

Project1 排序实验报告

徐煜森 PB16110173

一. 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} 。

二. 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)

三. 实验过程

0. Makefile 和生成数据

- i. 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。

```
Makefile x
1  all: insert heap quick counting radix
2
3  insert:
4      g++ -std=c++11 insert_sort.cpp -o insert
5
6  heap:
7      g++ -std=c++11 heap_sort.cpp -o heap
8
9  quick:
10     g++ -std=c++11 quick_sort.cpp -o quick
11
12  counting:
13     g++ -std=c++11 counting_sort.cpp -o counting
14
15  radix:
16     g++ -std=c++11 radix_sort.cpp -o radix
17
18  clean:
19     rm insert heap quick counting radix
20
```

- ii. 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。

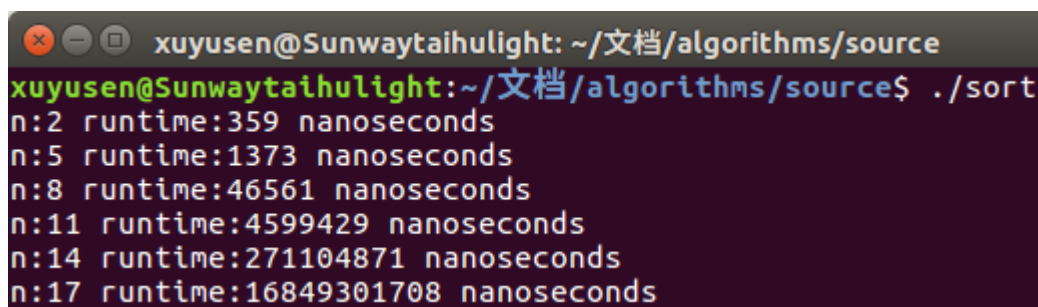
```
int main()
{
    ofstream fileout;
    fileout.open("../input/input_integer.txt");

    srand(unsigned(time(0)));
    for (int i = 0; i < (1 << 18); i++) {
        fileout << rand() + rand() + 1 << endl;
    }

    return 0;
}
```

1. 插入排序

- i. 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。
- ii. 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。
- iii. 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。
- iv. 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。



```
xuyusen@Sunwaytaihulight: ~/文档/algorithms/source
xuyusen@Sunwaytaihulight:~/文档/algorithms/source$ ./sort
n:2 runtime:359 nanoseconds
n:5 runtime:1373 nanoseconds
n:8 runtime:46561 nanoseconds
n:11 runtime:4599429 nanoseconds
n:14 runtime:271104871 nanoseconds
n:17 runtime:16849301708 nanoseconds
```

2. 堆排序

- i. 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。

- ii. 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。
- iii. 以下为打印出的测试结果。

```
xuyusen@Sunwaytaihlighthouse:~/文档/algorithms/source$ ./heap
n:2 runtime:998 nanoseconds
n:5 runtime:3027 nanoseconds
n:8 runtime:123977 nanoseconds
n:11 runtime:446399 nanoseconds
n:14 runtime:3372604 nanoseconds
n:17 runtime:43535899 nanoseconds
```

3. 快速排序

- i. 按照课本上的方法实现快排算法。
- ii. 以下为打印出的测试结果

```
xuyusen@Sunwaytaihlighthouse:~/文档/algorithms/source$ ./quick
n:2 runtime:392 nanoseconds
n:5 runtime:2659 nanoseconds
n:8 runtime:26776 nanoseconds
n:11 runtime:274224 nanoseconds
n:14 runtime:3255078 nanoseconds
n:17 runtime:30242881 nanoseconds
```

4. 计数排序

- i. 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。
- ii. 值得注意的是数组从 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。
- iii. 以下为打印出的测试结果。

```
xuyusen@Sunwaytaihlighthouse:~/文档/algorithms/source$ ./counting
n:2 runtime:461039 nanoseconds
n:5 runtime:298735 nanoseconds
n:8 runtime:299811 nanoseconds
n:11 runtime:357858 nanoseconds
n:14 runtime:656309 nanoseconds
n:17 runtime:2607846 nanoseconds
```

- iv. 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

```
xuyusen@Sunwaytaihligh:~/文档/algorithms/source$ ./counting
n:2 runtime:312832 nanoseconds
n:5 runtime:260272 nanoseconds
n:8 runtime:232666 nanoseconds
n:11 runtime:245016 nanoseconds
n:14 runtime:326425 nanoseconds
n:17 runtime:2508663 nanoseconds
n:18 runtime:3218248 nanoseconds
n:19 runtime:5383090 nanoseconds
n:20 runtime:9962013 nanoseconds
n:21 runtime:32268055 nanoseconds
n:22 runtime:66585611 nanoseconds
n:23 runtime:411246892 nanoseconds
```

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。

5. 基数排序

- i. 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。
- ii. 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。
- iii. 以下为打印出的测试结果。

```
xuyusen@Sunwaytaihlight: ~/文档/algorithms/source
xuyusen@Sunwaytaihlight:~/文档/algorithms/source$ make radix
g++ -std=c++11 radix_sort.cpp -o radix
xuyusen@Sunwaytaihlight:~/文档/algorithms/source$ ./radix
n:2 runtime:994754 nanoseconds
n:5 runtime:69250 nanoseconds
n:8 runtime:120732 nanoseconds
n:11 runtime:719691 nanoseconds
n:14 runtime:3687948 nanoseconds
n:17 runtime:21206540 nanoseconds
```

6. 使用 Excel 处理实验数据，并画图分析。

四. 实验关键代码

1. 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。

```
int input_arr[1 << 17], sort_arr[1 << 17];
string result_out_path = "../output/insert_sort/result_";
```

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。

```

int main()
{
    int n[6] = { 2, 5, 8, 11, 14, 17 };

    ifstream input;
    input.open("../input/input_integer.txt");
    ofstream time_out;
    time_out.open("../output/insert_sort/time.txt");
    if (!time_out) {
        cout << "fail to open file!" << endl;
    }

    for (int i = 0; i < 1 << 17; i++) {
        input >> input_arr[i];
    }
}

```

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。

```

long long runtime;

for (int i = 0; i < 6; i++) {
    // copy array
    for (int j = 0; j < 1 << n[i]; j++) {
        sort_arr[j] = input_arr[j];
    }

    runtime = InsertSort(sort_arr, 1 << n[i]);
    printf("n:%d runtime:%lld nanoseconds\n", n[i], runtime);

    // output time and result
    time_out << runtime << endl;
    output_result(sort_arr, n[i]);
}

```

2. 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。

```
long long InsertSort(int arr[], int length) {
    auto start = system_clock::now();

    int i, j;
    int tmp;

    for (i = 1; i < length; i++) {
        tmp = arr[i];
        for (j = i - 1; j >= 0 && arr[j] > tmp; j--) {
            arr[j + 1] = arr[j];
        }
        arr[j + 1] = tmp;
    }

    auto end = system_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    return duration.count();
}
```

3. 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。


```

void max_heapify(int arr[], int start, int end) {
    int current = start;
    int num = arr[start];
    // lc = 2*i+1  rc = 2*i+2
    int child = 2 * start + 1;
    for (; child <= end; current = child, child = 2 * child + 1) {
        // if rc > lc
        if (child < end && arr[child] < arr[child + 1]) {
            child += 1;
        }
        if (num > arr[child]) {
            // num at its right position
            break;
        }
        else {
            arr[current] = arr[child];
            arr[child] = num;
        }
    }
}

```

`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。

```

void build_max_heap(int arr[], int end) {
    // father = (i-1)/2
    for (int i = (end - 1) / 2; i >= 0; i--) {
        max_heapify(arr, i, end);
    }
}

```

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。

```

build_max_heap(arr, length - 1);
for (int i = length - 1; i > 0; i--) {
    swap(arr[0], arr[i]);
    max_heapify(arr, 0, i - 1);
}

```

4. 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：

```
int partition(int arr[], int start, int end) {
    int lastnum = arr[end];
    int i = start - 1;
    for (int j = start; j < end; j++) {
        if (arr[j] <= lastnum) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[end]);
    return i + 1;
}
```

快排算法主体：

```
// [start, length]
void quick(int arr[], int start, int end) {
    if (start < end) {
        int q = partition(arr, start, end);
        quick(arr, start, q - 1);
        quick(arr, q + 1, end);
    }
}
```

5. 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。

```

int count[65536] = {0};
for (int i = 0; i <= length - 1; i++) {
    count[A[i]]++;
}
for (int i = 2; i <= 65535; i++) {
    count[i] += count[i - 1];
}
// ps: count[i] contains the number of elements less than or equal to i
// but array start at 0, so before write to B, --count[i]
for (int i = length - 1; i >= 0; i--) {
    B[--count[A[i]]] = A[i];
    //count[A[i]]--;
}

```

6. 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。

```

int count[16] = { 0 };
int tmp[1 << 17] = { 0 };
int radix = 1;

```

```

// 取16为基, [1, 2^16-1]需要进行4次排序
for (int j = 0; j < 4; j++) {

    // 初始化
    for (int i = 0; i <= 15; i++) {
        count[i] = 0;
    }

    // 计数排序
    for (int i = 0; i <= length - 1; i++) {
        count[(arr[i] / radix) % 16]++;
    }
    for (int i = 1; i <= 15; i++) {
        count[i] += count[i - 1];
    }
    // ps: count[i] contains the number of elements less than or equal to i
    // but array start at 0, so before write to tmp[], --count[i]
    for (int i = length - 1; i >= 0; i--) {
        tmp[--count[(arr[i] / radix) % 16]] = arr[i];
        //count[arr[i]]--;
    }

    // copy tmp to arr
    for (int i = 0; i <= length - 1; i++) {
        arr[i] = tmp[i];
    }
    radix *= 16;
}

```

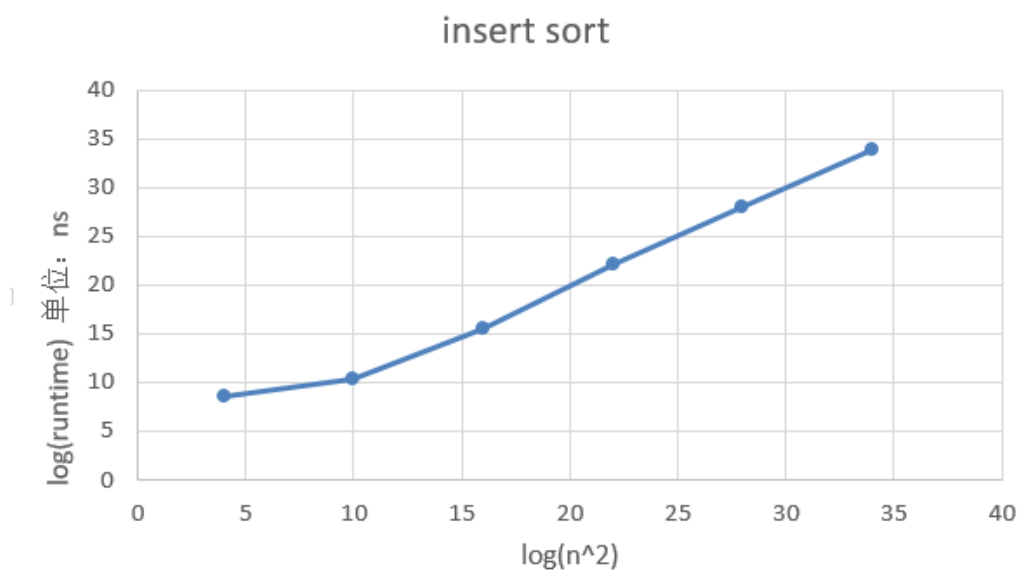
五. 实验结果及分析

1. 插入排序

- i. 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。

insert sort					
exp	n	n ²	runtime(ns)	log(n ²)	log(runtime)
2	4	16	359	4	8.48784
5	32	1024	1373	10	10.42312
8	256	65536	46561	16	15.50683
11	2048	4194304	4599429	22	22.13302
14	16384	2.68E+08	271104871	28	28.01428
17	131072	1.72E+10	16849301708	34	33.97197

ii. 将数据画图可以看出，测试运行时间符合 $\Theta(n^2)$ 。



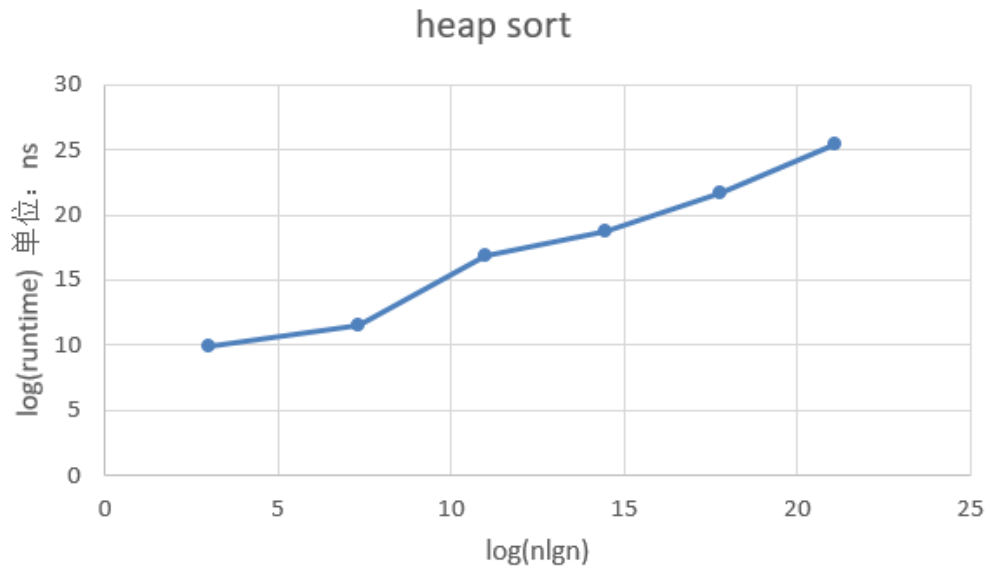
2. 堆排序

i. 实验数据

heap sort					
exp	n	n lgn	runtime(ns)	log(n lgn)	log(runtime)
2	4	8	998	3	9.962896
5	32	160	3027	7.321928	11.56367
8	256	2048	123977	11	16.91971
11	2048	22528	446399	14.45943	18.76797
14	16384	229376	3372604	17.80735	21.68543
17	131072	2228224	43535899	21.08746	25.3757

- ii. 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 $\Theta(n \lg n)$

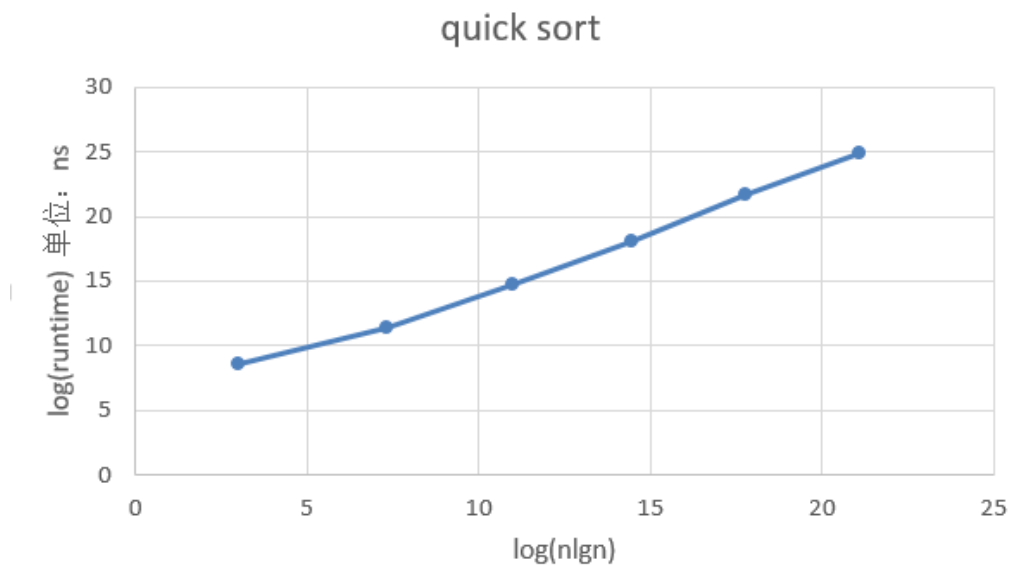


3. 快速排序

- i. 实验数据

quick sort					
exp	n	nlgn	runtime(ns)	log(nlgn)	log(runtime)
2	4	8	392	3	8.61471
5	32	160	2659	7.321928	11.37667
8	256	2048	26776	11	14.70865
11	2048	22528	274224	14.45943	18.065
14	16384	229376	3255078	17.80735	21.63426
17	131072	2228224	30242881	21.08746	24.85009

- ii. 画图后可以看出运行时间测试结果符合 $\Theta(n \lg n)$, 快速排序得到的实验结果是五种里面最好的。

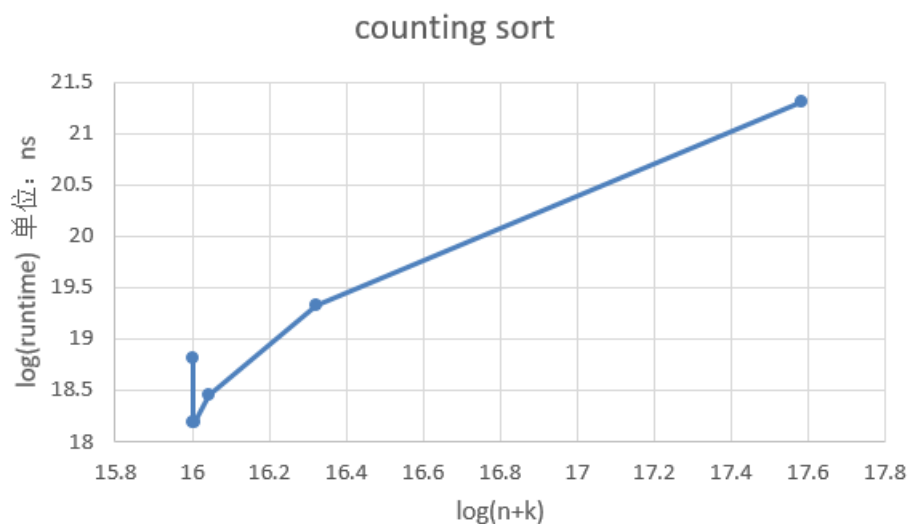


4. 计数排序

i. 实验数据

counting sort					
exp	n	n+k	runtime(ns)	$\log(n+k)$	$\log(\text{runtime})$
2	4	65539	461039	16.00007	18.81453
5	32	65567	298735	16.00068	18.18851
8	256	65791	299811	16.0056	18.19369
11	2048	67583	357858	16.04437	18.44903
14	16384	81919	656309	16.32191	19.32402
17	131072	196607	2607846	17.58496	21.31443

- ii. 画图后看出，在 n 为 2,5,8 时运行时间产生了与期望 $\Theta(k + n)$ 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。

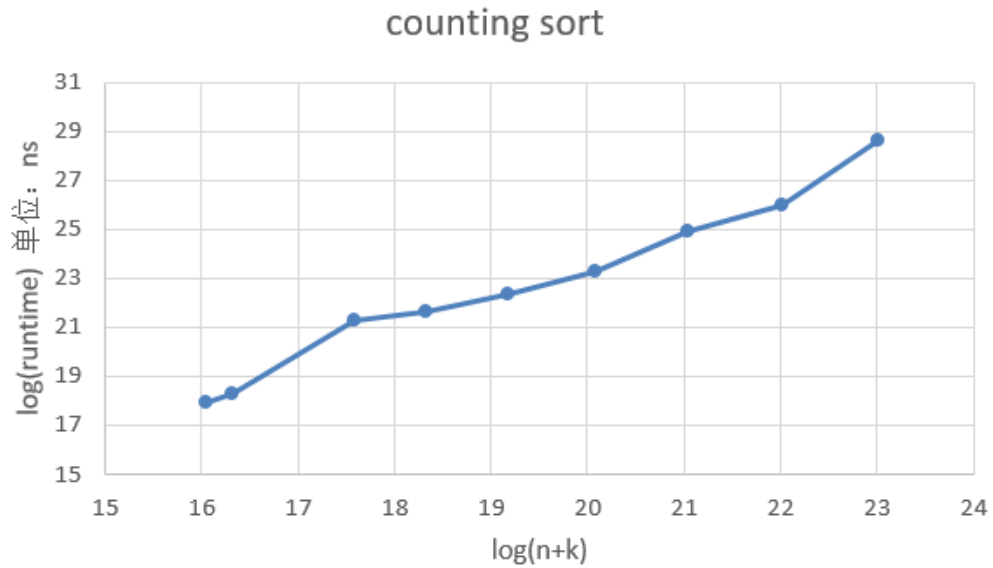


iii. 增测数据

counting sort					
exp	n	n+k	runtime(ns)	log (n+k)	log(runtime)
2	4	65539	312832	16.00007	18.25503
5	32	65567	260272	16.00068	17.98966
8	256	65791	232666	16.0056	17.8279
11	2048	67583	245016	16.04437	17.90252
14	16384	81919	326425	16.32191	18.31639
17	131072	196607	2508663	17.58496	21.25849
18	262144	327679	3218248	18.32192	21.61784
19	524288	589823	5383090	19.16992	22.36
20	1048576	1114111	9962013	20.08746	23.24801
21	2097152	2162687	32268055	21.04439	24.9436
22	4194304	4259839	66586511	22.02237	25.98873
23	8388608	8454143	411246892	23.01123	28.61543

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 $\Theta(k + n)$



分析原因可能是：

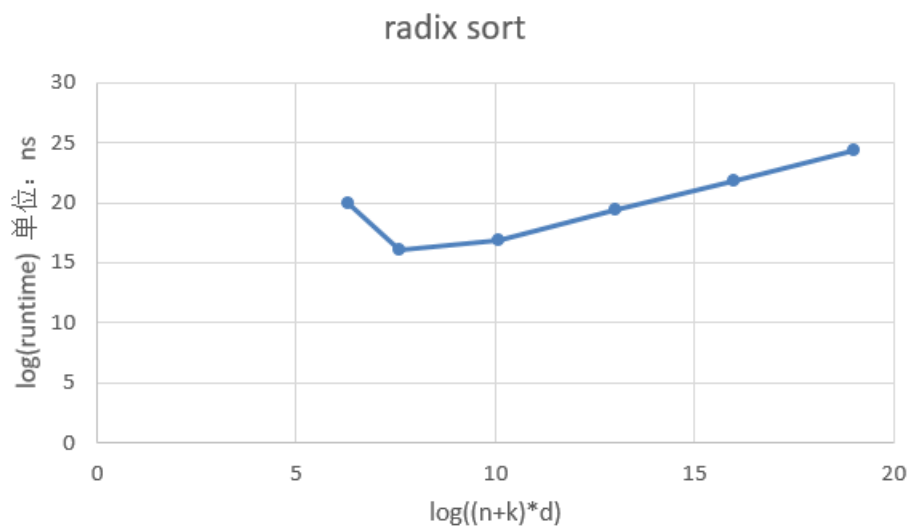
- a. n 为 2,5,8 时与 k 值 (65535) 相差过大, 此时对 `count` 数组 (大小为 65535) 的自增和求和成为主要时间开销, 导致运行时间反常波动。
- b. 硬件 `cache` 存在, 而程序运行前期有大量的 `cache miss`, 从内存把数据调入 `cache` 也成为不可忽视的时间开销。程序运行至后期, `cache hit` 的概率增加, 再加上排序所需时间增加, 导致数据调入 `cache` 内的时间几乎可以忽略, 也可能使运行时间反常波动。

5. 基数排序

i. 实验数据

radix sort				log((n+k)*d)	
exp	n	(n+k)*d	runtime(ns)		log(runtime)
2	4	80	994754	6.321928	19.92398
5	32	192	69250	7.584963	16.07953
8	256	1088	120732	10.08746	16.88145
11	2048	8256	719691	13.01123	19.45702
14	16384	65600	3687948	16.00141	21.81439
17	131072	524352	21206540	19.00018	24.33801

- ii. 画图发现运行时间测试结果基本符合 $\Theta(d(n+k))$ ，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。



6. 对比

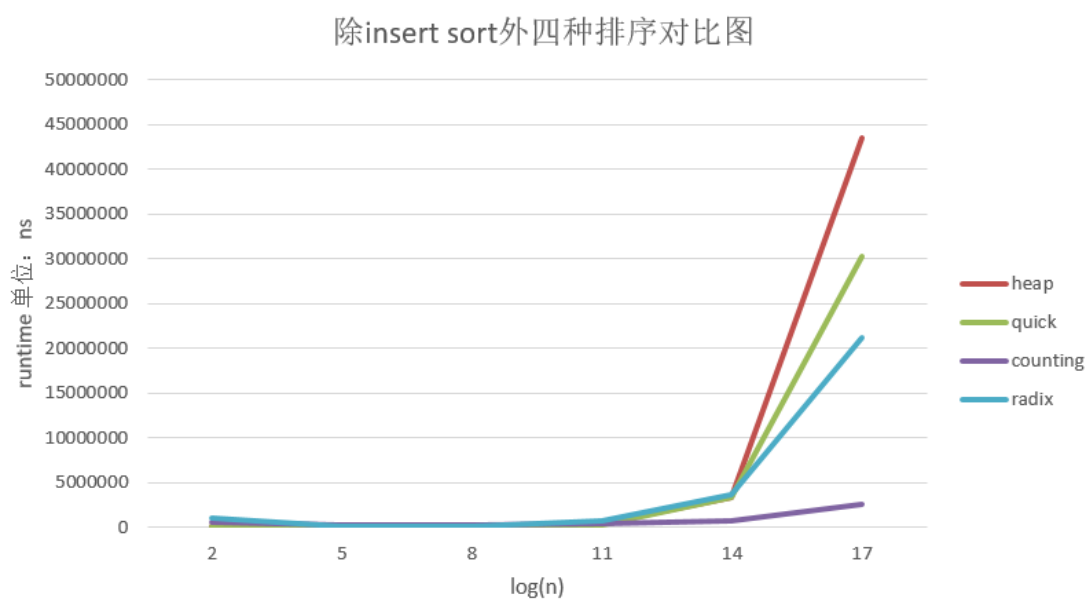
- i. 从实验数据中可以看出数据量较小时，插入排序、堆排序、快速排序的表现非常好。 $n=2$ 时这三个算法的时间都在 1000ns 以下，在 n 小于等于 8 时插入排序、快速排序的表现远好于另外三种排序。

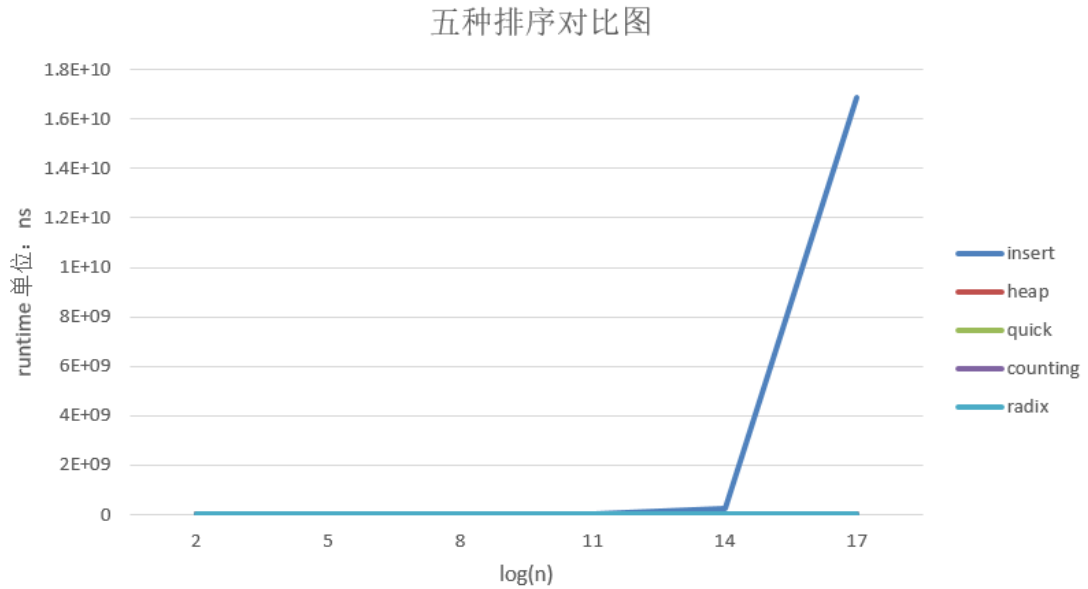
- ii. 当 n 逐渐增大至 14,17 时，计数排序和基数排序的线性时间复杂度优势逐渐体现出来，此时要优于其他三种排序。
- 另外计数排序的时间要小于基数排序，原因可能是运行时间的常数比较小，基数排序的常数的作用明显大于计数排序。

对比	insert	heap	quick	counting	radix
exp	runtime	runtime	runtime(ns)	runtime	runtime(ns)
2	359	998	392	461039	994754
5	1373	3027	2659	298735	69250
8	46561	123977	26776	299811	120732
11	4599429	446399	274224	357858	719691
14	2.71E+08	3372604	3255078	656309	3687948
17	1.68E+10	43535899	30242881	2607846	21206540

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。





六. 实验总结

1. Windows 下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 $1 < n < 17$ 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。
2. 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。
3. 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。
4. 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，调试了一个小时。