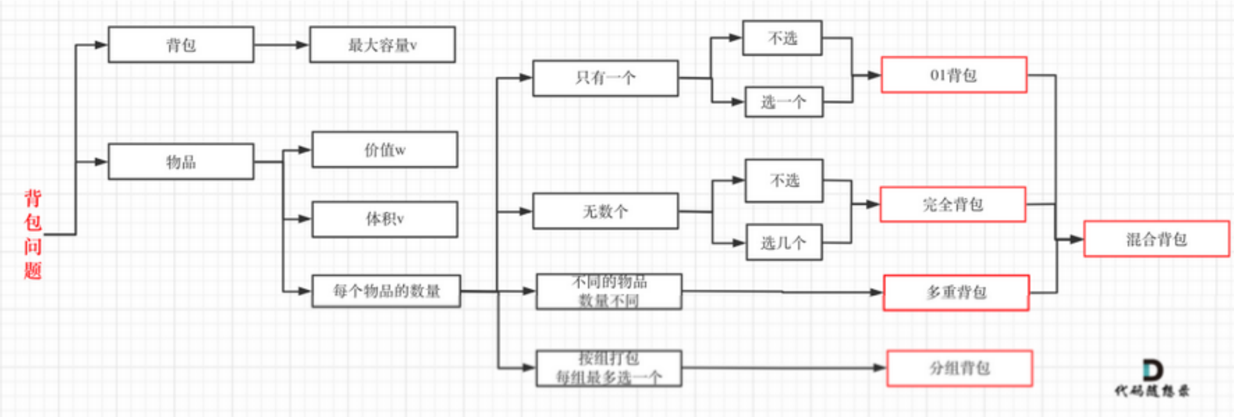
背包问题大总结

背包问题有01背包问题、完全背包问题以及多重背包问题等.



> 完全背包是01背包稍作变化而来的,即:完全背包的**物品数量是无限的**

01背包原理

有N件物品和一个最多能被重量为W 的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。每件物品只能用一次，求解将哪些物品装入背包里物品价值总和最大。

1、确定dp数组以及下标的含义

> dp[i][j]表示从下标为[0-i]的物品里任意取,放进容量为j的背包，价值总和最大。

2、确定递推公式

> 可以有两个方向推出来dp[i][j]
> 由dp[i - 1][j]推出，即背包容量为j，里面不放物品i的最大价值，此时dp[i][j]就是dp[i - 1][j]
> 由dp[i - 1][j - weight[i]]推出，dp[i - 1][j - weight[i]] 为背包容量为j - weight[i]的时候不放物品i的最大价值，那么dp[i - 1][j - weight[i]] + value[i] （物品i的价值），就是背包放物品i得到的最大价值
> 所以dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i]]+value[i]);

3、dp数组如何初始化

> 首先从dp[i][j]的定义触发，如果背包容量j为0的话，即dp[i][0]，无论是选取哪些物品，背包价值总和一定为0。
> dp[0][j]，即：i为0，存放编号0的物品的时候，各个容量的背包所能存放的最大价值。
> if j>=weight[0] then dp[0][j]=value[i];else dp[0][j]=0;

```
for(int j=weight[0];j<=bagWeight;j++) dp[0][j]=value[i];
```

> dp数组的其他下标,由于递推公式是取max所以初始值设为最小,如果物品的价值为负号,初始值为负无穷大;如果物品的价值为正数,初始值为0.

这样才能让dp数组在递归公式的过程中取最大的价值，而不是被初始值覆盖了。

```
//初始化dp
vector<vector<int>> dp(weight.size()+1,vector<int>(bagWeight + 1, 0));
for(int j=weight[0];j<=bagWeight;j++) dp[0][j]=value[0];
```

4、确定遍历顺序

在如下图中,可以看出,有两个遍历的维度：**物品和背包重量**

> 先遍历物品还是先遍历背包重量?
> 当然是先**遍历物品**然后**背包重量**

```cpp
vector<vector<int>> dp(weight.size()+1,vector<int>(bagWeight+1,0));
for(int j=weight[0];j<=bagWeight;j++) dp[0][j]=value[0];
for(int i=1;i<=weight.size();i++){
 for(int j=1;j<=bagWeight;j++){
 if(j>=weight[i]) dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i]+value[i]);
 else dp[i][j]=dp[i-1][j];
 }
}
return dp[weight.size()][bagWeight];

```

> 其实背包问题里，两个for循环的先后循环是非常有讲究,理解遍历顺序其实比推导公式更难理解。

动态规划：关于01背包问题，你该了解这些！（滚动数组）

> 今天我们就说一说滚动数组,其实在前面的题目中我们已经用到过滚动数组了，就是把二维dp降为一维dp。
> 压缩的一般是第一个维度
> 背包问题的压缩如下：

在使用二维数组的时候，递推公式：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);

其实可以发现如果把dp[i - 1]那一层拷贝到dp[i]上，表达式完全可以是：dp[i][j] = max(dp[i][j], dp[i][j - weight[i]] + value[i]);

于其把dp[i - 1]这一层拷贝到dp[i]上，不如只用一个一维数组了，只用dp[j]（一维数组，也可以理解是一个滚动数组）。

滚动数组使用条件:

上一层可以重复利用,直接拷贝到当前层。

在01背包问题中一维dp数组遍历顺序:是从后面到前面遍历

```cpp
for(int i=0;i<weight.size();i++){//遍历物品
 for(int j=bagWeight;j>=0;j--){
 dp[j]=max(dp[j],dp[j-weight[i]]+value[i]);
 }
}
```

为什么从后向前遍历呢?

由于每次递推的时候都是需要左上角的数据，如果从前向后遍历,dp[j]=max(dp[j],dp[j-weight]+value[i])实际上是dp[i][j]=max(dp[i-1][j],dp[i][j-weight]+value[i]);

面试题目：

> 就是本文中的题目，要求先实现一个纯二维的01背包，如果写出来了，然后再问为什么两个for循环的嵌套顺序这么写？反过来写行不行？再讲一讲初始化的逻辑。
>
> 然后要求实现一个一维数组的01背包，最后再问，一维数组的01背包，两个for循环的顺序反过来写行不行？为什么？
>
> 注意以上问题都是在候选人把代码写出来的情况下才问的。
>
> 就是纯01背包的题目，都不用考01背包应用类的题目就可以看出候选人对算法的理解程度了。

416. 分割等和子集

题目链接：https://leetcode-cn.com/problems/partition-equal-subset-sum/

题目难易：中等

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].
示例 2:
输入: [1, 2, 3, 5]
输出: false
解释: 数组不能分割成两个元素和相等的子集.

> 把01背包问题套到本题上来：
> 1、背包的体积为sum/2
> 2、背包要放入的商品(集合里的元素)重量为元素的数值，价值也为元素的数值

> 3、背包如何正好装满，说明找到了总和为sum/2的子集

确定dp数组以及下标的含义

> dp[i]表示背包总容量是i，最大可以凑成i的子集和为dp[i];

确定递推公式

> 01背包的问题是:
> dp[j]=max(dp[j],dp[j-weight[i]]+nums[i]);
> 本题,相当于背包里放入数值,那么物品i的重量是nums[i],其价值也是nums[i]
> 从dp[j]的定义来看，首先dp[0]一定是0。
> 如果如果题目给的价值都是正整数那么非0下标都初始化为0就可以了，如果题目给的价值有负数，那么非0下标就要初始化为负无穷。

确定遍历顺序

```
for(int i=0;i<nums.size();i++){
 for(int j=target;j>=nums[i];j--){
 //对于j小于nums[i]的,此时dp[i][j]=dp[i-1][j];在一维数组里面就根本不需要进行改变
 dp[j]=max(dp[j],dp[j-nums[i]]+nums[i]);
 }
}
```

最后判断结果:

dp[i]的值一定是小于等于i的

就是**dp[target]==taget时**表示可以分为**两个相同的整数解**

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
 int target=0;
 for(int i=0;i<nums.size();i++){
 target+=nums[i];
 }
 //如果sum的和不能为一个整数
 if(target%2==1) return false;
  target=target/2;//先计算一半的和是多少
  //由于nums数组内部的值为正整数,那么dp数组的初始值均设为0
  vector<int> dp(target+100,0);
  for(int i=0;i<nums.size();i++){
  for(int j=target;j>=nums[i];j--){
  dp[j]=max(dp[j],dp[j-nums[i]]+nums[i]);
  }
  }
  return dp[target]==target?true:false;
  }
```

```
20  };
```

1049. 最后一块石头的重量 II

题目链接：https://leetcode-cn.com/problems/last-stone-weight-ii/

题目难度：中等

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：

如果 x == y，那么两块石头都会被完全粉碎；
如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
最后，最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下，就返回 0。

示例：
输入：[2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，

> 由题意可知,
> 只有当x==y时候，那么剩下没有石头了
> 当x!=y时候,那么剩下的石头就是比较重的石头，其重量为y-x
> 转换为将两边的石头分成尽量相同的两组,输出结果为(sum/2-dp[sum/2])*2;
> 因为在执行过程中dp[i]的值一直小于等于i
> 1、确定dp数组及其下标
> dp[j]表示从0....i个石头里面放入到容量为j的背包里面,最大重量。
> 2、递推公式
> dp[j]=max(dp[j],dp[j-nums[i]+nums[i]);
> 3、确定初始化
> 由于石头的重量为正数,那么dp的初始值为0。
> 4、遍历顺序
> 石头的外层遍历顺序为从小到大

> 背包的内层遍历顺序为从大到小

```
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
 int sum=0;
 for(int i=0;i<stones.size();i++){
 sum+=stones[i];
 }
 int target=sum/2;
 vector<int> dp(target+10,0);
  for(int i=0;i<stones.size();i++){
  for(int j=target;j>=stones[i];j--){
  //这个内层遍历是有有意思的
  //当j>=stones[i]时此时背包的容量可以放入stones_i,此时可以从其中选取最大值和最小值
  //当j<stones[i]时没有任何操作那么就等价于dp[i][j]=dp[i-1][j]
  dp[j]=max(dp[j],dp[j-stones[i]]+stones[i]);
  }
  }
  return (sum-dp[target])-dp[target];
     }
};
```

494. 目标和

题目链接：https://leetcode-cn.com/problems/target-sum/

难度：中等

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

> 一方面这个是回溯算法的排序问题,另一方面也是动态规划的01问题
> 其实**如果回溯算法的每个节点的分支**只有**两个可以考虑动态规划01背包问题**或者**二叉树问题**
> **该问题为什么可以用回溯算法因为每个数都要做出一个决定**
> **既然为target,那么就一定有left组合-right组合=target**
> **Letf+Right=sum,而sum是固定的。**
> **那么left-right=left-(sum-left)=target即left=(target+sum)/2;(将加法和减法转换为只有加法了,非常有意思)**

01背包问题的分析

> 这次和之前遇到的背包问题不一样了，之前都是**求容量为j的背包**,**最多能装多少**。
> 现在其实就是一个组合问题了:
> 1、确定**dp数组**以及**下标**的含义
> dp[j]表示:填满j(包括j)这么大容积的包，有dp[i]种方法
> 2、**确定递推公式**
> dp[j]+=dp[j-nums[i]]
> 3、**初始化为0**
> 4、**确定遍历顺序**，外层从小到大，内层从大到小
> 5、**举例推导dp数组**

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
 int sum=0;
 for(int i=0;i<nums.size();i++) sum+=nums[i];
 if(target>sum) return 0;//此时没有方案
 if((sum+target)%2==1) return 0;//此时没有方案,因为此时背包容量不为整数
 int bagSize=(target+sum)/2;
 vector<int> dp(bagSize+1,0);
  //初始化
  dp[0]=1;
  for(int i=0;i<nums.size();i++){
  for(int j=bagSize;j>=nums[i];j--){
  dp[j]+=dp[j-nums[i]];
  }
  }
  return dp[bagSize];
      }
 };
```

474.一和零

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 。

> 如果本题目中只有一个限制条件m个0那么转换为背包问题就是
> 1、确定dp数组及其下标
> dp[j]从下标0...i的元素中选择字符串放入容量为j个0个背包里，此时的最大值
> 2、确定递推公式(此时m个0的限制就是重量,数值的大小就是价值)
> dp[j]=max(dp[j],dp[j-weight[j]]+value[i])

正解

判断是**01背包问题还是完全背包以及多重背包问题的方法**就是根据**物品的数量**以及**选取的个数**选择

> 这不过这个背包有两个维度,一个是m，一个是n，而不同长度的**字符串**就是不同大小的**待装物品**

非常有意思的问题：

> 1、确定dp数组以及下标的含义
> dp[i][j]:最多有i个0和j个1的strs的最大子集的大小为**dp[i][j]。**
> 2、确定递推公式
> dp[i][j]=max(dp[i][j],dp[i-zeroNum][j-oneNum]+1);
> 3、dp数组如何初始化
> 01背包的dp数组初始化为0就可以
> 4、确定遍历顺序
> 外层循环就是表示字符串的编号
> 内层循环就是对zeroNum和oneNum两个维度分别进行遍历

```
class Solution {
```

```
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
 //dp[i][j]表示把字符串数组的中的元素放进i个0,j个1的背包里。
 vector<vector<int>> dp(m+1,vector<int>(n+1,0));
 for(int i=0;i<strs.size();i++){
 int zeroNum=0,oneNum=0;
 for(int p=0;p<strs[i].size();p++){
 if(strs[i][p]=='0') zeroNum++;
  else oneNum++;
  }
  for(int j=m;j>=zeroNum;j--){
  for(int k=n;k>=oneNum;k--){
  dp[j][k]=max(dp[j][k],dp[j-zeroNum][k-oneNum]+1);
  }
  }
  }
  return dp[m][n];
     }
 };
```