A

CIA REPORT

ON

COIN CHANGE USING DYNAMIC PROGRAMMING

Submitted by,

74 Chandar Pratiksha

75 Navale Pragati

76 Parjane Krutika

77 Phopse Akshada

(TY BTECH COMPUTER)

DIVISION
A

Guided by,
Prof. P. B. Dhanwate

Subject- Design and Analysis of Algorithms

In Academic Year 2024-25

Department of Computer Engineering
Sanjivani College of Engineering,
Kopargaon

Sanjivani College of Engineering, Kopargaon

CERTIFICATE

This is to certify that

74 Chandar Pratiksha

75 Navale Pragati

76 Parjane Krutika

77 Phopse Akshada

(T.Y. Computer)

Successfully completed their CIA Report on

COIN CHANGE USING DYNAMIC PROGRAMMING

Towards the partial fulfilment of

Bachelor's Degree in Computer Engineering

During the academic year 2024-25

Prof. P. B. Dhanwate
[Guide]

Dr. D. B. KSHIRSAGAR
[H.O.D. Comp Engg]

Dr. A. G. THAKUR

[Director]

CONTENTS

1. Introduction

The Coin Change Problem is a classic combinatorial optimization problem with applications in various domains such as finance, gaming, and resource allocation. The challenge lies in determining how many unique ways a given sum can be formed using an infinite supply of a set of coin denominations. This problem demonstrates the power of dynamic programming, which systematically breaks down the problem into smaller, manageable sub-problems and stores intermediate results for reuse. The paper highlights the importance of efficient algorithms in solving such problems and explores the dynamic programming approach to achieve optimal solutions.

Dynamic programming is chosen because of the inherent properties of the problem, such as **optimal substructure** and **overlapping sub-problems**. Through this approach, we illustrate how to build a solution incrementally and efficiently. This paper also explores the time and space complexities associated with the algorithm, presenting solved examples and a practical implementation to understand its utility better.

2. Problem Statement & Description

Problem Statement:

Given:

- A target amount sum to be formed.
- An array coins[] containing denominations of coins, with an infinite supply of each denomination.

The objective is to determine the total number of ways to form the exact sum using the available denominations, with the constraint that the order of coins does not matter.

Description:

The problem is a combinatorial challenge where permutations of the same combination are considered equivalent. For instance, for sum = 4 and coins[] = {1, 2, 3}, combinations {1, 3} and {3, 1} are treated as the same.

This problem is significant in scenarios like:

- **Financial Systems**: Determining the number of ways to provide change for a transaction.
- **Gaming**: Computing ways to achieve specific scores or outcomes.
- **Optimization Problems**: Resource allocation with constraints.

Example

Input: sum = 4, coins[] = {1, 2, 3}
Output: 4
Explanation: The possible combinations are:

1. {1, 1, 1, 1}
2. {1, 1, 2}
3. {2, 2}
4. {1, 3}

3. Problem Prerequisite

Input:

- Coin denominations: `[1, 2, 3]`
- Target sum: `4`

Output:

We need to determine how many ways we can achieve the sum of `4` using the given denominations.

To understand and solve the problem effectively, consider:

Constraints: The coin values must be positive integers. The target sum must be a non-negative integer.

Overlapping Subproblems: Many subproblems share results. For example, finding the number of ways to make sum - 2 will likely be needed multiple times for different coins.

Optimal Substructure: The problem can be broken into smaller subproblems whose solutions can be combined to solve the original problem efficiently.

4. Requirement Analysis of Problem

1. **Dynamic Programming Approach**
    a. Use a DP array to store the minimum coins required for each amount from 0 to the target amount.
    b. Transition formula: $dp[j]=\min(dp[j],dp[j-coin]+1)$
    c. Complexity: $O(n \times m)$, where n = target amount, m= number of denominations.

2. **Recursive Approach with Tabulation**
    a. Explore all possible combinations using recursion and store already-computed results to avoid redundant calculations.
    b. Complexity: Exponential without memoization, reduced to $O(n \times m)$ with it.

3. **Breadth-First Search (BFS)**
    a. Treat the problem as a graph traversal, where each state represents a current amount.
    b. Complexity: $O(n \times m)$

4. **Tools**
    a. IDEs: PyCharm, Eclipse, Visual Studio, IntelliJ IDEA.
    b. Compilers: GCC, Java Runtime.
    c. Debugging Tools: Visual Debuggers in IDEs.

5. Solved example of given application

1. **Initialization:**

- We create a dp[] array of size sum + 1 (i.e., dp[0] to dp[4]).
- Initialize dp[0] = 1 because there is one way to make sum 0 (using no coins).
- All other values in dp[] will be initially set to 0.

Initial DP Array: dp = [1, 0, 0, 0, 0]

2. **Process Coin 1:**

    - For each sum from 1 to 4, we update dp[j] by adding dp[j - 1] (the number of ways to make the sum j - 1).

DP Update after Coin 1:

- For j = 1: dp[1] += dp[1 - 1] = dp[0] = 1
- For j = 2: dp[2] += dp[2 - 1] = dp[1] = 1
- For j = 3: dp[3] += dp[3 - 1] = dp[2] = 1
- For j = 4: dp[4] += dp[4 - 1] = dp[3] = 1

DP Array after Coin 1: dp = [1, 1, 1, 1, 1]

3. **Process Coin 2:**

- For each sum from 2 to 4, we update dp[j] by adding dp[j - 2] (the number of ways to make the sum j - 2).

DP Update after Coin 2:

- For j = 2: dp[2] += dp[2 - 2] = dp[0] = 1
- For j = 3: dp[3] += dp[3 - 2] = dp[1] = 1
- For j = 4: dp[4] += dp[4 - 2] = dp[2] = 2

DP Array after Coin 2: dp = [1, 1, 2, 2, 3]

4. **Process Coin 3:**
    a. For each sum from 3 to 4, we update dp[j] by adding dp[j - 3] (the number of ways to make the sum j - 3).

DP Update after Coin 3:

    b. For j = 3: dp[3] += dp[3 - 3] = dp[0] = 1
    c. For j = 4: dp[4] += dp[4 - 3] = dp[1] = 1

DP Array after Coin 3: dp = [1, 1, 2, 3, 4]

5. **Final Output:**
    a. The value dp[4] (i.e., dp[4] = 4) represents the total number of ways to make the sum 4 using the coins {1, 2, 3}.

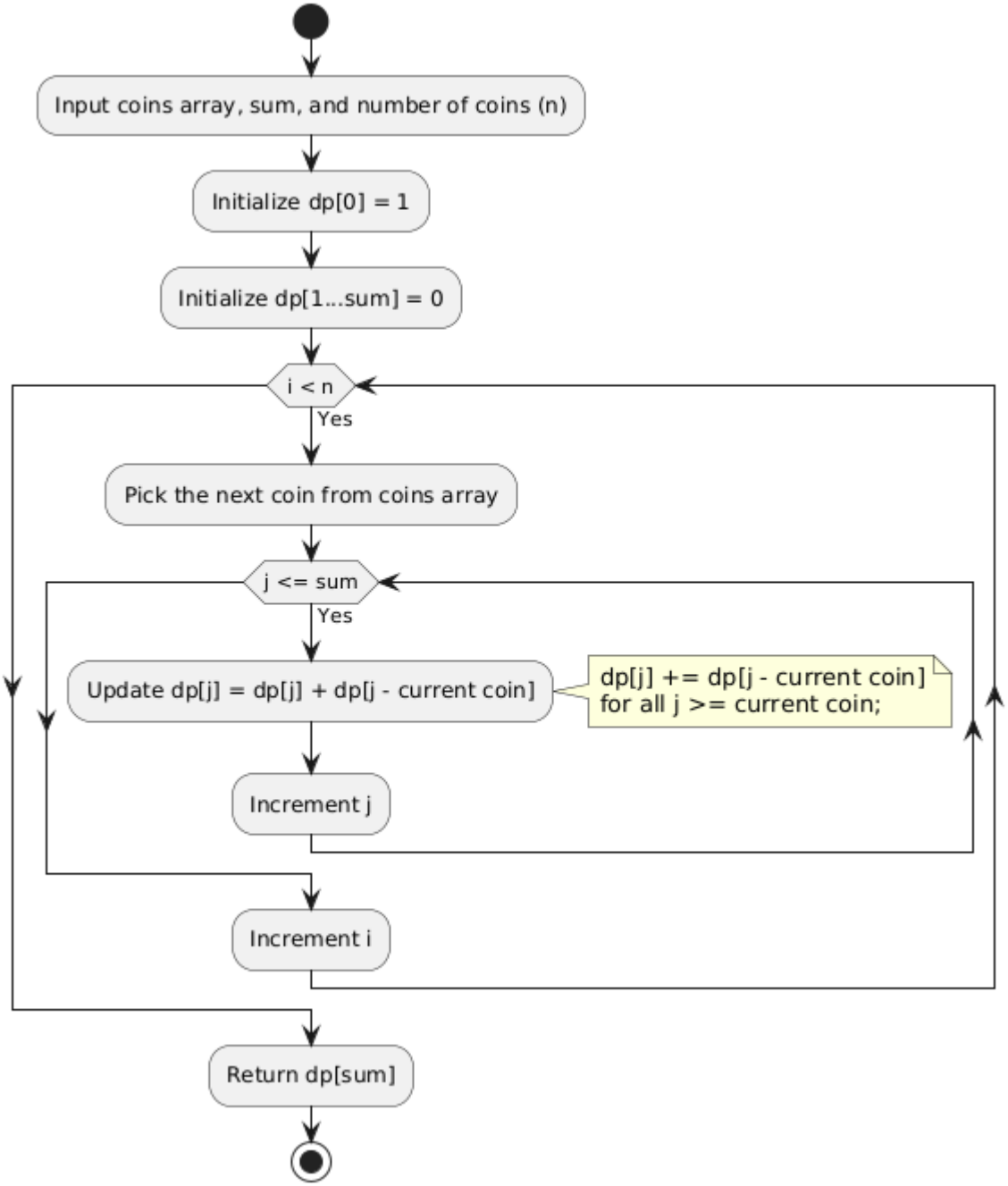
The number of ways to make the sum 4 using coins {1, 2, 3} is **4**. These ways are:

1. 1 + 1 + 1 + 1
2. 1 + 1 + 2
3. 2 + 2
4. 1 + 3

Thus, the final answer is 4.

6. Control abstraction of the Algorithm:

Flowchart for the problem statement:



7. Explanation of algorithmic steps or code of entire program

Algorithm for the considered problem statement is as follows

The problem is solved using Dynamic Programming. The approach involves maintaining a dp array where dp[i] represents the number of ways to make the sum i using the given coins.

Steps:

1. **Initialization**:

- Create an array dp of size sum + 1.
- Set dp[0] = 1 because there is exactly one way to make the sum 0 (by choosing no coins).
- Set all other dp[i] values to 0 because, initially, there are no ways to make other sums.

2. **Processing Each Coin**:

- For each coin in the given array, update the dp array to include the number of ways to make sums using this coin.

3. **Inner Loop for Updating Sums**:

- For a given coin coin, iterate through the dp array starting from index coin to sum.
- Update dp[j] as dp[j] += dp[j - coin]. This means:
    - The number of ways to make the sum j is increased by the number of ways to make the sum j - coin.
    - This is because adding the coin to every way of making j - coin creates a new way of making j.

4. **Final Answer**:

- After processing all coins, the value at dp[sum] contains the total number of ways to make the target sum.

8. Implementation

Program Code:

```
#include <iostream>
using namespace std;

// Function to calculate the number of ways to make the target sum
int countWays(int coins[], int n, int sum) {
    int dp[sum + 1]; // Create a dp array to store the solutions to subproblems
    dp[0] = 1;       // There's 1 way to make the sum 0 (using no coins)

    // Initialize all other values in the dp array as 0
    for (int i = 1; i <= sum; i++) {
        dp[i] = 0;
    }

    // Iterate through all coins
    for (int i = 0; i < n; i++) {
        // Update the dp array for sums from coins[i] to sum
        for (int j = coins[i]; j <= sum; j++) {
            dp[j] += dp[j - coins[i]];
        }
    }

    return dp[sum]; // Return the total number of ways to make the sum
}

int main() {
    // Test Case 1
    int coins1[] = {1, 2, 3};
    int sum1 = 4;
    int n1 = sizeof(coins1) / sizeof(coins1[0]);
    cout << "Number of ways to make sum " << sum1 << " is " << countWays(coins1, n1, sum1) <<
endl;

    // Test Case 2
    int coins2[] = {2, 5, 3, 6};
    int sum2 = 10;
    int n2 = sizeof(coins2) / sizeof(coins2[0]);
    cout << "Number of ways to make sum " << sum2 << " is " << countWays(coins2, n2, sum2) <<
endl;

    return 0;
}
```

Output:

```
PS C:\Users\lenovo> cd "c:\Users\lenovo\
ile }
Number of ways to make sum 4 is 4
Number of ways to make sum 10 is 5
```

9. Time and space complexity of problem statement

Time Complexity:

In the solution, we use dynamic programming to calculate the number of ways to make a given sum. Here's a step-by-step breakdown of the time complexity:

1. **Initialization of the dp array:**

```
dp[0] = 1;
for (int i = 1; i <= sum; i++) {
   dp[i] = 0;
}
```

   - **Loop runs for sum times** (since i ranges from 1 to sum).
   - Time complexity: **O(sum)**.

2. **Nested Loops to Fill the dp Array:**

```
for (int i = 0; i < n; i++) {
   for (int j = coins[i]; j <= sum; j++) {
      dp[j] += dp[j - coins[i]];
   }
}
```

   - The outer loop runs **n times** where n is the number of available coins.
   - The inner loop runs **sum - coins[i] + 1 times** for each coin. In the worst case, this could be up to sum times (if a coin has a value of 1).
   - Therefore, in the worst case, for every coin, the inner loop runs **sum times**.

Thus, the total time complexity is the number of iterations in the nested loops:

- **Outer loop**: runs n times (for each coin).
- **Inner loop**: runs up to sum times for each coin.

Therefore, the total time complexity is: $O(n \times sum)$

Where n is the number of coins and sum is the target sum we want to achieve.

Space Complexity:

The space complexity is determined by the amount of extra space (memory) used by the algorithm in addition to the input.

1. **dp Array:**
    - The dp array has a size of **sum + 1** (because we need to store values for all sums from 0 to sum).
    - So, the space used by the dp array is **O(sum)**.
2. **Other Variables:**
    - Other variables (i, j, coins[], and n) require constant space.
    - So, they contribute **O(1)** space complexity.

Thus, the total space complexity is determined by the size of the dp array: O(sum)O

10. Conclusion

The Coin Change Problem is a classic dynamic programming challenge that efficiently computes the number of ways to form a given sum using a set of coin denominations with an infinite supply. Through dynamic programming, the problem is broken down into manageable subproblems, with solutions stored in a table to avoid redundant calculations. The algorithm's time complexity is $O(n \times sum)$, where n is the number of coins and sum is the target amount, while its space complexity is O(sum) due to the storage of intermediate results in the dp array.

This approach optimizes the computation, making it suitable for practical applications in finance, gaming, and resource allocation where determining the number of combinations of values is essential. The problem illustrates the power of dynamic programming in solving combinatorial optimization problems efficiently.

11. References

1. https://www.geeksforgeeks.org/coin-change-dp-7/
2. https://leetcode.com/problems/coin-change/description/
3. https://stackoverflow.com/questions/25773103/how-to-solve-coin-change-task-when-order-does-matters