算法设计与分析实验报告

实验题目：排序算法

姓名：马天成

学号：2020211376

日期：2022-10-10

目录

一、实验环境

1.1 设备规格

设备规格

Legion R7000P2020H

设备名称	PC
处理器	AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
机带 RAM	16.0 GB (15.9 GB 可用)
设备 ID	370678B7-1FA2-4C35-8BAB-F92D3429406B
产品 ID	00342-35932-44511-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器
笔和触控	为 10 触摸点提供笔和触控支持

1.2 操作系统

Windows 规格

版本	Windows 10 家庭中文版
版本号	21H2
安装日期	2022/4/5
操作系统内部版本	19044.2006
序列号	PF2660CA
体验	Windows Feature Experience Pack 120.2212.4180.0

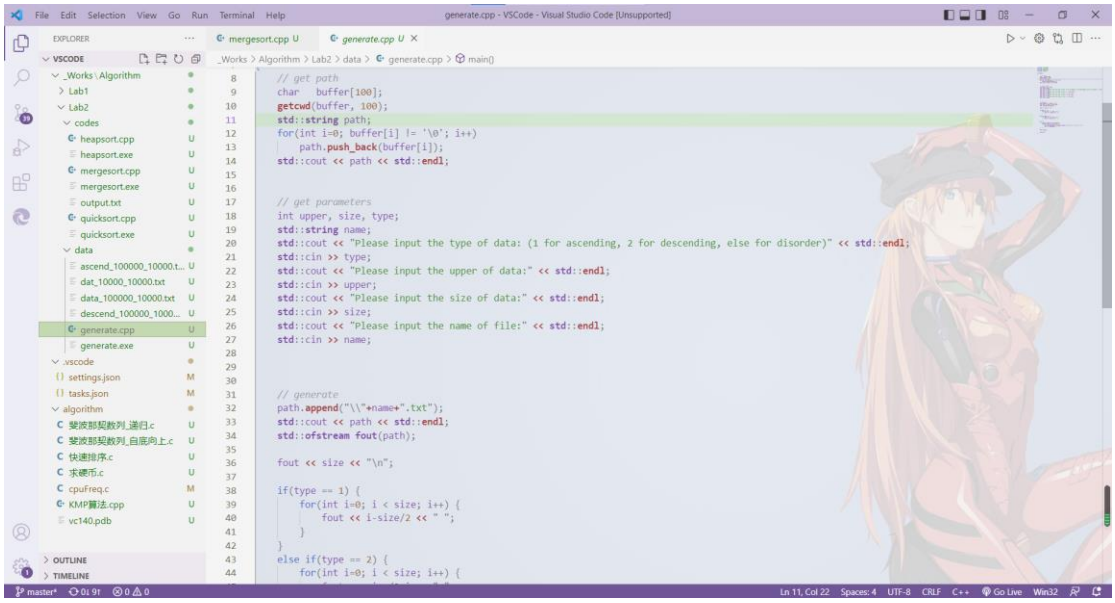
1.3 编程语言&编译器

C++11

```
version : MinGW-W64-builds-4.3.5
user    : nixman
date    : 05.12.2018-10:29:36 AM
```

1.4 开发工具

VSCode



二、实验内容

2.1 实验目的

理解分治法的策略，掌握基于递归的分治算法的实现方法；
掌握基于分治法的合并排序、快速排序的实现方法；
理解并掌握在渐进意义下的算法复杂性的评价方法；
掌握算法测试的基本流程。

2.2 实验内容及要求

- 算法的设计与实现

  设计并实现堆排序、归并排序（合并排序）、快速排序算法，通过比较三种排序算法在不同数据量的情况下所需的移动次数、比较次数，分析算法在最差情况、平均情况下的算法复杂度。

- 测试要求

  设计测试数据集，编写测试程序，用于测试：

  a) 正确性：所实现的三种算法的正确性；

b) 算法复杂性：三种排序算法中，设计测试数据集，评价各个算法在复杂性上的表现
c) 效率：在三种排序算法中，设计测试数据集，评价各个算法中比较，移动的频率。

2.3 快速排序

快速排序的思路也是**分治法**：以标杆元素为中心插入（左边都小，右边都大）分为两边，继续找标杆元素分为两边，直到只剩一个元素。这就是分治的思想。

下面我觉得递归没有意思，本人也非常讨厌递归，总觉得十分冗杂还带有一丝危险性。所以我写的是非递归。

```
79      // sort
80      SizeType left, right, i, j;
81      while(myQueue.size()) {
82          left = myQueue.front().l_index;
83          right = myQueue.front().r_index;
84          i = left, j = right;
85          myQueue.pop();
86
87          int x = data[left]; // fisrt as
88          while (i < j) ···
122         // insert the last one
123         data[i] = x;
124         count_insert++;
125
126         // create two pair into queue
127         count_compare++;
128         if(i-1 > left) {
129             Pair l_pair;
130             l_pair.l_index = left, l_pair.r_index = i-1;
131             myQueue.push(l_pair);
132         }
133         count_compare++;
134         if(i+1 < right) {
135             Pair r_pair;
136             r_pair.l_index = i+1, r_pair.r_index = right;
137             myQueue.push(r_pair);
138         }
139     }
140 }
```

其实快排没有那么难以理解，也不是很难，主要讲的是非递归的写法。

在实验一中，其实我已经用了队列的思路来进行消除非递归。这是一种非常好用的方式。对于分治这种层级性的问题，每一次队列循环意味着处理了一个层级。

所以先把 data 数组的两个左右 index 入队列。然后循环取出队列元素，进行分治再把分治后的子问题入队列。当子问题规模缩小到一则解决，不入队列。这样就可以解决问题了。

2.4 归并排序

归并排序的思路也是分治法。相较于快速排序，他有一个放和收的过程。我们把整个数组分成最小子问题，然后一直进行子问题的合成。这里的分治体现在最初的分成最小子问题。

```cpp
void mergesort(DataType data[], SizeType size)
{
    // initialize queue with each unit(devide)
    std::queue<Pair> myQueue;
    for(SizeType i=0; i < size; i++) {
        Pair tmp;
        tmp.l_index = i, tmp.r_index = i;
        myQueue.push(tmp);
    }

    // merge
    SizeType l_left, l_right, r_left, r_right;
    DataType* tempArr = new DataType[size]; // memory temp merge
    while(myQueue.size() > 1) {

        // even: skip merging the first one-----------------------------------
        count_compare++;
        if(myQueue.size()&1) {
            myQueue.push(myQueue.front());
            myQueue.pop();
        }
        SizeType times = myQueue.size()/2;

        for(SizeType i=0; i < times; i++) {
            // get index of start and end & push merged one-------------------
            l_left = myQueue.front().l_index;
            l_right = myQueue.front().r_index;
            myQueue.pop();
            r_left = myQueue.front().l_index;
            r_right = myQueue.front().r_index;
            myQueue.pop();
            Pair tmp;
            tmp.l_index = l_left, tmp.r_index = r_right;
            myQueue.push(tmp);

            // merge into tempArr(combine)----------------------------------
            SizeType left = l_left, j=0;

            while(1) {
                count_compare++;        // l_left > l_right
                if(l_left > l_right)
                    break;

                count_compare++;        // r_left > r_right
                if(r_left > r_right)
                    break;

                count_compare++;        // l_left <= l_right, r_left <= r_right, data[l_left] < data[r_left]
                if(data[l_left] < data[r_left])
                    tempArr[j++] = data[l_left++];
                else
                    tempArr[j++] = data[r_left++];
            }

            while(1) {
                count_compare++;        // l_left <= l_right
                if(l_left > l_right)
                    break;
                tempArr[j++] = data[l_left++];
            }
            while(1) {
                count_compare++;        // r_left <= r_right
                if(r_left > r_right)
                    break;
                tempArr[j++] = data[r_left++];
            }

            // insert into data---------------------------------------------
            j=0;
            while(1) {
                if(left+j > r_right)
                    break;
                count_insert++;
                //std::cout << data[left+j] << " ";
                data[left+j++] = tempArr[j];
            }
            //std::cout << "\n";
        }
    }

    delete tempArr;
}
```

归并就是把两段排排序，合成一段，然后放到原来的位置上。这里肯定需要一段特殊的暂存空间。这里最后一段 combine 需要整个数组相等的空间，那就肯定得开个一样的空间。

combine：把两段按序排号放在暂存数组空间里，然后把复制回去。

2.5 堆排序

堆排序对我来说是比较难以理解的。因为它牵扯到很多我不知道的知识。主要是完全

堆排序其实也是一种选择排序，是一种树形选择排序。树形选择排序恰好利用树形的特点保存了部分前面的比较结果，因此可以减少比较次数。

所以现在的核心问题是如何解决建堆和 sort 的问题。

- 调整堆

```
77  void adjust(DataType data[], SizeType len, SizeType index)
78  {
79      SizeType left = 2*index + 1;    // index的左子节点
80      SizeType right = 2*index + 2;   // index的右子节点
81
82      SizeType maxIdx = index;
83      count_compare += 3;             // left<len, right<len, maxIdx != index
84      if(left<len) {
85          count_compare++;            // data[left] > data[maxIdx]
86          if(data[left] > data[maxIdx])
87              maxIdx = left;
88      }
89      if(right<len) {
90          count_compare++;            // data[right] > data[maxIdx]
91          if(data[right] > data[maxIdx])
92              maxIdx = right;
93      }
94      if(maxIdx != index) {
95          swap(&data[maxIdx], &data[index]);
96          adjust(data, len, maxIdx);
97      }
98  }
```

主要就是保证每一个父节点都大于子节点就行。

调整堆的思路：子节点必须在排序范围内做调整，不能把排序排好的给弄了。还有就是逐个往上找把路过的都排一排，不路过的就没事了。

- 排序

```
100 void heapsort(DataType data[], SizeType size)
101 {
102     // 构建大根堆（从最后一个非叶子节点向上）
103     for(SizeType i=size/2 - 1; i >= 0; i--) {
104         adjust(data, size, i);
105     }
106
107     // 调整大根堆
108     for(SizeType i = size - 1; i >= 1; i--) {
109         swap(&data[0], &data[i]);           // 将当前最大的放置到数组末尾
110         adjust(data, i, 0);                 // 将未完成排序的部分继续进行堆排序
111     }
112 }
```

就很简单一个排序，调整完之后把最大的放在底端，然后更新底端 index。更新到底端为开头那就更完了。

2.6 数据生成

首先大体思路为测试乱序，正序，倒序。此外还有数据量的测试。乱序正序倒序对标的是大体的算法性能（最好最差平均）；数据量则更多体现在稳定排序和不稳定排序的差别上。

所以我使用了自动生成，能够生成不同规模和不同排序属性的数据集。

```
17      // get parameters
18      int upper, size, type;
19      std::string name;
20      std::cout << "Please input the type of data: (1 for ascending, 2 for descending, else for disorder)" << std::endl;
21      std::cin >> type;
22      std::cout << "Please input the upper of data:" << std::endl;
23      std::cin >> upper;
24      std::cout << "Please input the size of data:" << std::endl;
25      std::cin >> size;
26      std::cout << "Please input the name of file:" << std::endl;
27      std::cin >> name;
```

- 进行当前 path 的读取，以便生成文件
- 进行数据规模，数据 random，文件名称的读取（从 io）
- 生成数据，写入文件：

1. 生成文件路径并打开文件：

```
path.append("\\"+name+".txt");
std::cout << path << std::endl;
std::ofstream fout(path);
```

2. 写入 dataSize 和 data 并关闭文件：

```
fout << size << "\n";

srand((unsigned int)time(NULL));
for(int i=0; i < size; i++) {
    int randomNum=rand()%(2*upper)-upper;//产生[-upper, upper]的随机数
    fout << randomNum << " ";
}
fout.close();
```

3. 根据实验要求，数据范围都在[-1000, 1000]

命名格式：dtat_dataSize_dataRandom.txt

形如：data_10_1000.txt

```
_Works > Algorithm > Lab1 > data_10_1000.txt
1   10
2   348 -28 930 -762 979 751 767 436 -14 150
```

此外，还有正序和倒序的数据生成。这个就比较简单了。直接获得数据规模，以 1 为中心展开等差为 1 或-1 的数列。

2.7 白盒测试

快速排序，归并排序和堆排序基本普通样例都可以测到所有分支。但是快排不一样。我们需要设置一个快排的方案，必须要使得有左右交换数据。

直接手写数据集：

5

3 5 4 2 1

运行结果：

```
PS D:\_Codes\VSCode> cd "d:\_Codes\VSCode\_Works\A
Get data in 0.0027766s.
Size: 5, calculate in 8.5e-06s.
Compare: 30
Insert: 7
PS D:\_Codes\VSCode\_Works\Algorithm\Lab2\codes>
```

2.8 黑盒测试

首先是测试逻辑-乱序测试

我们这里使用乱序的 100000 容量数据进行测试。

- 快速排序

```
PS D:\_Codes\VSCode\_Works\Algorithm\La
 }
Get data in 0.0156996s.
Size: 100000, calculate in 0.0129545s.
Compare: 5139985
Insert: 721824
```

- 归并排序

```
PS D:\_Codes\VSCode\_Works\Algorithm\La
 }
Get data in 0.0149517s.
Size: 100000, calculate in 0.0205751s.
Compare: 5176152
Insert: 1692992
```

- 堆排序

```
PS D:\_Codes\VSCode> cd "d:\_Codes\VSCo
Get data in 0.0177653s.
Size: 100000, calculate in 0.0186733s.
Compare: 7895996
Swap: 1575250
```

然后是正序和倒序测试

归并和堆的算法并没有太大变化。而快速排序简直是灾难性的。

ascend：

```
PS D:\_Codes\VSCode\_Works\Algorithm
sort } ; if ($?) { .\quicksort }
Get data in 0.0249622s.
Size: 100000, calculate in 20.1779s.
Compare: 1410565402
Insert: 99999
```

descend：

```
PS D:\_Codes\VSCode\_Works\Algorithm
sort } ; if ($?) { .\quicksort }
Get data in 0.028211s.
Size: 100000, calculate in 20.8403s.
Compare: 1410515402
Insert: 149999
```

这简直就是无比离谱。显然都达到了最坏的时间复杂度。

所以快速排序虽然好，但并不是什么时候都适用的。最好的排序算法一定是先观察数据集的属性，然后进行算法的预测和选择。

三、出现问题及解决

3.1 快速排序的非递归

其实就是一个队列的思路，层级往下。以标杆元素为中心插入（左边都小，右边都大）分为两边，继续找标杆元素分为两边，直到只剩一个元素。这就是分治的思想。这是一种非常好用的方式。对于分治这种层级性的问题，每一次队列循环意味着处理了一个层级。

3.2 归并排序写的繁琐

归并排序一开始写的是用什么空间开辟什么空间。最后发现我们只需要开辟最大的就可以了。直接重复利用该空间，也不需要清零操作。

3.3 无法理解堆排序的 adjust

堆排序的 adjust 确实是一大难点。只要理解：调整过后的子节点需要调整相应的父节点。这样才会有大根堆或者小根堆。

3.4 计数问题的设计和修改代码

计数问题让我直接修改了大部分的代码。因为逻辑测试，特别是&&和||，这里两个符号计数的时候一定要拆开，所以我只能写成这样的等价形式：

```
while(A && B)   变成：

while(1) {
   if(!A) {
      count_compare++;
      …
   }
   if(!B) {
      count_compare++;
      …
   }
}
```

3.5 测试数据的设计

对于测试数据，我们秉承黑盒白盒测试的精髓。而且数据量一定不能太小。否则会有很大的偶然误差以及数据结构的开销在算法运行中占比较重，造成理解运行数量级的偏差。

四、总结

4.1 时间复杂度和空间复杂度以及推导

	平均时间复杂度	最好时间复杂度	最坏时间复杂度	空间复杂度
快速	O(nlog2n)	O(nlog2n)	O(n2)	O(log2n)
归并	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)
堆	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)

- 快速排序

  T(1) = C，n=1

  T(n) = 2*T(n/2) + n， n>1

  推导出 O(nlog2n)

我的非递归空间复杂度也是通过入队列得到的。最大情况为每个下标都入队列，但最好情况是只有一个入队列，平均情况算出来为 O(log2n)

- 归并排序

  时间复杂度：**O(nlog$_2$n)**，因为为两两分治。

  空间复杂度：**O(n)---**需要对应的暂时存储空间。

- 堆排序

总时间=建堆的时间耗费 + 堆调整(排序)的时间耗费

建堆的时间耗费：设树根处于第 1 层，该堆共有 h= log2n+1 层。建堆从第 h-1 层开始进行。只要知道了每一层的结点数（建的小堆的个数），和每建一个小堆所需的比较次数，就可以求得建堆的时间耗费。

第 i 层上小堆的高度 = h-i+1

建第 i 层上每个小堆最多所需的比较次数 = 2×(h-i)

建堆的时间耗费:

$$S(n) <= \sum_{i=h-1}^{1} 2^{i-1} \times (h-2) = \sum_{j=1}^{h-1} 2^{h-j} \times j$$

则建堆的时间复杂性为 <= 4n =O(n).

最坏情况 O(nlog2n)：建初始堆时，比较次数<=4n，反复调整堆时，比较次数<nlog2n。

4.2 稳定性

稳定性的意义

1、如果只是简单的进行数字的排序，那么稳定性将毫无意义。

2、如果排序的内容仅仅是一个复杂对象的某一个数字属性，那么稳定性依旧将毫无意义

3、如果要排序的内容是一个复杂对象的多个数字属性，但是其原本的初始顺序毫无意义，那么稳定性依旧将毫无意义。

4、除非要排序的内容是一个复杂对象的多个数字属性，且其原本的初始顺序存在意义，那么我们需要在二次排序的基础上保持原有排序的意义，才需要使用到稳定性的算法，例如要排序的内容是一组原本按照价格高低排序的对象，如今需要按照销量高低排序，使用稳定性算法，可以使得想同销量的对象依旧保持着价格高低的排序展现，只有销量不同的才会重新排序。

快速排序：不稳定

归并排序：稳定

堆排序：不稳定

4.3 理解与思考

实际上这些算法都是分治算法的体现。分治则天生与递归挂钩。所以我对快速排序和归并排序采用了非递归的写法，手动模拟入栈，才能更深刻的理解这个过程。而且老师所说的黑盒白盒测试以及数据集要求，我也深受启发。