一．选择问题

1. 题目：

问题定义：给定线性序集中 n 个元素和一个整数 k，1≤k≤n，要求找出这 n 个元素中第 k 小的元素，（这里给定的线性集是无序的）。下面三种是可行的方法：

（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)。

请给出以上三种方法的算法描述，用你熟悉的编程语言实现上述三种方法。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）。

2. 算法思想：
   ① 基于堆的选择：创建一个最大堆，并将数组中的前 k 个元素插入堆中。从第 k+1 个元素开始遍历数组，对于每个元素，如果它小于堆顶元素（即最大值），则将堆顶元素弹出，将该元素插入堆中。遍历完整个数组后，堆顶元素即为数组中第 k 小的元素。
   ② 随机划分线性选择：选择一个随机数 pivot，将数组划分为左右两个子数组，其中左边子数组的元素都小于等于 pivot，右边子数组的元素都大于等于 pivot。如果左边子数组的长度等于 k-1，则 pivot 即为数组中第 k 小的元素。如果左边子数组的长度小于 k-1，则第 k 小的元素位于右边子数组中，将右边子数组递归地作为输入，寻找第 k-leftSize-1 小的元素。如果左边子数组的长度大于 k-1，则第 k 小的元素位于左边子数组中，将左边子数组递归地作为输入，寻找第 k 小的元素。
   ③ 利用中位数的线性时间选择： 算法的核心思想是通过选取数组的中位数作为枢轴元素，将数组划分为三个部分：小于中位数的元素、等于中位数的元素和大于中位数的元素。然后根据第 k 小元素的位置与中位数的大小关系，递归地在左半边或右半边继续查找，或者直接返回中位数。
3. 编程语言及环境：
   编程语言：C++，环境：DevC++
4. 输入输出结果：
   当 n 越来越大时：

	基于堆	随机划分	中位数
100-5	0.001s	0.001s	0.001s
1000-5	0.001s	0.002s	0.001s
10000-5	0.007s	0.008s	0.012s
100000-5	0.051s	0.056s	0.105s
1000000-5	0.506s	0.521s	1.08s

   当 k 越来越大时：

	基于堆	随即划分	中位数
1000000-5	0.494s	0.512s	1.071s
1000000-50	0.494s	0.517s	1.076s
1000000-500	0.496s	0.517s	1.065s
1000000-5000	0.509s	0.503s	1.069s
1000000-50000	0.588s	0.502s	1.075s

5. 算法分析：

① 基于堆的选择：该算法的期望时间复杂度为 O(n)，其中 n 是数组的长度。由于每次选择的随机数是等概率的，因此左边子数组的长度的期望值是 n/2，右边子数组的长度的期望值也是 n/2。因此，每次递归的长度都是 n/2 的期望，而且在每一次递归中，都只需要处理一个子数组，因此期望时间复杂度是线性的。

② 随机划分线性选择：该算法的时间复杂度是线性的 O(n)。在步骤 1 中，将 n 个元素划分为 ⌈n/5⌉ 组，每组最多包含 5 个元素。因此，每组中使用插入排序或者任意其他排序算法排序的时间复杂度是 O(1)，共需 O(n)时间。在步骤 3 中，递归查找第 k 小元素的过程中，每次选择一个子集的中位数来进行划分，因此在每一层递归中，所需要的比较次数最多为 n/2，共需 O(n)时间。由于每次递归只处理一个子集，因此总共需要的递归次数是 O(log n)。因此，该算法的总时间复杂度为 O(n log n)。

③ 利用中位数的线性选择：在递归过程中，每次选择中位数都需要对整个数组进行排序，时间复杂度为 O(nlogn)。但由于选择中位数是一个随机过程，可以证明选择的中位数的期望值是数组中的第(1/4)n ~ (3/4)n 个元素，因此算法的期望时间复杂度为线性时间 O(n)。

6. 其他说明：

根据理论分析，随机划分的线性选择算法算法时间复杂度期望为 O(n)，但最坏情况下可能会退化到 O(n^2)。这个算法在平均情况下的表现比堆排序要好。另外利用中位数的线性时间选择算法该算法时间复杂度期望为 O(n)，并且由于使用了中位数，其最坏情况下也能保证线性时间复杂度。在最坏情况下，该算法的表现比随机划分的线性选择要好。

但是在实际情况中并不是这样的，结果恰恰相反，这是一个令人困惑的地方。

二．博物馆警卫巡逻问题

1. 题目：

凸多边形是每个内角小于 180 度的多边形。

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。

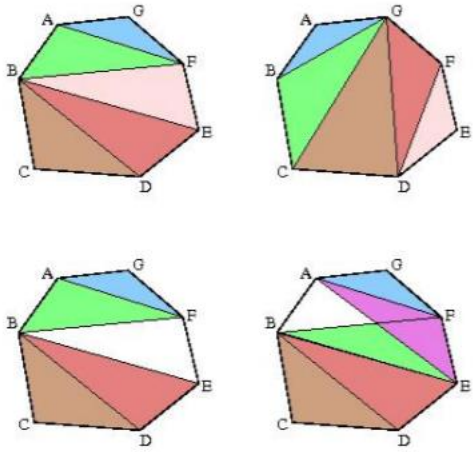
（3）博物馆内的每一处都必须由警卫人员守护。

（4）任何两个警卫巡逻所在的三角形在其内部不会重叠，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。



上面是说明本问题的四个图形。博物馆是多边形 ABCDEFG（顶点的逆时针序）。每个彩色（阴影）三角形对应一个警卫，

上面的两个图显示了一组警卫（它们的三角形），它们满足安全保卫规则。在左上方，警卫队巡逻了三角形 AFG（蓝色），ABF（绿色），BEF（淡红色），BDE（浅红色）和 BCD（棕色）的边界。在右上方，守卫巡逻 ABG（蓝色），BCG（绿色），CDG（棕色），DFG（浅红色）和 DEF（浅红色）。

底部的两个图显示了一组不满足这些规则的三角形：在左下图中，博物馆的一部分没有任何警卫守护（覆盖）（无阴影三角形 BEF），而在右下图中，粉色三角形（AEF）和绿色三角形（BEF）相交。

2. 算法思想: 算法核心思想是将凸多边形分割成三角形, 并计算每种分割方式的得分, 最终求出最优得分。该算法使用了自底向上的动态规划方法，利用一个二维数组 array 来存储已经计算过的最优得分。对于每一次计算，通过递归实现计算子问题的

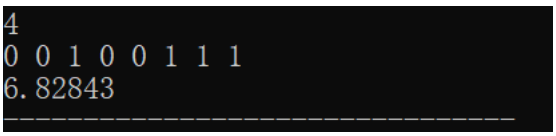
最优得分，并使用 array 数组来存储已经计算过的最优得分，避免了重复计算。最终，通过 array[0][n-1]来获取整个凸多边形的最优得分。

具体实现中，通过枚举最后一个划分的三角形，计算该三角形的得分，利用递归计算剩余子问题的最优得分，并选出得分最小的三角形作为当前分割的最优方案。特别地，当分割成三角形时，可以直接计算得分，无需递归。

3. 编程语言及环境：

   编程语言：C++，环境：DevC++

4. 输入输出结果：



5. 算法分析：这个算法采用了动态规划的思想，用递归函数来求解凸多边形的最优三角剖分。算法中利用了一个二维数组来存储每个子问题的解，从而避免重复计算，提高算法的效率。

   具体来说，算法将凸多边形看作一个有序点集合，其中每个点表示凸多边形的一个顶点。对于一个凸多边形，假设其顶点序列为 P[0], P[1], ..., P[n-1]，则任意一个三角剖分都可以由一条边 P[i]P[j]和两个子问题 P[i], P[i+1], ..., P[j]和 P[j], P[j+1], ..., P[i+n-1]的三角剖分组成。因此，问题的求解可以通过对子问题的求解来递归地求解整个凸多边形的最优三角剖分。

   在求解子问题的过程中，可以通过一个数组来存储子问题的解，避免重复计算。具体地，设 S(i,j)表示将顶点集合 P[i], P[i+1], ..., P[j]划分为三角形所需的最小权值和，则 S(i,j)可以通过枚举边 P[i]P[j]所对应的三角形顶点 k 来计算，即 S(i,j) = min{S(i,k)+S(k,j)+w(i,j,k)}，其中 w(i,j,k)表示三角形 P[i]P[j]P[k]的权值，即从 P[i]到 P[j]到 P[k]再回到 P[i]的路径长度之和。在计算 S(i,j)的同时，可以将其存储在数组 array[i][j]中，以备后续使用。

   整个算法的时间复杂度为 O(n^3)，其中 n 为凸多边形的顶点数。虽然该算法的时间复杂度较高，但是它具有较好的可扩展性和适应性，可以应用于各种不同类型的凸多边形。

6. 其他说明：通过动态规划的方法解决凸多边形最优划分问题，不仅可以得到最优解，而且其时间复杂度为 $O(n^3)$，在实际应用中表现出了较高的效率和可扩展性。

三．主元素问题

1. 题目

设 A 是含有 n 个元素的数组，如果元素 x 在 A 中出现的次数大于 n/2，则称 x 是 A 的主元素，

（1）如果 A 中的元素是可以排序的，设计一个 O(nlogn)时间的算法，判断 A 中是否存在主元素；

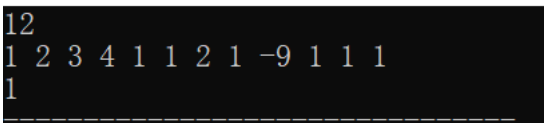
（2）对于（1）中可排序的数组，能否设计一个 O(n)时间的算法；

（3）如果 A 中元素只能进行“是否相等”的测试，但是不能排序，设计一个算法判断 A 中是否存在主元素。

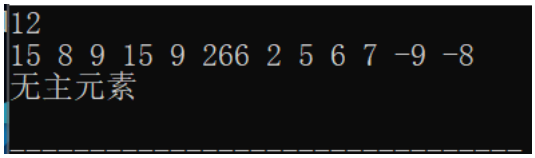
2. 算法思想：
   ① 可排序 1：将数组进行排序，再依次遍历，计算出出现次数最多的元素，判断其是否时主元素。
   ② 可排序 2：遍历数组，使用 map 记录下每个元素的出现次数，当次数超过数组元素数量的一半时即可返回主元素。
   ③ 不可排序：思想同可排序 2 ，因为可排序 2 没有用到可排序这一特性。
3. 编程语言及环境：

编程语言：C++，环境：DevC++

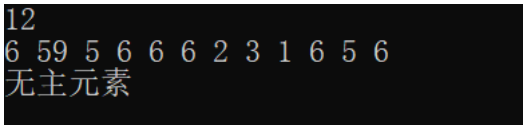
4. 输入输出结果：
   ① 可排序 1：



   ② 可排序 2：



   ③ 不可排序：



5. 算法分析：
   ① 可排序 1 中我时先将序列进行堆排序（O(nlogn)），再进行遍历（O(n)），总时间开销为（O(nlogn)），空间开销为 O（n）
   ② 可排序 2 中我并未对元素进行排序，直接进行遍历，统计每个元素的出现次数，时间复杂度为 O(n)，但是由于需要存储元素的出现次数，空间复杂度会相对高一些。
   ③ 不可排序的算法和可排序 2 相同，所以时间复杂度也是 O(n)。
6. 其他说明：在寻求 O（n）的算法时，我采用了使用更多的空间来存储出现次数的方式，这使得运行速度更快了，但响应的占用的空间也更多了，这种用空间来换取时间的做法在某些需要提升速度的情况下是可以考虑的。