

2024/11/05 动态规划

Updated 1411 GMT+8 Nov 5, 2024

2024 fall, Complied by Hongfei Yan

Log:

2024/11/4 示例题目课上讲，练习同学课后做

2024/10/24 部分内容取自, https://github.com/GMyhf/2023fall-cs101/blob/main/dp_questions.md

说明：

讲dp，3小时有5+个典型题目，对于零基础同学来说，很烧脑。dp与递归，经常同时出现。本课程最难的部分是DP。因为没有定式，只有框架。从经典模型讲起，结合例题。

零基础同学，最好提前预习。预习内容可以是《算法图解》第9章，里面的两个题目，我们都有。题目是：
OJ23421: 小偷背包，OJ02806:公共子序列。

有100+个逐行讲解的题目视频，顺着这个链接都能找到，【oj02754/八皇后_TA胡扬-哔哩哔哩】 <https://b23.tv/s933Y5c> 如果有缺少并且需要，可以在群里反馈，TA还可以录制的。

Recap

课前思考：作业中 OJ02754：八皇后 是加了约束的 sy132：全排列 I （全排列保证了不同行不同列）， OJ23421：小偷背包 是 0-1 背包， CF189A.Cut Ribbon 是完全背包最优化。本节 找出最少零钱组合 是完全背包最优化。

02945：拦截导弹 带出了 狄尔沃斯定理， https://github.com/GMyhf/2024fall-cs101/blob/main/other/Dilworth_theorem.md

示例02754: 八皇后 穷举法

八皇后的递归写法，不容易理解，可以先把穷举的方法掌握了。

在八皇后问题中，每个皇后放在不同的行且不同的列，因此可以使用排列 (permutation) 来建立解空间。每种排列代表每行中的一个皇后所在的列。通过检查排列是否满足对角线的约束（即两个皇后不能在同一对角线上），可以筛选出合法的解。

```
1 # 02754:八皇后 http://cs101.openjudge.cn/practice/02754/    穷举法
2 from itertools import permutations
3
4 def solve_n_queens(n):
5     solutions = []
6     cols = range(n)
7
8     # 生成每一行皇后位置的排列
9     for perm in permutations(cols):
10        # 检查是否有两个皇后在同一对角线上
11        if n == len(set(perm[i] + i for i in cols)) == len(set(perm[i] - i for i in
cols)):
12            # 如果满足条件，加入解
13            solutions.append(perm)
14
15    return solutions
16
17 solutions = solve_n_queens(8)
18
19 for _ in range(int(input())):
20    n = int(input())
21    queen_string = ''.join(str(col + 1) for col in solutions[n - 1])
22    print(queen_string)
23
```

具体来说，排列 (c_1, c_2, \dots, c_n) 表示第 1 行的皇后在第 c_1 列，第 2 行的皇后在第 c_2 列，以此类推。

对角线检查

为了确保没有两个皇后在同一对角线上，我们需要检查以下两种类型的对角线：

- 主对角线** (从左上到右下) : 对于位置 (i, j) , 主对角线上的其他位置 (i', j') 满足 $i - j = i' - j'$ 。
- 副对角线** (从右上到左下) : 对于位置 (i, j) , 副对角线上的其他位置 (i', j') 满足 $i + j = i' + j'$ 。

具体实现

在代码中, 我们使用集合来检查对角线冲突:

- 主对角线检查**: 使用 `set(perm[i] + i for i in cols)` 来检查主对角线上的冲突。
- 副对角线检查**: 使用 `set(perm[i] - i for i in cols)` 来检查副对角线上的冲突。

如果这两个集合的大小都等于 n , 说明没有两个皇后在同一对角线上。

通过对角线检查, 我们可以确保生成的排列中没有两个皇后在同一对角线上。这种方法利用了集合的唯一性特性, 高效地检查了对角线冲突。

详细解释一下主对角线和副对角线的原理。

主对角线 (从左上到右下)

对于位置 (i, j) , 主对角线上的其他位置 (i', j') 满足 $i - j = i' - j'$ 。这是因为主对角线上的所有位置在直角坐标系中具有相同的 $i - j$ 值。

举例说明

考虑一个 4×4 的棋盘, 我们标记出所有的主对角线:

1	(0,0)	(0,1)	(0,2)	(0,3)
2	(1,0)	(1,1)	(1,2)	(1,3)
3	(2,0)	(2,1)	(2,2)	(2,3)
4	(3,0)	(3,1)	(3,2)	(3,3)

对于每个位置 (i, j) , 计算 $i - j$ 的值:

- $(0,0) \rightarrow 0 - 0 = 0$
- $(0,1) \rightarrow 0 - 1 = -1$
- $(0,2) \rightarrow 0 - 2 = -2$
- $(0,3) \rightarrow 0 - 3 = -3$
- $(1,0) \rightarrow 1 - 0 = 1$
- $(1,1) \rightarrow 1 - 1 = 0$
- $(1,2) \rightarrow 1 - 2 = -1$
- $(1,3) \rightarrow 1 - 3 = -2$
- $(2,0) \rightarrow 2 - 0 = 2$
- $(2,1) \rightarrow 2 - 1 = 1$
- $(2,2) \rightarrow 2 - 2 = 0$

- $(2, 3) \rightarrow 2 - 3 = -1$
- $(3, 0) \rightarrow 3 - 0 = 3$
- $(3, 1) \rightarrow 3 - 1 = 2$
- $(3, 2) \rightarrow 3 - 2 = 1$
- $(3, 3) \rightarrow 3 - 3 = 0$

可以看到，主对角线上的位置具有相同的 $i - j$ 值。例如，主对角线 $(0,0), (1,1), (2,2), (3,3)$ 上的所有位置的 $i - j$ 值都是 0。

副对角线（从右上到左下）

对于位置 (i, j) ，副对角线上的其他位置 (i', j') 满足 $i + j = i' + j'$ 。这是因为副对角线上的所有位置在直角坐标系中具有相同的 $i + j$ 值。

举例说明

继续考虑同一个 4×4 的棋盘，我们标记出所有的副对角线：

1	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$
2	$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$
3	$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$
4	$(3,0)$	$(3,1)$	$(3,2)$	$(3,3)$

对于每个位置 (i, j) ，计算 $i + j$ 的值：

- $(0,0) \rightarrow 0 + 0 = 0$
- $(0,1) \rightarrow 0 + 1 = 1$
- $(0,2) \rightarrow 0 + 2 = 2$
- $(0,3) \rightarrow 0 + 3 = 3$
- $(1,0) \rightarrow 1 + 0 = 1$
- $(1,1) \rightarrow 1 + 1 = 2$
- $(1,2) \rightarrow 1 + 2 = 3$
- $(1,3) \rightarrow 1 + 3 = 4$
- $(2,0) \rightarrow 2 + 0 = 2$
- $(2,1) \rightarrow 2 + 1 = 3$
- $(2,2) \rightarrow 2 + 2 = 4$
- $(2,3) \rightarrow 2 + 3 = 5$
- $(3,0) \rightarrow 3 + 0 = 3$
- $(3,1) \rightarrow 3 + 1 = 4$
- $(3,2) \rightarrow 3 + 2 = 5$
- $(3,3) \rightarrow 3 + 3 = 6$

可以看到，副对角线上的位置具有相同的 $i + j$ 值。例如，副对角线 $(0,3), (1,2), (2,1), (3,0)$ 上的所有位置的 $i + j$ 值都是 3。

示例02945: 拦截导弹

02945: 拦截导弹

dp, greedy <http://cs101.openjudge.cn/practice/02945/>

这个题目最优解是greedy, $O(n \log n)$ 。

```
1 """
2 与这个题目思路相同:
3 28389: 跳高, http://cs101.openjudge.cn/practice/28389
4
5 拦截导弹 求最长不升LIS, 可以相等所以用 bisect_right。如果求最长上升LIS, 用 bisect_left
6 """
7
8 from bisect import bisect_right
9
10 def min_testers_needed(scores):
11     scores.reverse() # 反转序列以找到最长下降子序列的长度
12     lis = [] # 用于存储最长上升子序列
13
14     for score in scores:
15         pos = bisect_right(lis, score)
16         if pos < len(lis):
17             lis[pos] = score
18         else:
19             lis.append(score)
20
21     return len(lis)
22
23
24 N = int(input())
25 scores = list(map(int, input().split()))
26
27 result = min_testers_needed(scores)
28 print(result)
```

写法特别高明, bisect需要排序, 但是代码中看不到sort。

使用bisect时候, 有时候不需要显示排序。类似的, 在递归中, 在某些情况下, 终止条件可以通过外部条件来控制, 而不是在递归函数内部显式地定义基准情况。

理解你的意思了。在递归中, 确实可以通过外部条件来控制递归的终止, 而不是在递归函数内部显式地定义基准情况。这种做法在某些情况下可以使代码更加简洁和灵活。下面是一些示例, 展示如何通过外部条件来控制递归的终止。

示例 1：深度优先搜索（DFS）

在深度优先搜索中，递归的终止条件可以通过一个外部的访问集合来控制。

```
1 def dfs(graph, node, visited):
2     visited.add(node)
3     print(node, end=' ')
4     for neighbor in graph[node]:
5         if neighbor not in visited:
6             dfs(graph, neighbor, visited)
7
8 # 定义图的邻接表
9 graph = {
10    'A': ['B', 'C'],
11    'B': ['A', 'D', 'E'],
12    'C': ['A', 'F'],
13    'D': ['B'],
14    'E': ['B', 'F'],
15    'F': ['C', 'E']
16 }
17
18 # 调用深度优先搜索
19 visited = set()
20 dfs(graph, 'A', visited)
```

在这个例子中，递归的终止条件是由 `visited` 集合控制的。当一个节点已经被访问过时，递归自然终止，不会继续深入。

示例 2：生成所有子集

在生成所有子集的问题中，递归的终止条件可以通过当前子集的长度来控制。

```
1 def generate_subsets(nums, index, current_subset, all_subsets):
2     all_subsets.append(current_subset.copy())
3     for i in range(index, len(nums)):
4         current_subset.append(nums[i])
5         generate_subsets(nums, i + 1, current_subset, all_subsets)
6         current_subset.pop()
7
8 # 测试生成所有子集
9 nums = [1, 2, 3]
10 all_subsets = []
11 generate_subsets(nums, 0, [], all_subsets)
12 print(all_subsets)
```

在这个例子中，递归的终止条件是由 `index` 控制的。当 `index` 达到 `nums` 的长度时，递归自然终止。`current_subset` 的状态通过外部参数传递，而不是在递归函数内部显式地定义基准情况。

示例 3：生成所有排列

在生成所有排列的问题中，递归的终止条件可以通过当前排列的长度来控制。

```
1 def generate_permutations(nums, index, current_permutation, all_permutations):
2     if index == len(nums):
3         all_permutations.append(current_permutation.copy())
4         return
5     for i in range(index, len(nums)):
6         nums[index], nums[i] = nums[i], nums[index]
7         current_permutation.append(nums[index])
8         generate_permutations(nums, index + 1, current_permutation,
9         all_permutations)
10        current_permutation.pop()
11        nums[index], nums[i] = nums[i], nums[index]
12
13 # 测试生成所有排列
14 nums = [1, 2, 3]
15 all_permutations = []
16 generate_permutations(nums, 0, [], all_permutations)
17 print(all_permutations)
```

在这个例子中，递归的终止条件是由 `index` 控制的。当 `index` 达到 `nums` 的长度时，递归自然终止。`current_permutation` 的状态通过外部参数传递，而不是在递归函数内部显式地定义基准情况。

总结

通过外部条件来控制递归的终止，可以使代码更加简洁和灵活。这些外部条件可以是访问集合、索引、长度等，具体取决于问题的性质。希

全排列、八皇后，对照着学习

全排列I, <https://sunnywhy.com/sfbj/4/3/132>, 02754 八皇后, <http://cs101.openjudge.cn/practice/02754/>

```
1 # 全排列I, https://sunnywhy.com/sfbj/4/3/132
2 list1 = []
3
4 def sequ(s, nums):
5     if len(s) == nums:
6         list1.append(s)
7         return
8     for i in range(1, nums + 1):
9         if str(i) not in s:
10             sequ(s + str(i), nums)
11
12 num = int(input())
13 sequ('', num)
14 for k in list1:
15     print(' '.join(k))
```

```

1 # 全排列I, https://sunnywhy.com/sfbj/4/3/132
2 list1 = []
3
4 def sequ(s, nums):
5     if len(s) == nums:
6         list1.append(s)
7         return
8     for i in range(1, nums + 1):
9         if str(i) not in s:
10            sequ(s + str(i), nums)
11
12 num = int(input())
13 sequ('', num)
14 for k in list1:
15     print(''.join(k))

# 02754 八皇后, http://cs101.openjudge.cn/practice/02754/
1 list1 = []
2
3
4 def queen(s):
5     if len(s) == 8:
6         list1.append(s)
7         return
8     for i in range(1, 9):
9         if all(str(i) != s[j] and abs(len(s) - j) != abs(i - int(s[j]))) for j in range(len(s)):
10            queen(s + str(i))

11 queen('')
12 samples = int(input())
13 for k in range(samples):
14     print(list1[int(input()) - 1])

"""
abs(len(s) - j) != abs(i - int(s[j])) for j in range(len(s)) 是一个生成器表达式,
用于检查当前尝试放置的皇后是否与已经放置的皇后在同一条对角线上。具体解释如下:

- len(s) 表示当前已经放置的皇后的数量, 即当前正在尝试放置的皇后的行号。
- j 是已经放置的皇后的列号。
- i 是当前尝试放置的皇后的列号。
- s[j] 是已经放置的皇后所在的列号。

对于每一个已经放置的皇后, 检查以下条件:
- abs(len(s) - j) 计算当前尝试放置的皇后与已经放置的皇后之间的行差。
- abs(i - int(s[j])) 计算当前尝试放置的皇后与已经放置的皇后之间的列差。

如果行差和列差相等, 说明两皇后在同一条对角线上, 返回 `False`, 否则返回 `True`。
"""


```

测试输入 提交结果 历史提交

完美通过

100% 数据通过测试
运行时长: 0 ms

1 Dynamic Programming

<https://runestone.academy/ns/books/published/pythonds3/Recursion/DynamicProgramming.html?mode=bowing>

Many programs in computer science are written to optimize some value; for example, find the shortest path between two points, find the line that best fits a set of points, or find the smallest set of objects that satisfies some criteria. There are many strategies that computer scientists use to solve these problems. One of the goals of this book is to expose you to several different problem-solving strategies. **Dynamic programming** is one strategy for these types of optimization problems.

许多计算机科学程序都是为了优化某个值而编写的；例如，找到两点之间的最短路径，找到最能拟合一组点的直线，或者找到满足某些条件的最小对象集。计算机科学家们使用了许多策略来解决这些问题。本书的一个目标是让你接触到几种不同的问题解决策略。**动态规划**是解决这类优化问题的一种策略。

A classic example of an optimization problem involves making change using the fewest coins. Suppose you are a programmer for a vending machine manufacturer. Your company wants to streamline effort by giving out the fewest possible coins in change for each transaction. Suppose a customer puts in a dollar bill and purchases an item for 37 cents. What is the smallest number of coins you can use to make change? The answer is six coins: two quarters, one dime, and three pennies. How did we arrive at the answer of six coins? We start with the largest coin in our arsenal (a quarter) and use as many of those as possible, then we go to the next lowest coin value and use as many of those as possible. This first approach is called a **greedy method** because we try to solve as big a piece of the problem as possible right away.

一个经典的优化问题涉及使用最少的硬币找零。假设你是一名自动售货机制造商的程序员。你的公司希望通过在每次交易中给出最少的硬币来找零来简化工作。假设一位顾客投入了一美元并购买了一件价值37美分的商品。你能用最少的硬币找零吗？答案是六枚硬币：两枚25美分硬币，一枚10美分硬币和三枚1美分硬币。我们是如何得出六枚硬币的答案的呢？我们从最大的硬币（25美分硬币）开始，尽可能多地使用它们，然后转向下一个面值较小的硬币，并尽可能多地使用它们。这种方法称为**贪婪方法**，因为我们试图立即解决尽可能大的问题部分。

Greedy找不准

The greedy method works fine when we are using U.S. coins, but suppose that your company decides to deploy its vending machines in Lower Elbonia where, in addition to the usual 1, 5, 10, and 25 cent coins they also have a 21 cent coin. In this instance our greedy method fails to find the optimal solution for 63 cents in change. With the addition of the 21 cent coin the greedy method would still find the solution to be six coins. However, the optimal answer is three 21 cent pieces.

贪婪方法在美国硬币系统中效果很好，但假设你的公司决定在下埃博尼亚部署其自动售货机，除了常用的1美分、5美分、10美分和25美分硬币外，他们还有一种21美分的硬币。在这种情况下，我们的贪婪方法无法找到63美分找零的最佳解决方案。增加21美分硬币后，贪婪方法仍然会找到六枚硬币的解决方案。然而，最佳答案是三枚21美分的硬币。

```
1 # Recursive example of trying to get the least amount of coins
2
3 def recMC_greedy(coinValueList: list, change: int) -> int:
4     if change == 0: # base case
```

```

5         return 0
6
7     # use the maximum in the list
8     cur_max = max(coinValueList)
9
10    # find how many of the max is needed to make the change
11    count = change // cur_max
12    index = coinValueList.index(cur_max)
13    del coinValueList[index] # erasing the current max s
14
15    # returns the counts of the coins using recursion
16    return count + recMC_greedy(coinValueList, change - cur_max * count)
17
18
19 # using the greedy algorithm
20 # but greedy algorithm gives 6 coins which is not the most optimum solution
21 print(recMC_greedy([1, 5, 10, 21, 25], 63))
22

```

<https://stackoverflow.com/questions/43233535/explicitly-define-datatype-in-python-function>



Python is a strongly-typed dynamic language, which associates types with *values*, not names. If you want to force callers to provide data of specific types the only way you can do so is by adding explicit checks inside your function.

55



Fairly recently [type annotations](#) were added to the language. and now you can write syntactically correct function specifications including the types of arguments and return values.



The annotated version for your example would be



```
def add(x: float, y: float) -> float:
    return x+y
```

找出最少零钱组合问题可以被视为一个完全背包问题的变种，具体来说，它是一个“完全背包”问题的最优解问题。下面我将详细解释这个问题的背景和解决方法。

问题描述

给定一组不同面额的硬币和一个总金额，找出组成该总金额所需的最少硬币数。如果不能组成该总金额，返回 -1。

完全背包问题

完全背包问题是一种背包问题，其中每种物品可以无限次选择。在最少零钱组合问题中，每种面额的硬币可以无限次使用，因此它可以被视为一个完全背包问题。

动态规划解决方法

我们可以使用动态规划来解决这个问题。具体步骤如下：

1. 定义状态:

- $dp[i]$ 表示组成金额 i 所需的最少硬币数。

2. 初始化:

- $dp[0] = 0$, 因为组成金额 0 所需的硬币数为 0。
- 其他 $dp[i]$ 初始化为一个较大的值 (如 `float('inf')`) , 表示初始状态下无法组成这些金额。

3. 状态转移方程:

- 对于每个金额 i , 遍历所有可用的硬币面额 $coin$, 更新 $dp[i]$:

```
[  
    dp[i] = min(dp[i], dp[i - coin] + 1)  
]
```
- 这个方程的含义是: 如果当前金额 i 可以通过选择一个面额为 $coin$ 的硬币来减少到 $i - coin$, 那么 $dp[i]$ 就是 $dp[i - coin] + 1$ 和当前 $dp[i]$ 的最小值。

4. 返回结果:

- 如果 $dp[amount]$ 仍然是初始的大值, 说明无法组成该金额, 返回 -1。
- 否则, 返回 $dp[amount]$ 。

代码实现

```
1 def min_coins_for_change(amount, coins):  
2     # 初始化 dp 数组, dp[i] 表示组成金额 i 所需的最少硬币数  
3     dp = [float('inf')] * (amount + 1)  
4     dp[0] = 0 # 组成金额 0 所需的硬币数为 0  
5  
6     # 遍历每个金额 i  
7     for i in range(1, amount + 1):  
8         # 遍历每个硬币面额 coin  
9         for coin in coins:  
10             if i >= coin:  
11                 dp[i] = min(dp[i], dp[i - coin] + 1)  
12  
13     # 返回结果  
14     return dp[amount] if dp[amount] != float('inf') else -1  
15  
16 # 示例  
17 amount = 11  
18 coins = [1, 2, 5]  
19 print(min_coins_for_change(amount, coins)) # 输出 3 (5 + 5 + 1)
```

时间复杂度

- 时间复杂度为 $O(n * m)$, 其中 n 是金额 $amount$, m 是硬币面额的数量。
- 空间复杂度为 $O(n)$, 因为我们使用了一个长度为 $amount + 1$ 的 dp 数组。

Recursion找的慢

Let's look at a method where we could be sure that we would find the optimal answer to the problem. Let's start with identifying the base case. If we are trying to make change for the same amount as the value of one of our coins, the answer is easy, one coin.

让我们看看一种可以确保找到问题最优解的方法。让我们从确定基准情况开始。如果我们试图找零的金额正好等于我们某一种硬币的价值，那么答案很简单，只需要一枚硬币。

If the amount does not match we have several options. What we want is the minimum of a penny plus the number of coins needed to make change for the original amount minus a penny, or a nickel plus the number of coins needed to make change for the original amount minus five cents, or a dime plus the number of coins needed to make change for the original amount minus ten cents, and so on. So the number of coins needed to make change for the original amount can be computed according to the following:

如果金额不匹配，我们有多个选项。我们想要的是：一枚1美分硬币加上找零所需硬币数量（原始金额减去1美分），或一枚5美分硬币加上找零所需硬币数量（原始金额减去5美分），或一枚10美分硬币加上找零所需硬币数量（原始金额减去10美分），依此类推。所以，找零所需硬币数量可以根据以下公式计算：

$$num_coins = \min \begin{cases} 1 + num_coins(original\ amount - 1) \\ 1 + num_coins(original\ amount - 5) \\ 1 + num_coins(original\ amount - 10) \\ 1 + num_coins(original\ amount - 25) \end{cases}$$

The algorithm for doing what we have just described is shown in Listing 17. In line 3 we are checking our base case; that is, we are trying to make change in the exact amount of one of our coins. If we do not have a coin equal to the amount of change, we make recursive calls for each different coin value less than the amount of change we are trying to make. Line 6 shows how we filter the list of coins to those less than the current value of change using a list comprehension. The recursive call also reduces the total amount of change we need to make by the value of the coin selected. The recursive call is made in line 7. Notice that on that same line we add 1 to our number of coins to account for the fact that we are using a coin. Just adding 1 is the same as if we had made a recursive call asking where we satisfy the base case condition immediately.

该算法如列表17所示。在第3行，我们检查基准情况，即我们试图找零的金额正好等于我们某一种硬币的价值。如果金额不匹配，我们对每种小于找零金额的硬币值进行递归调用。第6行显示了我们如何使用列表推导式过滤出小于当前找零金额的硬币。递归调用也在第7行进行。注意，在同一行中，我们加1来表示我们使用了一枚硬币。加1的效果就像我们立即满足了基准情况条件一样。

Listing 17: Recursive Version of Coin Optimization Problem

```

1 def make_change_1(coin_denoms, change):
2     if change in coin_denoms:
3         return 1
4     min_coins = float("inf")
5     for i in [c for c in coin_denoms if c <= change]:
6         num_coins = 1 + make_change_1(coin_denoms, change - i)
7         min_coins = min(num_coins, min_coins)
8     return min_coins
9
10
11 print(make_change_1([1, 5, 10, 25], 63))

```

The screenshot shows a LeetCode playground interface. The top navigation bar includes '力扣' (LeetCode), '学习' (Learning), '题库' (Problem Set), '讨论' (Discussion), '竞赛' (Contest), '求职' (Job Hunting), '商店' (Shop), '新功能' (New Features), 'Plus 会员' (Plus Member), and '我是' (I am). The main area has tabs for '执行代码' (Execute Code) and '未命名' (Untitled). The code editor contains the recursive function 'recMC'. The status bar at the bottom right indicates '输出: Time Limit Exceeded'.

```

def recMC(coinValueList,change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList,change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins

print(recMC([1,5,10,21,25],63))

```

The trouble with the algorithm in Listing 17 is that it is extremely inefficient. In fact, it takes 67,716,925 recursive calls to find the optimal solution to the 4 coins, 63 cents problem! To understand the fatal flaw in our approach look at Figure 14, which illustrates a small fraction of the 377 function calls needed to find the optimal set of coins to make change for 26 cents.

列表17中的算法极其低效。事实上，它需要67,716,925次递归调用才能找到四枚硬币63美分问题的最优解！要理解我们方法中的致命缺陷，可以看图14，它展示了找到26美分最优硬币组合所需的377次函数调用的一小部分。

Each node in the graph corresponds to a call to `make_change_1`. The label on the node indicates the amount of change for which we are computing the number of coins. The label on the arrow indicates the coin that we just used. By following the graph we can see the combination of coins that got us to any point in the graph. The main problem is that we are redoing too many calculations. For example, the graph shows that the algorithm would recalculate the optimal number of coins to make change for 15 cents at least three times. Each of these computations to find the optimal number of coins for 15 cents itself takes 52 function calls. Clearly we are wasting a lot of time and effort recalculating old results.

图中的每个节点对应一次 `make_change_1` 调用。节点上的标签表示正在计算找零所需硬币数量的金额。箭头上的标签表示我们刚刚使用的硬币。通过跟随图中的路径，可以看到组合哪些硬币使我们到达图中的任何一点。主要问题是我们在重做太多计算。例如，图显示算法会至少三次重新计算15美分找零的最优硬币数量。每次计算15美分找零的最优硬币数量本身就需要52次函数调用。显然，浪费了很多时间和精力重新计算旧结果。

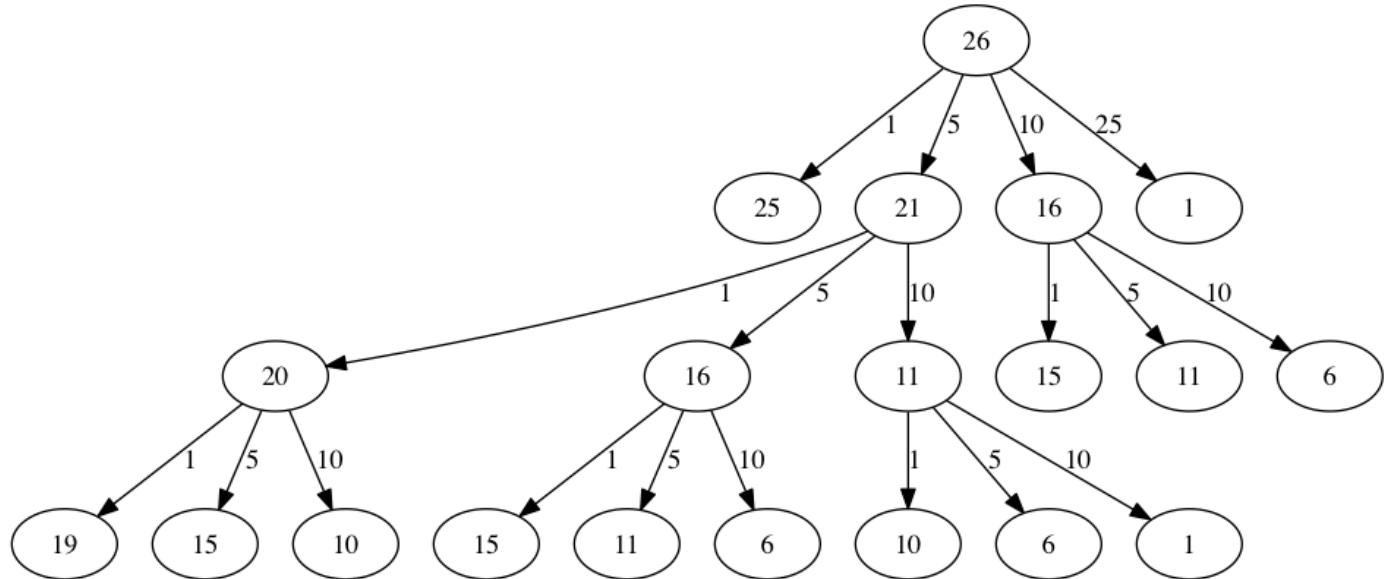


Figure 14: Call Tree for Listing 17

Recursion with memoization 记忆化搜索

The key to cutting down on the amount of work we do is to remember some of the past results so we can avoid recomputing results we already know. A simple solution is to store the results for the minimum number of coins in a table when we find them. Then before we compute a new minimum, we first check the table to see if a result is already known. If there is already a result in the table, we use the value from the table rather than recomputing. ActiveCode 1 shows a modified algorithm to incorporate our table lookup scheme.

减少工作量的关键是记住一些过去的结果，以避免重新计算已知的结果。一个简单的解决方案是在找到最小硬币数量时将其存储在表格中。然后在计算新最小值之前，首先检查表格中是否有已知结果。如果表格中有结果，使用表格中的值而不是重新计算。活动代码4.12.1显示了修改后的算法，以包含我们的表格查找方案。

```

1 def make_change_2(coin_value_list, change, known_results):
2     min_coins = change
3     if change in coin_value_list:
4         known_results[change] = 1
5         return 1
6     elif known_results[change] > 0:
7         return known_results[change]
8     else:
9         for i in [c for c in coin_value_list if c <= change]:
  
```

```

10     num_coins = 1 + make_change_2(coin_value_list, change - i, known_results)
11     if num_coins < min_coins:
12         min_coins = num_coins
13     known_results[change] = min_coins
14 return min_coins
15
16 print(make_change_2([1, 5, 10, 25], 63, [0] * 64))

```

Activity: 4.12.1 Recursively Counting Coins with Table Lookup

Notice that in line 6 we have added a test to see if our table contains the minimum number of coins for a certain amount of change. If it does not, we compute the minimum recursively and store the computed minimum in the table. Using this modified algorithm reduces the number of recursive calls we need to make for the four coin, 63 cent problem to 221 calls!

注意，在第6行，添加了一个测试，以查看我们的表格是否包含特定找零金额的最小硬币数量。如果没有，递归计算最小值并将计算出的最小值存储在表格中。使用这种修改后的算法将四枚硬币63美分问题所需的递归调用次数减少到221次！

The screenshot shows a Python code editor on the LeetCode playground. The code uses `lru_cache` from the `functools` module to implement memoization. The code defines a function `recMC` that takes a list of coin values and a target change. It returns the minimum number of coins required. The code includes a check to update the `known_results` table if the current value is less than the previously recorded minimum. The output panel shows the code was executed successfully, with a note about a `TypeError` due to an unhashable list being passed to `lru_cache`, and a final result of 3.

```

1 from functools import lru_cache
2
3 @lru_cache(maxsize = None)
4 def recMC(coinValueList, change):
5     minCoins = change
6     if change in coinValueList:
7         return 1
8     else:
9         for i in [c for c in coinValueList if c <= change]:
10            numCoins = 1 + recMC(coinValueList, change-i)
11            if numCoins < minCoins:
12                minCoins = numCoins
13
14 return minCoins
15
16 print(recMC(tuple([1,5,10,21,25]),63))

```

Although the algorithm in ActiveCode 1 is correct, it looks and feels like a bit of a hack. Also, if we look at the `known_results` lists we can see that there are some holes in the table. In fact the term for what we have done is not dynamic programming but rather we have improved the performance of our program by using a technique known as *memoization*, or more commonly called *caching*.

尽管活动代码1中的算法是正确的，但它看起来和感觉像是一种修补。此外，如果查看`known_results`列表，可以看到表格中有一些空白。事实上，所做的技术被称为记忆化（memoization），更常见的是缓存（caching），而不是动态规划。

Truly dynamic programming递推写法

A truly dynamic programming algorithm will take a more systematic approach to the problem. Our dynamic programming solution is going to start with making change for one cent and systematically work its way up to the amount of change we require. This guarantees that at each step of the algorithm we already know the minimum number of coins needed to make change for any smaller amount.

一个真正的动态规划算法将对问题采取更系统的方法。我们的动态规划解决方案将从找零1美分开始，系统地向上推进到所需的找零金额。这保证了在算法的每一步，我们已经知道任何较小金额的最小硬币数量。

Let's look at how we would fill in a table of minimum coins to use in making change for 11 cents. Figure 15 illustrates the process. We start with one cent. The only solution possible is one coin (a penny). The next row shows the minimum for one cent and two cents. Again, the only solution is two pennies. The fifth row is where things get interesting. Now we have two options to consider, five pennies or one nickel. How do we decide which is best? We consult the table and see that the number of coins needed to make change for four cents is four, plus one more penny to make five, equals five coins. Or we can look at zero cents plus one more nickel to make five cents equals one coin. Since the minimum of one and five is one we store 1 in the table. Fast forward again to the end of the table and consider 11 cents. Figure 16 shows the three options that we have to consider:

让我们看看如何填充一个最小硬币数量表，以用于11美分的找零。图15展示了这个过程。从1美分开始。唯一的解决方案是一枚硬币（1美分）。下一行显示了1美分和2美分的最小值。同样，唯一的解决方案是两枚1美分硬币。第五行是有趣的地方。现在有两个选择：五枚1美分硬币或一枚5美分硬币。如何决定哪个更好？我们查阅表格，看到找零4美分所需的硬币数量是四枚，再加上一枚1美分硬币，总共五枚硬币。或者可以看0美分加上一枚5美分硬币，总共一枚硬币。由于1和5的最小值是1，我们在表格中存储1。跳到表格的末尾，考虑11美分。图16显示了我们需要考虑的三个选项：

1. A penny plus the minimum number of coins to make change for $11 - 1 = 10$ cents (1)
2. A nickel plus the minimum number of coins to make change for $11 - 5 = 6$ cents (2)
3. A dime plus the minimum number of coins to make change for $11 - 10 = 1$ cent (1)

Either option 1 or 3 will give us a total of two coins which is the minimum number of coins for 11 cents.

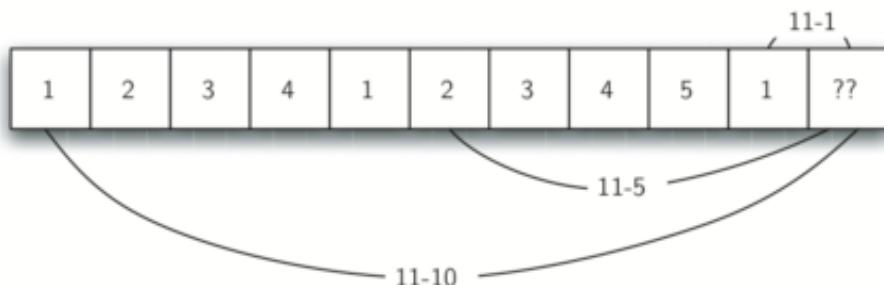
选项1或3都会给我们总共两枚硬币，这是11美分的最小硬币数量。

Figure 15: Minimum Number of Coins Needed to Make Change

Change to Make

1	2	3	4	5	6	7	8	9	10	11
1										
1	2									
1	2	3								
1	2	3	4							
1	2	3	4	1						
...										
1	2	3	4	1	2	3	4	5	1	
1	2	3	4	1	2	3	4	5	1	2

Figure 16: Three Options to Consider for the Minimum Number of Coins for Eleven Cents



Listing 19 is a dynamic programming algorithm to solve our change-making problem. `make_change_3` takes three parameters: a list of valid coin values, the amount of change we want to make, and a list of the minimum number of coins needed to make each value. When the function is done, `min_coins` will contain the solution for all values from 0 to the value of `change`.

列表19是一个动态规划算法，用于解决我们的找零问题。`make_change_3`接受三个参数：有效的硬币值列表、我们要找零的金额，以及一个列表，用于存储制作每个值所需的最小硬币数量。当函数完成时，`min_coins`将包含从0到`change`值的所有值的解决方案。

Listing 19: Dynamic Programming Solution

```

1 def make_change_3(coin_value_list, change, min_coins):
2     for cents in range(change + 1):
3         coin_count = cents
4         for j in [c for c in coin_value_list if c <= cents]:
5             if min_coins[cents - j] + 1 < coin_count:
6                 coin_count = min_coins[cents - j] + 1
7         min_coins[cents] = coin_count
8     return min_coins[change]
9
10 print(make_change_3([1, 5, 10, 21, 25], 63, [0]*64))

```

Note that `make_change_3` is not a recursive function, even though we started with a recursive solution to this problem. It is important to realize that a recursive solution to a problem will not necessarily be the most efficient solution. The bulk of the work in this function is done by the loop that starts on line 4. In this loop we consider using all possible coins to make change for the amount specified by `cents`. Like we did for the 11 cent example above, we remember the minimum value and store it in our `min_coins` list.

请注意，`make_change_3`不是一个递归函数，即使我们是从递归解决方案开始的。重要的是要意识到，递归解决方案不一定是解决问题的最有效方法。该函数的主要工作是由从第4行开始的循环完成的。在这个循环中，我们考虑使用所有可能的硬币来制作指定金额的找零。就像上面11美分的例子一样，我们记住最小值并将其存储在`min_coins`列表中。

Although our making change algorithm does a good job of figuring out the minimum number of coins, it does not help us make change since we do not keep track of the coins we use. We can easily extend `make_change_3` to keep track of the coins used by simply remembering the last coin we add for each entry in the `min_coins` table. If we know the last coin added, we can simply subtract the value of the coin to find a previous entry in the table that tells us the last coin we added to make that amount. We can keep tracing back through the table until we get to the beginning.

尽管我们的找零算法在计算最小硬币数量方面做得很好，但它并没有帮助我们实际找零，因为我们没有跟踪所使用的硬币。我们可以通过简单地记住`min_coins`表中每个条目的最后一枚硬币来轻松扩展`make_change_3`。如果我们知道最后一枚添加的硬币，我们只需减去该硬币的价值即可找到表中的前一个条目，该条目告诉我们添加最后一枚硬币以制作该金额所需的硬币。我们可以一直回溯到表的开头。

ActiveCode 2 shows `make_change_4`, based on the `make_change_3` algorithm but modified to keep track of the coins used, along with a function `print_coins` that walks backward through the table to print out the value of each coin used. This shows the algorithm in action solving the problem for our friends in Lower Elbonia. The first two lines of `main` set the amount to be converted and create the list of coins used. The next two lines create the lists we need to store the results. `coins_used` is a list of the coins used to make change, and `coin_count` is the minimum number of coins used to make change for the amount corresponding to the position in the list.

活动代码2 显示了基于 `make_change_3` 算法但修改为跟踪所使用硬币的 `make_change_4`，以及一个函数 `print_coins`，该函数向后遍历表以打印出每个使用的硬币的值。这展示了算法如何实际解决我们下埃博尼亞朋友的问题。`main` 函数的前两行设置了要转换的金额并创建了使用的硬币列表。接下来的两行创建了我们存储结果所需的列表。`coins_used` 是用于找零的硬币列表，而 `coin_count` 是用于找零对应金额的最小硬币数量。

Notice that the coins we print out come directly from the `coins_used` array. For the first call we start at array position 63 and print 21. Then we take and look at the 42nd element of the list. Once again we find a 21 stored there. Finally, element 21 of the array also contains 21, giving us the three 21 cent pieces.

注意，我们打印出的硬币直接来自 `coins_used` 数组。对于第一次调用，我们从数组位置63开始，打印出 21。然后我们查看列表中的第42个元素，再次找到存储的21。最后，数组的第21个元素也包含21，这给了我们三枚21美分的硬币。

```
1 def make_change_4(coin_value_list, change, min_coins, coins_used):
2     for cents in range(change + 1):
3         coin_count = cents
4         new_coin = 1
5         for j in [c for c in coin_value_list if c <= cents]:
6             if min_coins[cents - j] + 1 < coin_count:
7                 coin_count = min_coins[cents - j] + 1
8                 new_coin = j
9             min_coins[cents] = coin_count
10            coins_used[cents] = new_coin
11    return min_coins[change]
12
13
14 def print_coins(coins_used, change):
15     coin = change
16     while coin > 0:
17         this_coin = coins_used[coin]
18         print(this_coin, end=" ")
19         coin = coin - this_coin
20     print()
21
22
23 def main():
24     amnt = 63
25     clist = [1, 5, 10, 21, 25]
26     coins_used = [0] * (amnt + 1)
27     coin_count = [0] * (amnt + 1)
28
29     print(
30         "Making change for {} requires the following {} coins: ".format(
31             amnt, make_change_4(clist, amnt, coin_count, coins_used)
32         ),
33         end="",
34     )
```

```
35     print_coins(coins_used, amnt)
36     print("The used list is as follows:")
37     print(coins_used)
38
39
40 main()
```

2 动态规划的递归写法和递推写法

《算法笔记》第11.1节

动态规划是一种非常精妙的算法思想，它没有固定的写法、极其灵活，常常需要具体问题具体分析。学习方式是先接触一些经典模型，这样会有更好的效果。在介绍一些动态规划经典模型中，穿插动态规划的概念，慢慢接触动态规划。同时多练习、多思考、多总结是学习动态规划的重点。

什么是动态规划

动态规划(Dynamic Programming, DP)是一种用来解决一类最优化问题的算法思想。简单来说，动态规划将一个复杂的问题分解成若干个子问题，通过综合子问题的最优解来得到原问题的最优解。需要注意的是，动态规划会将每个求解过的子问题的解记录下来，这样当下一次碰到同样的子问题时，就可以直接使用之前记录的结果，而不是重复计算。注意：虽然动态规划采用这种方式来提高计算效率，但不能说这种做法就是动态规划的核心。

一般可以使用递归或者递推的写法来实现动态规划，其中递归写法在此处又称作记忆化搜索。

2.1 动态规划的递归写法

先来讲解递归写法。通过这部分内容的学习，理解动态规划是如何记录子问题的解，来避免下次遇到相同的子问题时的重复计算的。

以斐波那契 (Fibonacci) 数列为例。

示例02753: 斐波那契数列

math,recursion, dp, <http://cs101.openjudge.cn/practice/02753>

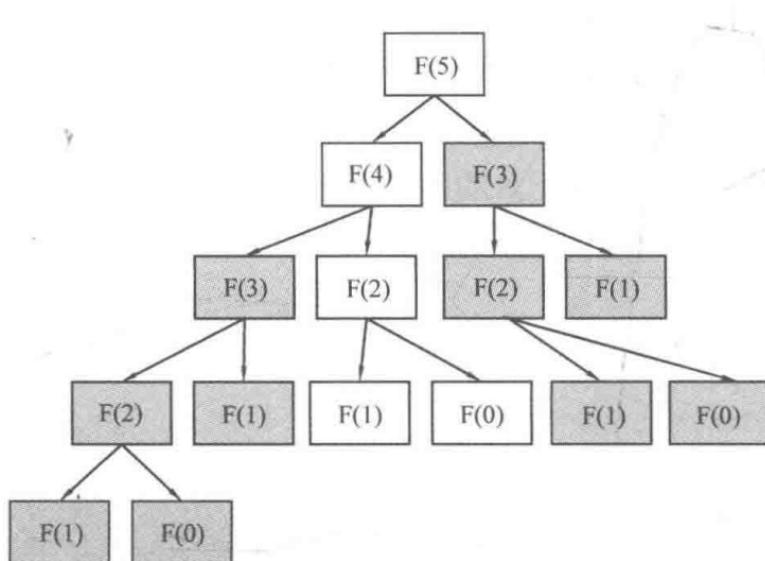
斐波那契数列是指这样的数列：数列的第一个和第二个数都为1，接下来每个数都等于前面2个数之和。
给出一个正整数a，要求斐波那契数列中第a个数是多少。

斐波那契 (Fibonacci) 数列的定义为 $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ ($n \geq 2$)

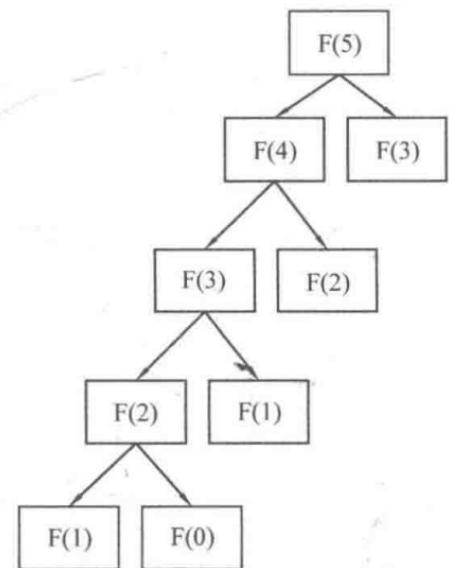
```
1 def f(n):
2     if n <= 2:
3         return 1
4     else:
5         return f(n-1)+f(n-2)
6
7
8 n = int(input())
9 ans = []
10 for _ in range(n):
11     num = int(input())
12     ans.append(f(num))
13
14 print('\n'.join(map(str, ans)))
```

事实上，这个递归会涉及很多重复的计算。如图A所示，当n=5时，可以得到 $F(5) = F(4) + F(3)$ ，接下来在计算 $F(4)$ 时又会有 $F(4) = F(3) + F(2)$ 。这时候如果不采取措施， $F(3)$ 将会被计算两次。可以推知，如果 n 很大，重复计算的次数将难以想象。事实上，由于没有及时保存中间计算的结果，实际复杂度会高达 $O(2^n)$ ，即每次都会计算 $F(n-1)$ 和 $F(n-2)$ 这两个分支，基本不能承受 n 较大的情况。

为了避免重复计算，可以开一个一维数组 dp，用以保存已经计算过的结果，其中 $dp[n]$ 记录 $F(n)$ 的结果，并用 $dp[n] = -1$ 表示 $F(n)$ 当前还没有被计算过。然后就可以在递归当中判断 $dp[n]$ 是否是 -1：如果不是 -1，说明已经计算过 $F(n)$ ，直接返回 $dp[n]$ 就是结果，否则，按照递归式进行递归。



图A 斐波那契数列递归图



图B 斐波那契数列记忆化搜索示意图

这样就把已经计算过的内容记录了下来，于是当下次再碰到需要计算相同的内容时，就能直接使用上次计算的结果，这可以省去大半无效计算，而这也是记忆化搜索这个名字的由来。如图B所示，通过记忆化搜索，把复杂度从 $O(2^n)$ 降到了 $O(n)$ ，也就是说，用一个 $O(n)$ 空间的力量就让复杂度从指级别降低到了线性级别。

```

1 def f(n):
2     if n <= 2:
3         return 1
4
5     if dp[n] != -1:
6         return dp[n]
7     else:
8         dp[n] = f(n-1)+f(n-2)
9     return dp[n]
10
11
12 dp = [-1]*21
13 n = int(input())
14 ans = []
15 for _ in range(n):
16     num = int(input())
17     ans.append(f(num))
18
19 print('\n'.join(map(str, ans)))
  
```

通过上面的例子可以引申出一个概念：如果一个问题可以被分解为若干个子问题，且这些子问题会重复出现，那么就称这个问题拥有重叠子问题（Overlapping Subproblems）。动态规划通过记录重叠子问题的解，来使下次碰到相同的子问题时直接使用之前记录的结果以此避免大量重复计算。因此，**一个问题必须拥有重叠子问题，才能使用动态规划去解决。**

```
1  '''
2 Python Functools - lru_cache(),
3 https://www.geeksforgeeks.org/python-functools-lru-cache/
4
5 The LRU caching scheme is to remove the least recently used frame when the
6 cache is full and a new page is referenced which is not there in the cache.
7 https://www.geeksforgeeks.org/python-lru-cache/
8 '''
9 from functools import lru_cache
10
11 @lru_cache(maxsize = 128)
12 def f(n):
13     if n <= 2:
14         return 1
15     else:
16         return f(n-1)+f(n-2)
17
18
19 n = int(input())
20 list_1 = []
21 for i in range(n):
22     num = int(input())
23     list_1.append(f(num))
24 for i in list_1:
25     print(i)
```

2.2 动态规划的递推写法

以经典的数塔问题为例，如图 11-3 所示，将一些数字排成数塔的形状，其中第一层有一个数字，第二层有两个数字……第n层有n 个数字。现在要从第一层走到第n 层，每次只能走向下一层连接的两个数字中的一个，问：最后将路径上所有数字相加后得到的和最大是多少？

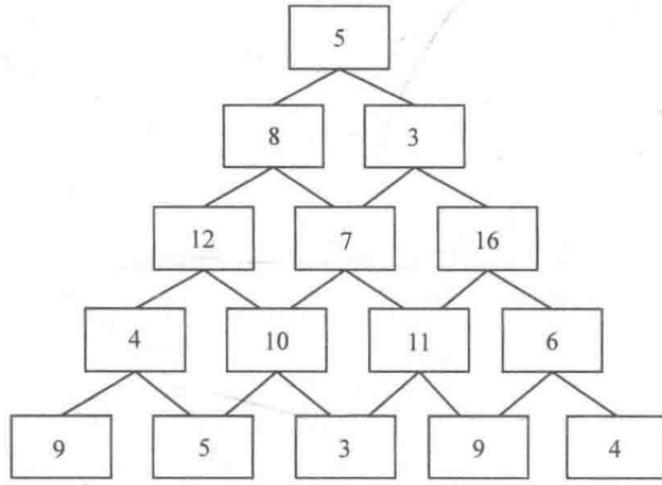


图 11-3 数塔问题示意图

按照题目的描述，如果开一个二维数组 f ，其中 $f[i][j]$ 存放第 i 层的第 j 个数字，那么就有 $f[1][1] = 5$, $f[2][1] = 8$, $f[2][2] = 3$, $f[3][1] = 12$, ..., $f[5][4] = 9$, $f[5][5] = 4$ 。

此时，如果尝试穷举所有路径，然后记录路径上数字和的最大值，那么由于每层中的每个数字都会有两条分支路径，因此可以得到时间复杂度为 $O(2^n)$ ，这在 n 很大的情况下是不可接受的。那么，产生这么大复杂度的原因是什么？下面来分析一下。一开始，从第一层的 5 出发，按 $5 \rightarrow 8 \rightarrow 7$ 的路线来到 7，并枚举从 7 出发的到达最底层的所有路径。但是，之后当按 $5 \rightarrow 3 \rightarrow 7$ 的路线再次来到 7 时，又会去枚举从 7 出发的到达最底层的所有路径，这就导致了从 7 出发的到达最底层的所有路径都被反复地访问，做了许多多余的计算。事实上，可以在第一次枚举从 7 出发的到达最底层的所有路径时就把路径上能产生的最大和记录下来，这样当再次访问到 7 这个数字时就可以直接获取这个最大值，避免重复计算。

由上面的考虑，不妨令 $dp[i][j]$ 表示从第 i 行第 j 个数字出发的到达最底层的所有路径中能得到的最大和，例如 $dp[3][2]$ 就是图中的 7 到最底层的路径最大和。在定义这个数组之后 $dp[1][1]$ 就是最终想要的答案，现在想办法求出它。

注意到一个细节：如果要求出“从位置(1,1)到达最底层的最大和” $dp[1][1]$ ，那么一定要先求出它的两个子问题“从位置(2,1)到达最底层的最大和 $dp[2][1]$ ”和“从位置(2,2)到达最底层的最大和 $dp[2][2]$ ”，即进行了一次决策：走数字 5 的左下还是右下。于是 $dp[1][1]$ 就是 $dp[2][1]$ 和 $dp[2][2]$ 的较大值加上 5。写成式子就是：

$$dp[1][1] = \max(dp[2][1], dp[2][2]) + f[1][1]$$

由此可以归纳得到这么一个信息：如果要求 $dp[i][j]$ ，那么一定要先求出它的两个子问题“从位置($i+1,j$)到达最底层的最大和 $dp[i+1][j]$ ”和“从位置($i+1,j+1$)到达最底层的最大和 $dp[i+1][j+1]$ ”，即进行了一次决策：走位置(i,j)的左下还是右下。于是 $dp[i][j]$ 就是 $dp[i+1][j]$ 和 $dp[i+1][j+1]$ 的较大值加上 $f[i][j]$ 。写成式子就是：

$$dp[i][j] = \max(dp[i+1][j], dp[i+1][j+1]) + f[i][j]$$

把 $dp[i][j]$ 称为问题的状态，而把上面的式子称作状态转移方程，它把状态 $dp[i][j]$ 转移为 $dp[i+1][j]$ 和 $dp[i+1][j+1]$ 。可以发现，状态 $dp[i][j]$ 只与第 $i+1$ 层的状态有关，而与其他层的状态无关，这样层号为 i 的状态就总是可以由层号为 $i+1$ 的两个子状态得到。那么，如果总是将层号增大，什么时候会到头呢？可以发现，数塔的最后一层的 dp 值总是等于元素本身即 $dp[n][j] = f[n][j]$ ($1 \leq j \leq n$)，把这种可以直接确定其结果的部分称为边界，而动态规划的递推写法总是从这些边界出发，通过状态转移方程扩散到整个 dp 数组。

这样就可以从最底层位置的dp值开始，不断往上求出每一层各位置的dp值，最后就会得到 $dp[1][1]$ ，即为想要的答案。

示例02760: 数字三角形

我们结合 02760: 数字三角形，先给出超时的递归写法，然后给出递归写法实现的动态规划，再给出递推写法实现的动态规划。

dp/dfs similar, <http://cs101.openjudge.cn/practice/02760>

```
1      7
2      3   8
3      8   1   0
4      2   7   4   4
5      4   5   2   6   5
6
7      (图1)
```

图1给出了一个数字三角形。从三角形的顶部到底部有很多条不同的路径。对于每条路径，把路径上面的数加起来可以得到一个和，你的任务就是找到最大的和。

注意：路径上的每一步只能从一个数走到下一层上和它最近的那个数或者右边的那个数。

只递归，不用dp，立马超时。

```
1 def f(i, j):                      # Time Limit Exceeded, 9953ms
2     if i == N-1:
3         return tri[i][j]
4
5     return max(f(i+1, j), f(i+1, j+1)) + tri[i][j]
6
7
8 N = int(input())
9 tri = []
10 for _ in range(N):
11     tri.append([int(i) for i in input().split()])
12 print(f(0, 0))
```

使用递归写法来实现动态规划，又称作记忆化搜索。没错，即使用lru_cache，也是纯正的dp。零基础同学应该容易理解。至少希望是。

```
1 from functools import lru_cache
2
3 @lru_cache(maxsize = 128)
```

```

4 def f(i, j):
5     if i == N-1:
6         return tri[i][j]
7
8     return max(f(i+1, j), f(i+1, j+1)) + tri[i][j]
9
10
11 N = int(input())
12 tri = []
13 for _ in range(N):
14     tri.append([int(i) for i in input().split()])
15 print(f(0, 0))

```

递推写法实现的动态规划。

```

1 N = int(input())
2 tri = []
3 for _ in range(N):
4     tri.append([int(i) for i in input().split()])
5
6 dp = [[0]*N for _ in range(N)]
7 for j in range(N):
8     dp[N-1][j] = tri[N-1][j]
9
10 for i in range(N-2, -1, -1):
11     for j in range(i+1):
12         dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + tri[i][j]
13
14 print(dp[0][0])

```

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

Python 3.6
known limitations

```

1 N = int(input())
2 tri = []
3 for _ in range(N):
4     tri.append([int(i) for i in input().split()])
5
6 dp = [[0]*N for _ in range(N)]
7 for j in range(N):
8     dp[N-1][j] = tri[N-1][j]    初值
9
10 for i in range(N-2, -1, -1):
11     for j in range(i+1):
12         dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + tri[i][j]
13
14 print(dp[0][0])

```

[Edit this code](#)

line that just executed
next line to execute

<< First < Prev Next >> Last >>

Step 63 of 93

NEW: follow our [YouTube](#), [TikTok](#), and [Instagram](#) for free tutorials

[Get AI Help](#)

[Move and hide objects](#)

Print output (drag lower right corner to resize)

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

Frames Objects

Global frame

- N 5
- tri [● 4]
- 4
- dp [● 4]
- j 4

显然，使用递归也可以实现上面的例子(即从 $dp[0][0]$ 开始递归，直至到达边界时返回结果)。两者的区别在于：使用递推写法的计算方式是自底向上(**Bottom-up Approach**)，即从边界开始，不断向上解决问题，直到解决了目标问题；而使用递归写法的计算方式是自顶向下(**Top-down Approach**)，即从目标问题开始，将它分解成子问题的组合，直到分解至边界为止。

通过上面的例子再复习一个概念：如果一个问题的最优解可以由其子问题的最优解有效地构造出来，那么称这个问题拥有**最优子结构(Optimal Substructure)**。最优子结构保证了动态规划中原问题的最优解可以由子问题的最优解推导而来。因此，一个问题必须拥有最优子结构，才能使用动态规划去解决。例如数塔问题中，每一个位置的 dp 值都可以由它的两个子问题推导得到。

至此，重叠子问题和最优子结构的内容已介绍完毕。需要指出，一个问题必须拥有重叠子问题和最优子结构，才能使用动态规划去解决。下面指出这两个概念的区别：

① 分治与动态规划。分治和动态规划都是将问题分解为子问题，然后合并子问题的解得到原问题的解。但是不同的是，分治法分解出的子问题是不重叠的，因此分治法解决的问题不拥有重叠子问题，而动态规划解决的问题拥有重叠子问题。例如，归并排序和快速排序都是分别处理左序列和右序列，然后将左右序列的结果合并，过程中不出现重叠子问题，因此它们使用的都是分治法。另外，分治法解决的问题不一定是最优化问题，而动态规划解决的问题一定是最优化问题。

② 贪心与动态规划。贪心和动态规划都要求原问题必须拥有最优子结构。二者的区别在于，贪心法采用的计算方式类似于上面介绍的“自顶向下”，但是并不等待子问题求解完毕后再选择使用哪一个，而是通过一种策略直接选择一个子问题去求解，没被选择的子问题就不去求解了，直接抛弃。也就是说，它总是只在上一步选择的基础上继续选择，因此整个过程以一种单链的流水方式进行，显然这种所谓“最优选择”的正确性需要用归纳法证明。例如对数塔问题而言，贪心法从最上层开始，每次选择左下和右下两个数字中较大的一个，一直到最底层得到最后结果，显然这不一定可以得到最优解。而动态规划不管是采用自底向上还是自顶向下的计算方式，都是从边界开始向上得到目标问题的解。也就是说，它总是会考虑所有子问题，并选择继承能得到最优结果的那个，对暂时没被继承的子问题，由于重叠子问题的存在，后期可能会再次考虑它们，因此还有机会成为全局最优的一部分，不需要放弃。所以贪心是一种壮士断腕的决策，只要进行了选择，就不后悔；动态规划则要看哪个选择笑到了最后，暂时的领先说明不了什么。

随着动态规划的学习，会对上面的内容不断深化理解，因此可以暂时不必太过拘泥于部分细节，之后再回过头来看，可能会有更深的理解。

3 最大连续子序列和

Longest Continuous Subsequence Sum/ Kadane's Algorithm

示例：最大连续子序列和（LCSS）

<https://sunnywhy.com/sfbj/11/2>

现有一个整数序列 a_1, a_2, \dots, a_n , 求连续子序列 $a_i + \dots + a_j$ 的最大值。

输入

第一行一个正整数 $n (1 \leq n \leq 10^4)$, 表示序列长度;

第二行为用空格隔开的 n 个整数 $a_i (-10^5 \leq a_i \leq 10^5)$, 表示序列元素。

输出

输出一个整数, 表示最大连续子序列和。

样例1

输入

1	6
2	-2 11 -4 13 -5 -2

输出

1	20
---	----

解释: 连续子序列和的最大值为: $11 + (-4) + 13 = 20$

这个问题如果暴力来做, 枚举左端点和右端点(即枚举 i, j)需要 $O(n^2)$ 的复杂度, 而计算 $A[i] + \dots + A[j]$ 需要 $O(n)$ 的复杂度, 因此总复杂度为 $O(n^3)$ 。就算采用记录前缀和的方法(预处理 $S[i] = A[0] + A[1] + \dots + A[i]$, 这样 $A[i] + \dots + A[j] = S[j] - S[i-1]$)使计算的时间变为 $O(1)$, 总复杂度仍然有 $O(n^2)$, 这对 n 为 10^5 大小的题目来说是无法承受的。

下面介绍动态规划的做法, 复杂度为 $O(n)$ 。

通过设置这么一个 dp 数组, 要求的最大和其实就是 $dp[0], dp[1], \dots, dp[n-1]$ 中的最大值, 想办法求解 dp 数组。因为 $dp[i]$ 要求是必须以 $A[i]$ 结尾的连续序列, 那么只有两种情况:

①这个最大和的连续序列只有一个元素, 即以 $A[i]$ 开始, 以 $A[i]$ 结尾。

②这个最大和的连续序列有多个元素, 即从前面某处 $A[p]$ 开始($p < i$), 一直到 $A[i]$ 结尾。

对第一种情况, 最大和就是 $A[i]$ 本身。

对第二种情况, 最大和是 $dp[i-1] + A[i]$, 即 $A[p] + \dots + A[i-1] + A[i] = dp[i-1] + A[i]$ 。

由于只有这两种情况, 于是得到状态转移方程:

$$dp[i] = \max(A[i], dp[i - 1] + A[i])$$

这个式子只和 `i` 与 `i之前` 的元素有关，且边界为 `dp[0]=A[0]`，由此从小到大枚举 `i`，即可得到整个 `dp` 数组。接着输出 `dp[0],dp[1],...,dp[n-1]` 中的最大值即为最大连续子序列的和。

只用 $O(n)$ 的时间复杂度就解决了原先需要 $O(n^2)$ 复杂度问题，这就是动态规划的魅力。

```

1 n = int(input())
2 *a, = map(int, input().split())
3
4 dp = [0]*n
5 dp[0] = a[0]
6
7 for i in range(1, n):
8     dp[i] = max(dp[i-1]+a[i], a[i])
9
10 print(max(dp))

```

此处顺便介绍无后效性的概念。状态的无后效性是指：当前状态记录了历史信息，一旦当前状态确定，就不会再改变，且未来的决策只能在已有的一个或若干个状态的基础上进行，历史信息只能通过已有的状态去影响未来的决策。而针对上面的问题来说，每次计算状态 `dp[i]`，都只会涉及 `dp[i-1]`，而不直接用到 `dp[i-1]` 蕴含的历史信息。对动态规划可解的问题来说，总会有很多设计状态的方式，但并不是所有状态都具有无后效性，因此必须设计一个拥有无后效性的状态以及相应的状态转移方程，否则动态规划就没有办法得到正确结果。事实上，**如何设计状态和状态转移方程，才是动态规划的核心，而它们也是动态规划最难的地方。**

题面如果问，最大连续子序列和的最优方案。

```

1 n = int(input())
2 *a, = map(int, input().split())
3
4 dp = [0]*n
5 start =[0]*n
6 dp[0] = a[0]
7
8 for i in range(1, n):
9     if (dp[i-1] >= 0):
10         dp[i] = dp[i-1] + a[i]
11         start[i] = start[i-1]
12     else:
13         dp[i] = a[i]
14         start[i] = i
15
16 max_val = max(dp)
17 pos = dp.index(max_val)
18
19 print(max_val, start[pos]+1, pos+1)

```

练习02766: 最大子矩阵

dp, <http://cs101.openjudge.cn/practice/02766/>

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵，你的任务是找到最大的非空(大小至少是 $1 * 1$)子矩阵。

比如，如下 $4 * 4$ 的矩阵

```
0 -2 -7 0  
9 2 -6 2  
-4 1 -4 1  
-1 8 0 -2
```

的最大子矩阵是

```
9 2  
-4 1  
-1 8
```

这个子矩阵的大小是15。

输入

输入是一个 $N * N$ 的矩阵。输入的第一行给出 N ($0 < N \leq 100$)。再后面的若干行中，依次（首先从左到右给出第一行的 N 个整数，再从左到右给出第二行的 N 个整数……）给出矩阵中的 N^2 个整数，整数之间由空白字符分隔（空格或者空行）。已知矩阵中整数的范围都在 $[-127, 127]$ 。

输出

输出最大子矩阵的大小。

样例输入

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

样例输出

1	15
---	----

来源：翻译自 Greater New York 2001 的试题

Kadane's Algorithm 是一种高效的算法，用于在一维数组中找到具有最大和的连续子数组。它的核心思想是通过一次遍历来实现，时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

```

1  """
2  为了找到最大的非空子矩阵，可以使用动态规划中的Kadane算法进行扩展来处理二维矩阵。
3  基本思路是将二维问题转化为一维问题：可以计算出从第i行到第j行的列的累计和，
4  这样就得到了一个一维数组。然后对这个一维数组应用Kadane算法，找到最大的子数组和。
5  通过遍历所有可能的行组合，我们可以找到最大的子矩阵。
6  """
7  def max_submatrix(matrix):
8      def kadane(arr):
9          max_end_here = max_so_far = arr[0]
10         for x in arr[1:]:
11             max_end_here = max(x, max_end_here + x)
12             max_so_far = max(max_so_far, max_end_here)
13         return max_so_far
14
15     rows = len(matrix)
16     cols = len(matrix[0])
17     max_sum = float('-inf')
18
19     for left in range(cols):
20         temp = [0] * rows
21         for right in range(left, cols):
22             for row in range(rows):
23                 temp[row] += matrix[row][right]
24             max_sum = max(max_sum, kadane(temp))
25     return max_sum
26
27 n = int(input())
28 nums = []
29
30 while len(nums) < n * n:
31     nums.extend(input().split())
32 matrix = [list(map(int, nums[i * n:(i + 1) * n])) for i in range(n)]
33
34 max_sum = max_submatrix(matrix)
35 print(max_sum)

```

4 最大上升子序列 (LIS)

Longest Increasing Subsequence

示例02533: Longest Ordered Subsequence

dp, <http://cs101.openjudge.cn/practice/02533>

与这个题目相同：

OJ2757: 最长上升子序列

dp, <http://cs101.openjudge.cn/practice/02757>

A numeric sequence of a_i is ordered if $a_1 < a_2 < \dots < a_N$. Let the subsequence of the given numeric sequence (a_1, a_2, \dots, a_N) be any sequence $(a_{i_1}, a_{i_2}, \dots, a_{i_K})$, where $1 \leq i_1 < i_2 < \dots < i_K \leq N$. For example, sequence $(1, 7, 3, 5, 9, 4, 8)$ has ordered subsequences, e. g., $(1, 7), (3, 4, 8)$ and many others. All longest ordered subsequences are of length 4, e. g., $(1, 3, 5, 8)$.

Your program, when given the numeric sequence, must find the length of its longest ordered subsequence.

输入

The first line of input file contains the length of sequence N . The second line contains the elements of sequence - N integers in the range from 0 to 10000 each, separated by spaces. $1 \leq N \leq 1000$

输出

Output file must contain a single integer - the length of the longest ordered subsequence of the given sequence.

样例输入

1	7
2	1 7 3 5 9 4 8

样例输出

1	4
---	---

来源

Northeastern Europe 2002, Far-Eastern Subregion

对于这个问题，可以用最原始的办法来枚举每种情况，即对于每个元素有取和不取两种选择，然后判断序列是否为上升序列。如果是上升序列，则更新最大长度，直到枚举完所有情况并得到最大长度。但是很严峻的一个问题是，由于需要对每个元素都选择取或者不取，那么如果元素有 n 个，时间复杂度将高达 $O(2^n)$ ，这显然是不能承受的。

事实上这个枚举过程包含了大量重复计算。那么这些重复计算源自哪里呢？不妨先来看动态规划的解法，之后就会容易理解为什么会有重复计算产生了（下文中出现的 LIS 均指最大上升子序列）。

令 $dp[i]$ 表示以 $A[i]$ 结尾的最长上升子序列长度（和最大连续子序列一样，以 $A[i]$ 结尾是强制的要求）。这样对 $A[i]$ 来说就会有两种可能：

① 如果存在 $A[i]$ 之前的元素 $A[j] (j < i)$ ，使得 $A[j] < A[i]$ 且 $dp[j] + 1 > dp[i]$ （即把 $A[i]$ 跟在以 $A[j]$ 结尾的 LIS 后面时能比当前以 $A[i]$ 结尾的 LIS 长度更长），那么就把 $A[i]$ 跟在以 $A[j]$ 结尾的 LIS 后面，形成一条更长的上升子序列（令 $dp[i] = dp[j] + 1$ ）。

② 如果 $A[i]$ 之前的元素都比 $A[i]$ 大，那么 $A[i]$ 就只好自己形成一条 LIS，但是长度为 1，即这个子序列里面只有一个 $A[i]$ 。

最后以 $A[i]$ 结尾的 LIS 长度就是①②中能形成的最大长度。

由此可以写出状态转移方程：

$$dp[i] = \max(1, dp[j] + 1), (j = 1, 2, \dots, i - 1 \ \&\& \ A[j] < A[i])$$

上面的状态转移方程中隐含了边界： $dp[i] = 1 (1 \leq i \leq n)$ 。显然 $dp[i]$ 只与小于 i 的 j 有关，因此只要让 i 从小到大遍历即可求出整个 dp 数组。由于 $dp[i]$ 表示的是以 $A[i]$ 结尾的 LIS 长度，因此从整个 dp 数组中找出最大的那个才是要寻求的整个序列的 LIS 长度，整体复杂度为 $O(n^2)$ 。

到此就可以想象究竟重复计算出现在哪里了：每次碰到子问题“以 $A[i]$ 结尾的最大上升子序列”时，都去重新遍历所有子序列，而不是直接记录这个子问题的结果。

```
1 n = int(input())
2 *b, = map(int, input().split())
3 dp = [1]*n
4
5 for i in range(1, n):
6     for j in range(i):
7         if b[j] < b[i]:
8             dp[i] = max(dp[i], dp[j]+1)
9
10 print(max(dp))
```

bisect 用法，Maintain lists in sorted order, <https://pymotw.com/2/bisect/>

```
1 import bisect
2 n = int(input())
3 *lis, = map(int, input().split())
4 dp = [1e9]*n
5 for i in lis:
6     dp[bisect.bisect_left(dp, i)] = i
7 print(bisect.bisect_left(dp, 1e8))
```

Bisect_left 返回的位置，如果不是升序，值会被覆盖

pythontutor.com/visualize.html#mode=display

p in the [Python Discord](#) chat

Python 3.6
(known limitations)

```

1 import bisect
2 n = int(input())
3 *lis, = map(int, input().split())
4 dp = [1e9]*n
5 for i in lis:
6     dp[bisect.bisect_left(dp, i)] = i
7 print(bisect.bisect_left(dp, 1e8))

```

[Edit this code](#)

it just executed
ie to execute

<< First < Prev Next > >>

Done running (20 steps)

Print output (drag lower right corner to resize)

7
1 7 3 5 9 4 8
4

Frames Objects

Global frame

bisect	module instance
n	7
lis	list [0 1 2 3 4 5 6]
dp	list [0 1 3 4 8 1000000000.0 5 1000000000.0 6 1000000000.0]
i	8

示例03532: 最大上升子序列和

dp, <http://cs101.openjudge.cn/practice/03532>

一个数的序列 b_i , 当 $b_1 < b_2 < \dots < b_s$ 的时候, 我们称这个序列是上升的。对于给定的一个序列 (a_1, a_2, \dots, a_N) , 我们可以得到一些上升的子序列 $(a_{i_1}, a_{i_2}, \dots, a_{i_K})$, 这里 $1 \leq i_1 < i_2 < \dots < i_K \leq N$ 。比如, 对于序列 $(1, 7, 3, 5, 9, 4, 8)$, 有它的一些上升子序列, 如 $(1, 7), (3, 4, 8)$ 等等。这些子序列中序列和最大为18, 为子序列 $(1, 3, 5, 9)$ 的和。

你的任务, 就是对于给定的序列, 求出最大上升子序列和。**注意, 最长的上升子序列的和不一定是最大的, 比如序列 $(100, 1, 2, 3)$ 的最大上升子序列和为100, 而最长上升子序列为 $(1, 2, 3)$ 。**

输入

输入的第一行是序列的长度N ($1 \leq N \leq 1000$)。第二行给出序列中的N个整数, 这些整数的取值范围都在0到10000 (可能重复)。

输出

最大上升子序列和

样例输入

```

1 | 7
2 | 1 7 3 5 9 4 8

```

样例输出

```

1 | 18

```

思路：从第一个数开始逐次递推，考虑第 i 个数的情况时，再从第一个数开始逐个检验，如果第 i 个数大于前 i 个数中的第 j 个数，那么将前 j 个数的最大上升子序列和再加上第 i 个数，即构成前 i 个数上升子序列和的一种情况，再取这些情况中的最大值，即得到前 i 个数的最大上升子序列和。最后依次递推，即可得到整个序列的最大上升子序列和。

主要思路就是记录把每个数作为序列最后一位时的序列和，取 max。即，以每一项为末项的最大上升子序列和。

2020fall-cs101，邹思清。感觉跟我之前做的dp不太一样。之前的dp大多是计算n位之前满足的答案，而这道题使用的递推公式也不仅仅是相邻几项，而且还使用了max，做的时候没有想到，又学到新方法了。

```
1 input()
2 a = [int(x) for x in input().split()]
3
4 n = len(a)
5 dp = [0]*n
6
7 for i in range(n):
8     dp[i] = a[i]
9     for j in range(i):
10         if a[j] < a[i]:
11             dp[i] = max(dp[j]+a[i], dp[i])
12
13 print(max(dp))
```

5 背包DP

这是最经典的DP问题，会衍生出各种不同的变式。我们可以掌握0-1背包，完全背包，和“恰好型”。

详见<https://oi-wiki.org/dp/knapsack/>，其中没有“恰好型”。

5.1 0-1背包（每个物品选或者不选）

示例23421: 小偷背包

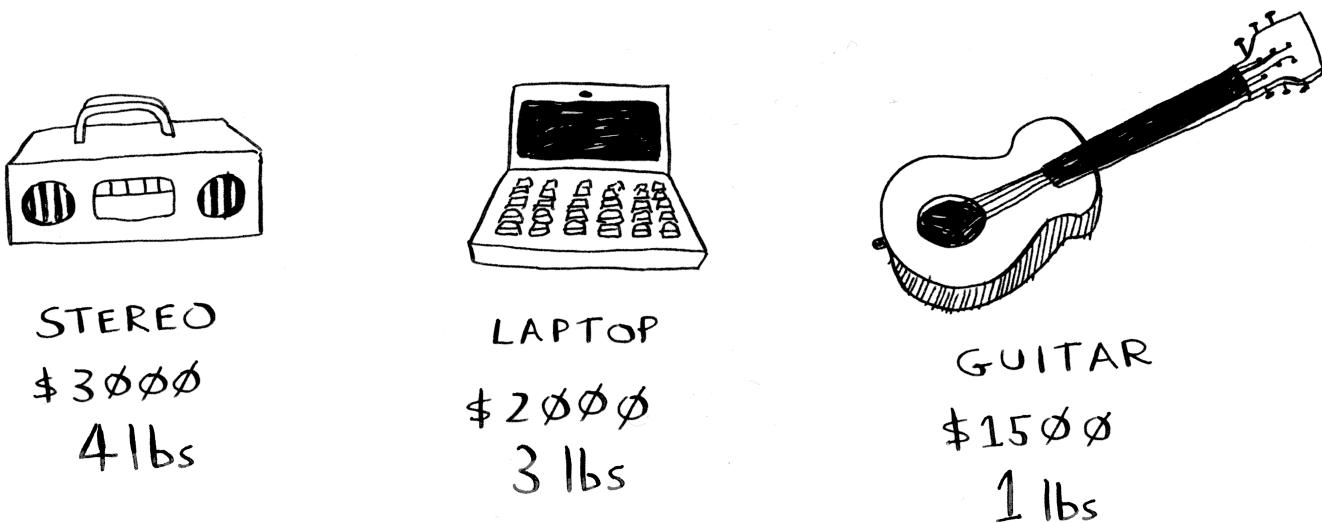
dp, <http://cs101.openjudge.cn/practice/23421>

这是《算法图解》[1]书中第9章动态规划的例子：一个小贼正在一家店里偷商品。

假设一种情况如下：

一个小偷背着一个可装 4 磅东西的背包。商场有三件物品分别为：

价值 3000 美元重 4 磅的音响，价值 2000 美元重 3 磅的笔记本，价值 1500 美元重 1 磅的吉他。



问小偷应该怎样选择商品，才能使得偷取的价值最高？

[1]Grokking Algorithms by Aditya Bhargava, published by Manning Publications. Copyright © 2016 by Manning Publications.

Simplified Chinese-language edition copyright © 2017 by Posts & Telecom Press.

输入

第一行是两个整数N和B，空格分隔。N表示物品件数，B表示背包最大承重。

第二行是N个整数，空格分隔。表示各个物品价格。

第三行是N个整数，空格分隔。表示各个物品重量（是与第二行物品对齐的）。

输出

输出一个整数。保证在满足背包容量的情况下，偷的价值最高。

样例输入

1	3 4
2	3000 2000 1500
3	4 3 1

样例输出

1	3500
---	------

基本思路

最简单的算法如下：尝试各种可能的商品组合，并找出价值最高的组合。这样可行，但速度非常慢。在有3件商品的情况下，你需要计算8个不同的集合；有4件商品时，你需要计算16个集合。每增加一件商品，需要计算的集合数都将翻倍！这种算法的运行时间为 $O(2^n)$ ，真的是慢如蜗牛。

答案是使用动态规划！下面来看看动态规划算法的工作原理。动态规划先解决子问题，再逐步解决大问题。对于背包问题，你先解决小背包（子背包）问题，再逐步解决原来的问题。

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

每个动态规划算法都从一个网格开始，背包问题的网格如下。Every dynamic-programming algorithm starts with a grid. Here's a grid for the knapsack problem.

COLUMNS ARE KNAPSACK
SIZES FROM 1 TO 4 POUNDS

ONE ROW
FOR EACH
ITEM TO
CHOOSE
FROM

GUITAR
STEREO
LAPTOP

1	2	3	4

The rows of the grid are the items, and the columns are knapsack weights from 1 lb to 4 lb. You need all of those columns because they will help you calculate the values of the sub-knapsacks.

网格的各行为商品，各列为不同容量（1~4磅）的背包。所有这些列你都需要，因为它们将帮助你计算子背包的价值。

The grid starts out empty. You're going to fill in each cell of the grid. Once the grid is filled in, you'll have your answer to this problem! Please follow along. Make your own grid, and we'll fill it out together.

网格最初是空的。你将填充其中的每个单元格，网格填满后，就找到了问题的答案！你一定要跟着做。请你创建网格，我们一起来填满它。

THE GUITAR ROW

I'll show you the exact formula for calculating this grid later. Let's do a walkthrough first. Start with the first row.

后面将列出计算这个网格中单元格值的公式。我们先来一步一步做。首先来看第一行。

	1	2	3	4
GUITAR				
STEREO				
LAPTOP				

This is the guitar row, which means you're trying to fit the guitar into the knapsack. At each cell, there's a simple decision: do you steal the guitar or not? Remember, you're trying to find the set of items to steal that will give you the most value.

这是吉他行， 意味着你将尝试将吉他装入背包。在每个单元格， 都需要做一个简单的决定：偷不偷吉他？别忘了， 你要找出一个价值最高的商品集合。

该不该偷音响呢？

背包的容量为1磅， 能装下音响吗？ 音响太重了， 装不下！ 由于容量1磅的背包装不下音响， 因此最大价值依然是1500美元。

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G			
LAPTOP				

Same thing for the next two cells. These knapsacks have a capacity of 2 lb and 3 lb. The old max value for both was \$1,500.

接下来的两个单元格的情况与此相同。在这些单元格中，背包的容量分别为2磅和3磅，而以前的最大价值为1500美元。

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	
LAPTOP				

The stereo still doesn't fit, so your guesses remain unchanged. What if you have a knapsack of capacity 4 lb? Aha: the stereo finally fits! The old max value was \$1,500, but if you put the stereo in there instead, the value is \$3,000! Let's take the stereo.

由于这些背包装不下音响，因此最大价值保持不变。

背包容量为4磅呢？终于能够装下音响了！原来的最大价值为1500美元，但如果在背包中装入音响而不是吉他，价值将为3000美元！因此还是偷音响吧。

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP				

You just updated your estimate! If you have a 4 lb

你更新了最大价值！如果背包的容量为4磅，就能装入价值至少3000美元的商品。在这个网格中，你逐步地更新最大价值。

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G	\$2000 L	

At 3 lb, the old estimate was \$1,500. But you can choose the laptop instead, and that's worth \$2,000. So the new max estimate is \$2,000!

对于容量为4磅的背包，情况很有趣。这是非常重要的部分。当前的最大价值为3000美元，你可不偷音响，而偷笔记本电脑，但它只值2000美元。价值没有原来高。但等一等，笔记本电脑的重量只有3磅，背包还有1磅的容量没用！

根据之前计算的最大价值可知，在1磅的容量中可装入吉他，价值1500美元。因此，你需要做如下比较。

$$\$3000 \text{ vs } (\$2000 + \$1500)$$

音响 笔记本电脑 吉他

你可能始终心存疑惑：为何计算小背包可装入的商品的最大价值呢？但愿你现在明白了其中的原因！余下了空间时，你可根据这些子问题的答案来确定余下的空间可装入哪些商品。笔记本电脑和吉他的总价值为3500美元，因此偷它们是更好的选择。最终的网格类似于下面这样。

	1	2	3	4
GUITAR	\$1500 ↓ G	\$1500 ↓ G	\$1500 ↓ G	\$1500 G
STEREO	\$1500	\$1500	\$1500	\$3000
LAPTOP	\$1500 ↓ G	\$1500 ↓ G	\$2000 G	\$3500 L G
				↑ THE ANSWER!

There's the answer: the maximum value that will fit in the knapsack is \$3,500, made up of a guitar and a laptop! Maybe you think that I used a different formula to calculate the value of that last cell. That's because I skipped some unnecessary complexity when filling in the values of the earlier cells. Each cell's value gets calculated with the same formula. Here it is.

答案如下：将吉他和笔记本电脑装入背包时价值最高，为3500美元。你可能认为，计算最后一个单元格的价值时，我使用了不同的公式。那是因为填充之前的单元格时，我故意避开了一些复杂的因素。其实，计算每个单元格的价值时，使用的公式都相同。这个公式如下。

$$\text{CELL}[i][j] = \max \text{ of } \left\{ \begin{array}{l} 1. \text{ THE PREVIOUS MAX (VALUE AT CELL } [i-1][j] \text{)} \\ \quad \quad \quad \text{vs} \\ 2. \text{ VALUE OF CURRENT ITEM + VALUE OF THE REMAINING SPACE} \\ \quad \quad \quad \uparrow \\ \text{CELL}[i-1][j - \text{ITEM'S WEIGHT}] \end{array} \right.$$

用子问题定义状态：即 $\text{CELL}[i][j]$ 表示前 i 件物品恰放入一个容量为 j 的背包可以获得的最大价值。则其状态转移方程便是：

$$\text{CELL}[i][j] = \max(\text{CELL}[i-1][j]; \text{CELL}[i-1][j - W_i] + V_i)$$

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前 i 件物品放入容量为 j 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只和前 $i - 1$ 件物品相关的问题。如果不放第 i 件物品，那么问题就转化为“前 $i - 1$ 件物品放入容量为 j 的背包中”，价值为 $\text{CELL}[i-1][j]$ ；如果放第 i 件物品，那么问题就转化为“前 $i - 1$ 件物品放入剩下的容量为 $j - W_i$ 的背包中”，此时能获得的最大价值就是 $\text{CELL}[i-1][j - W_i]$ 再加上通过放入第 i 件物品获得的价值 V_i 。

You can use this formula with every cell in this grid, and you should end up with the same grid I did. Remember how I talked about solving subproblems? You combined the solutions to two subproblems to solve the bigger problem.

你可以使用这个公式来计算每个单元格的价值，最终的网格将与前一个网格相同。现在你明白了为何要求解子问题吧？你可以合并两个子问题的解来得到更大问题的解。



```

1 # 动态规划之背包问题 (算法图解书中例子实现)
2
3 #第一步建立网格(横坐标表示[0,c]整数背包承重):(n+1)*(c+1)
4 def knapsack(n, c, w, p):
5     cell = [[0 for j in range(c+1)] for i in range(n+1)]
6     for j in range(c+1):
7         #第0行全部赋值为0, 物品编号从1开始.为了下面赋值方便
8         cell[0][j] = 0
9     for i in range(1, n+1):
10        for j in range(1, c+1):
11            #生成了n*c有效矩阵, 以下公式w[i-1],p[i-1]代表从第一个元素w[0],p[0]开始取。
12            if j >= w[i-1]:
13                cell[i][j] = max(cell[i-1][j], p[i-1] + cell[i-1][j - w[i-1]])
14            else:
15                cell[i][j] = cell[i-1][j]
16    return cell
17
18
19 goodsnum, bagsize = map(int, input().split())
20 #goodsnum, bagsize = 3, 4
21 *value, = map(int, input().split())
22 *weight, = map(int, input().split())
23 #value, weight = [1500, 3000, 2000], [1, 4, 3] # guitar, stereo, laptop
24
25 cell = knapsack(goodsnum, bagsize, weight, value)
26 print(cell[goodsnum][bagsize])

```

如果价格、重量的第一个元素从1开始

```

1 n,b=map(int, input().split())
2 price=[0]+[int(i) for i in input().split()]
3 weight=[0]+[int(i) for i in input().split()]
4 bag=[[0]*(b+1) for _ in range(n+1)]
5 for i in range(1,n+1):
6     for j in range(1,b+1):
7         if weight[i]<=j:
8             bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
9         else:
10            bag[i][j]=bag[i-1][j]
11 print(bag[-1][-1])

```

滚动数组优化空间复杂度

背包九讲.pdf, 及<https://oi-wiki.org/dp/knapsack/>

例题中已知条件有第 i 个物品的重量 w_i , 价值 v_i , 以及背包的总容量 W 。

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$ ；当其放入背包时，背包的剩余容量会减小 w_i ，背包中物品的总价值会增大 v_i ，故这种情况的最大价值为 $f_{i-1,j-w_i} + v_i$ 。

由此可以得出状态转移方程：

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对 i 有影响的只有 $i-1$ ，可以去掉第一维，直接用 f_j 来表示处理到当前物品时背包容量为 j 的最大价值，得出以下方程：

$$f_j = \max(f_j, f_{j-w_i} + v_i)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

实现

还有一点需要注意的是，很容易写出这样的 错误核心代码：

```
1 for i in range(1, n + 1):
2     for l in range(0, w - w[i] + 1):
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
4 # 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
5 # f[i][l + w[i]]) 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$ ，在 $j \geq w_i$ 时，是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i ，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新。

因此实际核心代码为

```
1 for i in range(1, n + 1):
2     for l in range(W, w[i] - 1, -1):
3         f[l] = max(f[l], f[l - w[i]] + v[i])
```

优化23421: 小偷背包

从 大到小更新，我们总是基于“之前未包含当前物品的最优解”来更新新的状态，因此能保证每个物品在每次主循环中只会被计算一次。

```

1 # 压缩矩阵/滚动数组 方法
2 N,B = map(int, input().split())
3 *p, = map(int, input().split())
4 *w, = map(int, input().split())
5
6 dp=[0]*(B+1)
7 for i in range(N):
8     for j in range(B, w[i] - 1, -1):
9         dp[j] = max(dp[j], dp[j-w[i]]+p[i])
10
11 print(dp[-1])

```

5.2 完全背包（每种物品可以选0个-无限个）

将0-1背包中内层循环改为正着遍历即可（这里其实就利用了先前已经得到的信息来简化转移：在先前的转移中物品i可能已经用过若干次了）

练习01384: Piggy-Bank

<http://cs101.openjudge.cn/practice/01384/>

Before ACM can do anything, a budget must be prepared and the necessary financial support obtained. The main income for this action comes from Irreversibly Bound Money (IBM). The idea behind is simple.

Whenever some ACM member has any small money, he takes all the coins and throws them into a piggy-bank. You know that this process is irreversible, the coins cannot be removed without breaking the pig.

After a sufficiently long time, there should be enough cash in the piggy-bank to pay everything that needs to be paid.

But there is a big problem with piggy-banks. It is not possible to determine how much money is inside. So we might break the pig into pieces only to find out that there is not enough money. Clearly, we want to avoid this unpleasant situation. The only possibility is to weigh the piggy-bank and try to guess how many coins are inside. Assume that we are able to determine the weight of the pig exactly and that we know the weights of all coins of a given currency. Then there is some minimum amount of money in the piggy-bank that we can guarantee. Your task is to find out this worst case and determine the minimum amount of cash inside the piggy-bank. We need your help. No more prematurely broken pigs!

5.3 多重背包（每个物品数量有上限）

最简单的思路是将多个同样的物品看成多个不同的物品，从而化为0-1背包。稍作优化：可以改善拆分方式，譬如将m个1拆成x_1,x_2,...,x_t个1，只需要这些x_i中取若干个的和能组合出1至m即可。最高效的拆分方式是尽可能拆成2的幂，也就是所谓“二进制优化”

练习01742: Coins

dp, <http://cs101.openjudge.cn/routine/01742/>

People in Silverland use coins. They have coins of value A1,A2,A3...An Silverland dollar. One day Tony opened his money-box and found there were some coins. He decided to buy a very nice watch in a nearby shop. He wanted to pay the exact price (without change) and he known the price would not more than m. But he didn't know the exact price of the watch.

You are to write a program which reads n,m,A1,A2,A3...An and C1,C2,C3...Cn corresponding to the number of Tony's coins of value A1,A2,A3...An then calculate how many prices (from 1 to m) Tony can pay use these coins.

5.4 “恰好”型暨最优解

示例189A. Cut Ribbon

brute force/dp, 1300, <https://codeforces.com/problemset/problem/189/A>

Polycarpus has a ribbon, its length is n . He wants to cut the ribbon in a way that fulfills the following two conditions:

- After the cutting each ribbon piece should have length a , b or c .
- After the cutting the number of ribbon pieces should be maximum.

Help Polycarpus and find the number of ribbon pieces after the required cutting.

Input

The first line contains four space-separated integers n , a , b and c ($1 \leq n, a, b, c \leq 4000$) — the length of the original ribbon and the acceptable lengths of the ribbon pieces after the cutting, correspondingly. The numbers a , b and c can coincide.

Output

Print a single number — the maximum possible number of ribbon pieces. It is guaranteed that at least one correct ribbon cutting exists.

思路：就是一个需要刚好装满的完全背包问题，只有三种商品a, b, c, 能取无限件物品，每件物品价值是1，求最大价值。

```

1 n, a, b, c = map(int, input().split())
2 dp = [0]+[float('-inf')]*n
3
4 for i in range(1, n+1):
5     for j in (a, b, c):
6         if i >= j:
7             dp[i] = max(dp[i-j] + 1, dp[i])
8
9 print(dp[n])

```

练习21458: 健身房 (dp)

dp, <http://cs101.openjudge.cn/routine/21458/>

小嘎是大不列颠及北爱尔兰联合王国大力士，为了完成增肌计划，他需要选择一些训练组进行训练：有n个训练组，每天做第i个训练需要耗费 t_i 分钟，每天坚持做第i个训练一个月后预计可增肌 w_i 千克。因为会导致效果变差，小嘎一天不会做相同的训练组多次。由于小嘎是强迫症，他希望每天用于健身的时间恰好为T分钟，他希望在一个月后获得最多的增肌量，请帮助小嘎计算：他训练一个月后最大增肌量是多少呢？

输入

第一行两个整数 T,n。

第 2 行到第 $n+1$ 行，每行两个整数 t_i, w_i 。

保证 $0 < t_i \leq T \leq 10^3$, $0 < n \leq 10^3$, $0 < w_i < 20$ 。

输出

如果不存在满足条件的训练计划，输出-1。

如果存在满足条件的训练计划，输出一个整数，表示训练一个月后的最大增肌量。

样例输入

```

1 sample1 in
2 6 4
3 2 1
4 4 7
5 3 5
6 3 5
7
8 sample1 out
9 10

```

样例输出

```

1 sample2 in
2 700 4
3 450 5
4 340 1
5 690 10
6 9 2
7
8 sample2 out
9 -1
10 样例2解释：无法找出一种方案满足训练时间恰好等于T。

```

来源：cs101 2020 Final Exam

"恰好"型dp。类似方法：最开始的设为0，其余的都为设为负无穷。。。 https://zhuanlan.zhihu.com/p/560690993?utm_id=0

```

1 # 23n2300011031, 黄源森
2 t,n=map(int,input().split())
3 dp=[0]+[-1]*(t+1)
4 for i in range(n):
5     k,w=map(int,input().split())
6     for j in range(t,k-1,-1):
7         if dp[j-k]!=-1:
8             dp[j]=max(dp[j-k]+w,dp[j])
9 print(dp[t])

```

练习20089: NBA门票

dp, <http://cs101.openjudge.cn/practice/20089/>

六月，巨佬甲正在加州进行暑研工作。恰逢湖人和某东部球队进NBA总决赛的对决。而同为球迷的老板大发慈悲给了甲若干美元的经费，让甲同学用于购买球票。然而由于球市火爆，球票数量也有限。共有七种档次的球票（对应价格分别为50 100 250 500 1000 2500 5000美元）而同学甲购票时这七种票也还分别剩余 ($n_1, n_2, n_3, n_4, n_5, n_6, n_7$ 张)。现由于甲同学与同伴关系恶劣。而老板又要求甲同学必须将所有经费恰好花完，请给出同学甲可买的最少的球票数 X 。

输入

第一行老板所发的经费 N ,其中 $50 \leq N \leq 1000000$ 。

第二行输入 n_1-n_7 ，分别为七种票的剩余量，用空格隔开

输出

假若余票不足或者有余额，则输出'Fail'

而假定能刚好花完，则输出同学甲所购买的最少的票数 X 。

样例输入

```
1 Sample1 Input:  
2 5500  
3 3 3 3 3 3 3  
4  
5 Sample1 Output:  
6 2
```

样例输出

```
1 Sample2 Input:  
2 125050  
3 1 2 3 1 2 5 20  
4  
5 Sample2 Output:  
6 Fail
```

来源: cs101-2019 龚世棋

```
1 # 2200015481, 陈涛  
2 n=int(input())  
3 tickets=list(map(int,input().split()))  
4 price=[50,100,250,500,1000,2500,5000]  
5 dp={0:0}  
6 path={0:[0,0,0,0,0,0,0]}  
7 for i in range(n):  
8     if i in dp:  
9         for k in range(7):  
10            if path[i][k]<tickets[k]:  
11                if i+price[k] in dp:  
12                    if dp[i]+1<dp[i+price[k]]:  
13                        dp[i+price[k]]=dp[i]+1  
14                        path[i+price[k]]=path[i][:]  
15                        path[i+price[k]][k]+=1  
16                else:  
17                    dp[i+price[k]]=dp[i]+1  
18                    path[i+price[k]]=path[i][:]  
19                    path[i+price[k]][k]+=1  
20        if n in dp:  
21            print(dp[n])  
22        else:  
23            print('Fail')
```

6 最长公共子串 Longest common substring

《算法图解》9.3 最长公共子串

通过前面的动态规划问题，你得到了哪些启示呢？

- 动态规划可帮助你在给定约束条件下找到最优解。在背包问题中，你必须在背包容量给定的情况下，偷到价值最高的商品。
- 在问题可分解为彼此独立且离散的子问题时，就可使用动态规划来解决。

要设计出动态规划解决方案可能很难，这正是本节要介绍的。下面是一些通用的小贴士。

- 每种动态规划解决方案都涉及网格。
- 单元格中的值通常就是你要优化的值。在前面的背包问题中，单元格的值为商品的价值。
- 每个单元格都是一个子问题，因此你应考虑如何将问题分成子问题，这有助于你找出网格的坐标轴。

下面再来看一个例子。假设你管理着网站dictionary.com。用户在该网站输入单词时，你需要给出其定义。

但如果用户拼错了，你必须猜测他原本要输入的是什么单词。例如，Alex想查单词fish，但不小心输入了hish。在你的字典中，根本就没有这样的单词，但有几个类似的单词。

在这个例子中，只有两个类似的单词，真是太小儿科了。实际上，类似的单词很可能有数千个。Alex输入了hish，那他原本要输入的是fish还是vista呢？

6.1 基本思路

1 绘制网格

用于解决这个问题的网格是什么样的呢？要确定这一点，你得回答如下问题。

单元格中的值是什么？

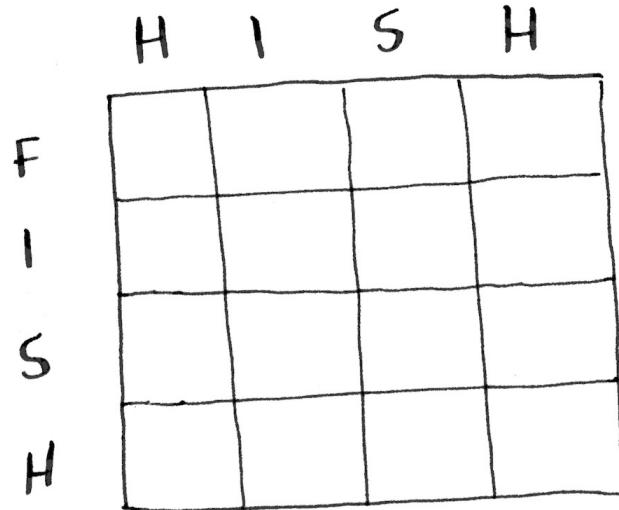
如何将这个问题划分为子问题？

网格的坐标轴是什么？

在动态规划中，你要将某个指标最大化。在这个例子中，你要找出两个单词的最长公共子串。hish和fish都包含的最长子串是什么呢？hish和vista呢？这就是你要计算的值。

别忘了，单元格中的值通常就是你要优化的值。在这个例子中，这很可能是一个数字：两个字符串都包含的最长子串的长度。

如何将这个问题划分为子问题呢？你可能需要比较子串：不是比较hish和fish，而是先比较his和fis。每个单元格都将包含这两个子串的最长公共子串的长度。这也给你提供了线索，让你觉得坐标轴很可能是这两个单词。因此，网格可能类似于下面这样。

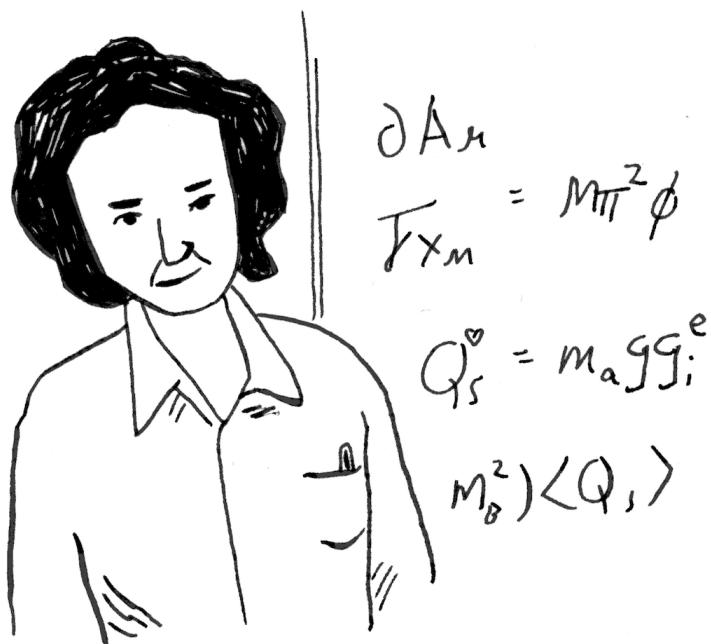


2 填充网格

现在，你很清楚网格应是什么样的。填充该网格的每个单元格时，该使用什么样的公式呢？由于你已经知道答案——hish和fish的最长公共子串为ish，因此可以作弊。

即便如此，你还是不能确定该使用什么样的公式。计算机科学家有时会开玩笑说，那就使用费曼算法（Feynman algorithm）。这个算法是以著名物理学家理查德·费曼命名的，其步骤如下。

- (1) 将问题写下来。
- (2) 好好思考。
- (3) 将答案写下来。



计算机科学家真是一群不按常理出牌的人啊！实际上，根本没有找出计算公式的简单办法，你必须通过尝试才能找出管用的公式。有些算法并非精确的解决步骤，而只是帮助你理清思路的框架。

请尝试为这个问题找到计算单元格值的公式。给你一点提示吧：下面是这个单元格的一部分。

	H	I	S	H
F	0	0		
I				
S			2	0
H				3

其他单元格的值呢？别忘了，每个单元格都是一个子问题的值。为何单元格(3, 3)的值为2呢？又为何单元格(3, 4)的值为0呢？

请找出计算公式，再接着往下读。这样即便你没能找出正确的公式，后面的解释也将容易理解得多。

3 揭晓答案

最终的网格如下。

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

我使用下面的公式来计算每个单元格的值。

1. 如果两个字母不相同，值为0

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

2. 如果两个字母相同，值
为左上角邻居的值加1

实现这个公式的伪代码类似于下面这样。

```
if word_a[i] == word_b[j]:    -----两个字母相同
    cell[i][j] = cell[i-1][j-1] + 1
else:   -----两个字母不同
    cell[i][j] = 0
```

6.2 最长公共子序列 Longest common subsequence

假设Alex不小心输入了fosh，他原本想输入的是fish还是fort呢？

我们使用最长公共子串公式来比较它们。

	F	O	S	H
F	I	O	O	O
O	O	2	O	O
R	O	O	O	O
T	O	O	O	O

vs

	F	O	S	H
F	I	O	O	O
I	O	O	O	O
S	O	O	I	O
H	O	O	O	2

最长公共子串的长度相同，都包含两个字母！但fosh与fish更像。

F O S H
 ↓ ↓ ↓ = 3
 F I S H

F O S H
 ↓ ↓ = 2
 F O R T

这里比较的是最长公共子串，但其实应比较最长公共子序列：两个单词中都有的序列包含的字母数。如何计算最长公共子序列呢？

下面是用于计算fish和fosh的最长公共子序列的网格的一部分。

	F	O	S	H
F	I	I		
I	I			
S		I	2	2
H				

你能找出填充这个网格时使用的公式吗？最长公共子序列与最长公共子串很像，计算公式也很像。请试着找出这个公式——答案稍后揭晓。

最长公共子序列之解决方案

最终的网格如下。

	F	O	S	H
F	1 → 1 → 1 → 1			
O	↓ 1 → 2 → 2 → 2			
R	↓ 1 → 2 → 2 → 2			
T	↓ 1 → 2 → 2 → 2			

LONGEST COMMON SUBSEQUENCE = 2

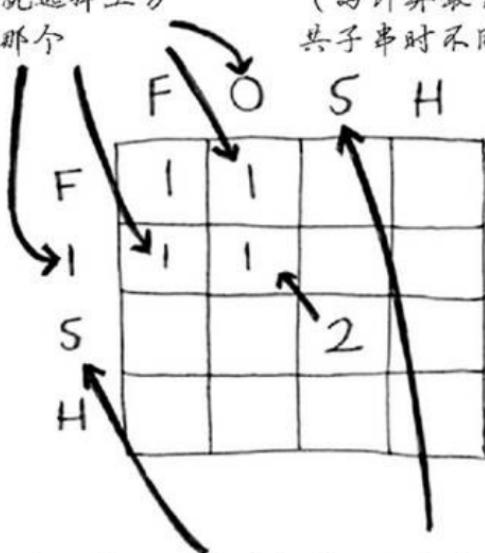
	F	O	S	H
F	1 → 1 → 1 → 1			
I	↓ 1 → 1 → 1 → 1			
S	↓ 1 → 1 → 2 → 2			
H	↓ 1 → 1 → 2 → 3			

LONGEST COMMON SUBSEQUENCE = 3

下面是填写各个单元格时使用的公式。

1. 如果两个字母不同，就选择上方和左方邻居中较大的那个

(与计算最长公共子串时不同)



2. 如果两个字母相同，就将当前单元格的值设置为左上方单元格的值加1 (与计算最长公共子串时类似)

伪代码如下。

```

if word_a[i] == word_b[j]:           ←-----两个字母相同
    cell[i][j] = cell[i-1][j-1] + 1
else:                                ←-----两个字母不同
    cell[i][j] = max(cell[i-1][j], cell[i][j-1])

```

本章到这里就结束了！它绝对是本书最难理解的一章。动态规划都有哪些实际应用呢？

- 生物学家根据最长公共序列来确定DNA链的相似性，进而判断两种动物或疾病有多相似。最长公共序列还被用来寻找多发性硬化症治疗方案。
- 你使用过诸如git diff 等命令吗？它们指出两个文件的差异，也是使用动态规划实现的。
- 前面讨论了字符串的相似程度。编辑距离（Levenshtein distance）指出了两个字符串的相似程度，也是使用动态规划计算得到的。编辑距离算法的用途很多，从拼写检查到判断用户上传的资料是否是盗版，都在其中。
- 你使用过诸如Microsoft Word等具有断字功能的应用程序吗？它们如何确定在什么地方断字以确保行长一致呢？使用动态规划！

示例OJ02806:公共子序列

<http://cs101.openjudge.cn/practice/02806/>

我们称序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ 是序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ 的子序列当且仅当存在严格上升的序列 $\langle i_1, i_2, \dots, i_k \rangle$ ，使得对 $j = 1, 2, \dots, k$ ，有 $x_{i_j} = z_j$ 。比如 $Z = \langle a, b, f, c \rangle$ 是 $X = \langle a, b, c, f, b, c \rangle$ 的子序列。

现在给出两个序列X和Y，你的任务是找到X和Y的最大公共子序列，也就是说要找到一个最长的序列Z，使得Z既是X的子序列也是Y的子序列。

输入

输入包括多组测试数据。每组数据包括一行，给出两个长度不超过200的字符串，表示两个序列。两个字符串之间由若干个空格隔开。

输出

对每组输入数据，输出一行，给出两个序列的最大公共子序列的长度。

样例输入

1	abcfbc	abfcab
2	programming	contest
3	abcd	mnp

样例输出

1	4
2	2
3	0

来源：翻译自Southeastern Europe 2003的试题

这题目输入没有明确结束，需要套在try ... except里面。测试时候，需要模拟输入结束，看你是window还是mac。If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.

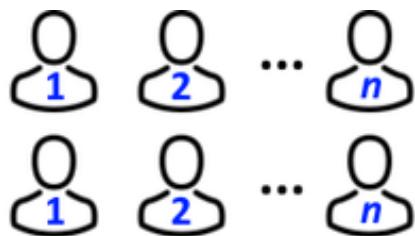
```
1 while True:
2     try:
3         a, b = input().split()
4     except EOFError:
5         break
6
7     aLEN = len(a)
8     bLEN = len(b)
9
10    dp = [[0] * (bLEN + 1) for i in range(aLEN + 1)]
11
12    for i in range(1, aLEN + 1):
13        for j in range(1, bLEN + 1):
14            if a[i - 1] == b[j - 1]:
15                dp[i][j] = dp[i - 1][j - 1] + 1
16            else:
17                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
18
19
20    print(dp[aLEN][bLEN])
```

7 定义多个dp数组

示例1195C. Basketball Exercise

dp, 1400, <https://codeforces.com/problemset/problem/1195/C>

Finally, a basketball court has been opened in SIS, so Demid has decided to hold a basketball exercise session. $2 \cdot n$ students have come to Demid's exercise session, and he lined up them into two rows of the same size (there are exactly n people in each row). Students are numbered from 1 to n in each row in order from left to right.



Now Demid wants to choose a team to play basketball. He will choose players from left to right, and the index of each chosen player (excluding the first one **taken**) will be strictly greater than the index of the previously chosen player. To avoid giving preference to one of the rows, Demid chooses students in such a way that no consecutive chosen students belong to the same row. The first student can be chosen among all $2n$ students (there are no additional constraints), and a team can consist of any number of students.

Demid thinks, that in order to compose a perfect team, he should choose students in such a way, that the total height of all chosen students is maximum possible. Help Demid to find the maximum possible total height of players in a team he can choose.

Input

The first line of the input contains a single integer n ($1 \leq n \leq 10^5$) — the number of students in each row.

The second line of the input contains n integers $h_{1,1}, h_{1,2}, \dots, h_{1,n}$ ($1 \leq h_{1,i} \leq 10^9$), where $h_{1,i}$ is the height of the i -th student in the first row.

The third line of the input contains n integers $h_{2,1}, h_{2,2}, \dots, h_{2,n}$ ($1 \leq h_{2,i} \leq 10^9$), where $h_{2,i}$ is the height of the i -th student in the second row.

Output

Print a single integer — the maximum possible total height of players in a team Demid can choose.

Examples

input

1	5
2	9 3 5 7 3
3	5 8 1 4 5

output

input

1	3
2	1 2 9
3	10 1 1

output

1	19
---	----

input

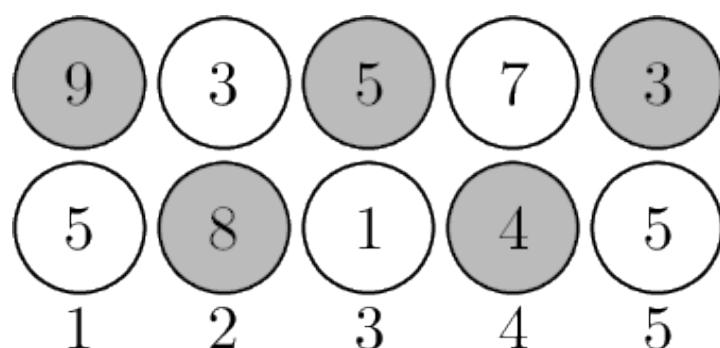
1	1
2	7
3	4

output

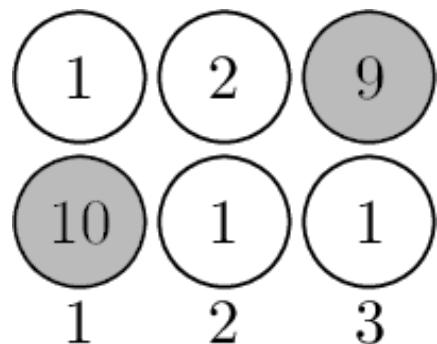
1	7
---	---

Note

In the first example Demid can choose the following team as follows:



In the second example Demid can choose the following team as follows:



```

1 n = int(input())
2 h1 = list(map(int, input().split()))
3 h2 = list(map(int, input().split()))
4
5 dp1 = [0] * n
6 dp2 = [0] * n
7
8 dp1[0] = h1[0]
9 dp2[0] = h2[0]
10
11 for i in range(1, n):
12     dp1[i] = max(dp2[i - 1] + h1[i], dp1[i - 1])
13     dp2[i] = max(dp1[i - 1] + h2[i], dp2[i - 1])
14
15 print(max(dp1[-1], dp2[-1]))

```

练习25573: 红蓝玫瑰

dp, greedy, <http://cs101.openjudge.cn/practice/25573/>

“玫瑰的红，容易受伤的梦，握在手中却流失于指缝，又落空”

有n ($n < 500000$)支玫瑰从左到右排成一排，它们的颜色是红色或蓝色，红色玫瑰用R表示，蓝色玫瑰用B表示

作为魔法女巫的你，掌握两种魔法：

魔法1：对一支玫瑰施加颜色反转咒语

魔法2：对从左数前k支玫瑰同时施加颜色反转咒语（每次施法时的k值可以不同）

颜色反转咒语将使红玫瑰变成蓝玫瑰，蓝玫瑰变成红玫瑰

请你求出，最少使用多少次魔法，能使得这一排玫瑰全都变为红玫瑰

输入

一个字符串，由R和B组成

输出

一个整数，最少使用多少次魔法

样例输入

```

1 Sample Input1:
2 RRRRRBR
3
4 Sample Output1:
5 1

```

样例输出

```
1 Sample Input2:  
2 RRRBBBRRRB  
3  
4 Sample Output2:  
5 4  
6  
7 解释: 先使用魔法2令k=12, 得到BBBRRRBBBRRR, 然后使用魔法2令k=9, 得到RRRB  
8 然后使用魔法2令k=6, 得到BBBRRRRRRRR, 然后使用魔法2令k=3, 得到RRRRRRRRRRR。  
9 共使用了4次魔法
```

提示

tags: dp, greedy

来源: 2022fall-cs101, gdr

25573: 红蓝玫瑰, 有点像 蒋子轩23工学院 推荐的CF那两个dp题目: 698A-vacations, 1195C-Basketball Exercise。

2022fall-cs101, 姜鑫。

思路的关键是建了两个一维dp, 一个是前n朵玫瑰全变红, 记为Rn, 一个是前n朵玫瑰全变蓝, 记为Bn。如果n+1朵玫瑰是红色, R(n+1)=Rn,B(n+1)可以通过魔法一由前n朵全是蓝色的玫瑰变来, 也可以通过魔法二由前n朵全是红色的玫瑰变来。所以B(n+1)=min(Rn,Bn)+1。如果n+1朵玫瑰是蓝色就反过来。最后对R1, B1赋个值就可以快乐dp了。

```
1 r=list(input())  
2 n=len(r)  
3 R=[0]*n  
4 B=[0]*n  
5 if r[0]=="R":R[0]=0;B[0]=1  
6 else:R[0]=1;B[0]=0  
7 for i in range(n-1):  
8     if r[i+1]=="R":  
9         R[i+1]=R[i]  
10        B[i+1]=min(R[i],B[i])+1  
11    else:  
12        R[i+1]=min(R[i],B[i])+1  
13        B[i+1]=B[i]  
14 print(R[-1])
```

9 小结

- 需要在给定约束条件下优化某种指标时， 动态规划很有用。
- 问题可分解为离散子问题时， 可使用动态规划来解决。
- 每种动态规划解决方案都涉及网格。
- 单元格中的值通常就是你要优化的值。
- 每个单元格都是一个子问题， 因此你需要考虑如何将问题分解为子问题。
- 没有放之四海皆准的计算动态规划解决方案的公式。

10 More Problems

Top 20 Dynamic Programming Interview Questions

<https://www.geeksforgeeks.org/top-20-dynamic-programming-interview-questions/>

Following are the most important Dynamic Programming problems.

1. [Longest Common Subsequence](#)
2. [Longest Increasing Subsequence](#)
3. [Edit Distance](#)
4. [Minimum Partition](#)
5. [Ways to Cover a Distance](#)
6. [Longest Path In Matrix](#)
7. [Subset Sum Problem](#)
8. [Optimal Strategy for a Game](#)
9. [0-1 Knapsack Problem](#)
10. [Boolean Parenthesization Problem](#)
11. [Shortest Common Supersequence](#)
12. [Matrix Chain Multiplication](#)
13. [Partition problem](#)
14. [Rod Cutting](#)
15. [Coin change problem](#)
16. [Word Break Problem](#)
17. [Maximal Product when Cutting Rope](#)
18. [Dice Throw Problem](#)
19. [Box Stacking](#)
20. [Egg Dropping Puzzle](#)

Last Updated : 22 Jun, 2022

Other Problems:

OJ02773: 采药

dp, <http://cs101.openjudge.cn/practice/02773>

OJ02711: 合唱队形

<http://cs101.openjudge.cn/practice/02711/>

OJ02995: 登山

dp , <http://cs101.openjudge.cn/practice/02995>

OJ9267: 核电站

<http://cs101.openjudge.cn/practice/09267>

选自《挑战程序设计竞赛》第2版 Page 135

OJ2229: Sumsets

<http://cs101.openjudge.cn/routine/02229/>

OJ2385: Apple Catching

<http://bailian.openjudge.cn/practice/2385/>

OJ1742: Coins

<http://bailian.openjudge.cn/practice/1742/>

References:

《算法笔记》，胡凡，曾磊。机械工业出版社，2016年7月。

《算法图解》，[美]Aditya Bhargava。2017年。

《Python数据结构与算法分析》，[美]布拉德利·米勒（Bradley N. Miller）,戴维·拉努。人民邮电出版社，2019年9月。

<https://runestone.academy/ns/books/published/pythonds3/Recursion/DynamicProgramming.html?mode=bowing#lst-change2>

<https://www.geeksforgeeks.org/top-20-dynamic-programming-interview-questions/>

<https://www.geeksforgeeks.org/top-50-dynamic-programming-coding-problems-for-interviews/>

Introduction to Knapsack Problem, its Types and How to solve them

<https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/>

Complexity of Python Operations

<https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt>