

2025/10/14 Week6 贪心和矩阵

Updated 2025-10-14 14:43 GMT+8

Compiled by Hongfei Yan (2024 Spring)

Log:

2024/10/18 增加了3.2.1 two pointers, 3.2.2 binary search的讲解。其中二分查找内容移自前一讲 week5_dawn.md

2024/10/15 复制自 20231017_notes.md, 根据本学期进度进行修改。

10月份，主要是掌握矩阵、贪心题目，最好能涉及到简单的DP题目。时间复杂度理解 $O(1)$, $O(n)$, $O(\log n)$ 。

一、Greedy

贪心算法是用来解决一类最优化问题，并希望由局部最优策略来推得全局最优结果。贪心法适用的问题一定满足最优子结构性质，即一个问题的最优解可以通过其子问题的最优解来构建。

严谨使用贪心法来求解最优问题需要对采取的策略进行证明。证明往往比贪心本身更难，因此一般来说，如果想到某个似乎可行的策略，并且自己无法举出反例，那么就编码实现尝试。

<https://oi-wiki.org/basic/greedy/>

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如 $n=1$ ）的最优解 F_1 ，然后再证明：对于每个 n ， F_{n+1} 都可以由 F_n 推导出结果。

常见题型

在提高组难度以下的题目中，最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

1526C1. Potions (Easy Version)

greedy, dp, data structures, brute force, *1500, <https://codeforces.com/problemset/problem/152/6/C1>

与动态规划的区别

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

Introduction_to_algorithms-3rd Edition.pdf * - Foxit PDF Editor

Home Convert Edit Organize Comment View Form Protect Share Accessibility Help

Find

Bookmarks

- Contents
- Preface
- I Foundations
- II Sorting and Order Statistics
- III Data Structures
- IV Advanced Design and Anal...
- 15 Dynamic Programming
- 16 Greedy Algorithms**
- 17 Amortized Analysis
- V Advanced Data Structures
- VI Graph Algorithms
- VII Selected Topics
- VIII Appendix: Mathematical ...
- A Summations
- B Sets,Etc.
- C Counting and Probability
- D Matrices
- Bibliography
- Index

Algorithms for **optimization problems** typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it **makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution**. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly **Section 15.3**.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the **theory underlying combinatorial structures called “matroids,” for which a greedy algorithm always produces an optimal solution**. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra’s algorithm for shortest paths from a single source (Chapter 24), and Chvátal’s greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read

优化问题的算法通常会经历一系列步骤，每一步都有若干选择。对于许多优化问题，使用动态规划来确定最佳选择是过度的；更简单、更高效的贪心算法就足够了。贪心算法在每一步总是做出当前看起来最优的选择。也就是说，它做出局部最优选择，希望这种选择最终能导致全局最优解。本章探讨那些贪心算法能够提供最优解的优化问题。在阅读本章之前，你应该先阅读第15章关于动态规划的内容，特别是**第15.3节**。

贪心算法并不总能得到最优解，但对于许多问题，它们确实可以得到最优解。我们将在第16.1节首先考察一个简单但非平凡的问题——活动选择问题，该问题可以通过贪心算法高效地计算出最优解。我们将通过首先考虑动态规划方法，然后证明我们总是可以做出贪心选择以达到最优解，从而得出贪心算法。第16.2节回顾贪心方法的基本要素，并给出一种直接证明贪心算法正确性的方法。第16.3节介绍贪心技术的一个重要应用：设计数据压缩（哈夫曼）编码。在第16.4节中，我们将研究一些称为“拟阵”（matroids）的组合结构背后的理论，这些结构对于贪心算法总是能产生最优解。最后，第16.5节将拟阵应用于解决带截止时间和惩罚的单位时间任务调度问题。

1.1 Sortings

排序可以按照greedy来理解，因为都有不同的优化策略。

Python十大排序算法源码，https://github.com/GMyhf/2024spring-cs201/blob/main/code/ten_sort_algorithms.md



Generative AI is experimental. Info quality may vary. ::

Yes, bubble sort is considered a greedy algorithm. A greedy algorithm makes the best possible choice at each step in the hope of finding a global optimum. 

Bubble sort is a simple sorting algorithm that compares each pair of adjacent elements. It optimizes the sortedness of adjacent pairs of elements. It needs to iterate over the list several times to reach a global optimum. 

Other sorting algorithms that are considered greedy include: Selection sort, Insertion sort. 

Other examples of greedy algorithms include: 

- Kruskal's algorithm
- Prim's algorithm
- The algorithm for finding optimum Huffman trees

Prim算法和Kruskal算法主要用于解决无向图中的最小生成树（Minimum Spanning Tree, MST）问题。最小生成树是指在一个加权无向图中找到一棵包含所有顶点的树，且这棵树的所有边的权重之和最小。

- **Prim算法**: 从任意一个顶点开始构建最小生成树，逐步将距离当前树最近的一个顶点加入到树中，直到所有顶点都被包含进来。该算法适用于边数较多的稠密图。
- **Kruskal算法**: 首先将所有的边按照权重从小到大排序，然后依次选取权重最小的边，只要这条边不会与已选择的边构成回路，就将其加入到最小生成树中，直到选择了 $n-1$ 条边（ n 为顶点数）。此算法对稀疏图较为适用。

这两种算法都能有效地找出无向图的最小生成树，但在处理有向图时则需要转换成其他形式的问题或者使用不同的算法来求解。

<https://stackoverflow.com/questions/47238823/why-selection-sort-is-not-greedy>



A selection sort could indeed be described as a greedy algorithm, in the sense that it:

9



- tries to choose an output (a permutation of its inputs) that optimizes a certain measure ("sortedness", which could be measured in various ways, e.g. by number of inversions), and
- does so by breaking the task into smaller subproblems (for selection sort, finding the k -th element in the output permutation) and picking the locally optimal solution to each subproblem.



As it happens, the same description could be applied to most other sorting algorithms, as well — the only real difference is the choice of subproblems. For example:

- insertion sort locally optimizes the sortedness of the permutation of k first input elements;
- bubble sort optimizes the sortedness of adjacent pairs of elements; it needs to iterate over the list several times to reach a global optimum, but this still falls within the broad definition of a greedy algorithm;
- merge sort optimizes the sortedness of exponentially growing subsequences of the input sequence;
- quicksort recursively divides its input into subsequences on either side of an arbitrarily chosen pivot, optimizing the division to maximize sortedness at each stage.

Indeed, off the top of my head, I can't think of any practical sorting algorithm that *wouldn't* be greedy in this sense. (Bogosort isn't, but can hardly be called practical.) Furthermore, formulating these sorting algorithms as greedy optimization problems like this rather obscures the details that actually matter in practice when comparing sorting algorithms.

Thus, I'd say that characterizing selection sort, or any other sorting algorithm, as greedy is technically valid but practically useless, since such classification provides no real useful information.

Share Improve this answer Follow

answered Nov 11, 2017 at 15:02



Ilmari Karonen

49.3k ● 9 ● 93 ● 153

1.2 双指针和二分查找

双指针和二分查找是贪心算法中常用的技巧。常规贪心题目，例如：

1.2.1 two pointers

参考《算法笔记.胡凡》4.6

two pointers 是算法编程中一种非常重要的思想，但是很少会有教材单独拿出来讲，其中一个原因是它更倾向于是一种编程技巧，而长得不太像是一个“算法”的模样。two pointers的思想十分简洁，但却提供了非常高的算法效率。

以一个例子引入：给定一个递增的正整数序列和一个正整数 M，求序列中的两个不同位置的数a和 b，使得它们的和恰好为 M，输出所有满足条件的方案。例如给定序列{1,2,3,4,5,6}和正整数M=8，就存在 $2+6=8$ 与 $3+5=8$ 成立。

本题的一个最直观的想法是，使用二重循环枚举序列中的整数a和 b，判断它们的和是否为 M，如果是，输出方案；如果不是，则继续枚举。代码如下：

```
1 n = int(input())
2 a = list(map(int, input().split()))
3 M = int(input())
4
5 for i in range(n):
6     for j in range(i + 1, n):
7         if a[i] + a[j] == M:
8             print(a[i], a[j])
9 """
10 5
11 1 2 3 4 5
12 7
13
14 2 5
15 3 4
16 """
```

显然，这种做法的时间复杂度为 $O(n^2)$ ，对n在 10^5 的规模时是不可承受的。

two pointers 将利用有序序列的枚举特性来有效降低复杂度。它针对本题的算法过程是：令下标i的初值为 0，下标j的初值为n-1，即令i、j分别指向序列的第一个元素和最后一个元素，接下来根据 $a[i] + a[j]$ 与M的大小来进行下面三种选择，使i不断向右移动、使j不断向左移动，直到 $i > j$ 成立。

① 如果满足 $a[i] + a[j] == M$ ，说明找到了其中一组方案。由于序列递增，不等式 $a[i+1]+a[j]>M$ 与 $a[i] + a[j-1] < M$ 均成立，但是 $a[i+1]+a[j-1]$ 与M的大小未知，因此剩余的方案只可能在 $[i+1,j-1]$ 区间内产生，令 $i=i+1$ 、 $j=j-1$ （即令i向右移动，j向左移动）。

② 如果满足 $a[i] + a[j] > M$ ，由于序列递增，不等式 $a[i+1]+ a[j]>M$ 成立，但是 $a[i]+ a[j-1]$ 与M的大小未知，因此剩余的方案只可能在 $[i,j-1]$ 区间内产生，令 $j=j-1$ （即令j向左移动）。

③ 如果满足 $a[i]+a[j] < M$ ，由于序列递增，不等式 $a[i]+ a[j-1]< M$ 成立，但是 $a[i+1]+a[j]$ 与M的大小未知，因此剩余的方案只可能在 $[i+1,j]$ 区间内产生，令 $i=i+1$ （即令i向右移动）。

反复执行上面三个判断，直到 $i > j$ 成立。代码如下：

```
1 n = int(input())
2 a = list(map(int, input().split()))
3 M = int(input())
4
5 i = 0
6 j = n - 1
7
8 while i < j:
9     if a[i] + a[j] == M:
10        print(a[i], a[j])
11        i += 1
12        j -= 1
```

```

13     elif a[i] + a[j] < M:
14         i += 1
15     else:
16         j -= 1
17
18 """
19 5
20 1 2 3 4 5
21 7
22
23 2 5
24 3 4
25 """

```

分析算法的复杂度，由于i的初值为0，j的初值为n-1，而程序中变量i只有递增操作、变量j只有递减操作，且循环当i>j时停止，因此i和j的操作次数最多为n次，时间复杂度为O(n)。可以发现，two pointers的思想充分利用了序列递增的性质，以很浅显的思想降低了复杂度。

再来看序列合并问题。假设有两个递增序列A与B，要求将它们合并为一个递增序列C。同样的，可以设置两个下标i和j，初值均为0，表示分别指向序列A的第一个元素和序列B的第一个元素，然后根据A[i]与B[j]的大小来决定哪一个放入序列C。

- ① 若 A[i] < B[j]，说明 A[i]是当前序列A与序列B的剩余元素中最小的那个，因此把A[i]加入序列C中，并让i加1（即让i右移一位）。
- ② 若 A[i] > B[j]，说明 B[j]是当前序列A与序列B的剩余元素中最小的那个，因此把B[j]加入序列C中，并让j加1（即让j右移一位）。
- ③ 若 A[i] == B[j]，则任意选一个加入到序列C中，并让对应的下标加1。上面的分支操作直到i、j中的一个到达序列末端为止，然后将另一个序列的所有元素依次加入序列C中，代码如下：

```

1 def merge(A, B):
2     i, j = 0, 0
3     c = []
4
5     # 合并两个有序数组
6     while i < len(A) and j < len(B):
7         if A[i] <= B[j]:
8             c.append(A[i])
9             i += 1
10        else:
11            c.append(B[j])
12            j += 1
13
14     # 将 A 的剩余元素加入 c
15     c.extend(A[i:])
16
17     # 将 B 的剩余元素加入 c
18     c.extend(B[j:])
19

```

```
20     return len(c), c
21
22 # 示例
23 A = [1, 3, 5, 7]
24 B = [2, 4, 6, 8]
25
26 length, c = merge(A, B)
27 print(c)
```

two pointers 到底是怎样的一种思想？事实上，two pointers 最原始的含义就是针对本节第一个问题而言的，而广义上的 two pointers 则是利用问题本身与序列的特性，使用两个下标i、j对序列进行扫描（可以同向扫描，也可以反向扫描），以较低的复杂度（一般是 $O(n)$ 的复杂度）解决问题。在实际编程时要能够有使用这种思想的意识。

1.2.2 Binary Search

查找操作是编程中的基本技能，根据数据集的大小和结构选择合适的查找方法可以显著提高效率。线性查找适用于较小或无序的数据集，而二分查找适用于较大的有序数据集。

我发现二分查找容易理解，但是细节部分不容易写对（while的条件是 \leq ，还是 $<$ ；折半后是 $mid+1$ ， $mid-1$ ，还是 mid ）。

常见的查找方法

1. 线性查找 (Linear Search) :

- 适用范围：适用于较小的数据集或无序的数据集。
- 原理：逐个检查数据集中的每个元素，直到找到满足条件的元素或遍历完所有元素。
- 时间复杂度： $O(n)$ ，其中 n 是数据集的大小。

2. 二分查找 (Binary Search) :

- 适用范围：适用于有序的数据集。
- 原理：通过将数据集分成两半，逐步缩小查找范围，直到找到满足条件的元素或确定不存在。
- 时间复杂度： $O(\log n)$ ，其中 n 是数据集的大小。

示例代码

线性查找

```

1 def linear_search(arr, target):
2     for i, element in enumerate(arr):
3         if element == target:
4             return i # 返回目标元素的索引
5     return -1 # 如果未找到目标元素, 返回 -1
6
7 # 示例
8 arr = [3, 5, 2, 8, 1, 9, 4]
9 target = 8
10 result = linear_search(arr, target)
11 print(f"Target {target} found at index {result}")
12 # Target 8 found at index 3

```

二分查找

```

1 def binary_search(arr, target):
2     left, right = 0, len(arr) - 1
3
4     while left <= right:
5         mid = (left + right) // 2
6         if arr[mid] == target:
7             return mid # 返回目标元素的索引
8         elif arr[mid] < target:
9             left = mid + 1
10        else:
11            right = mid - 1
12
13    return -1 # 如果未找到目标元素, 返回 -1
14
15 # 示例
16 arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
17 target = 8
18 result = binary_search(arr, target)
19 print(f"Target {target} found at index {result}")
20 # Target 8 found at index 7

```

详细步骤

线性查找

1. 初始化:
 - 遍历数据集中的每个元素。
2. 查找过程:
 - 逐个检查每个元素是否等于目标元素。
 - 如果找到目标元素, 返回其索引。
 - 如果遍历完所有元素仍未找到目标元素, 返回 -1。

二分查找

1. 初始化：

- 设置左边界 `left` 为 0，右边界 `right` 为数据集的最后一个索引。

2. 查找过程：

- 计算中间位置 `mid`。
- 如果中间位置的元素等于目标元素，返回其索引。
- 如果中间位置的元素小于目标元素，调整左边界 `left` 为 `mid + 1`。
- 如果中间位置的元素大于目标元素，调整右边界 `right` 为 `mid - 1`。
- 重复上述步骤，直到找到目标元素或左边界超过右边界。

3. 未找到目标元素：

- 如果左边界超过右边界，返回 -1。

参考 `bisect` 源码的二分查找写法，

<https://github.com/python/cpython/blob/main/Lib/bisect.py>

当然可以！下面是一个基于 `bisect_left` 函数的实现，并提供一个详细的二分查找样例。

二分查找实现

```
1 def bisect_left(a, x, lo=0, hi=None, *, key=None):
2     """Return the index where to insert item x in list a, assuming a is sorted.
3
4     The return value i is such that all e in a[:i] have e < x, and all e in
5     a[i:] have e >= x. So if x already appears in the list, a.insert(i, x) will
6     insert just before the leftmost x already there.
7
8     Optional args lo (default 0) and hi (default len(a)) bound the
9     slice of a to be searched.
10
11    A custom key function can be supplied to customize the sort order.
12    """
13
14    if lo < 0:
15        raise ValueError('lo must be non-negative')
16    if hi is None:
17        hi = len(a)
18    # Note, the comparison uses "<" to match the
19    # __lt__() logic in list.sort() and in heapq.
20    if key is None:
21        while lo < hi:
22            mid = (lo + hi) // 2
23            if a[mid] < x:
24                lo = mid + 1
25            else:
26                hi = mid
```

```

27     else:
28         while lo < hi:
29             mid = (lo + hi) // 2
30             if key(a[mid]) < x:
31                 lo = mid + 1
32             else:
33                 hi = mid
34     return lo
35
36 # 二分查找函数
37 def binary_search(arr, target):
38     index = bisect_left(arr, target)
39     if index != len(arr) and arr[index] == target:
40         return index # 返回目标值的索引
41     else:
42         return -1 # 如果未找到目标值, 返回 -1
43
44 # 示例
45 arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
46 target = 8
47 result = binary_search(arr, target)
48 print(f"Target {target} found at index {result}")

```

详细步骤

1. 定义 `bisect_left` 函数:

- 参数:

- `a`: 已排序的列表。
- `x`: 要查找的目标值。
- `lo`: 搜索范围的起始索引, 默认为 0。
- `hi`: 搜索范围的结束索引, 默认为 `len(a)`。
- `key`: 可选的键函数, 用于自定义排序顺序。

- 逻辑:

- 检查 `lo` 是否非负。
- 如果 `hi` 为 `None`, 则设置 `hi` 为 `len(a)`。
- 使用二分查找算法找到目标值 `x` 应该插入的位置。
- 如果 `key` 为 `None`, 直接比较 `a[mid]` 和 `x`。
- 如果 `key` 不为 `None`, 比较 `key(a[mid])` 和 `x`。

2. 定义 `binary_search` 函数:

- 使用 `bisect_left` 找到目标值在已排序列表中第一次出现的位置。

- 检查目标值是否存在于列表中:

- 如果 `index` 不等于列表的长度且 `arr[index]` 等于目标值, 返回 `index`。
- 否则, 返回 -1。

自定义键函数示例

假设你有一个包含元组的列表，并且你希望根据元组的第二个元素进行二分查找：

```
1 def binary_search_with_key(arr, target, key):
2     index = bisect_left(arr, target, key=key)
3     if index != len(arr) and key(arr[index]) == target:
4         return index # 返回目标值的索引
5     else:
6         return -1 # 如果未找到目标值，返回 -1
7
8 # 示例
9 arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
10 target = 'c'
11 result = binary_search_with_key(arr, target, key=lambda x: x[1])
12 print(f"Target {target} found at index {result}")
```

- 输入：

```
1 arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
2 target = 'c'
```

- 输出：

```
1 Target c found at index 2
```

总结

二分查找是一种高效的查找算法，适用于已排序的数据集。你可以使用 `bisect` 模块中的 `bisect_left` 函数来快速实现二分查找，也可以手动实现以学习算法的细节。

进一步优化

如果你的 `key` 函数比较复杂，可以考虑使用 `functools.cmp_to_key` 来定义一个比较函数。这样可以更灵活地处理复杂的比较逻辑。

使用 `functools.cmp_to_key` 的示例

Python 的 `bisect.bisect_left` 函数从 Python 3.10 版本开始才支持 `key` 参数。

```
1 from bisect import bisect_left
2 from functools import cmp_to_key
3
4 def compare_items(x, y):
5     return (x[1] > y[1]) - (x[1] < y[1])
6
7 def binary_search_with_key(arr, target, key):
8     # 找到目标值应该插入的位置
```

```

9     index = bisect_left(arr, target, key=cmp_to_key(key))
10
11    # 检查是否找到了目标值
12    if index < len(arr) and key(arr[index], (0, target)) == 0:
13        return index # 返回目标值的索引
14    else:
15        return -1 # 如果未找到目标值, 返回 -1
16
17    # 示例
18 arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
19 target = 'c'
20 result = binary_search_with_key(arr, target, key=compare_items)
21 print(f"Target {target} found at index {result}")

```

详细解释

1. `compare_items` 函数:

- 定义一个比较函数 `compare_items`, 用于比较两个元组的第二个元素。

2. `cmp_to_key` 函数:

- 将 `compare_items` 转换为 `key` 函数, 传递给 `bisect_left`。

3. `if index < len(arr) and key(arr[index], (0, target)) == 0 :`

- 使用 `key` 函数比较 `arr[index]` 和 `(0, target)`, 确保它们的第二个元素相等。

1.3 编程题目

在问题求解时, 总是做出在当前看来是最好的选择, 不从整体最优上考虑。贪心算法没有固定的算法框架, 关键是贪心策略的选择, 贪心策略使用的前提是局部最优能导致全局最优。

练习CF1221A. 2048 Game

brute force/greedy/math, 1000, <http://codeforces.com/problemset/problem/1221/A>

You are playing a variation of game 2048. Initially you have a multiset s of n integers. Every integer in this multiset is a power of two.

You may perform any number (possibly, zero) operations with this multiset.

During each operation you choose two **equal** integers from s , remove them from s and insert the number equal to their sum into s .

For example, if $s=\{1,2,1,1,4,2,2\}$ and you choose integers 2 and 2, then the multiset becomes $\{1,1,1,4,4,2\}$.

You win if the number 2048 belongs to your multiset. For example, if $s=\{1024,512,512,4\}$ you can win as follows: choose 512 and 512, your multiset turns into $\{1024,1024,4\}$. Then choose 1024 and 1024, your multiset turns into $\{2048,4\}$ and you win.

You have to determine if you can win this game.

You have to answer q independent queries.

练习01017: 装箱问题

greedy, <http://cs101.openjudge.cn/practice/01017>

一个工厂制造的产品形状都是长方体，它们的高度都是 h ，长和宽都相等，一共有六个型号，他们的长宽分别为 $1*1, 2*2, 3*3, 4*4, 5*5, 6*6$ 。这些产品通常使用一个 $6*6*h$ 的长方体包裹包装然后邮寄给客户。因为邮费很贵，所以工厂要想方设法的减小每个订单运送时的包裹数量。他们很需要有一个好的程序帮他们解决这个问题从而节省费用。现在这个程序由你来设计。

输入： 输入文件包括几行，每一行代表一个订单。每个订单里的一行包括六个整数，中间用空格隔开，分别为11至66这六种产品的数量。输入文件将以6个0组成的一行结尾。

输出： 除了输入的最后一行6个0以外，输入文件里每一行对应着输出文件的一行，每一行输出一个整数代表对应的订单所需的最小包裹数。

解题思路： $4*4, 5*5, 6*6$ 这三种的处理方式较简单，就是每一个箱子至多只能有其中1个，根据他们的数量添加箱子，再用 $2*2$ 和 $1*1$ 填补。 $1*1, 2*2, 3*3$ 这些就需要额外分情况讨论，若有剩余的 $3*3$,每4个 $3*3$ 可以填满一个箱子，剩下的 $3*3$ 用 $2*2$ 和 $1*1$ 填补装箱。剩余的 $2*2$ ，每9个可以填满一个箱子，剩下的与 $1*1$ 一起装箱。最后每36个 $1*1$ 可以填满一个箱子，剩下的为一箱子。

样例输入

1	0 0 4 0 0 1
2	7 5 1 0 0 0
3	0 0 0 0 0 0

样例输出

1	2
2	1

来源：Central Europe 1996

直接用总数把bcdef占的位置都减掉就可以了，思路就清晰起来了。**运用列表，避免多个 if else。

```
1 import math
2 rest = [0,5,3,1]
3
4 while True:
5     a,b,c,d,e,f = map(int,input().split())
6     if a + b + c + d + e + f == 0:
7         break
8     boxes = d + e + f          #装4*4, 5*5, 6*6
9     boxes += math.ceil(c/4)      #填3*3
10    spaceforb = 5*d + rest[c%4] #能和4*4 3*3 一起放的2*2
11    if b > spaceforb:
```

```

12     boxes += math.ceil((b - spaceforb)/9)
13     spacefora = boxes*36 - (36*f + 25*e + 16*d + 9*c + 4*b)      #和其他箱子一起的填的
14
15     if a > spacefora:
16         boxes += math.ceil((a - spacefora)/36)
17     print(boxes)

```

练习12559: 最大最小整数 v0.3

greedy, strings, sortings, math <http://cs101.openjudge.cn/practice/12559>

假设有n个正整数，将它们连成一片，将会组成一个新的大整数。现需要求出，能组成的大整数。

比如，有4个正整数，23, 9, 182, 79，连成的最大整数是97923182，最小的整数是18223799。

输入

第一行包含一个整数n, $1 \leq n \leq 1000$ 。

第二行包含n个正整数，相邻正整数间以空格隔开。

输出

输出为一行，为这n个正整数能组成的最大的多位整数和最小的多位整数，中间用空格隔开。

样例输入

```

1 Sample1 in:
2 4
3 23 9 182 79
4
5 Sample1 out:
6 97923182 18223799

```

样例输出

```

1 Sample2 in:
2 2
3 11 113
4
5 Sample2 out:
6 11311 11113

```

提示

位数不同但前几位相同的时候。例如：898 8987，大整数是898+8987，而不是8987+898。

来源：cs10116 final exam

思路：先拼接出最小值：即字典序最小；要保证每一个小的字符串，左移到合适位置，需要两两比较（刚好是冒泡排序）。这个题目是个不容易的，字符串处理题目。

求minimum时，对相邻两strA[k]与A[k+1]，比较A[k]+A[k+1]与A[k+1]+A[k]的大小，若A[k+1]+A[k]大，颠倒A[k]与A[k+1]；最多交换len(A)-1次。求maximum时，颠倒求minimum时的有序序列即可。使用冒泡排序，循环(n-1)次。

把这些数当成字符串处理，然后采用类似冒泡排序的做法排出大小。

```
1 # O(n^2)
2 n = int(input())
3 nums = input().split()
4 for i in range(n - 1):
5     for j in range(i+1, n):
6         #print(i,j)
7         if nums[i] + nums[j] < nums[j] + nums[i]:
8             nums[i], nums[j] = nums[j], nums[i]
9
10 ans = ''.join(nums)
11 nums.reverse()
12 print(ans + " " + ''.join(nums))
```

2020fall-cs101，黄旭

思路：这道题的关键应该是找到排序的方式，前一个数和后一个数比较，如果位数不足，就要重新从第一位开始比，所以说我就先取这个数列的最大位数，然后把每个数都扩充到相同位数进行比较，就可以了。

```
1 # 虽然能AC，但实际上不对。两倍长度是正确的。
2 from math import ceil
3 input()
4 lt = input().split()
5
6 max_len = len(max(lt, key = lambda x:len(x)))
7 lt.sort(key = lambda x: tuple([int(i) for i in x]) * ceil(max_len/len(x)))
8 lt1 = lt[::-1]
9 print(''.join(lt1),''.join(lt))
```

```

1 # 两倍长度是正确的。O(nlogn)
2 from math import ceil
3 input()
4 lt = input().split()
5
6 max_len = len(max(lt, key = lambda x:len(x)))
7 lt.sort(key = lambda x: x * ceil(2*max_len/len(x)))
8 lt1 = lt[::-1]
9 print(''.join(lt1),''.join(lt))

```

练习19948: 因材施教

greedy, <http://cs101.openjudge.cn/practice/19948>

有一所魔法高校招入一批学生，为了贯彻因材施教的理念，学校打算根据他们的魔法等级进行分班教育。在确定班级数目的情况下，班级内学生的差异要尽可能的小，也就是各个班级内学生的魔法等级要尽可能的接近。例如：现在有($n = 7$)位学生，他们的魔法等级分别为($r = [2, 7, 9, 9, 16, 28, 45]$)，我们要将他们分配到($m = 3$)个班级，如果按照([2, 7], [9, 9], [16, 28, 45])的方式分班，则他们的总体差异为($d = (7 - 2) + (9 - 9) + (45 - 16) = 34$)。

练习1000B. Light It Up

greedy, 1500, <https://codeforces.com/problemset/problem/1000/B>

Recently, you bought a brand new smart lamp with programming features. At first, you set up a schedule to the lamp. Every day it will turn power on at moment 0 and turn power off at moment M. Moreover, the lamp allows you to set a program of switching its state (states are "lights on" and "lights off"). Unfortunately, some program is already installed into the lamp.

The lamp allows only *good* programs. Good program can be represented as a non-empty array a , where $0 < a_1 < a_2 < \dots < a_{|a|} < M$. All a_i must be integers. Of course, preinstalled program is a good program.

The lamp follows program a in next manner: at moment 0 turns power and light on. Then at moment a_i the lamp flips its state to opposite (if it was lit, it turns off, and vice versa). The state of the lamp flips instantly: for example, if you turn the light off at moment 1 and then do nothing, the total time when the lamp is lit will be 1. Finally, at moment M the lamp is turning its power off regardless of its state.

Since you are not among those people who read instructions, and you don't understand the language it's written in, you realize (after some testing) the only possible way to alter the preinstalled program. You can **insert at most one** element into the program a , so it still should be a *good* program after alteration. Insertion can be done between any pair of consecutive elements of a , or even at the beginning or at the end of a .

Find such a way to alter the program that the total time when the lamp is lit is maximum possible. Maybe you should leave program untouched. If the lamp is lit from x till moment y , then its lit for $y-x$ units of time. Segments of time when the lamp is lit are summed up.

练习18211: 军备竞赛

greedy/two pointers, <http://cs101.openjudge.cn/practice/18211>

鸣人是木叶村的村长，最近在跟敌国进行军备竞赛，他手边有N份武器设计图，每张设计图有制作成本（大于等于零）且最多使用一次，可以选择花钱制作或是以同样的价钱卖给敌国，同时任意时刻敌国的武器不能比我国更多，鸣人的目标是在不负债的前提下武器种类比敌国越多越好。

输入

第一行为起始整数经费p,并且 $0 \leq p$ 。且要求任何时刻p不能小于0.

第二行为n个整数，以空格分隔，并且 $0 \leq$ 每个整数。代表每张设计图的制作成本，同时也是卖价，最多用一次(无法又制作又卖).

输出

一个整数，代表武器种类最多比敌国多多少.

练习CF1364A. XXXXX

brute force/data structures/number theory/two pointers, 1200, <https://codeforces.com/problemset/problem/1364/A>

Ehab loves number theory, but for some reason he hates the number x . Given an array a , find the length of its longest subarray such that the sum of its elements **isn't** divisible by x , or determine that such subarray doesn't exist.

An array a is a subarray of an array b if a can be obtained from b by deletion of several (possibly, zero or all) elements from the beginning and several (possibly, zero or all) elements from the end.

Input

The first line contains an integer t ($1 \leq t \leq 5$) — the number of test cases you need to solve. The description of the test cases follows.

The first line of each test case contains 2 integers n and x ($1 \leq n \leq 10^5$, $1 \leq x \leq 10^4$) — the number of elements in the array a and the number that Ehab hates.

The second line contains n space-separated integers a_1, a_2, \dots, a_n ($0 \leq a_i \leq 10^4$) — the elements of the array a .

Output

For each testcase, print the length of the longest subarray whose sum isn't divisible by x . If there's no such subarray, print -1 .

Example

input

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

output

```
1 | 2
2 | 3
3 | -1
```

Note

In the first test case, the subarray 2, 3 has sum of elements 5, which isn't divisible by 3.

In the second test case, the sum of elements of the whole array is 6, which isn't divisible by 4.

In the third test case, all subarrays have an even sum, so the answer is -1.



```

1 # 查达闻
2 def r(i):return int(i)%b
3 for z in range(int(input())):
4     a,b=map(int,input().split());a=list(map(r,input().split()))
5     if sum(a)%b:print(len(a))
6     else:
7         n=1
8         for i in range(len(a)):
9             if a[i]or a[~i]:print(len(a)-i-1);n=0;break
10        if n:print(-1)

```

2023fall-cs101: Algo DS(325)



查达闻

The screenshot shows a Codeforces submission page. The submission ID is 226703706, made by user 'AgWOW' (张达闻) on Oct 05, 2023, at 16:09:25 UTC. The problem is '1356A - 15' (Practice). The code was accepted with a time of 218 ms and memory usage of 13184 kB. The verdict is 'Accepted'. The language used is Python 3.

Oct 5, 2023 23:25



数院胡睿诚

(但是我感觉很难看懂 😅)

"23-花园-何雨轩" 拍了拍 "数院胡睿诚"

头大，都是1个字母的

查达闻: def r(i):return int(i)%b for z in
range(int(input())): a,b=map(int,input().split())... >

```

1 def prefix_sum(nums):
2     prefix = []
3     total = 0
4     for num in nums:
5         total += num

```

```

6         prefix.append(total)
7     return prefix
8
9 def suffix_sum(nums):
10    suffix = []
11    total = 0
12    # 首先将列表反转
13    reversed_nums = nums[::-1]
14    for num in reversed_nums:
15        total += num
16        suffix.append(total)
17    # 将结果反转回来
18    suffix.reverse()
19    return suffix
20
21
22 t = int(input())
23 for _ in range(t):
24     N, x = map(int, input().split())
25     a = [int(i) for i in input().split()]
26     aprefix_sum = prefix_sum(a)
27     asuffix_sum = suffix_sum(a)
28
29     left = 0
30     right = N - 1
31     if right == 0:
32         if a[0] % x != 0:
33             print(1)
34         else:
35             print(-1)
36         continue
37
38     leftmax = 0
39     rightmax = 0
40     while left != right:
41         total = asuffix_sum[left]
42         if total % x != 0:
43             leftmax = right - left + 1
44             break
45         else:
46             left += 1
47
48     left = 0
49     right = N - 1
50     while left != right:
51         total = aprefix_sum[right]
52         if total % x != 0:
53             rightmax = right - left + 1
54             break
55         else:
56             right -= 1
57

```

```
58     if leftmax == 0 and rightmax == 0:  
59         print(-1)  
60     else:  
61         print(max(leftmax, rightmax))
```

二、Matrices 矩阵

在学习编程的过程中，经常遇到输入的数据是矩阵的形式，所以我们首先来明确矩阵的概念。

2.1 知识点：矩阵

这段矩阵知识点的讲解，借鉴自《数学要素》的1.4和1.5节，作者：姜伟生，2023-06-01出版。

万物皆数。

All is Number.

——毕达哥拉斯(Pythagoras) | 古希腊

哲学家、数学家 | 570 B.C.—495 B.C.

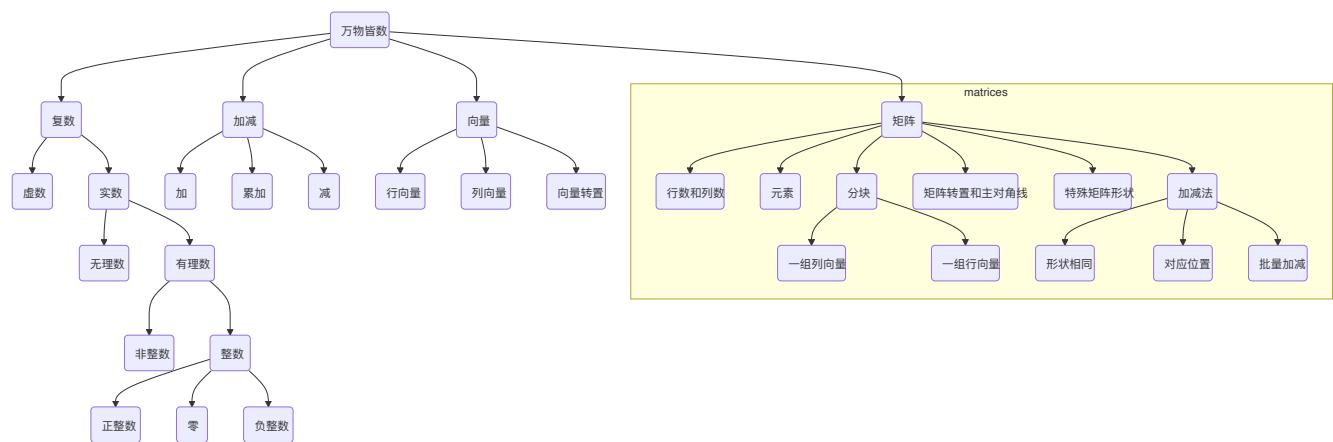


图1 数的结构

2.1.1 向量：数字排成行、列

向量、矩阵等线性代数概念对于数据科学和机器学习至关重要。在机器学习中，数据几乎都以矩阵形式存储、运算。毫不夸张地说，没有线性代数就没有现代计算机运算。逐渐地，大家会发现算数、代数、解析几何、微积分、概率统计、优化方法并不是一个个孤岛，而线性代数正是连接它们的重要桥梁之一。

行向量、列向量

若干数字排成一行或一列，并且用中括号括起来，得到的数组叫作向量(vector)。

排成一行的叫作行向量(row vector)，排成一列的叫作列向量(column vector)。

通俗地讲，行向量就是表格的一行数字，列向量就是表格的一列数字。以下两例分别展示了行向量和列向量，即

$$[1 \quad 2 \quad 3]_{1 \times 3}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}_{3 \times 1} \quad (1)$$

式(1)中，下角标“ 1×3 ”代表“1行、3列”，“ 3×1 ”代表“3行、1列”。

转置

转置符号为上标“T”。行向量转置(transpose)可得到列向量；同理，列向量转置可得到行向量。举例如下，有

$$[1 \ 2 \ 3]^T = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}^T = [1 \ 2 \ 3] \quad (2)$$

2.1.2 矩阵：数字排列成长方形

矩阵(matrix)将一系列数字以长方形方式排列，如

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2 \times 3}, \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}_{3 \times 2}, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{2 \times 2} \quad (3)$$

通俗地讲，矩阵将数字排列成表格，有行、有列。式(3)给出了三个矩阵，形状分别是2行3列（记作 2×3 ）、3行2列（记作 3×2 ）和2行2列（记作 2×2 ）。

通常用大写字母代表矩阵，比如矩阵A和矩阵B。

图2所示为一个 $n \times D$ 矩阵X。n是矩阵的行数(number of rows in the matrix)，D是矩阵的列数(number of columns in the matrix)。X可以展开写成表格形式，即

$$X_{n \times D} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,D} \end{bmatrix} \quad (4)$$

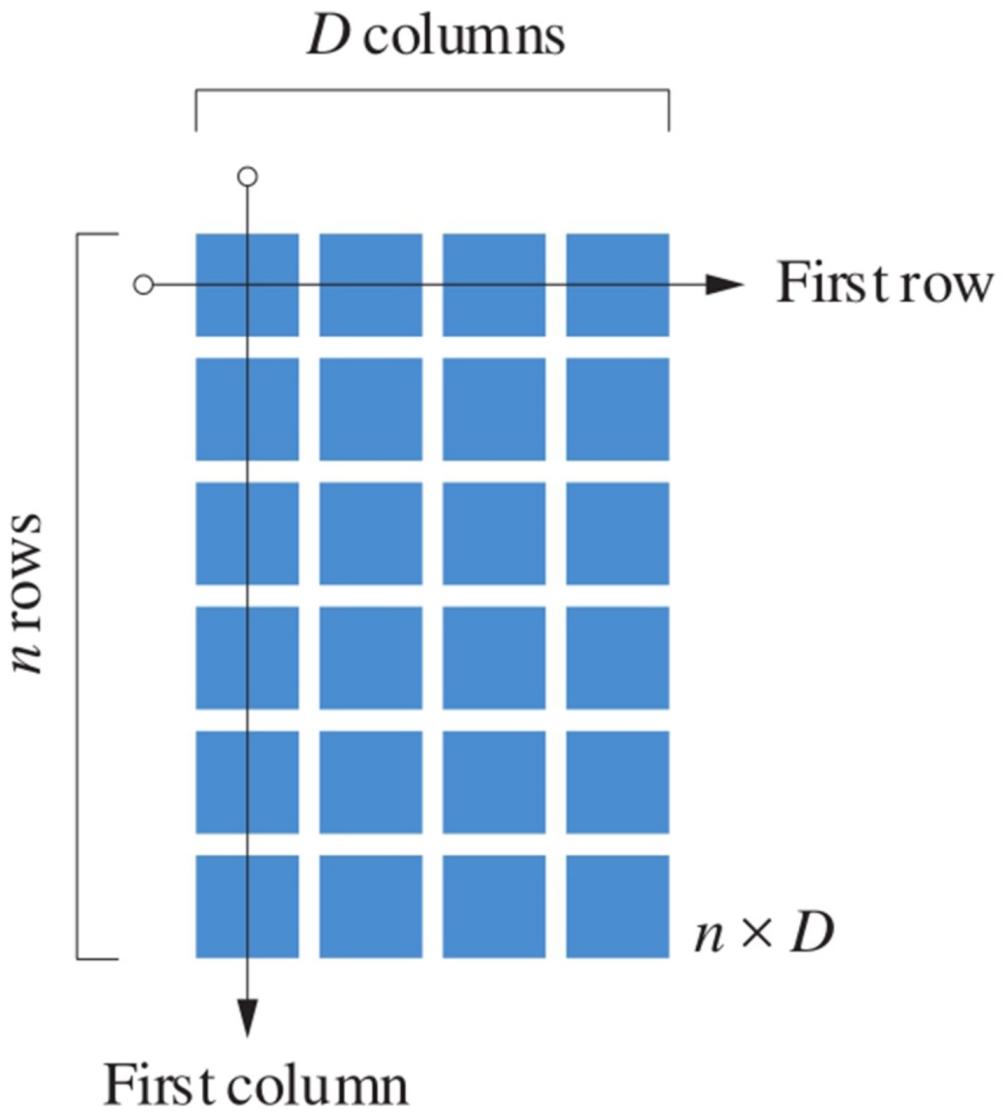


图2 $n \times D$ 矩阵X

再次强调：先说行序号，再说列序号。数据矩阵一般采用大写X表达。

矩阵X中，元素(element) $x_{i,j}$ 被称作i,j元素 (i j entry或i j element)，也可以说 $x_{i,j}$ 出现在i行j列(appears in row i and column j)。比如， $x_{n,1}$ 是矩阵X的第n行、第1列元素。

表1.4总结了如何用英文读矩阵和矩阵元素。

表1.4 矩阵有关英文表达

数学表达	英文表达
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	Two by two matrix, first row one two, second row three four
$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$	m by n matrix, first row a sub one one, a sub one two, dot dot dot, a sub one n , second row a sub two one, a sub two two, dot dot dot, a sub two n dot dot dot last row a sub m one, a sub m two, dot dot dot a sub m n
$a_{i,j}$	Lowercase (small) a sub i comma j
$a_{i,j+1}$	Lowercase a double subscript i comma j plus one
$a_{i,j-1}$	Lowercase a double subscript i comma j minus one

2.2 编程题目

2.2.1 矩阵乘法运算符

示例E18161: 矩阵运算(先乘再加)

matrices, <http://cs101.openjudge.cn/pctbook/E18161/>

现有三个矩阵A, B, C, 要求矩阵运算A·B+C并输出结果

矩阵运算介绍:

矩阵乘法运算必须要前一个矩阵的列数与后一个矩阵的行数相同,
如 m 行 n 列的矩阵A与 n 行 p 列的矩阵B相乘, 可以得到 m 行 p 列的矩阵C,
矩阵C的每个元素都由A的对应行中的元素与B的对应列中的元素一一相乘并求和得到,
即 $C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + \dots + A[i][n-1]*B[n-1][j]$

($C[i][j]$ 表示C矩阵中第*i*行第*j*列元素)。

矩阵的加法必须在两个矩阵行数列数均相等的情况下进行,
如 m 行 n 列的矩阵A与 m 行 n 列的矩阵B相加, 可以得到 m 行 n 列的矩阵C,
矩阵C的每个元素都由A与B对应位置的元素相加得到,
即 $C[i][j] = A[i][j] + B[i][j]$

输入

输入分为三部分, 分别是A,B,C三个矩阵的内容。

每一部分的第一行为两个整数, 代表矩阵的行数row和列数col

接下来row行, 每行有col个整数, 代表该矩阵这一行的每个元素

输出

如果可以完成矩阵计算，输出计算结果，与输入格式类似，不需要输出行数和列数信息。
如果不能完成矩阵计算，输出"Error!"

样例输入

```
1 Sample Input1:  
2 3 1  
3 0  
4 1  
5 0  
6 1 2  
7 1 1  
8 3 2  
9 3 1  
10 3 1  
11 3 1  
12  
13 Sample Output1:  
14 3 1  
15 4 2  
16 3 1
```

样例输出

```
1 Sample Input2:  
2 1 1  
3 0  
4 2 1  
5 1  
6 3  
7 1 1  
8 9  
9  
10 Sample Output2:  
11 Error!
```

提示

sample1 计算过程

$$|0| \quad |00|$$

$$|1| \cdot |11| = |11|$$

$$|0| \quad |00|$$

$$|00| \quad |31| \quad |31|$$

$$|11| + |31| = |42|$$

$$|00| \quad |31| \quad |31|$$

来源：cs101-2017 期末机考备选 & 2018 Mock Exam 2

思路：矩阵运算，如果没有学过可以百度下矩阵乘法（这是线性代数/高等代数的初步）

```
1 A,B,C = [],[],[]
2
3 a,b = map(int, input().split())
4 for i in range(a):
5     A.append(list(map(int, input().split())))
6
7 c,d = map(int, input().split())
8 for i in range(c):
9     B.append(list(map(int, input().split())))
10
11 e,f = map(int, input().split())
12 for i in range(e):
13     C.append(list(map(int, input().split())))
14
15 if b!=c or a!=e or d!=f:
16     print("Error!")
17 else:
18     D = [[0 for j in range(f)] for i in range(e)]
19     for i in range(e):
20         for j in range(f):
21             for k in range(b):
22                 D[i][j] += A[i][k] * B[k][j]
23                 D[i][j] += C[i][j]
24
25     for i in range(e):
26         print(' '.join([str(j) for j in D[i]]))
```

💡 运算符在 Python 中就是矩阵乘法（从 Python 3.5 开始引入 PEP 465），但它只在支持 `__matmul__` 的对象中有效，比如 `list` 自己封装的矩阵类。

⚠ 标准库里的 `list` 并没有定义 `__matmul__`，所以直接对普通嵌套列表写 `A @ B` 会报错。

办法就是自己封装一个矩阵类，实现 `__matmul__` 和 `__add__` 方法。

代码：

```
1 class Matrix:
2     def __init__(self, data):
3         self.data = data
4         self.rows = len(data)
5         self.cols = len(data[0]) if self.rows else 0
6
7     def __matmul__(self, other): # 定义 A @ B
8         if self.cols != other.rows:
9             raise ValueError("Matrix dimensions do not match for multiplication")
10        result = [[0] * other.cols for _ in range(self.rows)]
11        for i in range(self.rows):
12            for j in range(other.cols):
```

```

13         for k in range(self.cols):
14             result[i][j] += self.data[i][k] * other.data[k][j]
15     return Matrix(result)
16
17     def __add__(self, other): # 定义 A + B
18         if self.rows != other.rows or self.cols != other.cols:
19             raise ValueError("Matrix dimensions do not match for addition")
20     result = [
21         [self.data[i][j] + other.data[i][j] for j in range(self.cols)]
22         for i in range(self.rows)
23     ]
24     return Matrix(result)
25
26     def __str__(self): # 打印友好
27         return "\n".join(" ".join(map(str, row)) for row in self.data)
28
29
30 # === 读入 ===
31     def read_matrix():
32         r, c = map(int, input().split())
33         data = [list(map(int, input().split())) for _ in range(r)]
34         return Matrix(data)
35
36
37 A = read_matrix()
38 B = read_matrix()
39 C = read_matrix()
40
41 # === 计算 ===
42 try:
43     D = A @ B + C
44     print(D) # 自动调用 __str__
45 except ValueError:
46     print("Error!")

```

`A @ B` 调用 `__matmul__`，就是矩阵乘法

`(A @ B) + C` 调用 `__add__`，就是矩阵加法

如果维度不合法，抛 `ValueError`，就能捕获并输出 "Error!"

2.2.2 保护圈

示例12560: 生存游戏

matrices, <http://cs101.openjudge.cn/practice/12560/>

有如下生存游戏的规则：

给定一个 $n*m(1 \leq n, m \leq 100)$ 的数组，每个元素代表一个细胞，其初始状态为活着(1)或死去(0)。

每个细胞会与其相邻的8个邻居（除数组边缘的细胞）进行交互，并遵守如下规则：

任何一个活着的细胞如果只有小于2个活着的邻居，那它就会由于人口稀少死去。

任何一个活着的细胞如果有2个或者3个活着的邻居，就可以继续活下去。

任何一个活着的细胞如果有超过3个活着的邻居，那它就会由于人口拥挤而死去。

任何一个死去的细胞如果有恰好3个活着的邻居，那它就会由于繁殖而重新变成活着的状态。

请写一个函数用来计算所给定初始状态的细胞经过一次更新后的状态是什么。

注意：所有细胞的状态必须同时更新，不能使用更新后的状态作为其他细胞的邻居状态来进行计算。

输入

第一行为 n 和 m ，而后 n 行，每行 m 个元素，用空格隔开。

输出

n 行，每行 m 个元素，用空格隔开。

样例输入

1	3 4
2	0 0 1 1
3	1 1 0 0
4	1 1 0 1

样例输出

1	0 1 1 0
2	1 0 0 1
3	1 1 1 0

来源：cs10116 final exam

函数

```
1 # http://cs101.openjudge.cn/practice/12560/
2 def check(board, y, x):
3     c = board[y-1][x-1]+board[y-1][x]+board[y-1][x+1]+ \
4         board[y][x-1]+board[y][x+1]+ \
5         board[y+1][x-1]+board[y+1][x]+board[y+1][x+1]
6     if board[y][x] and (c<2 or c>3):
7         return 0
8     elif board[y][x]==0 and c==3:
9         return 1
10
11    return board[y][x]
12
13 n,m=[int(i) for i in input().split()]
14
15 board=[]
16 board.append( [0 for x in range(m+2)] ) →
17 for y in range(n):
18     board.append([0] +[int(x) for x in input().split()] + [0])
19
20 board.append( [0 for x in range(m+2)] )
21 →
22
23 bn = [[0]*m for y in range(n)]
24 for y in range(n):
25     for x in range(m):
26         bn[y][x] = check(board, y+1, x+1)
27
28 for y in range(n):
29     print(' '.join([str(x) for x in bn[y]])) ←
```

保护圈

' '.join(map(str, bn[y]))

加保护圈，八个邻居步长用dx,dy对表示。

```
1 dx = [-1, -1, -1, 0, 1, 1, 1, 0]
2 dy = [-1, 0, 1, 1, 1, 0, -1, -1]
3
4 def check(board, y, x):
5     c = 0
6     for i in range(8):
7         nx = x + dx[i]
8         ny = y + dy[i]
9         c += board[ny][nx]
10
11     if board[y][x] and (c<2 or c>3):
12         return 0
13     elif board[y][x]==0 and c==3:
14         return 1
15
16     return board[y][x]
17
18 n, m = map(int, input().split())
19
20 board=[]
21 board.append( [0 for x in range(m+2)] )
22 for _ in range(n):
```

```

23     board.append([0] +[int(_) for _ in input().split()] + [0])
24
25 board.append( [0 for _ in range(m+2)] )
26
27 # in place solver
28 bn = [[0]*m for y in range(n)]
29 for i in range(n):
30     for j in range(m):
31         bn[i][j] = check(board, i+1, j+1)
32
33 for row in bn:
34     print(*row)

```

练习508A. Pasha and Pixels

brute force, 1100, <http://codeforces.com/problemset/problem/508/A>

Pasha loves his phone and also putting his hair up... But the hair is now irrelevant.

Pasha has installed a new game to his phone. The goal of the game is following. There is a rectangular field consisting of n rows with m pixels in each row. Initially, all the pixels are colored white. In one move, Pasha can choose any pixel and color it black. In particular, he can choose the pixel that is already black, then after the boy's move the pixel does not change, that is, it remains black. Pasha loses the game when a 2×2 square consisting of black pixels is formed.

Pasha has made a plan of k moves, according to which he will paint pixels. Each turn in his plan is represented as a pair of numbers i and j , denoting respectively the row and the column of the pixel to be colored on the current move.

Determine whether Pasha loses if he acts in accordance with his plan, and if he does, on what move the 2×2 square consisting of black pixels is formed.

Input

The first line of the input contains three integers n, m, k ($1 \leq n, m \leq 1000, 1 \leq k \leq 10^5$) — the number of rows, the number of columns and the number of moves that Pasha is going to perform.

The next k lines contain Pasha's moves in the order he makes them. Each line contains two integers i and j ($1 \leq i \leq n, 1 \leq j \leq m$), representing the row number and column number of the pixel that was painted during a move.

Output

If Pasha loses, print the number of the move when the 2×2 square consisting of black pixels is formed.

If Pasha doesn't lose, that is, no 2×2 square consisting of black pixels is formed during the given k moves, print 0.

Examples

input

1	2	2	4
2	1	1	
3	1	2	
4	2	1	
5	2	2	

output

1	4
---	---

input

1	2	3	6
2	2	3	
3	2	2	
4	1	3	
5	2	2	
6	1	2	
7	1	1	

output

1	5
---	---

input

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

output

1	0
---	---

→ Source

```

n,m,k = map(int, input().split())
mx = [(m+2)*[0] for i in range(n+2)] 保护圈

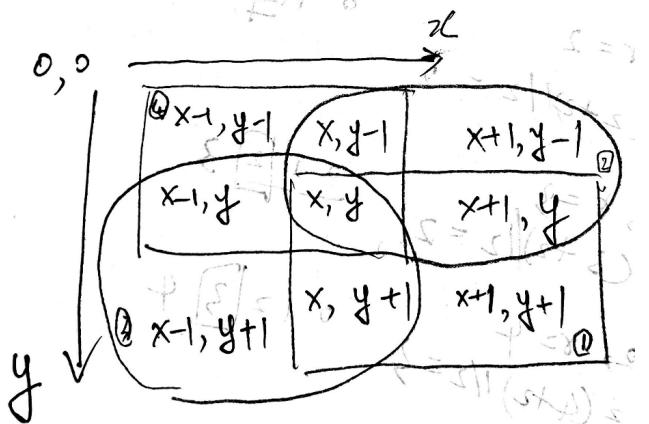
# if square 2 × 2 formed from black cells appears, and
# cell (i, j) will upper-left, upper-right, bottom-left
# or bottom-right of this squares.

def square_check(i, j):
    if mx[i][j+1] and mx[i+1][j] and mx[i+1][j+1]:
        return True
    if mx[i][j-1] and mx[i+1][j-1] and mx[i+1][j]:
        return True
    if mx[i-1][j] and mx[i-1][j+1] and mx[i][j+1]:
        return True
    if mx[i-1][j-1] and mx[i-1][j] and mx[i][j-1]:
        return True
    return False

for i in range(k):
    x,y = map(int, input().split())
    mx[x][y] = 1
    if square_check(x,y):
        print(i+1)
        break
else:
    print(0)

```

函数



练习加保护圈

```

1 # http://codeforces.com/contest/508/submission/44603553
2 n,m,k = map(int, input().split())
3 mx = [(m+2)*[0] for i in range(n+2)]
4
5 # if square 2 × 2 formed from black cells appears, and
6 # cell (i, j) will upper-left, upper-right, bottom-left
7 # or bottom-right of this squares.
8
9 def square_check(i,j):
10     if mx[i][j+1] and mx[i+1][j] and mx[i+1][j+1]:
11         return True
12     if mx[i][j-1] and mx[i+1][j-1] and mx[i+1][j]:
13         return True
14     if mx[i-1][j] and mx[i-1][j+1] and mx[i][j+1]:
15         return True
16     if mx[i-1][j-1] and mx[i-1][j] and mx[i][j-1]:
17         return True
18     return False
19
20 for i in range(k):
21     x,y = map(int, input().split())
22     mx[x][y] = 1
23     if square_check(x,y):
24         print(i+1)
25         break

```

```
26 |     else:  
27 |         print(0)
```

2.2.3 range中使用min、max

练习02659:Bomb Game

matrices, <http://cs101.openjudge.cn/practice/02659/>

Bosko and Susko are playing an interesting game on a board made of rectangular fields arranged in A rows and B columns.

When the game starts, Susko puts its virtual pillbox in one field one the board. Then Bosko selects fields on which he will throw his virtual bombs. After each bomb, Susko will tell Bosko whether his pillbox is in the range of this bomb or not.

The range of a bomb with diameter P (P is always odd), which is thrown in field (R, S), is a square area. The center of the square is in the field (R, S), and the side of the square is parallel to the sides of the board and with length P.

After some bombs have been thrown, Bosko should find out the position of Susko's pillbox. However, the position may be not unique, and your job is to help Bosko to calculate the number of possible positions.

输入

First line of input contains three integers: A, B and K, $1 \leq A, B, K \leq 100$. A represents the number of rows, B the number of columns and K the number of thrown bombs.

Each of the next K lines contains integers R, S, P and T, describing a bomb thrown in the field at R-th row and S-th column with diameter P, $1 \leq R \leq A$, $1 \leq S \leq B$, $1 \leq P \leq 99$, P is odd. If the pillbox is in the range of this bomb, T equals to 1; otherwise it is 0.

输出

Output the number of possible fields, which Susko's pillbox may stay in.

样例输入

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

样例输出

```
1 | 5
```

来源

Croatia OI 2002 National - Juniors

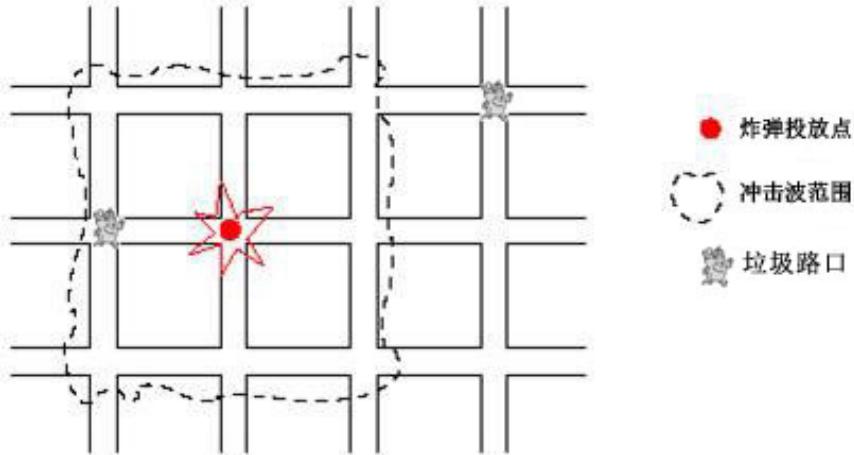
```
1 def max_count(matrix):
2     maximum = max(max(row) for row in matrix)
3     count = sum(row.count(maximum) for row in matrix)
4     return count
5
6 def calculate_possible_positions(A, B, K, bombs):
7     positions = [[0] * B for _ in range(A)]
8
9     for (R, S, P, T) in bombs:
10        for i in range(max(0, R - (P - 1) // 2), min(A, R + (P + 1) // 2)):
11            for j in range(max(0, S - (P - 1) // 2), min(B, S + (P + 1) // 2)):
12                if T == 1:
13                    positions[i][j] += 1
14
15                elif T == 0:
16                    positions[i][j] -= 1
17
18    #for row in positions:
19    #    print(row)
20    return max_count(positions)
21
22 A, B, K = map(int, input().split())
23 bombs = []
24 for _ in range(K):
25     R, S, P, T = map(int, input().split())
26     bombs.append((R - 1, S - 1, P, T))
27
28 result = calculate_possible_positions(A, B, K, bombs)
29 print(result)
```

练习04133:垃圾炸弹

matrices, <http://cs101.openjudge.cn/practice/04133/>

2018年俄罗斯世界杯（2018 FIFA World Cup）开踢啦！为了方便球迷观看比赛，莫斯科街道上很多路口都放置了的直播大屏幕，但是人群散去后总会在这些路口留下一堆垃圾。为此俄罗斯政府决定动用一种最新发明——“垃圾炸弹”。这种“炸弹”利用最先进的量子物理技术，爆炸后产生的冲击波可以完全清除波及范围内的所有垃圾，并且不会产生任何其他不良影响。炸弹爆炸后冲击波是以正方形方式扩散的，炸弹威力（扩散距离）以d给出，表示可以传播d条街道。

例如下图是一个d=1的“垃圾炸弹”爆炸后的波及范围。



图：“垃圾炸弹”爆炸冲击波范围

假设莫斯科的布局为严格的 1025×1025 的网格状，由于财政问题市政府只买得起一枚“垃圾炸弹”，希望你帮他们找到合适的投放地点，使得一次清除的垃圾总量最多（假设垃圾数量可以用一个非负整数表示，并且除设置大屏幕的路口以外的地点没有垃圾）。

输入

第一行给出“炸弹”威力 $d(1 \leq d \leq 50)$ 。第二行给出一个数组 $n(1 \leq n \leq 20)$ 表示设置了大屏幕(有垃圾)的路口数目。接下来 n 行每行给出三个数字 x, y, i , 分别代表路口的坐标 (x, y) 以及垃圾数量 i . 点坐标 (x, y) 保证是有效的 (区间在0到1024之间) , 同一坐标只会给出一次。

输出

输出能清理垃圾最多的投放点数目，以及能够清除的垃圾总量。

样例输入

```

1 | 1
2 | 2
3 | 4 4 10
4 | 6 6 20

```

样例输出

```

1 | 1 30

```

```

1 #gpt
2 ...
3 过遍历方式计算出在每个点投掷炸弹能清理的垃圾数量，并用max_point存储垃圾数量的最大值,
4 res存储清理垃圾数量最大时的点的数量。最后输出结果。
5 是一个比较经典的滑动窗口问题
6 ...
7 d = int(input())

```

```
8 n = int(input())
9 square = [[0]*1025 for _ in range(1025)]
10 for _ in range(n):
11     x, y, k = map(int, input().split())
12     #for i in range(x-d if x-d >= 0 else 0, x+d+1 if x+d <= 1024 else 1025):
13         #for j in range(y-d if y-d >= 0 else 0, y+d+1 if y+d <= 1024 else 1025):
14             for i in range(max(x-d, 0), min(x+d+1, 1025)):
15                 for j in range(max(y-d, 0), min(y+d+1, 1025)):
16                     square[i][j] += k
17
18 res = max_point = 0
19 for i in range(0, 1025):
20     for j in range(0, 1025):
21         if square[i][j] > max_point:
22             max_point = square[i][j]
23             res = 1
24         elif square[i][j] == max_point:
25             res += 1
26 print(res, max_point)
```

三、其他知识点

3.1 Regular expression 甜点

不要求必须掌握，但是会了可以用，有时候很便捷。

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

正则表达式，又称规则表达式，(Regular Expression，在代码中常简写为regex、regexp或RE)，是一种文本模式，包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为“元字符”），是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串，通常被用来检索、替换那些符合某个模式（规则）的文本。

Regulex正则表达式在线测试工具，<https://regex101.com>

Python正则表达式详解

https://blog.csdn.net/weixin_43347550/article/details/105158003

简单的正则表示使用，需要掌握，有时候做题很方便。两个例题：

LC3484.设计电子表格

OOP, RE, <https://leetcode.cn/contest/biweekly-contest-152/problems/design-spreadsheet/>

04015: 邮箱验证

RE, <http://cs101.openjudge.cn/practice/04015>

正则表达式 (Regular Expression, 简称regex或regexp) 是一种强大的文本处理工具，广泛应用于字符串的搜索、匹配和替换操作。

正则表达式基础

1. 普通字符：指的是可以直接匹配的字符，如字母、数字等。

2. 元字符：具有特殊意义的字符，用于表示复杂的模式。例如：

- . 匹配除换行符外的任意字符。
- ^ 表示匹配输入字符串的开始位置。
- \$ 表示匹配输入字符串的结束位置。
- * 匹配前面的子表达式零次或多次。
- + 匹配前面的子表达式一次或多次。
- {n} 匹配确定的 n 次。
- [abc] 字符集，匹配方括号内的任意字符。
- \d 匹配一个数字字符，等价于 [0-9]。
- \w 匹配包括下划线的任何单词字符，等价于 [A-Za-z0-9_]。

应用场景

- **数据验证**: 例如验证电子邮件地址格式是否正确。
- **查找与替换**: 在文档中查找特定模式的文本，并进行替换。
- **文本提取**: 从大量文本中提取所需的信息，如日志文件分析。

工具和资源

- **Regulex/Regex101**: 提供了在线测试正则表达式的平台，可以帮助用户构建和调试正则表达式。Regex101, <https://regex101.com> 是一个非常流行的在线工具，支持多种正则表达式语法（包括Python），并提供详细的解释和分步调试功能。
- **学习资源**: 对于想要深入了解Python正则表达式的开发者，可以参考CSDN上的这篇文章, http://blog.csdn.net/weixin_43347550/article/details/105158003, 它详细介绍了Python中正则表达式的使用方法，包括如何使用 `re` 模块进行匹配、搜索、分割和替换操作。

re.match, re.search区别?

匹配范围：

- `re.match` 只检查字符串的开始部分。
- `re.search` 检查整个字符串，直到找到第一个匹配为止。

示例04015: 邮箱验证

strings, <http://cs101.openjudge.cn/practice/04015>

POJ 注册的时候需要用户输入邮箱，验证邮箱的规则包括：

- 1)有且仅有一个'@'符号
 - 2)'@'和'.'不能出现在字符串的首和尾
 - 3)'@'之后至少要有一个'.'，并且'@'不能和'.'直接相连
- 满足以上3条的字符串为合法邮箱，否则不合法，
编写程序验证输入是否合法

输入

输入包含若干行，每一行为一个待验证的邮箱地址，长度小于100

输出

每一行输入对应一行输出

如果验证合法，输出 YES

如果验证非法：输出 NO

样例输入

```
1 .a@b.com
2 pku@edu.cn
3 cs101@gmail.com
4 cs101@gmail
```

样例输出

```
1 NO
2 YES
3 YES
4 NO
```

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.

题目给的要求是[^@.], 也就是说正常字段只需要不是“@”和“.”即可。以前遇到的要求是：正常字段只能是大小写字母或“-”，所以也试了试[\w-]。虽然regulation需要前后match，也就是说前面加一个“^”，后面加一个“\$”，但是.match函数本身就是从头开始检索的，所以“^”可以删去。

```
1 # https://www.tutorialspoint.com/python/python_reg_expressions.htm
2 # https://www.geeksforgeeks.org/python-regex/
3
4 import re
5 while True:
6     try:
7         s = input()
8         reg = r'^[\w-]+(\.[\w-]+)*@[ \w-]+\(\.[\w-]+\)+$'
9         print('YES' if re.match(reg, s) else 'NO')
10    except EOFError:
11        break
```

^: 匹配字符串的开始。

[\w-]+: 匹配用户名部分的第一个子部分，允许字母、数字、下划线和连字符，至少有一个字符。

([\w-]+)*: 匹配用户名部分的其余子部分，每个子部分由点分隔，可以有零个或多个这样的子部分。

@: 匹买单个 @ 符号。

[\w-]+: 匹配域名的第一部分，允许字母、数字、下划线和连字符，至少有一个字符。

([\w-]+)+: 匹配域名的后续部分，每个部分由点分隔，至少有一个这样的部分，并且每个部分都至少包含一个字符。

\$: 匹配字符串的结尾，确保整个字符串都被匹配到结尾，不允许多余的字符。

```

1 # https://www.tutorialspoint.com/python/python_reg_expressions.htm
2 # https://www.geeksforgeeks.org/python-regex/
3 import re
4 while True:
5     try:
6         s = input()
7         reg = r'^[^@\.]+(\.[^@\.]+)*@[^\@\.]+(\.[^\@\.]+)+$'
8         print('YES' if re.match(reg, s) else 'NO')
9     except EOFError:
10        break

```

正则表达式遵循以下规则：

^：匹配字符串的开始。

[^@\.]+：匹配不包含 @ 和 . 的字符序列，确保用户名部分不以 @ 或 . 开始。

(\.[^@\.]+)*：允许用户名部分有多个子部分，每个子部分由点分隔。

@：匹配单个 @ 符号。

[^\@\.]+：匹配域名的第一部分，确保其不包含 @ 和 ..

(\.[^\@\.]+)+：匹配域名的后续部分，每个部分必须至少包含一个非 @ 和 . 的字符，并且必须至少有一个这样的部分。

\$：确保整个字符串都被匹配到结尾，不允许多余的字符。

[^xyz]，匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”任一字符。

\$匹配输入行尾。

(pattern)，匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到。

<https://regex101.com>

The screenshot shows the regex101.com interface. The left sidebar includes options for saving, sharing, and selecting programming languages (Python, PCRE2, PCRE, ECMAScript, etc.). The main area displays the regular expression pattern: `^[^@\.]+(\.[^@\.]+)*@[^\@\.]+(\.[^\@\.]+)+$`. The 'EXPLANATION' panel provides a detailed breakdown of the regex components:

- `^`: matches the previous token between one and unlimited times, as many times as possible, giving back as needed (greedy)
- `[^@\.]+`: matches the character @ with index 64₁₀ (40₁₆ or 100₈) literally (case sensitive)
- `\.`: matches the character . with index 46₁₀ (2E₁₆ or 56₈) literally (case sensitive)
- `*@`: matches the character @ with index 64₁₀ (40₁₆ or 100₈) literally (case sensitive)
- `[^\@\.]+`: matches the character . with index 46₁₀ (2E₁₆ or 56₈) literally (case sensitive)
- `+$/`: matches the character @ with index 64₁₀ (40₁₆ or 100₈) literally (case sensitive)

练习24834: 通配符匹配

<http://cs101.openjudge.cn/practice/24834/>

给定一个字符串s和一个字符模式p, 请实现一个支持'?'和'*'的通配符匹配功能。

其中'?'可以匹配任何单个字符, 如'a?c'可以成功匹配'aac','abc'等字符串, 但不可匹配'ac','aaac'等字符串。

'*'可以匹配任意长度字符串(包括空字符串), 如'a*c'可以成功匹配'ac','abdc','abc','aaac'等字符串, 但不可匹配'acb', 'cac'等字符串。

两个字符串完全匹配才算匹配成功。

输入

输入为一个数字n表示测试字符串与字符模式对数, 换行。(n ≤ 30)

后续2n行为每组匹配的s与p, 每行字符串后换行。

s 非空, 只包含从 a-z 的小写字母。

p 非空, 只包含从 a-z 的小写字母, 以及字符 ? 和 *。

字符串s和p的长度均小于50

输出

每一组匹配串匹配成功输出'yes', 否则输出'no'。

样例输入

1	3
2	abc
3	abc
4	abc
5	a*c
6	abc
7	a??c

样例输出

1	yes
2	yes
3	no

```
1 #23n2300017735(夏天明BrightSummer)
2 import re
3
4 for i in range(int(input())):
5     s, p = input(), input().replace("?", ".{1}").replace("*", ".*") + "$"
6     print("yes" if re.match(p, s) else "no")
```

.点，匹配除“\n”和“\r”之外的任何单个字符。要匹配包括“\n”和“\r”在内的任何字符，请使用像“[\s\S]”的模式。

*, 匹配前面的子表达式任意次。例如，z能匹配“z”，也能匹配“zo”以及“zoo”。等价于{0,}。

练习58A. Chat room

greedy/strings, 1000, <http://codeforces.com/problemset/problem/58/A>

Vasya has recently learned to type and log on to the Internet. He immediately entered a chat room and decided to say hello to everybody. Vasya typed the word s. It is considered that Vasya managed to say hello if several letters can be deleted from the typed word so that it resulted in the word "hello". For example, if Vasya types the word "ahhellllloou", it will be considered that he said hello, and if he types "hlelo", it will be considered that Vasya got misunderstood and he didn't manage to say hello. Determine whether Vasya managed to say hello by the given word s.

Input

The first and only line contains the word s, which Vasya typed. This word consists of small Latin letters, its length is no less than 1 and no more than 100 letters.

Output

If Vasya managed to say hello, print "YES", otherwise print "NO".

Examples

input

```
1 | ahhellllloou
```

output

```
1 | YES
```

input

```
1 | hlelo
```

output

```
1 | NO
```

```
1 import re
2 s = input()
3 r = re.search('h.*e.*l.*o', s)
4 print(['YES', 'NO'][r==None])
```

练习LeetCode 65. 有效数字

<https://leetcode.cn/problems/valid-number/description/>

<https://leetcode.cn/problems/valid-number/solutions/564188/you-xiao-shu-zi-by-leetcode-solution-298/>

这个正则表达式 pattern 用于判断一个字符串是否是有效数字。下面我来详细解释一下其中的各个部分：

- `^` 表示匹配字符串的开始位置。
- `[+-]?` 表示一个可选的符号字符，可以是正号 `+` 或负号 `-`。
- `(\d+(\.\d*)?|\.\d+)` 表示有效数字的主要部分，可以分成三种情况：
 - `\d+(\.\d*)?` 表示至少一位数字，后面可选的小数部分，小数部分可以没有或有多个小数位。
 - `|` 表示或的关系。
 - `.\d+` 表示以点 `.` 开始，后面至少一位数字的小数形式。
- `([eE][+-]?\d+)?` 表示指数部分，也是一个可选项，可以是 `e` 或 `E` 开头，后面可选的符号字符，以及至少一位数字。
- `$` 表示匹配字符串的结束位置。

综合起来，整个正则表达式可以解释为：

- 首先可以匹配一个可选的符号字符。
- 接下来是有效数字的主要部分，可以是整数或小数形式。
- 最后是可选的指数部分。

因此，该正则表达式可以匹配符合有效数字要求的字符串。在 Python 中使用 `re.match` 方法进行匹配时，如果匹配成功，说明字符串是一个有效数字，返回 `True`；否则，返回 `None`，表示不是一个有效数字。

```
1 class Solution:
2     def isNumber(self, s: str) -> bool:
3         import re
4         pattern = r'^[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?d+)?$'
5
6         ans = re.match(pattern, s)
7         if ans is not None:
8             return True
9         else:
10            return False
```

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

22 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.

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
```

3.2 DP的影子

优化问题除了使用时间复杂度更低的算法（如：线性筛/欧拉筛），还可以用DP。

```
from functools import lru_cache; lru_cache(maxsize = None)
```

Introduction_to_algorithms-3rd_Edition.pdf * - Foxit PDF Editor

File Home Convert Edit Organize Comment View Form Protect Share Accessibility Help

Introduction_to_algo... X

Bookmarks

- Contents
- Preface
- I Foundations
- II Sorting and Order Statistics
- III Data Structures
- IV Advanced Design and Anal...
- 15 Dynamic Programming
- 16 Greedy Algorithms
- 17 Amortized Analysis
- V Advanced Data Structures
- VI Graph Algorithms
- VII Selected Topics
- VIII Appendix: Mathematical ...
- A Summations
- B Sets, Etc.
- C Counting and Probability
- D Matrices
- Bibliography
- Index

Dynamic programming, like the divide-and-conquer method, solves problems by **combining the solutions to subproblems**. (“Programming” in this context refers to a **tabular** method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when **the subproblems overlap—that is, when subproblems share subsubproblems**. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm **solves each subsubproblem just once and then saves its answer in a table**, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to **optimization problems**. Such problems can have many possible solutions. Each solution has a value, and we wish to **find a solution with the optimal (minimum or maximum) value**. We call such a solution *an optimal solution* to the problem, as opposed to *the optimal solution*, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, **typically in a bottom-up fashion**.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 15.1 examines the problem of cutting a rod into

“Programming” 指的是一种表格法，不是写计算机程序

- Dp应用于自问题重叠的情况，即不同的子问题具有公共的子问题
- Dp对每个子问题只求解一次，并将其解保存在一个表格中，从而无需每次求解一个子问题时都重新计算
- 最优化问题 (optimization problems) 可以有很多可行解，每个解都有一个值，希望寻找具有最优值 (最小值或最大值) 的解。
- 称这样的解为问题的一个最优解 (an optimal solution)，而不是最优解 (the optimal solution)，因为可能有多个解都达到最优值

• 四个步骤来设计一个dp算法：

- 刻画一个最优解的结构特征
- 递推地定义最优解的值
- 计算最优的值，通常采用自底向上的方法
- 利用计算出的信息结构构造一个最优解

Contents
Preface
+ Foundations
+ II Sorting and Order Statistics
+ III Data Structures
+ IV Advanced Design and Anal...
- 15 Dynamic Programming
- 16 Greedy Algorithms
- 17 Amortized Analysis
+ V Advanced Data Structures
+ VI Graph Algorithms
+ VII Selected Topics
+ VIII Appendix: Mathematical ...
- A Summations
- B Sets, Etc.
- C Counting and Probability
- D Matrices
Bibliography
Index

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine **the two key ingredients that an optimization problem must have** in order for dynamic programming to apply: **optimal substructure and overlapping subproblems**. We also revisit and discuss more fully how memoization might help us take advantage of the overlapping-subproblems property in a top-down recursive approach.

15.3 Elements of dynamic programming

379

mization problem must have in order for dynamic programming to apply: **optimal substructure and overlapping subproblems**. We also revisit and discuss more fully how memoization might help us take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply. (As Chapter 16 discusses, it also might mean that a **greedy strategy applies**, however.) In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

15.3 动态规划的要素

尽管我们刚刚通过两个动态规划方法的例子进行了讲解，你可能仍然在思考这种方法适用的条件。从工程的角度来看，什么时候我们应该寻找一个问题的动态规划解法呢？在本节中，我们将探讨最优问题必须具备的两个关键要素，以便应用动态规划：**最优子结构**和**重叠子问题**。我们还将重新审视并更深入地讨论记忆化（memoization）如何帮助我们在自顶向下的递归方法中利用重叠子问题的特性。

最优子结构

使用动态规划求解优化问题的第一步是刻画最优解的结构。回忆一下，如果一个问题的最优解包含其子问题的最优解，则称该问题具有**最优子结构**。每当一个问题表现出最优子结构时，我们就有了一个很好的线索，表明动态规划可能适用。（如第16章所述，这也可能意味着贪心策略适用，但情况并非总是如此。）在动态规划中，我们通过子问题的最优解来构建原问题的最优解。因此，必须小心确保我们考虑的子问题范围包括了最优解中所使用的那些子问题。

Contents
Preface
+ Foundations
+ II Sorting and Order Statistics
+ III Data Structures
+ IV Advanced Design and Anal...
- 15 Dynamic Programming
- 16 Greedy Algorithms
- 17 Amortized Analysis
+ V Advanced Data Structures
+ VI Graph Algorithms
+ VII Selected Topics
+ VIII Appendix: Mathematical ...
- A Summations

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, **the total number of distinct subproblems is a polynomial in the input size**. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has **overlapping subproblems**.⁴ In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by **solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup**.

重叠子问题

动态规划适用于优化问题的第二个必要条件是：子问题的空间必须“较小”，也就是说，该问题的递归算法会反复求解相同的子问题，而不是总是生成新的子问题。通常情况下，不同子问题的总数是输入规模的多项式。当一个递归算法反复访问同一个问题时，我们就说这个优化问题具有重叠子问题。相比之下，适合使用分治法的问题通常在递归的每一步都会生成全新的问题。动态规划算法通常通过每次只解决每个子问题一次，然后将解存储在一个表中，以便在需要时以常数时间进行查找，从而利用重叠子问题的特性。

230B. T-primes

binary search, implementation, math, number theory, 1300, <http://codeforces.com/problemset/problem/230/B>

求解素数的三种方法，包括：试除法（trial division）、埃氏筛（Sieve of Eratosthenes）、欧拉筛（Sieve of Euler，线性法），<https://blog.dotcpp.com/a/69737>

@lru_cache(maxsize = None)

除余法，pypy3可以AC 230B。lru_cache 如果屏蔽了，超时。

Contest status :						
#	When	Who	Problem	Lang	Verdict	Time Memory
227551469	Oct/10/2023 22:21 UTC+8	GMyhf	230B - T-primes	PyPy 3-64	Time limit exceeded on test 63	2000 ms 19100 KB
227551224	Oct/10/2023 22:20 UTC+8	GMyhf	230B - T-primes	PyPy 3-64	Accepted	998 ms 20400 KB

```
1 import math
2 from functools import lru_cache
3
4 @lru_cache(maxsize = None)
5 def prime(x):
6     for i in range(2, int(math.sqrt(x))+1):
7         if x % i == 0:
8             return False
9     return True
10
11 input()
12
13 *a, = map(int, input().split())
14 ans = []
15 for i in a:
16     if i == 1:
17         ans.append('NO')
18         continue
19     tmp = int(math.sqrt(i))
20     if tmp**2 != i:
21         ans.append('NO')
22         continue
23
24     if prime(tmp):
```

```
25     ans.append('YES')
26 else:
27     ans.append('NO')
28
29 print('\n'.join(ans))
```

02810: 完美立方

- 1) lru_cache 有作用，时间接近先算好的方法。完美立方，<http://cs101.openjudge.cn/practice/02810/> 2)
- 今天课件里面用lru_cache的程序没有写对，因为它对函数的参数起缓存作用，所以作用的函数一定要有参数。

状态: Accepted

源代码

```
from functools import lru_cache

@lru_cache(maxsize = 128)
def cube(i):
    return i**3

def solv():
    N = int(input())
    ans = []
    for a in range(2,N+1):
        for b in range(2,a):
            for c in range(b,a):
                for d in range(c,a):
                    #if a ** 3 == b ** 3 + c ** 3 + d ** 3:
                    if cube(a) == cube(b) + cube(c) + cube(d):
                        #print('Cube = %d, Triple = (%d,%d,%d)' % (a,b,c))
                        ans.append((a,b,c,d))

    return ans

for a,b,c,d in solv():
    print(f"Cube = {a}, Triple = ({b},{c},{d})")
```

基本信息

#: 41573897
题目: 02810
提交人: HFYan(GMyhf)
内存: 3652kB
时间: 1352ms
语言: Python3
提交时间: 2023-10-10 23:29:55

状态



cs101.openjudge.cn/practice/solution/41519922/



OpenJudge

题目ID, 标题, 描述



GMyhf

信箱(4)



CS101 / 题库

题目 排名 状态 提问

#41519922 提交状态

查看 提交 统计

状态: Accepted

源代码

```
n = int(input())
cube = [i**3 for i in range(n+1)]
for a in range(3,n+1):
    for b in range(2,a):
        for c in range(b,a):
            for d in range(c,a):
                if cube[a]==cube[b]+cube[c]+cube[d]:
                    print(f"Cube = {a}, Triple = ({b},{c},{d})")
```

基本信息

#: 41519922
题目: 02810
提交人: HFYan(GMyhf)
内存: 3656kB
时间: 875ms
语言: Python3
提交时间: 2023-10-08 10:33:52

3.3 Data Structure

三个常用的数据结构: stack, queue, heap

Stack in Python

<https://www.geeksforgeeks.org/stack-in-python/>

Stack in Python

Read

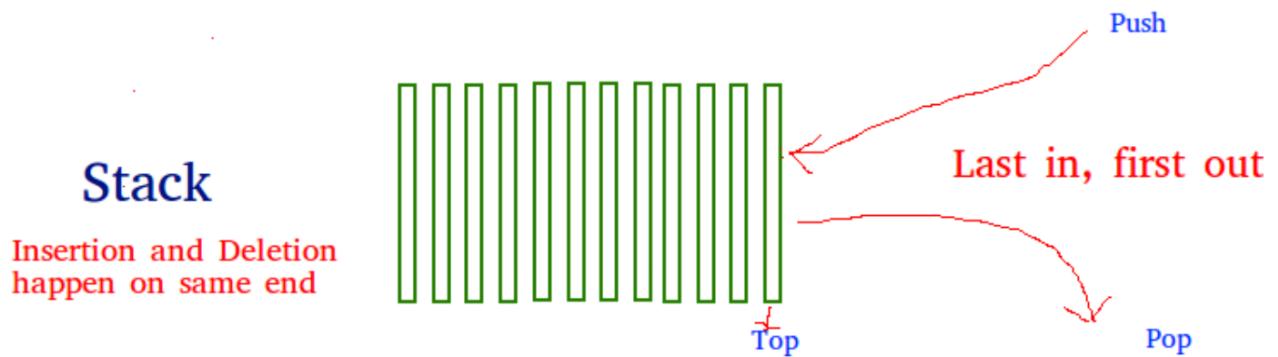
Discuss

Courses

Practice

⋮

A **stack** is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)
- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

Queue in Python

<https://www.geeksforgeeks.org/queue-in-python/>

Queue in Python

Read

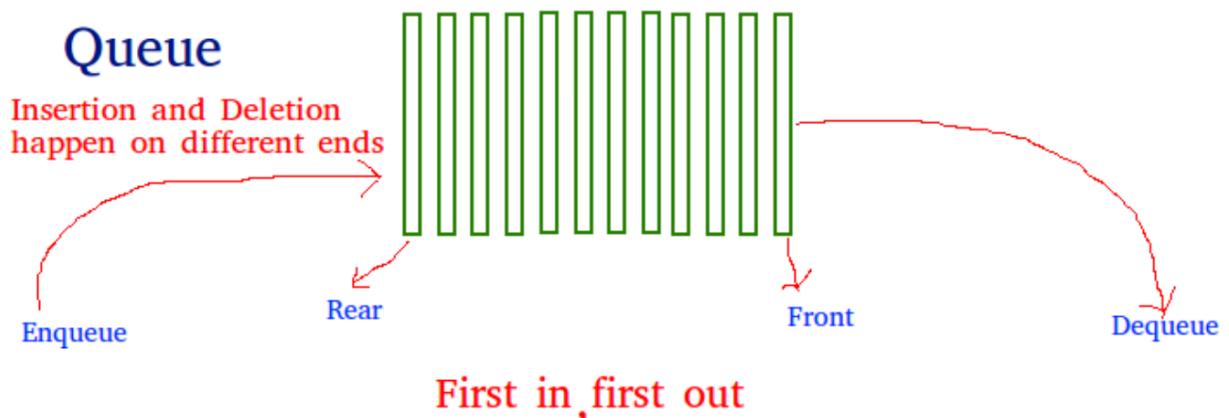
Discuss

Courses

Practice

⋮

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
- **Front:** Get the front item from queue – Time Complexity : O(1)
- **Rear:** Get the last item from queue – Time Complexity : O(1)

Heap queue (or heapq) in Python

<https://www.geeksforgeeks.org/heap-queue-or-heappq-in-python/>

Heap queue (or heapq) in Python

Read Discuss Courses Practice Video

⋮

Heap data structure is mainly used to represent a priority queue. In Python, it is available using the “heapq” module. The property of this data structure in Python is that each time the smallest heap element is popped(min-heap). Whenever elements are pushed or popped, heap structure is maintained. The `heap[0]` element also returns the smallest element each time. Let's see various Operations on the heap in [Python](#).

Creating a simple heap

The `heapify(iterable)`:- This function is used to convert the iterable into a heap data structure. i.e. in heap order.

Python3

```
# importing "heapq" to implement heap queue
import heapq

# initializing list
li = [5, 7, 9, 1, 3]

# using heapify to convert list into heap
heapq.heapify(li)

# printing created heap
print ("The created heap is : ",list(li))
```

Output

```
The created heap is : [1, 3, 9, 7, 5]
```

3.4 程序生成测试数据

练习26971:分发糖果

greedy, <http://cs101.openjudge.cn/routine/26971/>

n 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的 **最少糖果数目**。

输入

第一行包含一个整数n。 $1 \leq n \leq 2 * 10^4$

第二行包含n个整数，相邻整数间以空格隔开。 $0 \leq \text{ratings}[i] \leq 2 * 10^4$

输出

一个整数

样例输入

```

1 Sample1 input:
2 3
3 1 0 2
4 Sample1 output:
5 5

```

样例输出

```

1 Sample2 input:
2 3
3 1 2 2
4 Sample2 output:
5 4

```

提示

tags: greedy

来源

LeetCode 135.分发糖果：<https://leetcode.cn/problems/candy/>

```

1 import random
2 import time
3
4 random.seed(0)
5
6 for epoch in range(20):
7     n = random.randint(10 + epoch * 2, 900 + 200 * (epoch // 2) ** 2)
8     nums = [random.randint(0, 800 + 200 * (epoch // 2) ** 2) for _ in range(n)]
9     inlines = [f'{n}\n'] + [' '.join([str(num) for num in nums]) + '\n']
10
11     with open(f'data/{epoch}.in', 'w') as f:

```

```

12         f.writelines(inlines)
13
14     def candy(ratings):
15         n = len(ratings)
16         left = [0] * n
17         for i in range(n):
18             if i > 0 and ratings[i] > ratings[i - 1]:
19                 left[i] = left[i - 1] + 1
20             else:
21                 left[i] = 1
22
23         right = ret = 0
24         for i in range(n - 1, -1, -1):
25             if i < n - 1 and ratings[i] > ratings[i + 1]:
26                 right += 1
27             else:
28                 right = 1
29             ret += max(left[i], right)
30
31     return ret
32
33 #input()
34 #*nums, = map(int, input().split())
35 start = time.time()
36 ans = candy(nums)
37 end = time.time() - start
38
39 print(f"[{epoch}] {end:.3f}sec")
40 print(ans)
41
42 with open(f'data/{epoch}.out', 'w') as f:
43     f.writelines([str(ans) + '\n'])

```

练习26976:摆动序列

greedy, <http://cs101.openjudge.cn/routine/26976/>

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为 **摆动序列**。第一个差（如果存在的话）可能是正数或负数。仅有一个元素或者含两个不等元素的序列也视作摆动序列。

- 例如，

[1, 7, 4, 9, 2, 5] 是一个 **摆动序列**，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。

- 相反，

[1, 4, 7, 2, 5]

[1, 7, 4, 5, 5]

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 `nums`，返回 `nums` 中作为 摆动序列 的 最长子序列的长度 。

输入

第一行包含一个整数n。 $1 \leq n \leq 1000$

第二行包含n个整数，相邻整数间以空格隔开。 $0 \leq \text{nums}[i] \leq 1000$

输出

一个整数

样例输入

```
1 sample1 input:  
2 6  
3 1 7 4 9 2 5  
4 sample1 output:  
5 6  
6  
7 sample2 input:  
8 10  
9 1 17 5 10 13 15 10 5 16 8  
10 sample2 output:  
11 7
```

样例输出

```
1 sample3 input:  
2 9  
3 1 2 3 4 5 6 7 8 9  
4 sample3 output:  
5 2
```

提示

tags: greedy

来源

LeetCode 376. 摆动序列: <https://leetcode.cn/problems/wiggle-subsequence/>

```
1 import random  
2 import time  
3  
4 random.seed(0)  
5  
6 for epoch in range(20):  
7     n = random.randint(1, min(1000, 10 + epoch*50))  
8     nums = [random.randint(0, 1000) for _ in range(n)]
```

```

9     inlines = [f'{n}\n'] + [' '.join([str(num) for num in nums]) + '\n']
10
11    with open(f'data/{epoch}.in', 'w') as f:
12        f.writelines(inlines)
13
14    def wiggleMaxLength(nums):
15        n = len(nums)
16        if n < 2:
17            return n
18
19        prevdiff = nums[1] - nums[0]
20        ret = (2 if prevdiff != 0 else 1)
21        for i in range(2, n):
22            diff = nums[i] - nums[i - 1]
23            if (diff > 0 and prevdiff <= 0) or (diff < 0 and prevdiff >= 0):
24                ret += 1
25            prevdiff = diff
26
27    return ret
28
29
30 #input()
31 #*nums, = map(int, input().split())
32 start = time.time()
33 ans = wiggleMaxLength(nums)
34 end = time.time() - start
35
36 print(f"[{epoch}] {end:.3f}sec")
37 print(ans)
38
39 with open(f'data/{epoch}.out', 'w') as f:
40     f.writelines([str(ans) + '\n'])

```

多解的题目

tags: constructive algorithms (If there are multiple solutions, you may output any of them. 多解的题目需要spj来处理, special judge)

请问这种题面的测试数据是如何构建的? <https://codeforces.com/problemset/problem/1868/A> If there are multiple solutions, you may output any of them. 多解
的题目需要spj来处理, special judge



测试数据的构建过程可以根据具体需求和题目的要求来进行设计。在这个问题的示例中, 链接指向的是一个Codeforces上的题目。

通常情况下, 测试数据的构建包括以下步骤:

1. 理解题目要求和约束条件: 首先要仔细阅读题目, 理解问题的需求和限制条件。这些内容将指导你在后续设计测试数据时的选择。
2. 设计输入数据: 根据题目的要求, 设计合适的输入数据。这些数据应该包含边界情况、特殊情况以及一般情况, 以覆盖题目中可能出现的各种情况。
3. 生成输出数据: 根据输入数据, 在考虑特殊情况的前提下, 生成对应的输出数据。确保输出数据符合题目的要求, 包括满足多解情况的要求。
4. 确保数据的正确性: 使用特殊判定程序 (Special Judge, SPJ) 来验证生成的输出数据是否正确。SPJ是一个自定义的程序, 用于验证参赛者的程序所产生的输出是否符合题目的要求。
5. 随机化和扩充数据: 为了增加测试的多样性和复杂性, 可以对输入数据进行随机化处理, 或者增加更多的边界情况和特殊情况。

需要注意的是, 测试数据的构建是一个相对复杂且需要经验的过程, 需要综合考虑题目要求、时间复杂度、空间复杂度等因素。有时候, 对于某些特殊类型的问题, 还需要特定的方法来构建足够复杂和完备的测试数据。

在竞赛中, 测试数据的设计非常重要, 因为它直接影响参赛者的解题效果和算法的正确性。因此, 测试数据的构建需要谨慎和充分的测试和验证。

练习CF1868A. Fill in the Matrix

constructive algorithms, implementation, 1300, <https://codeforces.com/problemset/problem/1868/A>

There is an empty matrix M of size $n \times m$.

Zhongkao examination is over, and Daniel would like to do some puzzle games. He is going to fill in the matrix using permutations of length m . That is, each row of M must be a permutation of length m .

Define the *value* of the i -th column in M as $v_i = MEX(M_{1,i}, M_{2,i}, \dots, M_{n,i})$. Since Daniel likes diversity, the *beauty* of M is $s = MEX(v_1, v_2, \dots, v_m)$.

You have to help Daniel fill in the matrix M and **maximize** its beauty.

† A permutation of length m is an array consisting of m distinct integers from 0 to $m-1$ in arbitrary order. For example, $[1, 2, 0, 4, 3]$ is a permutation, but $[0, 1, 1]$ is not a permutation (1 appears twice in the array), and $[0, 1, 3]$ is also not a permutation ($m-1=2$ but there is 3 in the array).

‡ The MEXMEX of an array is the smallest non-negative integer that does not belong to the array. For example, $MEX(2, 2, 1)=0$ because 0 does not belong to the array, and $MEX(0, 3, 1, 2)=4$ because 0, 1, 2 and 3 appear in the array, but 4 does not.

Input

The first line of input contains a single integer t ($1 \leq t \leq 1000$) — the number of test cases. The description of test cases follows.

The only line of each test case contains two integers n and m ($1 \leq n, m \leq 2 \cdot 10^5$) — the size of the matrix.

It is guaranteed that the sum of $n \cdot m$ over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, in the first line output a single integer — the maximum beauty of M .

Then output the matrix M of size $n \times m$ — the matrix you find.

If there are multiple solutions, you may output any of them.

Example

input

1	4
2	4 3
3	1 16
4	6 6
5	2 1

output

1	3
2	1 0 2
3	0 2 1
4	1 0 2
5	0 2 1
6	2
7	14 7 15 4 10 0 8 6 1 2 3 5 9 11 12 13
8	6
9	3 0 1 4 2 5
10	5 2 1 0 4 3
11	1 3 2 4 5 0
12	4 1 3 2 5 0
13	4 2 5 3 0 1
14	2 4 0 5 1 3
15	0
16	0
17	0

Note

In the first test case:

- $v_1 = \text{MEX}(1, 0, 1, 0) = 2$;
- $v_2 = \text{MEX}(0, 2, 0, 2) = 1$;
- $v_3 = \text{MEX}(2, 1, 2, 1) = 0$.

Therefore, $s = \text{MEX}(2, 1, 0) = 3$.

It can be shown that 33 is the maximum possible beauty of M .

In the second test case, any permutation will make $s=2$.

In the third test case:

- $v_1 = \text{MEX}(3, 5, 1, 4, 4, 2) = 0$;
- $v_2 = \text{MEX}(0, 2, 3, 1, 2, 4) = 5$;

- $v_3 = \text{MEX}(1, 1, 2, 3, 5, 0) = 4$;
- $v_4 = \text{MEX}(4, 0, 4, 2, 3, 5) = 1$;
- $v_5 = \text{MEX}(2, 4, 5, 5, 0, 1) = 3$;
- $v_6 = \text{MEX}(5, 3, 0, 0, 1, 3) = 2$.

Therefore, $s = \text{MEX}(0, 5, 4, 1, 3, 2) = 6$.

3.5 testing_code.py

https://github.com/GMyhf/2025fall-cs101/blob/main/code/testing_code.py

```

1 import subprocess
2 import difflib
3 import os
4 import sys
5
6 def test_code(script_path, infile, outfile):
7     command = [sys.executable, script_path] # 使用Python解释器运行脚本
8     with open(infile, 'r') as fin, open(outfile, 'r') as fout:
9         expected_output = fout.read().strip()
10        # 启动一个新的子进程来运行指定的命令
11        process = subprocess.Popen(command, stdin=fin, stdout=subprocess.PIPE)
12        actual_output, _ = process.communicate()
13        if actual_output.decode().strip() == expected_output:
14            return True
15        else:
16            print(f"Output differs for {infile}:")
17            diff = difflib.unified_diff(
18                expected_output.splitlines(),
19                actual_output.decode().splitlines(),
20                fromfile='Expected', tofile='Actual', lineterm='\n'
21            )
22            print('\n'.join(diff))
23            return False
24
25
26 if __name__ == "__main__":
27     # 检查命令行参数的数量
28     if len(sys.argv) != 2:
29         print("Usage: python testing_code.py <filename>")
30         sys.exit(1)
31
32     # 获取文件名
33     script_path = sys.argv[1]
34
35     #script_path = "class.py" # 你的Python脚本路径
36     #test_cases = ["d.in"] # 输入文件列表

```

```
37     #expected_outputs = ["d.out"] # 预期输出文件列表
38     # 获取当前目录下的所有文件
39     files = os.listdir('.')
40
41     # 建立输入/输出字典
42     input_files = {}
43     output_files = {}
44
45     for file in files:
46         if file.endswith('.in'):
47             key = file[:-3] # 去掉 ".in"
48             input_files[key] = file
49         elif file.endswith('.out'):
50             key = file[:-4] # 去掉 ".out"
51             output_files[key] = file
52
53     # 遍历公共 key, 按字典序排序
54     for key in sorted(set(input_files.keys()) & set(output_files.keys())):
55         infile = input_files[key]
56         outfile = output_files[key]
57         if not test_code(script_path, infile, outfile):
58             break
59
```