北京郵電大學

实验报告

题目：分治法的应用

班　　级：20222113xx

学　　号：20222113xx

姓　　名：DoctMean

学　　院：计算机学院（国家示范性软件学院）

2024 年 1月 9 日

目录

1.实验原理

1.1 定义

"分治" (Divide and Conquer) 的字面意思是“分而治之”，也就是说将一个复杂的问题分解为两个或更多个相同或相似的子问题，直到这些子问题变得足够简单，能够直接解决。之后，通过将这些子问题的解合并，最终得到原问题的解。

1.2 过程

分治策略通常包含三个步骤：

1. **分解 (Divide)**：将一个大问题划分成若干个规模较小的子问题。

目标：将一个大问题划分成若干个规模较小的子问题。
递归：通常通过递归方式，反复将问题拆解成更小的子问题。
子问题的相似性：这些子问题与原问题的结构是相同或相似的，因此可以用同样的方法继续分解。
停止条件：分解过程会继续，直到子问题规模变得足够小（例如，只有一个元素，或问题简单到可以直接解决）。

2. **解决 (Conquer)**：递归地解决子问题，直到子问题足够简单，可以直接求解。

直接解决：当问题规模缩小到可以直接解决的程度时，通常通过基础操作来完成这些子问题的求解。
递归求解：如果子问题仍然过于复杂，则递归调用分治算法继续分解，直到问题足够简单。
终止递归：每个递归调用最终会达到一个基线条件，解决的是最小规模的子问题。

3. **合并 (Combine)**：将子问题的解合并，构造出原问题的解。

合并子问题解：将各个子问题的解整合起来，形成对原问题的整体解答。这个合并过程的逻辑取决于问题的性质。

- 在归并排序中，合并步骤是将两个已经排序的子数组合并为一个更大的排序数组。
- 在快速排序中，合并过程则是通过正确地将枢轴元素放在适当位置，并连接左子数组与右子数组的结果。

合并的复杂性：合并的过程有时需要一定的逻辑处理，但在其他情况下，它可能是一个简单的操作（例如拼接列表）。

总结：

分治法的核心思想是通过递归分解大问题，将其转化为小问题并逐步解决，再将小问题的结果组合起来。这种方法适用于具有子问题独立性、可以合并子问题解的复杂问题，且具有较好的时间复杂度表现。

分治法擅长解决的问题包括：

- 排序问题（如归并排序、快速排序）
- 查找问题（如二分查找）
- 图论中的分治算法（如求最近点对问题）
- 数值计算问题（如矩阵乘法的 Strassen 算法）

优化策略：

子问题规模平衡：通常将问题分解成大小相等的子问题能够保持算法的高效性。这种平衡子问题规模的做法可以减少递归深度，从而有效地提高算法的性能。

对于一些复杂的分治问题，如果每个子问题之间存在依赖关系，可以考虑使用动态规划来避免重复计算，从而提高效率。

1.3 辨析

递归与分治：

递归是一种通用的解决问题的思维方式和编程技巧，而分治算法是建立在递归基础上的一种具体算法模式。分治算法利用递归来分解问题、递归求解子问题，最终通过合并子问题的解来解决原问题。总之，分治算法和递归之间的关系反映了递归思想在算法设计中的广泛应用。

分治与动态规划：

分治算法通常通过递归实现，**自顶向下**逐步分解问题，然后**自底向上**合并子问题的解。而动态规划通常通过**自底向上**的迭代方式实现。它通过解决**互相重叠**的较小的子问题，逐步构建较大问题的解。

2.实验任务

2.1 实验题目

给定一个有 n 个数的**无序**的数组，查找其中排序好后的**第 k 大**的元素(排序从 1 开始)。

比较不同规模下 n=1000,100000,1000000 到 10000000,且 k 取在有序后的头部、中间和尾部时(比如，间隔为 n/8 的方式)各自的时间复杂度，比较相关的差异，画出曲线图。

2.2 实验要求

(1)随机选择方式的实现

(2)确定的线性时间的选择算法，通过比较说明相关算法的特点，可能的改进(如果有，再实现加以验证)。

(3)算法执行时会改变元素的位置，执行“partition”（分区操作）后，数组相对会“更有序”一些，试探究执行多次查找后可能的查找复杂度的变化。

3. 实验过程

3.1 随机选择

3.1.1 标准库排序函数实现

可以使用 C++标准库排序函数对数组 numbers 进行从小到大的排序，直接输出 numbers[k-1]即为第 k 大的元素。

Algorithm 1：C++标准库排序函数

输入：数组 numbers，第 k 大的元素位置 k

输出：第 k 大的元素

```
std::sort(numbers.begin(),numbers.end(),greater<int>());
print(numbers[k-1]);
```

已知默认排序算法的平均时间复杂度下界 nlogn，故该算法的复杂度为 O(nlogn)，其中 n 是待排序序列的大小。

3.1.2 随机选择关键值的分治法实现

每次随机从数组 numbers 中选择一个值作为关键值，将数组 numbers 中的数据按照关键值的大小分为三段：小于关键值，等于关键值，大于关键值。基于这样的划分，系统再在分得的子问题中进行求解。

Algorithm 2：以随机选择方式的分治法查找第 k 大元素

输入：数组 numbers，第 k 大的元素位置 k，随机数生成器 gen，左边界 l，右边界 r

输出：第 k 大的元素

```
if left == right then
    return numbers[left]
end if

pivotIndex = gen(left, right)
pivot = numbers[pivotIndex]
交换 numbers[pivotIndex] 和 numbers[right]

IndexMem = left
for i = left to right - 1 do
    if numbers[i] > pivot then
        交换 numbers[i] 和 numbers[IndexMem]
        IndexMem = IndexMem + 1
    end if
end for

交换 numbers[IndexMem] 和 numbers[right]
```

```
bound = IndexMem - left + 1
if bound == k then
    return numbers[IndexMem]
else if k < bound then
    return QuickSelect(numbers, left, IndexMem - 1, k, gen)
else
    return QuickSelect(numbers, IndexMem + 1, right, k - bound, gen)
end if
```

1. 基线情况：

判断条件：如果当前查找范围内只有一个元素，即 left == right，那么这个元素必然是你要找的第 k 大元素。

处理方法：直接返回这个唯一的元素，不需要进一步递归。

2. 随机选择枢轴：

操作：从当前查找范围 [left, right] 内随机选取一个元素作为**枢轴**（pivot），这是为了避免最坏情况（即每次选择的枢轴不平衡，导致递归深度增加）。

步骤：选取完枢轴元素后，将其与当前范围的最后一个元素交换，将枢轴移动到数组的末尾，方便后续的分区操作。

3. 分区操作：

操作：将所有**大于枢轴**的元素移动到数组的左侧。通过维护一个 IndexMem 来记录比枢轴大的元素的最终位置。遍历当前范围的所有元素，将比枢轴大的元素交换到左侧。

步骤：

- 遍历数组 [left, right-1] 中的元素。
- 如果 numbers [i] > pivot，则交换 numbers [i] 和 numbers [IndexMem]，并将 IndexMem 递增。
- 遍历完成后，将枢轴元素放置在 IndexMem 的位置（即将 numbers [IndexMem] 和 numbers [right] 交换），完成分区。

4. 递归查找：

判断：分区完成后，IndexMem 表示枢轴元素的最终位置。现在根据

IndexMem 与目标 k 进行比较：

- 如果 IndexMem 恰好是第 k 大元素的位置：即 IndexMem - left + 1 == k，直接返回枢轴元素 numbers [IndexMem]。
- 如果 k 小于 IndexMem 左侧的元素个数：第 k 大的元素在左侧子数组中，递归查找左侧。
- 如果 k 大于 IndexMem：第 k 大的元素在右侧子数组中，递归查找右侧，并将 k 调整为 k - 左侧子数组的元素个数，即 k - (IndexMem - left + 1)，因为左侧子数组已经排除了部分元素。

3.1.3 复杂度分析

最好情况分析：

在最好情况下，每次选择的枢轴能够将数组均匀分为两个大小接近的子数组，左右子数组的大小接近 $\frac{n}{2}$。

每次递归的时间为 $O(n)$，因为需要遍历整个数组进行分区操作。

由于问题规模每次递归时减半，递归深度为$O(logn)$，每层递归的时间复杂度为 $O(n)$，所以总的时间复杂度为：

$$T(n) = O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \cdots = O(n) \times 2 = O(n)$$

因此，在最好情况下，QuickSelect 的时间复杂度为 $O(n)$。

最坏情况分析：

在最坏情况下，每次选择的枢轴是当前范围内的最大或最小元素，这会导致一侧子数组的大小为 1，另一侧子数组的大小为 $n-1$=，每次递归**只减少一个**元素。

递归深度为 $O(n)$ ，每次分区需要 $O(n)$ 的时间，所以总的时间复杂度为：

$$T(n) = T(n-1) + O(n)$$

展开递归关系式，得到：

$$T(n) = O(n) + O(n-1) + O(n-2) + \cdots + O(1) = O(n^2)$$

因此，最坏情况下，QuickSelect 的时间复杂度为 $O(n^2)$。

平均情况分析：

在平均情况下，枢轴的选择是随机的，数组在每次分区时左右子数组的大小接近均匀分布。例如，枢轴平均将数组分为大小约为 εn 和 $(1-\varepsilon)n$ 的子数组，其中 $0<\varepsilon<1$。

递归关系包括：

$$E(n)=E\big(sizeof(numbers)\big)+O(n)$$

$$E(n)=\frac{1}{n}\sum_{i=1}^{n}\begin{cases}E(i-1)+O(n), i>k\\E(n-i)+O(n), i<k\\O(n), i=k\end{cases}$$

$$E(n)=O(n)+\frac{1}{n}\sum_{i=1}^{n}E(max\ (i-1,n-i))$$

可以得到：

$$\sum_{i=1}^{n}max\ (i-1,n-i)=\sum_{i=1}^{n/2}(n-i)+\sum_{\frac{n}{2}+1}^{n}(i-1)\approx 2\times(\frac{n^2}{4}-\frac{n}{4})\approx\frac{n^2}{2}$$

$$E(n)=O(n)+\frac{1}{n}\sum_{i=1}^{n}E(\max(i-1,n-i))\le cn+\frac{c}{n}*\frac{n^2}{2}=\frac{3n}{2}$$

即$E(n)=O(n)$

通过对递归关系的求解以及概率分析，最终可以得到平均时间复杂度为$O(n)$。

总结：

- **最好情况下的时间复杂度：**$O(n)$
- **最坏情况下的时间复杂度：**$O(n^2)$
- **平均情况下的时间复杂度：**$O(n)$

通过随机选择枢轴，QuickSelect 在实际应用中通常表现出接近线性时间的效率，尽管存在最坏情况的可能性，但其发生的概率非常低。

3.2 确定的线性时间选择

3.2.1 BFPRT 算法

BFPRT 算法，也称为中位数的中位数算法，是一种用于在最坏情况下时间复杂度为 O(n)时找到数组中第 k 大的数的算法。它主要通过改进快速排序中基准值（pivot）的选择来避免快速排序在最坏情况下的性能退化。

Algorithm 3：基于确定性选择的分治法查找第 k 大元素

```
输入：数组 numbers，第 k 大的元素位置 k，左边界 l，右边界 r
输出：第 k 大的元素
if (right - left + 1) ≤ 5 then
    return sorted(numbers[left..right])[⌈(right - left + 1)/2⌉]

Medians = []
for i from left to right step 5 do
    sub_right = min(i + 4, right)
    group = numbers[i..sub_right]
    sorted_group = sort(group)
    median = sorted_group[⌈(sub_right - i + 1)/2⌉ - 1]
    向 Medians 加入 median

medianOfMedians = determinedSelect(Medians, 0, Medians.size() - 1, ⌈Medians.size()/2⌉)

pivotIndex = findIndex(numbers, left, right, medianOfMedians)
交换 numbers[pivotIndex]和 numbers[right]

IndexMem = left
for i from left to right – 1 do
```

```
        if numbers[i] > medianOfMedians:
            交换 numbers[i]和 numbers[IndexMem] //将枢轴放到正确的位置
            IndexMem += 1


交换 numbers[IndexMem]  和 numbers[right]
bound = IndexMem - left + 1


if bound == k then
    return numbers[IndexMem]
else if k < bound
    return determinedSelect(numbers, left, IndexMem - 1, k)
else
    return determinedSelect(numbers, IndexMem + 1, right, k - bound)
```

Algorithm 4：查找子数组的中位数

```
输入：数组 numbers，左边界 l，右边界 r
输出：输入数组的中位数元素
Sort(numbers [left..right]) #  对子数组  [left, right]  进行排序
MediansIndex = left + ⌊(right - left) / 2⌋ #  计算中位数的索引
return numbers[MediansIndex] #  返回排序后子数组的中位数元素
```

1. 分组与中位数收集：

将数组分成每组最多 5 个元素的子组。

对每组找到它的中位数，形成一个新的数组 Medians。

2. 递归找到“中位数的中位数”：

对 Medians 数组递归调用 determinedSelect 函数，找到其中的中位数，也称为“中位数的中位数”，用作分区的枢轴。

3. 分区操作：

使用“中位数的中位数”作为枢轴，将数组分区，使得所有小于或等于枢轴的元素位于左侧，所有大于枢轴的元素位于右侧。

记录分割点 IndexMem。

4. 递归查找：

- 如果枢轴的位置正好是第 k 大的位置，返回该元素。
- 如果第 k 大的元素位于左侧子数组，则在左侧子数组中递归查找。
- 如果第 k 大的元素位于右侧子数组，则在右侧子数组中递归查找，并更新 k 的值。

3.2.2 BFPRT 算法复杂度分析

1. 最好情况时间复杂度分析

每次选择的枢轴 "中位数的中位数" 能够将数组近乎均匀地分为两部分，即每次分割后，左右子数组的大小接近 $\frac{n}{2}$。

将数组分为大小不超过 5 的组，每组找到中位数需要 $O(1)$ 时间。总共 $\frac{n}{5}$ 组，时间复杂度为 $O(n)$。

在 Medians 数组中找到中位数，需要递归调用 **determinedSelect**，子问题规模为 $\frac{n}{5}$；在分区操作时，需要对数组进行一次线性扫描和分区，时间复杂度为 $O(n)$。

设 $T(n)$=为大小为 n 的数组上执行 **determinedSelect** 的时间复杂度，递归关系为：

$$T(n) = T(\frac{n}{5}) + O(n)$$

我们可以应用主定理（Master Theorem）来解决此递归关系。

根据递推关系 $T(n) = a \cdot T(\frac{n}{b}) + f(n)$，我们有：

- $a = 1$：表示每次递归调用的子问题数量。
- $b = 5$：表示问题规模被分割的比例。
- $f(n) = O(n)$：表示分治步骤中花费的额外时间。

根据主定理：

- 计算 $\log_b a = \log_5 1 = 0$。

- 比较 $f(n)$ 与 $n^{\log_b a} = n^0 = 1$，我们有 $f(n) = O(n)$。

因此根据主定理的第三种情况（$f(n) = \Omega(n^{\log_b a + \epsilon})$，且满足正则条件），递归关系的解为：

$$T(n) = O(n)$$

因此，在最好情况下，**determinedSelect** 算法的时间复杂度为线性时间 $O(n)$。

2. 最坏情况时间复杂度分析

由于 "中位数的中位数" 的选择策略，保证每次分区都能至少排除一定比例的元素，避免极端不均匀的分割。

每次分区至少能够排除掉 $\frac{3n}{10}$ 的元素，这是因为中位数的中位数策略保证了一定比例的元素被排除，分区后还会残存$\frac{7n}{10}$个数组元素；每次递归都能排除至少 $\frac{3n}{10}$ 的元素，因此递归深度为 $O(logn)$。

设 $T(n)$=为大小为 n 的数组上执行 **determinedSelect** 的时间，递归关系为：

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

但是实际上算法只会选择一侧数组进行递归调用，故实际上的递归关系为：

$$T(n) \leq T\left(\frac{n}{5}\right) + O(n)$$

同上，根据主定理，仍然可以得出：

$$T(n) = O(n)$$

因此，在最坏情况下，**determinedSelect** 的时间复杂度仍为线性时间 $O(n)$。这是因为“中位数的中位数”策略避免了极端不均匀分割，从而保证每次都能有效地减少问题规模。

3. 平均情况时间复杂度分析

由于 determinedSelect 的分割策略是确定的（通过 "中位数的中位数" 选择枢轴），不依赖输入的随机性，因此在平均情况下，其表现与最坏情况相似。

我们可以假设每次分区后，递归的子问题规模减少约为 $\frac{n}{5}$，分区操作时

间为 $O(n)$。

设 $T(n)$ 为在大小为 n 的数组上执行 determinedSelect 的时间，理论上的递归关系为：

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

与最坏情况分析类似，实际上的递归关系为：

$$T(n) \leq T(\frac{n}{5}) + O(n)$$

根据归纳假设可以证明，对于所有 $m < n$ 成立 $T(m) \leq a \cdot m$，并且递归关系满足：

$$T(n) \leq a \cdot \frac{n}{5} + b \cdot n$$

选择合适的常数 a 和 b(当$\frac{a}{5} + b \leq$a，即$b \leq \frac{4a}{5}$的情况下)，可以确保 $T(n) \leq a \cdot n$。

因此，平均情况下，determinedSelect 的时间复杂度为 $O(n)$。

总结：

- **最好情况时间复杂度：**$O(n)$
- **最坏情况时间复杂度：**$O(n)$
- **平均情况时间复杂度：**$O(n)$

determinedSelect 算法通过使用 "中位数的中位数" 策略，能够保证在任何情况下都具有线性时间的复杂度，且其空间复杂度为 $O(logn)$，是一种非常高效且稳定的选择算法。

3.3 多次查询复杂度的变化分析

1.数据序列有序化影响的理论分析

在随机选择基准值的方法中，每次选择任意一个元素作为基准值的概率是相等的。这意味着无论序列的初始顺序如何，每次选择基准值都是均匀随机的。由于随机选择的特性，即使多次查询后序列趋向于有序，这种有序化并不会影响查询效率，因为每次选择基准值的概率分布不会改变。

BFPRT 算法通过选择中位数的中位数作为基准值来优化查询效率，以期望在最坏情况下也能保持线性时间复杂度。如果使用 BFPRT 算法多次查询，可能

会导致序列逐渐有序化。在极限情况下，如果序列完全有序，那么每次选择的基准值总能将序列平均分成两部分。在这种情况下，可以计算出平均的计算次数为 ，这意味着最坏情况下的时间复杂度可以降低。

然而，对于通常情况，这种有序化实际上可能会提高 BFPRT 算法的时间复杂度，因为其原本就是为了处理无序序列而设计的。

2.数组 K 值对算法时间复杂度的影响分析

对于 BFPRT 算法，无论 K 的取值如何，算法的递归结束条件都是当序列缩减到只有一个元素时。这意味着无论 K 的值是多少，算法的递归深度是相似的。

因此，算法的时间复杂度与 K 的取值无关，对于不同的 K 值，算法的复杂度保持一致。

4. 运行效率测试

4.1 单次查询的平均时长

图 1 数组大小为 1000 时，不同算法运行时间比较

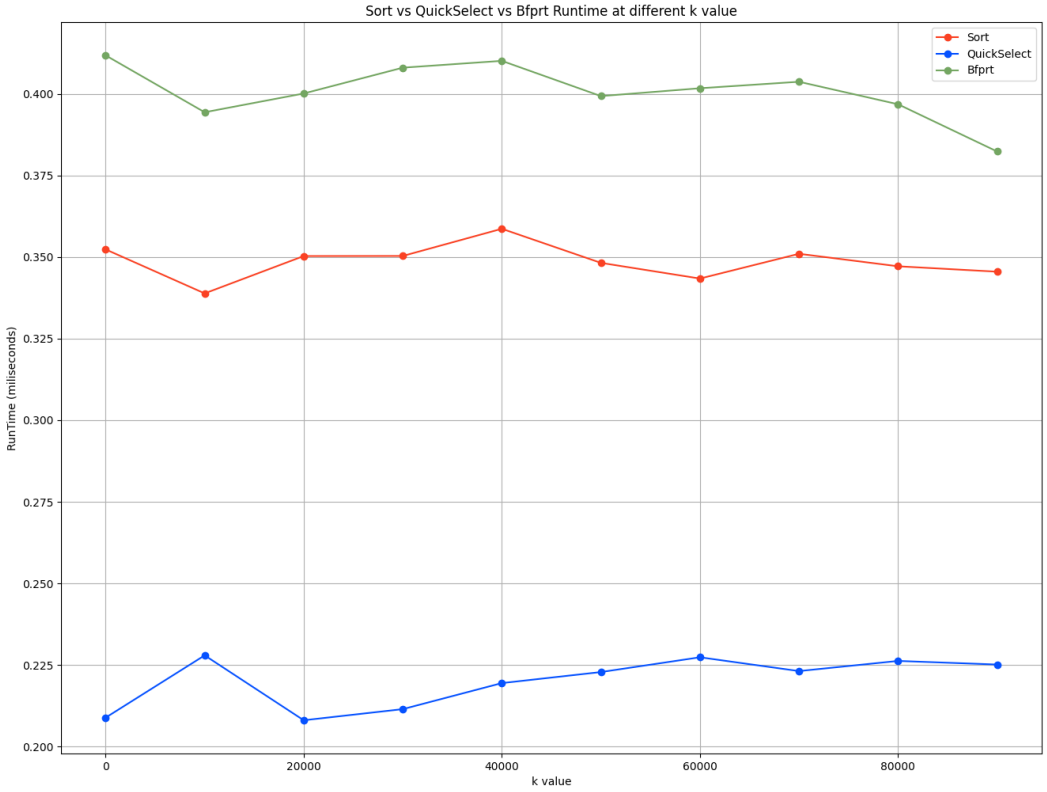

图 2 数组大小为 100000 时，不同算法运行时间比较

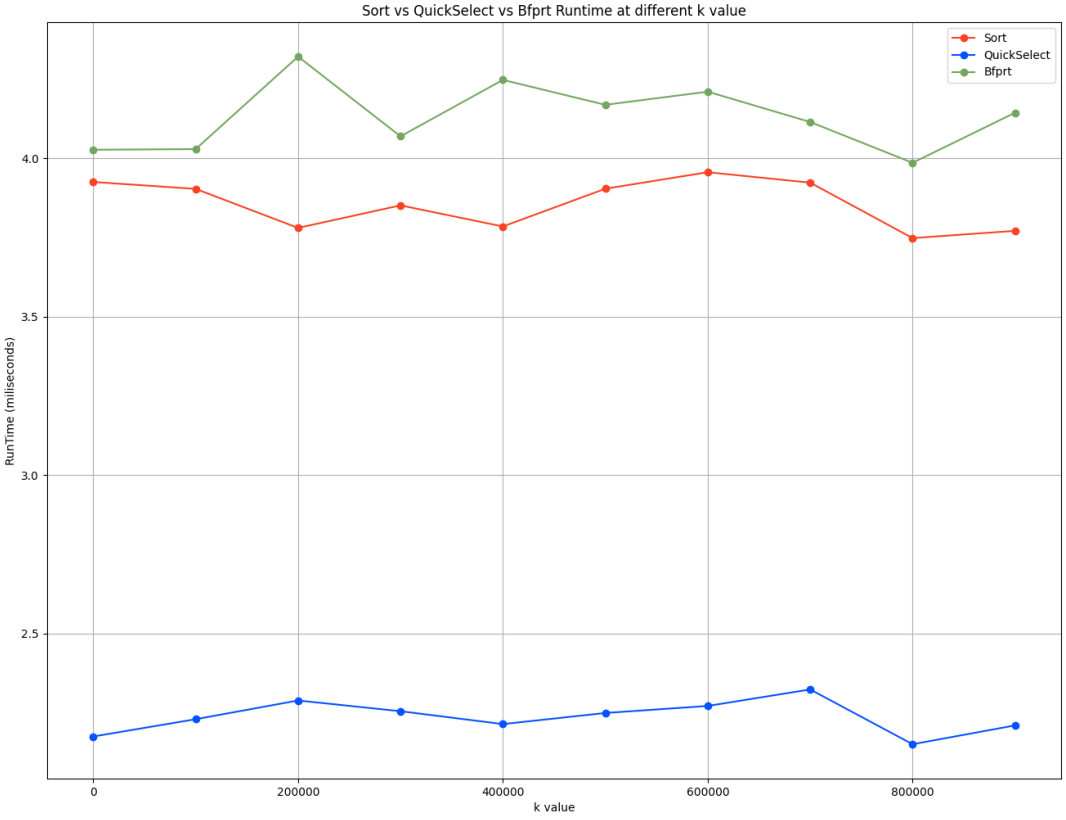

图 3 数组大小为 1000000 时，不同算法运行时间比较

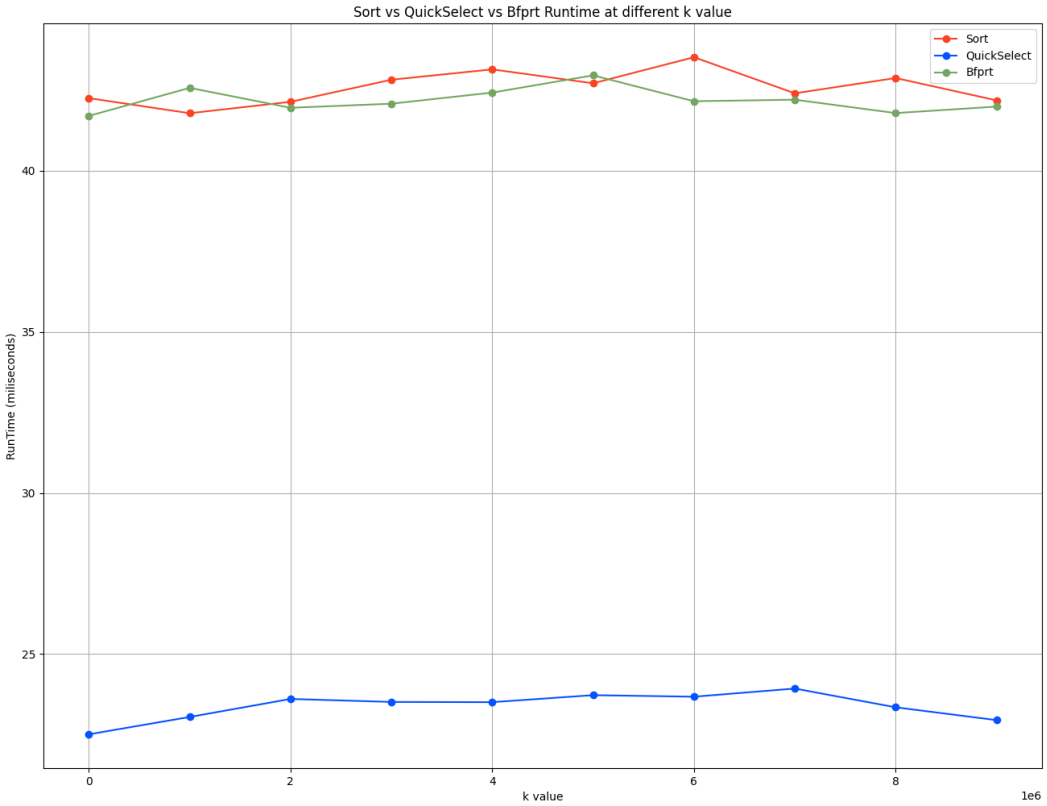

图 4 数组大小为 10000000 时，不同算法运行时间比较

所有实验均进行 10 轮取平均值；

综合以上四张图，在随机选择数值作为基准元素的情况下，当 k 从小到大地取值时，程序运行时间呈现出**先上升后下降**的趋势。在确定性地选择数值作为基准元素的情况下，即每次选择一个固定位置的元素作为基准，程序运行时间基本保持稳定，并在随机选择方法所需时间的 **2 倍**左右波动。

除此以外，随着 k 值增大，Sort 方法的**上移**非常明显：当 k 值较小时，Sort 方法程序运行时间比随机选择程序还要短；当 k 值>大于 1000 后，sort 方法程序运行时间长于随机选择程序，慢慢向 Bfprt 方法程序运行时间靠拢；而当 k 值很大接近千万的时候，Sort 方法程序已经比 Bfprt 方法低效。

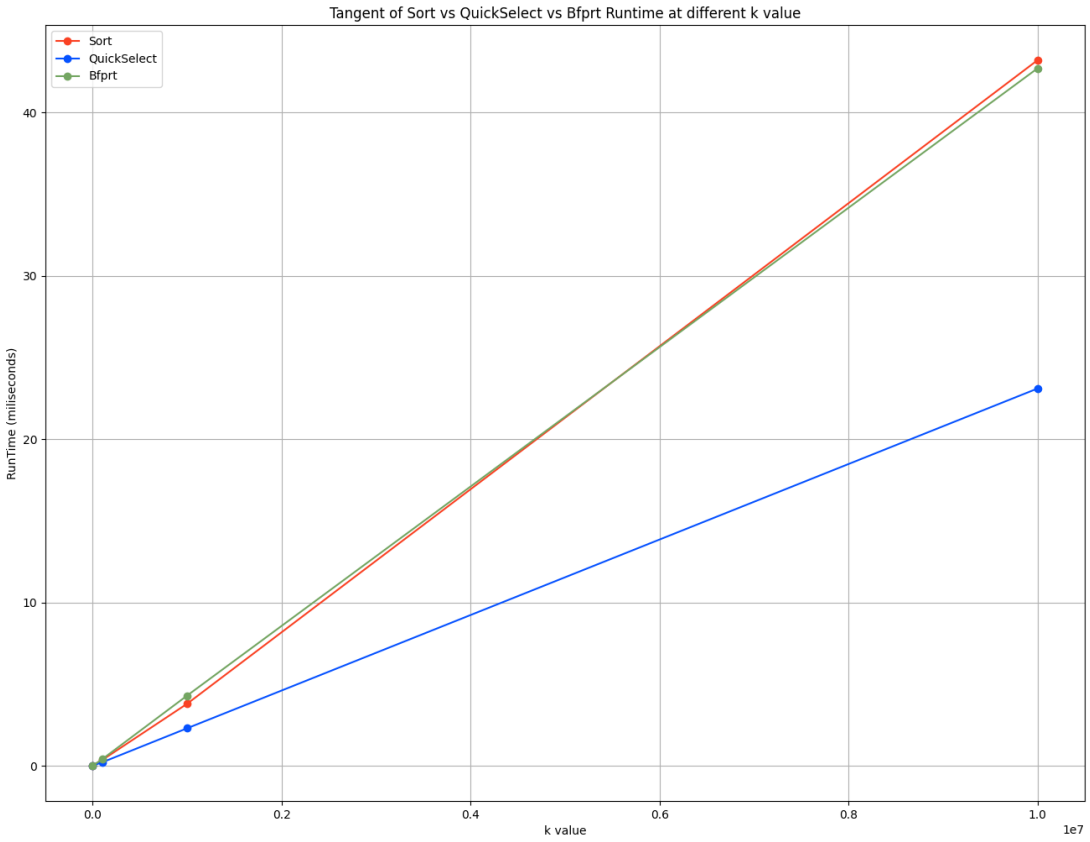

图 5 不同算法运行时长与 k 值的函数关系

从图 5 我们可以进一步确定，三种方法的运行时长随着 k 值的增长而呈**线性**增长趋势，由此我们可以反过来验证之前我们的理论推导，随机选择方法与 Bfprt 方法的平均时间复杂度几乎都是 O(n)，Sort 函数则在 k 值处于不同区间而表现出不同的时间复杂度。

4.2 测试结果分析

确定性选择算法通过每次分区都能有效缩小搜索范围，确保了运行时间的线性稳定性。这意味着，无论我们要查找的是哪个位置的值，算法的运行时间都不会因为 k 值的位置而有太大波动。然而，这种算法在实现上相对**复杂**，需要将数组分成固定大小的组，排序并递归地选择中位数。这一过程不仅涉及大量的交换和比较操作，还增加了**递归调用**的开销，特别是在处理大规模数据时，开销尤为显著。

相比之下，随机选择算法由于其枢轴选择的随机性，表现出了更为**灵活和高效**的特性。当我们要查找接近最大值或最小值的元素时，随机选择能够迅速将较大的元素推向数组的一侧，从而**快速缩小**待查找区间。这种方法不仅减少了递归

的深度，还提升了运行效率。此外，随机化选择算法在内存访问模式上更为友好，通常进行顺序扫描和交换，符合现代处理器的**缓存机制**，进一步提升了性能。

在实际应用中，虽然确定性选择算法在理论上能够保证线性时间复杂度，但由于复杂度的常数较高、缓存性能较差以及其实现过于复杂，导致确定性选择算法在实际运行中并不如随机选择算法高效。随机化选择算法不仅**实现简单**，代码简洁，还能够在大多数情况下需求更少的运行时间，尽管在极少数情况下可能退化为线性时间。

总而言之，确定性选择算法在理论研究中具有重要意义，确保了算法的稳定性和可靠性。然而，在实际应用中，随机选择算法凭借其**较低的常数因子**、更好的缓存利用率和**简洁高效**的实现，往往成为更为优选的选择。因此，对于大多数实际数据分布情况，随机化选择方法不仅表现出更高的效率，也更加符合现代计算环境的需求。

4.3 数列逐步有序后运行时长的变化

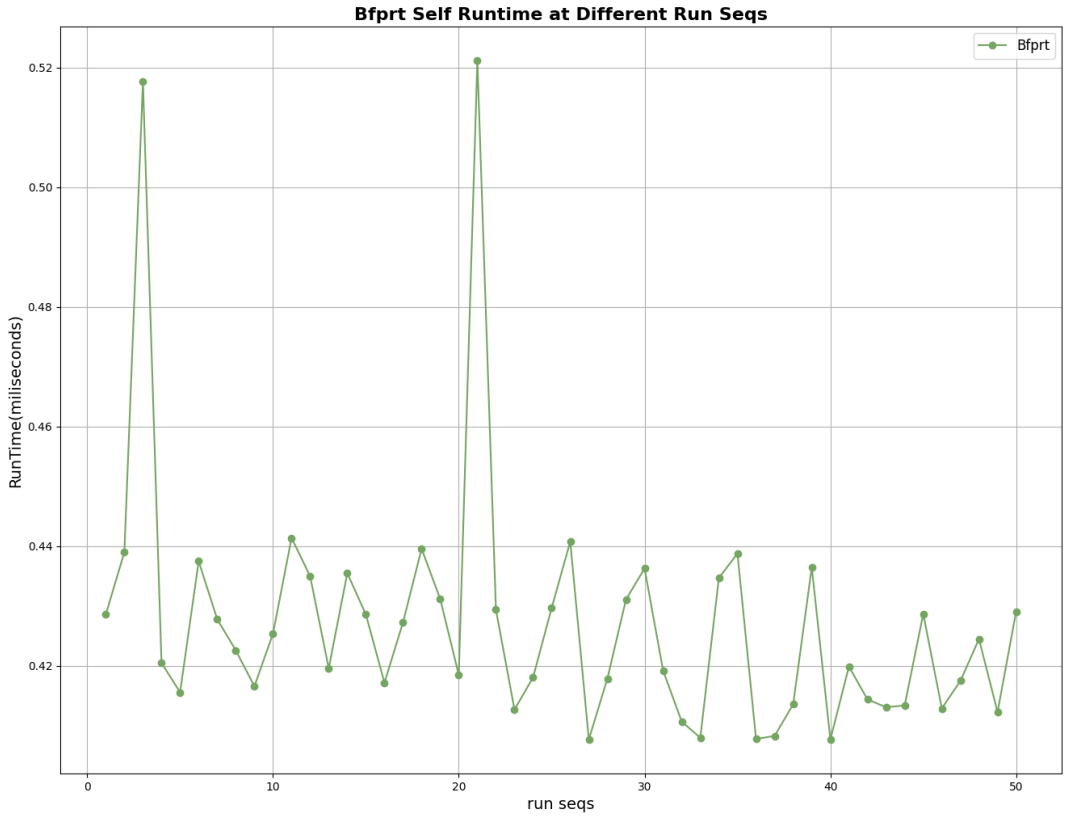

图 6 针对同一数组，确定性选择算法程序多次运行的运行时长

如图 6，可以观察到，随着反复查找次数的增加，采用确定性选择方法的

时间表现出在波动中**降低**的趋势。即使数据本身是有序的，BFPRT 算法通过使用“中位数的中位数”方法来选择枢轴，这意味着算法的性能并不依赖于数据的初始有序性。

不过从运行结果来看，尽管 BFPRT 算法**不依赖**数据的有序性，但在有序数据中进行分区操作会更加**高效**。原理上来说，这是因为当数据接近有序时，分区过程中需要移动的数据量减少，现代计算机的缓存机制对连续内存访问进行了优化。因此，在数据接近有序时，分区操作的内存访问更加一致，提高了缓存命中率，从而减少了执行时间。此外，数据的有序性有助于每次选择的枢轴“中位数的中位数”将数组近乎**均匀**地分割为两部分，这意味着每次分割后左右子数组的大小大致相等，接近算法的最佳情况。

同时还需要考虑 searchMedians 函数中的排序操作。对于完全有序的数据，排序过程可能更快。但是，这种排序操作仅在极小的子数组中进行，因此对整体复杂度的影响非常有限。这表明数据的有序性会对 BFPRT 算法的整体效率有促进作用，但是是**常数级别**的微小影响。

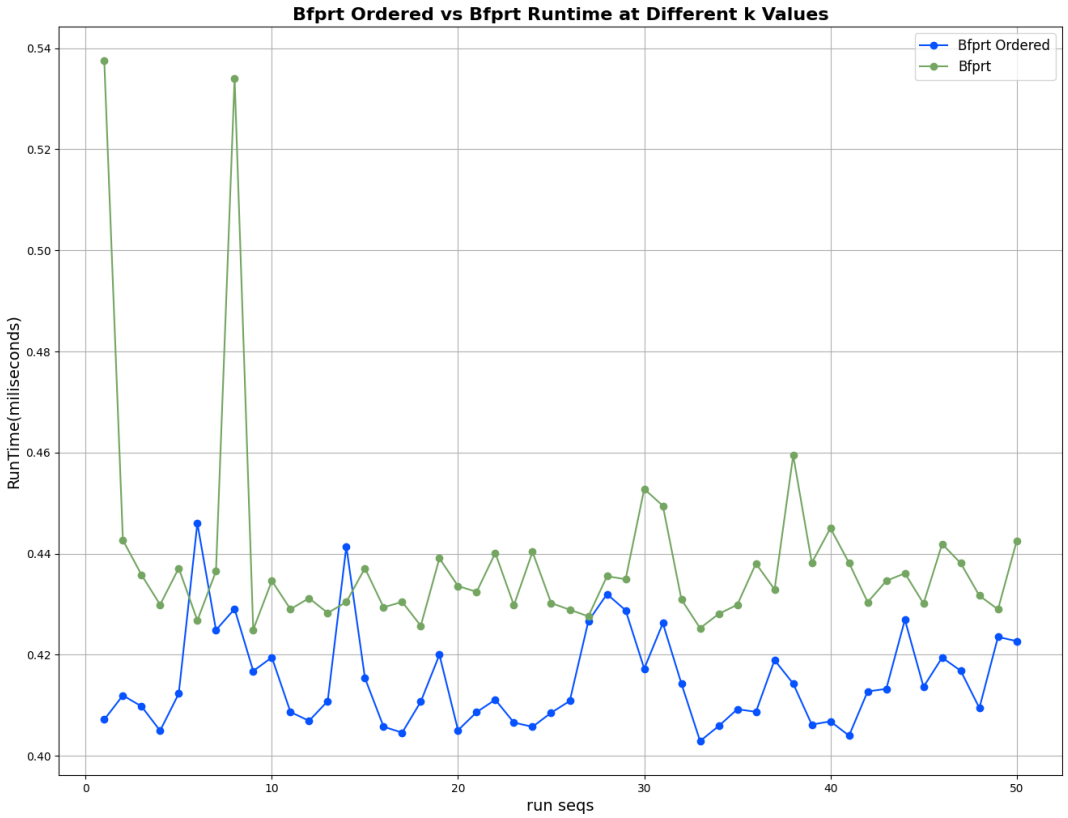

图 7 分别在有序数组和无序数组情况下，确定性选择算法程序的运行时长

如图 7，我们尝试了在有序数组上运行 BFPRT 算法。从图中可以看出，当数据有序时，使用确定性选择方法的运行时间显著低于数据无序时的情况。此外，有序数据下的运行时间更加稳定，波动较小，这说明数据的有序性能够提高算法

的执行效率并减少运行时间的不稳定性。

我们甚至可以把完全有序的数组当作是逐步有序的**极端**例子，这样图 6 和图 7 的联系非常显著。

4.4 BFPRT 算法改进

1. 使用插入排序优化中位数查找

原始代码的问题：

在原始代码中，当处理每组最多 5 个元素时，使用了标准库中的 std::sort 进行排序；尽管 std::sort 通常具有优秀的性能，但对于极小规模的数组而言，常数因子较大，整体效率不如预期。

改进思想：

将排序方法从通用的 std::sort 替换为插入排序，针对小规模数据进行优化。此外，插入排序在处理几乎有序的数据时，具有更好的缓存局部性。

2. 正确计算中位数索引

原始代码的问题：

在原始代码中，中位数的索引计算存在潜在的问题：原计算方式在某些情况下可能导致返回错误的中位数，尤其是在子数组大小为偶数时。

改进思想：

在处理偶数个元素时，返回靠前的中位数；确保在所有情况下，中位数的选择都是正确的。

3. 模块化设计与辅助函数的引入

原始代码的问题：

原始代码将所有功能集中在一个函数中，中位数的查找和枢轴的管理混杂在一起，增加了代码的复杂性。

改进思想：

引入辅助函数来实现模块化：

- searchMedians：用于对子数组进行排序并返回中位数；
- findIndex：用于在指定范围内查找特定值的索引。

通过模块化设计，函数 determinedSelect 的逻辑更清晰，增强了代码的可读性和可维护性。

4. 处理枢轴未找到的情况

原始代码的问题：

在原始代码中，使用 findIndex 查找 medianOfMedians 的索引，但未处理查找失败的情况。如果 medianOfMedians 不存在于当前子数组中，会导致返回-1，从而引发错误。

改进思想：

在新代码中，添加了对 findIndex 返回值的检查，确保在枢轴未找到时，选择最后一个元素作为枢轴，避免无限循环或错误的分区。

5.总结

5.1 算法比较总结

算法类型	时间复杂度	空间复杂度	实现难度	适用场景
直接排序法	O(n log n)	O(n)	简单	小规模数据集，一次性查询
快速选择法	平均 O(n)，最坏 $O(n^2)$	O(1)	简单	中等规模数据集，动态查询
确定性选择算法	O(n)	O(n)	复杂	大规模数据集，对时间稳定性有高要求的场景

1.直接排序法

优点

实现简单：利用现有的排序算法（如快速排序、归并排序、堆排序等）即可轻松实现。

稳定性高：在大多数编程语言中，内置排序函数经过高度优化，性能稳定可靠。

适用于小规模数据：对于数据量较小的情况，排序的时间开销较低，且无需额外的复杂操作。

缺点

时间复杂度高：一般排序算法的时间复杂度为 O(nlogn)，对于需要频繁查找第 k 大元素的场景，效率较低。

不适合大规模数据：当数据量极大时，排序的时间和空间开销显著增加，影响性能。

适用场景

小规模数据集：当数据量较小（如几百或几千个元素）时，排序法依然高效且简洁。

一次性查询：如果只需要进行一次查询，且不频繁变动数据集，排序法是一个不错的选择。

2.快速选择法

优点

平均时间复杂度低：快速选择法的平均时间复杂度为 O(n)，在实际应用中表现优异。

实现相对简单：与快速排序类似，快速选择法的实现较为直观，代码简洁。

缺点

最坏情况时间复杂度高：在极端情况下（如每次选择的枢轴都是最小或最大的元素），时间复杂度可能退化到 $O(n^2)$。

依赖枢轴选择策略：算法的性能高度依赖于枢轴的选择策略，随机选择枢轴虽然能在大多数情况下避免最坏情况，但仍无法完全杜绝。

适用场景

中等规模数据集：对于中等规模的数据集，快速选择法能够高效地完成查询。

动态数据集：当需要频繁进行第 k 大元素查询，且数据集动态变化时，快速选择法因其较低的平均时间复杂度而表现优异。

3.确定性选择算法

优点

线性时间复杂度：在最坏情况下，BFPRT 算法的时间复杂度为 O(n)，提供了理论上的时间保证。

稳定性高：由于枢轴选择的策略能够避免最坏情况的发生，算法在所有输入情况下表现一致。

缺点

常数因子较大：由于需要多次递归调用和中位数的计算，算法的常数因子较大，实际运行时间可能比快速选择法更长。

实现复杂：相较于快速选择法，BFPRT 算法的实现更加复杂，涉及多个步骤和递归调用，增加了实现和维护的难度。

空间开销：需要额外的空间来存储中位数，尤其是在处理大规模数据集时，内存开销较为显著。

适用场景

大规模数据集：在处理非常大规模（如数千万到数亿个元素）的数据集时，BFPRT 算法因其线性时间复杂度的优势显得尤为重要。

对时间稳定性有高要求的场景：在一些对时间稳定性和最坏情况有严格要求的应用中，BFPRT 算法能够提供可靠的性能表现。

5.2 经验与总结

在以上实验中，我对**分治算法、常数级优化**方面有了更为深入的理解，体会到了在算法在理论与实际应用之间的差异。

首先我熟悉了确定性选择算法，它的最大特点是通过选取中位数的中位数来确保在最坏情况下实现线性时间复杂度。这种方法在**理论上**具有极高的吸引力，因为它消除了随机化算法中可能出现的最坏情况。然而，实验结果显示，确定性选择算法在实际操作中的复杂度较高：尽管算法的时间复杂度是线性的，但常数因子较大；确定性选择算法需要多次递归调用，增加了实现的复杂性。

与之相反，随机选择算法在实验中表现出了更优的**实际效率**，尽管其最坏情况下的时间复杂度为 $O(n^2)$。在一般分布下，随机选择算法的平均时间复杂度依然保持在 $O(n)$附近。

此外，实验还揭示了数据有序性对两种算法性能的显著影响。具体表现为：

在处理**部分有序**的数据时，确定性选择算法的性能略微有所提升。这是因为有序数据减少了中位数选择过程中的复杂度，使得递归调用的次数减少，但是对整体的影响实在过于**微小**。

对应地，尽管随机选择算法对数据**有序敏感性**更低，但在有序数据上的表现仍略优于完全无序的数据。那么我们可以归纳出，输入数据的特性在一定程度上可以影响随机选择算法的效率，尤其是在数据具有某种内在规律或部分排序的情况下。

通过一系列优化，确定性选择算法在实际应用中的性能得到了显著提升。然而，与随机选择算法相比，仍存在一定的性能差距。

总之，此次实验不仅加深了我对分治算法在第 k 大元素问题上的理解，还让我认识到理论优化与实际应用之间的复杂关系。通过对比随机选择与确定性选择算法，我学会了在实际应用中如何根据具体需求和数据特性，**选择最合适**的算法。同时，实验过程中对实现细节的关注也让我意识到，精细的优化也可以提升算法的实际性能。这些经验和认识将在我未来的算法设计与优化工作中发挥重要作用，帮助我更好地熟悉**理论优化手段和实际优化手段**。

源代码

1. Sort

```
    std::sort(numbers.begin(), numbers.end(),greater<int>());
    int result = numbers[k-1];
```

2. QuickSelect

```
int quickSelect(std::vector& numbers, int left, int right, int k, std::
ranlux48_base& gen){
    if (left == right) {
        return numbers[left];
    }
    int pivotIndex = gen(left, right);
    int pivot = numbers[pivotIndex];
    std::swap(numbers[pivotIndex], numbers[right]);

    int IndexMem = left;
    for (int i = left; i < right; i++) {
        if (numbers[i] > pivot) {
            std::swap(numbers[i], numbers[IndexMem]);
            IndexMem++;
        }
    }
    std::swap(numbers[IndexMem], numbers[right]);
    int bound = IndexMem - left + 1;
    if (bound == k) {
        return numbers[IndexMem];
    } else if (k < bound) {
        return QuickSelect(numbers, left, IndexMem - 1, k, gen);
    } else {
        return QuickSelect(numbers, IndexMem + 1, right, k - bound, gen);
    }
}
```

3. BFPRT

```
int determinedSelect(std::vector<int>& numbers, int left, int right, int k) {
    //数组大小过小直接排序并返回中位数
    if (right - left + 1 <= 5) {
        // 提取子数组
        std::vector<int> subarray(numbers.begin() + left, numbers.begin() + right +
1);
```

```
        std::sort(subarray.begin(), subarray.end(), std::greater<int>());
        return subarray[k - 1];
    }
    // 将数组分成每组最多5个元素，找到每组的中位数
    std::vector<int> Medians;
    for (int i = left; i <= right; i += 5) {
        int sub_right = std::min(i + 4, right);
        std::vector<int> group(numbers.begin() + i, numbers.begin() + sub_right + 1);
        std::sort(group.begin(), group.end(), std::greater<int>());
        // 计算中位数索引
        int medianIndex = static_cast<int>(std::ceil(static_cast<double>(sub_right -
i + 1) / 2)) - 1;
        int median = group[medianIndex];
        Medians.push_back(median);
    }
    // 找到中位数的中位数
    int medianOfMedians;
    if (Medians.size() == 1) {
        medianOfMedians = Medians[0];
    } else {
        medianOfMedians = determinedSelect(Medians, 0, Medians.size() - 1,
static_cast<int>(std::ceil(Medians.size() / 2.0)));
    }

    // 找到medianOfMedians在原数组中的索引
    int pivotIndex = findIndex(numbers, left, right, medianOfMedians);
    std::swap(numbers[pivotIndex], numbers[right]);

    // 分区操作
    int IndexMem = left;
    for (int i = left; i < right; ++i) {
        if (numbers[i] > medianOfMedians) { // 查找第k大，因此使用 > 符号
            std::swap(numbers[i], numbers[IndexMem]);
            IndexMem++;
        }
    }
    std::swap(numbers[IndexMem], numbers[right]);
    int bound = IndexMem - left + 1;

    if (bound == k) {
        return numbers[IndexMem];
    } else if (k < bound) {
        return determinedSelect(numbers, left, IndexMem - 1, k);
    } else {
```

```
        return determinedSelect(numbers, IndexMem + 1, right, k - bound);
    }
}
int findMedians(const std::vector<int>& numbers, int left, int right, int value) {
    for (int i = left; i <= right; ++i) {
        if (numbers[i] == value) {
            return i;
        }
    }
    return -1;
}
```

4. 改进后的 BFPRT

```
// 定义组大小为5
const int groupSize = 5;

// 插入排序找中位数（降序排序后返回中位数）
int searchMedians(std::vector<int>& numbers, int left, int right) {
    for (int i = left + 1; i <= right; ++i) {
        int key = numbers[i];
        int j = i - 1;
        while (j >= left && numbers[j] < key) { // 改为降序排序
            numbers[j + 1] = numbers[j];
            --j;
        }
        numbers[j + 1] = key;
    }
    int size = right - left + 1;
    int medianIndex = left + (size - 1) / 2;
    return numbers[medianIndex];
}

// 在指定范围内查找特定值的索引
int findIndex(const std::vector<int>& numbers, int left, int right, int value) {
    for (int i = left; i <= right; ++i) {
        if (numbers[i] == value) {
            return i;
        }
    }
    return -1;
}

// 确定性选择算法的迭代版本
int determinedSelect(std::vector<int>& numbers, int left, int right, int k) {
    while (true) {
```

```
        // 如果数组足够小，直接排序并返回第k大的元素
        if (right - left + 1 <= groupSize) {
            std::vector<int> subarray(numbers.begin() + left, numbers.begin() + right
+ 1);
            // 降序排序
            std::sort(subarray.begin(), subarray.end(), std::greater<int>());
            // 返回第k大的元素
            return subarray[k - 1];
        }

        // 将数组分成每组最多5个元素，找到每组的中位数
        std::vector<int> Medians;
        Medians.reserve((right - left + groupSize) / groupSize); // 预分配空间
        for (int i = left; i <= right; i += groupSize) {
            int sub_right = std::min(i + groupSize - 1, right);
            Medians.push_back(searchMedians(numbers, i, sub_right));
        }

        // 找到中位数的中位数
        int MediansOfMedians;
        if (Medians.size() == 1) {
            MediansOfMedians = Medians[0];
        } else {
            // 计算中位数的中位数时，k应为ceil(Medians.size() / 2.0)
            int medianK = (Medians.size() + 1) / 2; // 等同于 ceil(Medians.size() /
2.0)
            MediansOfMedians = determinedSelect(Medians, 0, Medians.size() - 1,
medianK);
        }

        // 找到 MediansOfMedians 在原数组中的索引
        int pivotIndex = findIndex(numbers, left, right, MediansOfMedians);
        if (pivotIndex == -1) {
            pivotIndex = right;
        }
        std::swap(numbers[pivotIndex], numbers[right]); // 将枢轴元素移到末尾

        // 分区操作
        int IndexMem = left;
        for (int i = left; i < right; ++i) {
            if (numbers[i] > MediansOfMedians) {
                std::swap(numbers[i], numbers[IndexMem]);
                IndexMem++;
            }
```

```
        }
        std::swap(numbers[IndexMem], numbers[right]);
        int bound = IndexMem - left + 1;

        if (bound == k) {
            return numbers[IndexMem];
        } else if (k < bound) {
            right = IndexMem - 1;
        } else {
            k -= bound;
            left = IndexMem + 1;
        }
    }
}
```