20241224-Week16 编程练习

Updated 1429 GMT+8 Dec 24, 2024

2024 fall, Complied by Hongfei Yan

20089: NBA门票（二进制分解）

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≤N≤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 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

第一版：超时代码

这是一个典型的“完全背包问题”，在此问题中，你需要选择一些球票，使得总花费正好等于给定的经费，并且买的票数最少。每种票的数量有限，因此你还需要考虑剩余的票数限制。具体步骤：

1. **定义状态**：定义 `dp[i]` 表示能否用 `i` 美元购买票，且 `dp[i]` 的值表示最少需要的票数。
2. **转移方程**：从每种票的最小数量到最大数量，尝试购买这类票，更新 `dp` 数组。
3. **初始化和边界条件**：初始化 `dp[0] = 0`（表示没有钱时，买票数为0）。其它 `dp[i]` 初始为无穷大（表示不可能的状态）。
4. **输出结果**：若 `dp[N]` 还是无穷大，则输出 "Fail"。否则，输出 `dp[N]`。

```python
def min_tickets(N, tickets_remaining):
    # 票价数组
    prices = [50, 100, 250, 500, 1000, 2500, 5000]

    # dp[i] 表示花费 i 美元时，最少需要多少票
    # 初始化为一个较大的值（无穷大），表示不可能的状态
    dp = [float('inf')] * (N + 1)
    dp[0] = 0  # 0 美元时需要 0 张票

    # 遍历每种票的价格
    for i in range(7):
        price = prices[i]
        max_count = tickets_remaining[i]

        # 对每种票的数量进行优化
        for j in range(N, -1, -1):  # 从大到小避免重复使用同张票
            # 购买每种票时的状态更新
            for k in range(1, max_count + 1):
                if j >= price * k:
                    dp[j] = min(dp[j], dp[j - price * k] + k)
                else:
                    break

    # 结果判断
    return dp[N] if dp[N] != float('inf') else "Fail"

N = int(input())  # 老板发的经费
tickets_remaining = list(map(int, input().split()))  # 各种票的剩余量

result = min_tickets(N, tickets_remaining)
print(result)
```

代码解释

1. **价格和剩余量**：`prices` 数组存储了每种票的价格，`tickets_remaining` 存储了每种票的剩余量。
2. **动态规划数组 `dp`**：初始化 `dp[0] = 0`（表示0美元时买票数为0），其它 `dp[i]` 初始为无穷大，表示不可能花费 `i` 美元。
3. **双重循环**：

- 外层循环遍历每种票的价格。
- 内层循环用来尝试购买不同数量的票，并更新 `dp` 数组。

4. **最终输出**：根据 `dp[N]` 的值判断是否有解，若是无穷大则输出 "Fail"，否则输出最小的票数。

第二版：pypy3可以过

使用一种优化方法：**完全背包问题中的多重背包优化**，即通过二进制分解（Binary Decomposition）的方法来减少时间复杂度。

优化思路

1. **优化背包问题**：将每种票的数量 `n_i` 分解为多个小的子问题，这样可以减少重复的计算。具体来说，每种票的数量 `n_i` 可以分解成 1, 2, 4, 8, ..., 形式的子问题，这样能够减少更新状态的次数。
2. **二进制分解法**：对于每种票，如果剩余票数 `n_i` 大于1，我们可以将其按二进制分解为多个“包”，从而有效减少每次计算的重复性。
3. **背包状态更新**：通过这个方法减少了每种票的处理时间，从而避免超时。

优化后的代码

```python
def min_tickets(N, tickets_remaining):
    # 票价数组
    prices = [50, 100, 250, 500, 1000, 2500, 5000]

    # dp[i] 表示花费 i 美元时，最少需要多少票
    dp = [float('inf')] * (N + 1)
    dp[0] = 0  # 0 美元时需要 0 张票

    # 遍历每种票的价格
    for i in range(7):
        price = prices[i]
        max_count = tickets_remaining[i]

        # 对每种票的数量进行优化
        # 使用二进制分解法，减小每次更新的次数
        k = 1
        while k <= max_count:
            # 处理 k 张票
            for j in range(N, price * k - 1, -1):
                dp[j] = min(dp[j], dp[j - price * k] + k)
            max_count -= k
            k *= 2

        # 处理剩余的票
        if max_count > 0:
            for j in range(N, price * max_count - 1, -1):
                dp[j] = min(dp[j], dp[j - price * max_count] + max_count)

    # 结果判断
```

```
    return dp[N] if dp[N] != float('inf') else "Fail"

# 输入处理
N = int(input())  # 老板发的经费
tickets_remaining = list(map(int, input().split()))  # 各种票的剩余量

# 输出结果
result = min_tickets(N, tickets_remaining)
print(result)
```

第三版：Python AC，倒序遍历时取-50为步长

1845ms AC

```
def min_tickets(N, tickets_remaining):
    # 票价数组
    prices = [50, 100, 250, 500, 1000, 2500, 5000]

    # dp[i] 表示花费 i 美元时，最少需要多少票
    dp = [float('inf')] * (N + 1)
    dp[0] = 0  # 0 美元时需要 0 张票

    # 遍历每种票的价格
    for i in range(7):
        price = prices[i]
        max_count = tickets_remaining[i]

        # 对每种票的数量进行优化
        # 使用二进制分解法，减小每次更新的次数
        k = 1
        while k <= max_count:
            # 处理 k 张票
            for j in range(N, price * k - 1, -50):
                dp[j] = min(dp[j], dp[j - price * k] + k)
            max_count -= k
            k *= 2

        # 处理剩余的票
        if max_count > 0:
            for j in range(N, price * max_count - 1, -1):
                dp[j] = min(dp[j], dp[j - price * max_count] + max_count)

    # 结果判断
    return dp[N] if dp[N] != float('inf') else "Fail"
```

```python

# 输入处理
N = int(input())  # 老板发的经费
tickets_remaining = list(map(int, input().split()))  # 各种票的剩余量

# 输出结果
result = min_tickets(N, tickets_remaining)
print(result)
```

第四版：Python AC，事先除以50优化

277ms AC

```python
def min_tickets(N, tickets_remaining):
    # 票价数组
    # prices = [50, 100, 250, 500, 1000, 2500, 5000]
    prices = [1, 2, 5, 10, 20, 50, 100]
    N //= 50

    # dp[i] 表示花费 i 美元时，最少需要多少票
    dp = [float('inf')] * (N + 1)
    dp[0] = 0  # 0 美元时需要 0 张票

    # 遍历每种票的价格
    for i in range(7):
        price = prices[i]
        max_count = tickets_remaining[i]

        # 对每种票的数量进行优化
        # 使用二进制分解法，减小每次更新的次数
        k = 1
        while k <= max_count:
            # 处理 k 张票
            for j in range(N, price * k - 1, -1):
                dp[j] = min(dp[j], dp[j - price * k] + k)
            max_count -= k
            k *= 2

        # 处理剩余的票
        if max_count > 0:
            for j in range(N, price * max_count - 1, -1):
                dp[j] = min(dp[j], dp[j - price * max_count] + max_count)

    # 结果判断
    return dp[N] if dp[N] != float('inf') else "Fail"


```

```python
# 输入处理
N = int(input())  # 老板发的经费
tickets_remaining = list(map(int, input().split()))  # 各种票的剩余量

# 输出结果
result = min_tickets(N, tickets_remaining)
print(result)
```

25572: 螃蟹采蘑菇（bfs变形）

bfs, http://cs101.openjudge.cn/practice/25572/

“采蘑菇的小螃蟹，背着一个大竹框”

一只螃蟹小呆想要从迷宫中的一个地方走到另一个地方采蘑菇，请你帮小呆判断一下它能否到达

迷宫所在的区域为一个n*n的格子矩阵，0的格子表示路，1的格子表示不能穿过的墙体，整个区域的外围也由墙体包围不能穿过。

小呆的身体占据相邻的两个格子，且小呆只能沿前后左右四个方向平移，不能斜着走也不能旋转，用两个数字5表示小呆的初始位置，用一个数字9表示蘑菇的位置，即小呆想要到达的地方（只要小呆两侧身体的任意一侧能够到达蘑菇的位置即可认为能够到达）

输入

第一行一个正整数n (n < 30)，表示迷宫区域的大小
接下来n行，每行n个整数，代表迷宫各个位置的情况。

输出

如果小呆能够到达终点，则输出yes
否则输出no

样例输入

```
Sample Input1:
6
0 0 0 0 0 9
0 0 1 0 1 1
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 1 0 0
0 0 0 1 5 5

Sample Output1:
yes
```

样例输出

```
Sample Input2:
6
0 0 0 0 0 9
0 0 1 0 1 1
0 0 0 0 0 0
0 0 0 1 0 0
0 0 0 1 0 5
0 0 0 1 0 5

Sample Output2:
no
```

提示 tags: bfs, dfs

来源: 2022fall-cs101, gdr

bfs，只是多加了一些东西，没有本质区别。

```python
from collections import deque

# 定义四个方向：右、下、左、上
dire = [(0, 1), (1, 0), (0, -1), (-1, 0)]

def bfs(a, x1, y1, x2, y2):
    visit = set()  # 使用集合来避免重复访问
    queue = deque([(x1, y1, x2, y2)])
    visit.add((x1, y1, x2, y2))  # 初始点加入访问集合

    while queue:
        xa, ya, xb, yb = queue.popleft()
        # 遍历四个方向
        for xi, yi in dire:
            # 计算新位置
            nx1, ny1 = xa + xi, ya + yi
            nx2, ny2 = xb + xi, yb + yi

            # 判断新位置是否合法
            if 0 <= nx1 < a and 0 <= ny1 < a and 0 <= nx2 < a and 0 <= ny2 < a:
                if (nx1, ny1, nx2, ny2) not in visit and Matrix[nx1][ny1] != 1 and Matrix[nx2][ny2] != 1:
                    # 加入队列并标记访问
                    queue.append((nx1, ny1, nx2, ny2))
                    visit.add((nx1, ny1, nx2, ny2))
                    # 检查是否到达目标
                    if Matrix[nx1][ny1] == 9 or Matrix[nx2][ny2] == 9:
                        return True
    return False

# 读取输入
a = int(input())
```

```python
Matrix = [list(map(int, input().split())) for _ in range(a)]

# 找到第一个和第二个 '5' 的位置
x1, y1, x2, y2 = -1, -1, -1, -1
found_first = False

for i in range(a):
    for j in range(a):
        if Matrix[i][j] == 5:
            if not found_first:
                x1, y1 = i, j
                Matrix[i][j] = 0  # 标记为已访问
                found_first = True
            else:
                x2, y2 = i, j
                Matrix[i][j] = 0  # 标记为已访问
                break
    if x2 != -1:  # 如果第二个 5 已经找到
        break

# 运行 BFS 检查是否可以从 (x1, y1) 到 (x2, y2)
check = bfs(a, x1, y1, x2, y2)
print('yes' if check else 'no')

```

思路:一位信科学长告诉我的，别管有几个点要移动，每个点都去判定就行，果真能行，这题就变成了一题很简单的bfs，之前做过一次，还是参照着解答，写了type1 type2区分横的竖的，如今看来没必要

```python
# 马凯权 24元培学院
from collections import deque

move = [(0, 1), (0, -1), (1, 0), (-1, 0)]


def bfs(s_x1, s_y1, s_x2, s_y2):

    if not ((abs(s_x1 - s_x2) == 1 and s_y1 == s_y2) or (s_x1 == s_x2 and abs(s_y1 - s_y2) == 1)):
        return False

    q = deque()
    q.append((s_x1, s_y1, s_x2, s_y2))
    inq = set()
    inq.add((s_x1, s_y1, s_x2, s_y2))

    while q:
        x1, y1, x2, y2 = q.popleft()

        if maze[x1][y1] == 9 or maze[x2][y2] == 9:
            return True
```

```
22
23          for dx, dy in move:
24              nx1, ny1 = x1 + dx, y1 + dy
25              nx2, ny2 = x2 + dx, y2 + dy
26
27              if 0 <= nx1 < n and 0 <= ny1 < n and 0 <= nx2 < n and 0 <= ny2 < n:
28                  if maze[nx1][ny1] != 1 and maze[nx2][ny2] != 1:
29                      if (nx1, ny1, nx2, ny2) not in inq:
30                          inq.add((nx1, ny1, nx2, ny2))
31                          q.append((nx1, ny1, nx2, ny2))
32
33      return False
34
35
36
37  n = int(input())
38  maze = [list(map(int, input().split())) for _ in range(n)]
39  a = []
40  for i in range(n):
41      for j in range(n):
42          if maze[i][j] == 5:
43              a.append([i, j])
44
45
46  if len(a) == 2:
47      result = bfs(a[0][0], a[0][1], a[1][0], a[1][1])
48      print('yes' if result else 'no')
49  else:
50      print('no')
51
```

01088: 滑雪（lru_cache, sorting）

dp/dfs similar, http://cs101.openjudge.cn/practice/01088

Michael喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael想知道载一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

```
 1  2  3  4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。

输入

输入的第一行表示区域的行数R和列数C(1 ≤ R,C ≤ 100)。下面是R行，每行有C个整数，代表高度h，0≤ h≤ 10000。

输出

输出最长区域的长度。

样例输入

```
5 5
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

样例输出

```
25
```

来源：USACO

```
"""
算法基础与在线实践：
递推的顺序是，讲所有点按高度从小到大排序，然后按照高度从小到大计算所有点的L值。
计算点(i,j)时，与它相邻的比它低的点必然已经计算过，因此可以递推
"""
import heapq

rows, cols = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(rows)]

# 使用最小堆存储元素及其坐标
heap = [(matrix[i][j], i, j) for i in range(rows) for j in range(cols)]
heapq.heapify(heap)

# 每个点的L值初始化为1
dp = [[1] * cols for _ in range(rows)]

# 定义方向数组，用于遍历上下左右
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# 记录最长递增路径长度
longest_path = 1

# 遍历堆中的元素，按高度从小到大处理
while heap:
    height, x, y = heapq.heappop(heap)
    for dx, dy in directions:
```

```
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols and matrix[nx][ny] < height:
            dp[x][y] = max(dp[x][y], dp[nx][ny] + 1)
    longest_path = max(longest_path, dp[x][y])

print(longest_path)
```

可以用 sort 代替 heapq，因为 heapq 的主要优势是动态获取最小/最大值，而这里并不需要动态插入或删除元素。我们只需按照高度从小到大处理所有点，使用排序即可实现相同的逻辑。

```
rows, cols = map(int, input().split())
matrix = [list(map(int, input().split())) for _ in range(rows)]

# 将所有点按高度从小到大排序
points = sorted([(matrix[i][j], i, j) for i in range(rows) for j in range(cols)])

# 每个点的L值初始化为1
dp = [[1] * cols for _ in range(rows)]

# 定义方向数组，用于遍历上下左右
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# 记录最长递增路径长度
longest_path = 1

# 从低到高，前面的不会对后面造成影响！
for height, x, y in points:
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols and matrix[nx][ny] < height:
            dp[x][y] = max(dp[x][y], dp[nx][ny] + 1)
    longest_path = max(longest_path, dp[x][y])

print(longest_path)
```

dfs即可，状态相同的位置的搜索结果永远相同，故可以用数组记录（进而只用搜索一次）（其实本质就是@lru_cache(maxsize=None)，因此也可以只用这一行语句解决）

```
import sys
sys.setrecursionlimit(1 << 30)
from functools import lru_cache
@lru_cache(maxsize=None)
def dfs(x, y):
    if d[x][y] > 0: return d[x][y]
    ans = 1
```

```
    for nx, ny in directions:
        tx, ty = x + nx, y + ny
        if 0 <= tx < n and 0 <= ty < m and a[tx][ty] < a[x][y]:
            ans = max(ans, dfs(tx, ty) + 1)
    d[x][y] = ans
    return ans
n, m = map(int, input().split())
a = [list(map(int, input().split())) for _ in range(n)]
d = [[0] * m for _ in range(n)]
directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
ans = 1
for i in range(n):
    for j in range(m):
        ans = max(ans, dfs(i, j))
print(ans)
```

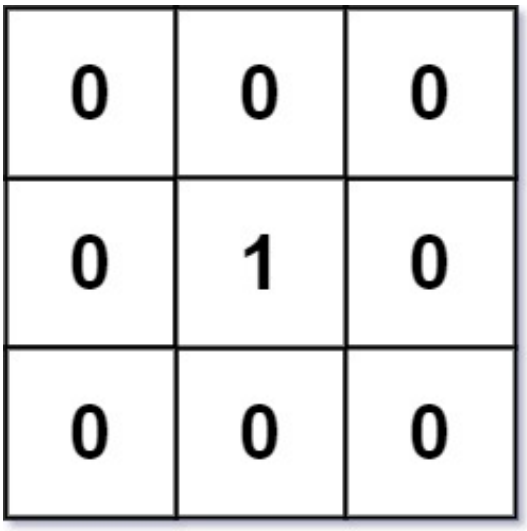
LCR 107.01 矩阵（滑雪升级版）

dp, https://leetcode.cn/problems/2bCMpM/

给定一个由 `0` 和 `1` 组成的矩阵 `mat` ，请输出一个大小相同的矩阵，其中每一个格子是 `mat` 中对应位置元素到最近的 `0` 的距离。

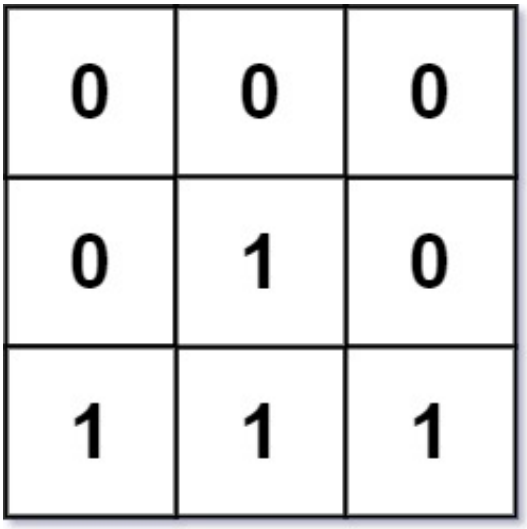
两个相邻元素间的距离为 `1` 。

示例 1：




```
输入：mat = [[0,0,0],[0,1,0],[0,0,0]]
输出：[[0,0,0],[0,1,0],[0,0,0]]
```

示例 2：




```
输入：mat = [[0,0,0],[0,1,0],[1,1,1]]
输出：[[0,0,0],[0,1,0],[1,2,1]]
```

提示：

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 104`
- `1 <= m * n <= 104`
- `mat[i][j] is either 0 or 1.`
- `mat` 中至少有一个 `0`

注意：本题与主站 542 题相同：https://leetcode-cn.com/problems/01-matrix/

是 OJ01088:滑雪 的升级版。因为矩阵每个点的高度有更新，不能只用sort一次，需要使用heapq。

```python
import heapq
from typing import List

class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        dp = [[float('inf')] * n for _ in range(m)]
        heap = []

        # 初始化，所有的0加入到堆中
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
```

```
                    dp[i][j] = 0
                    heapq.heappush(heap, (0, i, j))  # (distance, x, y)

        # 定义四个方向的移动
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        # 使用堆进行更新
        while heap:
            dist, x, y = heapq.heappop(heap)

            # 如果当前的距离大于 dp[x][y]，说明这个位置已经被更新过，不需要再次处理
            if dist > dp[x][y]:
                continue

            # 对当前点的四个方向进行处理
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n:
                    # 如果新位置的dp值可以更新（即发现更短的路径）
                    if dp[nx][ny] > dp[x][y] + 1:
                        dp[nx][ny] = dp[x][y] + 1
                        heapq.heappush(heap, (dp[nx][ny], nx, ny))

        return dp

# 测试用例
if __name__ == "__main__":
    mat = [[0,0,0],[0,1,0],[1,1,1]]
    print(Solution().updateMatrix(mat))
```

```
from typing import List
from collections import deque

class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        dp = [[float('inf')] * n for _ in range(m)]
        queue = deque()

        # 将所有0的元素加入队列并初始化dp数组
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    dp[i][j] = 0
                    queue.append((i, j))

        # 定义四个方向的移动
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

```

```
        # BFS开始
        while queue:
            x, y = queue.popleft()
            # 对当前点的四个方向进行处理
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n:
                    # 如果新位置的dp值可以更新（即发现更短的路径）
                    if dp[nx][ny] > dp[x][y] + 1:
                        dp[nx][ny] = dp[x][y] + 1
                        queue.append((nx, ny))

        return dp

# 测试用例
if __name__ == "__main__":
    mat = [[0,0,0],[0,1,0],[1,1,1]]
    print(Solution().updateMatrix(mat))
```

```
from typing import List
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        dp = [[float('inf') for _ in range(n)] for _ in range(m)]
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    dp[i][j] = 0
                else:
                    if i > 0:
                        dp[i][j] = min(dp[i][j], dp[i-1][j]+1)
                    if j > 0:
                        dp[i][j] = min(dp[i][j], dp[i][j-1]+1)
        for i in range(m-1, -1, -1):
            for j in range(n-1, -1, -1):
                if mat[i][j] == 0:
                    dp[i][j] = 0
                else:
                    if i < m-1:
                        dp[i][j] = min(dp[i][j], dp[i+1][j]+1)
                    if j < n-1:
                        dp[i][j] = min(dp[i][j], dp[i][j+1]+1)
        return dp

if __name__ == "__main__":
    mat = [[0,0,0],[0,1,0],[0,0,0]]
    print(Solution().updateMatrix(mat))
```

04102: 宠物小精灵之收服（二维费用01背包）

dp, http://cs101.openjudge.cn/practice/04102/

宠物小精灵是一部讲述小智和他的搭档皮卡丘一起冒险的故事。

一天，小智和皮卡丘来到了小精灵狩猎场，里面有很多珍贵的野生宠物小精灵。小智也想收服其中的一些小精灵。然而，野生的小精灵并不那么容易被收服。对于每一个野生小精灵而言，小智可能需要使用很多个精灵球才能收服它，而在收服过程中，野生小精灵也会对皮卡丘造成一定的伤害（从而减少皮卡丘的体力）。当皮卡丘的体力小于等于0时，小智就必须结束狩猎（因为他需要给皮卡丘疗伤），而使得皮卡丘体力小于等于0的野生小精灵也不会被小智收服。当小智的精灵球用完时，狩猎也宣告结束。

我们假设小智遇到野生小精灵时有两个选择：收服它，或者离开它。如果小智选择了收服，那么一定会扔出能够收服该小精灵的精灵球，而皮卡丘也一定会受到相应的伤害；如果选择离开它，那么小智不会损失精灵球，皮卡丘也不会损失体力。

小智的目标有两个：主要目标是收服尽可能多的野生小精灵；如果可以收服的小精灵数量一样，小智希望皮卡丘受到的伤害越小（剩余体力越大），因为他们还要继续冒险。

现在已知小智的精灵球数量和皮卡丘的初始体力，已知每一个小精灵需要的用于收服的精灵球数目和它在被收服过程中会对皮卡丘造成的伤害数目。请问，小智该如何选择收服哪些小精灵以达到他的目标呢?

输入

输入数据的第一行包含三个整数：N(0 < N < 1000)，M(0 < M < 500)，K(0 < K < 100)，分别代表小智的精灵球数量、皮卡丘初始的体力值、野生小精灵的数量。
之后的K行，每一行代表一个野生小精灵，包括两个整数：收服该小精灵需要的精灵球的数量，以及收服过程中对皮卡丘造成的伤害。

输出

输出为一行，包含两个整数：C，R，分别表示最多收服C个小精灵，以及收服C个小精灵时皮卡丘的剩余体力值最多为R。

样例输入

```
样例输入1:
10 100 5
7 10
2 40
2 50
1 20
4 20

样例输入2:
10 100 5
8 110
12 10
20 10
5 200
1 110
```

样例输出

```
样例输出1:
3 30

样例输出2:
0 100
```

提示

对于样例输入1：小智选择：(7,10) (2,40) (1,20) 这样小智一共收服了3个小精灵，皮卡丘受到了70点伤害，剩余100-70=30点体力。所以输出3 30
对于样例输入2：小智一个小精灵都没法收服，皮卡丘也不会收到任何伤害，所以输出0 100

二维费用01背包

- `dp` 是一个二维列表，`dp[i][j]` 表示在收服了 `i` 个小精灵且皮卡丘剩余 `j` 体力的情况下，小智还剩下的精灵球数量。
- `dp[0][M] = N` 表示初始状态，没有收服任何小精灵，皮卡丘的初始体力为 `M`，小智有 `N` 个精灵球。

`K*M*K`, 732ms AC.

```python
N, M, K = map(int, input().split())
dp = [[-1] * (M + 1) for i in range(K + 1)]
dp[0][M] = N
for i in range(K):
    cost, dmg = map(int, input().split())
    for p in range(M):
        for q in range(i + 1, 0, -1):
            if p + dmg <= M and dp[q - 1][p + dmg] != -1:
                dp[q][p] = max(dp[q][p], dp[q - 1][p + dmg] - cost)


def find():
    for i in range(K, -1, -1):
        for j in range(M, -1, -1):
            if dp[i][j] != -1:
                return i, j


captured, remaining_life = find()
print(captured, remaining_life)
```

`K*M*K`, 665ms AC

```python
# 官祺云 24 物理学院
"""
四个参数：个数K，限制N，限制M，值num。如果以MN为限制去找最大num，那么K*N*M会TLE。
这时注意到num也有限制num<=K，故可以选择把num改成其中一个限制，去寻找最小的N，
时间就会变成K*M*K，就不会TLE
"""
N, M, K = map(int, input().split())

dp = [[float('inf')] * (M + 1) for i in range(K + 1)]
dp[0][0] = 0
for i in range(K):
    cost, dmg = map(int, input().split())
    for j in range(M, dmg - 1, -1):
        for k in range(i + 1, 0, -1):
            if dp[k - 1][j - dmg] + cost <= N:
                dp[k][j] = min(dp[k][j], dp[k - 1][j - dmg] + cost)


def find():
    for i in range(K, -1, -1):
        for j in range(M + 1):
            if dp[i][j] != float('inf'):
                return i, M - j


captured, remaining_life = find()
print(captured, remaining_life)
```

`K*N*K` 1984ms AC

```python
N, M, K = map(int, input().split())

# 初始化DP数组
dp = [[float('inf')] * (K + 1) for _ in range(N + 1)]
for i in range(N + 1):
    dp[i][0] = 0  # 当不收服任何小精灵时，皮卡丘的体力消耗为0

# 读取每个小精灵的信息
for i in range(1, K + 1):
    u, v = map(int, input().split())
    for j in range(N, u - 1, -1):  # 从后向前遍历以避免重复计算
        for r in range(i, 0, -1):
            if dp[j - u][r - 1] != float('inf'):
                dp[j][r] = min(dp[j][r], dp[j - u][r - 1] + v)

# 找到最大收服数量
max_captured = 0
min_energy_used = M
for i in range(K, -1, -1):
    if dp[N][i] < M:
        max_captured = i
        min_energy_used = M - dp[N][i]
        break

# 输出结果
print(max_captured, min_energy_used)
```

`N*M*K`, Time Limit Exceeded

```python
def solve(ball, life, k, info):
    # 初始化动态规划表
    dp = [[0] * (life + 1) for _ in range(ball + 1)]

    # 动态规划
    for m in range(k):
        cost, damage = info[m]
        for i in range(ball, cost - 1, -1):
            for j in range(life, damage - 1, -1):
                dp[i][j] = max(dp[i][j], dp[i - cost][j - damage] + 1)

    # 寻找最优解
```

```
    for i in range(life + 1):
        for j in range(ball + 1):
            if dp[j][i] == dp[ball][life]:
                return dp[ball][life], life - i


# 读取输入
N, M, K = map(int, input().split())
pokemons = [tuple(map(int, input().split())) for _ in range(K)]

# 解决问题
captured, health = solve(N, M, K, pokemons)
print(captured, health)
```

66ms AC

```
# 刘思昊，24工学院
n, m, k = map(int, input().split())
wild_pokemon = []
for _ in range(k):
    balls, damage = map(int, input().split())
    wild_pokemon.append((balls, damage))

# 按伤害升序排序
wild_pokemon.sort(key=lambda x: x[1])

# 初始化 DP 数组
dp = [[0, 0] for _ in range(n + 1)]  # dp[i] = [最大收服数, 总伤害]

for balls, damage in wild_pokemon:
    for i in range(n, balls - 1, -1):
        prev_num, prev_damage = dp[i - balls]
        if prev_damage + damage >= m:  # 超过体力限制，跳过
            continue
        # 更新 DP：选择更优方案
        if prev_num + 1 > dp[i][0]:
            dp[i] = [prev_num + 1, prev_damage + damage]
        elif prev_num + 1 == dp[i][0]:
            dp[i][1] = min(dp[i][1], prev_damage + damage)


max_captured, total_damage = dp[-1]
print(max_captured, m - total_damage)
```

> 思路是 **动态规划 + 贪心**，并通过对小精灵列表按伤害值（`hp`）升序排序来优化结果。以下是对其正确性的分析和改进建议：
>
> **算法说明**
>
> 1. **排序优化**:

- 按照伤害值升序排列小精灵后，优先考虑那些对皮卡丘伤害较小的小精灵。
- 这种排序确保了在尽量收服更多小精灵的情况下，皮卡丘的剩余体力尽可能多。

2. **动态规划**:
   - `dp[i] = [num, hp]`：表示使用了 `i` 个精灵球时，最多能收服的精灵数 `num` 和相应情况下皮卡丘所受的总伤害 `hp`。
   - 状态转移公式：
     - 如果当前精灵可以被收服（即精灵球数足够，且皮卡丘体力不至于小于等于 0）：

       ```
       dp[i] = max(dp[i], [dp[i-ball][0] + 1, dp[i-ball][1] + hp])
       ```

       在数量相等时比较伤害值，取伤害值更小的方案。
3. **复杂度分析**:
   - 预排序的时间复杂度为 $O(K \log K)$。
   - 动态规划部分的复杂度为 $O(K \times N)$，因为每个小精灵需要遍历精灵球的使用情况。

正确性分析

该算法的思想与经典的 **0-1 背包问题** 类似，目的是在有限资源（精灵球和皮卡丘体力）的约束下，优化两个目标：

- 收服小精灵的数量（主目标）；
- 剩余体力的最大化（次目标）。

排序的贪心性

排序确保了优先考虑对皮卡丘伤害较小的小精灵，减少了高伤害精灵对后续选择的影响。因为动态规划的顺序遍历会保留最优解，所以排序不会影响结果的正确性。

1883D. In Love

data structure, greedy, 1500, https://codeforces.com/problemset/problem/1883/D

Initially, you have an empty multiset of segments. You need to process q operations of two types:

- + l r — Add the segment (l,r) to the multiset,
- − l r — Remove **exactly** one segment (l,r) from the multiset. It is guaranteed that this segment exists in the multiset.

After each operation, you need to determine if there exists a pair of segments in the multiset that do not intersect. A pair of segments (l,r) and (a,b) do not intersect if there does not exist a point x such that $l \le x \le r$ and $a \le x \le b$.

Input

The first line of each test case contains an integer q ($1 \le q \le 10^5$) — the number of operations.

The next q lines describe two types of operations. If it is an addition operation, it is given in the format + l r. If it is a deletion operation, it is given in the format − l r ($1 \le l \le r \le 10^9$).

Output

After each operation, print "YES" if there exists a pair of segments in the multiset that do not intersect, and "NO" otherwise.

You can print the answer in any case (uppercase or lowercase). For example, the strings "yEs", "yes", "Yes", and "YES" will be recognized as positive answers.

Example

input

```
12
+ 1 2
+ 3 4
+ 2 3
+ 2 2
+ 3 4
- 3 4
- 3 4
- 1 2
+ 3 4
- 2 2
- 2 3
- 3 4
```

output

```
NO
YES
YES
YES
YES
YES
NO
NO
YES
NO
NO
NO
```

Note

In the example, after the second, third, fourth, and fifth operations, there exists a pair of segments (1,2) (1,2) and (3,4)(3,4) that do not intersect.

Then we remove exactly one segment (3,4)(3,4), and by that time we had two segments. Therefore, the answer after this operation also exists.

```python
'''
The claim is that if the answer exists, we can take the segment with
the minimum right boundary and the maximum left boundary
(let's denote these boundaries as r and l). Therefore, if r<l
, it is obvious that this pair of segments is suitable for us.
Otherwise, all pairs of segments intersect because they have common
points in the range l…r.

先写了个超时的算法，然后看tutorial及其他人引入dict, heap的代码。
按照区间右端点从小到大排序。从前往后依次枚举每个区间。
假设当前遍历到的区间为第i个区间 [li, ri]，如果有li > ed,
说明当前区间与前面没有交集。
'''

import sys
import heapq
from collections import defaultdict
input = sys.stdin.readline

minH = []
maxH = []

ldict = defaultdict(int)
rdict = defaultdict(int)

n = int(input())

for _ in range(n):
    op, l, r = map(str, input().strip().split())
    l, r = int(l), int(r)
    if op == "+":
        ldict[l] += 1
        rdict[r] += 1
        heapq.heappush(maxH, -l)
        heapq.heappush(minH, r)
    else:
        ldict[l] -= 1
        rdict[r] -= 1

    '''
    使用 while 循环，将最大堆 maxH 和最小堆 minH 中出现次数为 0 的边界移除。
    通过比较堆顶元素的出现次数，如果出现次数为 0，则通过 heappop 方法将其从堆中移除。
    '''
    while len(maxH) > 0 >= ldict[-maxH[0]]:
        heapq.heappop(maxH)
    while len(minH) > 0 >= rdict[minH[0]]:
        heapq.heappop(minH)

    '''
    判断堆 maxH 和 minH 是否非空，并且最小堆 minH 的堆顶元素是否小于
```

```
    最大堆 maxH 的堆顶元素的相反数。
    '''
    if len(maxH) > 0 and len(minH) > 0 and minH[0] < -maxH[0]:
        print("Yes")
    else:
        print("No")
```

26646: 建筑修建

greedy, http://cs101.openjudge.cn/practice/26646/

小雯打算对一个线性街区进行开发，街区的坐标为[0,m)。

现在有n个开发商要承接建筑的修建工作，第i个承包商打算修建宽度为y[i]的建筑，并保证街区包含了x[i]这个整数坐标。

建筑为一个左闭右开的区间，为了方便规划建筑的左侧必须为整数坐标，且左右边界不能超出街区范围。

例如，当m=7, x[i]=5, y[i]=3时，[3,6),[4,7)是仅有的两种合法建筑，[2,5),[5,8)则是不合法的建筑。

两个开发商修建的建筑不能有重叠。例如，[3,5)+[4,6)是不合法的，而[3,5)+[5,7)则是合法的。

小雯想要尽量满足更多开发商的修建工作，请问在合理安排的情况下，最多能满足多少个开发商的需求?

输入

第一行两个整数n,m（n, m ≤ 1000）

之后n行，每行两个整数表示开发商的计划，其中第i行的整数为x[i],y[i]。

输入保证x[i]从小到大排列，且都在[0,m)之间。并且保证y[i] > 0。

输出

一个整数，表示最多能满足多少个开发商的需求。

样例输入

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

样例输出

```
2
```

```
# 23n2300011072(X)
def generate_intervals(x, width, m):
    temp = []
```

```
    for start in range(max(0, x-width+1), min(m, x+1)):
        end = start+width
        if end <= m:
            temp.append((start, end))
    return temp


n, m = map(int, input().split())
plans = [tuple(map(int, input().split())) for _ in range(n)]
intervals = []
for x, width in plans:
    intervals.extend(generate_intervals(x, width, m))
intervals.sort(key=lambda x: (x[1], x[0]))
cnt = 0
last_end = 0
for start, end in intervals:
    if start >= last_end:
        last_end = end
        cnt += 1
print(cnt)
```

27310: 积木

implementation, brute force, http://cs101.openjudge.cn/practice/27310

为了提高她的词汇量，奶牛 Bessie 拿来了一套共四块积木，每块积木都是一个正方体，六面各写着一个字母。她正在将积木排成一排，使积木顶部的字母拼出单词，以此来学习拼写。

给定 Bessie 四块积木上的字母，以及她想要拼写的单词列表，请判断她可以使用积木成功拼写列表中的哪些单词。

输入

输入的第一行包含 N（1 ≤ N ≤ 10），为 Bessie 想要拼写的单词数。以下四行，每行包含一个包含六个大写字母的字符串，表示 Bessie 的一块积木六面上的字母。以下 N 行包含 Bessie 想要拼写的 N 个单词。每一个均由 1 到 4 个大写字母组成。

输出

对于 Bessie 的列表中的每一个单词，如果她可以拼写这个单词则输出 YES，否则输出 NO。

样例输入

```
6
MOOOOO
OOOOOO
ABCDEF
UVWXYZ
COW
MOO
ZOO
MOVE
CODE
FARM
```

样例输出

```
YES
NO
YES
YES
NO
NO

解释：在这个例子中，Bessie 可以拼写 COW，ZOO 和 MOVE。令人难过地，她无法拼出 MOO，
因为唯一包含 M 的积木不能同时用于 O。她无法拼出 FARM，因为没有包含字母 R 的积木。
她无法拼出 CODE，因为 C，D 和 E 属于同一块积木。
```

提示

tags: implementation, brute force

来源

2023fall cwy. http://usaco.org/index.php?page=viewproblem2&cpid=1205&lang=en

```python
from collections import defaultdict
from itertools import permutations

a = defaultdict(int)
b = defaultdict(int)
c = defaultdict(int)
d = defaultdict(int)
n = int(input())

for i in input():
    a[i] += 1
for i in input():
    b[i] += 1
for i in input():
    c[i] += 1
for i in input():
```

```
    d[i] += 1

dicts = [a, b, c, d]

def check(word):
    for perm in permutations(dicts, len(word)):
        for i, d in enumerate(perm):
            if word[i] not in d:
                break
        else:
            return 'YES'
    else:
        return 'NO'

for _ in range(n):
    word = input()
    print(check(word))
```

27103: 最短的愉悦旋律长度

greedy, http://cs101.openjudge.cn/practice/27103/

江凯同学推荐了计概A的一个题目，题面如下：

在去年的计概期末，大家知道了P大富哥李哥，而李哥有个好朋友陈哥。

与李哥不同，陈哥喜欢钻研音乐，他创造了一种特殊的音乐，这种音乐可以包含 M种 不同的音符 (1<=M<=10000)，因此，每个音符都可以唯一编码为1到10000之间的一个整数。陈哥创作的每首乐曲都可以看做由 N个 (1<=N<=100000)这种音符所组成的音符序列。

例如，可以利用3种音符组成如下的音符序列：1,2,2,3,2,3,3,1

根据陈哥敏感的音乐细胞，他发现，针对每首给定的音符序列，有些子序列（不一定连续）会出现在该序列中，而有些子序列则不会出现在该序列中。

例如，对于上面例子中的音符序列，子序列"1,2,2,1"就出现在该序列中，而子序列“2,1,3”就没有出现在该序列中。

热心的陈哥告诉你：正如子序列"1,2,2,1"一样，子序列中的数不需要在原始音符序列中“连续出现”，只要其遵循原本在音符序列中的先后次序，即使子序列的各个数之间穿插有其他数字，也可认为这个子序列出现于音符序列中。

现在，给定一首已经创作完成的包含 N个 音符的音符序列，陈哥想用 M种 不同的音符构造一个子序列，使之“不出现”在给定的乐曲序列中。李哥想将其称为“崭新的旋律”，但陈哥还是喜欢称为“愉悦的旋律”，请问这个“愉悦的旋律”的最短长度是多少?

输入

第1行输入两个整数 N 和 M ，接下来1行输入音符序列。

输出

输出一行，包含一个整数，代表最短的“愉悦的旋律”的长度。

样例输入

```
14 5
1 5 3 2 5 1 3 4 4 2 5 1 2 3
```

样例输出

```
3

样例解释：
所有长度为1和2的可能的子序列都出现了，但长度为3的子序列"2,2,4"却没有出现。
```

提示

只需要告诉陈哥最短的“愉悦的旋律”的长度就行。

来源：计概A 2023

思路：如果没有n+1套，谁导致n+1套凑不齐

```
N, M = map(int, input().split())
*melody, = map(int, input().split())

cnt = 1
note = set()
for i in melody:
    note.add(i)
    if len(note) == M:
        cnt += 1
        note.clear()

print(cnt)
```

> 要去寻找这个“未出现的序列"的最短长度，不妨这样去看待一个问题，以M=3为例，既有3种音符 123。首先这样去想，长度为1的子序列，是不是 1和2和3？长度为2的子序列 是不是[123]和[123]两个集合中任选一个？按照前后顺序排起来？长度为3的子序列，是不是集合[123] 和[123] 和[123]三个集合从前往后，每次取一个，按照前后顺序排起来？采用分块出现的思想，那么本题就很清晰了，例如，对于1523444512533，可以分成几个部分，15234/4451253/3，发现，第一个分隔号前，已经出现了1,2,3,4,5一次全部数字，第一个分隔号到第二个分隔号，又出现了1,2,3,4,5，完整的一次？那么我们可以肯定，长度为1和为2的全部子序列已经可以得到，所以只有长度为3的子序列没有被全部枚举，答案就是3。

27104: 世界杯只因

greedy/dp, http://cs101.openjudge.cn/practice/27104/

卡塔尔世界杯正在火热进行中，P大富哥李哥听闻有一种叫"肤白·态美·宇宙无敌·世界杯·预测鸡"的鸡品种（以下简称为只因）有概率能准确预测世界杯赛果，一口气买来无数只只因，并把它们塞进了N个只因窝里，但只因窝实在太多了，李哥需要安装摄像头来观测里面的只因的预测行为。

具体来说，李哥的只因窝可以看作分布在一条直线上的N个点，编号为1到N。由于每个只因窝的结构不同，在编号为i的只因窝处安装摄像头，观测范围为a_i，其中a是长为N的整数列，表示若在此安装摄像头，可以观测到编号在 [i - a_i , i + a_i]（闭区间）内的所有只因窝。

李哥觉得摄像头成本高，决定抠门一下，请你来帮忙看看最少需要安装多少个摄像头，才能观测到全部N个只因窝。作为回报，他会请你喝一杯芋泥波波牛乳茶。

输入

第一行：一个正整数，代表有N个只因窝。
第二行给出数列a：N个非负整数，第i个数代表a_i，也就是在第i个只因窝装摄像头能观测到的区间的半径。

数据保证 N ≤ 500000，0 ≤ a_i ≤ N

输出

一个整数，即最少需要装的摄像头数量。

样例输入

```
10
2 0 1 1 0 3 1 0 2 0
```

样例输出

```
3
```

提示：彩蛋：只因们很喜欢那个穿着蓝白球衣长得像黄金矿工的10号

来源：计概A 2022期末


```python
# 计概B不A不C故没AC      224ms
def min_cameras_to_cover_all(ranges):
    n = len(ranges)
    ptr = 0
    num = 0
```

```
    mx = max(ranges)
    while ptr < n:
        # 假设下一个指针位置为当前指针加上当前观测范围再加一
        nxt = ptr + ranges[ptr] + 1

        # 遍历一个以当前指针为中心的大窗口，考虑到最大观测范围mx的影响
        for i in range(max(0, ptr - mx), min(n, ptr + mx + 1)):
            if 0 <= i < n and i - ranges[i] <= ptr and i + ranges[i] + 1 > nxt:
                nxt = i + ranges[i] + 1  # 更新最远可达位置

        num += 1  # 每次循环代表安装了一个摄像头
        ptr = nxt  # 移动到最远可达位置继续搜索

    return num


# 输入处理
if __name__ == "__main__":
    _ = int(input())  # 忽略第一行的N值
    ranges = list(map(int, input().strip().split()))
    print(min_cameras_to_cover_all(ranges))
```

27141: 完美的爱

http://cs101.openjudge.cn/practice/27141/

一天，小帅和小美来到一处世外桃源，映入眼帘的是岛主为二人准备好的n个礼物，排成一排。但是岛主说他们只能选 一部分的礼物带走，具体规则如下：

如果他们能从这n个礼物中选出连续相邻的若干个礼物，且这些礼物的平均价值是520，那么他们就可以带走这些礼物。

那么问题来了，他们这次能带走总价值最多为多少的礼物呢。

输入

第一行为一个正整数n，n<10^5
第二行为n个整数，表示每个礼物的价值

输出

总价值(整数)

样例输入

```
10
520 521 519 12 124 512 520 519 518 522
```

样例输出

```
1560
```

来源

2023 课程微信群

超时代码

```python
def find_max_value(n, gifts):
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i + 1] = prefix_sum[i] + gifts[i]

    max_value = 0
    for i in range(n):
        for j in range(i + 1, n + 1):
            subarray_sum = prefix_sum[j] - prefix_sum[i]
            subarray_length = j - i
            if subarray_sum / subarray_length == 520:
                max_value = max(max_value, subarray_sum)

    return max_value

# 读取输入
n = int(input())
gifts = list(map(int, input().split()))

# 计算结果
result = find_max_value(n, gifts)
print(result)
```

为了使用滑动窗口优化这段代码，我们需要改变策略来寻找平均值为520的子数组。原始代码中嵌套循环的时间复杂度是O(n^2)，这在处理大量数据时会非常慢。我们可以通过维护一个滑动窗口，并且只遍历一次数组（线性时间）来优化它。

但是，对于这个问题来说，直接应用传统的滑动窗口并不合适，因为我们要找的是一个平均值等于特定值的连续子数组，而不是简单的最大或最小和。这里的关键在于转换问题：我们可以将原问题转化为求解前缀和的问题，然后利用哈希表存储已经访问过的前缀和及其对应的索引，从而快速判断是否存在满足条件的子数组。

下面是优化后的代码实现：

```python
from collections import defaultdict

def find_max_value(n, gifts):
    target_average = 520
    # 计算需要的偏移量使得目标变为0
```

```
    gifts_offset = [x - target_average for x in gifts]

    prefix_sum = 0
    max_length = 0
    sum_indices = defaultdict(list)
    sum_indices[0].append(-1)  # 初始化，表示从开始到-1的和为0

    for i, gift in enumerate(gifts_offset):
        prefix_sum += gift
        if prefix_sum in sum_indices:
            # 如果当前前缀和之前出现过，说明存在一个子数组其元素平均值为target_average
            length = i - sum_indices[prefix_sum][0]  # 取最早的索引来获得最长的子数组
            max_length = max(max_length, length)
        sum_indices[prefix_sum].append(i)

    # 计算最大子数组的总和
    max_value = max_length * target_average if max_length > 0 else 0
    return max_value


# 读取输入
n = int(input())
gifts = list(map(int, input().split()))

# 计算结果
result = find_max_value(n, gifts)
print(result)
```

> - **gifts_offset**：将每个礼物的价值减去目标平均值520，这样做的目的是让目标变成找到一个和为0的子数组。
> - **prefix_sum**：这是从前到当前位置所有元素的累积和。
> - **sum_indices**：这是一个字典，用来记录每个前缀和第一次出现的位置。如果同一个前缀和再次出现，则意味着在这两个位置之间的子数组的平均值就是目标平均值。
> - **max_length**：用于跟踪最长的有效子数组长度。
> - **max_value**：最后计算的最大子数组的总和。
>
> 这种方法确保了算法能够在O(n)时间内完成，极大地提高了效率。请注意，这里的`max_value`实际上是基于`max_length`和`target_average`计算得出的，因为真正关心的是最长的子数组，而不是具体的数值总和。

27384: 候选人追踪

http://cs101.openjudge.cn/practice/27384/

超大型偶像团体HIHO314159总选举刚刚结束了。制作人小Hi正在复盘分析投票过程。

小Hi获得了N条投票记录，每条记录都包含一个时间戳Ti以及候选人编号Ci，代表有一位粉丝在Ti时刻投了Ci一票。

给定一个包含K名候选人集合S={S1, S2, ... SK}，小Hi想知道从投票开始(0时刻)，到最后一张票投出的时刻(max{Ti})，期间有多少时间得票最多的前K名候选人恰好是S中的K名候选人。

注意这里对前K名的要求是"严格"的，换句话说，S中的每一名候选人得票都要大于任何一名S之外的候选人。S集合内名次先后不作要求。

注：HIHO314159这个团体有314159名团员，编号是1~314159。

输入

第一行包含两个整数N和K。

第二行包含2N个整数：T1, C1, T2, C2, ... TN, CN。

第三行包含K个整数：S1, S2, ... SK。

对于30%的数据，1 ≤ N, K ≤ 100

对于60%的数据，1 ≤ N, K ≤ 1000

对于100%的数据, 1 ≤ N, K ≤ 314159 1 ≤ Ti ≤ 1000000 1 ≤ Ci, SK ≤ 314159

输出

一个整数，表示前K名恰好是S一共持续了多少时间。

样例输入

```
10 2
3 1 4 1 5 1 4 3 6 5 8 3 7 5 8 5 9 1 10 5
1 5
```

样例输出

```
3
```

来源：HC

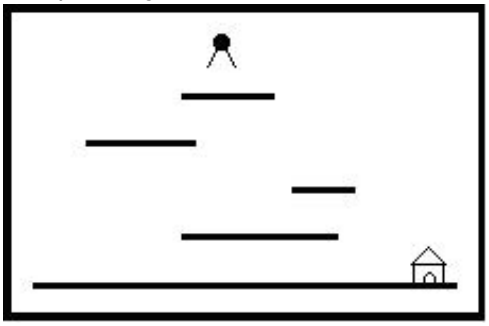
```python
import heapq

maxn = 320000
cnt = [0] * maxn
n, k = 0, 0
vis = [False] * maxn

n, k = map(int, input().split())
*records, = map(int, input().split())
arr = [(records[i], records[i+1]) for i in range(0, 2*n, 2)]

Q = []
candidates = list(map(int, input().split()))
for i in range(k):
```

```
    heapq.heappush(Q, (0, candidates[i]))
    vis[candidates[i]] = True

arr = sorted(arr[:n])
if k == 314159:
    print(arr[n-1][0])
    exit()

rmx = 0
rs = 0
for i in range(n):
    c = arr[i][1]
    cnt[c] += 1
    if vis[c]:
        while cnt[Q[0][1]]: # 懒更新，每次只更新到堆中的最小值是实际的最小值
            f = heapq.heappop(Q)
            f = (f[0] + cnt[f[1]], f[1])
            heapq.heappush(Q, f)
            cnt[f[1]] = 0
    else:
        rmx = max(rmx, cnt[c])
    if i != n-1 and arr[i+1][0] != arr[i][0] and Q[0][0] > rmx:
        rs += arr[i+1][0] - arr[i][0]

print(rs)
```

01661: Help Jimmy

dfs/dp, http://cs101.openjudge.cn/practice/01661

"Help Jimmy" 是在下图所示的场景上完成的游戏：



场景中包括多个长度和高度各不相同的平台。地面是最低的平台，高度为零，长度无限。

Jimmy老鼠在时刻0从高于所有平台的某处开始下落，它的下落速度始终为1米/秒。当Jimmy落到某个平台上时，游戏者选择让它向左还是向右跑，它跑动的速度也是1米/秒。当Jimmy跑到平台的边缘时，开始继续下落。Jimmy每次下落的高度不能超过MAX米，不然就会摔死，游戏也会结束。

设计一个程序，计算Jimmy到底地面时可能的最早时间。

输入

第一行是测试数据的组数t（0 ≤ t ≤ 20）。每组测试数据的第一行是四个整数N，X，Y，MAX，用空格分隔。N是平台的数目（不包括地面），X和Y是Jimmy开始下落的位置的横竖坐标，MAX是一次下落的最大高度。接下来的N行每行描述一个平台，包括三个整数，X1[i]，X2[i]和H[i]。H[i]表示平台的高度，X1[i]和X2[i]表示平台左右端点的横坐标。1 ≤ N ≤ 1000，-20000 ≤ X, X1[i], X2[i] ≤ 20000，0 < H[i] < Y ≤ 20000（i = 1..N）。所有坐标的单位都是米。

Jimmy的大小和平台的厚度均忽略不计。如果Jimmy恰好落在某个平台的边缘，被视为落在平台上。所有的平台均不重叠或相连。测试数据保证问题一定有解。

输出

对输入的每组测试数据，输出一个整数，Jimmy到底地面时可能的最早时间。

样例输入

```
1
3 8 17 20
0 10 8
0 10 13
4 14 3
```

样例输出

```
23
```

来源：POJ Monthly--2004.05.15, CEOI 2000, POJ 1661, 程序设计实习2007

```python
# 查达闻 2300011813
from functools import lru_cache

@lru_cache
def dfs(x, y, z):
    for i in range(z+1, N+1):
        if y - MaxVal > p[i][2]:
            return 1 << 30
        elif p[i][0] <= x <= p[i][1]:
            left = x - p[i][0] + dfs(p[i][0], p[i][2], i)
            right = p[i][1] - x + dfs(p[i][1], p[i][2], i)
            return min(left,right)

    if y <= MaxVal:
        return 0
    else:
        return 1 << 30


for _ in range(int(input())):
    N, ini_x, ini_y, MaxVal = map(int, input().split())

```

```
    p = []      #platform
    p.append( [0, 0, 1 << 30] ) # 1<<30 大于 20000*2*1000
    for _ in range(N):
        p.append([int(x) for x in input().split()])
    p.sort(key = lambda x:-x[2])

    print(ini_y + dfs(ini_x, ini_y, 0))
```

01664: 放苹果

dp, dfs, http://cs101.openjudge.cn/practice/01664/

把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用K表示）5，1，1和1，5，1 是同一种分法。

输入

第一行是测试数据的数目t（0 <= t <= 20）。以下每行均包含二个整数M和N，以空格分开。1<=M，N<=10。

输出

对输入的每组数据M和N，用一行输出相应的K。

样例输入

```
1
7 3
```

样例输出

```
8
```

来源

lwx@POJ

将问题转化为动态规划（DP）形式，可以通过构造一个二维DP数组来解决问题，其中 `dp[m][n]` 表示将 `m` 个苹果放入 `n` 个盘子的分法数。

状态转移方程

1. 如果没有苹果或者只有一个盘子：
    - `dp[m][n]=1`（基础情况）。
2. 如果盘子数大于苹果数：
    - `dp[m][n]=dp[m][m]`（因为多余的盘子没有意义）。
3. 一般情况下：

- `dp[m][n]=dp[m][n-1]+dp[m-n][n]`，
- 其中：
  - `dp[m][n-1]` 表示至少有一个盘子空着的情况。
  - `dp[m-n][n]` 表示每个盘子至少放一个苹果的情况。

```python
def apple_distribution(t, cases):
    # 最大苹果数和盘子数
    max_m = max(c[0] for c in cases)
    max_n = max(c[1] for c in cases)

    # 初始化DP数组
    dp = [[0] * (max_n + 1) for _ in range(max_m + 1)]

    # 基础情况
    for m in range(max_m + 1):
        dp[m][1] = 1  # 只有一个盘子
    for n in range(max_n + 1):
        dp[0][n] = 1  # 没有苹果

    # 填表
    for m in range(1, max_m + 1):
        for n in range(2, max_n + 1):
            if n > m:
                dp[m][n] = dp[m][m]  # 盘子多于苹果
            else:
                dp[m][n] = dp[m][n-1] + dp[m-n][n]

    # 处理每个测试用例
    results = []
    for m, n in cases:
        results.append(dp[m][n])

    return results

# 主函数
def main():
    t = int(input())  # 测试数据数目
    cases = []
    for _ in range(t):
        m, n = map(int, input().split())
        cases.append((m, n))
    results = apple_distribution(t, cases)
    for res in results:
        print(res)

# 样例测试
if __name__ == "__main__":
    main()
```

时间复杂度

- 预处理 DP 表的复杂度为 $O(M \times N)$，其中 M,N 是最大苹果数和盘子数。

dfs

```python
def count_ways(m, n):
    # 如果只有一个盘子，只能有一种分法
    if n == 1:
        return 1
    # 如果苹果数为0，只有一种分法：所有盘子空
    if m == 0:
        return 1
    # 如果盘子数大于苹果数，最多只需用前m个盘子分苹果
    if n > m:
        return count_ways(m, m)
    # 分为两种情况：至少每个盘子放一个苹果；有盘子空着
    return count_ways(m, n - 1) + count_ways(m - n, n)


t = int(input())  # 测试数据数目
results = []
for _ in range(t):
    m, n = map(int, input().split())
    results.append(count_ways(m, n))
# 输出结果
for res in results:
    print(res)
```

09267: 核电站

dfs, dp, http://cs101.openjudge.cn/practice/09267/

一个核电站有 N 个放核物质的坑，坑排列在一条直线上。如果连续 M 个坑中放入核物质，则会发生爆炸，于是，在某些坑中可能不放核物质。

任务：对于给定的 N 和 M ，求不发生爆炸的放置核物质的方案总数。

输入

只有一行，两个正整数 N，M (1 < N < 50, 2 <= M <= 5)。

输出

一个正整数 S，表示方案总数。

样例输入

```
4 3
```

样例输出

```
13
```

参考：https://blog.csdn.net/weixin_50624971/article/details/117337552

这题和放苹果挺类似的，都是要分情况讨论i和m的大小关系
case1：如果i<m
这说明在这段区间里怎么放都不会炸，那么状态转移方程就是：a[i]=2*a[i-1].（乘2是因为每个坑有放和不放两种情况）
case2：如果i==m
情况同case1，只是要减去区间全放（会炸）的情况，那么状态转移方程就是：a[i]=2*a[i-1]-1.
case3：如果i>m
这是就要考虑会炸的情况了。我们采用减法的方式——即从总情况里减去会炸的情况。
如果第i位是不能放的，那么说明i-m这段区间肯定全放了，而且i-m-1这一位一定是0，因为如果该位是1的话就会在之前被处理掉，所以如果i为不能放，那么i-m-1这段区间的所有可能都需要从答案里减掉。那么状态转移方程就是：a[i]=2*a[i-1]-a[i-1-m]

```python
# 23n1900014516
n, m = map(int, input().split())
DP = [0] * 60
DP[0] = 1 #DP[i]是第i个位置的方案数。

for i in range(1, n + 1):
    if i < m: #达不到连续放置m个的情况
        DP[i] = DP[i - 1] * 2  # 从第1个到第m-1个，方案都可以选择放/不放
    elif i == m: #第m个要小心了
        DP[i] = DP[i - 1] * 2 - 1
    else:#i>m
        DP[i] = DP[i - 1] * 2 - DP[i - m - 1]
print(DP[n])
```

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def dfs(i, j, n, m):
    if j == m:
        return 0  # 如果有连续的m个坑都有物质，此方案不可行
    if i == n:
        return 1  # 如果能到n，说明之前没有连续的m个坑都有物质，此方案可行

    # 不在第i个坑放置物质
    no_place = dfs(i + 1, 0, n, m)
```

```
    # 在第i个坑放置物质
    place = dfs(i + 1, j + 1, n, m)

    # 计算总数
    return no_place + place

if __name__ == "__main__":
    n, m = map(int, input().split())
    result = dfs(0, 0, n, m)
    print(result)
```

```
n = 0
m = 0
ans = 0
memo = {}

def dfs(i, j):
    if j == m:
        return 0  # 如果有连续的m个坑都有物质，此方案不可行
    if i == n:
        return 1  # 如果能到n，说明之前没有连续的m个坑都有物质，此方案可行
    if (i, j) in memo:
        return memo[(i, j)]
    ans = dfs(i+1, 0)  # 第i个坑里没有物质，之后的坑里是否放物质与前面没有联系了
    ans += dfs(i+1, j+1)  # 前i个坑中最后连续j个坑里都有物质
    memo[(i, j)] = ans
    return ans

if __name__ == "__main__":
    res = 0
    n, m = map(int, input().split())
    res = dfs(0, 0)  # 从第0个坑里开始放
    print(res)
```

```

```

```
1
```