#### MINIMUM COIN CHANGE PROBLEM

#### REPORT

***Submitted by***

###### AASTIK SHARMA (RA2011003010718)

###### PRERNA SHARMA (RA2011031010086)

###### ROHAN KISHOR SHINDE (RA2011031010122)

***Under the Guidance of***

#### Dr. N. Arunachalam

###### (Assistant Professor Department of Computing Technologies)

***in partial fulfilment of the requirement for the IV semester for subject***

**DESIGN AND ANALYSIS OF ALGORITHM**

of

###### BACHELOR OF TECHNOLOGY

###### in

###### COMPUTER SCIENCE ENGINEERING



###### DEPARTMENT OF NETWORKING AND COMMUNICATIONS

###### COLLEGE OF ENGINEERING AND TECHNOLOGY

###### SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

**KATTANKULATHUR- 603 203**

###### SESSION- 2021-22

**ACKNOWLEDGMENTS**

We sincerely thank Dr. N. Arunachalam, Assistant Professor, Department of Computing Technologies, who have been the greatest inspiration and have provided sufficient background knowledge and understanding of this subject.

Our humble prostration goes to him, for providing all the necessary resources and environment, which have aided us to complete this project successfully.

**CONTRIBUTIONS**

|  |  |
| --- | --- |
| CONTRIBUTION | MEMBER |
| Design Techniques, collection of references and conclusion of project | Aastik Sharma |
| Problem statement selection, definition and explanation, final review of project | Prerna Sharma |
| Algorithm Design and Time complexity Analysis, final documentation of project | Rohan Kishor Shinde |

**ABSTRACT**

The classical problem “Coin change” in Computer Science has become a key problem to a number of subsequent problems in different areas: finance, algorithm study, sports, etc. Mathematicians have been paying attention to only two possible outcomes of the problem: the most time/resource efficient solution and the total number of solutions. However, solutions among the “normal solutions” can be beneficial in certain situations, if carefully considered with math and economic phenomena in the past. Our work describes some of such possible beneficial solutions that are worth paying attention to and its application in finance and fiscal policy.

**CONTENTS**

**Introduction**

**1. Problem Definition**  **1**

**2. Problem Explanation**  **2**

2.1 Problem 2

2.2 Diagram and Example 3

**3. Design Techniques** **4**

3.1 Brute Force Approach using Recursion 4

3.2 Efficient Approach using Dynamic Programming 6

**4. Algorithm**  **8**

4.1 Explanation 8

4.1.1 Code 9

4.2 Complexity Analysis 10

**5. Conclusions 11**

**6. References** **12**

**Introduction**

1. **Problem Definition**

The name of the problem is **Minimum Coin Change** :

If we want to make change for a given value (N) of cents, and we have an infinite supply of each of C = { C1 , C2 , C3 , … , Cn } valued coins, what is the minimum number of coins required to make the change?

In this problem, we will consider a set of different coins C{1, 2, 5, 10} are given, There is an infinite number of coins of each type. To make change the requested value we will try to take the minimum number of coins of any type.

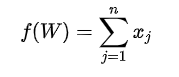
As an example, for value 22 − we will choose {10, 10, 2}, 3 coins as the minimum.

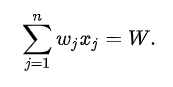
It is also the most common variation of the coin change problem, a general case of partition in which, given the available denominations of an infinite set of coins, the objective is to find out the number of possible ways of making a change for a specific amount of money, without considering the order of the coins.

It is weakly NP-hard, but may be solved optimally in pseudo-polynomial time by dynamic programming.

**2. Problem Explanation**

The change-making problem addresses the question of finding the minimum number of coins (of certain denominations) that add up to a given amount of money. It is a special case of the integer knapsack problem, and has applications wider than just currency.



****

W is the total amount and a set of non-negative (positive or zero) integers {x1, x2, ..., xn}, with each xj representing how often the coin with value wj is used, which minimize the total number of coins f(W).

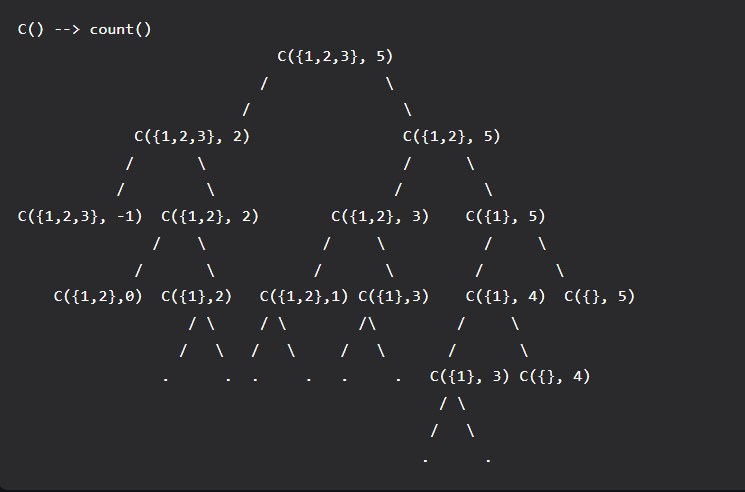
We are given an amount W representing a total amount of money and an integer array coin[] of size n representing coins of different denominations.

Find the minimum number of coins required to make the change.

We assume that we have an infinite supply of each kind of coin with the value of coin[0] to coin[n-1]. If any combination of coins cannot make up amount W of money , we return -1.

**2.2 Diagram and Example**

**1.**

****

For example , the following picture is the recursion tree diagram for W=5 and coins of the given values [1,2,3] . The Sub-problem of size 3 is coming 2 times , the sub-problem of size 2 is coming 4 times , the sub-problem of size 1 is coming 7 times.

The required solution is {2,3} which gives us our required sum 5.

**3. Design Techniques**

There are 2 approaches to the problem statement’s solution discussed in this project report :

1. Brute Force (recursion)
2. Efficient (Dynamic Programming)

3.1 **Brute Force Approach using Recursion**

This is an optimization problem because there can be several ways to provide change, but we need to return the change using the minimum number of coins. In different words :  solution space is huge and only a few solutions will provide the optimal solution. So one basic idea would be to explore all solutions or possibilities of the change and return the count of the solution with a minimum number of coins.

In such a situation, when we need to explore all possibilities, we can think about solving the problem recursively, i.e., the solution of the problem using smaller sub-problems.

Initially available choices are important for building the recursive solution. Here we have ‘m’ choices of the coin in the start i.e. we can pick any coin among m coins.

Suppose **minCoin(coin[], m, K)** returns the minimum number of coins required to make a change of value K (This is our larger problem). If we select any coin[i] first, then the smaller sub-problem is **minCoin(coin[], m, K  - coin[i])**i.e. the minimum number of coins required to make a change of amount K -  coin[i].

So for i = 0 to m-1, whichever choice provides the change using the minimum number of coins, we shall add 1 and return the value. But before selecting any coin, make sure whether the value of the coin is less than equal to the amount needed i.e. **coin[i] <= K**.

**Recursive structure:**

minCoin(coin[], m, K) = min (for i = 0 to m-1) { 1 + minCoin(coin[], m, K - coin[i]) }

Where coin[i] <= K Base case: If K == 0, then 0 coins required.

**PSEUDOCODE:**

*int count( int S[], int m, int n )*

*{*

*// If n is 0 then there is 1 solution (do not include any coin)*

*if (n == 0)*

*return 1;*

*// If n is less than 0 then no solution exists*

*if (n < 0)*

*return 0; // If there are no coins and n is greater than 0, then no solution exist*

*if (m <=0 && n >= 1)*

*return 0; // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]*

*return count( S, m - 1, n ) + count( S, m, n-S[m-1] );*

*}*

Time Complexity : O(2^n) – Exponential

3.2 **Efficient Approach using Dynamic Programming**

Solution Idea and Steps

Since we have identified that it is a dynamic programming problem, we can solve it using the bottom-up approach. Our aim here is to calculate the solution of the smaller problems in an iterative way and store their result in a table.

**Table structure**: The state of the smaller sub-problems depends on the one variable **K** because it decreases after each recursive call. So we need to construct a 1-D table to store the solution of the sub-problems.

**Table size**: The size of the 1-D table is equal to the total different subproblems. According to the recursion tree, there can be a total (K + 1) of different sub-problems i.e sub-problems of size (K, K-1,…2, 1, 0). **int Change[K + 1]**

**Table initialization**: Before building the solution using the iterative structure of the bottom-up approach, we need to initialize the table by the smaller version of the solution i.e base case. **Change[0] = 0**

**Iterative structure to fill the table**: Now, we need to define the iterative structure to fill the table Change[i] i.e a relation by which we build the solution of the larger problem using the solution of smaller problems in a bottom-up manner. We can easily define the iterative structure by using the recursive structure of the recursive solution.

Change[i] = min (for j = 0 to m-1) { 1 + Change[i - coin[j]] }, where coin[j] < K

**Returning final solution**: After filling the table iteratively, our final solution gets stored at the last Index of the array i.e. return **Change[K].**

**PSEUDOCODE:**

*int minCoin(int coin[], int m, int K)*

*{ if(K == 0)*

*return 0;*

*int Change[K + 1];*

*Change[0] = 0;*

*for(int i = 1; i <= K; i = i + 1)*

*Change[i] = INT\_MAX*

*for(int i = 1; i <= K; i = i + 1)*

*{*

*for(int j = 0; j<m; j = j +1)*

*{*

*if(coin[j] <= i)*

*{*

*int currCount = Change[i - coin[j]];*

*if(currCount != INT\_MAX && currCount + 1 < Change[i])*

*Change[i] = currCount + 1;*

*}*

*}*

*} if(Change[K] == INT\_MAX) return -1;*

*else*

*return Change[K];*

*}*

**4. Algorithm – Dynamic Programming Implementation**

The minimum number of coins for a value V can be computed using the below recursive formula.

If V == 0, then 0 coins required.

If V > 0

minCoins(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}

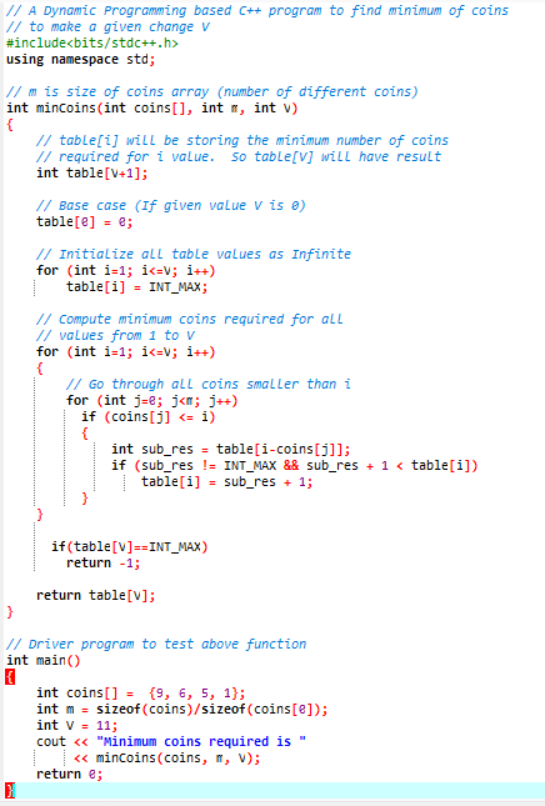
where i varies from 0 to m-1

and coin[i] <= V

The time complexity of the above solution is exponential and space complexity is way greater than O(n). If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from V = 11, we can reach 6 by subtracting one 5 times and by subtracting 5 one time. So the subproblem for 6 is called twice.

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the min coins problem has both properties re-computations of the same subproblems can be avoided by constructing a temporary array table[][] in a bottom-up manner.

**Time complexity: O(mV)**

****

**4.2 Time Complexity Analysis:**

The complexity of solving the coin change problem using recursive time and space will be:

Problems: Overlapping subproblems + Time complexity

O(2^n) is the time complexity, where n is the number of coins

Time and space complexity will be reduced by using dynamic programming to solve the coin change problem :

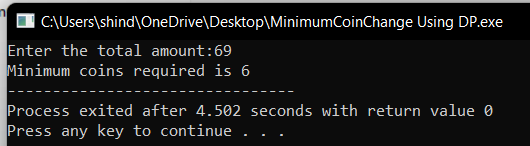
**Time complexity: O(mV):**

O(numberOfCoins\*TotalAmount

**Space complexity:**

O(numberOfCoins\*TotalAmount)

**Sample Output:- coins{1,2,3,5,10,20}**

****

**Conclusions**

Through this project, we were given the chance to understand and implement the minimum coin change problem. We needed to make change for a given value (N) of cents while we had infinite supply of each C={ C1, C2, C3, … Cn} valued coins, we were successfully able to implement an algorithm that provided the minimum number of coins required to make the change. We solved this problem using two different approaches Brute Force which was implemented using recursion and Efficient Force which was implemented using Dynamic Programming. We found that time and space complexity was reduced by using dynamic programming to solve the minimum coin change problem.

This algorithm problem has real life applications like this algorithm can be used to distribute change i.e. In a soda pop vending machine that could accept bills and coins and dispense coins. One example is, buying a 60 cent soda pop with a dollar. This leaves 40 cents change, or in the US, 1 quarter, 1 dime, and 1 nickel for least coin pay. If the nickel tube were empty though, the machine would dispense 4 dimes.

It was a wonderful and learning experience for us while working on this project. This project took us through the various phases of project development and gave us real insight into the world of coding. The joy of work and the thrill involved while tackling the various problems and challenges gave us a feel of developers industry.

**References**

* Google
* Wikipedia
* Geeks For Geeks
* Stack Exchange
* Let Us “C” by Yashwant Kanetkar
* “C++” by Sumita Arora