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。