# 算法设计与分析实验报告

# 实验一 快速排序

|  |  |
| --- | --- |
| **院系：** | **信息工程学院** |
| **班级：** | **15级网络工程2班** |
| **学号：** | **2015551621** |
| **姓名：** | **王 康** |
| **任课教师：** | **王 婷** |
| **成绩：** |  |

**湘 潭 大 学**

**2018年6月实验一 快速排序**

1. **实验内容**

运用分治递归思想，编程实现快速排序算法。通过不同实例测试，分析最坏、最好、平均情况下的时间复杂度，并设计实验程序验证分析结果。

**二．实验目的**

1、理解递归的概念及实现方法；

2、掌握分治策略的基本思想以及用分治法解决问题的一般技巧；

3、掌握求解递归算法时间复杂度的一般方法。

**三. 算法描述**

1、在待排数组a[0..n]中挑选基准元素并保存到变量flag中，如：挑选第一个元素a[low]存入flag中。

2、将被挑选出的基准位置a[low]赋值为0，表示当前位置为空，可放入数据。

3、判断：如果a[low]为0时，则从high到low方向找出一个比基准元素小的数的位置i，然后使得a[low]与a[i]的值进行交换。相反，如果a[high]为0，则从相反的方向寻找一个比基准元素大的数的位置j，使得a[high]与a[j]交换值。

4、当步骤3中i大于或等于j时，将基准元素放到空位，待排数组[low..high]则完成了一次分块，使得待排数组以基准为界前后总体有序。

5、将分得的块分别调用排序函数，递归完成排序。

6、返回运行时间（执行次数）。

改进：利用随机数挑选基准flag，然后将第一个数放到被挑选的基准位置，第一个位置空出赋值为0,。

**四. 算法实现**

1. **数据结构及函数说明**

Ⅰ、数据结构

A、待排数组为一个一位数组a[0..n]

Ⅱ、函数说明

**A、分块函数Part()**

/\*\*

\* @author Swking

\* @method Part

\* @parame

\* int a[] 待排数组

\* int low 待排数组的最低位

\* int high 待排数组的最高位

\* int &timeNum 执行次数

\* @return j 基准位置

\*/

int Part(int a[], int low, int high, int &timeNum);

说明：将待排数组a[]分块，使得前后总体有序。

1. **排序函数FastSort()**

/\*\*

\* @author Swking

\* @method FastSort

\* @parame

\* int a[] 待排数组

\* int low 待排数组的最低位

\* int high 待排数组的最高位

\* @return timeNum 执行次数

\*/

int FastSort(int a[], int low, int high, int &timeNum);

说明：利用Part()函数分块后，递归处理分块后的数组，直至完成排序。

1. **源程序代码**

#include<iostream>

#include<cstdio>

#include<ctime>

#include<time.h>

#include<random>

#include<fstream>

using namespace std;

/\*\*

\* @author Swking

\* @method Part

\* @parame

\* int a[] 待排数组

\* int low 待排数组的最低位

\* int high 待排数组的最高位

\* int &timeNum 执行次数

\* @return j 基准位置

\*/

int Part(int a[], int low, int high, int &timeNum){

int i=low,j=high;

srand(time(NULL));

int f = low + rand()%(high-low+1);//随机挑选基准数位置

//int f = low; //普通方法挑选基准，挑选第一个,用于模拟最坏情况

//int f = (low + high)/2; //用于模拟最好情况

int flag = a[f]; //选出基准数

a[f] = a[low]; //空出第一个位置

a[low] = 0;

//cout << f << " ";

for(; i<=j ;){

if(a[i]==0){ //如果开头位置空出，则要从末尾挑选比基准数小的数，空出末尾位置

if(flag >= a[j]){ //当要排的数据全部相等时，如果不交换就如同最坏情况，交换则是最好情况，每个数都是中位数

a[i] = a[j];

a[j] = 0;

i++;

}else{

j--;

}

timeNum++; //计算执行次数

if(i>=j){

break;

}

}

if(a[j]==0){ //如果末尾位置空出，则要从开头挑选比基准数大的数，空出开头位置

if(flag < a[i]){

a[j] = a[i];

a[i] = 0;

j--;

}else{

i++;

}

timeNum++; //计算执行次数

if(i>=j){

break;

}

}

}

a[j] = flag; //将基准数放回数组

//cout<< j<<endl;

return j;

}

/\*\*

\* @author Swking

\* @method FastSort

\* @parame

\* int a[] 待排数组

\* int low 待排数组的最低位

\* int high 待排数组的最高位

\* @return timeNum 执行次数

\*/

int FastSort(int a[], int low, int high, int &timeNum){

if(low < high){

int flag = Part(a, low, high, timeNum); //找基准元素

FastSort(a, low, flag-1, timeNum); //对左半部排序

FastSort(a, flag+1, high, timeNum); //对右半部排序

}

return timeNum;

}

int main()

{

fstream fio;

ofstream fout;

fout.open("out.txt",ios::out);

/\*\*

\* 模拟数据文件

\*/

fio.open("in.txt",ios::out);

cout << "producting the file..." << endl;

int eNum = 1000;//元素个数

//模拟最好情况和最坏情况，由于最好情况的数据难以模拟，所以在取基准时做模拟

fio << eNum << endl;

for(int i=1; i<=eNum; i++){

fio << i << ' ';

}

fio << endl;

//模拟普通数据

srand(time(NULL));

int exampleSum = 50 + rand()%50;

while(exampleSum--){

//int elementNum = 20 + rand()%80;

int elementNum = eNum;

fio << elementNum << endl;

while(elementNum--){

fio << rand()%1000 << ' ';

}

fio << endl;

}

fio.close();

/\*\*

\* 读取数据文件

\*/

fio.open("in.txt",ios::in);

cout << "reading the file..." << endl;

cout << "sorting..." << endl;

while(!fio.eof()){

int n = 0;

fio >> n;

if(n!=0){ //控制最后一行不多输出

int a[n];

for(int i=0; i<n; i++){

fio >> a[i];

// cout << a[i] << ' ';

}

int exec = 0;

clock\_t start = clock();

int execNum = FastSort(a, 0, n-1, exec); //排序

clock\_t ends = clock();

//输出

fout << n << endl;

for(int i=0; i<n; i++){

fout << a[i] << ' ';

}

fout << endl;

fout << "execNum:" << execNum << " time:" << (double)(ends - start)\*1000/CLOCKS\_PER\_SEC << "ms" << endl << endl;

}

}

cout << "sorted!" << endl;

fio.close();

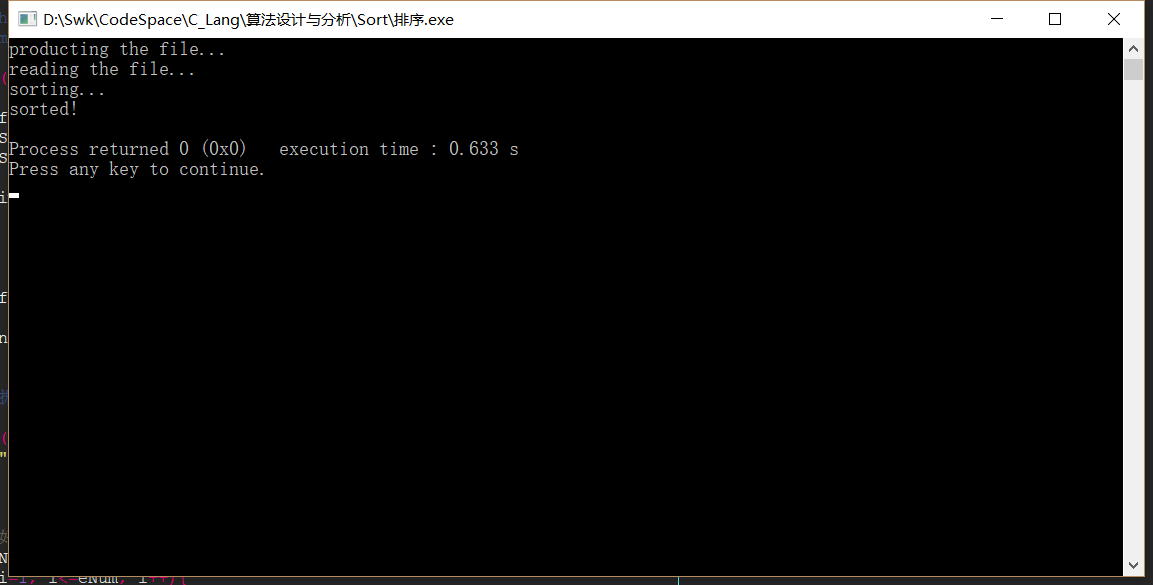
fout.close();

return 0;

}

1. **程序运行结果**

1、控制台输出信息



2、样本文件in.txt，第一组数据为了最好情况和最坏情况作对比，是手动模拟的数据，后面的数据全为随机生成。奇数行为待排数据元素的个数，偶数行为对应的元素值。



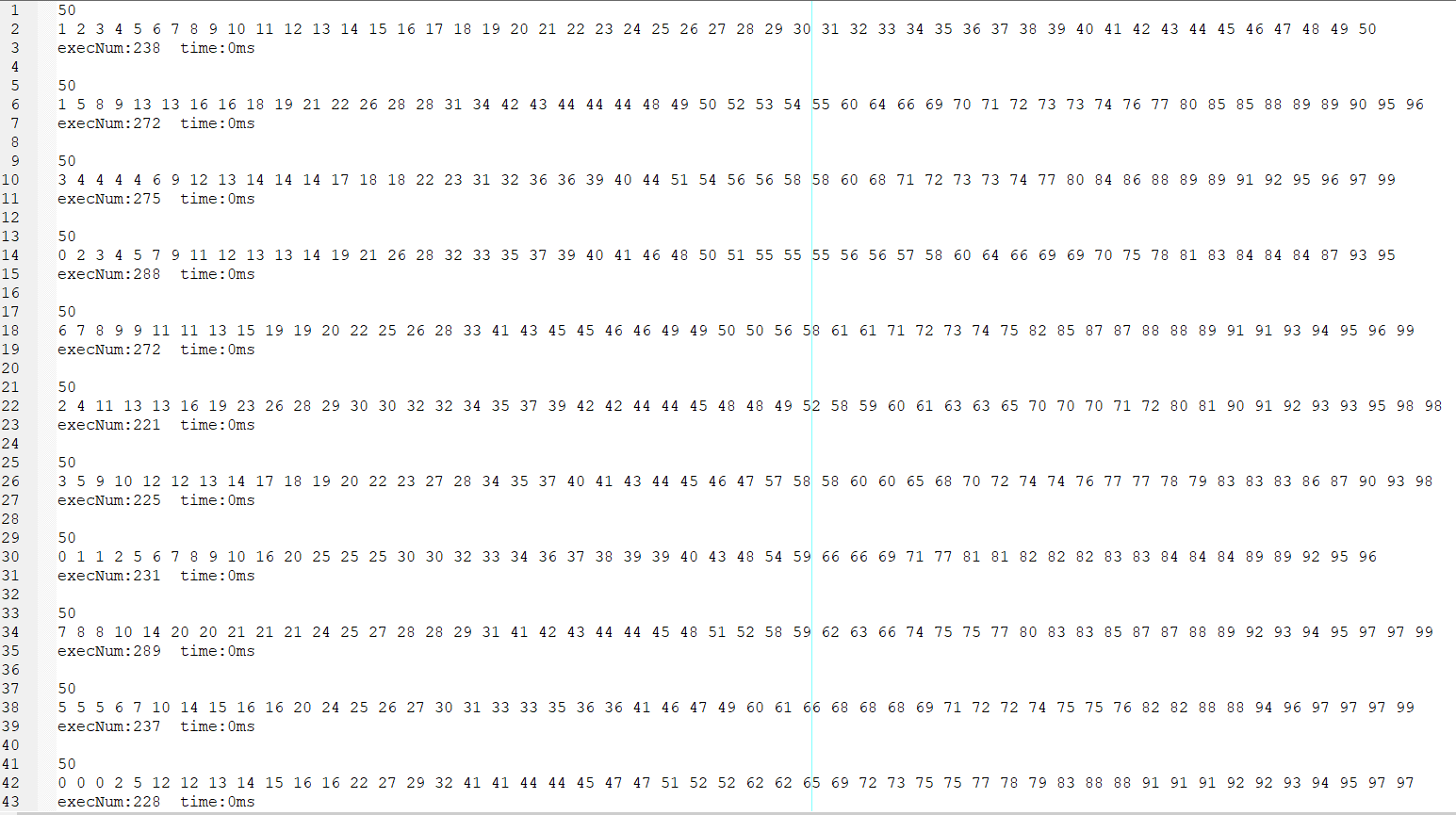
1. 用普通方法挑选基准，即挑选第一个位置做基准时，运行情况数据如下：
   1. 模拟最坏情况，即待排数据是有序的。



* 1. 模拟最好情况，每次选择的基准都将待排数组分为大小相同的两半。



1. 采用改进方法使用随机挑选基准



1. **实验结果分析**

分析算法的时间复杂度并与实验结果对比

1. 由分治法时间复杂度可知，程序最好情况是每次将待排数组分为相同的两块，且最后合并时间为O(n)，则有

T(n) 好= 2T(n/2) + O(n)=O(nlog2n)

1. 最坏情况是每次只将待排数组分为两块：一块只有一个元素，另一块包含剩下的元素，则有

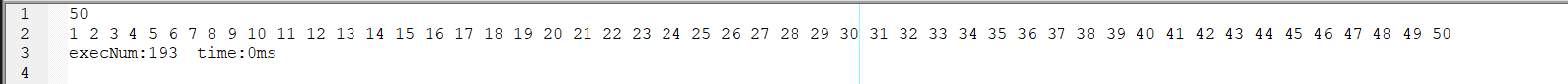
T(n)坏 = 2T(n-1) + O(n)=O(n2)

1. 以随机方法挑选基准可避免分块极度不均的情况，有

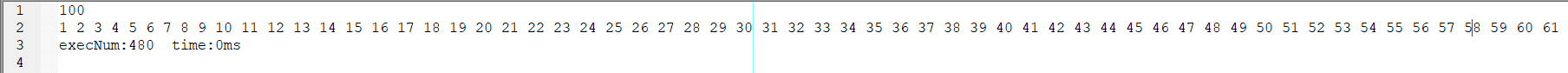
T(n) 好≦T(n)均≦T(n)坏

1. 以普通方法挑选基准的比较
   1. 最好情况

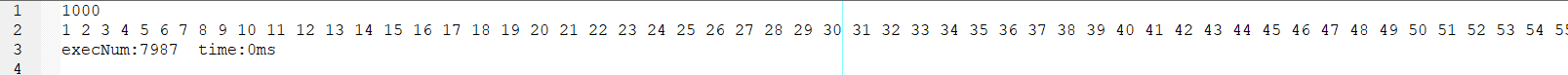
共50个元素，最好情况执行次数为193次，约小于50log250≈250



共100个元素，最好情况执行次数为480次，约小于100log2100≈600

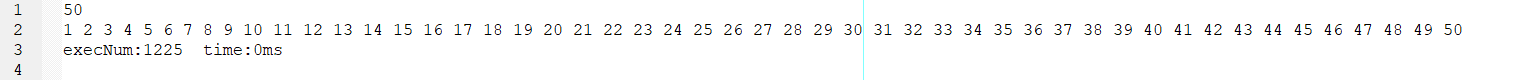


共1000个元素，最好情况执行次数为7987次，约小于1000log21000≈10000

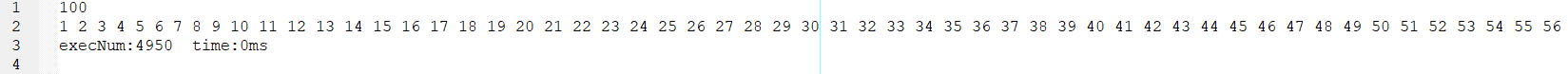


* 1. 最坏情况

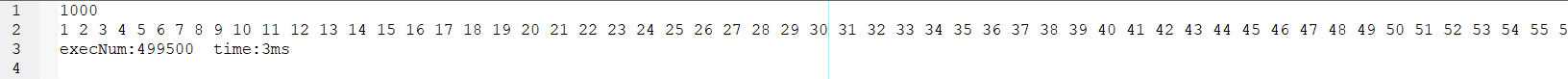
共50个元素，最坏情况执行次数为1225次，约小于502/2=1259



共100个元素，最坏情况执行次数为4950次，约小于1002/2=5000



共1000个元素，最坏情况执行次数为499500次，约小于10002/2=500000

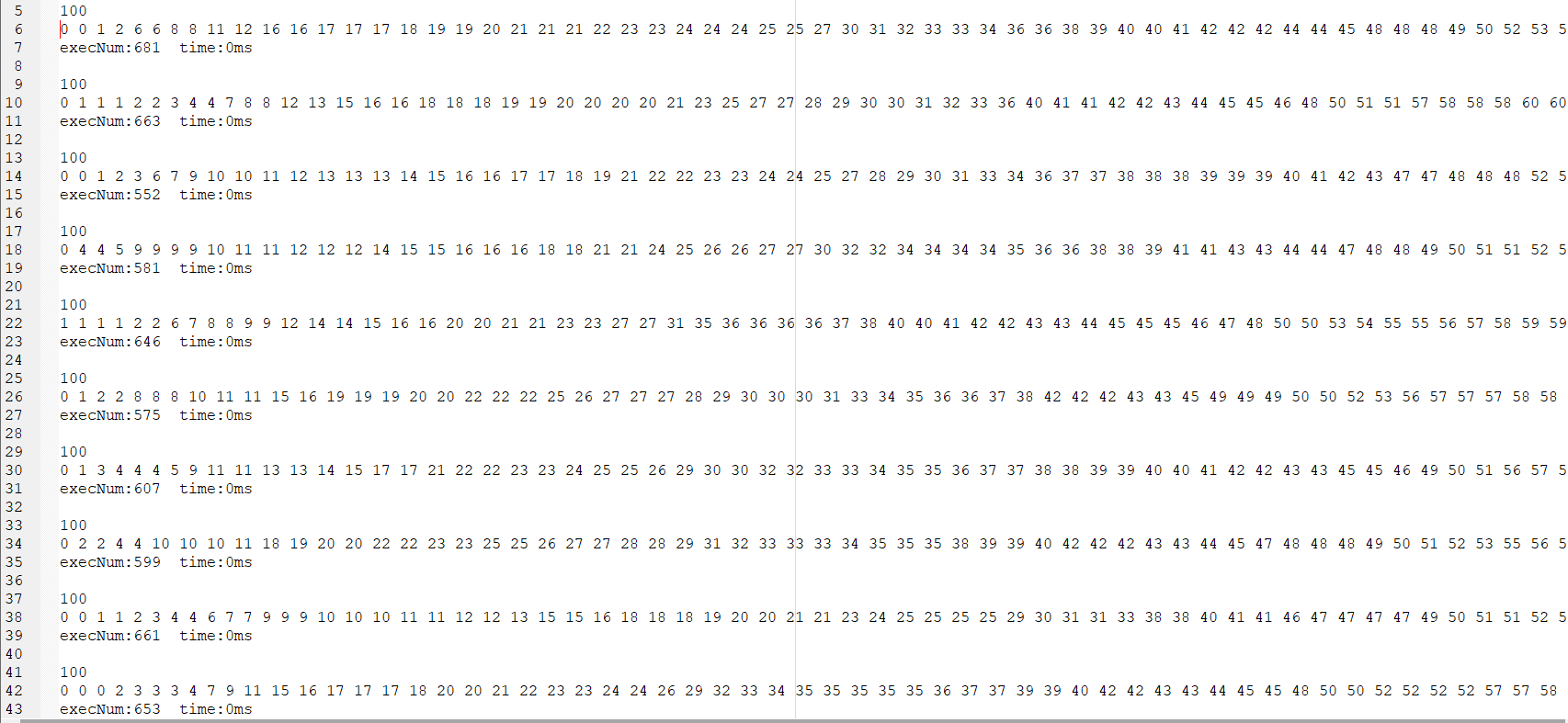


* 1. 平均情况

共50个元素，平均情况执行次数在200~300次



共100个元素，平均情况执行次数为500~700次



共1000个元素，最坏情况执行次数为10000~12000次



1. 以随机的方法挑选基准，相当于普通方法的平均情况

共50个元素，平均情况执行次数在250次左右



共100个元素，平均情况执行次数为600次左右



共1000个元素，平均情况执行次数为10000次左右



1. **结论**
2. 有上述对实验的结果的分析可以得出，快速排序的时间复杂度的影响因素：
   1. 第一个是以何种方式挑选基准，
      1. 用普通方法挑选基准，运行的时间则取决于待排数组的原有顺序，若待排数组本来就是有序时，则该方法效率会达到最低。
      2. 用随机方法挑选基准时，可避免待排数组原有顺序带来的问题，挑选随机化可达到算法的平均时间。
   2. 第二个是待排数组元素的状态：
      1. 当待排数组是整体有序时，若采用普通方法挑选基准，那将成为最坏情况，每次分块极度不均（一块元素为1个，一块为剩下的），若采用随机挑选基准则打破了待排数组的原有顺序，避免出现分块不均的情况。
      2. 当待排数组无序时，采用普通方法和随机方法的时间复杂度相差不大。
3. 针对影响因素改进算法
   1. 改变挑选基准的方式，上面已经有提到过了。
   2. 打乱待排数组原有的顺序
      1. 利用舍伍德算法对待排数组进行洗牌