经典动态规划-背包问题

base case 就是 `dp[..][0] = true` 和 `dp[0][..] = false`，因为背包没有空间的时候，就相当于装满了，而当没有物品可选择的时候，肯定没办法装满背包。

0-1背包是否重量能否装满的问题

```
package com.lcz.contest.alibaba;

import java.util.Scanner;

public class Contest_01 {
    // 限重m，个人是wi 恰好等于就上船
    // 恰好背包问题 不重复背人
    public static boolean check(int[] nums,int target) {
        int n = nums.length;
        boolean[][] dp = new boolean[n+1][target+1];
        dp[0][0] = true;

        for(int i=1;i<=n;i++) {
            for(int j=1;j<=target;j++) {
                // 判断能不能放
                if(j>=nums[i-1]) {
                    //可以放 也可以不放
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                }else {
                    // 不能放
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        // 默认
        return dp[n][target];
    }


    public static boolean check_2(int[] nums,int target) {
        int n = nums.length;
        boolean[] dp = new boolean[target+1];
        for(int num:nums) {
            // 开始准备
            for(int j=target;j>=num;j--) {
                dp[j] = dp[j] || dp[j-num];
            }
        }
        return dp[target];
    }
```

```
    // 主函数
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while(in.hasNextInt()) {
            int n = in.nextInt();
            int target = in.nextInt();
            int[] weight = new int[n];
            for(int i=0;i<n;i++) {
                weight[i] = in.nextInt();
            }

            boolean flag = check(weight,target);
            if(flag) {
                System.out.println("YES");
            }else {
                System.out.println("NO");
            }
        }
    }
}
```

0-1背包装满的最大价值问题

```
package com.lcz.contest.alibaba;
// 给你一个可装载重量为W和N个物体，每个物品有重量和质量两个属性
// 其中第i个物体的重量有wt[i] 价值为val[i]
// 最多能装多少
import java.util.*;
public class Contest_03 {
    // 处理
    public static int process(int[] wt,int[] val,int target) {
        //动态规划 选择数组前ig容量为w
        int n = wt.length;
        int[][] dp = new int[n+1][target+1];
        // 初始化最大最小值不需要
        // 对其进行选择
        for(int i=1;i<=n;i++) {
            for(int j=1;j<=target;j++) {
                // 看能否放下去
                if(j>=wt[i-1]) {
                    // 能放下去数组的值存放价值
                    dp[i][j] = Math.max(dp[i-1][j],dp[i-1][j-wt[i-1]] + val[i-
1]);
                }else {
                    // 放不下去
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][n];

    }
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
```

```
        while(in.hasNextInt()) {
            // n=3 w=4 wt=[2,1,3] val=[4,2,3]
            int N = in.nextInt();
            int W = in.nextInt();
            int[] wt = new int[N];
            int[] val = new int[N];
            for(int i=0;i<N;i++) {
                wt[i] = in.nextInt();
            }
            for(int i=0;i<N;i++) {
                val[i] = in.nextInt();
            }
            System.out.println(process(wt,val,W));
        }

    }
}
```

0-1背包存放组合数量问题，从中挑选几个 使其等于某个值，组合数量为多少

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        // 对其找值
        int sum = 0;
        for(int num:nums) {
            sum += num;
        }
        // 找其值
        if((sum+S)%2==1 || sum<S) {
            return 0;
        }
        int target = (sum + S)/2;


        //nums的数组，target为
        int n = nums.length;
        int[][] dp = new int[n+1][target+1];
        // 初始化 如果target为0则代表都弄完了
        for(int i=0;i<=n;i++) {
            dp[i][0] = 1;
        }
        // 开始搜索
        for(int i=1;i<=n;i++) {
            for(int j=0;j<=target;j++) {
                // 看是否能放下
                if(nums[i-1]<=j) {
                    dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i-1]];
                }else {
                    dp[i][j] = dp[i-1][j];
                }
            }
        }

        return dp[n][target];

    }
```

```
}
```

> 也可以用回溯来做

```
class Solution {
    int result = 0;
    public int findTargetSumWays(int[] nums, int S) {
        // 回溯解题
        int index = 0;
        dfs(nums,index,S);
        return result;
    }
    public void dfs(int[]nums,int i,int target){
        if(i==nums.length){
            // 遍历到最后了
            if(target==0){
                result++;
            }
            return;
        }

        // 选择正负号
        target += nums[i];
        // 回溯
        dfs(nums,i+1,target);
        target -= nums[i];


        target -= nums[i];
        dfs(nums,i+1,target);
        target += nums[i];

    }
}
```

完全背包-从中挑选数量使其等于某个数，有多少个?

```
class Solution {
    public int change(int amount, int[] coins) {
        // 组合数据 前coins组成的价格
        int n =coins.length;
        int[][] dp = new int[n+1][amount+1];
        // 初始化 如果amount为0则表示凑满一次了
        for(int i=0;i<=n;i++){
            dp[i][0] = 1;
        }
        for(int i=1;i<=n;i++){
            for(int j=1;j<=amount;j++){
                if(j>=coins[i-1]){
                    dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
```

```
            }
        }
        return dp[n][amount];
    }
}
```

一、动态规划 0-1背包问题（不重复使用）

> 给定一个可装载重量为W的背包和N个物品，每个物品有重量和质量两个属性。其中第i个物体的重量为wt[i], 质量为val[i]. 现在让你用这个背包装物品，最多能装的价值是多少?

```
// W 为背包总体积
// N 为物品数量
// weights 数组存储 N 个物品的重量
// values 数组存储 N 个物品的价值
public int knapsack(int W, int N, int[] weights, int[] values) {
    int[][] dp = new int[N + 1][W + 1];
    for (int i = 1; i <= N; i++) {
        int w = weights[i - 1], v = values[i - 1];
        for (int j = 1; j <= W; j++) {
            if (j >= w) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - w] + v);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[N][W];
}
```

空间优化

```
public int knapsack(int W, int N, int[] weights, int[] values) {
    int[] dp = new int[W + 1];
    for (int i = 1; i <= N; i++) {
        int w = weights[i - 1], v = values[i - 1];
        for (int j = W; j >= 1; j--) {
            if (j >= w) {
                dp[j] = Math.max(dp[j], dp[j - w] + v);
            }
        }
    }
    return dp[W];
}
```

1.1 分割等和子集问题（无顺序 true/false问题 ）

Leetcode416

> 题目：给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
>
> 注意:
>
> 每个数组中的元素不会超过 100
> 数组的大小不会超过 200
> 示例 1:
>
> 输入: [1, 5, 11, 5]
>
> 输出: true
>
> 解释: 数组可以分割成 [1, 5, 5] 和 [11].

> 解题思路：从数组中挑选值 使其满足target的问题。即0-1背包问题

```
class Solution {
    public boolean canPartition(int[] nums) {
        int sum = 0;
        for(int num:nums){
            sum+= num;
        }
        if(sum%2==1){
            return false;
        }


        // 动态规划解题即从数组中挑选数字 使其满足等于target
        int target = sum/2;
        int length = nums.length;
        boolean[][] dp = new boolean[length+1][target+1];
        // 初始化
        for(int i=0;i<=length;i++){
            dp[i][0] = true;
        }
        // 对其选择
        for(int i=1;i<=length;i++){
            for(int j=1;j<=target;j++){
                if(j>=nums[i-1]){
                    // 能放（可放可不放）
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                }else{
                    // 不能放
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[length][target];
    }
}
```

> 对上述代码空间上的优化

```
class Solution {
    public boolean canPartition(int[] nums) {
```

```
        int sum = 0;
        for(int num:nums){
            sum+= num;
        }
        if(sum%2==1){
            return false;
        }


        // 动态规划解题即从数组中挑选数字 使其满足等于target
        int target = sum/2;
        int length = nums.length;
        // 空间上优化
        boolean[] dp = new boolean[target+1];
        // 初始化
        dp[0] = true;
        // 选择
        for(int num:nums){
            for(int j=target;j>=num;j--){
                    dp[j] = dp[j] || dp[j-num];
            }
        }
        return dp[target];
    }
}
```

1.2 目标和问题（ / 组合问题）

Leetcode494

> 题目：给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。
>
> 返回可以使最终数组和为目标数 S 的所有添加符号的方法数。
>
> 示例：
>
> 输入：nums: [1, 1, 1, 1, 1], S: 3
> 输出：5
> 解释：
>
> -1+1+1+1+1 = 3
> +1-1+1+1+1 = 3
> +1+1-1+1+1 = 3
> +1+1+1-1+1 = 3
> +1+1+1+1-1 = 3
>
> 一共有5种方法让最终目标和为3。

> 解题思路：递归

```
class Solution {
    int count = 0;
    public int findTargetSumWays(int[] nums, int S) {
        process(nums,0,0,S);
```

```
        return count;
    }
    public void process(int[] nums,int i,int sum,int S) {
            if(i==nums.length){
                if(sum==S) {
                    count++;
                }
            }else{
                process(nums, i+1, sum+nums[i], S);
                process(nums, i+1, sum-nums[i], S);
            }
    }
}
```

> 解题思路2：将其转为动态规划的问题：
>
> 即X+Y = SUM; X-Y =S; 得出2X =(SUM+S);即X=(SUM+S)/2，即从里面找出整数使其等于（sum+s）/2

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        int sum = 0;
        for(int num:nums){
            sum+=num;
        }
        if(sum<S || (sum+S)%2==1){
            return 0;
        }


        int target = (sum+S)/2;
        int length = nums.length;
        int[] dp = new int[target+1];
        dp[0]    = 1;
        for(int num:nums){
            for(int j=target;j>=num;j--){
                dp[j] = dp[j] + dp[j-num];
            }
        }

        return dp[target];
    }
}
```

1.3 01字符构成最多的字符串(无顺序 最大最小问题)

Leetcode474

> 题目：给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
>
> 请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。
>
> 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。

> 示例 1：
>
> 输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
> 输出：4
> 解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
> 其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3 。
> 示例 2：
>
> 输入：strs = ["10", "0", "1"], m = 1, n = 1
> 输出：2
> 解释：最大的子集是 {"0", "1"} ，所以答案是 2 。

```
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        // 多维0-1背包问题
        if(strs==null ||strs.length==0){
            return 0;
        }

        // 动态规划
        int[][] dp = new int[m+1][n+1];
        for(String s:strs){// 每个只能用一次
            int ones =0, zeros = 0;
            for(char c:s.toCharArray()){
                if(c=='0'){
                    zeros++;
                }else{
                    ones++;
                }
            }
            for(int i=m;i>=zeros;i--){
                for(int j=n;j>=ones;j--){
                    dp[i][j] = Math.max(dp[i][j],dp[i-zeros][j-ones]+1);
                }
            }
        }
        return dp[m][n];
    }
}
```

1.4 0-1背包划分三类

- 恰好target-组合问题的转移公式：

```
dp[i] = dp[i] + dp[i-num];
```

- 恰好target-True/False问题的转移公式：

```
dp[i] = dp[i] || dp[i-num];
```

- 最大target-最大最小问题的转移公式：

```
dp[i] = Math.max(dp[i],dp[i-num]+1)
```

> 恰好问题记得初始化dp[0] = 1 or dp[0] = true;

二、动态规划 完全背包问题 （重复使用）

2.1 零钱兑换（无顺序 最大最小的问题）

Leetcode

> 题目：给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
>
> 你可以认为每种硬币的数量是无限的。
>
> 示例 1：
>
> 输入：coins = [1, 2, 5], amount = 11
> 输出：3
> 解释：11 = 5 + 5 + 1
> 示例 2：
>
> 输入：coins = [2], amount = 3
> 输出：-1
> 示例 3：
>
> 输入：coins = [1], amount = 0
> 输出：0
> 示例 4：
>
> 输入：coins = [1], amount = 1
> 输出：1
> 示例 5：
>
> 输入：coins = [1], amount = 2
> 输出：2

```
class Solution {
    public int coinChange(int[] coins, int amount) {
            int[] dp = new int[amount+1];
            Arrays.fill(dp, amount+1);
            dp[0] = 0;
            for(int coin:coins){
                for(int j=coin;j<=amount;j++){
                    // 可重复使用
                    dp[j] = Math.min(dp[j],dp[j-coin]+1);
                }
            }
            return dp[amount]>amount?-1:dp[amount];
    }
}
```

2.2 零钱兑换II(无顺序 组合问题)

Leetcode

> 题目：给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

> 示例 1:
>
> 输入: amount = 5, coins = [1, 2, 5]
> 输出: 4
> 解释: 有四种方式可以凑成总金额:
> 5=5
> 5=2+2+1
> 5=2+1+1+1
> 5=1+1+1+1+1
> 示例 2:
>
> 输入: amount = 3, coins = [2]
> 输出: 0
> 解释: 只用面额2的硬币不能凑成总金额3。
> 示例 3:
>
> 输入: amount = 10, coins = [10]
> 输出: 1。

```
class Solution {
    public int change(int amount, int[] coins) {
        // 完全背包问题-组合问题
        int[] dp = new int[amount+1];
        dp[0] = 1;


        for(int coin:coins){
            for(int j=coin;j<=amount;j++){
                // 转移方程
                dp[j] = dp[j] + dp[j-coin];
            }
        }

        return dp[amount];
    }
}
```

2.3 单词拆分（有顺序 true/false问题）

Leetcode139

> 给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。
>
> 说明：
>
> 拆分时可以重复使用字典中的单词。
> 你可以假设字典中没有重复的单词。
> 示例 1：
>
> 输入: s = "leetcode", wordDict = ["leet", "code"]
> 输出: true
> 解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
> 示例 2：

> 输入: s = "applepenapple", wordDict = ["apple", "pen"]
> 输出: true
> 解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
> 注意你可以重复使用字典中的单词。
> 示例 3：
>
> 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
> 输出: false

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        // 完全背包问题-有顺序 true/false问题
        int target =s.length();
        boolean[] dp = new boolean[target+1];
        dp[0] = true;

        for(int i=1;i<=target;i++){
            //有顺序
            for(String word:wordDict){
                int len = word.length();
                // 判断
                if(len<=i&&word.equals(s.substring(i-len,i))){
                    dp[i] = dp[i] || dp[i-len];
                }
            }
        }
        return dp[target];
    }
}
```

2.4 组合总和(IV)（有顺序/组合问题）

Leetcode

> 题目：给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。
>
> 示例:
>
> nums = [1, 2, 3]
> target = 4
>
> 所有可能的组合为：
> (1, 1, 1, 1)
> (1, 1, 2)
> (1, 2, 1)
> (1, 3)
> (2, 1, 1)
> (2, 2)
> (3, 1)
>
> 请注意，顺序不同的序列被视作不同的组合。
>
> 因此输出为 7。

```
class Solution {
    public int combinationSum4(int[] nums, int target) {
```

```
        // 完全背包问题-有顺序-组合问题
        int[] dp = new int[target+1];
        dp[0]    = 1;
        for(int i=1;i<=target;i++){// 背包
            for(int num:nums){// 物品放在
                // 进行判断
                if(i>=num){
                    dp[i] = dp[i] + dp[i-num];
                }
            }
        }
        return dp[target];
    }
}
```

三、背包问题整理

先判定是0-1背包问题 还是完全背包问题，以及是否考虑元素之间的顺序。

- 若不考虑元素顺序
  - 0-1背包问题(元素不重复)

    ```
    for num in nums:
        for(int j=target;j>=num;j--){
    ```

  - 完全背包问题(元素可重复使用)

    ```
    for num in nums:
        for(int j=num;j<=target;j++)
    ```

- 若考虑元素顺序(目前只在完全背包问题中遇到了)

  ```
  for(int j=num;j<=target;j++): 先是背包
     for num in nums:后是物体
  ```

问题类型：

- 恰好target-组合问题的转移公式：

```
dp[i] = dp[i] + dp[i-num];
```

- 恰好target-True/False问题的转移公式：

```
dp[i] = dp[i] || dp[i-num];
```

- 最大target-最大最小问题的转移公式：

```
dp[i] = Math.max(dp[i],dp[i-num]+1)
```

> 恰好问题记得初始化dp[0] = 1 or dp[0] = true;
>
> 最大最小问题默认初始化的方式即可