# 实验一——基于蒙特卡罗模拟的渗透问题阈值估计

姓名：王越洋 学号：22009200894

## 1. 实验背景与目的

渗透问题（Percolation）是一种物理现象，它可以用来描述液体通过多孔介质流动的过程。通过计算机模拟，可以估算随机分布系统中渗透的临界值（渗透阈值）。在本实验中，我们通过使用并查集数据结构和蒙特卡罗模拟来估计二维网格系统的渗透阈值。

实验任务是实现一个 Percolation 类和 PercolationStats 类，利用这两个类来模拟渗透过程，并通过多次实验来估算渗透阈值及其置信区间。

## 2. 实验内容

实现 Percolation 数据类型，用于模拟一个 N×N 的网格，并使用并查集判断该系统是否渗透。

实现 PercolationStats 类，通过进行多次独立实验估算渗透阈值，并计算其均值、标准差和 95% 置信区间。

使用 QuickFind 和 WeightedQuickUnion 算法分别实现并查集，分析这两种算法在渗透问题中的效率差异。

## 3. 实验原理

在一个 N×N 的网格中，随机选择格点并将其设置为开放状态，直到系统形成从顶部到底部的连通路径。渗透阈值是指网格系统从不渗透状态转变为渗透状态时，开放格点所占的比例。

为了估计渗透阈值，通过蒙特卡罗模拟进行多次独立实验，记录每次实验中系统渗透时的开放格点比例，并统计多次实验的平均值、标准差和置信区间。

## 4. 程序设计

### 4.1 Percolation 类

（1）功能：模拟 N×N 网格系统中的渗透过程，使用并查集（QuickFindUF 或 WeightedQuickUnionUF）来判断系统是否渗透。

（2）主要方法：

Percolation(int N)：创建一个 N×N 的网格，所有格点初始状态为封闭。

void open(int i, int j)：打开指定位置的格点。

boolean isOpen(int i, int j)：检查某个格点是否为开放状态。

boolean isFull(int i, int j)：检查某个格点是否与顶部连通。

boolean percolates()：判断系统是否渗透，即从顶部到底部是否存在连通路径。

### 4.2 PercolationStats 类

（1）功能：进行多次独立实验，通过调用 Percolation 类的相关方法估计渗

透阈值，并计算统计结果。
（2）主要方法：
PercolationStats(int N, int T)：执行 T 次独立实验，在 N×N 的网格上估算渗透阈值。
double mean()：计算渗透阈值的平均值。
double stddev()：计算渗透阈值的标准差。
double confidenceLo()：计算 95% 置信区间的下界。
double confidenceHi()：计算 95% 置信区间的上界。

## 5. 代码实现

### 5.1 Percolation 类（基于 QuickFindUF/ WeightedQuickUnionUF）

（1）代码

```
1. package edu.princeton.cs.algs4;
2. /**
3.  * 该类模拟一个渗透系统，基于 N×N 网格，使
   用 QuickFindUF/WeightedQuickUnionUF 算法判断系统是否渗透。
4.  */
5.
6. public class Percolation {// WeightedQuickUnionUF
7.     private int N;
8.     private boolean[][] grid;
9.     private WeightedQuickUnionUF uf;
10.    private int virtualTop;
11.    private int virtualBottom;
12.
13.    public Percolation(int N) {
14.        if (N <= 0) throw new IllegalArgumentException("N 必须大
   于 0");
15.        this.N = N;
16.        grid = new boolean[N][N];
17.        uf = new WeightedQuickUnionUF(N * N + 2);  // 使用加权的并查集
18.        virtualTop = N * N;
19.        virtualBottom = N * N + 1;
20.    }
21.//public class Percolation { // QuickFindUF
22.//    private int N; // 网格的大小
23.//    private boolean[][] grid; // 网格，记录各格点是否为 open
24.//    private QuickFindUF uf;
25.//    private int virtualTop; // 虚拟顶点，用于快速检查顶行格点是否连接
26.//    private int virtualBottom; // 虚拟底点，用于快速检查底行格点是否连
   接
27.//
28.//    /**
29.//     * 构造函数，创建一个 N×N 网格，所有格点初始化为 blocked
```

```
30.//     * @param N 网格的大小
31.//     */
32.//    public Percolation(int N) {
33.//        if (N <= 0) throw new IllegalArgumentException("N 必须大
   于 0");
34.//        this.N = N;
35.//        grid = new boolean[N][N];
36.//        uf = new QuickFindUF(N * N + 2); // N*N + 2: 加上两个虚拟节
   点
37.//        virtualTop = N * N; // 顶部虚拟节点
38.//        virtualBottom = N * N + 1; // 底部虚拟节点
39.//    }
40.
41.    /**
42.     * 将格点 (i, j) 打开
43.     * @param i 行号
44.     * @param j 列号
45.     */
46.    public void open(int i, int j) {
47.        validate(i, j);
48.        if (!isOpen(i, j)) {
49.            grid[i - 1][j - 1] = true;
50.            int currentSite = xyTo1D(i, j);
51.
52.            // 如果是第一行，连接到虚拟顶点
53.            if (i == 1) uf.union(currentSite, virtualTop);
54.
55.            // 如果是最后一行，连接到虚拟底点
56.            if (i == N) uf.union(currentSite, virtualBottom);
57.
58.            // 连接周围的 open 格点
59.            connectAdjacent(i, j);
60.        }
61.    }
62.
63.    /**
64.     * 检查格点 (i, j) 是否是 open
65.     * @param i 行号
66.     * @param j 列号
67.     * @return true 如果该格点是 open
68.     */
69.    public boolean isOpen(int i, int j) {
70.        validate(i, j);
71.        return grid[i - 1][j - 1];
```

```
72.     }
73.
74.     /**
75.      * 检查格点 (i, j) 是否与顶行连接，即是否为 full
76.      * @param i 行号
77.      * @param j 列号
78.      * @return true 如果该格点是 full
79.      */
80.     public boolean isFull(int i, int j) {
81.         validate(i, j);
82.         int currentSite = xyTo1D(i, j);
83.         return uf.find(currentSite) == uf.find(virtualTop);
84.     }
85.
86.     /**
87.      * 判断系统是否渗透，即顶行与底行是否连接
88.      * @return true 如果系统渗透
89.      */
90.     public boolean percolates() {
91.         return uf.find(virtualTop) == uf.find(virtualBottom);
92.     }
93.
94.     // 将二维坐标 (i, j) 转换为一维
95.     private int xyTo1D(int i, int j) {
96.         return (i - 1) * N + (j - 1);
97.     }
98.
99.     // 连接与 (i, j) 相邻的 open 格点
100.        private void connectAdjacent(int i, int j) {
101.            int currentSite = xyTo1D(i, j);
102.
103.            if (i > 1 && isOpen(i - 1, j)) uf.union(currentSite, xyT
    o1D(i - 1, j)); // 上
104.            if (i < N && isOpen(i + 1, j)) uf.union(currentSite, xyT
    o1D(i + 1, j)); // 下
105.            if (j > 1 && isOpen(i, j - 1)) uf.union(currentSite, xyT
    o1D(i, j - 1)); // 左
106.            if (j < N && isOpen(i, j + 1)) uf.union(currentSite, xyT
    o1D(i, j + 1)); // 右
107.        }
108.
109.        // 验证 (i, j) 是否在有效范围内
110.        private void validate(int i, int j) {
111.            if (i < 1 || i > N || j < 1 || j > N) {
```

```
112.                 throw new IllegalArgumentException("坐
    标 (" + i + ", " + j + ") 超出范围");
113.             }
114.         }
115.     }
```

## （2）分析

Percolation 类负责模拟一个 N × N 的网格，并通过使用并查集（Union-Find）数据结构来判断系统是否渗透。它模拟了渗透问题中的格点状态变化，并为动态连通性提供了支持。以下对 Percolation 类的关键部分进行分析：

### 5.1.1 open(int i, int j) 方法

将网格中的某个格点设置为开放状态。每当一个格点被打开时，它需要与其相邻的开放格点进行连接，以保证连通性。在调用该方法时，网格的状态会更新，并将该格点与相邻的开放格点合并到同一个连通集。

**关键点：**

二维坐标转换为一维索引：通过 xyTo1D(i, j) 方法将 (i, j) 转换为一维索引。这是因为并查集的数据结构是用一维数组表示的。

连接虚拟顶点和虚拟底点：为了简化渗透判断，open() 方法会将顶行的开放格点与虚拟顶点相连，底行的开放格点与虚拟底点相连。这确保了我们在判断系统是否渗透时，只需要检查虚拟顶点和虚拟底点是否连通即可，而不必遍历整个网格。

连接相邻的开放格点：每当一个格点被打开时，它会与上下左右相邻的开放格点进行合并。这通过 connectAdjacent(i, j) 方法来实现。相邻的开放格点被合并到同一个连通集后，可以确保系统的渗透连通性。

### 5.1.2 isOpen(int i, int j) 方法

该方法用于判断网格中某个格点是否处于开放状态。实现方式简单明了，通过检查 grid[i - 1][j - 1] 数组中的布尔值来判断格点是否已经被打开。

**关键点：**

输入的合法性检查：该方法首先通过 validate(i, j) 来检查输入的行列索引是否在有效范围内，以防止数组越界。

返回值：该方法直接返回 grid 数组中相应位置的值。true 表示格点开放，false 表示格点仍处于封闭状态。

### 5.1.3 xyTo1D(int i, int j) 方法

将二维坐标 (i,j) 转换为一维数组中的索引。这种转换是为了适应并查集的实现，因为并查集通常是通过一维数组来管理不同的集合。

关键点：

通过公式 index = (i - 1) * N + (j - 1)，二维网格中的每个格点都映射到并查集中的一维数组索引。这个公式的含义是：前面 i-1 行有 (i-1) * N 个格点，再加上第 i 行的第 j 个格点。

### 5.1.4 percolates() 方法

判断系统是否渗透。渗透的定义是：如果从网格的顶部（第一行）到底部（最后一行）存在一条连通路径，则系统渗透。通过引入虚拟顶点和虚拟底点，我们可以简化这一判断，只需检查虚拟顶点和虚拟底点是否连通即可。

**关键点：**

效率高：无需遍历整个网格，只需检查 find(virtualTop) 和 find(virtualBottom) 是否相等，即可判断系统是否渗透。这样能够大幅提高渗透判断的效率。

### 5.1.5 两个虚拟节点的作用

虚拟顶点：连接网格第一行所有的格点，便于快速判断是否有路径从顶部延伸到该格点。

虚拟底点：连接网格最后一行所有的格点，用于判断从底部是否存在到某个格点的连通路径。

简化渗透判断：通过判断虚拟顶点与虚拟底点的连通性，简化了整个系统的渗透状态的判断过程。如果没有虚拟节点，判断系统是否渗透需要遍历最后一行所有的格点。

## 5.2 PercolationStats 类

### （1）代码

```
1. package edu.princeton.cs.algs4;
2.
3. import edu.princeton.cs.algs4.StdRandom;
4. import edu.princeton.cs.algs4.StdStats;
5.
6. public class PercolationStats {
7.     private double[] thresholds; // 记录每次实验的渗透阈值
8.     private int T; // 实验次数
9.
10.    /**
11.     * 构造函数，进行 T 次独立实验
12.     * @param N 网格大小
13.     * @param T 实验次数
14.     */
15.    public PercolationStats(int N, int T) {
16.        if (N <= 0 || T <= 0) throw new IllegalArgumentException("N
   和 T 必须大于 0");
17.        this.T = T;
18.        thresholds = new double[T];
19.
20.        // 记录开始时间
```

```
21.         long startTime = System.nanoTime();
22.
23.         // 进行 T 次实验
24.         for (int t = 0; t < T; t++) {
25.             Percolation percolation = new Percolation(N);
26.             int openSites = 0;
27.
28.             // 打开格点直到系统渗透
29.             while (!percolation.percolates()) {
30.                 int i = StdRandom.uniform(N) + 1; // 生成随机行
31.                 int j = StdRandom.uniform(N) + 1; // 生成随机列
32.
33.                 if (!percolation.isOpen(i, j)) {
34.                     percolation.open(i, j);
35.                     openSites++;
36.                 }
37.             }
38.             thresholds[t] = (double) openSites / (N * N); // 计算渗透
    阈值
39.         }
40.
41.         // 记录结束时间
42.         long endTime = System.nanoTime();
43.
44.         // 计算总运行时间并输出
45.         long elapsedTime = endTime - startTime;
46.         double elapsedTimeInSeconds = elapsedTime / 1e9; // 转换为秒
47.         System.out.println("总运行时间
    （秒）：  " + elapsedTimeInSeconds);
48.     }
49.
50.     /**
51.      * 返回渗透阈值的均值
52.      * @return 均值
53.      */
54.     public double mean() {
55.         return StdStats.mean(thresholds);
56.     }
57.
58.     /**
59.      * 返回渗透阈值的标准差
60.      * @return 标准差
61.      */
62.     public double stddev() {
```

```
63.         return StdStats.stddev(thresholds);
64.     }
65.
66.     /**
67.      * 返回 95% 置信区间的下界
68.      * @return 下界
69.      */
70.     public double confidenceLo() {
71.         return mean() - (1.96 * stddev() / Math.sqrt(T));
72.     }
73.
74.     /**
75.      * 返回 95% 置信区间的上界
76.      * @return 上界
77.      */
78.     public double confidenceHi() {
79.         return mean() + (1.96 * stddev() / Math.sqrt(T));
80.     }
81.
82.     /**
83.      * 主函数，用于从命令行接收参数并执行实验
84.      * @param args 输入参数
85.      */
86.     public static void main(String[] args) {
87.         if (args.length < 2) {
88.             System.out.println("用法: java PercolationStats <网格大
    小 N> <实验次数 T>");
89.             return;
90.         }
91.
92.         int N = Integer.parseInt(args[0]); // 网格大小
93.         int T = Integer.parseInt(args[1]); // 实验次数
94.
95.         PercolationStats stats = new PercolationStats(N, T);
96.
97.         System.out.println("渗透阈值均值 = " + stats.mean());
98.         System.out.println("渗透阈值标准差 = " + stats.stddev());
99.         System.out.println("95% 置信区
    间 = [" + stats.confidenceLo() + ", " + stats.confidenceHi() + "]");
100.        }
101.    }
```

## （2）分析

PercolationStats 类负责使用 Percolation 模拟渗透问题，并通过蒙特卡罗实验来估计渗透阈值。

### 5.2.1 构造函数 PercolationStats(int N, int T)

进行 T 次独立实验。在每次实验中，它使用 Percolation 模拟 N×N 网格，并记录每次实验中系统首次渗透时的开放格点比例。

关键点：

初始化参数检查：如果输入的网格大小 N 或实验次数 T 小于等于 0，程序会抛出异常。

实验过程：在每次实验中，不断随机开放格点，直到系统渗透。然后将开放格点数与总格点数的比例记录为一次实验的渗透阈值。

### 5.2.2 mean() 方法

计算渗透阈值的均值。实验多次后，系统的渗透阈值不会每次相同，因此需要计算所有实验中渗透阈值的平均值，作为对真实渗透阈值的估计。

公式：mean() 调用了 StdStats.mean(thresholds)，其中 thresholds 是所有实验的渗透阈值数组。均值的计算公式为：

其中，T 是实验次数，threshold_i 是第 i 次实验的渗透阈值。

### 5.2.3 stddev() 方法

计算渗透阈值的标准差，表示实验结果的波动性。标准差越大，说明实验结果的波动越大。

公式：stddev() 调用了 StdStats.stddev(thresholds)。标准差的计算公式为：

$$\text{mean} = \frac{1}{T}\sum_{i=1}^{T} \text{threshold}_i$$

### 5.2.4 confidenceLo() 和 confidenceHi() 方法

计算 95% 置信区间的下界和上界，即估计渗透阈值的波动范围。在 95% 的置信水平下，真实的渗透阈值有 95% 的概率落在这个区间内。

公式：

$$\text{stddev} = \sqrt{\frac{1}{T-1}\sum_{i=1}^{T} (\text{threshold}_i - \text{mean})^2}$$

下界：confidenceLo() = mean() - (1.96 * stddev() / Math.sqrt(T))

上界：confidenceHi() = mean() + (1.96 * stddev() / Math.sqrt(T))

其中 1.96 是标准正态分布中 95% 置信区间的对应系数，T 是实验次数，stddev() 是标准差。

关键点：

置信区间的计算反映了实验结果的波动情况。如果标准差较小，则置信区间会较窄，表明实验结果较为稳定。

通过多次实验和计算均值、标准差，可以对渗透阈值的准确度有更好的把

握。

## 6.结果分析

设置 150*150 大小的方格，进行 100 次试验。对于不同算法结果如下：

### 6.1 实验 1：WeightedQuickUnionUF（高效率）

```
总运行时间（秒）：  0.1556943
渗透阈值均值 = 0.5933391111111109
渗透阈值标准差 = 0.011506865659310063
95% 置信区间 = [0.5910837654418861, 0.59559445678033357]
```

运行时间非常短，仅为 0.1556943 秒，主要由于 WeightedQuickUnionUF 通过加权合并策略，使得树的高度不会过度增长，这保证了查找（find）和合并（union）操作的效率。随着网格规模增大，算法的时间复杂度仍保持在 O(log N)。

因此表现了加权合并策略在大规模网格中的高效性。

### 6.1 实验 2：QuickFindUF（低效率）

```
总运行时间（秒）：  4.6963348
渗透阈值均值 = 0.5924306666666667
渗透阈值标准差 = 0.013076989127623012
95% 置信区间 = [0.5898675767976526, 0.5949937565356808]
```

QuickFindUF 的每次 union 操作都需要遍历整个数组，导致在处理大规模网格时效率非常低，尤其是当系统中有大量的连通操作时，时间复杂度为 O(N)。

在 150 × 150 的网格上，QuickFindUF 的算法随着网格规模的增加而变得更加低效，从而大幅增加了运行时间。

### 6.2 渗透阈值均值的对比

实验 1 的渗透阈值均值为 0.593339，而实验 2 为 0.592431。两者的差异很小，基本都接近理论值 0.592746。这表明，两种算法在估算渗透阈值时，尽管效率不同，但得到的结果是一致且合理的。

无论使用哪种并查集算法，最终的渗透阈值均值应接近理论值。这表明即使 QuickFindUF 算法效率较低，它仍能正确估计出渗透阈值。

### 6.3 渗透阈值标准差的对比

实验 1 的标准差为 0.011506，而实验 2 为 0.013077。

标准差反映了实验结果的波动性。较小的标准差意味着实验结果更加集中，波动较小，而较大的标准差表示实验结果有更多的波动。

虽然实验 1 的标准差略小于实验 2，但差异不大。这说明使用 Weighted QuickUnionUF 算法的结果稍微更稳定一些，但总体上两者的波动性都在合理范围内。

### 6.4 95% 置信区间的对比

实验 1 的置信区间为 [0.5910837654418861, 0.5955944567803357]。

实验 2 的置信区间为 [0.5898675767976526, 0.5949937565356808]。

两组置信区间的范围有轻微差异，但总体上它们都包含理论值 0.592746。实验 1 的置信区间略微靠上，而实验 2 的置信区间范围略大。置信区间的差异与标准差一致：标准差越大，置信区间的范围越广。

说明两组实验都能够给出合理的渗透阈值估计，且都提供了对实际渗透阈值的较好估计。

## 6.5 总结

运行时间的差异表明 WeightedQuickUnionUF 在大规模网格上的性能远优于 QuickFindUF，尤其是在需要频繁进行连通判断的大规模渗透问题中。QuickFindUF 由于其 union 操作效率低，在处理较大网格时需要更长的时间。

渗透阈值的均值和标准差在两种算法中非常接近，说明即使是低效的算法 QuickFindUF，它在正确性上仍然能够较好地估算渗透阈值。

95% 置信区间的结果说明两种算法都能够提供合理的渗透阈值估计，并且估计值在多次实验中趋于一致。

# 7. 实验反思与体会

本次实验通过实现渗透问题并结合蒙特卡罗模拟，我们深入了解了并查集数据结构的作用，并分析了不同算法在渗透问题中的效率表现。实验的主要目的在于估算二维网格系统的渗透阈值，并分析其置信区间。

## 1. 理论与实践的结合

在理论学习中，渗透问题的核心概念是“渗透阈值”，这在统计物理学中有重要应用。然而，通过实际编程实现 Percolation 类和 PercolationStats 类，我们对这些理论概念有了更加直观的理解。在代码实现过程中，如何通过不断打开网格中的格点、判断系统是否渗透，结合并查集的优化策略，让理论不再抽象，而是通过每次实验的结果进行验证。

## 2. 并查集算法的深刻认识

本实验中的一个关键任务是选择并查集算法来解决动态连通性问题。通过实现和对比 QuickFindUF 和 WeightedQuickUnionUF，我们能够切实感受到不同算法在效率上的巨大差异。具体体会如下：

QuickFindUF 的局限性：在理论上我们知道 QuickFindUF 的时间复杂度较高，特别是在处理较大规模网格时表现不佳。然而在实际运行中，看到运行时间长达数秒甚至更多时，这种算法的劣势被直观地放大。这让我进一步认识到算法效率的重要性，特别是在处理大规模数据时，低效算法可能会导致不可接受的性能问题。

WeightedQuickUnionUF 的效率优势：加权合并策略有效避免了树的高度无限增长，从而显著提高了查找和合并操作的效率。在实验中，这一点通过极短的运行时间得到了验证。特别是在 N = 150 的网格上，WeightedQuickUnionUF 能够在不到一秒钟内完成实验，而 QuickFindUF 则需要数秒，充分体现了加权优化的优势。

## 3. 实验中的挑战

在实验中，主要的挑战来自：

（1）坐标转换与数组处理：二维网格的坐标与一维数组的映射在实现中是一个需要特别注意的细节。通过 xyTo1D 函数，准确地将二维坐标转换为一维索引，并使用并查集管理这些索引，使我们能够更高效地进行动态连通性判断。

（2）置信区间的计算：置信区间是统计实验中的重要概念，它反映了实验结果的稳定性。在实验中，计算渗透阈值的均值、标准差和 95% 置信区间时，我们需要使用准确的数学公式，并通过 PercolationStats 类进行多次实验来确保结果的可靠性。对于置信区间的实际意义，通过代码的实现，我进一步理解了如何在统计学中估计参数，并解释结果的波动性。

## 4. 实验的收获

算法效率的直观体会：通过对比两种不同的并查集算法，深刻体会到算法优化的重要性。即使在相对简单的问题场景中，优化后的算法依然能够大幅提高性能。在渗透问题这样的动态连通性场景下，正确选择并查集算法至关重要。

理论验证：通过蒙特卡罗模拟多次实验后，得出的渗透阈值结果接近理论值 0.592746，这让我们认识到，即使在现实世界中，随机现象的统计特性可以通过大量实验进行逼近和验证。这也是统计物理学和计算模拟中常见的工作方法。

## 5. 未来的改进与思考

更高效的数据结构：在这次实验中，虽然 WeightedQuickUnionUF 相比 QuickFindUF 有显著的性能提升，但仍然存在进一步优化的空间。通过路径压缩进一步减少树的高度，能够在大型网格上进一步提高效率。

更复杂的应用场景：渗透问题是物理学中的一个经典问题，但在实际应用中，渗透问题的模型可以更加复杂。未来实验可以考虑三维网格的渗透模拟，或者引入更加复杂的动态规则，这些都将是进一步研究的方向。

## 6. 实验体会

通过这次实验，我不仅巩固了对并查集算法的理解，还对如何通过模拟实验解决现实问题有了新的认识。算法不仅仅是理论知识，它在实际应用中决定了程序的效率和可行性。优化数据结构、设计高效的算法、处理大量的实验数据是本次实验带来的重要收获。

# 实验二——排序算法性能比较

姓名：王越洋 学号：22009200894

## 1. 实验背景与目的

本实验的目的是通过比较几种经典排序算法的时间和空间复杂度，深入了解它们在不同输入数据规模和排序顺序下的性能表现。这些算法包括：

（1）插入排序（Insertion Sort，IS）
（2）自顶向下归并排序（Top-down Merge Sort，TDM 或 MergeX）
（3）自底向上归并排序（Bottom-up Merge Sort，BUM 或 MergeBU）
（4）随机快速排序（Random QuickSort，RQ 或 Quick）
（5）Dijkstra 三向划分快速排序（3-way QuickSort，QD3P 或 Quick3way）

## 2. 实验内容

（1）针对不同输入规模的数据（1000、5000、10000 个元素）进行实验。
（2）排序算法运行 10 次，记录每次的时间和空间占用，取指令。

**实验数据要求：**

记录排序算法的运行时间（以微秒为单位）和空间使用量（以 KB 为单位）。

**实验分析：**

（1）比较不同算法在已排序数据和几乎排序数据上的表现。
（2）分析数据排序的初始顺序对算法性能的影响。
（3）比较算法在小规模（n=1000）和大规模（n=10000）数据集上的表现。
（4）对每个算法的整体性能提出假设，解释不同算法的表现差异。
（5）检查结果中是否存在不一致的情况，并分析原因。

## 3. 代码实现

### 3.1 算法实现

（1）插入排序（Insertion Sort，IS）

```
1. /**
2.      * 使用自然顺序对数组进行升序排列。
3.      * @param a 要排序的数组
4.      */
5.     public static void sort(Comparable[] a) {
6.         int n = a.length;
7.         // 从数组的第二个元素开始
8.         for (int i = 1; i < n; i++) {
9.             // 通过二分查找确定要插入的位置
10.            Comparable v = a[i];  // 要插入的元素
11.            int lo = 0, hi = i;   // 二分查找的范围
12.            while (lo < hi) {
13.                int mid = lo + (hi - lo) / 2;  // 计算中间索引
```

```
                if (less(v, a[mid])) hi = mid;  // 如果v比中间值小，继续在左侧查找
                else                 lo = mid + 1;  // 否则在右侧查找
            }

            // 插入排序的"半交换"部分
            // 将a[i]插入到位置lo，并将a[lo]到a[i-1]的元素向右移动
            for (int j = i; j > lo; --j)
                a[j] = a[j-1];  // 将元素右移
            a[lo] = v;  // 插入元素
        }
        assert isSorted(a);  // 检查数组是否已排序
    }
```

（2）自顶向下归并排序（MergeX）

```
/**
     * 将 src[lo..mid] 和 src[mid+1..hi] 两个有序子数组合并为一个有序数组，结果存入 dst。
     * @param src  原数组（包含两个有序子数组）
     * @param dst  目标数组，用于存放合并后的结果
     * @param lo   左边界索引
     * @param mid  中间索引
     * @param hi   右边界索引
     */
    private static void merge(Comparable[] src, Comparable[] dst, int lo, int mid, int hi) {

        // 前置条件：src[lo..mid] 和 src[mid+1..hi] 是有序的
        assert isSorted(src, lo, mid);
        assert isSorted(src, mid+1, hi);

        int i = lo, j = mid + 1;
        // 将两个子数组合并到 dst[lo..hi]
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              dst[k] = src[j++];  // 左半部分用完，取右半部分的元素
            else if (j > hi)               dst[k] = src[i++];  // 右半部分用完，取左半部分的元素
            else if (less(src[j], src[i])) dst[k] = src[j++];  // 右边元素小于左边元素，取右边的元素
            else                           dst[k] = src[i++];  // 否则取左边的元素
        }

```

```
24.         // 后置条件：dst[lo..hi] 是有序的
25.         assert isSorted(dst, lo, hi);
26.     }
27. 
28.     /**
29.      * 使用递归的归并排序对数组 src[lo..hi] 进行排序，结果存
    入 dst[lo..hi]。
30.      * @param src  原数组
31.      * @param dst  目标数组，用于存放排序结果
32.      * @param lo   左边界索引
33.      * @param hi   右边界索引
34.      */
35.     private static void sort(Comparable[] src, Comparable[] dst, in
    t lo, int hi) {
36.         if (hi <= lo + CUTOFF) {  // 小数组使用插入排序
37.             insertionSort(dst, lo, hi);
38.             return;
39.         }
40.         int mid = lo + (hi - lo) / 2;
41.         sort(dst, src, lo, mid);      // 递归排序左半部分
42.         sort(dst, src, mid+1, hi);    // 递归排序右半部分
43. 
44.         // 如果左右部分已经有序，则直接复制回目标数组，跳过 merge 操作
45.         if (!less(src[mid+1], src[mid])) {
46.             System.arraycopy(src, lo, dst, lo, hi - lo + 1);
47.             return;
48.         }
49. 
50.         // 否则执行归并操作
51.         merge(src, dst, lo, mid, hi);
52.     }
53. 
54.     /**
55.      * 对数组进行升序排序，使用自然顺序。
56.      * @param a 要排序的数组
57.      */
58.     public static void sort(Comparable[] a) {
59.         Comparable[] aux = a.clone();   // 复制原数组到辅助数组
60.         sort(aux, a, 0, a.length-1);    // 调用递归排序函数
61.         assert isSorted(a);             // 验证数组是否已排序
62.     }
63. 
64.     /**
65.      * 使用插入排序对数组 a[lo..hi] 进行排序。
```

```
     * 插入排序适用于小数组，效率高于归并排序。
     * @param a  要排序的数组
     * @param lo 左边界索引
     * @param hi 右边界索引
     */
    private static void insertionSort(Comparable[] a, int lo, int hi) {
        for (int i = lo; i <= hi; i++) {
            for (int j = i; j > lo && less(a[j], a[j-1]); j--) {
                exch(a, j, j-1);  // 交换 a[j] 和 a[j-1]
            }
        }
    }
```

（3）自底向上归并排序（MergeBU）

```
/**
     * 将 a[lo..mid] 和 a[mid+1..hi] 两个有序的子数组合并为一个有序数组
     * @param a    原数组，包含两个有序的子数组
     * @param aux  辅助数组，用于存放临时结果
     * @param lo   左边界索引
     * @param mid  中间索引
     * @param hi   右边界索引
     */
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {

        // 将 a[lo..hi] 的元素复制到辅助数组 aux[lo..hi]
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }

        // 合并回原数组 a[lo..hi]
        int i = lo, j = mid + 1;  // i 指向左半部分的起点，j 指向右半部分的起点
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              a[k] = aux[j++];  // 左半部分用完，取右半部分的元素
            else if (j > hi)               a[k] = aux[i++];  // 右半部分用完，取左半部分的元素
            else if (less(aux[j], aux[i])) a[k] = aux[j++];  // 右边元素小于左边，取右边的元素
            else                           a[k] = aux[i++];  // 否则取左边的元素
        }
```

```
    }

    /**
     * 对数组进行升序排序，使用自然顺序。
     * 采用自底向上的归并排序算法。
     * @param a 要排序的数组
     */
    public static void sort(Comparable[] a) {
        int n = a.length;  // 获取数组长度
        Comparable[] aux = new Comparable[n];  // 创建辅助数组
        // len 表示归并的子数组的长度，初始为 1，逐步扩展
        for (int len = 1; len < n; len *= 2) {
            // lo 表示每次归并的起始位置
            for (int lo = 0; lo < n - len; lo += len + len) {
                int mid  = lo + len - 1;  // 左子数组的结束位置
                int hi = Math.min(lo + len + len - 1, n - 1);  // 右子数组的结束位置，确保不越界
                merge(a, aux, lo, mid, hi);  // 合并两个长度为 len 的子数组
            }
        }
        assert isSorted(a);  // 验证数组是否已排序
    }
```

（4）随机快速排序（Quick）

```
/**
     * 对数组进行升序排列，使用自然顺序。
     * @param a 要排序的数组
     */
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);  // 随机打乱数组以避免最坏情况
        sort(a, 0, a.length - 1);  // 调用递归排序函数
        assert isSorted(a);  // 确认数组已排序
    }

    // 对子数组 a[lo..hi] 进行快速排序
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;  // 递归基：如果子数组只有一个元素，则返回
        int j = partition(a, lo, hi);  // 进行划分并获取划分后的索引
        sort(a, lo, j - 1);  // 递归排序左半部分
        sort(a, j + 1, hi);  // 递归排序右半部分
        assert isSorted(a, lo, hi);  // 确认子数组已排序
    }
```

```
19.
20.     // 对子数组 a[lo..hi] 进行划分，使得 a[lo..j-
    1] <= a[j] <= a[j+1..hi]
21.     // 返回划分的索引 j
22.     private static int partition(Comparable[] a, int lo, int hi) {
23.         int i = lo;  // 左指针
24.         int j = hi + 1;  // 右指针
25.         Comparable v = a[lo];  // 选取第一个元素为基准值
26.
27.         while (true) {
28.             // 找到左边的元素（大于基准值）进行交换
29.             while (less(a[++i], v)) {
30.                 if (i == hi) break;  // 到达右边界
31.             }
32.
33.             // 找到右边的元素（小于基准值）进行交换
34.             while (less(v, a[--j])) {
35.                 if (j == lo) break;  // 到达左边界
36.             }
37.
38.             // 如果指针交叉，结束循环
39.             if (i >= j) break;
40.
41.             exch(a, i, j);  // 交换不符合条件的元素
42.         }
43.
44.         // 将基准值放到正确的位置
45.         exch(a, lo, j);
46.
47.         // 现在 a[lo..j-1] <= a[j] <= a[j+1..hi]
48.         return j;  // 返回基准值的最终位置
49.     }
50.
51.     /**
52.      * 将数组重新排列，使得 a[k] 为第 k 小的元素；
53.      * a[0] 到 a[k-1] 是小于（或等于） a[k] 的元素；
54.      * a[k+1] 到 a[n-1] 是大于（或等于） a[k] 的元素。
55.      *
56.      * @param  a 要排序的数组
57.      * @param  k 第 k 小元素的索引
58.      * @return 第 k 小的元素
59.      * @throws IllegalArgumentException 当 k 不在 [0, a.length) 范围
    内时抛出异常
60.      */
```

```
    public static Comparable select(Comparable[] a, int k) {
        if (k < 0 || k >= a.length) {
            throw new IllegalArgumentException("index is not between 0 and " + a.length + ": " + k);
        }
        StdRandom.shuffle(a);  // 随机打乱数组
        int lo = 0, hi = a.length - 1;
        while (hi > lo) {
            int i = partition(a, lo, hi);  // 划分
            if      (i > k) hi = i - 1;  // 如果划分位置大于 k，继续在左半部分查找
            else if (i < k) lo = i + 1;  // 如果划分位置小于 k，继续在右半部分查找
            else return a[i];  // 找到第 k 小的元素
        }
        return a[lo];  // 返回第 k 小的元素
    }
```

（5）三向划分快速排序（Quick3way）

```
/**
     * 对字符串数组进行升序排列。
     *
     * @param a 要排序的字符串数组
     */
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);  // 随机打乱数组以避免最坏情况
        sort(a, 0, a.length - 1);  // 调用递归排序函数
        assert isSorted(a);  // 确认数组已排序
    }

    // 对子数组 a[lo..hi] 使用 3-way 快速排序算法进行排序
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;  // 递归基：如果子数组只有一个元素，则返回

        int lt = lo, gt = hi;  // lt 和 gt 分别表示小于和大于基准值的指针
        Comparable v = a[lo];  // 选取基准值
        int i = lo + 1;  // 当前处理元素的指针
        while (i <= gt) {
            int cmp = a[i].compareTo(v);  // 比较当前元素与基准值
            if      (cmp < 0) exch(a, lt++, i++);  // 小于基准值，交换并移动 lt 和 i 指针
            else if (cmp > 0) exch(a, i, gt--);  // 大于基准值，交换并移动 gt 指针
```

```
            else              i++;  // 等于基准值，移动 i 指针
        }

        // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]
        sort(a, lo, lt - 1);  // 递归排序小于基准值的部分
        sort(a, gt + 1, hi);  // 递归排序大于基准值的部分
        assert isSorted(a, lo, hi);  // 确认子数组已排序
    }
```

## 3.2 数据生成 & 性能测试

```
// 生成随机整数数组
    private static Integer[] generateRandomArray(int n) {
        Random random = new Random();
        Integer[] array = new Integer[n];
        for (int i = 0; i < n; i++) {
            array[i] = random.nextInt(10000); // 生成0到9999之间的随机数
        }
        return array;
    }

    // 生成已排序的整数数组
    private static Integer[] generateSortedArray(int n) {
        Integer[] array = new Integer[n];
        for (int i = 0; i < n; i++) {
            array[i] = i; // 生成已排序的数组
        }
        return array;
    }

    // 生成反向排序的整数数组
    private static Integer[] generateReverseSortedArray(int n) {
        Integer[] array = new Integer[n];
        for (int i = 0; i < n; i++) {
            array[i] = n - i - 1; // 生成反向排序的数组
        }
        return array;
    }

    // 记录算法执行时间
    private static long timeSort(Comparable[] array, String algorithm) {
        long start = System.nanoTime();
```

```
32.         switch (algorithm) {
33.             case "insertion":
34.                 Insertion.sort(array);
35.                 break;
36.             case "mergeX":
37.                 MergeX.sort(array);
38.                 break;
39.             case "mergeBU":
40.                 MergeBU.sort(array);
41.                 break;
42.             case "quick":
43.                 Quick.sort(array);
44.                 break;
45.             case "quick3way":
46.                 Quick3way.sort(array);
47.                 break;
48.             case "quick3string":
49.                 Quick3string.sort(Arrays.stream(array).map(String::va
    lueOf).toArray(String[]::new));
50.                 break;
51.             default:
52.                 throw new IllegalArgumentException("未知的排序算
    法: " + algorithm);
53.         }
54.         return System.nanoTime() - start; // 返回执行时间（纳秒）
55.     }
56.
57.     // 计算每个算法的空间使用情况
58.     private static long getMemoryUsage() {
59.         Runtime runtime = Runtime.getRuntime();
60.         return (runtime.totalMemory() - runtime.freeMemory()) / 1024;
     // 以 KB 为单位
61.     }
62.
63.     public static void main(String[] args) {
64.         String[] algorithms = {"insertion", "mergeX", "mergeBU", "qui
    ck", "quick3way"};
65.         int[] sizes = {1000, 5000, 10000}; // 不同的输入规模
66.
67.         for (int size : sizes) {
68.             System.out.println("测试数组大小: " + size);
69.             for (String algorithm : algorithms) {
70.                 System.out.println("算法: " + algorithm);
71.
```

```
72.                 // 测试随机数组
73.                 long totalTimeRandom = 0;
74.                 long totalMemoryRandom = 0;
75.                 for (int run = 1; run <= 10; run++) {
76.                     Integer[] randomArray = generateRandomArray(size)
    ; // 生成新的随机数组
77.                     long memoryBefore = getMemoryUsage();
78.                     long time = timeSort(randomArray, algorithm);
79.                     long memoryAfter = getMemoryUsage();
80.
81.                     long memoryUsage = memoryAfter - memoryBefore;
82.                     totalTimeRandom += time;
83.                     totalMemoryRandom += memoryUsage;
84.
85.                     System.out.printf("随机 - 运行%d - 耗时: %d 纳
    秒, 内存: %d KB%n", run, time, memoryUsage);
86.                 }
87.                 long averageTimeRandom = totalTimeRandom / 10; // 计
    算平均时间
88.                 long averageMemoryRandom = totalMemoryRandom / 10; //
     计算平均内存使用
89.                 System.out.printf("随机 - 平均耗时: %d 纳秒, 平均内
    存: %d KB%n", averageTimeRandom, averageMemoryRandom);
90.
91.                 // 测试已排序数组
92.                 long totalTimeSorted = 0;
93.                 long totalMemorySorted = 0;
94.                 Integer[] sortedArray = generateSortedArray(size); //
     生成已排序数组
95.                 for (int run = 1; run <= 10; run++) {
96.                     long memoryBefore = getMemoryUsage();
97.                     long time = timeSort(sortedArray.clone(), algorit
    hm);
98.                     long memoryAfter = getMemoryUsage();
99.
100.                      long memoryUsage = memoryAfter - memoryBefore;
101.                      totalTimeSorted += time;
102.                      totalMemorySorted += memoryUsage;
103.
104.                      System.out.printf("顺 - 运行%d - 耗时: %d 纳
    秒, 内存使用: %d KB%n", run, time, memoryUsage);
105.                  }
106.                  long averageTimeSorted = totalTimeSorted / 10; //
    计算平均时间
```

```
                long averageMemorySorted = totalMemorySorted / 10; // 计算平均内存使用
                System.out.printf("顺 - 平均耗时: %d 纳秒, 平均内存使用: %d KB%n", averageTimeSorted, averageMemorySorted);

                // 测试反向排序数组
                long totalTimeReverse = 0;
                long totalMemoryReverse = 0;
                Integer[] reverseSortedArray = generateReverseSortedArray(size); // 生成反向排序数组
                for (int run = 1; run <= 10; run++) {
                    long memoryBefore = getMemoryUsage();
                    long time = timeSort(reverseSortedArray.clone(), algorithm);
                    long memoryAfter = getMemoryUsage();

                    long memoryUsage = memoryAfter - memoryBefore;
                    totalTimeReverse += time;
                    totalMemoryReverse += memoryUsage;

                    System.out.printf("反 - 运行%d - 耗时: %d 纳秒, 内存使用: %d KB%n", run, time, memoryUsage);
                }
                long averageTimeReverse = totalTimeReverse / 10; // 计算平均时间
                long averageMemoryReverse = totalMemoryReverse / 10; // 计算平均内存使用
                System.out.printf("反 - 平均耗时: %d 纳秒, 平均内存使用: %d KB%n", averageTimeReverse, averageMemoryReverse);

                System.out.println(); // 输出空行以分隔不同算法的结果
            }
            System.out.println(); // 输出空行以分隔不同数组规模的结果
        }
    }
```

# 4. 结果分析

## 4.1 结果展示

时间（纳秒）

| | 随机-1000 | 顺序-1000 | 逆序-1000 | 随机-5000 | 顺序-5000 | 逆序-5000 | 随机-10000 | 顺序-10000 | 逆序-10000 |
|---|---|---|---|---|---|---|---|---|---|
| IS | 2492990 | 3920 | 1880010 | 21662350 | 12360 | 41983930 | 75871810 | 21340 | 136812840 |
| TDM | 302640 | 34670 | 199160 | 1487910 | 301440 | 584350 | 2369500 | 274600 | 476630 |
| BUM | 353650 | 147180 | 249970 | 4108430 | 3387710 | 3231390 | 2421100 | 1331660 | 1652370 |
| RQ | 500370 | 190280 | 201780 | 2362670 | 600760 | 530580 | 944620 | 1086250 | 1021060 |
| QD3P | 344940 | 232750 | 540680 | 2722730 | 1945570 | 817220 | 1558720 | 1495550 | 1447400 |

空间（KB）

| | 随机-1000 | 顺序-1000 | 逆序-1000 | 随机-5000 | 顺序-5000 | 逆序-5000 | 随机-10000 | 顺序-10000 | 逆序-10000 |
|---|---|---|---|---|---|---|---|---|---|
| IS | 8 | 0 | 49 | 10 | 0 | 60 | 12 | 0 | 70 |
| TDM | 10 | 10 | 12 | 51 | 51 | 55 | 101 | 89 | 96 |
| BUM | 10 | 10 | 12 | 50 | 51 | 55 | 101 | 100 | 58 |
| RQ | 5 | 2 | 10 | 20 | 10 | 40 | 30 | 15 | 50 |
| QD3P | 5 | 2 | 12 | 20 | 10 | 45 | 30 | 14 | 42 |

## 4.2 复杂度分析

已知各个算法的时空复杂度如表：

| 排序算法 | 最坏时间复杂度 | 最优时间复杂度 | 平均时间复杂度 | 空间复杂度 |
|---|---|---|---|---|
| **插入排序（Insertion Sort, IS）** | O(n^2) | O(n) | O(n^2) | O(1) |
| **自顶向下归并排序（MergeX）** | O(n log n) | O(n log n) | O(n log n) | O(n) |
| **自底向上归并排序（MergeBU）** | O(n log n) | O(n log n) | O(n log n) | O(n) |
| **随机快速排序（Quick）** | O(n^2) | O(n log n) | O(n log n) | O(log n) |
| **三向划分快速排序（Quick3way）** | O(n^2) | O(n log n) | O(n log n) | O(log n) |

### 1. 插入排序（IS）

空间复杂度：O(1)，是就地排序算法，不需要额外的存储空间，除了一些用于变量的常量空间。这意味着它在测试中表现出极低的内存占用。

插入排序在内存使用上将保持稳定，空间占用较少。然而，在大规模数据时，由于其时间复杂度较高，排序时间会显著增加。

### 2. 自顶向下归并排序（MergeX）

空间复杂度：O(n)，需要使用额外的辅助数组来存储中间结果，无法实现就

地排序。这导致在处理大规模数组时，它的空间消耗显著增加，特别是大数据规模时。

虽然归并排序在内存使用上占用较高，但其稳定的时间复杂度 O(nlogn) 确保了即使在大规模数据下也能保持较好的性能。

**3.自底向上归并排序（BUM）**

空间复杂度：O(n)，与自顶向下归并排序类似，自底向上归并排序同样需要额外的空间来存储合并操作中的中间结果。

实验表现：自底向上的归并排序在不同数据集上的内存消耗与自顶向下的归并排序相似。然而由于它采用了迭代而非递归方式，其空间使用虽然与 MergeX 相似，但在极大规模数据集上稍有优化。

**4.随机快速排序（RQ）**

空间复杂度：O(log n)，快速排序的空间复杂度来自递归调用栈。在随机化快速排序中，划分的子数组更均匀，因此递归深度较浅，内存消耗相对较小。

实数据集规模较大时，由于递归深度依赖于划分的平衡性，内存使用波动较小。

**5.三向划分快速排序（Quick3way）**

空间复杂度：O(logn)，与普通快速排序类似，但三向划分快速排序通过减少重复元素的比较，进一步减少了递归深度，因此在数据集中有大量重复元素时，内存消耗会较普通快速排序更低。

在处理大量重复元素的随机数据集时，三向划分快速排序表现出色。其内存占用与普通快速排序相似，递归深度较浅，且在重复元素多的场景下，内存使用有所优化。

## 4.3 总结

插入排序：虽然空间占用极低，但它的时间复杂度在大规模数据上导致了性能瓶颈，尤其是在反向排序的数据上。

归并排序（TDM 和 BUM）：虽然内存占用较大，但在大规模数据集上的时间表现稳定，适合对稳定性和性能要求较高的场景。

快速排序：在空间方面表现优秀，尤其是在处理大规模数据时，内存消耗维持在较低水平。时间复杂度表现稳定，特别是三向划分快速排序在有大量重复元素的数据集上表现最佳。

## 5. 问题回答

（1）哪种排序在已排序数据上表现最好？为什么？

插入排序在已排序数据上表现最好，因为它只需一次遍历，不需要进行任何交换操作，时间复杂度为 O(n)。

（2）在大部分有序数据上，是否表现相同？为什么？

插入排序在部分有序数据上仍表现良好。对于大部分排序算法，输入数据的部分有序性有助于减少不必要的交换和比较，尤其是归并排序和插入排序。

（3）输入数据的有序性是否影响排序算法的性能？

是的。有序数据通常会优化插入排序和归并排序的表现，快速排序（特别是随机化快速排序）在有序数据上的性能有所下降，但三向划分快速排序在有大量重复元素时表现良好。

（4）哪种排序在较小的数据集上表现最好？在较大数据集上是否相同？

插入排序在小数据集上表现较好，但在较大数据集上归并排序和快速排序更为有效。插入排序的时间复杂度 O(n^2)限制了其在大规模数据上的性能。

（5）总体上哪种排序表现更好？为什么会有差异？

快速排序（Quick3way）总体表现最好，特别是在大规模且包含重复元素的随机数据集上。快速排序的随机化策略避免了最坏情况的发生，而三向划分的优化则有效应对了重复元素。归并排序也表现稳定，但其需要更多的内存空间。

（6）结果中是否有不一致情况？为什么？

会出现性能突变的情况，尤其是在小数据集上进行递归排序时。可能与运行时环境、内存管理或 CPU 任务调度、JAVA 内存回收机制相关，具体表现为某些运行比预期时间长得多或内存占用突然增加或内存空间利用计算不准确。

## 6.实验总结

## 6.1 对各算法特性总结

**1.插入排序（Insertion Sort）**

特点：插入排序是一个简单的排序算法，适合小规模数据或部分有序的数据集。它的最坏时间复杂度为 O(n^2)，在已排序或几乎有序的数据集上表现较好。

已排序数据：在完全有序的情况下，插入排序的时间复杂度为 O(n)，表现最优。

随机和反向排序数据：由于插入排序的**逐一比较**和**交换机制**，性能会迅速下降，特别是在反向排序的情况下。

**2.自顶向下归并排序（MergeX）**

特点：使用**递归**的归并排序，最坏时间复杂度为 O(n log n)，并且在合并时**跳过已排序的部分**来优化性能。适合处理大规模数据。

随机数据：MergeX 在大规模的随机数据上表现较为稳定，时间复杂度维持在 O(nlogn)。

已排序数据：由于跳过已排序部分的优化，在这种情况下表现接近最佳。

反向排序数据：归并排序对逆序数据的性能与随机数据相同，表现较为稳定。

**3.自底向上归并排序（MergeBU）**

特点：使用迭代的方式进行归并排序，避免了递归带来的栈空间消耗，但仍然需要额外的 O(n)空间来存储辅助数组。

性能表现与 MergeX 相似，适合处理大规模数据，尤其是在递归深度可能成为瓶颈的情况下，MergeBU 更有优势。

**4.随机快速排序（Quick）**

特点：通过**随机化**避免最坏情况，平均时间复杂度为 O(nlogn)，但在最坏情况下（例如已经有序的数据），性能可能退化为 O(n^2)。

随机数据：表现最佳，运行速度快，特别适合处理大规模无序数据。

已排序数据：即使随机化，在处理完全有序的数据时，性能可能稍差，但仍然优于没有随机化的快速排序。

反向排序数据：与已排序数据类似，虽然随机化能改善性能，但仍可能退化。

**5.三向划分快速排序（Quick3way）**

特点：专门针对有大量重复元素的数据集，三向划分快速排序将数组划分为小于、等于和大于基准值的**三部分**，避免了重复元素带来的性能问题。

随机数据：表现优异，特别是在处理包含大量重复元素的数组时，效率比普

通快速排序高。

已排序和反向排序数据：表现相对稳定，不易退化。

## 6.2 收获&反思

**（1）收获**

时间复杂度的实际影响：通过实验更直观地感受到不同算法在不同规模数据下的表现差异。

空间复杂度的重要性：归并排序需要大量的辅助空间，尤其是在大规模数据下。虽然归并排序的时间复杂度较为稳定，但其空间消耗成为了一个潜在的问题。相比之下，快速排序及其优化版本（如三向划分快速排序）在空间效率上表现更好。

三向划分快速排序的优势：三向划分快速排序在处理包含大量重复元素的数据时表现尤为突出。通过减少重复元素的比较和交换，极大地提升了性能。

实验设计的重要性：排序算法的选择不仅仅依赖于理论上的复杂度，还应考虑输入数据的特点。比如插入排序在小规模、已排序数据上表现优异，但在大规模、无序数据上表现较差。

**（2）反思**

在选择排序算法时，不能仅仅根据最优的理论时间复杂度来判断。尽管归并排序和快速排序在理论上表现较好，但如果数据规模较小或输入数据有序，插入排序反而能更高效。反之，对于大规模、无序数据，归并排序和快速排序更为合适。

个别测试运行时间波动较大，可能与系统环境、内存调度及 Java 虚拟机的垃圾回收机制有关。必须控制好实验环境的变量，以确保测试结果的稳定性和可重复性。在实际应用中，算法的空间效率也非常重要，特别是在内存资源有限的情况下。

# 实验三——地图路由

学号：22009200894 姓名：王越洋

## 1.实验背景与目的

**（1）实验背景**

最短路径问题是图论中重要的经典问题之一，广泛应用于交通网络、路由选择、物流调度等场景。Dijkstra 算法是求解单源最短路径的著名算法之一，通常用于加权有向图且权重为非负数的情况。该算法的效率在很大程度上依赖于优先队列的性能，因此在不同实现的优先队列下，算法的时间和空间复杂度也有所不同。

**（2）实验目的**

比较在不同优先队列（Binary Heap、Multiway Heap）实现下 Dijkstra 算法的性能差异。本次实验研究不同维度的多路堆对算法效率的影响，从而分析选择合适优先队列的意义。通过多次实验测量运行时间和内存占用情况，验证并评估不同实现的实际表现。

## 2.实验内容

**（1）实现 Dijkstra 算法**

基于加权有向图数据结构，编写 Dijkstra 算法的实现类 DijkstraSP，使用优先队列管理候选节点并执行放松操作。

**（2）实现多种优先队列**

Binary Heap（IndexMinPQ）：二叉堆的优先队列实现。

Multiway Heap（IndexMultiwayMinPQ）：支持多个子节点的多路堆优先队列，并允许自定义维度 d。

**（3）实验设计与运行**

使用不同优先队列类型（Binary、Multiway（d=3）、Multiway（d=4））运行 Dijkstra 算法。

测量每种配置下的运行时间和内存使用。

结果分析与总结：对实验结果进行分析，包括各配置下的运行效率和资源占用情况。

## 3. 代码实现与原理

## 3.1 DijkstraSP 类

**（1）原理**

DijkstraSP 类实现了 Dijkstra 算法的核心逻辑。算法的核心思想是通过维护源点到各节点的最短路径长度，不断从未访问节点集合中选择当前最短路径的节点进行扩展。

**（2）关键代码说明**

1.初始化源节点：将源节点距离初始化为 0，其他节点距离为正无穷。

2.优先队列操作：优先队列的主要操作包括：

- insert(s, distTo[s])：将源节点 s 插入优先队列。

- delMin()：取出当前距离源节点最近的节点。
- decreaseKey()：若发现更短路径，更新节点优先级。

3.放松操作：检查是否可以通过某条边找到更短路径并更新邻接节点的路径长度。

```
1. public class DijkstraSP {
2.         private double[] distTo;
3.         private DirectedEdge[] edgeTo;
4.         private IndexMinPQ<Double> pq; // 使用二叉堆优先队列
5.
6.         public DijkstraSP(EdgeWeightedDigraph G, int s) {
7.                 distTo[s] = 0.0;
8.                 pq.insert(s, distTo[s]);
9.                 while (!pq.isEmpty()) {
10.                         int v = pq.delMin();
11.                         for (DirectedEdge e : G.adj(v))
12.                                 relax(e);
13.                 }
14.         }
15.
16.         private void relax(DirectedEdge e) {
17.                 int v = e.from(), w = e.to();
18.                 if (distTo[w] > distTo[v] + e.weight()) {
19.                         distTo[w] = distTo[v] + e.weight();
20.                         edgeTo[w] = e;
21.                         if (pq.contains(w)) pq.decreaseKey(w, d
   istTo[w]);
22.                         else pq.insert(w, distTo[w]);
23.                 }
24.         }
25.}
```

## 3.2 IndexMultiwayMinPQ 类

**（1）原理**

IndexMultiwayMinPQ 实现了多路堆优先队列，允许设定堆的维度 d，使得每个节点最多拥有 d 个子节点。理论上，多路堆可以减少堆的高度，从而降低插入和删除最小值的操作复杂度。

**（2）代码片段**

```
1. public class IndexMultiwayMinPQ<Key> {
2.         private final int d;        // 堆的维度
3.         private int[] pq;             // 堆数组
4.         private int[] qp;             // 逆数组
5.         private Key[] keys;         // 优先级数组
6.
```

```
        public IndexMultiwayMinPQ(int N, int D) {
            this.d = D;
            pq = new int[N + D];
            qp = new int[N + D];
            keys = (Key[]) new Comparable[N + D];
        }

        private void swim(int i) {
            while (i > 0 && greater((i - 1) / d, i)) {
                exch(i, (i - 1) / d);
                i = (i - 1) / d;
            }
        }

        private void sink(int i) {
            while (d * i + 1 < n) {
                int j = minChild(i);
                if (!greater(i, j)) break;
                exch(i, j);
                i = j;
            }
        }
}
```

## 3.3 EdgeWeightedDigraph 类

EdgeWeightedDigraph 管理了图的顶点和边以及每条边的权重。该类使用邻接表存储各顶点的邻接边，从而使边的管理和遍历更加高效。以下是代码的逻辑和关键代码的简述：

**（1）构造方法**

EdgeWeightedDigraph(int V)：初始化一个包含 V 个顶点且没有边的空图。

EdgeWeightedDigraph(int V, int E)：创建包含 V 个顶点、E 条随机边的图。

EdgeWeightedDigraph(In in)：从输入流中读取顶点和边的定义并构建图。

EdgeWeightedDigraph(EdgeWeightedDigraph G)：通过深拷贝另一个图 G 来创建新的图对象。

**（2）添加和获取方法**

addEdge(DirectedEdge e)：将有向边 e 添加到图中，并更新邻接表和入度数组。

adj(int v)：返回与顶点 v 相连的所有有向边。

indegree(int v) 和 outdegree(int v)：分别返回顶点 v 的入度和出度。

edges()：返回图中的所有有向边集合。

```
public Iterable<DirectedEdge> edges() {
```

```
        Bag<DirectedEdge> list = new Bag<DirectedEdge>();
        for (int v = 0; v < V; v++) {
            for (DirectedEdge e : adj(v)) {
                list.add(e);    // 将每条边添加到集合中
            }
        }
        return list;
    }
```

## 3.4 DijkstraSPMap 类

**（1）从文件读取数据并初始化图**

**1.文件读取和顶点、边信息解析：**函数从文件中读取顶点和边的数量，初始化一个带权有向图 EdgeWeightedDigraph。

- 时间复杂度：假设文件中包含 V 个顶点和 E 条边，则读取数据的时间复杂度约为 O(V+E)。
- 空间复杂度：图结构和数据存储使用 O(V+E) 的空间，顶点数和边数均会影响内存需求。

**2.存储顶点的坐标：**使用 HashMap 将每个顶点的编号与其坐标（x 和 y）关联，以便计算边的权重（距离）。

- 时间复杂度：使用 HashMap 插入 V 个顶点坐标，时间复杂度约为 O(V)。
- 空间复杂度：HashMap 中存储了 V 个顶点的坐标信息，空间复杂度为 O(V)。

3.构建边和计算边权重：读取边信息，并根据顶点坐标计算每条边的权重（使用欧几里得距离）。边的权重计算完毕后，通过 addEdge 方法将边加入到图中。

- 时间复杂度：对于每条边进行一次距离计算和插入操作，总时间复杂度为 O(E)。
- 空间复杂度：邻接表存储 E 条边，空间复杂度为 O(E)。

**（2）执行 Dijkstra 算法**

**1.选择优先队列类型：**根据输入参数 pqType，函数选择使用 binary 或 multiway（多路堆）作为优先队列实现来执行 Dijkstra 算法。

- 二叉堆：当 pqType 为 binary 时，选择 D=2 的多路堆，相当于一个二叉堆。
- 多路堆：当 pqType 为 multiway 时，函数使用指定的 D 值作为多路堆的维度。

**2.Dijkstra 算法的执行：**基于选择的优先队列类型，创建 DijkstraSP 对象并计算最短路径。

- 时间复杂度：Dijkstra 算法的复杂度主要依赖于优先队列操作。对于二叉堆实现，时间复杂度是 O((V+E)logV)；对于多路堆，时间复杂度与 D 相关，较大的 D 可以减少树的高度，但单次插入和删除操作的时间会增大。
- 空间复杂度：优先队列存储了图中顶点和边的路径信息，其空间复杂度约为 O(V+E)。

**（3）记录结束时间和内存使用**

executionTime：算法总执行时间，计算方法为 endTime - startTime，单

位为毫秒。

memoryUsed：算法执行过程中实际使用的内存，通过 endMemory - startMemory 计算（单位为 MB）。这里将字节转换为 MB 方便展示。

## 4.结果分析

### （1）数据

对 binary 和 multiway (d=3 和 d=4) 队列类型下的Dijkstra算法分别进行了运行，记录了它们的执行时间和内存使用情况。结果如下：

| 优先队列类型 | 执行时间（毫秒） | 内存使用（MB） |
|---|---|---|
| Binary Heap | 699 | 10 |
| Multiway Heap (d=3) | 567 | 11 |
| Multiway Heap (d=4) | 298 | 11 |

### （2）分析

1.执行时间：实验结果显示，multiway 优先队列的执行时间随维度 d 的增加而减小。这是因为随着 d 的增大，堆的高度降低了，使得 insert 和 delMin 操作变得更快。

2.内存使用：d=3 的多路堆使用了最多的内存，而 d=4 内存使用最少。这可能是由于更大的维度导致了更高的内存利用率，尤其是在降低堆高度和优化数组访问时，使堆结构更紧凑。

3.比较：多路堆在执行时间方面有显著优势，尤其是在更大维度的设置下，但内存消耗也需要权衡。因此，如果对内存使用要求较高，d=4 是较优的选择。

### （3）复杂度

插入、减少键值、获取最小键复杂度：$O(d \cdot \log_d(n))$。

删除最小键复杂度：$O(d \cdot \log_d(n))$。

删除键、增加键复杂度：$O(d \cdot \log_d(n))$。

随着 d 增加，堆的层数减少，使得插入和删除的复杂度下降，但由于每层比较的子节点增多，删除操作会有额外的开销。

## 5.总结

本实验展示了不同优先队列实现对 Dijkstra 算法的性能影响。通过实验结果可以得出以下结论：

1.在执行时间方面，multiway 优先队列优于 binary 优先队列，且多路堆的维度越大，性能提升越显著。

2.在内存消耗方面，multiway 的内存需求在 d=3 时显著高于 d=4，更大的维度反而有助于降低内存开销。

3.实际应用中应根据具体需求选择合适的优先队列。对于需要频繁操作大量节点的场景，多路堆（d=4）是一种较优选择，可以在性能和内存占用之间取得平衡。

## 6. 收获与反思

### （1）收获

通过本实验，我掌握了如何在 Dijkstra 算法中使用优先队列优化性能。深入理解了多路堆的结构与操作的细节，进一步加深了对数据结构效率和算法实现

之间关系的理解。同时，学习到如何通过实验数据分析来验证和总结算法的效率。

**（2）反思**

在实验过程中，我发现提高维度虽然可以减少堆高度，但会增加每层的子节点数，这在实际场景中需要动态调整以平衡时间和内存的需求。

# 实验四——文本索引

## 姓名：王越洋 学号：22009200894

## 1.实验背景与目的

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。

## 2.实验内容

从程序 BoyerMoore.java 开始。 这段代码基本上提供了一个完整的解决方案，但为了使其正常工作，你必须进行一些小的更改，因为它们在许多细节上与此处指定的问题不同，并且因为缺少各种小的代码。 你可能需要编辑此代码，或从头开始编写自己的代码。 再次，必须仔细考虑“比较”功能。

## 3.代码实现

1. 算法概述

Boyer-Moore 算法是一种高效的字符串匹配算法，它通过利用模式串与文本之间的不同部分进行跳跃来减少比较次数。其核心思想是通过预处理模式串来实现跳跃，优化搜索过程。该实现使用了坏字符规则，但不包括强健后缀规则。

2. 算法实现

在这个实现中：

R 是字符集的基数，这里是 256，表示使用的是 ASCII 字符集。

right 数组是坏字符跳过数组，它存储了模式串中每个字符最后一次出现的位置。初始化时，如果某个字符不在模式串中，其值为 -1。

search 方法用来在给定文本中搜索模式串的所有完整匹配，返回匹配的次数。

main 方法处理输入，读取文本和查询文件，统计模式串在文本中的匹配次数并输出。

3. 算法流程

（1）预处理阶段：

创建 right 数组，记录模式串中每个字符最后出现的位置。

（2）搜索阶段：

从文本的左端开始匹配模式串，逐字符从右向左比较。如果遇到不匹配的字符，使用 right 数组来确定跳跃距离。

如果整个模式串匹配成功，则计数器加一，继续搜索下一个位置。

如果没有匹配，则返回 0。

4. 代码分析

```
1. // 构造方法，预处理模式串
2. public BoyerMoore(String pat) {
3.     this.R = 256;  // ASCII 字符集大小
4.     this.pat = pat;
5.     right = new int[R];
```

```
    for (int c = 0; c < R; c++) right[c] = -1;
    for (int j = 0; j < pat.length(); j++) right[pat.charAt(j)] = j;
}

// 在文本中搜索模式串的所有完整匹配
public int search(String txt) {
    int m = pat.length();
    int n = txt.length();
    int skip;
    int count = 0;

    for (int i = 0; i <= n - m; i += skip) {
        skip = 0;
        for (int j = m - 1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i + j)) {
                skip = Math.max(1, j - right[txt.charAt(i + j)]);
                break;
            }
        }
        if (skip == 0) {
            count++;
            i++;  // 查找下一个位置
        }
    }

    return count;
}
```

## 比较功能：

（1）字符逐一比较

在 Boyer-Moore 算法中，比较是通过从模式串的右端开始，与文本中当前位置的字符进行逐一比较的方式来实现的。具体步骤如下：

假设模式串 pat 的长度是 m，文本 txt 的长度是 n，要在 txt 中查找 pat 的出现位置。

算法从文本 txt 中的每个可能位置（即从 i = 0 到 i = n - m）开始，逐个字符与模式串 pat 进行比较。

比较从模式串的最右侧字符（即 pat[m-1]）开始，逐步向左移动，直到匹配或不匹配为止。

```
for (int j = m - 1; j >= 0; j--) {
    if (pat.charAt(j) != txt.charAt(i + j)) {
        skip = Math.max(1, j - right[txt.charAt(i + j)]);
        break;
    }
```

```
6. }
```

- pat.charAt(j) 是模式串的当前字符。
- txt.charAt(i + j) 是文本中的字符。
- 当遇到不匹配时，skip 计算跳过的字符数，通过坏字符规则确定下一个可能的匹配位置。

（2）坏字符规则

坏字符规则是 Boyer-Moore 算法中最核心的“比较”功能，基于模式串中每个字符的最右出现位置来跳过一些不必要的字符。这个规则大大提高了算法的效率，避免了对每个字符的逐个比较。

右移规则： 当某个字符在模式串中没有匹配时，算法将模式串向右移动，跳过所有不可能匹配的位置。具体来说，若模式串中的字符在文本中没有匹配到，则通过坏字符规则（即当前字符在模式串中的最右位置）计算出**跳跃的距离**。

```
skip = Math.max(1, j - right[txt.charAt(i + j)]);
```

在这个过程中：

right[txt.charAt(i + j)] 是文本中当前字符在模式串中的最右位置。

如果当前字符不存在于模式串中，则 right[txt.charAt(i + j)] 会为 -1。

skip 是计算出的跳跃步数，保证模式串能够跳过不必要的匹配位置

（3）模式串匹配

在每一次字符比较时，从模式串的右端开始向左端比较字符，直到找出模式串的完整匹配或者发现某个字符不匹配。

如果所有字符匹配，则模式串完全匹配文本的一部分，算法返回匹配的起始位置。

如果发现不匹配，则根据坏字符规则计算跳跃步数，跳过已经匹配过的字符。

（4）核心逻辑

比较功能的关键在于通过跳跃减少不必要的比较，使得在每次不匹配时，能够尽量跳过尽可能多的字符。

字符逐一比较：从模式串的右侧字符开始，逐个比较文本中的字符。

坏字符规则：当发现不匹配时，计算跳跃步数，跳过不必要的字符。

高效跳跃：通过最右位置的坏字符规则来跳过文本中的多个字符，从而避免逐个字符的比较，显著提高效率。

## 4. 结果分析

输出结果如图所示：

```
2 (Boyer-Moore) falling Time: 14 μs
31 (Boyer-Moore) March Hare Time: 12 μs
1 (Boyer-Moore) miserable Mock Turtle Time: 7
31 (Boyer-Moore) moment Time: 12 μs
134 (Boyer-Moore) own Time: 10 μs
87 (Boyer-Moore) went Time: 8 μs
-- (Boyer-Moore) ??? Time: 9 μs
Boyer-Moore average time: 1060 μs
Brute Force average time: 1504 μs
```

## 5.总结

在本次实验中，我实现了基于 Boyer-Moore 算法的字符串匹配程序，并通过与其他算法进行对比，分析了该算法在实际应用中的性能优势与不足。以下是本次实验的几个关键收获与总结：

1. Boyer-Moore 算法的原理与应用

Boyer-Moore 算法是一种非常高效的字符串匹配算法，特别适用于模式串较短而文本较长的情况。其核心思想基于两个规则：

坏字符规则：当模式串与文本的字符不匹配时，根据不匹配字符在模式串中最后出现的位置来决定跳跃的距离，从而避免了模式串和文本中已经匹配的部分重新比较。

好后缀规则（未实现）：如果出现了部分匹配的后缀，可以利用模式串中相同后缀的位置信息来跳过不必要的比较，进一步提高匹配效率。

在实验中，我只实现了坏字符规则，通过预处理模式串来生成一个数组记录每个字符在模式串中最后一次出现的位置。然后，在实际匹配过程中，利用这个信息决定跳过多少个字符，减少无效的比较。

2. 性能比较与优化

在实验过程中，我将 Boyer-Moore 算法与暴力匹配算法进行了对比。暴力匹配算法的时间复杂度为 O(mn)，其中 m 是模式串的长度，n 是文本的长度。暴力算法每次都从文本的每个字符开始逐一比较，效率较低。而 Boyer-Moore 算法通过利用坏字符规则的跳跃，能够在某些情况下将匹配时间大幅减少，尤其是在模式串与文本不匹配的部分时，能够跳过大量字符比较。

我通过分析实验结果，发现 Boyer-Moore 算法的性能在处理大规模文本时展现出了明显的优势，尤其是当模式串较短且文本较长时。相比暴力匹配算法，Boyer-Moore 能够大幅度减少字符比较次数，从而提高整体匹配效率。

3. 处理大规模文本的挑战与解决方案

在实验中，我使用了一个相对较大的文本文件进行测试，并处理了多个查询模式串。面对大规模文本时，如何保证匹配的效率是一个挑战。通过 Boyer-Moore 算法的优化策略，能够有效地减少不必要的字符比较，显著提高了查询速度。

尽管 Boyer-Moore 算法在很多情况下表现出色，但仍然存在一些局限性。对于某些特殊模式串（如模式串中有大量重复字符的情况），该算法可能无法充分发挥其跳跃的优势，仍然会进行较多的比较。因此，对于不同类型的文本和模式串，选择合适的算法非常重要。

4. 实验中的问题与思考

输入格式处理：如何从文件中正确读取文本和查询模式串，并确保格式一致性，避免读取错误。

5. 未来的改进

引入好后缀规则：进一步完善 Boyer-Moore 算法，结合好后缀规则，使得在大规模文本中进行模式串匹配时，效率能够得到进一步提升。

并行处理：对于非常大的文本文件，可以考虑使用并行算法来加速字符串匹配过程，例如使用多线程将文本分段处理，进而加速查询过程。

与其他算法对比：除了 Boyer-Moore，还可以尝试实现并比较其他字符串匹配算法，如 KMP 算法、Rabin-Karp 算法等，进一步优化匹配效率。