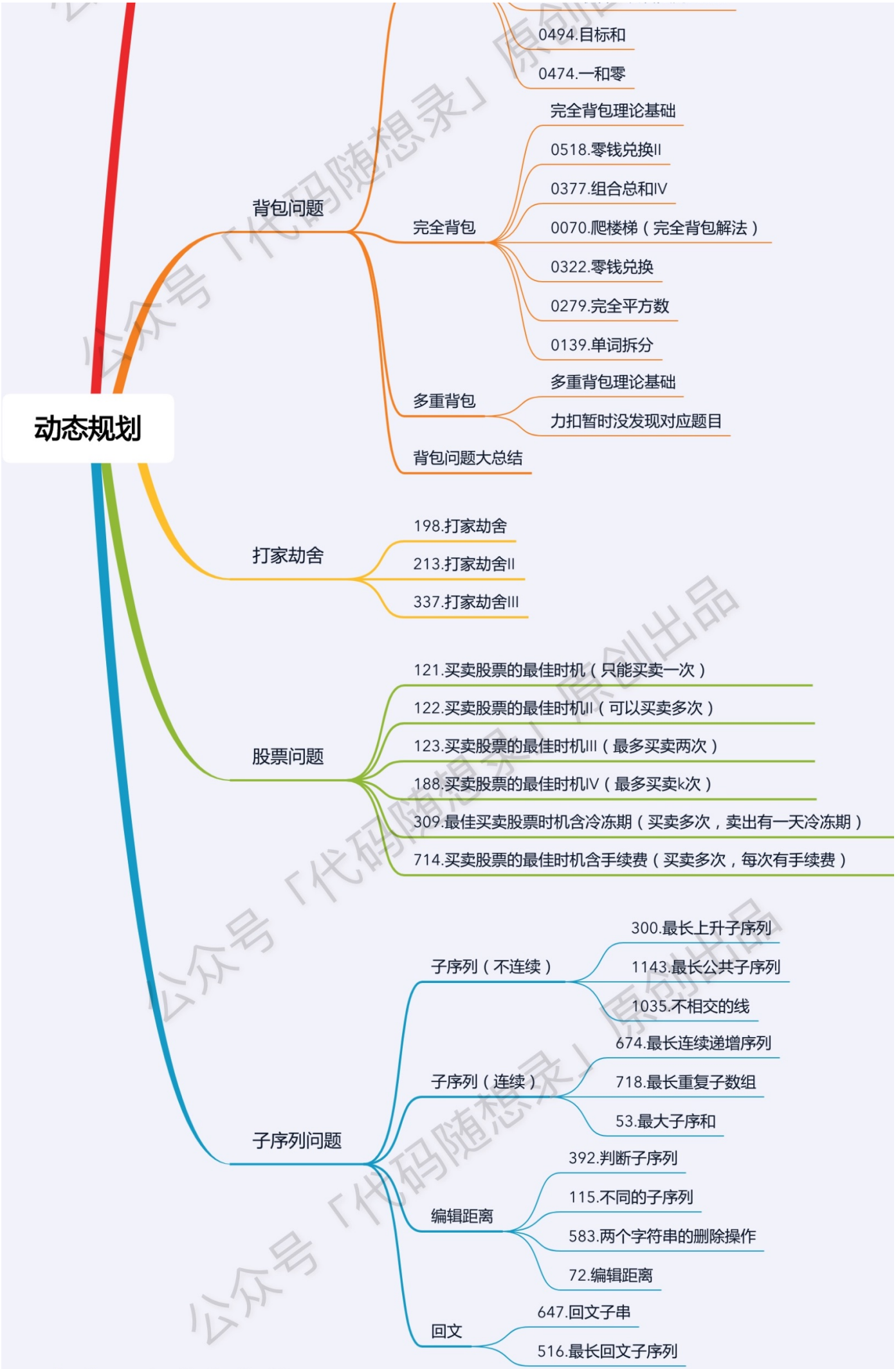
动态规划

动态规划理论基础

动态规划刷题大纲



#什么是动态规划

动态规划，英文：Dynamic Programming，简称DP，如果某一问题有很多重叠子问题，使用动态规划是最有效的。

所以动态规划中每一个状态一定是由上一个状态推导出来的， **这一点就区分于贪心** ，贪心没有状态推导，而是从局部直接选最优的，

在关于贪心算法，你该了解这些！ **(opens new window)**中我举了一个背包问题的例子。

例如：有N件物品和一个最多能背重量为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。

动态规划中dp[j]是由dp[j-weight[i]]推导出来的，然后取max(dp[j], dp[j - weight[i]] + value[i])。

但如果是贪心呢，每次拿物品选一个最大的或者最小的就完事了，和上一个状态没有关系。

所以贪心解决不了动态规划的问题。

其实大家也不用死扣动规和贪心的理论区别，后面做做题目自然就知道了 。

而且很多讲解动态规划的文章都会讲最优子结构啊和重叠子问题啊这些，这些东西都是教科书的上定义，晦涩难懂而且不实用。

大家知道动规是由前一个状态推导出来的，而贪心是局部直接选最优的，对于刷题来说就够用了。

上述提到的背包问题，后序会详细讲解。

#动态规划的解题步骤

做动规题目的时候，很多同学会陷入一个误区，就是以为把状态转移公式背下来，照葫芦画瓢改改，就开始写代码，甚至把题目AC之后，都不太清楚dp[i]表示的是什么。

这就是一种朦胧的状态，然后就把题给过了，遇到稍稍难一点的，可能直接就不会了，然后看题解，然后继续照葫芦画瓢陷入这种恶性循环中 。

状态转移公式（递推公式）是很重要，但动规不仅仅只有递推公式。

对于动态规划问题，我将拆解为如下五步曲，这五步都搞清楚了，才能说把动态规划真的掌握了！

1. 确定dp数组（dp table）以及下标的含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

一些同学可能想为什么要先确定递推公式，然后在考虑初始化呢？

因为一些情况是递推公式决定了dp数组要如何初始化！

后面的讲解中我都是围绕着这五点来进行讲解。

可能刷过动态规划题目的同学可能都知道递推公式的重要性，感觉确定了递推公式这道题目就解出来了。

其实 确定递推公式 仅仅是解题里的一步而已！

一些同学知道递推公式，但搞不清楚dp数组应该如何初始化，或者正确的遍历顺序，以至于记下来公式，但写的程序怎么改都通过不了。

后序的讲解的大家就会慢慢感受到这五步的重要性了。

#动态规划应该如何debug

相信动规的题目，很大部分同学都是这样做的。

看一下题解，感觉看懂了，然后照葫芦画瓢，如果能正好画对了，万事大吉，一旦要是没通过，就怎么改都通过不了，对 dp数组的初始化，递推公式，遍历顺序，处于一种黑盒的理解状态。

写动规题目，代码出问题很正常！

找问题的最好方式就是把dp数组打印出来，看看究竟是不是按照自己思路推导的！

一些同学对于dp的学习是黑盒的状态，就是不清楚dp数组的含义，不懂为什么这么初始化，递推公式背下来了，遍历顺序靠习惯就是这么写的，然后一鼓作气写出代码，如果代码能通过万事大吉，通过不了的话就凭感觉改一改。

这是一个很不好的习惯！

做动规的题目，写代码之前一定要把状态转移在dp数组的上具体情况模拟一遍，心中有数，确定最后推出的是想要的结果 。

然后再写代码，如果代码没通过就打印dp数组，看看是不是和自己预先推导的哪里不一样。

如果打印出来和自己预先模拟推导是一样的，那么就是自己的递归公式、初始化或者遍历顺序有问题了。

如果和自己预先模拟推导的不一样，那么就是代码实现细节有问题。

这样才是一个完整的思考过程，而不是一旦代码出问题，就毫无头绪的东改改西改改，最后过不了，或者说是稀里糊涂的过了 。

这也是我为什么在动规五步曲里强调推导dp数组的重要性。

举个例子哈：在「代码随想录」刷题小分队微信群里，一些录友可能代码通过不了，会把代码抛到讨论群里问：我这里代码都已经和题解一模一样了，为什么通过不了呢？

发出这样的问题之前，其实可以自己先思考这三个问题：

- 这道题目我举例推导状态转移公式了么？
- 我打印dp数组的日志了么？
- 打印出来了dp数组和我想的一样么？

如果这灵魂三问自己都做到了，基本上这道题目也就解决了 ，或者更清晰的知道自己究竟是哪一点不明白，是状态转移不明白，还是实现代码不知道该怎么写，还是不理解遍历dp数组的顺序。

然后在问问题，目的性就很强了，群里的小伙伴也可以快速知道提问者的疑惑了。

注意这里不是说不让大家问问题哈， 而是说问问题之前要有自己的思考，问题要问到点子上！

大家工作之后就会发现，特别是大厂，问问题是一个专业活，是的，问问题也要体现出专业！

如果问同事很不专业的问题，同事们会懒的回答，领导也会认为你缺乏思考能力，这对职场发展是很不利的。

所以大家在刷题的时候，就锻炼自己养成专业提问的好习惯。

#总结

这一篇是动态规划的整体概述，讲解了什么是动态规划，动态规划的解题步骤，以及如何debug。

动态规划是一个很大的领域，今天这一篇讲解的内容是整个动态规划系列中都会使用到的一些理论基础。

在后序讲解中针对某一具体问题，还会讲解其对应的理论基础，例如背包问题中的01背包，leetcode上的题目都是01背包的应用，而没有纯01背包的问题，那么就需要在把对应的理论知识讲解一下。

大家会发现，我讲解的理论基础并不是教科书上各种动态规划的定义，错综复杂的公式。

这里理论基础篇已经是非常偏实用的了，每个知识点都是在解题实战中非常有用的内容，大家要重视起来哈。

刷题

509. 斐波那契数 - 力扣（Leetcode）

```
class Solution {
public:
    vector<int> fibNum;
    int fib(int n) {
        fibNum.push_back(0);
        fibNum.push_back(1);
        for (int i=2;i<=n;i++)
        {
            fibNum.push_back(fibNum[i-1]+fibNum[i-2]);
        }
        return fibNum[n];
    }

};
```

对比一下卡哥写的代码，我的代码可能有一些问题吧，但是这样还是可以接收

```
class Solution {
public:
    int fib(int N) {
        if (N <= 1) return N;
        vector<int> dp(N + 1);
        dp[0] = 0;
        dp[1] = 1;
```

```
        for (int i = 2; i <= N; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[N];
    }
};
```

用动态规划进行求解，这是一种典型的以空间去换时间的方式，这样还是非常优秀的。

70. 爬楼梯 - 力扣（Leetcode）

这个问题还是已经反复做过无数遍了，这里我直接写出来就没问题了
1.dp数组的含义，dp[i]代表到达第i个台阶有多少种方式
2.确定递推公式 dp[i]=dp[i-1]+dp[i-2]
3.如何初始化 dp[0]=1 dp[1]=1
4.确定遍历顺序，一维数组，哪里来的顺序，直接遍历就行
5.自行推导尝试。

```
#include<iostream>
#include <vector>
using namespace std;
class Solution {
public:
    vector<int> dp;
    int climbStairs(int n) {
        dp.push_back(1);
        dp.push_back(1);
        for(int i=2;i<=n;i++)
        {
            dp.push_back(dp[i-1]+dp[i-2]);
        }
        return dp[n];
    }
};
```

这里可以轻松求解出来答案，也就没有更多的问题了。

746. 使用最小花费爬楼梯 - 力扣（Leetcode）

1.确定dp数组的含义，这里很简单就可以确定出其含义，dp[i]就是到达第i级的最少消费
2.确定递推公式，dp[i]=min{dp[i-1]+cost[i-1],dp[i-2]+cost[i-2]}
3.如何初始化dp[0]=0 dp[1]=0
4.确定遍历顺序，一维的不用遍历
5.自行推导尝试，这里也不用尝试。
解决方法已经熟练到不行了，这个就这样美好的通过了。

```
#include<iostream>
#include <vector>
using namespace std;
class Solution {
```

```
public:
    vector<int> dp;
    int minCostClimbingStairs(vector<int>& cost) {
        dp.push_back(0);
        dp.push_back(0);
        for(int i=2;i<=cost.size();i++)
        {
            dp.push_back(min(dp[i-1]+cost[i-1],dp[i-2]+cost[i-2]));
        }
        return dp[cost.size()];
    }
};
```

62. 不同路径 - 力扣（Leetcode）

这里其实可以看到其实是一种深度优先搜索的思路，但是我们学习了动态规划，就可以使用动态规划的方式进行求解。
1.确定dp数组的含义，dp[i][j]代表到达i,j格子的时候的路线的数量。
2.确定递推公式，由于机器人只能向右或者向下一步，那么我们可以知道dp[i][j]=dp[i-1][j]+dp[i][j-1]
3.初始化，首先到达[0][0]位置的方式为1，之后的[0][1]位置的方式为1，到达[1][0]位置的方式为1
4.确定遍历顺序，按照行或者列遍历都可以，那么我们就按照行进行遍历了
5.自行推导尝试
出现了一些小问题，这里的问题在于初始化的问题，还是需要历练啊

```
class Solution {
public:
    vector<vector<int>> dp;
    int uniquePaths(int m, int n) {
        dp=vector<vector<int>>(100,vector<int>(100,0));
        for(int i=0;i<n;i++)
        {
            dp[0][i]=1;
        }
        for(int i=0;i<m;i++)
        {
            dp[i][0]=1;
        }
        for(int i=1;i<m;i++)
        {
            for(int j=1;j<n;j++){
                dp[i][j]=dp[i][j-1]+dp[i-1][j];
            }
        }
        return dp[m-1][n-1];
    }
};
```

63. 不同路径 II - 力扣（Leetcode）

这里本质上其实跟上面那个题是一样的逻辑，这里直接使用卡哥得代码放到这里了，我就不写了

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        if (obstacleGrid[m - 1][n - 1] == 1 || obstacleGrid[0][0] == 1) //如果在起点或终点出现了障碍，直接返回0
            return 0;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) dp[i][0] = 1;
        for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) dp[0][j] = 1;
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (obstacleGrid[i][j] == 1) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

343. 整数拆分 - 力扣（Leetcode）

1.首先确认dp数组的含义 dp[i]代表正整数i拆分成的两个整数的乘积最大值。
2.确认递推公式 首先第i项可以拆成dp[i-j] * j 这里采用遍历的方式就可以进行，或者就可以拆成j*(i-j) 这样就确认了方法，同时取两个值里面的最大值就可以求解了。
3.如何初始化 首先可以确定的是dp[0]=0 dp[1]=0 dp[2]=1 dp[3]=2 ... 剩下的可以利用递推公式进行计算了
4.确定遍历顺序 不需要确定 就只有一个未知数
5.自行推导

下面的代码还是一遍通过了，当然这个题还是非常简单的（看过代码随想录）

```
class Solution {
public:

    int integerBreak(int n) {
        //init
        vector<int> dp(60,0);
        dp[0]=0;
        dp[1]=0;
        dp[2]=1;

        //calculate
        //i代表dp的索引
        //j代表进行遍历使用的变量
        for (int i =3;i<=n;i++)
        {
            for(int j=0;j<=i;j++)
            {
                int tmp1=j*(i-j);
                int tmp2=j*dp[i-j];
```

```
                //返回三者之间的最大值
                dp[i]=dp[i]>(tmp1>tmp2?tmp1:tmp2)?dp[i]:(tmp1>tmp2?tmp1:tmp2);
            }
        }
        return dp[n];
    }
};
```

卡哥说这里也可以使用贪心，代码如下：

```
class Solution {
public:
    int integerBreak(int n) {
        if (n == 2) return 1;
        if (n == 3) return 2;
        if (n == 4) return 4;
        int result = 1;
        while (n > 4) {
            result *= 3;
            n -= 3;
        }
        result *= n;
        return result;
    }
};
```

※96. 不同的二叉搜索树 - 力扣（Leetcode）

1.确定dp数组的含义　dp[i]的含义是当有i个节点的时候有多少种二叉搜索树
2.确定递推公式　这里的思路也是不太好想到的结果，这个题相对困难，这里先放到后面再回来复习的时候进行
3.初始化 这里也不用想 dp[0]=0 dp[1]=1 dp[2]=2
4.确定遍历顺序
5.自行推导尝试

0-1背包问题

这里为什么没有链接呢，不是我懒昂，因为leetcode上面只有背包问题的应用，并没有简单的背包问题，这里直接放一段完整的代码，同时在这里我也学习一下优化空间的结构，同时将老师上课优化的代码也在这里放出来

练习到这里，我们总结出一个规律，作为dp[i][j][k]中这些索引的变量是限制变量，也就是限制我们的结果，而装入的值是我们说的所谓的最优值，这里也就是动态规划dp数组确定的根本方式。

为什么叫做0-1背包问题?
假如我们把一个物品装入背包记为1，不装入记为0，那么此时也就是0-1背包问题的由来，也就是放入的物品不能重复。
一、普通的形态（基本形态）
1.这是一个二维的问题，我们也不多说直接开始确定dp[i][j]的含义，含义是装入物品i同时背包的容量为j的时候最大的价值，为什么采用二维，我们要达到的结果是价值最大，也就是dp[i][j]里面放入的内容是价值最大，而价值又是跟物体重量以及背包容量绑定的。
2.确定递推关系，dp[i][j]=max(dp[i][j-weight[i]] +value[i] , dp[i-1][j])
3.初始化，首先对于dp[0]这一行，也就是此时对于j>=weight[0] 的时候才需要装入，此时价值就是value[0]，其余的均初始化为0
4.行列扫描应该都是可以的，咱们只需要保证j-weight[i]已经填入数据就可以了，但是当然需要对于大于>weight[i]的情况，这样才能说行列扫描都可以
5.自行推导。
代码如下：

```
#include<iostream>
#include<vector>
using namespace std;

class Solution {
public:
    static int backPack(vector<int> values,vector<int> weights,int n) {
        // init
        vector<vector<int>> dp(values.size(),vector<int>(n+1,0));

        for (int j=0;j<=n;j++)
        {
            if(j>=weights[0])
            {
                dp[0][j]=values[0];
            }
        }

        for(int i=1;i<values.size();i++)
        {
            for(int j=0;j<n+1;j++)
            {
                if(j<weights[i])
                {
                    dp[i][j]=dp[i-1][j];
                }else
                {
                    dp[i][j]=max(dp[i-1][j],dp[i][j-weights[i]]+values[i]);
                }
            }
        }
        return dp[values.size()-1][n];
    }
};

int main(int argc, char* argv[])
{
    int result=Solution::backPack({1,2,3},{10,20,30},40);
    cout<<result;
    return 0;
}
```

二、空间优化形态
其实这个我也考虑了一下，其实上层完全可以使用下层的东西，因为下层无论如何也不过就是dp[i][j-weight[i]]+values[i] 或者是 dp[i-1][j] 那么我们第二种情况也必然是寻找上面一行的某一个值进行拷贝，这样其实并没有太多的问题，那我们如果弄成一行的话，也不过是在当前行寻找所有的东西，是完全没有问题的，不多说了，直接开整

```
#include<iostream>
#include<vector>
using namespace std;

class Solution {
public:
    static int backPack(vector<int> values,vector<int> weights,int n) {
        //背包问题空间优化
        vector<int> dp(100,0);//进行初始化
        for(int i=0;i<=n;i++)
        {
            if(i>=weights[0])
            {
                dp[i]=values[0];
            }
        }

        for(int i=0;i<values.size();i++)
        {
            for(int j=0;j<=n;j++)
            {
                if(j>=weights[i])
                {
                    dp[j]=dp[j]>dp[j-weights[i]]+values[i] ? dp[j] : dp[j-
weights[i]]+values[i];
                }
            }
        }
        return dp[n];
    }
};

int main(int argc, char* argv[])
{
    int result=Solution::backPack({1,2,3},{10,20,30},40);
    cout<<result;
    return 0;
}
```

这里就求解完成了
三、数学计算之后进行优化的形式
最后这种优化的方式个人感觉没怎么好，这个之后再进行补充吧，今天要往后做了

416. 分割等和子集 - 力扣（Leetcode）

这里使用回溯法可以解决，但是我们这里还没有学习到，所以我们这里利用动态规划做一做看看
本质上这个是一个0-1背包问题，来试一试
我只看了一点就知道思路了，害，看来还是需要多练习啊，
这里重要的一个点是，我们要的结果是sum/2，这里也可以认为是我们的背包容量，好了，不多说，直接上代码看看。

做到这个题确实有点平复不了心情了，做的挺差的，做的还是太少了，还是要多看才知道

```
#include<iostream>
#include <numeric>
#include<vector>
using namespace std;

class Solution {
public:

    bool canPartition(vector<int>& nums) {
        int sum=0;
        vector<int> dp(10001,0);
        for(int i=0;i<nums.size();i++)
        {
            sum+=nums[i];
        }
        //背包容量
        if(sum%2==1) return false;
        int target=sum/2;
        // 开始 01背包
        for(int i = 0; i < nums.size(); i++) {
            for(int j = target; j >= nums[i]; j--) { // 每一个元素一定是不可重复放入，所以从大到小遍历
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }
        // 集合中的元素正好可以凑成总和target
        if (dp[target] == target) return true;
        return false;
    }
};
```

1049. 最后一块石头的重量 II - 力扣（Leetcode）

1.确定dp数组的含义
2.确定递推公式
3.初始化
4.确定遍历顺序
5.自行推导尝试

这里其实就是将石头分成两堆，然后相减的算法，如下：

```
#include<iostream>
#include <numeric>
#include<vector>
using namespace std;
/*
 * 最后一块石头的重量 这里修正了无数次啊 真的折磨，需要考虑的部分还是很多的，幸好
 */
class Solution {
public:
    int lastStoneWeightII(vector<int>& nums) {
        if(nums.size()==1)
        {
            return nums[0];
```

```
        }
        if(nums.size()==2)
        {
            return abs(nums[0]-nums[1]);
        }
        int sum=0;
        for (auto iter : nums)
        {
            sum+=iter;
        }
        //目标是达到有和可以是sum/2
        int target;
        if(sum%2==0)
        {
            target=sum/2;
        }else
        {
            target=sum/2+1;
        }


        vector<vector<int>> dp(nums.size(),vector<int>(sum/2+2,0));

        for(int i=0;i<=sum/2;i++)
        {
            if(i>=nums[0])
            {
                dp[0][i]=nums[0];
            }
        }

        for (int i=1;i<nums.size();i++)
        {
            for(int j=0;j<=sum/2+1;j++)
            {
                if(j<=nums[i])
                {
                    dp[i][j]=dp[i-1][j];
                }else
                {
                    dp[i][j]=max(dp[i-1][j],dp[i-1][j-nums[i]]+nums[i]);
                }
            }
        }
        return abs(sum-2*dp[nums.size()-1][target]);
    }
};

int main(int argc, char* argv[])
{

    vector<int> test={1,1,4,2,2};
    Solution s;
    int result=s.lastStoneWeightII(test);
    cout<<result;
    return 0;
```

```
}
```

494. 目标和 - 力扣（Leetcode）

这里其实本质上还是一个背包问题的变式，我们其实可以将一个这个题目拆成一个数组的一部分减去另外一个部分等于某一个确定的值，那么我们此时设我们需要等于的这个值为target，我们取+符号的和为left，这样我们可以有left-right=target，而对于我们拥有的所有变量的和我们设置成为sum，这样right=sum-left，则我们left-sum+left=target，这样我们可以得到推导的形式是left=(target+sum)/2，这样我们就将题目又转换成为对于一组数，其和等于(target+sum)/2的数量，和我们上面的题目如出一辙

```
#include<iostream>
#include <numeric>
#include<vector>
using namespace std;
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum=0;//求解一下结果
        for (auto iter:nums)
        {
            sum+=iter;
        }
        if((target+sum)%2!=0) return 0;//此时我们就是无法做到达到
        //这里是背包的容量
        int result= (target+sum)/2;
        if(result>sum) return 0;//此时如果我们要求解的值大于我们能达到的值，那么我们就无法达到，则直接返回
        if(result<0) return 0;
        vector<int> dp(result+1,0);//这里打算利用一次滚动数组让算法更加方便


        //初始化
        dp[0]=1;
        for (int i=0;i<nums.size();i++)
        {
            for(int j=result;j>=nums[i];j--)
            {
                dp[j]+=dp[j-nums[i]];
            }
        }
        return dp[result];
    }
};
int main(int argc, char* argv[])
{
    Solution s;
    vector<int> test={0,0,0,0,0,0,0,0,1};
    int result=s.findTargetSumWays(test,1);
    cout<<result;
```

```
    return 0;
}
```

小总结1

通过上面几个题目的练习，我们可以看出，可以使用背包问题模板的几种情况，首先，就是对于一般的背包问题，放入其中的物品是有二维属性的，例如重量和价值，第二种是对于我们后面数组的数字计算那几种类型，这种类型主要是针对于题目可以等价于将数组拆分成两个子数组差值的情况，或者是子数组等于某个特定值的情况，此时将我们需要达到的目标值作为背包的容量，同时放入的物品的重量也是数组中元素的大小，又是其价值，这样也达成了二维的限制条件。

这个题明天再复习一下子，实在是自己跟傻子一样，啥都不会

474. 一和零 - 力扣（Leetcode）

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>> dp(m+1,vector<int>(n+1,0));
        for(string str:strs)
        {
            int zeroNum=0,oneNum=0;
            for(char c:str)
            {
                if(c=='0')  zeroNum++;
                else oneNum++;
            }
            for(int i=m;i>=zeroNum;i--)
            {
                for(int j=n;j>=oneNum;j--)
                {
                    dp[i][j]=max(dp[i][j],dp[i-zeroNum][j-oneNum]+1);
                }
            }
        }
        return dp[m][n];
    }
};
```

完全背包问题

这个问题其实也就是某一天我骑车的时候思考的问题，如何保证添加的一定是一个物品呢，对于0-1背包问题，这个思考了很久，最后发现，只要我们从后向前遍历就能完成这个了，但是如果我们从前向后遍历的话，那我们就是个完全背包问题了，因为此时我们不再需要保证，我们每次只添加一个物品，这样的话，我们就解决了我们的问题，这里直接用卡哥网站上的内容来。

完全背包

有N件物品和一个最多能背重量为W的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。 **每件物品都有无限个（也就是可以放入背包多次）** ，求解将哪些物品装入背包里物品价值总和最大。

完全背包和01背包问题唯一不同的地方就是，每种物品有无限件 。

同样leetcode上没有纯完全背包问题，都是需要完全背包的各种应用，需要转化成完全背包问题，所以我这里还是以纯完全背包问题进行讲解理论和原理。

在下面的讲解中，我依然举这个例子：

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

每件商品都有无限个！

问背包能背的物品最大价值是多少？

01背包和完全背包唯一不同就是体现在遍历顺序上，所以本文就不去做动规五部曲了，我们直接针对遍历顺序经行分析！

关于01背包我如下两篇已经进行深入分析了：

- 动态规划：关于01背包问题，你该了解这些！(opens new window)
- 动态规划：关于01背包问题，你该了解这些！（滚动数组）(opens new window)

首先在回顾一下01背包的核心代码

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

我们知道01背包内嵌的循环是从大到小遍历，为了保证每个物品仅被添加一次。

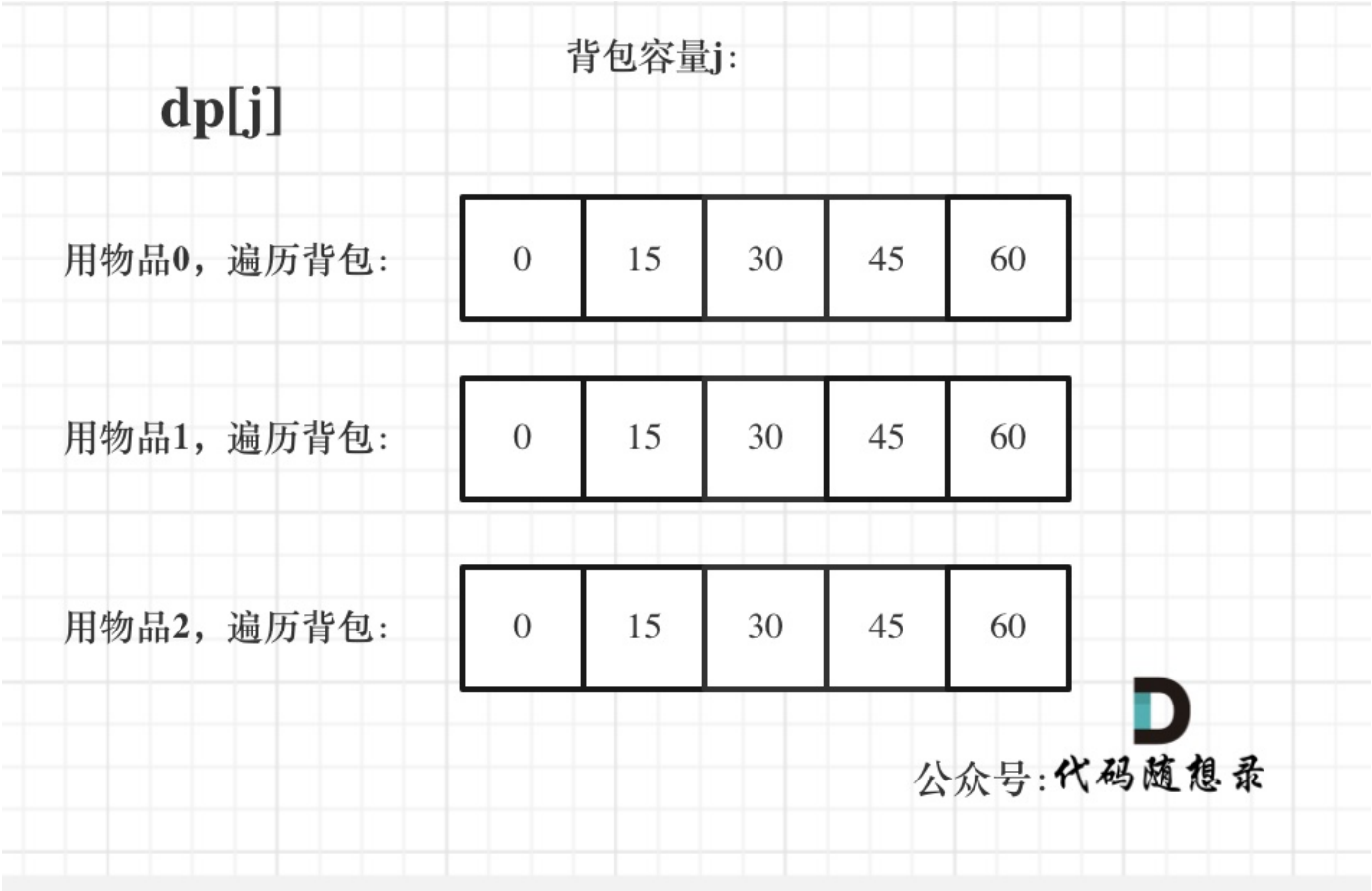
而完全背包的物品是可以添加多次的，所以要从小到大去遍历，即：

```
// 先遍历物品，再遍历背包
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j <= bagWeight ; j++) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

    }
}
```

至于为什么，我在动态规划：关于01背包问题，你该了解这些！（滚动数组） **(opens new window)**中也做了讲解。

dp状态图如下：



相信很多同学看网上的文章，关于完全背包介绍基本就到为止了。

其实还有一个很重要的问题，为什么遍历物品在外层循环，遍历背包容量在内层循环？

这个问题很多题解关于这里都是轻描淡写就略过了，大家都默认 遍历物品在外层，遍历背包容量在内层，好像本应该如此一样，那么为什么呢？

难道就不能遍历背包容量在外层，遍历物品在内层？

看过这两篇的话：

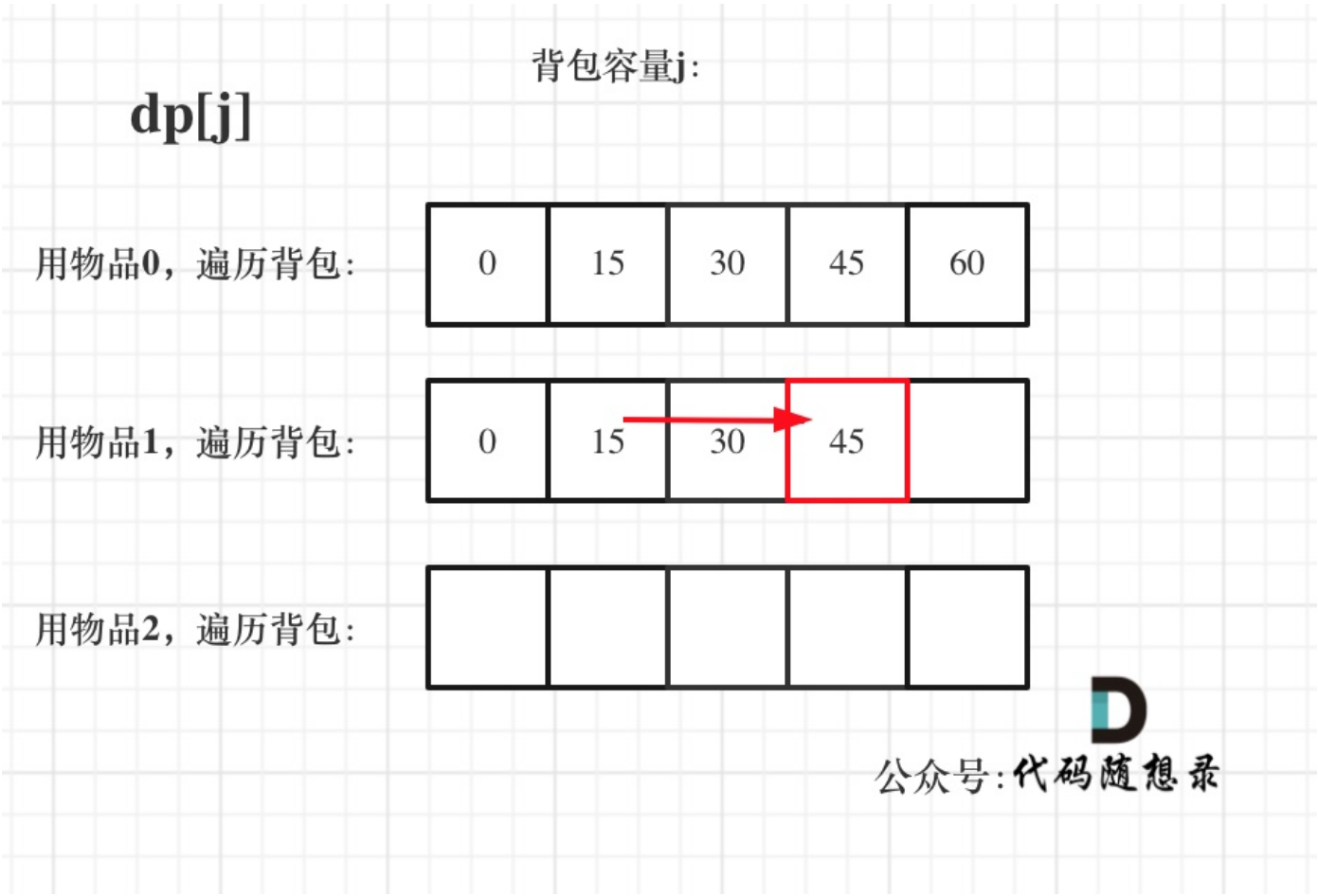
- 动态规划：关于01背包问题，你该了解这些！(opens new window)
- 动态规划：关于01背包问题，你该了解这些！（滚动数组）(opens new window)

就知道了，01背包中二维dp数组的两个for遍历的先后循序是可以颠倒了，一维dp数组的两个for循环先后循序一定是先遍历物品，再遍历背包容量。

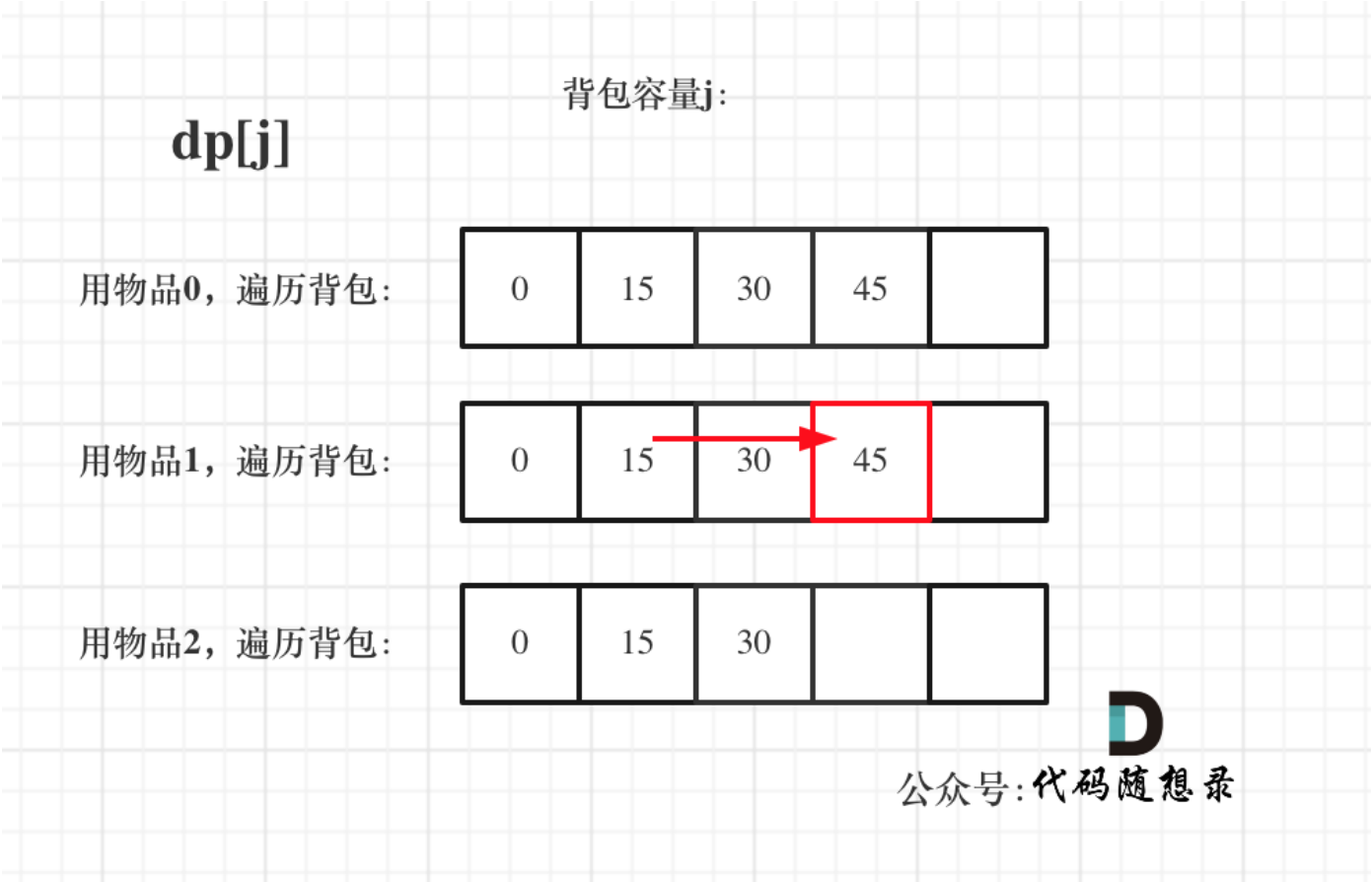
在完全背包中，对于一维dp数组来说，其实两个for循环嵌套顺序同样无所谓！

因为dp[j] 是根据 下标j之前所对应的dp[j]计算出来的。 只要保证下标j之前的dp[j]都是经过计算的就可以了。

遍历物品在外层循环，遍历背包容量在内层循环，状态如图：



遍历背包容量在外层循环，遍历物品在内层循环，状态如图：



看了这两个图，大家就会理解，完全背包中，两个for循环的先后循序，都不影响计算dp[j]所需要的值（这个值就是下标j之前所对应的dp[j]）。

先遍历背包在遍历物品，代码如下：

```
// 先遍历背包，再遍历物品
for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
    cout << endl;
}
```

#C++测试代码

完整的C++测试代码如下：

```
// 先遍历物品，在遍历背包
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;
    vector<int> dp(bagWeight + 1, 0);
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = weight[i]; j <= bagWeight; j++) { // 遍历背包容量
```

```
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    cout << dp[bagWeight] << endl;
}
int main() {
    test_CompletePack();
}
```

```
// 先遍历背包，再遍历物品
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;

    vector<int> dp(bagWeight + 1, 0);

    for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
        for(int i = 0; i < weight.size(); i++) { // 遍历物品
            if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] +
value[i]);
        }
    }
    cout << dp[bagWeight] << endl;
}
int main() {
    test_CompletePack();
}
```

#总结

细心的同学可能发现，**全文我说的都是对于纯完全背包问题，其for循环的先后循环是可以颠倒的！**

但如果题目稍稍有点变化，就会体现在遍历顺序上。

如果问装满背包有几种方式的话？ 那么两个for循环的先后顺序就有很大区别了，而leetcode上的题目都是这种稍有变化的类型。

这个区别，我将在后面讲解具体leetcode题目中给大家介绍，因为这块如果不结合具题目，单纯的介绍原理估计很多同学会越看越懵！

别急，下一篇就是了！哈哈

最后，又可以出一道面试题了，就是纯完全背包，要求先用二维dp数组实现，然后再用一维dp数组实现，最后在问，两个for循环的先后是否可以颠倒？为什么？ 这个简单的完全背包问题，估计就可以难住不少候选人了。

518. 零钱兑换 II - 力扣（Leetcode）

这个一眼看去就是一个完全背包问题，但是思路不一样的是，我们需要计算的是并不是我们最高价值，而是需要计算我们当前有多少种方式，这样我们dp数组里面装进去就是我们当前的次数。

```
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        vector<int> dp(amount+1,0);

        dp[0]=1;
        for(int i=0;i<coins.size();i++)
        {
            for(int j=coins[i];j<=amount;j++)
            {
                dp[j]+=dp[j-coins[i]];
            }
        }

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

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

377. 组合总和 Ⅳ - 力扣（Leetcode）

这个题一眼看上去跟上面的没有什么区别，但是，这里的区别主要在于，顺序不同的序列被视为不同的组合，这个是一个重大的问题。

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> dp(target+1,0);
        dp[0]=1;

        for (int i = 0; i <= target; i++) { // 遍历背包
            for (int j = 0; j < nums.size(); j++) { // 遍历物品
                if (i - nums[j] >= 0 && dp[i] < INT_MAX - dp[i - nums[j]]) {
                    dp[i] += dp[i - nums[j]];
                }
            }
        }
```

```
        return dp[target];
    }
};
```

完全背包问题的排列与组合

这里牵扯到一个排列与组合的问题，我们在写这个组合总和和这个零钱兑换的题目，本质上都是完全背包问题，但是我们通过调整遍历顺序让其有了不同的结果，所以我们不得不开始分析这个问题了。

首先如果我们第一层循环是先遍历物品，第二层是先遍历背包的话，也就是我们是横向的更新，同时每一个横向更新取决于前面的某一个的更新，那么我们每次增加一个物品，那么这个物品必然在前一个物品后面，也就是我们现在只能是一种顺序，那么此时我们就是利用的组合。

反之，如果我们第一层循环是遍历背包大小，也就是我们的纵向更新，此时我们就包含了两者换序的情况，认为，我们在放入后一个物品的情况下，我们重新再放入前一个物品，那么我们也就是实现了组合，这样非常方便理解。

70. 爬楼梯 - 力扣（Leetcode）

这个问题我们可以将我们的总的楼梯级数看成是我们背包的容量，我们放进去的物品是一级或者两级台阶，那么这样的话，我们就可以更快的解决了，同时我们发现，我们需要的是排列数，这样我们也确定了循环顺序，那就可以直接开始了，当然用我们原本的方法也是可以的，这里先放原本的方法，再放我们的背包问题。
原本方法：

```
class Solution {
public:
    vector<int> dp;
    int climbStairs(int n) {
        dp.push_back(1);
        dp.push_back(1);
        for(int i=2;i<=n;i++)
        {
            dp.push_back(dp[i-1]+dp[i-2]);
        }
        return dp[n];
    }
};
```

背包问题：

```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n+1,0);
        dp[0]=1;
        vector<int> stairs={1,2};
```

```
        for(int i=0;i<=n;i++)
        {
            for(int j=0;j<2;j++)
            {
                if(i>=stairs[j])
                {
                    dp[i]+=dp[i-stairs[j]];
                }

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

322. 零钱兑换 - 力扣（Leetcode）

这个题目一眼看上去跟咱们上面写的零钱的问题还是有一点点区别的，但是dp数组里面存储的应该是最少的硬币个数，这样我们也就有了基本的思路了。这个问题我第一遍自己做的时候并没有做出来，因为在dp数组的含义和递推公式的设计上有一些偏差
首先，dp数组是我们最少的硬币数量，这样我们开始应该给除了dp[0]之外的所有元素设置一个足够大的值，这样我们在比较的时候会方便很多，如果我们使用的是0的话，那么开始的时候也就没有比0更加小的数字，这样会变成无法更新的情况，同时对于我们无法组成的情况，也不能很好的标识。
其次，在自己递推公式上面，应该采用min的形式，因为我们混合了两种情况，也就是使用了当前这个硬币和没有使用这个硬币的情况，这样我们更新的时候需要使用min来进行操作
但是，为什么我们每次写入的刚好是装满背包的呢，这是因为我们开始的时候，第一次循环，我们一定能够让dp[i-coins[j]]替换我们原本的位置，这样我们可以更好的解决我们的问题，对于INT_MAX项我们也不参与运算，这样就保证了我们每次填入的是刚好装满背包的，好的，那么算法在下面，加油啊！

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount+1,INT_MAX);//传统艺能dp数组
        dp[0]=0;

        for (int i=0;i<coins.size();i++)
        {
            for (int j=coins[i];j<=amount;j++)
            {
                if(dp[j-coins[i]]!=INT_MAX)
                {
                    dp[j]=min(dp[j-coins[i]]+1,dp[j]);
                }
            }
        }
        if(dp[amount]==INT_MAX) return -1;
        return dp[amount];

    }
};
```

279. 完全平方数 - 力扣（Leetcode）

```
这个题跟上面那个题完全一样
class Solution{
public:
    int numSquares(int n){
        vector<int> dp(n+1,INT_MAX);
        dp[0]=0;
        for(int i=0;i<=n;i++){
            for(int j=1;j*j<=i;j++){
                dp[i]=min(dp[i],dp[i-j*j]+1);
            }
        }
        return dp[n];
    }

};
```

单词拆分明天再接着进行

139. 单词拆分 - 力扣（Leetcode）

```

```