同濟大學
TONGJI UNIVERSITY

算法分析与设计
期中大作业实验报告

学　　号：2154312
姓　　名：郑博远
教　　师：程大伟
完成日期：2023 年 4 月 21 日

1. 题目 1 第 k 小问题

1.1. 题目描述

假设有一组数据，每个数据都是一个字母和一个数字的组合。现在要求按照数字从小到大的顺序找出其中第 k 小的数据，写一个算法来解决这个问题。下面三种是可行的算法：

（1） 基于堆的选择：不需要对全部 n 个元素排序，只需要维护 k 个元素的最大堆，即用容量为 k 的最大堆存储最小的 k 个数，总费时 O(k + (n − k)logk)。

（2） 随机划分线性选择 (教材上的 RandomizedSelect)：在最坏的情况下时间复杂度为 $O(n^2)$ ，平均情况下期望时间复杂度为 O(n)。

（3） 利用中位数的线性时间选择：选择中位数的中位数作为划分的基准，在最坏情况下时间复杂度为 O(n)。

请给出以上三种算法的算法描述，用你熟悉的编程语言实现上述三种算法。并通过实际用例测试，给出三种算法的运行时间随 k 和 n 变化情况的对比图（表）。

1.2. 算法思想

1.2.1. 基于堆的选择

基于堆的选择可以在 O(k + (n − k)logk)的时间复杂度内找到第 k 小的数据。该算法的主要思想是使用一个容量为 k 的最大堆，存储最小的 k 个元素。在处理数据时，对于每个新的元素，如果它小于堆中最大元素，则将该元素插入堆中，并将堆中最大元素删除。这样，最终堆中剩下的 k 个元素就是前 k 个最小的元素。

下面是基于堆的选择的具体步骤：

- 对数据中的前 k 个数据，创建一个容量为 k 的最大堆；
- 对于第 k+1 到第 n 个元素，依次进行以下操作：如果该元素比堆中最大元素小，则将该元素插入堆中，并删除堆中最大元素；如果该元素比堆中最大元素大，则直接跳过该元素；
- 最终，堆中剩余的 k 个元素就是前 k 个最小的元素，其中堆顶元素即为第 k 小的数据。

基于堆的选择算法的时间复杂度为 O(k + (n − k)logk)，其中 O(k)为对堆中前 k 个数据建堆的时间复杂度，O((n − k)logk)为对之后元素插入堆中进行调整所带来的时间复杂度。

1.2.2. 随机划分线性选择

随机划分线性选择是一种快速选择算法，其思想类似于排序算法中的快速排序。在一般情况下，它可以在 O(n)的时间复杂度内找到第 k 小的数据；在最坏情况下，它能在 $O(n^2)$的时间复杂度内找到第 k 小的数据。该算法的主要思想是在数据集合中随机选择一个元素作为枢轴（pivot），将数据集合划分为两个子集：小于等于枢轴的元素集合和大于枢轴的元素集合。如果小于等于枢轴的元素的数量小于 k，则递归地在大于枢轴的元素集合中查找第 k-（小于等于枢轴的元素数量）小的元素；否则，在小于等于枢轴的元素集合中查找第 k 小的元素。

下面是随机划分线性选择的具体步骤：

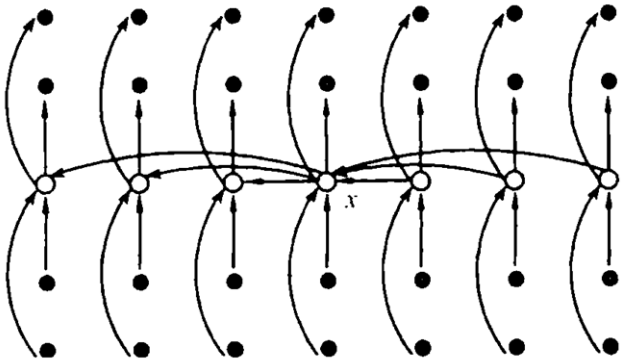
- 对于当前的数组，执行 RandomizedPartition。即随机选择一个枢轴，将数组 a[p:r]划分成两个子数组 a[p:i]和 a[i+l:r]，使 a[p:i]中每个元素都不小于等于枢轴元素的值，a[i+l:r]中每个元素都大于枢轴元素的值。
- 计算子数组 a[p:i]中元素个数 j，并递归地调用 RandomizedSelect 算法：若 k≤j，则 a[p:r]中第 k 小元素落在子数组 a[p:i]中；若 k>j，则要找的第 k 小元素落在子数组 a[i+l:r]中。由于此时已知道子数组 a[p]中元素均小于要找的第

k 小元素，因此要找的 a[p:r]中第 k 小元素是 a[i+l:r]中的第 k - j 小元素。

容易看出，在最坏情况下，算法 RandomizedSelect 需要 $\Omega(O^2)$计算时间。例如，在找最小元素时，总是在最大元素处划分，从而导致递归树退化成链。尽管如此，由于随机选择枢轴的过程是随机的，该算法的平均性能很好。

1.2.3. 利用中位数的线性时间选择

利用中位数的线性时间选择算法是一种优化版的随机划分线性选择算法，它能够保证在最坏的情况下以 O(n)的时间复杂度内找到第 k 小的元素。该算法的主要思想是在数据集合中选择一个中位数的中位数作为枢轴，将数据集合划分为两个子集，从而使得每一次划分都能尽可能将数组一分为二（实际上能让划分的子数组长度至少缩短四分之一）。



下面介绍中位数线性时间选择算法对于每次 Select 的枢轴选择：

· 将 n 个输入元素划分成⌈n/5⌉个组，每组 5 个元素，除可能有一个组不是 5 个元素外。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共⌈n/5⌉个。

· 递归调用 Select 找出这⌈n/5⌉个元素的中位数。如果⌈n/5⌉是偶数，就找它的两个中位数中较大的一个。然后以这个元素作为枢轴。

当数组长度小于 75 时，直接调用排序算法返回当前第 k 小的元素。综上所述可以得到关于 $T(n)$ 的表达递归式：

$$T(n) \leq \begin{cases} C_1 & n < 75 \\ C_2 n + T(n/5) + T(3n/4) & n \geq 75 \end{cases}$$

解递归式可得该算法的时间复杂度为 O(n)。

1.3. 选择的编程语言及环境

编程语言：C++

环境：Windows10　Microsoft Visual Studio 2022 Debug x86

1.4. 典型测试用例

输入数据格式：

第一行为两个数字，分别代表数据总数 n 以及所求的第 k 小数字 k；

接下来 n 行，每行一个数据，分别有一个字母和一个数字。

输出数据格式：

共一行，一个字母和一个数字，代表第 k 小的数据。

说明：由于算法不同，当数字大小相同时不会选择相同的字母；为方便观察，若在比较大小时数字相同，则**进一步以字母的 ASCII 码作为比较依据**。

1.4.1. 测试用例 1

输入数据	10000 6310 M 226 H 406 R 30 D 817 L 405 C 794 R 724 Y 345

	I 639 O 254 R 190 L 60 Z 795 G 786 Y 513 W 260 W 815 J 106 Z 717 J 344 D 967 P 822 U 63 E 112 P 575 I 632 I 832 F 938 H 773 A 625 （此处省略后续数据，详细可见 input1.txt）
正确结果	Q 631
算法 1 结果	Q 631
算法 2 结果	Q 631
算法 3 结果	Q 631

1.4.2. 测试用例 2

输入数据	1000000 524790

	M 636 S 640 Q 649 P 145 C 226 I 268 K 337 K 855 B 137 I 37 Z 628 T 578 X 634 U 24 F 393 O 673 C 744 D 942 X 369 X 679 Y 286 N 308 P 291 R 884 X 164 A 875 P 872 Q 65 M 510 Q 225 K 663 （此处省略后续数据，详细可见 input10.txt）
正确结果	T 521

算法 1 结果	T 521
算法 2 结果	T 521
算法 3 结果	T 521

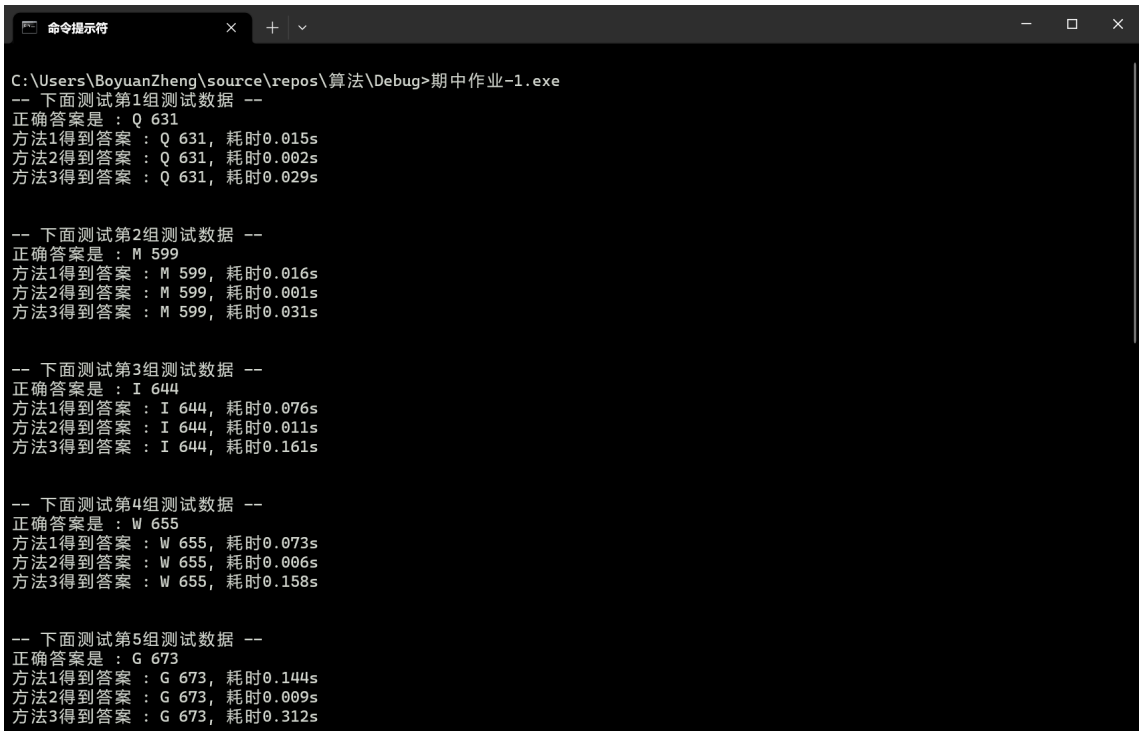
1.4.3. 测试用例 3

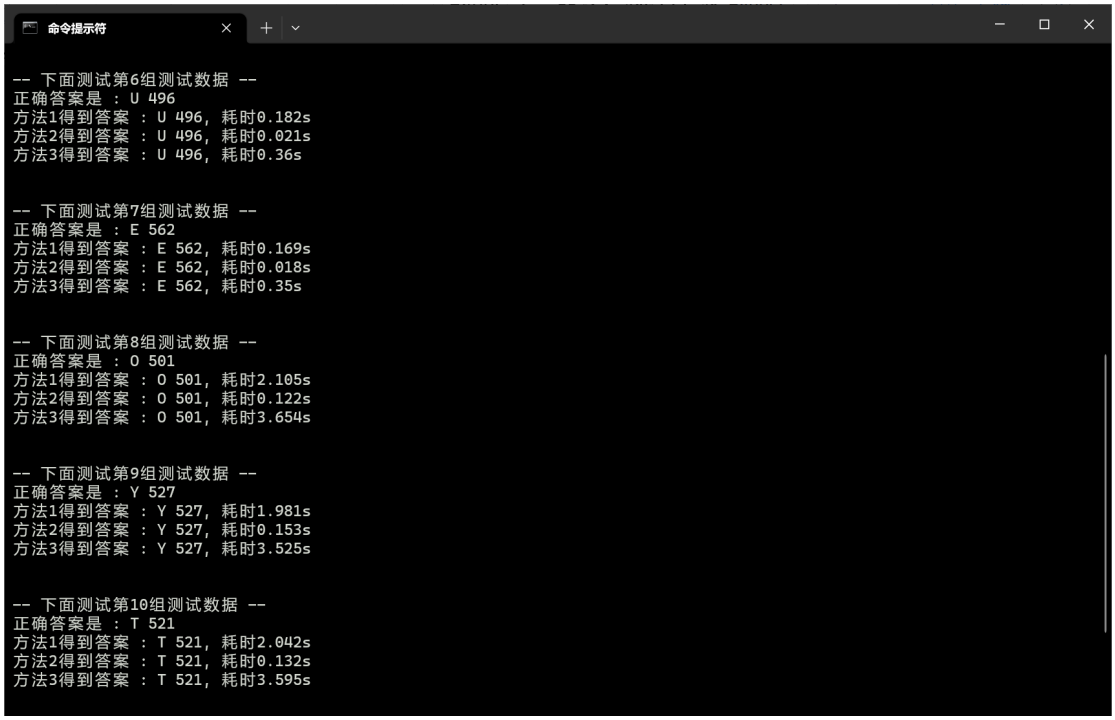
输入数据	1000 500 M 1000 L 999 K 998 J 997 I 996 H 995 G 994 F 993 E 992 D 991 C 990 B 989 A 988 Z 987 Y 986 X 985 W 984 V 983 U 982 T 981 S 980 R 979 Q 978 P 977 O 976 N 975

	M 974 L 973 K 972 J 971 （以此类推，此处省略后续数据）
正确结果	G 500
算法 1 结果	G 500
算法 2 结果	G 500
算法 3 结果	G 500

1.5. 系统输入输出运行结果截图

（输入文件见附件中的 input1.txt 至 input11.txt）

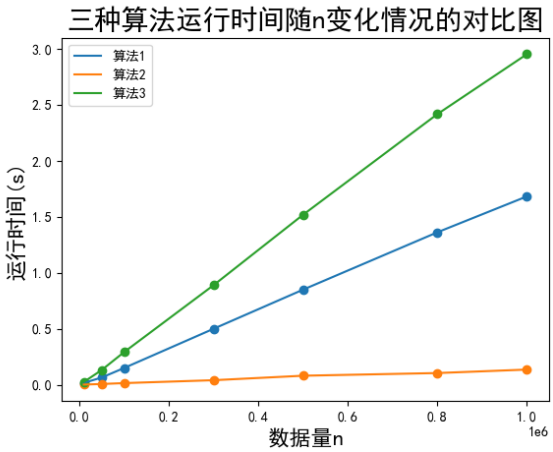




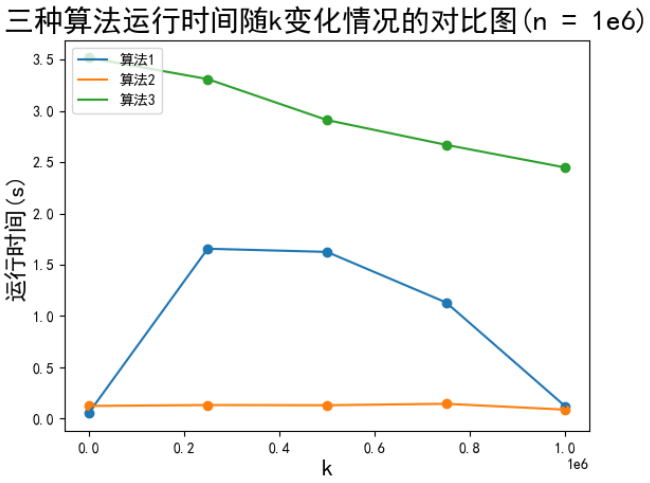
1.6. 算法分析

表 1 三种算法运行时间随 n 变化情况的对比表

n	10000	50000	100000	1000000
算法 1 (s)	0.016	0.064	0.149	1.684
算法 2 (s)	0.001	0.006	0.018	0.135
算法 3 (s)	0.024	0.131	0.291	2.956



根据上方图表分析，可以观察到三种算法随数据量 n 的增长都近似于线性增长。尽管算法三在常数上较大，运行时间普遍较长，但其随时间增长的斜率稳定，符合 O(n)的时间复杂度。此外，算法二的最坏时间复杂度为 $O(n^2)$，但由于其随机选择，绝大多数的情况下都能表现出 O(n)的时间复杂度；尽管算法三的时间复杂度能稳定在 O(n)，但由于其排序、求中位数等操作带来了较大的常数，因此总的运行时间表现不佳。算法一的时间复杂度与 k 也有关系，因此下面分析三种算法运行时间随 k 变化情况。



可以观察到，算法一的运行时间随 k 变化明显。这是由于，当 k 较小时将其视为常熟，O(k + (n − k)logk)也趋近于线性的时间复杂度；当 k 较大时，时间复杂度主要在建堆所带来的 O(k)上，因此也近似于 O(n)；当 k 在 n/2 附近时，时间复杂度近似于 O(nlogn)，因此所耗时间较久。

此外，为了模拟算法 2 的极端情况，我将其选择的枢纽固定为第一个，并测试了倒序的（如上一部分展示的测试用例 3 中的）数据，可以明显观察到算法 2 的运行时间增长（由于其时间复杂度退化到 $O(n^2)$）。

1.7. 其他说明

本题我总体完成的比较顺利，在实现算法一、三时基本没有遇到问题。

问题二的 RandomizedSelect 中，Partition 和二分的 Select 选择的边界问题尤为重要。在快速排序时，每一次 Partition 将数组分为两个部分，枢轴左侧的部分值小于它，枢轴右侧的部分值大于它。二分时，两次调用 Select 函数区间都不覆盖当前的枢轴；因此，区间大小一定会不断缩短，不会陷入死循环。但是在本题背景中，若当前选择的枢轴正好与区间末尾的元素值相同，则会导致区间大小不更新。此时若区间内所有数的值都相同，Select 函数将无限递归下去，从而导致死循环。这有两种解决方案：1) 若保持 Partition 函数不变，则在 Select 函数中可以增加条件：若枢轴位置正好是当前要求的 k，则直接输出 nums[i]，否则让区间右边界收缩 1 防止死循环；2) 修改 Partition 函数的思路，不要求一定将枢轴至于左侧全小于它，右侧全大于它的位置上，只保证从数列左侧开始的某个子序列满足所有数都小于枢轴，返回的是这个子序列的最后一个坐标，这样也能防止陷入死循环。

2. 题目 3 警卫巡逻问题

2.1. 题目描述

博物馆是具有 n 个顶点的凸多边形的形状。博物馆由警卫队通过巡逻来确保馆内物品的安全。博物馆的安全保卫工作遵循以下规则，以尽可能时间经济的方式确保最大的安全性：

（1） 警卫队中每个警卫巡逻都沿着一个三角形的路径；该三角形的每个顶点都必须是多边形的顶点。

（2） 警卫可以观察其巡逻路径三角形内的所有点，并且只能观察到这些点；我们说这些点由该警卫守护并覆盖。

（3） 博物馆内的每一处都必须由警卫人员守护。

（4） 任何两个警卫巡逻所在的三角形在其内部不重叠，但它们可能具有相同的边。在这些限制条件下，警卫的成本是警卫巡逻所沿路径的三角形的周长。我们的目标是找到一组警卫，以使警卫队的总成本（即各个警卫的成本之和）尽可能小。给定博物馆顶点的 x 坐标和 y 坐标以及这些顶点沿博物馆边界的顺序，设计一种算法求解该问题，并给出算法的时间复杂性。

请注意，我们并未试图最小化警卫人数。我们希望使警卫队巡逻的路线的总长度最小化，假定任何线段的长度都是线段端点之间的欧几里得距离，并且可以在恒定时间内计算该长度。

2.2. 算法思想

设 dp[i][[j] 的值为由顶点 i 到顶点 j 所构成的 j – i + 1 边形进行三角形划分后可以得到的最小警卫巡逻成本。考虑终止情况，即当 j = i + 2 时，凸多边形退化为三角形。其他情况下需要再进一步进行剖分，假设剖分得到的三角形中，顶点 i、j 和另一个顶点 k（i < k < j）构成了一个三角形，那么三角形 ikj 就将这个凸多边形分成了三部分:

1.顶点 i 到顶点 k 构成的凸 k – i + 1 边形。当 k = i + 1 时，这部分不存在；

2.顶点 i、k、j 构成的三角形；

3.顶点 k 到顶点 j 构成的凸 j – k + 1 边形。当 k = j – 1 时，这部分不存在。

凸多边形的值就是这三部分的值之和。容易得到转移方程：

$$dp[i][j] = \max_{k=i+1}^{j-1}\{dp[i][k] + dp[k][j] + C(i,j,k)\}$$

其中 $C(i,j,k)$ 表示 i、j、k 为顶点的三角形周长。

2.3. 编程语言及环境

编程语言：C++

环境：Windows10　Microsoft Visual Studio 2022 Debug x86

2.4. 典型测试用例

输入数据格式：

第一行为一个数字，代表凸多边形的边数 n；

接下来 n 行按照**逆时针顺序**输入顶点信息，每行分别有一个字母和两个数字。字母表示该顶点的编号，两个数字分别表示该顶点的横、纵坐标。

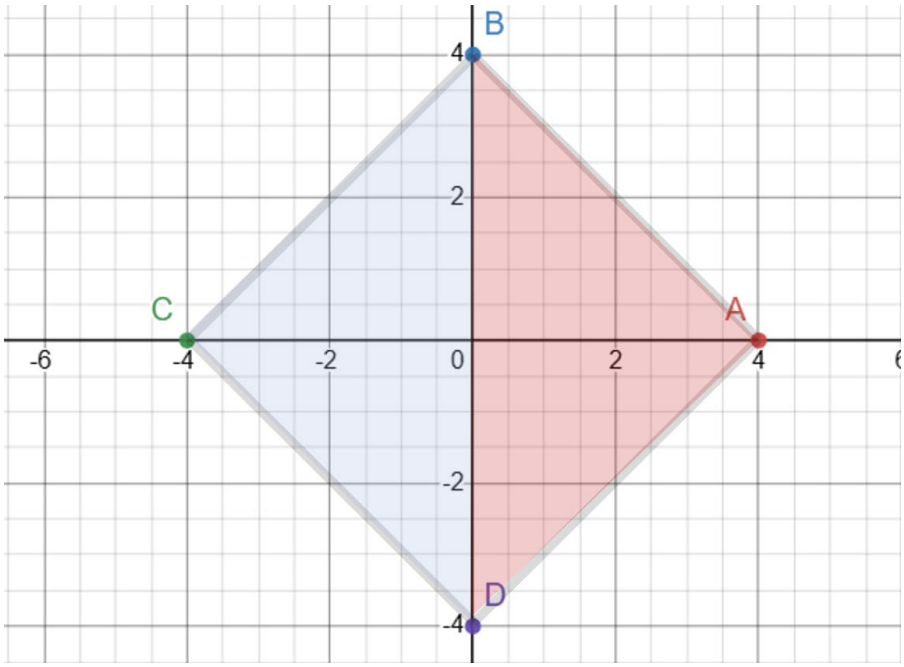
输出数据格式：

第一行为一个数字，表示警卫巡逻的最小总成本。

接下来 n-2 行，每行三个字母表示对应的警卫巡逻三角形。

说明：程序未对凸边形合法性做判断，需要输入数据保证输入的顶点按逆时针（或顺时针）顺序输入，且顶点之间构成的多边形为凸多边形。

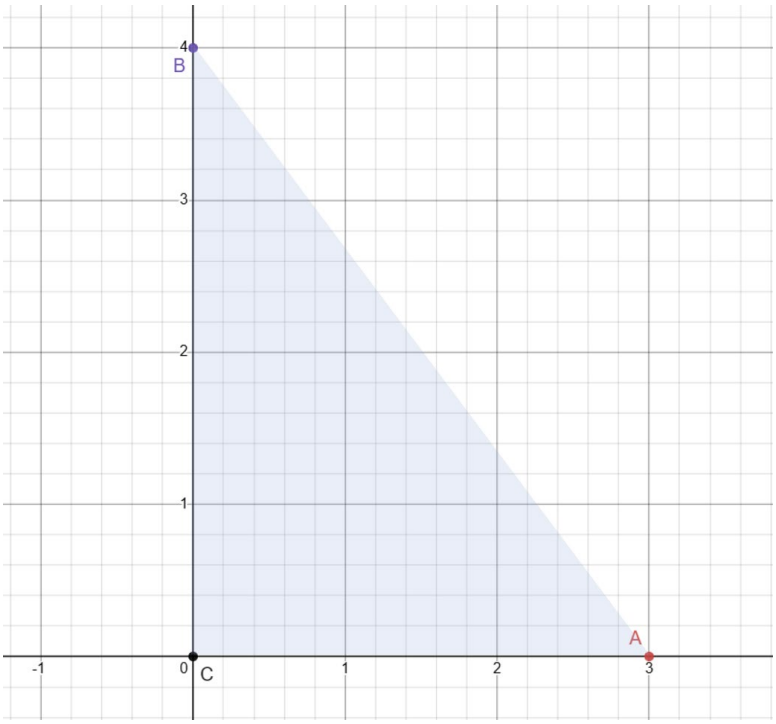
2.4.1. 测试用例 1



输入数据	4 A 4 0

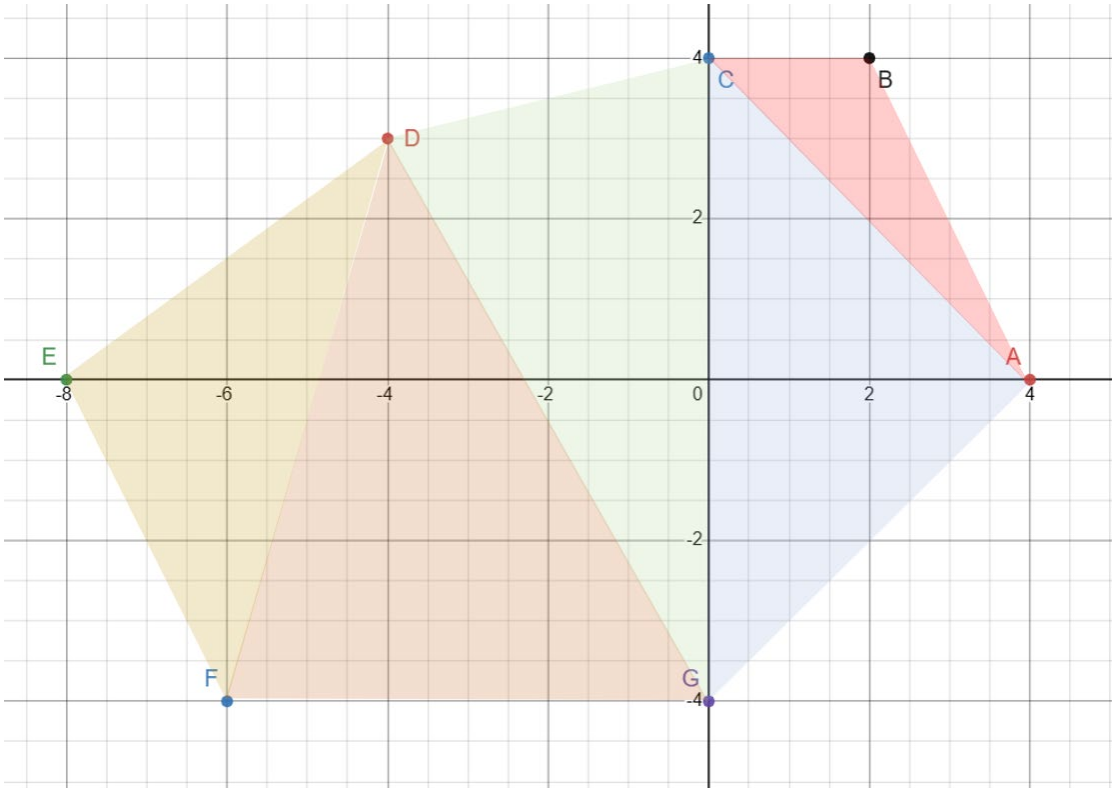
	B 0 4 C -4 0 D 0 -4
输出结果	38.6274 ADB BDC

2.4.2. 测试用例 2



输入数据	3 A 3 0 B 0 4 C 0 0
输出结果	12 ACB

2.4.3. 测试用例 3



输入数据	7 A 4 0 B 2 4 C 0 4 D -4 3 E -8 0 F -6 -4 G 0 -4
输出结果	89.7227 AGC CGD DGF DFE ACB

2.5. 系统输入输出运行结果截图

Microsoft Visual Studio 调试控制台

```
4
A 4 0
B 0 4
C -4 0
D 0 -4
38.6274
ADB
BDC
```

Microsoft Visual Studio 调试控制台

```
3
A 3 0
B 0 4
C 0 0
12
ACB
```

Microsoft Visual Studio 调试控制台

```
7
A 4 0
B 2 4
C 0 4
D -4 3
E -8 0
F -6 -4
G 0 -4
89.7227
AGC
CGD
DGF
DFE
ACB
```

2.6. 算法分析

下面进行该算法的时间复杂度分析。算法中的核心是求出凸多边形从 i 到 j 最小的警卫巡逻成本 dp[i][j]（其中 i 和 j 是多边形的顶点编号）。

为了计算 dp[i][j]，我们需要枚举一个从顶点 i 到顶点 j 之间的顶点 k（不包括 i 和 j），然后根据 dp[i][k]、dp[k][j]与以顶点 i、j、k 构成的三角形边长三者之和来计算 dp[i][j]，在 k 滑动过程中取最小值进行记录。

对于动态规划算法，其时间复杂度是动态规划的状态数×计算每个状态的时间复杂度。遍历 dp[i][j]的每个状态会带来 $O(n^2)$的时间复杂度，而对于每个 i、j，k 在二者之间滑动取最大值也会带来 O(n)的时间复杂度。因此，该算法的时间复杂度为 $O(n^3)$。

2.7. 其他说明

本题比较简单，我在完成时没有遇到太多的问题。完成本算法时，主要需要注意 dp 数组所表示的下标含义对应的区间开闭；此外在进行动态规划时，注意三层 for 循环的边界细节即可。由于本题中我增加了对划分方案的输出，因此还需要额外使用一个数组进行记录，即在记录 dp 数组的同时，同步更新 s 数组每次 k 滑动时所带来最佳方案的 k 值。在 dp 算法结束后，可以通过 s[0][n-1]找到最外层的最佳 k 值，再逐层寻找（顺序没有要求，用队列或栈均可）每一次的最优方案进行输出即可。