

实验一 排序算法

陈俊驰 PB18000051

一、实验内容：

本次实验要求排序 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} 。

对于该排序功能，需要通过 C/C++ 实现以下的算法：直接插入排序，堆排序，快速排序，归并排序，计数排序。

对于输入，input 文件夹中的 input.txt 文件每行存储一个随机数，总行数大于等于 2^{18} 。每次输入将顺序读取 n 个数据进行排序。

对于输出，分别用这五种算法对于这六种规模的数据进行排序，将结果输出到 output 文件夹中的该算法对应子文件夹。排序结果数据用 result_n.txt 存储， n 为数据规模的指数。六种规模的排序的时间结果统一用 time.txt 存储。

注意，其中元素为由 stdlib.h 中的 rand() 函数生成的随机数，以保证元素的均匀分布。不会出现重复太多导致的排序结果受到影响。

二、实验设备 and 环境：

本次实验所使用的实验设备为笔记本电脑，型号为 2018 版联想拯救者，处理器为 Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz，内存为 8.00GB。

所使用的实验环境为 Windows 10，所使用的 IDE 为 Visual Studio 2019。

三、实验方法和步骤：

1. 首先，根据实验文档的名称与结构要求建立根文件夹，文件夹名称为 36-陈俊驰-PB18000051-project1，在根文件夹下建立实验报告文件和 ex1 子文件夹，在子文件夹中创立三个子文件夹，分别为 input 文件夹，用于存储生成的随机数文件 input.txt；src 文件夹，用于存储源程序，包含项目文件夹及 cpp 文件，由于所使用的 IDE 为 Visual Studio，每个源程序对应一个项目文件夹，由于相对路径问题，在执行时需要打开该项目在 VS 中执行，而不能直接通过使用 Debug 文件夹中存储的 exe 文件；output 文件夹，其中为每个算法在 output 文件夹下构建一个子文件夹用于存储排序结果数据和时间结果。

2. 建立一个项目 sortInput，通过使用 stdlib.h 中的 rand() 函数，生成 2^{18} 个介于 0 和 $2^{15}-1$ 之间的随机数，每行一个，输入到 input/input.txt 文件中，作为之后排序算法的待排序输入数据。核心代码段如下

```
count = pow(2, 18);  
//生成并写入 $2^{18}$ 个随机数  
for (i = 1; i <= count; i++)  
    fprintf(fp, "%d\n", rand());
```

图 1 生成 input.txt

3. 分别为五种算法：直接插入排序，堆排序，快速排序，归并排序，计数排序建立一个项目。具体对于每一种算法的实现见相关 cpp 文件。以直接插入排序为例，说明实现方法。

A) 首先通过文件指针打开输入输出文件，其中后缀 i 为读入 input.txt，后缀 o3 到 o18 分别对应 3 到 18 的指数规模输出文件，后缀 time 对应时间结果文件；

```
FILE* fp_i, * fp_o3, * fp_o6, * fp_o9, * fp_o12, * fp_o15, * fp_o18, * fp_time;
int i, k;
fp_i = fopen("../..\\input\\input.txt", "r");
fp_o3 = fopen("../..\\output\\insert_sort\\result_3.txt", "w");
fp_o6 = fopen("../..\\output\\insert_sort\\result_6.txt", "w");
fp_o9 = fopen("../..\\output\\insert_sort\\result_9.txt", "w");
fp_o12 = fopen("../..\\output\\insert_sort\\result_12.txt", "w");
fp_o15 = fopen("../..\\output\\insert_sort\\result_15.txt", "w");
fp_o18 = fopen("../..\\output\\insert_sort\\result_18.txt", "w");
fp_time = fopen("../..\\output\\insert_sort\\time.txt", "w");
```

图 2 打开文件

B) 然后对不同规模的数据重复如下操作。首先利用申请的全局指针变量在堆中申请一个空间作为数组来存储数据；然后从 input.txt 中顺序读入要排序的数据并将文件指针重置到开始，为下次读取做准备；开始计时并调用直接插入排序，输入为数组和规模，结果为将排序过的数据存储在该数组中返回，停止计时；根据计时的结果（这里用的是纳秒级精度），写入到 time.txt 文件；再将排序的结果写入到相应的 result_n.txt 文件；最后释放空间。

```
//对应规模2^3
//读取数据
A3 = (int*)malloc((8 + 1) * sizeof(int));
for (i = 1; i <= 8; i++)
{
    fscanf(fp_i, "%d", &k);
    A3[i] = k;
}
//将文件指针重置到开始，为下次读取做准备
fseek(fp_i, 0, SEEK_SET);
//开始计时
auto start = system_clock::now();
//对2^3规模数据调用插入排序
Insert_Sort(A3, 8);
//结束计时，输出计时结果
auto end = system_clock::now();
auto duration = duration_cast<nanoseconds>(end - start);
fprintf(fp_time, "对于2^3规模花费了 %lf us\n", 1000000*double(duration.count()) * nanoseconds::period::num / nanoseconds::period::den);
//输出排序结果
for (i = 1; i <= 8; i++)
{
    fprintf(fp_o3, "%d\n", A3[i]);
}
//释放分配的空间
free(A3);
```

图 3 对于一定规模的处理

C) 其中，对于不同的算法具体实现代码如下：

```
void Insert_Sort(int a[], int n)
//插入排序，其中a[]为所要排序的数组，n为数据量，通过main函数传入
{
    int i, j, key;
    for (j = 2; j <= n; j++)
    {
        key = a[j];
        //将a[j]插入到已排序的序列a[1...j-1]
        for (i = j - 1; i > 0; i--)
            if (a[i] > key)
                //将每一个比a[j]大的右移一位，直到找到一个key介于左右两元素大小之间的
                a[i + 1] = a[i];
            else
                break;
        a[i + 1] = key;
    }
}
```

图 4 直接插入排序代码

```

int Parent(int i) { ... }
int Left(int i) { ... }
int Right(int i) { ... }

void Max_Heapify(int A[], int i, int size)
{
    //输入一个数组和规模，对某一个结点i进行调整以维护最大堆
    int l = Left(i);
    int r = Right(i);
    int temp;
    int largest;
    //通过比较获得该结点i和其左右孩子中最大节点的序号
    if (l <= size && A[l] > A[i])
        largest = l;
    else
        largest = i;
    if (r <= size && A[r] > A[largest])
        largest = r;
    //通过该最大节点和i的交换，使得i及其孩子满足最大堆性质，然后再去调整以该结点i新位置为根的子树
    if (largest != i)
    {
        temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        Max_Heapify(A, largest, size);
    }
}

void Build_Max_Heap(int A[], int size)
{
    //输入一个数组和规模，使之建立一个最大堆
    int i;
    for (i = size / 2; i >= 1; i--)
        //由于i+1到size都为叶子，故只要调整剩余结点即可
        Max_Heapify(A, i, size);
}

void Heap_Sort(int A[], int size)
{
    //输入一个数组及规模，获得其排序结果
    int i, temp;
    //首先将其建为最大堆
    Build_Max_Heap(A, size);
    //每次取出一个最大值然后再对剩余进行调整以维护最大堆性质
    for (i = size; i >= 2; i--)
    {
        temp = A[i];
        A[i] = A[1];
        A[1] = temp;
        size = size - 1;
        Max_Heapify(A, 1, size);
    }
}

```

图 5.1 及 5.2 堆排序代码

```

int Partition(int A[], int p, int r)
{
    //选中A[r]作为主元，来划分大小
    int x = A[r];
    int i = p - 1;
    int j, tem;
    for (j = p; j <= r - 1; j++)
    {
        //用j来统计所遍历的元素，i来统计小于等于x的元素
        if (A[j] <= x)
        {
            //将小于等于x的元素移到左侧
            i = i + 1;
            tem = A[i];
            A[i] = A[j];
            A[j] = tem;
        }
    }
    //确定了小于等于x的元素序号到i截止，故用A[i+1]存储x
    tem = A[r];
    A[r] = A[i + 1];
    A[i + 1] = tem;
    //返回划分的结果
    return i + 1;
}

void Quick_Sort(int A[], int p, int r)
{
    int q;
    if (p < r)
    {
        //当数组范围合理时，首先进行划分，将数组分为A[p, q-1],
        //使之每个元素都小于等于A[q], 和A[q+1, r], 每个元素都大于等于A[q]
        q = Partition(A, p, r);
        //对划分出的两个子数组排序
        Quick_Sort(A, p, q - 1);
        Quick_Sort(A, q + 1, r);
    }
}

```

图 6 快速排序代码

```

void MERGE(int A[], int p, int q, int r)
{
    //输入一个数组A[p, r], 其中A[p, q]A[q+1, r]已经有序，将二者合并为一个有序数组
    int n1 = q - p + 1;
    int n2 = r - q;
    int* L = (int*)malloc((n1 + 2) * sizeof(int));
    int* R = (int*)malloc((n2 + 2) * sizeof(int));
    int i, j, k;
    //将数组数据拷贝下来
    for (i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];
    for (j = 1; j <= n2; j++)
        R[j] = A[q + j];
    //设置哨兵
    L[n1 + 1] = 65536;
    R[n2 + 1] = 65536;
    i = 1; j = 1;
    //比较并将L和R中最小的元素按序抄入A[p, q]
    for (k = p; k <= r; k++)
    {
        if (L[i] <= R[j])
        {
            A[k] = L[i];
            i = i + 1;
        }
        else
        {
            A[k] = R[j];
            j = j + 1;
        }
    }
}

void Merge_Sort(int A[], int p, int r)
{
    //输入一个数组及其上下限，对该范围内的元素进行排序
    int q;
    if (p < r)
    {
        //在该范围合规的情况下，先对左右两部分分别排序
        q = (p + r) / 2;
        Merge_Sort(A, p, q);
        Merge_Sort(A, q + 1, r);
        //再对排序好的左右两部分进行合并
        MERGE(A, p, q, r);
    }
}

```

图 7 归并排序代码

```

void Counting_Sort(int A[], int size)
{
    //输入一个数组和其规模，对数组进行排序
    int i, j;
    //申请数组C存储各数值元素个数
    C = (int*)malloc(32768 * sizeof(int));
    //申请数组B存储排序结果
    B = (int*)malloc((size + 1) * sizeof(int));
    //对数组C初始化
    for (i = 0; i <= 32767; i++)
        C[i] = 0;
    //统计等于各数值的元素个数
    for (j = 1; j <= size; j++)
        C[A[j]] += 1;
    //现在修改C，使之C[i]存储小于等于i的元素个数
    for (i = 1; i <= 32767; i++)
        C[i] = C[i] + C[i - 1];
    for (j = size; j >= 1; j--)
    {
        //遍历数组A，根据A[j]的值与表征位置的C来获得它在排序后的B的正确位置上
        B[C[A[j]]] = A[j];
        //使该值的可用位置减一
        C[A[j]] -= 1;
    }
    for (i = 1; i <= size; i++)
        A[i] = B[i];
    free(B);
    free(C);
}

```

图 8 计数排序代码

4. 对于上述的程序，由于在最初几次运行时，排序耗时不稳定，故重复多次运行取最后稳定的结果，以增加结果的可靠性。

5. 最终获得了一系列排序结果及不同算法的运行时间，对于该数据进行比较分析并撰写实验报告。

四、实验结果与分析：

首先对于每个算法的运行结果进行展示，即展示相关排序和时间结果的截图

1.直接插入排序

对于直接插入排序，通过运行代码获得六个 result_n.txt 文件，通过检查，发现排序结果正确，故成功实现了直接插入排序算法。其中对于 2^3 规模的排序结果如下图

36-陈俊驰-PB18000051-project1 > ex1 > output > insert_sort

result_3.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```

41
6334
11478
15724
18467
19169
26500
29358

```

图 9 直接插入排序 2^3 规模排序结果

其中，直接插入排序对于六个输入规模的运行时间如下

36-陈俊驰-PB18000051-project1 > ex1 > output > insert_sort



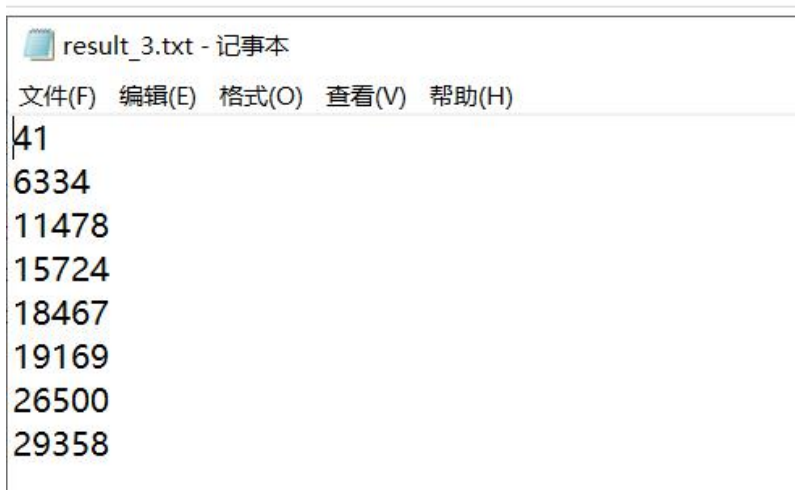
```
time.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
对于2^3规模花费了 0.900000 us
对于2^6规模花费了 3.200000 us
对于2^9规模花费了 150.200000 us
对于2^12规模花费了 9005.900000 us
对于2^15规模花费了 573174.600000 us
对于2^18规模花费了 36135133.200000 us
```

图 10 直接插入排序运行时间

2.堆排序

对于堆排序，通过运行代码获得六个 result_n.txt 文件，通过检查，发现排序结果正确，故成功实现了堆排序算法。其中对于 2^3 规模的排序结果如下图

36-陈俊驰-PB18000051-project1 > ex1 > output > heap_sort



```
result_3.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41
6334
11478
15724
18467
19169
26500
29358
```

图 11 堆排序 2^3 规模排序结果

其中，堆排序对于六个输入规模的运行时间如下

36-陈俊驰-PB18000051-project1 > ex1 > output > heap_sort



```
time.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
对于2^3规模花费了 9.600000 us
对于2^6规模花费了 100.800000 us
对于2^9规模花费了 492.200000 us
对于2^12规模花费了 3321.100000 us
对于2^15规模花费了 26280.700000 us
对于2^18规模花费了 245765.400000 us
```

图 12 堆排序运行时间

3.快速排序

对于快速排序，通过运行代码获得六个 result_n.txt 文件，通过检查，发现排序结果正确，故成功实现了快速排序算法。其中对于 2^3 规模的排序结果如下图

36-陈俊驰-PB18000051-project1 > ex1 > output > quick_sort



```
result_3.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41
6334
11478
15724
18467
19169
26500
29358
```

图 13 快速排序 2^3 规模排序结果

其中，快速排序对于六个输入规模的运行时间如下

36-陈俊驰-PB18000051-project1 > ex1 > output > quick_sort



```
time.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
对于2^3规模花费了 2.200000 us
对于2^6规模花费了 10.400000 us
对于2^9规模花费了 47.200000 us
对于2^12规模花费了 437.300000 us
对于2^15规模花费了 4140.900000 us
对于2^18规模花费了 39932.300000 us
```

图 14 快速排序运行时间

4.归并排序

对于归并排序，通过运行代码获得六个 result_n.txt 文件，通过检查，发现排序结果正确，故成功实现了归并排序算法。其中对于 2^3 规模的排序结果如下图

36-陈俊驰-PB18000051-project1 > ex1 > output > merge_sort

```
result_3.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41
6334
11478
15724
18467
19169
26500
29358
```

图 15 归并排序 2^3 规模排序结果

其中，归并排序对于六个输入规模的运行时间如下

36-陈俊驰-PB18000051-project1 > ex1 > output > merge_sort

```
time.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
对于2^3规模花费了 12.900000 us
对于2^6规模花费了 37.700000 us
对于2^9规模花费了 263.300000 us
对于2^12规模花费了 2664.900000 us
对于2^15规模花费了 16965.700000 us
对于2^18规模花费了 136900.300000 us
```

图 16 归并排序运行时间

5.计数排序

对于计数排序，通过运行代码获得六个 result_n.txt 文件，通过检查，发现排序结果正确，故成功实现了计数排序算法。其中对于 2^3 规模的排序结果如下图

36-陈俊驰-PB18000051-project1 > ex1 > output > counting_sort

```
result_3.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41
6334
11478
15724
18467
19169
26500
29358
```

图 17 计数排序 2^3 规模排序结果

其中，计数排序对于六个输入规模的运行时间如下

time.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

对于2^3规模花费了 317.600000 us
 对于2^6规模花费了 194.300000 us
 对于2^9规模花费了 237.800000 us
 对于2^12规模花费了 234.400000 us
 对于2^15规模花费了 697.500000 us
 对于2^18规模花费了 3022.500000 us

图 18 计数排序运行时间

然后对实验进行分析：

对于排序结果，通过人工检查发现元素均按照从小到大顺序输出，故算法实现正确。

对于时间分析

首先，单独对各算法的运行时间进行分析（即实验文档中的第一个分析题目：画出各算法在不同输入规模下的运行时间曲线图。比较并分析是否与课本上算法渐进性能相同）

1 直接插入排序

根据直接插入排序输出的 time.txt 文件中的数据可以获得如下的运行时间规模图

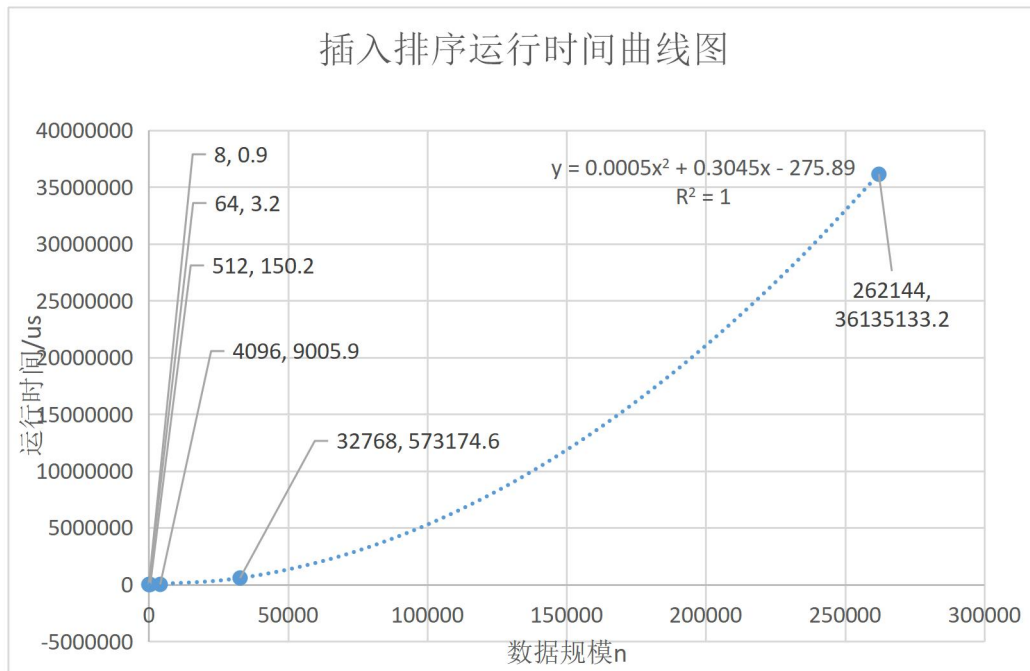


图 19 直接插入排序时间曲线图

其中横坐标为数据规模 n ，纵坐标为运行时间，单位为 μs 。通过程序拟合，发现其运行时间与规模的关系符合表达式 $t=0.0005n^2+0.3045n-275.89$ 的函数，该函数为 $\Theta(n^2)$ 。其中 $R^2=1$ ，说明拟合程度极高，结果很准确。由于输入数据为随机生成，故可以视为平均情况，通过课本查询知，直接插入排序的平均情况运行

时间为 $\Theta(n^2)$ ，故测量结果与理论的算法渐进性能相同。

2.堆排序

根据堆排序输出的 `time.txt` 文件中的数据可以获得如下的运行时间规模图

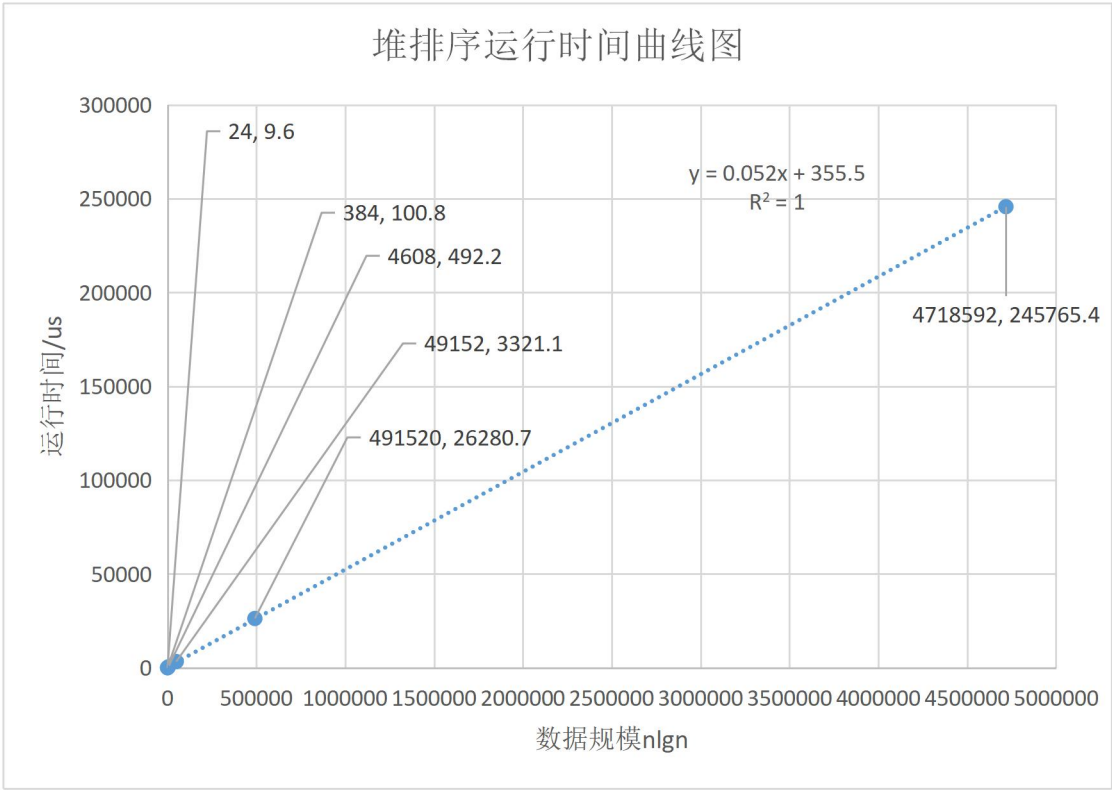


图 20 堆排序时间曲线图

其中横坐标为数据规模 $nlgn$ ，纵坐标为运行时间，单位为 `us`。通过程序拟合，发现其运行时间与规模的关系符合表达式 $t=0.052nlgn+355.5$ 的函数，该函数为 $\Theta(nlgn)$ 。其中 $R^2=1$ ，说明拟合程度极高，结果很准确。由于输入数据为随机生成，故可以视为平均情况，通过课本查询知，堆排序的运行时间为 $O(n^2)$ ，故测量结果与理论的算法渐进性能相同。

3.快速排序

根据快速排序输出的 `time.txt` 文件中的数据可以获得如下的运行时间规模图

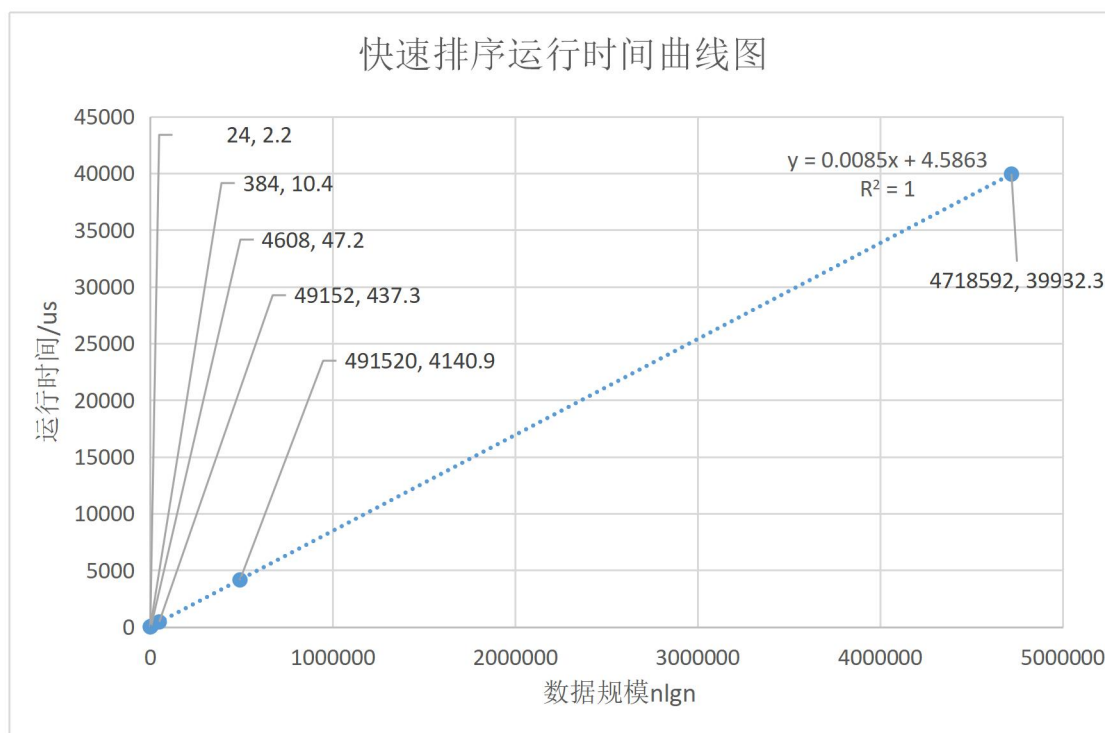


图 21 快速排序时间曲线图

其中横坐标为数据规模 $n \lg n$ ，纵坐标为运行时间，单位为 μs 。通过程序拟合，发现其运行时间与规模的关系符合表达式 $t=0.0085n \lg n+4.5863$ 的函数，该函数为 $\Theta(n \lg n)$ 。其中 $R^2=1$ ，说明拟合程度极高，结果很准确。由于输入数据为随机生成，故可以视为平均情况，通过课本查询知，快速排序的平均情况运行时间为 $\Theta(n \lg n)$ ，故测量结果与理论的算法渐进性能相同。

4. 归并排序

根据归并排序输出的 `time.txt` 文件中的数据可以获得如下的运行时间规模图

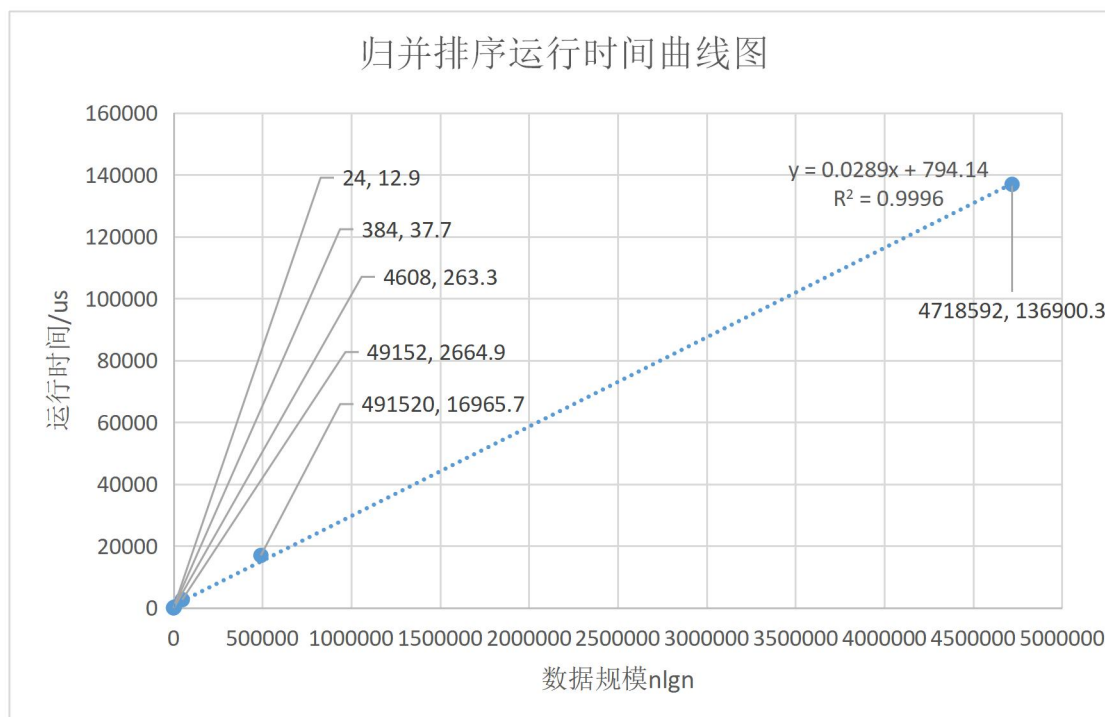


图 22 归并排序时间曲线图

其中横坐标为数据规模 $n \lg n$ ，纵坐标为运行时间，单位为 μs 。通过程序拟合，发现其运行时间与规模的关系符合表达式 $t = 0.0289n \lg n + 794.14$ 的函数，该函数为 $\Theta(n \lg n)$ 。其中 $R^2 = 1$ ，说明拟合程度极高，结果很准确。由于输入数据为随机生成，故可以视为平均情况，通过课本查询知，归并排序的平均情况运行时间为 $\Theta(n \lg n)$ ，故测量结果与理论的算法渐进性能相同。

5. 计数排序

根据计数排序输出的 `time.txt` 文件中的数据可以获得如下的运行时间规模图

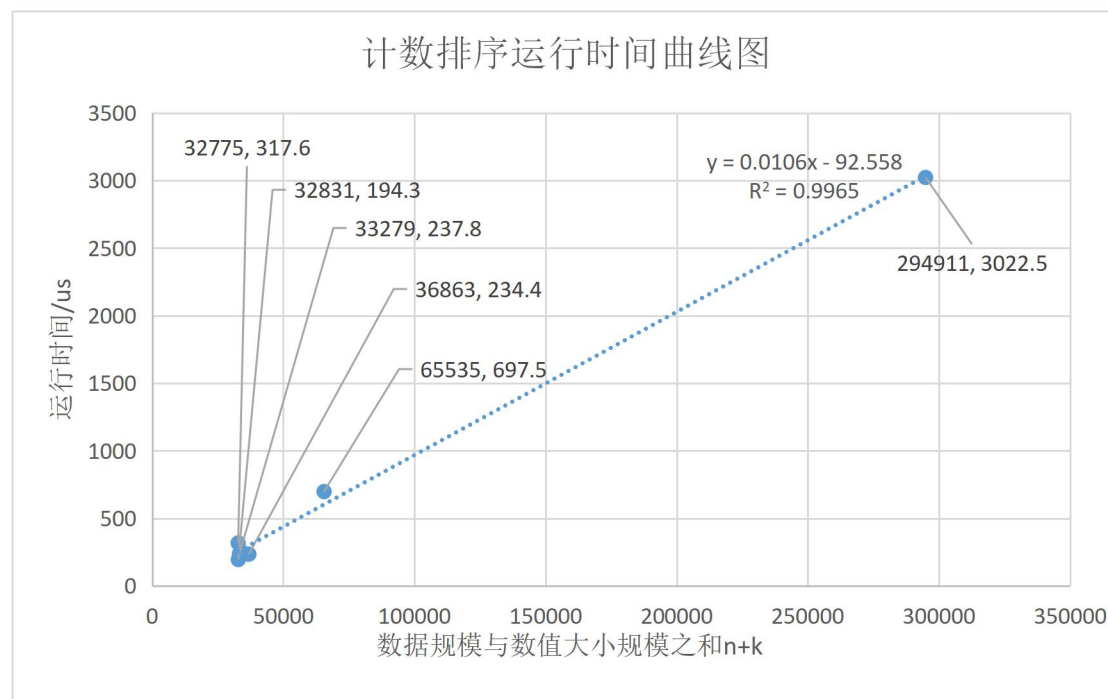


图 23 计数排序时间曲线图

其中横坐标为数据规模 $n+k$ ，纵坐标为运行时间，单位为 μs 。通过程序拟合，发现其运行时间与规模的关系符合表达式 $t = 0.0106(n+k) - 92.558$ 的函数，该函数为 $\Theta(n+k)$ 。其中 $R^2 = 0.9965$ ，说明拟合程度较好，结果较为准确。由于输入数据为随机生成，故可以视为平均情况，通过课本查询知，计数排序的平均情况运行时间为 $\Theta(n+k)$ ，故测量结果与理论的算法渐进性能相同。

但是通过数据我们发现，在数据规模较小的时候， 2^3 的数据耗时甚至多余 2^6 和 2^9 。这与我们理论分析的随着 n 增大，时间将会线性增大不同。一方面是由于本身渐进时间复杂度的分析是针对较大数据量的情况，数据量大到忽略其他的低阶项，故对于较小的数量，本身该规律并不明显；另一方面，由于在我的程序中最先对 2^3 规模进行调用，且函数中用于存储排序结果的堆数组 `B` 和用于统计各数值元素数量的数组 `C` 此时第一次被申请、赋值、释放，故推测由于本程序内存中对于空间的初次分配与操作所导致耗时较大。在此后的调用中，由于对应的内存已经被申请赋值过一次，所以相同操作耗时将会减少。此外，还有可能是程序运行次数较少和系统 CPU 算力不稳定所导致的结果误差出现。

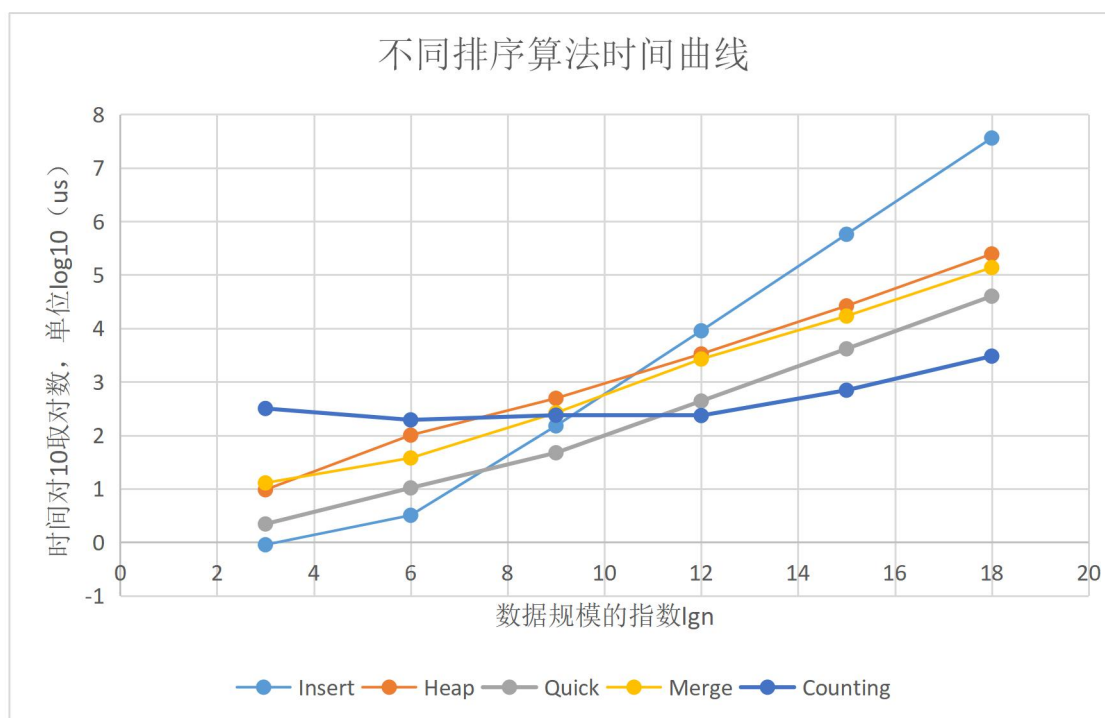
对于上述算法的运行时间单独分析完后，再将他们结合起来进行分析（即实验文档第二个分析题目：比较不同的排序算法的时间曲线，分析在不同输入规模

下哪个更占优)

通过各算法的输出 time.txt 文件，我们可以获得如下的规模与时间表格

时间单位为 us	输入规模					
	2^3	2^6	2^9	2^12	2^15	2^18
直接插入排序	0.9	3.2	150.2	9005.9	573174.6	36135133.2
堆排序	9.6	100.8	492.2	3321.1	26280.7	245765.4
快速排序	2.2	10.4	47.2	437.3	4140.9	39932.3
归并排序	12.9	37.7	263.3	2664.9	16965.7	136900.3
计数排序	317.6	194.3	237.8	234.4	697.5	3022.5

根据上述结果可以画出不同排序算法的时间曲线如下图所示，其中为了便于区分各点的差距，横坐标为数据规模的指数，即 $\lg n$ ，纵坐标为时间对 10 取对数的结果，其中由于时间的单位为 us，则纵坐标单位为 $\log_{10}(\text{us})$ 。



对于数据规模为 2^3 和 2^6 的数据，此时**直接插入排序**耗时最少，最占优势。

主要是由于直接插入排序为原址排序不需要额外的空间申请与赋值，在规模较少的情况下，如计数排序，申请空间和对所申请的空间进行处理的耗时已远大于本身排序的时间。此外，直接插入排序过程简单，不需要额外的复杂操作以及对于函数的递归调用，同时节省了时间与空间。

对于数据规模为 2^9 的数据，此时**快速排序**耗时最少，最占优势。

此时数据规模已经开始增大，快速排序作为 $\Theta(n \lg n)$ 的时间复杂的优势开始显现。此时直接插入排序由于其 $\Theta(n^2)$ 的时间复杂度，耗时快速上升；堆排序和归并排序由于有着大量的递归与额外对于数组的操作，导致其隐藏的常数项以及 $n \lg n$ 项的系数较高，耗时较多；时间复杂度 $\Theta(n+k)$ 的计数排序由于需要对对应 k 种数值的数组进行处理，且 k 相对 n 来说仍然很大，故 k 引入的耗时使之耗时仍

然较多，不占优势。

对于数据规模为 2^{12} , 2^{15} , 2^{18} 的数据，此时**计数排序**耗时最少，最占优势。

此时，数据规模 n 已经较大了，计数排序由于 k 所引入的时间消耗已经由于其 $O(n+k)$ 的时间复杂度的优势弥补回来。而其他四种算法由于时间复杂度为 $\Theta(n^2)$, $\Theta(n \lg n)$ ，导致耗时很大。相较之下，计数排序最占优势。由于计数排序需要额外申请大量的空间，故如果需要为了空间牺牲时间，其他的 $\Theta(n \lg n)$ 算法也不失为一种权宜之计。

五、反省与总结：

本次实验中，复习了排序算法的相关知识，并在实际编程中对于各类算法进行了运用与实践，加深了对于课本排序部分相关知识的理解。此外，对于不同算法、不同规模下排序时间的比较，更为实际且形象的理解了时间复杂度这一概念，并且对于不同算法的时间性能的差异有了更深的认识。此外，通过实验过程中的学习，还初步掌握到了对于计时函数的使用。

在实验过程中，遇到了栈空间不足的问题，通过查询以前的资料和上网搜索，发现可以使用全局指针来在堆中申请空间，从而解决了这个问题。在今后的实验中，应当更加细心认真，对于不了解不清楚的知识应当抓紧学习。

