LeetCode 322

https://leetcode.com/problems/coin-change/description/

Yifeng Zeng

Description

322. Coin Change

Idea Report

The primitive idea is to search all posible solutions and use a variable min to record the minimum value. There are two options, 1. choose the current index, then we look for amount - coins[index] at the index location in the next level of recursion. 2. not choose the current index then we look for the amount at the index + 1 location in the next level of recursion. Once the amount becomes 0, we compair the number of coins used to the global minimum. And if amount is less than 0 or current index is out of boundary, no recursion is further needed so we return.

Code

```
class Solution {
  // TLE
  public int coinChange(int[] coins, int amount) {
      Arrays.sort(coins);
      int[] min = new int[1];
      min[0] = Integer.MAX_VALUE;
      coinChange(coins, amount, 0, 0, min);
      return min[0] == Integer.MAX_VALUE ? -1 : min[0];
  }

  private void coinChange(int[] coins, int amount, int numCoins,
                          int index, int[] min) {
      if (amount == 0) {
          min[0] = Math.min(min[0], numCoins);
          return;
      }
      if (amount < 0 || index >= coins.length) {
```

```
            return;
        }

        // Choose current index
        coinChange(coins, amount - coins[index], numCoins + 1, index, min);
        // Not choose current index
        coinChange(coins, amount, numCoins, index + 1, min);
    }
}
```

We look for how many coins (x) can make the amount, then we can see how many coins (x - 1) can make the amount - coins[i]. Define int[] f, where f[i] is the least coins we need to make up the amount of i. So we are looing for f[amount]. The induction rule would be f[i] = f[i - coins[j]] + 1, where j is from 0 to coins.length - 1 (all different coin denominations we have). The base case is f[0] = 0 because we need 0 coins to make up amount of 0.

Code

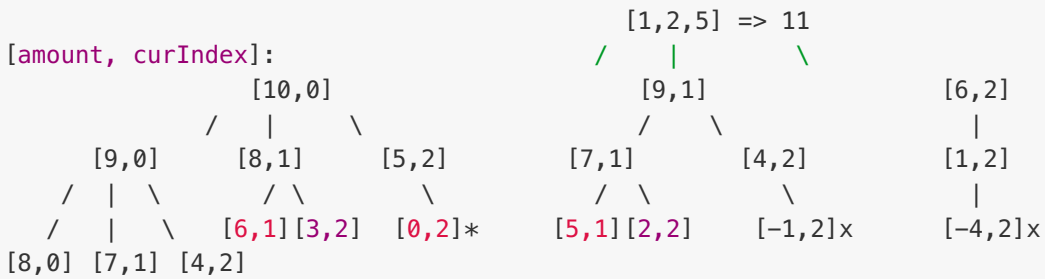
```
public int coinChange(int[] coins, int amount) {
    if (amount <= 0) {
        return 0;
    }
    Arrays.sort(coins);
    int[] f = new int[amount + 1];
    Arrays.fill(f, Integer.MAX_VALUE);
    f[0] = 0;
    for (int i = 1; i <= amount; i++) {
        for (int j = 0; j < coins.length; j++) {
            if (coins[j] <= i && f[i - coins[j]] != Integer.MAX_VALUE) {
                f[i] = Math.min(f[i], f[i - coins[j]] + 1);
            }
        }
    }
    return f[amount] > amount ? -1 : f[amount];
}
```

Summary

- After drawing a solution space tree, a DFS most likely will need some kind of memorization to prune the tree.

Follow up

Use the primitive DFS with no memoization.



Code

```java
class Solution {
    // TLE
    public int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int[] min = new int[1];
        min[0] = Integer.MAX_VALUE;
        helper(coins, amount, 0, 0, min);
        return min[0] == Integer.MAX_VALUE ? -1 : min[0];
    }

    private void helper(int[] coins, int amount, int index,
                        int count, int[] min) {
        if (amount == 0) {
            min[0] = Math.min(min[0], count);
            return;
        }

        if (amount < 0 || index > coins.length) {
            return;
        }

        for (int i = index; i < coins.length; i++) {
            if (amount - coins[i] < 0) {
                break;
            }
            helper(coins, amount - coins[i], i, count + 1, min);
        }
    }
}
```

DFS with memoization.

```java
class Solution {
    // for loop dfs with memo AC
    public int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int[] memo = new int[amount + 1];
        Arrays.fill(memo, Integer.MAX_VALUE);

        return helper(coins, amount, 0, memo);
    }

    private int helper(int[] coins, int amount, int index, int[] memo) {
        if (amount == 0) {
            return 0;
        }
        if (amount < 0 || index > coins.length) {
            return -1;
        }
        if (memo[amount] != Integer.MAX_VALUE) {
            return memo[amount];
        }

        int res = Integer.MAX_VALUE;
        for (int i = 0; i < coins.length; i++) {
            if (amount - coins[i] < 0) {
                break;
            }

            int temp = helper(coins, amount - coins[i], i, memo);
            if (temp != -1) {
                res = Math.min(res, temp + 1);
            }
        }
        memo[amount] = res == Integer.MAX_VALUE ? -1 : res;
        return memo[amount];
    }
}
```