**西安电子科技大学**

**算法上机报告**



姓名： 秦龙

学号： 15030120044

# 实验题目

（一）渗透问题（Percolation）

使用合并-查找（union-find）数据结构，编写程序通过蒙特卡罗模拟（Monte Carlo simulation）来估计渗透阈值的值。

## 程序分析

程序要求实现对一个NxN矩阵的连通性判断问题，则可以使用quick-find算法和加权quick-union算法来实现，因为算法中的数组是一维的，所以首要问题就是将NxN矩阵中的点经过变换转换到一位数组中对应的位置来完成之后的算法求解。将它们在数组中的编号依次设置为1~NxN。为了之后检验连通性的问题，有一个非常巧妙的方法。抽象出在矩阵的顶部有一个单独的注水口，它和第一行的所有点都是连通的，在矩阵的底部有一个出水口，它和最后一行的所有点是连通的。按照题目的要求每次随机打开矩阵中的一个点，然后判断与它邻近的点是否已经被打开，若已经打开就将它们连接起来。那么每次打开一个新的结点之后检验连通性，只需要检验注水口和出水口是否连通即可。

## 程序实现

**public** **class** Percolation {

**private** **int**[] isOp; // 各个节点是否打开，0为否，1为是

**private** WeightedQuickUnionUF qUf; //使用的算法结构

**private** **int** row; // 行数

**private** **int** column; // 列数

**private** **int** count; // 连通分量的数量

**private** **int** num; // 已经打开的节点数目

// 初始化

**public** Percolation(**int** v,**int** e) {}

//打开节点 (row i, column j)

**public** **int** open(**int** i, **int** j) {}

// 判断节点 (row i, column j) 是否打开

**public** **boolean** isOpen(**int** i, **int** j) {}

//连接两个打开的结点

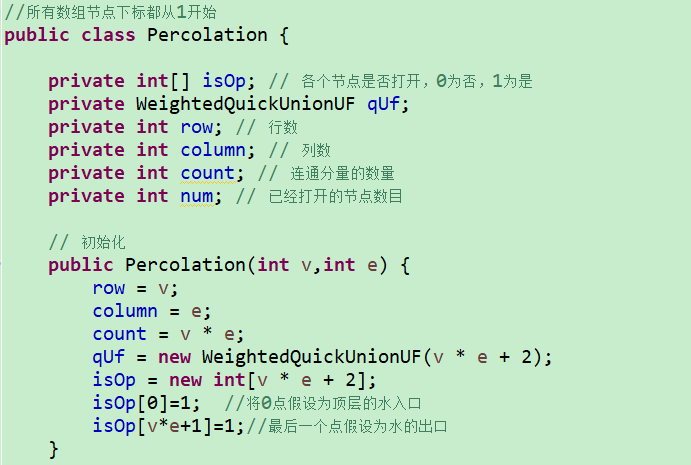
**public** **void** union(**int** i, **int** j, **int** m, **int** n) {}

//判断当前矩阵是否可以渗漏

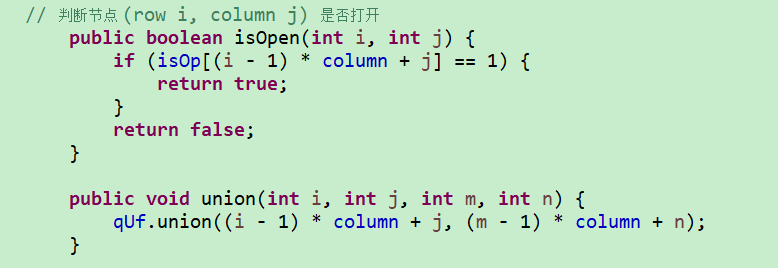
**public** **boolean** percolates() {}

}

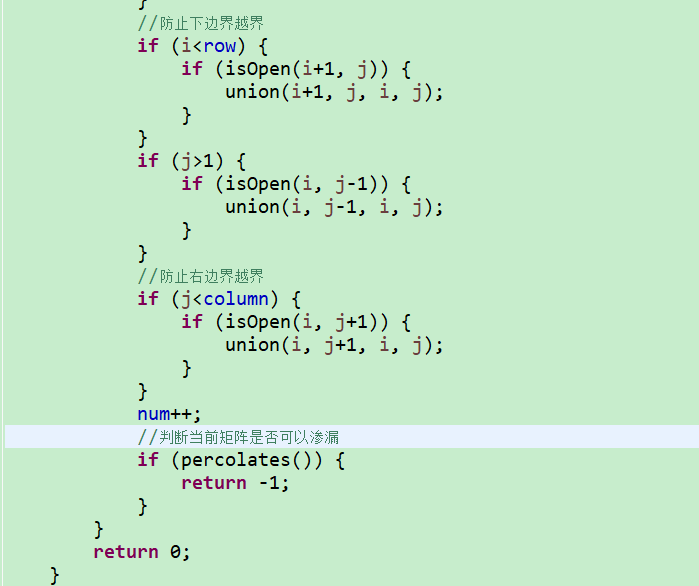
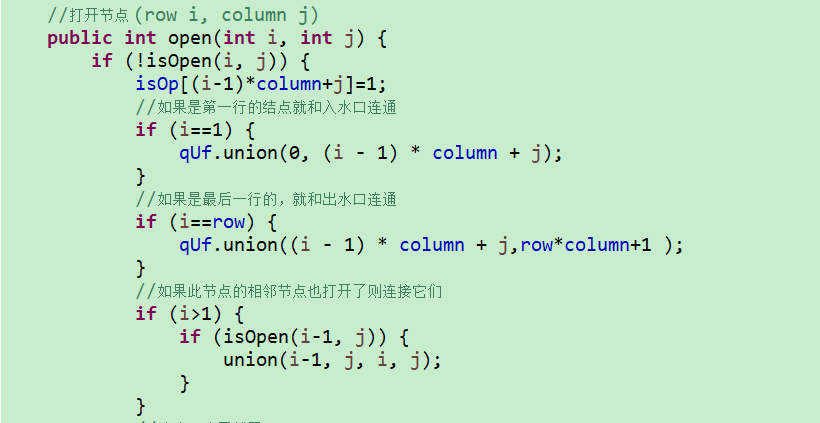
首先构建出Percolation数据类型结构，里面包含对矩阵节点的一些操作函数，还有对连通图操作的加权quick-union算法结构类WeightedQuickUnionUF，同理若使用quick-find算法，则对应包含的结构类为QuickFindUF，如下所示在对Percolation的构造函数初始化中初始NxN矩阵对应的数组isOp,将出水口和入水口直接打开，对应为isOp数组中的下标为0与下标为N\*N+1，其余的结点都初始化为闭塞。



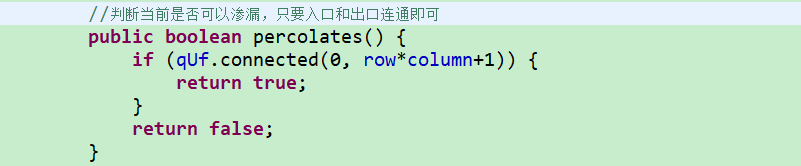
在isOpen函数中实现检测一个节点是否被打开，在union方法中来使用WeightedQuickUnionUF连接两个已经打开的结点，使相互连通



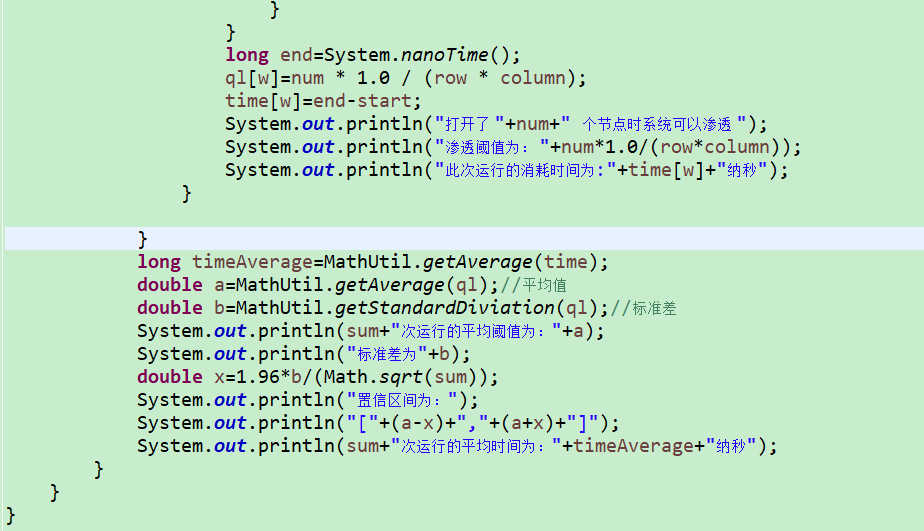
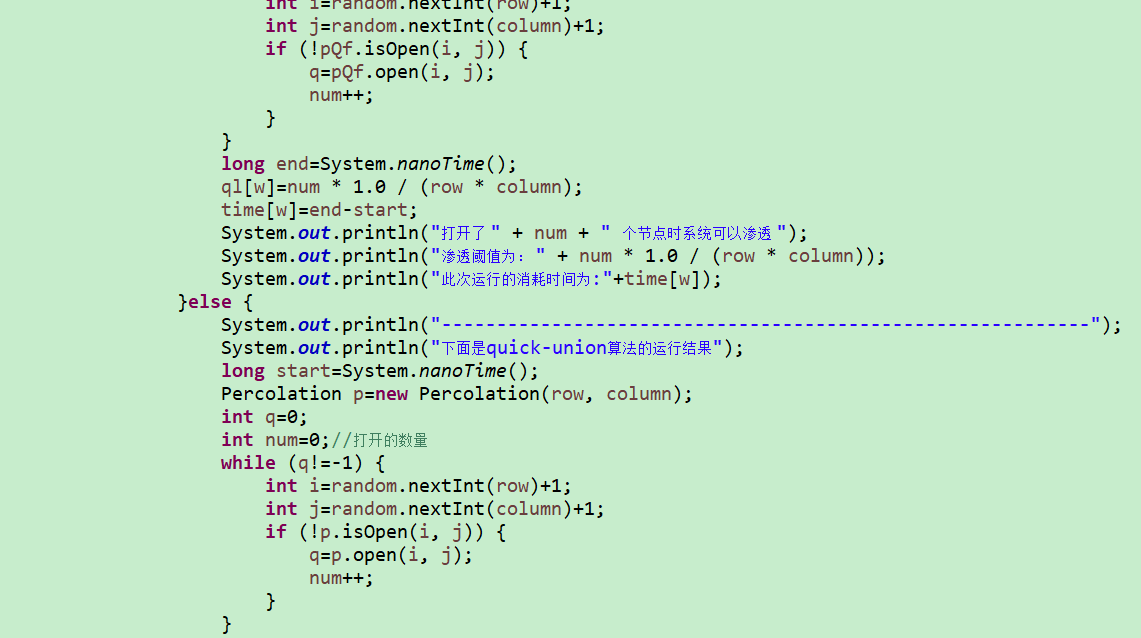
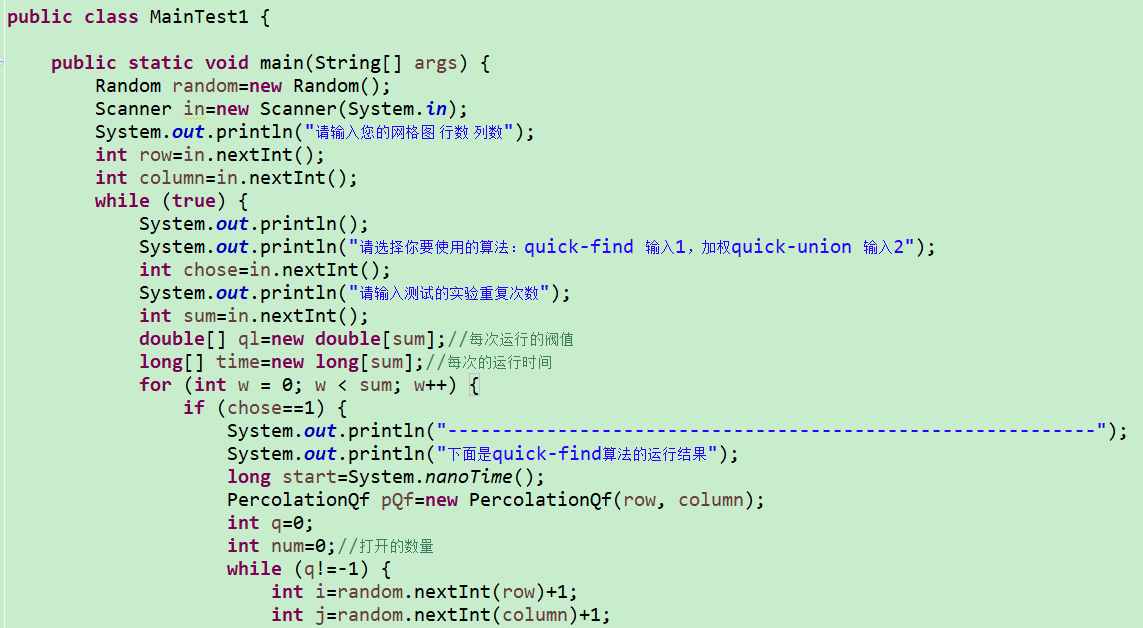
当要新打开一个节点时调用open方法，将对对应的isOp数组中的值置为1，若打开的节点为位于第一行，则将其与注水口连通,如位于最后一行则将其与出水口连通。然后判断此节点的邻近节点是否已经打开，若打开则union将它们连通。



每次新打开一个节点后用percolates方法来检测当前矩阵是否可以渗漏，若可以返回-1.

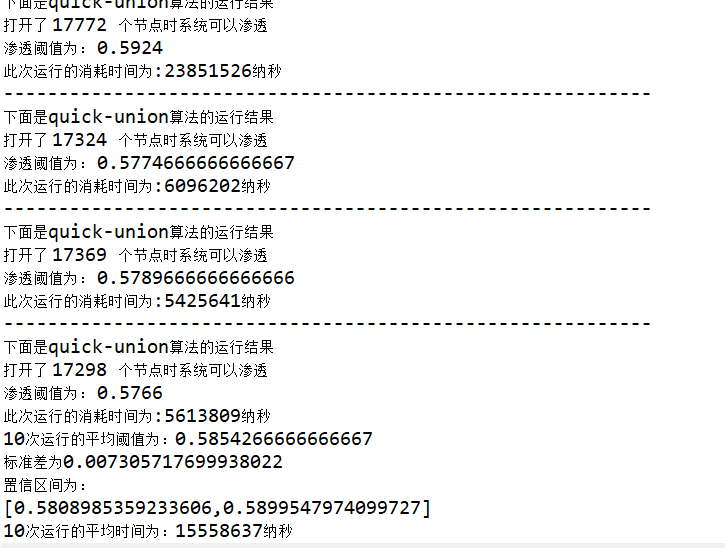
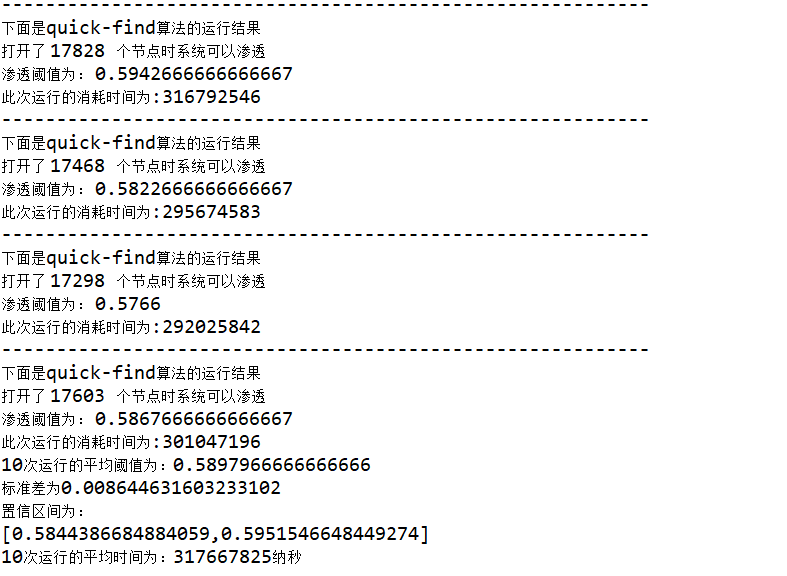
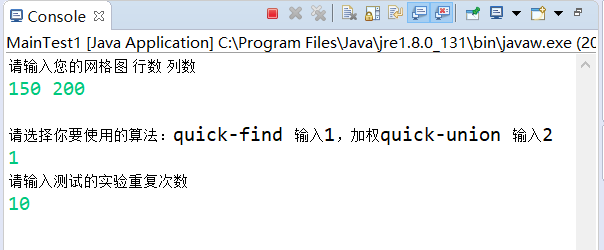


注：对应的WeightedQuickUnionUF与QuickFindUF算法结构详见附带文件



如上为主函数MainTest1,在主函数中通过键盘输入矩阵的行数与列数，然后可以选择所使用的算法，输入实验测试的次数，利用Random随机数来生成每次打开的节点坐标，while循环不断的打开新节点，若已经可以渗透则退出循环，记录下每次实验的阀值，以及运行时间，最后通过公式算出置信区间

## 测试结果



## 运行时间分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N=50时间单位(纳秒) | N | 2N | 3N | 4N | 5N |
| Quick-find | 6478054 | 40263203 | 202000264 | 593925736 | 1260870505 |
| *weighted quick-union* | 1382775 | 4628837 | 8562823 | 12626302 | 23190971 |

设N时的运行时间为t

得出运行时间与输入矩阵宽度aN的近似表达式：

(1) Quick-find: T=5a2t

(2) weighted quick-union: T=3(a-1)t

（二）几种排序算法的实验性能比较

实现插入排序（Insertion Sort，IS），自顶向下归并排序（Top-down Mergesort，TDM），自底向上归并排序（Bottom-up Mergesort，BUM），随机快速排序（Random Quicksort，RQ），Dijkstra 3-路划分快速排序（Quicksortwith Dijkstra 3-way Partition，QD3P）。在你的计算机上针对不同规模的数据进行实验，对比上述排序算法的时间及空间占用性能。要求对于每次输入运行10次，记录每次时间/空间占用，取平均值。

## 程序分析

要实现各种排序算法的性能比较，首先要实现所有的排序算法，将所有的排序算法放进一个应用类中，并将每种算法函数声明为静态。在主函数中通过随机生成数组数据，然后使用同样的数据来对不同的排序算法进行测试。

## 程序实现

public class AllSorts {

public static int length;

public static Comparable[] aux;

public AllSorts(int n) {

length=n;

aux=new Comparable[n];

}

//插入排序

public static void IS(Comparable[] a) {}

//自顶向下归并排序

public static void TDM(Comparable[] a,int lo,int hi) {}

public static void merge(Comparable[] a,int lo,int mid,int hi) {}

//自底向上归并排序

public static void BUM(Comparable[]a) {}

//随机快速排序

public static void RQ(Comparable[] a,int lo,int hi) {}

public static int partition(Comparable[] a,int lo,int hi) {}

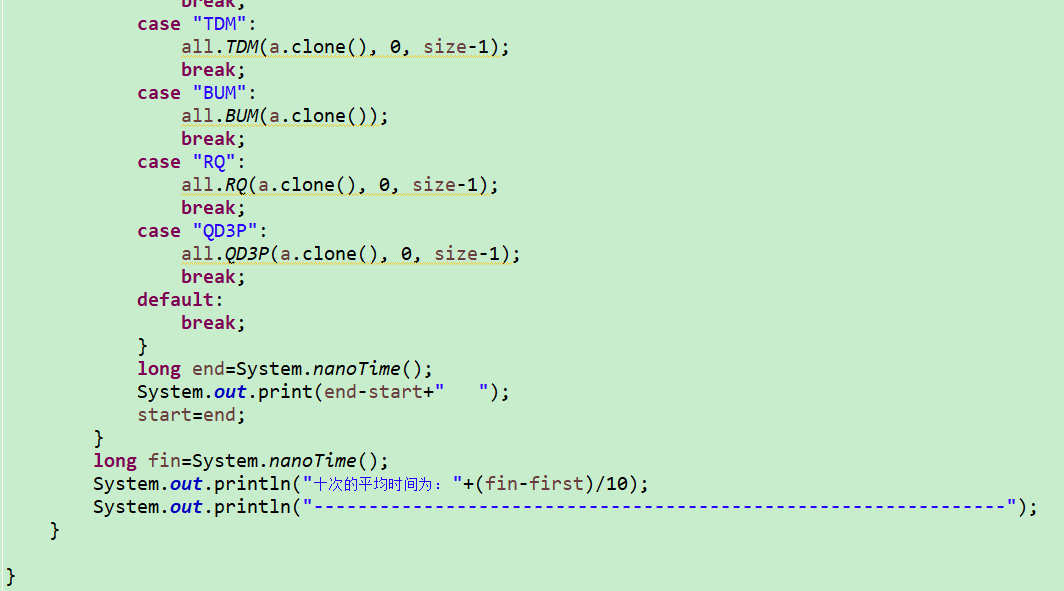
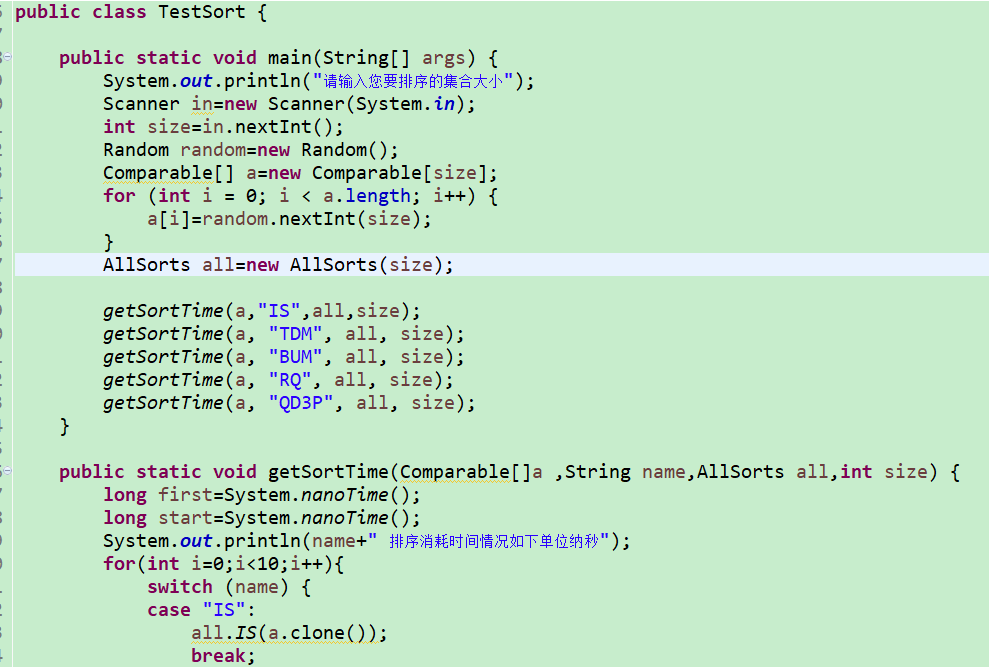
//Dijkstra 3-路划分快速排序

public static void QD3P(Comparable[] a,int lo,int hi) {}

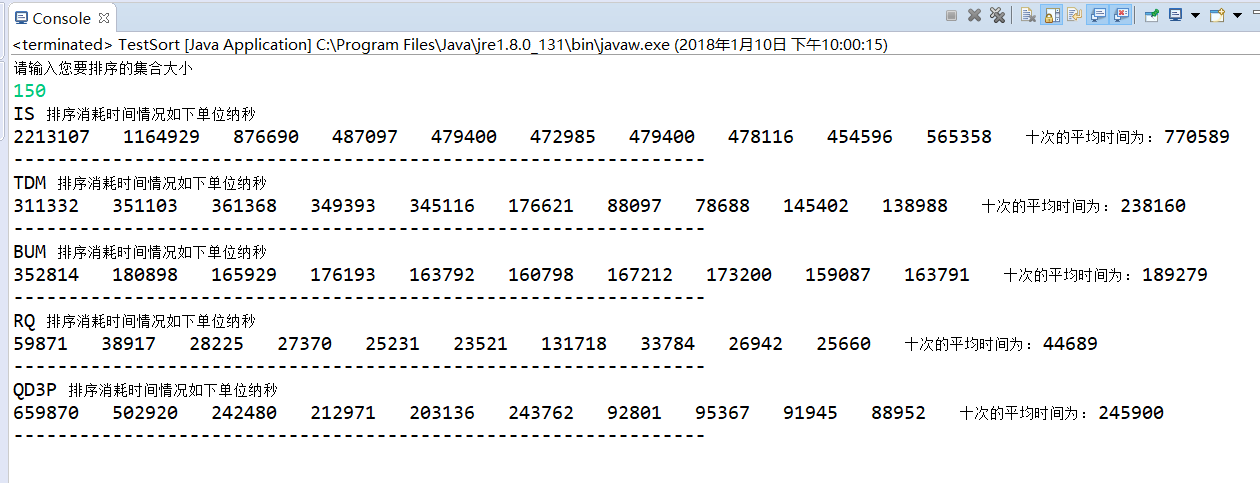
}

首先实现上述AllSorts类数据结构，在里面包含各种排序的方法，排序方法的具体实现见源代码附件。

在主函数中键盘输入所要排序的集合大小，然后通过Random随机数来对数组进行初始化，然后分别调用getSortTime函数来测试每一种排序算法，调用系统函数来计算每种算法的消耗时间。



## 测试结果



## 回答问题

1. Which sort worked best on data in constant or increasing order (i.e., already sorted data)? Why do you think this sort worked best?

我认为是插入排序，因为每次插入要处理的数字只需要一次比较就确定位置。

2. Did the same sort do well on the case of mostly sorted data? Why or why not?

不，初始数据基本有序情况对每种排序算法的影响很大，比较次数会相差很多

3. In general, did the ordering of the incoming data affect the performance of the sorting algorithms? Please answer this question by referencing specific data from your table to support your answer.

输入数据的初始排序情况会影响算法的性能，不同的排序算法实现机制不同。

4. Which sort did best on the shorter (i.e.,*n*=1,000) data sets? Did the same one do better on the longer (i.e.,*n*=10,000) data sets? Why or why not? Please use specific data from your table to support your answer.

当数据较短时快速排序算法性能较好，当数据长度很大时，快速排序算法性能较好，综合来说随即快速排序的性能最好

5. In general, which sort did better? Give a hypothesis as to why the difference in performance exists.

总的来说三项切分快速排序性能最好，时间复杂度介于N和NlogN之间，并且空间复杂度为lgN为原地排序

6. Are there results in your table that seem to be inconsistent? (e.g., If I get run times for a sort that look like this {1.3, 1.5, 1.6, 7.0, 1.2, 1.6, 1.4, 1.8, 2.0, 1.5] the 7.0 entry is not consistent with the rest). Why do you think this happened?

对于出现的偶然性相差较大的情况，我认为是由于数据的初始有序情况不同的原因，造成偶然性结果。

（三）地图路由（Map Routing）

实现经典的Dijkstra最短路径算法，并对其进行优化。这种算法广泛应用于地理信息系统（GIS），包括MapQuest和基于GPS的汽车导航系统。

**目标**：优化Dijkstra算法，使其可以处理给定图的数千条最短路径查询。

## 程序分析

因为要实现地图路由，地图是一个加权无向图，图中所用的边为加权无向边，实现一个加权无向图的数据结构，初始化地图后，利用Dijkstra算法来找出最短路径，经典的Dijkstra算法是初始化时就把所有节点的最短路径找出来了，程序优化则可以重用代码，多次查询两节点间最短路径用同一对象，并且每次查询当找到目标节点后便停止，不再遍历其他的节点。然后将上一次查询改变的成员变量部分还原，以供下一次的查询使用，这样便大大的优化了传统的Dijkstra算法。

## 程序实现

public class DijkstraUndirectedSP {

private double[] distTo;

private Edge[] edgeTo;

private IndexMinPQ<Double> pq;

private EdgeWeightedGraph mGraph;

private int from;//开始的原点

public DijkstraUndirectedSP(EdgeWeightedGraph G) {}

//设置算法的起点

public void setFrom(int from) {}

//更新到节点的最短路径

private void relax(Edge e, int v) {}

获取到某一节点的最短距离

public double distTo(int v) {}

//在这里才执行相关的路径初始化

public boolean hasPathTo(int v) {}

//还原上一次查询被修改的部分

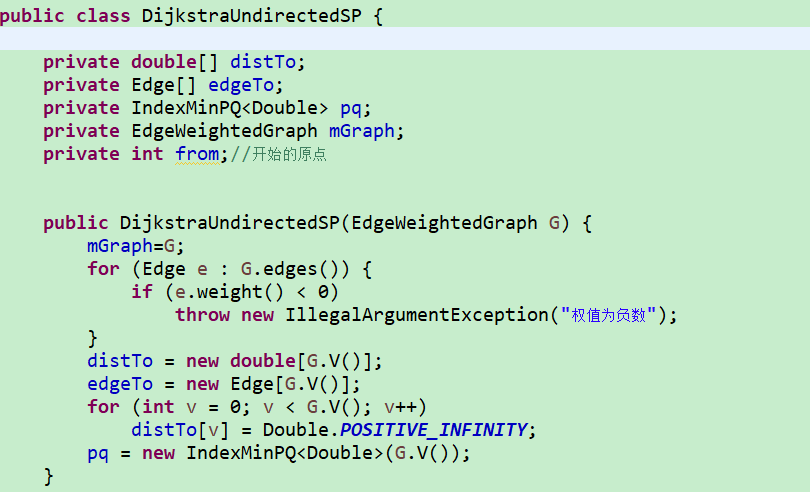
public void initDijkstra() {}

//遍历到一个节点所经过的边

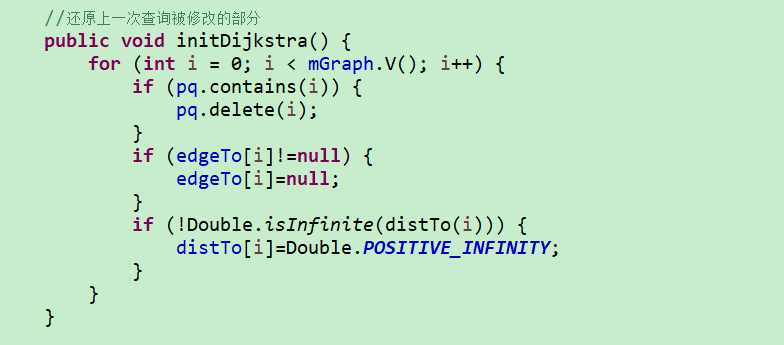
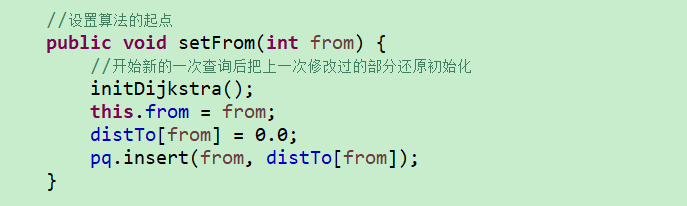
public Iterable<Edge> pathTo(int v) {}

}

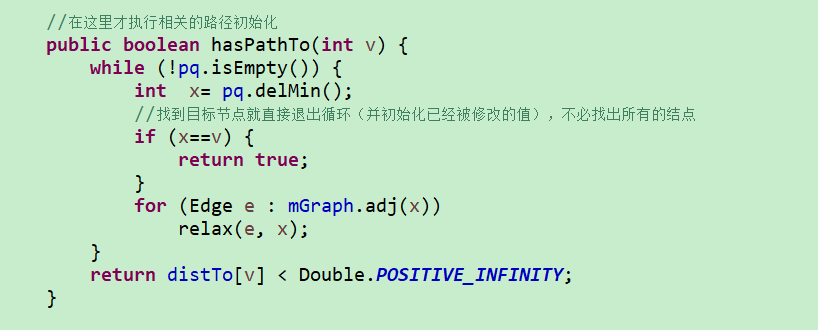
首先实现上面的数据结构类型，对相关的加权无向图进行操作，在初始化构造函数中传入传入无向图，初始化distTo数组，构建面向最小值的优先队列。



调用setFrom函数设置算法的起点，每次设置新的起点后，调用initDijkstra方法还原上次查询操作被修改的数据。



当执行查询源点到目的地的最短路径时调用hasPathTo方法来查询，不断的从优先队列pq中找出当前最短路径最小的节点，然后对此节点的邻节点进行relax松弛。如果从优先队列中得到的最小权值点就是目标节点的话即为找到两节点的最短路劲退出函数循环。



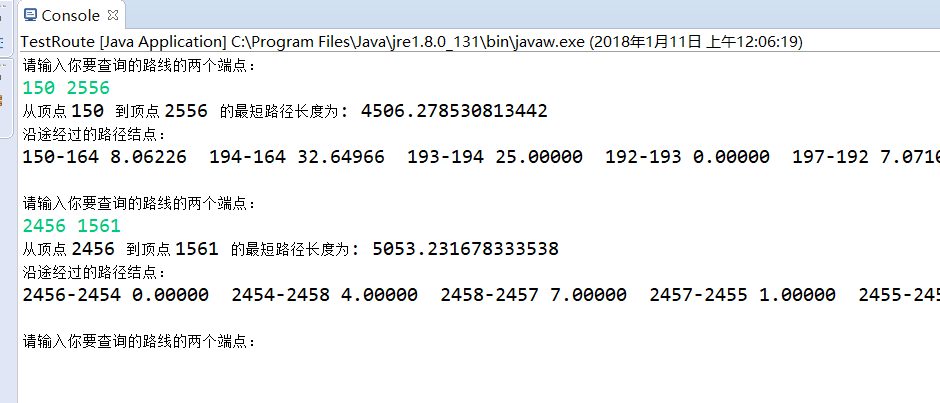
在主函数中从文本文件中读取相应的数据初始化所有顶点，无向边，构造出加权无向图。



调用fingRoute方法，while循环实现循环的查询给定的两个节点间的最短路径。每次调用hasPathTo方法查询，一旦找到目的节点就退出循环，下一次查询的时候setFrom方法会还原上次查询修改掉的数据，然后再进行hasPathTo查询目的节点。依次循环下去便实现了对传统Dijkstra算法的优化。使其可以处理给定图的数千条最短路径查询，并且在亚线性时间内解决最短路径问题。



## 测试结果



## 优化Dijkstra

优化方法采用的想法1减少检查的顶点数量，一旦发现目的地的最短路径就停止搜索。通过这种方法使每个最短路径查询的运行时间与*E*'log *V*'成比例，在不断执行查询，每次重新初始化在先前查询中改变的那些值来大大加速查询。具体的优化实现方法见上述程序实现。

上机心得体会

此次算法上机三道大题的综合性都比较强，通过这次的上机作业对面向对象编程思想有了更加深入的了解，熟悉了连通性问题中quick-find和加权quick-union算法数据结构。对各种排序算法的具体实现有了更好的掌握，知道了各种排序的性能，适用情况。熟悉了基于加权无向图的Dijkstra算法，优先级队列的使用，并且自己思考对传统的Dijkstra算法进行了优化，无论是时间上还是空间上性能都有了很大的提升。这三次试验对自己的综合编程能力有了很大的提升，熟悉了基本的算法与数据结构，收获颇丰。