

# الجامعة العربية الأمريكية ARAB AMERICAN UNIVERSITY



*Algorithms Analysis and Design*

WORKED BY :-

SAMA SAMMAR 202110795

DANA SAMMAR 202111117

AYAH ZAYED 202111486

## Table of Contents

3	.....	<b>Coin Change Problem</b>
3	.....	<b>Introduction =&gt;</b>
3	.....	<b>Background=&gt;</b>
3	.....	<b>Coin Change Problem:</b>
3	.....	<b>Problem Definition:</b>
3	.....	<b>Significance of Dynamic Programming:</b>
3	.....	<b>Goals of the Project:</b>
4	.....	<b>Table for Explain how to Find the minimum # of Coins=&gt;</b>
4	.....	<b>The Tree that explain the sub problems</b>
5	.....	<b>Implementatation=&gt;</b>
6	.....	<b>Test for the example=&gt;</b>
6	.....	<b>Some details about structures and algorithms using in the code=&gt;</b>
9	.....	<b>How the code run to find the result =&gt;</b>
10	.....	<b>Time Complexity=&gt;</b>
10	.....	<b>Conclusion =&gt;</b>

# Coin Change Problem

## Introduction =>

Welcome to the Coin Change Problem Solver project! This project aims to implement a solution to the classic coin change problem using dynamic programming.

## Background=>

**Coin Change Problem:** The coin change problem is a classic algorithmic problem in the field of computer science and mathematics. It involves finding the minimum number of coins needed to make up a given amount of money, using a given set of coin denominations. The problem is often encountered in real-world scenarios such as making change at a cashier or vending machine.

**Problem Definition:** Given a set of coin denominations  $d_1, d_2, \dots, d_n$  and a target amount  $A$ , the goal is to find the minimum number of coins needed to make up  $A$ . Each coin denomination  $d_i$  can be used an unlimited number of times.

**Significance of Dynamic Programming:** Dynamic programming is a powerful technique for solving optimization problems like the coin change problem. It involves breaking down a complex problem into simpler subproblems and solving each subproblem only once, storing the solutions to subproblems in a table to avoid redundant calculations. This approach can significantly improve the efficiency of algorithms by eliminating unnecessary recomputation.

**Goals of the Project:** The primary goal of this project is to implement a dynamic programming solution to the coin change problem, leveraging the efficiency and effectiveness of dynamic programming to solve. The project aims to provide a reliable and efficient solution that can handle various coin denominations and target amounts, with a focus on correctness, scalability, and usability.

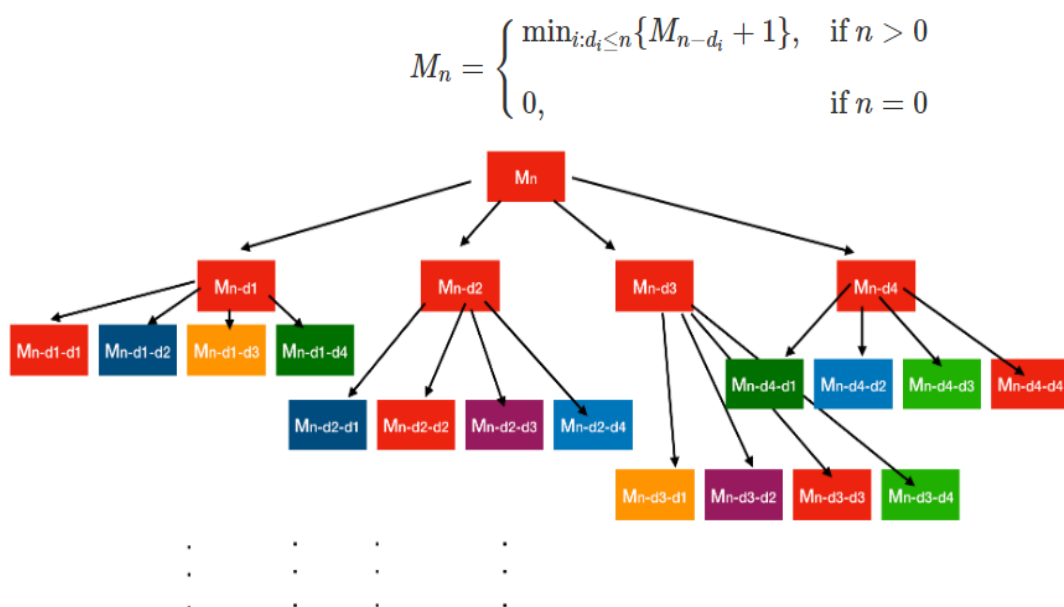
Table for Explain how to Find the minimum # of Coins=>

• Coins = {1, 5, 6, 9} and w=10.

	0	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9	10
5	0	1	2	3	4	1	2	3	4	5	2
6	0	1	2	3	4	1	1	2	3	4	2
9	0	1	2	3	4	1	1	2	3	1	2

If coin > w, then copy value from above

Minimum number of coins



The Tree that explain the sub problems



DP ingredients :

- 1-Optimal Substructure
- 2-Overlapping Subproblems

## Implementation=>

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

int coinChange(vector<int>& coins, int numCoins, int amount) {
    vector<int> dp(amount + 1, INT_MAX); // Initialize dp table
    with INT_MAX and size = amount+1
    dp[0] = 0; // Base case: It takes 0 coins to make up the amount
    0

    for (int i = 1; i <= amount; ++i) {
        for (int j = 0; j < numCoins; ++j) {
            int coin = coins[j];
            if (coin <= i && dp[i - coin] != INT_MAX) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }

    return dp[amount] == INT_MAX ? -1 : dp[amount];
}

int main() {
    int numCoins;
    cout << "Enter the number of coins: ";
    cin >> numCoins;

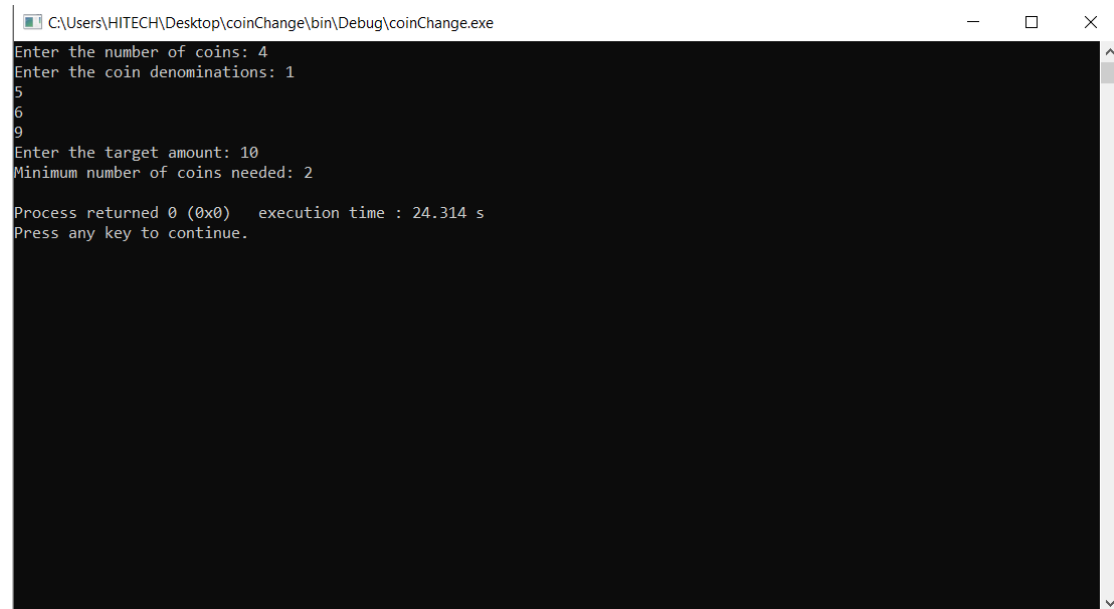
    vector<int> coins(numCoins);
    cout << "Enter the coin denominations: ";
    for (int i = 0; i < numCoins; ++i) {
        cin >> coins[i];
    }

    int amount;
    cout << "Enter the target amount: ";
    cin >> amount;

    cout << "Minimum number of coins needed: " << coinChange(coins,
numCoins, amount) << endl;

    return 0;
}
```

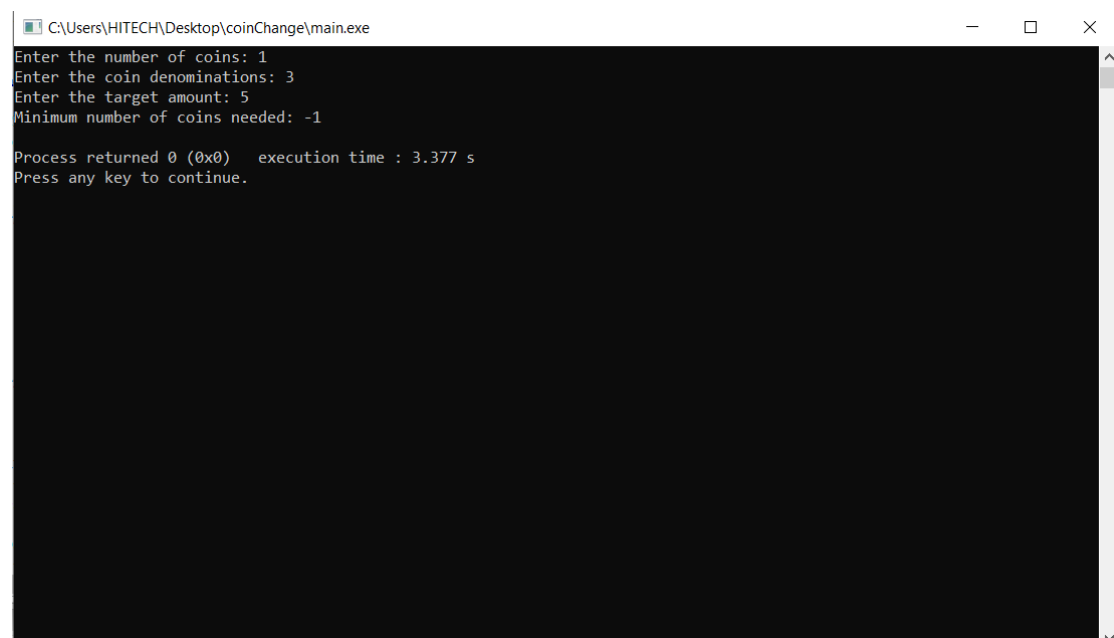
## *Test for the example=>*



```
C:\Users\HITECH\Desktop\coinChange\bin\Debug\coinChange.exe
Enter the number of coins: 4
Enter the coin denominations: 1
5
6
9
Enter the target amount: 10
Minimum number of coins needed: 2

Process returned 0 (0x0)   execution time : 24.314 s
Press any key to continue.
```

This another example when the process not succeed :



```
C:\Users\HITECH\Desktop\coinChange\main.exe
Enter the number of coins: 1
Enter the coin denominations: 3
5
Enter the target amount: 5
Minimum number of coins needed: -1

Process returned 0 (0x0)   execution time : 3.377 s
Press any key to continue.
```

## *Some details about structures and algorithms using in the code=>*

-> Firstly, the coin change problem exhibits the characteristics of an **optimization problem** because it involves finding the optimal solution (best solution), subject to constraints on available coin denominations and the target amount.

-> The core design technique used in this code is **dynamic programming**.

-> We used data structures like **vector** because vectors are dynamic arrays that can grow and shrink in size dynamically, making them suitable for storing a variable number of elements, In the code, **the vector "coins" stores the coin denominations input by the user.**

-> **A dynamic programming array "dp" is used to store the minimum number of coins needed to make up each amount from 0 to the target amount "amount"** (The dynamic programming array is implemented as a vector) , the **"dp"** is initialized with all elements set to **"INT\_MAX"** except for `dp[0]` which is set to **0( Base Case )**.

The `limits` library provides constants for specifying the limits of various data types, We use it here to get the maximum value representable by an integer **"INT\_MAX"** to indicate that they are unreachable by default and provides a clear starting point for the DP process.

### **Why we use passing by reference of the vector "coins" ?**

This to avoid making a copy of the vector when passing it to the **"coinchange"** function. **This can be more efficient, especially if the vector is large, as it avoids the overhead of copying the entire vector.**

-> **Nested loop Structure** :This allows us to consider all possible combinations of coin denominations and amounts to determine the minimum number of coins needed for each amount.

-> **Recurrence relation:** For each amount  $i$ , the algorithm considers all coin denominations  $coin$  and updates  $dp[i]$  by taking the minimum of its current value and  $dp[i-coin]+1$ , which represents the number of coins needed to make up the amount  $i-coin$ , plus one additional coin (the current coin denomination).

-> For each amount  $i$  and coin denomination  $coin$ , if  $coin$  is less than or equal to  $i$  and the amount  $i-coin$  is reachable (i.e.,  $dp[i - coin] \neq INT\_MAX$ ), we update  $dp[i]$  by taking the minimum of its current value and  $dp[i-coin]+1$ , which represents the number of coins needed to make up the amount  $i-coin$ , plus one additional coin (the current coin denomination).

-> Finally, we return  $dp[amount]$ , which represents the minimum number of coins needed to make up the target amount. If it's not possible to make up the amount, we return  $-1$ .



## How the code run to find the result =>

```
dp = [0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX]
```

Let's see how dp is updated:

For amount 1:  $dp[1] = \min(INT\_MAX, dp[1 - 1] + 1) = \min(INT\_MAX, dp[0] + 1) = \min(0, 1) = 0$

For amount 2:  $dp[2] = \min(INT\_MAX, dp[2 - 1] + 1) = \min(INT\_MAX, dp[1] + 1) = \min(0, 1 + 1) = 0$

For amount 3:  $dp[3] = \min(INT\_MAX, dp[3 - 1] + 1) = \min(INT\_MAX, dp[2] + 1) = \min(0, INT\_MAX) = 0$

For amount 4:  $dp[4] = \min(INT\_MAX, dp[4 - 1] + 1) = \min(INT\_MAX, dp[3] + 1) = \min(0, 0 + 1) = 0$

For amount 5:  $dp[5] = \min(INT\_MAX, dp[5 - 1] + 1, dp[5 - 5] + 1) = \min(INT\_MAX, dp[4] + 1, dp[0] + 1) = \min(0, 0 + 1, 0 + 1) = 0$

For amount 6:  $dp[6] = \min(INT\_MAX, dp[6 - 1] + 1, dp[6 - 5] + 1, dp[6 - 6] + 1) = \min(INT\_MAX, dp[5] + 1, dp[1] + 1, dp[0] + 1) = \min(0, 0 + 1, INT\_MAX, 0 + 1) = 0$

For amount 7:  $dp[7] = \min(INT\_MAX, dp[7 - 1] + 1, dp[7 - 5] + 1, dp[7 - 6] + 1) = \min(INT\_MAX, dp[6] + 1, dp[2] + 1, dp[1] + 1) = \min(0, 0 + 1, INT\_MAX, 0 + 1) = 1$

For amount 8:  $dp[8] = \min(INT\_MAX, dp[8 - 1] + 1, dp[8 - 5] + 1, dp[8 - 6] + 1) = \min(INT\_MAX, dp[7] + 1, dp[3] + 1, dp[2] + 1) = \min(1, 1 + 1, INT\_MAX, 0 + 1) = 1$

For amount 9:  $dp[9] = \min(INT\_MAX, dp[9 - 1] + 1, dp[9 - 5] + 1, dp[9 - 6] + 1) = \min(INT\_MAX, dp[8] + 1, dp[4] + 1, dp[3] + 1) = \min(1, 1 + 1, 0 + 1, INT\_MAX) = 2$

For amount 10:  $dp[10] = \min(INT\_MAX, dp[10 - 1] + 1, dp[10 - 5] + 1, dp[10 - 6] + 1, dp[10 - 9] + 1) = \min(INT\_MAX, dp[9] + 1, dp[5] + 1, dp[4] + 1, dp[1] + 1) = \min(2, 2 + 1, 0 + 1, 0 + 1, 0 + 1) = 2$

After updating dp, the value of dp[10] is 2, indicating that the minimum number of coins needed to make up the amount 10 is 2.

## Time Complexity=>

	Cost	Time
<code>int coinChange(vector&lt;int&gt;&amp; coins, int numCoins, int amount) {</code>	C1	m
<code>1 vector&lt;int&gt; dp(amount + 1, INT_MAX);</code>	C2	1
<code>2 dp[0] = 0;</code>	C3	m+1
<code>3 for (int i = 1; i &lt;= amount; ++i) {</code>	C4	m*(n+1)
<code>4     for (int j = 0; j &lt; numCoins; ++j) {</code>	C5	m*n
<code>5         int coin = coins[j];</code>	C6	m*n
<code>6         if (coin &lt;= i &amp;&amp; dp[i - coin] != INT_MAX) {</code>	C7	m*n
<code>7             dp[i] = min(dp[i], dp[i - coin] + 1);</code>	C8	1
<code>             }</code>		
<code>         }</code>		
<code>8 return dp[amount] == INT_MAX ? -1 : dp[amount];</code>		
<code>}</code>		

Where n is the number of coin denominations and m is the target amount.

To find the total time complexity :-

$$C1(m) + C2(1) + C3(m+1) + C4(m*(n+1)) + C5(m*n) + C6(m*n) + C7(m*n) + C8(1)$$

$$C1m + C2 + C3m + C3 + C4(m*n) + C4m + C5(m*n) + C6(m*n) + C7(m*n) + C8$$

$$C2 + C3 + C8 + m(C1 + C3 + C4) + (m*n)(C4 + C5 + C6 + C7)$$

$$C + Am + B(m*n) \Rightarrow T(n) = Am + B(m*n) + C \quad \text{where A, B and C are constants}$$

**So, the time complexity of this code is  $O(n*m)$ .**

This algorithm is the most efficient and commonly used approaches for this problem.

Also, the **space complexity** of this solution is  **$O(m)$**  due to the dynamic programming array, because the space complexity of the input vector coin is  $O(n)$  but  $O(m) + O(n) = O(\max(m, n))$ .

## Conclusion =>

In conclusion, the coin change problem serves as an excellent example of how dynamic programming can be leveraged to efficiently solve optimization problems. By applying dynamic programming principles, we can devise algorithms that provide optimal solutions while mitigating computational complexities. This project not only deepened our understanding of dynamic programming but also equipped us with valuable problem-solving skills essential for tackling complex algorithmic challenges.