

华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析

年级：22 级

上机实践成绩：

指导教师：金澈清

姓名：郭夏辉

上机实践名称：排序算法

学号：10211900416

上机实践日期：2023 年 3 月 16 日

上机实践编号：No.3

组号：1-416

一、目的

1. 熟悉算法设计的基本思想
2. 掌握堆排序、快速排序的基本思想，并且能够分析算法性能

二、内容与设计思想

1. 编程实现堆排序和快速排序。
2. 随机生成 1000、10000、100000、1000000 个数，在不同数据规模情况下，两种算法的运行时间各是多少，画图描述不同情况下的运行时间差异。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

1. 写出堆排序、快速排序算法；
2. 分别画出各个实验结果的折线图

1.随机数生成器

为了方便后续的实验，输出的随机数文档 data.txt 的第一个数字是生成的随机数的数量。由于我的电脑上的 stdlib.h 库中 RAND_MAX 很小(仅三万多)，生成的数据在规模较大时会出现大量的重复，有失客观性，所以我生成随机数时加了一些处理。

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
using namespace std;
int s=0,t=100000000;
int main(){
    ofstream fout("data.txt");
    int n;
    srand(time(NULL));
    cin>>n;fout<<n;
    fout<<endl;
    int tmp;
    for(int i=1;i<=n;i++){
        tmp=((rand()*(rand()))%(t-s+1);
        fout<<tmp<<" ";
```

```

    }
    fout.close();
    return 0;
}

```

2.堆排序的实现以及花费时间统计

堆是一种基于完全二叉树的数据结构，可以通过数组来实现。它中任何一个节点的值或者是不大于或者不小于其父节点的值。前者叫大根堆，后者叫小根堆。

1.将待排序序列构造成为一个大顶堆(这里使用的是数组，而不是一颗二叉树)

此时整个序列的最大值就是堆顶的根节点

2.将根结点与末尾元素进行交换，此时末尾就是最大值

3.然后将剩余 $n-1$ 个元素重新构造成为一个堆，这样就会得到 n 个元素的次小值。如此反复，便能得到一个有序序列。

堆排序算法包括两个步骤：建堆和重建堆。建堆过程的时间复杂度分析根据算法导论，结果是 $O(n)$ 。而在重建堆的过程中，需要把堆的最大值和堆的最后一个结点的值进行交换，将堆的大小减一，然后向下调整堆，使之成为重新成为一个大根堆。周而复始，直到堆的大小变成 0。每次向下翻转的最大次数为 $\lg n$ ，所以这一过程的时间复杂度为 $O(n \lg n)$ 。综合考虑，可以得到堆排序的时间复杂度是 $O(n \lg n)$ ，即使在最坏的情况(向下翻转了最多层, $\lg n$ 次)也是如此。与此同时，在整个堆排序的过程中只需要对两个变量操作，所以总的空间复杂度为 $O(1)$ 。

```

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <time.h>
using namespace std;
int a[10000000+6],n;

void heapify(int a[],int i,int n){
    int tmp,tmp1;
    if((2*i<=n) && (a[2*i]>a[i]))tmp=2*i;
    else tmp=i;

    if((2*i+1<= n) && (a[2*i+1]>a[tmp]))tmp=2*i+1;

    if(tmp!=i){
        tmp1=a[i];a[i]=a[tmp];a[tmp]=tmp1;
        heapify(a,tmp,n);
    }
}

void buildheap(int a[],int n){
    for(int i=(n/2);i>=1;i--)heapify(a,i,n);
}

void heapsort(int a[],int n){
    buildheap(a,n);
    int tmp,tmp1,n=n,cnt=1;

```

```

        for(int i=n;i>=2;i--){
            int tmp=a[1];a[1]=a[i];a[i]=tmp;
            --tmplen;++cnt;
            heapify(a,1,tmplen);
        }
        return;
    }
}

int main(){
    ifstream fin("data.txt");
    ofstream fout("result.txt");
    fin>>n;for(int i=1;i<=n;i++)fin>>a[i];
    clock_t start,stop;
    start=clock();
    heapsort(a,n);
    stop=clock();
    for(int i=1;i<=n;i++)fout<<a[i]<<" ";
    cout<<"Time:"<<1000*(((double) (stop -
start)))/CLOCKS_PER_SEC)<<"ms"<<endl;
    fin.close();
    fout.close();
    return 0;
}

```

3.快速排序的实现以及花费时间统计

快速排序和堆排序类似，是一种基于比较的排序方法，也运用了分治的思想。快速排序的两个步骤是分解和归纳

分解:数组 $A[p,r]$ 被划分为为了两个子数组 $A[p...q-1]$ 和 $A[q+1...r]$ ，其中 $A[p..q-1]$ 中的每一个元素都小于 $A[q]$ ，而 $A[q+1...r]$ 中每一个元素都大于 $A[q]$

归纳:递归调用快速排序，对子数组进行排序

1.选取一个基准元素 (pivot)

2.比 pivot 小的放到 pivot 左边，比 pivot 大的放到 pivot 右边

3.对 pivot 左边的序列和右边的序列分别递归的执行步骤 1 和步骤 2

可以看到，快速排序不同于归并排序，最后不需要归并操作，因为快速排序就是在原数组之上进行操作的。

快速排序的最好情况和最坏情况之时间复杂度差异还是巨大的。在最好情况下，每一次划分都正正好将数组分成长度相等的两半，可以写出递归式

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

通过主定理可以得出时间复杂度为 $O(n \lg n)$

但是在最坏情况下，当划分的两个子问题分别包含 $n-1$ 个元素和 0 个元素的时候，可以写出递归式

$$T(n) = T(n-1) + \Theta(n)$$

通过递推可以得到时间复杂度为 $O(n^2)$

但是在平均情况时，假设任意一种划分情况出现的概率相等，然后对于所有可能的划分进行计算，求其期望可得时间复杂度为 $O(n \lg n)$

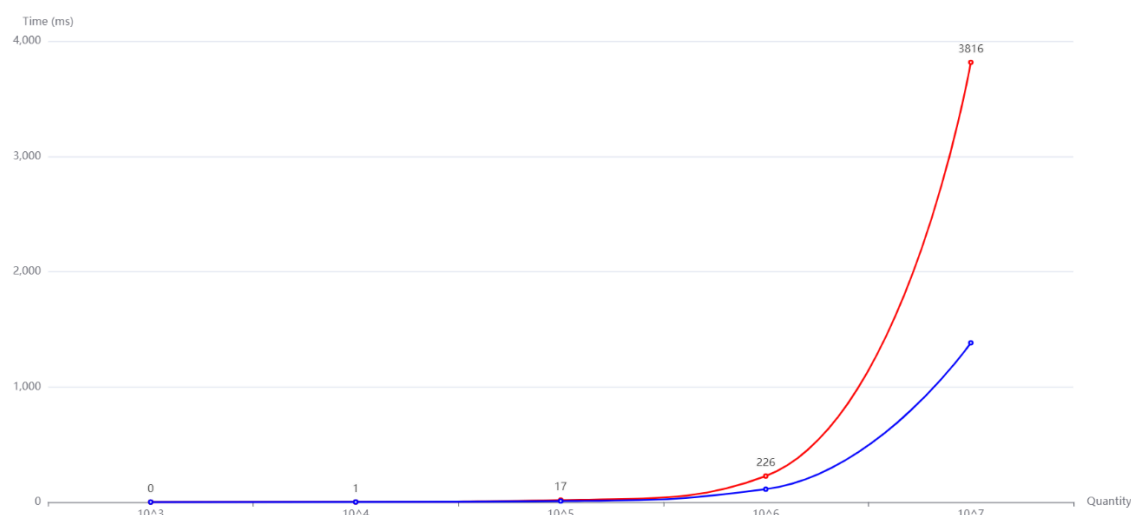
快速排序的空间复杂度和时间复杂度密切相关，因为快排的实现是递归调用的，而且每次函数调用中只使用了常数的空间，因此空间复杂度等于递归深度，最坏情况下空间复杂度为 $O(n)$ 而最好情况下空间复杂度为 $O(\lg n)$

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <time.h>
using namespace std;
int a[10000000+6],n;
int partition(int a[],int left,int right){
    int tmp=a[right];
    int i=left-1;
    int tmp1;
    for(int j=left;j<right;j++){
        if(a[j]<=tmp){
            ++i;
            tmp1=a[i];a[i]=a[j];a[j]=tmp1;
        }
    }
    tmp1=a[i+1];a[i+1]=a[right];a[right]=tmp1;
    return i+1;
}
void quicksort(int a[],int left,int right){
    if(left<right){
        int mid=partition(a,left,right);
        quicksort(a,left,mid-1);
        quicksort(a,mid+1,right);
    }
}
int main(){
    ifstream fin("data.txt");
    ofstream fout("result.txt");
    fin>>n;for(int i=0;i<n;i++)fin>>a[i];
    clock_t start,stop;
    start=clock();
    quicksort(a,0,n-1);
    stop=clock();
    for(int i=0;i<n;i++)fout<<a[i]<<" ";
    cout<<"Time:"<<1000*(((double) (stop -
start))/CLOCKS_PER_SEC)<<"ms"<<endl;
    fin.close();
```

```
fout.close();
return 0;
}
```

4.两种排序算法的对比

我生成了 10^3 到 10^7 规模的随机数据，然后分别使用两种排序算法，统计之后得到了各自的运行时间，如下图所示(图线已平滑处理)。



五、总结

对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。

快速排序的最坏时间复杂度是 $O(n^2)$ ，平均时间复杂度是 $O(n \lg n)$ ，而归并排序在最坏情况下的时间复杂度也是 $O(n \lg n)$ 。采用归并排序，确实可以在特殊的输入数据下避免最坏时间复杂度情况的发生，但是归并排序真的在平均情况下优于快速排序吗？根据实验结果，可以很轻易地发现快速排序往往比归并排序效率更高。

快速排序之所以在平均情况下优于归并排序，也是有一些原因的。首先，快速排序过程中每次访问的是临近的内存位置，有较好的空间局部性，而归并排序访问的内存位置很分散。根据存储金字塔原理，快速排序能更好地利用高速缓存，使得 CPU 能够快速访问数据。而且，快速排序是就地操作的，不需要像归并排序那样额外地创建辅助数组来保存临时值，这就节省了分配和删除辅助数组的时间。

快速排序也可以随机化地选择主元，尽力降低极端的情况发生的概率。通过这种方式，快速排序在弥补自身缺点的同时，又拉大了和堆排序之间的差异。

总而言之，这次实验的过程还算是简单的，但是通过实际的操作，我发现不能仅仅依赖时间复杂度分析一个算法在平均情况下的优劣程度，还是要结合实际才能更好地捕获它的运行情况。后来我翻阅了一些网上的资料，发现 C++、Python、Java 等主流编程语言的函数库中快速排序也是官方采取的排序方式，广泛地使用证明了其优越性。知行合一之中，希望我的算法学习之路更加稳妥。