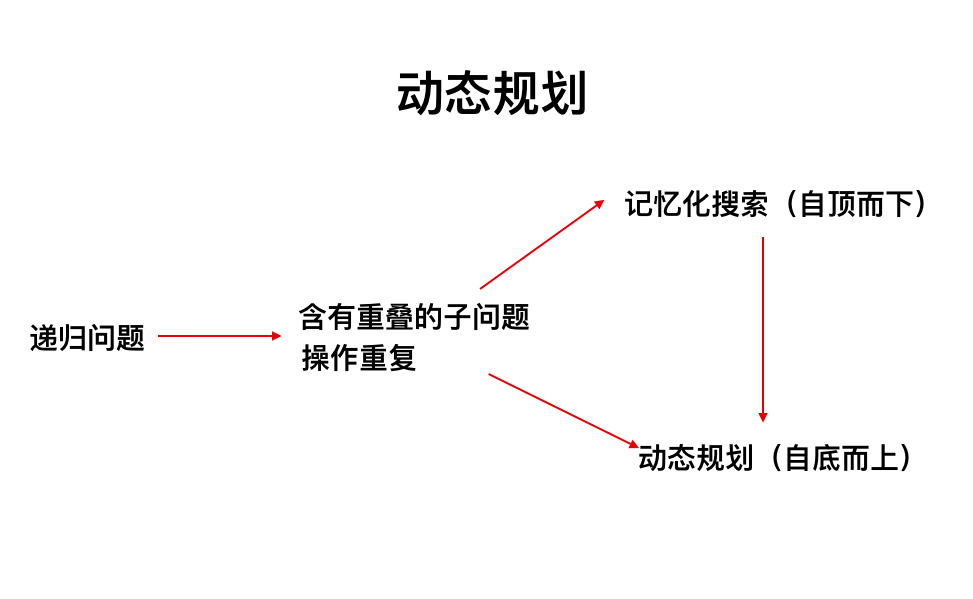
https://assets.leetcode-cn.com/aliyun-lc-upload/users/sugar-31/avatar_1583465537.pngJava 递归、记忆化搜索、动态规划

[sugar](https://leetcode-cn.com/u/sugar-31/)发布于 21 天前5.1k**精选**Java 动态规划 回溯 算法

**解题思路：**

1. 这种找路径，找方法的题一般可以使用回溯法来解决，回溯法也可以说是树形图法，解题的时候使用类似于树状图的结构，使用 **自顶而下** 的方法。
2. 而在回溯法中，如果含有很多的重复的计算的时候，就可以使用记忆化的搜索，将可能出现的重复计算大状态使用一个数组来保存其值，在进行重复的计算的时候，就可以直接的调用数组中的值，较少了不必要的递归。
3. 使用了记忆化搜索后，一般还可以进行优化，在记忆化搜索的基础上，变成 **自底而上** 的动态规划。



**递归**

使用递归的关键是知道递归函数是用来干什么的，从宏观的角度去理解递归。  
直接使用递归超出时间限制

class Solution {

int res = Integer.MAX\_VALUE;

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

if(coins.length == 0){

return -1;

}

findWay(coins,amount,0);

// 如果没有任何一种硬币组合能组成总金额，返回 -1。

if(res == Integer.MAX\_VALUE){

return -1;

}

return res;

}

public void findWay(int[] coins,int amount,int count){

if(amount < 0){

return;

}

if(amount == 0){

res = Math.min(res,count);

}

for(int i = 0;i < coins.length;i++){

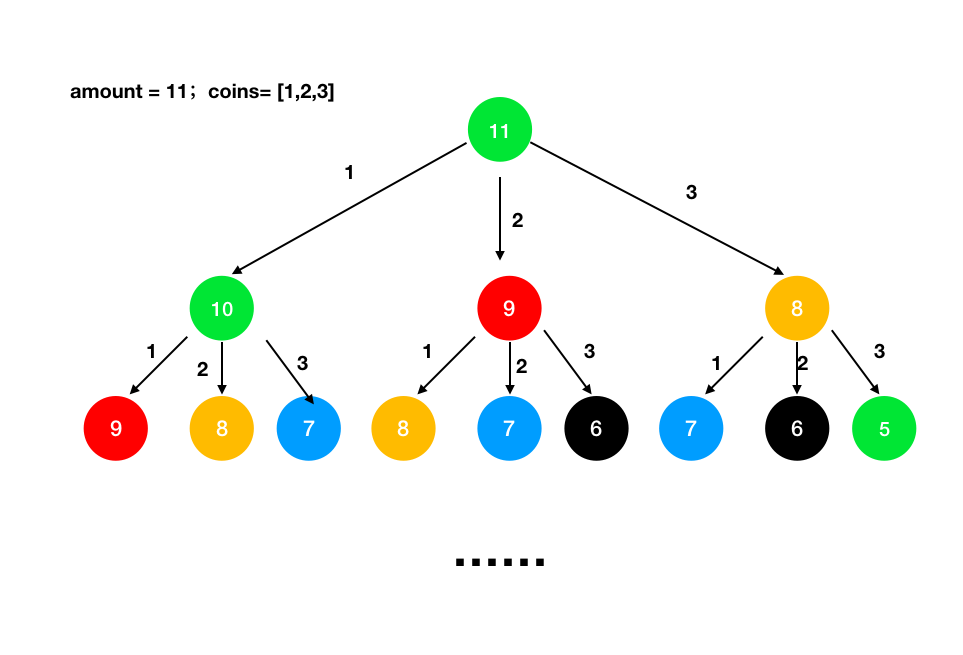
findWay(coins,amount-coins[i],count+1);

}

}

}

**记忆化搜索**



可以看出在进行递归的时候，有很多重复的节点要进行操作，这样会浪费很多的时间。  
使用数组 memo[\ ]*memo*[ ] 来保存节点的值  
memo[n]*memo*[*n*] 表示钱币 n*n* 可以被换取的最少的硬币数，不能换取就为 -1−1  
findWay 函数的目的是为了找到 amount*amount* 数量的零钱可以兑换的最少硬币数量，返回其值 int*int*

在进行递归的时候，memo[n]被复制了，就不用继续递归了，可以直接的调用

class Solution {

int[] memo;

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

if(coins.length == 0){

return -1;

}

memo = new int[amount];

return findWay(coins,amount);

}

// memo[n] 表示钱币n可以被换取的最少的硬币数，不能换取就为-1

// findWay函数的目的是为了找到 amount数量的零钱可以兑换的最少硬币数量，返回其值int

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

if(amount < 0){

return -1;

}

if(amount == 0){

return 0;

}

// 记忆化的处理，memo[n]用赋予了值，就不用继续下面的循环

// 直接的返回memo[n] 的最优值

if(memo[amount-1] != 0){

return memo[amount-1];

}

int min = Integer.MAX\_VALUE;

for(int i = 0;i < coins.length;i++){

int res = findWay(coins,amount-coins[i]);

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

min = res + 1; // 加1，是为了加上得到res结果的那个步骤中，兑换的一个硬币

}

}

memo[amount-1] = (min == Integer.MAX\_VALUE ? -1 : min);

return memo[amount-1];

}

}

**动态规划**

上面的记忆化搜索是先从 memo[amonut-1]*memo*[*amonut*−1] 开始，从上到下  
动态规划从 memo[0]*memo*[0] 开始，从下到上

class Solution {

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

// 自底向上的动态规划

if(coins.length == 0){

return -1;

}

// memo[n]的值： 表示的凑成总金额为n所需的最少的硬币个数

int[] memo = new int[amount+1];

memo[0] = 0;

for(int i = 1; i <= amount;i++){

int min = Integer.MAX\_VALUE;

for(int j = 0;j < coins.length;j++){

if(i - coins[j] >= 0 && memo[i-coins[j]] < min){

min = memo[i-coins[j]] + 1;

}

}

// memo[i] = (min == Integer.MAX\_VALUE ? Integer.MAX\_VALUE : min);

memo[i] = min;

}

return memo[amount] == Integer.MAX\_VALUE ? -1 : memo[amount];

}

}

**另一种实现**

memo[i]*memo*[*i*] 有两种实现的方式，去两者的最小值

1. 包含当前的 coins[i]*coins*[*i*]，那么剩余钱就是 i-coins[i]*i*−*coins*[*i*]，这种操作要兑换的硬币数是 memo[i-coins[j]] + 1*memo*[*i*−*coins*[*j*]]+1
2. 不包含，要兑换的硬币数是 memo[i]*memo*[*i*]

class Solution {

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

// 自底向上的动态规划

if(coins.length == 0){

return -1;

}

// memo[n]的值： 表示的凑成总金额为n所需的最少的硬币个数

int[] memo = new int[amount+1];

// 给memo赋初值，最多的硬币数就是全部使用面值1的硬币进行换

// amount + 1 是不可能达到的换取数量，于是使用其进行填充

Arrays.fill(memo,amount+1);

memo[0] = 0;

for(int i = 1; i <= amount;i++){

for(int j = 0;j < coins.length;j++){

if(i - coins[j] >= 0){

// memo[i]有两种实现的方式，

// 一种是包含当前的coins[i],那么剩余钱就是 i-coins[i],这种操作要兑换的硬币数是 memo[i-coins[j]] + 1

// 另一种就是不包含，要兑换的硬币数是memo[i]

memo[i] = Math.min(memo[i],memo[i-coins[j]] + 1);

}

}

}

return memo[amount] == (amount+1) ? -1 : memo[amount];

}

}