lab1 排序算法 --实验报告

实验内容

- 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：$2^3, 2^6, 2^9, 2^{12}, 2^{15}, 2^{18}$
- 实现以下算法

    a. 直接插入排序
    b. 堆排序
    c. 快速排序
    d. 归并排序
    e. 计数排序

实验设备和环境

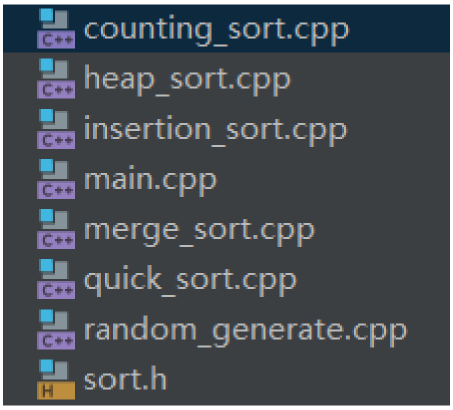
- 编译运行环境
    - Windows10-mingw-w64
    - Clion 2020.2.4

- 硬件
    - 处理器: 英特尔 Core i7-8750H @ 2.20GHz 六核
    - 速度 2.21 GHz (100 MHz x 22.0)
    - 处理器数量 核心数: 6 / 线程数: 12
    - 一级数据缓存 6 x 32 KB, 8-Way, 64 byte lines
    - 一级代码缓存 6 x 32 KB, 8-Way, 64 byte lines
    - 二级缓存 6 x 256 KB, 4-Way, 64 byte lines
    - 三级缓存 9 MB, 12-Way, 64 byte lines
    - 内存： 海力士 DDR4 2666MHz 8GB

实验方法和步骤

整体代码框架

- 由main.cpp来读取input.txt的数据，调用各个算法并完成output文件夹下的写入以及每次算法执行时间的测量。
- sort.h头文件中声明了每个算法的函数(均单独在一个文件里实现)，使得main.cpp能够调用这些算法
- random_generate.cpp主要随机生成input.txt
- 其余5个文件为5个算法的实现



代码设计思路

- 随机生成input.txt文件：

  考虑用stdlib.h下的rand()函数来生成随机数，不过为了更好的随机化，考虑使用当前时钟作为随机数种子。

  ```
  int n=262144;
  srand((unsigned)time(NULL));//随机数生成
  for(int i = 0; i < n;i++ ){
      write_file << rand() << endl;
  }
  ```

  在做实验时在main代码里会调用random_generate()函数来随机生成input.txt,为了与本实验报告数据一致，已经把此部分代码注释了。

- 文件读写
  - 考虑使用fsteam头文件下的ofstream以及ifstream流来轻松实现文件读写。
  - 需注意文件目录组织下，采用相对路径较为简单，个人因为使用Clion缘故，其运行时当前目录为cmake-build-debug,所以文件路径设置为

    `"..\\..\\input\\input.txt"`等
  - 此外，Clion不支持中文路径，需要注意
- 计时：采用网上推荐的格式，在经过试验后，发现微秒级的计时即可让结果显示较为方便。

  但是一开始选用的计时方式如下：

  ```
  auto start = system_clock::now();

  auto end   = system_clock::now();
  auto duration = duration_cast<microseconds>(end - start);
   time_count[time_cur++]=(double)(duration.count());
  ```

  但可能是因为其测量精度不够的原因，导致实验里测量的时间总是有很多time=0us的情况，这在数据规模n=3,6,9时尤为明显，而且数据极不稳定。

  所以后续检索后考虑采用<windows.h>头文件下一个us级的计时方式

```
//clock计时参数的声明
LARGE_INTEGER nFreq;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
double dt;

QueryPerformanceFrequency(&nFreq);
QueryPerformanceCounter(&t1);
quick_sort(data3_sorted,0,num_3-1);
QueryPerformanceCounter(&t2);
dt =(t2.QuadPart-t1.QuadPart)/(double)nFreq.QuadPart;
time_count[time_cur++]=dt*1000000;
```

实验结果发现此时计时精度较高，每个数据规模下均有较为合理的测量结果。

为方便计算，计时操作均在main.cpp里完成，对每一次算法调用做一次计时。在每一类(一类6个time)算法执行完毕后，统一通过file_print()函数将数据写入time.txt文件。

- 考虑在main函数里读入随机生成的input.txt文件，放入数组中，数组均有3,6,9,12,15,18的量级。

  因为排序算法里有原址排序，也有需要额外存储空间的算法，所以考虑开了两组数组，一组存放从input.txt读入的原始数据`datan[]`，一组用来做排序`datan_sorted`。这样，每次调用算法之前，把`datan[]`中数据拷贝到`datan_sorted`中，再传入相应算法的函数即可。

- 各个算法函数里完成算法实现

  - counting_sort():与书上所述类似，主要注意这里k的取值已定，以及数组下标从0开始
  - heap_sort():
    - 与书上不同的地方在于，维护的最大堆数组下标从0开始，所以PARENT,LEFT,RIGHT函数需要对应做改变
    - 考虑简化代码，未用结构体实现，所以直接把堆的length以及heap_size作为参数传递给需要的函数
  - insertion_sort():此算法最为简单，不做过多阐述。
  - merge_sort():
    - 因为已知数据范围，所以选取40000作为哨兵
    - 另外注意动态内存分配后一定要delete
  - quick_sort():

    较为简单，但自己犯了一个低级错误，导致输出的时间不对，但是排序结果正确，debug了一定时间：

    主要就是这种需要递归调用的算法实现，不能直接在递归调用函数里完成计时操作。所以考虑在原函数外面包装一层函数，如在`quick_sort()`中完成计时并调用`QUICK_SORT(A,p,r);`完成真正的排序。但自己改动代码的时候`void QUICK_SORT(int A[],int p,int r)`里递归调用写的是`quick_sort()`

    当然，在改进后的实现里，自己把计时又放回了main.cpp，所以也不用再考虑上述情况。

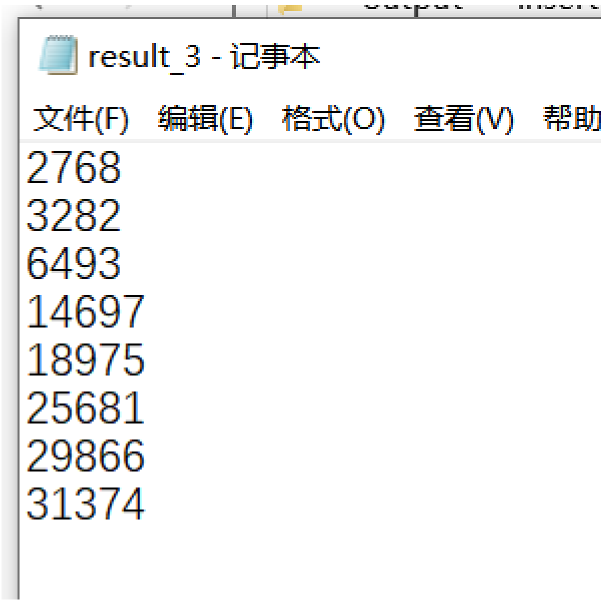
实验结果与分析

结果截图

- 五个排序算法$n = 2^3$时截图

  因为5个算法排序结果都是一样的，所以看到下面各个算法文件夹下的result_3.txt是一样的数据

  - 插入排序

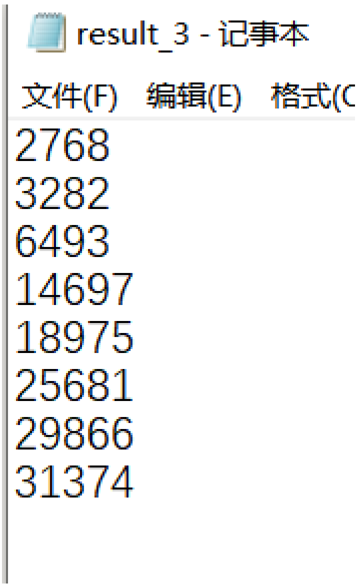


result_3 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助

2768
3282
6493
14697
18975
25681
29866
31374

  - 归并排序

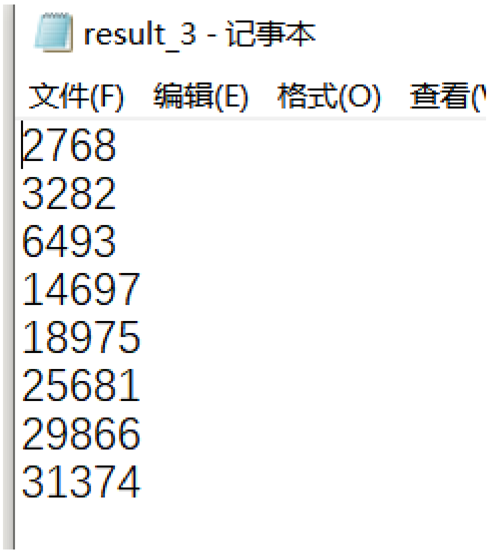


result_3 - 记事本

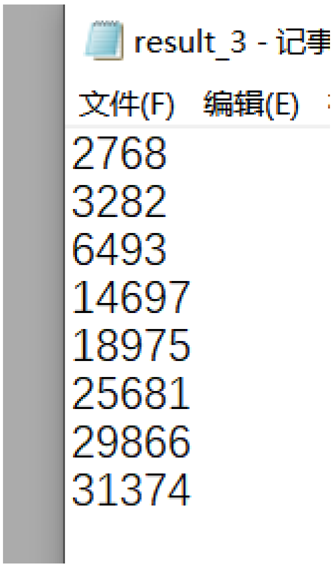
文件(F) 编辑(E) 格式(C

2768
3282
6493
14697
18975
25681
29866
31374

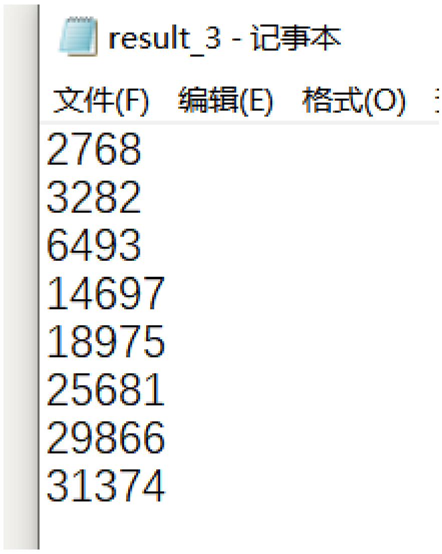
  - 堆排序



- 快速排序



- 计数排序



- 归并排序6个输入规模运行时间

time - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

单位:us
4.3
153.7
140.3
1215.7
9605.1
80373

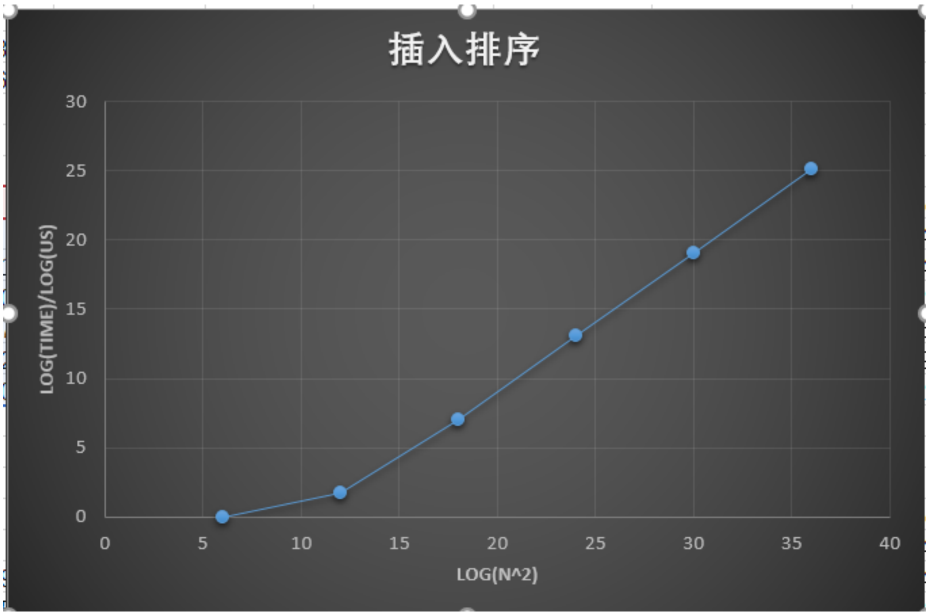
不同输入规模时记录的数据

注：

1. 以下图像里纵坐标中time单位均为us(微妙)
2. 诸如插入排序理论时间复杂度为$\Theta(n^2)$，为了减小数据取值大小，便于作图直观，可以将横纵坐标同时取对数后再作图，效果不变
3. 以下的对数均以2为底
4. 因为数据规模n=3时由于时间过小，有0.5us这样的数据，在取对数时为复数，为了图像直观，把这些数据取值改为0(这样的数据点也比较少)

- 插入排序

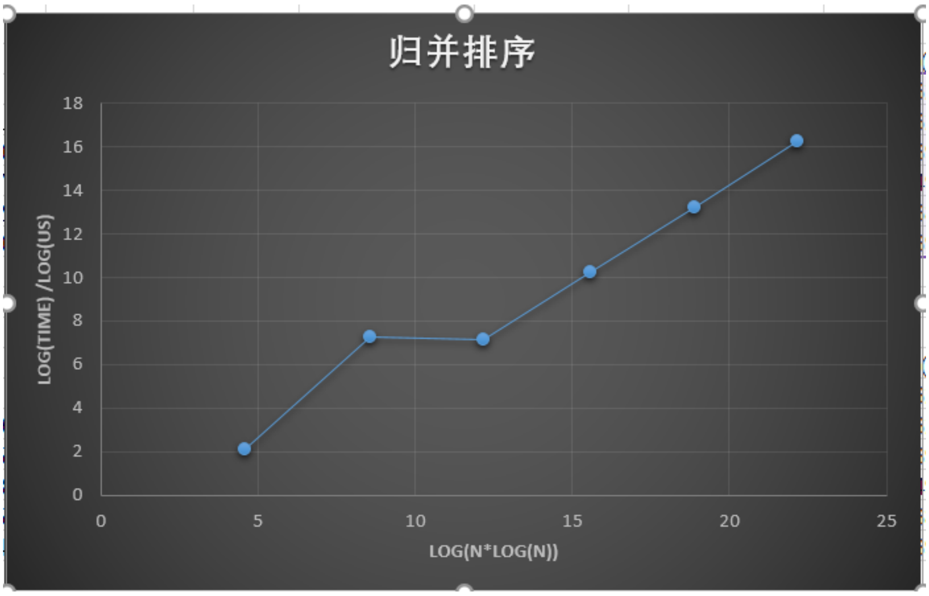
插入排序				
数据规模n	n^2	time(us)	log(n^2)	log(time)
8	64	0.5	6	0
64	4096	3.2	12	1.678071905
512	262144	133.2	18	7.057450272
4096	16777216	8785.3	24	13.10087584
32768	1073741824	560061	30	19.09522444
262144	68719476736	37685400	36	25.16750237



由图像可知较为符合$\Theta(n^2)$

- 归并排序

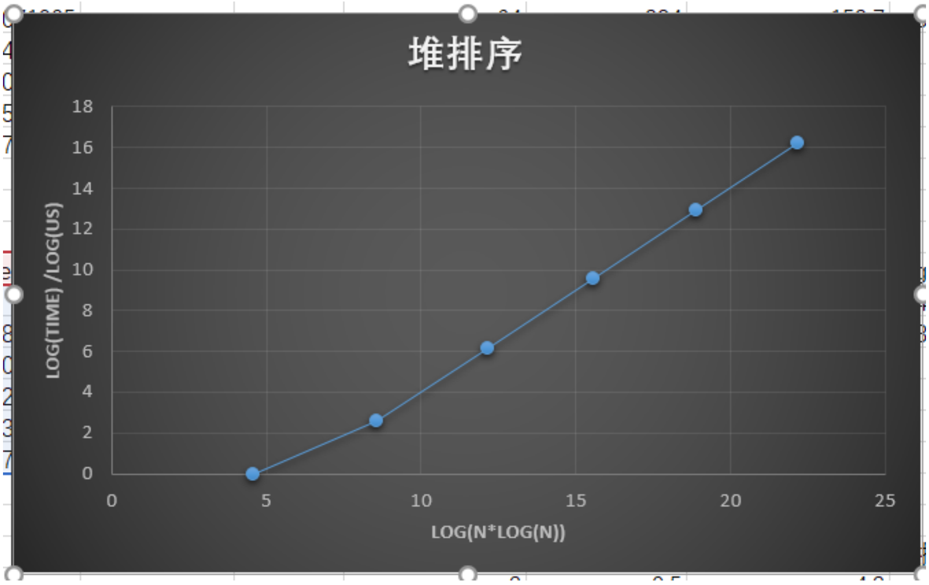
归并排序				
数据规模n	n*log(n)	time(us)	log(n*log(n))	log(time)
8	24	4.3	4.584962501	2.10433666
64	384	153.7	8.584962501	7.263973355
512	4608	140.3	12.169925	7.132371199
4096	49152	1215.7	15.5849625	10.24757154
32768	491520	9605.1	18.9068906	13.22958492
262144	4718592	80373	22.169925	16.29442331



可知较为符合$\Theta(nlog(n))$

- 堆排序

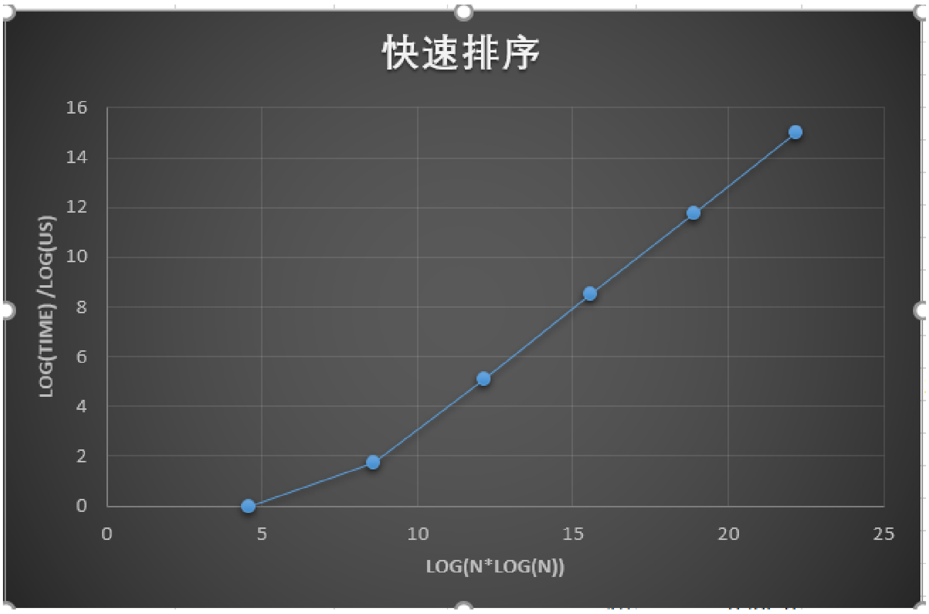
堆排序				
数据规模n	n*log(n)	time(us)	log(n*log(n))	log(time)
8	24	1	4.584962501	0
64	384	6.1	8.584962501	2.608809243
512	4608	69.6	12.169925	6.121015401
4096	49152	748.7	15.5849625	9.548243944
32768	491520	7610.9	18.9068906	12.89385135
262144	4718592	76212.5	22.169925	16.21774002



可知较为符合$\Theta(nlog(n))$

- 快速排序

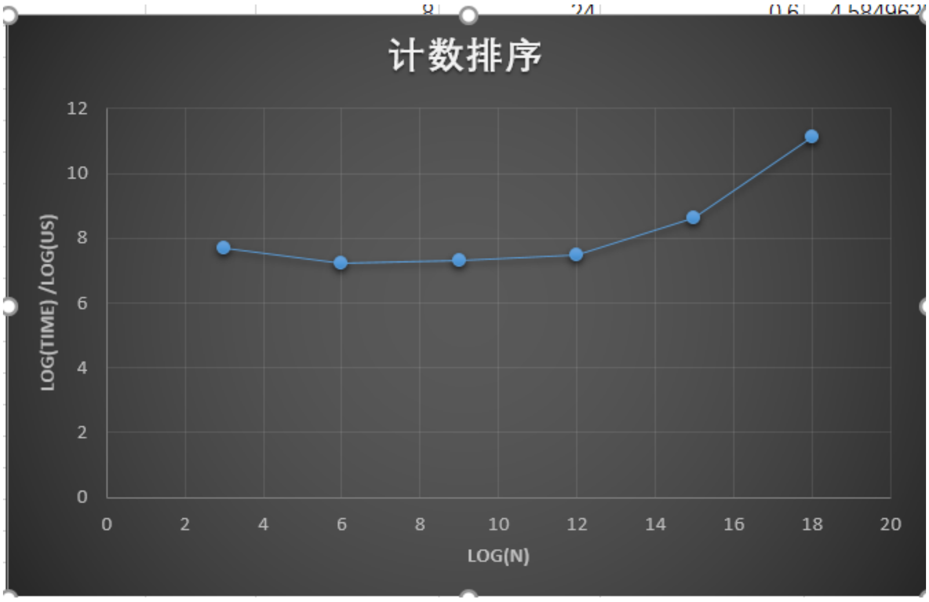
快速排序				
数据规模n	n*log(n)	time(us)	log(n*log(n))	log(time)
8	24	0.6	4.584962501	0
64	384	3.4	8.584962501	1.765534746
512	4608	33.6	12.169925	5.070389328
4096	49152	366	15.5849625	8.515699838
32768	491520	3397.6	18.9068906	11.7303003
262144	4718592	32641.8	22.169925	14.994433



可知较为符合$\Theta(nlog(n))$

- 计数排序

计数排序			
数据规模n	time(us)	log(n)	log(time)
8	207.2	3	7.694880193
64	151.7	6	7.245077275
512	160.1	9	7.322829498
4096	177.5	12	7.471675214
32768	389.9	15	8.606960345
262144	2225.1	18	11.11965446



由图像发现计数排序的结果与期待的$\Theta(k+n)$性能有一定偏差，

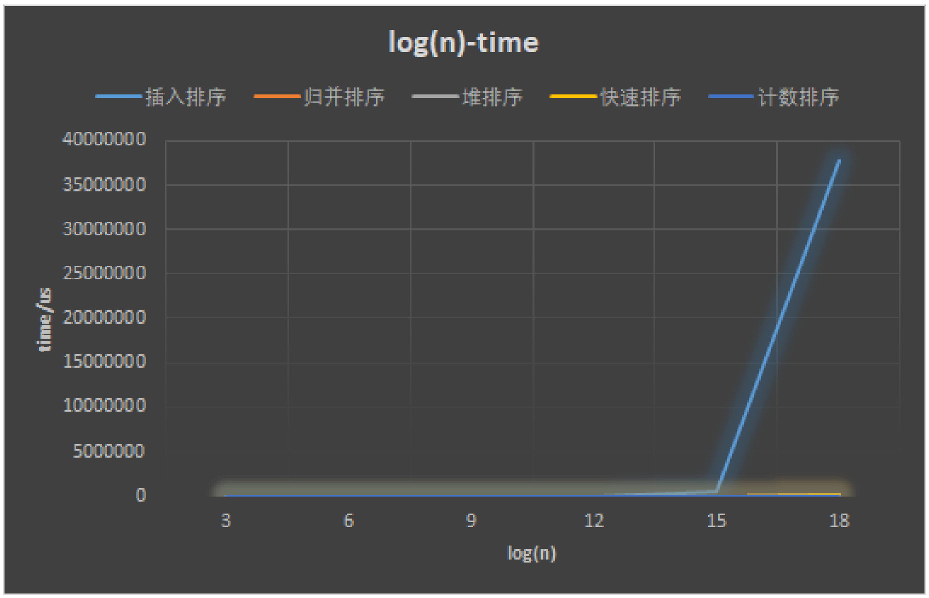
分析原因如下：

- 分析计数排序代码：

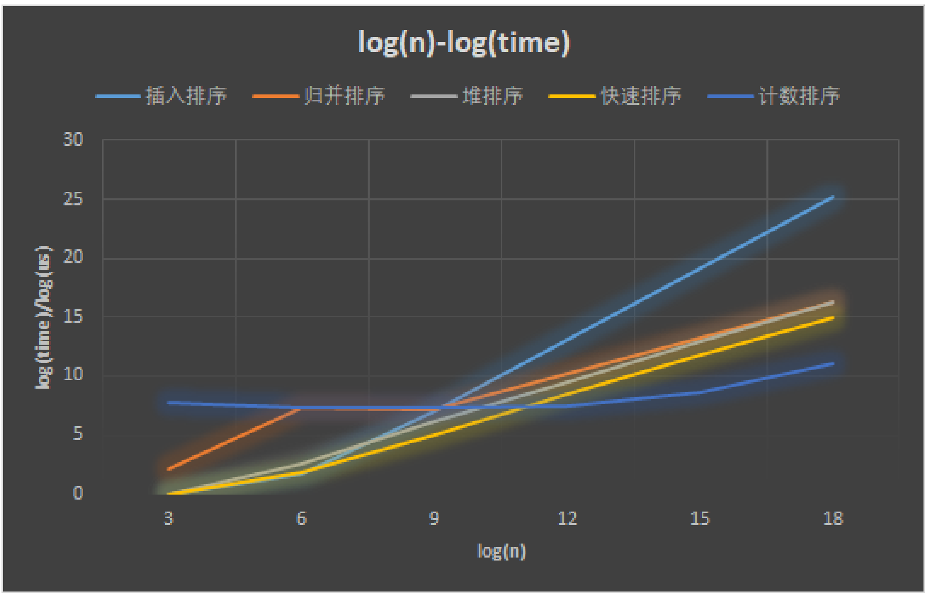
```
for(i=0;i<=k;i++){
        C[i]=0;
}
for(j=0;j<length;j++){
    C[A[j]]=C[A[j]]+1;
}
for(i=1;i<=k;i++){
    C[i]=C[i]+C[i-1];
}
for(j=length-1;j>=0;j--){
    B[C[A[j]]-1]=A[j];//B数组下标从0开始
    C[A[j]]=C[A[j]]-1;
}
```

- 当n较小时，k的取值(实验里k=32767)占据了主导因素，算法的主要时间花在对C[i]这个数组的赋值、计算上，而k的取值一定，所以n较小时(除了$2^{15}, 2^{18}$量级以外的数据)运行时间相对稳定在180us, 此时$\Theta(n+k) \approx \Theta(k)$
- 在数据规模较小时(n=3,6,9,12),时间测量存在一定波动，n=3时的时间反而更久一些，一方面是因为k占据时间的主导，本来这些规模的数据执行时间就相近。更进一步，可能是由于硬件客观存在的cache环境以及cache策略的影响，导致在运行时会有cache miss次数上的区别，使得时间波动较大
- 注意到当$n=2^{15}, 2^{18}$时，可明显发现用时上升，这也证明了此时n占据了算法耗时的主导。

不同排序算法的时间对比

- 

-



- 由图像可以看到
  - n较小时(n规模在3,6时)，运行时间：

    计数排序>归并排序>堆排序≈快速排序≈插入排序

    数据规模过小，计数排序用时最大，是因为其运行时间受固定的k的取值影响。此数据规模下选择快速排序或者插入排序有较好性能。
  - n规模在9-12时，运行时间：

    插入排序>归并排序>堆排序≈快速排序≈计数排序

    在这个输入规模下，选择堆排序、快速排序更优，因为考虑到计数排序或者归并排序还需额外的空间，但效率并未有显著区别。
  - n在2^{12}量级往后，运行时间：

    插入排序>归并排序≈堆排序>快速排序>计数排序

    - 当然，可以看到，此时曲线明显分为了3个梯队，最大是的插入排序，中间梯队是归并、堆、快速排序，最小的是计数排序，这也与理论上的时间复杂度分析一致，所以在此输入规模下，若无空间上的限制，优先选择计数排序(当然，计数排序还有其最大的限制：数据取值范围已知并且取值为整数)。
    - 由图像也可以看到，快速排序的性能会稍稍优于堆排序和归并排序，这是因为快速排序能更好的利用硬件特性，有效减少Cache miss，这也是其被广泛使用的原因所在。所以在输入规模较大时，在$\Theta(nlog(n))$的算法中，优先选择快速排序。

实验总结

- 通过本次实验，我对课上所学的各个算法有了更深刻的认识，除了亲手实现算法的具体代码以外， 也对各个算法遇到不同数据规模时的性能表现有了一定的认识，对算法的选择有了一定的判断方法。

  比如计数排序$\Theta(n+k)$的理论时间复杂度，课上学时还是一知半解，只有真正测得了数据后，看到具体的图像并分析后才能明白其中的缘由。
- 同时，也是对excel绘图做了一定熟悉，这对于以后写实验报告也是有很大帮助的。

参考文档

- http://blog.sina.com.cn/s/blog_79ab4be10100uzrj.html
- http://c.biancheng.net/view/299.html
- https://www.jianshu.com/p/a70ca5dc80d3