算法与设计上机实验报告

姓名：党子清

学号：22009201170

实验一：渗透问题（Percolation）

一、实验内容

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。

二、实验环境

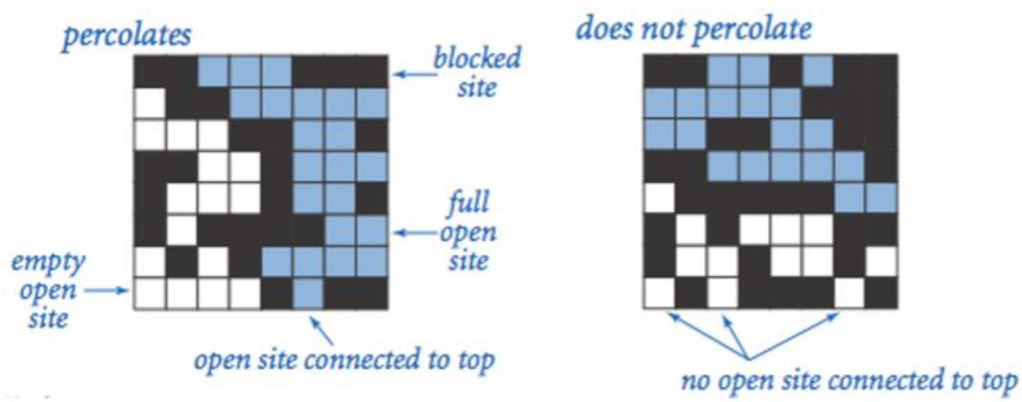
IntelliJ IDEA

Windows

三、实验步骤

1. **构建 Percolation 类**

   n*n 个点组成的网格，每个点是 Open 或 Closed 状态。假如最底部的点和最顶端的点连通，就说明这个网格系统是渗透的。
   比如图中黑色表示 Closed 状态，白色表示 Open，蓝色表示与顶部连通。所以左图是渗透的，右图不是：



   创建一个 Percolation 类，通过对 N*N 个网格中的点进行操作，来模拟判断渗透情况

```
public class Percolation {

    private final WeightedQuickUnionUF realQU;//不包含 virtual bot
```

```
tom
    private final WeightedQuickUnionUF virtualQU;//包含2个虚节点
    private final boolean[] isOpen;//方块的状态参数
    private final int virtualtop;//顶部虚节点
    private final int virtualbtm;//底部虚节点
    private final int n;
    private int openCount;
}
```

判断图是否渗透，关键是要判断顶部和底部是否连通。根据所学知识，使用并查集可以快速完成判断。每次打开网格中的点时，就讲该点与其上下左右四个相邻网格中开放的点并入同一集合。可以在顶部和底部创建两个虚拟节点，在初始化时将其分别与顶部和底部的节点并入同一集合，每次只需判断这两个虚拟节点是否在同一集合里，即可判断图是否渗透

Percolation 类实现的代码见附录

1. **蒙特卡洛模拟**

本实验通过蒙特卡洛算法，估算渗透阈值，具体做法为：

- 初始化 n*n 全为 Blocked 的网格系统
- 随机 Open 一个点，重复执行，直到整个系统变成渗透的为止
- 上述过程重复 T 次，计算平均值、标准差、95% 置信区间

对大小为 1500 的网格进行 50 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community Edition 2024.2.1\
please enter the size of the array:
1500
please enter the number of the trials:
50
mean                    = 0.5926039466666667
stddev                  = 0.0021132123108476494
95% confidence interval = 0.5920181940986725, 0.5931896992346609
运行时间为24337ms

进程已结束，退出代码为 0
```

对大小为 1000 的网格进行 50 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community Edition
please enter the size of the array:
1000
please enter the number of the trials:
50
mean                    = 0.5923323
stddev                  = 0.0035445708484949183
95% confidence interval = 0.5913497950873202, 0.5933148049126798
运行时间为11774ms

进程已结束，退出代码为 0
```

对大小为 200 的网格进行 500 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community
please enter the size of the array:
200
please enter the number of the trials:
50
mean                    = 0.591314
stddev                  = 0.00950346628491791
95% confidence interval = 0.5886797735417015, 0.5939482264582985
运行时间为4343ms
```

通过多次试验发现，随着模拟规模的增大，渗透阈值方差趋于稳定，95%置信区间稳定在 0.591~0.594，最终渗透阈值稳定在 0.5925 附近。并且，网格规模对渗透阈值无明显影响。

2. **不同的并查集算法性能比较**

为了研究不同的并查集算法性能，本实验重新构建了并查集的类，上述代码使用的是 WeightedQuickUnionUF，本实验还测试了当使用 QuickFindUF 时的情况，以下是实验结果：

对大小为 1500 的网格进行 50 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Commun
please enter the size of the array:
1500
please enter the number of the trials:
50
mean                    = 0.5927261066666667
stddev                  = 0.0024711048725185947
95% confidence interval = 0.5920411513418133, 0.59341106199152
运行时间为23871ms

进程已结束，退出代码为 0
```

对大小为 1000 的网格进行 50 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community Edition 2024.2.1\lib
please enter the size of the array:
1000
please enter the number of the trials:
50
mean                    = 0.59315034
stddev                  = 0.002852066752302937
95% confidence interval = 0.5923597874295475, 0.5939408925704525
运行时间为10603ms

进程已结束，退出代码为 0
```

对大小为 200 的网格进行 500 次模拟，结果如下

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA C
please enter the size of the array:
200
please enter the number of the trials:
50
mean                    = 0.5912345
stddev                  = 0.009005488533360622
95% confidence interval = 0.5887383059321439, 0.5937306940678561
运行时间为3043ms

进程已结束，退出代码为 0
```

由分析可得，使用 WeightedQuickUnionUF 速度会优于 QuickFindUF，得知，不同并查集的实现算法对解决实际问题会有影响，这提醒我们在设计算法时设计合理的数据结果以及算法的重要性。

附：实验源码部分（详见附件）

```java
package my.percolation;

import edu.princeton.cs.algs4.StdOut;
import edu.princeton.cs.algs4.WeightedQuickUnionUF;
//import edu.princeton.cs.algs4.QuickFindUF;

public class Percolation {

    private final WeightedQuickUnionUF realQU;//不包含 virtual bottom
    private final WeightedQuickUnionUF virtualQU;//包含 2 个虚节点
    private final boolean[] isOpen;//方块的状态参数
    private final int virtualtop;//顶部虚节点
    private final int virtualbtm;//底部虚节点
    private final int n;
    private int openCount;


    public Percolation(int n) {
        if (n <= 0) {
            throw new IllegalArgumentException("n 必须大于 0！");
        }
        this.n = n;
        virtualtop = 0;//顶部虚节点存在数组头部
        virtualbtm = n * n + 1;//底部虚节点存在数组尾部
        virtualQU = new WeightedQuickUnionUF(n * n + 2);
        realQU = new WeightedQuickUnionUF(n * n + 1);
        isOpen = new boolean[n * n + 2];
        isOpen[virtualtop] = true;
        isOpen[virtualbtm] = true;//2 个虚节点均置为 open
    }

    //将二维下标转化为一维下标
    private int linearIndex(int row, int col) {
        if (row < 1 || row > n) {
            throw new IndexOutOfBoundsException("行越界！");
        }
        if (col < 1 || col > n) {
            throw new IndexOutOfBoundsException("列越界！");
        }
        return (row - 1) * n + col;
    }


    public void open(int row, int col) {
        int curIndex = linearIndex(row, col);
        isOpen[curIndex] = true;
        openCount++;
```

```
        if (row == 1) {//第1 行
            virtualQU.union(curIndex, virtualtop);
            realQU.union(curIndex, virtualtop);
        }
        if (row == n) {//最后一行
            virtualQU.union(curIndex, virtualbtm);
        }
        neighborConnect(row, col, row - 1, col);  // 上
        neighborConnect(row, col, row + 1, col);  // 下
        neighborConnect(row, col, row, col - 1);  // 左
        neighborConnect(row, col, row, col + 1);  // 右
    }

    //将a 与open 的邻居相连接
    private void neighborConnect(int rowA, int colA, int rowB, int colB)
 {
        if (0 < rowB && rowB <= n && 0 < colB && colB <= n
                && isOpen(rowB, colB)) {
            virtualQU.union(linearIndex(rowA, colA), linearIndex(rowB,
colB));
            realQU.union(linearIndex(rowA, colA), linearIndex(rowB, col
B));
        }
    }

    //计数
    public int numberOfOpenSites() {
        return openCount;
    }


    public boolean isOpen(int row, int col) {
        return isOpen[linearIndex(row, col)];
    }


    public boolean isFull(int row, int col) {
        return realQU.connected(virtualtop,
                                linearIndex(row, col));//必须用realQU 判
断而不能用virtualQU，否则出现backwash 现象
    }


    public boolean percolates() {
        return virtualQU.connected(virtualtop, virtualbtm);//判断渗透用
virtualQU 判断
    }
```

```java
    public static void main(String[] args) {
        StdOut.println("请运行 PercolationStats 程序！");
    }
}

package my.percolation;

import edu.princeton.cs.algs4.StdIn;
import edu.princeton.cs.algs4.StdOut;
import edu.princeton.cs.algs4.StdRandom;
import edu.princeton.cs.algs4.StdStats;

import java.util.Scanner;

public class PercolationStats {

    private final double[] fractions;
    private final double CONFIDENCE_95 = 1.96;

    public PercolationStats(int n, int trials) {
        if (n <= 0) {
            throw new IllegalArgumentException("n 必须大于 0！");
        }
        if (trials <= 0) {
            throw new IllegalArgumentException("trials 必须大于 0！");
        }
        fractions = new double[trials];
        for (int i = 0; i < trials; i++) {//进行 trials 次随机实验
            Percolation percolation = new Percolation(n);
            int openedSites = 0;
            while (!percolation.percolates()) {//每次开 1 个方块直到perco
lates 为止
                int row = StdRandom.uniform(n) + 1;
                int col = StdRandom.uniform(n) + 1; //随机 open 方块
                if (!percolation.isOpen(row, col)) {
                    percolation.open(row, col);
                    openedSites++;//每开 1 个格子 opensites 加 1
                }
            }
            fractions[i] = openedSites * 1.0 / (n * n);//恰好渗透时 open
的格子数与总格子数之比
        }
    }

    //*p 均值
    public double mean() {
        return StdStats.mean(fractions);
```

```
    }

    //标准偏差
    public double stddev() {
        return StdStats.stddev(fractions);
    }

    //置信区间
    public double confidenceLo() {
        return mean() - CONFIDENCE_95 * stddev() / Math.sqrt(fractions.
length);
    }


    public double confidenceHi() {
        return mean() + CONFIDENCE_95 * stddev() / Math.sqrt(fractions.
length);
    }


    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        Scanner sc = new Scanner(System.in);
        StdOut.println("please enter the size of the array:");
        int n = StdIn.readInt();
        StdOut.println("please enter the number of the trials:");
        int trials = StdIn.readInt();
        PercolationStats stats = new PercolationStats(n, trials);
        long endTime = System.currentTimeMillis();

        StdOut.println("mean                    = " + stats.mean());
        StdOut.println("stddev                  = " + stats.stddev());
        StdOut.println("95% confidence interval = "
                           + stats.confidenceLo() + ", "
                           + stats.confidenceHi());
        long duration = endTime - startTime;
        StdOut.println("运行时间为"+duration+"ms");
    }
}
```

实验二：几种排序算法的实验性能比较（详见附件）

一、实验内容

实现插入排序（Insertion Soxt，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），Dikstra3-路划分快速排序（Quicksot with Dikstca 3-

way Partition，QD3P）。在你的计算机上针对不同输入规模数据进行实验，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。

二、实验环境

IntelliJ IDEA

Windows

三、实验步骤

五种基本排序的实现

1. 按照题目要求，设计题目要求的插入排序（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（Quicksort with Dijkstra 3-way Partition，QD3P）五种排序算法

2. 编写 `generateRandom()` 函数，实现产生指定大小的随机数组功能，用于排序

3. 记录每种算法的时间使用和空间占用

四、实验结果

测试不同规模数组的排序，从中选取部分测试结果如下所示

测试排序规模：10000 个元素

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\D
10000
插入排序用时为81毫秒，内存使用为0字节
自顶向下的归并排序用时为4毫秒，内存使用为0字节
自底向上的归并排序用时为3毫秒，内存使用为0字节
快速排序用时为53毫秒，内存使用为461384字节
三切分快速排序用时为17毫秒，内存使用为0字节
```

测试排序规模：100000 个元素

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA
100000
插入排序用时为8405毫秒，内存使用为0字节
自顶向下的归并排序用时为13毫秒，内存使用为400016字节
自底向上的归并排序用时为21毫秒，内存使用为400016字节
快速排序用时为5177毫秒，内存使用为6414192字节
三切分快速排序用时为199毫秒，内存使用为0字节

进程已结束，退出代码为 0
```

测试排序规模：150000 个元素

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Communit
150000
插入排序用时为25276毫秒，内存使用为0字节
自顶向下的归并排序用时为62毫秒，内存使用为600016字节
自底向上的归并排序用时为85毫秒，内存使用为600016字节
快速排序用时为12766毫秒，内存使用为11382880字节
三切分快速排序用时为507毫秒，内存使用为0字节

进程已结束，退出代码为 0
```

按照原理来说，由于插入排序的算法时间复杂度最高，因此在不同规模的测试中所用时间均最久。快速排序以及在几种排序中所用时间最少，归并排序次之，但两者差距不大。但是因为 intellj 的 ide 会限制栈大小，因此我在递归调用时使用了显式栈，因此对快速排序的影响很大。

同样按原理来说，按照从空间占用情况看，因为归并排序需要额外的数组空间实现归并操作，因此其空间占用一直是所有排序算法中最高，且其空间占用随着测试规模的增大而增大。但是因为 intellj 的 ide 会限制栈大小，因此我在递归调用时使用了显式栈，因此对快速排序的影响很大。

附：部分源代码

```java
package my.sort;

import java.util.Random;
import java.util.Scanner;

public class Main {
    public static Integer[] generateRandomArray(int n) {
        Integer[] array = new Integer[n];
        Random random = new Random();

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

```java
            array[i] = random.nextInt(n * 2);
        }

        return array;
    }

    public static void main(String[] args) {
        // 输入初始数据
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        Integer[] randomArray = generateRandomArray(n);

        // 插入排序
        long start_time = System.currentTimeMillis();
        long start_memory = getUsedMemory();
        InsertionSort.sort(randomArray);
        long end_time = System.currentTimeMillis();
        long end_memory = getUsedMemory();
        long duration = end_time - start_time;
        long memoryUsed = end_memory - start_memory;
        System.out.println("插入排序用时为" + duration + "毫秒，内存使用为
" + memoryUsed + "字节");

        // 自顶向下的归并排序
        start_time = System.currentTimeMillis();
        start_memory = getUsedMemory();
        TopDownMerge.sort(randomArray);
        end_time = System.currentTimeMillis();
        end_memory = getUsedMemory();
        duration = end_time - start_time;
        memoryUsed = end_memory - start_memory;
        System.out.println("自顶向下的归并排序用时为" + duration + "毫秒，
内存使用为" + memoryUsed + "字节");

        // 自底向上的归并排序
        start_time = System.currentTimeMillis();
        start_memory = getUsedMemory();
        ButtomUpMerge.sort(randomArray);
        end_time = System.currentTimeMillis();
        end_memory = getUsedMemory();
        duration = end_time - start_time;
        memoryUsed = end_memory - start_memory;
        System.out.println("自底向上的归并排序用时为" + duration + "毫秒，
内存使用为" + memoryUsed + "字节");

        // 快速排序
        start_time = System.currentTimeMillis();
        start_memory = getUsedMemory();
        Quick.sort(randomArray);
```

```java
        end_time = System.currentTimeMillis();
        end_memory = getUsedMemory();
        duration = end_time - start_time;
        memoryUsed = end_memory - start_memory;
        System.out.println("快速排序用时为" + duration + "毫秒，内存使用为
" + memoryUsed + "字节");

        // 三路快速排序
        start_time = System.currentTimeMillis();
        start_memory = getUsedMemory();
        QD3P.sort(randomArray);
        end_time = System.currentTimeMillis();
        end_memory = getUsedMemory();
        duration = end_time - start_time;
        memoryUsed = end_memory - start_memory;
        System.out.println("三切分快速排序用时为" + duration + "毫秒，内存
使用为" + memoryUsed + "字节");
    }

    public static long getUsedMemory() {
        Runtime runtime = Runtime.getRuntime();
        return runtime.totalMemory() - runtime.freeMemory();
    }
}
```

实验三：最短路（Map Routing）

一、实验内容

实现经典的 Dijkstra 最短路径算法，并对其进行优化。

地图。 本次实验对象是图 maps 或 graphs，其中顶点为平面上的点，这些点由权值为欧氏距离 的边相连成图。

可将顶点视为城市，将边视为相连的道路。

二、实验环境

IntelliJ IDEA

Windows

三、 实验步骤

1. Dijkstra 基本算法

在 dijkstra.java 文件中，对 Dijkstra 算法进行了基本实现。实现了以 s 为起点，到其余各点的最短路径。

该算法是未经优化过的最简实现，虽然能完成最短路径的计算，但是，求得每条最短路径的复杂度为 N2。

2. 优化，发现最短路之后就停止搜索

由 Dijkstra 算法可知，从优先队列里面取出的最小顶点，其最短路一定已经确定，因此，可以直接停止搜索，提前停止搜索后，算法的执行时间有了显著的减少：

3. 进一步优化，初始化只进行一次

因为每次查询最短路，整个 dist 数组，整个 pred 数组，全部都需要重新初始化，而这个初始化时间是与 V 成正比的，因此，如果能每次只初始化上次修改过的元素，还可以进一步提高速度。可以将 dist pred 两个数组的初始化，以及优先队列的初始化操作在 dijstra 对象的构造函数中执行

四、 实验结果

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community Editio
please enter the source point:23
please enter the destination point:45
the shortest distance from 23 to 45 is 164.21819939767659
the path is: 23->86->100->115->132->135->134->90->53->59->54->45
the processing time is 32.0ms
please enter the source point:33
please enter the destination point:100
the shortest distance from 33 to 100 is 314.1730670165184
the path is: 33->27->26->31->25->22->21->24->35->41->47->48->49
->52->63->64->66->69->67->54->59->53->90->134->135
->132->115->100
the processing time is 29.0ms
please enter the source point:100
please enter the destination point:1000
the shortest distance from 100 to 1000 is 569.0809401756168
the path is: 100->115->138->146->158->204->216->219->221->225->227->248->275
->317->368->378->389->476->494->497->501->504->608->609->614
->628->632->634->686->695->698->714->725->763->765->766->764
->767->769->774->826->831->866->913->1000
the processing time is 21.0ms
please enter the source point:
```

经过改进之后，速度得到优化，经过本次实验，理解了最短路径算法 dijkstra 的原理，并且尝试对其进行优化。也从中学到了很多。

附：部分源代码（详见附件）

```java
package my.Dijkstra;

import edu.princeton.cs.algs4.In;
import edu.princeton.cs.algs4.Stack;
import edu.princeton.cs.algs4.StdIn;
```

```java
public class Main {
    public static void main (String args[]) {
        String file_path = "D:\\DZQ\\java_learn\\sort\\src\\my\\Dijkstr
a\\usa.txt";
        while (true) {
            System.out.print("please enter the source point:");
            int start = StdIn.readInt();
            System.out.print("please enter the destination point:");
            int destination = StdIn.readInt();
            EdgeWeightedDiagraph G = new EdgeWeightedDiagraph(new In(fi
le_path));

            double start_time = System.currentTimeMillis();
            DijkstraSPImple sp = new DijkstraSPImple(G, start);
            double end_time = System.currentTimeMillis();
            double duration = end_time - start_time;
            if (sp.hasPathTo(destination)) {
                System.out.println("the shortest distance from " + star
t + " to " + destination + " is " + sp.distTo(destination));
            }
            Stack<DirectedEdge> path = sp.pathTo(destination);
            System.out.print("the path is: " + start);
            int cnt = 0;
            while (!path.isEmpty()) {
                DirectedEdge e = path.pop();
                System.out.print("->" + e.to());
                cnt++;
                if (cnt % 12 == 0) System.out.println();
            }
            System.out.println();
            System.out.println("the processing time is " + duration + "
ms");
        }
    }
}
package my.Dijkstra;

import edu.princeton.cs.algs4.IndexMinPQ;
import edu.princeton.cs.algs4.Stack;

public class DijkstraSPImple implements DijkstraSP{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSPImple (EdgeWeightedDiagraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<>(G.V());
```

```java
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Double.POSITIVE_INFINITY;
        }
        distTo[s] = 0.0;

        pq.insert(s, 0.0);
        while (!pq.isEmpty()) {
            relax(G, pq.delMin());
        }
    }

    @Override
    public void relax(EdgeWeightedDiagraph G, int v) {
        for (DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.changeKey(w, distTo(w));
                else pq.insert(w, distTo[w]);
            }
        }
    }

    @Override
    public double distTo(int v) {
        return distTo[v];
    }

    @Override
    public boolean hasPathTo(int v) {
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

    @Override
    public Stack<DirectedEdge> pathTo(int v) {
        if (!hasPathTo(v)) return null;
        Stack<DirectedEdge> path = new Stack<DirectedEdge>();
        for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from
()]) {
            path.push(e);
        }
        return path;
    }
}
```

实验四：文本索引

一、实验内容

编写一个构建大块文本索引的程序，然后进行快速搜索，来查找某个字符串在该文本中的出现位置。

二、实验环境

IntelliJ IDEA

Windows

三、实验步骤

1. 编写 suffixSort() 函数，实现后缀数组的排序

后缀数组保存在 pos[] 数组中，后缀数组的逆保存在 rank 数组中。使用 MSD 算法进行实现

```
void suffixSort(int n){
  //sort suffixes according to their first characters
  for (int i=0; i<n; ++i){
    pos[i] = i;
  }
  std::sort(pos, pos + n, smaller_first_char);
  //{pos contains the list of suffixes sorted by their first character}

  for (int i=0; i<n; ++i){
    bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
    b2h[i] = false;
  }

  for (int h = 1; h < n; h <<= 1){
    //{bh[i] == false if the first h characters of pos[i-1] == the firs
t h characters of pos[i]}
    int buckets = 0;
    for (int i=0, j; i < n; i = j){
      j = i + 1;
      while (j < n && !bh[j]) j++;
      next[i] = j;
      buckets++;
    }
    if (buckets == n) break; // We are done! Lucky bastards!
    //{suffixes are separted in buckets containing strings starting wit
h the same h characters}

    for (int i = 0; i < n; i = next[i]){
      cnt[i] = 0;
      for (int j = i; j < next[i]; ++j){
```

```
        rank[pos[j]] = i;
      }
    }

    cnt[rank[n - h]]++;
    b2h[rank[n - h]] = true;
    for (int i = 0; i < n; i = next[i]){
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0){
          int head = rank[s];
          rank[s] = head + cnt[head]++;
          b2h[rank[s]] = true;
        }
      }
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0 && b2h[rank[s]]){
          for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
        }
      }
    }
    for (int i=0; i<n; ++i){
      pos[rank[i]] = i;
      bh[i] |= b2h[i];
    }
  }
  for (int i=0; i<n; ++i){
    rank[pos[i]] = i;
  }
}
void getHeight(int n){
  for (int i=0; i<n; ++i) rank[pos[i]] = i;
  height[0] = 0;
  for (int i=0, h=0; i<n; ++i){
    if (rank[i] > 0){
      int j = pos[rank[i]-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      height[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
```

2. 编写二分查找函数

编写 binarychop 函数，利用二分查找，实现对输入的 key 关键字的查找匹配

```
int binarychop(char* key, int key_lengh, int left, int right){
    if (left > right){
```

```java
            return -1;
        }
        int mid = (right-left)/2+left;
        int p = pos[mid];
        for (int i = 0; i < key_lengh; i++){
            if (key[i]<str[p+i]){
                return binarychop(key,key_lengh,left,mid-1);
            } else if (key[i] > str[p+i]){
                return binarychop(key,key_lengh,mid+1,right);
            }
        }
        return p;
    }
```

四、实验结果

使用 1.txt 文本数据进行测试（长度：152089），使用 11.txt 中的测试数据。

```
D:\DZQ\java\jdk-17.0.12\bin\java.exe "-javaagent:D:\DZQ\java\IntelliJ IDEA Community Edition 2024.2.1\lib\idea_rt.jar=56842:D:
请输入语料库文件名:
1.txt
请输入查询文件名:
11.txt
conversation:暴力搜索结果: 3570, 耗时: 1736400 纳秒
二分搜索结果: 3570, 耗时: 154100 纳秒

considering:暴力搜索结果: 56, 耗时: 71400 纳秒
二分搜索结果: 56, 耗时: 48900 纳秒

very remarkable:暴力搜索结果: 107, 耗时: 95900 纳秒
二分搜索结果: 107, 耗时: 88900 纳秒

how:暴力搜索结果: 243, 耗时: 186200 纳秒
二分搜索结果: 243, 耗时: 74600 纳秒

world:暴力搜索结果: 246, 耗时: 179100 纳秒
二分搜索结果: 246, 耗时: 43000 纳秒

sobs:暴力搜索结果: 12263, 耗时: 5231100 纳秒
二分搜索结果: 12263, 耗时: 33800 纳秒

cdcjdjcdjvvkfjvkf:暴力搜索结果: -1, 耗时: 2090600 纳秒
二分搜索结果: -1, 耗时: 34800 纳秒

alice:暴力搜索结果: 1, 耗时: 40000 纳秒
二分搜索结果: 1, 耗时: 135300 纳秒
```

附：实验部分代码（详见附件）

```java
package my.String;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.stream.Stream;
```

```java
import static java.lang.System.in;

public class Main {
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);

        // Read corpus file name from user input
        System.out.println("请输入语料库文件名:");
        String corpusFileName = scanner.nextLine();
        corpusFileName = "D:\\DZQ\\java_learn\\sort\\src\\my\\String\\"
+corpusFileName;

        // Create a File object for the corpus file
        File corpusFile = new File(corpusFileName);

        // Check if the corpus file exists and is readable
        if (!corpusFile.exists() || !corpusFile.canRead()) {
            System.err.println("无法读取语料库文件: " + corpusFileName);
            return;
        }

        // Read the content of the corpus file into a string
        StringBuilder corpusContent = new StringBuilder();
        try (Scanner fileScanner = new Scanner(corpusFile)) {
            while (fileScanner.hasNextLine()) {
                corpusContent.append(fileScanner.nextLine());
            }
        } catch (IOException e) {
            System.err.println("读取语料库文件时发生错误: " + e.getMessag
e());
            return;
        }

        // Read query file name from user input
        System.out.println("请输入查询文件名:");
        String queryFileName = scanner.nextLine();
        queryFileName = "D:\\DZQ\\java_learn\\sort\\src\\my\\String\\"+
queryFileName;

        // Create a File object for the query file
        File queryFile = new File(queryFileName);

        // Check if the query file exists and is readable
        if (!queryFile.exists() || !queryFile.canRead()) {
            System.err.println("无法读取查询文件: " + queryFileName);
            return;
        }

        // Read the content of the query file into a list of strings
```

```
        List<String> queries = new ArrayList<>();
        try (Scanner fileScanner = new Scanner(queryFile)) {
            while (fileScanner.hasNextLine()) {
                queries.add(fileScanner.nextLine().trim());
            }
        } catch (IOException e) {
            System.err.println("读取查询文件时发生错误: " + e.getMessage
());
            return;
        }

        // Perform searches using both algorithms
        Search violenceSearch = new ViolenceSearch();
        Search binarySearch = new BinarySearch(corpusContent.toString
());

        for (String query : queries) {
            long startTime = System.nanoTime();
            int violenceResult = violenceSearch.search(corpusContent.to
String(), query);
            long endTime = System.nanoTime();
            long duration = endTime - startTime;
            System.out.println(query + ":暴力搜索结果: " + violenceResul
t + ", 耗时: " + duration + " 纳秒");

            startTime = System.nanoTime();
            int binaryResult = binarySearch.search(corpusContent.toStri
ng(), query);
            endTime = System.nanoTime();
            duration = endTime - startTime;
            System.out.println("二分搜索结果: " + binaryResult + ", 耗时:
 " + duration + " 纳秒");
            System.out.println();
        }
    }
}
```