ACM-ICPC Team Reference Document
3-STOOD

Contents

1 General

1.1 C++ Template

```
#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

#define int long long
#define mod 1000000007

#define inf INT_MAX
#define llinf LONG_LONG_MAX

#define en '\n'
#define ll long long
#define ld long double
#define ff first
#define ss second
#define rep(i, a, b) for (int i = a; i < b; i++)
#define rrep(i, a, b) for (int i = a; i >= b; i--)
```

```
#define pii pair<int, int>
#define vi vector<int>
#define vc vector<char>
#define vvi vector<vector<int>>
#define vvc vector<vector<char>>
#define vpi vector<pair<int, int>>
#define all(x) (x).begin(), (x).end()
#define rall(x) (x).rbegin(), (x).rend()

template <typename T>
using min_heap = priority_queue<T, vector<T>,
    greater<T>>;

template <typename T>
using max_heap = priority_queue<T, vector<T>,
    less<T>>;

template <typename T>
using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag,
    tree_order_statistics_node_update>;
// order_of_key(k) -> index of k
// *find_by_order(i) -> value at index i

void solve() {}

int32_t main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    cout << fixed;
    cout << setprecision(8);

    int t = 1;
    cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}
```

1.2 Python Template

```
import sys
import re
from math import ceil, log, sqrt, floor

___local_run___ = False
if ___local_run___:
    sys.stdin = open('input.txt', 'r')
    sys.stdout = open('output.txt', 'w')

def main():
    a = int(input())
    b = int(input())
    print(a*b)

main()
```

1.3 Compilation

```
# Simple compile
g++ -DLOCAL -O2 -o main.exe -std-c++17 -Wall
    -Wno-unused-result -Wshadow main.cpp
# Debug
g++ -DLOCAL -std=c++17 -Wshadow -Wall -o
    main.exe main.cpp -fsanitize=address -
    fsanitize=undefined -fuse-ld=gold -
    D_GLIBCXX_DEBUG -g
```

2 Data Structures

2.1 Disjoin Set Union

```
struct DSU {
    vector<int> parent, rank, size;

    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        size.resize(n, 1);
        rep(i, 0, n) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] == x) return x;
        return parent[x] = find(parent[x]);
    }

    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
                size[rootX] += size[rootY];
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
                size[rootY] += size[rootX];
            } else {
                parent[rootY] = rootX;
                size[rootX] += size[rootY];
                rank[rootX]++;
            }
        }
    }
};
```

2.2 Fenwick Tree PURQ

```
struct FenwickTreePURQ {
    int n;
```

```
    vector<int> bit;

    FenwickTreePURQ(int n) {
        this->n = n;
        bit.assign(n + 1, 0);
    }

    void update(int idx, int val) {
        for (; idx <= n; idx += idx & -idx) {
            bit[idx] += val;
        }
    }

    int prefixSum(int idx) {
        int sum = 0;
        for (; idx > 0; idx -= idx & -idx) {
            sum += bit[idx];
        }
        return sum;
    }

    int rangeSum(int l, int r) { return prefixSum(r)
         - prefixSum(l - 1); }
};
```

2.3 Fenwick Tree RUPQ

```
struct FenwickTreeRUPQ {
    vector<int> bit, arr;
    int n;

    FenwickTreeRUPQ(int n, const vector<int>&
        initialArray) {
        this->n = n;
        arr = initialArray;
        bit.assign(n + 1, 0);

        for (int i = 1; i <= n; i++) {
            pointUpdate(i, arr[i - 1]);
        }
    }

    void rangeUpdate(int l, int r, int val) {
        pointUpdate(l, val);
        pointUpdate(r + 1, -val);
    }

    void pointUpdate(int idx, int val) {
        for (; idx <= n; idx += idx & -idx) {
            bit[idx] += val;
        }
    }

    int pointQuery(int idx) {
        int sum = 0;
        for (; idx > 0; idx -= idx & -idx) {
            sum += bit[idx];
        }
        return sum;
    }

    int getValue(int idx) { return arr[idx - 1] +
        pointQuery(idx); }

    void setValue(int idx, int newValue) {
        int diff = newValue - arr[idx - 1];
        arr[idx - 1] = newValue;
        pointUpdate(idx, diff);
    }
};
```

2.4 Fenwick Tree RURQ

```
struct FenwickTreeRURQ {
    int n;
    FenwickTreePURQ bit1, bit2;

    FenwickTreeRURQ(int n) : n(n), bit1(n), bit2(
        n) {}

    void rangeUpdate(int l, int r, int val) {
        bit1.update(l, val);
        bit1.update(r + 1, -val);
        bit2.update(l, val * (l - 1));
        bit2.update(r + 1, -val * r);
    }

    int rangeQuery(int l, int r) {
        return bit1.rangeSum(l, r) * r - bit2.
            rangeSum(l, r) -
              (bit1.rangeSum(1, l - 1) * (l - 1) -
                  bit2.rangeSum(1, l - 1));
    }
};
```

2.5 Fenwick Tree 2D

```
struct FenwickTree2D {
    vector<vector<int>> bit;
    int n, m;

    FenwickTree2D(int n, int m) {
        this->n = n;
        this->m = m;
        bit.assign(n + 1, vector<int>(m + 1, 0));
    }

    void update(int x, int y, int val) {
        for (int i = x; i <= n; i += i & -i) {
            for (int j = y; j <= m; j += j & -j) {
                bit[i][j] += val;
            }
        }
    }

    int prefixQuery(int x, int y) {
```

```
        int sum = 0;
        for (int i = x; i > 0; i -= i & -i) {
            for (int j = y; j > 0; j -= j & -j) {
                sum += bit[i][j];
            }
        }
        return sum;
    }

    int rangeQuery(int x1, int y1, int x2, int y2) {
        return prefixQuery(x2, y2) - prefixQuery(
            x1 - 1, y2) -
              prefixQuery(x2, y1 - 1) +
                  prefixQuery(x1 - 1, y1 - 1);
    }
};
```

2.6 Segment Tree PURQ

```
struct SegmentTreePURQ {
    int n;
    vector<int> tree;

#define left 2 * node + 1
#define right 2 * node + 2
#define mid (start + end) / 2

    void build(int node, int start, int end, vector<
        int> &arr) {
        if (start == end) {
            tree[node] = arr[start];
            return;
        }

        build(left, start, mid, arr);
        build(right, mid + 1, end, arr);
        tree[node] = tree[left] + tree[right];
    }

    int query(int node, int start, int end, int l, int r
        ) {
        if (end < l || start > r) return 0;

        if (start >= l && end <= r) return tree[
            node];

        return (query(left, start, mid, l, r) +
                query(right, mid + 1, end, l, r));
    }

    void update(int node, int start, int end, int ind,
         int val) {
        if (ind < start || ind > end) return;

        if (start == end) {
            tree[node] = val;
            return;
        }
        update(left, start, mid, ind, val);
        update(right, mid + 1, end, ind, val);
        tree[node] = tree[left] + tree[right];
    }

    SegmentTreePURQ(vector<int> &arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(0, 0, n - 1, arr);
    }

    int query(int l, int r) { return query(0, 0, n - 1,
         l, r); }

    void update(int ind, int val) { update(0, 0, n -
        1, ind, val); }

#undef left
#undef right
#undef mid
};
```

2.7 Segment Tree RURQ

```
struct SegmentTreeRURQ {
    int n;
    vector<int> tree, lazy;

#define left 2 * node + 1
#define right 2 * node + 2
#define mid (start + end) / 2

    void build(int node, int start, int end, vector<
        int> &arr) {
        if (start == end) {
            tree[node] = arr[start];
            return;
        }

        build(left, start, mid, arr);
        build(right, mid + 1, end, arr);
        tree[node] = tree[left] + tree[right];
    }

    void pushdown(int node, int start, int end) {
        if (lazy[node]) {
            tree[node] += lazy[node] * (end - start
                + 1);
            if (start != end) {
                lazy[left] += lazy[node];
                lazy[right] += lazy[node];
            }
            lazy[node] = 0;
        }
    }
```

```
    void update(int node, int start, int end, int l,
         int r, int inc) {
        pushdown(node, start, end);

        if (end < l || start > r) return;

        if (start >= l && end <= r) {
            lazy[node] += inc;
            pushdown(node, start, end);
            return;
        }

        update(left, start, mid, l, r, inc);
        update(right, mid + 1, end, l, r, inc);
        tree[node] = tree[left] + tree[right];
    }

    int query(int node, int start, int end, int ind) {
        pushdown(node, start, end);

        if (ind < start || ind > end) return 0;

        if (start == end) return tree[node];

        return (query(left, start, mid, ind) + query
             (right, mid + 1, end, ind));
    }

    SegmentTreeRURQ(vector<int> &arr) {
        n = arr.size();
        tree.resize(4 * n);
        lazy.assign(4 * n, 0);

        build(0, 0, n - 1, arr);
    }

    void update(int l, int r, int inc) { update(0, 0,
         n - 1, l, r, inc); }

    int query(int ind) { return query(0, 0, n - 1,
         ind); }

#undef left
#undef right
#undef mid
};
```

2.8 Sparse Table

```
struct SparseTable {
    vector<vector<int>> st;
    vector<int> logValues;
    int n;

    SparseTable(const vector<int>& arr) {
        n = arr.size();
        int maxLog = log2(n) + 1;
        st = vector<vector<int>>(n, vector<int
             >(maxLog));
        logValues = vector<int>(n + 1);

        logValues[1] = 0;
        for (int i = 2; i <= n; i++) logValues[i] =
             logValues[i / 2] + 1;

        for (int i = 0; i < n; i++) st[i][0] = arr[i];

        for (int j = 1; j <= maxLog; j++) {
            for (int i = 0; i + (1 << j) <= n; i
                 ++) {
                st[i][j] = min(st[i][j - 1], st[i + (1
                     << (j - 1))][j - 1]);
            }
        }
    }

    int rangeMinQuery(int l, int r) {
        int j = logValues[r - l + 1];
        return min(st[l][j], st[r - (1 << j) + 1][j]);
    }

};
```

2.9 Trie

```
struct TrieNode {
    int ending, holding;
    vector<TrieNode*> next;

    TrieNode() : ending(0), holding(0), next(26,
         nullptr) {}
};

struct Trie {
    TrieNode* head;

    Trie() { head = new TrieNode(); }

    void insert(string& s) {
        TrieNode* temp = head;
        for (auto& c : s) {
            temp->holding++;
            if (temp->next[c - 'a'] == nullptr) {
                temp->next[c - 'a'] = new
                     TrieNode();
            }
            temp = temp->next[c - 'a'];
        }
        temp->holding++;
        temp->ending++;
    }

    int findPrefixes(string& s) {
        TrieNode* temp = head;
        for (auto& c : s) {
```

```
            if (temp->next[c - 'a'] == nullptr)
                return 0;
            temp = temp->next[c - 'a'];
        }
        return temp->holding;
    }
};
```

3 Graphs

3.1 Binary Lifting

```
struct info {
    // all the data required
    // for merging and answering queries
    int sum, minPrefL, maxPrefL, minPrefR,
        maxPrefR, minSeg, maxSeg;

    info(int el = 0) {
        sum = el;
        minSeg = minPrefL = minPrefR = min(el,
            0LL);
        maxSeg = maxPrefL = maxPrefR = max(
            el, 0LL);
    }
};

info merge(info &a, info &b) {
    // a's ending node is same as b's starting node
    // however info doesn't include ending node's
        value
    info res;
    res.sum = a.sum + b.sum;
    res.minPrefL = min(a.minPrefL, a.sum + b.
        minPrefL);
    res.maxPrefL = max(a.maxPrefL, a.sum + b.
        maxPrefL);
    res.minPrefR = min(a.minPrefR + b.sum, b.
        minPrefR);
    res.maxPrefR = max(a.maxPrefR + b.sum, b.
        maxPrefR);
    res.minSeg = min({a.minSeg, b.minSeg, a.
        minPrefR + b.minPrefL});
    res.maxSeg = max({a.maxSeg, b.maxSeg, a.
        maxPrefR + b.maxPrefL});
    return res;
}

const int MAXN = 2e5 + 5;
const int MAXLOG = 20;

// logCache[2^i] = i
map<int, int> logCache;
// depth of each node
vector<int> lvl(MAXN);
// par[i][0] -> immediate parent of i
vector<vector<int>> par(MAXN, vector<int>(
    MAXLOG));
// lift[i][j] -> value from i (inclusive) to 2^j th
    parent (exclusive)
// lift[i][0] -> i node's value only
vector<vector<info>> lift(MAXN, vector<info>(
    MAXLOG));

int lca(int a, int b) {
    if (lvl[a] < lvl[b]) swap(a, b);

    int diff = lvl[a] - lvl[b];
    for (int i = MAXLOG - 1; i >= 0; i--) {
        if ((diff >> i) & 1) {
            a = par[a][i];
        }
    }

    if (a == b) return b;

    for (int i = MAXLOG - 1; i >= 0; i--) {
        if (par[a][i] != par[b][i]) {
            a = par[a][i];
            b = par[b][i];
        }
    }
    return par[a][0];
}

void reverseData(info &u) {
    swap(u.minPrefL, u.minPrefR);
    swap(u.maxPrefL, u.maxPrefR);
}

info segmentData(int a, int b) {
    if (lvl[a] < lvl[b]) swap(a, b);

    info u, v;

    int diff = lvl[a] - lvl[b];
    for (int i = MAXLOG - 1; i >= 0; i--) {
        if ((diff >> i) & 1) {
            u = merge(u, lift[a][i]);
            a = par[a][i];
        }
    }

    if (a == b) {
        u = merge(u, lift[a][0]);
        return u;
    }

    for (int i = MAXLOG - 1; i >= 0; i--) {
        if (par[a][i] != par[b][i]) {
            u = merge(u, lift[a][i]);
            v = merge(v, lift[b][i]);
            a = par[a][i];
            b = par[b][i];
        }
    }
    u = merge(u, lift[a][1]);
```

```
    v = merge(v, lift[b][0]);
    reverseData(v);
    return merge(u, v);
}
```

3.2 Lowest Common Ancestor

```
int timer;
vector<vector<int>> adj;
vector<int> tin, tout;

void dfs(int v, int p) {
    tin[v] = ++timer;

    for (int u : adj[v]) {
        if (u != p) dfs(u, v);
    }

    tout[v] = ++timer;
}

bool isAncestor(int u, int v) {
    return tin[u] <= tin[v] && tout[v] <= tout[u];
}
```

3.3 Strongly Connected Components

```
vector<vector<int> > g, gr; // adjList and
    reversed adjList
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;
    for (size_t i = 0; i < g[v].size(); ++i)
        if (!used[g[v][i]]) dfs1(g[v][i]);
    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);
    for (size_t i = 0; i < gr[v].size(); ++i)
        if (!used[gr[v][i]]) dfs2(gr[v][i]);
}

int main() {
    int n;
    for (;;) {
        int a, b;
        g[a].push_back(b);
        gr[b].push_back(a);
    }

    used.assign(n, false);
    for (int i = 0; i < n; ++i)
        if (!used[i]) dfs1(i);
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        int v = order[n - 1 - i];
        if (!used[v]) {
            dfs2(v);
            // do something with the found
                component
            component.clear(); // components are
                generated in toposort-order
        }
    }
}
```

3.4 Bellman Ford Algorithm

```
struct Edge {
    int a, b, cost;
};

int n, m, v;
vector<Edge> e;

// To find a negative cycle: perform one more
    relaxation step.
// If anything changes - a negative cycle exists.
void solve() {
    vector<int> d(n, INT_MAX);
    d[v] = 0;
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < m; ++j)
            if (d[e[j].a] < INT_MAX)
                d[e[j].b] = min(d[e[j].b], d[e[j].a] +
                    e[j].cost);
}
```

3.5 Finding Articulation Points

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> inTime, lowTime;
int timer;

void processCutpoint(int v) {
    // problem-specific logic goes here
    // it can be called multiple times for the same
        v
}

void dfs(int v, int p = -1) {
    visited[v] = true;
    inTime[v] = lowTime[v] = timer++;
    int children = 0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
```

```
            lowTime[v] = min(lowTime[v], inTime[
                to]);
        } else {
            dfs(to, v);
            lowTime[v] = min(lowTime[v],
                lowTime[to]);
            if (lowTime[to] >= inTime[v] && p !=
                -1) processCutpoint(v);
            ++children;
        }
    }
    if (p == -1 && children > 1) processCutpoint(
        v);
}

void findCutpoints() {
    timer = 0;
    visited.assign(n, false);
    inTime.assign(n, -1);
    lowTime.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs(i);
    }
}
```

3.6 Finding Bridges

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> inTime, lowTime;
int timer;

void processBridge(int u, int v) {
    // do something with the found bridge
}

void dfs(int v, int p = -1) {
    visited[v] = true;
    inTime[v] = lowTime[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;;
        if (visited[to]) {
            lowTime[v] = min(lowTime[v], inTime[
                to]);
        } else {
            dfs(to, v);
            lowTime[v] = min(lowTime[v],
                lowTime[to]);
            if (lowTime[to] > inTime[v])
                processBridge(v, to);
        }
    }
}

// Doesn't work with multiple edges
// But multiple edges are never bridges, so it's
    easy to check
void findBridges() {
    timer = 0;
    visited.assign(n, false);
    inTime.assign(n, -1);
    lowTime.assign(n, -1);
    bridges.clear();
    FOR(i, 0, n) {
        if (!visited[i]) dfs(i);
    }
}
```

3.7 Number Of Paths Of Fixed Length

Let G be the adjacency matrix of a graph. Then $C_k = G^k$ gives a matrix, in which the value $C_k[i][j]$ gives the number of paths between i and j of length k.

3.8 Shortest Paths Of Fixed Length

Define $A \bigodot B = C \iff C_{ij} = min_{p=1..n}(A_{ip} + B_{pj})$. Let G be the adjacency matrix of a graph. Also, let $L_k = G \bigodot \ldots \bigodot G = G^{\bigodot k}$. Then the value $L_k[i][j]$ denotes the length of the shortest path between i and j which consists of exactly k edges.

3.9 Dijkstra

```
vector<vector<pair<int, int>>> adj;
void dijkstra(int s, vector<int>& d, vector<int>&
    p) {
    int n = adj.size();
    d.assign(n, INT_MAX);
    p.assign(n, -1);

    d[s] = 0;
    min_heap<pii> q;
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v]) continue;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;
            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

3.10 Floyd Warshall

```
vector<vector<int>> floydWarshall(vector<vector<int>>& graph, int V) {
    vector<vector<int>> dist = graph;

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    return dist;
}
```

4 Geometry

4.1 2d Vector

```
template <typename T>
struct Vec {
    T x, y;
    Vec(): x(0), y(0) {}
    Vec(T _x, T _y): x(_x), y(_y) {}
    Vec operator+(const Vec& b) {
        return Vec<T>(x+b.x, y+b.y);
    }
    Vec operator-(const Vec& b) {
        return Vec<T>(x-b.x, y-b.y);
    }
    Vec operator*(T c) {
        return Vec(x*c, y*c);
    }
    T operator*(const Vec& b) {
        return x*b.x + y*b.y;
    }
    T operator^(const Vec& b) {
        return x*b.y-y*b.x;
    }
    bool operator<(const Vec& other) const {
        if(x == other.x) return y < other.y;
        return x < other.x;
    }
    bool operator==(const Vec& other) const {
        return x==other.x && y==other.y;
    }
    bool operator!=(const Vec& other) const {
        return !(*this == other);
    }
    friend ostream& operator<<(ostream& out, const Vec& v) {
        return out << "(" << v.x << ", " << v.y << ")";
    }
    friend istream& operator>>(istream& in, Vec<T>& v) {
        return in >> v.x >> v.y;
    }
    T norm() { // squared length
        return (*this)*(*this);
    }
    ld len() {
        return sqrt(norm());
    }
    ld angle(const Vec& other) { // angle between this and other vector
        return acosl((*this)*other/len()/other.len());
    }
    Vec perp() {
        return Vec(-y, x);
    }
};
/* Cross product of 3d vectors: (ay*bz-az*by, az*bx-ax*bz, ax*by-ay*bx)
 */
```

4.2 Line

```
template <typename T>
struct Line { // expressed as two vectors
    Vec<T> start, dir;
    Line() {}
    Line(Vec<T> a, Vec<T> b): start(a), dir(b-a) {}

    Vec<ld> intersect(Line l) {
        ld t = ld((l.start-start)^l.dir)/(dir^l.dir);
        // For segment-segment intersection this should be in range [0, 1]
        Vec<ld> res(start.x, start.y);
        Vec<ld> dirld(dir.x, dir.y);
        return res + dirld*t;
    }
};
```

4.3 Convex Hull Gift Wrapping

```
vector<Vec<int>> buildConvexHull(vector<Vec<int>>& pts) {
    int n = pts.size();
    sort(pts.begin(), pts.end());
    auto currP = pts[0]; // choose some extreme point to be on the hull

    vector<Vec<int>> hull;
    set<Vec<int>> used;
    hull.pb(pts[0]);
    used.insert(pts[0]);
```

```
    while(true) {
        auto candidate = pts[0]; // choose some
            point to be a candidate

        auto currDir = candidate-currP;
        vector<Vec<int>> toUpdate;
        FOR(i, 0, n) {
            if(currP == pts[i]) continue;
            // currently we have currP->candidate
            // we need to find point to the left of
                this
            auto possibleNext = pts[i];
            auto nextDir = possibleNext - currP;
            auto cross = currDir ^ nextDir;
            if(candidate == currP || cross > 0) {
                candidate = possibleNext;
                currDir = nextDir;
            } else if(cross == 0 && nextDir.norm
                () > currDir.norm()) {
                candidate = possibleNext;
                currDir = nextDir;
            }
        }
        if(used.find(candidate) != used.end())
            break;
        hull.pb(candidate);
        used.insert(candidate);
        currP = candidate;
    }
    return hull;
}
```

4.4 Convex Hull With Graham's Scan

```
// Takes in >= 3 points
// Returns convex hull in clockwise order
// Ignores points on the border
vector<Vec<int>> buildConvexHull(vector<Vec<
    int>> pts) {
    if(pts.size() <= 3) return pts;
    sort(pts.begin(), pts.end());
    stack<Vec<int>> hull;
    hull.push(pts[0]);
    auto p = pts[0];
    sort(pts.begin()+1, pts.end(), [&](Vec<int> a,
        Vec<int> b) -> bool {
        // p->a->b is a ccw turn
        int turn = sgn((a-p)^(b-a));
        //if(turn == 0) return (a-p).norm() > (b-
            p).norm();
        // ^ among collinear points, take the
            farthest one
        return turn == 1;
    });
    hull.push(pts[1]);
    FOR(i, 2, (int)pts.size()) {
        auto c = pts[i];
        if(c == hull.top()) continue;
        while(true) {
            auto a = hull.top(); hull.pop();
            auto b = hull.top();
            auto ba = a-b;
            auto ac = c-a;
            if((ba^ac) > 0) {
                hull.push(a);
                break;
            } else if((ba^ac) == 0) {
                if(ba*ac < 0) c = a;
                // ^ c is between b and a, so it
                    shouldn't be added to the hull
                break;
            }
        }
        hull.push(c);
    }
    vector<Vec<int>> hullPts;
    while(!hull.empty()) {
        hullPts.pb(hull.top());
        hull.pop();
    }
    return hullPts;
}
```

4.5 Circle Line Intersection

```
double r, a, b, c; // ax+by+c=0, radius is at (0, 0)
// If the center is not at (0, 0), fix the constant c
    to translate everything so that center is at (0,
    0)
double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b
    );
if (c*c > r*r*(a*a+b*b)+eps)
    puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < eps) {
    puts ("1 point");
    cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b));
    double ax, ay, bx, by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx <<
        ' ' << by << '\n';
}
```

4.6 Circle Circle Intersection

Let's say that the first circle is centered at $(0, 0)$ (if it's not, we can move the origin to the center of the first circle and adjust the coordinates), and the second one is at (x_2, y_2). Then, let's construct a line

$Ax + By + C = 0$, where $A = -2x_2, B = -2y_2, C = x_2^2 + y_2^2 + r_1^2 - r_2^2$. Finding the intersection between this line and the first circle will give us the answer. The only tricky case: if both circles are centered at the same point. We handle this case separately.

4.7 Common Tangents To Two Circles

```
struct pt {
    double x, y;

    pt operator- (pt p) {
        pt res = { x-p.x, y-p.y };
        return res;
    }
};
struct circle : pt {
    double r;
};
struct line {
    double a, b, c;
};
void tangents (pt c, double r1, double r2, vector<
    line> & ans) {
    double r = r2 - r1;
    double z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -eps) return;
    d = sqrt (abs (d));
    line l;
    l.a = (c.x * r + c.y * d) / z;
    l.b = (c.y * r - c.x * d) / z;
    l.c = r1;
    ans.push_back (l);
}
vector<line> tangents (circle a, circle b) {
    vector<line> ans;
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents (b-a, a.r*i, b.r*j, ans);
    for (size_t i=0; i<ans.size(); ++i)
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    return ans;
}
```

4.8 Number Of Lattice Points On Segment

Let's say we have a line segment from (x_1, y_1) to (x_2, y_2). Then, the number of lattice points on this segment is given by

$$gcd(x_2 - x_1, y_2 - y_1) + 1.$$

4.9 Pick's Theorem

We are given a lattice polygon with non-zero area. Let's denote its area by S, the number of points with integer coordinates lying strictly inside the polygon by I and the number of points lying on the sides of the polygon by B. Then:

$$S = I + \frac{B}{2} - 1.$$

4.10 Usage Of Complex

```
typedef long long C; // could be long double
typedef complex<C> P; // represents a point or
    vector
#define X real()
#define Y imag()
...
P p = {4, 2}; // p.X = 4, p.Y = 2
P u = {3, 1};
P v = {2, 2};
P s = v+u; // {5, 3}
P a = {4, 2};
P b = {3, -1};
auto l = abs(b-a); // 3.16228
auto plr = polar(1.0, 0.5); // construct a vector of
    length 1 and angle 0.5 radians
v = {2, 2};
auto alpha = arg(v); // 0.463648
v *= plr; // rotates v by 0.5 radians
    counterclockwise. The length of plt must be 1
    to rotate correctly.
auto beta = arg(v); // 0.963648
a = {4, 2};
b = {1, 2};
C p = (conj(a)*b).Y; // 6 <- the cross product of
    a and b
```

4.11 Misc

Distance from point to line.

We have a line $l(t) = \vec{a} + \vec{b}t$ and a point $\vec{p}$. The distance from this point to the line can be calculated by expressing the area of a triangle in two different ways. The final formula: $d = \frac{(\vec{p}-\vec{a})\times(\vec{p}-\vec{b})}{|\vec{b}-\vec{a}|}$

Point in polygon.

Send a ray (half-infinite line) from the points to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, otherwise it's outside.

Using cross product to test rotation direction.

Let's say we have vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$. Let's define $\vec{ab} = b - a, \vec{bc} = c - b$ and $s = sgn(\vec{ab} \times \vec{bc})$. If $s = 0$, the three points are collinear. If $s = 1$, then $\vec{bc}$ turns in the counterclockwise direction compared to the direction of $\vec{ab}$. Otherwise it turns in the clockwise direction.

Line segment intersection.

The problem: to check if line segments ab and cd intersect. There are three cases:

1. **The line segments are on the same line.** Use cross products and check if they're zero - this will tell if all points are on the same line. If so, sort the points and check if their intersection is non-empty. If it is non-empty, there are an infinite number of intersection points.
2. **The line segments have a common vertex.** Four possibilities: $a = c, a = d, b = c, b = d$.
3. **There is exactly one intersection point that is not an endpoint.** Use cross product to check if points c and d are on different sides of the line going through a and b and if the points a and b are on different sides of the line going through c and d.

Angle between vectors.

$arccos(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|})$.

Dot product properties.

If the dot product of two vectors is zero, the vectors are orthogonal. If it is positive, the angle is acute. Otherwise it is obtuse.

Lines with line equation.

Any line can be described by an equation $ax + by + c = 0$.

- Construct a line using two points A and B:
  1. Take vector from A to B and rotate it 90 degrees ($(x, y) \to (-y, x)$). This will be (a, b).
  2. Normalize this vector. Then put A (or B) into the equation and solve for c.
- Distance from point to line: put point coordinates into line equation and take absolute value. If (a, b) is not normalized, you still need to divide by $\sqrt{a^2 + b^2}$.
- Distance between two parallel lines: $|c_1 - c_2|$ (if they are not normalized, you still need to divide by $\sqrt{a^2 + b^2}$).
- Project a point onto a line: compute signed distance d between line L and point P. Answer is $P - d(\vec{a, b})$.
- Build a line parallel to a given one and passing through a given point: compute the signed distance d between line and point. Answer is $ax + by + (c - d) = 0$.
- Intersect two lines: $d = a_1b_2 - a_2b_1, x = \frac{c_2b_1 - c_1b_2}{d}, y = \frac{c_1a_2 - c_2a_1}{d}$. If $abs(d) < \epsilon$, then the lines are parallel.

Half-planes.

Definition: define as line, assume a point (x, y) belongs to half plane iff $ax + by + c \geq 0$.

Intersecting with a convex polygon:

1. Start at any point and move along the polygon's traversal.
2. Alternate points and segments between consecutive points.
3. If point belongs to half-plane, add it to the answer.
4. If segment intersects the half-plane's line, add it to the answer.

Some more techniques.

- Check if point A lies on segment BC:
  1. Compute point-line distance and check if it is 0 (abs less than ϵ).
  2. $\vec{BA} \cdot \vec{BC} \geq 0$ and $\vec{CA} \cdot \vec{CB} \geq 0$.
- Compute distance between line segment and point: project point onto line formed by the segment. If this point is on the segment, then the distance between it and original point is the answer. Otherwise, take minimum of distance between point and segment endpoints.

5 Math

5.1 Linear Sieve

```
ll minDiv[MAXN+1];
vector<ll> primes;

void sieve(ll n){
    FOR(k, 2, n+1){
        minDiv[k] = k;
    }
    FOR(k, 2, n+1) {
        if(minDiv[k] == k) {
            primes.pb(k);
        }
        for(auto p : primes) {
            if(p > minDiv[k]) break;
            if(p*k > n) break;
            minDiv[p*k] = p;
        }
    }
}
```

5.2 Extended Euclidean Algorithm

```
// ax+by=gcd(a,b)
void solveEq(ll a, ll b, ll& x, ll& y, ll& g) {
    if(b==0) {
        x = 1;
        y = 0;
        g = a;
        return;
    }
    ll xx, yy;
    solveEq(b, a%b, xx, yy, g);
    x = yy;
    y = xx-yy*(a/b);
}
// ax+by=c
bool solveEq(ll a, ll b, ll c, ll& x, ll& y, ll& g) {
    solveEq(a, b, x, y, g);
    if(c%g != 0) return false;
    x *= c/g; y *= c/g;
    return true;
}
// Finds a solution (x, y) so that x >= 0 and x is
    minimal
bool solveEqNonNegX(ll a, ll b, ll c, ll& x, ll &y, ll
    & g) {
    if(!solveEq(a, b, c, x, y, g)) return false;
    ll k = x*g/b;
    x = x - k*b/g;
    y = y + k*a/g;
    if(x < 0) {
        x += b/g;
        y -= a/g;
    }
    return true;
}
```

All other solutions can be found like this:

$$x' = x - k\frac{b}{g}, y' = y + k\frac{a}{g}, k \in \mathbb{Z}$$

5.3 Chinese Remainder Theorem

Let's say we have some numbers m_i, which are all mutually coprime. Also, let $M = \prod_i m_i$. Then the system of congruences

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \ldots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

is equivalent to $x \equiv A \pmod{M}$ and there exists a unique number A satisfying $0 \le A \le M$.

Solution for two: $x \equiv a_1 \pmod{m_1}, x \equiv a_2 \pmod{m_2}$. Let $x = a_1 + km_1$. Substituting into the second congruence: $km_1 \equiv a_2 - a_1 \pmod{m_2}$. Then, $k = (m_1)^{-1}_{m_2}(a_2 - a_1) \pmod{m_2}$. and we can easily find x. This can be extended to multiple equations by solving them one-by-one.

If the moduli are not coprime, solve the system $y \equiv 0 \pmod{\frac{m_1}{g}}, y \equiv \frac{a_2 - a_1}{g} \pmod{\frac{m_2}{g}}$ for y. Then let $x \equiv gy + a_1 \pmod{\frac{m_1 m_2}{g}}$.

5.4 Euler Totient Function

```
// Number of numbers x < n so that gcd(x, n) = 1
ll phi(ll n) {
    if(n == 1) return 1;
    auto f = factorize(n);
    ll res = n;
    for(auto p : f) {
        res = res - res/p.first;
    }
    return res;
}
```

5.5 Factorization With Sieve

```
// Use linear sieve to calculate minDiv
vector<pll> factorize(ll x) {
    vector<pll> res;
    ll prev = -1;
    ll cnt = 0;
    while(x != 1) {
        ll d = minDiv[x];
        if(d == prev) {
            cnt++;
        } else {
            if(prev != -1) res.pb({prev, cnt});
            prev = d;
            cnt = 1;
        }
        x /= d;
    }
    res.pb({prev, cnt});
    return res;
}
```

5.6 Modular Inverse

```
bool invWithEuclid(ll a, ll m, ll& aInv) {
    ll x, y, g;
    if(!solveEqNonNegX(a, m, 1, x, y, g)) return
        false;
    aInv = x;
    return true;
}
// Works only if m is prime
ll invFermat(ll a, ll m) {
    return pwr(a, m-2, m);
}
// Works only if gcd(a, m) = 1
ll invEuler(ll a, ll m) {
    return pwr(a, phi(m)-1, m);
}
```

5.7 Simpson Integration

```
const int N = 1000 * 1000; // number of steps (
    already multiplied by 2)

double simpsonIntegration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b =
        x_2n
    for (int i = 1; i <= N - 1; ++i) {
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

5.8 Burnside's Lemma

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Burnside's lemma asserts the following formula for the number of orbits:

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

Example. Coloring a cube with three colors.

Let X be the set of 3^6 possible face color combinations. Let's count the sizes of the fixed sets for each of the 24 rotations:

- one 0-degree rotation which leaves all 3^6 elements of X unchanged
- six 90-degree face rotations, each of which leaves 3^3 elements of X unchanged
- three 180-degree face rotation, each of which leaves 3^4 elements of X unchanged
- eight 120-degree vertex rotations, each of which leaves 3^2 elements of X unchanged
- six 180-degree edge rotations, each of which leaves 3^3 elements of X unchanged

The average is then $\frac{1}{24}(3^6 + 6\cdot 3^3 + 3\cdot 3^4 + 8\cdot 3^2 + 6\cdot 3^3) = 57$. For n colors: $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$.

Example. Coloring a circular stripe of n cells with two colors.

X is the set of all colored striped (it has 2^n elements), G is the group of rotations (n elements - by 0 cells, by 1 cell, ..., by $(n-1)$ cells). Let's fix some K and find the number of stripes that are fixed by the rotation by K cells. If a stripe becomes itself after rotation by K cells, then its 1st cell must have the same color as its $(1 + K \mod n)$-th cell, which is in turn the same as its $(1 + 2K \mod n)$-th cell, etc., until $mK \mod n = 0$. This will happen when $m = n/gcd(K, n)$. Therefore, we have $n/gcd(K, n)$ cells that must all be of the same color. The same will happen when starting from the second cell and so on. Therefore, all cells are separated into $gcd(K, n)$ groups, with each group being of one color, and that yields $2^{gcd(K,n)}$ choices. That's why the answer to the original problem is $\frac{1}{n}\sum_{k=0}^{n-1} 2^{gcd(k,n)}$.

5.9 FFT

```
namespace FFT {
    int n;
    vector<int> r;
    vector<complex<ld>> omega;
    int logN, pwrN;

    void initLogN() {
        logN = 0;
        pwrN = 1;
        while (pwrN < n) {
            pwrN *= 2;
            logN++;
        }
        n = pwrN;
    }

    void initOmega() {
        FOR(i, 0, pwrN) {
            omega[i] = { cos(2 * i*PI / n), sin(2 *
                i*PI / n) };
        }
    }

    void initR() {
        r[0] = 0;
        FOR(i, 1, pwrN) {
            r[i] = r[i / 2] / 2 + ((i & 1) << (logN
                - 1));
        }
    }

    void initArrays() {
        r.clear();
        r.resize(pwrN);
        omega.clear();
        omega.resize(pwrN);
    }

    void init(int n) {
        FFT::n = n;
        initLogN();
        initArrays();
        initOmega();
        initR();
    }

    void fft(complex<ld> a[], complex<ld> f[]) {
        FOR(i, 0, pwrN) {
            f[i] = a[r[i]];
        }
        for (ll k = 1; k < pwrN; k *= 2) {
```

```
            for (ll i = 0; i < pwrN; i += 2 * k) {
                for (ll j = 0; j < k; j++) {
                    auto z = omega[j*n / (2 * k)]
                        * f[i + j + k];
                    f[i + j + k] = f[i + j] - z;
                    f[i + j] += z;
                }
            }
        }
    }
}
```

5.10 FFT With Modulo

```
bool isGenerator(ll g) {
    if (pwr(g, M - 1) != 1) return false;
    for (ll i = 2; i*i <= M - 1; i++) {
        if ((M - 1) % i == 0) {
            ll q = i;
            if (isPrime(q)) {
                ll p = (M - 1) / q;
                ll pp = pwr(g, p);
                if (pp == 1) return false;
            }
            q = (M - 1) / i;
            if (isPrime(q)) {
                ll p = (M - 1) / q;
                ll pp = pwr(g, p);
                if (pp == 1) return false;
            }
        }
    }
    return true;
}

namespace FFT {
    ll n;
    vector<ll> r;
    vector<ll> omega;
    ll logN, pwrN;

    void initLogN() {
        logN = 0;
        pwrN = 1;
        while (pwrN < n) {
            pwrN *= 2;
            logN++;
        }
        n = pwrN;
    }

    void initOmega() {
        ll g = 2;
        while (!isGenerator(g)) g++;
        ll G = 1;
        g = pwr(g, (M - 1) / pwrN);
        FOR(i, 0, pwrN) {
            omega[i] = G;
            G *= g;
            G %= M;
        }
    }

    void initR() {
        r[0] = 0;
        FOR(i, 1, pwrN) {
            r[i] = r[i / 2] / 2 + ((i & 1) << (logN
                - 1));
        }
    }

    void initArrays() {
        r.clear();
        r.resize(pwrN);
        omega.clear();
        omega.resize(pwrN);
    }

    void init(ll n) {
        FFT::n = n;
        initLogN();
        initArrays();
        initOmega();
        initR();
    }

    void fft(ll a[], ll f[]) {
        for (ll i = 0; i < pwrN; i++) {
            f[i] = a[r[i]];
        }
        for (ll k = 1; k < pwrN; k *= 2) {
            for (ll i = 0; i < pwrN; i += 2 * k) {
                for (ll j = 0; j < k; j++) {
                    auto z = omega[j*n / (2 * k)]
                        * f[i + j + k] % M;
                    f[i + j + k] = f[i + j] - z;
                    f[i + j] += z;
                    f[i + j + k] %= M;
                    if (f[i + j + k] < 0) f[i + j + k]
                        += M;
                    f[i + j] %= M;
                }
            }
        }
    }
}
```

5.11 Big Integer Multiplication With FFT

```
complex<ld> a[MAX_N], b[MAX_N];
complex<ld> fa[MAX_N], fb[MAX_N], fc[
    MAX_N];
complex<ld> cc[MAX_N];

string mul(string as, string bs) {
```

```
    int sgn1 = 1;
    int sgn2 = 1;
    if (as[0] == '-') {
        sgn1 = -1;
        as = as.substr(1);
    }
    if (bs[0] == '-') {
        sgn2 = -1;
        bs = bs.substr(1);
    }
    int n = as.length() + bs.length() + 1;
    FFT::init(n);
    FOR(i, 0, FFT::pwrN) {
        a[i] = b[i] = fa[i] = fb[i] = fc[i] = cc[i] = 0;
    }
    FOR(i, 0, as.size()) {
        a[i] = as[as.size() - 1 - i] - '0';
    }
    FOR(i, 0, bs.size()) {
        b[i] = bs[bs.size() - 1 - i] - '0';
    }
    FFT::fft(a, fa);
    FFT::fft(b, fb);
    FOR(i, 0, FFT::pwrN) {
        fc[i] = fa[i] * fb[i];
    }
    // turn [0,1,2,...,n-1] into [0, n-1, n-2, ..., 1]
    FOR(i, 1, FFT::pwrN) {
        if (i < FFT::pwrN - i) {
            swap(fc[i], fc[FFT::pwrN - i]);
        }
    }
    FFT::fft(fc, cc);
    ll carry = 0;
    vector<int> v;
    FOR(i, 0, FFT::pwrN) {
        int num = round(cc[i].real() / FFT::pwrN)
            + carry;
        v.pb(num % 10);
        carry = num / 10;
    }
    while (carry > 0) {
        v.pb(carry % 10);
        carry /= 10;
    }
    reverse(v.begin(), v.end());
    bool start = false;
    ostringstream ss;
    bool allZero = true;
    for (auto x : v) {
        if (x != 0) {
            allZero = false;
            break;
        }
    }
    if (sgn1*sgn2 < 0 && !allZero) ss << "-";
    for (auto x : v) {
        if (x == 0 && !start) continue;
        start = true;
        ss << abs(x);
    }
    if (!start) ss << 0;
    return ss.str();
}
```

5.12 Gaussian Elimination

```
// The last column of a is the right-hand side of
    the system.
// Returns 0, 1 or oo - the number of solutions.
// If at least one solution is found, it will be in ans
int gauss (vector < vector<ld> > a, vector<ld> &
    ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++
        col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < eps)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                ld c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        ld sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > eps)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return oo;
    return 1;
}
```

5.13 Sprague Grundy Theorem

We have a game which fulfills the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

Grundy Numbers. The idea is to calculate Grundy numbers for each game state. It is calculated like so: $mex(\{g_1, g_2, ..., g_n\})$, where $g_1, g_2, ..., g_n$ are the Grundy numbers of the states which are reachable from the current state. mex gives the smallest nonnegative number that is not in the set ($mex(\{0, 1, 3\}) = 2, mex(\emptyset) = 0$). If the Grundy number of a state is 0, then this state is a losing state. Otherwise it's a winning state.

Grundy's Game. Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of a game state is $mex(\{g_1, g_2, ..., g_n\}), g_k = a_{k,1} \oplus a_{k,2} \oplus ... \oplus a_{k,m}$ meaning that move k divides the game into m subgames whose Grundy numbers are $a_{i,j}$.

Example. We have a heap with n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game. Let $g(n)$ denote the Grundy number of a heap of size n. The Grundy number can be calculated by going though all possible ways to divide the heap into two parts. E.g. $g(8) = mex(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\})$. Base case: $g(1) = g(2) = 0$, because these are losing states.

5.14 Formulas

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(2n+1)(n+1)}{6}$; $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$; $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$; $\sum_{i=a}^{b} c^i = \frac{c^{b+1}-c^a}{c-1}, c \neq 1$; $\sum_{i=1}^{n} a_1 + (i-1)d = \frac{n(a_1+a_n)}{2}$; $\sum_{i=1}^{n} a_1 r^{i-1} = \frac{a_1(1-r^n)}{1-r}, r \neq 1$; $\sum_{i=1}^{\infty} ar^{i-1} = \frac{a_1}{1-r}, |r| \leq 1$.

6 Strings

6.1 Hashing

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

struct HashedString {
    const ll A1 = 999999929, B1 = 1000000009, A2
        = 1000000087, B2 = 1000000097;
    vector<ll> A1pwrs, A2pwrs;
    vector<pll> prefixHash;
    HashedString(const string& _s) {
        init(_s);
        calcHashes(_s);
    }
    void init(const string& s) {
        ll a1 = 1;
        ll a2 = 1;
        FOR(i, 0, (int)s.length() + 1) {
            A1pwrs.pb(a1);
            A2pwrs.pb(a2);
            a1 = (a1 * A1) % B1;
            a2 = (a2 * A2) % B2;
        }
    }
    void calcHashes(const string& s) {
        pll h = {0, 0};
        prefixHash.pb(h);
        for (char c : s) {
            ll h1 = (prefixHash.back().first * A1 +
                c) % B1;
            ll h2 = (prefixHash.back().second * A2
                + c) % B2;
            prefixHash.pb({h1, h2});
        }
    }
    pll getHash(int l, int r) {
        ll h1 = (prefixHash[r + 1].first -
                prefixHash[l].first * A1pwrs[r + 1
                    - l]) %
            B1;
        ll h2 = (prefixHash[r + 1].second -
                prefixHash[l].second * A2pwrs[r +
                    1 - l]) %
            B2;
        if (h1 < 0) h1 += B1;
        if (h2 < 0) h2 += B2;
        return {h1, h2};
    }
};
```

6.2 Z Function

```
// z[i] stores the length of the longest substring
// starting from s[i] that matches the prefix of s.
vector<int> zFunction(const string& s) {
    int n = s.size();
    vector<int> z(n, 0);
    int l = 0, r = 0;

    for (int i = 1; i < n; i++) {
        if (i <= r) {
            z[i] = min(r - i + 1, z[i - l]);
        }
```

```
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            {
            z[i]++;
        }
        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }

    return z;
}
```

6.3 KMP

```
// LPS (Longest Prefix Suffix) indicates the length
    of the longest prefix
// that is also a suffix for any prefix of the pattern.
vector<int> computeLPS(const string& pattern) {
    int m = pattern.size();
    vector<int> lps(m, 0);
    int len = 0;
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
}

void KMP(const string& text, const string&
    pattern) {
    int n = text.size();
    int m = pattern.size();

    vector<int> lps = computeLPS(pattern);

    int i = 0;
    int j = 0;
    while (i < n) {
        if (text[i] == pattern[j]) {
            i++;
            j++;
        }

        if (j == m) {
            cout << "Pattern␣found␣at␣index␣"
                << (i - j) << endl;
            j = lps[j - 1];
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}
```

6.4 Manacher

```
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 1, r = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while (s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if (i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}

vector<int> manacher(string s) {
    string t;
    for (auto c : s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) -
        1);
}
```

7 Dynamic Programming

7.1 Convex Hull Trick

```
class CHT {
    // slope in increasing order
    deque<pair<Line, int>> dq;
public:
    void insert(int m, int c) {
        Line newLine(m, c);
        if(dq.empty()) {
            dq.push_back({newLine, INT_MN});
            return;
        }
```

```
        while(dq.size() > 1 && dq.back().second
            >= dq.back().first.intersect(newLine))
            {
            dq.pop_back();
        }

        dq.push_back({newLine, dq.back().first.
            intersect(newLine)});
    }

    int query1(int x) {
        while(dq.size() > 1) {
            if(dq[1].second <= x) dq.pop_front();
            else break;
        }
        return dq[0].first.getVal(x);
    }

    int query2(int x) {
        auto it = upper_bound(dq.begin(), dq.end
            (), make_pair(Line(0,0), x),
                            [&](const pair<Line,
                                int> &a, const
                                pair<Line, int>
                                &b) {
                                return a.
                                    second <
                                    b.second;
                            });
        it--;
        return (*it).first.getVal(x);
    }

};
```

7.2 Divide And Conquer

```
/*
Let A[i][j] be the optimal answer for using i objects
    to satisfy j first
requirements.
The recurrence is:
A[i][j] = min(A[i-1][k] + f(i, j, k)) where f is some
    function that denotes the
cost of satisfying requirements from k+1 to j using
    the i-th object.
Consider the recursive function calc(i, jmin, jmax,
    kmin, kmax), that calculates
all A[i][j] for all j in [jmin, jmax] and a given i
    using known A[i-1][*].
*/

void calc(int i, int jmin, int jmax, int kmin, int
    kmax) {
    if(jmin > jmax) return;
    int jmid = (jmin+jmax)/2;
    // calculate A[i][jmid] naively (for k in kmin...
        min(jmid, kmax){...})
    // let kmid be the optimal k in [kmin, kmax]
    calc(i, jmin, jmid-1, kmin, kmid);
    calc(i, jmid+1, jmax, kmid, kmax);
}

int main() {
    // set initial dp values
    FOR(i, start, k+1){
        calc(i, 0, n-1, 0, n-1);
    }
    cout << dp[k][n-1];
}
```

7.3 Optimizations

1. Convex Hull 1:
   - Recurrence: $dp[i] = \min_{j<i}\{dp[j] + b[j]\cdot a[i]\}$
   - Condition: $b[j] \geq b[j+1], a[i] \leq a[i+1]$
   - Complexity: $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n)$
2. Convex Hull 2:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + b[k]\cdot a[j]\}$
   - Condition: $b[k] \geq b[k+1], a[j] \leq a[j+1]$
   - Complexity: $\mathcal{O}(kn^2) \rightarrow \mathcal{O}(kn)$
3. Divide and Conquer:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + C[k][j]\}$
   - Condition: $A[i][j] \leq A[i][j+1]$
   - Complexity: $\mathcal{O}(kn^2) \rightarrow \mathcal{O}(kn\log(n))$
4. Knuth:
   - Recurrence: $dp[i][j] = \min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$
   - Condition: $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
   - Complexity: $\mathcal{O}(n^3) \rightarrow \mathcal{O}(n^2)$

Notes:

- $A[i][j]$ - the smallest k that gives the optimal answer
- $C[i][j]$ - some given cost function

8 Misc

8.1 Mo's Algorithm

Mo's algorithm processes a set of range queries on a static array. Each query is to calculate something base on the array values in a range $[a, b]$. The queries have to be known in advance. Let's divide the array into blocks of size $k = O(\sqrt{n})$. A query $[a_1, b_1]$ is processed before query $[a_2, b_2]$ if $\lfloor \frac{a_1}{k} \rfloor < \lfloor \frac{a_2}{k} \rfloor$ or $\lfloor \frac{a_1}{k} \rfloor = \lfloor \frac{a_2}{k} \rfloor$ and $b_1 < b_2$.

Example problem: counting number of distinct values in a range. We can process the queries in the described order and keep an array count, which knows how many times a certain value has appeared. When moving the boundaries back and forth, we either increase count$[x_i]$ or decrease it. According to value of it, we will know how the number of distinct values has changed (e.g. if count[x_i] has just become 1, then we add 1 to the answer, etc.).

8.2 Big Integer

```
/*
bigint a = "12345678901234567890123456789 0";
bigint b = "98765432109876543210987654321";

bigint sum = a + b; // Addition
bigint diff = a - b; // Subtraction
bigint prod = a * b; // Multiplication
bigint quot = a / b; // Division
bigint rem = a % b; // Modulo
*/

const int base = 1000000000;
const int base_digits = 9;
struct bigint {
    vector<int> a;
    int sign;
    int size() {
        if (a.empty()) return 0;
        int ans = (a.size() - 1) * base_digits;
        int ca = a.back();
        while (ca) ans++, ca /= 10;
        return ans;
    }
    bigint operator^(const bigint &v) {
        bigint ans = 1, x = *this, y = v;
        while (!y.isZero()) {
            if (y % 2) ans *= x;
            x *= x, y /= 2;
        }
        return ans;
    }
    string to_string() {
        stringstream ss;
        ss << *this;
        string s;
        ss >> s;
        return s;
    }
    int sumof() {
        string s = to_string();
        int ans = 0;
        for (auto c : s) ans += c - '0';
        return ans;
    }
    bigint() : sign(1) {}
    bigint(long long v) { *this = v; }
    bigint(const string &s) { read(s); }
    void operator=(const bigint &v) {
        sign = v.sign;
        a = v.a;
    }
    void operator=(long long v) {
        sign = 1;
        a.clear();
        if (v < 0) sign = -1, v = -v;
        for (; v > 0; v = v / base) a.push_back(v
            % base);
    }
    bigint operator+(const bigint &v) const {
        if (sign == v.sign) {
            bigint res = v;
            for (int i = 0, carry = 0;
                 i < (int)max(a.size(), v.a.size()) ||
                     carry; ++i) {
                if (i == (int)res.a.size()) res.a.
                    push_back(0);
                res.a[i] += carry + (i < (int)a.size
                    () ? a[i] : 0);
                carry = res.a[i] >= base;
                if (carry) res.a[i] -= base;
            }
            return res;
        }
        return *this - (-v);
    }
    bigint operator-(const bigint &v) const {
        if (sign == v.sign) {
            if (abs() >= v.abs()) {
                bigint res = *this;
                for (int i = 0, carry = 0; i < (int)v
                    .a.size() || carry; ++i) {
                    res.a[i] -= carry + (i < (int)v.
                        a.size() ? v.a[i] : 0);
                    carry = res.a[i] < 0;
                    if (carry) res.a[i] += base;
                }
                res.trim();
                return res;
            }
            return -(v - *this);
        }
        return *this + (-v);
    }
    void operator*=(int v) {
        if (v < 0) sign = -sign, v = -v;
        for (int i = 0, carry = 0; i < (int)a.size() ||
             carry; ++i) {
            if (i == (int)a.size()) a.push_back(0);
            long long cur = a[i] * (long long)v +
                carry;
            carry = (int)(cur / base);
            a[i] = (int)(cur % base);
        }
        trim();
    }
    bigint operator*(int v) const {
        bigint res = *this;
        res *= v;
        return res;
    }
    void operator*=(long long v) {
        if (v < 0) sign = -sign, v = -v;
```

```
    for (int i = 0, carry = 0; i < (int)a.size() ||
        carry; ++i) {
        if (i == (int)a.size()) a.push_back(0);
        long long cur = a[i] * (long long)v +
            carry;
        carry = (int)(cur / base);
        a[i] = (int)(cur % base);
    }
    trim();
}
bigint operator*(long long v) const {
    bigint res = *this;
    res *= v;
    return res;
}
friend pair<bigint, bigint> divmod(const
    bigint &a1, const bigint &b1) {
    int norm = base / (b1.a.back() + 1);
    bigint a = a1.abs() * norm;
    bigint b = b1.abs() * norm;
    bigint q, r;
    q.a.resize(a.a.size());
    for (int i = a.a.size() - 1; i >= 0; i--) {
        r *= base;
        r += a.a[i];
        int s1 = r.a.size() <= b.a.size() ? 0 : r.
            a[b.a.size()];
        int s2 = r.a.size() <= b.a.size() - 1 ? 0
            : r.a[b.a.size() - 1];
        int d = ((long long)base * s1 + s2) / b
            .a.back();
        r -= b * d;
        while (r < 0) r += b, --d;
        q.a[i] = d;
    }
    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim();
    r.trim();
    return make_pair(q, r / norm);
}
bigint operator/(const bigint &v) const {
    return divmod(*this, v).first; }
bigint operator%(const bigint &v) const {
    return divmod(*this, v).second; }
void operator/=(int v) {
    if (v < 0) sign = -sign, v = -v;
    for (int i = (int)a.size() - 1, rem = 0; i >=
        0; --i) {
        long long cur = a[i] + rem * (long long
            )base;
        a[i] = (int)(cur / v);
        rem = (int)(cur % v);
    }
    trim();
}
bigint operator/(int v) const {
    bigint res = *this;
    res /= v;
    return res;
}
int operator%(int v) const {
    if (v < 0) v = -v;
    int m = 0;
    for (int i = a.size() - 1; i >= 0; --i)
        m = (a[i] + m * (long long)base) % v;
    return m * sign;
}
void operator+=(const bigint &v) { *this = *
    this + v; }
void operator-=(const bigint &v) { *this = *
    this - v; }
void operator*=(const bigint &v) { *this = *
    this * v; }
void operator/=(const bigint &v) { *this = *
    this / v; }
bool operator<(const bigint &v) const {
    if (sign != v.sign) return sign < v.sign;
    if (a.size() != v.a.size())
        return a.size() * sign < v.a.size() * v.
            sign;
    for (int i = a.size() - 1; i >= 0; i--)
        if (a[i] != v.a[i]) return a[i] * sign < v.a
            [i] * sign;
    return false;
}
bool operator>(const bigint &v) const { return
    v < *this; }
bool operator<=(const bigint &v) const {
    return !(v < *this); }
bool operator>=(const bigint &v) const {
    return !(*this < v); }
bool operator==(const bigint &v) const {
    return !(*this < v) && !(v < *this);
}
bool operator!=(const bigint &v) const {
    return *this < v || v < *this; }
void trim() {
    while (!a.empty() && !a.back()) a.
        pop_back();
    if (a.empty()) sign = 1;
}
bool isZero() const { return a.empty() || (a.size
    () == 1 && !a[0]); }
bigint operator-() const {
    bigint res = *this;
    res.sign = -sign;
    return res;
}
bigint abs() const {
    bigint res = *this;
    res.sign *= res.sign;
    return res;
}
long long longValue() const {
    long long res = 0;
    for (int i = a.size() - 1; i >= 0; i--) res =
        res * base + a[i];
```

```
        return res * sign;
    }
    friend bigint gcd(const bigint &a, const bigint
         &b) {
        return b.isZero() ? a : gcd(b, a % b);
    }
    friend bigint lcm(const bigint &a, const bigint
         &b) {
        return a / gcd(a, b) * b;
    }
    void read(const string &s) {
        sign = 1;
        a.clear();
        int pos = 0;
        while (pos < (int)s.size() && (s[pos] == '-'
             || s[pos] == '+')) {
            if (s[pos] == '-') sign = -sign;
            ++pos;
        }
        for (int i = s.size() - 1; i >= pos; i -=
             base_digits) {
            int x = 0;
            for (int j = max(pos, i - base_digits +
                 1); j <= i; j++)
                x = x * 10 + s[j] - '0';
            a.push_back(x);
        }
        trim();
    }
    friend istream &operator>>(istream &stream,
         bigint &v) {
        string s;
        stream >> s;
        v.read(s);
        return stream;
    }
    friend ostream &operator<<(ostream &stream,
         const bigint &v) {
        if (v.sign == -1) stream << '-';
        stream << (v.a.empty() ? 0 : v.a.back());
        for (int i = (int)v.a.size() - 2; i >= 0; --i)
            stream << setw(base_digits) <<
                 setfill('0') << v.a[i];
        return stream;
    }
    static vector<int> convert_base(const vector<
         int> &a, int old_digits,
                                    int new_digits
                                        ) {
        vector<long long> p(max(old_digits,
             new_digits) + 1);
        p[0] = 1;
        for (int i = 1; i < (int)p.size(); i++) p[i] =
             p[i - 1] * 10;
        vector<int> res;
        long long cur = 0;
        int cur_digits = 0;
        for (int i = 0; i < (int)a.size(); i++) {
            cur += a[i] * p[cur_digits];
            cur_digits += old_digits;
            while (cur_digits >= new_digits) {
                res.push_back(int(cur % p[
                     new_digits]));
                cur /= p[new_digits];
                cur_digits -= new_digits;
            }
        }
        res.push_back((int)cur);
        while (!res.empty() && !res.back()) res.
             pop_back();
        return res;
    }
    typedef vector<long long> vll;
    static vll karatsubaMultiply(const vll &a, const
         vll &b) {
        int n = a.size();
        vll res(n + n);
        if (n <= 32) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++) res[i + j
                     ] += a[i] * b[j];
            return res;
        }
        int k = n >> 1;
        vll a1(a.begin(), a.begin() + k);
        vll a2(a.begin() + k, a.end());
        vll b1(b.begin(), b.begin() + k);
        vll b2(b.begin() + k, b.end());

        vll a1b1 = karatsubaMultiply(a1, b1);
        vll a2b2 = karatsubaMultiply(a2, b2);

        for (int i = 0; i < k; i++) a2[i] += a1[i];
        for (int i = 0; i < k; i++) b2[i] += b1[i];

        vll r = karatsubaMultiply(a2, b2);
        for (int i = 0; i < (int)a1b1.size(); i++) r[i]
             -= a1b1[i];
        for (int i = 0; i < (int)a2b2.size(); i++) r[i]
             -= a2b2[i];

        for (int i = 0; i < (int)r.size(); i++) res[i +
             k] += r[i];
        for (int i = 0; i < (int)a1b1.size(); i++) res
             [i] += a1b1[i];
        for (int i = 0; i < (int)a2b2.size(); i++) res
             [i + n] += a2b2[i];
        return res;
    }
    bigint operator*(const bigint &v) const {
        vector<int> a6 = convert_base(this->a,
             base_digits, 6);
        vector<int> b6 = convert_base(v.a,
             base_digits, 6);
        vll x(a6.begin(), a6.end());
        vll y(b6.begin(), b6.end());
        while (x.size() < y.size()) x.push_back(0);
        while (y.size() < x.size()) y.push_back(0);
```

```
        while (x.size() & (x.size() - 1)) x.
            push_back(0), y.push_back(0);
        vll c = karatsubaMultiply(x, y);
        bigint res;
        res.sign = sign * v.sign;
        for (int i = 0, carry = 0; i < (int)c.size(); i
            ++) {
            long long cur = c[i] + carry;
            res.a.push_back((int)(cur % 1000000));
            carry = (int)(cur / 1000000);
        }
        res.a = convert_base(res.a, 6, base_digits)
            ;
        res.trim();
        return res;
    }
};
```

8.3 Binary Exponentiation

```
ll pwr(ll a, ll b, ll m) {
    if(a == 1) return 1;
    if(b == 0) return 1;
    a %= m;
    ll res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

8.4 Builtin GCC Stuff

- __builtin_clz(x): the number of zeros at the beginning of the bit representation.
- __builtin_ctz(x): the number of zeros at the end of the bit representation.
- __builtin_popcount(x): the number of ones in the bit representation.
- __builtin_parity(x): the parity of the number of ones in the bit representation.
- __gcd(x, y): the greatest common divisor of two numbers.
- __int128_t: the 128-bit integer type. Does not support input/output.

9 Flow

9.1 Edmond Karp

```
// TC: O(E^2V)

int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur
                ][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[
                    cur][next]);
                if (next == t) return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

9.2 Dinics Algorithm

```
// TC: O(V^2E)

class FlowEdge {
public:
    int v, u;
    int cap, flow = 0;
    FlowEdge(int v, int u, int cap) : v(v), u(u), cap
        (cap) {}
};
```

```
class Dinic {
private:
    const int flow_inf = 1e15;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

public:
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, int cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap == edges[id].flow)
                     continue;
                if (level[edges[id].u] != -1) continue
                    ;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    int dfs(int v, int pushed) {
        if (pushed == 0) return 0;
        if (v == t) return pushed;
        for (int& cid = ptr[v]; cid < adj[v].size();
             cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u]) continue;
            int tr = dfs(u, min(pushed, edges[id].
                 cap - edges[id].flow));
            if (tr == 0) continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    int flow() {
        int f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (int pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

9.3 Push Relabel

```
// TC: > O(VE + V^2sqrt(E)), which in the worst
     case is O(V^3)

const int inf = 1000000000;

int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess;

void push(int u, int v)
{
    int d = min(excess[u], capacity[u][v] - flow[u][v
        ]);
    flow[u][v] += d;
    flow[v][u] -= d;
    excess[u] -= d;
    excess[v] += d;
}

void relabel(int u)
{
    int d = inf;
    for (int i = 0; i < n; i++) {
        if (capacity[u][i] - flow[u][i] > 0) d = min(d
             , height[i]);
    }
    if (d < inf) height[u] = d + 1;
}

vector<int> find_max_height_vertices(int s, int t
    ) {
    vector<int> max_height;
    for (int i = 0; i < n; i++) {
        if (i != s && i != t && excess[i] > 0) {
            if (!max_height.empty() && height[i]
                 > height[max_height[0]])
                 max_height.clear();
```

```
            if (max_height.empty() || height[i] ==
                height[max_height[0]])
                max_height.push_back(i);
        }
    }
    return max_height;
}

int max_flow(int s, int t)
{
    height.assign(n, 0);
    height[s] = n;
    flow.assign(n, vector<int>(n, 0));
    excess.assign(n, 0);
    excess[s] = inf;
    for (int i = 0; i < n; i++) {
        if (i != s) push(s, i);
    }

    vector<int> current;
    while (!(current = find_max_height_vertices(s
        , t)).empty()) {
        for (int i : current) {
            bool pushed = false;
            for (int j = 0; j < n && excess[i]; j++)
                {
                if (capacity[i][j] - flow[i][j] > 0 &&
                    height[i] == height[j] + 1) {
                    push(i, j);
                    pushed = true;
                }
            }
            if (!pushed) {
                relabel(i);
                break;
            }
        }
    }

    return excess[t];
}
```

9.4 Capacity Scaling

```
// TC: O(E^2 logCm)

class CapacityScalingSolver : public
    NetworkFlowSolverBase {
private:
    long long delta;

    long long dfs(int node, long long flow) {
        if (node == t) return flow;
        visit(node);
        for (Edge* edge : graph[node]) {
            long long cap = edge->
                remainingCapacity();
            if (cap >= delta && !visited(edge->to)
                ) {
                long long bottleNeck = dfs(edge->
                    to, min(flow, cap));
                if (bottleNeck > 0) {
                    edge->augment(bottleNeck);
                    return bottleNeck;
                }
            }
        }
        return 0;
    }

public:
    CapacityScalingSolver(int n, int s, int t) :
        NetworkFlowSolverBase(n, s, t), delta(0)
        {}

    void solve() override {
        delta = 1;
        while (delta <= INF) delta <<= 1;
        delta >>= 1;

        while (delta > 0) {
            long long f = 0;
            do {
                markAllNodesAsUnvisited();
                f = dfs(s, INF);
                maxFlow += f;
            } while (f != 0);
            delta >>= 1;
        }
    }
};
```