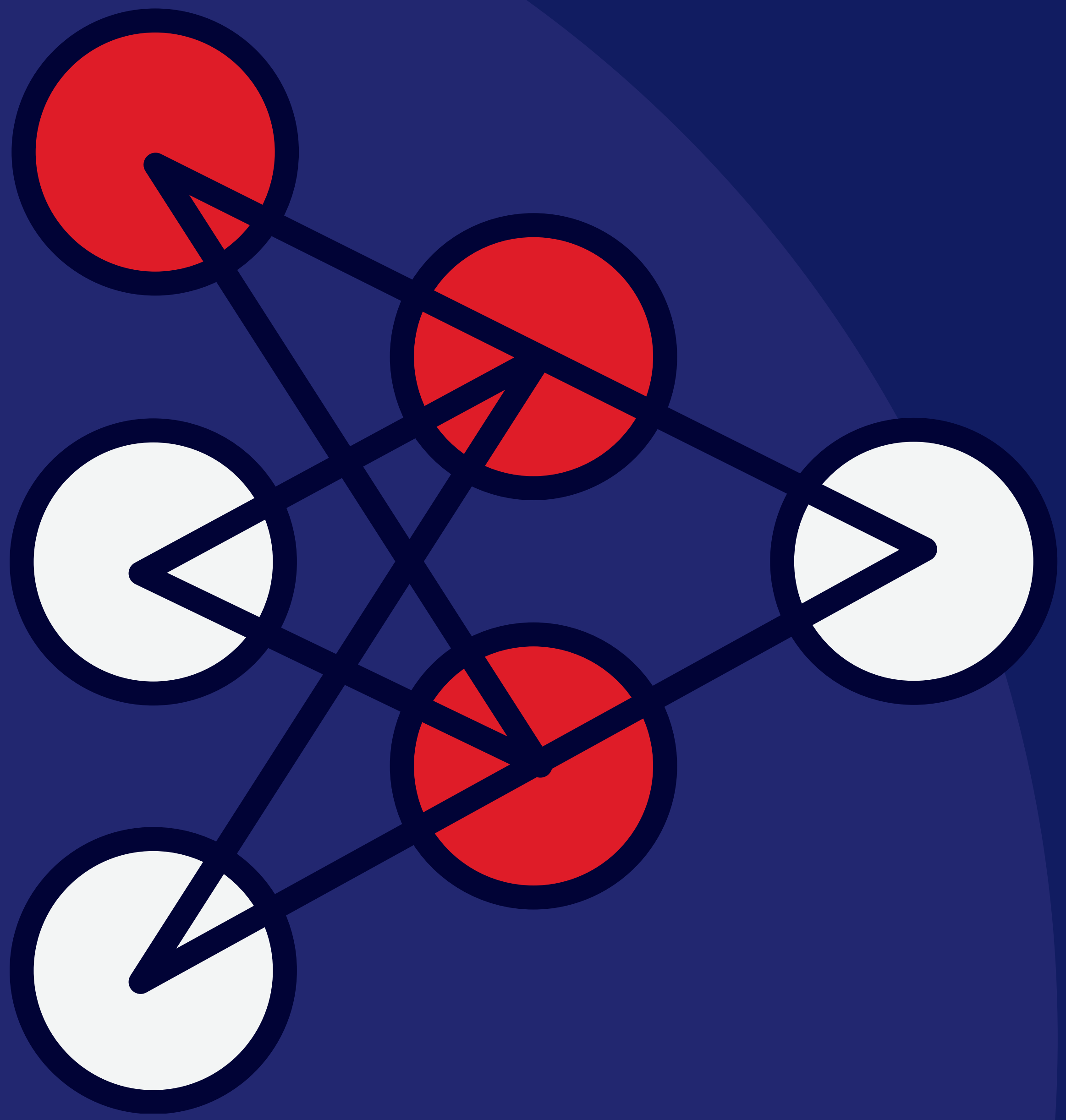
Kathan Modh
23CS046

WAYS TO REACH TARGET ELEMENTS USING ARRAY ELEMENTS

23CS046



Introduction

LET'S CONSIDER WE ARE GIVEN SOME ELEMENTS SUCH AS ARRAY { 5, 3, -6, 2 } WE NEED TO FIND WAYS TO ACHIEVE OUR DESIRED OUTPUT OF NUMBER 6 FOR EXAMPLE AND FROM THE ARRAY WE CAN TAKE ANY NUMBER OF ELEMENTS

THE TOTAL NUMBER OF WAYS TO ACHIEVE A TARGET SUM OF 6 USING ONLY + AND – OPERATORS IS 4 AS:

(-)-6 = 6

(+) 5 (+) 3 (-) 2 = 6

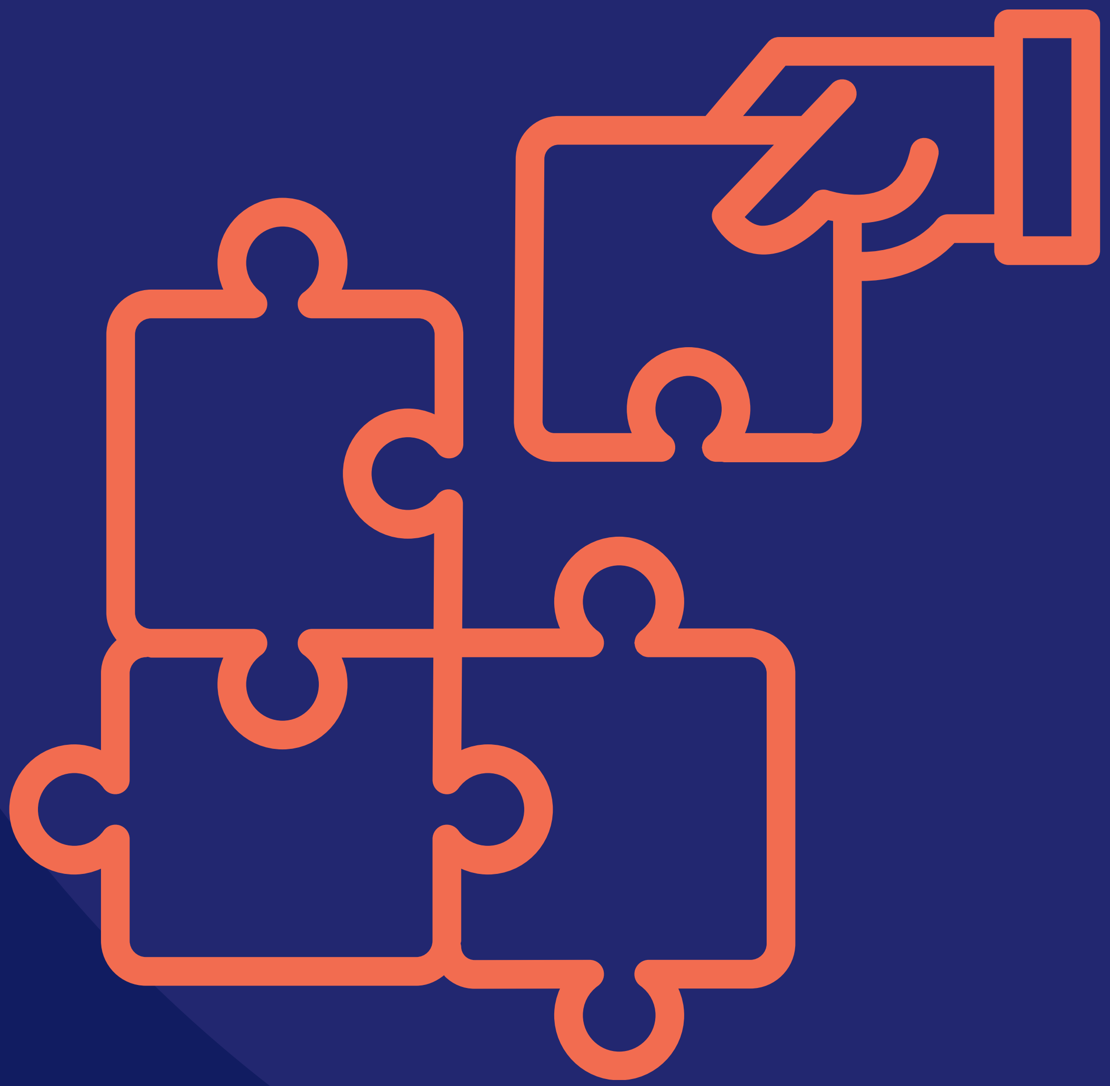
(+) 5 (-) 3 (-) -6 (-) 2 = 6

(-) 5 (+) 3 (-) -6 (+) 2 = 6

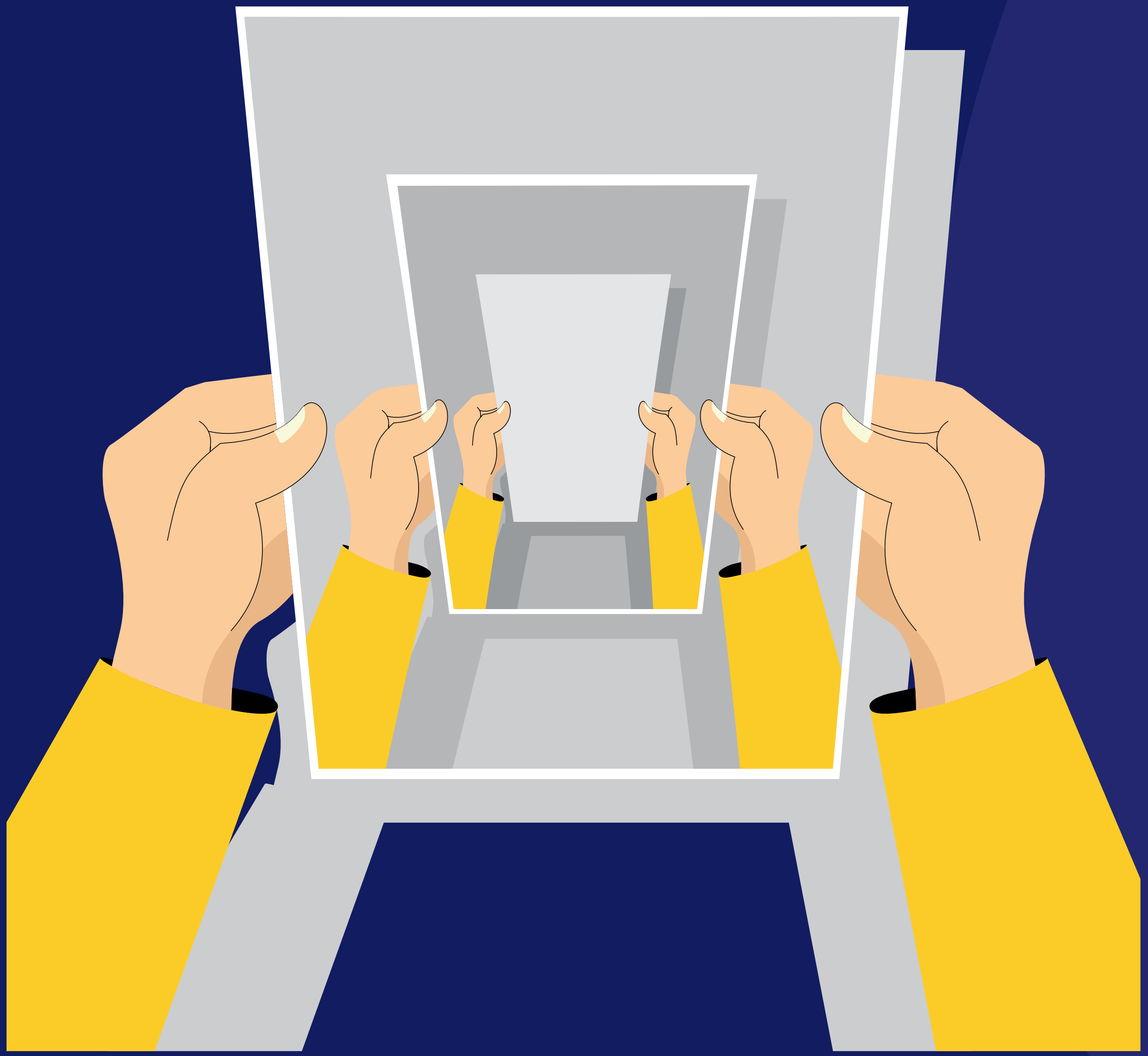
23CS046

THERE ARE MAINLY FOUR WAYS TO SOLVE THIS PROBLEM

- Brute Force (Recursion) – Tries all possible + and - combinations.
- Recursion + Memoization – Caches results to avoid recomputation.
- Dynamic Programming (Subset Sum) – Converts problem into a subset sum variation.
- Optimized DP – Reduces space complexity for larger inputs.



23CS046



BRUTE FORCE(RECURSION)

🔹 Concept: Try all + and - combinations using recursion.

🔹 Example (nums = [1, 2, 3], target = 3):

+1 +2 -3 = 0 ❌,

+1 -2 +3 = 2 ❌,

-1 +2 +3 = 4 ❌, ... (Total 2^N cases!)

🔹 Code:

```
if (index == nums.size()) return target == 0 ? 1 : 0;
return countWays(nums, index + 1, target - nums[index]) +
    countWays(nums, index + 1, target + nums[index]);
```

🔹 Time Complexity: $O(2^{N})$ ❌ (Slow for large N)

✅ Good for N ≤ 15, ❌ Use DP for larger inputs!

Recursive + Memoization

Concept:

- Stores results of subproblems to avoid redundant calculations (Top-Down DP).
- Improves efficiency over brute force recursion.

🔹 Optimized Recursive Code (C++):

```
unordered_map<string, int> dp;
int countWays(vector<int>& nums, int i, int target) {
   string key = to_string(i) + "," + to_string(target);
   if (dp.count(key)) return dp[key];
   if (i == nums.size()) return target == 0 ? 1 : 0;
   return dp[key] = countWays(nums, i+1, target - nums[i]) +
                    countWays(nums, i+1, target + nums[i]);
}
```

🔹 Time Complexity: O(N×S) ✅ (Faster than brute force)

✅ Best for 15 < N ≤ 30, ❌ Use DP for even larger inputs!

23CS046

DYNAMIC PROG
(BOTTOM UP)

◆ Concept:

- Uses a 2D DP table to store solutions iteratively.
- Converts problem into a subset sum variation.

◆ Formula:

dp[i][j] = dp[i-1][j] + dp[i-1][j - nums[i-1]]

```
vector<vector<int>> dp(n+1, vector<int>(S+1, 0));
dp[0][0] = 1;
for (int i = 1; i <= n; i++) {
   for (int j = 0; j <= S; j++) {
      dp[i][j] = dp[i-1][j];
      if (j >= nums[i-1]) dp[i][j] += dp[i-1][j - nums[i-1]];
   }
}
```

◆ Time Complexity: $O(N \times S)$ ✅ (Faster for large N)

✅ Best for N > 30, ❌ Uses extra memory

23CS046

Space-Optimized DP Approach

Space-Optimized DP Approach

🔹 Concept:

- Reduces 2D DP table to 1D array (since we only need the previous row).
- Uses rolling array technique to save space.

🔹 Formula:

dp[j] = dp[j] + dp[j - nums[i-1]] (Iterate backward to avoid overwriting).

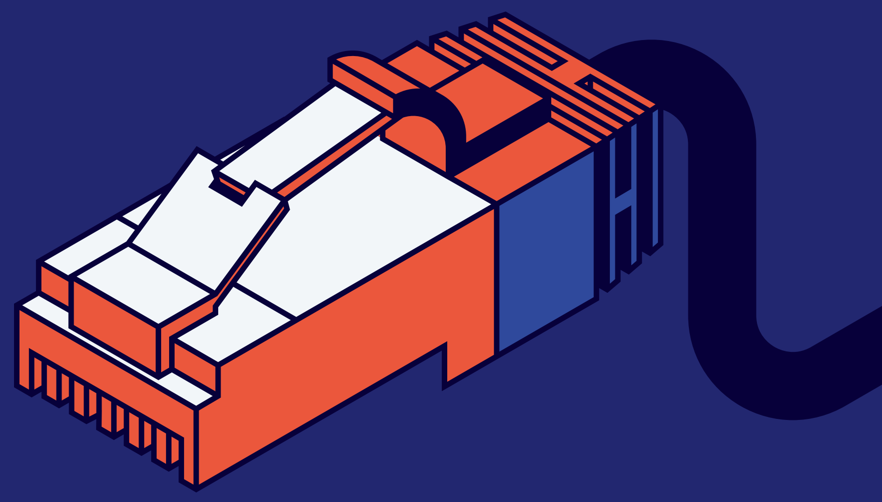
```
vector<int> dp(S+1, 0);
dp[0] = 1;
for (int num : nums) {
  for (int j = S; j >= num; j--) {
    dp[j] += dp[j - num];
  }
}
```

🔹 Time Complexity: $O(N \times S)$ ✅ (Same as DP)

🔹 Space Complexity: $O(S)$ ✅ (Much better than 2D DP)

✅ Best for N > 30, ❌ Still not optimal for very large S

23CS046



Optimization Techniques

Brute Force 🛠️

📌 When to Use?
✔ Small N (≤ 20).
❌ Inefficient for large N.

Memoization 💾

📌 When to Use?
✔ Medium-sized N (≤ 30).
✔ Saves time compared to brute force.

DP ⚡

📌 When to Use?
✔ Large N (≤ 1000).
✔ Best for efficiency but requires extra space.

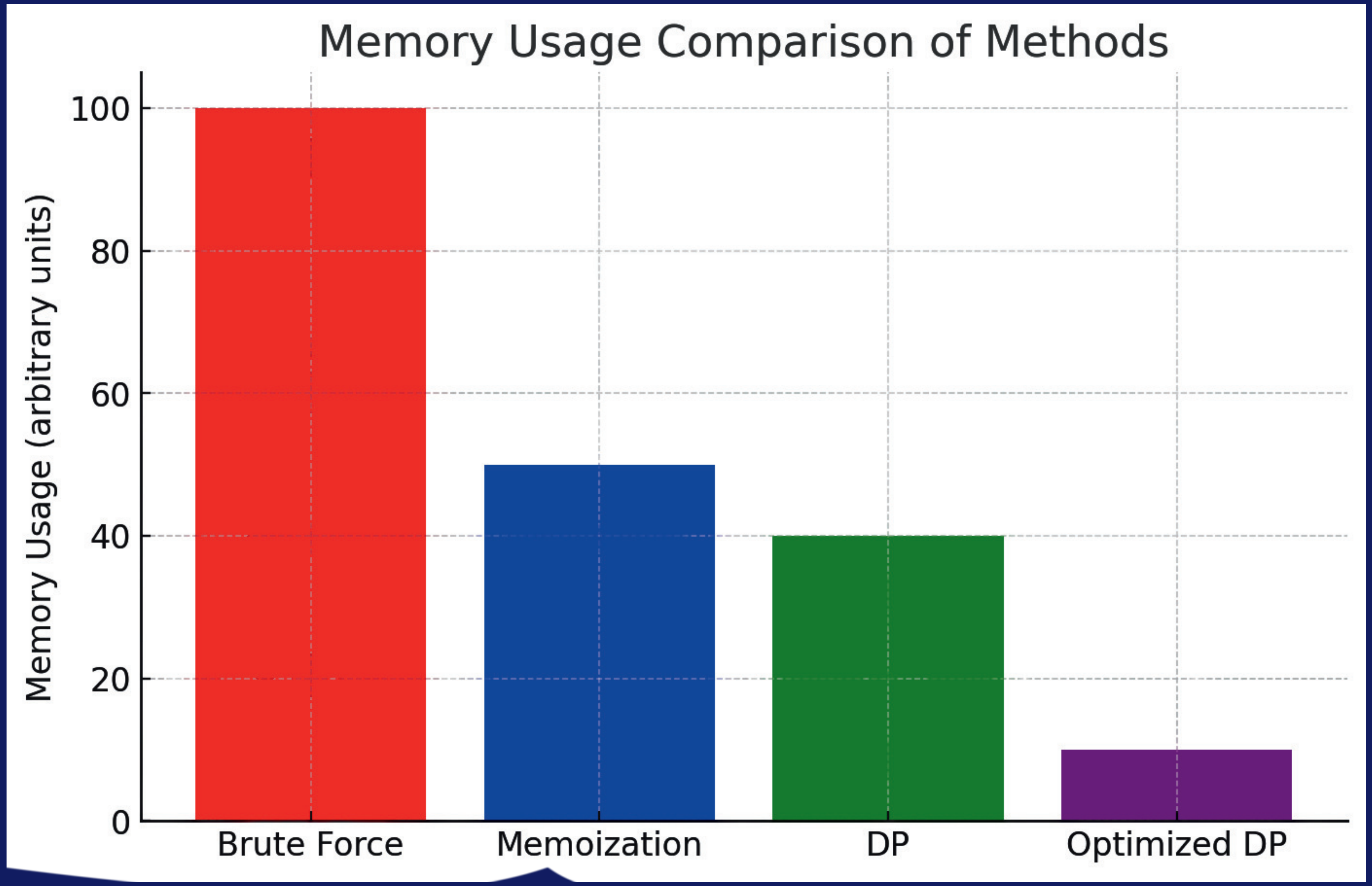
C++ IMPLEMENTATION

Explaination of Code

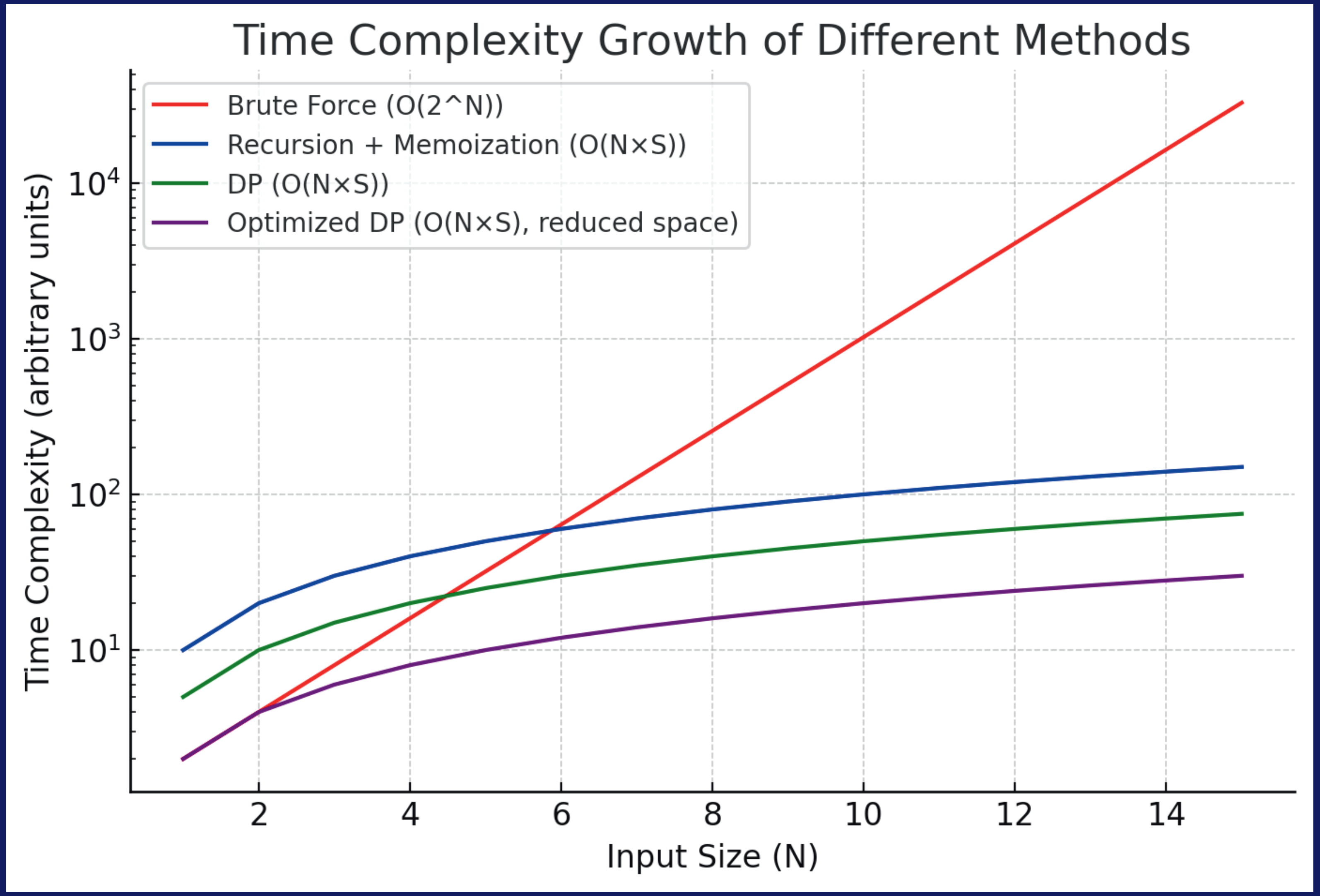
KEY TAKEAWAY

- For small inputs → Use recursion.
- For medium inputs → Use recursion with memoization.
- For large inputs → Use Dynamic Programming.
- Optimized DP is the best for handling large data.

Space Complexity



Time Complexity



THANK YOU

KATHAN MODH

23CS046