参与本项目，**贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！**

动态规划：01背包理论基础（滚动数组）

《代码随想录》算法视频公开课：带你学透0-1背包问题！（滚动数组），**相信结合视频再看本篇题解，更有助于大家对本题的理解。**

昨天动态规划：关于01背包问题，你该了解这些！ 中是用二维dp数组来讲解01背包。

今天我们就来说一说滚动数组，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，一些录友当时还表示比较困惑。

那么我们通过01背包，来彻底讲一讲滚动数组！

接下来还是用如下这个例子来进行讲解

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

一维dp数组（滚动数组）

对于背包问题其实状态都是可以压缩的。

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]);

与其把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）。

这就是滚动数组的由来，需要满足的条件是上一层可以重复利用，直接拷贝到当前层。

读到这里估计大家都忘了 dp[i][j]里的i和j表达的是什么了，i是物品，j是背包容量。

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，价值总和最大是多少。

一定要时刻记住这里i和j的含义，要不然很容易看懵了。

动规五部曲分析如下：

1. 确定dp数组的定义

在一维dp数组中，dp[j]表示：容量为j的背包，所背的物品价值可以最大为dp[j]。

2. 一维dp数组的递推公式

dp[j]为 容量为j的背包所背的最大价值，那么如何推导dp[j]呢？

dp[j]可以通过dp[j - weight[i]]推导出来，dp[j - weight[i]]表示容量为j - weight[i]的背包所背的最大价值。

dp[j - weight[i]] + value[i] 表示 容量为 j - 物品i重量 的背包 加上 物品i的价值。（也就是容量为j的背包，放入物品i了之后的价值即：dp[j]）

此时dp[j]有两个选择，一个是取自己dp[j] 相当于 二维dp数组中的dp[i-1][j]，即不放物品i，一个是取dp[j - weight[i]] + value[i]，即放物品i，指定是取最大的，毕竟是求最大价值，

所以递归公式为：

可以看出相对于二维dp数组的写法，就是把dp[i][j]中i的维度去掉了。

3. 一维dp数组如何初始化

关于初始化，一定要和dp数组的定义吻合，否则到递推公式的时候就会越来越乱。

dp[j]表示：容量为j的背包，所背的物品价值可以最大为dp[j]，那么dp[0]就应该是0，因为背包容量为0所背的物品的最大价值就是0。

那么dp数组除了下标0的位置，初始为0，其他下标应该初始化多少呢？

看一下递归公式：dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

dp数组在推导的时候一定是取价值最大的数，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。

这样才能让dp数组在递归公式的过程中取的最大的价值，而不是被初始值覆盖了。

那么我假设物品价值都是大于0的，所以dp数组初始化的时候，都初始为0就可以了。

4. 一维dp数组遍历顺序

代码如下：

```
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]);

    }
}
```

这里大家发现和二维dp的写法中，遍历背包的顺序是不一样的！

二维dp遍历的时候，背包容量是从小到大，而一维dp遍历的时候，背包是从大到小。

为什么呢？

倒序遍历是为了保证物品i只被放入一次！。但如果一旦正序遍历了，那么物品0就会被重复加入多次！

举一个例子：物品0的重量weight[0] = 1，价值value[0] = 15

如果正序遍历

dp[2] = dp[2 - weight[0]] + value[0] = 30

此时dp[2]就已经是30了，意味着物品0，被放入了两次，所以不能正序遍历。

为什么倒序遍历，就可以保证物品只放入一次呢？

倒序就是先算dp[2]

dp[2] = dp[2 - weight[0]] + value[0] = 15 （dp数组已经都初始化为0）

dp[1] = dp[1 - weight[0]] + value[0] = 15

所以从后往前循环，每次取得状态不会和之前取得状态重合，这样每种物品就只取一次了。

那么问题又来了，为什么二维dp数组历的时候不用倒序呢？

因为对于二维dp，dp[i][j]都是通过上一层即dp[i - 1][j]计算而来，本层的dp[i][j]并不会被覆盖！

（如何这里读不懂，大家就要动手试一试了，空想还是不靠谱的，实践出真知！）

再来看看两个嵌套for循环的顺序，代码中是先遍历物品嵌套遍历背包容量，那可不可以先遍历背包容量嵌套遍历物品呢？

不可以！

因为一维dp的写法，背包容量一定是要倒序遍历（原因上面已经讲了），如果遍历背包容量放在上一层，那么每个dp[j]就只会放入一个物品，即：背包里只放入了一个物品。

倒序遍历的原因是，本质上还是一个对二维数组的遍历，并且右下角的值依赖上一层左上角的值，因此需要保证左边的值仍然是上一层的，从右向左覆盖。

（这里如果读不懂，就再回想一下dp[j]的定义，或者就把两个for循环顺序颠倒一下试试！）

所以一维dp数组的背包在遍历顺序上和二维其实是有很大差异的！，这一点大家一定要注意。

5. 举例推导dp数组

一维dp，分别用物品0，物品1，物品2 来遍历背包，最终得到结果如下：

 动态规划-背包问题9

一维dp01背包完整C++测试代码

```
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;

    // 初始化
    vector<int> dp(bagWeight + 1, 0);
    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]);
        }
    }
    cout << dp[bagWeight] << endl;
}

int main() {
    test_1_wei_bag_problem();
}
```

可以看出，一维dp 的01背包，要比二维简洁的多！ 初始化 和 遍历顺序相对简单了。

所以我倾向于使用一维dp数组的写法，比较直观简洁，而且空间复杂度还降了一个数量级！

在后面背包问题的讲解中，我都直接使用一维dp数组来进行推导。

总结

以上的讲解可以开发一道面试题目（毕竟力扣上没原题）。

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。

相信大家读完这篇文章，应该对以上问题都有了答案！

大家可以发现其实信息量还是挺大的。

如果把动态规划：关于01背包问题，你该了解这些！ 和本篇的内容都理解了，后面我们在做01背包的题目，就会发现非常简单了。

不用再凭感觉或者记忆去写背包，而是有自己的思考，了解其本质，代码的方方面面都在自己的掌控之中。

即使代码没有通过，也会有自己的逻辑去debug，这样就思维清晰了。

接下来就要开始用这两天的理论基础去做力扣上的背包面试题目了，录友们握紧扶手，我们要上高速啦！

其他语言版本

Java

```java
public static void main(String[] args) {
    int[] weight = {1, 3, 4};
    int[] value = {15, 20, 30};
    int bagWight = 4;
    testWeightBagProblem(weight, value, bagWight);
}

public static void testWeightBagProblem(int[] weight, int[] value, int bagWeight){
    int wLen = weight.length;
    //定义dp数组：dp[j]表示背包容量为j时，能获得的最大价值
    int[] dp = new int[bagWeight + 1];
    //遍历顺序：先遍历物品，再遍历背包容量
    for (int i = 0; i < wLen; i++){
        for (int j = bagWeight; j >= weight[i]; j--){
            dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    //打印dp数组
    for (int j = 0; j <= bagWeight; j++){
        System.out.print(dp[j] + " ");
    }
}
```

Python

```py
def test_1_wei_bag_problem():
    weight = [1, 3, 4]
    value = [15, 20, 30]
    bag_weight = 4
    # 初始化: 全为0
    dp = [0] * (bag_weight + 1)

    # 先遍历物品，再遍历背包容量
    for i in range(len(weight)):
        for j in range(bag_weight, weight[i] - 1, -1):
            # 递归公式
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i])

    print(dp)

test_1_wei_bag_problem()
```

Go

```go
func test_1_wei_bag_problem(weight, value []int, bagWeight int) int {
    // 定义 and 初始化
    dp := make([]int,bagWeight+1)
    // 递推顺序
    for i := 0 ;i < len(weight) ; i++ {
        // 这里必须倒序,区别二维,因为二维dp保存了i的状态
        for j:= bagWeight; j >= weight[i] ; j-- {
            // 递推公式
            dp[j] = max(dp[j], dp[j-weight[i]]+value[i])
        }
    }
    //fmt.Println(dp)
    return dp[bagWeight]
}

func max(a,b int) int {
    if a > b {
        return a
    }
```

```
func main() {
    weight := []int{1,3,4}
    value := []int{15,20,30}
    test_1_wei_bag_problem(weight,value,4)
}
```

JavaScript

```js

function testWeightBagProblem(wight, value, size) {
  const len = wight.length,
    dp = Array(size + 1).fill(0);
  for(let i = 1; i <= len; i++) {
    for(let j = size; j >= wight[i - 1]; j--) {
      dp[j] = Math.max(dp[j], value[i - 1] + dp[j - wight[i - 1]]);
    }
  }
  return dp[size];
}


function test () {
  console.log(testWeightBagProblem([1, 3, 4, 5], [15, 20, 30, 55], 6));
}

test();
```

C

```c
#include <stdio.h>
#include <string.h>

#define MAX(a, b) (((a) > (b)) ? (a) : (b))
#define ARR_SIZE(arr) ((sizeof((arr))) / sizeof((arr)[0]))
#define BAG_WEIGHT 4

void test_back_pack(int* weights, int weightSize, int* values, int valueSize, int bagWeigl
```

```
    int i, j;
    // 先遍历物品
    for(i = 0; i < weightSize; ++i) {
        // 后遍历重量。从后向前遍历
        for(j = bagWeight; j >= weights[i]; --j) {
            dp[j] = MAX(dp[j], dp[j - weights[i]] + values[i]);
        }
    }

    // 打印最优结果
    printf("%d\n", dp[bagWeight]);
}

int main(int argc, char** argv) {
    int weights[] = {1, 3, 4};
    int values[] = {15, 20, 30};
    test_back_pack(weights, ARR_SIZE(weights), values, ARR_SIZE(values), BAG_WEIGHT);
    return 0;
}
```

TypeScript

```ts
function testWeightBagProblem(
  weight: number[],
  value: number[],
  size: number
): number {
  const goodsNum: number = weight.length;
  const dp: number[] = new Array(size + 1).fill(0);
  for (let i = 0; i < goodsNum; i++) {
    for (let j = size; j >= weight[i]; j--) {
      dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
    }
  }
  return dp[size];
}
const weight = [1, 3, 4];
const value = [15, 20, 30];
const size = 4;
```

Scala

```scala
object Solution {
  // 滚动数组
  def test_1_wei_bag_problem(): Unit = {
    var weight = Array[Int](1, 3, 4)
    var value = Array[Int](15, 20, 30)
    var baseweight = 4

    // dp数组
    var dp = new Array[Int](baseweight + 1)

    // 遍历
    for (i <- 0 until weight.length; j <- baseweight to weight(i) by -1) {
      dp(j) = math.max(dp(j), dp(j - weight(i)) + value(i))
    }

    // 打印数组
    println("[" + dp.mkString(",") + "]")
  }

  def main(args: Array[String]): Unit = {
    test_1_wei_bag_problem()
  }
}
```