottom left corner (x, y) plus its width w and height h,
// we can use the following checks to determine if another point (a, b) is inside, at the border,
// or outside this rectangle:
int insideRectangle(int x, int y, int w, int h, int a, int b)
{
    if ((x < a) && (a < x + w) && (y < b) && (b < y + h))
        return 1; // strictly inside
    else if ((x <= a) && (a <= x + w) && (y <= b) && (b <= y + h))
        return 0; // at border
    else
        return -1; // outside
}
```

5.7 Triangle

```
/**
 * Author: Ulf Lundstrom
 * Date: 2009-02-26
 * License: CC0
 * Source: My head with inspiration from tinyKACTL
 * Description: Triangle
 * Status: Works fine, used a lot
 */

/*
To check if three line segments of length a, b and c can form a triangle, we can simply
check these triangle inequalities: (a+b>c)&&(a+c>b)&&(b+c>a).
If the three lengths are sorted, with a being the smallest and c the largest, then we can
simplify the check to just (a + b > c).
*/

// Heron: A = sqrt(s(s-a)(s-b)(s-c)) with s is nua chu vi

// A triangle with area A and semi-perimeter s has an inscribed circle (incircle) with radius r = A/s.
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)); }

// The center of incircle is the meeting point between the triangle's angle bisectors.
// We can get the center if we have two angle bisectors and find their intersection point
// 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; }

// A triangle with 3 sides: a, b, c and area A has a circumscribed circle (circumcircle)
// with radius R = abc/(4A).
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)); }

// The center of circumcircle is the meeting point between the triangle's perpendicular bisectors
```

5.8 Polygon

```
/**
 * Description: Polygon
 */

// Polygon representation
// 6(+1) points, entered in counter clockwise order, 0-based indexing
vector<point> P;
P.emplace_back(1, 1);                           // P0
P.emplace_back(3, 3);                           // P1
P.emplace_back(9, 1);                           // P2
P.emplace_back(12, 4);                          // P3
P.emplace_back(9, 7);                           // P4
P.emplace_back(1, 7);                           // P5
P.push_back(P[0]);                              // loop back, P6 = P0

// returns the perimeter of polygon P, which is the sum of
// Euclidian distances of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {      // by ref for efficiency
  double ans = 0.0;
  for (int i = 0; i < (int)P.size()-1; ++i)     // note: P[n-1] = P[0]
    ans += dist(P[i], P[i+1]);                  // as we duplicate P[0]
  return ans;
}

/*
The signed area A of a (convex or concave) polygon with n vertices given in some order
(either clockwise or counter-clockwise)
*/
// returns the area of polygon P
double area(const vector<point> &P) {
  double ans = 0.0;
  for (int i = 0; i < (int)P.size()-1; ++i)     // Shoelace formula
    ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
  return fabs(ans)/2.0;                         // only do / 2.0 here
}

// returns the area of polygon P, which is half the cross products
// of vectors defined by edge endpoints
double area_alternative(const vector<point> &P) {
  double ans = 0.0; point O(0.0, 0.0);          // O = the Origin
  for (int i = 0; i < (int)P.size()-1; ++i)     // sum of signed areas
    ans += cross(toVec(O, P[i]), toVec(O, P[i+1]));
  return fabs(ans)/2.0;
}


// Checking if a Polygon is Convex
/*
We can simply check whether all three consecutive vertices of the polygon form the same turns
(all left turns/ccw if the vertices are listed in counterclockwise order-the default setting-or
all right turns/cw if the vertices are listed in clockwise order). If we can find at least one
triple where this is false, then the polygon is concave.
*/

// 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 n = (int)P.size();
  // a point/sz=2 or a line/sz=3 is not convex
  if (n <= 3) return false;
  bool firstTurn = ccw(P[0], P[1], P[2]);       // remember one result,
  for (int i = 1; i < n-1; ++i)                 // compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) != firstTurn)
      return false;                             // different -> concave
  return true;                                  // otherwise -> convex
}

// Checking if a Point is Inside a Polygon
// returns 1/0/-1 if point p is inside/on (vertex/edge)/outside of
// either convex/concave polygon P
int insidePolygon(point pt, const vector<point> &P) {
  int n = (int)P.size();
  if (n <= 3) return -1;                        // avoid point or line
  bool on_polygon = false;
  for (int i = 0; i < n-1; ++i)                 // on vertex/edge?
    if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) - dist(P[i], P[i+1])) < EPS)
      on_polygon = true;
  if (on_polygon) return 0;                     // pt is on polygon
  double sum = 0.0;                             // first = last point
  for (int i = 0; i < n-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(sum) > M_PI ? 1 : -1;             // 360d->in, 0d->out
}

// compute the intersection point between line segment p-q and line A-B
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y-A.y, b = A.x-B.x, 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 (order matters)
// (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
    if (left1*left2 < -EPS)                      // 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());                      // wrap around
  return P;
}
```

5.9 Convex hull

```
/**
 * Description: Finding the Convex Hull of a Set of Points.
 * The Convex Hull of a set of points Pts is the smallest
 * convex polygon CH(Pts) for which each point in Pts is either
 * on the boundary of CH(Pts) or in its interior.
 */


// Graham Scan
vector<point> CH_Graham(vector<point> &Pts) {    // overall O(n log n)
  vector<point> P(Pts);                          // copy all points
  int n = (int)P.size();
  if (n <= 3) {                                  // point/line/triangle
    if (!(P[0] == P[n-1])) P.push_back(P[0]);    // corner case
    return P;                                    // the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = min_element(P.begin(), P.end())-P.begin();
  swap(P[0], P[P0]);                             // swap P[P0] with P[0]

  // second, sort points by angle around P0, O(n log n) for this sort
  sort(++P.begin(), P.end(), [&](point a, point b) {
    return ccw(P[0], a, b);                      // use P[0] as the pivot
  });

  // third, the ccw tests, although complex, it is just O(n)
  vector<point> S({P[n-1], P[0], P[1]});         // initial S
  int i = 2;                                     // then, we check the rest
  while (i < n) {                                // n > 3, O(n)
    int j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i]))                 // CCW turn
      S.push_back(P[i++]);                       // accept this point
    else                                         // CW turn
      S.pop_back();                              // pop until a CCW turn
  }
  return S;                                      // return the result
}

// Monotone Chain algorithm
vector<point> CH_Andrew(vector<point> &Pts) {    // overall O(n log n)
  int n = Pts.size(), k = 0;
  vector<point> H(2*n);
  sort(Pts.begin(), Pts.end());                  // sort the points by x/y
  for (int i = 0; i < n; ++i) {                  // build lower hull
    while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  for (int i = n-2, t = k+1; i >= 0; --i) {       // build upper hull
    while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  H.resize(k);
  return H;
}
```

6 Bitwise

6.1 Bit Manipulation

```
/**
 * Description: Bitwise Trick
 */


/*
x << k corresponds to multiplying x by 2k
x >> k corresponds to dividing x by 2k rounded down to an integer.

x & 1 = 1 if x is odd, = 0 if x is even
x | (1 << k): sets the kth bit of x to one
x & (2^k-1) = 0 <=> x is divisible by 2^k
~x = -x-1
x << k: multiplying x by 2k
x >> k: dividing x by 2k rounded down to an integer.
x &~(1 << k): sets the kth bit of x to zero
x ^ (1 << k): inverts the kth bit of x.
x & (x-1): sets the last one bit of x to zero
x & -x: sets all the one bits to zero, except for the last one bit
x | (x-1): inverts all the bits after the last one bit

T = ((S) & -(S)): get the value of the least significant bit of S that is on.
T = LSOne(S) is a power of 2, i.e., 2^j. To get the actual index j (from the right),
we can use __builtin_ctz(T)

a positive number x is a power of two exactly when x & (x-1) = 0.

111..1 (k one bits) = 2^k-1

__builtin_clz(x): the number of zeros at the beginning of the number
__builtin_ctz(x): the number of zeros at the end of the number
__builtin_popcount(x): the number of ones in the number
__builtin_parity(x): the parity (even or odd) of the number of ones
there are also long long versions of the functions available with the suffix ll.
cout << bitset<8>(x); prints a number after converting it into a bitset, which can be printed
*/


// print a number in binary format.
string to_binary(int x) {
        string s;
        while(x > 0) {
                s += (x % 2 ? '1' : '0');
                x /= 2;
        }
        reverse(s.begin(), s.end());
        return s;
}



// Representing sets
/*
The following code declares an int variable x that can contain a subset of
{0,1,2,...,31}. After this, the code adds the elements 1, 3, 4 and 8 to the set
and prints the size of the set.
*/
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4

for (int i = 0; i < 32; i++) {
    if (x&(1<<i)) cout << i << " ";
}
// output: 1 3 4 8

// Set operations
// Intersection: a & b
// Union: a | b
// Complement: ~a
// Difference: a&(~b)

//The following code first constructs the sets x = {1,3,4,8} and
// y = {3,6,8,9}, and then constructs the set z = xUy = {1,3,4,6,8,9}:
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6

// goes through the subsets of {0,1,...,n-1}:
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}

// goes through the subsets with exactly k elements:
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
```

```
}

// goes through the subsets of a set x:
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);

// Note: 2^k is just 1 << k or 1LL << k if you need long longs
```

6.2 Optimization using Bit

```
/**
 * Description: Optimization trick
 */

// Hamming distances
/*
Given a list of n bit strings, each of length k,
calculate the minimum Hamming distance between two strings in the list.
The Hamming distance hamming(a,b) between two strings a and b of equal
length is the number of positions where the strings differ.

if k is small, we can optimize the code by storing the bit strings
as integers and calculating the Hamming distances using bit operations. In
particular, if k <= 32, we can just store the strings as int values and use the
following function to calculate distances
*/

int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}

// Count intersection of two sets (O(1))
__builtin_popcount(a & b)

// For bigger set: Use unsigned long long. we can simulate a long binary number
//  with multiple unsigned long longs. The implementation isn't that bad but doing
// binary shifts would be quite ugly. Turns out all of this can be done with bitsets easily.
// bitset<365> is a binary number with 365 bits available, and it supports most of binary operations.
bitset<MAX_D> x[MAX_N];
int intersection(int i, int j) {
        return (x[i] & x[j]).count();
}
/*
construct a bitset from int or string bitset<100> a(17); bitset<100> b("1010");.
You can even access particular bits with b[i]
*/


// given a sequence of N <= 10^7 numbers, each from interval [0,10^9]. How many
// different values appear in the sequence? Don't use set or unordered_set because they quite slow.
// Create bitset<1000000001> visited, mark every given number visited[x] = 1, and print
// visited.count(). The time complexity is O(N + MAX_X/32), space is O(MAX_X/32). This will use
// 128 MB memory (one billion bits).
```

6.3 Other trick

```
/**
 * Description: other bitwise trick
 */

a + b = 2*(a & b) + (a^b)
/*
Proof:
a^b is essentially just a + b in base 2 but we never carry over to the next bit. Recall a bit
in a^b is 1 only if the bit in a is different from the bit in b, thus one of them must be a 1
. However, when we xor two 1 bits, we yield a 0, but we do not carry that 1 to the next bit.
This is where a&b comes in. a&b is just the carry bits themselves, since a bit is 1 only if
it's a 1 in both a and b, which is exactly what we need. We multiply this by 2 to shift all
the bits to the left by one, so every value carries over to the next bit.
*/

a ^ b = ~(a & b) & (a | b)

// addition
// If we perform addition without carrying, then we are simply performing the XOR operator.
// Then, the bits that we carry over are those equivalent to 1 in both numbers: a&b
int add(int a, int b) {
        while (b > 0) {
                int carry = a & b;
                a ^= b;
                b = carry << 1;
        }
        return a;
}

// multiplication
int prod(int a, int b) {
        int c = 0;
        while (b > 0) {
                if ((b & 1) == 1) {
                        c = add(c, a);  // Use the addition function we coded previously
                }
                a <<= 1;
                b >>= 1;
        }
        return c;
}

// Biggest power of 2 that is a divisor of x
1 << __builtin_ctz(x)

// The smallest power of 2 that is not smaller than x but this is UB (undefined behavior) for x <= 1
// so you'll often need an if for x == 1
1 << (32 - __builtin_clz (x - 1))
```

7 STL

7.1 Vector

```
/**
 * Description: Vector and iterator
 */

vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]

vector<int> v = {2,4,2,5,1};
vector<int> a(8); // size 8, initial value 0
vector<int> b(8,2); // size 8, initial value 2

for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}

for (auto x : v) {
    cout << x << "\n";
}

vector<int> v = {2,4,2,5,1};
cout << v.back() << "\n"; // 1
v.pop_back();
cout << v.back() << "\n"; // 5

// An iterator is a variable that points to an element of a data structure.
// The iterator begin points to the first element of a data structure, and the
// iterator end points to the position after the last element. A range is a
// sequence of consecutive elements in a data structure. The usual way to specify
// a range is to give iterators to its first element and the position after its last element.

sort(v.begin(),v.end());
reverse(v.begin(),v.end());
random_shuffle(v.begin(),v.end());

cout << *v.begin() << "\n";

vector<int> v = {2,3,3,5,7,8,8,8};
// If there is no such element, the functions return an iterator to the element after the last element
      in the range.
auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);
cout << *a << " " << *b << "\n";

// creates a vector that contains the unique elements of the original vector in a sorted order:
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());

// reverse iterator
sort(A.rbegin(), A.rend());

// insert val to position of iterator
vi::iterator it;
it = v.begin() + 2;
v.insert(it, 9);
v.erase(it); // erase an element
```

```
v.clear(); // delete all element
```

7.2 Unordered Set

```
/**
 * Description: In unordered sets, elements are stored in an arbitrary order through hashing.
 * Insertions, deletions, and searches are all O(1) (with a high constant factor).
 * unordered set actually has worst case O(N). In any case, just default to using ordered sets.
 * Unordered sets work with primitive types, but require a custom hash function for structures/classes
 * like vectors and pairs.
 */

unordered_set<int> s;
s.insert(1);                    // {1}
s.insert(4);                    // {1, 4}
s.insert(2);                    // {1, 4, 2}
s.insert(1);                    // does nothing because 1's already in the set
cout << s.count(1) << endl;  // 1
s.erase(1);                     // {2, 4}
cout << s.count(5) << endl;  // 0
s.erase(0);                     // does nothing because 0 wasn't in the set
```

7.3 Tuple

```
/**
 * Description: Tuple
 */

// tuple has built-in comparison function
typedef tuple<int, string, string> iss; // combine the 3 fields
auto &[ageA, lastA, firstA] = A; // decompose the tuple

vector<iii> EL(E);
for (auto &[w, u, v] : EL) // C++17 style

tuple <char, int, float> geek;
geek = make_tuple('a', 10, 15.5);
cout << get<0>(geek) << " " << get<1>(geek);
get<0>(geek) = 'b';
get<2>(geek) =  20.5;

int i_val;
char ch_val;
float f_val;

// Initializing tuple
tuple<int,char,float> tup1(20,'g',17.5);

// Use of tie() without ignore
tie(i_val,ch_val,f_val) = tup1;
// ignores char value
tie(i_val,ignore,f_val) = tup1;
```

7.4 String

```
/**
 * Description: String
 */

/*
size()/length() : return size
empty() : check empty string
clear() : delete string
insert(pos, string)
erase(pos, len)
find(string)
substr(pos, len)
append(string, pos, len)
*/

int result = isalpha(char c)
int result = isdigit(char c)
int result = islower(char c)
int result = isupper(char c)

string s("12");
// atof(char *str)
int number = atoi(s.c_str()); // turn string into integer. For double, use atof
s = to_string(number);
char c = toupper(s[2]); // tolower is the same
```

7.5 Stack

```
/**
 * Description: Stack
 */

stack<int> s;
s.push(2); // [2]
s.push(5); // [2,5]
cout << s.top() << "\n"; // 5
s.pop(); // [2]
cout << s.top() << "\n"; // 2
```

7.6 Set

```
/**
 * Description: In sorted sets, the elements are sorted in order of element.
 * Insertions, deletions, and searches are all O(logN)
 * all the essential operations that unordered set has
 */

set<int> s;
s.insert(1);                        // [1]
s.insert(14);                       // [1, 14]
s.insert(9);                        // [1, 9, 14]
s.insert(2);                        // [1, 2, 9, 14]

// lower_bound: returns an iterator to the least element greater than or equal to some element k.
// upper_bound: returns an iterator to the least element strictly greater than some element k.
// if the requested element does not exist, the return value is end().
cout << *s.upper_bound(7) << '\n';  // 9
cout << *s.upper_bound(9) << '\n';  // 14
cout << *s.lower_bound(5) << '\n';  // 9
cout << *s.lower_bound(9) << '\n';  // 9
cout << *s.begin() << '\n';         // 1
auto it = s.end();
cout << *(--it) << '\n';    // 14
s.erase(s.upper_bound(6));  // [1, 2, 14]

for (int element : s) { cout << element << endl; }

auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}

// Real Runtime: Set > unordered_set > normal sorting
```

7.7 Queue

```
/**
 * Description: Queue
 */

queue<int> q;
q.push(2); // [2]
q.push(5); // [2,5]
cout << q.front() << "\n"; // 2
q.pop(); // [5]
cout << q.back() << "\n"; // 5
```

7.8 Heap (Prority Queue)

```
/**
 * Description: A priority queue (or heap) supports the following operations: insertion
 * of elements, deletion of the element considered highest priority, and retrieval
 * of the highest priority element, all in O(logN), retrieval takes O(1) time.
 */

priority_queue<int> pq;
pq.push(7);                 // [7]
```

```
pq.push(2);                 // [2, 7]
pq.push(1);                 // [1, 2, 7]
pq.push(5);                 // [1, 2, 5, 7]
cout << pq.top() << endl;  // 7
pq.pop();                   // [1, 2, 5]
pq.pop();                   // [1, 2]
pq.push(6);                 // [1, 2, 6]

priority_queue<int,vector<int>,greater<int>> q; // min heap

// Note: Priority queue is about five times faster than a multiset.
```

7.9 Order Statistic Tree

```
/**
 * Description: The one that we will use to solve the selection and ranking problems easily.
 * A data structure that supports all the operations as a set in C++ in addition to the following:
 * 1. orderofkey(x): counts the number of elements in the set that are strictly less than x.
 * 2. findbyorder(k): similar to find, returns the iterator corresponding to the k-th lowest
 * element in the set (0-indexed).
 */


#include <bits/stdc++.h>
using namespace std;

#include <bits/extc++.h>                    // pbds
// #include <ext/pb_ds/assoc_container.hpp> // Common file
// #include <ext/pb_ds/tree_policy.hpp>     // Including tree_order_statistics_node_update

using namespace __gnu_pbds;
// for multiset, change less to less_equal
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int main() {
    ordered_set X;
    // X = {1, 2, 4, 8, 16}

    for(int i = 1; i <= 16; i *= 2)
        X.insert(i); // O(n log n)

    // O(log n) select
    cout << *X.find_by_order(0) << endl; // 1 , 1-smallest = 1
    cout << *X.find_by_order(1) << endl; // 2
    cout << *X.find_by_order(2) << endl; // 4
    cout << *X.find_by_order(4) << endl; // 16, 5-smallest or largest = 16
    cout << (X.end()==X.find_by_order(6)) << endl; // true

    // O(log n) rank
    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
}
```

7.10 Multiset

```
/**
 * Description: A multiset is a sorted set that allows multiple copies of the same element.
 */


// Note that the functions count and erase have an additional O(k) factor
//  where k is the number of elements counted/removed.
// To remove a value once, use ms.erase(ms.find(val)).
// To remove all occurrences of a value, use ms.erase(val).


multiset<int> ms;
ms.insert(1);                   // [1]
ms.insert(14);                  // [1, 14]
ms.insert(9);                   // [1, 9, 14]
ms.insert(2);                   // [1, 2, 9, 14]
ms.insert(9);                   // [1, 2, 9, 9, 14]
ms.insert(9);                   // [1, 2, 9, 9, 9, 14]
cout << ms.count(4) << '\n';   // 0
cout << ms.count(9) << '\n';   // 3
cout << ms.count(14) << '\n';  // 1
ms.erase(ms.find(9));
cout << ms.count(9) << '\n';  // 2
ms.erase(9);
cout << ms.count(9) << '\n';  // 0
```

7.11 Map

```
/**
 * Description: A map is a set of entries, each consisting of a key and a value.
 * In a map, all keys are required to be unique, but values can be repeated.
 * Maps have three primary methods: add, retrive, remove.
 * In sorted maps, the pairs are sorted in order of key.
 * Insertions, deletions, and searches are all O(logN)
 * In unordered maps, the pairs aren't kept in sorted order
 * and all insertions, deletions, and searches are all O(1)
 */

map<int, int> m;
// unordered_map<int, int> m;
m[1] = 5;                       // [(1, 5)]
m[3] = 14;                      // [(1, 5); (3, 14)]
m[2] = 7;                       // [(1, 5); (2, 7); (3, 14)]
m[0] = -1;                      // [(0, -1); (1, 5); (2, 7); (3, 14)]
m.erase(2);                     // [(0, -1); (1, 5); (3, 14)]
cout << m[1] << endl;          // 5
cout << m.count(7) << endl;  // 0
cout << m.count(1) << endl;  // 1
cout << m[2] << endl;          // 0

for (const auto &x : m) { cout << x.first << " " << x.second << endl; }

for (auto &x : m) {
        x.second = 1234;  // Change all values to 1234
}

// Note: it is generally a bad idea to insert or remove elements of a map while iterating over it.
// One way to get around this is to just create a new map instead of removing from the old one.
// Another is to maintain a list of all the keys you want to erase and erase them after
// the iteration finishes

map<int, int> m;
m[3] = 5;      // [(3, 5)]
m[11] = 4;     // [(3, 5); (11, 4)]
m[10] = 491;  // [(3, 5); (10, 491); (11, 4)]
cout << m.lower_bound(10)->first << " " << m.lower_bound(10)->second << '\n';  // 10 491
cout << m.upper_bound(10)->first << " " << m.upper_bound(10)->second << '\n';  // 11 4
m.erase(11);   // [(3, 5); (10, 491)]
if (m.upper_bound(10) == m.end()) {
        cout << "end" << endl;  // Prints end
}

// Note: unordered_map is about three times faster than map, an array is almost a hundred times faster
      .
```

7.12 Deque

```
/**
 * Description: Deque
 */

deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

7.13 Custom Comparators

```
/**
 * Description: Comparators in sorting and ordered STL DS
 */

// Essentially, the comparator determines whether object x belongs to the left
// of object y in a sorted ordering.

// Method 1: Overloading the Less Than Operator. Only works for objects (not primitives)
struct Edge {
        int a, b, w;
        bool operator<(const Edge &y) { return w < y.w; }
};

// We can also overload the operator outside of the class:
struct Edge {
```

```
        int a, b, w;
};
bool operator<(const Edge &x, const Edge &y) { return x.w < y.w; }

// Method 2: Comparison Function. Works for both objects and primitives,
// can declare many different comparators for the same object.
struct Edge {
        int a, b, w;
};

// If object x is less than object y , return true
bool cmp(const Edge &x, const Edge &y) { return x.w < y.w; }
sort(begin(v), end(v), cmp);

sort(begin(v), end(v), [](const Edge &x, const Edge &y) { return x.w < y.w; }); // lambda expression


// C++ Sets with Custom Comparators
/*
make sure to include the second const or you'll get a compilation error.
[The second const] means you cannot modify member variables of the current object.
*/
struct Edge {
        int a, b, w;
        bool operator<(const Edge &y) const { return w < y.w; }
};

int main() {
        int M = 4;
        set<Edge> v;
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.insert({a, b, w});
        }
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}

// With a function
struct Edge {
        int a, b, w;
};

bool cmp(const Edge &x, const Edge &y) { return x.w < y.w; }

int main() {
        int M = 4;
        set<Edge, bool (*)(const Edge &, const Edge &)> v(cmp);
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.insert({a, b, w});
        }
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}
// You can also use the following syntax to declare set v using a function:
set<Edge,decltype(&cmp)> v(cmp);

// With lambda expression
auto cmp = [](const Edge &x, const Edge &y) { return x.w < y.w; };

int main() {
        int M = 4;
        set<Edge, bool (*)(const Edge &, const Edge &)> v(cmp);
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.insert({a, b, w});
        }
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}
// You can also use the following syntax to declare set v using a lambda
// even though decltype(cmp) is not actually equivalent to bool(*)(const Edge&,const Edge&).
set<Edge,decltype(cmp)> v(cmp);

// Functors. One functor can be used for multiple objects.
struct Edge {
        int a, b, w;
};

struct cmp {
        bool operator()(const Edge &x, const Edge &y) const { return x.w < y.w; }
};

int main() {
        int M = 4;
        set<Edge, cmp> v;
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.insert({a, b, w});
        }
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}

// We can also use cmp like a normal function by adding () after it.
int main() {
        int M = 4;
        vector<Edge> v;
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.push_back({a, b, w});
        }
        sort(begin(v), end(v), cmp());
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}

// Built-in functor
// Overloading the less than operator (<) automatically generates the functor less<Edge>.
// Similarly, overloading (>) automatically generates the functor greater<Edge>.
// We can use this to store a set in reverse order.

struct Edge {
        int a, b, w;
        bool operator>(const Edge &y) const { return w > y.w; }
};

int main() {
        int M = 4;
        set<Edge, greater<Edge>> v;
        for (int i = 0; i < M; ++i) {
                int a, b, w;
                cin >> a >> b >> w;
                v.insert({a, b, w});
        }
        for (Edge e : v) cout << e.a << " " << e.b << " " << e.w << "\n";
}

/* Output:
2 3 10
1 2 9
1 3 7
2 4 3
*/

// other Containers
set<int, greater<int>> a;
map<int, string, greater<int>> b;
priority_queue<int, vector<int>, greater<int>> c;
```

8 Number Theory

8.1 ModInt template

```
/**
 * Description: templates that implement integer types that automatically
 * wrap around when they exceed a certain modulus
 */

/**
 * Description: Modular arithmetic. Assumes $MOD$ is prime.
 * Source: Benq Template
 * Verification: https://open.kattis.com/problems/modulararithmetic
 * Usage: mi a = MOD+5; inv(a); // 400000003
 */

#include <bits/stdc++.h>
using namespace std;

const int MOD = 1e9 + 7;

struct mi {
        int v;
        explicit operator int() const { return v; }
        mi() { v = 0; }
        mi(long long _v) : v(_v % MOD) { v += (v < 0) * MOD; }
};
mi &operator+=(mi &a, mi b) {
        if ((a.v += b.v) >= MOD) a.v -= MOD;
        return a;
}
mi &operator-=(mi &a, mi b) {
        if ((a.v -= b.v) < 0) a.v += MOD;
        return a;
}
mi operator+(mi a, mi b) { return a += b; }
```

```
mi operator-(mi a, mi b) { return a -= b; }
mi operator*(mi a, mi b) { return mi((long long)a.v * b.v); }
mi &operator*=(mi &a, mi b) { return a = a * b; }
mi pow(mi a, long long p) {
        assert(p >= 0);
        return p == 0 ? 1 : pow(a * a, p / 2) * (p & 1 ? a : 1);
}
mi inv(mi a) {
        assert(a.v != 0);
        return pow(a, MOD - 2);
}
mi operator/(mi a, mi b) { return a * inv(b); }
// EndCodeSnip

int main() {
        {
                int a = 1e8, b = 1e8, c = 1e8;
                cout << (long long)a * b % MOD * c % MOD << "\n";  // 49000000
        }
        {
                mi a = 1e8, b = 1e8, c = 1e8;
                // cout << a * b * c << "\n";  // Errors- we have to cast this an an int
                cout << (int)(a * b * c) << "\n";  // 49000000
        }
}

// Kactl template is another option
```

8.2 Modular Inverse

```
/**
 * Description: Finding modular inverse
 * Time: O(logN)
 */

// Note: Nghich dao cua b theo modulo m ton tai khi b va m la nguyen to cung nhau
// O(log(m))
// If the modular inverse of the same number(s) is/are being used many times,
// it is a good idea to precalculate it.

// Find mod inverse, using fermat little theorem
int modInverse(int b, int m) {
    int res = modpow(b, m - 2, m);
    if ((res * b) % m == 1) return res;
    return -1; // inverse doesnt exist
}


// Using extended Euclidean algorithm
vi extendedEuclid(int b, int m) {
    vi res;
    int q, r;
    int x1 = 1, y1 = 0;
    int x2 = 0, y2 = 1;
    int x3 = 1, y3 = 0;
    while (m != 0) {
        q = b / m;
        r = b % m;
        x3 = x1 - q * x2;
        y3 = y1 - q * y2;
        x1 = x2, y1 = y2;
        x2 = x3, y2 = y3;
        b = m, m = r;
    }
    res.PB(b); res.PB(x1); res.PB(y1);
    return res;
}

void modInverse(int b, int m) {
    vi res = extendedEuclid(b, m);
    int gcd = res[0];
    int x = res[1];
    int y = res[2];
    if (gcd != 1) cout << "Inverse doesnt exist";
    else cout << "Mod inverse: " << (x + m) % m << endl;
}
```

8.3 Modular exponentiation

```
/**
 * Description: Finding a^b%c
 * Time: O(logN)
 */

// Modular exponentiation
// efficiently calculate the value of x^n mod m. (CPH)
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}

// Code 2:
ll exp(ll x, ll n, ll m) {
        assert(n >= 0);
        x %= m;  // note: m * m must be less than 2^63 to avoid ll overflow
        ll res = 1;
        while (n > 0) {
                if (n % 2 == 1) { res = (res * x) % m; }
                x = (x * x) % m;
                n /= 2;
        }
        return res;
}
```

8.4 Prime

```
bool isPrime(int n) {
    for (int i = 2; i*i <= n; i++)
        if (n % i == 0) return false;
    return n > 1;
}

// Sieve of Eratosthenes, Find all prime <= N
vi sieve(int n) { // O(nlogn)
    vector<bool> mark;
    vi primes;
    mark.resize(n + 1, true);
    mark[0] = mark[1] = false;
    for (int i = 2; i * i <= n; i++) {
        if (mark[i]) {
            // Mark all the multiples of i as composite numbers
            for (int j = i * i; j <= n; j += i) {
                mark[j] = false;
            }
        }
    }

    // return all primes <= n
    for (int i = 2; i <= n; i++) {
        if (mark[i]) primes.push_back(i);
    }
    return primes;
}

// Segmented Sieve: Find all prime in [left, right]
// 1 <= left <= right <= 10^12 and right-left <= 10^6
vi segmentedSieve(int left, int right, vi primes) {
    if (left == 1) left++;
    vector<bool> mark;
    mark.resize(right - left + 1, true);

    for (int i = 0; i < sz(primes) && primes[i] <= sqrt(right); i++) {
        int base = (left / primes[i]) * primes[i];
        if (base < left) base += primes[i];
        for (int j = base; j <= right; j += primes[i]) {
            if (j != primes[i]) mark[j - left] = false;
        }
    }

    vi res;
    for (int i = left; i <= right; i++) {
        if (mark[i - left]) res.push_back(i);
    }
    return res;
}

int main() {
    int left = 11, right = 34;
    vi primes = sieve(sqrt(right));
    vi res = segmentedSieve(left, right, primes);
}


// Prime Factorization. Factorization of 252 = {2,2,3,3,7}
vector<int> factor(int n) { // O(sqrt(N))
        vector<int> ret;
        for (int i = 2; i * i <= n; i++) {
                while (n % i == 0) {
```

```
                ret.push_back(i);
                n /= i;
            }
        }
        if (n > 1) { ret.push_back(n); }
        return ret;
}

// Prime factorization using sieve: O(logN)
// x can have O(logx) distinct prime factors
int minPrime[n + 1];
for (int i = 2; i * i <= n; ++i) {
    if (minPrime[i] == 0) { //if i is prime
        for (int j = i * i; j <= n; j += i) {
            if (minPrime[j] == 0) {
                minPrime[j] = i;
            }
        }
    }
}
for (int i = 2; i <= n; ++i) {
    if (minPrime[i] == 0) {
        minPrime[i] = i;
    }
}

vector<int> factorize(int n) {
    vector<int> res;
    while (n != 1) {
        res.push_back(minPrime[n]);
        n /= minPrime[n];
    }
    return res;
}
```

8.5 GCD, LCM

```
/**
 * Description: GCD and LCM
 * Time: O(logab) for gcd
 */

// For C++14, you can use the built-in __gcd(a,b).
// In C++17, there exists std::gcd and std::lcm in the <numeric> header
// Note: lcm(a, b) = ab/gcd(a, b)

// Also, these two functions are associative, meaning that
// if we want to take the GCD or LCM of more than two elements, we can do so two at
// a time, in any order. For example,

// gcd(a1, a2, a3, a4) = gcd(a1, gcd(a2, gcd(a3, a4)))
```

8.6 Euler's Totient Function

```
int phi(int n) {
    int res = n;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            while (n % i == 0) {
                n /= i;
            }
            res = res / i * (i - 1);
        }
    }
    if (n > 1) {
        res = res / n * (n - 1);
    }
    return res;
}
```

9 Combinatorics

9.1 Binomial Coefficients

```
/**
 * Description: Binomial coefficient
 */

const int MAXN = 1e6;
const int MOD = 1e9 + 7;

ll fac[MAXN + 1];
ll inv[MAXN + 1];

/** Computes x^n modulo m in O(log mod) time. */
ll exp(ll x, ll n, ll m) {
        x %= m;
        ll res = 1;
        while (n > 0) {
                if (n % 2 == 1) { res = res * x % m; }
                x = x * x % m;
                n /= 2;
        }
        return res;
}

/** Precomputes n! from 0 to MAXN. */
void factorial() {
        fac[0] = 1;
        for (int i = 1; i <= MAXN; i++) { fac[i] = fac[i - 1] * i % MOD; }
}

/**
 * Precomputes all modular inverse factorials
 * from 0 to MAXN in O(n + log mod) time
 */
void inverses() {
        inv[MAXN] = exp(fac[MAXN], MOD - 2, MOD);
        for (int i = MAXN; i >= 1; i--) { inv[i - 1] = inv[i] * i % MOD; }
}

/** Computes nCr mod p */
ll choose(int n, int r) { return fac[n] * inv[r] % MOD * inv[n - r] % MOD; }


factorial();
inverses();
choose(a, b);
```

10 Other

10.1 Coordinate Compression

```
/**
 * Description: coordinate Compression
 */

// Coordinate compression describes the process of mapping each value in a list to
// its index if that list was sorted. (labelling coordinate with compressed values)
// When we have values from a large range, but we only care about their relative order
// (for example, if we have to know if one value is above another), coordinate compression
// is a simple way to help with implementation.

typedef pair<int, int> Point;
bool ycomp(Point p, Point q) { return p.second < q.second; }

// Perform compression on a set of points => Can be used as array indices
sort(P, P + N);
for (int i = 0; i < N; i++) P[i].first = i + 1;
sort(P, P + N, ycomp);
for (int i = 0; i < N; i++) P[i].second = i + 1;

// coordinate compression sometimes will also require remembering values in addition
// to compressing them (as opposed to just replacing the original values)
```

10.2 Backtracking

```
/**
 * Description: Backtracking model
 */

const int maxn = 33;
int x[maxn];
int n;

void printRes(){
```

```
    for (int i = 1; i <= n; i++){
        cout << x[i];
    }
    cout << endl;
}

// find all binary string of length N
void backtrack(int i){
    for (int j = 0; j <= 1; j++){ // all values can be assigned to x[i]
        x[i] = j; // try x[i] = j
        if (i == n) printRes(); // tim thay 1 cau hinh
        else backtrack(i + 1); // recursively try values for x[i + 1]
    }
}

// backtrack(1)

// Liet ke cac tap con k phan tu
void backtrack(int i){
    for (int j = x[i - 1] + 1; j <= n - k + i; j++){
        x[i] = j;
        if (i == k) printRes();
        else backtrack(i + 1);
    }
}

// Liet ke cac chinh hop ko lap chap k
void backtrack(int i){
    for (int j = 1; j <= n ;j++){
        if (c[j]){
            x[i] = j;
            if (i == k) printRes();
            else {
                c[j] = false;
                backtrack(i + 1);
                c[j] = true;
            }
        }
    }
}
```

11 Dynamic Programming

11.1 Coin Exchange

```
// Returns number of ways we can exchange k using set of coins {{{
template<typename T>
T coin_exchange(int k, std::vector<int> coins) {
    std::vector<T> f(k + 1);
    f[0] = 1;
    for (int coin : coins) {
        for (int i = coin; i <= k; ++i) {
            f[i] += f[i-coin];
        }
    }
    return f.back();
}
// }}}
```

11.2 Edit Distance

```
// Minimum number of operations to transform string a => string b
// In one operation, we can either modify 1 character, delete 1 character or
// insert 1 character (insert is not needed in this case)
//
// Edit distance {{{
int edit_distance(std::string a, std::string b) {
    int la = a.size();
    int lb = b.size();
    a = " " + a + " ";
    b = " " + b + " ";

    std::vector<std::vector<int>> f(la + 1, std::vector<int> (lb + 1, la + lb));

    // corner cases
    for (int j = 0; j <= lb; ++j) f[0][j] = j;
    for (int i = 0; i <= la; ++i) f[i][0] = i;

    // DP
    for (int i = 1; i <= la; ++i) {
        for (int j = 1; j <= lb; ++j) {
            if (a[i] == b[j]) f[i][j] = f[i-1][j-1];
            else f[i][j] = 1 + std::min({
                    f[i-1][j-1],  // modify
                    f[i][j-1],    // remove b[j]
                    f[i-1][j],    // remove a[i]
                    });
        }
    }

    return f.back().back();
}
// }}}
```

11.3 Knapsack

```
// Knapsack 01 {{{
// Select subset of items, such that sum(weights) <= capacity
// and sum(values) is maximum
int knapsack01(
        int capacity,
        const std::vector<int>& weights,
        const std::vector<int>& values) {
    int n = weights.size();
    std::vector<int> f(capacity + 1, 0);
    for (int i = 0; i < n; ++i) {
        for (int j = capacity; j >= weights[i]; --j) {
            f[j] = max(f[j], f[j-weights[i]] + values[i]);
        }
    }

    return *max_element(f.begin(), f.end());
}
// }}}
// Knapsack unbounded {{{
// Select subset of items, such that sum(weights) <= capacity
// and sum(values) is maximum
// An item can be selected unlimited number of times
int knapsack_unbounded(
        int capacity,
        const std::vector<int>& weights,
        const std::vector<int>& values) {
    int n = weights.size();
    std::vector<int> f(capacity + 1, 0);
    for (int i = 0; i < n; ++i) {
        for (int j = weights[i]; j <= capacity; ++j) {
            f[j] = max(f[j], f[j-weights[i]] + values[i]);
        }
    }

    return *max_element(f.begin(), f.end());
}
// }}}
```

11.4 LIS

```
int find_lis(vi &a) {
    vi dp;
    for (int i : a) {
        int pos = lower_bound(dp.begin(), dp.end(), i) - dp.begin();
        if (pos == sz(dp)) dp.push_back(i);
        else {
            dp[pos] = i;
        }
    }
    return dp.size();
}
```

12 Divide and Conquer

12.1 Binary Search

```
// find target in an monotonic array
int binary_search(int A[], int sizeA, int target) {
    int lo = 1, hi = sizeA;
    while (lo <= hi) {
```

```
        int mid = lo + (hi - lo)/2;
        if (A[mid] == target)
            return mid;
        else if (A[mid] < target)
            lo = mid+1;
        else
            hi = mid-1;
    }
    // target not found in A
    return -1;
}

// Find smallest x such that P(x) = true
// If P(S) is of the form true-false, just reverse P(x)
bool P(int x) {
    // Ham danh gia x
    return true;
}

int binary_search(int lo, int hi) {
    while (lo < hi) {
        int mid = lo + (hi-lo)/2;
        if (P(mid) == true)
            hi = mid;
        else
            lo = mid+1;
    }

    if (P(lo) == false)
        return -1; // P(x) = false for all x in S, no solution

   return lo; // lo is smallest x that P(x) = true
}

// Find biggest x such that P(x) = false
int binary_search(int lo, int hi) {
    while (lo < hi) {
        int mid = lo + (hi-lo+1)/2;
        if (P(mid) == true)
            hi = mid - 1;
        else
            lo = mid;
    }

    if (P(lo) == true)
        return -1; // P(x) = true for all x in S, no solution

   return lo; // lo is largest x that P(x) = false
}

// Binary search on real range
bool isTerminated(double lo, double hi) {
    // return result of check
    // Do lo and hi satisfy stop condition?
}

double binary_search(double lo, double hi) {
    while (isTerminated(lo, hi) == false) {
        double mid = lo + (hi-lo)/2;
        if (P(mid) == true)
            hi = mid;
        else
            lo = mid;
    }
    // Approximate mean of lo and hi
    // boundary between false and true
    return lo + (hi-lo)/2;
}

// Binary search implementation with open range
// hi must be greater than actual search range by 1
// In the case we want to find the last false then the open range will be (lo, hi]
// and search returns lo if every values in range are true
int binary_search(int lo, int hi) {
    while (lo < hi) {
        int mid = lo + (hi-lo)/2;
        if (P(mid) == true)
            hi = mid;
        else
            lo = mid+1;
    }

   return lo; // lo is smallest x that P(x) = true
}
```