长河乱语 三

OTTF 2024 2 6 起写

OTTF 2024 5 1 终写

第0章 引子

太阳浸泡在橙色的天空里，好像打翻高台上的了火炉，这把何乐乌与沃柔德的热情也点起来了，她们又一次走到那条河的旁边。

何乐乌说道：“啊，这一次应该是第三次了——也可以说是‘很多次’。”

沃柔德点了点头，“今天我们的内容是什么呢？”

何乐乌开口了。

第1章 贪心

> 这么轻松，开局就能吃一个马？
>
> ——在下象棋时开局炮打马

这里有五个苹果，甜度分别为1、2、3、4、5，你只能吃一个苹果，并且想要让自己吃的越甜越好，你会怎么选呢？

这是一个简单的问题，显然我们应该选择甜度为5的那个。那么我扩展一下——现在你可以吃三个苹果，你该如何选择呢？

还是很简单的问题，显然我们应该选择3、4、5这三个。注意到，我们解决的这两个问题都涉及到“决策”，或者说是“选择”。而且，这种决策是十分直接的——我们肯定都知道要选择最大的几个苹果。

再来看另一个问题，好比说，这里有一个由非负整数组成的“数字三角形”

```
   1
  4 3
 5 2 0
```

我们从最上方的“1”开始一层层往下行走，每次可以走到左下方或右下方的数字，最终我们要使经过的数字的和最大，并给出这个和。显然，这个问题可以被拆解成一系列决策。

首先，在从“1”往下走的时候，我们可以选择“4”或“3”，不如选择其中较大的，也就是“4”。然后，在从“4”往下走的时候，我们进行同样的选择，选择“5”。于是，我们最终得到的和是$1+4+5$，也就是10。

看起来不错，只要看一下别的路，我们就能发现，这个和是最大的。但是，如果情况是：

```
   1
  4 3
 5 2 9
```

我们还是会选择$1+4+5$的路径，但实际上，还有一条$1+3+9$，和为13，更大。看起来，每次选择最大的策略并不总能得到最大的和，或者说“最优解”。这是为什么呢？

问题就在于我们每次做出选择时，只是考虑到了下方一层的两个数，后面的则没有考虑——当我们在“1”的位置做选择时，我们只是考虑到“‘4’比‘3’大”，而没有考虑再下面一层的“9”。故而，即使在“3”的右下方有一个“100”，我们也不会选择经过它的路径。

不得不说，这种策略是一种“鼠目寸光”的策略——每次只考虑最近的利益。这看起来似乎是贬义的，而且也不太能解决实际问题。但实际上，这种策略叫做“贪心”，并且能在某些情况解决实际问题、找到最优解。

开头的选苹果就是一个例子——每次选择最大的苹果就是一种“贪心”。当然，贪心也可以解决更多问题，来看这个：

有一堆苹果，我们从小到大给出它们的大小a_i，现在我们要把这些苹果打包装箱，具体来说，一个箱子中可以有1个，也可以有2个苹果，而不管怎么样，箱子中苹果的总体积都不能超过一个给定的值k，所以显然，每一个给出的a_i其本身首先不会大于这个数。现在问题来了，我们把使用的箱子数设为res，求其最小的值。

怎样把苹果装入最少的箱子里呢？我们好像并不知道，但我们知道如何把苹果装入最多的箱子里——每个箱子都只放一个苹果。而我们可以尝试从这种方案开始一步步“变形”，从而达到箱子最少的方案。由于一个箱子最多有两个苹果，我们可以尝试“合并”，即决定是否把两个苹果放到一个箱子里。

比如，来看$6, 7, 8, 9$这四个苹果，假如$k = 15$的话，我们可以合并6和9、7和8，这样就能让箱子数最少。但是，如果我们合并了6和8的话，剩下的7和9就不能合并了。所以，看起来我们应该尽可能每一次让最小的数与最大的数合并。

为什么呢？我们刚才想出了一种选择策略，或者说是贪心策略，而它当然是需要证明的。那么让我们来想想，如果我们剩下了n个苹果，对于这些剩下的苹果的大小$a_1, a_2 \cdots a_n$，我们刚才的策略意味着尝试合并a_1与a_n，那么来分析一下：如果这两个苹果大小之和大于k，那么由于a_1是目前最小的数，我们只能让a_n自己待着——如果我们去寻找之前已经分到箱子里的苹果的话，即使它能与a_n合并，我们也会造出一个新的孤单的苹果。而如果和大小之和小于等于k，我们就让两个苹果合并到了一个盒子，一个盒子放两个苹果，这是尽可能优的。

所以说，我们的这个贪心策略是能使用的，现在我这样书写它：

```
苹果装箱（一列数a，有n个数，一个数k）：
    装箱数res，一开始为0
    最小的苹果的编号l一开始为1
    最大的苹果的编号r一开始为n

    重复：
        如果l等于r：
            res为res+1
            返回res

        如果l大于r：
            返回res

        如果a[l]+a[r]<=k：
            res为res+1
            l为l+1
            r为r-1
        否则：
```

```
        res为res+1
        r为r-1
```

第2章 继续贪心

> “贪”，就是关心“今”“贝”。
>
> ——我

让我们再来考虑一个可以用贪心解决的问题：国王领着n个大臣们做游戏，这些人像小学生站队一样排成一列，国王自己一定站在最前面，而后面大臣的位置则可以调整。国王和大臣们的左右手都有正整数，这会使得大臣们获得赏赐，每位大臣获得的赏赐等于其前面的人左手数字之和除以大臣自己右手数字的结果。现在，我们需要决定大臣们的位置，使所获赏赐最多的人获得的赏赐尽可能少。

真是一个复杂的问题，我们不如先考虑简单的情况。如果只有一个大臣的话，他的站位是一定的。而如果有两个大臣的话，我们可以把他们编号为1号和2号，先让1号站在2号的后面：

```
l0 r0
l1 r1
l2 r2
```

l_i代表第i个人左手上的数，r_i代表第i个人右手边上的数，特殊地，国王是第0个人。这样看来，第1个人得到的赏赐就是$\frac{l_0}{r_1}$，第2个人得到的赏赐就是$\frac{l_0 \times l_1}{r_2}$，其中的最大值也就是$max(\frac{l_0}{r_1}, \frac{l_0 \times l_1}{r_2})$。

而如果让两个大臣互换位置，也就是：

```
l0 r0
l2 r2
l1 r1
```

显然，两个大臣所得赏赐分别是$\frac{l_0}{r_2}$和$\frac{l_0 \times l_2}{r_1}$，其中的最大值也就是$max(\frac{l_0}{r_2}, \frac{l_0 \times l_2}{r_1})$。

我们把这两个最大值中的l_0都约掉，毕竟它对这两个式子的大小关系没有影响，于是得到$max(\frac{1}{r_1}, \frac{l_1}{r_2})$和$max(\frac{1}{r_2}, \frac{l_2}{r_1})$。我们再去它们的分母，也就得到$max(r_2, l_1 \times r_1)$与$max(r_1, l_2 \times r_2)$。

在这两个式子里的四项中，显然$r_2 \leq l_2 \times r_2$，而如果r_2是最大的，我们就能得到$r_2 = l_2 \times r_2$这代表无论采用哪种站位都是一样的。而r_1最大的情况也一样。所以，我们可以剔除这两项，也就得到$l_1 \times r_1$和$l_2 \times r_2$。我们只要比较它们的大小就能确定使用哪种站位了，实际上，这意味着让左右手两个数字乘积较小的站在前面。

我们刚才分析了有两个大臣的情况，那么对于有很多大臣时呢？事实上，两个大臣的站位并不会影响别的大臣所获得的赏赐，这很显然。所以，不管有多少大臣，我们只需要按手上数字乘积排序他们就好了。

该如何按照这个有点奇怪的需求排序呢？实际上，当我们排序时，一个“大小的定义”十分重要，我们经常提到某数小于某数，但这也可以写成小于（某数，某数），也就是说，我们可以把所谓的“小于”当作一个函数传进去。

第3章 记忆化搜索

> 把字刻在石头上。
>
> ——刘慈欣《三体》

对于贪心的讲解好像有些枯燥了，你头晕吗？好吧，让我们来看一些简单的问题。

还记得数字三角形问题吗？这是一个无法用贪心解决的问题。既然这样，我们怎样解决它呢？既然无法用贪心这样简便的方法解决，要不我们暴力吧！

是的！从最上面开始依次往下递归，计算出每一种路径所能得到的和，然后取其中的最大值就可以了。可惜的是这种方法的时间复杂度显然很大。所以，我们需要使用一些技巧。

来看这个数字三角形：

```
   1
  4 3
 5 2 9
6 7 8 0
```

在我们从“1”开始往下走时，我们可以认为是依次尝试了以“4”开始的众多路径和以“3”开始的众多路径，并选择了其中和最大的一个。也就是说，对于最上面的位置，我们为了求出经过的数字之和最大的路径，其实是对它下面的两个位置分别进行了一次“数字三角形”问题的求解。而这可以推广到任何位置上。

也就是说，我们可以认为：

```
一列元素nums，共n个元素，每个元素是一个数组，其长度等于它的编号
数字三角形（现在的位置是第i行，第j个）：
    如果i等于n：
        返回 nums[j][j]
    否则：
        返回 nums[i][j] + 最大（数字三角形（i+1，j），数字三角形（i_1，j+1））
```

这有什么“技巧”吗？时间复杂度应该如刚才所说那样，是很高的，这确实没错。但是，你肯定发现了，在上面的例子中，我们会重复访问“2”那个位置，也就重复计算了这里的答案。而如果我们使用记忆化搜索，把这里的答案储存起来，就能减少计算量：

```
一列元素nums，共n个元素，每个元素是一个数组，其长度等于它的编号
一列元素ress，共n个元素，每个元素是一个数组，其长度等于它的编号，初始时全为-1
数字三角形（现在的位置是第i行，第j个）：
    如果 ress[i][j] 等于 -1：
        如果i等于n：
           ress[i][j] 为 nums[i][j]
        否则：
            ress[i][j] 为 nums[i][j] + 最大（数字三角形（i+1，j），数字三角形（i_1，
j+1））

    返回 ress[i][j]
```

这样的话，每一个位置的答案只会被计算一遍，之后都能直接调用。我们可以调用数字三角形（1，1）来完成计算。

第4章 从数字三角形到背包

> 宙斯的一银元，赫拉的两银元，赫尔墨斯的白送。
>
> ——雕像者在赫尔墨斯前说道

你或许记得，我们第一次提到“记忆化”搜索实在说斐波那契的时候，而在其后，我们迅速引入了递推——一种不用递归就能算斐波那契数的方法。所以，我们能不能消除数字三角形算法中的递归呢？

显然可以。审视这个代码：

```
一列元素nums，共n个元素，每个元素是一个数组，其长度等于它的编号
一列元素ress，共n个元素，每个元素是一个数组，其长度等于它的编号，初始时全为-1
```

```
数字三角形（现在的位置是第i行，第j个）：
    如果 ress[i][j] 等于 -1：
        如果i等于n：
           ress[i][j] 为 nums[i][j]
        否则：
            ress[i][j] 为 nums[i][j] + 最大（数字三角形（i+1，j），数字三角形（i+1，
j+1））

    返回 ress[i][j]

数字三角形（1，1）
```

我们开始计算“第1行，第1个”这个位置的答案，而为了计算这个位置，我们需要计算“第2行，第1个”与“第2行，第2个”这两个位置的答案。可想而知，为了计算所有答案，我们最先得到的是“第n行”的所有答案，然后才能得到其他答案。所以，为了消除递归，我们先从第n行算起，自然得到：

```
一列元素nums，共n个元素，每个元素是一个数组，其长度等于它的编号
一列元素ress，共n个元素，每个元素是一个数组，其长度等于它的编号，初始时全为-1
数字三角形（）：
    i从1到n循环：
        ress[n][i] 为 nums[n][i]
```

而剩下的部分，我们也可以一层层反推，这意味着我们需要从第n-1行推到第1行，也就是：

```
一列元素nums，共n个元素，每个元素是一个数组，其长度等于它的编号
一列元素ress，共n个元素，每个元素是一个数组，其长度等于它的编号，初始时全为-1
数字三角形（）：
    i从1到n循环：
        ress[n][i] 为 nums[n][i]

    i从n-1到1循环：
        j从1到i循环：
            ress[i][j] 为 最大（ress[i+1][j]，ress[i+1][j+1]）
```

你看，这样，我们就把这个使用记忆化搜索的算法转变成了直接使用循环的算法。可这有什么深意吗？我们来看另一个问题：

有一堆苹果，我们从小到大给出它们的大小a_i，现在我们要把这些苹果打包装箱，具体来说，一个箱子中可以有1个，也可以有2个苹果，而不管怎么样，箱子中苹果的总体积都不能超过一个给定的值k，所以显然，每一个给出的a_i其本身首先不会大于这个数——稍等，这不就是我们能用贪心解决的那个问题吗？确实，不过现在我加上一个条件：每个苹果有自己的价值v_i，我们要做的事让箱子中价值的和最大。

比如说，如果有3个苹果，k等于10，a_i分别是$1, 9, 11$，而v_i分别是$2, 2, 3$，最终我们就能装总价值为4的苹果。

该怎么做呢？你看，我们在解决数字三角形的问题时，把这个大问题先定了个性——它是“如何在第1行第1列及其下方找到最优的方案”，而要解决它，我们就需要解决“如何在第2行第1列及其下方找到最优的方案”等问题。同时，为了减少重复的计算，我们把每次计算的结果记录下来，以后就可以直接使用。

所以先来拆分一下这个苹果问题吧：注意到，我们可以认为自己是在从头到尾一个一个决定要不要选每一个苹果。所以说，我们要解决的问题是“如何在选择第n个及其前面的苹果时找到最佳方案”。但是稍等，我们的容量最大值k貌似也会对我们的选择产生影响，所以，我们可以说要解决的问题是“如何在容量最大值为k，选择第n个及其前面的苹果时找到最佳方案”。

这是一个很大、需要在最后解决的问题。我们能直接解决的问题有两种——如果我们的最大容量是0，我们能得到的价值就是0；如果我们能选择的苹果数是0，那么我们能得到的价值也是0。这些东西可以被称为“边界条件”。

求解数字三角形问题，我们写了`ress[i][j] 为 最大（ress[i+1][j]，ress[i+1][j+1]）`这个式子，它能利用两个小问题的答案得到一个大问题的答案。现在，让我们给苹果问题写一个这样的式子。嗯……我们还是用数字来表示问题的答案，所以它的左边是`ress[i][j]`，这表示“选择第`i`个及其前面的苹果，容量最大值为`j`”。

对于一个苹果，我们可以选也可以不选。而为了得到最大的价值，我们应该去这两种情况中较大的。如果不取这个苹果，我们肯定能得到`ress[i-1][j]`的价值，而如果取呢？嗯哼，如果取这个苹果，我们能得到`ress[i-1][j-a[i]]+v[i]`的价值。其中，`j-a[i]`表示着我们把`a[i]`的空间用来放这一个苹果，这也暗示着如果要取，`j`必须大于等于`a[i]`；而`+v[i]`就表示我们得到了这个苹果的价值。

这意味着我们找到了式子——`ress[i][j] 为 最大（ress[i-1][j]，ress[i-1][j-a[i]]+v[i]）`！为了凸显它用来“转移”我们计算的答案的功能，我们可以叫它“转移方程”。而有因为所谓“选择第`i`个及其前面的苹果，容量最大值为`j`”既可以被视为是一个“小问题”，也可以被视为一个“状态”，所以，我们把式子叫做“状态转移方程”。

把其他的部分补上，我们自然发现苹果问题的解法是：

```
一列数a，共n个
一列数v，共n个
一列元素ress，共n个元素，每个元素是一个数组，其长度等于k
苹果（）：
    i从1到n循环：
        j从1到k循环：
```

```
        如果j小于a[i]:
            ress[i][j]为ress[i-1][j]
        否则:
            ress[i][j] 为 最大 (ress[i-1][j], ress[i-1][j-a[i]]+v[i])
```

很好，看起来我们解决了如何装苹果的问题，这是一个伟大的开始！啊哈！

第5章 背包的空间优化

> 有心栽花花不开，无心插柳柳成荫。
>
> ——古话

我们一直把刚才解决的问题叫做“苹果问题”，但实际上他有一个更通用的名称——“背包问题”，意思是我们考虑要不要把一些“东西”装进“背包”里，从而达到一个最优解。再细分一下，我们解决的这个问题是个“01背包问题”，因为每种苹果只能选一个，我们不可能把“第1个苹果”选两次；并且每一个苹果可以选，也可以不选，不可以“选又不选”。

注意到，我们刚才的解法的时间和空间复杂度都是$O(nk)$。嗯，时间复杂度或许不能降了，因为我们必定需要计算$ress$的每一个值，那么空间复杂度呢？

看看我们的状态转移方程——`ress[i][j] 为 最大 (ress[i-1][j], ress[i-1][j-a[i]]+v[i])`，不得不注意到它只是用到了“这一行”——第i行、和“上一行”——第$i-1$行的数据。所以，我们可以考虑把它削减成只有两行：

```
一列数a，共n个
一列数v，共n个
一列元素ress，共2个元素，每个元素是一个数组，其长度等于k
01背包 () :
    现在的行数now为1
    上一行的行数last为2
    i从1到n循环:
        j从1到k循环:
            如果j小于a[i]:
                ress[now][j]为ress[last][j]
            否则:
                ress[now][j] 为 最大 (ress[last][j], ress[last][j-a[i]]+v[i])
        交换 (now, last)

    交换 (now, last)
```

我们在函数的末尾也进行了一次`交换 (now, last)`，是为了让最终的答案等于`ress[now][k]`。不得不说，这个代码真的把空间复杂度减少了很多——它已经变成

$O(k)$了。不过，其实我们只用一行数组就能搞定这个问题：

```
一列数a，共n个
一列数v，共n个
一列元素ress，共k个元素
01背包（）：
    i从1到n循环：
        j从k到a[i]循环：
            ress[j] 为 最大（ress[j]，ress[j-a[i]]+v[i]）
```

很好，我们把第一维干脆直接剔除了，但是为什么里面的那一层循环被倒了过来呢？你看，我们的状态转移方程在使用“容器”那一维时，只会使用第`j-a[i]`个，也就是第`j`个之前的。如果我们正着循环，第`j-a[i]`个就已经被第`i`个物体更新过了——我们会再考虑一边要不要取它，这不符合要求。而我们倒着循环，第`j-a[i]`个实际上就是“上一行”的数据，我们就可以放心使用了。现在我们拥有了一个异常简洁的“01背包问题”解决方案了。

等一下，我们刚才谈到“我们会再考虑一边要不要取它，这不符合要求”，这显然是因为每一个东西只能取一次。但是，如果每一个东西可以取很多次，不把循环倒过来的方法不就可以使用了吗？

```
一列数a，共n个
一列数v，共n个
一列元素ress，共k个元素
完全背包（）：
    i从1到n循环：
        j从a[i]到k循环：
            ress[j] 为 最大（ress[j]，ress[j-a[i]]+v[i]）
```

呃，不过在哪里会有这种需求呢？在我们讨论的东西是一种一种的时候。比如说，现在我们眼前的不是几个价值和大小不同的苹果，而是几棵苹果树。每一棵树上的苹果都有相同的价值和大小，并且每一个树上的苹果都有很多，怎么摘也摘不完。现在我们需要摘一些苹果放到有容量限制的箱子里，使得总价值最大。这种问题叫做“完全背包问题”，可以使用上面的方法解决。

第6章 略显复杂的背包

> 假如世上只有一个肥皂泡，其价值会是多少呢？
>
> ——马克吐温《登勃朗峰》

啊哈！我们已经能解决每种东西只有一个的“01背包问题”，和每种东西有无限个的“完全背包问题”这两种背包问题了。话说——我们能解决每种东西有大于一的有限个的问题吗？当然可以。

我们完全可以把它转换为一个“01背包问题”，要想这么做，我们只需要把每一种物品拆成多种每种只有一个的物品。比如说，如果我们有1种苹果，而这种总共有9个，我们就可以看成我们拥有9种每种只有一个的苹果，只是每一种的价值相同、体积相同罢了。

很不错的注意！然而，我们有一个问题——如果一种的数量十分多，甚至比种数还要多得多，我们就将会看成我们有十分多种类的物品，这势必导致时间复杂度升高。我们需要找到另一种解决方式。

幸运的是，我们只需要做一点小小的改动。显而易见，把每种物品直接暴力拆分还是显得过于粗暴了，我们可以使用另一种拆分方式——将某些物品“整合”在一起，作为“01背包问题”考虑的个体。而我们的拆分需求，显然是在拆分之后，我们最终还是可以表示取任意个某种物品的方案。

如果1种苹果有9个，每一个价值为w、体积为v，我们应该怎样拆分呢？首先，需要有1种只有1个的苹果，其价值为w、体积为v，因为我们肯定要表示“选1个这种苹果”。接下来，我们需要表示“选2个这种苹果”，我们当然可以再拆分出1种只有1个的苹果，其价值为w、体积为v，但这太不划算。我们显然可以拆分出1种2个苹果组合成的“聚合苹果”，其价值为$2w$、体积为$2v$，这样我们不仅可以表示“选2个这种苹果”，还可以将刚才的第1种苹果和现在的第2种一起选，表示“选3个这种苹果”。

接下来，我们再拆分出1种4个苹果组合成的“聚合苹果”，其价值为$4w$、体积为$4v$，这导致我们可以一路表示到“选7个这种苹果”。现在只剩2个苹果了，我们把它们组合成价值为$2w$、体积为$2v$的“聚合苹果”，于是我们可以一路表示到“选9个这种苹果”——我们可以表示所有选择方式了！

这种拆分方式叫做“二进制优化”，因为它让我们可以快速地处理这种背包问题。事实上，这种背包问题的正式名称叫做“多重背包”。不过细想一下，为什么“二进制优化”一定可以满足我们的拆分要求呢？

```
1 2 4 8 5
```

首先，设我们有n个物品，而它被拆分成了$2^0, 2^1, 2^2 \cdots 2^{k-1}, 2^k, m$个物品的聚合体，其中$m = n - 2^k$。我们显然可以一路表示到“选$2^{k+1} - 1$个物品”，不过怎么从表示“选$2^{k+1}$个物品”到“选$n$个物品”呢？其实，我们可以认为我们是相对于选所有2的正整数次幂的聚

合体的方案，先选上m，然后不选一些本来选了的聚合体，从而达到从表示“选2^{k+1}个物品”到“选n个物品”的每种状态。

所以，我们要做的就是，对于每一个从1到m的数a，想出一个方法用$m-b$表示，其中b可以用从2^0到2^{k-1}的2的正整数次幂相加表示。而毕竟2^0到2^{k-1}的正整数次幂可以表示从1到$2^{k+1}-1$的数，且$b<m$，而$m<2^{k-1}$是清晰明了的。所以，二进制优化是可行的。

再来看另一种背包——“分组背包”。每种物品依然只有选和不选，但某些物品有互斥关系。嗯，我们可以认为这是比较花哨的“01背包问题”，只需要便历每一组，然后找到对于这一组最优的选择方案就好了。这种问题被称为“分组背包问题”。

第7章 更多动规

> 会动的乌龟。
>
> ——一个学长如此向我解释“动规”。

看啊，我们到现在已经解决了很多“背包问题”，而在解决这些问题时，我们都是把大问题划分为小问题，然后先解决独立的小问题，再利用小问题的答案来得到大问题的答案。这种解决问题的思路叫做“动态规划”，简称“动规”，或者英文缩写“DP”。值得一提的是，所谓“规划”在这里的的最初含义其实是在纸上画方格算数。

动态规划可以解决很多问题，让我们再来看一个：现在有一个数列a，共n个正整数，我们要得到“最长上升子序列”的长度。所谓“最长上升子序列”需要好好解释一下：在数列中，随便选取几个数，组成的就是一个子序列。如果其中每一个都比前面的大，那么它就是个“上升子序列”。“最长上升子序列”就是所有上升子序列中最长的，我们要求的是它的长度。比如说，对于数列$1,2,3,0,10,100,10$，答案应该是5，因为最长上升子序列是$1,2,3,10,100$。

来用动规解决这个问题吧！对于只有1个数的数列，我们问题的答案当然为1。而我们可以设立dp这一数组，让它表示以每一项作为结尾的最长上升子序列的长度。这样，我们就可以思考如何划分问题：显然，对于一个有i个数的数列，我们可以认为前$i-1$个位置的答案已经被算出，只需要计算一下第i个数那里的了。那么该怎么计算呢？

不难发现，我们可以遍历一下前$i-1$个数，这是因为如果遍历到的数字a_j小于a_i，我们就可以把a_i接到以a_j为结尾的最长上升子序列的后面，这意味着dp_i可以取dp_j+1，不过我们当然要取一个较大的值。而最终当我们获得答案的时候，应当取dp中最大的那个长度。

现在，不难写出伪代码了：

```
一列数a，长度为n
一列数dp，长度为n，初始时全为0
最长不上升子序列（）：
    i从1到n循环：
        j从1到i-1循环：
            dp[i] 为 最大（dp[i]，dp[j]+1）

    res为0
    i从1到n循环：
        res 为 最大（res，dp[i]）

    返回res
```

这样的动规可以被称为“线性DP”，因为它在枚举时一般枚举的是数组的下标。而让我们再来看另一个问题：还是一列数a，共n个，我们可以把相邻的两个数a_i与a_{i+1}合并。合并后这两个数字消失，但是其位置会诞生一个新数，其值等于a_i+a_{i+1}。而我们每次合并都需要一个代价，代价也等于a_i+a_{i+1}。现在我们需要求出把这些数字最终都合并在一起所需的最小代价。

嗯哼哼，我们需要开动脑筋了。我们可以设立dp，使得$dp[i][j]$表示把从第i个数到第j个数都合并起来所需的最小代价，显然如果$i=j$则这个数等于0。现在，让我们来看看如何推导出这之后的结果。

啊哈！“从第i个数到第j个数”，说白了就是一段数，或者说一个区间。而既然它的长度应该大于一——不然我们就不用计算了——我们就可以把它劈成两段更小的区间。这样，我们能得到这两段区间所对应的答案，我们可以认为这两段区间已经被合并成两个数了，现在只需要再计算一下把这两个数合并所需的代价就好了——不难发现这等于两端区间中所有数字——合并前的数字——的和。知道了这些，我们只要枚举所有区间划分方式就好了。当然，还有一点需要注意的是，我们显然需要从短到长枚举区间——所以我们会从小到大枚举区间长度：

```
一列数a，长度为n
一列元素dp，长度为n，每个元素都是一列数，长度为n，每个数初始都为0
合并（）：
    len从2到n循环：
        i从1到n-len+1循环：
            j为i+len-1

            k从i到j-1循环：
                dp[i][j] 为 最小（dp[i][j]，dp[i][k]+dp[k+1][j]+从a[i]到a[j]的和）
```

看起来不错，可是……从a[i]到a[j]的和计算起来大概是$O(n)$的，这会导致整个算法的时间复杂度是$O(n^4)$……啊呀，这还是我们第一次见到如此大的时间复杂度！能不能优化呢？

优化的点就在从a[i]到a[j]的和上，我们可以用$O(1)$的时间得到它——显然，我们可以得到“从a_i到a_j的和”等于“从a_1到a_j的和”减去“从a_1到a_{i-1}的和”，所以只要我们预先处理好从数列中第一个数到每一个数的和，就能用较快的时间得出区间求和的结果。我们称这种技巧为“前缀和”。

现在，我们可以写出：

```
一列数a，长度为n
一列数sum，长度为n
一列元素dp，长度为n，每个元素都是一列数，长度为n，每个数初始都为0
合并（）：
    i从1到n循环：
        sum[i]为sum[i-1]+a[i]

    len从2到n循环：
        i从1到n-len+1循环：
            j为i+len-1

            k从i到j-1循环：
                dp[i][j] 为 最小（dp[i][j]，dp[i][k]+dp[k+1][j]+sum[j]-sum[i-1]）
```

顺嘴一提，如果在数组的访问时下标为0，我们可以认为得到的值为0。不管怎样，我们又用DP解决了一个问题。而由于这种DP是处理区间的，并且总要枚举一下区间的强度，我们便叫它“区间DP”。

第-1章 结语

“好酷，好玄妙，”沃柔德右手托腮，“知识像雨滴一样泼洒进我的大脑。”

“而它平息之后，会留下一道绚丽的彩虹。”何乐乌说道。她接着又补上：“老样子，我并不认为这都是对的——去网上搜索一遍也是一种复习……对。”

比平时早了一点，两人依旧在月光下离开了。