COINS CHANGE PROBLEM

Submitted by

Viswesh Mekala
[RA2111003010009]

Anubhav Pathak
[RA2111003010120]

Rithvee chandak
[RA2111003010145]

Under the Guidance of

Dr.kishore Anthuvan Sahayaraj K
Assistant professor,
Department of computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
In
COMPUTER SCIENCE ENGINEERING
Of
FACULTY OF ENGINEERING AND TECHNOLOGY

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUE – 603 203
Chengalpattu District

BONAFIDE CERTIFICATE

Register No. RA2111003010009 Certified to be the bonafide work done by VISWESH MEKALA of II Year/IV Sem B.Tech Degree Course in the **Practical DESIGN AND ANALYSIS OF ALGORITHMS 18CSC204J** in **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY,** Kattankulathur during the academic year 2022 – 2023.

LAB INCHARGE **Head of the Department**

Name of the Faculty : Dr.kishore Anthuvan Sahayaraj K

Designation

Department of Computing Technologies

SRMIST – KTR.

TABLE OF CONTENTS

VISWESH-A1-009
ANUBHAV-B1-120
RITHVEE – C1-145
CSE CORE

COIN *CHANGE PROBLEM*

ABSTRACTION:

Given an integer array of coins[ ] of size N representing different types of currency and an integer sum, The task is to find the number of ways to make sum by using different combinations from coins[].

Note: Assume that you have an infinite supply of each type of coin.

Examples:

Input: sum *= 4, coins[] = {1,2,3},*

Output: 4

Explanation: there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}. of each type of coin.

According to the coin change problem, we are given a set of coins of various denominations. Consider the below array as the set of coins where each element is basically a denomination.

{1, 2, 5, 10, 20, 50, 100, 500}

Our task is to use these coins to form a sum of money using the minimum (or optimal) number of coins. Also, we can assume that a particular denomination has an infinite number of coins. In other words, we can use a particular denomination as many times as we want.

As an example, if we have to achieve a sum of 93, we need a minimum of 5 coins as below:

50 20 20 2 1

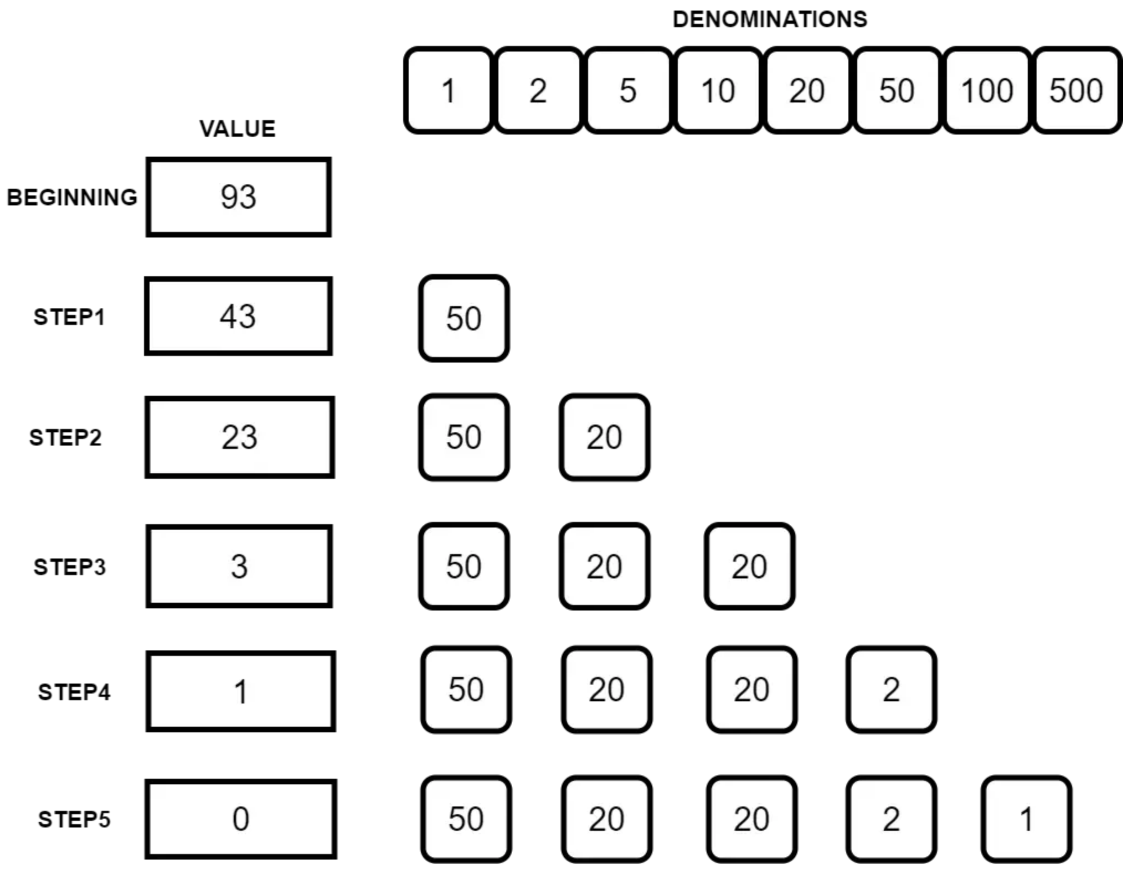
GREEDY METHOD:

ALGORITHM:

Going by the greedy approach, we first pull out a coin of denomination 50. We have 43 left to achieve. The next highest coin is 20 and therefore we pull that out. Then, we have 23 left. After that, we pull out another coin of 20 resulting in 3 left. We can then pull out 2 & 1 to finally reach 0 meaning we have reached a sum of 93.

As you can see, at each step, the algorithm makes the best possible choice. In this case, the highest denomination possible.

Below is an illustration that depicts the overall process:



As you can see, at each step we use the highest possible coin from the denominations. At last, we are able to reach the value of 93 just by using 5 coins.

Coin Change Problem Greedy Approach Implementation

Below is an implementation of the above algorithm using C++. However, you can use any programming language of choice.

```
#include <bits/stdc++.h>

std::vector<int> denominations = {1, 2, 5, 10, 20, 50, 100, 500};

void findMinCoins(int value)
{
    sort(denominations.begin(), denominations.end());

    std::vector<int> answer;
    for (int i = denominations.size() - 1; i >= 0; i--)
    {
        while (value >= denominations[i])
        {
            value = value - denominations[i];
answer.push_back(denominations[i]);
        }
    }
                  "The value can be achieved in " << answer.size() << " coins as be
  low:"std::cout
       <<
      << std::endl;

    for (int i = 0; i < answer.size(); i++)
    {
        std::cout << answer[i] <<  ;" "
    }
}
int main()
{
    int value = 93;
findMinCoins(value);     return
0;
}
```

We store the denominations in a vector. The values are kept in ascending order and then, we make sure to traverse from the end. This is because the higher denominations will be at the end of the array.

At each iteration, we keep comparing whether the remaining value is greater than the denomination. If yes, we deduct otherwise, we move to the next lower denomination. Finally, we have list of all the denominations we have used to reach the given value.

TIME COMPLEXITY ANALYSIS:

The time complexity of this code is **O(n log n),** where n is the number of denominations.

The reason for this is because the first operation in the findMinCoins function is to sort the denominations vector using the std::sort function. The time complexity of the std::sort function is O(n log n), where n is the number of elements in the vector.

After sorting, the function then loops through the denominations vector in reverse order, starting with the largest denomination. Within this loop, there is another loop that will execute at most n times, where n is the number of coins needed to make up the target value. Therefore, the time complexity of the loop is O(n).

Overall, the time complexity of the findMinCoins function is O(n log n + n) = O(n log n), since the **O(n log n)** term dominates.

Backtracking:

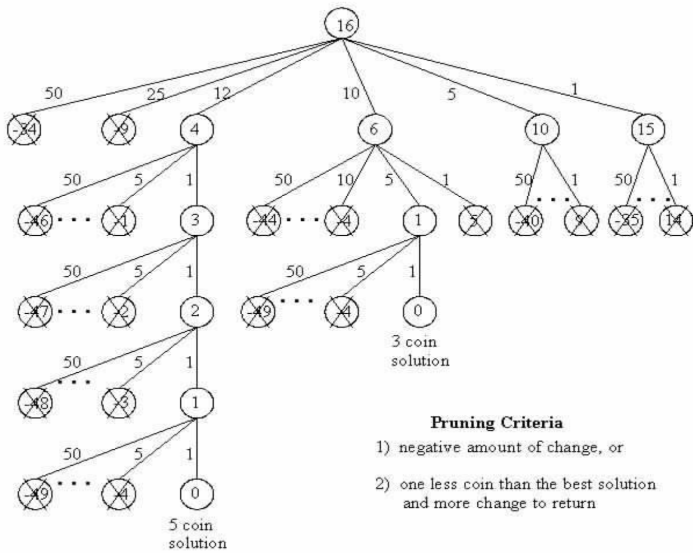
- Search the "state-space tree" using depth-first search
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time.

In the worst case, the algorithm is still exponential.

Backtracking State-Space Tree

State-Space Tree for 16-cents coins {1, 5, 10, 12, 25, 50}



Pruning Criteria

1) negative amount of change, or

2) one less coin than the best solution and more change to return

ALGORITHM:

```
Expand_Backtrack( treeNode n ) {
    treeNode c;
    for each child c of n do
        if promising(c) then
            if c is a solution that's better than best then
                best = c
            else
                Expand_Backtrack(c)
           end if
        end if
    end for
}
```

BACKTRACKING IMPLEMENTATION:

```
void Expand(int numberOfCoins, int changeToBeReturned) {
     int i, j;
     for (i = 0; i < numberOfCoinTypes; i++) {
          partialSolution[i] = partialSolution[i] + 1;
          changeToBeReturned = changeToBeReturned - coinValues[i];
          numberOfCoins++;
          if (Promising(numberOfCoins, changeToBeReturned)) {
              if ((changeToBeReturned) == 0) {
                  if (!solutionFound || numberOfCoins < bestNumberOfCoins) {
                         bestNumberOfCoins = numberOfCoins;
                         solutionFound = TRUE;
                         for (j = 0; j < numberOfCoinTypes; j++) {
                              bestSolutionSoFar[j] = partialSolution[j];
                         }
                  }
              } else {
                   Expand(numberOfCoins,changeToBeReturned);
              }
           }
           partialSolution[i] = partialSolution[i] - 1;
```

```
            changeToBeReturned = changeToBeReturned + coinValues[i];
            numberOfCoins--;
      }
}
int Promising(int numberOfCoins, int changeToBeReturned) {
     if (changeToBeReturned < 0) {
          return FALSE;
     } else if (solutionFound && changeToBeReturned > 0 && numberOfCoins+1 >=
bestNumberOfCoins) {
          return FALSE;
     }
     return TRUE;
}
```

TIME COMPLEXITY: The time complexity of the above code depends on the values of numberOfCoins and changeToBeReturned and the number of coin types, denoted by numberOfCoinTypes.

In the worst-case scenario, the algorithm needs to consider all possible combinations of coins that add up to the required change. The number of recursive calls required to solve the problem grows exponentially with the number of coin types and the amount of change. Therefore, the worst-case time complexity is O(2^(N)) where N denotes numberof coin types.

Dynamic Programming approach

The above problem lends itself well to a **dynamic programming approach**.

The dynamic programming solution finds all possibilities of forming a particular sum. Basically, this is quite similar to a brute-force approach.

Then, you might wonder how and why dynamic programming solution is efficient.

This is because the dynamic programming approach uses **memoization**. Due to this, it calculates the solution to a sub-problem only once. In other words, we can derive a particular sum by dividing the overall problem into sub-problems.

ALGORITHM:

Let’s work with the second example from previous section where the greedy approach did not provide an optimal solution.

In the above illustration, we create an initial array of size **sum + 1**. Also, we assign each element with the value **sum + 1**. Since we are trying to reach a sum of 7, we create an array of size 8 and assign 8 to each element's value. This array will basically store the answer to each value till 7. So, for example, the index 0 will store the minimum number of coins required to achieve a value of 0. Next, index 1 stores the minimum number of coins to achieve a value of 1.

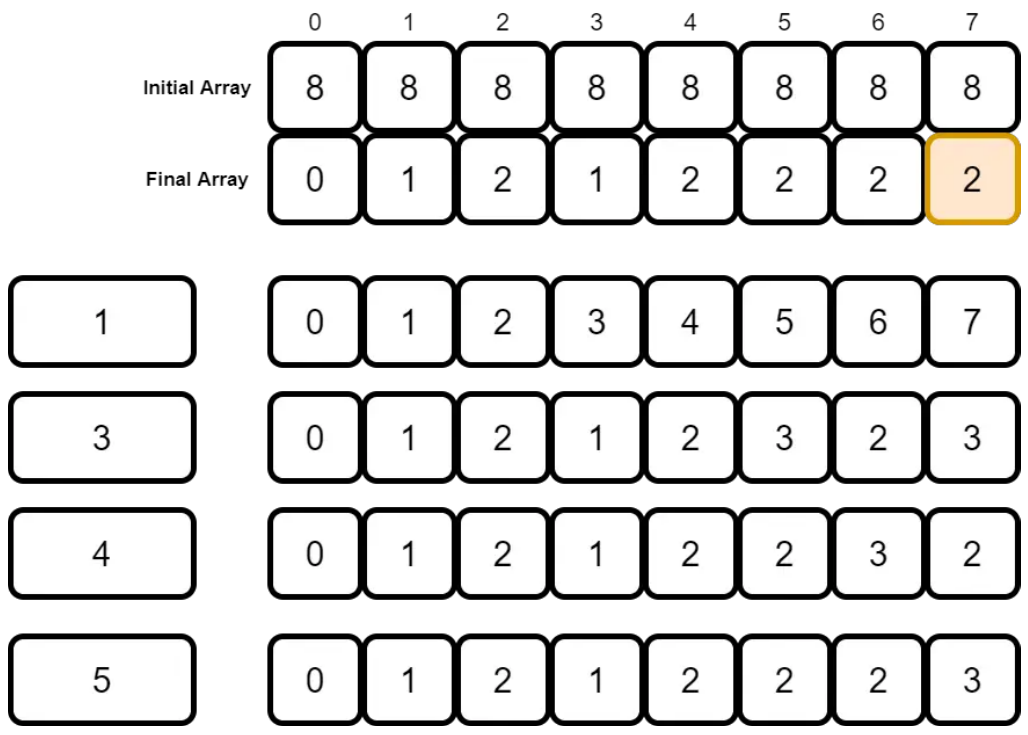
Lastly, index 7 will store the minimum number of coins to achieve value of 7. And that will basically be our answer. However, we will also keep track of the solution of every value from 0 to 7.

To fill the array, we traverse through all the denominations one-by-one and find the minimum coins needed using that particular denomination. As an example, first we take the coin of value 1 and decide how many coins needed to achieve a value of 0. The answer, of course is 0. Next, we look at coin having value of 3. The answer is still 0 and so on.

Once we check all denominations, we move to the next index. Using coin having value 1, we need 1 coin. Using other coins, it is not possible to make a value of 1. Hence, the minimum stays at 1. The main change, however, happens at value 3. Using coins of value 1, we need 3 coins. However, if we use a single coin of value 3, we just need 1 coin which is the optimal solution.

Completing the Approach

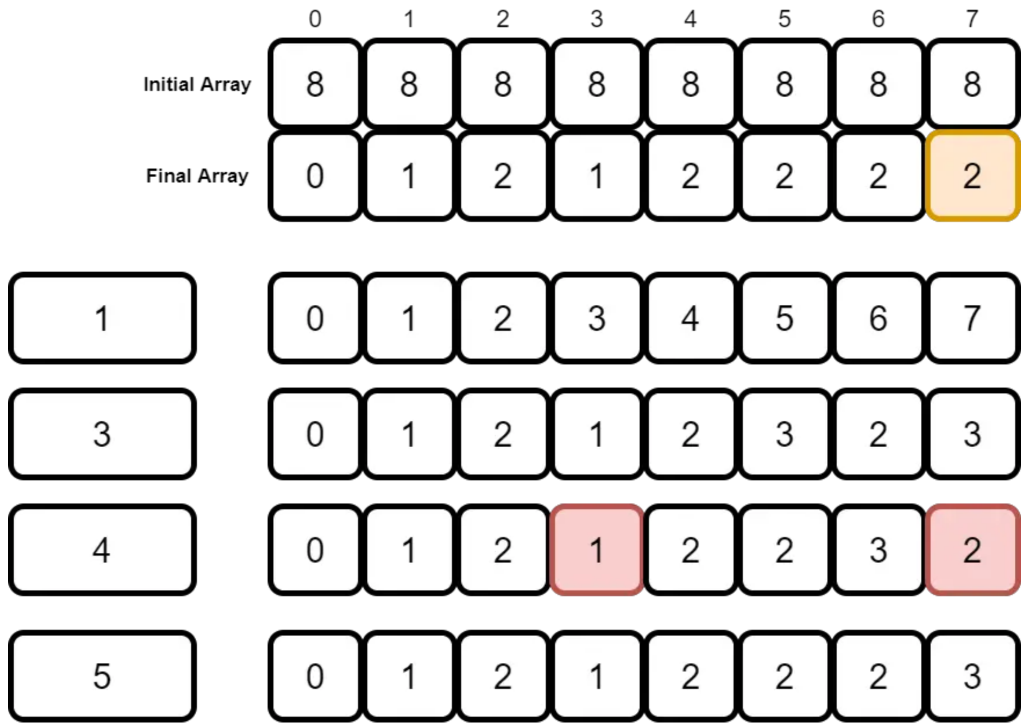
Following this approach, we keep filling the above array as below:



	0	1	2	3	4	5	6	7
Initial Array	8	8	8	8	8	8	8	8
Final Array	0	1	2	1	2	2	2	2
1	0	1	2	3	4	5	6	7
3	0	1	2	1	2	3	2	3
4	0	1	2	1	2	2	3	2
5	0	1	2	1	2	2	2	3

As you can see, we finally find our solution at index 7 of our array. Basically, 2 coins. And that is the most optimal solution.

Considering the above example, when we reach denomination 4 and index 7 in our search, we check that excluding the value of 4, we need 3 to reach 7. And using our stored results, we can easily see that the optimal solution to achieve 3 is 1 coin. Hence, the optimal solution to achieve 7 will be 2 coins (1 more than the coins required to achieve 3). See below highlighted cells for more clarity.



	0	1	2	3	4	5	6	7
Initial Array	8	8	8	8	8	8	8	8
Final Array	0	1	2	1	2	2	2	2
1	0	1	2	3	4	5	6	7
3	0	1	2	1	2	3	2	3
4	0	1	2	1	2	2	3	2
5	0	1	2	1	2	2	2	3

With this understanding of the solution, let’s now implement the same using C++.

Implementation of Coin Change Problem in C++

Below is an implementation of the **coin change problem using dynamic programming**.

```
#include <bits/stdc++.h>

std::vector<int> denominations = {1, 3, 4, 5};

int findMinCoins(int value)
{
    //Initializing  the  result  array           std::vector<int>
dp(value+1, value+1);

    //Setting the first element to 0     dp[0] = 0;

    for(int i = 1; i < dp.size(); i++)
    {
        for(int j = 0; j < denominations.size(); j++)
        {
            if(denominations[j] <= i)
            {
                dp[i] = std::min(dp[i], dp[i-denominations[j]] + 1);             }

        }
    }
    return dp[value] == (value + 1) ? - 1 : dp[value];
}
int main()
{
    int minimum_coins = findMinCoins(7);

    std::cout << minimum_coins << std::endl;

    return 0;
}
```

Time complexity analysis:

Basically, here we follow the same approach we discussed. The final results will be present in the **vector** named **dp**. We return that at the end.

The time complexity of this solution is O(A * n). Here, **A** is the amount for which we want to calculate the coins. Also, **n** is the number of denominations.

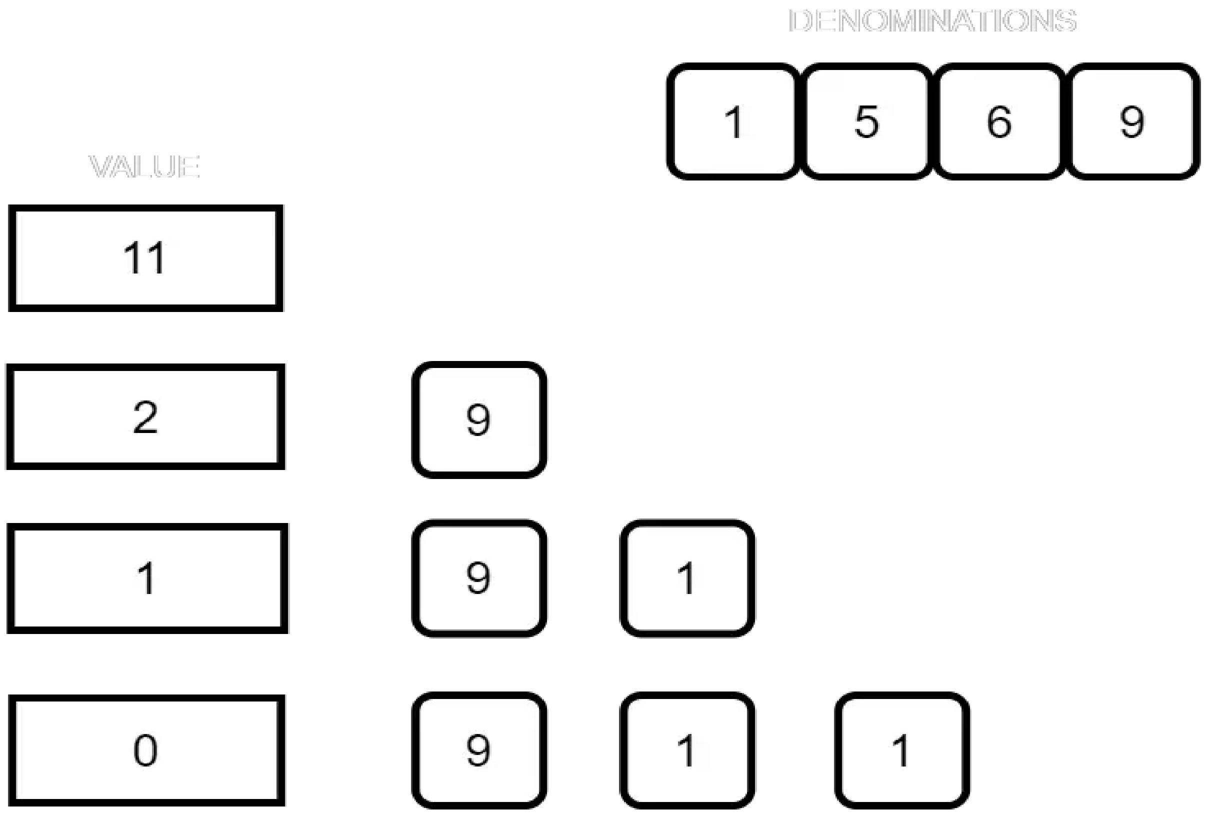
With this, we have successfully understood the solution of coin change problem using dynamic programming approach. Also, we implemented a solution using C++.

conclusion

While the coin change problem can be solved using Greedy algorithm, there are scenarios in which it does not produce an optimal result.

{1, 5, 6, 9}

Now, using these denominations, if we have to reach a sum of 11, the greedy algorithm will provide the below answer. See below illustration.



Here, accordingly to the Greedy algorithm, Backtracking we will end up the denomination 9, 1, 1 i.e. 3 coins to reach the value of 11. However, if you look closely, there is a more optimal solution. And that is by using the denominations 5 & 6. Using them, we can reach 11 with only 2 coins.

However, the way greedy approach works, this solution was never considered. And this can be thought of as a shortcoming of greedy approach. It does not work in the general cases.

we end up with 3 coins (5, 1, 1) for the above denominations. This is because the greedy algorithm always gives priority to local optimization.

Result: The specialty of dynamic approach is that it takes care of all types of input denominations. This is unlike the **Coin change problem using greedy algorithm** where certain cases resulted in a non-optimal solution.

REFERENCES:

https://progressivecoder.com/coin-change-problem-using-greedy-algorithm/

https://leetcode.com/problems/coin-change/solutions/1101485/java-multiple-approaches-backtracking-memoisation-and-dp/

http://www.cs.uni.edu/~fienup/cs270s04/lectures/lec13_2-24-04.htm