

# 2025/10/21 编程技巧与算法实战：区间问题、排序、单调栈与动态规划

---

Updated 1443 GMT+8 Nov 04, 2025

2024 fall, Complied by Hongfei Yan

Log: 扩充了滑动窗口最大值、前缀和、Kadane算法

# 一、调试程序

<https://pythontutor.com> 很好用，适合还不会用Pycharm调试工具的，当然后者也好用。另外就是print变量输出。

计算中心机房 PyCharm 比 VS Code 补全功能强。

## 1 Pycharm调试

在行号处点击，设置端点；然后点击右上角绿色小虫子debug模式运行。

The screenshot shows the PyCharm IDE interface. In the top right corner, there is a GitHub Copilot Chat window with a red circled '2' icon, displaying a task about implementing a function to find a digit at a given position. Below it is an 'Implementation' section with sample code using the decimal module. In the bottom right corner, there is a 'Scratches' window showing a list of words: 6, MO0000, 000000, ABCDEF, UVWXYZ, COW, MOO, ZOO, MOVE, CODE, FARM. At the bottom of the screen, the status bar displays: Scratches > scratch\_976.py, 14:1 LF UTF-8 4 spaces Python 3.10 (base), and a few other icons.

debug运行后，停在设置了短点的语句，变量中的值都显示出来。

The screenshot shows the PyCharm IDE interface. On the left, the code editor displays `scratch_976.py` with several variables and loops defined. A red arrow points from the code editor to the GitHub Copilot Chat panel on the right. The GitHub Copilot Chat panel contains a task titled "Implementing a Function to Find the Digit at a Given Position" with three steps: 1. Set precision, 2. Perform calculation, and 3. Print result. Below this is the "Implementation" section with sample code using the `decimal` module. Another red arrow points from the code editor to the Debug tool window on the right. The Debug tool window shows the current state of variables in the Main Thread, including `N=6`, `dc=[...]`, and `m1=[0, 0, 0, 0]`. The status bar at the bottom indicates the file is 11:1 LF, in UTF-8 encoding, with 4 spaces, and is written in Python 3.10 (base).

## 2 pythontutor可视化运行

递归程序运行过程，不容易理解。<https://pythontutor.com>，完美展示 归并排序 的递归过程。

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

Ads keep tt  
for ad cont

Python 3.6 known limitations

```

1 def MergeSort(arr):
2     if len(arr) <= 1:
3         return arr
4     mid = len(arr) // 2
5     left = MergeSort(arr[:mid])
6     right = MergeSort(arr[mid:])
7     return merge(left, right)
8
9 def merge(left, right):
10    result = []
11    i = j = 0
12    while i < len(left) and j < len(right):
13        if left[i] < right[j]:
14            result.append(left[i])
15            i += 1
16        else:
17            result.append(right[j])
18            j += 1
19    result.extend(left[i:])
20    result.extend(right[j:])
21    return result

```

[Edit this code](#)

line that just executed  
next line to execute

<< First | < Prev | Next > | Last >>

Step 44 of 694

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

Get AI Help

Move and hide objects

Print output (drag lower right corner to resize)

[6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9,

Frames Objects

Global frame → function MergeSort(arr)  
MergeSort → function merge(left, right)  
arr\_in → list [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]  
MergeSort arr [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 11]  
MergeSort arr [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7]  
MergeSort arr [6, 5, 18, 2, 16]  
MergeSort arr [6, 5]  
merge left [6, 5]  
merge right [5]  
merge result [0, 5, 6]  
merge i [0]  
merge j [1]

## 二、常见区间问题

一文读懂五类常见区间问题, <https://zhuanlan.zhihu.com/p/446371757>

练习网址, <https://leetcode.cn>。leetcode只需要完成核心代码就可以, 不用写输入输出部分。

核心代码已经指定好类名、方法名、参数名, 请勿修改或重命名, 直接返回值即可。

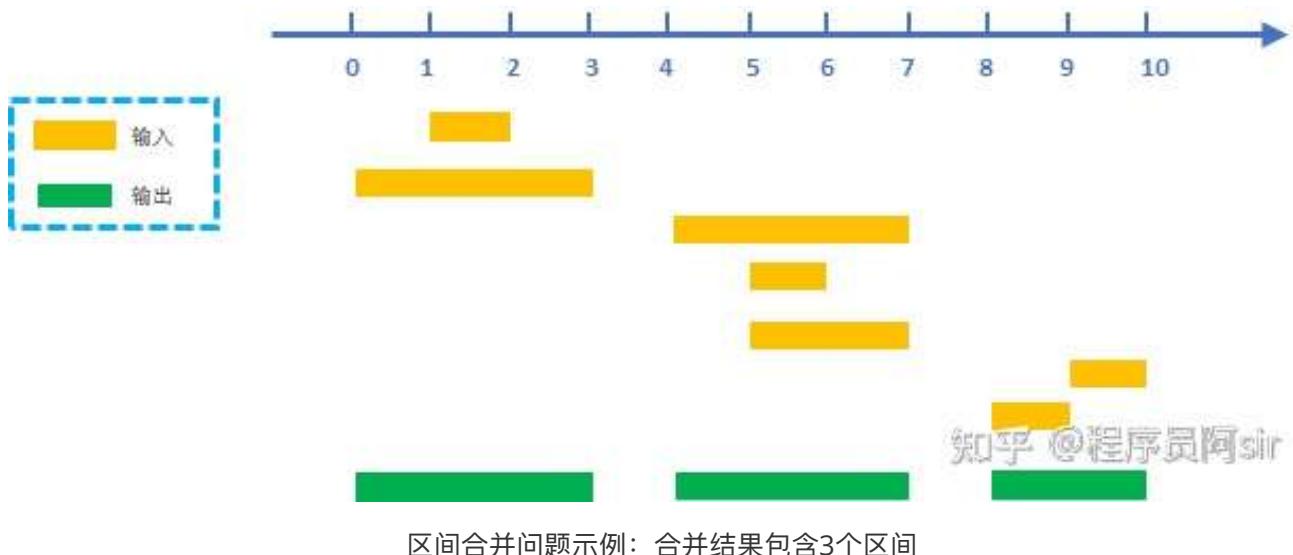
### 1 区间合并

#### 1.1 题意描述

区间合并问题大概题意就是:

给出一堆区间, 要求合并所有有交集的区间 (端点处相交也算有交集)。最后问合并之后的区间。

如下图所示:



#### 1.2 解题步骤

【步骤一】：按照区间左端点从小到大排序。

【步骤二】：维护前面区间中最右边的端点为ed。从前往后枚举每一个区间, 判断是否应该将当前区间视为新区间。

假设当前遍历到的区间为第*i*个区间  $[l_i, r_i]$ , 有以下两种情况:

- $l_i \leq ed$ : 说明当前区间与前面区间有交集。因此不需要增加区间个数, 但需要设置  $ed = \max(ed, r_i)$ 。
- $l_i > ed$ : 说明当前区间与前面没有交集。因此需要增加区间个数, 并设置  $ed = \max(ed, r_i)$ 。

### 练习LC M56.合并区间

<https://leetcode.cn/problems/merge-intervals/>

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

### 示例 1：

```
1 | 输入: intervals = [[1,3],[2,6],[8,10],[15,18]]
2 | 输出: [[1,6],[8,10],[15,18]]
3 | 解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

### 示例 2：

```
1 | 输入: intervals = [[1,4],[4,5]]
2 | 输出: [[1,5]]
3 | 解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

### 提示：

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 10^4`

```
1 | from typing import List
2 | import sys
3 |
4 | class Solution:
5 |     def merge(self, intervals: List[List[int]]) -> List[List[int]]:
6 |         # 对区间进行排序
7 |         intervals.sort()
8 |
9 |         res = []
10 |        st, ed = -sys.maxsize, -sys.maxsize
11 |
12 |        for v in intervals:
13 |            if ed == -sys.maxsize:
14 |                st, ed = v[0], v[1]
15 |            elif v[0] <= ed:
16 |                ed = max(v[1], ed)
17 |            elif v[0] > ed:
18 |                res.append([st, ed])
19 |                st, ed = v[0], v[1]
20 |
```

```
21         if ed != -sys.maxsize:  
22             res.append([st, ed])  
23  
24     return res
```

这里有几个需要注意的地方：

- `-sys.maxsize` 用于表示最小整数值。
- 类型注解（如 `List[List[int]]`）是可选的，但有助于提高代码的可读性和类型检查工具的效率。

## M29947:校门外的树又来了

greedy, interval merging, stack, <http://cs101.openjudge.cn/practice/29947/>

某校大门外长度为L的马路上有一排树，每两棵相邻的树之间的间隔都是1米。我们可以把马路看成一个数轴，马路的一端在数轴0的位置，另一端在L的位置；数轴上的每个整数点，即0, 1, 2, ……, L，都种有一棵树。马路上有一些区域要用来建地铁，这些区域用它们在数轴上的起始点和终止点表示。已知任一区域的起始点和终止点的坐标都是整数，区域之间可能有重合的部分。现在要把这些区域中的树（包括区域端点处的两棵树）移走。你的任务是计算将这些树都移走后，马路上还有多少棵树。

### 输入

输入的第一行有两个整数L ( $1 \leq L \leq 10^9$ ) 和 M ( $1 \leq M \leq 100$ )，L代表马路的长度，M代表区域的数目，L和M之间用一个空格隔开。接下来的M行每行包含两个不同的整数，用一个空格隔开，表示一个区域的起始点和终止点的坐标。

### 输出

输出包括一行，这一行只包含一个整数，表示马路上剩余的树的数目。

### 样例输入

```
1 500 3  
2 150 300  
3 100 200  
4 470 471
```

### 样例输出

```
1 298
```

### 来源

yan

思路：区间合并。排序是为了保证取交集的方向唯一，这种单向关系可以避免分类讨论。

```
1 L, M = map(int, input().split())
```

```

2 intervals = []
3
4 for _ in range(M):
5     s, e = map(int, input().split())
6     if s > e:
7         s, e = e, s
8     intervals.append((s, e))
9
10 # 按起点排序
11 intervals.sort()
12
13 # 合并区间
14 merged = []
15 for s, e in intervals:
16     if not merged or s > merged[-1][1] + 1:
17         merged.append([s, e])
18     else:
19         merged[-1][1] = max(merged[-1][1], e)
20
21 # 计算被砍掉的树总数
22 cut = sum(e - s + 1 for s, e in merged)
23 remain = (L + 1) - cut
24
25 print(remain)

```

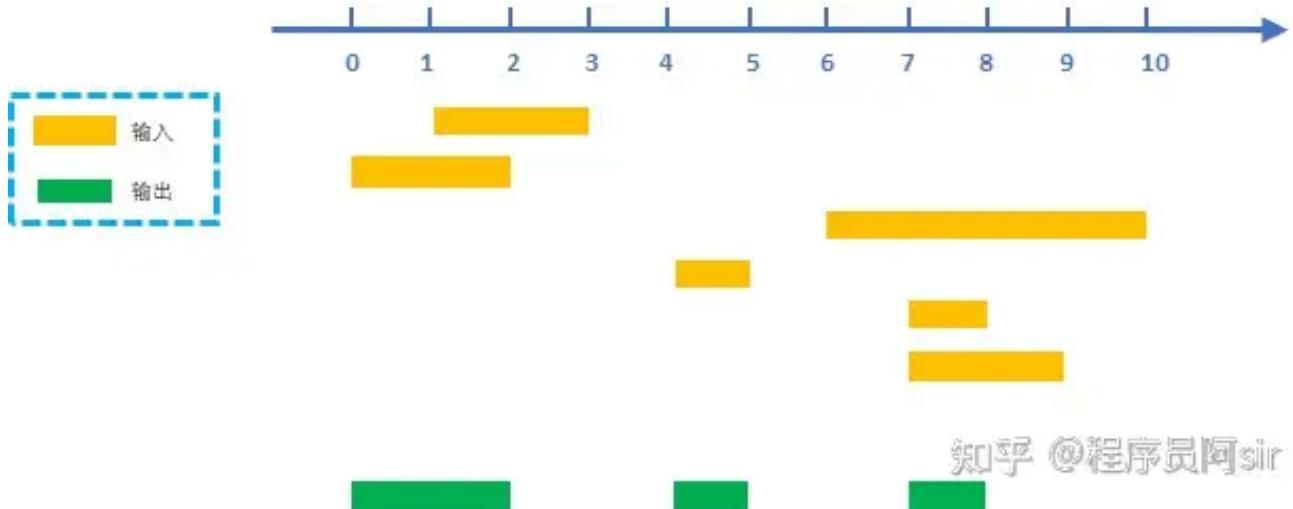
## 2 选择不相交区间

### 2.1 题意描述

选择不相交区间问题大概题意就是：

给出一堆区间，要求选择尽量多的区间，使得这些区间互不相交，求可选取的区间的最大数量。这里端点相同也算有重复。

如下图所示：



选择不相交区间问题示例：结果包含3个区间

## 2.2 解题步骤

**【步骤一】**：按照区间右端点从小到大排序。

**【步骤二】**：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  $[l_i, r_i]$ ，有以下两种情况：

- $l_i \leq ed$ : 说明当前区间与前面区间有交集。因此直接跳过。
- $l_i > ed$ : 说明当前区间与前面没有交集。因此选中当前区间，并设置  $ed = r_i$ 。

## 练习LC M435.无重叠区间

<https://leetcode.cn/problems/non-overlapping-intervals/>

给定一个区间的集合 `intervals`，其中 `intervals[i] = [starti, endi]`。返回 需要移除区间的最小数量，使剩余区间互不重叠。

注意 只在一点上接触的区间是 不重叠的。例如 `[1, 2]` 和 `[2, 3]` 是不重叠的。

### 示例 1:

```

1 | 输入: intervals = [[1,2],[2,3],[3,4],[1,3]]
2 | 输出: 1
3 | 解释: 移除 [1,3] 后, 剩下的区间没有重叠。

```

### 示例 2:

```

1 | 输入: intervals = [ [1,2], [1,2], [1,2] ]
2 | 输出: 2
3 | 解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

```

### 示例 3:

```
1 | 输入: intervals = [ [1,2], [2,3] ]  
2 | 输出: 0  
3 | 解释: 你不需要移除任何区间, 因为它们已经是无重叠的了。
```

### 提示:

- `1 <= intervals.length <= 10^5`
- `intervals[i].length == 2`
- `-5 * 10^4 <= starti < endi <= 5 * 10^4`

```
1 | from typing import List  
2 | import sys  
3 |  
4 | class Solution:  
5 |     def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:  
6 |         # 按照右端点从小到大排序  
7 |         intervals.sort(key=lambda x: x[1])  
8 |  
9 |         res = 0  
10 |        ed = -sys.maxsize  
11 |  
12 |        for v in intervals:  
13 |            if ed <= v[0]:  
14 |                res += 1  
15 |                ed = v[1]  
16 |  
17 |        return len(intervals) - res
```

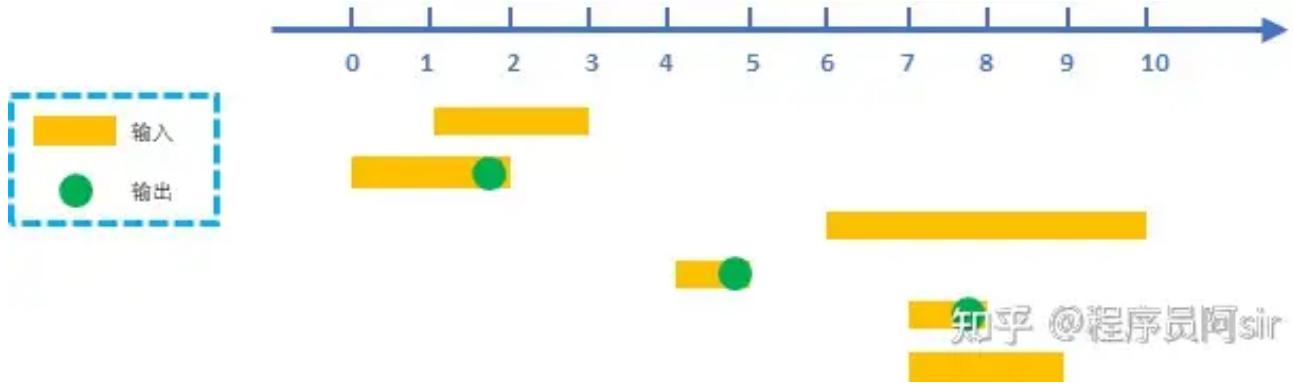
## 3 区间选点问题

### 3.1 题意描述

区间选点问题大概题意就是：

给出一堆区间，取尽量少的点，使得每个区间内至少有一个点（不同区间内含的点可以是同一个，位于区间端点上的点也算作区间内）。

如下图所示：



区间选点问题示例，最终至少选择3个点

这个题可以转化为上一题的求最大不相交区间的数量。

对于这些最大的不相交区间，肯定是每个区间都需要选出一个点。而其他的区间都是和这些选出的区间有重复的，我们只需要把点的位置选在重合的部分即可。

也可以换一种思路：

我们将区间按照右端点从小到大排序，这时我们应该尽量选择当前区间最右边的点。

因为最右边的点可能和下面的其他区间重复，所以至少不比选择区间靠前位置的点差。

所以，最后的解法与选择不相交区间问题解法完全一样。

## 3.2 解题步骤

**【步骤一】**：按照区间右端点从小到大排序。

**【步骤二】**：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  $[l_i, r_i]$ ，有以下两种情况：

- $l_i \leq ed$ : 说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- $l_i > ed$ : 说明当前区间与前面没有交集。因此选中当前区间，并设置  $ed = r_i$ 。

## 练习LC M452. 用最少量的箭引爆气球

<https://leetcode.cn/problems/minimum-number-of-arrows-to-burst-balloons/>

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points`，其中 `points[i] = [xstart, xend]` 表示水平直径在 `xstart` 和 `xend` 之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点 完全垂直 地射出。在坐标 `x` 处射出一支箭，若有一个气球的直径的开始和结束坐标为 `xstart, xend`，且满足 `xstart ≤ x ≤ xend`，则该气球会被引爆。可以射出的弓箭的数量 没有限制。弓箭一旦被射出之后，可以无限地前进。

给你一个数组 `points`，返回引爆所有气球所必须射出的 最小 弓箭数。

### 示例 1:

```
1 | 输入: points = [[10,16],[2,8],[1,6],[7,12]]  
2 | 输出: 2  
3 | 解释: 气球可以用2支箭来爆破:  
4 | -在x = 6处射出箭, 击破气球[2,8]和[1,6]。  
5 | -在x = 11处发射箭, 击破气球[10,16]和[7,12]。
```

### 示例 2:

```
1 | 输入: points = [[1,2],[3,4],[5,6],[7,8]]  
2 | 输出: 4  
3 | 解释: 每个气球需要射出一支箭, 总共需要4支箭。
```

### 示例 3:

```
1 | 输入: points = [[1,2],[2,3],[3,4],[4,5]]  
2 | 输出: 2  
3 | 解释: 气球可以用2支箭来爆破:  
4 | - 在x = 2处发射箭, 击破气球[1,2]和[2,3]。  
5 | - 在x = 4处射出箭, 击破气球[3,4]和[4,5]。
```

### 提示:

- `1 <= points.length <= 105`
- `points[i].length == 2`
- `-231 <= xstart < xend <= 231 - 1`

```
1 | class Solution:  
2 |     def findMinArrowShots(self, points: List[List[int]]) -> int:  
3 |         # 按照右端点从小到大排序  
4 |         points.sort(key=lambda x: x[1])  
5 |  
6 |         res = 0  
7 |         ed = -sys.maxsize  
8 |  
9 |         for v in points:  
10 |             if ed < v[0]:  
11 |                 res += 1  
12 |                 ed = v[1]  
13 |  
14 |         return res
```

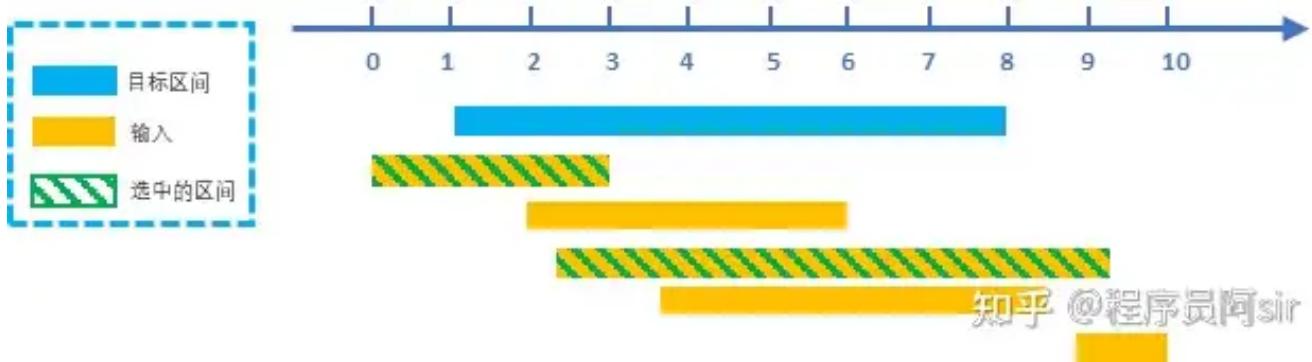
## 4 区间覆盖问题

### 4.1 题意描述

区间覆盖问题大概题意就是：

给出一堆区间和一个目标区间，问最少选择多少区间可以覆盖掉题中给出的这段目标区间。

如下图所示：



区间覆盖问题示例，最终至少选择2个区间才能覆盖目标区间

### 4.2. 解题步骤

**【步骤一】**：按照区间左端点从小到大排序。

**步骤二】**：从前往后依次枚举每个区间，在所有能覆盖当前目标区间起始位置start的区间之中，选择右端点最大的区间。

假设右端点最大的区间是第 $i$ 个区间，右端点为 $r_i$ 。

最后将目标区间的start更新成 $r_i$

## M01328: Radar Installation

greedy, <http://cs101.openjudge.cn/pctbook/M01328/>

Assume the coasting is an infinite straight line. Land is in one side of coasting, sea in the other. Each small island is a point locating in the sea side. And any radar installation, locating on the coasting, can only cover  $d$  distance, so an island in the sea can be covered by a radius installation, if the distance between them is at most  $d$ .

We use Cartesian coordinate system, defining the coasting is the x-axis. The sea side is above x-axis, and the land side below. Given the position of each island in the sea, and given the distance of the coverage of the radar installation, your task is to write a program to find the minimal number of radar installations to cover all the islands. Note that the position of an island is represented by its x-y coordinates.

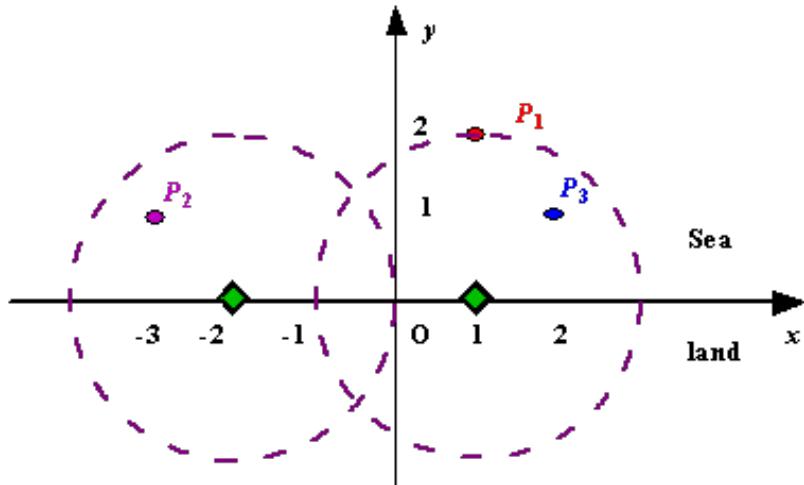


Figure A Sample Input of Radar Installations

### 输入

The input consists of several test cases. The first line of each case contains two integers  $n$  ( $1 \leq n \leq 1000$ ) and  $d$ , where  $n$  is the number of islands in the sea and  $d$  is the distance of coverage of the radar installation. This is followed by  $n$  lines each containing two integers representing the coordinate of the position of each island. Then a blank line follows to separate the cases.

The input is terminated by a line containing pair of zeros

### 输出

For each test case output one line consisting of the test case number followed by the minimal number of radar installations needed. "-1" installation means no solution for that case.

### 样例输入

```

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

```

### 样例输出

```

1 Case 1: 2
2 Case 2: 1

```

来源: Beijing 2002

映射到x轴，排序，左右端点互相看看。程序逻辑解释：

1. 计算岛屿的覆盖区间：对于每个岛屿，先根据其x和y坐标计算出在x轴上的区间范围。这是通过

$\sqrt{d^2 - y^2}$ 来确定的。

2. **排序**: 将所有岛屿的区间按照右端点进行排序，目的是尽可能让新的雷达覆盖更多的岛屿。
3. **更新覆盖范围**: 逐个岛屿进行遍历，尝试更新当前雷达能够覆盖的最远点。如果当前岛屿的左端点在已经覆盖的区间之外，则必须增加一个新的雷达，并将新的覆盖范围设为当前岛屿的区间。

```
1 import math
2
3 def solve(n, d, islands):
4     if d < 0:
5         return -1
6
7     ranges = []
8     for x, y in islands:
9         if y > d:
10            return -1
11     delta = math.sqrt(d * d - y * y)
12     ranges.append((x - delta, x + delta))
13
14     if not ranges:
15         return -1
16
17     ranges.sort(key=lambda x:x[1])
18
19     number = 1
20     r = ranges[0][1]
21     for start, end in ranges[1:]:
22         if r < start:
23             r = end
24             number += 1
25
26     return number
27
28 case_number = 0
29 while True:
30     n, d = map(int, input().split())
31     if n == 0 and d == 0:
32         break
33
34     case_number += 1
35     islands = []
36     for _ in range(n):
37         islands.append(tuple(map(int, input().split())))
38
39     result = solve(n, d, islands)
40     print(f"Case {case_number}: {result}")
41     input()
```

## 练习LC M1024. 视频拼接

<https://leetcode.cn/problems/video-stitching/>

你将会获得一系列视频片段，这些片段来自于一项持续时长为 `time` 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

使用数组 `clips` 描述所有的视频片段，其中 `clips[i] = [starti, endi]` 表示：某个视频片段开始于 `starti` 并于 `endi` 结束。

甚至可以对这些片段自由地再剪辑：

- 例如，片段 `[0, 7]` 可以剪切成 `[0, 1] + [1, 3] + [3, 7]` 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段 (`[0, time]`)。返回所需片段的最小数目，如果无法完成该任务，则返回 `-1`。

### 示例 1：

```
1 | 输入: clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], time = 10
2 | 输出: 3
3 | 解释:
4 | 选中 [0,2], [8,10], [1,9] 这三个片段。
5 | 然后, 按下面的方案重制比赛片段:
6 | 将 [1,9] 再剪辑为 [1,2] + [2,8] + [8,9] 。
7 | 现在手上的片段为 [0,2] + [2,8] + [8,10], 而这些覆盖了整场比赛 [0, 10]。
```

### 示例 2：

```
1 | 输入: clips = [[0,1],[1,2]], time = 5
2 | 输出: -1
3 | 解释:
4 | 无法只用 [0,1] 和 [1,2] 覆盖 [0,5] 的整个过程。
```

### 示例 3：

```
1 | 输入: clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],
2 | [2,6],[3,4],[4,5],[5,7],[6,9]], time = 9
3 | 输出: 3
4 | 解释:
5 | 选取片段 [0,4], [4,7] 和 [6,9] 。
```

### 提示：

- `1 <= clips.length <= 100`
- `0 <= starti <= endi <= 100`
- `1 <= time <= 100`

```

1  from typing import List
2
3  class Solution:
4      def videoStitching(self, clips: List[List[int]], time: int) -> int:
5          # 对 clips 按起点升序排序
6          clips.sort()
7
8          st, ed = 0, time
9          res = 0
10
11         i = 0
12         while i < len(clips) and st < ed:
13             maxR = 0
14             # 找到所有起点小于等于 st 的片段，并记录这些片段的最大终点 maxR
15             while i < len(clips) and clips[i][0] <= st:
16                 maxR = max(maxR, clips[i][1])
17                 i += 1
18
19             if maxR <= st:
20                 # 无法继续覆盖
21                 return -1
22
23             # 更新 st 为 maxR，并增加结果计数
24             st = maxR
25             res += 1
26
27             if maxR >= ed:
28                 # 已经覆盖到终点
29                 return res
30
31         # 如果没有成功覆盖到终点
32         return -1

```

## 练习T27104: 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  $N \leq 500000$ ,  $0 \leq a_i \leq N$

### 输出

一个整数，即最少需要装的摄像头数量。

#### 样例输入

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

#### 样例输出

1	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

是典型的“区间覆盖最小集合”问题，本质上是贪心算法在一维线段上的最优覆盖。

下面先解析题意→说明贪心思想→再给出代码。

---

### 题意解析

有  $N$  个窝在一条线上，每个窝  $i$  对应一个整数 ( $a_i$ )。

如果在窝  $i$  装摄像头，它能覆盖区间：

$[i - a_i, , i + a_i]$

目标：用尽量少的摄像头，使区间  $[1, N]$  被完全覆盖。

约束：

- $(N \leq 5 \times 10^5)$
- $(0 \leq a_i \leq N)$
- 必须线性或  $O(N \log N)$  算法。

---

### 💡 核心思想：区间覆盖贪心 (Interval Covering Greedy)

等价于：

给定一系列区间  $[L_i, R_i]$ ，要覆盖  $[1, N]$ ，求最少区间数量。

经典贪心策略：

1. 按 左端点 L 升序 排序。
2. 用一个变量 `rightmost` 表示当前已覆盖的最右端。
3. 在能“接上当前覆盖区间”的所有候选区间中，选右端点 R 最大的那个。
4. 每当 `rightmost` 无法继续延伸时，新增一个区间（摄像头）。

时间复杂度  $O(N \log N)$ ，如果不排序、直接预处理每个位置的最远可达右端，也能做到  $O(N)$ 。

内存: 96740kB, 时间: 1040ms

```
1 n = int(input())
2 a = list(map(int, input().split()))
3
4 # 1. 计算每个位置的覆盖区间
5 intervals = []
6 for i, x in enumerate(a, start=1):
7     left = max(1, i - x)
8     right = min(n, i + x)
9     intervals.append((left, right))
10
11 # 2. 按左端点排序
12 intervals.sort()
13
14 # 3. 贪心选择最少区间覆盖 [1, N]
15 ans = 0
16 i = 0
17 covered = 0
18
19 while covered < n:
20     best = covered
21     # 在所有能接上的区间中，选右端点最大的
22     while i < n and intervals[i][0] <= covered + 1:
23         best = max(best, intervals[i][1])
24         i += 1
25     if best == covered: # 无法延伸
26         break
27     covered = best
28     ans += 1
29
30 print(ans)
```

思路说明

- `covered` 是当前覆盖的最右边界。
- 每次扩展时：
  - 找所有  $L_i \leq covered + 1$  的区间。
  - 从中选  $R_i$  最大的那个。
  - 更新 `covered = R_i`。

- 计数器 `ans += 1`。
- 重复直到 `covered ≥ N`。

### 若需纯 O(N) 版本（无需排序）

利用「右端最远覆盖数组」：

内存: 59472kB, 时间: 627ms

```

1 n = int(input())
2 a = list(map(int, input().split()))
3
4 far = [0] * (n + 2)
5 for i, x in enumerate(a, start=1):
6     L = max(1, i - x)
7     R = min(n, i + x)
8     far[L] = max(far[L], R)
9
10 ans = 0
11 covered = 0
12 best = 0
13
14 for i in range(1, n + 1):
15     best = max(best, far[i])
16     if i > covered:
17         ans += 1
18         covered = best
19
20 print(ans)

```

这个写法等价于“跳台阶式区间扩展”，非常精炼、线性复杂度，是此类题的最优模板。

### ✓ 总结对比

特征	优化版 ( $O(N \log N)$ )	优化版 ( $O(N)$ )
核心思路	区间覆盖标准贪心	线性扫描右端最远覆盖
复杂度	$O(N \log N)$	$O(N)$
代码长度	20行	15行
可读性	★★★★★	★★★★★★
是否易记	是	是（推荐模板）

### 建议

今后遇到类似题型（如“最少灯照亮道路”“覆盖线段”等），直接记下面这两种模板：

- **排序版**：适合任意输入区间。
- **线性版**：适合“按位置定义区间”的题。

## 5 区间分组问题

### 5.1 题意描述

区间分组问题大概题意就是：给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。

如下图所示：



区间分组问题示例，最少分成3个组

### 5.2. 解题步骤

**【步骤一】**：按照区间左端点从小到大排序。

**【步骤二】**：从前往后依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

(即判断是否存在某个组的右端点在当前区间之中。如果可以，则不能放到这一组)

假设现在已经分了  $m$  组了，第  $k$  组最右边的一个点是  $r_k$ ，当前区间的范围是  $[L_i, R_i]$ 。则：

如果  $L_i \leq r_k$  则表示第  $i$  个区间无法放到第  $k$  组里面。反之，如果  $L_i > r_k$ ，则表示可以放到第  $k$  组。

- 如果所有  $m$  个组里面没有组可以接收当前区间，则当前区间新开一个组，并把自己放进去。
- 如果存在可以接收当前区间的组  $k$ ，则将当前区间放进去，并更新当前组的  $r_k = R_i$ 。

注意：

为了能快速的找到能够接收当前区间的组，我们可以使用**优先队列（小顶堆）**。

优先队列里面记录每个组的右端点值，每次可以在  $O(1)$  的时间拿到右端点中的的最小值。

## 练习NC147 主持人调度

<https://www.nowcoder.com/questionTerminal/4edf6e6d01554870a12f218c94e8a299>

有  $n$  个活动即将举办，每个活动都有开始时间与活动的结束时间，第  $i$  个活动的开始时间是  $\text{start}_i$ ，第  $i$  个活动的结束时间是  $\text{end}_i$ ，举办某个活动就需要为该活动准备一个活动主持人。

一位活动主持人在同一时间只能参与一个活动。并且活动主持人需要全程参与活动，换句话说，一个主持人参与了第  $i$  个活动，那么该主持人在  $(\text{start}_i, \text{end}_i)$  这个时间段不能参与其他任何活动。求为了成功举办这  $n$  个活动，最少需要多少名主持人。

数据范围:  $1 \leq n \leq 10^5$ ,  $-2^{32} \leq \text{start}_i \leq \text{end}_i \leq 2^{31}$

复杂度要求: 时间复杂度  $O(n \log n)$ ，空间复杂度  $O(n)$

示例1

输入

```
1 | 2,[[1,2],[2,3]]
```

输出

```
1 | 1
```

说明

```
1 | 只需要一个主持人就能成功举办这两个活动
```

示例2

输入

```
1 | 2,[[1,3],[2,4]]
```

输出

```
1 | 2
```

说明

```
1 | 需要两个主持人才能成功举办这两个活动
```

备注:

```
1 | 1≤n≤10^5  
2 | starti,endi在int范围内
```

## 解法1：将每个活动的开始时间和结束时间转换为事件

```
1  from typing import List
2
3  class Solution:
4      def minimumNumberOfHost(self, n: int, startEnd: List[List[int]]) -> int:
5          # 将每个活动的开始时间和结束时间转换为事件
6          events = []
7          for i in range(n):
8              start, end = startEnd[i]
9              events.append((start, 1)) # 活动开始, +1主持人
10             events.append((end, -1)) # 活动结束, -1主持人
11
12         # 对事件按照时间排序, 如果时间相同, 先处理结束事件
13         events.sort(key=lambda x: (x[0], x[1]))
14
15         min_hosts = 0
16         current_hosts = 0
17
18         # 遍历所有事件, 计算需要的主持人数
19         for time, event in events:
20             current_hosts += event
21             min_hosts = max(min_hosts, current_hosts)
22
23         return min_hosts
24
25 # 示例用法
26
27 #sol = Solution()
28 #print(sol.minimum_number_of_host(3, [[1, 5], [2, 7], [4, 5]])) # 输出应为 2
29 #print(sol.minimum_number_of_host(34,[[547,612],[417,551],[132,132],[168,446],
30 [95,747],[187,908],[115,712],[15,329],[612,900],[3,509],[181,200],[562,787],
31 [136,268],[36,784],[533,573],[165,946],[343,442],[127,725],[557,991],[604,613],
32 [633,721],[287,847],[414,480],[428,698],[437,616],[475,932],[652,886],[19,992],
33 [132,543],[390,869],[754,903],[284,925],[511,951],[272,739]]))
```

## 解法2：

```
1  from typing import List
2  import heapq
3
4  class Solution:
5      def minimumNumberOfHost(self, n: int, startEnd: List[List[int]]) -> int:
6          # 按左端点从小到大排序
7          startEnd.sort(key=lambda x: x[0])
8
9          # 创建小顶堆
10         q = []
```

```

12     for i in range(n):
13         if not q or q[0] > startEnd[i][0]:
14             heapq.heappush(q, startEnd[i][1])
15         else:
16             heapq.heappop(q)
17             heapq.heappush(q, startEnd[i][1])
18
19     return len(q)

```

解法3：

将活动开始时间写入一个列表starts，进行排序。

将活动结束时间写入一个列表ends，进行排序。

每次活动开始时，需要增加一个主持人上场，每次活动结束时候可以释放一个主持人。

所以按照时间先后顺序对starts进行遍历，每次有活动开始count++，每次有活动结束count--

在count最大的时候，即是需要主持人最多的时候

```

1 class Solution:
2     def minimumNumberOfHosts(self , n , startEnd ):
3         starts=[ ]
4         ends=[ ]
5         for start,end in startEnd:
6             starts.append(start);
7             ends.append(end);
8
9         starts.sort();
10        ends.sort()
11
12        i,j,count,res=0,0,0,0
13        for time in starts:
14            while(i<n and starts[i]<=time):
15                i+=1
16                count+=1
17            while(j<n and ends[j]<=time):
18                j+=1
19                count-=1
20            if res<count:
21                res=count
22        return res

```

# 三、Python十大排序算法源码

Logs:

2024/10/22 取自, [https://github.com/GMyhf/2024spring-cs201/blob/main/code/ten\\_sort\\_algorithm.s.md](https://github.com/GMyhf/2024spring-cs201/blob/main/code/ten_sort_algorithm.s.md)

## 1 前言

经常用到各种排序算法，但是网上的Python排序源码质量参差不齐。因此结合网上的资料和个人理解，整理了一份可直接使用的排序算法Python源码。

包括：冒泡排序（Bubble Sort），插入排序（Insertion Sort），选择排序（Selection Sort），希尔排序（Shell Sort），归并排序（Merge Sort），快速排序（Quick Sort），堆排序（Heap Sort），计数排序（Counting Sort），桶排序（Bucket Sort），基数排序（Radix Sort）

## 2 排序算法的选取规则

选择合适的排序算法取决于多种因素，包括数据的规模、特性、性能要求、稳定性要求、内存限制等。

### 数据规模

小规模，通常指数据量在几千到几万个元素。冒泡排序、插入排序、选择排序。

中规模数据，通常指数据量在几万到几百万个元素。希尔排序、快速排序、归并排序。

大规模数据，通常指数据量在几百万到几亿甚至更多个元素。归并排序、快速排序、堆排序、外部排序、分布式排序。

### 数据特性

几乎有序：插入排序。

数据范围小：计数排序。

数据分布均匀：桶排序。

固定长度的整数或字符串：基数排序。

### 性能要求

高时间效率：归并排序、快速排序、堆排序。

低空间复杂度：选择排序、堆排序。

### 稳定性要求

需要稳定排序：归并排序、计数排序、基数排序、桶排序、插入排序、冒泡排序。

### 内存限制

内存有限：选择排序、堆排序。

### Comparison sorts

在排序算法中，稳定性是指相等元素的相对顺序是否在排序后保持不变。换句话说，如果排序算法在排序过程中保持了相等元素的相对顺序，则称该算法是稳定的，否则是不稳定的。

对于判断一个排序算法是否稳定，一种常见的方法是观察交换操作。挨着交换（相邻元素交换）是稳定的，而隔着交换（跳跃式交换）可能会导致不稳定性。

Below is a table of [comparison sorts](#). A comparison sort cannot perform better than  $O(n \log n)$  on average.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Merge sort	$n \log n$	$n \log n$	$n \log n$	$n$	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarian's Algorithm)
Timsort	$n$	$n \log n$	$n \log n$	$n$	Yes	Insertion & Merging	Makes $n-1$ comparisons when the data is already sorted or reverse sorted.
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	$n$	$n^2$	$n^2$	1	Yes	Insertion	$O(n + d)$ , in the worst case over sequences that have $d$ inversions.
Bubble sort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	Tiny code size.
Selection sort	$n^2$	$n^2$	$n^2$	1	No	Selection	Stable with $O(n)$ extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

Highly tuned implementations use more sophisticated variants, such as [Timsort](#) (merge sort, insertion sort, and additional logic), used in [Android](#), [Java](#), and [Python](#), and [introsort](#) (quicksort and heapsort), used (in variant forms) in some [C++ sort](#) implementations and in [.NET](#).

计数排序，时间复杂度： $O(n + k)$ ，其中  $k$  是数据范围。空间复杂度： $O(k)$ 。稳定。适用于数据范围较小且数据分布均匀的情况。

桶排序，时间复杂度：平均情况  $O(n + k)$ ，最坏情况  $O(n^2)$ 。空间复杂度： $O(n + k)$ 。稳定。适用于数据分布均匀且已知数据范围的情况。

基数排序，时间复杂度： $O(nk)$ ，其中  $k$  是数字的位数。空间复杂度： $O(n + k)$ 。稳定。适用于数据范围较大但位数较少的情况，例如固定长度的整数或字符串。

## 3 十大排序算法的Python源码

### 3.1 冒泡排序(Bubble Sort)

方法：通过重复地遍历要排序的列表，比较相邻的元素并根据需要交换它们的位置来实现排序。（比较次数多，交换次数多）

主要思想：前后两两比较，大小顺序错误就交换位置

代码思路：

1. 比较相邻元素，如果前者大于后者，就交换位置。
2. 从队首到队尾，每一对相邻元素都重复上述步骤，最后一个元素为最大元素。
3. 针对前n-1个元素重复。

```
1 def BubbleSort(arr):  
2     for i in range(len(arr) - 1):  
3         for j in range(len(arr) - i - 1):  
4             if arr[j] > arr[j + 1]:  
5                 arr[j], arr[j + 1] = arr[j + 1], arr[j]  
6     return arr  
7  
8 if __name__ == "__main__":  
9     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,  
10    20, 17, 10]  
11    print(arr_in)  
12    arr_out = BubbleSort(arr_in)  
13    print(arr_out)
```

时间复杂度：平均和最坏情况  $O(n^2)$ ，最好情况  $O(n)$

空间复杂度： $O(1)$

稳定排序。适用于小规模数据或几乎有序的数据。

改进后的冒泡排序是对原始冒泡排序的一种优化。原始冒泡排序的基本思想是依次比较相邻的两个元素，如果它们的顺序错误就交换它们，直到没有需要交换的元素为止。这样的算法效率较低，因为即使序列已经有序，它仍然需要进行多轮的比较和交换。

改进后的冒泡排序通过增加一个标志位来优化。在每一轮比较中，如果没有发生任何交换，说明序列已经有序，不需要再进行后续的比较，因此可以提前结束排序过程。

改进后的冒泡排序实现如下所示：

```
1 def bubble_sort(arr):  
2     n = len(arr)  
3     for i in range(n):  
4         # 标记是否发生了交换  
5         swapped = False  
6         for j in range(0, n - i - 1):  
7             if arr[j] > arr[j + 1]:  
8                 # 交换元素  
9                 arr[j], arr[j + 1] = arr[j + 1], arr[j]  
10                swapped = True  
11                # 如果没有发生交换，说明数组已经排序完成  
12                if not swapped:
```

```
13     break
14 return arr
```

在这个改进后的冒泡排序算法中，如果在一轮比较中没有发生任何交换，就将标志位 `swapped` 设置为 `False`，并提前跳出循环，从而减少了不必要的比较次数，提高了效率。

## 3.2 选择排序(*Selection Sort*)

方法：在无序区找到最小的元素放到有序区的队尾（比较次数多，交换次数少）

主要思想：水果摊挑苹果，先选出最大的，再选出次大的，直到最后。

选择是对冒泡的优化，比较一轮只交换一次数据。

代码思路：

1. 找到无序待排序列中最小的元素，和第一个元素交换位置。
2. 剩下的待排无序序列（ $2-n$ ）选出最小的元素，和第二个元素交换位置。
3. 直到最后选择完成。

```
1 def SelectSort(arr):
2     for i in range(len(arr)):
3         minIndex = i
4         for j in range(i + 1, len(arr)):
5             if arr[j] < arr[minIndex]:
6                 minIndex = j
7             arr[i], arr[minIndex] = arr[minIndex], arr[i]
8     return arr
9
10 if __name__ == "__main__":
11     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
12     20, 17, 10]
13     print(arr_in)
14     arr_out = SelectSort(arr_in)
15     print(arr_out)
```

时间复杂度： $O(n^2)$

空间复杂度： $O(1)$

非稳定排序

## 3.3 插入排序(*Insertion Sort*)

方法：把无序区的第一个元素插入到有序区的合适位置（比较次数少，交换次数多）

主要思想：扑克牌打牌时的插入思想，逐个插入到前面的有序数中。

代码思路：

1. 选择待排无序序列的第一个元素作为有序数列的第一个元素。
2. 把第2个元素到最后一个元素看做无序待排序列。

3. 依次从待排无序序列取出每一个元素，与有序序列的每个元素比较（从右向左扫描），符合条件交换元素位置。

```
1 def InsertSort(arr):
2     for i in range(1, len(arr)):
3         for j in range(i, 0, -1):
4             if arr[j] < arr[j - 1]:
5                 arr[j], arr[j - 1] = arr[j - 1], arr[j]
6     return arr
7
8 if __name__ == "__main__":
9     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
10    20, 17, 10]
11    print(arr_in)
12    arr_out = InsertSort(arr_in)
13    print(arr_out)
```

时间复杂度:  $O(n^2)$

空间复杂度:  $O(1)$

稳定排序

上面代码并没有在找到正确位置后立即停止循环，而是一直循环直到内部的 for 循环完成。

改进后的插入排序应该在找到正确位置后立即停止循环。要实现这一点，可以在内部的 for 循环中添加一个判断条件来判断是否需要继续交换。如果当前元素已经大于（或等于）前一个元素，就可以停止内部的循环了。

下面是一个改进的插入排序版本：

```
1 def InsertSort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5         while j >= 0 and key < arr[j]:
6             arr[j + 1] = arr[j]
7             j -= 1
8         arr[j + 1] = key
9     return arr
10
11 if __name__ == "__main__":
12     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
13    20, 17, 10]
14     print(arr_in)
15     arr_out = InsertSort(arr_in)
16     print(arr_out)
```

这个版本的插入排序算法在找到正确位置后会立即停止内部的循环，从而提高了效率。

## 3.4 希尔排序(Shell Sort)

希尔排序是插入排序的一种更高效的改进版本，其核心思想是将待排序数组分割成若干个子序列，然后对各个子序列进行插入排序，最后再对整个序列进行一次插入排序。希尔排序的关键在于选择合适的间隔序列，以保证最终的排序效率。

代码思路：

1. 选择一个增量序列 $t_1, t_2, \dots, t_k$ , 其中 $t_i > t_j$ ,  $t_k = 1$ 。
2. 按增量序列个数 $k$ , 对序列进行 $k$ 趟排序。
3. 每趟排序, 根据对应的增量 $t_i$ , 将待排序序列分割成若干长度为 $m$ 的子序列, 分别对各子表进行直接插入排序。仅增量因子为1时, 整个序列作为一个表来处理, 表长度即为整个序列的长度。

```
1 def ShellSort(arr):  
2     n = len(arr)  
3     gap = n // 2  
4     while gap > 0:  
5         for i in range(gap, n):  
6             temp = arr[i]  
7             j = i  
8             while j >= gap and arr[j - gap] > temp:  
9                 arr[j] = arr[j - gap]  
10                j -= gap  
11                arr[j] = temp  
12            gap //= 2  
13     return arr  
14  
15 if __name__ == "__main__":  
16     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,  
17     20, 17, 10]  
18     print(arr_in)  
19     arr_out = ShellSort(arr_in)  
20     print(arr_out)
```

空间复杂度:  $O(1)$

非稳定排序

希尔排序的时间复杂度取决于所使用的增量序列。没有一个统一的最佳增量序列，不同的增量序列会导致不同的时间复杂度。然而，通常情况下，希尔排序的时间复杂度可以描述如下：

- 最坏情况时间复杂度:  $O(n^2)$ , 这通常发生在某些特定的增量序列上。
- 平均情况时间复杂度: 根据不同的增量序列, 希尔排序的平均时间复杂度可以有很大的变化, 但一般认为其优于简单的插入排序, 大约在 $O(n^{1.25})$ 到 $O(n^{1.6})$ 之间。
- 最佳情况时间复杂度: 如果数组已经是有序的, 或者接近有序, 那么希尔排序的时间复杂度可以接近线性, 即 $O(n)$ 。

值得注意的是, 对于一些特定的增量序列, 希尔排序的时间复杂度可以更接近于 $O(n \log n)$ , 但这并不普遍。因此, 希尔排序对于小到中等规模的数据集是一个不错的选择, 但对于非常大的数据集, 可能不如快速排序或归并排序等算法高效。希尔排序的空间复杂度为 $O(1)$ , 因为它是一种原地排序算法, 不需要额外的存储空间。

<https://pythontutor.com> 很好用，适合还不会用Pycharm调试工具的，当然后者也好用。另外就是print变量输出。

The screenshot shows the Python Tutor interface for visualizing code execution. On the left, the code for the `ShellSort` function is displayed:

```
1 def ShellSort(arr):
2     n = len(arr)
3     gap = n // 2
4     while gap > 0:
5         for i in range(gap, n):
6             temp = arr[i]
7             j = i
8             while j >= gap and arr[j - gap] > temp:
9                 arr[j] = arr[j - gap]
10                arr[j] = temp
11            j -= gap
12        gap //= 2
13    return arr
14
15 if __name__ == "__main__":
16     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9,
17     print(arr_in)
18     arr_out = ShellSort(arr_in)
19     print(arr_out)
```

The right side shows the state of variables and objects during the execution. The `arr` variable is a list containing the values: 6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9. The `ShellSort` function frame shows local variables: `n` (22), `gap` (11), `i` (12), `temp` (4), and `j` (11). The `Print output` box shows the sorted array: [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21].

## 3.5 归并排序(Merge Sort)

归并排序采用分治法，将待排序数组分成若干个子序列，分别进行排序，然后再合并已排序的子序列，直到整个序列都排好序为止。

代码思路：

1. 将待排序数组分成左右两个子序列，递归地对左右子序列进行归并排序。
2. 将两个已排序的子序列合并成一个有序序列。

```
1 def MergeSort(arr):
2     if len(arr) <= 1:
3         return arr
4     mid = len(arr) // 2
5     left = MergeSort(arr[:mid])
6     right = MergeSort(arr[mid:])
7     return merge(left, right)
8
9 def merge(left, right):
10    result = []
11    i = j = 0
12    while i < len(left) and j < len(right):
13        if left[i] < right[j]:
14            result.append(left[i])
15            i += 1
```

```

16         else:
17             result.append(right[j])
18             j += 1
19         result.extend(left[i:])
20         result.extend(right[j:])
21     return result
22
23 if __name__ == "__main__":
24     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
25     20, 17, 10]
26     print(arr_in)
27     arr_out = MergeSort(arr_in)
28     print(arr_out)

```

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$

稳定排序

递归程序运行过程, 不容易理解。<https://pythontutor.com>, 完美展示归并排序的递归过程。

The screenshot shows the Python Tutor interface visualizing the execution of the `MergeSort` function. The code on the left defines the `MergeSort` function and its helper `merge` function. The main part of the visualization shows the call stack and the state of the arrays at each step.

- Call Stack:** The call stack shows multiple frames of the `MergeSort` function. The top frame is for the initial call with `arr = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3, 20, 17, 10]`. Subsequent frames show the array being split into smaller halves: `[6, 5, 18]`, `[2, 16, 15]`, `[19, 13]`, `[10, 12]`, `[7, 9]`, `[4, 4]`, `[8, 1]`, `[11, 14]`, `[3, 20]`, and finally `[17, 10]`.
- Variables:** Global variables include `MergeSort` (function), `merge` (function), and `arr_in` (list). Local variables in each frame include `arr` (list) and `mid` (integer).
- Data Structures:** The arrays are shown as lists of integers, with some elements highlighted in yellow to indicate they are being compared or merged. Arrows show the flow of data from the left half to the right half during the merge process.
- Execution Details:** The bottom section shows the current step: `Step 44 of 694`. It indicates the `left` pointer is at index 0 and the `right` pointer is at index 1, with the `result` list containing `[0, 5, 6]`.

## 3.6 快速排序(Quick Sort)

快速排序是一种高效的排序算法, 采用分治法的思想, 通过将数组分割成较小的子数组, 然后分别对子数组进行排序, 最终将数组整合成有序序列。

代码思路:

1. 选择数组中的一个元素作为基准 (pivot)。
2. 将数组分割成两个子数组，使得左子数组中的所有元素都小于基准，右子数组中的所有元素都大于基准。
3. 对左右子数组递归地进行快速排序。

```

1 def QuickSort(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         pivot = arr[0] # Choose the first element as the pivot
6         left = [x for x in arr[1:] if x < pivot]
7         right = [x for x in arr[1:] if x >= pivot]
8         return QuickSort(left) + [pivot] + QuickSort(right)
9
10 if __name__ == "__main__":
11     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
12     20, 17, 10]
13     print(arr_in)
14     arr_out = QuickSort(arr_in)
15     print(arr_out)

```

时间复杂度：平均情况下为 $O(n \log n)$ ，最坏情况下为 $O(n^2)$ （当数组已经有序时）

空间复杂度：平均情况下为 $O(\log n)$ ，最坏情况下为 $O(n)$ （递归调用栈的深度）

不稳定排序

如果用双指针实现，在partition函数中用两个指针 `i` 和 `j` 的方式实现。

```

1 def quicksort(arr, left, right):
2     if left < right:
3         partition_pos = partition(arr, left, right)
4         quicksort(arr, left, partition_pos - 1)
5         quicksort(arr, partition_pos + 1, right)
6
7
8 def partition(arr, left, right):
9     i = left
10    j = right - 1
11    pivot = arr[right]
12    while i <= j:
13        while i <= right and arr[i] < pivot:
14            i += 1
15        while j >= left and arr[j] >= pivot:
16            j -= 1
17        if i < j:
18            arr[i], arr[j] = arr[j], arr[i]
19        if arr[i] > pivot:
20            arr[i], arr[right] = arr[right], arr[i]
21    return i

```

```

22
23
24 arr = [22, 11, 88, 66, 55, 77, 33, 44]
25 quicksort(arr, 0, len(arr) - 1)
26 print(arr)
27
28 # [11, 22, 33, 44, 55, 66, 77, 88]

```

## 3.7 堆排序(Heap Sort)

堆排序利用了堆这种数据结构的特性，将待排序数组构建成一个二叉堆，然后对堆进行排序。

代码思路：

1. 构建一个最大堆（或最小堆），将待排序数组转换成堆。
2. 从堆顶开始，每次将堆顶元素与堆的最后一个元素交换，然后重新调整堆。
3. 重复上述步骤，直到整个堆排序完成。

```

1 def heapify(arr, n, i):
2     largest = i
3     left = 2 * i + 1
4     right = 2 * i + 2
5
6     if left < n and arr[left] > arr[largest]:
7         largest = left
8     if right < n and arr[right] > arr[largest]:
9         largest = right
10
11    if largest != i:
12        arr[i], arr[largest] = arr[largest], arr[i]
13        heapify(arr, n, largest)
14
15 def HeapSort(arr):
16     n = len(arr)
17     for i in range(n // 2 - 1, -1, -1):
18         heapify(arr, n, i)
19     for i in range(n - 1, 0, -1):
20         arr[i], arr[0] = arr[0], arr[i]
21         heapify(arr, i, 0)
22     return arr
23
24 if __name__ == "__main__":
25     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
26     20, 17, 10]
27     print(arr_in)
28     arr_out = HeapSort(arr_in)
29     print(arr_out)

```

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(1)$

不稳定排序

## 3.8 计数排序(Counting Sort)

计数排序是一种非比较性的排序算法，适用于待排序数组的取值范围较小且已知的情况。该算法通过统计每个元素出现的次数，然后根据统计结果重构排序后的数组。

代码思路：

1. 统计数组中每个元素出现的次数，并存储在额外的计数数组中。
2. 根据计数数组中的统计结果，重构排序后的数组。

```
1 def CountingSort(arr):
2     max_value = max(arr)
3     count = [0] * (max_value + 1)
4     for num in arr:
5         count[num] += 1
6     sorted_arr = []
7     for i in range(max_value + 1):
8         sorted_arr.extend([i] * count[i])
9     return sorted_arr
10
11 if __name__ == "__main__":
12     arr_in = [6, 5, 18, 2, 16, 15, 19, 13, 10, 12, 7, 9, 4, 4, 8, 1, 11, 14, 3,
13     20, 17, 10]
14     print(arr_in)
15     arr_out = CountingSort(arr_in)
16     print(arr_out)
```

时间复杂度:  $O(n + k)$ , 其中n是数组的长度, k是数组中的最大值与最小值的差值

空间复杂度:  $O(n + k)$

## 3.9 桶排序(Bucket Sort)

桶排序是一种排序算法，它假设输入是由一个随机过程产生的，该过程将元素均匀、独立地分布在[0, 1)区间上。

代码思路：

1. 创建一个定量的桶数组，并初始化每个桶为空。
2. 将每个元素放入对应的桶中。
3. 对每个非空桶进行排序。
4. 从每个桶中将排序后的元素依次取出，得到排序结果。

```

1 def BucketSort(arr):
2     n = len(arr)
3     max_val = max(arr)
4     min_val = min(arr)
5     bucket_size = (max_val - min_val) / n
6
7     buckets = [[] for _ in range(n+1)]
8
9     for num in arr:
10         index = int((num - min_val) // bucket_size)
11         buckets[index].append(num)
12
13     sorted_arr = []
14     for bucket in buckets:
15         sorted_arr.extend(sorted(bucket))
16
17     return sorted_arr
18

```

```

19 if __name__ == "__main__":
20     arr_in = [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]
21     print(arr_in)
22     arr_out = BucketSort(arr_in)
23     print(arr_out)
24
25 #[0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897]

```

时间复杂度:  $O(n + k)$ , 其中n是数组的长度, k是桶的数量

空间复杂度:  $O(n)$

## 3.10 基数排序(*Radix Sort*)

基数排序是一种非比较型整数排序算法, 它通过按位处理数字来排序, 通常用于处理非负整数。

基数排序是一种多关键字的排序算法, 它将整数按位数切割成不同的数字, 然后按每个位数分别比较。

代码思路:

1. 找出数组中最大值, 并确定最大值的位数。
2. 使用计数排序或桶排序, 根据当前位数进行排序。

```

1 def RadixSort(arr):
2     max_val = max(arr)
3     digit = len(str(max_val))
4
5     for i in range(digit):
6         bucket = [[] for _ in range(10)]
7         for num in arr:
8             bucket[num // (10 ** i) % 10].append(num)
9         # for row in bucket:
10         #     print(*row)
11         arr = [num for sublist in bucket for num in sublist]
12         # arr = []
13         # for sublist in bucket:
14         #     for num in sublist:
15         #         arr.append(num)
16         #print(arr)
17
18     return arr
19
20 if __name__ == "__main__":
21     arr_in = [170, 45, 75, 90, 802, 24, 2, 66]
22     print(arr_in)
23     arr_out = RadixSort(arr_in)
24     print(arr_out)

```

时间复杂度:  $O(nk)$ , 其中n是数组的长度, k是最大值的位数

空间复杂度:  $O(n + k)$

## 实际运行示例

输入: [170, 45, 75, 90, 802, 24, 2, 66]

我们来模拟每一轮:

第 0 轮 (i=0, 按个位排序)

数字	个位	桶号
170	0	0
45	5	5
75	5	5
90	0	0
802	2	2
24	4	4
2	2	2
66	6	6

→ 合并后: [170, 90, 802, 2, 24, 45, 75, 66]

---

第 1 轮 (i=1, 按十位 (num//10 % 10) 排序)

数字	十位 (num//10 % 10)
170 → 7	
90 → 9	
802 → 0	
2 → 0	
24 → 2	
45 → 4	
75 → 7	
66 → 6	

→ 桶分布:

- bucket[0]: [802, 2]
- bucket[2]: [24]

- bucket[4]: [45]
- bucket[6]: [66]
- bucket[7]: [170, 75]
- bucket[9]: [90]

→ 合并后: [802, 2, 24, 45, 66, 170, 75, 90]

---

## 第 2 轮 (i=2, 按百位排序)

数字	百位 (num//100 % 10)
802 → 8	
2 → 0	
24 → 0	
45 → 0	
66 → 0	
170 → 1	
75 → 0	
90 → 0	

→ 桶分布:

- bucket[0]: [2, 24, 45, 66, 75, 90]
- bucket[1]: [170]
- bucket[8]: [802]

→ 合并后: [2, 24, 45, 66, 75, 90, 170, 802]

✓ 排序完成!

---

✓ 输出结果

```
1 | [170, 45, 75, 90, 802, 24, 2, 66]
2 | [2, 24, 45, 66, 75, 90, 170, 802]
```

# 四、单调栈 (monotonic stack)

## 1 Stack in Python

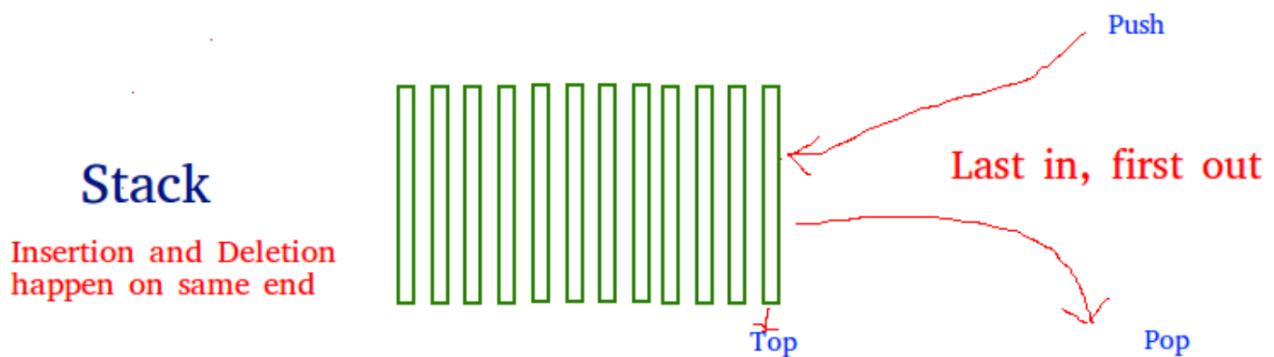
<https://www.geeksforgeeks.org/stack-in-python/>

### 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)

## 2 Monotonic Stack (单调栈)

单调栈是一种特殊的栈结构，其中的元素按照某种特定的顺序（如递增或递减）排列。在计算机科学中，单调栈常用于解决一类与数组或序列相关的优化问题，比如寻找下一个更大或更小的元素等。

### 单调栈的应用场景

1. 寻找下一个更大的元素：给定一个数组，对于每个元素，找到它右边第一个比它大的元素的位置。这类问题可以使用单调递减栈来高效解决。

2. 寻找下一个更小的元素：类似地，如果需要找到每个元素右边第一个比它小的元素的位置，则可以使用单调递增栈。
3. 直方图中的最大矩形：这是一个经典的问题，涉及到计算直方图中最大的矩形面积，可以使用单调栈来有效求解。
4. 滑动窗口的最大值：虽然这个问题通常使用双端队列来解决，但也可以通过单调栈的变形来处理。

### 单调栈的工作原理

- 入栈操作：当一个新的元素需要加入到栈中时，根据栈的性质（递增或递减），将所有不符合条件的栈顶元素弹出，然后再将新元素压入栈中。
- 出栈操作：通常情况下，出栈操作是自动发生的，即在执行入栈操作时，为了保持栈的单调性，会自动移除不满足条件的栈顶元素。

### 实现示例

这里以一个简单的例子说明如何使用单调栈来解决问题。假设我们需要找到数组 `[4, 5, 2, 25]` 中每个元素右边第一个更大的数。

```

1  def next_greater_element(nums):
2      stack = []
3      result = [0] * len(nums)
4
5      for i in range(len(nums)):
6          # 当栈不为空且当前考察的元素大于栈顶元素时
7          while stack and nums[i] > nums[stack[-1]]:
8              index = stack.pop()
9              result[index] = nums[i]
10             # 将当前元素的索引压入栈中
11             stack.append(i)
12
13         # 对于栈中剩余的元素，它们没有更大的元素
14         while stack:
15             index = stack.pop()
16             result[index] = -1
17
18     return result
19
20 nums = [4, 5, 2, 25]
21 print(next_greater_element(nums)) # 输出: [5, 25, 25, -1]

```

在这个例子中，我们维护了一个单调递减的栈，当遇到比栈顶元素大的数时，就找到了栈顶元素的“下一个更大的数”，然后将其从栈中弹出，并记录结果。最后，对于那些在栈中没有匹配到更大数的元素，它们的结果设置为 `-1`，表示没有更大的数。

## 3 编程题目

### 练习04137: 最小新整数

stack, greedy, <http://cs101.openjudge.cn/practice/04137/>

给定一个十进制正整数n(0 < n < 1000000000), 每个数位上数字均不为0。n的位数为m。

现在从m位中删除k位(0<k < m), 求生成的新整数最小为多少?

例如: n = 9128456, k = 2, 则生成的新整数最小为12456

## 输入

第一行t, 表示有t组数据;

接下来t行, 每一行表示一组测试数据, 每组测试数据包含两个数字n, k。

## 输出

t行, 每行一个数字, 表示从n中删除k位后得到的最小整数。

### 样例输入

```
1 | 2
2 | 9128456 2
3 | 1444 3
```

### 样例输出

```
1 | 12456
2 | 1
```

```
1 | # 蒋子轩23工学院
2 | def removeKDigits(num, k):
3 |     stack = []
4 |     for digit in num:
5 |         while k and stack and stack[-1] > digit:
6 |             stack.pop()
7 |             k -= 1
8 |         stack.append(digit)
9 |     # 如果还未删除k位, 从尾部继续删除
10 |    while k:
11 |        stack.pop()
12 |        k -= 1
13 |    return int(''.join(stack))
14 | t = int(input())
15 | results = []
16 | for _ in range(t):
17 |     n, k = input().split()
18 |     results.append(removeKDigits(n, int(k)))
19 | for result in results:
20 |     print(result)
```

## 练习21577: 护林员盖房子 加强版

matrix/implementation, <http://cs101.openjudge.cn/practice/21577>

在一片保护林中，护林员想要盖一座房子来居住，但他不能砍伐任何树木。  
现在请你帮他计算：保护林中所能用来盖房子的矩形空地的最大面积。

### 输入

保护林用一个二维矩阵来表示，长宽都不超过1000（即 $\leq 1000$ ）。

第一行是两个正整数 $m, n$ ，表示矩阵有 $m$ 行 $n$ 列。

然后是 $m$ 行，每行 $n$ 个整数，用1代表树木，用0表示空地。

### 输出

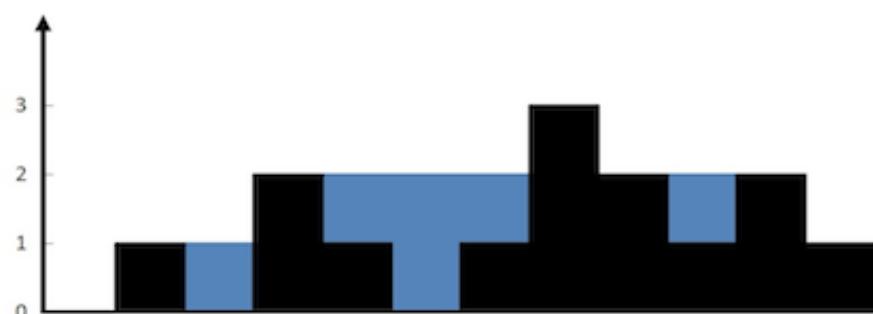
一个正整数，表示保护林中能用来盖房子的最大矩形空地面积。

## 练习T26977: 接雨水

stack, dp, math, <http://cs101.openjudge.cn/practice/26977/>

给定 $n$ 个非负整数表示每个宽度为1的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例：



height = [0,1,0,2,1,0,1,3,2,1,2,1]

由数组表示的高度图，在这种情况下，可以接6个单位的雨水（蓝色部分表示雨水）。

### 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。 $0 \leq \text{ratings}[i] \leq 2 * 10^5$

### 输出

一个整数

# 五、「滑动窗口最大值」三者合一

「滑动窗口最大值」问题——这是三者（双指针 + 滑动窗口 + deque）完美结合的代表。

## 1 问题描述

给你一个数组 `nums` 和一个整数 `k`，请输出每个长度为 `k` 的子数组的最大值。

例如：

```
1  nums = [1,3,-1,-3,5,3,6,7]
2  k = 3
3  输出: [3,3,5,5,6,7]
```

## 2 解法分层讲解

### 第一层：双指针思路

希望用两个指针维护一个区间 `[left, right]`：

- `right` 向右扩展窗口；
- `left` 根据需要右移缩小窗口；
- 每次窗口大小为 `k` 时，输出最大值。

但暴力求最大值会  $O(k)$ ，所以我们要想办法优化最大值维护。

### 第二层：滑动窗口结构

窗口的定义：

在任意时刻，`nums[left:right]` 就是当前窗口。

目标：

在每次窗口滑动一步时，高效地得到最大值。

于是引入一个辅助结构来维护窗口内的最大值——这时 `deque` 派上用场。

### 第三层：`deque` 维护单调性

用一个 单调递减队列（里面放的是下标，不是值）：

- 当右端加入新元素时，把所有比它小的元素都弹出（因为它们永远不会再成为最大值）；
- 当左端元素滑出窗口时，如果它正好是队首，就把它弹出。

这样，队首元素始终是当前窗口的最大值。

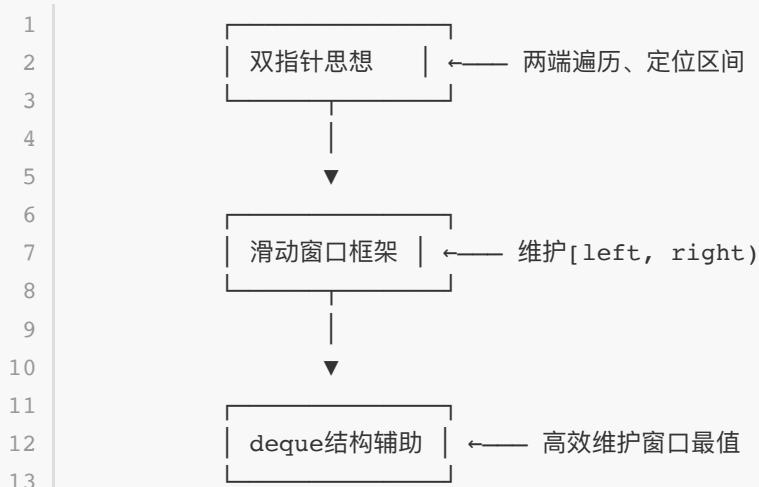
## 3 最终代码示例

```

1  from collections import deque
2
3  def maxSlidingWindow(nums, k):
4      dq = deque() # 存下标, 保证对应值递减
5      res = []
6      for right, x in enumerate(nums):
7          # step 1: 窗口右扩, 保持单调递减
8          while dq and nums[dq[-1]] <= x:
9              dq.pop()
10             dq.append(right)
11
12          # step 2: 移除滑出窗口的左端元素
13          if dq[0] <= right - k:
14              dq.popleft()
15
16          # step 3: 当窗口形成 (长度 >= k) 时, 记录最大值
17          if right >= k - 1:
18              res.append(nums[dq[0]])
19      return res
20
21 # ✅ 测试
22 print(maxSlidingWindow([1,3,-1,-3,5,3,6,7], 3))
23 # 输出: [3, 3, 5, 5, 6, 7]

```

## 4 三者关系图



### ✅ 一句话总结:

在「滑动窗口最大值」中,  
双指针定义窗口边界,  
滑动窗口描述窗口动态,  
deque维护窗口内部的最优状态。



# 六、前缀和优化区域统计

前缀和 (Prefix Sum) 是一种用于高效计算数组区间和的预处理技术。

## 1 一维前缀和

给定数组  $A[0..n-1]$ , 其前缀和数组  $P$  定义为:

- $P[0] = 0$
- $P[i] = A[0] + A[1] + \dots + A[i-1]$

这样, 区间  $[l, r]$  的和可以快速计算为:  $P[r+1] - P[l]$

## 2 二维前缀和

对于二维数组, 前缀和扩展为二维前缀和。 $\text{prefix}[i][j]$  表示从  $(0,0)$  到  $(i-1, j-1)$  矩形区域的元素和。

核心公式:

```
1 | prefix[i][j] = matrix[i-1][j-1] + prefix[i-1][j] + prefix[i][j-1] - prefix[i-1][j-1]
```

查询矩形区域  $(x_1, y_1)$  到  $(x_2, y_2)$  的和:

```
1 | sum = prefix[x2+1][y2+1] - prefix[x1][y2+1] - prefix[x2+1][y1] + prefix[x1][y1]
```

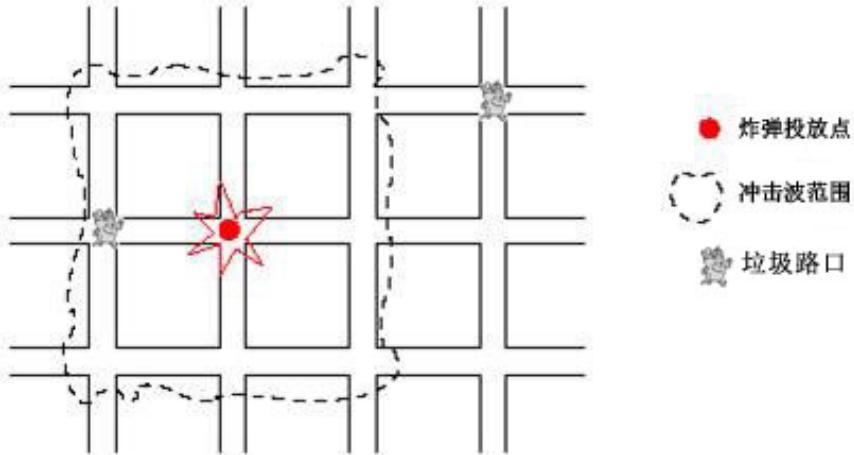
## 3 编程题目

### 练习M04133: 垃圾炸弹

matrices, <http://cs101.openjudge.cn/pctbook/M04133/>

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 $1025 \times 1025$ 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

```

**【夏子涵 元培学院】思路：**用二维前缀和数组避免多次暴力循环

这是一个典型的“用前缀和优化区域统计”问题，适合练习二维前缀和的应用。

```

1 | def main():
2 |     d = int(input().strip())
3 |     n = int(input().strip())
4 |

```

```

5      # 使用字典存储垃圾点，节省空间（稀疏数据时特别有效）
6      moskow = {}
7      max_coord = 1024
8      min_coord = 0
9
10     for _ in range(n):
11         x, y, weight = map(int, input().strip().split())
12         if (x, y) not in moskow:
13             moskow[(x, y)] = 0
14             moskow[(x, y)] += weight
15
16     # 构建二维前缀和数组（只构建 [0..1024] 范围）
17     size = max_coord + 1
18     prefix = [[0] * (size + 1) for _ in range(size + 1)] # prefix[0..1024+1]
19     [0..1024+1]
20
21     for i in range(1, size + 1):
22         for j in range(1, size + 1):
23             # 注意: prefix[i][j] 对应坐标 (i-1, j-1)
24             val = moskow.get((i - 1, j - 1), 0)
25             prefix[i][j] = val + prefix[i - 1][j] + prefix[i][j - 1] - prefix[i - 1][j - 1]
26
27     max_total = 0
28     count = 0
29
30     # 遍历所有可能的爆炸中心 (i, j)，范围 [0, 1024]
31     for i in range(min_coord, max_coord + 1):
32         for j in range(min_coord, max_coord + 1):
33             # 计算爆炸影响区域 [x1, x2] x [y1, y2]
34             x1 = max(min_coord, i - d)
35             y1 = max(min_coord, j - d)
36             x2 = min(max_coord, i + d)
37             y2 = min(max_coord, j + d)
38
39             # 查询前缀和：注意 prefix 是 1-indexed
40             total = prefix[x2 + 1][y2 + 1] \
41                     - prefix[x1][y2 + 1] \
42                     - prefix[x2 + 1][y1] \
43                     + prefix[x1][y1]
44
45             if total > max_total:
46                 max_total = total
47                 count = 1
48             elif total == max_total:
49                 count += 1
50
51
52
53     if __name__ == '__main__':
54         main()

```



# 七、Kadane 算法

## 1 理解 Kadane 算法（一维最大子数组和）

Kadane 算法用于解决“最大子数组和”问题，即在一个整数数组中找到连续子数组的最大和。

算法思想：

- `curr_max`：以当前元素结尾的最大连续和
- `total_max`：全局最大和

Python 实现：

```
1 def kadane(arr):  
2     curr_max = total_max = arr[0]  
3     for x in arr[1:]:  
4         curr_max = max(x, curr_max + x) # 要么重新开始，要么接上前面  
5         total_max = max(total_max, curr_max)  
6     return total_max
```

✓ 例子：[-2, 1, -3, 4, -1, 2, 1, -5, 4] → 最大子数组 [4, -1, 2, 1]，和为 6

## 2 扩展到二维 —— 最大子矩阵

将 Kadane 算法的思想扩展到二维：

总体策略：

1. 枚举所有可能的上边界 `top`
2. 对每个 `top`，枚举所有 `bottom >= top`
3. 对每一对 `(top, bottom)`，计算从第 `top` 行到第 `bottom` 行的每列的累加和，形成一个一维数组 `col_sum`
4. 在 `col_sum` 上运行 Kadane 算法，得到当前上下边界下的最大子矩阵和
5. 更新全局最大值

### 举例说明

原矩阵：

```
1  0 -2 -7  0  
2  9  2 -6  2  ← top=1  
3 -4  1 -4  1  
4 -1  8  0 -2  ← bottom=3
```

固定 `top=1, bottom=3`，计算每列的和：

- 第0列： $9 + (-4) + (-1) = 4$
- 第1列： $2 + 1 + 8 = 11$

- 第2列:  $-6 + (-4) + 0 = -10$
- 第3列:  $2 + 1 + (-2) = 1$

得到一维数组: [4, 11, -10, 1]

运行 Kadane:

- 最大子数组: [4, 11] 或 [11] → 和为 15

正好对应样例答案!

## 3 编程题目

### 练习M02766: 最大子矩阵

matrices, kadane, <http://cs101.openjudge.cn/pctbook/M02766/>

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

比如，如下 $4 * 4$ 的矩阵

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

的最大子矩阵是

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

这个子矩阵的大小是15。

#### 输入

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

#### 输出

输出最大子矩阵的大小。

#### 样例输入

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

## 样例输出

1 | 15

来源：翻译自 Greater New York 2001 的试题

```
1 def kadane(s):
2     curr_max = total_max = s[0]
3     for x in s[1:]:
4         curr_max = max(x, curr_max + x)
5         total_max = max(total_max, curr_max)
6     return total_max
7
8 def max_sum_matrix(mat):
9     max_sum = -float('inf')
10    row, col = len(mat), len(mat[0])
11    for top in range(row):
12        col_sum = [0] * col
13        for bottom in range(top, row):
14            for c in range(col):
15                col_sum[c] += mat[bottom][c]
16            max_sum = max(max_sum, kadane(col_sum))
17    return max_sum
18
19 n = int(input())
20 nums = []
21 while len(nums) < n**2:
22     nums.extend(input().split())
23 mat = [list(map(int, nums[i*n:(i+1)*n])) for i in range(n)]
24 print(max_sum_matrix(mat))
```

## 一、算法原理

二维最大子矩阵问题，可以通过「行压缩 + 一维 Kadane」解决。

### 思想：

- 固定子矩阵的上边界 `top`
- 固定子矩阵的下边界 `bottom`
- 对每一列求“从 `top` 到 `bottom` 行的列和”
  - 得到一个一维数组 `col_sum`
- 对 `col_sum` 使用 **Kadane** 算法，求最大子数组和（相当于固定上下边界后，在列方向找到左右边界）

### 举例：

```

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

```

比如 top=1, bottom=3, 则

`col_sum = [9-4-1, 2+1+8, -6-4+0, 2+1-2] = [4, 11, -10, 1]`

→ Kadane(`col_sum`) = 15, 对应矩阵正是样例输出。

## 二、核心代码结构分析

```

1 | def kadane(s):
2 |     curr_max = total_max = s[0]
3 |     for x in s[1:]:
4 |         curr_max = max(x, curr_max + x)
5 |         total_max = max(total_max, curr_max)
6 |     return total_max

```

一维 Kadane:

- `curr_max` 表示“以当前元素结尾的最大连续和”;
- `total_max` 表示“全局最大和”。

```

1 | def max_sum_matrix(mat):
2 |     max_sum = -float('inf')
3 |     row, col = len(mat), len(mat[0])
4 |     for top in range(row):
5 |         col_sum = [0] * col
6 |         for bottom in range(top, row):
7 |             for c in range(col):
8 |                 col_sum[c] += mat[bottom][c]
9 |             max_sum = max(max_sum, kadane(col_sum))
10 |    return max_sum

```

外层 `top` 和 `bottom` 控制上下边界 ( $O(n^2)$ ) ;

内层 Kadane 处理列方向 ( $O(n)$ ) ;

整体时间复杂度  **$O(n^3)$** , 适用于  $N \leq 100$  的题目。

## 三、输入处理逻辑说明

OJ 给的输入可能有空格和换行混合, 所以正确做法是——

不要逐行严格读取, 而是一直读取直到获取  $N^2$  个整数。

```

1 n = int(input())
2 nums = []
3 while len(nums) < n**2:
4     nums.extend(input().split())
5 mat = [list(map(int, nums[i*n:(i+1)*n])) for i in range(n)]
6 print(max_sum_matrix(mat))

```

#### 四、扩展思考（可选优化）

##### 1. 前缀和加速列和计算

- 可以预先构建行方向前缀和 `prefix[r][c]`
- 使得 `col_sum[c] = prefix[bottom][c] - prefix[top-1][c]`
- 时间复杂度仍然  $O(n^3)$ , 但常数项更小。

##### 2. 全负数矩阵

- Kadane 已正确处理（返回最大单个元素）。

##### 3. 空间复杂度

- $O(n)$ , 仅用到 `col_sum`。

这个问题是一个经典的“最大子矩阵和”问题，属于二维动态规划的应用场景。解决的核心思想是将二维问题降为多个一维的“最大子段和”问题（Kadane 算法），从而降低复杂度。

#### 解题思路（二维 Kadane 变种）

1. 固定两个行索引 `top` 和 `bottom`，将这两行之间（包含）的矩阵压缩成一个一维数组 `temp_col_sum`，其中每个元素是这几行中该列的和。
2. 在这个一维数组上应用“最大子段和”算法（Kadane）求出最大和。
3. 枚举所有可能的 `top` 和 `bottom` 组合，更新全局最大值。

#### 代码实现（Python）

```

1 ...
2 为了找到最大的非空子矩阵，可以使用动态规划中的Kadane算法进行扩展来处理二维矩阵。
3 基本思路是将二维问题转化为一维问题：可以计算出从第i行到第j行的列的累计和，
4 这样就得到了一个一维数组。然后对这个一维数组应用Kadane算法，找到最大的子数组和。
5 通过遍历所有可能的行组合，我们可以找到最大的子矩阵。
6 ...
7 def max_submatrix(matrix, n):
8     def kadane(arr):
9         # max_ending_here 用于追踪到当前元素为止包含当前元素的最大子数组和。
10        # max_so_far 用于存储迄今为止遇到的最大子数组和。
11        max_end_here = max_so_far = arr[0]
12        for x in arr[1:]:
13            # 对于每个新元素，我们决定是开始一个新的子数组（仅包含当前元素 x），
14            # 还是将当前元素添加到现有的子数组中。这一步是 Kadane 算法的核心。

```

```

15         max_end_here = max(x, max_end_here + x)
16         max_so_far = max(max_so_far, max_end_here)
17     return max_so_far
18
19     max_sum = float('-inf')
20
21     for top in range(n):
22         temp_col_num = [0] * n
23         for bottom in range(top, n):
24             for col in range(n):
25                 temp_col_num[col] += matrix[bottom][col]
26             max_sum = max(max_sum, kadane(temp_col_num))
27     return max_sum
28
29 # 输入处理
30 import sys
31 data = sys.stdin.read().split()
32 n = int(data[0])
33 numbers = list(map(int, data[1:]))
34 matrix = [numbers[i * n:(i + 1) * n] for i in range(n)]
35
36 max_sum = max_submatrix(matrix, n)
37 print(max_sum)

```

## 时间复杂度分析

- 外层双重循环 (`top` 和 `bottom`) :  $O(n^2)$
- 内层 Kadane:  $O(n)$
- 总体时间复杂度:  **$O(n^3)$** , 对于 `n <= 100` 是可接受的。

# End

---