几种排序算法的实验性能比较

1 问题描述

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。

2 问题分析

写出 5 种排序算法，并随机生成 100000 个随机数分别用 5 种排序算法进行排序，记录运行时间和内存空间占用

2.1 运行时间

使用 Jar 包中的 Stopwatch 来记录时间。

2.2 空间占用

使用 Runtime 的 totalMemory() - freeMemory() 来得到开辟的额外内存空间大小。

3 程序实现

3.1 Insertion sort

```
public class Insertion {
    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);
```

```
        }
    }
}
```

3.2 Top-down mergesort

```
public class MergeTD {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        assert isSorted(a, lo, mid);
        assert isSorted(a, mid + 1, hi);
        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++];
        }
        assert isSorted(a, lo, hi);
    }

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

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

3.3 Bottom-up mergesort

```
public class MergeBU {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
```

```
            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;
            Comparable[] aux = new Comparable[N];
            for (int sz = 1; sz < N; sz = sz + sz)
                for (int lo = 0; lo < N - sz; lo += sz + sz) {
                    merge(a, aux, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N - 1));
                }
        }
    }
```

3.4 Random Quicksort and threewaysort

```
    public class Quick {
        private static int partition(Comparable[] a, int lo, int hi) {
            int i = lo, j = hi + 1;
            while (true) {
                while (less(a[++i], a[lo]))
                    if (i == hi) break;
                while (less(a[lo], a[--j]))
                    if (j == lo) break;
                if (i >= j) break;
                exch(a, i, j);
            }
            exch(a, lo, j);
            return j;
        }

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

```
    private static void sort(Comparable[] a, int lo, int hi) {
        if (lo >= hi) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
    }

    public static void threeWaySort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int lt = lo, gt = hi;
        Comparable v = a[lo];
        int i = lo;
        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++;
        }
        threeWaySort(a, lo, lt - 1);
        threeWaySort(a, gt + 1, hi);
    }
}
```

3.5 Heapsort

```
public class Heap {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int k = N / 2; k >= 1; k--)
            sink(a, k, N);
        while (N > 1) {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }

    private static void exch(Comparable a[], int i, int j) {
        Comparable t = a[i - 1];
        a[i - 1] = a[j - 1];
        a[j - 1] = t;
```

```
        }

        private static void sink(Comparable[] a, int k, int N) {
            while (2 * k <= N) {
                int j = 2 * k;
                if (j < N && less(a, j, j + 1)) j++;
                if (!less(a, k, j)) break;
                exch(a, k, j);
                k = j;
            }
        }
    }
```

4 实验结果

4.1 Comparison of running time of sorting algorithms (in Micro Seconds)

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	10398.0	10979.0	18723.0	28015.0	10406.0	9250.0	16982.0	27876.0	27279.0	26465.0	18637.3
TDM	72.0	32.0	32.0	31.0	31.0	29.0	27.0	31.0	25.0	28.0	33.8
BUM	77.0	25.0	37.0	43.0	43.0	39.0	43.0	41.0	37.0	32.0	41.7
RQ	72.0	59.0	49.0	45.0	30.0	24.0	27.0	28.0	32.0	21.0	38.7
QD3P	24.0	16.0	8.0	8.0	0.0	9.0	10.0	5.0	5.0	11.0	9.6
System	48.0	72.0	119.0	71.0	55.0	56.0	44.0	32.0	51.0	57.0	60.5
Heapsort	61.0	23.0	47.0	32.0	35.0	35.0	28.0	29.0	23.0	33.0	34.6

4.2 Comparison of space usage of sorting algorithms (in Kilo Bytes)

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TDM	390.64	390.64	390.64	390.64	485.44	390.64	390.64	390.64	390.64	485.44	409.60
BUM	390.64	390.64	390.64	390.64	485.44	390.64	390.64	0.00	390.64	390.64	361.06
RQ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
QD3P	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
System	418.14	435.39	469.42	435.39	475.97	612.96	323.34	461.73	387.36	488.72	450.84
Heapsort	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

5 结果分析

由实验结果可以看到插入排序是最慢的，且与其他几种排序 NlgN 的排序算法相差很大，但它内存占用最小，若内存空间紧张且对时间性能要求不高，可以选择插入排序。两种归并排序由于需要辅助数组，内存空间占用很大。快速排序和三路快排时间和空间性能都较为优秀。根据 System Sort 的空间和时间性能可以推断出 System Sort 是由归并排序实现的。

6 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?
   对于已经有序的序列，我认为插入排序表现最好，因为每次插入要处理的数据只需要一次比较就能确定位置。

2. Did the same sort do well on the case of mostly sorted data? Why or why not?
   不是这样，对于基本有序的数据，每种排序算法比较次数相差很大。

3. In general, did the ordering of the incoming data affect the performance of the sorting algorithms? Please answer this question by referencing specific data from your table to support your answer.
   输入数据序列的初始排列顺序确实对排序算法的性能有影响，比如随机快排和三路随机快排，当输入序列重复的数据较多时，三路快排明显比随机快排表现好很多。

4. Which sort did best on the shorter (i.e., n = 1,000) data sets? Did the same one do better on the longer (i.e., n = 10,000) data sets? Why or why not? Please use specific data from your table to support your answer.
   当序列较短时，插入排序表现较好，从表格中可以看出来，当序列长度很大时，快排表现最好。

5. In general, which sort did better? Give a hypothesis as to why the difference in performance exists.
   总的来说三项切分的快排性能最好。

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
   我认为这可能是环境而导致的偶然情况。

7 心得体会

从本次实验我体会到了不同数量级时间复杂度的算法在实际应用中的差别十分之大。除此之外，我还了解到由于输入数据的差别，不同的算法的表现也会出现很大的差别。