深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验名称：排序算法性能分析

学院：计算机与软件学院 专业：软件工程腾班

报告人：郑杨 学号：2020151002 班级：软工 3 班

同组人：郑杨 陈敏涵

指导教师：李炎然

实验时间：2022.3.6-2022.3.19

实验报告提交时间：2022.3.20

教务处制

一、实验目的

1. 掌握选择排序、冒泡排序、合并排序、快速排序、插入排序算法原理
2. 掌握不同排序算法时间效率的经验分析方法，验证理论分析与经验分析的一致性。

二、实验概述

排序问题要求我们按照升序排列给定列表中的数据项，目前为止，已有多种排序算法提出。本实验要求掌握选择排序、冒泡排序、合并排序、快速排序、插入排序算法原理，并进行代码实现。通过对大量样本的测试结果，统计不同排序算法的时间效率与输入规模的关系，通过经验分析方法，展示不同排序算法的时间复杂度，并与理论分析的基本运算次数做比较，验证理论分析结论的正确性。

三．实验步骤与结果

1．实现插入排序、冒泡排序、选择排序、合并排序、快速排序等排序算法（从小到大）

（1）插入排序

设计思路：

从第二个数字开始，选取作为目标数字，与前面的数字从后往前逐个比较，若比目标数字大，则往后挪，直到找到不比目标数字大的位置并将目标数字插入进去。

伪代码：

$INSERTION\_SORT(A,n)$

$for\ i = 2\ to\ n$

$key = A[i]$

$j = i - 1$

$while\ j \geq 1\ and\ key < A[j]$

$A[j+1] = A[j]$

$j--$

$A[j+1] = key$

算法流程：

插入排序是将无序序列中的数据插入到有序序列中，在遍历无序序列时，首先拿出无序序列中的首元素去与有序序列中的每一个元素比较，若当前元素比拿出元素大，那么交换他们的位置，依次类推，直到遇到一个比当前小的元素，那么就是找到了合适的插入位置。如图 1 所示，一开始的无序序列为$\{3,7,6,9,1\}$,我们取出首元素‘3’，由于此时有序序列为空，那么我们直接把‘3’放入有序序列中即可。紧接着取出首元素‘7’，由于‘3’小于‘7’，所以找到了一个合适的插入位置，那么直接把‘7’插入于‘3’后面。依次类推，直到无序序列为空，算法停止。

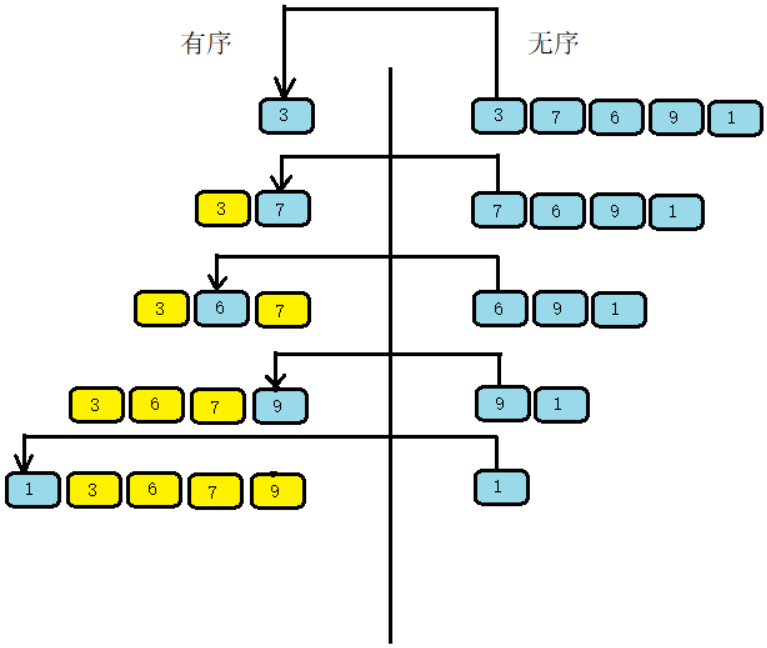

图 1 插入排序示意图

算法复杂度分析

i. 最坏情况下时间复杂度

我们通过伪代码语句运行情况来分析时间复杂度：

$\boldsymbol{INSERTION\_SORT}(A,n)$	$\boldsymbol{cost}$	$\boldsymbol{time}$
$for\ i = 2\ to\ n$	c_1	n
$key = A[i]$	c_2	$n-1$
$j = i-1$	c_3	$n-1$
$while\ j >= 1\ and\ key < A[j]$	c_4	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_5	$\sum_{i=2}^{n}(t_i-1)$
$j--$	c_6	$\sum_{i=2}^{n}(t_i-1)$
$A[j+1] = key$	c_7	$n-1$

那么有：

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{i=2}^{n} t_i + c_5\sum_{i=2}^{n}(t_i-1) + c_6\sum_{i=2}^{n}(t_i-1) + c_7(n-1)$$

最坏情况下，数组倒序，此时 $t_i = i$，故 $\sum_{i=2}^{n} t_i = \frac{n(n+1)}{2} - 1, \sum_{i=2}^{n}(t_i-1) = \frac{n(n-1)}{2}$

代入上式并化简有：

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1)$$

$$= \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

因此，最坏情况下的时间复杂度为$O(n^2)$

ii. 平均情况下时间复杂度

由上面最坏复杂度的分析可得，插入排序的复杂度取决于t_i，对于平均情况下，我们需要把t_i当作一个随机变量，t_i的取值范围为$[1,i]$，假设每一种情况的概率相同，为$p_i=\frac{1}{i}$。

那么 $E(t_i)=\frac{1}{i}(1+2+\cdots+i)=\frac{i+1}{2}\approx\frac{i}{2}$，将 $E(t_i)$ 带入 i 中公式得：

$$T(n)=\frac{1}{4}(c_4+c_5+c_6)n^2+(c_1+c_2+c_3+\frac{c_4}{4}-\frac{c_5}{4}-\frac{c_6}{4}+c_7)n-(c_2+c_3+\frac{1}{2}c_4+c_7)$$

因此，平均情况下的时间复杂度为$O(n^2)$，常数比最坏情况下略小。

iii. 空间复杂度

插入排序只使用了临时辅助变量，与问题规模无关，故空间复杂度为$O(1)$。

（2）冒泡排序

设计思路：

冒泡排序是一种最基础的交换排序，执行每一趟算法都可以把当前未归位的最大数通过相邻项交换的方式归位，每个元素都可以像小气泡一样，根据自身大小一点一点向数组的一侧移动。

伪代码：

```
BUBBLE_SORT(A,n)
    for i = n to 2
        for j = 2 to i
            If A[j] < A[j − 1]
                Swap(A[j],A[j − 1])
```

算法流程：

冒泡排序每一趟只能将一个数归位，也就是说，第一趟只能将末位上的数归位，第二趟只能将倒数第 2 位上的数归位，以此类推。如果有n个数进行排序，只需将$n-1$个数归位，也就是要进行$n-1$趟操作。对于每一趟操作，都需要从第一位开始进行相邻两个数的比较，将较大的数放在后面，重复直到最后一个还没归位的数，一趟循环结束。如图 2 所示，一开始序列还是{3, 7, 6, 9, 1}。进行第一趟操作之后，数字‘9’归位，进行第二趟操作之后，数字‘7’归位，依次类推。

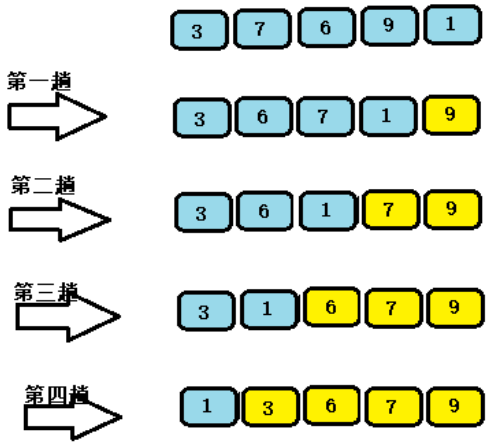

图 2 冒泡排序示意图

算法复杂度分析：

i. 最坏情况下时间复杂度

$\boldsymbol{BUBBLE\_SORT}(A,n)$	$\boldsymbol{cost}$	$\boldsymbol{time}$
$for\ i\ =\ n\ to\ 2$	c_1	n
$for\ j\ =\ 2\ to\ i$	c_2	$\sum_{i=2}^{n} i$
$If\ A[j]\ <\ A[j\ -\ 1]$	c_3	$(\sum_{i=2}^{n} i)-1$
$Swap(A[j],A[j\ -\ 1])$	c_4	$\sum_{i=2}^{n} t_i$

那么有：

$$T(n)=c_1n+c_2\sum_{i=2}^{n}i+c_3((\sum_{i=2}^{n}i)-1)+c_4\sum_{i=2}^{n}t_i$$

最坏情况下，$\sum_{i=2}^{n}t_i=(\sum_{i=2}^{n}i)-1=\frac{n(n+1)}{2}-2$

代入上式得：

$$T(n)=c_1n+c_2(\frac{n(n+1)}{2}-1)+(c_3+c_4)(\frac{n(n+1)}{2}-2)$$
$$=\frac{1}{2}(c_2+c_3+c_4)n^2+(c_1+\frac{c_2}{2}+\frac{c_3}{2}+\frac{c_4}{2})n-(c_2+2c_3+2c_4)$$

故最坏情况下的时间复杂度为：$O(n^2)$

ii. 平均情况下时间复杂度

类似的，我们把t_i看成随机变量，取值范围为$[0,i-1]$，假设每种情况出现的概率都相同，为$\frac{1}{i}$，那么$E(t_i)=\frac{1}{i}(0+1+\cdots+i-1)=\frac{i-1}{2}\approx\frac{i}{2}$

代入 i 中式子得：

$$T(n)=\frac{1}{2}(c_2+c_3+\frac{1}{2}c_4)n^2+(c_1+\frac{1}{2}c_2+\frac{1}{2}c_3+\frac{1}{4}c_4)n-(c_2+2c_3+\frac{1}{2}c_4)$$

我们发现，平均时间复杂度也为$O(n^2)$。

iii. 空间复杂度

显然，冒泡排序没有使用到额外空间，空间复杂度为$O(1)$。

（3）选择排序

设计思路：

类似冒泡排序每次将最大元素归位的思路，我们可以每次选出待排序序列中的最小值（最大值），把它放到待排序序列的最前面。以此类推直到整个序列有序为止。

伪代码：

$SELECTION\_SORT(A,n)$

$for\ i = 1\ to\ n-1$

$minval\ =\ A[i]$

$for\ j\ =\ i\ to\ n$

$if\ A[j]\ <\ minval$

$minval\ =\ A[j]$

$k\ =\ j$

$A[k]\ =\ A[i]$

$A[i]\ =\ key$

算法流程：

选择排序每一次可以把最小（最大）元素归位，那么对于长度为n的待排序序列而言，算法需要执行$n-1$趟。对于每一趟操作，我们需要遍历一遍待排序的序列，找到这个序列中的最小值，把这个最小值与待排序序列的首部进行交换。如图 3 所示，第一趟我们找到序列中的最小值‘1’，把它与序列首部的数字‘3’进行交换。以此类推，最后序列成功有序。

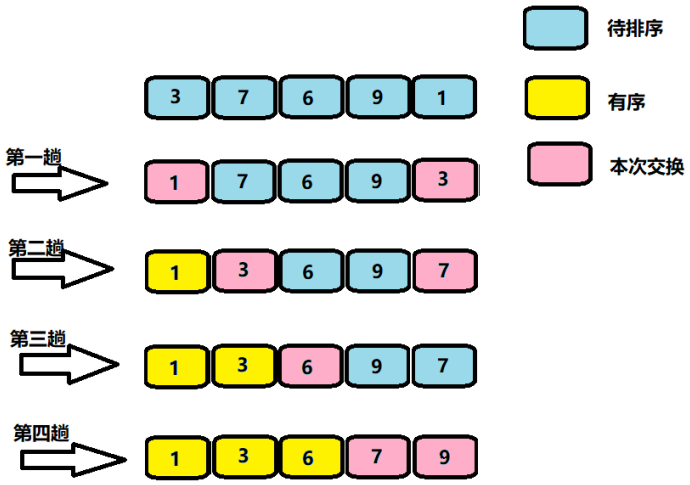

图 3 选择排序示意图

算法复杂度分析

i. 最坏情况下时间复杂度

$\boldsymbol{SELECTION\_SORT}(A,n)$	$\boldsymbol{cost}$	$\boldsymbol{time}$
$for\ i = 1\ to\ n-1$	c_1	n
$minval\ =\ A[i]$	c_2	$n-1$
$for\ j\ =\ i\ to\ n$	c_3	$\sum_{i=1}^{n-1} n-i+2$
$if\ A[j]\ <\ minval$	c_4	$\sum_{i=1}^{n-1} n-i+1$
$minval\ =\ A[j]$	c_5	$\sum_{i=1}^{n-1} t_i$
$k\ =\ j$	c_6	$\sum_{i=1}^{n-1} t_i$
$A[k]\ =\ A[i]$	c_7	$n-1$
$A[i]\ =\ minval$	c_8	$n-1$

那么有：

$$T(n)=c_1n+c_2(n-1)+c_3(\sum_{i=1}^{n-1}n-i+2)+c_4(\sum_{i=1}^{n-1}n-i+1)+c_5(\sum_{i=1}^{n-1}t_i)+c_6(\sum_{i=1}^{n-1}t_i)+c_7(n-1)+c_8(n-1)$$

最坏情况下，数组反序，$\sum_{i=1}^{n-1}t_i=\sum_{i=1}^{n-1}n-i+1=\frac{(n+2)(n-1)}{2}$

又因为：$\sum_{i=1}^{n-1}n-i+2=\frac{(n+4)(n-1)}{2}$

代入上式得：

$$T(n)=c_1n+c_2(n-1)+c_3\frac{(n+4)(n-1)}{2}+c_4\frac{(n+2)(n-1)}{2}+c_5\frac{(n+2)(n-1)}{2}+c_6\frac{(n+2)(n-1)}{2}+c_7(n-1)+c_8(n-1)$$

$$=\frac{1}{2}(c_3+c_4+c_5+c_6)n^2+(c_1+c_2+\frac{3}{2}c_3+\frac{1}{2}c_4+\frac{1}{2}c_5+\frac{1}{2}c_6)n-(c_2+2c_3+c_4+c_5+c_6+c_7+c_8)$$

故最坏情况下，选择排序的时间复杂度为$O(n^2)$。

ii. 平均情况下时间复杂度

类似的，我们把t_i看成随机变量，取值范围为$[0,n-i+1]$，假设每种情况出现的概率都相同，为$\frac{1}{n-i+2}$，那么 $E(t_i)=\frac{1}{n-i+2}(0+1+\cdots+n-i+1)=\frac{(n-i+1)}{2}$

代入 i 中式子得：

$$T(n)=\frac{1}{2}(c_3+c_4+\frac{1}{2}c_5+\frac{1}{2}c_6)n^2+(c_1+c_2+\frac{3}{2}c_3+\frac{1}{2}c_4+\frac{1}{4}c_5+\frac{1}{4}c_6)n-(c_2+2c_3+c_4+\frac{1}{2}c_5+\frac{1}{2}c_6+c_7+c_8)$$

我们发现，平均时间复杂度也为$O(n^2)$，常数比最坏情况下的复杂度略小。

iii. 空间复杂度

由于选择排序也只是使用了临时变量辅助运行算法，故空间复杂度也为$O(1)$。

（4）归并排序

设计思路：

归并排序的基本思想就是分治和递归，我们把待排序序列分为两部分，两部分就都是排序整个序列的一个子问题，那么我们求解完这两个子问题之后，再把结果**合并**起来构成一个有顺序的序列。

伪代码：

```
MERGE_SORT(A, l, r)
    if l < r
        mid = ⌊(l + r)/2⌋
        MERGE_SORT(A, l, mid)
        MERGE_SORT(A, mid + 1, r)
        MERGE(A, l, mid, r)
```

关于$MERGE$函数的伪代码，由于篇幅关系就不放下去了。

算法流程：

归并排序的本质是分治后合并，朴素的归并排序算法采用递归实现，首先我们把原序列分为均等的两部分（若原序列长度为奇数，则多余的一个数放在左半部分），然后我们想要先排序这两部分。排序这长度较小的两部分序列，本质上和排序原序列是一样的，所以他们都是‘排序原序列’的子问题，解决子问题的一个实现方式就是递归。那么当我们递归到序列长度为 1 的时候，就可以返回。返回时将两部分按顺序合并成原序列即可。如图 4 所示，我们首先把长度为 5 的原序列分为长度为 3 的左半边和长度为 2 的右半边，然后依次类推，直到序列长度为 1 为止。

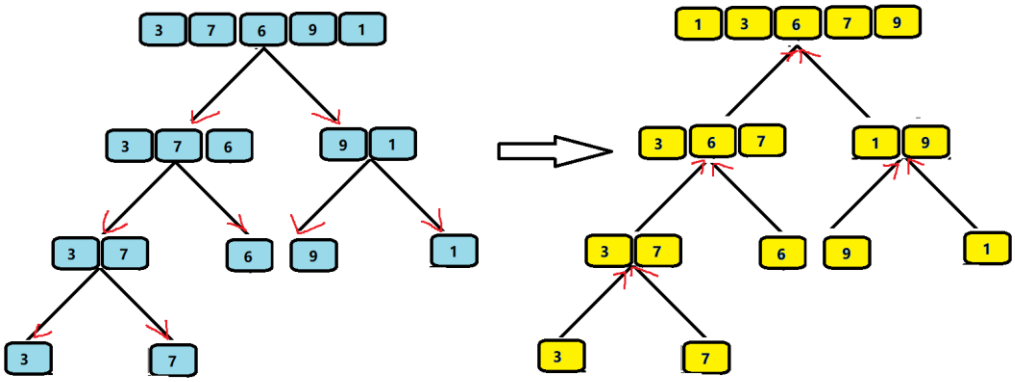

图 4 归并排序示意图（1）

合并的时候，**由于将要合并的两部分内部已经有序**，我们可以利用这个特征使用线性的方法进行合并。就是使用两个指针和一个临时数组，一个指针指向左半部分，一个指针指向右半部分。一开始两个指针都分别在两部分的起始位置，比较两个指针指向的数，把大的那个数放到临时数组里并且把对应指针往后移动；若有某个指针已经移动到尾部，则把另一个部分剩余的值依次放入临时数组中。如图 5 所示，我们合并‘3,6,7’和‘1,9’两部分的时候，首先把 1 放入临时数组中，移动‘ptr2’；然后依次把‘3，6，7’放入临时数组中，移动‘ptr1’。此时‘ptr1’已经到达左半部分的尾部，所以我们把右半部分剩余的数依次放入临时数组即可。

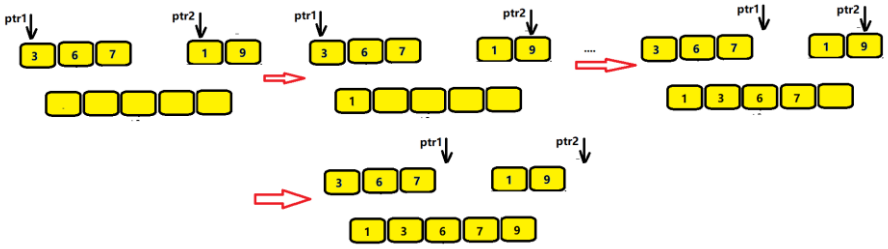

图 5 归并排序示意图（2）

算法复杂度分析

i. 时间复杂度

我们假设$T(n)$为使用归并排序解决规模为n的一个问题所用的运行时间。在归并排序里，我们总是将问题分为两个子问题，每个子问题的规模为$n/2$（如果n是奇数的话，子问题的规模分别为$\lfloor n/2 \rfloor$和$\lceil n/2 \rceil$，不过这并不影响复杂度的量级），故解决两个子问题的时间$2T(n/2)$。

我们再考虑一下分解步骤和合并步骤，分解步骤的所需时间量级为$\Theta(1)$，合并步骤的所需时间量级为$\Theta(n)$，故分解步骤和合并步骤合起来所需的时间为$\Theta(1)+\Theta(n)=\Theta(n)$，

故有：

$$T(n)=\begin{cases}\Theta(1), n=1\\ 2T(n/2)+\Theta(n), n>1\end{cases}$$

下面从比较直观的角度证明$T(n)=O(n\lg n)$

首先把$T(n)$进行更加具体化的表示：$T(n)=\begin{cases}c, n=1\\ 2T(n/2)+cn, n>1\end{cases}$

假设n是2的幂次（不会影响复杂度的量级），那么归并排序的递归树就是一颗满二叉树，我们可以把运行时间代价也按照递归树的形式画出来，如图 6 所示。

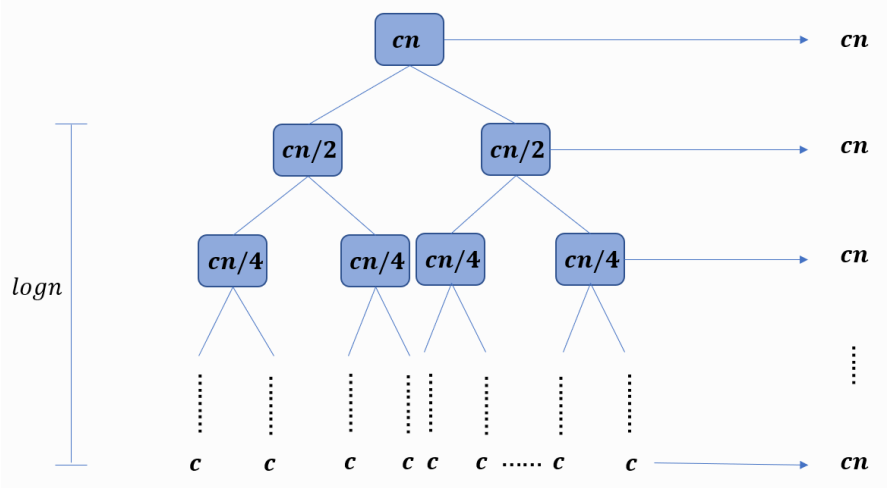

图 6 归并排序时间复杂度示意图

由于$T(n)=2T(n/2)+cn$，$T(n/2)=2T(n/4)+cn/4$，…，根据递归式可以画出上图。那么我们发现，递归树每一层的时间代价为cn，总共有$logn+1$层，那么得到：

$$T(n)=cn(\log n+1)=cn\log n+cn$$

故，归并排序的时间复杂度为$O(nlogn)$

ii. 空间复杂度

归并排序需要利用一个规模与n相同的临时数组，故空间复杂度为$O(n)$

（5）快速排序

设计思路：

与归并排序类似，快速排序的基本思想也是分治。对于一个待排序数组，快速排序首先选择一个枢纽（一般为序列首个元素），把小于这个枢纽的数放左边，把大于这个枢纽的数放右边，然后递归地去排序左部分与右部分。

伪代码：

$PARTITION(A,l,r)$

    $key = A[l]$

    $while(l < r)$

        $while(l < r\ and\ A[r] \geq key)\ r = r - 1$

        $if(l < r)\ A[l] = A[r]\ l = l + 1$

        $while(l < r\ and\ A[l] \leq key)\ l = l + 1$

        $if(l < r)\ A[r] = A[l]\ r = r - 1$

    $A[l] = key$

    $return\ l$

$QUICK\_SORT(A,l,r)$

    $if\ (l < r)$

        $mid = PARTITION(A,l,r)$

        $QUICK\_SORT(A,l,mid-1)$

        $QUICK\_SORT(A,mid+1,r)$

算法流程：

根据伪代码，首先我们需要把待排序的序列分成两部分。我们选择$A[l]$作为枢纽，用一个临时变量记录它的值。如图 7 所示，枢纽$key = A[l] = 3$，然后此时$A[l]$这个位置就空出来了，因为它的值已经被key记录下来了，所以我们从r位置开始从右到左找到第一个小于key的值，把它放到l位置。如图，我们找到第一个小于key的值为 1，此时把 1 放到l位置，然后把l往右移。我们发现把r位置的值空出来之后，需要从l位置开始从左往右找到第一个大于key的数，把它放到r，然后把r往左移动。依次类推，直到$l == r$时，划分结束，把key放到现在l（或者说r）的位置。

划分结束之后，我们就把序列分成了两部分，左部分的数都比key小，右部分的数都比key大，此时我们递归地去划分左边和右边即可。

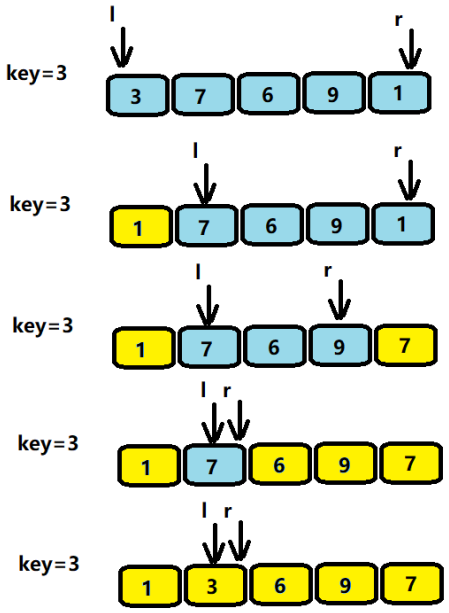

图 7 快速排序示意图

算法复杂度分析

快速排序包含两部分，划分和递归，其中划分两部分的时间复杂度为$O(n)$。假设使用快速排序解决规模为n的一个问题所使用的运行时间为$T(n)$，并假设某次划分产生的两个子问题规模为$n-a-1$和a，那么$T(n)=T(n-a-1)+T(n)+O(n)$。

i. 最坏情况下时间复杂度

最坏情况下，每次划分的两个子问题分别包含了$n-1$个元素和0个元素，那么递归式表示为：$T(n)=T(n-1)+T(0)+O(n)=T(n-1)+O(n)$，使用代入法可以得到这个递归式的解为：$T(n)=O(n^2)$。故最坏情况下，快速排序的时间复杂度为$O(n^2)$。

ii. 平均情况下时间复杂度

利用递归树的方法不难计算出平均情况下快速排序的时间复杂度是$O(nlogn)$，下面使用另一个方式证明这个复杂度的正确性。

我们不从局部进行考虑，而考虑整体，考虑$PARTITION$调用了多少次以及$PARTITION$中的比较执行了多少次。由于每一个元素都会被选为主元一次，每次都会调用$PARTITION$函数，故$PARTITION$函数会被调用n次。我们假设$PARTITION$中的总比较次数（考虑整体的比较次数而不是一次）为X，那么总复杂度就可以写成$O(n+X)$。

下面我们就只需要计算X的上界了。我们考虑每两个元素是否进行比较，由于每次$PARTITION$的主元之后就不会出现在待排序序列中了，所以任意两个元素至多比较一次。然后我们发现两个元素想要进行比较的话，必须是他们其中有某个被选为主元了。为了方便分析，我们假设这个序列的数字不重复，其中第i小和第j小的数字分别为x_i, x_j并且$i<j$，那么x_i和x_j想要进行比较的话，必须要是x_i被选为主元或者x_j被选为主元，如果是$x_k, i<k<j$被选为主元的话，那么经过这次x_i和x_j就被分到两个子问题中了，就没有可能再进行比较了。

我们定义一个布尔变量X_{ij}表示x_i和x_j是否进行比较，那么：

$$X=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}$$

我们可以通过计算X的期望来计算平均情况下的复杂度：

$$EX=E\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}EX_{ij}=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}P_{ij}$$

其中P_{ij}表示x_i和x_j进行比较的概率，通过上面的分析我们不难算出这个概率就是在划分$x_i \dots x_j$这一段时，x_i或者x_j被选为主元的概率，那么：

$$P_{ij}=P(x_i)+P(x_j)=\frac{1}{j-i+1}+\frac{1}{j-i+1}=\frac{2}{j-i+1}$$

代入上面的式子可以算出：

$$EX=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}P_{ij}=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1}=\sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k+1}<\sum_{i=1}^{n-1}\sum_{k=1}^{n-i}\frac{2}{k}=\sum_{i=1}^{n-1}O(\lg n)=O(n\lg n)$$

所以平均情况下快速排序的时间复杂度为$O(nlogn)$。

iii. 空间复杂度

空间上的开销主要是递归调用函数带来的开销，在最坏情况下，需要往下递归n层，空间复杂度为$O(n)$。在最好情况下和平均情况下，需要往下递归$logn$层，空间复杂度为$O(logn)$

2．对以上实现的所有排序算法进行随机数据实验测试，画出理论效率分析的曲线和实测的效率曲线，比较两曲线。

分别以$n=10000, n=20000, n=30000, n=50000, n=70000, n=100000$，对于每个$n$随机生成 20 组测试数据，统计每个算法的平均运行时间。

（1）插入排序

表 1 插入排序不同数据规模下理论值和实验值及其误差

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(s)	31.825	127.298	286.421	509.192	795.613	1559.402	3182.453
实验值(s)	31.825	128.369	295.547	512.032	801.783	1572.026	3203.018
误差	0.000%	0.842%	3.186%	0.558%	0.776%	0.810%	0.646%

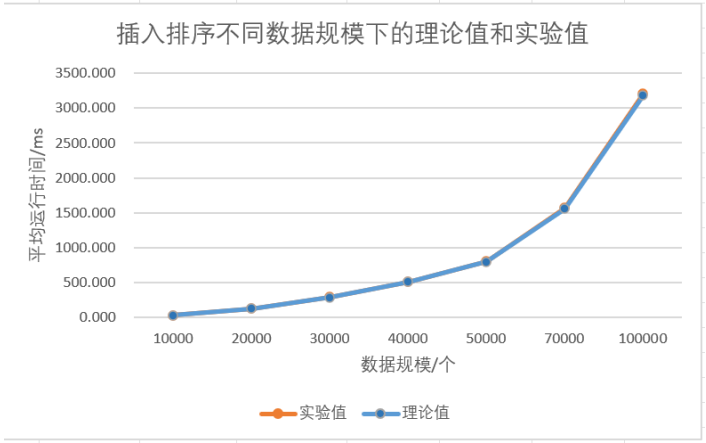

图 8 插入排序不同数据规模下理论值和实验值及其误差

我们发现，插入排序的实际平均运行时间随数据规模变换的曲线和理论值拟合的非常好，误差很小。

（2）冒泡排序

表 2 冒泡排序不同数据规模下理论值和实验值及其误差

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	158.437	633.748	1425.933	2534.992	3960.925	7763.413	15843.700
实验值(ms)	158.437	688.042	1752.603	3371.214	5520.637	11280.780	23598.231
误差	0.000%	8.567%	22.909%	32.987%	39.377%	45.307%	48.944%

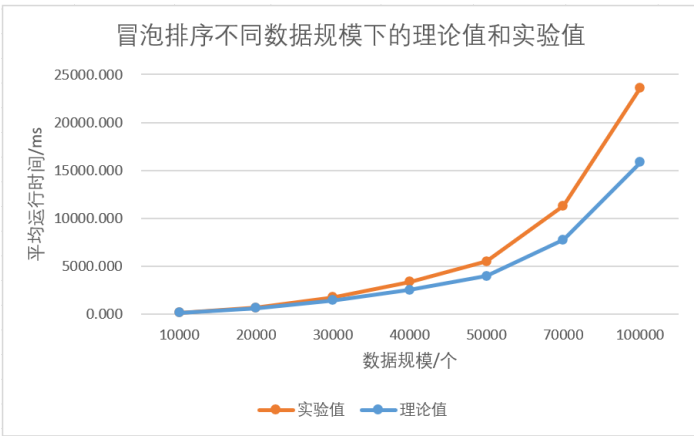

图 9 冒泡排序不同数据规模下理论值和实验值及其误差

我们发现，冒泡排序实验值和理论值的误差非常的大，而且随着数据规模的递增而递增。可能的原因是，冒泡排序本身复杂度的常数比较大，加上测试数据组数只有 20 组比较小，偶然性高导致误差较大。

（3）选择排序

表 3 选择排序不同数据规模下理论值和实验值及其误差

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	42.811	171.245	385.301	684.979	1070.280	2097.749	4281.120
实验值(ms)	42.811	168.967	378.807	674.519	1057.369	2066.199	4251.215
误差	0.000%	-1.330%	-1.685%	-1.527%	-1.206%	-1.504%	-0.699%

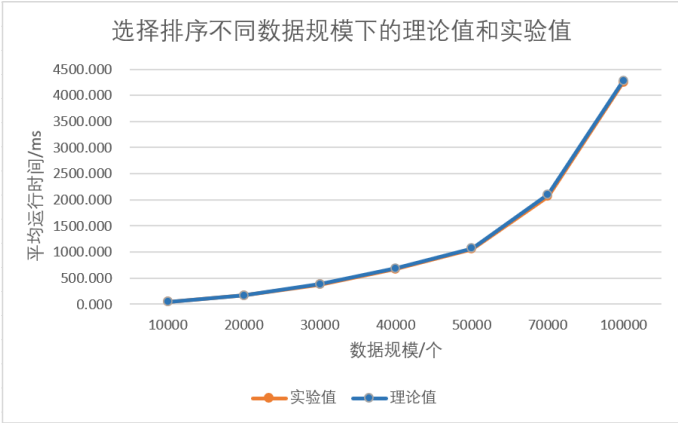

图 10 选择排序不同数据规模下理论值和实验值及其误差

选择排序的理论值和实验值拟合较好。

（4）归并排序

表 4 归并排序不同数据规模下理论值和实验值及其误差

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.85836	1.84592	2.88224	3.95022	5.04176	7.27797	10.72950
实验值(ms)	0.85836	1.84498	2.96577	4.03463	5.07342	7.47968	10.71969
误差	0.000%	-0.051%	2.898%	2.137%	0.628%	2.772%	-0.091%

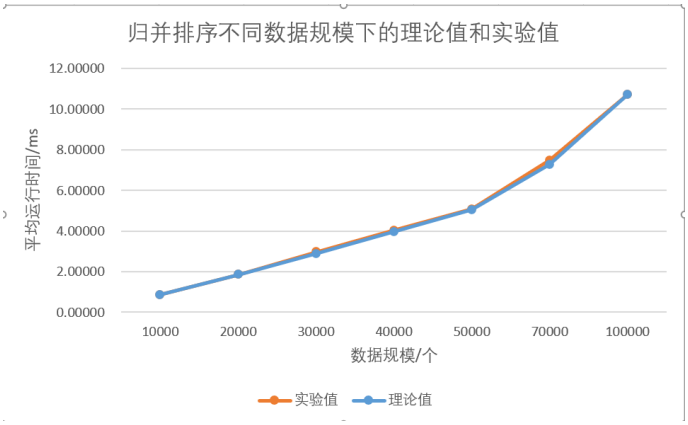

图 11 归并排序不同数据规模下理论值和实验值及其误差

归并排序实验值和理论值的拟合情况较为乐观，但也出现了少数波动，可能是因为数据的随机程度略微不同于其他数据规模导致归并排序合并中的数据比较过程较少而引起的略微差异。

（5）快速排序

表 5 快速排序不同数据规模下理论值和实验值及其误差

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.64522	1.38756	2.16655	2.96934	3.78984	5.47077	8.06525
实验值(ms)	0.64522	1.37764	2.12376	2.93745	3.65455	5.23880	7.59374
误差	0.000%	-0.715%	-1.975%	-1.074%	-3.570%	-4.240%	-5.846%

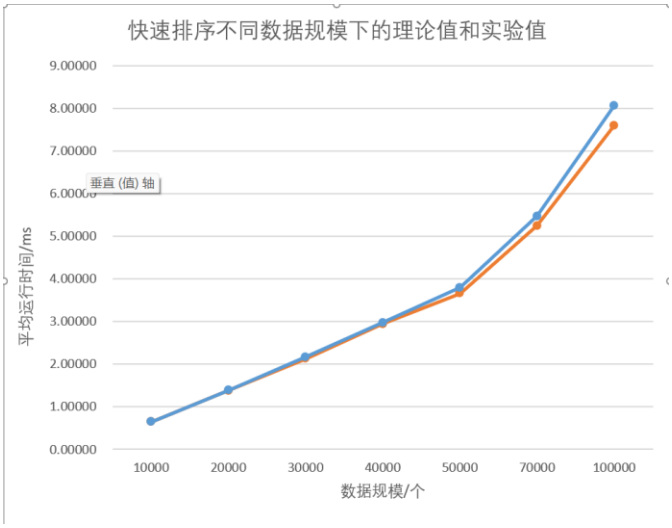

图 12 快速排序不同数据规模下理论值和实验值及其误差

我们发现，在数据规模增大的时候，快速排序实验值和理论值的误差逐渐增大，且实验值明显低于理论值，这是因为，我在计算理论值的时候，是以规模为 n=10000 的时间为基准，对数的底数为 2 进行计算的，而快速排序的实际对数底数比 2 略大，导致了理论值偏大。

3．分别以 n=10000，n=20000，n=30000，n=40000，n=50000 等等，重复 2 的实验，画出不同排序算法在 20 个随机样本的平均运行时间与输入规模 n 的关系

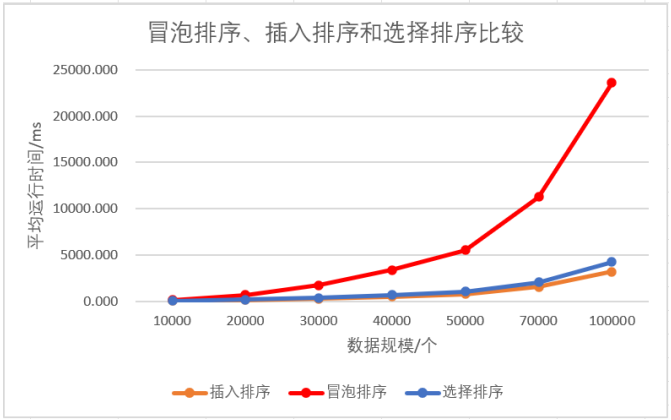

图 13 冒泡排序、插入排序和选择排序比较

我把三个时间复杂度量级都为$O(n^2)$的排序算法在相同数据下进行效率测试，插入排序与选择排序的效率都比较高，而冒泡排序的效率最为低下，由于冒泡排序需要进行的交换次数非常的多，所以常数较大，差不多是其他两个算法的三到四倍。

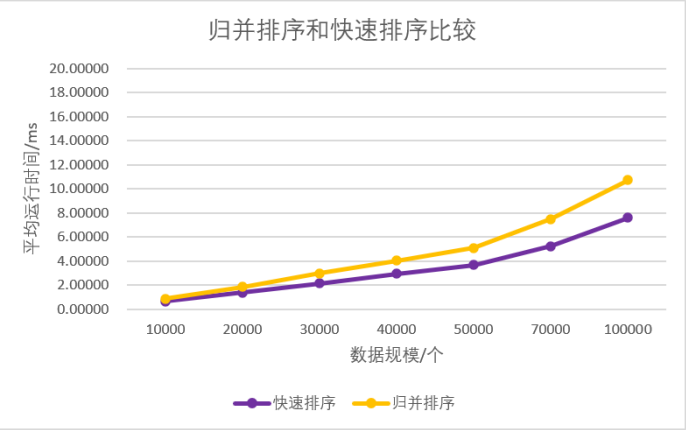

图 14 归并排序和快速排序比较

又把两个复杂度量级为$O(nlogn)$的算法进行了比较，可以看到快速排序的效率非常的高，这验证了理论结果，快速排序对数的底数会比归并排序大，所以它的常数比归并排序小一点，可以发现这两个算法的时间效率都远远高于复杂度量级为$O(n^2)$的三个算法的效率。

4、现在有 10 亿的数据（每个数据四个字节），请快速挑选出最大的十个数，并在小规模数据上验证算法的正确性。

我们把问题抽象为求解数据的**前k大问题**进行分析。在进行算法效率测试时，分别以以$n = 10000, n = 20000, n = 30000, n = 50000, n = 70000, n = 100000$，对于每个$n$随机生成 20 组测试数据，统计每个算法的平均运行时间。并在最后画出所有算法的运行时间比较图。

（1）冒泡排序暴力求解

算法流程

冒泡排序每一趟会把无序序列的最大值排到序列的最后一个位置，那么我们可以通过k次冒泡排序来找出k个最大数，他们排列在序列的最后k个位置。

复杂度分析

易得，k躺冒泡排序的时间复杂度为$O(kn)$。

算法正确性验证

我把 1-100 的所有数字随机排列了一下，然后找出前十大的数字，即 91-100。

```
Original data:
24 4 23 1 58 98 96 73 46 43 68 20 17 16 81 26 40 100 71 76 8 48 83 7 72 49 3 33 69 66 77 93 61 47 27 6 19 60 94 25 53 18
1 64 92 28 85 88 95 57 37 36 62 45 65 42 22 56 5 15 74 79 35 32 86 38 67 51 90 75 84 52 12 97 80 87 54 41 10 29 59 11 30
39 99 82 89 50 63 55 31 70 2 91 14 34 13 21 78 44 9
The top 10th numbers:
91 92 93 94 95 96 97 98 99 100
```

算法速率测试

表 6 冒泡排序求解前 k 大问题的运行效率随数据规模的变化表

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.48240	0.96480	1.44720	1.92960	2.41200	3.37680	4.82400
实验值(ms)	0.48240	0.97074	1.46375	1.98369	2.39991	3.36795	4.84143
误差	0.000%	0.616%	1.144%	2.803%	-0.501%	-0.262%	0.361%

（2）使用小根堆进行求解

算法流程

我们维护一个大小为k的小根堆，然后每读入一个数据，如果当前堆中元素数量小于k的话，那么就直接插入堆中；如果当前堆中元素数量大于等于k的话，就比较一下堆顶元素与当前元素的大小，如果堆顶元素比当前元素小的话，那么就把堆顶元素更换为当前元素，调整堆。当所有数据读入完毕后，堆中的k个元素就是前k大的数。比如图 15 所

示，我们想要找到 10 个数中的前 5 大的数。首先把前五个数字依次放入堆中，并调整堆。然后从第 6 个数开始，和堆顶元素比较大小（此时堆顶元素为堆中最小值），比如 0 比 1 小，那么跳过这个元素；比如 2 比 1 大，那么把 1 弹出堆，把 2 插入堆中并调整堆。

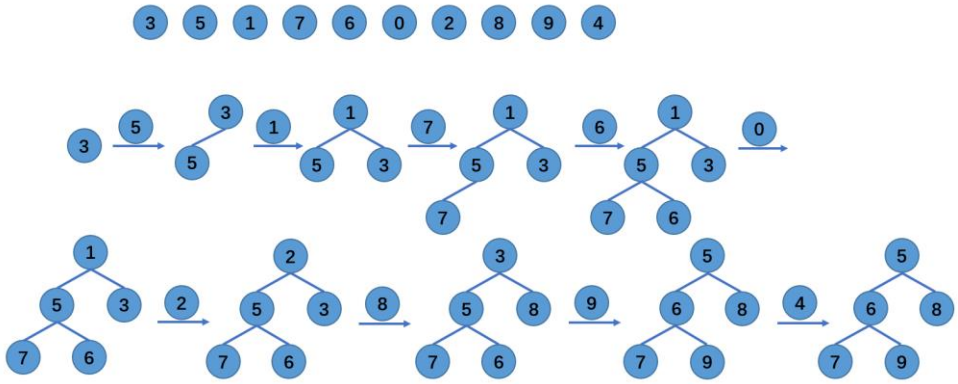

图 15 小根堆求解前 5 大示意图

复杂度分析

我们需要遍历n个数，每一次需要插入堆或者把元素弹出堆，插入堆和弹出堆之后需要调整堆，调整堆的运行时间上界为$logk$。首先要把前k个元素依次插入堆中，这个操作的复杂度为$klogk$。然后对剩下的$n-k$个元素进行分析：

最好情况下：

前面k个元素已经时前k大了，那么剩下的元素都不需要进行操作，时间复杂度为$O(klogk+n-k)$，若$k \ll n$，那么时间复杂度为$O(n)$。

最坏情况下：

剩下的$n-k$个元素都要进行弹出堆和插入堆的操作，运行时间为$(n-k)*2*logk$，故总运行时间为：

$$T(n) = klogk + (n-k)*2*logk = 2nlogk - klogk$$

那么时间复杂度为$O(nlogk)$。

平均情况下：

假设剩下的$n-k$个元素的操作次数为t，那么t的取值范围为$[0, n-k]$，假设每一种情况出现的概率相同且概率$p = \frac{1}{n-k+1}$，那么：

$$E(t) = \frac{1}{n-k+1}(0+1+\cdots+n-k) = \frac{n-k}{2}$$

故总运行时间为：

$$T(n) = k\log k + \frac{n-k}{2}2\log k = n\log k$$

故平均情况下的时间复杂度为$O(nlogk)$。

算法正确性验证

```
Original data:
26 48 8 85 62 75 63 66 82 30 29 97 81 13 61 2 71 74 3 37 22 89 54 23 11 25 80 44 32 12 17 99 68 77 27 94 70 35 78 67 90
43 72 60 28 76 52 16 92 96 38 57 56 58 84 9 7 18 64 83 21 55 24 79 41 95 45 1 91 47 42 31 5 34 87 98 69 20 46 100 59 36
15 53 19 50 39 65 49 88 10 4 86 33 73 14 51 40 6 93
The top 10th numbers:
91 92 93 94 95 96 97 98 99 100
```

算法速率测试

表 7 小根堆求解前 k 大问题的运行效率随数据规模的变化表

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.09340	0.18679	0.28019	0.37358	0.46698	0.65377	0.93395
实验值(ms)	0.09340	0.16938	0.25409	0.31050	0.37142	0.51343	0.72358
误差	0.000%	-9.321%	-9.315%	-16.887%	-20.464%	-21.466%	-22.525%

误差较大的原因应该是对于后面$n-k$个元素的操作次数不稳定，而计算理论值的时候是按照相同的操作次数去计算的。

（3）由快速排序演变的减治法求解

算法流程

我们回顾一下快速排序的过程，首先是调用$PARTITION$函数把整个待排序列划分为两部分，此时我们得知中间元素的左边都比中间元素小，右边都比中间元素大。对于我们的前k大问题，我们可以将前k大问题转化为把前k大排列在数组的前k个元素。也就是说，如果划分完整个序列之后（把大于中间元素的放左边），中间元素左边有num个数字（包括中间元素）。如果说$num == k$，那么当前前k个元素就是我们要找的前k大了；如果$num > k$，那么我们只需要递归左半部分求解；如果$num < k$，那么我们就只需要递归右半部分求解。如图 16 所示，我们想求出**十个元素中**的**前五小**的元素，假设我们默认以序列的左端点为主元进行$PARTITION$，那么第一次$PARTITION$之后，我们左边元素数量为 4，小于目标元素个数，故我们需要找到右半部分的第一小元素以凑齐五个元素。于是我们递归处理右边这个子问题，对右边进行$PARTITION$。此时左边元素比目标元素个数多了，那么我们就递归处理左边这个子问题，找出左边序列中最小的元素即可。我们可以看到，经过这个算法之后，前 5 小的元素就排列在数组前 5 个了，求前五大元素也是类似的道理。

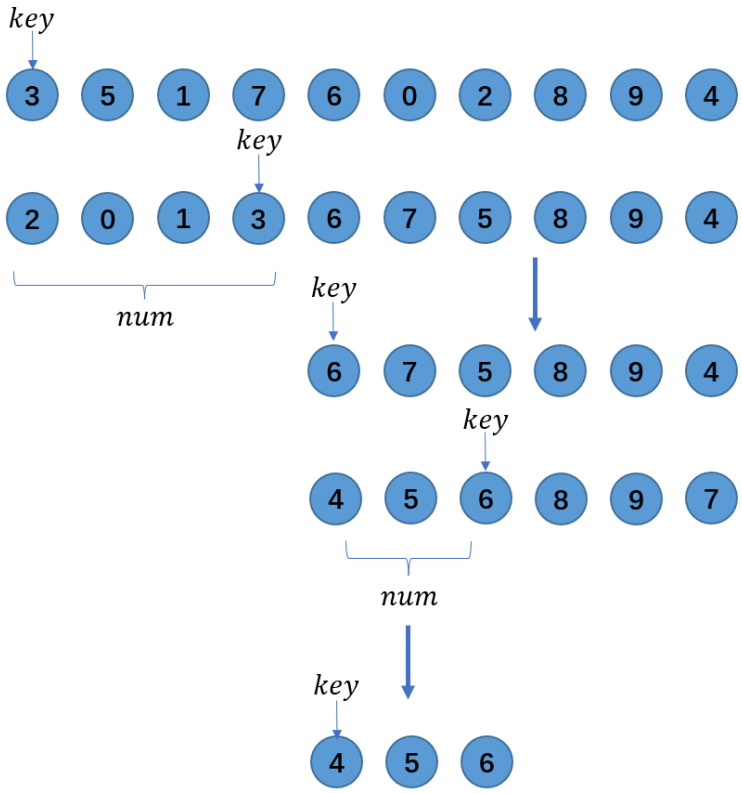

图 16 减治法求解前 5 小示意图

复杂度分析

对于一个规模为n的问题，我们在划分过程中需要比较n次，时间复杂度为$O(n)$，由于每一次划分之后，平均情况下子问题的规模近似为$\frac{n}{2}$，故总的运行时间为：

$$T(n)=n+\frac{n}{2}+\frac{n}{4}+\cdots+\frac{n}{2^{\log_2 n}}=n(1+\frac{1}{2}+\frac{1}{4}\cdots+\frac{1}{2^{\log_2 n}})$$

$$=n\frac{1-(\frac{1}{2})^{(\log_2 n+1)}}{1-\frac{1}{2}}=2n(1-\frac{1}{2}\frac{1}{n})=2n-1$$

故减治法求前k大的平均情况下的时间复杂度为$O(n)$。

算法正确性验证

```
Original data:
88 28 23 75 39 29 94 87 55 46 27 58 30 76 21 17 4 86 89 36 61 60 12 59 44 38 24 47 6 71 31 66 34 10 8 3 97 26 42 62 80 8
2 20 15 99 92 77 79 73 49 56 25 54 14 96 40 95 9 11 18 52 32 63 93 68 74 85 2 33 45 65 69 100 91 1 51 57 5 67 70 84 37 9
0 48 98 81 64 72 16 13 41 43 7 78 50 22 35 83 53 19
The top 10th numbers:
99 100 98 97 92 96 95 93 94 91
```

算法速率测试

表 8 减治法求解前 k 大问题的运行效率随数据规模的变化表

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.08636	0.17271	0.25907	0.34542	0.43178	0.60449	0.86355
实验值(ms)	0.08636	0.17274	0.29776	0.40940	0.42157	0.64111	1.03316
误差	0.000%	0.017%	14.936%	18.521%	-2.363%	6.058%	19.640%

（4）对减治法的优化

上面的减治法使用的$PARTITION$并没有特别稳定，最坏情况下，时间复杂度会达到$O(n^2)$，参考了算法导论，下面讲解最坏情况下时间复杂度上界为线性的选择算法。

算法流程

我们把优化之后的算法叫做选择算法，用$\boldsymbol{Select(a,\ l,r,k)}$表示在$\boldsymbol{a}$数组中的$\boldsymbol{[l,r]}$找出前$k$大的元素并且排在$\boldsymbol{[l,l+k-1]}$中。

我们的思路是使得最坏情况下$PARTITION$划分的两部分更加均匀，就是要选出一个合适的主元，因为主元决定了划分的结果。接下来介绍一下主元选取的规则：

- 首先我们把待划分序列分成$\lceil n/5\rceil$组，每一组有 5 个元素（最后一组有$n\ \%\ 5$个元素）。

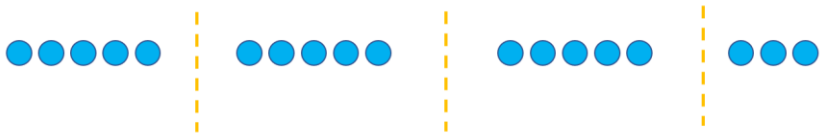

图 17 减治法优化示意图（1）

- 然后我们利用插入排序把每一组排序，求出每一组的中位数（排序后中间的数），默认偶数个数的组中的中位数为前一个。所以我们就获得了$\lceil n/5\rceil$个中位数。

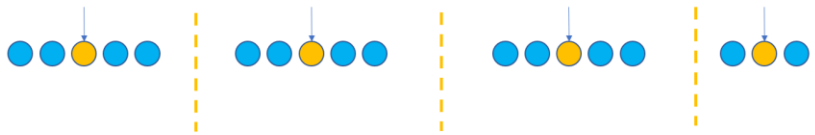

图 18 减治法优化示意图（2）

- 之后我们把所有中位数排在序列最前面，调用$\boldsymbol{Select(a,\ l,\ l+\lceil n/5\rceil-1,\frac{\lceil n/5\rceil}{2})}$求出所有中位数的中位数。把这个中位数作为$PARTITION$函数的主元。

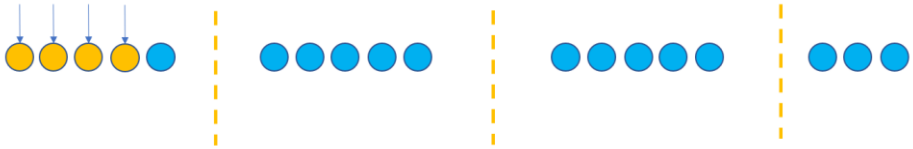

图 19 减治法优化示意图（3）

选好主元之后，剩下的两个步骤就和没优化前的减治法一样了。

复杂度分析

假设总体运行时间为$T(n)$

易得前两步的时间复杂度都为$O(n)$，第三步可以用$T\left(\left\lceil\frac{n}{5}\right\rceil\right)$表示，$PARTITION$的时间复杂度为$O(n)$，下面我们分析一下最后一步也就是分治那一步的时间复杂度表示。

如图 20，假设我们选出的主元为m，那么比主元大的数至少有阴影部分那么多。可以证明，至少有$\frac{3n}{10}-6$个，那么小于主元的元素最多就只有$\frac{7n}{10}+6$个，故分治下去的时间复杂度上界可以表示为$T\left(\frac{7n}{10}+6\right)$。

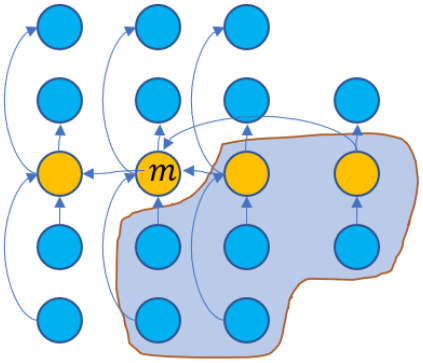

图 20 减治法时间复杂度分析示意图

故：$T(n)=O(n)+T(\lceil n/5\rceil)+T(7n/10+6)$

我们使用代入法证明$T(n)$的上界是线性的，我们先假设$T(n)\le cn$，$O(n)=an$

那么有：

$$T(n)<=an+c\lceil n/5\rceil+c(7n/10+6)<=an+cn/5+c+c7n/10+6c$$
$$=c9n/10+7c+an=cn+(-cn/10+7c+an)$$

故：$(-cn/10+7c+an)<=0$

得：$c(n-70)>=10an$

故：

$$n>70, c>=\frac{10an}{n-70}$$
$$n<=70, T(n)=O(1)<=cn$$

从这里也可以看出，在前 10 大的问题中，这个方法的常数较大，甚至大于暴力求解的方法，下面的数据测试也证明了这一点。但是如果推广到前 k 大的话，这一个方法就比较稳定。

算法正确性验证

```
Original data:
21 35 37 5 53 20 72 50 34 75 26 13 54 43 71 80 8 39 63 47 30 87 58 45 74 4 66 2 11 62 42 98 69 28 46 15 68 10 77 64 36 1
4 94 93 24 3 51 57 65 56 7 40 67 19 29 82 86 92 17 81 89 44 12 23 55 73 16 1 48 85 38 78 25 32 18 27 9 84 99 41 83 33 52
 70 31 96 76 6 91 88 59 22 79 97 49 90 95 61 60 100
The top 10th numbers:
98 96 99 95 100 97 94 92 93 91
```

算法速率测试

表 9 选择算法求解前 k 大问题的运行效率随数据规模的变化表

数据规模（个）	10000	20000	30000	40000	50000	70000	100000
理论值(ms)	0.47794	0.95587	1.43381	1.91174	2.38968	3.34555	4.77935
实验值(ms)	0.47794	0.92825	1.50164	1.86642	2.34334	3.56133	5.15998
误差	0.000%	-2.890%	4.731%	-2.371%	-1.939%	6.450%	7.964%

（5）各类算法比较

我们可以看到，减治法和小根堆方法的效率都比较高，且两者相差不大；反而优化之后的减治法由于常数较大，在小数据下与冒泡排序这样的暴力求解方法效率差不多。由于时间原因就没有在大数据下进行测试了。我们可以看到，并不是说线性算法就一定比暴力算法更优，在特定情况下，不复杂的暴力求解可能比复杂的优化更优。

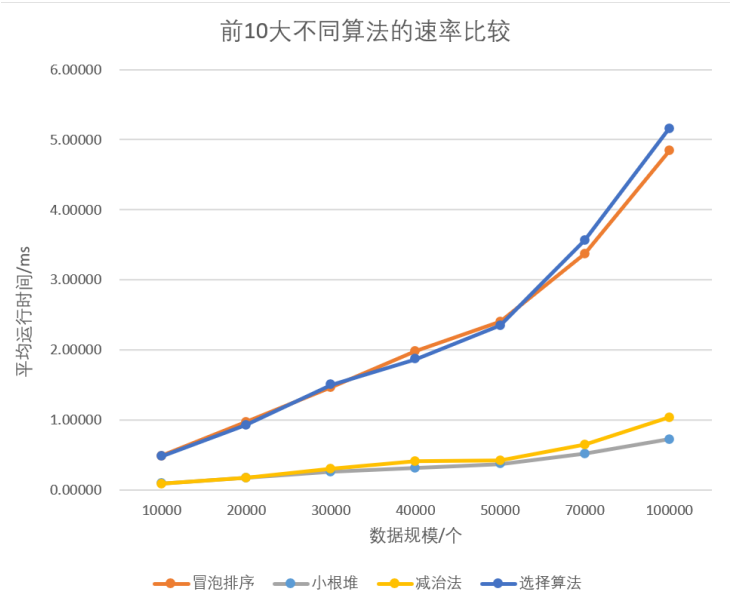

图 21 不同算法解决前 k 大问题的速率比较图

三．实验心得

这次实验花了比较多时间在复杂度分析上和测试实验数据上。在复杂度分析上，查阅了大量的参考资料，比如说《算法导论》，从头到尾搞懂了每一个算法的时间复杂度和空间复杂度。对于递归实现的算法的时间复杂度一开始一筹莫展，后来学习了递归树分析方法和代入法等等，问题也就迎刃而解。

其他的方面例如编程实现方面，倒也没遇到什么太大的问题。由于是第一次实验也比较简单。

四. 附件说明

- 讲解 PPT
- Code 文件夹
  - bubble_sort.cpp（冒泡排序实现及数据测试）
  - generatedata.cpp(随机数据生成器)
  - insert_sort.cpp（插入排序实现及数据测试）
  - merge_sort.cpp （归并排序实现及数据测试）
  - quick_sort.cpp （快速排序实现及数据测试）
  - selection_sort.cpp （选择排序实现及数据测试）
  - topK.cpp （解决前 k 大问题的四个算法实现及数据测试）
- result 测试结果数据表

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。