

# Minimum Coin Change

We'll cover the following



- Introduction
- Problem Statement
- Basic Solution
  - Code
- Top-down Dynamic Programming with Memoization
- Bottom-up Dynamic Programming
  - Code

## Introduction #

Given an infinite supply of 'n' coin denominations and a total money amount, we are asked to find the minimum number of coins needed to make up that amount.

### Example 1:

```
Denominations: {1,2,3}
Total amount: 5
Output: 2
Explanation: We need minimum of two coins {2,3} to make a total of '5'
```

### Example 2:

```
Denominations: {1,2,3}
Total amount: 11
Output: 4
Explanation: We need minimum four coins {2,3,3,3} to make a total of '11'
```

## Problem Statement #

Given a number array to represent different coin denominations and a total amount 'T', we need to find the minimum number of coins needed to make change for 'T'. We can assume an infinite supply of coins, therefore, each coin can be chosen multiple times.

This problem follows the Unbounded Knapsack

(<https://www.educative.io/collection/page/5668639101419520/5633779737559040/5745865499082752/>) pattern.

## Basic Solution #

A basic brute-force solution could be to try all combinations of the given coins to select the ones that give a total sum of 'T'. This is what our algorithm will look like:



```
1 for each coin 'c'
2   create a new set which includes one quantity of coin 'c' if it does not exceed 'T', and
3   recursively call to process all coins
4   create a new set without coin 'c', and recursively call to process the remaining coins
5   return the count of coins from the above two sets with a smaller number of coins
```

## Code #

Here is the code for the brute-force solution:

Java

JS

Python3

C++

```
1 import math
2
3
4 def count_change(denominations, total):
5     result = count_change_recursive(denominations, total, 0)
6     return -1 if result == math.inf else result
7
8
9 def count_change_recursive(denominations, total, currentIndex):
10    # base check
11    if total == 0:
12        return 0
13
14    n = len(denominations)
15    if n == 0 or currentIndex >= n:
16        return math.inf
17
18    # recursive call after selecting the coin at the currentIndex
19    # if the coin at currentIndex exceeds the total, we shouldn't process this
20    count1 = math.inf
21    if denominations[currentIndex] <= total:
22        res = count_change_recursive(
23            denominations, total - denominations[currentIndex], currentIndex)
24        if res != math.inf:
25            count1 = res + 1
26
27    # recursive call after excluding the coin at the currentIndex
28    count2 = count_change_recursive(denominations, total, currentIndex + 1)
29
30    return min(count1, count2)
31
32
33 def main():
34     print(count_change([1, 2, 3], 5))
35     print(count_change([1, 2, 3], 11))
36     print(count_change([1, 2, 3], 7))
37     print(count_change([3, 5], 7))
38
39
40 main()
41
```



The time complexity of the above algorithm is exponential  $O(2^{C+T})$ , where 'C' represents total coin denominations and 'T' is the total amount that we want to make change. The space complexity will be  $O(C + T)$ .



Let's try to find a better solution.

## Top-down Dynamic Programming with Memoization #

We can use memoization to overcome the overlapping sub-problems. We will be using a two-dimensional array to store the results of solved sub-problems. As mentioned above, we need to store results for every coin combination and for every possible sum:

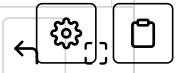
Java

JS

Python3

C++

```
1 import math
2
3
4 def count_change(denominations, total):
5     dp = [[-1 for _ in range(total+1)] for _ in range(len(denominations))]
6     result = count_change_recursive(dp, denominations, total, 0)
7     return -1 if result == math.inf else result
8
9
10 def count_change_recursive(dp, denominations, total, currentIndex):
11     # base check
12     if total == 0:
13         return 0
14     n = len(denominations)
15     if n == 0 or currentIndex >= n:
16         return math.inf
17
18     # check if we have not already processed a similar sub-problem
19     if dp[currentIndex][total] == -1:
20         # recursive call after selecting the coin at the currentIndex
21         # if the coin at currentIndex exceeds the total, we shouldn't process this
22         count1 = math.inf
23         if denominations[currentIndex] <= total:
24             res = count_change_recursive(
25                 dp, denominations, total - denominations[currentIndex], currentIndex)
26             if res != math.inf:
27                 count1 = res + 1
28
29         # recursive call after excluding the coin at the currentIndex
30         count2 = count_change_recursive(
31             dp, denominations, total, currentIndex + 1)
32         dp[currentIndex][total] = min(count1, count2)
33
34     return dp[currentIndex][total]
35
36
37 def main():
38     print(count_change([1, 2, 3], 5))
39     print(count_change([1, 2, 3], 11))
40     print(count_change([1, 2, 3], 7))
41     print(count_change([3, 5], 7))
42
43
44 main()
```



## Bottom-up Dynamic Programming #

Let's try to populate our array `dp[TotalDenominations][Total+1]` for every possible total with a minimum number of coins needed.

So for every possible total 't' ( $0 \leq t \leq \text{Total}$ ) and for every possible coin index ( $0 \leq \text{index} < \text{denominations.length}$ ), we have two options:

- Exclude the coin: In this case, we will take the minimum coin count from the previous set  
 $\Rightarrow dp[\text{index}-1][t]$
- Include the coin if its value is not more than 't': In this case, we will take the minimum count needed to get the remaining total, plus include '1' for the current coin  $\Rightarrow dp[\text{index}][t - \text{denominations}[\text{index}]] + 1$

Finally, we will take the minimum of the above two values for our solution:

$$dp[\text{index}][t] = \min(dp[\text{index}-1][t], dp[\text{index}][t - \text{denominations}[\text{index}]] + 1)$$

Let's draw this visually with the following example:

Denominations: [1, 2, 3]  
Total: 7

Let's start with our base case of zero total:

denominations\total	0	1	2	3	4	5	6	7
1	0							
{1, 2}	0							
{1, 2, 3}	0							

We don't need any coin to make zero total

1 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1						
{1, 2}	0							
{1, 2, 3}	0							

Total:1, Index:0 =>  $dp[Index][Total - denominations[Index] + 1]$ , we didn't consider  $dp[Index-1][Total]$  as Index = 0

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2					
{1, 2}	0							
{1, 2, 3}	0							

Total:2, Index:0 =>  $dp[Index][Total - denominations[Index] + 1]$ , we didn't consider  $dp[Index-1][Total]$  as Index = 0

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0							
{1, 2, 3}	0							

Total:3-7, Index:0 =>  $dp[Index][Total - denominations[Index] + 1]$ , we didn't consider  $dp[Index-1][Total]$  as Index = 0

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1						
{1, 2, 3}	0							

Total:1, Index:1 =>  $dp[Index-1][t]$ , we didn't consider  $dp[Index][Total - denominations[Index]]$  as  $Total < denominations[Index]$

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1					
{1, 2, 3}	0							

Total:2, Index:1 =>  $\min(dp[Index-1][Total], dp[Index][Total - denominations[Index] + 1])$



denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2				
{1, 2, 3}	0							

Total:3, Index:1 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

7 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0							

Total:4-7, Index:1 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

8 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1					

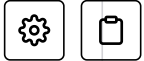
Total:1-2, Index:2 => dp[Index-1][t], we didn't consider dp[Index][Total-denominations[Index] as Total < denominations[Index]

9 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1	1				

Total:3, Index:2 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

10 of 14



denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1	1	2			

Total:4, Index:2 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

11 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1	1	2	2		

Total:5, Index:2 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

12 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1	1	2	2	2	

Total:6, Index:2 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

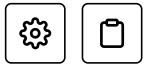
13 of 14

denominations\total	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
{1, 2}	0	1	1	2	2	3	3	4
{1, 2, 3}	0	1	1	1	2	2	2	3

Total:7, Index:2 => min( dp[Index-1][Total], dp[Index][Total-denominations[Index] + 1)

14 of 14

Code #



Here is the code for our bottom-up dynamic programming approach:

Java

JS

Python3

C++

```
1 import math
2
3
4 def count_change(denominations, total):
5     n = len(denominations)
6     dp = [[math.inf for _ in range(total+1)] for _ in range(n)]
7
8     # populate the total=0 columns, as we don't need any coin to make zero total
9     for i in range(n):
10         dp[i][0] = 0
11
12     for i in range(n):
13         for t in range(1, total+1):
14             if i > 0:
15                 dp[i][t] = dp[i - 1][t] # exclude the coin
16             if t >= denominations[i]:
17                 if dp[i][t - denominations[i]] != math.inf:
18                     # include the coin
19                     dp[i][t] = min(dp[i][t], dp[i][t - denominations[i]] + 1)
20
21     # total combinations will be at the bottom-right corner.
22     return -1 if dp[n - 1][total] == math.inf else dp[n - 1][total]
23
24
25 def main():
26     print(count_change([1, 2, 3], 5))
27     print(count_change([1, 2, 3], 11))
28     print(count_change([1, 2, 3], 7))
29     print(count_change([3, 5], 7))
30
31
32 main()
33
```

▶

📄 ↶ ⌂

The above solution has time and space complexity of  $O(C * T)$ , where 'C' represents total coin denominations and 'T' is the total amount that we want to make change.



