实验二 几种排序算法的实验性能比较

一、 实验目的

在计算机上针对不同输入规模数据进行实验，对比排序算法的时间及空间占用性能。

二、 实验内容

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（Quicksort with Dijkstra 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。

三、 实验方法

1. 时间占用计算

时间占用使用 JAVA 自带的 system.nanotime()函数即可。system.nanotime()函数返回正在运行的 Java 进程的高精度时间源的当前值，以纳秒为单位。

我们只需在排序进程运行前后各调用一次 system.nanotime()函数得到其起始时间和结束时间，两者做差即可求出时间占用。如下：

```
1.    public static double time(String hts, Comparable[] a) {
2.          long StartTime = System.nanoTime();
3.          //… … … … sort 进程… … … …
4.          long endTime = System.nanoTime();
5.          StdOut.println("时间占用->:" + (endTime - StartTime) / 1000 + " μs");
6.          return 0;
7.    }
```

2. 空间占用计算

空间计算需要使用 JAVA runtime 类实例的两个对象函数，即 runtime.freememory()和 runtime.totalmemory()。

runtime.freememory()返回 JVM 的空闲内存量，以字节为单位；

runtime.totalmemory()返回 JVM 已经从操作系统那里挖过来的内存大小，即进程当时所被分配的所有内存，以字节为单位；

由此我们可以得知，runtime.totalmemory()−runtime.freememory()即为当时所占用的内存大小。与时间占用方法类似，我们只需在排序进程运行前后各求一次所占内存得到其其实占用内存和结束占用内存，两者做差即可求出内存占用。如下：

```
1.  public static double time(String hts, Comparable[] a) {
2.          Runtime run = Runtime.getRuntime();
3.          run.gc();
4.          long startMemo = run.totalMemory() - run.freeMemory();
5.          //… … … … sort 进程… … … …
6.          long endMemo = run.totalMemory() - run.freeMemory();
7.          StdOut.println("内存占用->:" + (endMemo - startMemo) + " KB");
8.          return 0;
9.      }
10. }
```

四、 程序设计

1. 插入排序函数

```
class Insertion extends Sort {
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0 && less(a[j], a[j - 1]); j--) {
                exch(a, j, j - 1);
            }
        }
    }
}
```

2. 自顶向下归并排序函数

```
class MergeTD extends Sort {
    public static void merge(Comparable[] a, int lo, int mid, int hi) {
        Comparable[] aux = new Comparable[a.length];
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }

    }

    public static void sort(Comparable[] a) {
        sort(a, 0, a.length - 1);
    }

    public static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid + 1, hi);

        merge(a, lo, mid, hi);
    }
}
```

3. 自底向上归并排序函数

```
class MergeBU extends Sort {

    public static void merge(Comparable[] a, int lo, int mid, int hi) {
        Comparable[] aux = new Comparable[a.length];
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }
    }
```

```
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int len = 1; len < n; len *= 2) {
            for (int lo = 0; lo < n - len; lo += len + len) {
                int mid = lo + len - 1;
                int hi = Math.min(lo + len + len - 1, n - 1);
                merge(a, lo, mid, hi);
            }
        }
    }
}
```

4. 快速排序函数

```
class QuickSort extends Sort {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
        assert isSorted(a);
    }
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
    }
    private static int partition(Comparable[] a, int lo, int hi) {
        int i = lo;
        int j = hi + 1;
        Comparable v = a[lo];
        while (true) {
            while (less(a[++i], v)) {
                if (i == hi) break;
            }
            while (less(v, a[--j])) {
                if (j == lo) break;
            }
            if (i >= j) break;

            exch(a, i, j);
        }
        exch(a, lo, j);
        return j;
    }
}
```

5. 三路划分快速排序函数

```
class Quick3way extends Sort {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
        assert isSorted(a);
    }
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int lt = lo, gt = hi;
        Comparable v = a[lo];
        int i = lo + 1;
        while (i <= gt) {
            int cmp = a[i].compareTo(v);
            if (cmp < 0) exch(a, lt++, i++);
            else if (cmp > 0) exch(a, i, gt--);
            else i++;
        }
```

```
        sort(a, lo, lt - 1);
        sort(a, gt + 1, hi);
    }
}
```

6. 计算时间占用函数

```
public static double time(Comparable[] a) {
        for (int i = 0; i < 15; i++) {
            StdOut.println("第" + (i + 1) + "次: ");
            StdOut.println(" ");
            Comparable[] A1 = a.clone();
            Comparable[] B1 = a.clone();
            Comparable[] C1 = a.clone();
            Comparable[] D1 = a.clone();
            Comparable[] E1 = a.clone();

            long StartTime = System.nanoTime();
            Insertion.sort(A1);
            long endTime = System.nanoTime();
            StdOut.println("插入排序:");
            StdOut.println("时间占用->:"+(endTime - StartTime)/1000000.0 + " ms");

            StartTime = System.nanoTime();
            MergeBU.sort(B1);
            endTime = System.nanoTime();
            StdOut.println("自底向上归并:");
            StdOut.println("时间占用->:"+(endTime - StartTime)/1000000.0 + " ms");

            StartTime = System.nanoTime();
            MergeTD.sort(C1);
            endTime = System.nanoTime();
            StdOut.println("自顶向下归并:");
            StdOut.println("时间占用->:"+(endTime - StartTime)/1000000.0 + " ms");

            StartTime = System.nanoTime();
            QuickSort.sort(D1);
            endTime = System.nanoTime();
            StdOut.println("快速排序:");
            StdOut.println("时间占用->:"+(endTime - StartTime)/1000000.0 + " ms");

            StartTime = System.nanoTime();
            Quick3way.sort(E1);
            endTime = System.nanoTime();
            StdOut.println("3 路划分快排:");
            StdOut.println("时间占用->:"+(endTime - StartTime)/1000000.0 + " ms");
        }
        return 0;
}
```

7. 计算空间占用函数

```
public static double memo(Comparable[] a) {
        for (int i = 0; i < 15; i++) {
            StdOut.println("第" + (i + 1) + "次: ");
            StdOut.println(" ");
            Comparable[] A2 = a.clone();
            Comparable[] B2 = a.clone();
            Comparable[] C2 = a.clone();
            Comparable[] D2 = a.clone();
            Comparable[] E2 = a.clone();

            Runtime run = Runtime.getRuntime();
            run.gc();
            long startMemo = run.totalMemory() - run.freeMemory();
```

```
            Insertion.sort(A2);
            long endMemo = run.totalMemory() - run.freeMemory();
            StdOut.println("插入排序:");
            StdOut.println("辅助空间->:" + (endMemo - startMemo) / 1024 + " KB");

            run = Runtime.getRuntime();
            run.gc();
            startMemo = run.totalMemory() - run.freeMemory();
            MergeBU.sort(A2);
            endMemo = run.totalMemory() - run.freeMemory();
            StdOut.println("自底向上归并:");
            StdOut.println("辅助空间->:" + (endMemo - startMemo) / 1024 + " KB");

            run = Runtime.getRuntime();
            run.gc();
            startMemo = run.totalMemory() - run.freeMemory();
            MergeTD.sort(C2);
            endMemo = run.totalMemory() - run.freeMemory();
            StdOut.println("自顶向下归并:");
            StdOut.println("辅助空间->:" + (endMemo - startMemo) / 1024 + " KB");

            run = Runtime.getRuntime();
            run.gc();
            startMemo = run.totalMemory() - run.freeMemory();
            QuickSort.sort(D2);
            endMemo = run.totalMemory() - run.freeMemory();
            StdOut.println("快速排序:");
            StdOut.println("辅助空间->:" + (endMemo - startMemo) / 1024 + " KB");

            run = Runtime.getRuntime();
            run.gc();
            startMemo = run.totalMemory() - run.freeMemory();
            Quick3way.sort(E2);
            endMemo = run.totalMemory() - run.freeMemory();
            StdOut.println("3 路划分快排:");
            StdOut.println("辅助空间->:" + (endMemo - startMemo) / 1024 + " KB");
        }
        return 0;
}
```

8. 排序比较函数（主函数）

```
public class SortCompare {
    public static void main(String[] args) {
        StdOut.println("请输入待排序数量 N: ");
        int N = StdIn.readInt();
        Comparable array[] = new Comparable[N];

        for (int i = 0; i < N; i++) {
            array[i] = StdRandom.uniform(10000);
        }

        System.out.println(Arrays.toString(array));
        Comparable[] A1 = array.clone();
        Comparable[] B1 = array.clone();
        Comparable[] C1 = array.clone();
        Comparable[] D1 = array.clone();
        Comparable[] E1 = array.clone();

        Sort.calculate("插入排序", A1);
        Sort.calculate("自底向上归并排序", B1);
        Sort.calculate("自顶向下归并排序", C1);
        Sort.calculate("快速排序", D1);
        Sort.calculate("三路划分快速排序", E1);
    }
}
```

五、 实验结果记录

1. 随机数据（N=10000）

运行时间占用（ms）：

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	136.6	154.6	148.4	154.3	144.6	169.9	150.1	153.8	152.0	156.9	152.12
TDM	38.4	33.2	40.6	39.6	37.9	36.6	39.8	39.1	39.3	47.7	39.22
BUM	40.1	38.3	41.6	39.9	44.5	46.2	39.0	39.7	40.8	40.9	41.10
RQ	1.1	1.2	1.1	1.1	1.2	1.3	1.4	1.0	1.4	1.1	1.19
QD3P	1.9	2.0	1.9	1.9	2.1	2.0	2.4	1.8	2.0	2.0	2.00

表 5.1.1

运行空间占用（KB）：

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	0	0	0	0	0	0	0	0	0	0	0
TDM	65536	65536	65536	65536	65536	65536	65536	65536	65536	65536	65536
BUM	75303	74752	74791	74752	74752	74752	74752	74752	74752	74752	74806
RQ	0	0	0	0	0	0	0	0	0	0	0
QD3P	0	0	0	0	0	0	0	0	0	0	0

表 5.1.2

2. 大量递增数据（N=10000）

运行时间占用：

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	0.3	0.3	0.4	0.4	0.4	0.2	0.2	0.2	0.2	0.2	0.28
TDM	113.8	43.4	33.8	38.3	38.7	31.8	41.0	34.7	36.3	41.7	45.35
BUM	42.1	40.2	32.7	45.6	34.0	33.8	45.2	41.9	35.7	35.5	38.67
RQ	63.6	73.1	66.7	67.8	62.5	63.2	67.6	67.3	71.8	61.7	66.53
QD3P	7.6	6.9	7.1	7.4	8.1	8.5	7.4	7.1	8.6	7.7	7.64

表 5.2

表中可以看出，对于大量递增数据，插入排序算法要优异的多。

3. 少量随机数据（N=1000）

运行时间占用：

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	1.3	2.0	1.4	1.2	2.1	2.4	2.0	1.8	1.1	1.0	1.63
TDM	3.0	3.0	2.4	2.5	2.7	2.6	2.6	2.5	2.4	2.2	2.59
BUM	3.2	3.0	2.8	2.5	3.1	3.0	2.9	2.1	2.4	2.3	2.73
RQ	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.10
QD3P	0.2	0.2	0.2	0.1	0.2	0.2	0.2	0.1	0.1	0.1	0.16

表 5.3

六、 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?
哪种排序对不变或递增顺序的数据(即已经排序的数据)最有效?为什么你认为这种方法效果最好?

答：如表 5.2 所示，插入排序对不变或递增顺序的数据最有效。因为当待排数组有序时，只需要当前数跟前一个数比较一下就可以了，即每次排序只要比较 1 次即可确定，时间复杂度优化为 O(N)。

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	0.3	0.3	0.4	0.4	0.4	0.2	0.2	0.2	0.2	0.2	0.28
TDM	113.8	43.4	33.8	38.3	38.7	31.8	41.0	34.7	36.3	41.7	45.35
BUM	42.1	40.2	32.7	45.6	34.0	33.8	45.2	41.9	35.7	35.5	38.67
RQ	63.6	73.1	66.7	67.8	62.5	63.2	67.6	67.3	71.8	61.7	66.53
QD3P	7.6	6.9	7.1	7.4	8.1	8.5	7.4	7.1	8.6	7.7	7.64

表 5.2 大量（10000）递增数据各排序算法运行时间占用比较

2. Did the same sort do well on the case of mostly sorted data? Why or why not?
同样的排序算法在基本有序的数据中表现同样良好吗？说说你的理由。

答：不一定。数据的初始顺序对排序算法的性能有很大影响。例如，如表6.1所示，对于递增顺序的数据，快速排序的时间复杂度退化为$O(N^2)$。

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.
一般来说，传入数据的顺序会影响排序算法的性能吗？请引用表格中的具体数据来支持你的答案。

答：会。以快速排序为例，当传入的数据打乱无序时，测试结果平均运行时间为 1.19ms；当传入的数据有序时，测试结果平均运行时间为 66.53ms。

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.
对于少量数据(i.e., n = 1,000)来说，哪种排序方式最有效？这种排序对于大量数据而言也较为有效吗？请做出说明并用表格中具体的实验数据支撑你的观点。

答：实验结果如表 5.3 所示，对于少量（n=1000）数据，快速排序算法对于乱序数据在时间占用上表现突出，即使对于有序数据与插入排序算法也相差不大；对于大量（n=10000）数据，快速排序乱序时算法依旧表现突出，而有序时大大退化，完全不敌插入排序算法。

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	1.3	2.0	1.4	1.2	2.1	2.4	2.0	1.8	1.1	1.0	1.63
TDM	3.0	3.0	2.4	2.5	2.7	2.6	2.6	2.5	2.4	2.2	2.59
BUM	3.2	3.0	2.8	2.5	3.1	3.0	2.9	2.1	2.4	2.3	2.73
RQ	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.10
QD3P	0.2	0.2	0.2	0.1	0.2	0.2	0.2	0.1	0.1	0.1	0.16

表 5.3 各排序算法对少量（1000）乱序数据排序过程中的运行时间占用比较

5. In general, which sort did better? Give a hypothesis as to why the difference in performance exists.
总体来说，哪种排序算法性能最为优异？并对造成各个排序算法性能差异的原因给出你的猜想。

答：总体来说，三路划分的快速排序算法表现更为优异。因为三路划分快速排序算法作为随机快速排序算法的改进，虽然在运行时间上略逊于随机快速排序算法，但是在面对有较多重复键值的数据时，三路划分快速排序算法明显要优异的多。因此综合来说，三路划分快速排序算法应用面更为广泛，性能总体更为优异。

各个排序算法性能存在差异，是因为算法进程中中比较次数、交换次数、开辟的辅助空间大小等有差异。这些都对算法的时间复杂度和空间复杂度有很大的影响。

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?
你的记录表中是否有某些数据与其他数据不一致？（例如，在一次运行结果中，得到的时间占用列表为[1.3，1.5，1.6，7.0，1.2，1.6，1.4，1.8，2.0，1.5]，其中的数据“7.0”明显与其他数据不一致）。你认为为什么会出现这种现象？

答：在测试中出现了某些数据与其他数据不一致的现象，但我把他们当作坏值舍弃掉了，没有记录到表中。例如，在 n=10000 的乱序数据时间占用测试中，BUM 的平均时间占用为 41.10ms，而有一次的实验数据为 131.66ms。我认为，这种现象的发生体现了测试的偶然性。进程运行的时间由 CPU 性能释放、是否有其他进程抢占等许多复杂因素决定，任何一个因素的偶然改变都可能导致结果的巨大差距。