

# ICPC Team Notebook

typedef unsigned long long ll;  
Sorbonne Université  
SWERC 2025

---

---

*A curated reference of algorithms and data structures*



November 8, 2025

Contents

1

Number Theory

2

1.1

Euler totient . . . . .

2

1.2

Fast exponentiation . . . . .

2

1.3

Sieve of Eratosthenes . . . . .

2

2

Data Structures

2

2.1

Fenwick Tree . . . . .

2

2.2

Union find . . . . .

2

3

Graph Algorithms

2

3.1

Max Bipartite Matching . . . . .

2

3.2

Max Flow Min Cut . . . . .

3

3.3

Shortest Path . . . . .

4

3.4

TSP . . . . .

4

3.5

Toposort . . . . .

4

4

Arrays

4

4.1

Inversions . . . . .

4

# 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, 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 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 Max Bipartite Matching

**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;
}

```

## 3.2 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;
    }
}

```

```

void 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;
}

```

```
vector<int> bellmanFord(int n, vector<vector<int>>& edges, int src) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    for (int i = 0; i < n; i++) {
        for (vector<int> 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;
}
```

```
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;
}
```

### 3.4 TSP

```

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;
}

```

### 3.5 Toposort

```
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 = 1; i < n; i++)
        if (!vis[i]) dfs(i, adj, vis, res);
    reverse(res.begin(), res.end());
    return res;
}
```

## 4 Arrays

## 4.1 Inversions

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

```
11 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;  
}
```