算法分析与设计实验报告

实验题目：三种排序算法的设计与分析

姓名：陈俊卉

学号：2020212256

日期：2022.10.16

一、实验环境

(列出实验的操作环境，如操作系统，编程语言版本等，更多信息可视各自实际情况填写)
① 操作系统：windows 10
② 编程语言：c++
③ 编程工具：vscode 及其组件

二、实验内容

具体要求请参照实验指导书中各实验的“实验内容及要求”部分。
(示例：1.描述你对实验算法的设计思路;2.给出算法关键部分的说明以及代码实现截图；3.对测试数据、测试程序(没有要求则非必须)进行说明，如测试覆盖程度，最好最坏平均三种情况等等，并给出测试结果截图等信息)

1.算法的设计与实现

(1) 堆排序

① 实验代码：

```
void maxHeap(int* arr, int start, int end){
    // 记录当前节点
    int current = start;
    // 记录当前节点值
    int tmp = arr[current];
    // 找左右节点，交换后 current 指向被交换的子节点
    for (int i = 2 * start + 1; i<= end; i = 2 * i + 1){
        // 取较大的孩子节点
        if (i < end && arr[i] < arr[i + 1]){
            i++;
        }
        // 若子节点较大则交换
        if(tmp < arr[i]){
            arr[current] = arr[i];
            arr[i] = tmp;
            current = i;
        }
        else{
            break;
        }
```

```
    }
}
void heapSort(int* arr, int len){
    // 从后往前遍历所有非叶子节点,得到大根堆
    for (int i = len / 2 - 1; i >= 0; i--){
        maxHeap(arr, i, len-1);
    }
    // 对第一个元素（当前最大值）与大根堆最后一个元素进行交换，后将其排除出序列后继续构建大根堆
    for (int i = len - 1; i > 0; i--){
        int swap = arr[0];
        arr[0] = arr[i];
        arr[i] = swap;

        // 交换完只有最顶上的元素是不符合大根堆的，所以只需要 maxHeap(arr, 0, i-1)
        maxHeap(arr, 0, i-1);
    }
}
```

② 原理：

1. 本质上是将线性的数组根据下标构成一个二叉树，并通过不断构造大根堆来实现从小到大（大根堆）的排序。
2. 先**从后往前**遍历所有的非叶子节点，将较大的元素从二叉树的下方往上方移动，得到大根堆。**从后往前的原因是保证移动是自下往上进行的**。（最后一个非叶子节点的下标的计算公式为**数组长度/2-1**）
3. 在将较大的元素从下往上移动的时候【即 maxHeap()函数】，注意要先让叶子节点相比较，确定较大叶子节点的下标，再让父亲节点与叶子节点相比较、交换。
4. 在一轮大根堆排序结束后，数组的第一个值就为最大值。此时我们需要将最大值与**数组参与大根堆的最后一个值**交换，并将最大值排除出下一次大根堆构建的范围。

注意：此时除了栈顶刚刚交换的元素外，其余层均已经满足大根堆的要求（没有动过），所以只需要对栈顶进行 maxHeap(arr, 0, i-1)即可。

平均时间复杂度：O(nlogn)
最佳时间复杂度：O(nlogn)
最差时间复杂度：O(nlogn)
稳定性：不稳定

(2) 归并排序

① 实验代码：

```
void mergeSort(int* arr, int left, int mid, int right){
    // 定义两个数组的指针和 temp 数组及其指针
```

```
    int p1 = left;
    int p2 = mid + 1;
    int p = 0;
    int* tmp = new int[right - left + 1];

    // 比大小
    while (p1 <= mid && p2 <= right){
        if (arr[p1] <= arr[p2]){
            tmp[p] = arr[p1];
            p++;
            p1++;
        }
        else{
            tmp[p] = arr[p2];
            p++;
            p2++;
        }
    }

    // 若数组还有剩下的
    while (p1 <= mid){
        tmp[p] = arr[p1];
        p++;
        p1++;
    }
    while (p2 <= right){
        tmp[p] = arr[p2];
        p++;
        p2++;
    }

    // 赋值到原数组
    for (int i = 0;i < right - left + 1; i++){
        arr[left + i] = tmp[i];
    }

}
void mergeSortWrapper(int* arr, int left, int right){
    // 停止条件
    if (left >= right){
        return;
    }

    // long long:防溢出
```

```
    long long int mid = (left + right) / 2;
    mergeSortWrapper(arr, left, mid);
    mergeSortWrapper(arr, mid + 1, right);
    mergeSort(arr, left, mid , right);
}
```

② 原理：

1. 归并排序分为拆分和合并两部分。拆分利用递归分治减小数组规模，合并通过一个额外的数组空间对两个**已经排序完成的子数组**进行排序。
2. mergeSortWrapper()中的**停止条件为什么是“>=”而不是“=”**:右递归的左边界是 **mid+1**，mid 是整除得到的，当数组长度为 1 时，mid 为 0，此时右递归的左边界为 1，右边界为 0.
3. 先进行左右数组各自的排序，最后对左右数组进行排序：

```
mergeSortWrapper(arr, left, mid);
mergeSortWrapper(arr, mid + 1, right);
mergeSort(arr, left, mid , right);
```

4. 对左右数组进行排序【mergeSort()】:先建立左右数组的指针 p1，p2 和临时存放结果的 tmp 数组的指针 p.对左右数组指针所指的元素进行比较，并赋值在 tmp 数组指针的对应位置。经过一轮后，左右数组中可能有其中一组的指针还没有走到尾部（后面的数都是较大的数），所以需要将这些剩下的数也放进临时数组内，此时临时数组的排序就是正确的。最后将临时数组的数据原数组的对应位置上。

平均时间复杂度：O(nlogn)
最佳时间复杂度：O(nlogn)
最差时间复杂度：O(nlogn)
稳定性：稳定

(3) 快速排序

① 实验代码：

```
// 一遍快速排序
int partition(int* arr, int left, int right){
    // 选择基准
    int pivot = arr[left];
    while (left < right){
        // 先从右往左
        while (left < right && arr[right] >= pivot){
            right--;
        }
        // 找到右边小于 pivot 的值，赋值到左边的 left 位置处
```

```
        arr[left] = arr[right];

        // 再从左往右
        while (left < right && arr[left] <= pivot){
            left++;
        }
        // 找到左边大于 pivot 的值，复制到右边的 right 位置处
        arr[right] = arr[left];
    }

    // left 与 right 重合，将 pivot 赋值到此位置
    arr[left] = pivot;

    // 返回 pivot 元素的索引
    return left;
}
void quickSort(int* arr, int left, int right){
    if (left < right) {
        int mid = partition(arr, left, right);
        quickSort(arr, left, mid-1);
        quickSort(arr, mid+1, right);
    }
}
```

② 原理：

1. 每次选择数组第一个值作为基准，左指针指向第一个元素，右指针指向最后一个元素。首先右指针从右往左找比基准值小的元素，找到后赋值到左方指针；然后左指针从左往右找比基准值大的元素，找到后复制到右方指针……然后左指针与右指针将重合，将基准值赋值到此位置，则基准值左边的都为比基准值小的值，基准值右边的都为比基准值大的值。
2. 基准值的下标用 mid 表示。随后对基准值左边和右方的子数组分别继续进行上述步骤，最后将得到从小到大的有序数组。

平均时间复杂度：O(nlogn)
最佳时间复杂度：O(nlogn)
最差时间复杂度：O(n^2)
稳定性：不稳定

2. 算法测试

本次测试使用三种规模的数据集：1000、10000、100000。其中随机生成的数据大小范围为[-100,100].

考虑三种排序的最坏情况和最好情况：

堆排序：

最坏情况：根据大根堆的构造方式我们可以得知，二叉树下方越多数值大的数，构造大根堆所需要的移动比较次数越多。简单推理可以知道堆排序的最坏情况为顺序数据集。

最好情况：显然，最好情况为逆序数据集。

归并排序：

最坏情况：考虑到无论如何，归并排序的赋值次数相等，我们对比较次数进行考虑。可以得知，若合并时第一轮进行比较后只剩下一个数没有被放进临时数组中，则比较次数是最多的，为 n-1 次（n 为当前临时数组的长度），这就是最坏情况。一个实例为，对 1，5，3，7，2，6，4，8 进行归并排序，递归返回的每一组合并都需要最多次数的比较。最坏情况分法为：递归地对每一个数组分为下标为奇数的数组和下标为偶数的数组。

最好情况：同样显然，最好情况【的其中一种】为顺序数据集。

快速排序（基准值取数组第一个数）：

最坏情况：若数据为顺序或逆序，则快速排序的分治将失效，算法时间复杂度退化为 O(n^2)。这是最坏情况（每一次基准值都为最大或最小值）。

最好情况：每一次基准值都将数组均匀分为两半。（评分没要求，但尝试了一下，没构造出来，大概是一个顺序数组的二叉树中序遍历，但快排一次排序会交换位置，所以不知道怎么处理）

故我们需要构造的数据集种类为：

① 顺序、逆序数据集
② 随机数据集（模拟平均情况）
③ 归并排序的最坏情况的数据集

(1) 移动、比较次数

注意：比较次数只考虑数组元素本身的比较次数，下标的比较次数不统计在内。

移动次数这里定义为：若所赋的值（右值）是数组内的数，则算作一次移动（赋值给中间变量也算一次移动）

① 堆排序

1000 个数据、顺序（最坏情况）：

```
way:heapSort
compare_count:17626
assign_count:21914
```

1000 个数据、逆序（最好情况）：

```
way:heapSort
compare_count:15982
assign_count:19130
```

1000 个数据、随机生成（平均情况）：

```
way:heapSort
compare_count:16896
assign_count:20696
```

10000 个数据、顺序（最坏情况）：

```
way:heapSort
compare_count:244576
assign_count:288910
```

10000 个数据、逆序（最好情况）：

```
way:heapSort
compare_count:226720
assign_count:258390
```

10000 个数据、随机生成（平均情况）：

```
way:heapSort
compare_count:235000
assign_count:272806
```

100000 个数据、顺序（最坏情况）：

```
way:heapSort
compare_count:3112882
assign_count:3551706
```

100000 个数据、逆序（最好情况）：

```
way:heapSort
compare_count:2926754
assign_count:3244866
```

100000 个数据、随机生成（平均情况）：

```
way:heapSort
compare_count:3012026
assign_count:3387824
```

② 归并排序

1000 个数据、顺序（最好情况）：

```
way:mergeSort
compare_count:5044
assign_count:19952
```

1000 个数据、构造的最坏情况：

```
way:mergeSort
compare_count:8977
assign_count:19952
```

1000 个数据、随机生成（平均情况）：

```
way:mergeSort
compare_count:8690
assign_count:19952
```

10000 个数据、顺序（最好情况）：

```
way:mergeSort
compare_count:69008
assign_count:267232
```

10000 个数据、构造的最坏情况：

```
way:mergeSort
compare_count:123617
assign_count:267232
```

10000 个数据、随机生成（平均情况）：

```
way:mergeSort
compare_count:120464
assign_count:267232
```

100000 个数据、顺序（最好情况）：

```
way:mergeSort
compare_count:853904
assign_count:3337856
```

100000 个数据、构造的最坏情况：

```
way:mergeSort
compare_count:1568929
assign_count:3337856
```

100000 个数据、随机生成（平均情况）：

```
way:mergeSort
compare_count:1534608
assign_count:3337856
```

③ 快速排序

1000 个数据、顺序（最坏情况）：

```
way:quickSort
compare_count:501498
assign_count:2997
```

1000 个数据、逆序（最坏情况）：

```
way:quickSort
compare_count:501498
assign_count:2997
```

1000 个数据、随机生成（平均情况）：

```
way:quickSort
compare_count:15743
assign_count:5233
```

10000 个数据、顺序（最坏情况）：

```
way:quickSort
compare_count:50014998
assign_count:29997
```

10000 个数据、逆序（最坏情况）：

```
way:quickSort
compare_count:50014998
assign_count:29997
```

10000 个数据、随机生成（平均情况）：

```
way:quickSort
compare_count:365496
assign_count:57170
```

100000 个数据、顺序（最坏情况）：爆栈！

100000 个数据、逆序（最坏情况）：爆栈！

100000 个数据、随机生成（平均情况）：

```
way:quickSort
compare_count:22196827
assign_count:580936
```

制表如下：

比较次数：

N=1000、比较次数	堆排序	归并排序	快速排序
最坏情况	17626	8977	501498
最好情况	15982	5044	/
平均情况	16896	8690	15743

N=10000、比较次数	堆排序	归并排序	快速排序
最坏情况	244576	123617	501498
最好情况	226720	69008	/
平均情况	235000	120464	15743

N=100000、比较次数	堆排序	归并排序	快速排序
最坏情况	3112882	1568929	NA
最好情况	2926754	853904	/
平均情况	3012026	1534608	22196827

移动次数：

N=1000、移动次数	堆排序	归并排序	快速排序
最坏情况	21914	19952	2997
最好情况	19130	19952	/
平均情况	20696	19952	5233

N=10000、移动次数	堆排序	归并排序	快速排序
最坏情况	288910	267232	29997
最好情况	258390	267232	/
平均情况	272806	267232	57170

N=100000、移动次数	堆排序	归并排序	快速排序
最坏情况	3551706	3337856	NA
最好情况	3244866	3337856	/
平均情况	3387824	3337856	580936

移动、比较次数总结:

Ⅰ.堆排序的最好、最坏和平均情况之间的比较移动次数相差并不大。

Ⅱ.归并排序的最好、最坏和平均情况的比较次数相差较大，**移动次数永远一致。**

Ⅲ.快速排序的最坏情况与平均情况相比比较次数相差十分多，但**移动次数反而减少**，且在**数据集较大时就会出现爆栈现象**。

Ⅳ.通过多次实验，可以总结得出：

归并排序的比较次数是最优的，一般仅为堆排序的 1/2；快速排序的比较次数最多，且最差情况为 O(n^2).堆排序较为稳定。

归并排序的移动次数在任何情况都不变; 堆排序移动次数在好情况与坏情况也相差不大。但快排是相反的: 最坏情况反而移动次数较少。但快速排序的复杂度实际上由比较次数主导。

总的来说，归并排序和堆排序的复杂度都稳定在 O(nlogn),快速排序一般情况下也为 O(nlogn)，但快速排序最差情况可退化为 O(n^2).

(2) 时间测试

前提：去掉所有移动、比较次数累加语句；在 N=100000 下测试（三种算法效率都比较高，N 太小无法看出差别）

制表如下：

N=100000 单位：s	堆排序	归并排序	快速排序
最坏情况	0.014961s	0.025931s	NA
最好情况	0.010968s	0.019946s	/
平均情况	0.013965s	0.024966s	0.053855s

(3) 算法复杂度分析

① 堆排序

初始化堆：

设元素个数为 n，则堆的高度：

$$k = ceil(\log_2{(n+1)}) \approx ceil(\log_2 n)$$

非叶子节点的个数为：

$$2^{k-1}-1$$

假设每个非叶子节点都要调整，则第 i 层的非叶子节点需要的操作次数为 k-i.

第 i 层共有2^{i-1}个节点，则第 i 层的所有节点做的操作总数为：

$$(k-i)\times 2^{i-1}$$

假设为满二叉树（这对结论不产生影响），则共有 k-1 层非叶子节点。总操作数为：

$$\sum_{i=1}^{k-1}\left((k-i)\times 2^{i-1}\right)=2^k-k+1=n-\log_2 n+1$$

即为$O(n)$

调整堆：

显然总移动数：

$$\sum_{x=1}^{n-1}\log_2 x=\log_2{(n-1)!}$$

根据斯特林公式，$\log_2{(n-1)!}\approx n\log_2 n$

即堆排序总时间复杂度稳定在$O(n\log_2 n)$

② 归并排序

设数组长度为 n，递推式为：

$$T(n)=2T\left(\frac{n}{2}\right)+n$$

代入$\frac{n}{2}$、$\frac{n}{4}$……，可得到拓展式：

$$T(n)=2^kT\left(\frac{n}{2^k}\right)+kn$$

其中 k 为堆栈最大层数（顶层为第 0 层）.易知

$$k=\log_2 n$$

代入得

$$T(n)=nT(1)+n\log_2 n$$

T(1)=0.故有：

$$T(n)=n\log_2 n$$

即归并排序总时间复杂度稳定在$O(n\log_2 n)$

③ 快速排序

在最优情况，也就是每次基准值都为数组中间值时，递推式为：

$$T(n)=2T\left(\frac{n}{2}\right)+n$$

同归并排序。所以**最优情况时间复杂度为$O(n\log_2 n)$.**

在最差情况，也就是每次基准值都为最大或最小值时，每次划分只得到比上一次划分少一个数字的数组（另一边数组长度为 0）.则比较次数为

$$\sum_{i*1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

所以**最差情况时间复杂度为$\boldsymbol{O(n^2)}$**.

三、出现问题及解决

(列出你在实验中遇到了哪些问题以及是如何解决的)

问题一：快速排序的最优情况的数据集构建。

最优情况显然是每一次选中的基准值都是数组中间值。但因为快速排序时会有元素位置的交换，斟酌数个小时过后还是无法想到如何逆向表达位置的交换。询问老师，老师也暂时未能想出。但基本上确定是中序遍历为升序的平衡二叉树的变形。

问题二：快速排序最差情况在 N=100000 时无输出。

经查证，是爆栈问题。可以将基准值调整为 left、mid、right 中的中间值以改善快速排序的效率（但这就不是最差情况了）。

四、总结

(对所实现算法的总结评价，如时间复杂度，空间复杂度，是否有能够进一步提升的空间，不同实现之间的比较，不同情况下的效率，通过实验对此算法的认识与理解等等)

通过这次试验，堆排序、归并排序以及快速排序已经深深地印在我的脑海里。我对三种排序的腾挪次数和比较次数有了深刻的理解，对三种排序的时间复杂度的推导有了清晰的认识。我觉得本次实验最有挑战性的是数据集针对各种情况的设计。这需要对算法本身有着全面而清晰的认识。