实验一 排序算法

袁雨 PB20151804

一、实验内容和要求

1. 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：堆排序，快速排序， 归并排序，计数排序。

2. 编程要求

C/C++，排序算法要自己实现，不能直接调用qsort()等解决。

3. 目录格式

实验需建立根文件夹，文件夹名称为：编号-姓名-学号-project1，在根文件夹下需包括实验报告和ex1子文件夹。实验报告命名为编号-姓名-学号-project1.pdf， ex1子文件夹又包含3个子文件夹：

（1）input文件夹：存放输入数据

输入文件中每行一个随机数据，总行数大于等于2^{15} 。

顺序读取n个数据，进行排序。

Example：用快速排序对2^9个元素进行排序，其随机数据的输入文件路径为编号-姓名- 学号-project1/ex1/input/input.txt，顺序读取前2^9个元素进行排序。

（2）src文件夹：源程序

（3）output文件夹：输出数据

每种算法建立一个子文件夹，其输出结果数据导出到其对应子文件下面。

result_n.txt：排序结果的数据(N为数据规模的指数)，每个数据规模一个输出文件。

time.txt：运行时间效率的数据，五个规模的时间结果都写到同一个文件。

Example：用快速排序对29个元素进行排序，其排序结果文件路径为编号-姓名-学号-project1/ex1/output/quick_sort/result_9.txt。

二、实验设备和环境

1. 实验设备

设备：HUAWEI MateBook X Pro

处理器：Intel(R) Core(TM) i5-10210U CPU @1.60GHz 2.11 GHz

2. 实验环境

vscode, gcc

三、实验方法和步骤

参考课本堆排序、快速排序、归并排序、计数排序的伪代码，将其中的数组都改为从0开始，并修改相关代码。分别测试完成后，在main.c中引用为头文件。

用srand()、rand()等函数生成范围0到RAND_MAX=$2^{15}-1$的随机数。

用fopen()、 fscanf()、fprintf()、fclose()等函数完成对文件的打开、读写、关闭等操作。

用QueryPerformance()等函数完成计时。对输入数据的存取、初始化等均不计入运行时间。

用Excel进行数据处理，作出各算法在不同输入规模下的运行时间曲线图，并对结果进行分析讨论。

四、实验结果和分析

（一）实验结果

1. 目录结构

2. 四个排序算法$n = 2^3$时排序结果的截图

①堆排序：

Algorithm > 5-袁雨-PB20151804-project1 > ex1 > output > heap_sort > result_3.txt

```
2395
9102
10445
10994
12145
21761
29991
30266
```

②快速排序

Algorithm > 5-袁雨-PB20151804-project1 > ex1 > output > quick_sort > result_3.txt

```
2395
9102
10445
10994
12145
21761
29991
30266
```

③归并排序

Algorithm > 5-袁雨-PB20151804-project1 > ex1 > output > merge_sort > result_3.txt

```
2395
9102
10445
10994
12145
21761
29991
30266
```

④计数排序

Algorithm > 5-袁雨-PB20151804-project1 > ex1 > output > counting_sort > result_3.txt

```
2395
9102
10445
10994
12145
21761
29991
30266
```

3. 任一排序算法六个输入规模运行时间的截图

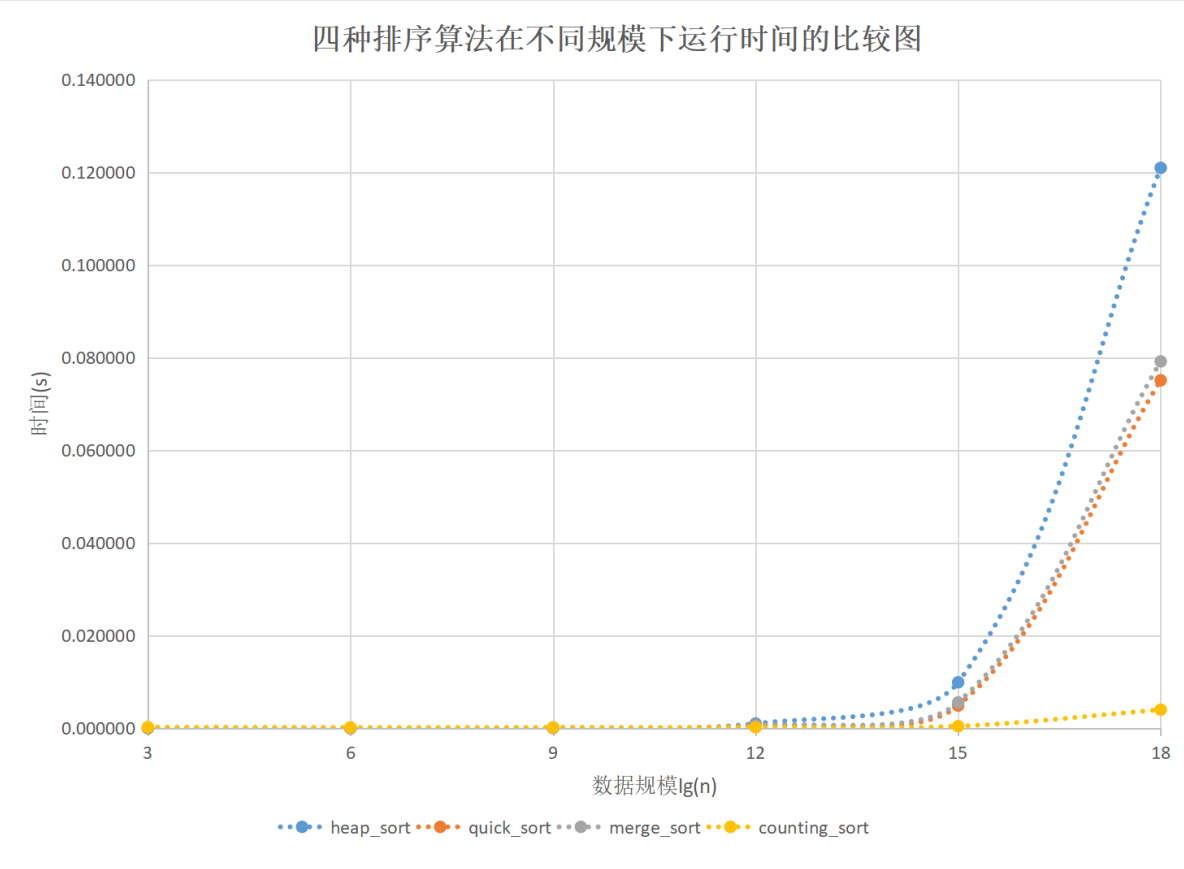
Algorithm > 5-袁雨-PB20151804-project1 > ex1 > output > heap_sort > time.txt

```
0.000001
0.000008
0.000095
0.000998
0.009903
0.120973
```

（二）结果分析

> 四种排序算法在不同规模下运行时间的比较表

lgn	heap_sort（s）	quick_sort（s）	merge_sort(s)	counting_sort(s)
3	0.000001	0.000001	0.000001	0.000226
6	0.000008	0.000004	0.000007	0.000156
9	0.000095	0.000048	0.000059	0.000162
12	0.000998	0.000612	0.000547	0.000186
15	0.009903	0.004879	0.005570	0.000437
18	0.120973	0.075114	0.079186	0.003986



> 课本中的算法渐进性能表

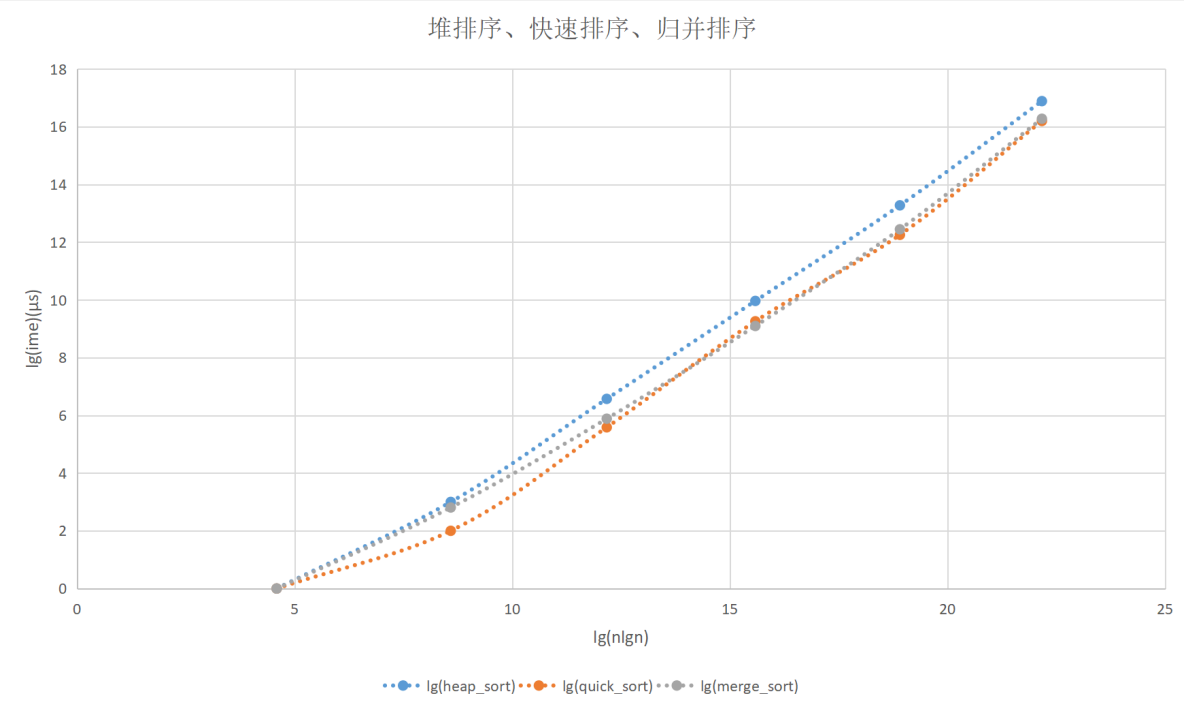
算法	最坏情况运行时间	平均情况 / 期望运行时间
堆排序	$O(nlgn)$	——
快速排序	$\theta(n^2)$	$\theta(nlgn)$（期望）
归并排序	$\theta(nlgn)$	$\theta(nlgn)$
计数排序	$\theta(k+n)$	$\theta(k+n)$

为了方便比较，对实验数据进行处理。

对于堆排序、快速排序和归并排序，计算数据规模lg(nlgn)；将时间的单位转换为微秒，然后计算对数得如下图表。

> 数据处理后的三种算法结果表

lg(nlgn)	heap_sort（μs)	quick_sort（μs)	merge_sort(μs)
4.584962501	0	0	0
8.584962501	3	2	2.807354922
12.16992500	6.569855608	5.584962501	5.882643049
15.58496250	9.962896005	9.257387843	9.095397023
18.90689060	13.27364992	12.25236977	12.44346161
22.16992500	16.88432556	16.19679421	16.27295777

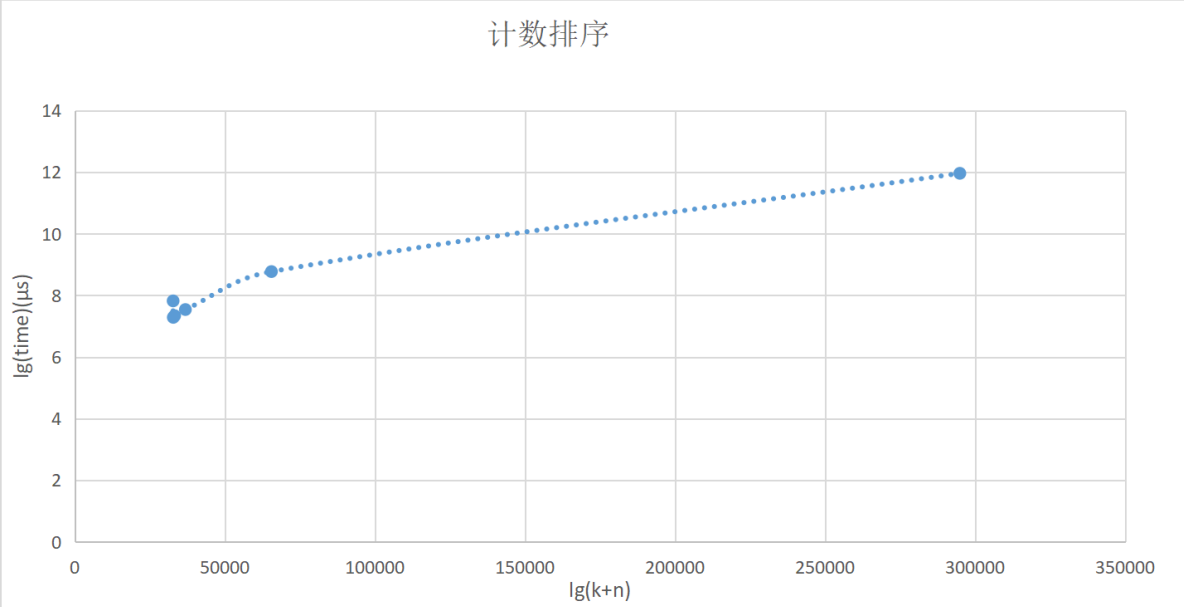


可见堆排序、快速排序、归并排序算法的lg(time)都近似与lg(nlgn)成正比，即运行时间近似与nlgn成正比，符合$\theta(nlgn)$的期望。

对于计数排序，计算数据规模lg(k+n)，其中$k = 2^{15} - 1$；将时间的单位转换为微秒，然后计算对数得如下图表。

> 数据处理后的计数排序结果表

lg(k+n)	counting_sort(μs)
32775	7.820178962
32831	7.285402219
33279	7.339850003
36863	7.539158811
65535	8.77148947
294911	11.96072599



除了第一次（2^3），其他的运行时间都是随数据规模的增长而递增的，因为这几次输入是通过循环依次进行的，故猜测可能是编译器进行了优化等，进行测试，如下图所示。

```c
void COUNTING_SORT(int* A,int* B, int k,int n)
{//A是输入,B存放输出，C提供临时存储空间
    double time;
    LARGE_INTEGER start, end, tc;
    QueryPerformanceFrequency(&tc); //获取机器内部计时器的时钟频率
    QueryPerformanceCounter(&start);
    int C[k];   // C[i] 保存等于i的元素的个数
    // int *C = (int *)malloc(sizeof(int) * k);
    QueryPerformanceCounter(&end);
    time = (end.QuadPart - start.QuadPart) * 1.0 / tc.QuadPart; // 利用两次获得的计数之差和时钟频率，计算出事件经历的精确时间
    printf("\n初始化C[k]耗时:%lf",time);
    QueryPerformanceCounter(&start);
    int i=0,j=0;
    for(i=0;i<=k;i++)
    {
        C[i]=0;
    }
    for(j=0;j<n;j++)
    {
        C[A[j]]=C[A[j]]+1;
    }
    for(i=1;i<=k;i++)
    {
        C[i]=C[i]+C[i-1];
    }
    for(j=n-1;j>=0;j--)
    {
        B[C[A[j]]-1]=A[j];
        C[A[j]]=C[A[j]]-1;
    }
    QueryPerformanceCounter(&end);
    time = (end.QuadPart - start.QuadPart) * 1.0 / tc.QuadPart;
    printf("\n排序耗时:%lf\n",time);
}
 // 顺序读取文件中的元素
 int Get_Num(int *A, int N)
 {
     int i = 0;
     FILE *fp = fopen("..\\input\\input.txt", "r");
     if (fp == NULL)
     {
         printf("文件读取无效.\n");
         return -1;
     }
     for (i = 0; i < N; i++)
         fscanf(fp, "%d", &A[i]);

     fclose(fp);
     return 0;
 }

 //测试
 int main()
 {
     int test_array1[8] = {0};
     int test_array2[64] = {0};
     int result_array1[8]={0};
     int result_array2[64]={0};
     Get_Num(test_array1,8);
     Get_Num(test_array2,64);
     printf("2^3:");
     COUNTING_SORT(test_array1, result_array1,pow(2, 15) - 1, 8);
     for (int i = 0; i < 8; i++)
     {
         printf("%d ", result_array1[i]);
     }
     printf("\n2^6:");
     COUNTING_SORT(test_array2, result_array2,pow(2, 15) - 1, 64);
     for (int i = 0; i < 64; i++)
     {
         printf("%d ", result_array2[i]);
     }
     return 0;
 }
```

```
[Running] cd "d:\vscodefile\c_multiple\Algorithm\5-袁雨-PB20151804-project1\ex1\src\" && gcc counting_sort.c -o counting_sort &&
"d:\vscodefile\c_multiple\Algorithm\5-袁雨-PB20151804-project1\ex1\src\"counting_sort
2^3:
初始化C[k]耗时:0.000127
排序耗时:0.000255
2395 9102 10445 10994 12145 21761 29991 30266
2^6:
初始化C[k]耗时:0.000000
排序耗时:0.000255
1 562 1230 1338 1590 2395 3206 3583 4999 5021 5121 5650 6348 7977 8274 8773 9102 10445 10447 10542 10921 10994 12145 12287 12372
12457 12725 16120 16341 16634 17474 17786 18007 18723 18961 19564 20809 21707 21761 22209 22576 22977 23289 23526 23735 25057 26343
26438 26678 27234 28532 28831 29413 29880 29991 30026 30071 30266 30377 31470 31781 32120 32151 32500
[Done] exited with code=0 in 3.36 seconds
```

测试后证实是编译器对C数组的初始化进行了优化，实际的排序时间应近似符合$\theta(k+n)$的期望。

综上，各算法在不同输入规模下的运行时间曲线与其渐进性能近似相同。

接下来比较不同的排序算法的时间曲线。

在规模较小时，比如$2^3, 2^6, 2^9$，堆排序、快速排序、归并排序更占优势，其中运行时间近似有快速排序≤归并排序 < 堆排序。可能是因为快速排序的代码很紧凑，运行时间中隐含的常数系数很小，故在实际应用中通常较快。

在规模较大时，比如$2^{12}, 2^{15}, 2^{18}$，$k = O(n)$，计数排序的运行时间与输入数组的规模呈线性关系，更占优势。