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: -

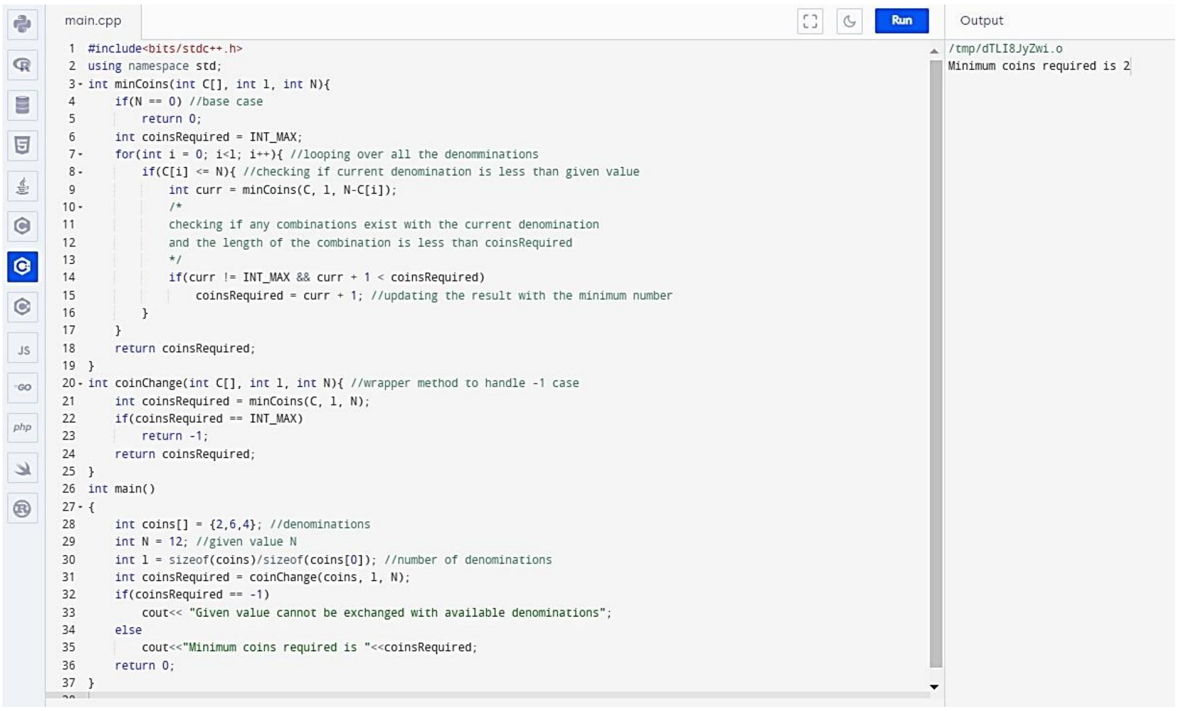
One way to solve this problem is by using the brute force approach. Brute force solution is recursive. Get Min Number Of Coins has one base case: if target sum is zero, then we need zero coins to get it. Otherwise, we try to use each coin and ask the function again to get min number of coins for a smaller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



```cpp
#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;
}
```

```
/tmp/dTLI8JyZwi.o
Minimum 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 *ln*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.

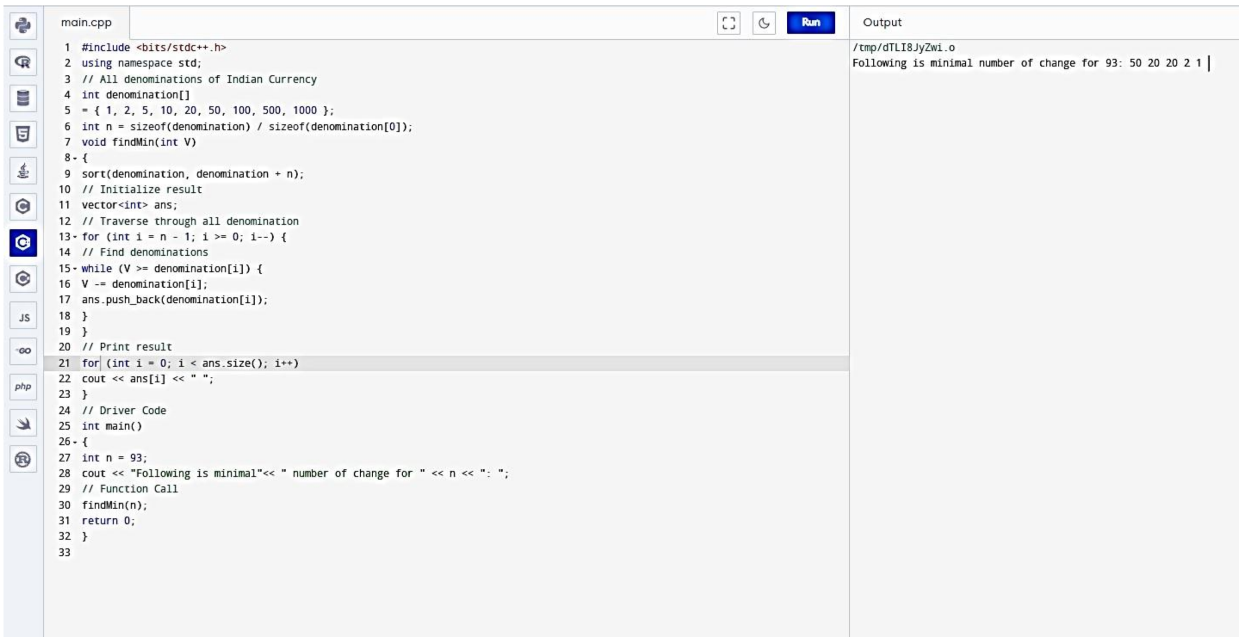
Code In C++ :-

```
#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*



```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)

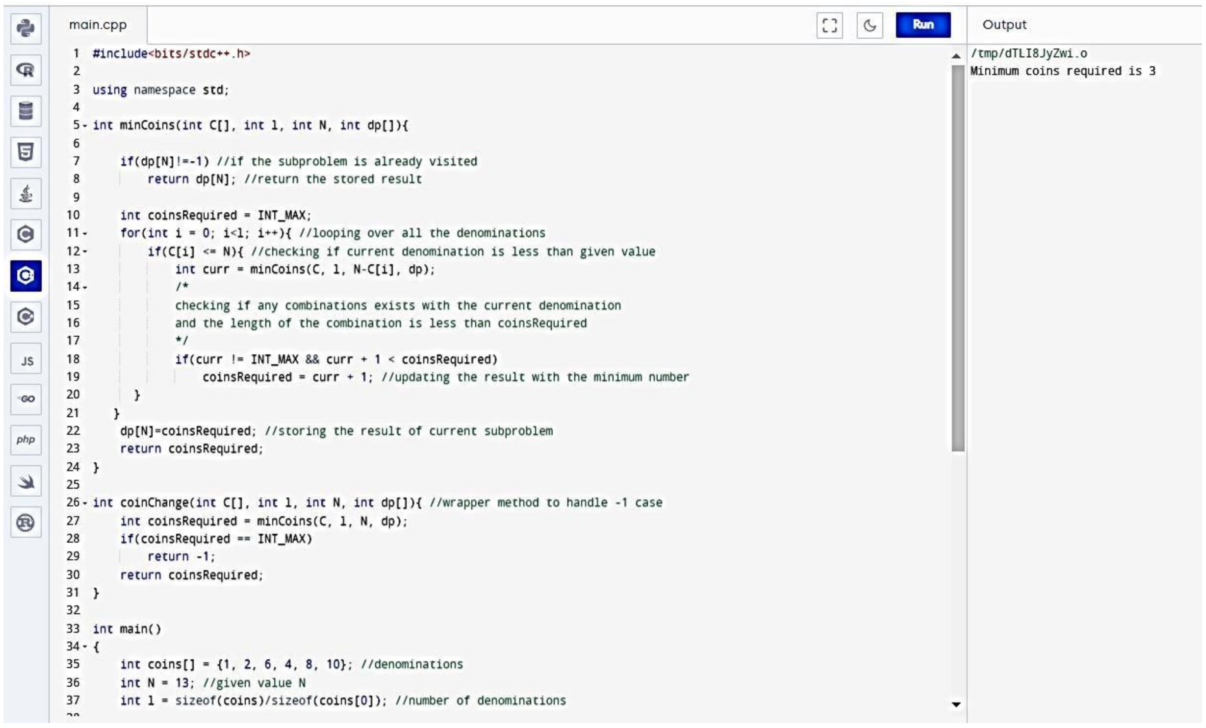
Dynamic Programming :-

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.



```
#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
```

```
/tmp/dTLI8JyZwi.o
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.