lab1-排序算法

姓名:裴启智

学号:PB18111793

1. 实验内容

1. 排序n个元素，元素为随机生成的0到$2^{15}-1$之间的整数， n的取值为：2^3 2^6 2^9 2^{12} 2^{15} 2^{18}
2. 实现以下算法
    - 直接插入排序
    - 堆排序
    - 快速排序
    - 归并排序
    - 计数排序

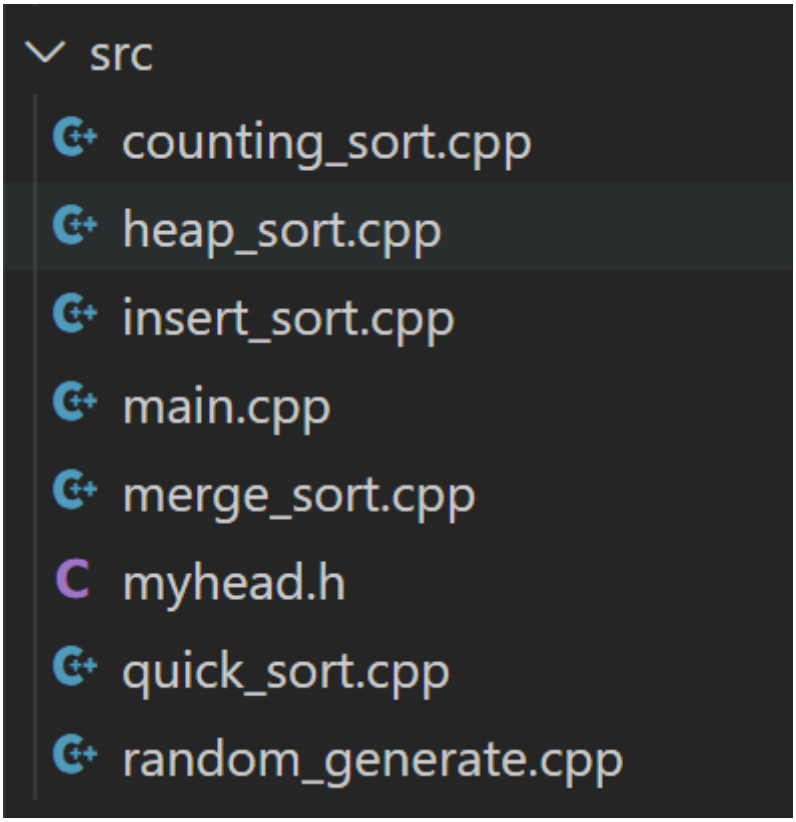
2. 实验设备和环境

- 编译环境：Windows10-mingw-w64
- 硬件详情

CPU	AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx
CPU核心数	4
逻辑处理器	8
主频	2100 Mhz
外频	100 Mhz
一级数据缓存	4 * 32KB
一级指令缓存	4 * 64KB
二级缓存	4 * 512KB
数据宽度	64 bit

3. 实验方法和步骤

整体架构



- 通过头文件`myhead.h`将不同的`cpp`文件联系起来,实现跨文件函数的调用
- `main.cpp`为整个工程的入口,执行次序如下
  1. 通过`random_generate.cpp`生成2^{18}个随机数放入`input.txt`中
  2. 在`main.cpp`中将`input.txt`读入到一个长度为2^{18}的数组中,再将这个数组部分复制给其他规模的数组,这样可以减少函数`IO`的开销. 每种规模数组都有一个副本,用于在副本上进行就地排序并存放结果(计数排序除外,这个副本直接作为计数排序存放结果的数组)
  3. 依次调用5种不同的排序算法,每次调用前对计时变量以及存放结果的数组进行初始化,调用结束后将结果输出到对应的文件
- 注意事项
  - 需要修改`CMakeList`,增加可用栈空间的大小,否则无法定义两组需要大小的数组
  - 有头文件时不能重复定义函数,如堆排序和快速排序都要用到的`exchange()`函数,只需定义一次并在头文件中声明即可
  - 一些全局变量可以只定义一次,并在头文件中用`extern`修饰即可
  - C++文件流中使用了绝对路径
  - 由于输入结果路径中不能含有中文,故在绝对路径中将文件名作了一定的修改

生成规模为2^{18}的随机数

- C++ int是4B，去掉符号位，最大$2^{31}-1$，所以元素可直接用int类型 即可满足实验要求
- `stdlib.h`中的`rand()`生成随机数范围是0到RAND_MAX=$2^{15}-1$,符合实验要求的范围
- 通过使用`srand()`,给`srand()`提供种子,可以在每次运行`main`函数时生成不同分布的输入,便于后续测试
- 具体生成代码为`random_generate.cpp`,通过C++的`ofstream`生成2^{18}个随机数

计时

- 为了代码可以在不同平台上运行,采用了C++11标准的计时方法:`chrono`库,其所有实现均在其所有实现均在`std::chrono namespace`下
- 这种计时方法可以灵活的变换单位,只需在

```
auto duration = duration_cast<>(end - start);
```

的 `<>` 内指定参数即可,可用的参数有 `hours, minutes, seconds, milliseconds, nanoseconds` ,再使用 `count()` 方法返回时间即可

- 计时被嵌入到每个排序函数的内部以减少函数调用对计时的影响. 对于递归的归并排序和快速排序, 为了方便计时,对它们外包了一个函数,用于计时的开始和结束

五种排序算法

- 整体思路和课本差不多,需要注意的如下
  - 书上各种数组都是从下标1开始的, 而实验中数组下标都是从0开始的.下标处理不当会出现数组访问越界的情况, 引起 `segmentation fault` 等. 我采用的是根据实际情况对循环变量等进行加一或减一的改动, 一种较为简便的思路是将每个数组大小多设置1, 而不使用0号元素
  - 堆排序数据结构的处理, 课本上二叉堆是从1开始编号的, 而实验中二叉堆是从0开始编号的, 这会导致求左右孩子编号的 `left()` 和 `right()` 函数有所变化

直接插入排序

- 对当前元素,通过多次比较,将当前元素放入到合适的位置

计数排序

- 每个元素范围已知, 通过计算每种大小的元素个数, 进而通过计算找到元素应该放置的位置, 来进行排序
- 没有元素间的直接比较

归并排序

- 采用分治的思想,将数组不断从"中间"划分直到通过一次比较就可以得到结果的子问题,然后将这些层层合并得到原问题的解

堆排序

- 通过维护二叉堆,每次从二叉堆中取出当前最大的元素并重新维护二叉堆,最终得到结果

快速排序

- 和归并排序思想类似,快速排序每次选择最后的元素作为主元对数组进行划分

4. 实验结果与分析

以下结果time的单位均为微秒(由于取对数后单位不好表示故在这里统一说明)

用**microsecond**(微秒)作为单位， 下面简写为**ms**（之后才发现这个和毫秒有歧义）

结果截图（取多次运行的某次结果）

- 五个排序算法$n = 2^3$时排序结果
  - 插入排序

Algorithms > 148-pqz-PB18111793-project1 > ex1 > output > insert_sort > result_3.txt

```
4710
4742
5587
14641
17000
19382
24108
29001
```

- 计数排序

Algorithms > 148-pqz-PB18111793-project1 > ex1 > output > counting_sort > result_3.txt

```
4710
4742
5587
14641
17000
19382
24108
29001
```

- 归并排序

Algorithms > 148-pqz-PB18111793-project1 > ex1 > output > merge_sort > result_3.txt

```
4710
4742
5587
14641
17000
19382
24108
29001

```

- 堆排序

Algorithms > 148-pqz-PB18111793-project1 > ex1 > output > heap_sort > result_3.txt

```
4710
4742
5587
14641
17000
19382
24108
29001
```

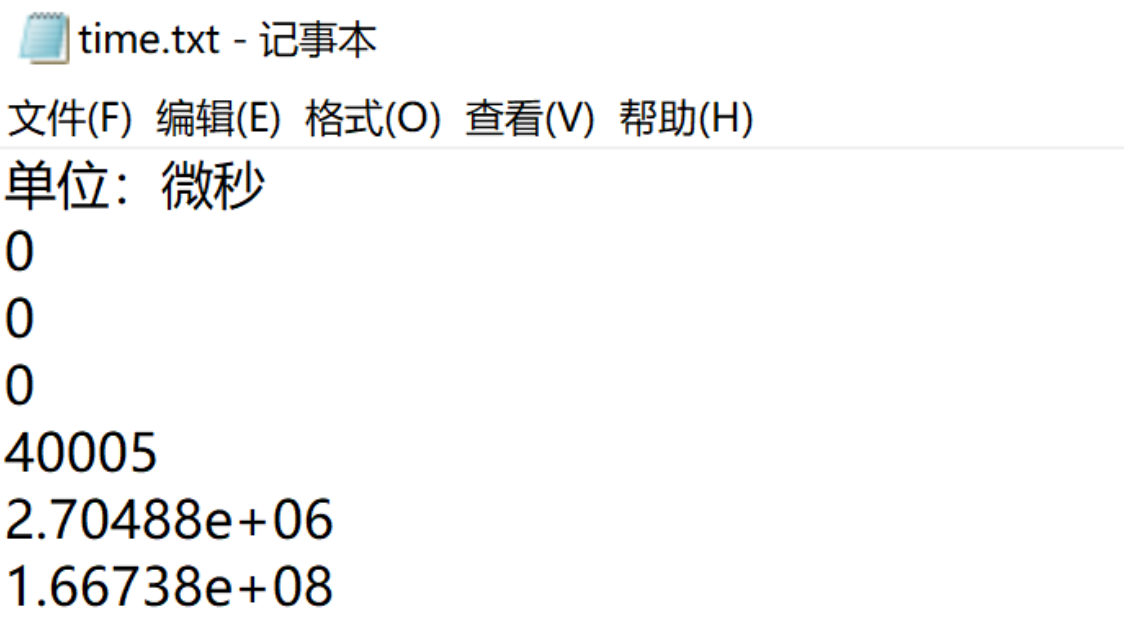
- 快速排序

Algorithms > 148-pqz-PB18111793-project1 > ex1 > output > quick_sort > result_3.txt

```
4710
4742
5587
14641
17000
19382
24108
29001

```

- 插入排序六个输入规模的运行时间



- 各排序算法时间复杂度如下表

算　法	最坏情况运行时间	平均情况/期望运行时间
插入排序	$\Theta(n^2)$	$\Theta(n^2)$
归并排序	$\Theta(n\lg n)$	$\Theta(n\lg n)$
堆排序	$O(n\lg n)$	—
快速排序	$\Theta(n^2)$	$\Theta(n\lg n)$(期望)
计数排序	$\Theta(k+n)$	$\Theta(k+n)$
基数排序	$\Theta(d(n+k))$	$\Theta(d(n+k))$
桶排序	$\Theta(n^2)$	$\Theta(n)$(平均情况)

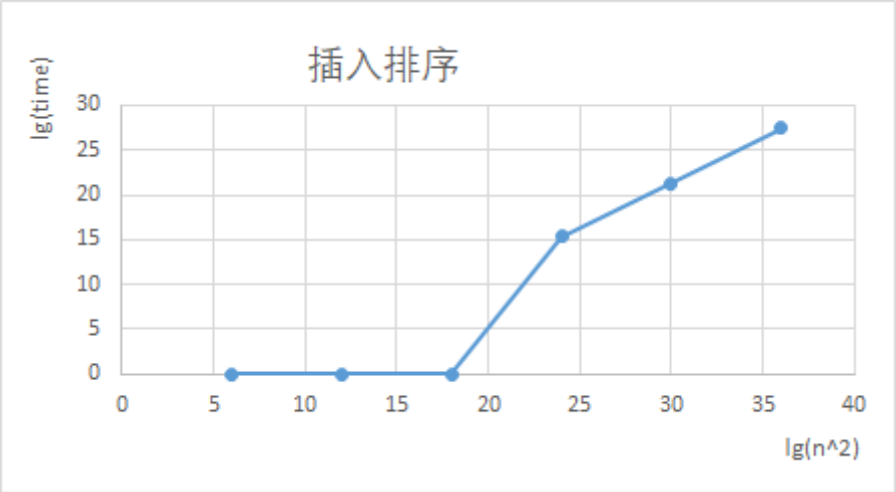
- 各排序结果分析如下(采用多次运行中具有代表性的一组作图)

输入规模n/时间(ms)	插入排序	计数排序	归并排序	堆排序	快速排序
2^3	0	0	0	0	0
2^6	0	0	0	0	0
2^9	0	1001	1000	0	0
2^(12)	41008	0	1999	2247	1958
2^(15)	2689210	2322	15016	25228	12840
2^(18)	188074000	9767	134027	292143	135654

  - 为了方便作图将横纵坐标取以2为底的对数
  - 对于为0的数据规定其对数为0.(即使我采用纳秒作为单位这些数据依然是0，故最后采用了微秒为单位（这里用ms表示)

插入排序

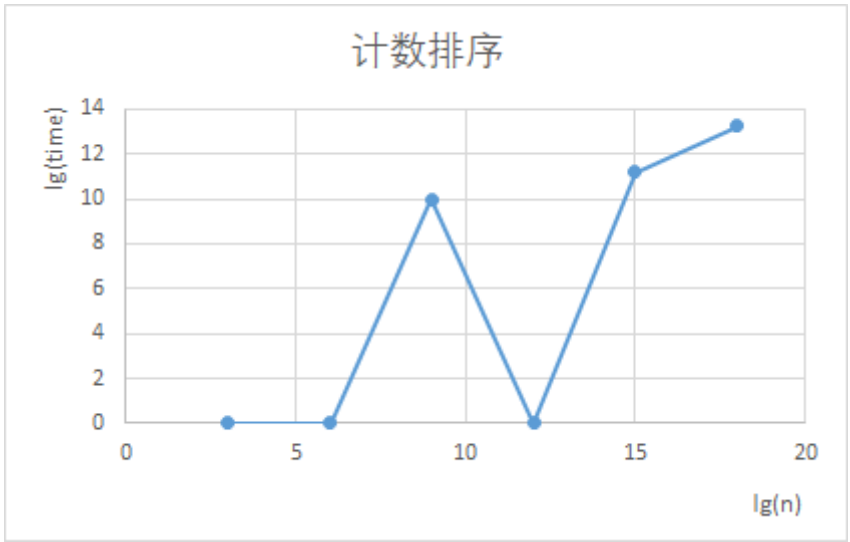
输入规模n	n^2	插入排序(time/ms)	log(n^2)	log(time)
8	64	0	6	0
64	4096	0	12	0
512	262144	0	18	0
4096	16777216	41008	24	15.32362
32768	1073741824	2689210	30	21.35875
262144	68719476736	188074000	36	27.48673



比较符合$\theta(n^2)$的预期

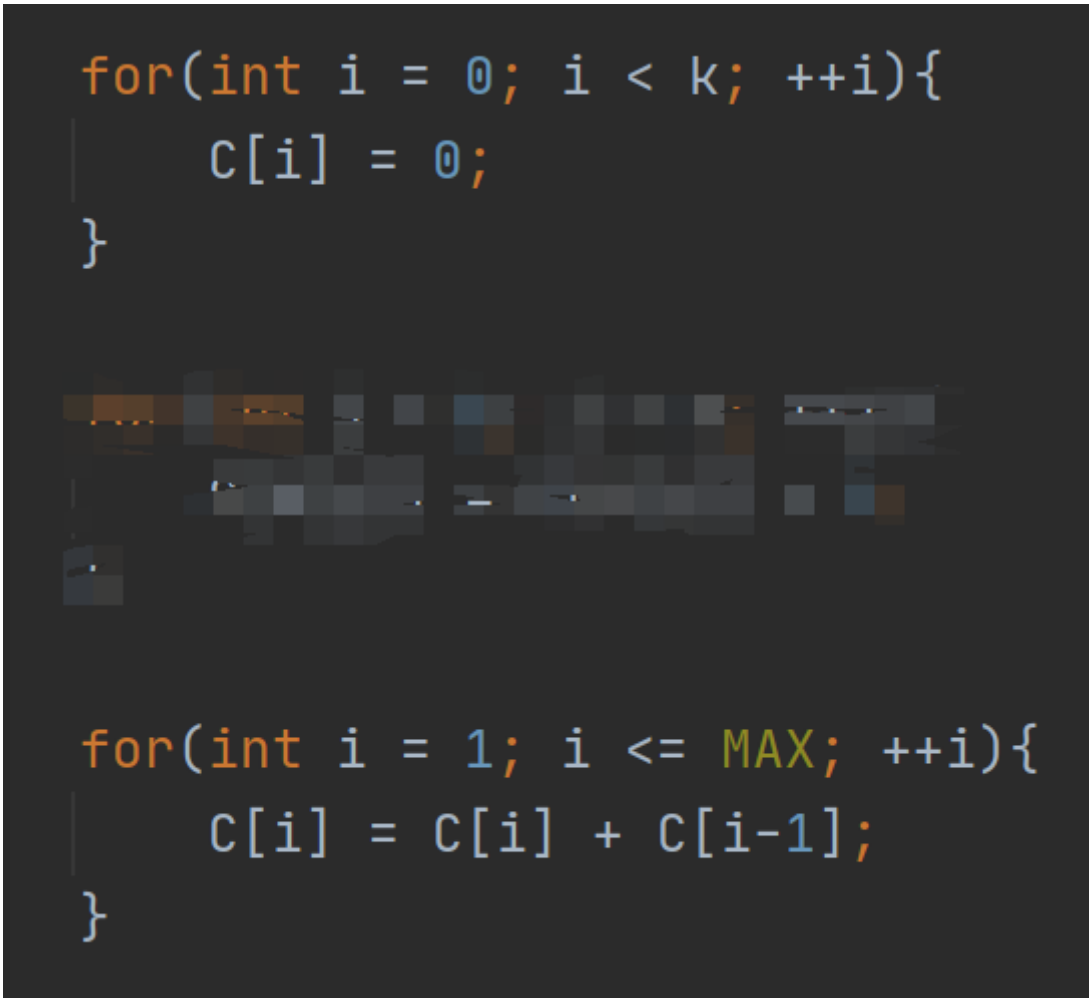
计数排序

输入规模n	计数排序(time/ms)	log(n)	log(time)
8	0	3	0
64	0	6	0
512	1001	9	9.967226
4096	0	12	0
32768	2322	15	11.18115
262144	9767	18	13.2537



如果不考虑第三个数据点,则较为符合$\theta(k+n)$的预期

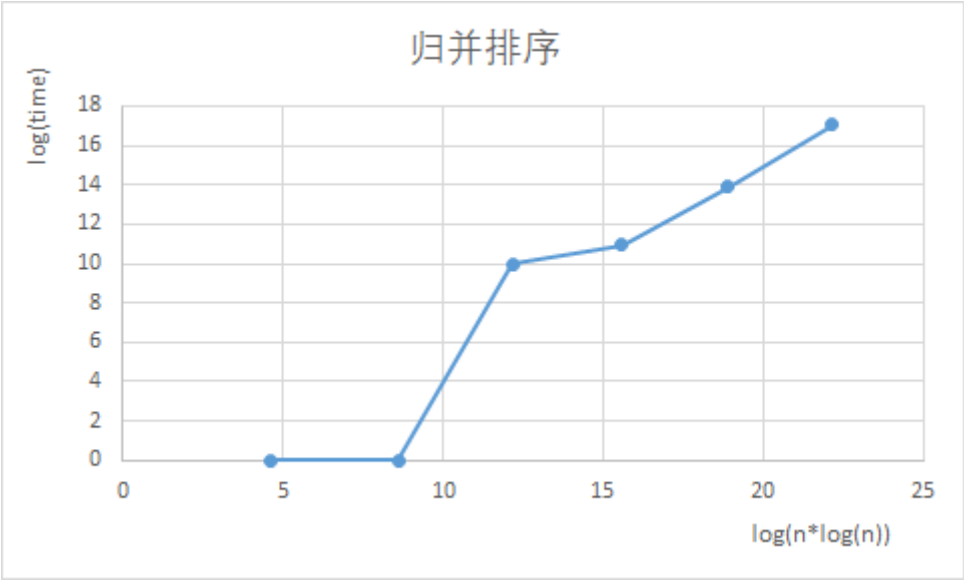
可以看到这里在第三个数据点产生了波动,在其他测试数据中也会出现类似的情况,从计数排序本身的操作来看,这很大程度上是因为无论要排序的规模有多大,数字的范围都是从0~32767,这导致



- 以上两个循环每次都要执行32768规模的次数,这对于排序规模较小时(如n=8,64,512)总体时间的影响是比较大的,远大于n次操作对总时间的影响. (即$\theta(k+n) \approx \theta(k)$)
- 本质原因很可能是机器本身的存储策略(主要是Cache策略),在时间波动段出现了大量的Cache Miss
- 当数据规模较大时,时间复杂度趋向于$\theta(n)$

归并排序

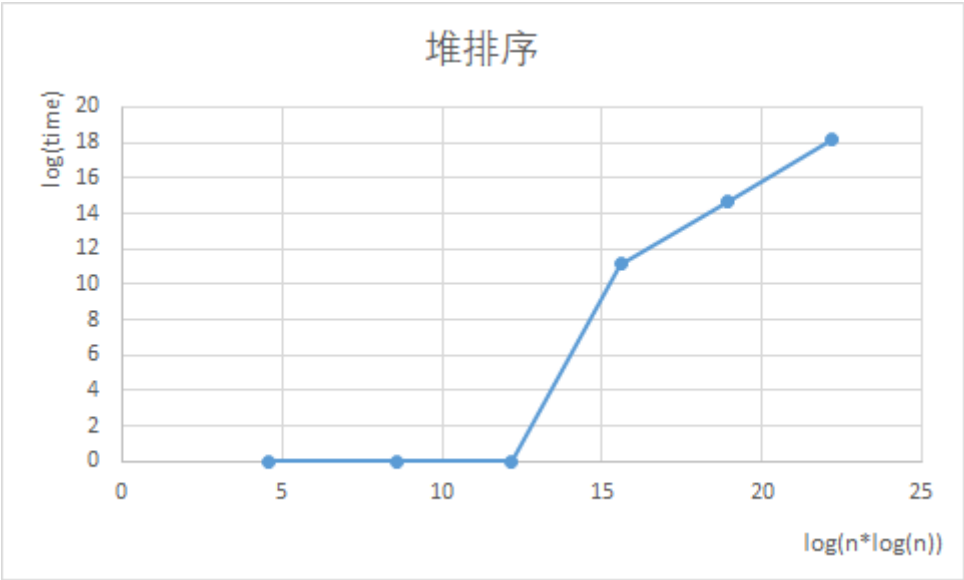
输入规模n	n*log(n)	归并排序(time/ms)	log(n*log(n))	log(time)
8	24	0	4.584962501	0
64	384	0	8.584962501	0
512	4608	1000	12.169925	9.965784
4096	49152	1999	15.5849625	10.96506
32768	491520	15016	18.9068906	13.87421
262144	4718592	134027	22.169925	17.03216



较为符合$\theta(nlgn)$的预期

堆排序

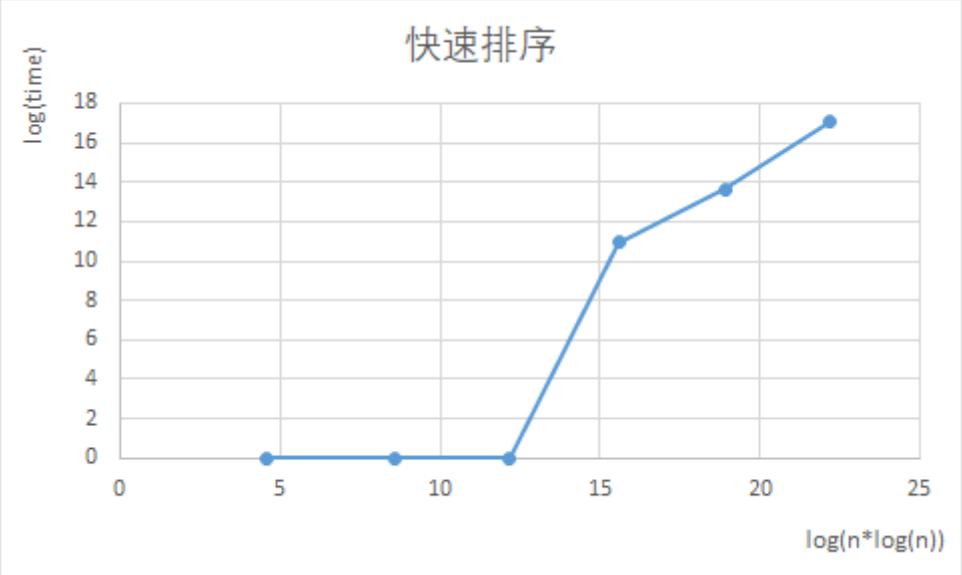
输入规模n	n*log(n)	堆排序(time/ms)	log(n*log(n))	log(time)
8	24	0	4.584962501	0
64	384	0	8.584962501	0
512	4608	0	12.169925	0
4096	49152	2247	15.5849625	11.13378
32768	491520	25228	18.9068906	14.62274
262144	4718592	292143	22.169925	18.15632



较为符合$O(nlgn)$的预期

快速排序

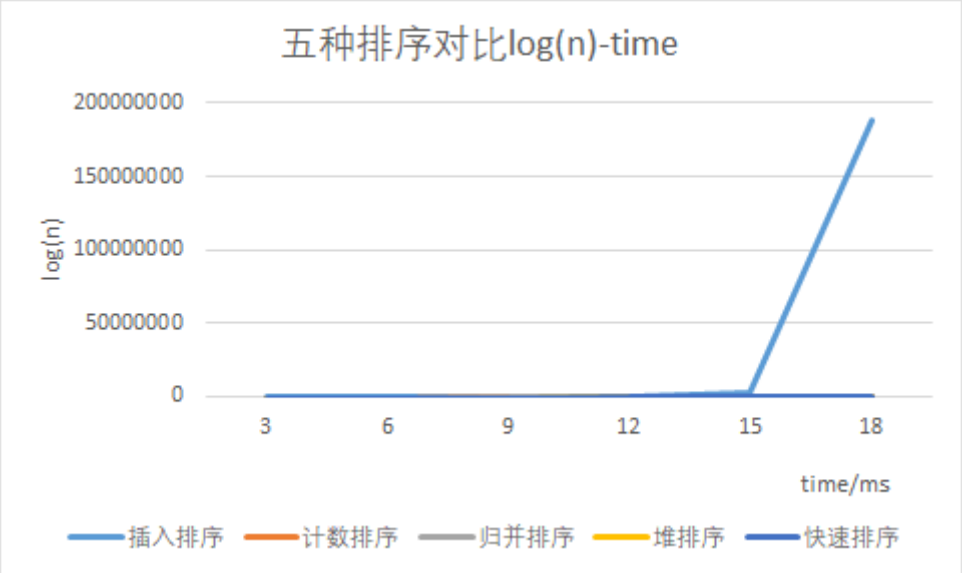
输入规模n	n*log(n)	快速排序(t	log(n*log(r	log(time)
8	24	0	4.584963	0
64	384	0	8.584963	0
512	4608	0	12.16993	0
4096	49152	1958	15.58496	10.93517
32768	491520	12840	18.90689	13.64836
262144	4718592	135654	22.16993	17.04957

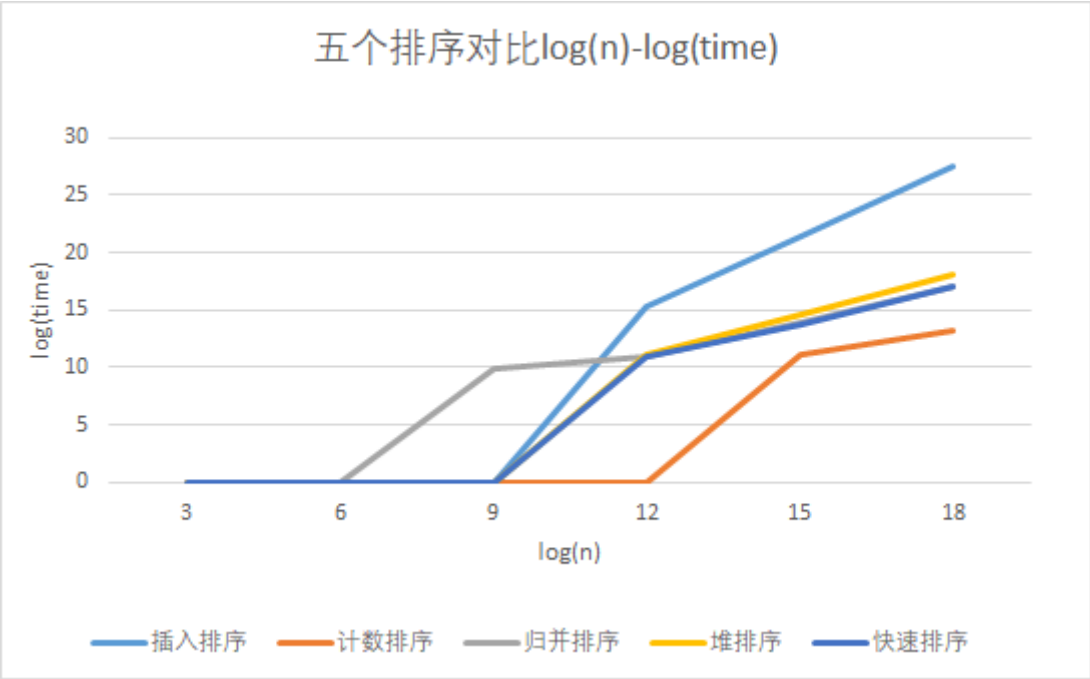


较为符合$\theta(nlgn)$的预期

综合对比

以下分析除去了计数排序不稳定的数据点





不同输入规模五个算法的比较(花费时间)

- 输入规模较小时(n在100的量级): 归并排序>插入排序>堆排序≈快速排序>计数排序
- 输入规模在1000附近:插入排序 > 归并排序>堆排序≈快速排序>计数排序
- 输入规模继续增大,大于4096附近:插入排序 > 堆排序>归并排序≈快速排序>计数排序

可以看到

- 计数排序时间复杂度最低,为$\theta(k+n)$,所需要的时间最少,但需要额外的存放结果的空间.且这种排序和其他四种排序有本质的不同,它没有元素间的直接比较,而是通过使用数组索引作为确定相对次序的工具. 也就是说如果可以事先确定数据的范围并且不在意空间, 计数排序是最优的选择
- 插入排序时间复杂度最高,为$\theta(n^2)$,所需要的时间最多. 由第一个图可以看出, 这种差距在数据规模较大时是巨大的. 但对于较小规模输入,由于插入排序的内层循环非常紧凑,插入排序可以很好地处理
- 归并排序、堆排序、快速排序时间复杂度均为$n(lgn)$量级, 在n较大时三者排序的时间基本相同. 且规模较大时归并排序和快速排序性能要好于堆排序. 但归并排序需要额外的存储空间,而快速排序是一种就地排序. 故在输入规模较大时采用快速排序比较合适

由于本人粗心刚开始将主函数中输出到time.txt文件的单位写成了毫秒（cout<< "单位：毫秒" << endl），实际上所有数据的单位都是微秒,故最后人工把这些“毫”字改成了“微”字，故output文件夹中各个time.txt文件的修改时间可能和其他文件不一致