SOC MID TERM REPORT

APPLIED PROBLEM SOLVING IN CP

NEHA

23B1801

Introduction to Competitive Programming

What is Competitive Programming?

Competitive Programming (CP) is a mental sport where participants solve well-defined problems using programming within a limited time. It is both a practice ground for improving algorithmic thinking and a competitive environment where speed, accuracy, and efficiency are tested

Key Features of CP

- **Time Constraints:** Solutions must be optimized for speed.
- **Memory Constraints:** Code must use memory efficiently.
- **Correctness:** The solution must work for all possible edge cases.
- **Languages Used:** Mostly C++, Python, and Java.

Purpose of CP

- **Sharpen problem-solving skills**
- **Improve algorithm and data structure knowledge**
- **Prepare for technical interviews**
- **Enhance coding speed and accuracy**
- **Participate in contests like Codeforces, AtCoder, LeetCode, CodeChef, etc.**

Why Should You Learn CP?

1. **Strong Foundation in Algorithms:** CP exposes you to a wide variety of problems requiring knowledge of sorting, searching, dynamic programming, graphs, trees, etc.
2. **Efficient Thinking:** You learn how to solve problems in the most optimal way.
3. **Career Advantage:** Many top companies like Google, Meta, Amazon, etc., value CP skills during hiring.
4. **Logical and Analytical Thinking:** It trains your brain to think clearly, logically, and quickly.
5. **Fun and Challenging:** CP is intellectually satisfying and addictive once you start solving problems successfully

Competitive Programming is not just for competitions; it builds a strong foundation that benefits every aspect of programming and software development. Whether you're aiming for tech interviews, academic improvement, or personal growth, CP is a powerful skill to develop

Basics of C++

Every C++ program follows a basic structure:

```
#include <iostream>   // Header file for input and output

using namespace std;  // Allows us to use standard library without std::

int main() {          // Main function - program execution starts here
    // Your code goes here
    return 0;         // Program ends successfully
}
```

1.Special Headers Commonly Used in CP:

Header	Purpose
#include <bits/stdc++.h>	Includes almost all standard C++ libraries (used in CP)
#include <vector>	For dynamic arrays
#include <algorithm>	For sort, max, min, etc.
#include <map>, <set>	For key-value pairs and unique containers

2.Understanding cout

cout stands for "character output" and is used to print values:

```
cout << "Hello World";    // prints Hello World
cout << x;                // prints value of x
```

Chaining:

```
cout << "Value is: " << x << "\n";
```

\n is used to move to the next line (faster than endl in CP).

3.Variables

Variables are names given to memory locations that store data.

```
int age = 20;
float pi = 3.14;
char grade = 'A';
```

Here age,pi,grade are variables

Rules:

- Must begin with a letter or underscore
- Cannot be a keyword
- Case-sensitive

4.Datatypes

Data Type	Example
int	1, 2, -10
long / long long	1e9+7
float	3.14
double	3.14159
char	'A', 'z'
bool	true,false

5.Simple Operators

Operator	Name	Example
+	Addition	a + b
-	Subtraction	a - b

Data Type		Example
*	Multiplication	a * b
/	Division	a / b
%	Modulo	a % b

6. Modulo Operator (%)

Used to get the remainder after division:

```
int a = 10;
int b = 3;
cout << a % b;    // Output: 1
```

Useful in:

- Checking even/odd: if (x % 2 == 0)
- Keeping numbers within range (% 1000000007)

7.Relational Operators

- Used to compare values. Return `true` or `false`.

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

8.Understanding cin

cin stands for "character input" – used to take user input:

```
int x;
cin >> x;
```

Multiple inputs:

```
int a, b;
cin >> a >> b;
```

Important: cin skips whitespaces. For reading full lines, use getline() instead:

```
string name;
getline(cin, name);  // reads entire line including spaces
```

9.Calculation Order in Datatypes

When an expression has different data types, C++ automatically promotes the smaller type to the larger one to maintain precision. This is called **type promotion**.

Type promotion hierarchy:

```
bool < char < short < int < long < long long < float < double < long double
```

- Always be aware of type conversions to avoid unexpected results, especially in divisions

10.Operator Precedence

Operator precedence determines which part of an expression is evaluated first when multiple operators are used.

Common precedence levels (from highest to lowest):

- () — Parentheses (overrides all)
- ++, --, ! — Unary operators
- *, /, % — Multiplicative

- +, - — Additive
- <, >, <=, >= — Relational
- ==, != — Equality
- && — Logical AND
- || — Logical OR
- =, +=, -= — Assignment

```
int x = 5 + 2 * 3;  // Output: 11, because * has higher precedence than +
```

11.Overflow

Overflow happens when a calculation exceeds the storage capacity of a data type.

```
int x = 2147483647; // Max value for 32-bit signed int
x = x + 1;
cout << x;  // Output: -2147483648 (wrap around)
```

- **Prefer long long in cases involving large calculations (like factorials, powers).**

Common maximum values:

- int: ±2,147,483,647
- long long: ±9,223,372,036,854,775,807
- Use unsigned to double the positive range but **cannot store negatives**

12.Conditional Statements

- Conditional logic is used to control the flow of the program based on conditions.

if-else:

```cpp
if (x > 0) {
    cout << "Positive";
} else if (x < 0) {
    cout << "Negative";
} else {
    cout << "Zero";
}
```

13.Loops

Loops help in executing a block of code multiple times.

for loop:

```cpp
for (int i = 0; i < 5; i++) {
    cout << i << " ";
}
```

while loop:

```cpp
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
```

do-while loop (runs at least once):

```cpp
int i = 0;
do {
    cout << i << " ";
    i++;
} while (i < 5);
```

14.Jump Statements

Used to modify normal flow inside loops:

- break: Exits the loop early.

- continue: Skips the current iteration.
- return: Exits the current function.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;  // Skip 3
    cout << i << " ";
}
```

15.Strings

The string class from <string> is commonly used in CP for text handling.

```
string s = "hello";
s += " world";        // Concatenation
cout << s[1];         // Access single character
cout << s.length();   // Get string length
```

16. getline()

To read full lines including spaces:

```
string line;
getline(cin, line);
```

Note: When using getline() after cin >>, flush newline using cin.ignore() before calling getline().

17.Arrays & Size Limits

Arrays are used for fast access and fixed-size storage.

Declaration:

```
int a[100005];  // Array of size 1e5
```

Input:

```
for (int i = 0; i < n; i++) cin >> a[i];
```

Local vs Global Limits:

- **Local arrays** (inside functions): Safe up to 10^5 elements
- **Global arrays** (outside main): Can be up to 10^7 or even more

For large data, **prefer vectors** or declare globally.

18.Functions

Functions allow modular code, reuse, and recursion.

Syntax:

```
int add(int a, int b) {
    return a + b;
}
```

Call:

```
int result = add(2, 3); // Output: 5
```

Recursive Function:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

19.C++ References (&)

References allow access to the **original variable**, not a copy.

Function by reference:

```
void update(int &x) {
    x += 5;
}
```

Usage:

```cpp
int a = 10;
update(a);  // a becomes 15
```

Used in:

- Swapping values
- Optimizing memory in large structures
- Passing arrays or vectors efficiently

20.Pointers & Memory in C++

1. Memory Allocation

- **Stack:** Local variables, ~1–8 MB limit.
- **Heap:** Dynamically allocated with new or malloc.
- Use **global** arrays or heap for large data (>10^6 elements)

2. Pointers Basics`

```cpp
int a = 10;
int *p = &a;    // p stores address of a
cout << *p;     // 10 (value)
```

- *p: Dereference – value at address
- &a: Address of variable

3. Pointer Arithmetic

```cpp
int arr[] = {1, 2, 3};
int *p = arr;
cout << *(p + 1);  // 2
```

- p + i points to arr[i].

4. Arrays & Pointers

```
int arr[3] = {1, 2, 3};
int *p = arr;
cout << *(p + 2);  // 3
```

- arr acts as pointer to the first element.

5. Double Pointers

```
int a = 5;
int *p = &a;
int **q = &p;
cout << **q;  // 5
```

- Used in 2D arrays or to modify pointers.

6. Function with Pointer

```
void update(int *p) {
    *p += 1;
}
```

- Modifies original variable via pointer.

7. Double Pointer Function

```
void update(int **p) {
    **p += 2;
}
```

- Can modify the original value using **p.

8. Passing Arrays to Functions

```
void print(int *arr, int n) {
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
}
```

- Arrays decay to pointers when passed.

9. All-in-One Example

```
void change(int *a, int **b) {
    *a += 1;
    **b += 2;
}

int x = 5, *p = &x;
change(p, &p);  // x becomes 8
```

Competitive Programming – Core Concepts

1. Basic Implementation

- **Understand problem statements carefully**: Look for input format, constraints, and required output.
- **Use loops (for, while), conditionals (if, else), and I/O (cin, cout)**.
- Start with clear input parsing and simple test cases.

2. Math & Number Theory

- **Factors & Multiples**: Loop till √n for efficiency.

```
for (int i = 1; i * i <= n; i++)
```

- **Prime Checking**:

```
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0) return false;
    return true;
}
```

- **GCD & LCM**:

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

- **Modular Arithmetic**:
    - Used to prevent overflow (% MOD)

- Apply while adding, multiplying, or subtracting large numbers.

3. Bit Manipulation

- Works directly on binary representation.
- **AND (&), OR (|), XOR (^)**

```
x & 1 → check if x is odd/even
x ^ y → toggles bits (used in odd frequency problems)
```

Common Problem:
Find number with odd frequency:

```
int result = 0;
for (int x : arr) result ^= x;
```

4. Greedy Algorithms

- **Make local optimal choices** hoping to reach global optimum.
- **Common Patterns**:
    - Sort elements
    - Pick best available option each step

Examples:

- **Activity Selection**: Sort by end time, pick non-overlapping intervals.
- **Coin Change (min coins)**: Pick largest possible coin first.

5. Sorting & Two Pointers

- **Sorting Algorithms**:
    - sort(arr, arr + n); // STL sort (uses intro sort)
    - Bubble, Merge, Quick sort (learn basics for interviews)
- **Two Pointers Technique**:
    - Works on sorted arrays.

- Use two indices moving toward each other or forward.

Example – Pair Sum:

```
int l = 0, r = n - 1;
while (l < r) {
    int sum = arr[l] + arr[r];
    if (sum == target) break;
    else if (sum < target) l++;
    else r--;
}
```

Searching Techniques in Competitive Programming

1.Binary Search on Arrays

Used to find an element in a sorted array in O(log n) time

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

2.Variants of Binary Search

- **First Occurrence**:
    - On finding the element, move high = mid - 1.
- **Last Occurrence**:
    - On finding, move low = mid + 1.

- **Lower Bound (floor)**:
    - Greatest element ≤ target.

- **Upper Bound (ceil)**:
    - Smallest element ≥ target.

STL:

```
int lb = lower_bound(arr, arr+n, x) - arr;
int ub = upper_bound(arr, arr+n, x) - arr;
```

3.Binary Search on Answer

Used when the **answer lies in a numeric range**, not in the array directly. Try to **check if a guess is valid** using a helper function.

Examples:

- **Aggressive Cows**: Maximize min distance between cows.
- **Koko Eating Bananas**: Minimize eating speed to finish in time.

Template:

```
int low = min_val, high = max_val, ans = -1;
while (low <= high) {
    int mid = (low + high) / 2;
    if (isValid(mid)) {
        ans = mid;
        high = mid - 1; // or low = mid + 1 based on problem
    } else {
        low = mid + 1;
    }
}
```

4.Search in Rotated Sorted Array

Modified binary search on arrays rotated at unknown pivot.

Key idea: One half is always sorted — use that to guide search.

```
if (arr[mid] >= arr[low]) {
    // left half is sorted
} else {
    // right half is sorted
}
```

Watch out for **duplicate values**, they may require shifting bounds slowly.

5.Peak Element Search

Find an element greater than both neighbors (local maximum).

Divide and Conquer approach:

```
int findPeak(vector<int> &arr) {
    int low = 0, high = arr.size() - 1;
    while (low < high) {
        int mid = (low + high) / 2;
        if (arr[mid] < arr[mid+1])
            low = mid + 1;
        else
            high = mid;
    }
    return low;  // or arr[low]
}
```

6.Nth Root using Binary Search

Find nth root of x using binary search with decimals.

```
double nthRoot(int n, int x) {
    double low = 1, high = x, eps = 1e-6;
    while (high - low > eps) {
        double mid = (low + high) / 2.0;
        if (pow(mid, n) < x)
            low = mid;
        else
            high = mid;
    }
    return low;
}
```

Array-Based Problem Solving

1. Prefix Sum / Suffix Sum

Used to **precompute cumulative values** for fast range queries.

Prefix Sum:

```
prefix[0] = arr[0];
for (int i = 1; i < n; i++)
    prefix[i] = prefix[i-1] + arr[i];
```

Range Sum Query [L...R]:

```
int sum = prefix[R] - (L > 0 ? prefix[L-1] : 0);
```

Applications:

- Sum of subarrays
- Count frequencies
- Balance checking (like left-right sums

2.Single Element in Sorted Array

- In a **sorted array with all elements repeated twice except one**, use XOR or binary search.
- **XOR Trick:**

```
int res = 0;
for (int x : arr) res ^= x;
```

- **Binary Search (when array is sorted and only one single):**

```
int low = 0, high = n - 1;
while (low < high) {
    int mid = low + (high - low) / 2;
    if ((mid % 2 == 0 && arr[mid] == arr[mid+1]) ||
        (mid % 2 == 1 && arr[mid] == arr[mid-1]))
        low = mid + 1;
    else
        high = mid;
}
return arr[low];
```

3.Split Arrays (Binary Search on Answer)

Problem: **Split array into k parts such that the maximum sum of any part is minimized**.

Approach:

- Use **binary search on possible max sums**.
- In each check, simulate splitting and count partitions.

```
bool isValid(vector<int>& a, int mid, int k) {
    int count = 1, sum = 0;
    for (int x : a) {
        if (x > mid) return false;
        if (sum + x > mid) { count++; sum = x; }
        else sum += x;
    }
    return count <= k;
}
```

4.Rotations and Count

Rotation means array is sorted and shifted.

Example: [4, 5, 6, 1, 2, 3]

Find number of rotations = index of minimum element

```
int findRotationCount(vector<int> &arr) {
    int low = 0, high = arr.size() - 1;
    while (low < high) {
        int mid = (low + high) / 2;
        if (arr[mid] > arr[high])
            low = mid + 1;
        else
            high = mid;
    }
    return low;  // Index of minimum element = rotations
}
```

String Algorithms in Competitive Programming

1. String Manipulation

Mastering string basics is essential in CP.

Common operations:

```
string s = "hello";

// Reversal
reverse(s.begin(), s.end());  // "olleh"

// Substring
string sub = s.substr(1, 3);  // "ell"

// Concatenation
string a = "hi", b = "there";
string c = a + b;             // "hithere"
```

2.Pattern Matching & Prefixes

Z-Algorithm (Z-array):
Used for pattern searching in linear time.

Use case: Find occurrences of a pattern in a string.

```cpp
string s = pattern + '$' + text;
vector<int> z = z_function(s);
```

Longest Common Prefix (LCP):
Used in suffix array construction and comparison of strings.

Basic prefix comparison:

```cpp
int commonPrefix(string a, string b) {
    int i = 0;
    while (i < a.size() && i < b.size() && a[i] == b[i]) i++;
    return i;
}
```

3.Isomorphic Strings & Anagrams

Anagram = strings with the same character counts.

Isomorphic Strings:
Two strings where character mapping is one-to-one.

Use two hash maps or fixed arrays to track mapping

Check anagram using frequency array:

```cpp
bool isAnagram(string a, string b) {
    if (a.size() != b.size()) return false;
    int freq[26] = {};
    for (char c : a) freq[c - 'a']++;
    for (char c : b) freq[c - 'a']--;
    for (int i : freq) if (i != 0) return false;
    return true;
}
```

4.Palindromes & Roman Conversion

Palindrome: Read the same backward and forward.

```
bool isPalindrome(string s) {
    int l = 0, r = s.size() - 1;
    while (l < r) {
        if (s[l++] != s[r--]) return false;
    }
    return true;
}
```

Roman Numeral to Integer:

```
unordered_map<char, int> roman = {{'I',1}, {'V',5}, ...};
int value = 0;
for (int i = 0; i < s.length(); i++) {
    if (roman[s[i]] < roman[s[i+1]])
        value -= roman[s[i]];
    else
        value += roman[s[i]];
}
```

5.Sort Characters by Frequency

Goal: Sort characters in descending order of frequency.

Use hash map + max heap:

```
unordered_map<char, int> freq;
for (char c : s) freq[c]++;

priority_queue<pair<int, char>> pq;
for (auto [ch, f] : freq) pq.push({f, ch});

string result = "";
while (!pq.empty()) {
    auto [f, ch] = pq.top(); pq.pop();
    result += string(f, ch);
}
```

Recursion & Backtracking in Competitive Programming

1. Subset & Power Set Generation

Used to generate **all possible subsets** (the power set) of a given set.

Recursive Inclusion/Exclusion Approach:

```
void generate(int idx, vector<int>& nums, vector<int>& curr) {
    if (idx == nums.size()) {
        print(curr);
        return;
    }

    // Include nums[idx]
    curr.push_back(nums[idx]);
    generate(idx + 1, nums, curr);

    // Exclude nums[idx]
    curr.pop_back();
    generate(idx + 1, nums, curr);
}
```

2. Combination Sum Variants

Choose numbers to sum to a target using **backtracking + pruning**.

Key Idea: Try all combinations, backtrack on failure.

```
void findCombinations(int idx, vector<int>& nums, int target, vector<int>& curr) {
    if (target == 0) {
        print(curr);
        return;
    }
    if (target < 0 || idx == nums.size()) return;

    // Include current
    curr.push_back(nums[idx]);
    findCombinations(idx, nums, target - nums[idx], curr);
    curr.pop_back();

    // Exclude current
    findCombinations(idx + 1, nums, target, curr);
}
```

3.Parentheses Generation

Generate all **valid combinations of parentheses**.

Use counts of open and close brackets:

```
void generate(int open, int close, string curr) {
    if (open == 0 && close == 0) {
        cout << curr << "\n";
        return;
    }
    if (open > 0) generate(open - 1, close, curr + "(");
    if (close > open) generate(open, close - 1, curr + ")");
}
```

4.Binary String Generation

Generate **all binary strings** of length n.

Simple DFS-style recursion:

```
void binaryStrings(int n, string curr) {
    if (n == 0) {
        cout << curr << "\n";
        return;
    }
    binaryStrings(n - 1, curr + "0");
    binaryStrings(n - 1, curr + "1");
}
```

** Use recursion when **choices build up** over time.

Use **backtracking** when you need to **undo choices** and explore other options.

Optimize using:

- **Pruning**: Stop early if solution is invalid.
- **Memoization**: Avoid re-computation if overlapping subproblems.