SCHOOL OF COMPUTING

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SRM NAGAR, KATTANKULATHUR-603202

DEPARTMENT OF DATA SCIENCE AND BUSINESS SYSTEM

PROJECT REPORT

Academic year: 2021-22

EVEN SEMESTER

DESIGN AND ANALYSIS OF ALGORITHMS – PROJECT REPORT

Program(UG/PG)	**:**	**UG**
Semester	**:**	**IV**
Course Code	**:**	**18CSC204J**
Student Name	**:**	**TALAT BINTI FIRDOUS** **UTKARSHA DIXIT**
Register Number	**:**	**RA2111027010115** **RA2111027010081**
Branch with Specialization	**:**	**CSE-BIG DATA ANALYTICS**
Section	**:**	**AB1**

DESIGN AND ANALYSIS OF ALGORITHMS

A COURSE PROJECT REPORT

By

Talat Binti Firdous
[RA2111027010115]
Utkarsha Dixit
[RA2111027010081]

Under the guidance of

Paul T Sheeba
(Associate Professor)

In partial fulfillment for the award of the degree of

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

SRM UNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled “SOLVING PROBLEM STATEMENTUSING 3 DIFFERENT ALGORITHMS” is the bonafide work of Talat Binti Firdous[RA2111027010115] and Utkarsha Dixit[RA2111027010081] who carried out the project work under our supervision. Certified further, thatto the best of our knowledge the work reported herein does not form any other project report or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Paul T Sheeba
Assistant Professor

ABSTRACT

The study of building effective algorithms to address complicated issues and evaluating their effectiveness is known as design and analysis of algorithms. An algorithm is a collection of guidelines that spells out how to carry out a certain activity or resolve a particular issue. Making a set of instructions that can be followed to solve a problem as quickly and precisely as feasible is the aim of constructing an algorithm.

Understanding the issue, discovering various solutions, choosing the best one, and then fine-tuning it to increase performance are all phases in the process of building an algorithm. The method is assessed in terms of its time and space complexity, as well as its accuracy and applicability for the issue at hand, during the analysis step.

In computer science and related subjects, efficient algorithms are crucial because they are used to address a variety of issues such as data analysis, optimisation, machine learning, and cryptography. Research is continuing in the design and study of algorithms, with the goal of creating new algorithms and improving those that already exist in order to solve problems more effectively.

A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found. It goes through all possible choices until a solution is found. The time complexity of a brute force algorithm is often proportional to the input size. Brute force algorithms are simple and consistent, but very slow. It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.

The greedy approach is used to solve optimization problems that require finding the best possible solution from a set of candidate solutions. In such problems, the greedy approach chooses the best option available at each step, without considering the future consequences of that decision. This is in contrast to other algorithms like dynamic programming or divide-and-conquer, which consider all possible solutions and choose the best one.

The technique of breaking a problem statement into subproblems and using the optimal result of subproblems as an optimal result of the problem statement is known as dynamic programming. This technique uses an optimized approach and results in optimal solutions. Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem.

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors. We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement. We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V. Gopal,** for bringing out novelty in all executions. We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project. We are highly thankful to my Course Project Faculty **Paul T Sheeba, Associate Professor,** for her assistance, timely suggestion, and guidance throughout the duration of this course project.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

PROBLEM STATEMENT

You are given n types of coin denominations of values v(1) < v(2) ..... < v(n) (all integers). Assume v(1) = 1, so you can always make change for any amount of money C. Give an algorithm which makes change for an amount of money C with as few coins as possible.

Create an algorithm that is best to find the best way to solve the give problem statement and Algorithm and evaluate the time complexity of each technique. Choose the algorithm that performs the best for this task by comparing its effectiveness to that of the other two.

DESIGN TECHNIQUES USED

1.Brute Force Approach
2. Greedy Algorithm
3, Dynamic Programming

Brute Force Approach: -

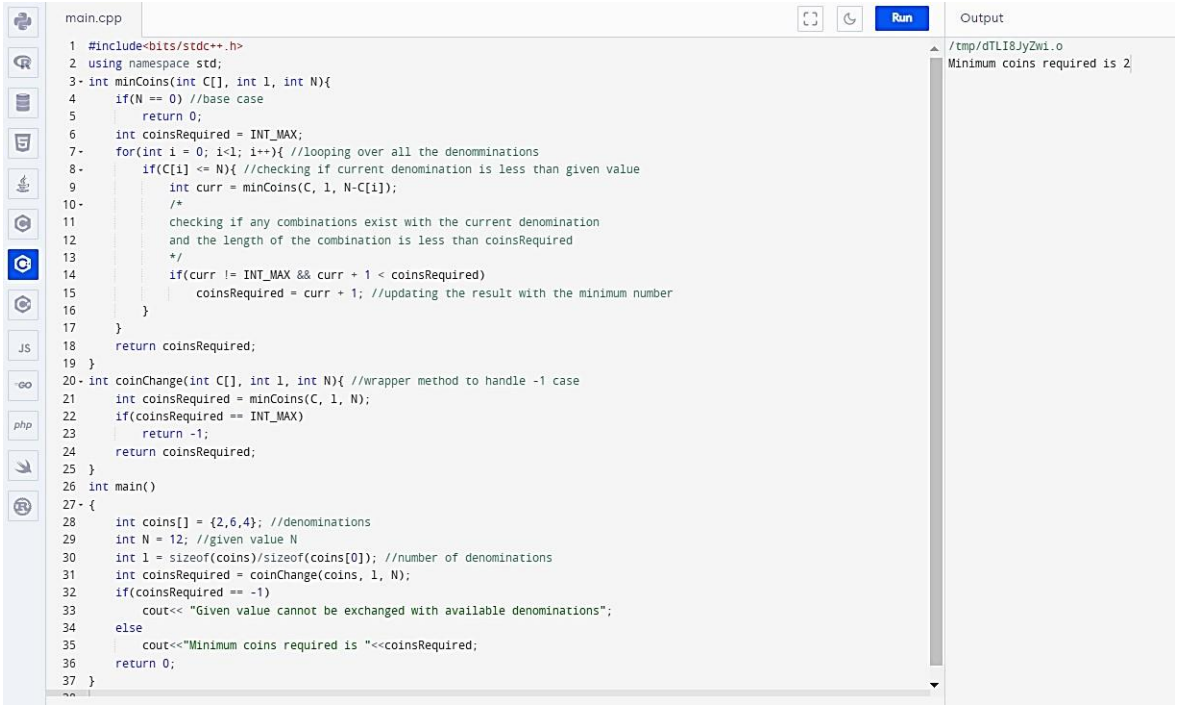
One way to solve this problem is by using the brute force approach. Brute force solution is recursive. Get Min Number Of Coins has one base case: if target sum is zero, then we need zero coins to get it. Otherwise, we try to use each coin and ask the function again to get min number of coins for a smaller sum (current sum minus coin value).

Code In C++ :-

```
#include<bits/stdc++.h>
using namespace std;
int minCoins(int C[], int l, int N){
   if(N == 0) //base case
      return 0;
   int coinsRequired = INT_MAX;
   for(int i = 0; i<l; i++){ //looping over all the denomminations
      if(C[i] <= N){ //checking if current denomination is less than given value
         int curr = minCoins(C, l, N-C[i]);
         /*
         checking if any combinations exist with the current denomination
         and the length of the combination is less than coinsRequired
         */
         if(curr != INT_MAX && curr + 1 < coinsRequired)
            coinsRequired = curr + 1; //updating the result with the minimum number
      }
   }
   return coinsRequired;
}
int coinChange(int C[], int l, int N){ //wrapper method to handle -1 case
   int coinsRequired = minCoins(C, l, N);
   if(coinsRequired == INT_MAX)
      return -1;
   return coinsRequired;
}
int main()
{
   int coins[] = {2,6,4}; //denominations
         int N = 12; //given value N
         int l = sizeof(coins)/sizeof(coins[0]); //number of denominations



         int coinsRequired = coinChange(coins, l, N);
         if(coinsRequired == -1)
            cout<< "Given value cannot be exchanged with available denominations";
         else
      cout<<"Minimum coins required is "<<coinsRequired;
   return 0;
}
```

Output:

Minimum number of coins required is 2



Complexity Analysis:

Time Complexity :

The above algorithm will make a recursive call for every denomination *Di*<=k and in the worst case, when all the denominations are less than the required value there will be *l* recursive calls.
This case implies that the first layer of the recursive tree will have *l*nodes in the worst case.
Similarly,n^2 nodes in the second lever, *n*^3 nodes in the third layer, and so on. It gives us the total number of recursive calls equal to (*n*^k−1)/(*n*−1).
So the worst-case time complexity is equivalent to ***O(n^k)***
Space Complexity :

This approach doesn't need any auxiliary space, but it maintains a recursion stack internally

Greedy Algorithm Approach:-

Another approach to solve this problem is by using greedy algorithm programming technique.

Code In C++ :-

```
#include <bits/stdc++.h>
using namespace std;
// All denominations of Indian Currency
int denomination[]
= { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };
int n = sizeof(denomination) / sizeof(denomination[0]);
void findMin(int V)
{
sort(denomination, denomination + n);
// Initialize result
vector<int> ans;
// Traverse through all denomination
for (int i = n - 1; i >= 0; i--) {
// Find denominations
while (V >= denomination[i]) {
V -= denomination[i];
ans.push_back(denomination[i]);
}
}
// Print result
or (int i = 0; i < ans.size(); i++)
cout << ans[i] << " ";
}
// Driver Code
int main()
{
int n = 93;
cout << "Following is minimal"<< " number of change for " << n << ": ";
// Function Call
findMin(n);
return 0;
}
```

Output:

Input: *V = 70*

Output: *2*

main.cpp

Run

Output

```cpp
#include <bits/stdc++.h>
using namespace std;
// All denominations of Indian Currency
int denomination[]
= { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };
int n = sizeof(denomination) / sizeof(denomination[0]);
void findMin(int V)
{
sort(denomination, denomination + n);
// Initialize result
vector<int> ans;
// Traverse through all denomination
for (int i = n - 1; i >= 0; i--) {
// Find denominations
while (V >= denomination[i]) {
V -= denomination[i];
ans.push_back(denomination[i]);
}
}
// Print result
for (int i = 0; i < ans.size(); i++)
cout << ans[i] << " ";
}
// Driver Code
int main()
{
int n = 93;
cout << "Following is minimal"<< " number of change for " << n << ": ";
// Function Call
findMin(n);
return 0;
}

```

```
/tmp/dTLI8JyZwi.o
Following is minimal number of change for 93: 50 20 20 2 1
```

Complexity Analysis:

Time Complexity: O(n)
Auxiliary Space: O(n)

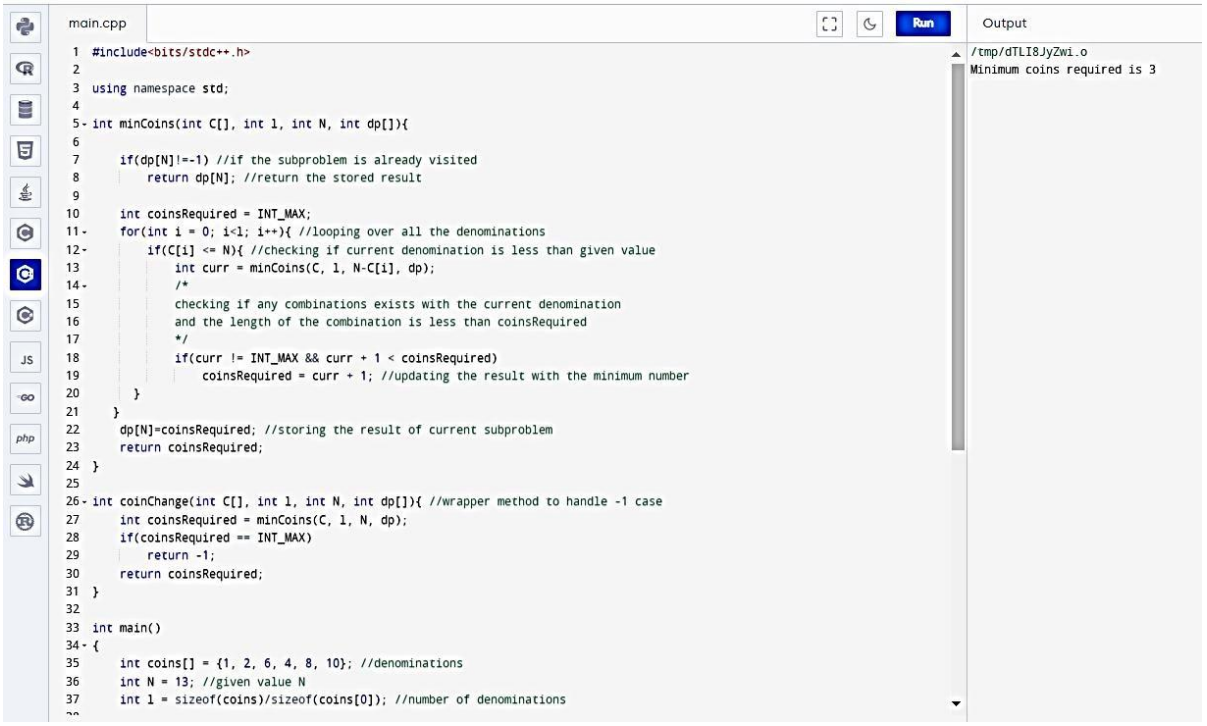
Dynamic Programming :-

Code In C:-

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

using namespace std;

int minCoins(int C[], int l, int N, int dp[]){

   if(dp[N]!=-1) //if the subproblem is already visited
      return dp[N]; //return the stored result

   int coinsRequired = INT_MAX;
   for(int i = 0; i<l; i++){ //looping over all the denominations
      if(C[i] <= N){ //checking if current denomination is less than given value
         int curr = minCoins(C, l, N-C[i], dp);
         /*
         checking if any combinations exists with the current denomination
         and the length of the combination is less than coinsRequired
         */
         if(curr != INT_MAX && curr + 1 < coinsRequired)
            coinsRequired = curr + 1; //updating the result with the minimum number
     }
   }
   dp[N]=coinsRequired; //storing the result of current subproblem
   return coinsRequired;
}

int coinChange(int C[], int l, int N, int dp[]){ //wrapper method to handle -1 case
   int coinsRequired = minCoins(C, l, N, dp);
   if(coinsRequired == INT_MAX)
      return -1;
   return coinsRequired;
}

int main()
{
   int coins[] = {1, 2, 6, 4, 8, 10}; //denominations
         int N = 13; //given value N
         int l = sizeof(coins)/sizeof(coins[0]); //number of denominations

         int dp[N+1]; //DP array to store results of subproblems

   /*
   Initially filling the whole dp array with -1
   Indicates that all the subproblems are unvisited
   */
```

```
    fill(dp, dp+N+1, -1);

    //base case
    dp[0]=0;

    int coinsRequired = coinChange(coins, l, N, dp);

    if(coinsRequired == -1){
        cout<<"Given value cannot be exchanged with available denominations";
    }
    else
            cout<<"Minimum coins required is "<<coinsRequired;

    return 0;
}
```

Output:
Minimum coins required is 3.



Complexity Analysis:

Time Complexity :

In the worst case we'll make a recursive call for all the values from 1 to k, i.e., minCoins(k, coins), minCoins($k-1$, coins), minCoins($k-2$, coins)..., minCoins(0, coins). And each recursive call has to loop over all denominations to find the minimum coins, so n x constant time. (constant time since we are memoizing the results of recursive calls).
So the time complexity of this algorithm will be $O(n.k)$.

Space Complexity :

This approach requires an auxiliary array of size k+1. So the space complexity will be $O(k)$.

Also, this approach will maintain a recursion stack of size N in the worst case.

COMPLEXITY ANALYSIS: -

All three methods have their own advantages and disadvantages. Here is a summary of the time complexity and space complexity of each method:

Algorithm	Time Complexity	Optimality
Brute Force	Exponential	Optimal
Dynamic Programming	O(n.k)	Optimal
Greedy Algorithm	O(n)	Not always optimal

CONCLUSION: -

Based on this comparison, **the dynamic programming approach is the best way to solve the coin change problem**. While the recursion method is optimal, its exponential time complexity makes it impractical for real-world applications. The greedy algorithm has a fast time complexity, but it is not always optimal, which means that it may not find the minimum number of coins required to make change for a given target amount. On the other hand, the dynamic programming approach has a polynomial time complexity, which makes it practical for real-world applications.
Additionally, it guarantees to find the optimal solution, which means that it always finds the minimum number of coins required to make change for a given target amount.

Therefore, the dynamic programming approach is the best way to solve the coin change problem.