深 圳 大 学 实 验 报 告

课程名称：算法设计

实验项目名称：实验 1 排序算法性能分析

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：刘刚

报告人：林宪亮 学号：2022150130 班级：国际班

实验时间：2024 年 3 月 17 日

实验报告提交时间：2024 年 3 月 22 日

教务处制

实验目的

1. 掌握选择排序、冒泡排序、插入排序、合并排序、快速排序算法原理
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。
3. 求解 TOP K 问题，并分析比较不同算法效率。

实验内容：

1. 实现选择排序、冒泡排序、插入排序、合并排序、快速排序算法。

（1）选择排序

原理：

选择排序的原理是每次都从待排序的序列中选择最小的一个元素，然后与待排序序列的第一个元素交换，之后待排序序列就是原本的待排序序列除去刚刚交换的第一个元素，确定完新的待排序序列后，重复之前的排序操作直到整个序列排序完成。

举例：

以升序排序为例，对序列 2，3，5，4，1 的选择排序过程如下：
第一步选择整个序列最小的元素 1 与第一个元素交换：1，3，5，4，2。
第二步选择除去第一个元素的序列中的最小元素与整个序列的第二个元素交换：1，2，5，4，3。
依次类推：
第三步：1，2，3，4，5。
第四步：1，2，3，4，5

伪代码：

Algorithm 选择排序
Input：待排序的数组
Output：选择排序后的代码

```
def selection_sort(arr):
    n = length of arr
    for i in range(n-1)://遍历
        min_index = i
        for j in range(i + 1, n)://find min element
            if arr[j] < arr[min_index]:
                min_index = j
        Swap the min element and the i element
    return arr
```

时间复杂度：

共有两层循环，外层循环需要遍历 n-1 次，内层循环则需要遍历 n-1-i 次，所以时间复杂度为 O（n^2）。

算法优化：

1.可以同时寻找未排序序列的最小值和最大值，最小值与序列首元素，最大值和序列末元素交换，这样可以加快加快排序的速度。

（2）冒泡排序：

原理：

冒泡排序的基本原理是每次通过对比相邻的元素，把比较大的元素像右移动，使得每次遍历后就可以把最大的元素移动到序列的末尾，依次类推，即可完成序列的排序。

举例：

以升序排序为例，对序列 2，3，5，4，1 的冒泡排序过程如下：

第一次排序：2，3，4，1，5

第二次排序：2，3，1，4，5

第三次排序：2，1，3，4，5

第四次排序：1，2，3，4，5

伪代码：

Algorithm 冒泡排序

Input：待排序的数组

Output：冒泡排序的数组

```
def maopao_sort(arr):
    n = length of arr
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
              Swap the large element and the small element.
    return arr
```

时间复杂度：

冒泡排序过程需要两层循环，外层循环需要 n-1 次，内层循环需要 n-1-i 次比较，因此冒泡排序的时间复杂度为 O（n^2）。

.

算法优化：

1. 在冒泡排序中，每次比较都可能进行交换操作。可以引入一个标志位来表示本轮是否进行过交换，如果没有进行交换，则说明数组已经有序，可以提前结束排序过程。

2. 冒泡排序的特点是每次都将最大（或最小）的元素往后（或往前）冒泡。在每一轮排序中，可以记录上一次发生交换的位置，该位置之后的元素已经有序，无需再进行比较。

（3）插入排序

原理：

插入排序把序列分为已排序部分和未排序部分，已排序部分一开始包含一个元素，之后通过遍历，把未排序部分的元素逐个插入已排序序列的合适位置。

举例：

以升序排序为例，对序列 2，3，5，4，1 的插入排序过程如下：

第一次插入：2，3，5，4，1.

第二次插入：2，3，5，4，1

第三次插入：2，3，4，5，1

第四次插入：1，2，3，4，5

伪代码：

Algorithm 插入排序

Input：待排序的数组

Output：插入排序后的数组

```
1.  def insert_sort(arr):
2.      n = length of arr
3.      for i in range(1, n):
4.          cmp = arr[i]
5.          j = i - 1
6.          while j >= 0 and arr[j] > cmp:
7.              arr[j + 1] = arr[j]
8.              j -= 1
9.          Insert the element to the correct position
10.     return arr
```

时间复杂度：

插入排序有两层循环，外层循环的次数为 n-1，而内层循环最差的情况则需要进行多次交换和比较，因此插入排序的时间复杂为 O（n^2）

算法优化：

1.可以使用二分查找法寻找元素应该插入的位置，这样可以减少比较次数，对于大规模数据，会有更好的效果。

（4）合并排序
原理：将待排序的序列递归的分成两个字序列，直到每个子序列都只有一个元素，然后对这些子序列递归的合并成有序的母序列，当递归结束，返回最终的母序列也会排序成功。

举例：
以升序排序序列 3，4，6，2，1，5 为例。
第一次分裂成：3，4，6 和 2，1，5
第二次分裂成：3，4 和 6，2，1 和 5
第三次分裂成：3 和 4，6，2 和 1，5
第一次合并成：3，4 和 6，1，2 和 5
第二次合并成：3，4，6 和 1，2，5
第三次合并成：1，2，3，4，5，6

伪代码：
Algorithm 合并排序
Input：待排序的数组
Output：合并排序后的数组

```
def hebing_sort(arr):
    if len of arr > 1:
        mid = len of arr // 2
        left_arr = arr[:mid]
        right_arr = arr[mid:]
        hebing_sort(left_arr)
        hebing_sort(right_arr)

        i, j, k = 0, 0, 0
        while i < mid and j < len(right_arr):
            合并 两个子序列
        while i < mid:
            子序列剩余元素直接排在后面
        while j < len(right_arr):
            子序列剩余元素直接排在后面
        return arr
```

时间复杂度：
合并排序的时间复杂度为 O(nlogn)。这是由于在每次合并的过程中，需要比较和移动的次数都与 n 成正比，而总共需要进行 logn 次合并操作。因此，总体时间复杂度为 O(nlogn)。

算法优化：
1. 当数组合并时，如果左子数组的最大值小于右数组的最小值，则不需要额外的比较，直接合并。
2. 可以使用并行技术，让递归的过程并行进行，提高效率。

（5）快速排序
原理：快速排序的原理是选择一个基准元素，将小于基准元素的元素放在基准元素的左边，大于基准元素的元素放在基准元素的右边，再对左右两边序列进行递归排序得到最终的有序序列。

举例：
以对序列 4，3，5，2，1 进行快速排序为例：
选择第一个元素 4 为基准元素，小的放左边，大的放右边。
左边部分为 3，2，1，右边部分为 5
对左边进行递归操作，选择基准元素为 3，得到左边为 2，1（再递归一次得到 1，2），右边部分没有元素。
对右边进行递归操作，得到 5.
最后进行递归合并，的带排序好的序列 1，2，3，4，5。

伪代码：
Algorithm 快速排序
Input：待排序的数组
Output：快速排序后的数组

```
def fast_sort(arr):
    if len(arr) <= 1:
        return arr
    基准元素 = arr[0]
    left, right = 0, len(arr) - 1
    while left < right:
        while left < right:
            if 基准元素 > arr[right]:
                Swap 基准元素 和 比较元素
                left += 1
                break
            right -= 1
        while left < right:
            if 基准元素 < arr[left]:
                 Swap 基准元素 和 比较元素
                right -= 1
                break
            left += 1

    return fast_sort(arr[:left]) + [基准元素] + fast_sort(arr[left + 1:])
```

时间复杂度：
快速排序的平均时间复杂度为 O(nlogn)，最坏情况下为 $O(n^2)$。平均情况下，快速排序通过每一次划分将待排序序列减少一半，因此时间复杂度为 O(nlogn)；最坏情况发生在每次划分时基准元素都是最大或最小元素，此时需

要进行 n 次划分，时间复杂度为 $O(n^2)$。

算法优化：
1. 可以随机选择基准元素，这样可以防止一些极端的情况的出现。
2. 可以对算法进行并行处理，缩短排序时间。

2. 以待排序数组的大小 n 为输入规模，固定 n，随机产生 20 组测试样本，统计不同排序算法在 20 个样本上的**平均运行时间。**

表 1：输入规模为 10w 时，不同算法的平均运行时间

算法	选择	冒泡	插入	合并	快速
平均时间（s）	166.97	435.50	206.64	0.23	0.20

说明：为了方便画图，我使用了 python 进行实验，所以在运行时间上会有些长。

3. 分别以 n=10 万，n=20 万，n=30 万，n=40 万，n=50 万等等（输入规模 n 可以直至极限），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。

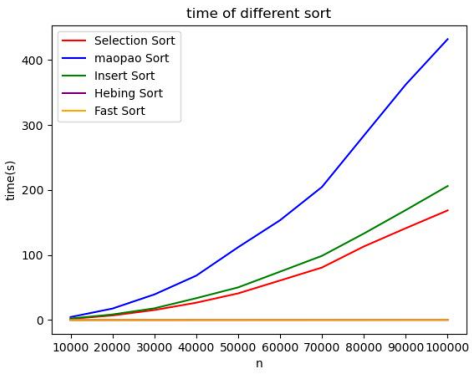

图 1 输入规模为 1w-10w 时关系

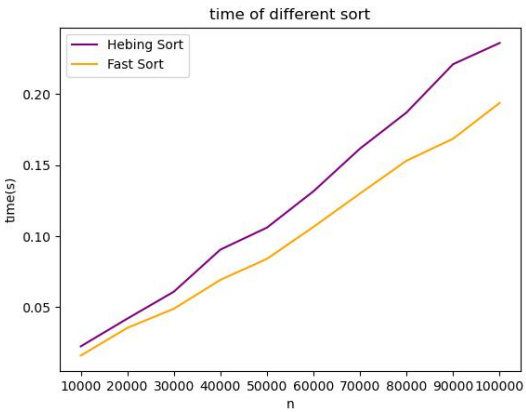

图 2 快排和合并排序时间规模关系

从图中可以看出，同等规模下，冒泡排序需要最多的时间，并且其时间增长率也是最高的，所以对于大规模数据的排序，冒泡排序是不适合的。插入和选择排序比起冒泡排序效率会高许多，但是对于大规模的数据依旧需要比较长的时间。相比之下，快速排序和合并排序就展现除了极高的性能优势，在图一中甚至无法观测到它们有什么时间上的变化，在图二中也只能看出只有微小的变化，因此这两种排序更加适合大规模数据的排序。

说明：因为在输入规模为几十万时，冒泡排序需要的时间实在是太长了，所

以我选取了规模为 1w 到 10w 进行了实验。

由于冒泡排序，选择排序以及插入排序在大规模数据下所需要的时间实在太长了，不具备时间可接受性，于是我只对快速排序以及合并排序在大规模数据下进行了实验。

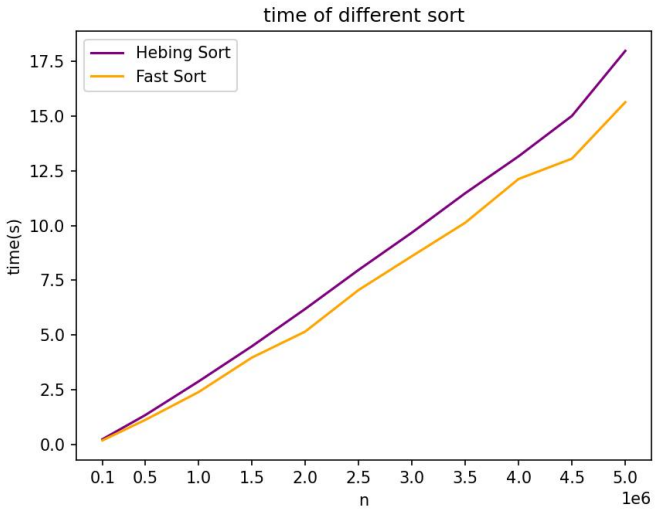

图 3 大规模数据下实验结果

从图 3 中可以看出，即使在输入规模为五百万时，快速排序只需要 15 秒左右，合并排序只需要 17.5 秒左右，而且随着规模变大，函数的斜率并没有出现很大的上升，快速排序和合并排序还是很适合大规模数据的排序的。

4、画出理论效率分析的曲线和实测的效率曲线，注意：由于实测效率是运行时间，而理论效率是基本操作的执行次数，两者需要进行对应关系调整。

（1）选择排序

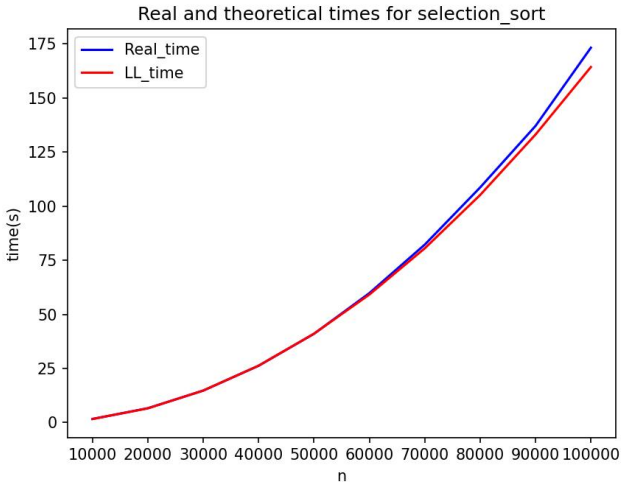

图 5 选择排序的理论曲线和实际曲线

如图 5，红色曲线是选择排序的理论效率曲线，蓝色曲线是实际的效率曲线。我选择了数据规模为 1w 的时候的时间作为基准，由于选择排序的时间复杂度为 O（n^2），所以我假设理论曲线的函数为：$Y=w*n*n^2$（n 是输入规模，w 是一个权重，y 是时间），然后通过 n 为 1w 的时候的时间计算出 w 的值。通过对比红色曲线和蓝色曲线可以得出结论实测值和理论值是基本一致的。

（2）冒泡排序

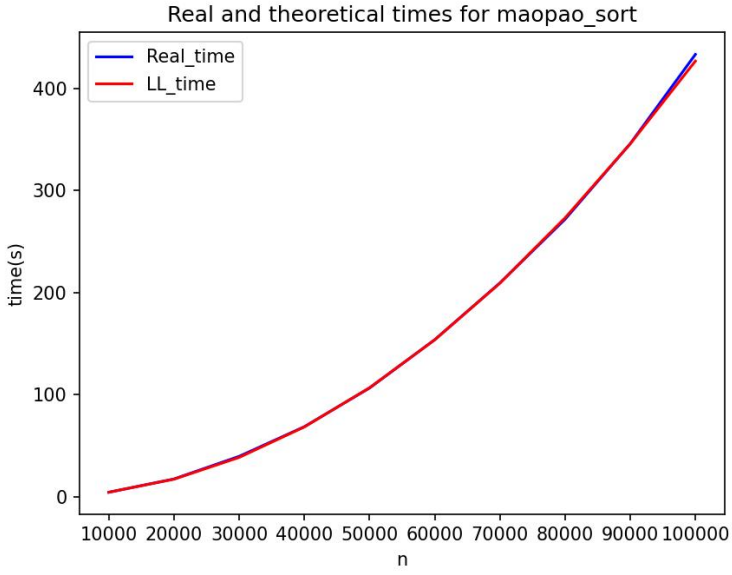

图 6 冒泡排序的理论曲线和实际曲线

如图 6，红色曲线是冒泡排序的理论效率曲线，蓝色曲线是实际的效率曲线。我选择了数据规模为 1w 的时候的时间作为基准，由于冒泡排序的时间复杂度

为 O（n^2），所以我假设理论曲线的函数为：Y=w*n*n^2（n 是输入规模，w 是一个权重，y 是时间），然后通过 n 为 1w 的时候的时间计算出 w 的值。通过对比红色曲线和蓝色曲线可以得出结论实测值和理论值是高度一致的。

（3）插入排序

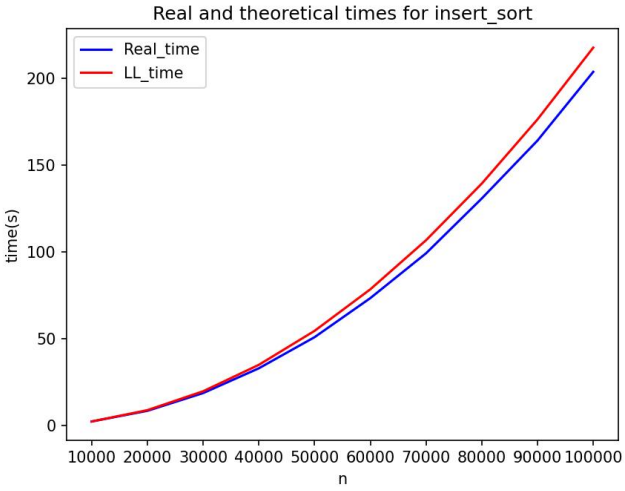

图 7 插入排序的理论曲线和实际曲线

如图 7，红色曲线是插入排序的理论效率曲线，蓝色曲线是实际的效率曲线。理论曲线的设计方式和冒泡排序一致。通过对比红色曲线和蓝色曲线可以得出结论实测值和理论值是基本一致的。

（4）合并排序

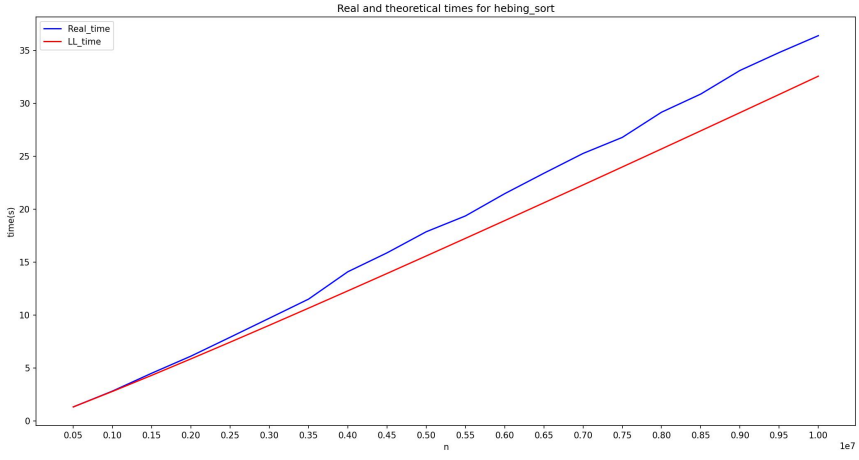

图 8 合并排序的理论曲线和实际曲线

如图 8，红色曲线是合并排序的理论效率曲线，蓝色曲线是实际的效率曲线。由于合并排序的时间复杂度为 O(nlogn)，所以我假设理论曲线为 Y=w*nlogn，设置输入规模为五十万时的时间为基准值，计算出 w 的值。通过对比实际曲

线和理论曲线可以看出，在规模比较小时（50w 到 350w 时），实际曲线和实际曲线是很接近的，但随着规模的增大，两条曲线的差距就开始变大了。合并算法的运行时间为 T（n）=nlogn+n，可能是我忽略了 n 导致当 n 增大时，实际值和理论值相差较大。

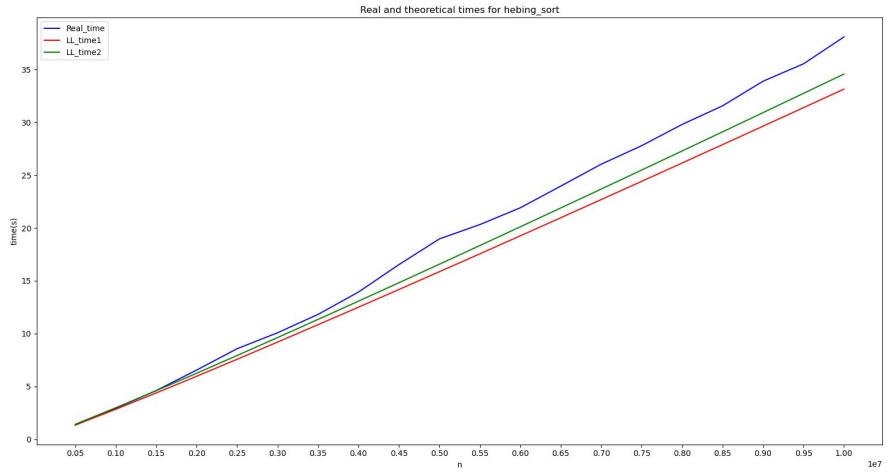

图 9 改良后的版本

绿色曲线为考虑了单独项 n 之后的函数，可以看出对比红色曲线有了一定的改进。

（5）快速排序

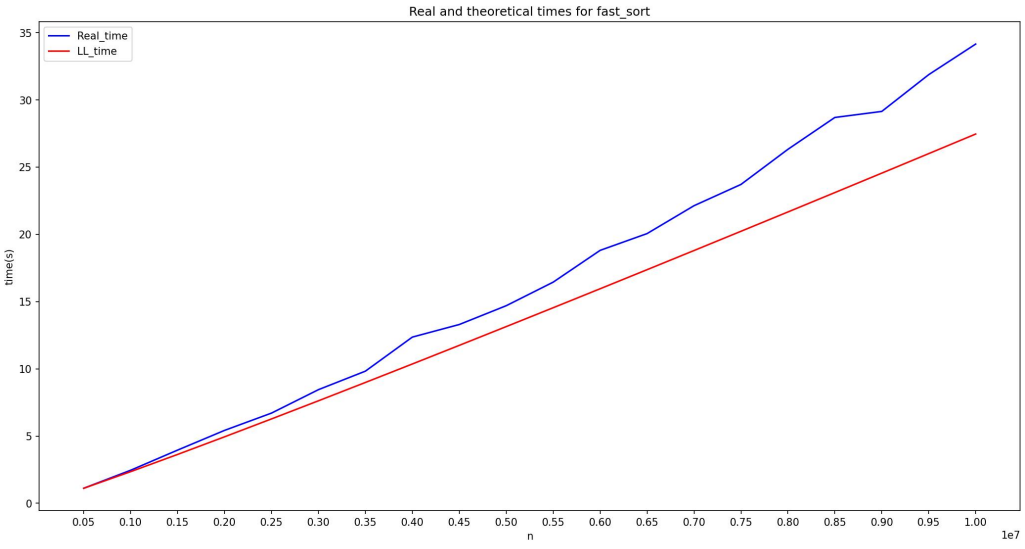

图 10 快速排序的理论曲线和实际曲线

如图 10，红色曲线是快速排序的理论效率曲线，蓝色曲线是实际的效率曲线。由于快速排序的时间复杂度为 O（nlogn），所以我假设理论曲线为 Y=w*nlogn，设置输入规模为五十万时的时间为基准值，计算出 w 的值。通过对比实际曲线和理论曲线可以看出，在规模比较小时（50w 到 350w 时），实际曲线和实际曲线是很接近的，但随着规模的增大，两条曲线的差距就开始变大了。存

在的问题和合并排序类似。可能与合并排序类似，我有忽略了后面的 a*n 这些项，也也能因为重复的数据过多，影响了快速排序的性能。

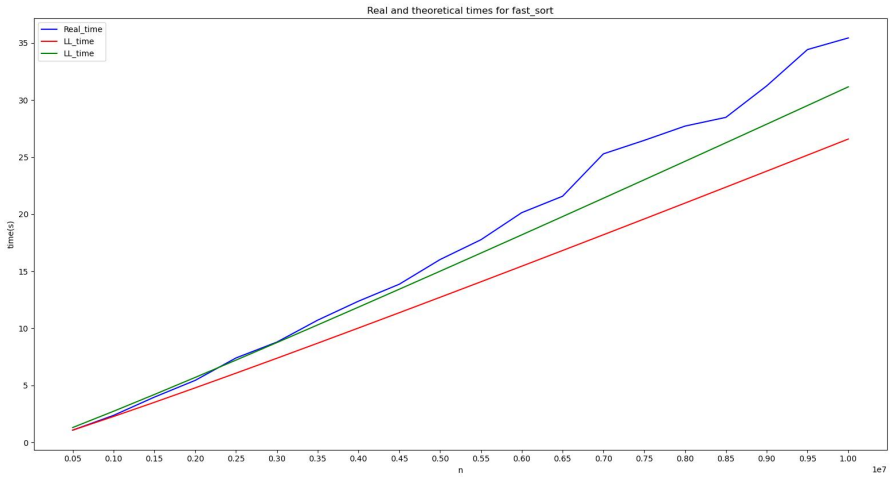

图 11 改良后的版本

加入了“n“这个项的考虑，以及降低了数据的重复率，改良后的曲线（绿色曲线）跟实际曲线更加相近了。

5、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。

算法 1：

对于此问题，我采用修改后的快速排序进行解决，可以把快速排序的返回条件由数组的长度小于等于 1 改成小于等于 10（100/1000/10000/100000），对这样快速排序后的数组取最后的 20（200/2000/20000/200000）个数据再进行一次正常的快速排序，取最后十个数据即是最大的 10 个数据。

算法解释：

下面证明当快速排序算法的返回长度为 1 时，快速排序后的序列的最后 2*L 个元素必然包含最大的 L 个元素。

若快速排序的返回长度为 1，则定义开始返回的序列为块（即长度小于等于 L 的元素序列），那么最后 2*L 个元素有以下两种情况：

1. 拥有 n 个完整的块，因为块间是有序的，所以这 n 个块一定是最大的那 2*L 的元素，所以最大的 1 个元素也会在其中。

2. 拥有 n 个完整块和一个不完整的块，因为不完整块的长度必然小于 10，那么这 n 个完整块的元素个数之和必然大于 L，且块间是有序的，那么在 n 个完整块必然包含最大的 L 的元素。

所以取最后的 2*L 个元素进行快速排序则可以得到最大的 L 个数据。

伪代码：
Algorithm 修改的快速排序
Input：待排序的数组
Output：可以方便找出最大十个数的数组

```
def fast_sort1(arr)://修改条件的快速排序
    if length of arr <= 10:
        return arr
    mid = arr[0]
    left, right = 0, length of arr - 1
    while left < right:
        while left < right:
            if mid > arr[right]:
                Swap mid and arr[right]
                left += 1
                break
            right -= 1
        while left < right:
            if mid < arr[left]:
                Swap mid and arr[left]
                right -= 1
                break
            left += 1
    return fast_sort1(arr[:left]) + [arr[left]] + fast_sort1(ar
r[left + 1:])
```

Algorithm 快速排序
Input：待排序的数组
Output：快速排序后的数组

```
def fast_sort2(arr)://正常的快速排序
    if length of arr <= 1:
        return arr
    mid = arr[0]
    left, right = 0, length of arr - 1
    while left < right:
        while left < right:
            if mid > arr[right]:
                Swap mid and arr[right]
                left += 1
                break
            right -= 1
        while left < right:
            if mid < arr[left]:
                Swap mid and arr[left]
                right -= 1
```

```
                break
            left += 1

    return fast_sort2(arr[:left]) + [arr[left]] + fast_sort2(ar
r[left + 1:])
```

测试结果： 我进行了 30 次测试，用一般快速排序得出的结果和使用我的算法得出的结果均是一致的。

性能分析：

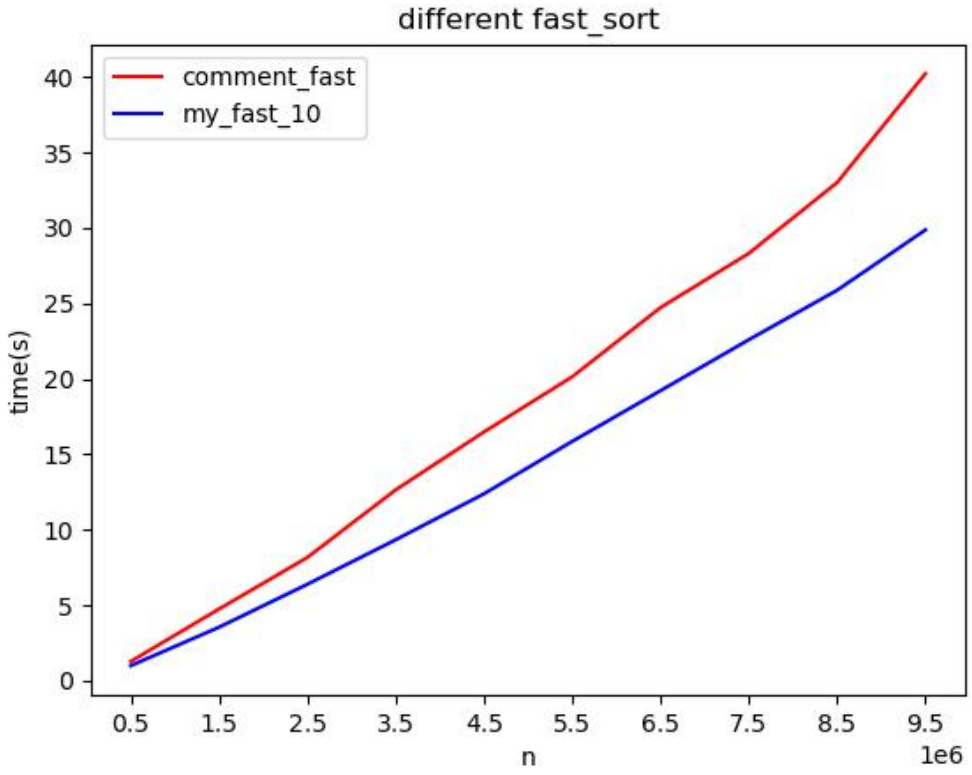

图 12 两种快速排序性能

如图 12，红色曲线是一般的快速排序，蓝色曲线是我修改后的快速排序（以 10 为返回长度，找出最大十个数），可以看出在绝大多数情况下我的快速排序都要快过一般的快速排序，拥有更好的时间效率。

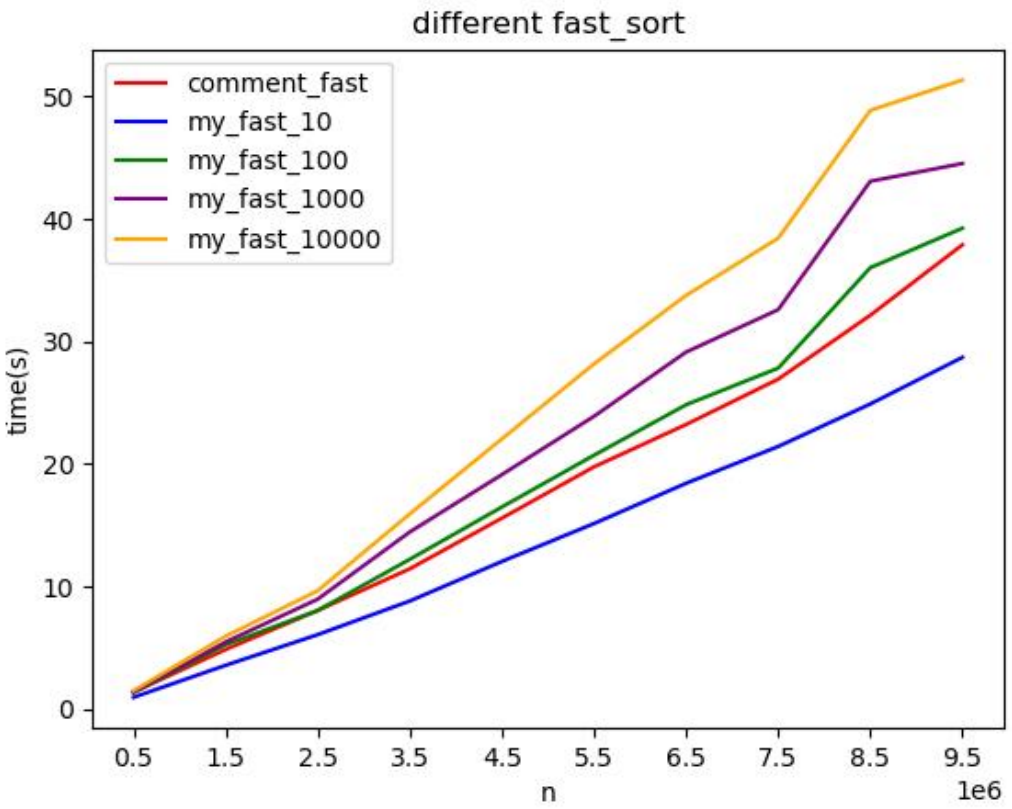

图 13 不同返回长度的效率曲线

如图 13，我对不同的取值下的效率曲线进行了比较，黄色曲线为 L 取为 10000 时的效率曲线，紫色曲线为 L 取 1000 时的效率曲线，绿色曲线为 L 取 100 的效率曲线，蓝色曲线为 L 取 10 的效率曲线，红色为正常快速排序的效率曲线，通过对比可以看出，L 取 10 时的效率会更好。

算法 2：
对于本题，只需要找出最大的十个数，那么使用堆排序也是合适的。

堆排序：堆排序是一种基于堆数据结构的排序算法，分为最大堆和最小堆两种类型。在最大堆中，父节点的值总是大于或等于其子节点的值；而在最小堆中，父节点的值总是小于或等于其子节点的值。本题可以使用最大堆实现。

性能分析：

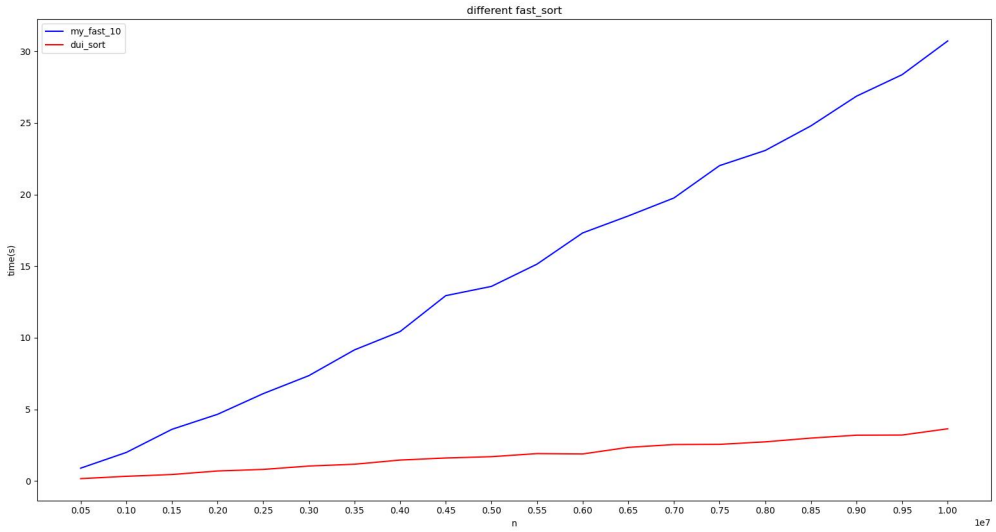

图 14 堆排序性能

如果所示，红色曲线代表的是堆排序，蓝色曲线代表的是我的快速排序，可以看出堆排序远远的快于我的快速排序，具有更佳的性能。时间复杂度只有 O（n），并且不需要额外的空间。

伪代码：
Algorithm 堆排序
Input：数组，数组长度，要调整的位置
Output：调整后的数组

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        Swap largest and i
```

```
        heapify(arr, n, largest)

```

Algorithm 堆排序
Input：数组
Output：最大的十个数

```
def heap_sort(arr):
    n = length of arr

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract the top 10 elements
    top_10 = []
    for i in range(n - 1, n - 11, -1):
       Put the largest to the end
        top_10.append(arr.pop())
        heapify(arr, i, 0)

    return top_10
```

经验总结：

1. 在五种算法中，快速排序有着最好的性能，合并排序次之，选择排序和插入排序排第三第四，冒泡排序性能最差。在小规模数据中它们的用时差不多，但是在大规模数据中，快速排序和合并排序具有巨大的优势。
2. 快速排序和合并排序的理论时间和实际实际随着输入规模的增大，会有愈发显著的差距，实际运行时间会大于理论的运行时间。在考虑 n 项后确实可以减小这种差距，但是在更大规模数据上估计考虑 n 项就不能有很好的减缓效果了。
3. Python 的对于这些算法运行效率会和用 c++运行相差 50 倍左右，python 还是相对低效。
4. 对于求解问题 5，对于如 10 亿这种大规模的数据，肯定得使用时间复杂度低的排序算法，那么就是从合并排序和快速排序中选择一个，因为相对合并排序，快速排序展现出了更优秀的性能，于是我选择使用快速排序来解决。但是用原版的快速排序时间性能也是不高的，那么有没有什么办法提高性能呢，因为题目说的是找出最大的十个数而非需要玩玩全全的排序，又根据快速排序的递归过程中块间有序的特性，于是可以对其进行修改，使其更加的适合我们的实际问题，降低时间成本，最后通过对比也是证明了相比一般的快速排序时间复杂度确实降低了。

算法并不是一成不变的，但也不一定需要取寻找全新的算法，可以对现有算法进行修改使其更加贴切待解决的问题。

对于自己想到的改进的算法，需要大胆尝试，如果发现几次测试下来结果都没有问题，再去想办法证明。

但是对于选出最大 k 个数这种问题，并不需要完全排序，所以堆排序还是更优的算法，尽管改进后的快速排序有着比原快速排序更好的性能，但是比起堆排序这种更加适合的算法，性能还是差了很多。

指导教师批阅意见：

成绩评定：

指导教师签字：

年　　月　　日

备注：