Project1排序实验报告

徐煜森 PB16110173

1. **实验要求**

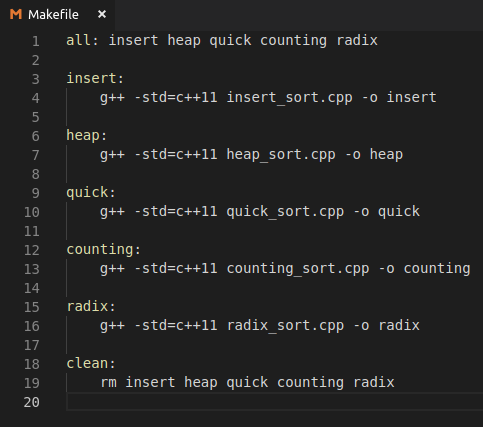
使用插入排序，堆排序，快速排序，基数排序，计数排序五种排序算法，排序n个元素，元素为随机生成的1到65535之间的整数，n的取值为： 2^2，2^5，2^8，2^11，2^14，2^17。

1. **实验环境**

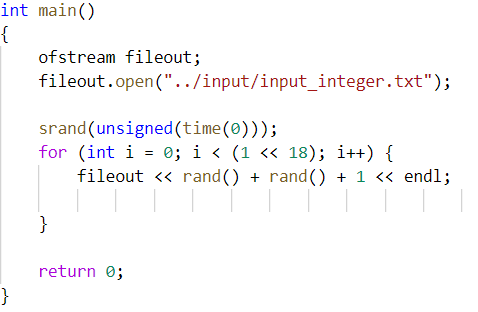
本实验在虚拟机中编译运行。

1. 宿主机：Windows10 64位x86，机器内存8G，时钟主频2.59GHz
2. 虚拟机VMware Workstation Pro：Ubuntu 16.04 LTS 64位，虚拟机内存2G
3. 编译环境：Ubuntu 16.04 LTS，G++ 5.4.0（编译选项-std=C++11）

1. **实验过程**
2. **Makefile和生成数据**
3. 为了方便进行实验，同时也方便助教进行测评，我写了一个简单的Makefile用于编译我的代码。可以直接执行相应的make命令来进行编译。如make insert可以对insert\_sort.cpp进行编译。另外，因为程序中使用C++11中的库函数，所以需要加上编译选项-std=C++11。



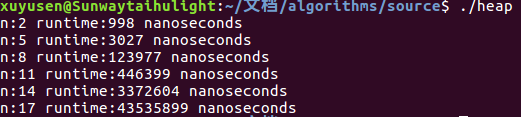
1. 生成数据使用stdlib.h中的rand()函数，函数生成[0, 32767]区间的随机数，使用 rand() + rand() + 1 来生成[1, 65535]区间的随机数。注意：不同环境下rand()函数的返回值可能不同。



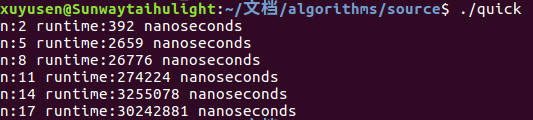
1. **插入排序**
2. 插入排序算法较为简单，我实现的方法与课本《算法导论》基本一致。
3. 作为本次实验实现的第一个排序算法，这个程序的主要任务是为后续几个算法的测试实现一个可以重复使用的框架。因此在之后编写测试程序时，只需要实现相应的排序算法即可。
4. 使用C++11中的chrono测量时间，测量结果在Linux下可以精确到纳秒级。实验中不考虑硬盘IO读取数组时间，从内存中已读入未排序数组开始计时，到排序结束后停止计时。
5. 以下为打印出的测试结果，排序结果和运行时间也按格式要求输出至相应位置。

****

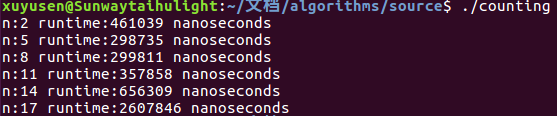
1. **堆排序**
2. 我使用了不同的方法实现堆排序，在Max\_Heapify函数中使用循环来替代递归，原因是递归可能增大时间开销，使用循环可以避免频繁的调用函数，从而节省函数间上下文切换的时间。
3. 数组从0开始，所以需要重新考虑父节点、左右子节点的计算方法。
4. 以下为打印出的测试结果。



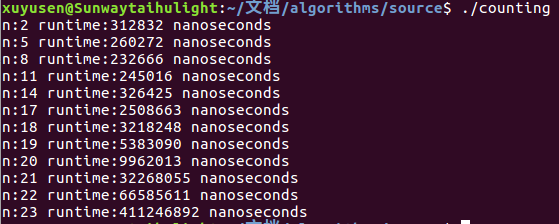
1. **快速排序**
2. 按照课本上的方法实现快排算法。
3. 以下为打印出的测试结果



1. **计数排序**
2. 按照课本上的方法实现计数排序，为基数排序提供稳定的线性时间的排序算法。
3. 值得注意的点是数组从0开始时，把结果写入前需要对count数组相应减1。因为count[i]中存的是小于等于i的数的个数，减1后对应的才是各个数的在数组中的位置。
4. 以下为打印出的测试结果。

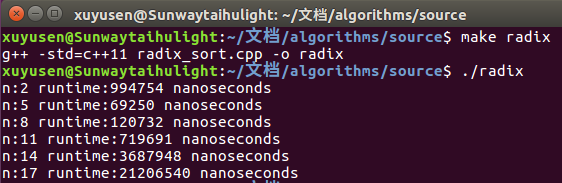


1. 实验过程中发现，运行时间在n较小时没有像期望那样与n成线性关系，如n为2,5,6时运行时间会产生波动，而在11之后逐渐稳定为线性。故增加测试数据至2^23，增测6个点18,19,20,21,22,23的数据。测试结果如下



注：因实验提交要求，为不影响实验代码评测，打包提交仍为测试前6个点的代码。不过增测代码仅改动了数组的大小。

1. **基数排序**
2. 以16为基，共需循环4次即可完成排序。从低位到高位使用计数排序对数组排序。
3. 值得注意的是count数组的大小从计数排序中的65536变为了16。另外，由于计数排序不是就地排序，所以每一遍排序后需要把结果复制给原数组。
4. 以下为打印出的测试结果。

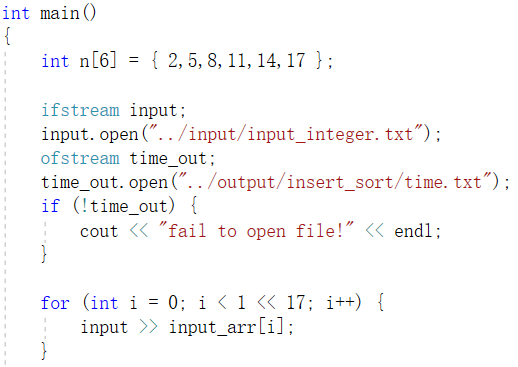


1. **使用Excel处理实验数据，并画图分析。**
2. **实验关键代码**
3. **整体测试框架**

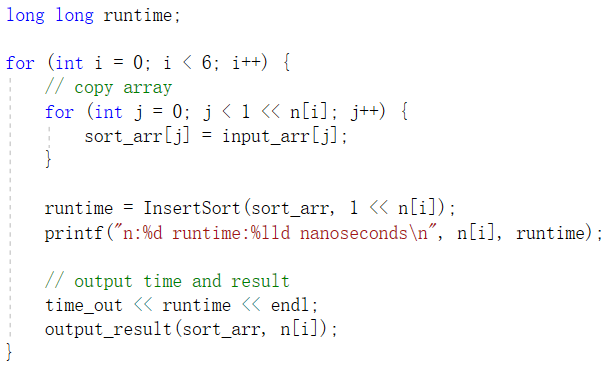
全局变量：存放输入的数组，待排序的数组和输出结果的路径前缀。其中大数组定义为全局变量可以防止线程栈溢出。



Main函数中设置好需要用到的n的数组，输入输出路径，并读入数据。

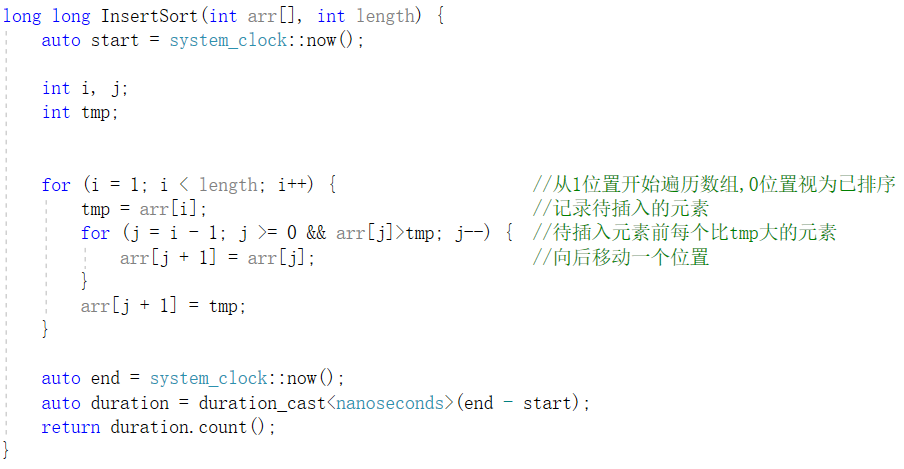


进入main函数主体，六次循环调用的排序函数（图中以插入排序为例）对不同规模的数据进行排序并返回运行时间，测试出n为6个不同值时的运行时间及排序结果，将运行时间和排序结果按要求输出至文件。



1. **插入排序**

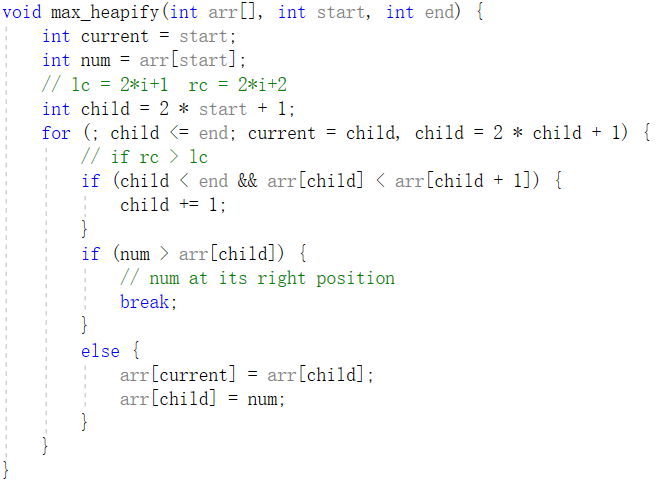
使用chrono中的system\_clock进行计时。插入排序的具体实现如图。



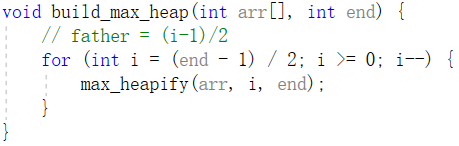
1. **堆排序**

数组从0开始时，节点i的左儿子为2\*i+1，右儿子为2\*i+2，父节点为（i-1）/2

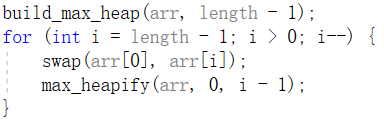
Max\_Heapify函数为元素arr[start]向下寻找其应在的位置，从而将堆重新调整为大根堆。end为堆的末尾。



Build\_Max\_Heap函数将一个乱序的数组构建为大根堆，通过不断调用max\_heapify函数自底向上建立。



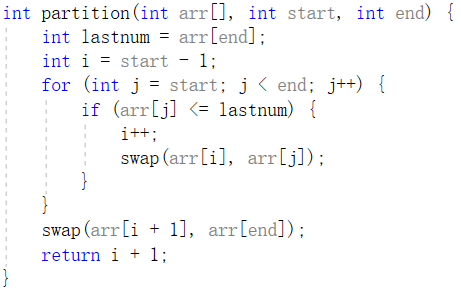
堆排序算法主体即为构建大根堆，之后不断的将第一个元素交换至堆尾，然后将堆再次调整为大根堆。



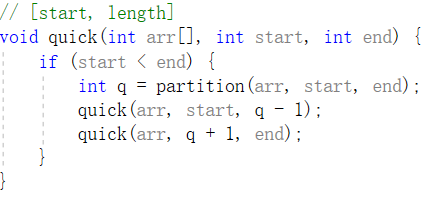
1. **快速排序**

快速排序实现与课本方法一致，start和end分别为当前区间的第一个元素和最后一个元素的索引。

划分函数：

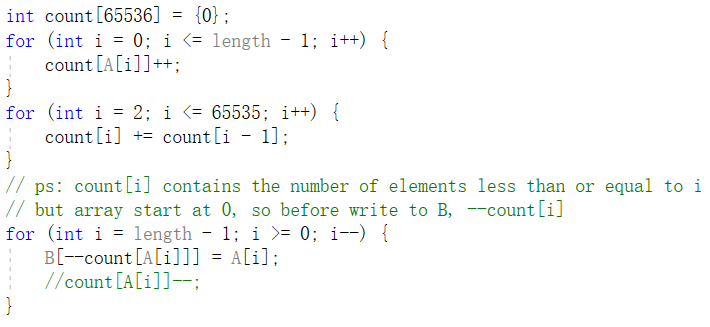


快排算法主体：



1. **计数排序**

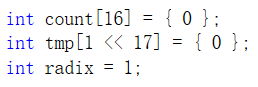
计数排序实现中值得注意的是数组从0开始，在将排序结果写回B数组前count首先需减1。其他与课本一致。

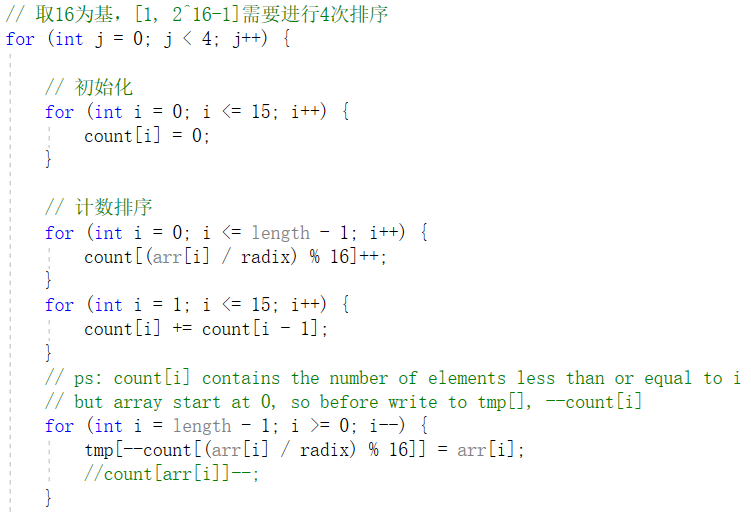


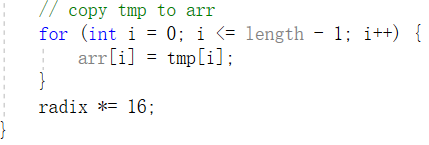
1. **基数排序**

基数排序中选取基为16，相当于4bit为一位，对最大值为65535（2^16-1）的数组，排序完成需要四次循环。

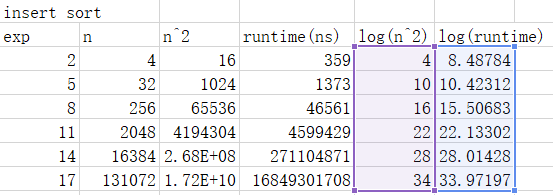
count数组的大小变为16，radix的值初始化为1，每次循环结束后自乘16。



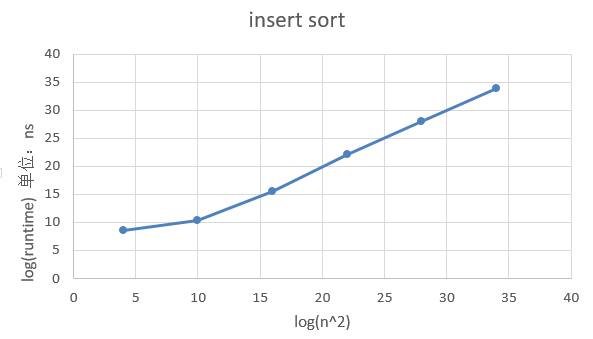




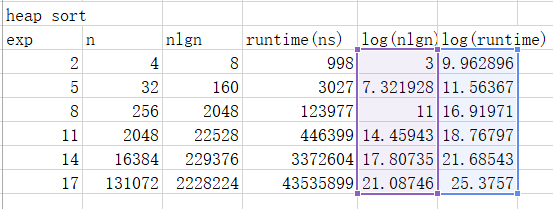
1. **实验结果及分析**
2. **插入排序**
3. 实验数据如下图，可以看出6个点之间数据相差较大，如果直接以n或n^2为横轴，以运行时间为纵轴画图会导致前几个点集中在图的左下角，画出的图只能看到最后三个点的变化趋势，无法观测整体。因此，对n^2和运行时间以2为底取对数（下图中框出部分），最终画图期望得到一条斜率为1的直线。



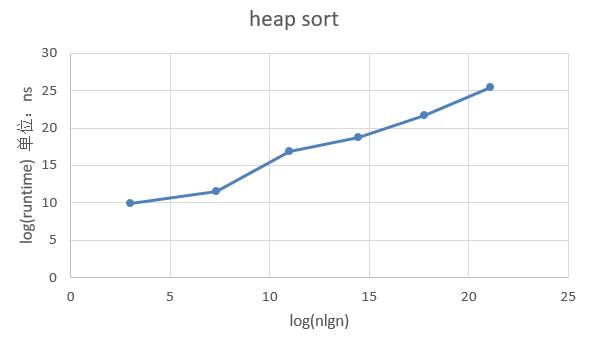
1. 将数据画图可以看出，测试运行时间符合。



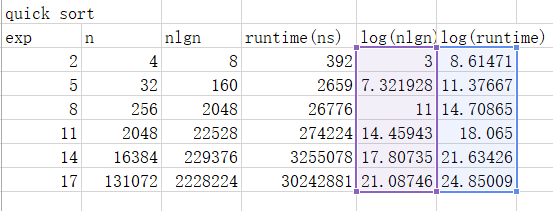
1. **堆排序**
2. 实验数据



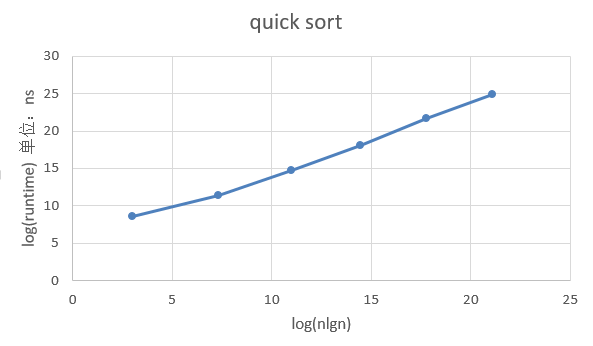
1. 同样使用上图中框出部分画图，可以看出运行时间测试结果符合



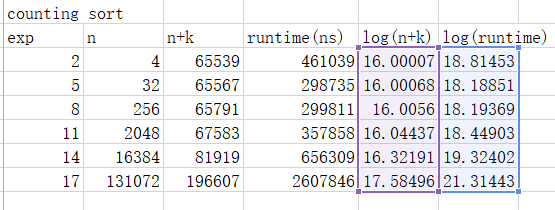
1. **快速排序**
2. 实验数据



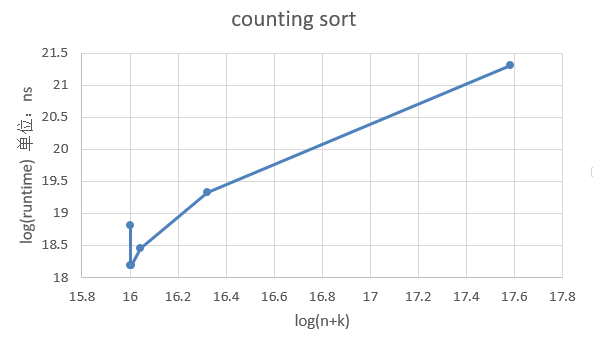
1. 画图后可以看出运行时间测试结果符合，快速排序得到的实验结果是五种里面最好的。



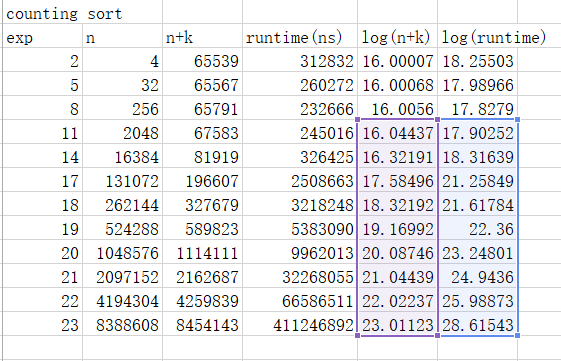
1. **计数排序**
2. 实验数据

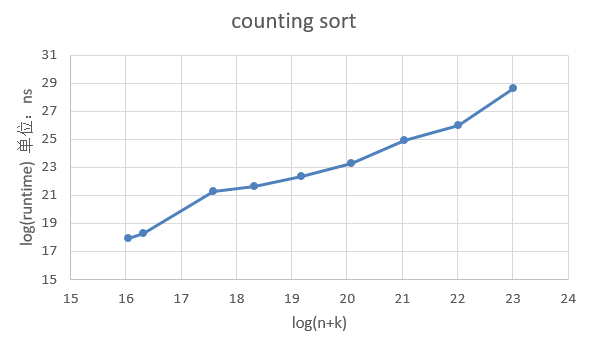


1. 画图后看出，在n为2,5,8时运行时间产生了与期望不符合的波动，而在n大于等于11后逐渐恢复线性。故增测了6个点，分别是n为18,19,20,21,22,23。



1. 增测数据

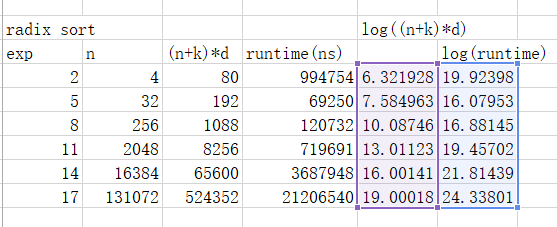


取n大于等于11的部分画图，可以发现其稳定后运行时间测试结果符合 

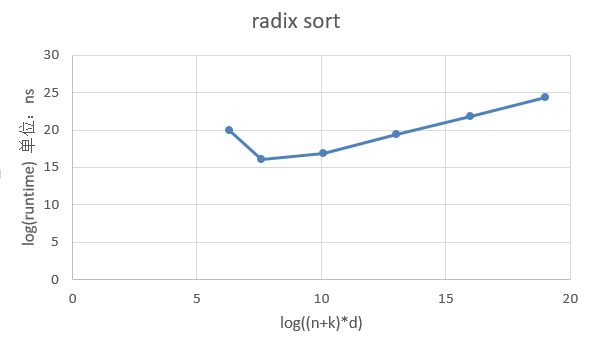
分析原因可能是：

* + 1. n为2,5,8时与k值（65535）相差过大，此时对count数组（大小为65535）的自增和求和成为主要时间开销，导致运行时间反常波动。
    2. 硬件cache存在，而程序运行前期有大量的cache miss，从内存把数据调入cache也成为不可忽视的时间开销。程序运行至后期，cache hit的概率增加，再加上排序所需时间增加，导致数据调入cache内的时间几乎可以忽略，也可能使运行时间反常波动。

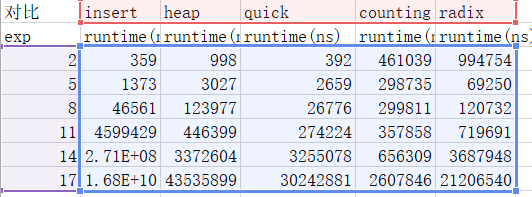
1. **基数排序**
2. 实验数据



1. 画图发现运行时间测试结果基本符合，只有n等于2时产生了异常，分析原因可能同样与cache有关，另外导致这个的原因应该与k取值关系不大，因为基数排序中k的取值为16，没有远超过待排序数据量。

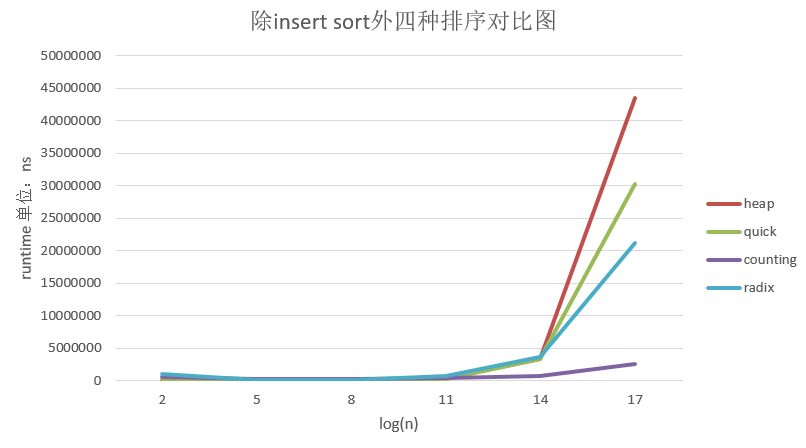


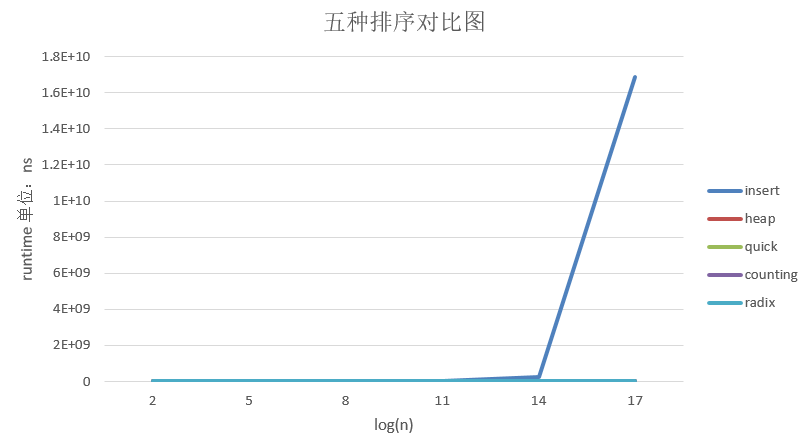
1. **对比**
2. 从实验数据中可以看出数据量较小时，插入排序、堆排序、快速排序的表现非常好。n=2时这三个算法的时间都在1000ns以下，在n小于等于8时插入排序、快速排序的表现远好于另外三种排序。
3. 当n逐渐增大至14,17时，计数排序和基数排序的线性时间复杂度优势逐渐体现出来，此时要优于其他三种排序。另外计数排序的时间要小于基数排序，原因可能是运行时间的常数比较小，基数排序的常数的作用明显大于计数排序。



以下为除插入排序外四种排序的对比图和加上插入排序五种排序的对比图，可以看出插入排序在排序量大时的运行时间，远超其他四种排序，导致图中其他排序的曲线几乎成为水平线。

注：图中log(n)即为2,5,8,11,14,17。





1. **实验总结**
2. Windows一个线程默认栈大小为1MB，如果在函数中定义了过多大数组就会导致栈溢出。我在main函数中定义了两个1<<17大小的int数组，导致报出栈溢出的错误。解决方法：将这两个数组定义为全局数组。
3. Ofstream的open方法只能自己创建文件，不会创建路径，实际上C++没有任何库有已封装好的方法用于创建路径，路径不存在时导致一直没有输出。解决方法：创建相应路径。
4. 在写计数排序时定义了三个2^17大小的int全局数组，会报出越界错误，程序无法正常退出。解决方案：按书上方法写计数排序，不使用临时大数组。之后在写基数排序时，发现临时大数组是可以使用的，不过在定义时需要初始化来分配存储空间。
5. 经常写一些基础的算法题是有必要的，这次实验中我用了不到十分钟写出来的快排，调试用了一个小时。