

# Competitive Programming Reference

ngmh

Last Updated: 07/01/2026

## Contents

<b>1 Data Structures</b>	<b>3</b>
1.1 Prefix Sums . . . . .	3
1.1.1 1D . . . . .	3
1.1.2 2D . . . . .	3
1.2 Fenwick Trees . . . . .	3
1.2.1 Point Update Range Query . . . . .	3
1.2.2 Range Update Point Query . . . . .	4
1.2.3 Range Update Range Query . . . . .	4
1.2.4 2D PURQ / RUPQ . . . . .	4
1.2.5 2D RURQ . . . . .	5
1.3 Segment Trees . . . . .	6
1.3.1 Standard . . . . .	6
1.3.2 Lazy Propagation . . . . .	6
1.3.3 Maxsum . . . . .	7
1.3.4 Merge Sort Tree / Order Statistics . . . . .	7
1.3.5 2D . . . . .	8
<b>2 Graph Theory</b>	<b>9</b>
2.1 Depth First Search . . . . .	9
2.2 Breadth First Search . . . . .	9
2.3 0-1 BFS . . . . .	10
2.4 Floyd-Warshall . . . . .	10
2.5 Bellman-Ford . . . . .	10
2.6 Dijkstra's Algorithm . . . . .	11
2.7 Shortest Path Faster Algorithm . . . . .	11
2.8 Prim's Algorithm . . . . .	11
2.9 Union Find Disjoint Subset . . . . .	12
2.10 Kruskal's Algorithm . . . . .	12
2.11 Topological Sort . . . . .	12
2.12 Floyd's Cycle Finding Algorithm . . . . .	13
2.13 Maximum Cardinality Bipartite Matching . . . . .	13
2.13.1 Kun's Algorithm . . . . .	13
2.13.2 Hopcroft-Karp . . . . .	14
2.14 Articulation Points and Bridges . . . . .	15
2.15 Strongly Connected Components . . . . .	16
2.16 Travelling Salesman Problem . . . . .	16
2.16.1 Bottom Up . . . . .	16
2.16.2 Top Down . . . . .	17
2.17 Trees . . . . .	17
2.17.1 Pre/Postorder Traversal . . . . .	17
2.17.2 Subtree to Range . . . . .	18
2.17.3 Weighted Maximum Independent Set . . . . .	18
2.17.4 Diameter . . . . .	19
2.17.5 $2^K$ Decomposition . . . . .	19
2.17.6 Lowest Common Ancestor . . . . .	19

2.17.7	Shortest Path . . . . .	20
2.17.8	Heavy Light Decomposition . . . . .	20
2.17.9	Centroid Decomposition . . . . .	21
<b>3</b>	<b>Dynamic Programming</b>	<b>22</b>
3.1	Maxsum . . . . .	22
3.1.1	1D . . . . .	22
3.2	Longest Increasing Subsequence . . . . .	22
3.2.1	$N^2$ DP . . . . .	22
3.2.2	Data Structure Speedup . . . . .	23
3.2.3	$N \log N$ DP . . . . .	23
3.3	Coin Combinations . . . . .	23
3.4	Coin Change . . . . .	23
3.5	Knapsack . . . . .	24
3.5.1	0-1 . . . . .	24
3.6	Digit DP . . . . .	24
3.7	Convex Hull Trick . . . . .	25
3.8	Li Chao Tree . . . . .	26
3.9	Divide and Conquer . . . . .	27
<b>4</b>	<b>Math</b>	<b>28</b>
4.1	Fast Exponentiation . . . . .	28
4.2	Prime Factorisation . . . . .	28
4.3	Sieve of Eratosthenes . . . . .	28
4.4	Greatest Common Divisor . . . . .	29
4.5	Lowest Common Multiple . . . . .	29
4.6	Modular Inverse . . . . .	29
4.7	$\binom{n}{k}$ . . . . .	29
4.8	Fibonacci . . . . .	30
<b>5</b>	<b>Algorithms</b>	<b>30</b>
5.1	Binary Search . . . . .	30
5.2	Binary Search using Lifting . . . . .	30
5.3	Sliding Set . . . . .	31
5.4	Set Merging . . . . .	31
5.5	Discretisation . . . . .	31
5.6	Meet in the Middle . . . . .	31
<b>6</b>	<b>Miscellaneous</b>	<b>32</b>
6.1	Fast I/O . . . . .	32
6.2	Superfast I/O . . . . .	32

# 1 Data Structures

Data Structure	Precomputation / Update	Query	Memory	Notes
Prefix Sum	$O(N) / X$	$O(1)$	$O(N)$	Associative Functions (+, XOR)
Sparse Table	$O(N \log N) / X$	$O(1)$	$O(N \log N)$	Non-Associative Functions (max, gcd)
Fenwick Tree	$X / O(\log N)$	$O(\log N)$	$O(N)$	Prefix Sum with Updates
Segment Tree	$X / O(\log N)$	$O(\log N)$	$O(4N)$	Allows more Information

Table 1: Quick Summary of Data Structures

## 1.1 Prefix Sums

### 1.1.1 1D

$O(N)$  precomputation,  $O(1)$  query.

---

```
//Query - 1-Indexed
int query(int s, int e){
    return ps[e]-ps[s-1];
}

//Precomputation
ps[0] = 0;
for(int i = 1; i <= n; i++) ps[i] = ps[i-1]+a[i];
```

---

### 1.1.2 2D

$O(R \cdot C)$  precomputation,  $O(1)$  query.

---

```
//Query - 1-Indexed
int query(int x1, int y1, int x2, int y2){
    return ps[x2][y2]-ps[x1-1][y2]-ps[x2][y1-1]+ps[x1-1][y1-1];
}

//Precomputation
for (int i = 0; i <= r; i++) ps[i][0] = 0;
for (int j = 0; j <= c; j++) ps[0][j] = 0;
for (int i = 1; i <= r; i++) {
    for (int j = 1; j <= c; j++) {
        ps[i][j] = ps[i-1][j]+ps[i][j-1]-ps[i-1][j-1]+a[i][j];
    }
}
```

---

## 1.2 Fenwick Trees

### 1.2.1 Point Update Range Query

$O(\log N)$  update and query.

---

```
inline int ls(int x){ return (x)&(-x); }

int fw[MAXN]; // 1-Indexed

void pu(int i, int v) {
    for(; i <= n; i += ls(i)) fw[i] += v;
}

int pq(int i) {
    int t = 0;
```

---

```

    for(; i; i -= ls(i)) t += fw[i];
    return t;
}

int rq(int s, int e) {
    return pq(e) - pq(s - 1);
}

```

---

### 1.2.2 Range Update Point Query

$O(\log N)$  update and query.

---

```

// Requires PURQ Code (PU, PQ)

void ru(int s, int e, int v) {
    pu(s, v);
    pu(e+1, -v);
}

```

---

### 1.2.3 Range Update Range Query

$O(\log N)$  update and query.

---

```

// Requires PURQ Code (PU, PQ)
// Functions need to be modified to take in array parameter
// e.g. int pu(*tree, int i, int v)

void ru(int s, int e, int v) {
    pu(fw1, s, v);
    pu(fw1, e+1, -v);
    pu(fw2, s, -v*(s-1));
    pu(fw2, e+1, v*e);
}

int ps(int i) {
    return pq(fw1, i)*i + pq(fw2, i);
}

int rq(int s, int e) {
    return ps(e) - ps(s - 1);
}

```

---

### 1.2.4 2D PURQ / RUPQ

$O(\log N \cdot \log M)$  update and query.

---

```

inline int ls(int x) { return x & -x; }

int fw[MAXN][MAXN]; // 1-indexed

void pu(int x, int y, int v) {
    for (int i = x; i < MAXN; i += ls(i)) {
        for (int j = y; j < MAXN; j += ls(j)) {
            fw[i][j] += v;
        }
    }
}

```

```

int pq(int x, int y) {
    int ans = 0;
    for (int i = x; i > 0; i -= ls(i)) {
        for (int j = y; j > 0; j -= ls(j)) {
            ans += fw[i][j];
        }
    }
    return ans;
}

// Range Query
int rq(int x1, int y1, int x2, int y2) {
    return pq(x2, y2) - pq(x1 - 1, y2) - pq(x2, y1 - 1) + pq(x1 - 1, y1 - 1);
}

// Range Update, Point Query
void ru(int x1, int y1, int x2, int y2, int v) {
    pu(x1, y1, v);
    pu(x1, y2 + 1, -v);
    pu(x2 + 1, y1, -v);
    pu(x2 + 1, y2 + 1, v);
}

```

---

### 1.2.5 2D RURQ

$O(\log N \cdot \log M)$  update and query.

---

```

// Requires 2D PURQ Code

void ru(int x1, int y1, int x2, int y2, long long v) {
    pu(fw1, x1, y1, v);
    pu(fw1, x1, y2+1, -v);
    pu(fw1, x2+1, y1, -v);
    pu(fw1, x2+1, y2+1, v);

    pu(fw2, x1, y1, v*(x1-1));
    pu(fw2, x1, y2+1, -v*(x1-1));
    pu(fw2, x2+1, y1, -v*x2);
    pu(fw2, x2+1, y2+1, v*x2);

    pu(fw3, x1, y1, v*(y1-1));
    pu(fw3, x1, y2+1, -v*y2);
    pu(fw3, x2+1, y1, -v*(y1-1));
    pu(fw3, x2+1, y2+1, v*y2);

    pu(fw4, x1, y1, v*(x1-1)*(y1-1));
    pu(fw4, x1, y2+1, -v*(x1-1)*y2);
    pu(fw4, x2+1, y1, -v*x2*(y1-1));
    pu(fw4, x2+1, y2+1, v*x2*y2);
}

long long ps(int x, int y) {
    return pq(fw1, x, y)*x*y - pq(fw2, x, y)*y - pq(fw3, x, y)*x + pq(fw4, x, y);
}

long long rq(int x1, int y1, int x2, int y2) {
    return ps(x2, y2) - ps(x1-1, y2) - ps(x2, y1-1) + ps(x1-1, y1-1);
}

```

---

## 1.3 Segment Trees

### 1.3.1 Standard

$O(\log N)$  point update and range query.

---

```
struct node {
    int s, e, m, v;
    node *l, *r;
    node(int _s, int _e) {
        s = _s; e = _e; m = (s+e)/2; v = 0;
        if (s != e) {
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    void pu(int x, int y) {
        if (s == e) { v = y; return; }
        if (x <= m) l->pu(x, y);
        if (x > m) r->pu(x, y);
        v = min(l->v, r->v);
    }
    int rq(int x, int y) {
        if (s == x && e == y) return v;
        if (y <= m) return l->rq(x, y);
        if (x > m) return r->rq(x, y);
        return min(l->rq(x, m), r->rq(m+1, y));
    }
} *root;
root = new node(0, n-1);
```

---

### 1.3.2 Lazy Propagation

$O(\log N)$  range update and range query.

---

```
struct node {
    int s, e, m, v, lazy;
    node *l, *r;
    node(int _s, int _e) {
        s = _s; e = _e; m = (s+e)/2; v = lazy = 0;
        if (s != e) {
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    int pu() {
        if (s == e) { v += lazy; lazy = 0; return v; }
        v += lazy;
        l->lazy += lazy; r->lazy += lazy;
        lazy = 0;
        return v;
    }
    void ru(int x, int y, int z) {
        if (s == x && e == y) { lazy += z; return; }
        if (y <= m) l->ru(x, y, z);
        else if (x > m) r->ru(x, y, z);
        else l->ru(x, m, z), r->ru(m+1, y, z);
    }
}
```

---

```

    v = max(l->pu(), r->pu());
}

int rq(int x, int y) {
    pu();
    if (s == x && e == y) return pu();
    if (y <= m) return l->rq(x, y);
    if (x > m) return r->rq(x, y);
    return max(l->rq(x, m), r->rq(m+1, y));
}
} *root;

root = new node(0, n-1);

```

---

### 1.3.3 Maxsum

$O(\log N)$  point update and range query.

---

```

struct node {
    ll s, e, m, ps, ss, ms, ts;
    node *l, *r;
    node(ll _s, ll _e) {
        s = _s; e = _e; m = (s+e)/2; ps = ss = ms = ts = 0;
        if (s != e) {
            l = new node(s, m);
            r = new node(m+1, e);
        }
    }
    void pu(ll x, ll y) {
        if (s == e) { ps = ss = ms = ts = y; return; }
        if (x <= m) l->pu(x, y);
        if (x > m) r->pu(x, y);
        //New Prefix Max -> Left Prefix, Left + Right Prefix
        ps = max(l->ps, l->ts+r->ps);
        //New Suffix Max -> Right Suffix, Left Suffix + Right
        ss = max(r->ss, r->ts+l->ss);
        //Total Sum -> Left + Right
        ts = l->ts+r->ts;
        //Maxsum - Left Suffix + Right Prefix, Left, Right,
        //          Total, Left Maxsum, Right Maxsum
        ms = max({l->ss+r->ps, ps, ss, ts, l->ms, r->ms});
    }
    ll ans() {
        return ms;
    }
} *root;

root = new node(0, n-1);

```

---

### 1.3.4 Merge Sort Tree / Order Statistics

Insertion:  $O(\log N)$ , Building:  $O(N \log^2 N)$ , Counting:  $O(\log^2 N)$ , Finding:  $O(\log V \cdot \log^2 N)$ , Range Max:  $O(\log N)$ .

---

```

struct node {
    int s, e, m;
    vector<int> v;
    node *l, *r;

```

```

node(int _s, int _e) {
    s = _s; e = _e; m = (s+e)/2;
    if (s != e) {
        l = new node(s, m);
        r = new node(m+1, e);
    }
}
void insert(int x, int y) {
    if (s == e) { v.push_back(y); return; }
    if (x > m) r->insert(x, y);
    if (x <= m) l->insert(x, y);
    v.push_back(y);
}
void build(){
    if (s == e) return;
    l->build();
    r->build();
    sort(v.begin(), v.end());
}
int countLessEqual(int x, int y, int k) {
    if (x > y) return 0;
    if (s == x && e == y) {
        return upper_bound(v.begin(), v.end(), k)-v.begin();
    }
    if (x > m) return r->countLessEqual(x, y, k);
    if (y <= m) return l->countLessEqual(x, y, k);
    return l->countLessEqual(x, m, k)+r->countLessEqual(m+1, y, k);
}
int kthSmallest(int x, int y, int k) {
    int mini = 0, maxi = (1 << 30);
    int ans = mini, gap = maxi;
    while (gap > 0) {
        while (ans + gap <= maxi && countLessEqual(x, y, ans + gap) < k) {
            ans += gap;
        }
        gap >>= 1;
    }
    return ans + 1;
}
int rangeMax(int x, int y) {
    if (x > y) return 0;
    if (s == x && e == y) return v.back();
    if (x > m) return r->rangeMax(x, y);
    if (y <= m) return l->rangeMax(x, y);
    return max(l->rangeMax(x, m), r->rangeMax(m+1, y));
}
} *root;
root = new node(0, n-1);

```

---

### 1.3.5 2D

$O(N \cdot \log N \cdot \log M)$  point update and range query.

---

```

struct node2D {
    int s, e, m;
    node1D *maxi;
    node2D *l, *r;
}

```

```

node2D(int a, int b, int c, int d) {
    s = a; e = b; m = (s+e)/2;
    maxi = new node1D(c, d);
    if (s != e) {
        l = new node2D(s, m, c, d);
        r = new node2D(m+1, e, c, d);
    }
}
void pu(int a, int b, int v) {
    if (s == e) { maxi->pu(b, v); return; }
    if (a <= m) l->pu(a, b, v);
    else r->pu(a, b, v);
    maxi->pu(b, max(l->maxi->rq(b, b), r->maxi->rq(b, b)));
}
int rq(int a, int b, int c, int d) {
    if (s == a && e == b) return maxi->rq(c, d);
    if (b <= m) return l->rq(a, b, c, d);
    if (a > m) return r->rq(a, b, c, d);
    return max(l->rq(a, m, c, d), r->rq(m+1, b, c, d));
}
} *root;
root = new node(0, n-1, 0, n-1);

```

---

## 2 Graph Theory

### 2.1 Depth First Search

Runs in  $O(V + E)$ .

---

```

// For adjacency lists
void dfs(int x, int p) {
    for (int y : adj[x]) {
        if (y != p) {
            dist[y] = dist[x]+1;
            dfs(y, x);
        }
    }
}

```

---

### 2.2 Breadth First Search

Runs in  $O(V + E)$ .

---

```

// For adjacency lists
visited[s] = 1;
dist[s] = 0;
q.push(s);
while (!q.empty()) {
    int f = q.front(); q.pop();
    for (int i : adjlist[f]) {
        if (!visited[i]) {
            q.push(i);
            visited[i] = 1;
            dist[i] = dist[f] + 1;
        }
    }
}

```

---

### 2.3 0-1 BFS

Runs in  $O(V + E)$ .

---

```
// For adjacency lists
deque<int> dq;
dist[s] = 0;
dq.push(s);
while (!dq.empty()) {
    int u = dq.front(); dq.pop();
    for (int e : adjlist[u]) {
        int v = e.first, w = e.second;
        if (dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            if (w == 0) dq.push_front(v);
            else dq.push_back(v);
        }
    }
}
```

---

### 2.4 Floyd-Warshall

Runs in  $O(N^3)$ .

---

```
// Initialise adjacency matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j) adj[i][j] = 0;
        else adj[i][j] = INF;
    }
}
// Floyd-Warshall
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
            if (adj[i][i] < 0) negCycle = true;
        }
    }
}
```

---

### 2.5 Bellman-Ford

Runs in  $O(VE)$ .

---

```
vector<int> dist(n, INF);
dist[s] = 0;
bool negCycle = false;
for (int i = 1; i <= n; i++) {
    bool update = false;
    for (Edge e : edges) {
        if (dist[e.u] < INF && dist[e.v] > dist[e.u] + e.w) {
            dist[e.v] = dist[e.u] + e.w;
            update = true;
        }
    }
    if (!update) break;
    if (update && i == n) negCycle = true;
}
```

---

## 2.6 Dijkstra's Algorithm

Runs in  $O(E \log V)$ .

```
priority_queue<pi, vector<pi>, greater<pi>> pq;
vector<int> dist(n, INF);
dist[s] = 0;
pq.push({0, s});
while (!pq.empty()) {
    pi f = pq.top(); pq.pop();
    int d = f.first, u = f.second;
    if (d != dist[u]) continue;
    for (pi x : adj[u]) {
        int v = x.first, w = x.second;
        if (dist[v] > d + w) {
            dist[v] = d + w;
            pq.push({dist[v], v});
        }
    }
}
```

## 2.7 Shortest Path Faster Algorithm

Runs in  $O(VE)$ .

```
vector<int> dist(n, INF);
vector<int> inQueue(n, 0);
queue<int> q;
dist[s] = 0;
q.push(s);
inQueue[s]++;
bool negCycle = false;
while (!q.empty()) {
    int u = q.front(); q.pop();
    inQueue[u]--;
    for (Edge e : adj[u]) {
        if (dist[e.v] > dist[u] + e.w) {
            dist[e.v] = dist[u] + e.w;
            if (inQueue[e.v] == 0) {
                q.push(e.v);
                inQueue[e.v]++;
                if (inQueue[e.v] > n) {
                    negCycle = true;
                    break;
                }
            }
        }
    }
    if (negCycle) break;
}
```

## 2.8 Prim's Algorithm

Runs in  $O(E \log V)$ .

```
priority_queue<pi, vector<pi>, greater<pi>> pq;
vector<int> dist(n, INF);
vector<bool> vis(n, false);
dist[s] = 0;
```

---

```

pq.push({0, s});
while (!pq.empty()) {
    pi f = pq.top(); pq.pop();
    int d = f.first, u = f.second;
    if (vis[u]) continue;
    vis[u] = true;
    for (pi x : adj[u]) {
        int v = x.first, w = x.second;
        if (!vis[v] && dist[v] > w) {
            dist[v] = w;
            pq.push({dist[v], v});
        }
    }
}

```

---

## 2.9 Union Find Disjoint Subset

With both path compression and union by rank, runs in  $O(\alpha(n))$  (basically constant time).

---

```

int p[MAXN];
int sz[MAXN];

int root(int x) {
    if (p[x] == -1) return x;
    return p[x] = root(p[x]);
}

void connect(int x, int y) {
    x = root(x); y = root(y);
    if (x == y) return;
    if (sz[x] < sz[y]) swap(x, y);
    p[y] = x;
    sz[x] += sz[y];
}

fill(p, p+MAXN, -1);
fill(sz, sz+MAXN, 1);

```

---

## 2.10 Kruskal's Algorithm

Runs in  $O(E \log E)$ .

---

```

sort(edges.begin(), edges.end());
for (Edge e : edges) {
    if (root(e.u) != root(e.v)) {
        connect(e.u, e.v);
        cost += e.w;
    }
}

```

---

## 2.11 Topological Sort

Runs in  $O(V + E)$ .

---

```

void dfs(int x) {
    if (v[x]) return;
    v[x] = 1;
    for (int y : adj[x]) dfs(y);

```

---

```

        topo.push_back(x);
    }
    for (int i = 0; i < n; i++) dfs(i);
    reverse(topo.begin(), topo.end());

```

---

## 2.12 Floyd's Cycle Finding Algorithm

For graphs with outdegree 1, runs in  $O(V + E)$ .

---

```

// detect cycle
int slow = s, fast = s;
do {
    slow = nxt[slow];
    fast = nxt[nxt[fast]];
} while (slow != fast);
// find start of cycle
slow = start;
while (slow != fast) {
    slow = nxt[slow];
    fast = nxt[fast];
}
// collect all nodes in cycle
vector<int> cycle;
int cur = slow;
do {
    cycle.push_back(cur);
    cur = nxt[cur];
} while (cur != slow);

```

---

## 2.13 Maximum Cardinality Bipartite Matching

### 2.13.1 Kun's Algorithm

Runs in  $O(V^3)$ .

---

```

int n, mcbms = 0, miss = 0;
vector<int> match;
vector<bool> vis;
vector<vector<int>> adj;
vector<int> left;

// dfs for augmenting path
bool dfs(int u) {
    for (int v : adj[u]) {
        if (!vis[v]) {
            vis[v] = true;
            if (match[v] == -1 || dfs(match[v])) {
                match[v] = u;
                match[u] = v;
                return true;
            }
        }
    }
    return false;
}

void mcbm() {
    match.assign(n, -1);

```

```

mcbms = 0;
// greedy initial matching
for (int u : left) {
    for (int v : adj[u]) {
        if (match[v] == -1) {
            match[v] = u;
            match[u] = v;
            mcbms++;
            break;
        }
    }
}
// dfs augmenting paths for unmatched
for (int u : left) {
    if (match[u] == -1) {
        vis.assign(n, false);
        if (dfs(u)) mcbms++;
    }
}
miss = n - mcbms;
}

```

---

### 2.13.2 Hopcroft-Karp

Runs in  $O(E \cdot \sqrt{V})$ .

```

int n, m;
vector<vector<int>> adj;
vector<int> pairU, pairV, dist;

bool bfs() {
    queue<int> q;
    for (int u = 0; u < n; u++) {
        if (pairU[u] == -1) {
            dist[u] = 0;
            q.push(u);
        } else {
            dist[u] = INF;
        }
    }
    bool found = false;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (pairV[v] == -1) {
                found = true;
            } else if (dist[pairV[v]] == INF) {
                dist[pairV[v]] = dist[u] + 1;
                q.push(pairV[v]);
            }
        }
    }
    return found;
}

bool dfs(int u) {
    for (int v : adj[u]) {
        if (pairV[v] == -1 || (dist[pairV[v]] == dist[u] + 1 && dfs(pairV[v]))) {

```

```

        pairU[u] = v;
        pairV[v] = u;
        return true;
    }
}
dist[u] = INF;
return false;
}

int hopcroftKarp() {
    pairU.assign(n, -1);
    pairV.assign(m, -1);
    dist.assign(n, 0);
    int mcbm = 0;
    while (bfs()) {
        for (int u = 0; u < n; u++) {
            if (pairU[u] == -1 && dfs(u)) {
                mcbm++;
            }
        }
    }
    return mcbm;
}

```

---

## 2.14 Articulation Points and Bridges

Using Tarjan's Algorithm, runs in  $O(V + E)$ .

```

vector<int> adj[MAXN];
vector<int> dep(MAXN, 0), low(MAXN, 0), par(MAXN, -1);
vector<int> chi(MAXN, 0), atp(MAXN, 0);
vector<bool> vis(MAXN, false);
vector<pi> bridges;

void tarjan(int u, int d) {
    vis[u] = true;
    dep[u] = low[u] = d;
    chi[u] = 0; atp[u] = 1;
    for (int v : adj[u]) {
        if (!vis[v]) {
            par[v] = u;
            chi[u]++;
            tarjan(v, d+1);
            low[u] = min(low[u], low[v]);
            if (low[v] >= dep[u]) atp[u]++;
            if (low[v] > dep[u]) bridges.push_back({u, v});
        } else if (v != par[u]) {
            low[u] = min(low[u], dep[v]);
        }
    }
}

tarjan(0, 0);
// handle root separately since it has no parents
atp[0] = chi[0];
// atp stores number of components separated upon removal
// bridges stores all bridges in the graph

```

---

## 2.15 Strongly Connected Components

Using Tarjan's Algorithm, runs in  $O(V + E)$ .

```
vector<int> adj[MAXN];
vector<int> comps[MAXN];
set<int> adjScc[MAXN];
vector<int> idxs(MAXN, -1), low(MAXN, 0);
vector<bool> onStack(MAXN, false);
stack<int> st;
int dfsidx = 0, sccidx;
int comp[MAXN];

void scc(int u) {
    idxs[u] = low[u] = dfsidx++;
    st.push(u);
    onStack[u] = true;
    for (int v : adj[u]) {
        if (idxs[v] == -1) {
            scc(v);
            low[u] = min(low[u], low[v]);
        } else if (onStack[v]) {
            low[u] = min(low[u], idxs[v]);
        }
    }
    if (low[u] == idxs[u]) {
        // u is a root of an scc
        comps[sccidx].clear();
        int w;
        do {
            w = st.top(); st.pop();
            onStack[w] = false;
            comps[sccidx].push_back(w);
            comp[w] = sccidx;
        } while (w != u);
        sccidx++;
    }
}

for (int i = 1; i <= n; i++) {
    if (idxs[i] == -1) scc(i);
}

for (int u = 1; u <= n; u++) {
    for (int v : adj[u]) {
        if (comp[u] != comp[v]) adjScc[comp[u]].insert(comp[v]);
    }
}

// dfs idxs now range from 0 to dfsidx-1
// scc idxs now range from 0 to sccidx-1
```

## 2.16 Travelling Salesman Problem

Both of these solutions run in  $O(N^2 \cdot 2^N)$  time.

### 2.16.1 Bottom Up

---

```

int full = (1 << n) - 1;
for (int mask = 0; mask <= full; mask++) {
    for (int i = 0; i < n; i++) {
        dp[mask][i] = INF;
    }
}
dp[1][0] = 0; // assume source is node 0
for (int mask = 1; mask <= full; mask++) {
    for (int i = 0; i < n; i++) {
        if (!(mask & (1 << i)) || dp[mask][i] >= INF) continue;
        for (int j = 0; j < n; j++) {
            if (mask & (1 << j)) continue;
            if (adj[i][j] >= INF) continue;
            int newMask = mask | (1 << j);
            dp[newMask][j] = min(dp[newMask][j], dp[mask][i] + adj[i][j]);
        }
    }
}
ll ans = INF;
for (int i = 0; i < n; i++) {
    if (adj[i][0] < INF) {
        // close the tour
        ans = min(ans, dp[full][i] + adj[i][0]);
    }
}

```

---

## 2.16.2 Top Down

---

```

ll adj[MAXN][MAXN], dp[1 << MAXN][MAXN];

ll tsp(int mask, int pos) {
    if (mask == (1 << n) - 1) {
        // assume node 0 is the start and end
        return adj[pos][0] < INF ? adj[pos][0] : INF;
    }
    if (dp[mask][pos] != -1) return dp[mask][pos];
    ll ans = INF;
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) == 0 && adj[pos][i] < INF) {
            ans = min(ans, adj[pos][i] + tsp(mask | (1 << i), i));
        }
    }
    return dp[mask][pos] = ans;
}

memset(dp, -1, sizeof(dp));
ll res = tsp(1, 0); // initial mask has source visited

```

---

## 2.17 Trees

### 2.17.1 Pre/Postorder Traversal

Runs in  $O(V)$ .

---

```

int prec = 0, postc = 0;
void dfs(int x, int p) {
    pre[x] = prec++;

```

---

```

        for(int y : adj[x]) {
            if (y != p) dfs(y, x);
        }
        post[x] = postc++;
    }

```

---

### 2.17.2 Subtree to Range

Runs in  $O(V)$ .

---

```

int dfs(int x, int p) {
    pre[x] = c++;
    rig[pre[x]] = pre[x];
    for (int y : adj[x]) {
        if (y != p) {
            rig[pre[x]] = max(rig[pre[x]], dfs(y, x));
        }
    }
    return rig[pre[x]];
}
// Subtree -> pre[x], rig[pre[x]]
// Node Index -> pre[x]
// Range of Children -> pre[x]+1, rig[pre[x]]

```

---

### 2.17.3 Weighted Maximum Independent Set

Runs in  $O(V)$ .

---

```

int dp[MAXN][2];

int mis(int v, bool take, int p) {
    if (dp[v][take] != -1) return dp[v][take];
    int ans = take * c[v];
    for (int u : adj[v]) {
        if (u == p) continue;
        int temp = mis(u, 0, v);
        if (!take) temp = max(temp, mis(u, 1, v));
        ans += temp;
    }
    return dp[v][take] = ans;
}
void ans(int v, bool take, int p) {
    for (int u : adj[v]) {
        if (u == p) continue;
        int temp0 = dp[u][0], temp1 = (take ? -1 : dp[u][1]);
        if (temp0 > temp1) ans(u, 0, v);
        else { a.push_back(u); ans(u, 1, v); }
    }
}

memset(dp, -1, sizeof(dp));
mis(0, 0, -1); // don't take root
mis(0, 1, -1); // take root
if (dp[0][1] > dp[0][0]) { a.push_back(0); ans(0, 1, -1); }
else ans(0, 0, -1);

```

---

#### 2.17.4 Diameter

Runs in  $O(V)$ .

---

```

pi dfs(int x, int p, int d) {
    pi b = {x, d};
    for (pi y : adj[x]) {
        if (y.first != p) {
            pi c = dfs(y.first, x, d + y.second);
            if (c.second > b.second) b = c;
        }
    }
    return b;
}
pi s = dfs(0, -1, 0);
pi e = dfs(s.first, -1, 0);
// e.second gives diameter
// For even diameter, centroid is at e.second / 2
// For odd diameter, centroid is at e.second / 2 and e.second / 2 + 1

```

---

#### 2.17.5 $2^K$ Decomposition

$O(N \log N)$  precomputation and memory,  $O(\log N)$  query.

---

```

int par(int x, int k) {
    for(int i = MAXLOGN; i >= 0; i--) {
        if (k >= (1 << i)) {
            if(x == -1) return x;
            x = p[x][i];
            k -= (1 << i);
        }
    }
    return x;
}

int p[MAXN][MAXLOGN];
memset(p, -1, sizeof(p));
dfs(0); // compute initial parent p[i][0]
for (int k = 1; k <= MAXLOGN; k++) {
    for (int i = 0; i < n; i++) {
        if(p[i][k-1] != -1) p[i][k] = p[p[i][k-1]][k-1];
    }
}

```

---

#### 2.17.6 Lowest Common Ancestor

Runs in  $O(\log N)$ .

---

```

int lca(int x, int y) {
    // make both nodes the same depth
    if (dep[x] < dep[y]) swap(x, y);
    for (int k = MAXLOGN; k >= 0; k--) {
        if (p[x][k] != -1 && dep[p[x][k]] >= dep[y]) x = p[x][k];
    }
    if (x == y) return x;
    // perform binary lifting while parents are different
    for (int k = MAXLOGN; k >= 0; k--) {
        if (p[x][k] != p[y][k]) {
            x = p[x][k];
        }
    }
}

```

---

```

        y = p[y][k];
    }
}

// find the next parent
return p[x][0];
}

```

---

### 2.17.7 Shortest Path

Runs in  $O(\log N)$ .

---

```

int distance(int x, int y) {
    return dist[x] + dist[y] - 2 * dist[lca(x, y)];
}

```

---

### 2.17.8 Heavy Light Decomposition

$O(N)$  precomputation,  $O(\log N)$  query (excluding  $O(\log N)$  from data structure).

---

```

// Requires segment tree, supports path maximum queries

vector<pi> adj[MAXN];
int par[MAXN], dep[MAXN], heavy[MAXN], head[MAXN], pos[MAXN], cpos = 0;
// heavy: heavy child, head: start of chain, pos: position in segment tree

int dfs(int x, int p) {
    int sz = 1, maxc = 0;
    par[x] = p;
    for (pi y : adj[x]) {
        if (y.s != p) {
            dep[y.s] = dep[x] + 1;
            int csz = dfs(y.s, x);
            sz += csz;
            if (csz > maxc) {
                maxc = csz;
                heavy[x] = y.s;
            }
        }
    }
    return sz;
}

void decomp(int x, int h) {
    head[x] = h; pos[x] = cpos++;
    if (heavy[x] != -1) decomp(heavy[x], h);
    for (pi y : adj[x]) {
        if (y.s == par[x]) continue;
        if (y.s != heavy[x]) decomp(y.s, y.s);
        root->update(pos[y.s], y.f);
    }
}

int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = par[head[b]]) {
        // maintain b as deeper
        if (dep[head[a]] > dep[head[b]]) swap(a, b);
        res = max(res, root->query(pos[head[b]], pos[b]));
    }
}

```

```

    }
    // a and b are now on the same chain
    if (dep[a] > dep[b]) swap(a, b);
    res = max(res, root->query(pos[a]+1, pos[b]));
    return res;
}

memset(heavy, -1, sizeof(heavy));
dep[0] = 0;
dfs(0, -1);
decomp(0, 0);

```

---

### 2.17.9 Centroid Decomposition

$O(N \log N)$  precomputation,  $O(\log N)$  updates and queries.

---

```

// Example: Xenia and Tree
// Update: Mark a node red
// Query: Find distance to closest red node to query node

int sz[MAXN], par[MAXN], lvl[MAXN];
ll dist[MAXN][MAXLOGN], best[MAXN];

// find subtree sizes
int dfsSize(int x, int p) {
    sz[x] = 1;
    for (int y : adj[x]) {
        if (lvl[y] != -1 || y == p) continue;
        sz[x] += dfsSize(y, x);
    }
    return sz[x];
}

// find centroid of each subtree
int dfsCentroid(int x, int p, int n) {
    for (int y : adj[x]) {
        if (lvl[y] != -1 || y == p) continue;
        if (sz[y] > n / 2) return dfsCentroid(y, x, n);
    }
    // current node is centroid
    return x;
}

void dfsDist(int x, int p, int level, ll d) {
    dist[x][level] = d;
    for (int y : adj[x]) {
        if (lvl[y] != -1 || y == p) continue;
        dfsDist(y, x, level, d + 1);
    }
}

void build(int x, int p, int level) {
    // find subtree sizes and centroid
    int size = dfsSize(x, -1);
    int cent = dfsCentroid(x, -1, size);
    if (p == -1) p = cent;
    par[cent] = p; // set parent of centroid to previous centroid
    lvl[cent] = level;
}

```

```

        dfsDist(cent, -1, level, 0);
        // recursively build subtrees
        for (int y : adj[cent]) {
            if (lvl[y] != -1) continue;
            build(y, cent, level + 1);
        }
    }

    void update(int x) {
        int y = x;
        int level = lvl[x];
        while (level != -1) {
            // shortest distance from y to any red node
            // update using new node x and its distance to the centroid y
            best[y] = min(best[y], dist[x][level]);
            y = par[y];
            level--;
        }
    }

    ll query(int x) {
        ll res = INF;
        int y = x;
        int level = lvl[x];
        while (level != -1) {
            // shortest distance to any red node in centroid chain
            res = min(res, best[y] + dist[x][level]);
            y = par[y];
            level--;
        }
        return res;
    }

    memset(lvl, -1, sizeof(lvl));
    fill(best, best+n, INF);
    build(0, -1, 0);
    update(0); // adding a node to be considered
    query(y); // querying with all considered nodes

```

---

### 3 Dynamic Programming

#### 3.1 Maxsum

##### 3.1.1 1D

Kadane's Algorithm. Runs in  $O(N)$ .

---

```

int ans = nums[0], cur = nums[0];
for (int i = 1; i < nums.size(); i++) {
    if (cur < 0) cur = 0;
    cur += nums[i];
    ans = max(ans, cur);
}

```

---

#### 3.2 Longest Increasing Subsequence

##### 3.2.1 $N^2$ DP

---

```

int ans = 0, dp[n];
memset(dp, 0, sizeof(dp));
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i]) {
            dp[i] = max(dp[i], dp[j]);
        }
    }
    dp[i]++;
    ans = max(ans, dp[i]);
}

```

---

### 3.2.2 Data Structure Speedup

---

```

// Data structure should support point max updates and range max queries
// Discretise values first
for (int i = 0; i < n; i++) {
    t = query(a[i] - 1) + 1;
    update(a[i], t);
    ans = max(ans, t);
}

```

---

### 3.2.3 $N \log N$ DP

---

```

int len = 0, dp[n];
memset(dp, 0, sizeof(dp));
for (int x : a) {
    int pos = lower_bound(dp, dp + len, x) - dp;
    dp[pos] = x;
    if (pos == len) len++;
}
cout << len;

```

---

## 3.3 Coin Combinations

Runs in  $O(N \cdot V)$ .

---

```

int ways[v+1];
memset(ways, 0, sizeof(ways));
ways[0] = 1;
for (int i = 0; i < n; i++) {
    int c = coins[i];
    for (int sum = c; sum <= v; sum++) {
        ways[sum] = (ways[sum] + ways[sum - c]) % MOD;
    }
}
cout << ways[v];

```

---

## 3.4 Coin Change

Runs in  $O(N \cdot V)$ .

---

```

const int INF = 1e9;
vector<int> dp(v + 1, INF);
dp[0] = 0;

```

---

---

```

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

```

---

## 3.5 Knapsack

### 3.5.1 0-1

Runs in  $O(N \cdot S)$ .

---

```

for (int i = 0; i < n; i++) {
    for (int j = s; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
cout << dp[s];

```

---

## 3.6 Digit DP

Runs in  $O(D)$ .

---

```

// Example - Numbers
// Compute the number of palindrome free numbers in a given range

vector<int> num;
ll dp[20][11][11][2][2]; // idx, last1, last2, tight, hasStarted

ll derp(int pos, int last1, int last2, bool tight, bool hasStarted) {
    if(pos == num.size()) return 1; // successfully populated whole number

    if(dp[pos][last1][last2][tight][hasStarted] != -1) {
        // state already visited
        return dp[pos][last1][last2][tight][hasStarted];
    }

    ll res = 0;
    int limit = tight ? num[pos] : 9; // do we need to keep to the range
    for(int d = 0; d <= limit; d++) { // try all next digits
        bool newHasStarted = hasStarted || (d != 0);
        bool newTight = tight && (d == limit);
        // skip palindromes only if the number has started
        if(newHasStarted) {
            if(d == last1) continue; // palindrome length 2
            if(d == last2) continue; // palindrome length 3
        }
        int newLast1 = newHasStarted ? d : 10;
        int newLast2 = hasStarted ? last1 : 10;
        res += derp(pos+1, newLast1, newLast2, newTight, newHasStarted);
    }
    return dp[pos][last1][last2][tight][hasStarted] = res;
}

// convert number to digits

```

---

```

void dcmp(ll x){
    num.clear();
    if(x == 0) num.push_back(0);
    while(x > 0) {
        num.push_back(x % 10);
        x /= 10;
    }
    reverse(num.begin(), num.end());
}

// Total Valid with value <= x
// Use PIE to get number within [a, b]
ll solve(ll x){
    dcmp(x);
    memset(dp, -1, sizeof(dp));
    return derp(0, 10, 10, true, false);
}

// To compute the kth string satisfying
// Either binary search or build character by character:
int count = 0;
vector<int> ans;
for (int i = 0; i < n; i++) {
    int x = 0;
    for (int j = 1; j < 10; j++) { // adjust to size of alphabet
        if (count + dp(i, j) > k) {
            break;
        }
        x = j;
    }
    count += dp(i, x); // position i is bounded by x
    ans.push_back(x);
}

```

---

### 3.7 Convex Hull Trick

Supports insertion and queries in amortised  $O(1)$ .

```

// Example: Commando
// Partition into contiguous groups with maximal effectiveness
// Group effectiveness is a quadratic function of their sum
// dp(x) = max(dp(i)+f(p(x)-p(i)))
//      = max(dp(i)+a(p(x)-p(i))^2+b(p(x)-p(i))+c)
//      = max(dp(i)+ap(x)^2-2ap(x)p(i)+ap(i)^2+bp(x)-bp(i))+c
//      = max(dp(i)+ap(i)^2-bp(i)-2ap(x)p(i)+ap(x)^2+bp(x)+c
//      = max([dp(i)+ap(i)^2-bp(i)][-2ap(i)][p(x)][+ap(x)^2+bp(x)+c])
//      = max(c(i)+m(i)p(x))+v(x)
// c(i) = dp(i)+ap(i)^2-bp(i)
// m(i) = -2ap(i)
// v(x) = ap(x)^2+bp(x)+c
// Lines have increasing gradients, queries have increasing x

deque<pi> hull;

ll func(pi line, ll x){
    return line.first*x+line.second;
}

```

```

ld intersection(ll m1, ll c1, ll m2, ll c2) {
    return (ld) (c2-c1) / (m1-m2);
}

ld intersect(pi x, pi y) {
    return intersection(x.first, x.second, y.first, y.second);
}

// query maximum y at x
ll query(ll x) {
    while (hull.size() > 1) {
        if (func(hull[0], x) < func(hull[1], x)) {
            hull.pop_front();
        } else break;
    }
    return func(hull[0], x);
}

// insert new line
void insert(ll m, ll c) {
    pi line = pi(m, c);
    while (hull.size() > 1) {
        ll s = hull.size();
        if (intersect(hull[s-1], line) <= intersect(hull[s-2], line)) {
            hull.pop_back();
        } else break;
    }
    hull.push_back(line);
}

insert(0, 0); // dp[0]
for(int i = 1; i <= n; i++){
    dp[i] = query(ps[i])+a*ps[i]*ps[i]+b*ps[i]+c; // max(mx + c) + v
    insert(-2*a*ps[i], dp[i]+a*ps[i]*ps[i]-b*ps[i]); // insert new (m, c)
}

```

---

### 3.8 Li Chao Tree

Supports insertion and queries in  $O(\log X)$  time where  $X$  is the domain.

---

```

struct Node {
    ll m, c;
    Node *lc, *rc;
    Node(ll _m = 0, ll _c = -INF) {
        m = _m; c = _c;
        lc = nullptr; rc = nullptr;
    }
    ll value(ll x) {
        return m * x + c;
    }
    void insert(ll nm, ll nc, ll l, ll r) {
        ll mid = (l+r)/2;
        bool leftCheck = nm * l + nc > value(l);
        bool midCheck = nm * mid + nc > value(mid);
        if (midCheck) {
            swap(m, nm);
            swap(c, nc);
        }
        if (leftCheck) {
            if (lc == nullptr) {
                lc = new Node(nm, nc);
            } else lc->insert(nm, nc, l, mid);
        }
        if (midCheck) {
            if (rc == nullptr) {
                rc = new Node(nm, nc);
            } else rc->insert(nm, nc, mid, r);
        }
    }
};

```

```

    }
    if (r-l == 1) return;
    if (leftCheck != midCheck) {
        if (!lc) lc = new Node();
        lc->insert(nm, nc, l, mid);
    } else {
        if (!rc) rc = new Node();
        rc->insert(nm, nc, mid, r);
    }
}
ll query(ll x, ll l, ll r) {
    ll res = value(x);
    if (r-l == 1) return res;
    ll mid = (l+r)/2;
    if (x < mid && lc) return max(res, lc->query(x, l, mid));
    if (x >= mid && rc) return max(res, rc->query(x, mid, r));
    return res;
}
};

// Queries are on half open [L, R)
// Example: Commando
Node *root = new Node();
root->insert(0, 0, MINX, MAXX);
for (int i = 1; i <= n; i++) {
    // query lct
    ll x = p[i];
    ll best = root->query(x, MINX, MAXX);
    dp[i] = best + a*x*x + b*x + c;
    // update lct
    ll m = -2*a*x;
    ll c = dp[i] + a*x*x - b*x;
    root->insert(m, c, MINX, MAXX);
}

```

---

### 3.9 Divide and Conquer

Reduces complexity from  $O(N^2 \cdot K)$  to  $O(N \log N \cdot K)$ .

```

// Example: Guards
// Minimise sum of costs where cost of partition is length * sum
// DP has form dp(i, j) = min dp(i-1, k-1) + C(k, j)
// i is the current layer, k to j is the new group
// Cost function satisfies quadrangle inequality:
// C(a, c) + C(b, d) <= C(a, d) + C(b, c) for a <= b <= c <= d

long long cost(int s, int e) {
    return (ps[e]-ps[s-1])*(e-s+1);
}

void dnc(int s, int e, long long x, int y, int k) {
    if(s > e) return;
    int m = (s+e)/2, best = 0;
    dp[m][k] = INF;
    for (int i = x; (i <= y && i <= m); i++) {
        ll val = dp[i][!k]+cost(i+1, m);
        if (dp[m][k] > val) {
            dp[m][k] = val;

```

```

        best = i;
    }
}
if (s < m) dnc(s, m-1, x, best, k);
if (m < e) dnc(m+1, e, best, y, k);
}

// Uses DP on the fly to save space
for (int i = 1; i <= n; i++) dp[i][0] = INF;
for (int i = 1; i <= g; i++) {
    for (int j = 1; j <= n; j++) dp[j][i%2] = INF;
    dnc(0, n, 0, n, i%2);
}

```

---

## 4 Math

### 4.1 Fast Exponentiation

Runs in  $O(\log b)$ .

```

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

```

---

### 4.2 Prime Factorisation

Runs in  $O(\sqrt{x})$ .

```

map<int, int> cnt;
while (x % 2 == 0) {
    cnt[2]++;
    x /= 2;
}
for (int i = 3; i * i <= x; i++) {
    while (x % i == 0) {
        cnt[i]++;
        x /= i;
    }
}
if (x > 1) {
    cnt[x]++;
}

```

---

### 4.3 Sieve of Eratosthenes

Runs in  $O(n \log \log n)$  with high constant.

```

bitset<MAXN> prime;
prime.set();
prime[0] = prime[1] = 0;
for (int i = 2; i < MAXN; i++) {

```

---

```

    if (prime[i]) {
        for (int j = i*i; j < MAXN; j += i) {
            prime[j] = 0;
        }
    }
}

```

---

#### 4.4 Greatest Common Divisor

Runs in  $O(\log \min(a, b))$ .

---

```

int gcd(int a, int b) {
    if (a > b) swap(a, b);
    while (a != 0) {
        b %= a;
        swap(a, b);
    }
    return b;
}

```

---

#### 4.5 Lowest Common Multiple

---

```

int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}

```

---

#### 4.6 Modular Inverse

For prime modulo.

---

```

ll modinv(ll a){
    return powmod(a, MOD-2, MOD);
}

```

---

#### 4.7 $\binom{n}{k}$

Precomputation takes  $O(MAXN)$  time, queries answered in  $O(1)$ .

---

```

ll fac[MAXN+1], modinv[MAXN+1];

ll nck(ll n, ll k) {
    if (n < k) return 0;
    ll res = fac[n];
    res = (res * modinv[k]) % MOD;
    res = (res * modinv[n-k]) % MOD;
    return res;
}

fac[0] = 1;
for(int i = 1; i <= MAXN; i++) {
    fac[i] = fac[i-1] * i % MOD;
}
modinv[MAXN] = powmod(fac[MAXN], MOD-2, MOD);
for(int i = MAXN; i > 0; i--) {
    modinv[i-1] = modinv[i] * i % MOD;
}

```

---

## 4.8 Fibonacci

Runs in  $O(\log N)$  time.

---

```
struct Mat {
    ll a, b, c, d; // 2x2 Matrix: [a b; c d]
};

Mat mul(Mat x, Mat y) {
    return {
        x.a*y.a + x.b*y.c,
        x.a*y.b + x.b*y.d,
        x.c*y.a + x.d*y.c,
        x.c*y.b + x.d*y.d
    };
}

Mat mpow(Mat base, long long exp) {
    Mat res = {1, 0, 0, 1}; // Identity Matrix
    while (exp) {
        if (exp & 1) res = mul(res, base);
        base = mul(base, base);
        exp >>= 1;
    }
    return res;
}

ll fib(long long n) {
    if (n == 0) return 0;
    Mat m = {1, 1, 1, 0}; // Fibonacci Seed Matrix
    return mpow(m, n-1).a;
}
```

---

## 5 Algorithms

### 5.1 Binary Search

Find the cuberoot of  $n$ . Runs in  $O(\log N)$ .

---

```
long long n; cin >> n;
long long mini = 0, maxi = 1e6, medi;
while (mini < maxi) {
    medi = mini+(maxi-mini)/2;
    if (medi * medi * medi >= n) maxi = medi;
    else mini = medi+1;
}
cout << mini << "\n";
```

---

### 5.2 Binary Search using Lifting

Find the cuberoot of  $n$ . Runs in  $O(\log N)$ .

---

```
long long n; cin >> n;
long long cur = 0, gap = 1e6, next;
while (gap > 0) {
    while (next = cur + gap, next * next * next < n) {
        cur = next;
    }
}
```

---

```

        gap >= 1;
}
cout << cur+1 << "\n";

```

---

### 5.3 Sliding Set

Speeds up DP from  $O(N^2)$  to  $O(N \log N)$ .

---

```

// Example - Candymountain
// Jump across with minimax candies
// Populate with initial window
for(int i = 0; i < k; i++) {
    dp[i] = candies[i];
    s.insert(candies[i]);
}
// Sliding Set
for(int i = k; i < n; i++){
    dp[i] = max(candies[i], *s.begin());
    s.erase(s.find(dp[i-k]));
    s.insert(dp[i]);
}

```

---

### 5.4 Set Merging

Reduces complexity from  $O(Q \cdot N \log N)$  to  $O(N \log^2 N)$ .

---

```

for (int i = 0; i < q; i++) {
    cin >> a >> b;
    // small to large merging
    if (s[a].size() > s[b].size()) swap(s[a], s[b]);
    for (int x : s[a]) s[b].insert(x);
    s[a].clear();
    cout << s[b].size() << "\n";
}

```

---

### 5.5 Discretisation

Runs in  $O(N \log N)$ .

---

```

vector<int> a(n);
vector<int> b = a;
sort(b.begin(), b.end());
b.erase(unique(b.begin(), b.end()), b.end());
for (int &i : a) {
    i = lower_bound(b.begin(), b.end(), i) - b.begin() + 1; // 1-indexed
}
// a now holds discretised values

```

---

### 5.6 Meet in the Middle

Reduces time complexity from  $O(2^N)$  to  $O(N \cdot 2^N)$ .

---

```

// Example: Bobek
// Count number of subsets with sum <= tgt
left = n/2; right = n-left;
vector<int> sums;
for (int i = 0; i < (1 << left); i++) {

```

```

int cur = 0;
for (int j = 0; j < left; j++) {
    if (i & (1 << j)) cur += a[j];
}
sums.push_back(cur);
}
sort(sums.begin(), sums.end());
for (int i = 0; i < (1 << right); i++) {
    int cur = 0;
    for (int j = 0; j < right; j++) {
        if (i & (1 << j)) cur += a[left + j];
    }
    ans += upper_bound(sums.begin(), sums.end(), tgt-cur) - sums.begin();
}

```

---

## 6 Miscellaneous

### 6.1 Fast I/O

Cannot use with `scanf`, `printf`.

---

```
ios_base::sync_with_stdio(false);
cin.tie(0);
```

---

### 6.2 Superfast I/O

Only for non-negative integer input.

---

```

inline ll ri () {
    ll x = 0;
    char ch = getchar_unlocked();
    while (ch < '0' || ch > '9') ch = getchar_unlocked();
    while (ch >= '0' && ch <= '9') {
        x = (x << 3) + (x << 1) + ch - '0';
        ch = getchar_unlocked();
    }
    return x;
}
```

---