

# ICPC Team Notebook

typedef unsigned long long uwu

Sorbonne Université

SWERC 2025

---

---

*A curated reference of algorithms and data structures*



November 15, 2025

# Contents

<b>1</b>	<b>Number Theory</b>
1.1	Euler totient . . . . .
1.2	Fast exponentiation . . . . .
1.3	Sieve of Eratosthenes . . . . .
<b>2</b>	<b>Data Structures</b>
2.1	Fenwick Tree . . . . .
2.2	RMQ . . . . .
2.3	Segment Tree(s) . . . . .
2.4	Trie . . . . .
2.5	Union find . . . . .
<b>3</b>	<b>Graph Algorithms</b>
3.1	2-SAT . . . . .
3.2	Articulation Points and Bridges . . . . .
3.3	Cycle detection . . . . .
3.4	Euler paths . . . . .
3.5	Hungarian . . . . .
3.6	Iterative dfs . . . . .
3.7	LCA . . . . .
3.8	Max Bipartite Matching . . . . .
3.9	Max Flow Min Cut . . . . .
3.10	Min Cost Flow . . . . .
3.11	Minimum Spanning Tree . . . . .
3.12	Shortest Path . . . . .
3.13	Strongly Connected Components . . . . .
3.14	TSP . . . . .
3.15	Toposort . . . . .
<b>4</b>	<b>Strings</b>
4.1	Aho-Corasick . . . . .
4.2	Knuth-Morris-Pratt . . . . .
4.3	Manacher . . . . .
4.4	SuffixArray . . . . .
<b>5</b>	<b>Arrays</b>
5.1	Inversions . . . . .
<b>6</b>	<b>Utilities</b>
6.1	STL . . . . .

## 1 Number Theory

### 1.1 Euler totient

Number of ints  $\leq n$  coprime to  $n$  (c.f. sieve for primes).

```
int phi(int n, vi& primes) {
    int res = n;
    for (int p : primes) {
        if (1LL * p * p > n) break;
        if (n % p == 0) {
            while (n % p == 0) n /= p;
            res -= res / p;
        }
    }
    if (n > 1) res -= res / n;
    return res;
}
```

### 1.2 Fast exponentiation

$O(\log b)$ : fast  $a^b \bmod p$ .

```
int modular_exp(int a, int b, int p){
    int res = 1;
    while(b > 0){
        if(b & 1) res = (1LL * a * res) % p;
        b = b >> 1;
        a = (1LL * a * a) % p;
    }
    return res;
}
```

### 1.3 Sieve of Eratosthenes

$O(n \log \log n)$  sieve[i] = 0 if i prime,  $\text{spf}$  otherwise for fast fact

```
vi sieve(int n){
    vi sieve(n,0);
    for(int i = 2; i*i < n ; i++)
        if(!sieve[i])
            for(int j = i*i ; j <n ; j += i)
                if(!sieve[j]) sieve[j] = i;
    return sieve;
}
```

## 2 Data Structures

### 2.1 Fenwick Tree

Point update, prefix and range sum,  $O(\log n)$ ,  $O(n)$  build

```
struct Fenwick {
    int n; vector<ll> t; //using long long, int might overflow.
    Fenwick(vector<ll>& a): n(a.size()), t(n+1,0) {
        for(int i = 1; i <=n; i++) { //Builds tree from array in O(n)
            t[i] += a[i-1];
            int p = i+(i&-i);
            if(p<=n) t[p] += t[i];
        }
        void add(int i,long long v){
            for(;i<=n;i+=i&-i) t[i]+=v;
        }
        long long sum(int i){
            long long r=0;
            for(; i>0 ;i -= i&-i) r += t[i];
            return r;
        }
        long long sum(int l,int r){ return sum(r)-sum(l-1); }
    };
};
```

### 2.2 RMQ

Preprocess an array in  $O(n \log n)$  to answer range minimum queries in  $O(1)$

```
struct RMQ {
    vvi jmp;
    RMQ(const vi& V) { // Python: depth = N.bit_length()
        int N = V.size(), depth = 32 - __builtin_clz(N);
        jmp.assign(depth, V);
        for(int i = 0; i < depth-1; ++i)
            for(int j = 0; j < N; ++j)
                jmp[i+1][j] = min(jmp[i][j],
                                   jmp[i][min(N-1, j + (1<<i))]);
    }// Returns min{V[a], V[a+1], ..., V[b-1]}
    int query(int a, int b) {
        if(b <= a) return INT_MAX;
        int dep = 31 - __builtin_clz(b - a); // python: (b-a).bit_length() - 1
    }
};
```

```
    return min(jmp[dep][a], jmp[dep][b - (1<<dep)]);
}
};
```

## 2.3 Segment Tree(s)

Regular seg tree: point update, range sum query,  $O(\log n)$

```
struct SegTree {
    int n; vector<ll> t;
    SegTree(vi &a): n(a.size()), t(4*n) { build(a,1,0,n); }
    void build(vi &a, int v, int l, int r){
        if(r-l == 1){ t[v]=a[1]; return; }
        int m = (l+r)/2;
        build(a, 2*v, l, m); build(a, 2*v+1, m, r);
        t[v] = t[2*v] + t[2*v+1];
    }
    ll sum_aux(int v, int l, int r, int ql, int qr){
        if(qr <= l || r <= ql) return 0;
        if(ql <= l && r <= qr) return t[v];
        int m = (l+r)/2;
        return sum_aux(2*v, l,m,ql,qr) + sum_aux(2*v+1, m,r,ql,qr);
    }
    ll sum(int ql, int qr){ return sum_aux(1,0,n,ql,qr); }
    void upd_aux(int v, int l, int r, int pos, ll val){
        if(r-l==1){ t[v]=val; return; }
        int m = (l+r)/2;
        if(pos<m) upd_aux(2*v,l,m,pos,val);
        else upd_aux(2*v+1,m,r,pos,val);
        t[v]=t[2*v]+t[2*v+1];
    }
    void upd(int pos,ll val){ upd_aux(1,0,n,pos,val); }
};
```

Lazy SegTree: range add, range sum query,  $O(\log n)$

```
struct LazySegTree {
    int n; vector<ll> t, lazy;
    LazySegTree(vi &a):n(a.size()),t(4*n),lazy(4*n){build(a,1,0,n);}
    void build(vi &a, int v, int l, int r){
        if(r-l == 1){ t[v]=a[1]; return; }
        int m = (l+r)/2;
        build(a,2*v,l,m); build(a,2*v+1,m,r);
        t[v] = t[2*v]+t[2*v+1];
    }
    void push(int v, int l, int r){
        if(lazy[v]==0) return;
        t[v] += lazy[v]*(r-l);
        if(r-l>1){
            lazy[2*v] += lazy[v];
            lazy[2*v+1] += lazy[v];
        }
        lazy[v]=0;
    }
    void upd_aux(int v, int l, int r, int ql, int qr, ll val){
        push(v,l,r);
        if(qr<=l || r<=ql) return;
        if(ql<=l && r<=qr){
            lazy[v] += val; push(v,l,r);
            return;
        }
        int m = (l+r)/2;
        upd_aux(2*v, l,m,ql,qr,val);
        upd_aux(2*v+1, m,r,ql,qr,val);
        t[v] = t[2*v] + t[2*v+1];
    }
    ll sum_aux(int v, int l, int r, int ql, int qr){
```

```
    push(v,l,r);
    if(qr<=l || r<=ql) return 0;
    if(ql<=l && r<=qr) return t[v];
    int m = (l+r)/2;
    return sum_aux(2*v, l,m,ql,qr) + sum_aux(2*v+1, m,r,ql,qr);
}
void upd(int ql,int qr,ll val){ upd_aux(1,0,n,ql,qr,val); }
ll sum(int ql,int qr){ return sum_aux(1,0,n,ql,qr); }
```

Iterative SegTree: point update, range sum query,  $O(\log n)$

```
struct IterSegTree {
    int n; vector<ll> t;
    IterSegTree(vi &a): n(a.size()), t(2*n) {
        for(int i=0;i<n;i++) t[n+i]=a[i];
        for(int i=n-1;i>0;i--) t[i]=t[i<<1]+t[i<<1|1];
    }
    void upd(int pos,ll val){
        pos+=n; t[pos]=val;
        for(pos>>=1; pos>0; pos>>=1) t[pos]=t[pos<<1]+t[pos<<1|1];
    }
    ll sum(int l,int r){
        ll res=0;
        for(l+=n,r+=n;l<r;l>>=1,r>>=1){
            if(l&1) res+=t[l++];
            if(r&1) res+=t[-r];
        }
        return res;
    }
};
```

## 2.4 Trie

Prefix tree: stores strings, insert/find in  $O(|s|)$ ; handles lowercase letters a-z

```
struct Node{
    int next[26]; bool isEnd=false;
    Node(){fill(begin(next), end(next), -1);}
};
struct Trie{
    vector<Node> t;
    Trie() {t.emplace_back();}
    void insert(string s){
        int cur = 0;
        for (auto c : s){
            if (t[cur].next[c-'a'] == -1){
                t.emplace_back(); // Adds new node
                t[cur].next[c-'a'] = t.size() - 1;
            }
            cur = t[cur].next[c-'a'];
        }
        t[cur].isEnd = true;
    }
    bool find(string s){
        int cur = 0;
        for (auto c : s){
            if (t[cur].next[c-'a'] == -1) return false;
            cur = t[cur].next[c-'a'];
        }
        return t[cur].isEnd;
    }
};
```

## 2.5 Union find

Complexity: effectively  $O(1)$

```

struct union_find{
    vector<int> rank, parent;
    union_find(int n){
        rank.resize(n, 0); parent.resize(n);
        for (int i = 0; i < n; i++) parent[i] = i;
    }
    int find(int i){
        int root = parent[i];
        if (parent[root] != root) return parent[i] = find(root);
        return root;
    }
    void unite(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot) return;
        if (rank[xRoot] < rank[yRoot]) parent[xRoot] = yRoot;
        else if (rank[yRoot] < rank[xRoot]) parent[yRoot] = xRoot;
        else{
            parent[yRoot] = xRoot;
            rank[xRoot]++;
        }
    }
};
```

## 3 Graph Algorithms

### 3.1 2-SAT

Check if  $n$  variables satisfy  $(x_i \vee x_j) \wedge (\neg x_k \vee x_l) \wedge \dots$

```

struct TwoSAT {
    int n; SCC g; // c.f. strongly connected components
    TwoSAT(int vars): n(vars), g(2*vars) {}
    int var(int i){ return i; }
    int neg(int i){ return i+n; }
    void add_clause(int a, bool a_val, int b, bool b_val){
        int u = a_val ? var(a) : neg(a);
        int v = b_val ? var(b) : neg(b);
        g.add_edge(u^1, v); //  $\neg a \rightarrow b$ . ^ bitwise XOR, same in python
        g.add_edge(v^1, u); //  $\neg b \rightarrow a$ 
    }
    bool satisfiable(vi& ans){
        g.kosaraju();
        ans.assign(n, 0);
        for (int i=0; i<n; i++){
            if (g.scc[var(i)] == g.scc[neg(i)]) return false;
            ans[i] = g.scc[var(i)] > g.scc[neg(i)];
        }
        return true;
    }
};
```

### 3.2 Articulation Points and Bridges

$O(V + E)$

AP = vertices whose removal increases # of connected components.

Bridge = edges whose removal increases # of connected components.

```

void tarjan(int u, vvi&adj, vi&vis, vi&disc, vi&low,
            int &time, int p, vi&isAP, set<pii>&bridges) {
    vis[u] = 1;
    disc[u] = low[u] = ++time;
    int children = 0;
    for (int v : adj[u]) {
        if (v == p) continue;
        if (!vis[v]) {
```

```

            children++;
            tarjan(v, adj, vis, disc, low, time, u, isAP, bridges);
            low[u] = min(low[u], low[v]);
            if (low[v] > disc[u]) // Bridge
                bridges.insert({min(u, v), max(u, v)});
            if (p != -1 && low[v] >= disc[u])
                isAP[u] = 1; // Articulation Point
            }
            else low[u] = min(low[u], disc[v]);
        }
        if (p == -1 && children > 1)
            isAP[u] = 1;
    }
    pair<vi, set<pii>> findAPandBridges(int V, vvi&adj) {
        vi disc(V), low(V), vis(V), isAP(V); // Initialize with 0s
        set<pii> bridges;
        int time = 0;
        for (int i = 0; i < V; i++)
            if (!vis[i])
                tarjan(i, adj, vis, disc, low, time, -1, isAP, bridges);
        return {isAP, bridges};
    }
}
```

### 3.3 Cycle detection

Returns a cycle in a directed graph, or empty if none exists.  $O(n + m)$

```

vi dfs_cycle(const vvi&adj) {
    int n = adj.size();
    vi color(n, 0), path;
    vi cycle;
    function<bool(int)> dfs = [&](int v) {
        color[v] = 1; path.push_back(v);
        for (int u : adj[v]){
            if (color[u] == 0 && dfs(u)) return true;
            else if (color[u] == 1) {
                auto it = find(path.begin(), path.end(), u);
                cycle.assign(it, path.end());
                return true;
            }
        }
        path.pop_back(); color[v] = 2;
        return false;
    };
    for (int i = 0; i < n; i++)
        if (color[i] == 0 && dfs(i)) break;
    return cycle;
}
```

Check if undirected graph contains a cycle in  $O(m)$ .

```

bool dsu_cycle(int n, vvi&edges) {
    union_find uf(n); //c.f. union find
    for (auto e : edges) {
        if (uf.find(e[0]) == uf.find(e[1])) return true;
        uf.unite(e[0], e[1]);
    }
    return false;
}
```

Tortoise & Hare: finds start of cycle in a functional graph (-1 if none).  $O(n)$

```

int floyd_cycle(int start, vi& nxt){
    int tort = start, hare = start;
    do {
        if (hare == -1 || nxt[hare] == -1) return -1;
        tort = nxt[tort];
```

```

    hare = nxt[nxt[hare]];
} while(tort!=hare);
tort=start;
while(tort!=hare){
    tort=nxt[tort]; hare=nxt[hare];
}
return tort;
}

```

### 3.4 Euler paths

Euler Path/Circuits:

Undirected:

- Circuit: all vertices even degree and connected.
- Path: exactly 2 odd-degree vertices and connected; start at one of them.
- Trick: to use circuit function for path, add dummy edge between odd vertices, find circuit, remove dummy.

Directed:

- Circuit: strongly connected,  $\text{indegree}[v] == \text{outdegree}[v] \forall v$ .
- Path: one vertex with  $\text{out-in}=1(\text{start})$ , one with  $\text{in-out}=1(\text{end})$ , rest balanced.
- Trick: add dummy edge end- $\text{start}$ , find circuit, rotate to remove dummy.

Find Euler path in directed graph.  $O(E)$  MODIFIES THE GRAPH!!!

```

vi euler_directed(vvi &adj, int start=0) {
    vi path; stack<int> st; st.push(start);
    while(!st.empty()) {
        int v = st.top();
        if(!adj[v].empty()) {
            int u = adj[v].back(); adj[v].pop_back();
            st.push(u);
        } else {
            path.push_back(v);
            st.pop();
        }
    }
    reverse(path.begin(), path.end());
    return path; // size = E+1 if valid Euler tour
}

```

Input:  $\text{adj}[u][i] = v, \text{id}$ ,  $\text{adj}[v][j] = u, \text{id}$  if  $u$  and  $v$  adjacent (same id for both edges).

```

vi euler_undirected(int n, vector<vpi>& adj) {
    vi used, path; stack<int> st; st.push(0);
    int m=0; for (auto &v:adj) m += v.size(); used.assign(m/2,0);
    while(!st.empty()){
        int v=st.top();
        while(!adj[v].empty() && used[adj[v].back().second])
            adj[v].pop_back();
        if(adj[v].empty()){
            path.push_back(v);
            st.pop();
        } else {
            auto e=adj[v].back(); adj[v].pop_back();
            used[e.second]=1;
            st.push(e.first);
        }
    }
    reverse(path.begin(), path.end());
    return path;
}

```

### 3.5 Hungarian

Min cost assignment  $O(n^3)$ . Input cost matrix, output (min cost, assignment)

```

pair<int, vi> hungarian(const vvi &a) {
    int n = a.size(), m = a[0].size();

```

```

    vi u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    for (int i = 1; i <= n; i++) {
        p[0] = i;
        vi minv(m + 1, 1e9); // INF constant
        vector<bool> used(m + 1, false);
        int j0 = 0;
        do {
            used[j0] = true;
            int i0 = p[j0], delta = 1e9, j1 = 0;
            for (int j = 1; j <= m; j++) if (!used[j]) {
                int cur = a[i0 - 1][j - 1] - u[i0] - v[j];
                if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                if (minv[j] < delta) delta = minv[j], j1 = j;
            }
            for (int j = 0; j <= m; j++) {
                if (used[j]) u[p[j]] += delta, v[j] -= delta;
                else minv[j] -= delta;
            }
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    vi match(n);
    for (int j = 1; j <= m; j++) if (p[j]) match[p[j] - 1] = j - 1;
    return { -v[0], match };
}

```

### 3.6 Iterative dfs

Iterative depth-first search

```

void dfs_iter(int start, vvi& adj, vi &output, vi& visited) {
    stack<int> st; st.push(start); visited[start] = 1;
    while (!st.empty()) {
        int v = st.top(); bool done = true;
        for (auto u : adj[v]) {
            if (!visited[u]) {
                visited[u] = 1; st.push(u);
                done = false; break;
            }
        }
        if (done) {
            st.pop(); output.push_back(v);
        }
    }
}

```

### 3.7 LCA

```

struct LCA {
    int n; vvi adj;
    vi depth, first, euler;
    RMQ rmq; // c.f. RMQ structure.
    LCA(int _n) : n(_n), adj(n), depth(n), first(n), rmq(vi()) {}
    void add_edge(int u,int v){ adj[u].push_back(v); adj[v].push_back(u); }
    void dfs(int v, int p, int d) {
        first[v] = euler.size();
        depth[v] = d;
        euler.push_back(v);
        for(int u: adj[v]) if(u != p) {

```

```

dfs(u, v, d+1);
euler.push_back(v);
}

void build(int root=0) {
    euler.clear();
    dfs(root,-1,0);
    vi D(euler.size());
    for(int i=0;i<euler.size();i++) D[i]=depth[euler[i]];
    rmq = RMQ(D);
} // dist(a,b) = depth[a] + depth[b] - 2 * depth[lca];
int query(int a,int b){
    int l = first[a], r = first[b];
    if(l>r) swap(l,r);
    return euler[rmq.query(l,r+1)];
}
};
```

## 3.8 Max Bipartite Matching

**König's theorem:** In a bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

Hopcroft–Karp  $O(E\sqrt{V})$  left  $[0..nL-1]$ , right  $[0..nR-1]$  adj from left to right.

```

int nL, nR;
vector<vi>adj;
vi dist, matchL, matchR;
bool bfs() {
    queue<int> q;
    dist.assign(nL, -1);
    for (int u = 0; u < nL; ++u)
        if (matchL[u] == -1) dist[u] = 0, q.push(u);
    bool found = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            int mu = matchR[v];
            if (mu == -1) found = 1;
            else if (dist[mu] == -1) dist[mu] = dist[u] + 1, q.push(mu);
        }
    }
    return found;
}
bool dfs(int u) {
    for (int v : adj[u]) {
        int mu = matchR[v];
        if (mu == -1 || (dist[mu] == dist[u] + 1 && dfs(mu))) {
            matchL[u] = v; matchR[v] = u;
            return 1;
        }
    }
    dist[u] = -1;
    return 0;
}
int hopcroftKarp() {
    matchL.assign(nL, -1);
    matchR.assign(nR, -1);
    int matching = 0;
    while (bfs())
        for (int u = 0; u < nL; ++u)
            if (matchL[u] == -1 && dfs(u))
                ++matching;
    return matching;
}
```

## 3.9 Max Flow Min Cut

Dinic's max flow  $O(V^2E)$ ,  $O(E\sqrt{V})$  for bipartite/unit. Edges from reachable nodes after flow form a min cut.

```

struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vi> adj;
    int n, m = 0;
    int s, t; // source, target
    vi level, ptr;
    queue<int> q;
    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, long long 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;
    }
    long long dfs(int v, long long pushed) {
        if (pushed == 0) return 0;
        if (v == t) return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid]; int u = edges[id].u;
            if (level[v] + 1 != level[u]) continue;
            long long 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;
    }
    long long flow() {
        long long 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 (long long pushed = dfs(s, flow_inf))
                f += pushed;
        }
    }
}
```

```

    return f;
}

vector<pii> min_cut_edges() {
    vector<bool> vis(n, false);
    queue<int> q;
    q.push(s); vis[s] = true;
    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int id : adj[v]) {
            auto &e = edges[id];
            if (!vis[e.u] && e.cap > e.flow) {
                vis[e.u] = true;
                q.push(e.u);
            }
        }
    }
    vector<pii> cut;
    for (auto &e : edges) {
        if (vis[e.v] && !vis[e.u] && e.cap > 0) {
            cut.push_back({e.v, e.u});
        }
    }
    return cut;
}

```

### 3.10 Min Cost Flow

Finds min cost to send up to F flow (set F=INF for MCMF)  $O(FE \log V)$ . Handles neg costs (no neg cycles).

```

struct MCF {
    struct E { int v, rev, cap; ll cost; };
    int n; vector<vector<E>> g;
    MCF(int n): n(n), g(n) {}
    void addEdge(int u,int v,int cap,ll cost){
        g[u].push_back({v,(int)g[v].size(),cap,cost});
        g[v].push_back({u,(int)g[u].size()-1,0,-cost});
    }
    pair<ll,ll> minCostFlow(int s,int t,int maxf){
        ll flow=0, cost=0; vll pot(n), dist(n); vi pv(n), pe(n);
        while(flow<maxf){
            fill(dist.begin(),dist.end(),LLONG_MAX);
            priority_queue<pii,vector<pii>,greater<pii>> pq;
            dist[s]=0; pq.push({0,s});
            while(!pq.empty()){
                auto [d,u]=pq.top(); pq.pop();
                if(d!=dist[u]) continue;
                for(int i=0;i<(int)g[u].size();i++){
                    E &e=g[u][i]; if(!e.cap) continue;
                    ll nd=d+e.cost+pot[u]-pot[e.v];
                    if(nd<dist[e.v]){
                        dist[e.v]=nd; pv[e.v]=u; pe[e.v]=i;
                        pq.push({nd,e.v});
                    }
                }
            }
            if(dist[t]==LLONG_MAX) break;
            for(int i=0;i<n;i++) if(dist[i]<LLONG_MAX) pot[i]+=dist[i];
            int add=maxf-flow;
            for(int v=t;v!=s;v=pv[v])
                add=min(add,g[pv[v]][pe[v]].cap);
            for(int v=t;v!=s;v=pv[v]){
                E &e=g[pv[v]][pe[v]];
                e.cap-=add; g[v][e.rev].cap+=add;
            }
        }
    }
}

```

```

        cost+=add*e.cost;
    }
    flow+=add;
}
return {flow,cost};
}

```

### 3.11 Minimum Spanning Tree

Kruskal  $O(E \log E)$  edges list of {u,v,weight}

```

int kruskal(int V, vvi &edges) {
    sort(edges.begin(), edges.end(), [] (auto a, auto b){return a[2]<b[2];});
    union_find uf(V);
    int cost = 0, count = 0;
    for (auto &e : edges) {
        if (uf.find(e[0]) != uf.find(e[1])) {
            uf.unite(e[0], e[1]);
            cost += e[2];
            if (++count == V - 1) break;
        }
    }
    return cost;
}

```

Prim's  $O((E + V) \log V)$  adj[u] list of (v,weight) adjacent to u.

```

int prim(int V, vector<vpi> &adj) {
    priority_queue<pii, vector<pii>, greater<pii>> q;
    vector<bool> vis(V, false);
    q.push({0, 0});
    int res = 0;
    while(!q.empty()){
        auto [w,u] = q.top(); q.pop();
        if(vis[u]) continue;
        res += w;
        vis[u] = true;
        for(auto v : adj[u]){
            if(!vis[v.first]){
                q.push({v.second, v.first});
            }
        }
    }
    return res;
}

```

### 3.12 Shortest Path

Find shortest paths from src (no negative weights).  $O((V+E)\log V)$

```

vi dijkstra(const vector<vector<pii>>& adj, int src) {
    vi dist(adj.size(), INT_MAX);
    priority_queue<pii, vector<pii>, greater<pii>> q;
    dist[src] = 0; q.push({0, src});
    while (!q.empty()) {
        auto [d, u] = q.top(); q.pop();
        if (d != dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (d+w < dist[v]) {
                dist[v] = d+w;
                q.push({d+w, v});
            }
        }
    }
    return dist;
}

```

$O(VE)$  Shortest Path+neg edges; BFS from nodes with dist  $-\infty$  for all neg-cycle reachable.

```
vi bellmanFord(int n, vvi& edges, int src) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    for (int i = 0; i < n; i++) {
        for (auto edge : edges) {
            int u = edge[0]; int v = edge[1]; int wt = edge[2];
            if (dist[u] != INT_MAX && dist[u] + wt < dist[v]) {
                if(i == n - 1) return {-1};
                dist[v] = dist[u] + wt;
            }
        }
    }
    return dist;
}
```

All-pairs shortest paths (neg edges ok, no neg cycles)  $O(V^3)$   $graph[i][i] = 0$ ,  $graph[i][j] = w$  if edge  $i \rightarrow j$  else  $INT\_MAX$

```
vector<vi> floydWarshall(vector<vi> graph) {
    int V = graph.size();
    auto 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] < INT_MAX && dist[k][j] < INT_MAX)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
    return dist;
}
```

### 3.13 Strongly Connected Components

Kosaraju SCC  $O(n+m)$

```
struct SCC {
    int n, ordn = 0, scc_cnt = 0;
    vvi adj, adjt; vi vis, ord, scc;
    SCC(int n): n(n), adj(n), adjt(n), vis(n,0), ord(n), scc(n) {}
    void add_edge(int u,int v){ adj[u].push_back(v); adjt[v].push_back(u); }
    void dfs(int u){
        vis[u]=1;
        for(int v:adj[u]) if(!vis[v]) dfs(v);
        ord[ordn++]=u;
    }
    void dfst(int u){
        scc[u]=scc_cnt; vis[u]=0; // SCCs IDs are in reverse topological order.
        for(int v:adjt[u]) if(vis[v]) dfst(v);
    }
    void kosaraju(){
        for(int i=0;i<n;i++) if(!vis[i]) dfs(i);
        for(int i=ordn-1;i>=0;i--) if(vis[ord[i]]){ scc_cnt++; dfst(ord[i]); }
    }
};
```

### 3.14 TSP

Traveling Salesman Problem  $O(n^2 2^n)$

```
int tsp(int n, vvi& dist) {
    int mask_limit = 1 << n;
    vvi dp(mask_limit, vi(n, INT_MAX));
    dp[1][0] = 0;
    for (int mask = 1; mask < mask_limit; mask++) {
        for (int last = 0; last < n; last++) {
            if (dp[mask][last] == INT_MAX) continue;
            for (int next = 0; next < n; next++) {
                if (mask & (1 << next)) continue;
```

```
                int new_mask = mask | (1 << next);
                dp[new_mask][next] = min(dp[new_mask][next],
                                         dp[mask][last] + dist[last][next]);
            }
        }
        int ans = INT_MAX;
        for (int last = 1; last < n; last++) {
            if (dp[mask_limit - 1][last] != INT_MAX && dist[last][0] != INT_MAX) {
                ans = min(ans, dp[mask_limit - 1][last] + dist[last][0]);
            }
        }
        return ans;
    }
}
```

## 3.15 Toposort

TopoSort via DFS  $O(V + E)$ .

```
void dfs(int u, vector<vi> &adj, vi &vis, vi &res) {
    vis[u] = 1;
    for (int v : adj[u])
        if (!vis[v])
            dfs(v, adj, vis, res);
    res.push_back(u);
}
vi toposort(vector<vi> &adj) {
    int n = adj.size();
    vi vis(n, 0), res;
    for (int i = 0; i < n; i++)
        if (!vis[i]) dfs(i, adj, vis, res);
    reverse(res.begin(), res.end());
    return res;
}
```

## 4 Strings

### 4.1 Aho-Corasick

Finds the occurrences of a set of patterns in a text simultaneously  $O(n + m + matches)$ , where  $n = \text{len}(\text{text})$ ,  $m = \text{total pattern length}$ .

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.output = []
        self.fail = None
    def build_automaton(keywords):
        root = TrieNode()
        for keyword in keywords:
            node = root
            for char in keyword:
                node = node.children.setdefault(char, TrieNode())
            node.output.append(keyword)
        queue = deque()
        for node in root.children.values():
            node.fail = root
            queue.append(node)
        while queue:
            current = queue.popleft()
            for ch, nxt in current.children.items():
                queue.append(nxt)
                f = current.fail
                while f and ch not in f.children:
                    f = f.fail
                f = f.fail
                nxt.fail = f.children[ch] if f else root
```

```

nxt.output += nxt.fail.output
return root
def search_text(text, keywords):
    root = build_automaton(keywords)
    res = {kw: [] for kw in keywords}
    node = root
    for i, ch in enumerate(text):
        while node and ch not in node.children:
            node = node.fail
        if not node:
            node = root
            continue
        node = node.children[ch]
        for kw in node.output:
            res[kw].append(i - len(kw) + 1)
    return res

```

## 4.2 Knuth-Morris-Pratt

Search for a pattern within a text in  $O(n + m)$ , where  $n$  is the size of the text and  $m$  the length of the pattern. Length of the longest proper prefix which is also a suffix of the substring ending at that position.

```

def constructLps(pat, lps):
    len_ = 0 ; m = len(pat) ; lps[0] = 0 ; i = 1
    while i < m:
        if pat[i] == pat[len_]:
            len_ += 1
            lps[i] = len_
            i += 1
        else:
            if len_ != 0:
                len_ = lps[len_ - 1]
            else:
                lps[i] = 0
                i += 1
def search(pat, txt):
    n = len(txt) ; m = len(pat) ; lps = [0] * m ; res = []
    constructLps(pat, lps)
    i = 0 ; j = 0
    while i < n:
        if txt[i] == pat[j]:
            i += 1 # If characters match, move both pointers forward
            j += 1
            if j == m: # found
                res.append(i - j)
                j = lps[j - 1] # Use LPS to skip unnecessary comparisons
        else: # mismatch
            if j != 0:
                j = lps[j - 1] # Use LPS to skip unnecessary comparisons
            else:
                i += 1
    return res

```

## 4.3 Manacher

Longest palindromic substring  $O(n)$ . Max of P corresponds to len(LPS)

```

def manacher(s):
    T = '#' + '#' .join(s) + '#' # handle even-length, remember to remove after
    n = len(T)
    P = [0]*n # P[i] = radius of palindrome centered at i
    C = R = 0 # current center, right boundary
    for i in range(n):
        mirr = 2*C - i
        if i < R: P[i] = min(R-i, P[mirr]) # use symmetry

```

```

while i+P[i]+1 < n and i-P[i]-1 >= 0 and T[i+P[i]+1] == T[i-P[i]-1]:
    P[i] += 1 # attempt to expand palindrome at i
    if i + P[i] > R: # update center and right boundary
        C, R = i, i + P[i]

```

## 4.4 SuffixArray

SuffixArray i.e. starting positions of suffixes in lexicographic order.  $O(n \log^2 n)$

```

def build_sa(s):
    n = len(s)
    sa = list(range(n))
    rank = [ord(ch) for ch in s] # initial ranks by 1 char
    k = 1
    while k < n: # sort by first k chars
        sa.sort(key=lambda x: (rank[x], rank[x+k] if x+k < n else -1))
        temp = [0]*n
        temp[sa[0]] = 0
        for i in range(1, n):
            prev = sa[i-1]
            cur = sa[i]
            prev_key = (rank[prev], rank[prev+k] if prev+k < n else -1)
            cur_key = (rank[cur], rank[cur+k] if cur+k < n else -1)
            temp[cur] = temp[prev] + (cur_key > prev_key)
        rank = temp
        if rank[sa[-1]] == n-1: break # all ranks distinct
        k <= 1
    return sa

```

Longest Common Prefix between suffixes  $sa[i]$  and  $sa[i+1]$   $O(n)$

```

def build_lcp(s, sa):
    n = len(s)
    rank = [0]*n
    for i in range(n): rank[sa[i]] = i
    lcp = [0]*n
    k = 0
    for i in range(n):
        if rank[i] == n-1: k = 0; continue
        j = sa[rank[i] + 1]
        while i+k < n and j+k < n and s[i+k] == s[j+k]:
            k += 1
        lcp[rank[i]] = k
        if k: k -= 1
    return lcp

```

Longest Repeated Substring = max LCP over all adjacent suffixes.

```

def lrs(s):
    sa = build_sa(s)
    lcp = build_lcp(s, sa)
    idx = lcp.index(max(lcp))
    return (lcp[idx], idx) # (length, position in SA)

```

Finds all occurrences of pat in s via binary search on the SA. Returns [L, R]: all SA positions where suffix starts with pat.

```

def search(s, pat, sa):
    n, m = len(s), len(pat)
    l, r = 0, n-1
    while l <= r:
        mid = (l+r)//2
        if s[sa[mid]:sa[mid]+m] < pat: l = mid+1
        else: r = mid-1
    start = l
    l, r = 0, n-1
    while l <= r:
        mid = (l+r)//2

```

```

if s[sa[mid]:sa[mid]+m] <= pat: l = mid+1
else: r = mid-1
return (start, r) if start <= r else (-1, -1)

```

Longest Common Substring between two strings. Build SA/LCP on  $s1\#s2\$$ , then the LCS occurs between adjacent suffixes coming from different strings.

```

def lcs(s1, s2):
    s = s1 + '#' + s2 + '$'
    sa = build_sa(s)
    lcp = build_lcp(s, sa)
    idx, max_len = 0, 0
    for i in range(1, len(s)):
        a = sa[i] < len(s1)
        b = sa[i-1] < len(s1)
        if a != b and lcp[i] > max_len:
            max_len, idx = lcp[i], i
    return (max_len, idx)

```

## 5 Arrays

### 5.1 Inversions

Count pairs where order flips between arrays.  $O(n \log n)$

```

ll inversions(vi& a, vi& b) {
    int n = a.size();
    unordered_map<int,int> pos;
    for (int i = 0; i < n; i++) pos[b[i]] = i + 1;
    Fenwick t(n); // C.f. Fenwick tree
}

```

```

ll inv = 0;
for (int i = 0; i < n; i++) {
    inv += i - t.sum(pos[a[i]]);
    t.add(pos[a[i]], 1);
}
return inv;
}

```

## 6 Utilities

### 6.1 STL

```

int main()
{

```

Lambda functions

```

int a;
auto f = [](){};           // empty capture: no outer variables, only parameters +
                           // globals
auto f = [=](int x){cout <<a;};
auto f = [&](int x){a=2;};

```

Sort using custom function

```

vector<pair<int,int>> P = {{1,2},{3,0},{2,5}};
sort(P.begin(), P.end(), [](auto &a, auto &b){ return a.second < b.second; });
}

```