Copy Copy

*** May 27, 2016 | 299.7K views Average Rating: 4.56 (196 votes)

LeetCode

You are given coins of different denominations and a total amount of money amount.

Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1. Example 1: Input: coins = [1, 2, 5], amount = 11

Output: 3 Explanation: 11 = 5 + 5 + 1Example 2: Input: coins = [2], amount = 3 Output: -1

Note:

You may assume that you have an infinite number of each kind of coin.

Summary This article is for intermediate users. It introduces the following ideas: Backtracking,

Solution

Dynamic programming.

Approach #1 (Brute force) [Time Limit Exceeded]

Intuition

The problem could be modeled as the following optimization problem: $\min_{x} \sum_{i=0}^{n-1} x_i$

subject to $\sum_{i=0}^{n-1} x_i * c_i = S$, where S is the amount, c_i is the coin denominations, x_i is the number of coins with denominations c_i used in change of amount S. We could easily see that $x_i = [0, \frac{S}{c_i}]$.

A trivial solution is to enumerate all subsets of coin frequencies $[x_0 \ldots x_{n-1}]$ that satisfy the constraints above, compute their sums and return the minimum among them.

To apply this idea, the algorithm uses backtracking technique to generate all combinations of coin frequencies
$$[x_0\dots x_{n-1}]$$
 in the range which satisfy the constraints above. It makes a sum of the combinations and returns their minimum or -1

Algorithm To apply this idea, the algorithm uses backtracking technique to generate all

7

8

9

10

13

14

15 16

17

18 19

public class Solution { public int coinChange(int[] coins, int amount) {

> if (idxCoin < coins.length && amount > 0) { int maxVal = amount/coins[idxCoin]; int minCost = Integer.MAX_VALUE; for (int x = 0; $x \leftarrow maxVal$; x++) {

if (amount >= x * coins[idxCoin]) {

minCost = Math.min(minCost, res + x);

private int coinChange(int idxCoin, int[] coins, int amount) {

in case there is no acceptable combination.

if (amount == 0)

if (res != -1)

coin denominations $[c_0 \dots c_{n-1}]$

them. The following recurrence relation holds:

return 0;

return (minCost == Integer.MAX_VALUE)? -1: minCost; 21 22 return -1; 23 24 // Time Limit Exceeded Complexity Analysis • Time complexity : $O(S^n)$. In the worst case, complexity is exponential in the number of the coins n. The reason is that every coin denomination c_i could have at most $\frac{S}{c_2}$ values. Therefore the number of possible combinations is : $\frac{S}{c_1} * \frac{S}{c_2} * \frac{S}{c_3} \dots \frac{S}{c_n} = \frac{S^n}{c_1 * c_2 * c_3 \dots c_n}$ Space complexity: O(n). In the worst case the maximum depth of recursion is n. Therefore we need O(n) space used by the system recursive stack. Approach #2 (Dynamic programming - Top down) [Accepted]

Could we improve the exponential solution above? Definitely! The problem could be solved with polynomial time using Dynamic programming technique. First, let's define:

F(S) - minimum number of coins needed to make change for amount S using

We note that this problem has an optimal substructure property, which is the key piece in solving any Dynamic Programming problems. In other words, the optimal solution can be

subproblems? Let's assume that we know F(S) where some change val_1, val_2, \ldots for S which is optimal and the last coin's denomination is C. Then the following equation

F(S) = F(S - C) + 1

 (c_i) for each possible denomination $(c_0, c_1, c_2 \dots c_{n-1})$ and choose the minimum among

 $F(S) = \min_{i=0...n-1} F(S - c_i) + 1$ subject to $S - c_i \ge 0$

F(S) = 0, when S = 0F(S) = -1, when n = 0

Recursive tree for finding coin change of amount 6 with coin denominations of (1,2,3).

constructed from optimal solutions of its subproblems. How to split the problem into

should be true because of optimal substructure of the problem:

int res = coinChange(idxCoin + 1, coins, amount - x * coins[idxCoin]);

But we don't know which is the denomination of the last coin C. We compute F(S-

Java

1

3

4

5

6 7

8

9

10

11 12

13

14

15 16

17

18 19

20 21 }

Algorithm

5

6 7

8

9

Complexity Analysis

public class Solution {

if (amount < 1) return 0;

if (rem < 0) return -1;

if (rem == 0) return 0;

for (int coin : coins) {

min = 1 + res;

return count[rem - 1];

the memoization table.

i AMOUNT

4

for coin in coins:

Analysis written by: @elmirap.

for x in range(coin, amount + 1):

dp[x] = min(dp[x], dp[x - coin] + 1)

int min = Integer.MAX_VALUE;

if (res >= 0 && res < min)

Therefore there is O(S * n) time complexity.

algorithm F(i) is computed as $\min_{j=0...n-1} F(i-c_j)+1$

5 F(4) F(3) F(2)

F(0)

Intuition

In the recursion tree above, we could see that a lot of subproblems were calculated multiple times. For example the problem F(1) was calculated 13 times. Therefore we should cache the solutions to the subproblems in a table and access them in constant time when necessary Algorithm

The idea of the algorithm is to build the solution of the problem from top to bottom. It applies the idea described above. It use backtracking and cut the partial solutions in the recursive tree, which doesn't lead to a viable solution. This happens when we try to make a change of a coin with a value greater than the amount S. To improve time complexity

we should store the solutions of the already calculated subproblems in a table.

public int coinChange(int[] coins, int amount) {

return coinChange(coins, amount, new int[amount]);

if (count[rem - 1] != 0) return count[rem - 1];

int res = coinChange(coins, rem - coin, count);

count[rem - 1] = (min == Integer.MAX_VALUE) ? -1 : min;

• Time complexity : O(S*n). where S is the amount, n is denomination count. In

algorithm solves only S subproblems because it caches precalculated solutions in a

table. Each subproblem is computed with n iterations, one by coin denomination.

• Space complexity : O(S), where S is the amount to change We use extra space for

the worst case the recursive tree of the algorithm has height of S and the

Approach #3 (Dynamic programming - Bottom up) [Accepted]

For the iterative solution, we think in bottom-up manner. Before calculating F(i), we have to compute all minimum counts for amounts up to i. On each iteration i of the

Copy

F(1) F(0) F(2) F(1) F(0)F(3) F(2) F(1)

F[i]

2

```
F(5) F(4) F(3)
             Finding minimal number of coins for amount i = 6.
                                    F[6] = 2
In the example above you can see that:
               F(3) = \min\{F(3-c_1), F(3-c_2), F(3-c_3)\} + 1
                     = \min\{F(3-1), F(3-2), F(3-3)\} + 1
                     = \min\{F(2), F(1), F(0)\} + 1
                     = \min\{1, 1, 0\} + 1
                     = 1
                                                                          Copy
        Python
 Java
  1
     class Solution:
  2
         def coinChange(self, coins: List[int], amount: int) -> int:
            dp = [float('inf')] * (amount + 1)
  3
            dp[0] = 0
  4
```

45 A V C Share Share SHOW 1 REPLY

numberMumbler # 25 @ November 1, 2017 1:29 AM also hit the time limit using bottom-up DP approach in Python 3 :(~20ms for the failing test case on my local machine 9 A V Share Share SHOW 1 REPLY

return dp[amount] if dp[amount] != float('inf') else -1 Complexity Analysis • Time complexity : O(S*n). On each step the algorithm finds the next F(i) in niterations, where $1 \leq i \leq S$. Therefore in total the iterations are S * n. • Space complexity : O(S). We use extra space for the memoization table.

96 A V C Share Share SHOW 11 REPLIES markov_r # 219 @ January 9, 2019 8:23 PM Good to note that the name of the problem is the Change-Making problem Change problem.

SHOW 3 REPLIES

Rate this article: * * * * * O Previous Next 0 Comments: 71 Sort By -Type comment here... (Markdown is supported)

(1 2 3 4 5 6 7 8 >

Preview Post It's a complete backpack problem. Here is a simplified Python solution - similar like Approach 3: class Solution(object): def coinChange(self, coins, amount): dn = [0] + [float('inf')] * amount Read More

SHOW 7 REPLIES 11 A V Share Share Reply

(https://en.wikipedia.org/wiki/Change-making_problem), the most common variation of Coin Change-Making problem is a variation of the Knapsack problem, more precisely - the Unbounded Read More

i

zhouyuanquaner * 47 ② September 16, 2017 12:13 AM Quick Question, can we sort first use the coin with highest value first? 13 A V C Share Share SHOW 5 REPLIES

gddh 🛊 37 🧿 June 11, 2018 12:19 AM hit time limit with top down approach using python.... Anyone else getting this? 37 A V C Share Share ŧ if amount == Integer.Integer, may this code int max = amount + 1 overflow?

i yajw * 8 @ March 20, 2019 6:55 AM this problem's statement is ambiguous for the upper limit of amount, dp doesn't work if amount is

too large, because of O(amount) space complexity.

8 A V E Share Share SHOW 2 REPLIES SNinja 🛊 5 🔘 September 12, 2018 10:06 AM i Thank you for the explanation. Why space complexity is O(n) in approach #1? It should be O(S) since the number of recursive calls in the worst case is S. 5 A V & Share Reply bluezebra ★ 274 ② November 29, 2017 11:29 AM

i Can anybody explain to me why we need count[rem - 1] = (min == Integer.MAX_VALUE) ? -1: min; ? Thanks 5 A V Share Reply SHOW 1 REPLY

RonanMacF * 20 @ December 16, 2018 11:22 PM i if (amount >= x * coins[idxCoin])

Am I right in saying that this line in the naice approach is not indeed as we already know what for every value of x it will divide into it

3 A V Share Share Reply SHOW 1 REPLY