Algorithms & Problem Solving

Tawhid Bin Omar

1 Time Complexity Quick Reference

Complexity	Max Input (1s)
$O(1)$	Any
$O(\log n)$	10^{18}
$O(n)$	10^8
$O(n \log n)$	10^6
$O(n^2)$	10^4
$O(n^3)$	500
$O(2^n)$	20
$O(n!)$	11

2 Binary Search

Time: $O(\log n)$ | **Space:** $O(1)$

Search in sorted arrays or find optimal values using predicate functions.

```cpp
// Standard binary search
int binarySearch(vector<int>& arr, int target) {
    int l = 0, r = arr.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) l = mid + 1;
        else r = mid - 1;
    }
    return -1; // not found
}

// Finding first position >= target
int lowerBound(vector<int>& arr, int target) {
    int l = 0, r = arr.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < target) l = mid + 1;
        else r = mid;
    }
    return l;
}

// Binary search on answer
bool check(int mid) {
    // Check if mid satisfies condition
    return true;
}

int binarySearchAnswer(int low, int high) {
    int ans = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (check(mid)) {
            ans = mid;
            high = mid - 1; // search left
        } else {
            low = mid + 1;
        }
    }
    return ans;
}
```

Problem: Square Root

Find $\lfloor \sqrt{n} \rfloor$ for $n \leq 10^{18}$ without using built-in sqrt.

Solution

```cpp
long long sqrtFloor(long long n) {
    long long l = 0, r = 2e9, ans = 0;
    while (l <= r) {
        long long mid = l + (r - l) / 2;
        if (mid * mid <= n) {
            ans = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return ans;
}
```

3 Two Pointers

Time: $O(n)$ | **Space:** $O(1)$

Efficient technique for array/string problems.

```cpp
// Two sum in sorted array
pair<int,int> twoSum(vector<int>& arr,
    int target) {
    int l = 0, r = arr.size() - 1;
    while (l < r) {
        int sum = arr[l] + arr[r];
        if (sum == target)
            return {l, r};
        else if (sum < target)
            l++;
        else
            r--;
    }
    return {-1, -1};
}

// Remove duplicates in-place
int removeDuplicates(vector<int>& arr)
    {
    if (arr.empty()) return 0;
    int j = 0;
    for (int i = 1; i < arr.size();
        i++) {
        if (arr[i] != arr[j]) {
            j++;
            arr[j] = arr[i];
        }
    }
    return j + 1;
}
```

4 Sliding Window

Time: $O(n)$ | **Space:** $O(k)$

For subarray/substring problems.

```cpp
// Maximum sum subarray of size k
int maxSumSubarray(vector<int>& arr,
    int k) {
    int n = arr.size();
    if (n < k) return -1;

    int maxSum = 0;
    for (int i = 0; i < k; i++)
        maxSum += arr[i];

    int windowSum = maxSum;
    for (int i = k; i < n; i++) {
        windowSum += arr[i] - arr[i - k];
        maxSum = max(maxSum, windowSum);
    }
    return maxSum;
}

// Longest substring with k distinct
    chars
int longestSubstringKDistinct(string s,
    int k) {
    unordered_map<char, int> freq;
    int l = 0, maxLen = 0;

    for (int r = 0; r < s.length();
        r++) {
        freq[s[r]]++;

        while (freq.size() > k) {
            freq[s[l]]--;
            if (freq[s[l]] == 0)
                freq.erase(s[l]);
            l++;
        }
        maxLen = max(maxLen, r - l + 1);
    }
    return maxLen;
}
```

Tip

Use sliding window when looking for subarrays or substrings with specific properties. Expand window by moving right pointer, shrink by moving left pointer.

5 Prefix Sums

Time: $O(n)$ build, $O(1)$ query

Precompute cumulative values for efficient range queries.

```cpp
// 1D prefix sum
```

```
vector<int> buildPrefix(vector<int>& arr) {
    int n = arr.size();
    vector<int> prefix(n + 1, 0);
    for (int i = 0; i < n; i++)
        prefix[i + 1] = prefix[i] + arr[i];
    return prefix;
}

// Range sum query [l, r]
int rangeSum(vector<int>& prefix, int l,
    int r) {
    return prefix[r + 1] - prefix[l];
}

// 2D prefix sum
vector<vector<int>> build2DPrefix(
    vector<vector<int>>& mat) {
    int n = mat.size(), m = mat[0].size();
    vector<vector<int>> prefix(n + 1,
        vector<int>(m + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            prefix[i][j] = mat[i-1][j-1]
                + prefix[i-1][j]
                + prefix[i][j-1]
                - prefix[i-1][j-1];
        }
    }
    return prefix;
}

// Query sum in rectangle (r1,c1) to (r2,c2)
int rectangleSum(vector<vector<int>>& prefix,
    int r1, int c1, int r2, int c2) {
    return prefix[r2+1][c2+1]
        - prefix[r1][c2+1]
        - prefix[r2+1][c1]
        + prefix[r1][c1];
}
```

6 Dynamic Programming

1. Fibonacci / Linear DP

```
// Climbing stairs (1 or 2 steps)
int climbStairs(int n) {
    if (n <= 2) return n;
    int a = 1, b = 2;
    for (int i = 3; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

2. Knapsack

```
// 0/1 Knapsack
int knapsack(vector<int>& wt,
    vector<int>& val, int W) {
    int n = wt.size();
    vector<vector<int>> dp(n + 1,
        vector<int>(W + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            dp[i][w] = dp[i-1][w];
            if (w >= wt[i-1]) {
                dp[i][w] = max(dp[i][w],
                    dp[i-1][w-wt[i-1]] +
                        val[i-1]);
            }
        }
    }
    return dp[n][W];
}
```

3. Longest Common Subsequence

```
int lcs(string s1, string s2) {
    int n = s1.length(), m =
        s2.length();
    vector<vector<int>> dp(n + 1,
        vector<int>(m + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i-1] == s2[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[n][m];
}
```

Problem: Coin Change

Given coins of denominations and amount n, find minimum coins needed to make amount n.

Solution

```cpp
int coinChange(vector<int>& coins,
    int amount) {
    vector<int> dp(amount + 1,
        INT_MAX);
    dp[0] = 0;

    for (int i = 1; i <= amount;
        i++) {
        for (int coin : coins) {
            if (i >= coin &&
                dp[i - coin] !=
                    INT_MAX)
                dp[i] = min(dp[i],
                    dp[i - coin] + 1);
        }
    }
    return dp[amount] == INT_MAX ?
        -1 : dp[amount];
}
```

7 Graph Algorithms

Graph Representation

```cpp
// Adjacency list
vector<vector<int>> adj(n);
adj[u].push_back(v);

// Weighted edges
vector<vector<pair<int,int>>> adj(n);
adj[u].push_back({v, weight});
```

DFS - Depth First Search

Time: $O(V + E)$

```cpp
vector<bool> visited;

void dfs(int u, vector<vector<int>>&
    adj) {
    visited[u] = true;
    // Process node u

    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v, adj);
        }
    }
}

// Iterative DFS
void dfsIterative(int start,
    vector<vector<int>>& adj) {
    stack<int> st;
    st.push(start);
    visited[start] = true;

    while (!st.empty()) {
        int u = st.top();
        st.pop();

        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                st.push(v);
            }
        }
    }
}
```

BFS - Breadth First Search

Time: $O(V + E)$

```cpp
void bfs(int start,
    vector<vector<int>>& adj) {
    queue<int> q;
    vector<bool> visited(adj.size(),
        false);
    vector<int> dist(adj.size(), -1);

    q.push(start);
    visited[start] = true;
    dist[start] = 0;

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

        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

Dijkstra's Algorithm

Time: $O((V + E) \log V)$

Shortest path in weighted graphs with non-negative weights.

```cpp
vector<int> dijkstra(int start, int n,
    vector<vector<pair<int,int>>>& adj)
        {
    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()) {
        auto [d, u] = pq.top();
        pq.pop();

        if (d > dist[u]) continue;

        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Topological Sort

Time: $O(V + E)$

For Directed Acyclic Graphs (DAG).

```
vector<int> topologicalSort(int n,
    vector<vector<int>>& adj) {
    vector<int> indegree(n, 0);

    for (int u = 0; u < n; u++) {
        for (int v : adj[u]) {
            indegree[v]++;
        }
    }

    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (indegree[i] == 0)
            q.push(i);
    }

    vector<int> result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        for (int v : adj[u]) {
            indegree[v]--;
            if (indegree[v] == 0)
                q.push(v);
        }
    }

    // If result.size() != n, cycle
        exists
    return result;
}
```

8 Greedy Algorithms

Make locally optimal choices at each step.

Problem: Activity Selection

Given start and end times of activities, select maximum non-overlapping activities.

Solution

```
int maxActivities(
    vector<pair<int,int>>&
        activities) {
    // Sort by end time
    sort(activities.begin(),
        activities.end(),
        [](auto& a, auto& b) {
            return a.second <
                b.second;
        });

    int count = 1;
    int lastEnd =
        activities[0].second;

    for (int i = 1; i <
        activities.size(); i++) {
        if (activities[i].first >=
            lastEnd) {
            count++;
            lastEnd =
                activities[i].second;
        }
    }
    return count;
}
```

9 Number Theory

GCD and LCM

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

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

Prime Numbers

```
// Sieve of Eratosthenes
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;
}

// Check if prime
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0)
        return false;

    for (int i = 5; i * i <= n; i += 6)
        {
        if (n % i == 0 || n % (i + 2) ==
            0)
            return false;
    }
    return true;
}
```

Modular Arithmetic

```
const int MOD = 1e9 + 7;

// Modular exponentiation
long long power(long long a, long long
    b) {
    long long res = 1;
    a %= MOD;
    while (b > 0) {
        if (b & 1) res = (res * a) % MOD;
        a = (a * a) % MOD;
        b >>= 1;
    }
    return res;
}

// Modular inverse (when MOD is prime)
long long modInv(long long a) {
    return power(a, MOD - 2);
}
```

10 Bit Manipulation

```
// Check if k-th bit is set
bool isSet(int n, int k) {
    return (n & (1 << k)) != 0;
}

// Set k-th bit
int setBit(int n, int k) {
    return n | (1 << k);
}

// Clear k-th bit
int clearBit(int n, int k) {
    return n & ~(1 << k);
}

// Toggle k-th bit
int toggleBit(int n, int k) {
    return n ^ (1 << k);
}

// Count set bits
int countSetBits(int n) {
    return __builtin_popcount(n);
}

// Check if power of 2
bool isPowerOf2(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}

// Get rightmost set bit
int rightmostSetBit(int n) {
    return n & -n;
}
```

Problem: Single Number

Every element appears twice except one. Find the single element. $O(n)$ time, $O(1)$ space.

Solution

```
int singleNumber(vector<int>&
    nums) {
    int result = 0;
    for (int num : nums)
        result ^= num; // XOR
            cancels pairs
    return result;
}
```

11 STL Essentials

Useful STL Functions

```
// Sort
sort(v.begin(), v.end());
sort(v.begin(), v.end(),
    greater<int>());

// Binary search
binary_search(v.begin(), v.end(), x);
lower_bound(v.begin(), v.end(), x);
upper_bound(v.begin(), v.end(), x);

// Min/Max
*min_element(v.begin(), v.end());
*max_element(v.begin(), v.end());
```

```
// Reverse
reverse(v.begin(), v.end());

// Next permutation
next_permutation(v.begin(), v.end());

// Unique (remove consecutive
    duplicates)
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()),
    v.end());

// Count
count(v.begin(), v.end(), x);

// Accumulate
accumulate(v.begin(), v.end(), 0);
```

Priority Queue (Heap)

```
// Max heap (default)
priority_queue<int> maxHeap;
maxHeap.push(5);
int top = maxHeap.top();
maxHeap.pop();

// Min heap
priority_queue<int, vector<int>,
    greater<int>> minHeap;

// Custom comparator
auto cmp = [](pair<int,int>& a,
    pair<int,int>& b) {
    return a.second > b.second;
};
priority_queue<pair<int,int>,
    vector<pair<int,int>>,
    decltype(cmp)> pq(cmp);
```

12 Problem-Solving Tips

Tip

Approach Strategy:

1. Read problem carefully, identify constraints
2. Consider time/space complexity limits
3. Think of brute force first
4. Optimize: Sort? Hash? DP? Greedy?
5. Implement and test edge cases

Common Patterns:

- **Array/String:** Two pointers, sliding window, prefix sums
- **Searching:** Binary search on answer
- **Optimization:** DP, greedy
- **Connectivity:** DFS, BFS, Union-Find
- **Shortest Path:** BFS (unweighted), Dijkstra (weighted)
- **Counting:** Combinatorics, DP

Practice Problems

Binary Search:

- Codeforces 1201C - Maximum Median
- CSES - Factory Machines

Two Pointers:

- LeetCode 15 - 3Sum
- Codeforces 279B - Books

Dynamic Programming:

- CSES - Coin Combinations
- LeetCode 322 - Coin Change
- AtCoder DP Contest

Graphs:

- CSES - Shortest Routes I
- LeetCode 207 - Course Schedule
- Codeforces 115A - Party

Greedy:

- CSES - Tasks and Deadlines
- LeetCode 435 - Non-overlapping Intervals