Coin Change Problem | Dynamic Programming

Coin change problem is the last algorithm we are going to discuss in this section of dynamic programming (/course/algorithms-dynamic-programming/). In the coin change problem, we are basically provided with coins with different denominations like 1¢, 5¢ and 10¢. Now, we have to make an amount by using these coins such that a minimum number of coins are used.

Let's take a case of making 10¢ using these coins, we can do it in the following ways:

1. Using 1 coin of 10¢
2. Using two coins of 5¢
3. Using one coin of 5¢ and 5 coins of 1¢
4. Using 10 coins of 1¢

10
5
5
5
1
10
10
10
10

Out of these 4 ways of making 10¢, we can see that the first way of using only one coin of 10¢ requires the least number of coins and thus it is our solution.

So in a coin change problem, we are provided with different denominations of coins:

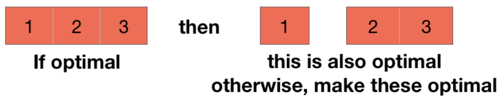
$$1 = d_1 < d_2 < d_3 < \ldots < d_k$$

$d_1 = 1$ ensures that we can make any amount using these coins.

Now, we have to make change for the value n using these coins and we need to find out the minimum number of coins required to make this change.

Approach to Solve the Coin Change Problem

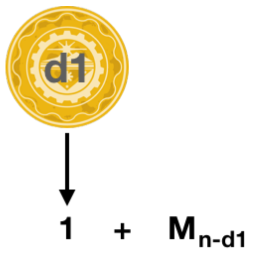
Like the rod cutting problem (/course/algorithms-rod-cutting/), coin change problem also has the property of the optimal substructure i.e., the optimal solution of a problem incorporates the optimal solution to the subproblems. For example, we are making an optimal solution for an amount of 8 by using two values - 5 and 3. So, the optimal solution will be the solution in which 5 and 3 are also optimally made, otherwise, we can reduce the total number of coins of optimizing the values of 5 and 8.



The reason we are checking if the problem has optimal substructure or not because if there is optimal substructure, then the chances are quite high that using dynamic programming will optimize the problem.

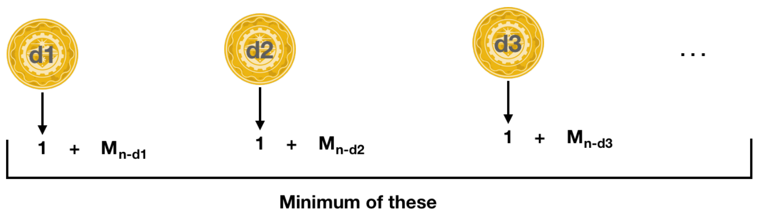
Let's say M_n is the minimum number of coins needed to make the change for the value n.

Let's start by picking up the first coin i.e., the coin with the value d_1. So, we now need to make the value of $n - d_1$ and M_{n-d_1} is the minimum number of coins needed for this purpose. So, the total number of coins needed are $1 + M_{n-d_1}$ (1 coin because we already picked the coin with value d_1 and M_{n-d_1} is the minimum number of coins needed to make the rest of the value).



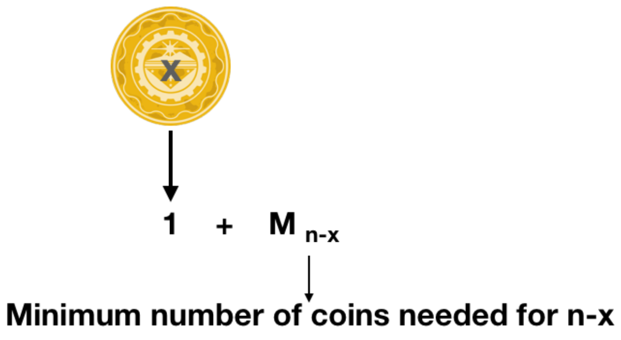
Similarly, we can pick the second coin first and then attempt to get the optimal solution for the value of $n - d_2$ which will require M_{n-d_2} coins and thus a total of $1 + M_{n-d_2}$.

We can repeat the process with all the k coins and then the minimum value of all these will be our answer. i.e., $\min_{i:d_i \le n}\{M_{n-d_i} + 1\}$.



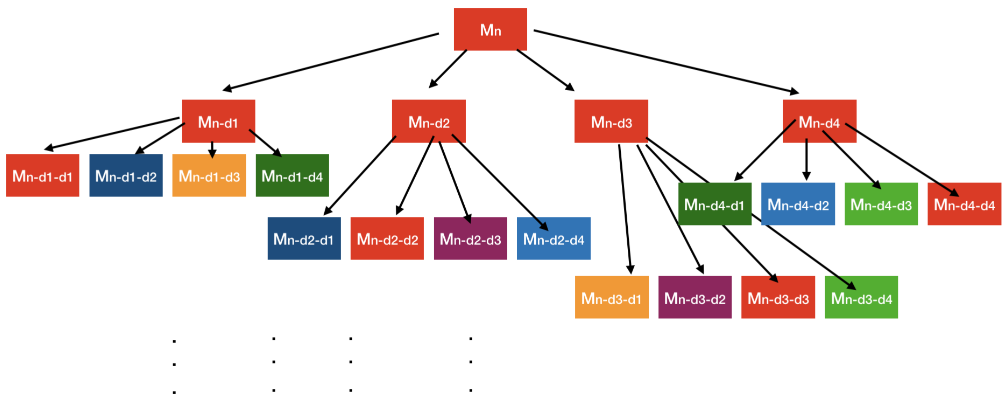
The above process can also be understood in a different way. Suppose, we have picked a coin with value x and we know that this coin is going to be in the solution. So, our next task is to find the minimum number of coins needed to make the change of value n-x i.e., M_{n-x}. Also, by choosing the coin with value x, we have already increased the total number of coins needed by 1. So, we can write:

$$M_n = 1 + M_{n-x}$$



But the real problem is that we don't know the value of x. So, we will try every coin for the value of x and then we will select the minimum among those.

$$M_n = \begin{cases} \min_{i:d_i \le n}\{M_{n-d_i} + 1\}, & \text{if } n > 0 \\ 0, & \text{if } n = 0 \end{cases}$$



You can see that there are overlapping subproblems in our solution and thus, we can use dynamic programming to optimize this. So, let's look at the coding implementation of the formula we derived above.

(/add_ques

Code for Coin Change Problem

We are going to use the bottom-up implementation of the dynamic programming to the code.

Our function is going to need the denomination vectors of coin (d), the value for which change has to be made (n) and number of denominations we have (k or number of elements in the array d) i.e., `COIN-CHANGE(d, n, k)`

Let's start by making an array to store the minimum number of coins needed for each value i.e., `M[n+1]`.

We know that for a value of 0, no coins are needed at all.

```
M[0] = 0
```

Now, we need to iterate for each value in array the M and fill it up. Also to store the minimum number of coins for each value, we will make a variable and initialize it with ∞ so that all other values are less than it in the starting. It is similar to the variable 'maximum_revenue' we made in the previous chapter.

```
for j in 1 to n
    minimum = INF
```

Now for each j, we need to choose the minimum of $M_{n-d_i}+1$, where d_i will change from d_1 to d_k and thus, i will change from 1 to k. Also, if the value of d_i is greater than j (value to be made), then of course, we can't use that coin.

```
for j in 1 to n
    minimum = INF
    for i in 1 to k
        if j >= d[i]
            minimum = min(minimum, 1 + M[j-d[i]])
```

`for i in 1 to n` → We are iterating to change the value of d_i from d_1 to d_k.

`if j >= d[i]` → If the value to be made (j) is greater than the value of the current coin (d[i]), only then we can use this coin.

`minimum = min(minimum, 1 + M[j-d[i]])` → If the current value of M[j-d[i]] (or M_{j-d_i}) is less than the current minimum, then we are changing the value of the 'minimum'.

Now we just need to change the value of the array M to the calculated minimum and return it.

```
for j in 1 to n
    minimum = INF
    for i in 1 to k
        ...
    M[j] = minimum
return M[n]
```

?

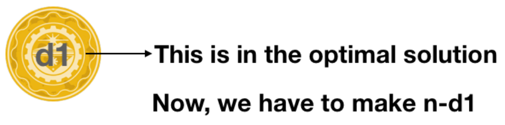
```
COIN-CHANGE(d, n, k)
  M[n+1]
  M[0] = 0

  for j in 1 to n
    minimum = INF

    for i in 1 to k
      if j >= d[i]
        minimum = min(minimum, 1+M[j-d[i]])

    M[j] = minimum
  return M[n]
```

C **Python** **Java**

```c
#include <stdio.h>
#define MIN(x, y) (((x) < (y)) ? (x) : (y))

const int INF = 100000;

//k is number of denominations of the coin or length of d
int coin_change(int d[], int n, int k) {
  int M[n+1];
  M[0] = 0;

  int i, j;
  for(j=1; j<=n; j++) {
    int minimum = INF;

    for(i=1; i<=k; i++) {
      if(j >= d[i]) {
        minimum = MIN(minimum, 1+M[j-d[i]]);
      }
    }
    M[j] = minimum;
  }
  return M[n];
}

int main() {
  // array starting from 1, element at index 0 is fake
  int d[] = {0, 1, 2, 3};
  printf("%d\n", coin_change(d, 5, 3)); //to make 5. Number of denominations = 3
  return 0;
}
```

Coins in Optimal Solution

Let's suppose we know the first coin needed to get the optimal solution of each value. Now, we can subtract the value of the first coin from n and then we will have a different value to be made.



As discussed above, the problem exhibits optimal substructure and we know the first coin needed to make this new value, so we know the second coin needed to make the value n.

d5
d5 is the first coin needed to make n-d1
It is the second coin for the value n

We can repeat this process to get all the coins needed.

So, the point is we only need to store the first coin needed for each value and then we can get the optimal coins for any value.

Thus, we will modify the above code and store the first coins of each value in an array S.

```
COIN-CHANGE-MODIFIED(d, n, k)
  M[n+1]
  M[0] = 0

  S[n+1]
  S[0] = 0

  for j in 1 to n
    minimum = INF

    for i in 1 to k
      if j >= d[i]
        if 1+M[j-d[i]] < minimum
          minimum = 1+M[j-d[i]]
          coin = i

    M[j] = minimum
    S[j] = coin

  //Code to print the coins, given below
  ...
  return M[n]
```

`S[n+1]` → We are declaring the array S.

We have just broken the 'min' function from the above code `minimum = min(minimum, 1+M[j-d[i]])` to `if 1+M[j-d[i]] < minimum → minimum = 1+M[j-d[i]]`. In this case, we are also assigning the value of i to the variable 'coin' because this coin 'i' is giving us the minimum value and thus, it will be the coin to be stored in the array S.

At last, we are storing this value in the array S `S[j] = coin`.

(/add_ques

```
COIN-CHANGE-MODIFIED(d, n, k)
  M[n+1]
  M[0] = 0

  S[n+1]
  S[0] = 0

  for j in 1 to n
    minimum = INF

    for i in 1 to k
      if j >= d[i]
        if 1+M[j-d[i]] < minimum
          minimum = 1+M[j-d[i]]
          coin = i

    M[j] = minimum
    S[j] = coin

  //Code to print the coins
  l = n
  while l>0
    print d[S[l]]
    l = l-d[S[l]]

  return M[n]
```

C **Python** **Java**

```c
#include <stdio.h>

const int INF = 100000;

//k is number of denominations of the coin or length of d
int coin_change_modified(int d[], int n, int k) {
  int M[n+1];
  M[0] = 0;

  int S[n+1];
  S[0] = 0;

  int i, j;
  for(j=1; j<=n; j++) {
    int minimum = INF;
    int coin=0;

    for(i=1; i<=k; i++) {
      if(j >= d[i]) {
        if((1+M[j-d[i]]) < minimum) {
          minimum = 1+M[j-d[i]];
          coin = i;
        }
      }
    }
    M[j] = minimum;
    S[j] = coin;
  }

  int l = n;
  while(l>0) {
    printf("%d\n",d[S[l]]);
    l = l-d[S[l]];
  }
  return M[n];
}

int main() {
  // array starting from 1, element at index 0 is fake
  int d[] = {0, 1, 2, 3};
  coin_change_modified(d, 5, 3);
  return 0;
}
```

As said above, we are first printing the value of the coin (S[l] will give us the coin and d[S[l]] is the value of that coin). After this, our initial value has decreased by the value of the coin i.e., `l = l-d[S[l]]` and we are doing this until we the value is greater than 0 `while l>0`.

Analysis of the Algorithm

The first loop is iterating from 1 to n and thus has a running time of $\Theta(n)$. The last loop will also run a total of n times in worst case (it can run faster depending upon the value of `l-d[S[l]]`). Thus it has a running time of $O(n)$.

Now, there is also a nested loop. The first loop if iterating from 1 to n and the second is iterating from 1 to k and thus, a total running time of $\Theta(nk)$. All the other statements are constant time taking statements. Thus the algorithm has a running time of $\Theta(nk)$.

There are many other interesting algorithms which are optimized by dynamic programming. You can write about them on BlogsDope (https://www.codesdope.com/blog/) or can download the BlogsDope app (https://play.google.com/store/apps/details?id=com.blogsdope) to never miss any new articles.

“ Leaders live by choice, not by accident. ”

- Mark Gorman