Coin Change

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:
coins = `[1, 2, 5]`, amount = `11`
return `3` (11 = 5 + 5 + 1)

Example 2:
coins = `[2]`, amount = `3`
return `-1`.

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

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int Max = amount + 1;
        vector<int> dp(amount + 1, Max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.size(); j++) {
                if (coins[j] <= i) {
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

written by wyattliu original link here

Solution 2

Recursive Method:

The idea is very classic dynamic programming: think of the last step we take. Suppose we have already found out the best way to sum up to amount `a`, then for the last step, we can choose any coin type which gives us a remainder `r` where `r = a-coins[i]` for all `i`'s. For every remainder, go through exactly the same process as before until either the remainder is 0 or less than 0 (meaning not a valid solution). With this idea, the only remaining detail is to store the minimum number of coins needed to sum up to `r` so that we don't need to recompute it over and over again.

Code in Java:

```
public class Solution {
public int coinChange(int[] coins, int amount) {
    if(amount<1) return 0;
    return helper(coins, amount, new int[amount]);
}

private int helper(int[] coins, int rem, int[] count) { // rem: remaining coins a
fter the last step; count[rem]: minimum number of coins to sum up to rem
    if(rem<0) return -1; // not valid
    if(rem==0) return 0; // completed
    if(count[rem-1] != 0) return count[rem-1]; // already computed, so reuse
    int min = Integer.MAX_VALUE;
    for(int coin : coins) {
        int res = helper(coins, rem-coin, count);
        if(res>=0 && res < min)
            min = 1+res;
    }
    count[rem-1] = (min==Integer.MAX_VALUE) ? -1 : min;
    return count[rem-1];
}
}
```

Iterative Method:

For the iterative solution, we think in bottom-up manner. Suppose we have already computed all the minimum counts up to `sum`, what would be the minimum count for `sum+1`?

Code in Java:

```java
public class Solution {
public int coinChange(int[] coins, int amount) {
    if(amount<1) return 0;
    int[] dp = new int[amount+1];
    int sum = 0;

    while(++sum<=amount) {
        int min = -1;
        for(int coin : coins) {
            if(sum >= coin && dp[sum-coin]!=-1) {
                int temp = dp[sum-coin]+1;
                min = min<0 ? temp : (temp < min ? temp : min);
            }
        }
        dp[sum] = min;
    }
    return dp[amount];
}
}
```

written by GWTW original link here

Solution 3

Java solution: O(amount) space, O(n*amount) time complexity

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if (coins == null || coins.length == 0)
            return -1;

        if (amount <= 0)
            return 0;

        int dp[] = new int[amount + 1];
        for (int i = 1; i < dp.length; i++) {
            dp[i] = Integer.MAX_VALUE;
        }

        for (int am = 1; am < dp.length; am++) {
            for (int i = 0; i < coins.length; i++) {
                if (coins[i] <= am) {
                    int sub = dp[am - coins[i]];
                    if (sub != Integer.MAX_VALUE)
                        dp[am] = Math.min(sub + 1, dp[am]);
                }
            }
        }
        return dp[dp.length - 1] == Integer.MAX_VALUE ? -1 : dp[dp.length - 1];
    }
}
```

written by RomanShakhmanaev original link here