ACM-ICPC Self-Study Plan with C++ Code

Overview

This 16-week comprehensive study plan covers all essential topics for ACM-ICPC preparation with complete C++ implementations.

Week 1-2: Foundation & Basic Data Structures

Topics Covered:

- Time & Space Complexity
- Arrays, Vectors, Strings
- Basic I/O optimization
- STL containers

Key Problems & Solutions:

1. Fast I/O Template

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

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;
    vector<int> arr(n);

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

    // Your solution here

    return 0;
}
```

2. Two Pointers Technique

```cpp
```

```cpp
// Find pair with sum equal to target
vector<int> twoSum(vector<int>& nums, int target) {
  int left = 0, right = nums.size() - 1;

  while(left < right) {
    int sum = nums[left] + nums[right];
    if(sum == target) {
      return {left, right};
    }
    else if(sum < target) {
      left++;
    }
    else {
      right--;
    }
  }
  return {-1, -1};
}
```

3. Sliding Window Maximum

```cpp
```

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    deque<int> dq;
    vector<int> result;

    for(int i = 0; i < nums.size(); i++) {
        // Remove elements outside window
        while(!dq.empty() && dq.front() <= i - k) {
            dq.pop_front();
        }

        // Remove smaller elements
        while(!dq.empty() && nums[dq.back()] <= nums[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        if(i >= k - 1) {
            result.push_back(nums[dq.front()]);
        }
    }

    return result;
}
```

Week 3-4: Advanced Data Structures

Topics Covered:

- Stack, Queue, Priority Queue
- Disjoint Set Union (DSU)
- Segment Trees
- Binary Indexed Trees (BIT/Fenwick Tree)

1. Disjoint Set Union with Path Compression

```cpp
```

```cpp
class DSU {
private:
    vector<int> parent, rank;

public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for(int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if(parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unite(int x, int y) {
        int px = find(x), py = find(y);
        if(px == py) return;

        if(rank[px] < rank[py]) {
            parent[px] = py;
        }
        else if(rank[px] > rank[py]) {
            parent[py] = px;
        }
        else {
            parent[py] = px;
            rank[px]++;
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }
};
```

2. Segment Tree (Range Sum Queries)

cpp

```cpp
class SegmentTree {
private:
    vector<long long> tree;
    int n;

    void build(vector<int>& arr, int node, int start, int end) {
        if(start == end) {
            tree[node] = arr[start];
        }
        else {
            int mid = (start + end) / 2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    void updateHelper(int node, int start, int end, int idx, int val) {
        if(start == end) {
            tree[node] = val;
        }
        else {
            int mid = (start + end) / 2;
            if(idx <= mid) {
                updateHelper(2*node, start, mid, idx, val);
            }
            else {
                updateHelper(2*node+1, mid+1, end, idx, val);
            }
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    long long queryHelper(int node, int start, int end, int l, int r) {
        if(r < start || end < l) {
            return 0;
        }
        if(l <= start && end <= r) {
            return tree[node];
        }
        int mid = (start + end) / 2;
        return queryHelper(2*node, start, mid, l, r) +
               queryHelper(2*node+1, mid+1, end, l, r);
    }

public:
```

```cpp
    SegmentTree(vector<int>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(arr, 1, 0, n-1);
    }

    void update(int idx, int val) {
        updateHelper(1, 0, n-1, idx, val);
    }

    long long query(int l, int r) {
        return queryHelper(1, 0, n-1, l, r);
    }
};
```

3. Binary Indexed Tree (Fenwick Tree)

```cpp
```

```
class BIT {
private:
    vector<int> tree;
    int n;

public:
    BIT(int size) {
        n = size;
        tree.assign(n + 1, 0);
    }

    void update(int idx, int val) {
        for(++idx; idx <= n; idx += idx & -idx) {
            tree[idx] += val;
        }
    }

    int query(int idx) {
        int sum = 0;
        for(++idx; idx > 0; idx -= idx & -idx) {
            sum += tree[idx];
        }
        return sum;
    }

    int rangeQuery(int l, int r) {
        return query(r) - query(l - 1);
    }
};
```

Week 5-6: Graph Algorithms - Traversal & Shortest Paths

Topics Covered:

- DFS, BFS
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm

1. DFS Implementation

```cpp
```

```cpp
vector<vector<int>> adj;
vector<bool> visited;

void dfs(int node) {
    visited[node] = true;

    for(int neighbor : adj[node]) {
        if(!visited[neighbor]) {
            dfs(neighbor);
        }
    }
}

// Usage
int main() {
    int n, m;
    cin >> n >> m;

    adj.resize(n);
    visited.assign(n, false);

    for(int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(0);
    return 0;
}
```

2. BFS Implementation

```
cpp
```

```cpp
vector<int> bfs(int start, int n) {
    vector<int> distance(n, -1);
    queue<int> q;

    distance[start] = 0;
    q.push(start);

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

        for(int neighbor : adj[node]) {
            if(distance[neighbor] == -1) {
                distance[neighbor] = distance[node] + 1;
                q.push(neighbor);
            }
        }
    }

    return distance;
}
```

3. Dijkstra's Algorithm

```cpp
```

```cpp
vector<int> dijkstra(int start, int n) {
    vector<int> dist(n, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[start] = 0;
    pq.push({0, start});

    while(!pq.empty()) {
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if(d > dist[u]) continue;

        for(auto& edge : adj[u]) {
            int v = edge.first;
            int w = edge.second;

            if(dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

4. Bellman-Ford Algorithm

```cpp
```

```cpp
bool bellmanFord(int start, int n, vector<tuple<int, int, int>>& edges) {
    vector<int> dist(n, INT_MAX);
    dist[start] = 0;

    // Relax edges n-1 times
    for(int i = 0; i < n - 1; i++) {
        for(auto& edge : edges) {
            int u, v, w;
            tie(u, v, w) = edge;

            if(dist[u] != INT_MAX && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Check for negative cycles
    for(auto& edge : edges) {
        int u, v, w;
        tie(u, v, w) = edge;

        if(dist[u] != INT_MAX && dist[u] + w < dist[v]) {
            return false; // Negative cycle found
        }
    }

    return true;
}
```

Week 7-8: Graph Algorithms - Advanced

Topics Covered:

- Minimum Spanning Tree (Kruskal's & Prim's)
- Topological Sort
- Strongly Connected Components
- Bridges and Articulation Points

1. Kruskal's MST Algorithm

```cpp
```

```cpp
struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

int kruskalMST(int n, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    DSU dsu(n);

    int mstWeight = 0;
    int edgesUsed = 0;

    for(Edge& e : edges) {
        if(!dsu.connected(e.u, e.v)) {
            dsu.unite(e.u, e.v);
            mstWeight += e.weight;
            edgesUsed++;

            if(edgesUsed == n - 1) break;
        }
    }

    return mstWeight;
}
```

2. Topological Sort (DFS-based)

```cpp
```

```
vector<int> topologicalSort(int n) {
  vector<int> result;
  vector<bool> visited(n, false);

  function<void(int)> dfs = [&](int node) {
    visited[node] = true;

    for(int neighbor : adj[node]) {
      if(!visited[neighbor]) {
        dfs(neighbor);
      }
    }

    result.push_back(node);
  };

  for(int i = 0; i < n; i++) {
    if(!visited[i]) {
      dfs(i);
    }
  }

  reverse(result.begin(), result.end());
  return result;
}
```

3. Tarjan's Algorithm for Strongly Connected Components

```cpp
```

```cpp
class TarjanSCC {
private:
    vector<vector<int>> adj;
    vector<int> disc, low, stackMember;
    stack<int> st;
    vector<vector<int>> sccs;
    int timer;

    void SCCUtil(int u) {
        disc[u] = low[u] = ++timer;
        st.push(u);
        stackMember[u] = true;

        for(int v : adj[u]) {
            if(disc[v] == -1) {
                SCCUtil(v);
                low[u] = min(low[u], low[v]);
            }
            else if(stackMember[v]) {
                low[u] = min(low[u], disc[v]);
            }
        }

        if(low[u] == disc[u]) {
            vector<int> component;
            int w;
            do {
                w = st.top();
                st.pop();
                stackMember[w] = false;
                component.push_back(w);
            } while(w != u);

            sccs.push_back(component);
        }
    }

public:
    TarjanSCC(int n) {
        adj.resize(n);
        disc.assign(n, -1);
        low.assign(n, -1);
        stackMember.assign(n, false);
        timer = 0;
    }
```

```cpp
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    vector<vector<int>> findSCCs() {
        for(int i = 0; i < adj.size(); i++) {
            if(disc[i] == -1) {
                SCCUtil(i);
            }
        }
        return sccs;
    }
};
```

Week 9-10: Dynamic Programming

Topics Covered:

- Basic DP concepts
- Classic DP problems
- Optimization techniques
- State space reduction

1. Longest Common Subsequence

```cpp
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.length(), n = text2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for(int i = 1; i <= m; i++) {
        for(int j = 1; j <= n; j++) {
            if(text1[i-1] == text2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[m][n];
}
```

2. 0/1 Knapsack Problem

```cpp
int knapsack(vector<int>& weights, vector<int>& values, int W) {
  int n = weights.size();
  vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

  for(int i = 1; i <= n; i++) {
    for(int w = 1; w <= W; w++) {
      if(weights[i-1] <= w) {
        dp[i][w] = max(dp[i-1][w],
               dp[i-1][w - weights[i-1]] + values[i-1]);
      }
      else {
        dp[i][w] = dp[i-1][w];
      }
    }
  }

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

3. Edit Distance

```cpp
```

```cpp
int editDistance(string word1, string word2) {
  int m = word1.length(), n = word2.length();
  vector<vector<int>> dp(m + 1, vector<int>(n + 1));

  for(int i = 0; i <= m; i++) dp[i][0] = i;
  for(int j = 0; j <= n; j++) dp[0][j] = j;

  for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= n; j++) {
      if(word1[i-1] == word2[j-1]) {
        dp[i][j] = dp[i-1][j-1];
      }
      else {
        dp[i][j] = 1 + min({dp[i-1][j],   // Delete
                  dp[i][j-1],    // Insert
                  dp[i-1][j-1]}); // Replace
      }
    }
  }

  return dp[m][n];
}
```

4. Matrix Chain Multiplication

```
cpp
```

```
int matrixChainMultiplication(vector<int>& dimensions) {
    int n = dimensions.size() - 1;
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for(int len = 2; len <= n; len++) {
        for(int i = 0; i < n - len + 1; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;

            for(int k = i; k < j; k++) {
                int cost = dp[i][k] + dp[k+1][j] +
                          dimensions[i] * dimensions[k+1] * dimensions[j+1];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    return dp[0][n-1];
}
```

Week 11-12: Number Theory & Mathematics

Topics Covered:

- GCD, LCM, Extended Euclidean Algorithm
- Modular Arithmetic
- Prime Numbers, Sieve of Eratosthenes
- Fast Exponentiation

1. Extended Euclidean Algorithm

```cpp
```

```cpp
int extgcd(int a, int b, int& x, int& y) {
    if(b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    int x1, y1;
    int gcd = extgcd(b, a % b, x1, y1);

    x = y1;
    y = x1 - (a / b) * y1;

    return gcd;
}

int modInverse(int a, int m) {
    int x, y;
    int gcd = extgcd(a, m, x, y);

    if(gcd != 1) return -1; // Inverse doesn't exist

    return (x % m + m) % m;
}
```

2. Sieve of Eratosthenes

```cpp
```

```
vector<bool> sieve(int n) {
    vector<bool> isPrime(n + 1, true);
    isPrime[0] = isPrime[1] = false;

    for(int i = 2; i * i <= n; i++) {
        if(isPrime[i]) {
            for(int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    return isPrime;
}

vector<int> getPrimes(int n) {
    vector<bool> isPrime = sieve(n);
    vector<int> primes;

    for(int i = 2; i <= n; i++) {
        if(isPrime[i]) {
            primes.push_back(i);
        }
    }

    return primes;
}
```

3. Fast Exponentiation

```cpp
```

```cpp
long long fastPow(long long base, long long exp, long long mod) {
    long long result = 1;
    base %= mod;

    while(exp > 0) {
        if(exp & 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp >>= 1;
    }

    return result;
}
```

4. Chinese Remainder Theorem

```cpp
long long chineseRemainder(vector<long long>& remainders, vector<long long>& moduli) {
    long long prod = 1;
    for(long long m : moduli) prod *= m;

    long long result = 0;
    for(int i = 0; i < remainders.size(); i++) {
        long long pp = prod / moduli[i];
        int x, y;
        extgcd(pp, moduli[i], x, y);
        result += remainders[i] * pp * x;
    }

    return ((result % prod) + prod) % prod;
}
```

Week 13-14: Strings & Advanced Algorithms

Topics Covered:

- String matching algorithms
- Suffix arrays and LCP arrays
- Trie data structure
- Advanced techniques

1. KMP Algorithm

cpp

```
vector<int> computeLPS(string pattern) {
  int m = pattern.length();
  vector<int> lps(m, 0);
  int len = 0, i = 1;

  while(i < m) {
    if(pattern[i] == pattern[len]) {
      len++;
      lps[i] = len;
      i++;
    }
    else {
      if(len != 0) {
        len = lps[len - 1];
      }
      else {
        lps[i] = 0;
        i++;
      }
    }
  }

  return lps;
}

vector<int> KMPSearch(string text, string pattern) {
  vector<int> result;
  vector<int> lps = computeLPS(pattern);
  int n = text.length(), m = pattern.length();
  int i = 0, j = 0;

  while(i < n) {
    if(pattern[j] == text[i]) {
      i++; j++;
    }

    if(j == m) {
      result.push_back(i - j);
      j = lps[j - 1];
    }
    else if(i < n && pattern[j] != text[i]) {
      if(j != 0) {
        j = lps[j - 1];
      }
      else {
        i++;
```

```
            }
        }
    }

    return result;
}
```

2. Trie Implementation

```cpp
```

```
class TrieNode {
public:
  unordered_map<char, TrieNode*> children;
  bool isEndOfWord;

  TrieNode() {
    isEndOfWord = false;
  }
};

class Trie {
private:
  TrieNode* root;

public:
  Trie() {
    root = new TrieNode();
  }

  void insert(string word) {
    TrieNode* current = root;
    for(char c : word) {
      if(current->children.find(c) == current->children.end()) {
        current->children[c] = new TrieNode();
      }
      current = current->children[c];
    }
    current->isEndOfWord = true;
  }

  bool search(string word) {
    TrieNode* current = root;
    for(char c : word) {
      if(current->children.find(c) == current->children.end()) {
        return false;
      }
      current = current->children[c];
    }
    return current->isEndOfWord;
  }

  bool startsWith(string prefix) {
    TrieNode* current = root;
    for(char c : prefix) {
      if(current->children.find(c) == current->children.end()) {
        return false;
```

```
        }
        current = current->children[c];
      }
      return true;
    }
};
```

3. Z Algorithm

```cpp
vector<int> zAlgorithm(string s) {
  int n = s.length();
  vector<int> z(n, 0);
  int l = 0, r = 0;

  for(int i = 1; i < n; i++) {
    if(i <= r) {
      z[i] = min(r - i + 1, z[i - l]);
    }

    while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }

    if(i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }

  return z;
}
```

Week 15-16: Geometry & Final Practice

Topics Covered:

- Basic geometry algorithms
- Convex hull
- Line intersection
- Contest strategies and optimization

1. Point and Line Structures

```cpp
struct Point {
    double x, y;

    Point(double x = 0, double y = 0) : x(x), y(y) {}

    Point operator+(const Point& p) const { return Point(x + p.x, y + p.y); }
    Point operator-(const Point& p) const { return Point(x - p.x, y - p.y); }
    Point operator*(double t) const { return Point(x * t, y * t); }

    double dot(const Point& p) const { return x * p.x + y * p.y; }
    double cross(const Point& p) const { return x * p.y - y * p.x; }

    double length() const { return sqrt(x * x + y * y); }
    double distance(const Point& p) const { return (p - *this).length(); }
};

struct Line {
    Point a, b;

    Line(Point a, Point b) : a(a), b(b) {}

    bool intersect(const Line& other, Point& intersection) const {
        Point dir1 = b - a;
        Point dir2 = other.b - other.a;

        double det = dir1.cross(dir2);
        if(abs(det) < 1e-9) return false; // Parallel lines

        double t = (other.a - a).cross(dir2) / det;
        intersection = a + dir1 * t;
        return true;
    }
};
```

2. Convex Hull (Graham Scan)

```cpp
```

```
int orientation(Point p, Point q, Point r) {
    double val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if(abs(val) < 1e-9) return 0;
    return (val > 0) ? 1 : 2; // 1 = Clockwise, 2 = Counterclockwise
}

vector<Point> convexHull(vector<Point> points) {
    int n = points.size();
    if(n < 3) return {};

    // Find bottom-most point
    int l = 0;
    for(int i = 1; i < n; i++) {
        if(points[i].y < points[l].y) l = i;
        else if(points[i].y == points[l].y && points[i].x < points[l].x) l = i;
    }

    swap(points[0], points[l]);

    // Sort points by polar angle
    sort(points.begin() + 1, points.end(), [&](Point a, Point b) {
        int o = orientation(points[0], a, b);
        if(o == 0) {
            return points[0].distance(a) < points[0].distance(b);
        }
        return o == 2;
    });

    vector<Point> hull;
    for(int i = 0; i < n; i++) {
        while(hull.size() > 1 &&
              orientation(hull[hull.size()-2], hull[hull.size()-1], points[i]) != 2) {
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }

    return hull;
}
```

Contest Tips & Template

Complete Contest Template

```
cpp
```

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

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef vector<ll> vll;

#define FOR(i, a, b) for(int i = (a); i < (b); i++)
#define RFOR(i, a, b) for(int i = (a); i >= (b); i--)
#define REP(i, n) FOR(i, 0, n)
#define ALL(v) v.begin(), v.end()
#define SZ(v) (int)v.size()
#define PB push_back
#define MP make_pair

const int INF = 1e9;
const ll LINF = 1e18;
const int MOD = 1e9 + 7;

void solve() {
    // Your solution here
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int t = 1;
    // cin >> t;

    while(t--) {
        solve();
    }

    return 0;
}
```

Practice Schedule

Daily Practice (2-3 hours):

1. **Week 1-8**: 5 easy + 3 medium problems daily
2. **Week 9-12**: 3 medium + 2 hard problems daily
3. **Week 13-16**: Virtual contests + upsolving

Weekly Contests:

- Participate in Codeforces rounds
- AtCoder Beginner/Regular contests
- Practice old ICPC regionals

Recommended Problem Sets:

- **Codeforces**: Div 2 A-D problems
- **AtCoder**: ABC problems
- **CSES Problem Set**: Complete all sections
- **UVa Online Judge**: Classic problems
- **Kattis**: ICPC-style problems

Study Resources

Books:

1. "Competitive Programming" by Steven Halim
2. "Introduction to Algorithms" by CLRS
3. "Guide to Competitive Programming" by Antti Laaksonen

Online Resources:

1. CP-Algorithms (https://cp-algorithms.com/)
2. USACO Guide (https://usaco.guide/)
3. Codeforces EDU section
4. YouTube: Errichto, SecondThread, William Lin

Practice Platforms:

- Codeforces
- AtCoder
- CodeChef
- SPOJ
- HackerRank
- LeetCode (for DP and data structures)

Advanced Topics & Implementations

Network Flows

1. Maximum Flow (Dinic's Algorithm)

```cpp
```

```cpp
struct Edge {
    int to, rev, cap;
};

class MaxFlow {
private:
    vector<vector<Edge>> graph;
    vector<int> level, iter;

    void bfs(int s) {
        fill(level.begin(), level.end(), -1);
        queue<int> q;
        level[s] = 0;
        q.push(s);

        while(!q.empty()) {
            int v = q.front();
            q.pop();
            for(auto& e : graph[v]) {
                if(e.cap > 0 && level[e.to] < 0) {
                    level[e.to] = level[v] + 1;
                    q.push(e.to);
                }
            }
        }
    }

    int dfs(int v, int t, int f) {
        if(v == t) return f;

        for(int& i = iter[v]; i < graph[v].size(); i++) {
            Edge& e = graph[v][i];
            if(e.cap > 0 && level[v] < level[e.to]) {
                int d = dfs(e.to, t, min(f, e.cap));
                if(d > 0) {
                    e.cap -= d;
                    graph[e.to][e.rev].cap += d;
                    return d;
                }
            }
        }
        return 0;
    }

public:
    MaxFlow(int n) {
```

```cpp
        graph.resize(n);
        level.resize(n);
        iter.resize(n);
    }

    void addEdge(int from, int to, int cap) {
        graph[from].push_back({to, (int)graph[to].size(), cap});
        graph[to].push_back({from, (int)graph[from].size() - 1, 0});
    }

    int maxflow(int s, int t) {
        int flow = 0;
        while(true) {
            bfs(s);
            if(level[t] < 0) return flow;
            fill(iter.begin(), iter.end(), 0);
            int f;
            while((f = dfs(s, t, INT_MAX)) > 0) {
                flow += f;
            }
        }
    }
};
```

2. Minimum Cost Maximum Flow

```cpp
```

```cpp
struct Edge {
    int to, rev, flow, cap, cost;
};

class MinCostMaxFlow {
private:
    vector<vector<Edge>> graph;
    vector<int> dist, parent, parent_edge;

    bool spfa(int s, int t) {
        fill(dist.begin(), dist.end(), INT_MAX);
        vector<bool> inQueue(graph.size(), false);
        queue<int> q;

        dist[s] = 0;
        q.push(s);
        inQueue[s] = true;

        while(!q.empty()) {
            int v = q.front();
            q.pop();
            inQueue[v] = false;

            for(int i = 0; i < graph[v].size(); i++) {
                Edge& e = graph[v][i];
                if(e.cap > e.flow && dist[v] + e.cost < dist[e.to]) {
                    dist[e.to] = dist[v] + e.cost;
                    parent[e.to] = v;
                    parent_edge[e.to] = i;
                    if(!inQueue[e.to]) {
                        q.push(e.to);
                        inQueue[e.to] = true;
                    }
                }
            }
        }

        return dist[t] != INT_MAX;
    }

public:
    MinCostMaxFlow(int n) {
        graph.resize(n);
        dist.resize(n);
        parent.resize(n);
        parent_edge.resize(n);
```

```
    }

    void addEdge(int from, int to, int cap, int cost) {
        graph[from].push_back({to, (int)graph[to].size(), 0, cap, cost});
        graph[to].push_back({from, (int)graph[from].size() - 1, 0, 0, -cost});
    }

    pair<int, int> minCostMaxFlow(int s, int t) {
        int flow = 0, cost = 0;

        while(spfa(s, t)) {
            int pushFlow = INT_MAX;
            for(int v = t; v != s; v = parent[v]) {
                Edge& e = graph[parent[v]][parent_edge[v]];
                pushFlow = min(pushFlow, e.cap - e.flow);
            }

            for(int v = t; v != s; v = parent[v]) {
                Edge& e = graph[parent[v]][parent_edge[v]];
                e.flow += pushFlow;
                graph[v][e.rev].flow -= pushFlow;
                cost += pushFlow * e.cost;
            }

            flow += pushFlow;
        }

        return {flow, cost};
    }
};
```

Matrix Operations

1. Matrix Multiplication and Exponentiation

```
cpp
```

```cpp
typedef vector<vector<ll>> Matrix;

Matrix multiply(const Matrix& A, const Matrix& B, int mod = MOD) {
    int n = A.size(), m = B[0].size(), p = B.size();
    Matrix C(n, vector<ll>(m, 0));

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            for(int k = 0; k < p; k++) {
                C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % mod;
            }
        }
    }

    return C;
}

Matrix matrixPower(Matrix A, ll n, int mod = MOD) {
    int size = A.size();
    Matrix result(size, vector<ll>(size, 0));

    // Initialize identity matrix
    for(int i = 0; i < size; i++) {
        result[i][i] = 1;
    }

    while(n > 0) {
        if(n & 1) {
            result = multiply(result, A, mod);
        }
        A = multiply(A, A, mod);
        n >>= 1;
    }

    return result;
}

// Example: Fibonacci using matrix exponentiation
ll fibonacci(ll n) {
    if(n <= 1) return n;

    Matrix fib = {{1, 1}, {1, 0}};
    Matrix result = matrixPower(fib, n - 1);
```

```
    return result[0][0];
}
```

Heavy-Light Decomposition

HLD Implementation

```cpp
```

```
class HeavyLightDecomposition {
private:
    vector<vector<int>> adj;
    vector<int> parent, depth, heavy, head, pos;
    SegmentTree segTree;
    int currentPos;

    int dfs(int v) {
        int size = 1, maxChildSize = 0;

        for(int u : adj[v]) {
            if(u != parent[v]) {
                parent[u] = v;
                depth[u] = depth[v] + 1;
                int childSize = dfs(u);

                if(childSize > maxChildSize) {
                    maxChildSize = childSize;
                    heavy[v] = u;
                }
                size += childSize;
            }
        }

        return size;
    }

    void decompose(int v, int h) {
        head[v] = h;
        pos[v] = currentPos++;

        if(heavy[v] != -1) {
            decompose(heavy[v], h);
        }

        for(int u : adj[v]) {
            if(u != parent[v] && u != heavy[v]) {
                decompose(u, u);
            }
        }
    }

public:
    HeavyLightDecomposition(vector<vector<int>>& graph, vector<int>& values)
        : adj(graph), segTree(values) {
        int n = adj.size();
```

```
        parent.assign(n, -1);
        depth.assign(n, 0);
        heavy.assign(n, -1);
        head.resize(n);
        pos.resize(n);
        currentPos = 0;

        dfs(0);
        decompose(0, 0);
    }

    void update(int v, int val) {
        segTree.update(pos[v], val);
    }

    ll query(int u, int v) {
        ll result = 0;

        while(head[u] != head[v]) {
            if(depth[head[u]] > depth[head[v]]) {
                result += segTree.query(pos[head[u]], pos[u]);
                u = parent[head[u]];
            }
            else {
                result += segTree.query(pos[head[v]], pos[v]);
                v = parent[head[v]];
            }
        }

        if(depth[u] > depth[v]) swap(u, v);
        result += segTree.query(pos[u], pos[v]);

        return result;
    }
};
```

Advanced String Algorithms

1. Suffix Array with LCP

```cpp
```

```
class SuffixArray {
private:
    string s;
    vector<int> sa, rank, lcp;

public:
    SuffixArray(string str) : s(str + "$") {
        int n = s.length();
        sa.resize(n);
        rank.resize(n);
        lcp.resize(n);

        buildSuffixArray();
        buildLCP();
    }

    void buildSuffixArray() {
        int n = s.length();
        vector<int> cnt(max(256, n), 0);

        // Initial sort by first character
        for(int i = 0; i < n; i++) cnt[s[i]]++;
        for(int i = 1; i < 256; i++) cnt[i] += cnt[i-1];
        for(int i = n-1; i >= 0; i--) sa[--cnt[s[i]]] = i;

        rank[sa[0]] = 0;
        for(int i = 1; i < n; i++) {
            rank[sa[i]] = rank[sa[i-1]] + (s[sa[i]] != s[sa[i-1]]);
        }

        for(int k = 1; k < n; k <<= 1) {
            vector<int> newSa(n), newRank(n);

            // Sort by second key
            int pos = 0;
            for(int i = n - k; i < n; i++) newSa[pos++] = i;
            for(int i = 0; i < n; i++) {
                if(sa[i] >= k) newSa[pos++] = sa[i] - k;
            }

            // Sort by first key
            fill(cnt.begin(), cnt.end(), 0);
            for(int i = 0; i < n; i++) cnt[rank[i]]++;
            for(int i = 1; i < n; i++) cnt[i] += cnt[i-1];
            for(int i = n-1; i >= 0; i--) sa[--cnt[rank[newSa[i]]]] = newSa[i];
```

```cpp
            // Update ranks
            newRank[sa[0]] = 0;
            for(int i = 1; i < n; i++) {
                newRank[sa[i]] = newRank[sa[i-1]] +
                    (rank[sa[i]] != rank[sa[i-1]] ||
                     sa[i] + k >= n || sa[i-1] + k >= n ||
                     rank[sa[i] + k] != rank[sa[i-1] + k]);
            }

            rank = newRank;
            if(rank[sa[n-1]] == n-1) break;
        }
    }

    void buildLCP() {
        int n = s.length();
        vector<int> invSa(n);

        for(int i = 0; i < n; i++) invSa[sa[i]] = i;

        int k = 0;
        for(int i = 0; i < n; i++) {
            if(invSa[i] == n-1) {
                k = 0;
                continue;
            }

            int j = sa[invSa[i] + 1];
            while(i + k < n && j + k < n && s[i + k] == s[j + k]) k++;

            lcp[invSa[i]] = k;
            if(k) k--;
        }
    }

    vector<int> getSuffixArray() { return sa; }
    vector<int> getLCP() { return lcp; }
};
```

2. Manacher's Algorithm (Palindromes)

```cpp
```

```cpp
vector<int> manacher(string s) {
    string modified = "#";
    for(char c : s) {
        modified += c;
        modified += "#";
    }

    int n = modified.length();
    vector<int> p(n, 0);
    int center = 0, right = 0;

    for(int i = 0; i < n; i++) {
        if(i < right) {
            p[i] = min(right - i, p[2 * center - i]);
        }

        while(i + p[i] + 1 < n && i - p[i] - 1 >= 0 &&
              modified[i + p[i] + 1] == modified[i - p[i] - 1]) {
            p[i]++;
        }

        if(i + p[i] > right) {
            center = i;
            right = i + p[i];
        }
    }

    return p;
}

// Find longest palindromic substring
string longestPalindrome(string s) {
    vector<int> p = manacher(s);
    int maxLen = 0, centerIndex = 0;

    for(int i = 0; i < p.size(); i++) {
        if(p[i] > maxLen) {
            maxLen = p[i];
            centerIndex = i;
        }
    }

    int start = (centerIndex - maxLen) / 2;
    return s.substr(start, maxLen);
}
```

Mo's Algorithm

Mo's Algorithm Implementation

```cpp
```

```
struct Query {
    int l, r, idx;
    int blockIdx;

    bool operator<(const Query& other) const {
        if(blockIdx != other.blockIdx) {
            return blockIdx < other.blockIdx;
        }
        return (blockIdx & 1) ? (r < other.r) : (r > other.r);
    }
};

class MoAlgorithm {
private:
    vector<int> arr;
    vector<int> freq;
    int currentAnswer;
    int blockSize;

    void add(int pos) {
        freq[arr[pos]]++;
        if(freq[arr[pos]] == 1) {
            currentAnswer++;
        }
    }

    void remove(int pos) {
        freq[arr[pos]]--;
        if(freq[arr[pos]] == 0) {
            currentAnswer--;
        }
    }

public:
    MoAlgorithm(vector<int>& array) : arr(array) {
        int n = arr.size();
        blockSize = sqrt(n) + 1;
        freq.resize(1000001, 0);
        currentAnswer = 0;
    }

    vector<int> processQueries(vector<pair<int, int>>& queries) {
        vector<Query> q;
        for(int i = 0; i < queries.size(); i++) {
            q.push_back({queries[i].first, queries[i].second, i,
                        queries[i].first / blockSize});
```

```cpp
    }

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

    vector<int> answers(queries.size());
    int currentL = 0, currentR = -1;

    for(Query& query : q) {
      while(currentL > query.l) {
        currentL--;
        add(currentL);
      }
      while(currentR < query.r) {
        currentR++;
        add(currentR);
      }
      while(currentL < query.l) {
        remove(currentL);
        currentL++;
      }
      while(currentR > query.r) {
        remove(currentR);
        currentR--;
      }

      answers[query.idx] = currentAnswer;
    }

    return answers;
  }
};
```

Square Root Decomposition

SQRT Decomposition for Range Updates

```cpp
```

```cpp
class SqrtDecomposition {
private:
    vector<ll> arr, lazy;
    int blockSize, numBlocks;

public:
    SqrtDecomposition(vector<int>& array) {
        arr.assign(array.begin(), array.end());
        int n = arr.size();
        blockSize = sqrt(n) + 1;
        numBlocks = (n + blockSize - 1) / blockSize;
        lazy.assign(numBlocks, 0);
    }

    void update(int l, int r, ll val) {
        int startBlock = l / blockSize;
        int endBlock = r / blockSize;

        if(startBlock == endBlock) {
            for(int i = l; i <= r; i++) {
                arr[i] += val;
            }
        }
        else {
            // Update partial first block
            for(int i = l; i < (startBlock + 1) * blockSize; i++) {
                arr[i] += val;
            }

            // Update complete middle blocks
            for(int i = startBlock + 1; i < endBlock; i++) {
                lazy[i] += val;
            }

            // Update partial last block
            for(int i = endBlock * blockSize; i <= r; i++) {
                arr[i] += val;
            }
        }
    }

    ll query(int pos) {
        int block = pos / blockSize;
        return arr[pos] + lazy[block];
    }
```

```cpp
    ll rangeSum(int l, int r) {
        ll sum = 0;
        for(int i = l; i <= r; i++) {
            sum += query(i);
        }
        return sum;
    }
};
```

Contest Strategy & Tips

Time Management

1. **First 15 minutes**: Read all problems, identify easy ones
2. **Next 2 hours**: Solve problems in order of difficulty
3. **Last hour**: Debug, optimize, attempt harder problems

Problem-Solving Approach

1. **Understand**: Read problem 2-3 times
2. **Examples**: Work through sample cases manually
3. **Algorithm**: Choose appropriate algorithm/data structure
4. **Edge cases**: Consider boundary conditions
5. **Implementation**: Code carefully with good variable names
6. **Testing**: Test with samples and edge cases

Common Mistakes to Avoid

- Integer overflow (use long long when needed)
- Array bounds (0-indexed vs 1-indexed)
- Infinite loops in graph algorithms
- Not handling empty input
- Wrong time complexity analysis

Debugging Techniques

```cpp
```

```cpp
#ifdef LOCAL
#define debug(x) cout << #x << " = " << x << endl
#else
#define debug(x)
#endif

// Print arrays
template<typename T>
void printArray(vector<T>& arr) {
  for(int i = 0; i < arr.size(); i++) {
    cout << arr[i] << (i == arr.size()-1 ? "\n" : " ");
  }
}
```

Final Recommendations

Last Month Before Contest:

1. **Virtual contests**: 3-4 per week
2. **Team practice**: Coordinate with teammates
3. **Upsolving**: Spend equal time on upsolving
4. **Weak areas**: Focus on your weakest topics
5. **Template**: Finalize your contest template

Contest Day:

1. **Sleep well**: 7-8 hours of sleep
2. **Eat properly**: Light meal before contest
3. **Arrive early**: Set up your workspace
4. **Stay calm**: Don't panic on hard problems
5. **Communicate**: Work effectively with teammates

Remember: Consistent practice is key. Solve at least 5-10 problems daily and participate in regular contests. The combination of theoretical knowledge and practical problem-solving experience will prepare you well for ACM-ICPC success.

Good luck with your preparation!