Algorithmic Notes For ICPC 2021

Oscar Skean

March 2022

1 Template

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

#define ll long long
#define SPEED ios::sync with stdio(false); cin.tie(0); cout.tie(0)
#define pb push back
#define rsz resize
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
#define FOR(i,a,b) for(int i=a;i<b;++i)
#define REP(i,n) FOR(i,0,n)


int main() {
    SPEED;
}
```

2 Data Structures

2.1 Segment Tree

```cpp
// load data directly into first row of seg tree
// make sure range is [start, end+1)
const int MAXN = 2e5 + 1;
ll seg[MAXN*4];
ll n;

void construct() {
    for (ll i = n-1; i > 0; i--) {
        seg[i] = seg[i<<1]  + seg[i<<1|1];
    }
}

void update(ll pos, ll val) {
    for (seg[pos += n] = val; pos > 1; pos >>= 1) {
        seg[pos>>1] = seg[pos] + seg[pos^1];
    }
}

void query(ll l, ll r) {
    ll sum = 0;
    for (l += n, r+= n; l < r; l >>= 1, r >>= 1) {
        if (l&1) sum += seg[l++];
        if (r&1) sum += seg[--r];
    }
    cout << sum << endl;
}
```

2.2 Minimum Sparse

```cpp
//read directly into first row of sparse table
#define MAXLOG 20
#define MAXN 200000
int sparse[MAXN][MAXLOG];
int logs[MAXN+1];

void construct(int n) {
    //build log table
    logs[1] = 0;
    for (int i = 2; i <= MAXN; i++) {
        logs[i] = logs[i/2] + 1;
    }

    //build sparse table
    for (int row = 1; row < MAXLOG; row++) {
        for (int i = 0; i + (1 << row) <= n; i++) {
            sparse[i][row] = min(sparse[i][row-1],
                                 sparse[i + (1 << (row-1))][row-1]);
        }
    }
}

int query(int l, int r) {
    int row = logs[r - l + 1];
    return min(sparse[l][row], sparse[r - (1 << row) + 1][row]);
}
```

2.3 Binary Jumping

```cpp
//read directly into first row of sparse table
#define MAXLOG 20
#define MAXN 200000
int sparse[MAXN][MAXLOG];
int logs[MAXN+1];

void construct(int n) {
    //build log table
    logs[1] = 0;
    for (int i = 2; i <= MAXN; i++) {
        logs[i] = logs[i/2] + 1;
    }

    //build sparse table
    for (int row = 1; row < MAXLOG; row++) {
        for (int i = 0; i + (1 << row) <= n; i++) {
            sparse[i][row] = min(sparse[i][row-1],
                                 sparse[i + (1 << (row-1))][row-1]);
        }
    }
}

int query(int l, int r) {
    int row = logs[r - l + 1];
    return min(sparse[l][row], sparse[r - (1 << row) + 1][row]);
}
```

3 Graph Algorithms

3.1 DFS with Cycle Detection

```cpp
void dfs(int s) {
    if (finished[s]) {
        return;
    }
    else if (seen[s]) {
        cout << "IMPOSSIBLE" << endl;
        exit(0);
    }


    seen[s] = true;
    for (int i : adj[s]) {
        dfs(i);
    }

    seen[s] = false;
    finished[s] = true;
```

3.2 BFS

```
const int MAXN = 1e5+1;
vector<int> adj[MAXN];

bool seen[MAXN];
int distances[MAXN];
int parents[MAXN];

void bfs(int start, int end) {
    queue<int> q;
    q.push(start);
    seen[start] = true;

    while (!q.empty()) {
        int a = q.front(); q.pop();

        //early break
        if (a == end) return;

        for (int b : adj[a]) {
            if (!seen[b]) {
                seen[b] = true;
                parents[b] = a;
                distances[b] = distances[a] + 1;
                q.push(b);
            }
        }
    }
}
```

3.3 BFS Route Reconstruction

```
//reconstruct the bfs route from parents array
void shortest route(int start, int end) {
    //run bfs
    bfs(start, end);

    if (distances[end] == 0) {
        cout << "IMPOSSIBLE" << endl;
        return;
    }

    // build route
    int length = distances[end];
    vector<int> route(length+1);

    int loc = end;
    for (int i = length; i >= 0; i--) {
        route[i] = loc;
        loc = parents[loc];
    }

    // print route
    cout << distances[end]+1 << endl;
    for(auto a : route) {
        cout << a << " ";
    }
    cout << endl;
}
```

3.4 Djikstra

```
//use with edges of form <node, weight>
const int MAXN = 1e5+1;
vector<pair<int, ll>> adj[MAXN];
ll distances[MAXN];
bool seen[MAXN];

void djikstra(int start, int n) {
    FOR(i, 2, n+1) {
        distances[i] = LONG MAX;
    }

    priority queue<pair<ll, int>> q;
    q.push({0, start});

    while (!q.empty()) {
        int a = q.top().second; q.pop();

        if (!seen[a]) {
            seen[a] = true;
            for (auto e : adj[a]) {
                int b; ll w;
                tie(b, w) = e;

                if (distances[a]+w < distances[b]) {
                    distances[b] = distances[a]+w;
                    q.push({-distances[b], b});
                }
            }
        }
    }
}
```

3.5 Bellman-Ford

```
void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    vector<int> p (n, -1);

    for (;;)
    {
        bool any = false;
        for (int j = 0; j < m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    p[e[j].b] = e[j].a;
                    any = true;
                }
        if (!any)  break;
    }

    if (d[t] == INF)
        cout << "No path from " << v << " to " << t << ".";
    else
    {
        vector<int> path;
        for (int cur = t; cur != -1; cur = p[cur])
            path.push back (cur);
        reverse (path.begin(), path.end());

        cout << "Path from " << v << " to " << t << ": ";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << ' ';
    }
}
```

3.6 Bellman-Ford with Negative Cycle Check

```
void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    vector<int> p (n - 1);
    int x;
    for (int i=0; i<n; ++i)
    {
        x = -1;
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
                    p[e[j].b] = e[j].a;
                    x = e[j].b;
                }
    }

    if (x == -1)
        cout << "No negative cycle from " << v;
    else
    {
        int y = x;
        for (int i=0; i<n; ++i)
            y = p[y];

        vector<int> path;
        for (int cur=y; ; cur=p[cur])
        {
            path.push back (cur);
            if (cur == y && path.size() > 1)
                break;
        }
        reverse (path.begin(), path.end());

        cout << "Negative cycle: ";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << ' ';
    }
}
```

3.7 Floyd-Warshall

```
const int MAXN = 505;
const ll bfn = 1e14;

ll distances[MAXN][MAXN];

void floyd(int n) {
    FOR(k, 1, n+1) {
        FOR(i, 1, n+1) {
            FOR(j, 1, n+1) {
                distances[i][j] = min(distances[i][j],
                        distances[i][k] + distances[k][j]);
            }
        }
    }
}

int main() {
    SPEED;
    int n,m,q;
    cin >> n >> m >> q;

    // prepare distances matrix
    REP(i, n+1) {
        REP(j, n+1) {
            distances[i][j] = bfn;
        }
    }
    REP(i, m) {
        ll a, b, c;
        cin >> a >> b >> c;

        distances[a][a] = 0; distances[b][b] = 0;
        distances[a][b] = distances[b][a] = min(distances[a][b], c);
    }

    floyd(n);

    REP(i, q) {
        int a, b;
        cin >> a >> b;
        ll res = distances[a][b];
        cout << (res < bfn ? res : -1) << endl;
    }
}
```

3.8 Find Bridges of Graph

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

vector<bool> visited;
vector<int> tin, low;
int timer;

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

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

3.9 Topological Sort

```
const int MAXN=1e5+1;
vector<int> adj[MAXN];
bool seen[MAXN];
bool finished[MAXN];

// remember to reverse sort this when printing it
vector<int> topo;

void dfs(int s) {
    if (finished[s]) {
        return;
    }
    else if (seen[s]) {
        cout << "IMPOSSIBLE" << endl;
        exit(0);
    }


    seen[s] = true;
    for (int i : adj[s]) {
        dfs(i);
    }

    seen[s] = false;
    finished[s] = true;
    topo.push back(s);
}

void solve(int n) {
    FOR(i, 1, n+1) {
        if (!finished[i]) dfs(i);
    }
}
```

3.10 Kruskal with DSU

```
// put weight first in edge tuple for sorting
vector<int> parent, ranks;
vector<tuple<ll, int, int>> edges;

void make set(int v) {
    parent[v] = v;
    ranks[v] = 0;
}

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

void union sets(int a, int b) {
    a = find set(a);
    b = find set(b);
    if (a != b) {
        if (ranks[a] < ranks[b])
            swap(a, b);
        parent[b] = a;
        if (ranks[a] == ranks[b])
            ranks[a]++;
    }
}

// can be modified to return cost or minimal edge set
ll kruskal(int n) {
    sort(edges.begin(), edges.end());

    parent.resize(n+1);
    ranks.resize(n+1);

    FOR(i, 1, n+1) {
        make set(i);
    }

    ll cost = 0;
    vector<pair<int,int>> result;

    for (auto e : edges) {
        ll w; int u, v;
        tie(w, u, v) = e;

        if (find set(u) != find set(v)) {
            cost += w;
            result.push back({u, v});
            union sets(u, v);
        }
    }

    // check for impossibility
    if (result.size() < n-1) {
        return -1;
    }

    return cost;
}
```

3.11 Connected Components

For counting, use DFS and increment whenever the recursive call is completely finished. For listing, keep a vector that gets appended to during DFS. Print the vector, then reset it for the next component.

3.12 Strongly Connected Components

```
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main() {
    int n;
    // ... read n ...

    for (;;) {
        int a, b;
        // ... read next directed edge (a,b) ...
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }

    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);

    used.assign(n, false);
    reverse(order.begin(), order.end());

    for (auto v : order)
        if (!used[v]) {
            dfs2 (v);

            // ... processing next component ...

            component.clear();
        }
```

3.13 Bipartite Check

```
// TWO COLORING
int n;
vector<vector<int>> adj;

//read in edges

vector<int> side(n, -1);
bool is_bipartite = true;
queue<int> q;
for (int st = 0; st < n; ++st) {
    if (side[st] == -1) {
        q.push(st);
        side[st] = 0;
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int u : adj[v]) {
                if (side[u] == -1) {
                    side[u] = side[v] ^ 1;
                    q.push(u);
                } else {
                    is_bipartite &= side[u] != side[v];
                }
            }
        }
    }
}

cout << (is_bipartite ? "YES" : "NO") << endl;
```

3.14 Maximum Flow

```
//FordFulkerson
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;
}
```

3.15 Minimum Cost Flow

```
//min cost flow
//finds cheapest flow from s to t with certain value K
//O(n^2 m^2)
struct Edge
{
    int from, to, capacity, cost;
};

vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
    }
}


int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF)
            break;

        // find max flow on that path
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K)
        return -1;
    else
        return cost;
}
```

3.16 Bipartite Matching

```
//Bipartite Matching
int n, k;
vector<vector<int>> g;
vector<int> mt; //keeps track of left side edge connections
vector<bool> used;

bool try_kuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    // ... reading the graph ...

    mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
        for (int to : g[v]) {
            if (mt[to] == -1) {
                mt[to] = v;
                used1[v] = true;
                break;
            }
        }
    }
    for (int v = 0; v < n; ++v) {
        if (used1[v])
            continue;
        used.assign(n, false);
        try_kuhn(v);
    }

    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1);
}
```

3.17 Number of Paths of Fixed Length

Suppose we have an adjacency matrix G, and we wish to find the number of paths with length k. G[i][j] is the number of edges from i to j. Raise the matrix G to the k-th power, then count the ones. Can use binary exponentiation if needed.

3.18 2SAT

```
// 2SAT
// looks for scc with kosaraju
int n;
vector<vector<int>> adj, adj_t;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : adj[v]) {
        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : adj_t[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    order.clear();
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }

    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}
```

```
void add_disjunction(int a, bool na, int b, bool nb) {
    // na and nb signify whether a and b are to be negated
    a = 2*a ^ na;
    b = 2*b ^ nb;
    int neg_a = a ^ 1;
    int neg_b = b ^ 1;
    adj[neg_a].push_back(b);
    adj[neg_b].push_back(a);
    adj_t[b].push_back(neg_a);
    adj_t[a].push_back(neg_b);
}
```

3.19 TSP

```
// dynamic tsp
#include<iostream>
using namespace std;
#define MAX 9999
int n=4; // Number of the places want to visit
 //Next distan array will give Minimum distance through all the position
int distan[10][10] = {
                    {0, 10, 15, 20},
                    {10, 0, 35, 25},
                    {15, 35, 0, 30},
                    {20, 25, 30, 0}
};
int completed_visit = (1<<n) - 1;
int DP[16][4];

int  TSP(int mark,int position){
  if(mark==completed_visit){      // Initially checking whether all
                                  // the places are visited or not
    return distan[position][0];
  }
  if(DP[mark][position]!=-1){
     return DP[mark][position];
  }
  //Here we will try to go to every other places to take the minimum
  // answer
  int answer = MAX;
  //Visit rest of the unvisited cities and mark the . Later find the
  //minimum shortest path
  for(int city=0;city<n;city++){
    if((mark&(1<<city))==0){
      int newAnswer = distan[position][city] + TSP( mark|(1<<city),city);
      answer = min(answer, newAnswer);
    }
  }
  return DP[mark][position] = answer;
}
int main(){
    /* initialize the DP array */
    for(int i=0;i<(1<<n);i++){
        for(int j=0;j<n;j++){
            DP[i][j] = -1;
        }
    }
    cout<<"Minimum Distance Travelled by you is "<<TSP(1,0);
    return 0;
}
```

3.20 Lowest Common Ancestor

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

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

    tout[v] = ++timer;
}

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

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

3.21 Eulerian Path

Eulerian cycle only exists if degree of every node is even. Eulerian path only exists if number of vertices with odd degree is 0 or 2.

```
stack St;
put start vertex in St;
until St is empty
  let V be the value at the top of St;
  if degree(V) = 0, then
    add V to the answer;
    remove V from the top of St;
  otherwise
    find any edge coming out of V;
    remove it from the graph;
    put the second end of this edge in St;
```

3.22 Hamiltonian Path

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

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

    tout[v] = ++timer;
}

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

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

3.23 Centers

Center of graph: set of vertices with minimum eccentricity. Use floyd warshall

Diameter of tree: BFS from v1. Last seen is v2. BFS from v2. Last seen is v3.

Center of tree: Do above and center is the middle of path from v2 to v3.

4 Dynamic Programming

4.1 Longest Increasing Subseqeuence

```
// performs DP algorithm
// initialize endings array to INT MAX
// endings array position i stores minimum ending to i-length increasing
void solve(int n) {
    int ans = 0;

    REP(i, n) {
        int bestLengthToAppendTo = binsearch(arr[i], 0, ans);

        if (arr[i] < endings[bestLengthToAppendTo]) {
            if (bestLengthToAppendTo == ans) {
                endings[ans] = arr[i];
                ans = max(1, ans+1);
            }
            else {
                endings[bestLengthToAppendTo] = arr[i];
            }

        }
    }

    cout << ans << endl;
}
```

4.2 Longest Common Subsequence

```
// LCS
#include <iostream>
#include <string>
using namespace std;

int LCSLength(string X, string Y)
{
    int m = X.length(), n = Y.length();

    int lookup[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        lookup[i][0] = 0;
    }
    for (int j = 0; j <= n; j++) {
        lookup[0][j] = 0;
    }

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (X[i - 1] == Y[j - 1]) {
                lookup[i][j] = lookup[i - 1][j - 1] + 1;
            }
            else  {
                lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1]);
            }
        }
    }

    return lookup[m][n];
}

int main()
{
    string X = "XMJYAUZ", Y = "MZJAWXU";

    cout << "The length of the LCS is " << LCSLength(X, Y);

    return 0;
}
```

4.3 Shortest Common Supersequence

```cpp
//SCS
#include <iostream>
#include <string>
using namespace std;

int SCSLength(string X, string Y)
{
    int m = X.length(), n = Y.length();

    int lookup[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        lookup[i][0] = i;
    }

    for (int j = 0; j <= n; j++) {
        lookup[0][j] = j;
    }

    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (X[i - 1] == Y[j - 1]) {
                lookup[i][j] = lookup[i - 1][j - 1] + 1;
            }

            else {
                lookup[i][j] = min(lookup[i - 1][j] + 1, lookup[i][j - 1] + 1);
            }
        }
    }

    return lookup[m][n];
}

int main()
{
    string X = "ABCBDAB", Y = "BDCABA";

    cout << "The length of the shortest common supersequence is "
         << SCSLength(X, Y);

    return 0;
}
```

4.4 Edit Distance

```cpp
int dist(string X, string Y)
{
    int m = X.length();
    int n = Y.length();

    int T[m + 1][n + 1];

    // initialize `T` by all 0's
    memset(T, 0, sizeof T);

    for (int i = 1; i <= m; i++) {
        T[i][0] = i;                // (case 1)
    }

    for (int j = 1; j <= n; j++) {
        T[0][j] = j;                // (case 1)
    }

    int substitutionCost;

    // fill the lookup table in a bottom-up manner
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (X[i - 1] == Y[j - 1]) {                 // (case 2)
                substitutionCost = 0;                   // (case 2)
            }
            else {
                substitutionCost = 1;                   // (case 3c)
            }
            T[i][j] = min(min(T[i - 1][j] + 1,          // deletion (case 3b)
                              T[i][j - 1] + 1),         // insertion (case 3a)
                          T[i - 1][j - 1] + substitutionCost);    // replace (case 2 & 3c)
        }
    }

    return T[m][n];
}

int main()
{
    string X = "kitten", Y = "sitting";

    cout << "The Levenshtein distance is " << dist(X, Y);

    return 0;
}
```

4.5 Coins Problem

```cpp
#define MAXN 1000005
#define INF 100000000
int main() {
    SPEED;
    int n, x;
    int c[100];
    int v[MAXN] = {0};
    cin >> n >> x;

    for (int i = 0; i < n; i++) {
        int q;
        cin >> q;
        c[i] = q;
    }

    for (int i = 1; i <= x; i++) {
        v[i] = INF;
        for (int j = 0; j < n; j++) {
            if (i-c[j] >= 0) {
                v[i] = min(v[i], v[i-c[j]]+1);
            }
        }
    }

    cout << (v[x] == INF ? -1 : v[x]) << endl;

}
```

4.6 Knapsack

```cpp
// KNAPSACK

// n is number of items. W is maximum cost allowed
int knapsack(vector<int>& cost, vector<int>& value, int n, int W) {
    // helper[i][j] stores maximum value that can be attained
    // with weight <= j and using only a subset of the first i
    int T[n+1][W+1] = {};

    FOR(i, 1, n+1) {
        FOR(j, 1, W+1) {

            T[i][j] = T[i-1][j];
            if (j - cost[i-1] >= 0) {
                T[i][j] = max(T[i][j],
                    T[i-1][j-cost[i-1]] + value[i-1]);
            }
        }
    }

    return T[n][W];
}
```

4.7 Partition

```cpp
//partition problem
// Returns true if there exists a subset of `array[0…n)` with the given
bool subsetSum(vector<int> const &nums, int sum)
{
    int n = nums.size();

    bool T[n + 1][sum + 1];

    for (int j = 1; j <= sum; j++) {
        T[0][j] = false;
    }

    for (int i = 0; i <= n; i++) {
        T[i][0] = true;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (nums[i - 1] > j) {
                T[i][j] = T[i - 1][j];
            }
            else {
                T[i][j] = T[i - 1][j] || T[i - 1][j - nums[i - 1]];
            }
        }
    }

    return T[n][sum];
}

// true if possible, false if not
bool partition(vector<int> const &nums)
{
    int sum = accumulate(nums.begin(), nums.end(), 0);
    return !(sum & 1) && subsetSum(nums, sum/2);
}
```

4.8 Rod Cutting

```cpp
// ROD CUTTING
int rodCut(int price[], int n) {
    int T[n + 1];

    for (int i = 0; i <= n; i++) {
        T[i] = 0;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            T[i] = max(T[i], price[j - 1] + T[i - j]);
        }
    }

    return T[n];
}

int main()
{
    int price[] = { 1, 5, 8, 9, 10, 17, 17, 20 };

    // rod length
    int n = 4;

    cout << "Profit is " << rodCut(price, n);

    return 0;
}
```

4.9 Word Break

```cpp
//WORD BREAK
bool wordBreak(unordered_set<string> const &dict, string word, vector<int> &lookup)
    int n = word.size();

    if (n == 0) {
        cout <<
        return true;
    }

    if (lookup[n] == -1) {
        lookup[n] = 0;

        for (int i = 1; i <= n; i++) {
            string prefix = word.substr(0, i);

            if (find(dict.begin(), dict.end(), prefix) != dict.end() &&
                wordBreak(dict, word.substr(i), lookup))
            {
                cout << prefix << endl;
                return lookup[n] = 1;
            }
        }
    }

    // return solution to the current subproblem
    return lookup[n];
}

// Word Break Problem Implementation in C++
int main() {
    unordered_set<string> dict = { "this", "th", "is", "famous", "Word", "break",
            "b", "r", "e", "a", "k", "br", "bre", "brea", "ak", "problem" };

    // input string
    string word = "Wordbreakproblem";

    vector<int> lookup(word.length() + 1, -1);

    if (wordBreak(dict, word, lookup)) {
        cout << "The string can be segmented";
    }
    else {
        cout << "The string can't be segmented";
    }

    return 0;
}
```

4.10 Counting Tilings

```cpp
// tilings
const int MOD = 1e9 + 7;

int dp[1 << 10][2];

int main() {
    cin.tie(0)->sync_with_stdio(0);
    int n, m;
    cin >> n >> m;
    dp[0][0] = 1;
    for (int j = 0; j < m; j++) for (int i = 0; i < n; i++) {
        for (int mask = 0; mask < (1 << n); mask++) {
            dp[mask][1] = dp[mask ^ (1 << i)][0]; // Vertical/no tile
            if (i && !(mask & (1 << i)) && !(mask & (1 << i - 1))) // Horizontal tile
                dp[mask][1] += dp[mask ^ (1 << i - 1)][0];

            if (dp[mask][1] >= MOD) dp[mask][1] -= MOD;
        }
        for (int mask = 0; mask < (1 << n); mask++) dp[mask][0] = dp[mask][1];
    }
    cout << dp[0][0];
    return 0;
}
```

5 Number Theoretic

5.1 Primality Testing

```
using u64 = uint64_t;
using u128 =   uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool MillerRabin(u64 n) { // returns true if n is prime, else retu
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

5.2 Euler Totient

```
// calculates phi from 1 to n
// uses sieve of eratosthenes
void calcPhi(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

5.3 Inclusion Exclusion

6 Strings

6.1 String Hashing

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

6.2 Count unique strings in array

```
vector<vector<int>> group_identical_strings(vector<string> const& s) {
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};

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

    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

6.3 Count unique substrings of string

```
int count_unique_substrings(string const& s) {
    int n = s.size();

    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i] - 'a' + 1) * p_pow[i]) % m;

    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l] + m - h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
    return cnt;
}
```

6.4 RabinKarp: String matching

```
vector<int> rabin karp(string const& s, string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();

    vector<long long> p pow(max(S, T));
    p pow[0] = 1;
    for (int i = 1; i < (int)p pow.size(); i++)
        p pow[i] = (p pow[i-1] * p) % m;

    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)
        h[i+1] = (h[i] + (t[i] - 'a' + 1) * p pow[i]) % m;
    long long h s = 0;
    for (int i = 0; i < S; i++)
        h s = (h s + (s[i] - 'a' + 1) * p pow[i]) % m;

    vector<int> occurences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur h = (h[i+S] + m - h[i]) % m;
        if (cur h == h s * p pow[i] % m)
            occurences.push back(i);
    }
    return occurences;
}
```

6.5 Knuth-Morris-Pratt

```
vector<int> prefix function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

Uses: 1) Check if t is in s. Run KMP with the string s+#+t.

6.6 Manacher Palindromes

```
vector<int> manacherodd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 0, 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 Miscellaneous

7.1 Binary Search

```
// find  what location key should go in array
int binsearch(int key, int l, int r) {
    while (l <= r) {
        int mid = (l + r) / 2;
        if (key < arr[mid]) r = mid - 1;
        else if (key > arr[mid]) l = mid + 1;
        else return mid;
    }

    return l;
}
```

7.2 Binary Exponentiation

```
const ll MOD = (ll) 1e9 + 7;

void exponentiation(ll a, ll b) {
    ll val = 1;
    while (b > 0) {
        if (b & 1) {
            val *= a;
        }
        a *= a;

        a %= MOD;
        val %= MOD;
        b >>= 1;
    }

    cout << val << endl;
}
```

7.3 Gray Code

```
vector<string> construct(int n) {
    vector<string> vec;

    //base case
    if (n == 1) {
        vec.pb("1");
        vec.pb("0");
        return vec;
    }

    // recusive reflection algorithm
    //
    vector<string> prev = construct(n-1);
    for (auto it = prev.begin(); it != prev.end(); it++) {
        vec.pb("0" + *it);
    }

    for (auto it = prev.rbegin(); it != prev.rend(); it++) {
        vec.pb("1" + *it);
    }

    return vec;

}
```

7.4 Towers of Hanoi

```
// call like hanoi(n, 1, 2, 3)
vector<pair<int,int>> moves;

void hanoi(int d, int l, int m, int r) {
    if (d == 1) {
        moves.pb(make pair(l, r));
        return;
    }

    else {
        hanoi(d-1, l, r, m);
        moves.pb(make pair(l, r));
        hanoi(d-1, m, l, r);
    }
}
```

7.5 Expression Parsing

```
bool delim(char c) {
    return c == ' ';
}

bool is op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}
```

```
int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is op(s[i])) {
            char cur op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur op)) {
                process op(st, op.top());
                op.pop();
            }
            op.push(cur op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }

    while (!op.empty()) {
        process op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

7.6 Balanced Sequences

```
//balanced sequence
bool next_balanced_sequence(string & s) {
    int n = s.size();
    int depth = 0;
    for (int i = n - 1; i >= 0; i--) {
        if (s[i] == '(')
            depth--;
        else
            depth++;

        if (s[i] == '(' && depth > 0) {
            depth--;
            int open = (n - i - 1 - depth) / 2;
            int close = n - i - 1 - open;
            string next = s.substr(0, i) + ')' + string(open, '(') + string(close, '
            s.swap(next);
            return true;
        }
    }
    return false;
}
```

7.7 Korder Statistic

C++ standard library has this implemented already. The function is called *nth_element*.

```
int main()
{
    std::vector<int> v{5, 10, 6, 4, 3, 2, 6, 7, 9, 3};
    printVec(v);

    auto m = v.begin() + v.size()/2;
    std::nth_element(v.begin(), m, v.end());
    std::cout << "\nThe median is " << v[v.size()/2] << '\n';
    // The consequence of the inequality of elements before/after the Nth one:
    assert(std::accumulate(v.begin(), m, 0) < std::accumulate(m, v.end(), 0));
    printVec(v);

    // Note: comp function changed
    std::nth_element(v.begin(), v.begin()+1, v.end(), std::greater{});
    std::cout << "\nThe second largest element is " << v[1] << '\n';
    std::cout << "The largest element is " << v[0] << '\n';
```

7.8 Josephus Queries

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

typedef tree<int,null_type,less<int>,rb_tree_tag, tree_order_statistics_node_update> indexed_set;
#define int long long
#define endl '\n'

signed main(){
    ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);
    #ifdef LOCAL
    freopen("input.txt", "r" , stdin);
    freopen("output.txt", "w", stdout);
    #endif

    indexed_set s;
    int n,k; cin>>n>>k;
    for (int i = 1; i <= n; i++)
        s.insert(i);

    int ind = k%n;
    while(n--) {
        auto y = s.find_by_order(ind);
        cout<<*y<<' ';
        s.erase(y);
        if (n) ind = (ind%n + k)%n;
    }
}
```

7.9 Convex Hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i], include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```