

2024/10/29 动态规划

Updated 1505 GMT+8 Oct 24, 2024

2024 fall, Complied by Hongfei Yan

Log:

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

2023/11/21 先讲递归，然后继续这个pdf的动规。

2023/11/10 说明：

提醒：下周二讲dp，3小时有5+个典型题目，对于零基础同学来说，很烧脑。最好提前预习！dp与递归，经常同时出现。

2023/11/9 说明：

本课程最难的部分是DP，下周二开始讲。建议零基础同学提前预习！比 segment tree, binary indexed tree, KMP都难，因为没有定式，只有框架。

下周二的课程很重要，因为是DP相关，ppt, notes我都会准备，但是这两个材料重叠度不大，都需要学习。从经典模型讲起，结合例题。另外一个重要的原因是，递归写法可以实现动态规划，递归也是需要掌握的。

2023/11/8 说明：

零基础同学，如果可能，可以提前预习，下周我们讲DP。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。

ppt: Recursion & Dynamic Programming

题目是：用最少的硬币找零，

1 动态规划的递归写法和递推写法

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==。

1.1 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。

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。**输入** 第1行是测试数据的组数n，后面跟着n行输入。每组测试数据占1行，包括一个正整数 \$a \ (1 \leq a \leq 20)\$ **输出** 输出有n行，每行输出对应一个输入。输出应是一个正整数，为斐波那契数列中第a个数的大小 样例输入

```
4
5
2
19
1
```

样例输出

```
5
1
4181
1
```

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

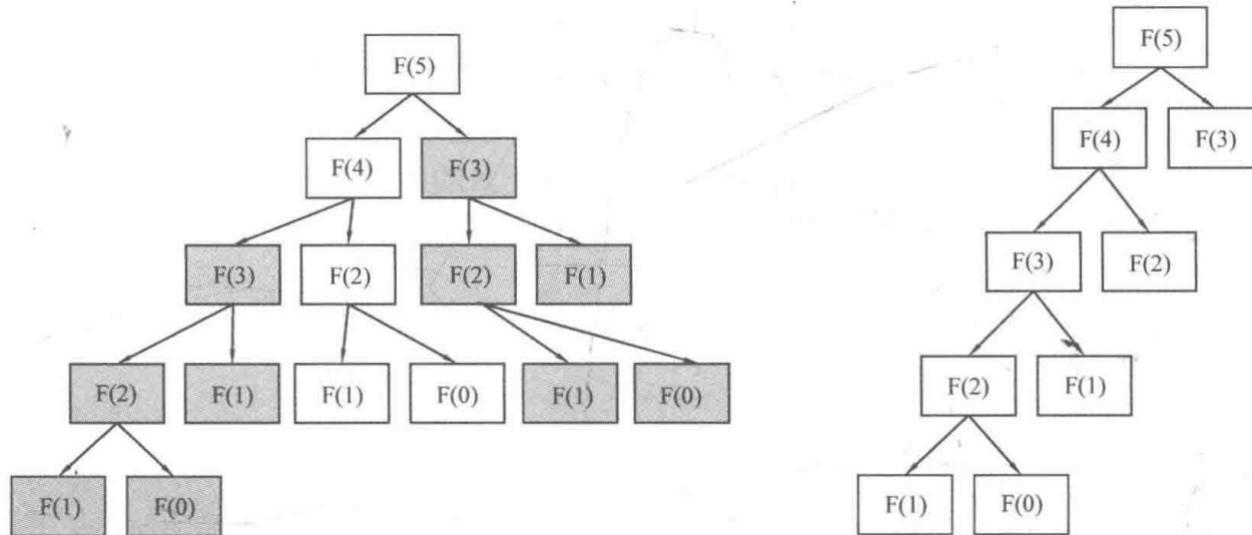
```
def f(n):
    if n <= 2:
        return 1
    else:
        return f(n-1)+f(n-2)

n = int(input())
ans = []
for _ in range(n):
    num = int(input())
    ans.append(f(num))

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数级别降低到了线性级别。

```
def f(n):
    if n <= 2:
        return 1

    if dp[n] != -1:
        return dp[n]
    else:
        dp[n] = f(n-1)+f(n-2)
        return dp[n]

dp = [-1]*21
n = int(input())
ans = []
for _ in range(n):
    num = int(input())
    ans.append(f(num))

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==。

```
...
lru_cache在0J可以用。 Python Functools - lru_cache(),
https://www.geeksforgeeks.org/python-functools-lru-cache/
```

The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in the cache.
<https://www.geeksforgeeks.org/python-lru-cache/>

```
...
from functools import lru_cache

@lru_cache(maxsize = 128)
def f(n):
    if n <= 2:
        return 1
    else:
        return f(n-1)+f(n-2)

n = int(input())
list_1 = []
for i in range(n):
    num = int(input())
    list_1.append(f(num))
for i in list_1:
    print(i)
```

1.2 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？

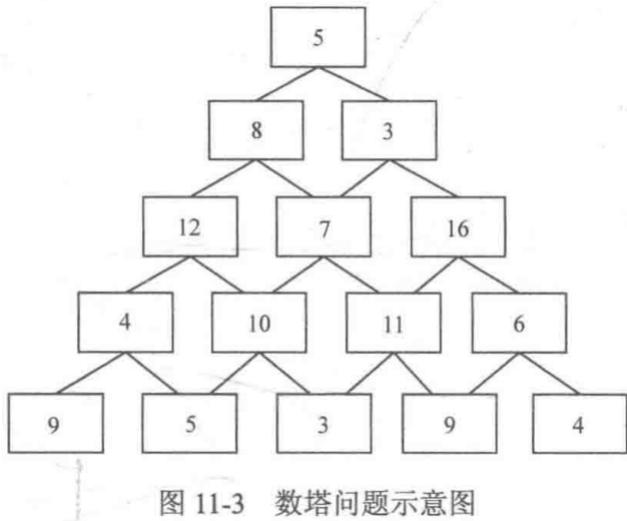


图 11-3 数塔问题示意图

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。

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。

02760: 数字三角形

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

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

(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左边的那个数或者右边的那个数。

输入

输入的是一行是一个整数N ($1 < N \leq 100$)，给出三角形的行数。下面的N行给出数字三角形。数字三角形上的数的范围都在0和100之间。

输出

输出最大的和。

样例输入

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

样例输出

```
30
```

来源：翻译自 IOI 1994 的试题

只递归，不用dp，立马超时。

```

def f(i, j):                                # Time Limit Exceeded, 9953ms
    if i == N-1:
        return tri[i][j]

    return max(f(i+1, j), f(i+1, j+1)) + tri[i][j]

N = int(input())
tri = []
for _ in range(N):
    tri.append([int(i) for i in input().split()])
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。

```

from functools import lru_cache

@lru_cache(maxsize = 128)
def f(i, j):
    if i == N-1:
        return tri[i][j]

    return max(f(i+1, j), f(i+1, j+1)) + tri[i][j]

N = int(input())
tri = []
for _ in range(N):
    tri.append([int(i) for i in input().split()])
print(f(0, 0))

```

递推写法实现的动态规划。

```

N = int(input())
tri = []
for _ in range(N):
    tri.append([int(i) for i in input().split()])

dp = [[0]*N for _ in range(N)]
for j in range(N):
    dp[N-1][j] = tri[N-1][j]

for i in range(N-2, -1, -1):
    for j in range(i+1):
        dp[i][j] = max(dp[i+1][j], dp[i+1][j+1]) + tri[i][j]

print(dp[0][0])

```

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。

2 最大连续子序列和

Longest Continuous Subsequence Sum/ Kadane's Algorithm

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

```
6
-2 11 -4 13 -5 -2
```

输出

```
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 $dp[0], dp[1], \dots, dp[n-1]$ 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。

```
n = int(input())
*a, = map(int, input().split())

dp = [0]*n
dp[0] = a[0]

for i in range(1, n):
    dp[i] = max(dp[i-1]+a[i], a[i])

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。

```
n = int(input())
*a, = map(int, input().split())

dp = [0]*n
start =[0]*n
dp[0] = a[0]

for i in range(1, n):
    if (dp[i-1] >= 0):
        dp[i] = dp[i-1] + a[i]
        start[i] = start[i-1]
    else:
        dp[i] = a[i]
        start[i] = i

max_val = max(dp)
pos = dp.index(max_val)

print(max_val, start[pos]+1, pos+1)
```

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

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

8 0 -2
```

样例输出

```
15
```

来源：翻译自 Greater New York 2001 的试题

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。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，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。通过遍历所有可能的行组合，我们可以找到最大的子矩阵。

```
def max_submatrix(matrix):
    def kadane(arr):
        max_end_here = max_so_far = arr[0]
        for x in arr[1:]:
            max_end_here = max(x, max_end_here + x)
            max_so_far = max(max_so_far, max_end_here)
        return max_so_far

    rows = len(matrix)
    cols = len(matrix[0])
    max_sum = float('-inf')

    for left in range(cols):
        temp = [0] * rows
        for right in range(left, cols):
            for row in range(rows):
                temp[row] += matrix[row][right]
            max_sum = max(max_sum, kadane(temp))
    return max_sum

n = int(input())
nums = []

while len(nums) < n * n:
```

```

nums.extend(input().split())
matrix = [list(map(int, nums[i * n:(i + 1) * n])) for i in range(n)]

max_sum = max_submatrix(matrix)
print(max_sum)

```

3 最大上升子序列 (LIS)

Longest Increasing Subsequence

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

```

7
1 7 3 5 9 4 8

```

样例输出

```

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（和最大连续子序列和问题一样，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 $A[j] < A[i] \wedge dp[j] + 1 > dp[i]$ （即把 $A[i]$ 跟在以 $A[i]$ 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：

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

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。

```

n = int(input())
*b, = map(int, input().split())
dp = [1]*n

for i in range(1, n):
    for j in range(i):
        if b[j] < b[i]:
            dp[i] = max(dp[i], dp[j]+1)

print(max(dp))

```

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

```

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

```

Bisect_left返回的位置，如果不是升序，值会被覆盖

The screenshot shows a Python Tutor visualization of the code:

```

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))
    
```

The output window shows the printed values:

```

7
1 7 3 5 9 4 8
4
    
```

The frames and objects section shows the state of variables:

- Global frame:**
 - bisect → module instance
 - n → 7
 - lis → list [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]
 - dp → list [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]
 - i → 8

Execution details:

- it just executed
- ie to execute
- << First | < Prev | Next > | Last >>
- Done running (20 steps)

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

```

7
1 7 3 5 9 4 8
    
```

样例输出

```

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。

```
input()
a = [int(x) for x in input().split()]

n = len(a)
dp = [0]*n

for i in range(n):
    dp[i] = a[i]
    for j in range(i):
        if a[j] < a[i]:
            dp[i] = max(dp[j]+a[i], dp[i])

print(max(dp))
```

4 背包问题

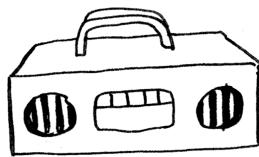
23421: 小偷背包

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。



STEREO

\$ 3000

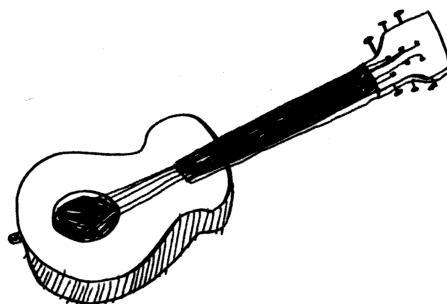
4 lbs



LAPTOP

\$ 2000

3 lbs



GUITAR

\$ 1500

1 lbs

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

```
3 4
3000 2000 1500
4 3 1
```

样例输出

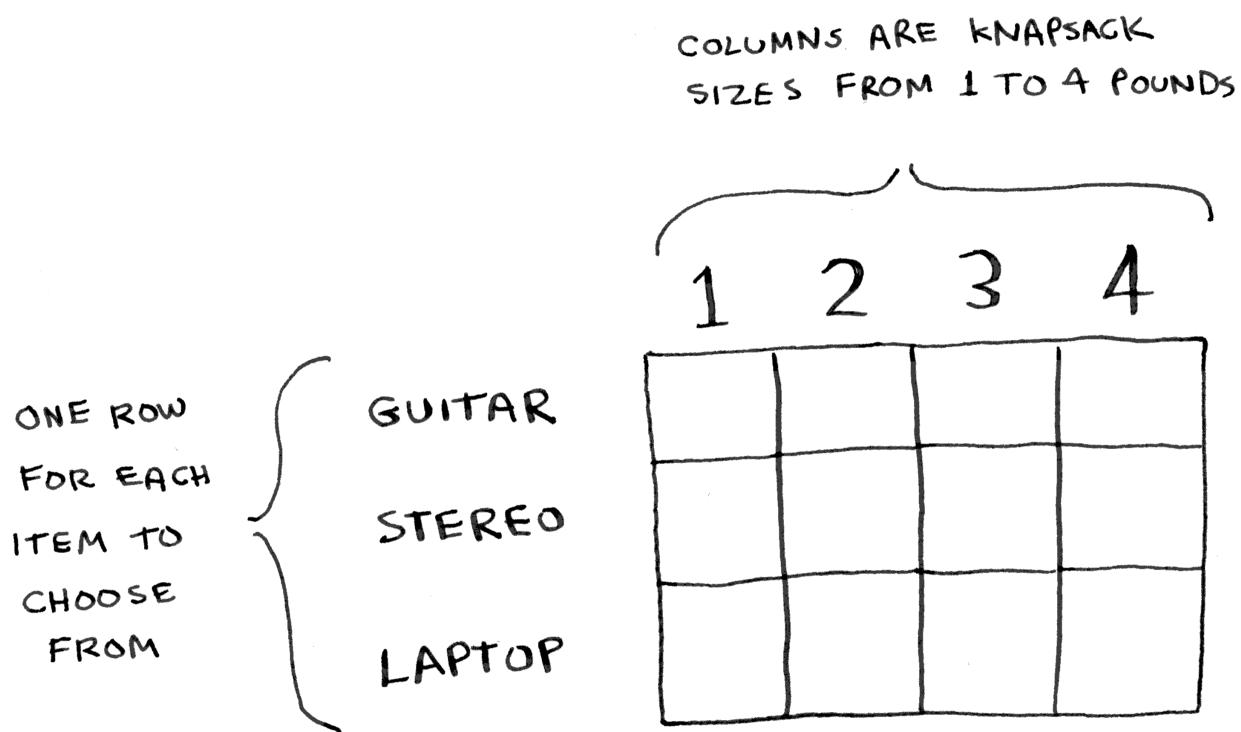
```
3500
```

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。



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 ' ' ' 5
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	

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。

$$\$3000 \text{ vs } (\underbrace{\$2000 + \$1500}_{\text{音响}} + \underbrace{\$1500}_{\text{吉他}})$$

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	\$1500	\$1500	\$1500
STEREO	G ↓ \$1500	G ↓ \$1500	G ↓ \$1500	G ↓ \$3000
LAPTOP	G ↓ \$1500	G ↓ \$1500	G ↓ \$2000	G ↓ \$3500
	G	G	L	L G
				5
				↑ 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。

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

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：

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

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。



```
# 动态规划之背包问题 (算法图解书中例子实现)
```

```
#第一步建立网格(横坐标表示[0,c]整数背包承重):(n+1)*(c+1)
def knapsack(n, c, w, p):
    cell = [[0 for j in range(c+1)] for i in range(n+1)]
    for j in range(c+1):
        #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
        cell[0][j] = 0
    for i in range(1, n+1):
        for j in range(1, c+1):
            #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。
            if j >= w[i-1]:
                cell[i][j] = max(cell[i-1][j], p[i-1] + cell[i-1][j - w[i-1]])
            else:
                cell[i][j] = cell[i-1][j]
    return cell
```

```
goodsnum, bagsize = map(int, input().split())
#goodsnum, bagsize = 3, 4
*value, = map(int, input().split())
*weight, = map(int, input().split())
#value, weight = [1500, 3000, 2000], [1, 4, 3] # guitar, stereo, laptop
```

```
cell = knapsack(goodsnum, bagsize, weight, value)
print(cell[goodsnum][bagsize])
```

```
n,b=map(int, input().split())
price=[0]+[int(i) for i in input().split()]
weight=[0]+[int(i) for i in input().split()]
bag=[[0]*(b+1) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,b+1):
        if weight[i]<=j:
            bag[i][j]=max(price[i]+bag[i-1][j-weight[i]], bag[i-1][j])
        else:
            bag[i][j]=bag[i-1][j]
print(bag[-1][-1])
```

优化空间复杂度

以上方法的时间和空间复杂度均为 $O(B N)$ ，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到 $O(B)$ 。先考虑上面讲的基本思路如何实现，肯定是有~~是~~一个主循环 $i \in 1...N$ ，每次算出来二维数组 $CELL[i;0...B]$ 的所有值。那么，如果只用一个数组 $CELL[0...B]$ ，能不能保证第 i 次循环结束后 $CELL[B]$ 中表示的就是我们定义的状态 $CELL[i; B]$ 呢？ $CELL[i; B]$ 是由 $CELL[i-1; B]$ 和 $CELL[i-1; B - W_i]$ 两个子问题递推而来，能否保证在推 $CELL[i; B]$ 时（也即在第 i 次主循环中推 $CELL[B]$ 时）能够取用 $CELL[i-1; B]$ 和 $CELL[i-1; B - W_i]$ 的值呢？事实上，这要求在每次主循环中我们以 $B...0$ 的递减顺序计算 $CELL[B]$ ，这样才能保证计算 $CELL[B]$ 时 $CELL[B - W_i]$ 保存的是状态 $CELL[i-1; B - W_i]$ 的值。

```
# 压缩矩阵/滚动数组 方法
N,B = map(int, input().split())
*p, = map(int, input().split())
*w, = map(int, input().split())

dp=[0]*(B+1)
for i in range(N):
    for j in range(B, w[i] - 1, -1):
        dp[j] = max(dp[j], dp[j-w[i]]+p[i])

print(dp[-1])
```

pythontutor.com

输入数据情况是：可装 $B=4$ 磅东西的背包。有 $N=3$ 件物品：价值 1500 美元重 1 磅的吉他，价值 3000 美元重 4 磅的音响，价值 2000 美元重 3 磅的笔记本。

Python 3.6
[known limitations](#)

```

1 # N,B = map(int, input().split())
2 # *p, = map(int, input().split())
3 # *w, = map(int, input().split())
4
5 N, B = 3, 4
6 p, w = [1500, 3000, 2000], [1, 4, 3]
7
8 dp=[0]*(B+1)
9 for i in range(N):
10     for j in range(B, w[i] - 1, -1):
11         dp[j] = max(dp[j], dp[j-w[i]]+p[i])
12
13 print(dp[-1])

```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First < Prev Next >> Last >>

Step 17 of 25

5 最长公共子串

《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。

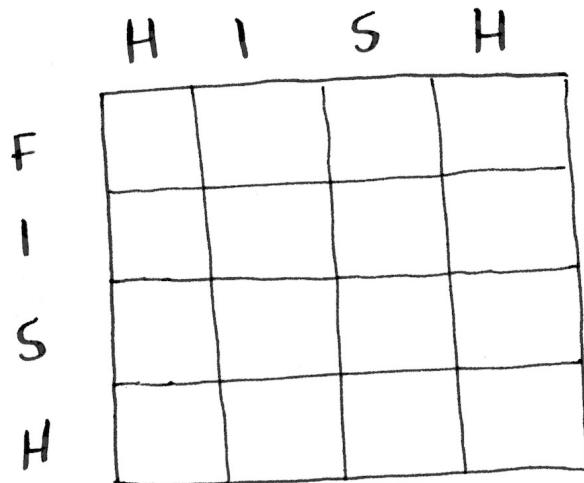
在这个例子中，只有两个类似的单词，真是太小儿科了。实际上，类似的单词很可能有数千个。Alex输入了hish，那他原本要输入的是fish还是vista呢？

基本思路

1 绘制网格

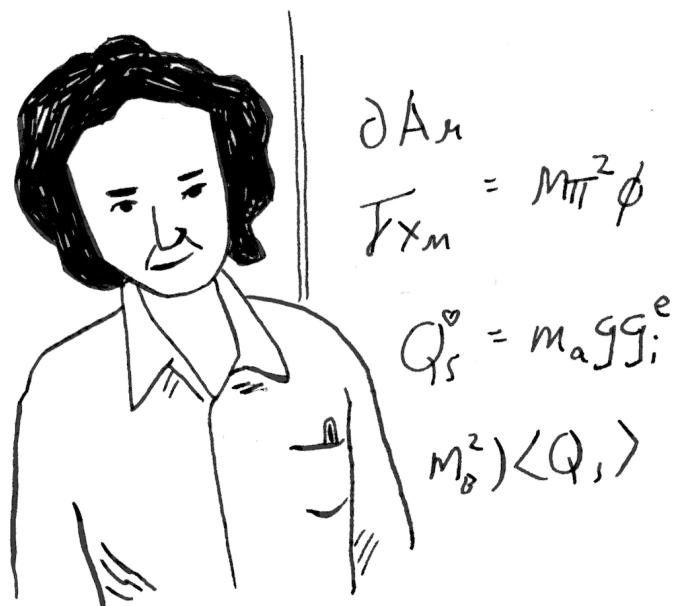
用于解决这个问题的网格是什么样的呢？要确定这一点，你得回答如下问题。单元格中的值是什么？如何将这个问题划分为子问题？网格的坐标轴是什么？在动态规划中，你要将某个指标最大化。在这个例子中，你要找出两个单词的最长公共子串。hish和fish都包含的最长子串是什么呢？hish和vista呢？这就是你要计算的值。别忘了，单元格中的值通常就是你要优化的值。在这个例子中，这很可能是一个数字：两个字符串都包

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

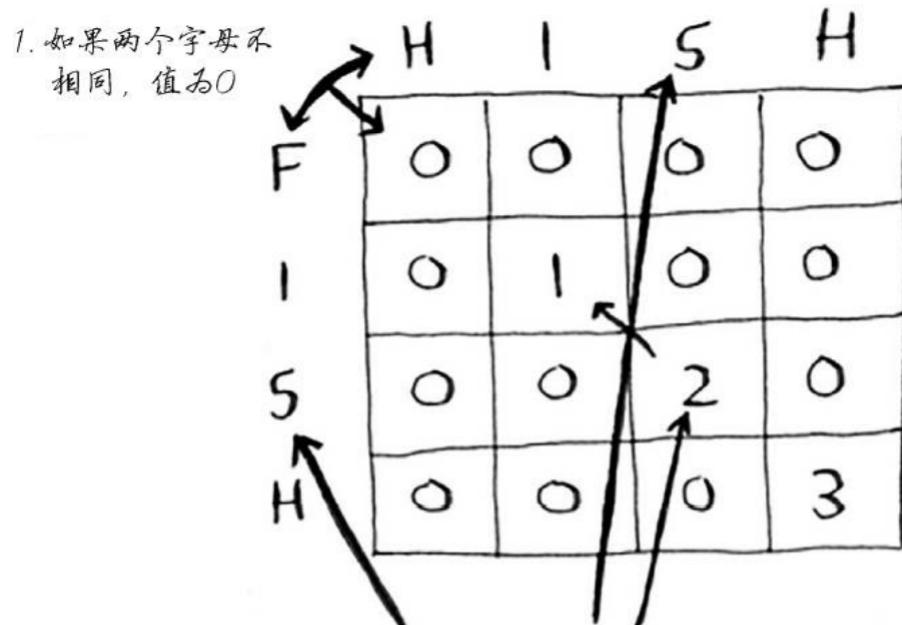
其他单元格的值呢？别忘了，每个单元格都是一个子问题的值。为何单元格(3, 3)的值为2呢？又为何单元格(3, 4)的值为0呢？请找出计算公式，再接着往下读。这样即便你没能找出正确的公式，后面的解释也将容易理解得多。

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。



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
```

最长公共子序列 (LCS)

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	1	0	0	0
O	0	2	0	0
R	0	0	0	0
T	0	0	0	0

vs

	F	O	S	H
F	1	0	0	0
I	0	0	0	0
S	0	0	1	0
H	0	0	0	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

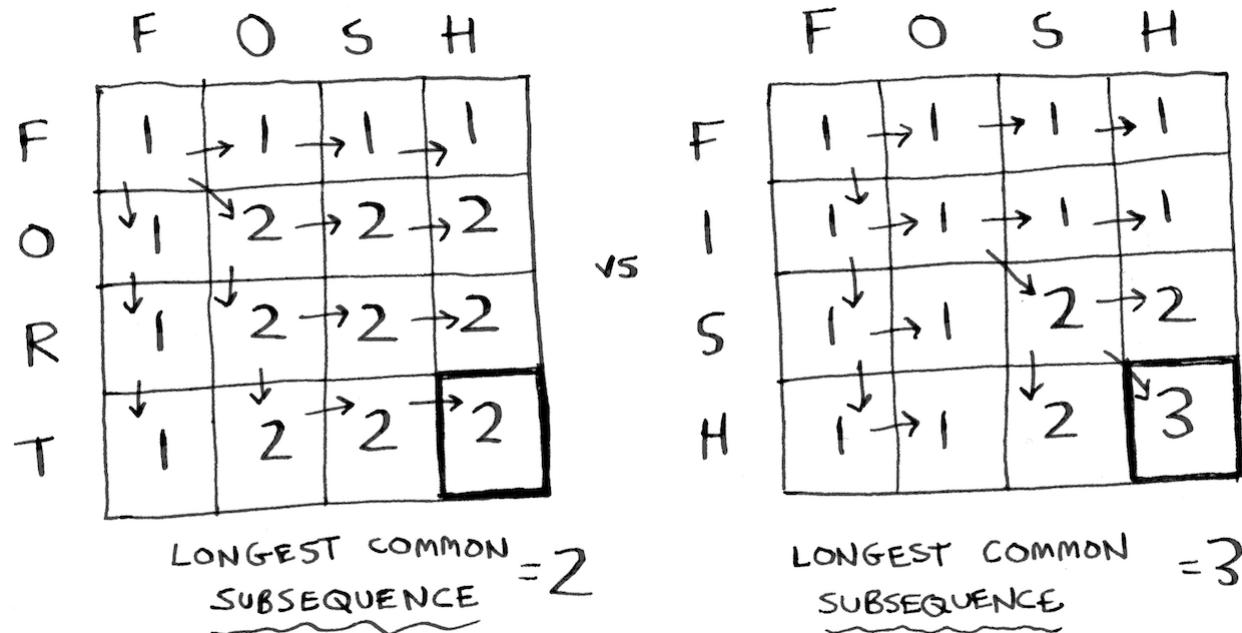
这里比较的是最长公共子串，但其实应比较最长公共子序列：两个单词中都有的序列包含的字母数。如何计算最长公共子序列呢？下面是用于计算fish和fosh的最长公共子序列的网格的一部分。

	F	O	S	H
F	1	1		
I	1			
S		1	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。

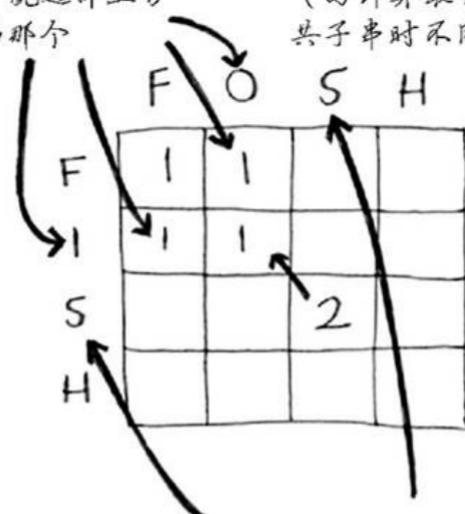
最长公共子序列之解决方案

最终的网格如下。



下面是填写各个单元格时使用的公式。

- 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)



- 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！

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

```
abcfbca      abfcab
programming   contest
abcd         mnp
```

样例输出

```
4
2
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.

```
while True:
    try:
        a, b = input().split()
    except EOFError:
        break
```

```

alen = len(a)
blen = len(b)

dp = [[0]*(blen+1) for i in range(alen+1)]

for i in range(1, alen+1):
    for j in range(1, blen+1):
        if a[i-1]==b[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

print(dp[alen][blen])

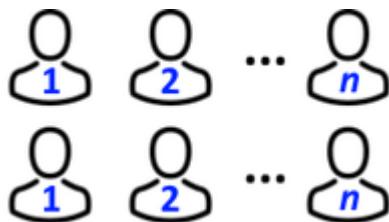
```

6 定义多个dp数组

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

```
5
9 3 5 7 3
5 8 1 4 5
```

output

```
29
```

input

```
3
1 2 9
10 1 1
```

output

```
19
```

input

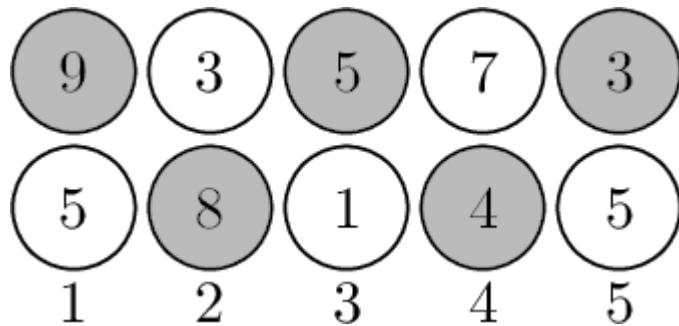
```
1
7
4
```

output

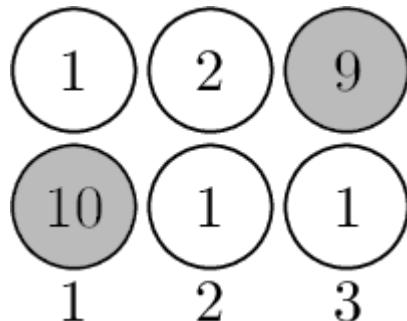
```
7
```

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:



```

n = int(input())
h1 = list(map(int, input().split()))
h2 = list(map(int, input().split()))

dp1 = [0] * n
dp2 = [0] * n

dp1[0] = h1[0]
dp2[0] = h2[0]

for i in range(1, n):
    dp1[i] = max(dp2[i - 1] + h1[i], dp1[i - 1])
    dp2[i] = max(dp1[i - 1] + h2[i], dp2[i - 1])

print(max(dp1[-1], dp2[-1]))

```

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

```
Sample Input1:
```

```
RRRRRBR
```

```
Sample Output1:
```

```
1
```

样例输出

```
Sample Input2:
```

```
RRRBBBRRRBBB
```

```
Sample Output2:
```

```
4
```

解释：先使用魔法2令 $k=12$ ，得到 $BBBRRRRBBBRRR$ ，然后使用魔法2令 $k=9$ ，得到 $RRRBBBRRRRRR$ ，然后使用魔法2令 $k=6$ ，得到 $BBBRRRRRRRRR$ ，然后使用魔法2令 $k=3$ ，得到 $RRRRRRRRRRRR$ 。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 R_n ，一个是前 n 朵玫瑰全变蓝，记为 B_n 。如果 $n+1$ 朵玫瑰是红色， $R(n+1)=R_n, B(n+1)$ 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 $B(n+1)=\min(R_n, B_n)+1$ 。如果 $n+1$ 朵玫瑰是蓝色就反过来。最后对 R_1, B_1 赋个值就可以快乐dp了。

```

r=list(input())
n=len(r)
R=[0]*n
B=[0]*n
if r[0]=="R":R[0]=0;B[0]=1
else:R[0]=1;B[0]=0
for i in range(n-1):
    if r[i+1]=="R":
        R[i+1]=R[i]
        B[i+1]=min(R[i],B[i])+1
    else:
        R[i+1]=min(R[i],B[i])+1
        B[i+1]=B[i]
print(R[-1])

```

7 “恰好”型dp

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.

Examples

input

```
5 5 3 2
```

output

2

input

7 5 5 2

output

2

Note

In the first example Polycarpus can cut the ribbon in such way: the first piece has length 2, the second piece has length 3.

In the second example Polycarpus can cut the ribbon in such way: the first piece has length 5, the second piece has length 2.

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。

```
n, a, b, c = map(int, input().split())
dp = [0]+[float('-inf')]*n

for i in range(1, n+1):
    for j in (a, b, c):
        if i >= j:
            dp[i] = max(dp[i-j] + 1, dp[i])

print(dp[n])
```

21458: 健身房 (dp)

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

小嘤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ti分钟，每天坚持做第i个训练一个月后预计可增肌wi千克。因为会导致效果变差，小嘤一天不会做相同的训练组多次。由于小嘤是强迫症，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，请帮助小嘤计算：他训练一个月后最大增肌量是多少呢？

对应英文描述：

Boingo, a bodybuilder from the United Kingdom of Great Britain and Northern Ireland, needs to choose a number of training sets in order to complete his muscle building program: there are **n** training groups, and it

takes **ti** minutes per day to do the **i**-th training set, and it's expected to gain **wi** kilograms of muscle after doing the first **i** training set every day for one month. Boingo would not do the same training set more than once a day because it's not effective. Since Boingo is obsessive-compulsive, he wants to spend **exactly t** minutes per day working out, he wants to gain the maximum amount of muscle mass after one month of training, please help him.

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 **ti,wi**。

保证 $0 < ti \leq T \leq 103$, $0 < n \leq 103$, $0 < wi < 20$ 。

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

```
sample1 in
6 4
2 1
4 7
3 5
3 5

sample1 out
10
```

样例输出

```
sample2 in
700 4
450 5
340 1
690 10
9 2

sample2 out
-1
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

```
# 23n2300011031, 黃源森
t,n=map(int,input().split())
dp=[0]+[-1]*(t+1)
for i in range(n):
    k,w=map(int,input().split())
    for j in range(t,k-1,-1):
        if dp[j-k]!=-1:
            dp[j]=max(dp[j-k]+w,dp[j])
print(dp[t])
```

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（n1, n2, n3, n4, n5, n6, n7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n1-n7，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

```
Sample1 Input:
5500
3 3 3 3 3 3
```

```
Sample1 Output:
2
```

样例输出

```
Sample2 Input:
125050
1 2 3 1 2 5 20
```

```
Smaple2 Output:  
Fail
```

来源: cs101-2019 龚世棋

```
# 2200015481, 陈涛
n=int(input())
tickets=list(map(int,input().split()))
price=[50,100,250,500,1000,2500,5000]
dp={0:0}
path={0:[0,0,0,0,0,0,0]}
for i in range(n):
    if i in dp:
        for k in range(7):
            if path[i][k]<tickets[k]:
                if i+price[k] in dp:
                    if dp[i]+1<dp[i+price[k]]:
                        dp[i+price[k]]=dp[i]+1
                        path[i+price[k]]=path[i][:]
                        path[i+price[k]][k]+=1
                else:
                    dp[i+price[k]]=dp[i]+1
                    path[i+price[k]]=path[i][:]
                    path[i+price[k]][k]+=1
    if n in dp:
        print(dp[n])
    else:
        print('Fail')
```

8 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。

9 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://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>