

算法设计实验报告



实验名称：递归与分治策略综合实验

姓名：蒋雪枫

班级：2017211314

学号：2017213508

专业：网络工程

指导教师：叶文老师

2019 年 10 月 16 日

一、综述

递归，即间接或直接调用自身的思想；分治，即分而治之，通过减小问题的规模来提高我们求解问题的效率与时间。在学生这两年的计算机方向的学习中，深深感受到这两种算法思想的重要性。同学们不论是在参加各种算法比赛，还是在假期去企业实习(当然大部分是刷题准备面试的时候)，都写过不少相关的代码，虽然或许并没有有意去深入了解其背后的思想。本月中，我们跟随叶文老师的教学，通过一些典型的案例深入研究其算法思想。在本次实验中，我们完成了快速排序(以及其随机化处理)，归并排序(以及其非递归实现)，线性时间选择以及平面最近点对的算法，并通过 LTE 基站的较为工程化的数据来辅助验证了算法的正确性。

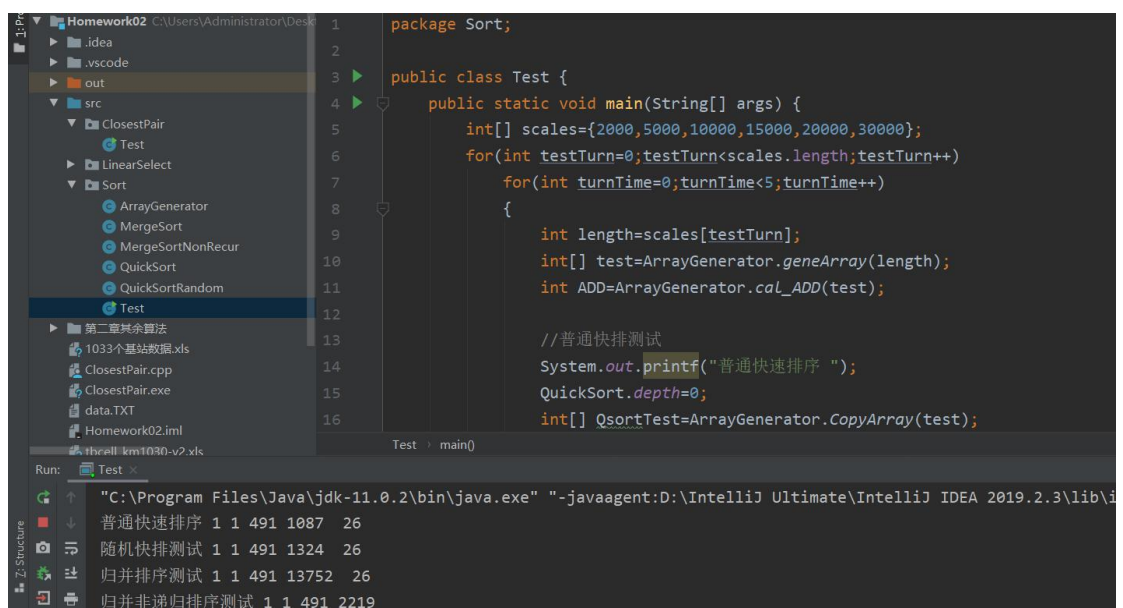
比起代码本身，算法的思想才是最为重要的，所以报告中未贴出全部代码，只贴出了核心算法部分，源代码见附件。

二、文件说明

本次作业主要使用 Java 完成，最近点对算法使用 C++ 算法完成。在主文件目录下，src 目录下是 Sort package(内有四种算法以及计算其运行效率的测试类 Test，其余的 Java 包同理)、LinearSelect package 以及 ClosestPair package。主文件目录还有 ClosestPair.cpp 以及其对应的可执行文件，使用 C++11 编写。其余是一些算法所需要的文件资源。另外，学生在这里也另外实现了第二章所讲的其他算法，在另外的附录中，兴趣使然：)

三、四种排序算法运行以及结果说明

首先，在我们的测试类中，我们生成了不同长度的数据，一共有 6 种长度，并且每一组长度我们都测试了五组样本。拿第一行举例子，1-1 为外部测试的轮数与样本序号，491 为该样本的 Average Distinct Degree，1087 为运行时间，单位为微妙，26 为递归最深层次。排序复杂度使用 $O(n^2)$ 方法遍历统计，递归深度通过设置全局变量实现，并且在调用自身时候传递参数 $k+1$ ，用于记录当前深度情况。



The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure includes a 'src' directory with a 'Sort' package containing 'ArrayGenerator', 'MergeSort', 'MergeSortNonRecur', 'QuickSort', 'QuickSortRandom', and 'Test'. The 'Test' class is selected, and its code is displayed in the editor. The code defines a 'Test' class with a 'main' method that tests various sorting algorithms. The execution results are shown in the bottom console window.

```
package Sort;

public class Test {
    public static void main(String[] args) {
        int[] scales={2000,5000,10000,15000,20000,30000};
        for(int testTurn=0;testTurn<scales.length;testTurn++)
            for(int turnTime=0;turnTime<5;turnTime++)
            {
                int length=scales[testTurn];
                int[] test=ArrayGenerator.geneArray(length);
                int ADD=ArrayGenerator.cal_ADD(test);

                //普通快排测试
                System.out.printf("普通快速排序 ");
                QuickSort.depth=0;
                int[] QsortTest=ArrayGenerator.CopyArray(test);
```

Run: Test

```
"C:\Program Files\Java\jdk-11.0.2\bin\java.exe" "-javaagent:D:\IntelliJ Ultimate\IntelliJ IDEA 2019.2.3\lib\i
普通快速排序 1 1 491 1087 26
随机快排测试 1 1 491 1324 26
归并排序测试 1 1 491 13752 26
归并非递归排序测试 1 1 491 2219
```

对于排序算法的运行时间详情，均已经把数据连接到文件夹当中，可以进行查看。这里截图

仅为部分，但基本服从整体的分布。

74	普通快速排序	4	4	3747	2221	31
75	随机快排测试	4	4	3747	2443	31
76	归并排序测试	4	4	3747	103750	31
77	归并非递归排序测试	4	4	3747	3046	
78	普通快速排序	4	5	3743	1475	31
79	随机快排测试	4	5	3743	2063	31
80	归并排序测试	4	5	3743	85365	31
81	归并非递归排序测试	4	5	3743	1836	
82	普通快速排序	5	1	4979	3384	35
83	随机快排测试	5	1	4979	3839	35
84	归并排序测试	5	1	4979	208215	35
85	归并非递归排序测试	5	1	4979	4127	
86	普通快速排序	5	2	4955	3486	35
87	随机快排测试	5	2	4955	3895	35
88	归并排序测试	5	2	4955	213022	35
89	归并非递归排序测试	5	2	4955	3874	
90	普通快速排序	5	3	5012	3667	31
91	随机快排测试	5	3	5012	3860	31
92	归并排序测试	5	3	5012	205145	31

在学生自己编写的代码中，测试发现递归式的归并排序时间较长，再咨询了其余同学后，得知他们也遇到了同样的情况，我将他们用 Java 实现的代码也丢进了我的测试类，发现时间都很长，比较新奇，至于原因，我们之后在总结部分再谈。而其余算法都差不多，甚至觉得随机优化的快速排序甚至整体上比不过最初的快速排序，原因我们也在后面的实验总结来谈。

这里附上这四种排序的核心算法：

```
public class MergeSort {
    public static int depth=0;
    public static void mergesort(int []array,int left,int right,int k)
    {
        if(k>=depth)
            depth=k;
        if(left<right)
        {
            int mid=(left+right)/2;
            mergesort(array, left, mid, k+1);
            mergesort(array, left: mid+1, right, k: k+1);
            merge(array,left,mid,right);
        }
    }
}
```

归并排序

```

public static void MergePass(int[] array, int gap, int length) {
    int i = 0;
    // 归并gap长度的两个相邻子表
    for (i = 0; i + 2 * gap - 1 < length; i = i + 2 * gap) {
        Merge(array, i, mid: i + gap - 1, high: i + 2 * gap - 1);
    }
    // 余下两个子表, 后者长度小于gap
    if (i + gap - 1 < length) {
        Merge(array, i, mid: i + gap - 1, high: length - 1);
    }
}

```

非递归的归并排序中 MergePass

```

public class QuickSort {
    // public static int count=0;
    public static int depth=0;
    public static void qsort(int p,int r,int[] a,int k)
    {
        // count+=1;
        if(k>=depth)
            depth=k;
        if(p<r)
        {
            int q=partition(p,r,a);
            qsort(p, r: q-1,a, k: k+1);
            qsort( p: q+1, r,a, k: k+1);
        }
    }
}

```

核心代码为四行的快速排序(其核心是划分函数)

```

public class QuickSortRandom {
    static Random random=new Random();
    public static int depth=0;
    public static void randomizedQuickSort(int p,int r,int[] a,int k)
    {
        if(k>=depth)
            depth=k;
        if(p<r)
        {
            int q=randomizedPartition(p,r,a);
            randomizedQuickSort(p, r: q-1,a, k: k+1);
            randomizedQuickSort( p: q+1, r,a, k: k+1);
        }
    }
}

```

随机优化的快速排序

四、线性时间选择以及最近点对问题

对于本问题，让我学习到的最重要的一个是算法本身的思想，一个是 Java 与 Excel 读写的交互。在实现时，学生采用了快排思想和书上线性时间选择求 MinK 的两个实现，结果一致，时间无疑是 $O(n)$ 优于 $O(n^2)$ 。

有一说一，线性时间选择算法将数学和算法结合得非常巧妙，要是没有学这一章，估计这辈子都想不到。同时，主定理也很重要，只是课上老师没怎么强调。当然，其实借助快排思想也挺快的，只是没有优化到 $O(n)$ 。另一方面，我们可以使用最大最小堆（Python 内的 `heapq` 可以很快实现这一要求）来解决 MinK 问题，当然它用得最多的情况还是滑动窗口。

```
public static void main(String[] args) throws IOException, BiffException {
    Workbook workbook = Workbook.getWorkbook(new File("1033个基站数据.xls"));
    Sheet sheet = workbook.getSheet(0);
    int targetRow = sheet.getRows();
    float[] Distances = new float[targetRow - 1];
    int targetCol = sheet.getColumns();
    for (int i = 1; i < targetRow; i++) // 开头是label, dont care
    {
        Cell cell = sheet.getCell(i, targetCol - 1);
        Distances[i - 1] = Float.parseFloat(cell.getContents());
    }
    Scanner in = new Scanner(System.in);
    System.out.println("想查询第几名? k=? ");
    int k = in.nextInt();
    float ans = LinearSelect.RandomizedSelect(Distances, 0, Distances.length - 1, k);
    System.out.printf("使用快排思想, 您查询的是第%d名(从小到大), 为%f\n", k, ans);
    float ans2 = LinearSelect.select(Distances, 0, Distances.length - 1, k);
    System.out.printf("使用线性时间选择思想, 您查询的是第%d名(从小到大), 为%f\n", k, ans);
}
```

```
// 线性时间选择算法
public static float select(float[] arr, int p, int r, int k)
{
    if (r - p < 5)
    {
        Arrays.sort(arr, p, r);
        return arr[p + k - 1];
    }

    for (int i = 0; i <= (r - p - 4) / 5; i++)
    {
        int s = p + 5 * i, t = s + 4;
        for (int j = 0; j < 3; j++)
            Arrays.sort(arr, s, t - j);
        swap(arr, i, p + i, j, s + 2);
    }

    float x = select(arr, p, p + (r - p - 4) / 5, k * (r - p + 6) / 10);
    int i = Partition(arr, p, r), j = i - p + 1;
    if (k <= j) return select(arr, p, i, k);
    else return select(arr, p, i + 1, k - j);
}
```


当我们查询 $k=5, 50, 1$ 的时候，结果分别如下：

想查询第几名？ $k=?$

1

使用快排思想，您查询的是第1名(从小到大)，为103.074997

使用线性时间选择思想，您查询的是第1名(从小到大)，为103.074997

想查询第几名？ $k=?$

5

使用快排思想，您查询的是第5名(从小到大)，为126.096001

使用线性时间选择思想，您查询的是第5名(从小到大)，为126.096001

想查询第几名？ $k=?$

50

使用快排思想，您查询的是第50名(从小到大)，为208.475006

使用线性时间选择思想，您查询的是第50名(从小到大)，为208.475006

随后是最近点对问题求解：

```
@ public static Pair cPair2(Point1[] x) {  
    if (x.length < 2) return null;  
  
    //依x进行排序  
    mergeSort(x);  
  
    Point2[] y = new Point2[x.length];  
    for (int i = 0; i < x.length; i++)  
        y[i] = new Point2(x[i].x, x[i].y, i);  
    //依y进行排序  
    mergeSort(y);  
  
    Point2[] z = new Point2[x.length];  
  
    return closedPair(x, y, z, 0, x.length - 1);  
}
```

而最近点对问题也是递归思想的一次升华，当我们最开始开始读到分为两个区域求出 d ，再在横坐标的 $X_{mid}-d, X_{mid}+d$ 区间内继续寻优，考虑到该区域可能集中了所有点，所以我们进一步优化，利用鸽巢原理，筛选周围的点。之后的学术论文中，甚至提出了可以继续优化的想法，这种想法是“画圆”，但其意义也没有“鸽巢原理”思想具有重大意义了。

使用 C++11 实现的最后版本的最近点对，结果如下：

```
C:\Users\Administrator\Desktop\Algorithm\Algorithm Practica\Homework02\ClosestPair.exe
the cloest points are :
idl:566803 , id2:567389
< (102. 741, 25. 05394) , (102. 741, 25. 053888) >
distance : 5.7886135835m
remove the first point of ans, the candidate ans is :
idl:566784 , id2:567222
< (102. 791, 25. 03979) , (102. 791, 25. 039722) >
distance : 7.5697252468m
remove the second point of ans, the candidate ans is :
idl:566784 , id2:567222
< (102. 791, 25. 03979) , (102. 791, 25. 039722) >
distance : 7.5697252468m
the second cloest points are :
idl:566784 , id2:567222
< (102. 791, 25. 03979) , (102. 791, 25. 039722) >
distance : 7.5697252468m
```

五、问题分析与思考

1. 为什么随机优化的快速排序会整体显得更慢？学生认为，这里是在我们排序的时候都需要传参，Java 是以数组整体进行传参，而我们在外面调用了又一层函数，可能就耽误了很多时间。可以进行代码优化。

2. 为什么有时候刚开始排序的时候，第一轮第一组时间会久一些？学生认为是生成 Random 类并第一次调用的时候比较费时间，并且另外编写了用于验证该想法的测试脚本也佐证了这一猜想。所以在最后分析的时候，可以不用参考第一次调用 Random 的第一个数组长度的第一轮第一个样本。

3. 为什么最后用 C++ 实现的最近点对？老实说，用 Java 没有 De 出 Bug 在哪里，只知道是在递归调用 closepair 方法的时候爆栈了 (throws StackOverflow Exception)。经过思考，学生认为，这并不是代码的问题，而就是爆栈了，因为传参都是开的对象数组。或许可以通过 JVM 调优来解决该问题。

4. 最近点对问题为什么跑出来是 0？因为有一对基站经纬度一样。本次运行时是去掉了其中一个基站的信息，因为个人觉得莫得意义。有一说一，其实用 KD tree 解决这种问题也挺快的，实在不行暴力 $O(n^2)$ 对于这种数据量也是可以做到的。但该算法的思想的确给了我很大的启发。

5. 排序的结果在哪？见文件夹内的 excel 文件。一目了然，无需多于分析。

6. 还可以怎么优化？可能需要多写点 Java 代码，读点 JVM 了。

7. 这些算法都是最佳的算法吗？显然不是。但是它们都很有意义，给学生以启发，只是随着后面我们高级数据结构和算法的使用，或许会有更方便的解决方案。

【备注】详细的代码在文件夹中，CCF CSP 第一次成绩的文件证明也在文件夹中，争取下次通过练习使用 C++/C 考出更好的成绩。

六、实验小结

以后再说起递归与分治，再也不会只知道 Hanoi 塔五行核心代码的例子和递归版的归并排序了。另外，也在实践中提高了自己 Java 编程与使用断点调试 Debug 的能力，也是头一次使用 jxl 这个 jar 包，在 idea 中导入，实现了 Excel 与 Java 之间的交互。