



## The 2025 ICPC Vietnam National Contest



HSGS Luong Van Tuy



LVTNB.Raumanian67

Giang Truong Vu - Thai Son Huu Vu - Nhat Minh Doan



# Contents

## Contents

<b>1 Data Structures</b>	<b>2</b>
1.1 Fenwick Tree (BIT)	2
1.2 Sparse Table	2
1.3 Disjoint Set Union (DSU)	2
1.4 Lazy Segment Tree	3
1.5 Trie	4
<b>2 Graph Theory</b>	<b>5</b>
2.1 DFS & BFS	5
2.2 Dijkstra's Algorithm	5
2.3 Bellman-Ford Algorithm	5
2.4 Floyd-Warshall Algorithm	6
2.5 Kruskal's Algorithm	6
2.6 Tarjan's SCC Algorithm	6
2.7 Bridges & Articulation Points	7
2.8 Topological Sort	7
2.9 Lowest Common Ancestor (LCA)	7
<b>3 Dynamic Programming</b>	<b>8</b>
3.1 Classic DP Patterns	8
3.2 Digit DP	8
3.3 Bitmask DP	9
3.4 Tree DP	9
<b>4 Mathematics &amp; Number Theory</b>	<b>10</b>
4.1 Modular Arithmetic	10
4.2 Modular Inverse	10
4.3 Prime Numbers	10
4.4 Combinatorics	11
4.5 Matrix Exponentiation	11
<b>5 String Algorithms</b>	<b>12</b>
5.1 KMP Algorithm	12
5.2 Z-Algorithm	12
5.3 String Hashing	13
5.4 Manacher's Algorithm	13
<b>6 Miscellaneous</b>	<b>14</b>
6.1 Mo's Algorithm	14
6.2 Meet in the Middle	14



## 1 Data Structures

### 1.1 Fenwick Tree (BIT)

```
1 struct Fenwick {
2     int n;
3     vector<ll> fen;
4
5     Fenwick(int _n) : n(_n), fen(n + 1, 0) {}
6
7     void modify(int p, ll val) {
8         for (; p <= n; p += p & -p) fen[p] += val;
9     }
10
11    ll query(int p, ll res = 0) {
12        for (; p > 0; p -= p & -p) res += fen[p];
13        return res;
14    }
15};
```

### 1.2 Sparse Table

```
1 struct SparseTable {
2     static constexpr int lg = 20;
3     int n;
4     vector<vector<int>> st;
5     vector<int> log;
6
7     SparseTable(const vector<int> &arr) : n(arr.size()) {
8         log.assign(n + 1, 0);
9         for (int i = 2; i <= n; ++i) log[i] = log[i >> 1] + 1;
10
11        st.assign(n, vector<int>(lg, 0));
12        for (int i = 0; i < n; ++i) st[i][0] = arr[i];
13
14        for (int j = 1; j < lg; ++j) {
15            for (int i = 0; i + (1 << j) <= n; ++i) {
16                st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
17            }
18        }
19    }
20
21    int query(int l, int r) const {
22        int j = log[r - l + 1];
23        return min(st[l][j], st[r - (1 << j) + 1][j]);
24    }
25};
```

### 1.3 Disjoint Set Union (DSU)

```
1 struct DSU {
2     vector<int> parent, rank;
3
4     void init(int n) {
5         rank.assign(n + 1, 0);
6         parent.resize(n + 1);
7         iota(parent.begin(), parent.end(), 0);
8     }
9
10    int find(int u) { return u == parent[u] ? u : parent[u] = find(parent[u]); }
11
12    void unite(int u, int v) {
13        u = find(u), v = find(v);
14
15        if (u == v) return;
16        if (rank[u] < rank[v]) swap(u, v);
17        if (rank[u] == rank[v]) ++rank[u];
18
19        parent[v] = u;
20    }
21};
```



## 1.4 Lazy Segment Tree

```
1 struct LazySegment {
2     int n;
3     vector<ll> tree, lazy;
4
5     LazySegment(const vector<ll> &arr) : n(arr.size()) {
6         tree.assign(n << 2, 0), lazy.assign(n << 2, 0);
7         build(1, 0, n - 1, arr);
8     }
9
10    void modify(int p, ll val) { modify(1, 0, n - 1, p, val); }
11    void range_modify(int l, int r, ll val) {
12        range_modify(l, 0, n - 1, l, r, val);
13    }
14    ll query(int l, int r) { return query(1, 0, n - 1, l, r); }
15
16 private:
17    void build(int node, int l, int r, const vector<ll> &arr) {
18        if (l == r) return tree[node] = arr[l], void();
19
20        int mid = l + r >> 1;
21
22        build(node << 1, l, mid, arr);
23        build(node << 1 | 1, ~mid, r, arr);
24        tree[node] = tree[node << 1] + tree[node << 1 | 1];
25    }
26
27    void push(int node, int l, int r) {
28        ll &val = lazy[node];
29
30        if (!val) return;
31        if (l ^ r) lazy[node << 1] += val, lazy[node << 1 | 1] += val;
32
33        tree[node] += val * (r - l + 1);
34        val = 0;
35    }
36
37    void range_modify(int node, int l, int r, int ql, int qr, ll val) {
38        push(node, l, r);
39
40        if (qr < l || r < ql) return;
41        if (ql <= l && r <= qr) {
42            lazy[node] += val;
43            return push(node, l, r), void();
44        }
45
46        int mid = l + r >> 1;
47
48        range_modify(node << 1, l, mid, ql, qr, val);
49        range_modify(node << 1 | 1, ~mid, r, ql, qr, val);
50        tree[node] = tree[node << 1] + tree[node << 1 | 1];
51    }
52
53    void modify(int node, int l, int r, int p, ll val) {
54        if (l == r) return tree[node] = val, void();
55
56        int mid = l + r >> 1;
57
58        (p <= mid) ? modify(node << 1, l, mid, p, val)
59        : modify(node << 1 | 1, ~mid, r, p, val);
60        tree[node] = tree[node << 1] + tree[node << 1 | 1];
61    }
62
63    ll query(int node, int l, int r, int ql, int qr) {
64        push(node, l, r);
65
66        if (qr < l || r < ql) return 0;
67        if (ql <= l && r <= qr) return tree[node];
68
69        int mid = l + r >> 1;
70
71        return query(node << 1, l, mid, ql, qr) +
72               query(node << 1 | 1, ~mid, r, ql, qr);
73    }
74};
```



## 1.5 Trie

```
1 struct Trie {
2     struct Node {
3         array<Node*, 26> child;
4         bool isEnd;
5         Node() : child{}, isEnd(false) {}
6     };
7
8     Node* root;
9
10    Trie() : root(new Node()) {}
11
12    void insert(const string &s) {
13        Node* cur = root;
14        for (char c : s) {
15            int idx = c - 'a';
16            if (!cur->child[idx]) cur->child[idx] = new Node();
17            cur = cur->child[idx];
18        }
19        cur->isEnd = true;
20    }
21
22    bool search(const string &s) {
23        Node* cur = root;
24        for (char c : s) {
25            int idx = c - 'a';
26            if (!cur->child[idx]) return false;
27            cur = cur->child[idx];
28        }
29        return cur->isEnd;
30    }
31};
```



## 2 Graph Theory

### 2.1 DFS & BFS

```
1 vector<int> adj[N];
2 bool vis[N];
3
4 void dfs(int u) {
5     vis[u] = true;
6     for (int v : adj[u]) if (!vis[v]) dfs(v);
7 }
8
9 void bfs(int start) {
10    queue<int> q;
11    q.push(start), vis[start] = true;
12
13    while (!q.empty()) {
14        int u = q.front(); q.pop();
15        for (int v : adj[u]) if (!vis[v]) {
16            vis[v] = true, q.push(v);
17        }
18    }
19 }
```

### 2.2 Dijkstra's Algorithm

```
1 vector<pair<int,int>> adj[N];
2 ll dist[N];
3
4 void dijkstra(int start, int n) {
5     fill(dist, dist + n + 1, INF);
6     priority_queue<pair<ll,int>, vector<pair<ll,int>>, greater<pair<ll,int>>> pq;
7
8     dist[start] = 0, pq.push({0, start});
9
10    while (!pq.empty()) {
11        auto [d, u] = pq.top(); pq.pop();
12
13        if (d > dist[u]) continue;
14
15        for (auto [v, w] : adj[u]) {
16            if (dist[u] + w < dist[v]) {
17                dist[v] = dist[u] + w;
18                pq.push({dist[v], v});
19            }
20        }
21    }
22 }
23 }
```

### 2.3 Bellman-Ford Algorithm

```
1 struct Edge { int u, v; ll w; };
2 vector<Edge> edges;
3 ll dist[N];
4
5 bool bellman_ford(int start, int n) {
6     fill(dist, dist + n + 1, INF);
7     dist[start] = 0;
8
9     for (int i = 0; i < n - 1; ++i) {
10         for (auto [u, v, w] : edges) {
11             if (dist[u] != INF && dist[u] + w < dist[v]) dist[v] = dist[u] + w;
12         }
13     }
14
15     for (auto [u, v, w] : edges) {
16         if (dist[u] != INF && dist[u] + w < dist[v]) return false;
17     }
18     return true;
19 }
```



## 2.4 Floyd-Warshall Algorithm

```
1 ll dist[N][N];
2
3 void floyd_warshall(int n) {
4     for (int k = 1; k <= n; ++k) {
5         for (int i = 1; i <= n; ++i) {
6             for (int j = 1; j <= n; ++j) {
7                 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
8             }
9         }
10    }
11 }
```

## 2.5 Kruskal's Algorithm

```
1 struct Edge {
2     int u, v; ll w;
3     bool operator<(const Edge &o) const { return w < o.w; }
4 };
5
6 vector<Edge> edges;
7 DSU dsu;
8
9 ll kruskal(int n) {
10     sort(edges.begin(), edges.end());
11     dsu.init(n);
12
13     ll cost = 0;
14     int cnt = 0;
15
16     for (auto [u, v, w] : edges) {
17         if (dsu.find(u) != dsu.find(v)) {
18             dsu.unite(u, v);
19             cost += w;
20             if (++cnt == n - 1) break;
21         }
22     }
23     return cost;
24 }
```

## 2.6 Tarjan's SCC Algorithm

```
1 vector<int> adj[N];
2 int low[N], disc[N], timer = 0;
3 bool onStack[N];
4 stack<int> st;
5 vector<vector<int>> sccs;
6
7 void tarjan(int u) {
8     disc[u] = low[u] = ++timer;
9     st.push(u), onStack[u] = true;
10
11     for (int v : adj[u]) {
12         if (!disc[v]) {
13             tarjan(v);
14             low[u] = min(low[u], low[v]);
15         } else if (onStack[v]) low[u] = min(low[u], disc[v]);
16     }
17
18     if (low[u] == disc[u]) {
19         vector<int> scc;
20         while (true) {
21             int v = st.top(); st.pop();
22             onStack[v] = false;
23             scc.push_back(v);
24             if (v == u) break;
25         }
26         sccs.push_back(scc);
27     }
28 }
```



## 2.7 Bridges & Articulation Points

```
1 vector<int> adj[N];
2 int low[N], disc[N], timer = 0;
3 bool vis[N], is_art[N];
4 vector<pair<int,int>> bridges;
5
6 void dfs_bridge(int u, int p = -1) {
7     vis[u] = true;
8     disc[u] = low[u] = ++timer;
9     int children = 0;
10
11    for (int v : adj[u]) {
12        if (v == p) continue;
13        if (vis[v]) {
14            low[u] = min(low[u], disc[v]);
15        } else {
16            dfs_bridge(v, u);
17            low[u] = min(low[u], low[v]);
18
19            if (low[v] > disc[u]) bridges.push_back({u, v});
20            if (low[v] >= disc[u] && p != -1) is_art[u] = true;
21
22            ++children;
23        }
24    }
25
26    if (p == -1 && children > 1) is_art[u] = true;
27 }
```

## 2.8 Topological Sort

```
1 vector<int> adj[N];
2 bool vis[N];
3 vector<int> topo;
4
5 void dfs_topo(int u) {
6     vis[u] = true;
7     for (int v : adj[u]) if (!vis[v]) dfs_topo(v);
8     topo.push_back(u);
9 }
10
11 void topological_sort(int n) {
12     for (int i = 1; i <= n; ++i) if (!vis[i]) dfs_topo(i);
13     reverse(topo.begin(), topo.end());
14 }
```

## 2.9 Lowest Common Ancestor (LCA)

```
1 constexpr int lg = 19;
2 vector<int> adj[N];
3 int depth[N], par[N][lg];
4
5 void dfs_lca(int u, int p) {
6     par[u][0] = p;
7     for (int j = 1; j < lg; ++j) par[u][j] = par[par[u][j - 1]][j - 1];
8     for (int v : adj[u]) if (v != p) {
9         depth[v] = depth[u] + 1;
10        dfs_lca(v, u);
11    }
12 }
13
14 int lca(int u, int v) {
15     if (depth[u] < depth[v]) swap(u, v);
16     for (int j = lg - 1; j >= 0; --j) {
17         if (depth[par[u][j]] >= depth[v]) u = par[u][j];
18     }
19     for (int j = lg - 1; j >= 0; --j) {
20         if (par[u][j] != par[v][j]) u = par[u][j], v = par[v][j];
21     }
22     return u == v ? u : par[u][0];
23 }
```



### 3 Dynamic Programming

#### 3.1 Classic DP Patterns

```
1 // 0/1 Knapsack
2 ll dp[N][N];
3
4 ll knapsack(vector<int> &wt, vector<int> &val, int W) {
5     int n = wt.size();
6     memset(dp, 0, sizeof dp);
7
8     for (int i = 1; i <= n; ++i) {
9         for (int w = 0; w <= W; ++w) {
10            dp[i][w] = dp[i - 1][w];
11            if (wt[i - 1] <= w) {
12                dp[i][w] = max(dp[i][w], val[i - 1] + dp[i - 1][w - wt[i - 1]]);
13            }
14        }
15    }
16    return dp[n][W];
17 }
18
19 // Longest Increasing Subsequence (O(n log n))
20 int lis(vector<int> &arr) {
21     vector<int> dp;
22     for (int x : arr) {
23         auto it = lower_bound(dp.begin(), dp.end(), x);
24         if (it == dp.end()) dp.push_back(x);
25         else *it = x;
26     }
27     return dp.size();
28 }
29
30 // Longest Common Subsequence
31 int lcs(string &s1, string &s2) {
32     int n = s1.size(), m = s2.size();
33     vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
34
35     for (int i = 1; i <= n; ++i) {
36         for (int j = 1; j <= m; ++j) {
37             dp[i][j] = (s1[i - 1] == s2[j - 1])
38                         ? dp[i - 1][j - 1] + 1
39                         : max(dp[i - 1][j], dp[i][j - 1]);
40         }
41     }
42     return dp[n][m];
43 }
```

#### 3.2 Digit DP

```
1 int K;
2 ll dp[20][200][2];
3
4 ll digit_dp(const string &num, int pos, int sum, int tight) {
5     if (pos == num.size()) return sum == K;
6     if (dp[pos][sum][tight] != -1) return dp[pos][sum][tight];
7
8     int limit = tight ? (num[pos] - '0') : 9;
9     ll res = 0;
10
11    for (int d = 0; d <= limit; ++d) {
12        res += digit_dp(num, pos + 1, sum + d, tight && (d == limit));
13    }
14
15    return dp[pos][sum][tight] = res;
16 }
```



### 3.3 Bitmask DP

```
1 int n, dist[20][20];
2 ll dp[1 << 20][20];
3
4 ll tsp(int mask, int pos) {
5     if (mask == (1 << n) - 1) return dist[pos][0];
6     if (dp[mask][pos] != -1) return dp[mask][pos];
7
8     ll ans = INF;
9     for (int i = 0; i < n; ++i) if (!(mask & (1 << i))) {
10         ans = min(ans, dist[pos][i] + tsp(mask | (1 << i), i));
11     }
12     return dp[mask][pos] = ans;
13 }
```

### 3.4 Tree DP

```
1 vector<int> adj[N];
2 ll dp[N][2];
3
4 void dfs_tree_dp(int u, int p) {
5     dp[u][0] = dp[u][1] = 0;
6     for (int v : adj[u]) if (v != p) {
7         dfs_tree_dp(v, u);
8         dp[u][0] += max(dp[v][0], dp[v][1]);
9         dp[u][1] += dp[v][0];
10    }
11    dp[u][1] += 1;
12 }
```



## 4 Mathematics & Number Theory

### 4.1 Modular Arithmetic

```
1 constexpr ll MOD = 1e9 + 7;
2
3 ll mod_add(ll a, ll b) { return ((a % MOD) + (b % MOD)) % MOD; }
4 ll mod_mul(ll a, ll b) { return ((a % MOD) * (b % MOD)) % MOD; }
5
6 ll mod_pow(ll base, ll exp) {
7     ll res = 1;
8     while (exp) {
9         if (exp & 1) res = mod_mul(res, base);
10        base = mod_mul(base, base);
11        exp >>= 1;
12    }
13    return res;
14 }
15
16 ll mod_inv(ll a) { return mod_pow(a, MOD - 2); }
```

### 4.2 Modular Inverse

```
1 ll extgcd(ll a, ll b, ll &x, ll &y) {
2     if (!b) return x = (a >= 0 ? 1 : -1), y = 0, llabs(a);
3     ll g = extgcd(b, a % b, y, x);
4     return y -= a / b * x, g;
5 }
6
7 ll modinv(ll a, ll m) {
8     a = ((a % m) + m) % m;
9     ll x, y, g = extgcd(a, m, x, y);
10    return g != 1 ? -1 : ((x % m) + m) % m;
11 }
```

### 4.3 Prime Numbers

```
1 vector<bool> is_prime;
2 vector<int> primes;
3
4 void sieve(int n) {
5     is_prime.assign(n + 1, true);
6     is_prime[0] = is_prime[1] = false;
7
8     for (int i = 2; i <= n; ++i) if (is_prime[i]) {
9         primes.push_back(i);
10        for (ll j = (ll)i * i; j <= n; j += i) is_prime[j] = false;
11    }
12 }
13
14 vector<pair<int,int>> factorize(int n) {
15     vector<pair<int,int>> factors;
16     for (int p : primes) {
17         if (p * p > n) break;
18         if (n % p == 0) {
19             int cnt = 0;
20             while (n % p == 0) n /= p, ++cnt;
21             factors.push_back({p, cnt});
22         }
23     }
24     if (n > 1) factors.push_back({n, 1});
25     return factors;
26 }
```



## 4.4 Combinatorics

```
1 ll fact[N], inv_fact[N];
2
3 void precompute(int n) {
4     fact[0] = 1;
5     for (int i = 1; i <= n; ++i) fact[i] = mod_mul(fact[i - 1], i);
6
7     inv_fact[n] = mod_inv(fact[n]);
8     for (int i = n - 1; i >= 0; --i) {
9         inv_fact[i] = mod_mul(inv_fact[i + 1], i + 1);
10    }
11 }
12
13 ll nCr(int n, int r) {
14     if (r < 0 || r > n) return 0;
15     return mod_mul(fact[n], mod_mul(inv_fact[r], inv_fact[n - r]));
16 }
17
18 ll catalan(int n) { return mod_mul(nCr(2 * n, n), mod_inv(n + 1)); }
```

## 4.5 Matrix Exponentiation

```
1 struct Matrix {
2     vector<vector<ll>> mat;
3     int n;
4
5     Matrix(int _n) : n(_n), mat(n, vector<ll>(n, 0)) {}
6
7     Matrix operator*(const Matrix &o) const {
8         Matrix res(n);
9         for (int i = 0; i < n; ++i) {
10             for (int j = 0; j < n; ++j) {
11                 for (int k = 0; k < n; ++k) {
12                     res.mat[i][j] = mod_add(res.mat[i][j],
13                                         mod_mul(mat[i][k], o.mat[k][j]));
14                 }
15             }
16         }
17         return res;
18     }
19 };
20
21 Matrix mat_pow(Matrix base, ll exp) {
22     Matrix res(base.n());
23     for (int i = 0; i < base.n(); ++i) res.mat[i][i] = 1;
24
25     while (exp) {
26         if (exp & 1) res = res * base;
27         base = base * base;
28         exp >>= 1;
29     }
30     return res;
31 }
32
33 ll fib(ll n) {
34     if (n <= 1) return n;
35     Matrix base(2);
36     base.mat = {{1, 1}, {1, 0}};
37     return mat_pow(base, n - 1).mat[0][0];
38 }
```



## 5 String Algorithms

### 5.1 KMP Algorithm

```
1 vector<int> kmp(const string &s) {
2     vector<int> pi(s.size(), 0);
3
4     for (int i = 1, j = 0; i < s.size(); ++i) {
5         while (j && s[i] != s[j]) j = pi[j - 1];
6         if (s[i] == s[j]) ++j;
7         pi[i] = j;
8     }
9
10    return pi;
11}
12
13 vector<int> kmp_search(const string &text, const string &pattern) {
14    vector<int> pi = kmp(pattern);
15    vector<int> matches;
16
17    for (int i = 0, j = 0; i < text.size(); ++i) {
18        while (j && text[i] != pattern[j]) j = pi[j - 1];
19        if (text[i] == pattern[j]) ++j;
20        if (j == pattern.size()) {
21            matches.push_back(i - j + 1);
22            j = pi[j - 1];
23        }
24    }
25    return matches;
26}
```

### 5.2 Z-Algorithm

```
1 vector<int> z_algorithm(const string &s) {
2     int n = s.size();
3     vector<int> z(n, 0);
4     int l = 0, r = 0;
5
6     for (int i = 1; i < n; ++i) {
7         if (i <= r) z[i] = min(r - i + 1, z[i - 1]);
8         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
9         if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
10    }
11    return z;
12}
13
14 vector<int> z_search(const string &text, const string &pattern) {
15    string combined = pattern + "$" + text;
16    vector<int> z = z_algorithm(combined);
17    vector<int> matches;
18    int m = pattern.size();
19
20    for (int i = m + 1; i < combined.size(); ++i) {
21        if (z[i] == m) matches.push_back(i - m - 1);
22    }
23    return matches;
24}
```



### 5.3 String Hashing

```
1 struct Rabinkarp {
2     static constexpr int mod = 1e9 + 7, base = 256;
3
4     int n;
5     vector<ll> hash, power;
6
7     Rabinkarp(const string &s) : n(s.size()), hash(n + 1, 0),
8                                   power(n + 1, 1) {
9         for (int i = 0; i < n; ++i) {
10             power[i + 1] = power[i] * base % mod;
11             hash[i + 1] = (hash[i] * base + (unsigned char)s[i]) % mod;
12         }
13     }
14
15     ll get(int l, int r) const {
16         return ((hash[r + 1] - hash[l] * power[r - l + 1]) % mod + mod) % mod;
17     }
18 }
```

### 5.4 Manacher's Algorithm

```
1 string preprocess(const string &s) {
2     string t = "@";
3     for (char c : s) t += "#", t += c;
4     return t += "#$";
5 }
6
7 vector<int> manacher(const string &s) {
8     string t = preprocess(s);
9     int n = t.size();
10    vector<int> p(n, 0);
11    int c = 0, r = 0;
12
13    for (int i = 1; i < n - 1; ++i) {
14        int mirror = 2 * c - i;
15        if (i < r) p[i] = min(r - i, p[mirror]);
16
17        while (t[i + p[i] + 1] == t[i - p[i] - 1]) ++p[i];
18
19        if (i + p[i] > r) c = i, r = i + p[i];
20    }
21    return p;
22 }
```



## 6 Miscellaneous

### 6.1 Mo's Algorithm

```
1 constexpr int BLOCK = 320;
2
3 struct Query {
4     int l, r, idx;
5     bool operator<(const Query &o) const {
6         int bl = l / BLOCK, br = o.l / BLOCK;
7         if (bl != br) return bl < br;
8         return (bl & 1) ? (r < o.r) : (r > o.r);
9     }
10 };
11
12 int cur_ans = 0;
13
14 void add(int pos) { /* update cur_ans */ }
15 void remove(int pos) { /* update cur_ans */ }
16
17 void mo_algorithm(vector<Query> &queries) {
18     sort(queries.begin(), queries.end());
19     vector<int> ans(queries.size());
20
21     int l = 0, r = -1;
22     for (auto q : queries) {
23         while (l > q.l) add(--l);
24         while (r < q.r) add(++r);
25         while (l < q.l) remove(l++);
26         while (r > q.r) remove(r--);
27         ans[q.idx] = cur_ans;
28     }
29 }
```

### 6.2 Meet in the Middle

```
1 bool subset_sum_mitm(const vector<int> &arr, int target) {
2     int n = arr.size(), mid = n >> 1;
3
4     unordered_set<int> first;
5     for (int mask = 0; mask < (1 << mid); ++mask) {
6         int sum = 0;
7         for (int i = 0; i < mid; ++i) {
8             if (mask & (1 << i)) sum += arr[i];
9         }
10        first.insert(sum);
11    }
12
13    for (int mask = 0; mask < (1 << (n - mid)); ++mask) {
14        int sum = 0;
15        for (int i = 0; i < n - mid; ++i) {
16            if (mask & (1 << i)) sum += arr[mid + i];
17        }
18        if (first.count(target - sum)) return true;
19    }
20    return false;
21 }
```