算法之排序问题

1. 初级排序算法

1.1 规则

1.1.1 排序算法简单介绍

排序问题关注的主要对象是重新排列数组元素的算法，其中每个元素都有一个主键，按照主键的大小进行对数组的重排序。因为对于排序问题的解决思路大体步骤一致，因此我们存在一个排序算法类的模板，主要包括sort()方法，以及辅助的less()和exch()方法*(可能还会其他辅助函数)*，以及示例用例main()。

对于不同算法的区分我们会有不同的类，如Insertion.sort()、Merge.sort()、Quick.sort()等等。

1.1.2 排序算法模板

```
public class Example{
    // 不同排序算法的主排序部分
    public static void sort(Comparable[] a)
    // 比较两个值的大小
    private static boolean less(Comparable v, Comprarble w) {
        return v.compareTo(w) < 0;
    }
    // 交换两个位置的值
    private static void exch(Comparable[] a, int i, int j) {
        Comparable t = a[i]; a[i] = a[j]; a[j] = t;
    }
    // 输出排序后的列表
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++)
            StdOut.print(a[i] + ' ');
        StdOut.println();
    }
    //  判断列表a是否有序
    public static boolean isSorted(Comparable[] a) {
        for (int i = 1; i < a.length; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }
    // 主程序测试
    public static void main (String[] args) {
        String[] a = In.readStrings();
        sort(a);
        assert isSorted(a);
        show(a);
    }
}
```

1.1.3 compareTo方法的实现

对于创建自己的数据类型时，我们只要实现Comparable接口就能够保证用例代码可以将其排序。要做到这一点，只需要实现一个CompareTo()方法来定义目标类型对象的自然次序；对于CompareTo()方法V<W,V=W和V>W三种情况，Java的习惯是在V.CompareTo(W)被调用时分别返回一个负整数，零和一个正整数（一般是-1，0，1）

```
public class Date implements Comparable<Date>{
    private final int day;
    private final int month;
    private final int year;
    public Date (int d, int m, int y) {
        day = d; month = m; year = y;
    }
    public int compareTo (Date that) {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
        if (this.month < that.month) return -1;
        // 类似实现年月日的比较
        // 其他返回相等
        return 0;
    }
}
```

1.2 选择排序

要点即是不断选择未排序列表中最小的元素提到最前面，因此注定效率并不是很高

1.2.1 步骤

第一步：找到最小的元素与第一个元素进行交换

第二步：在剩下的元素中找到第二小的元素与数组第二个元素进行交换

第三步：循环往复直到循环到最后一个元素

1.2.2 特点

运行时间与输入状态无关：并未考虑到数组**初始状态**，因此在对混乱和有序的数组进行排序时所用的排序时间是一样长的。

- **数据移动是最少的**：选择排序进行的交换是N次，即交换与数组大小成线性关系，这在其他任何算法中均不存在，其他成线性对数或是平方级别
- 对于长度为N的数组，选择排序需要进行比较$N^2/2$次，交换`N`次。

1.2.4 代码实现

```
public class Selection {
    public static void sort (Comparable[] a) {
        int N = a.length;
        for (int i= 0; i < N; i++) {
            // 将a[i]与a[i+1, ..., N-1]中最小的元素进行交换
            int min = i;
            for (int j = i; j < N; j++) {
                if (less(a[j], a[min])) min = j;
            }
            exch(a, i, min);
        }
    }
}
```

1.3 插入排序

要点是要像整理桥牌一样，把一个一个元素插入到已经有序的元素的适当位置，而已经有序的元素可能会被向右移，因此插入排序索引达到右端时即排序完成

1.3.1 方法

对于0~N-1之间的每一个i，将a[i]与a[0]与a[i-1]中比他小的所有元素依次交换，在i从左到右变化时，i左侧的元素总是有序的，因此当i到达最右侧时，即完成排序。

1.3.2 特点

- 插入排序与数组初始状态有关
- 平均情况下，比较需要$N^2/4$,交换需要$N^2/4$次。
- 最坏情况下，比较需要$N^2/2$次，交换需要$N^2/2$次。
- 最好情况下，比较需要N-1次比较，0次交换。

1.3.3 代码实现

```
public class Insertion {
    public static void sort (Comparable[] a) {
        // 将a[i]升序排序
        int N = a.length;
        for (int i = 0; i < N; i++) {
            // 将a[i]插入到a[i-1,...,0]之中去
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
        }
    }
}
```

1.4 排序算法的比较

要比较排序算法的优劣，通常情况下我们要比较其运行时间的长短，因此我们需要一个测量排序时间的一个类，即SortCompare类

以下为SortCompare类的模板

```
import com.google.common.base.Stopwatch;

public class SortCompare {
    public static double time (String alg, Double[] a) {
        Stopwatch timer = new Stopwatch();
        if (alg == "Insertion") Insertion.sort(a);
        if (alg == "Selection") Selection.sort(a);
        if (alg == "Shell") Shell.sort(a);
        ...
        return timer.elapsedTime();
    }
    public static timeRandomInput (String alg, int N, int T) {
        // 使用算法alg对T个长度为N的数组进行排序
        double total = 0.0;
        Double[] a = new Double[N];
        for (int t = 0; t < T; t++) {
            // 生成一个数组并进行一次排序
            for (int i = 0; i < N; i++)
                a[i] = StdRandom.uniform();
            total += time(alg, a);
        }
        return total;
    }
    public static void main (String[] args) {
        String alg1 = args[0];
        String alg2 = args[1];
        int T = Integer.parseInt(args[2]);
        int N = Integer.parseInt(args[3]);
        double t1 = timeRandomInput(alg1, N, T);
        double t2 = timeRandomInput(alg2, N, T);
    }
}

% 命令行：java SortCompare Insertion Selection 1000 100
```

1.5 希尔排序

实质上是对插入排序的改进，因为插入排序只能和临近的元素进行交换，因此注定效率并不是很高，而希尔排序存在h有序数组

1.5.1 希尔排序的特点

- 速度较快
- 代码量小
- 在常规情况下已经足够

1.5.2 代码模板

```
public class Shell {
    public static void sort(Comparable[] a) {
        // 将a[]按升序排序
        int N = a.length;
        int h = 1;
        while (h < N/3) h = h*3+1; // 1, 4, 13...
        while (h >= 1) {
            // 将数组变为h有序
            for (int i = h; i < N; i++) {
                //将a[i]插入到a[i-h],a[i-h*2]...中
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }
}
```

2. 归并排序

总体思路：将两个有序的数组合并成一个更大的数组

2.1 特点

- 优点：任意长度为*N*的数组排序所需时间与*NlogN*成正比
- 缺点：它所需的额外空间和N成正比

2.2 原地归并的抽象方法

需要辅助数组aux[]

2.2.1 步骤

1. 将所有元素复制到aux[]中
2. 归并回a[]中
3. 归并时进行四个判断
    1. 左半边用尽，则取右边的元素
    2. 右半边用尽，则取左边的元素
    3. 右半边当前元素小于左半边，取右半边元素
    4. 右半边当前元素大于等于左半边，取左半边元素

2.2.2 代码模板

```
public static merge(Comparable[] a, int lo, int mid, int hi) {
    // 将a[lo..mid]和a[mid+1..hi]归并
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    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++];
    }
}
```

2.3 自顶向下的归并排序

主要是通过递归的方法进行排序，体现分治思想

2.3.1代码模板

```
public class Merge
{
    private static Comparable[] aux; // 辅助数组
    public static void sort (Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, 0, a.length-1);
    }
    private static void sort (Comparable[] a, int lo, int hi) {
        // 将数组a[lo..hi]排序
        if (hi <= lo) return;
        int mid = lo + (hi - lo)/2;
        sort(a, lo, mid);  // 将左半边排序
        sort(a, mid+1, hi)  // 将右半边排序
        merge(a, lo, mid, hi); // 归并结果
    }
}
```

排序的数组长度从2，4，8，16。。

2.3.2 特点

- **对于长度为N的任意数组，自顶向下的归并排序需要*1/2NlgN*至*NlgN*次比较**
- **对于长度为N的任意数组，自顶向下的归并排序最多需要*6NlgN*次**

2.3.3 改进

- 对于小规模子数组使用插入排序
- 测试数组已经有序：if a[mid] <= a[mid+1]，认为已经有序，跳过merge()方法。

2.4 自底向上的归并排序

与自顶向下的分治思想相比，另一种思路是先归并那些微型数组，再成对归并得到子数组，最后合并

2.4.1 代码模板

```
public class MergeBU {
    private static Comparable[] aux;
    // merge()方法不变
    public static void sort (Comparable[] a) {
        // 进行lgN次两两归并
        int N = a.length;
        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, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

2.4.2 特点

- **对于长度为N的任意数组，自底向上的归并排序需要*1/2NlgN*至*NlgN*次比较**
- **对于长度为N的任意数组，自底向上的归并排序最多需要*6NlgN*次**

当数组长度为2的幂次时，自顶向下与自底向上比较和数组访问次数刚好一致

3. 快速排序

快速排序

1. 基本算法

- 归并排序将数组分为两个子数组分别排序，并将有序的子数组归并使得整个数组排序；
- 快速排序通过一个切分元素将数组分为两个子数组，左子数组小于等于切分元素，右子数组大于等于切分元素，将这两个子数组排序也就将整个数组排序了。

```
public class QuickSort<T extends Comparable<T>> extends Sort<T> {

    @Override
    public void sort(T[] nums) {
        shuffle(nums);
        sort(nums, 0, nums.length - 1);
    }

    private void sort(T[] nums, int l, int h) {
        if (h <= l)
            return;
        int j = partition(nums, l, h);
```

```
        sort(nums, l, j - 1);
        sort(nums, j + 1, h);
    }

    private void shuffle(T[] nums) {
        List<Comparable> list = Arrays.asList(nums);
        Collections.shuffle(list);
        list.toArray(nums);
    }
}
```

2. 切分

取 a[l] 作为切分元素，然后从数组的左端向右扫描直到找到第一个大于等于它的元素，再从数组的右端向左扫描找到第一个小于它的元素，交换这两个元素。不断进行这个过程，就可以保证左指针 i 的左侧元素都不大于切分元素，右指针 j 的右侧元素都不小于切分元素。当两个指针相遇时，将切分元素 a[l] 和 a[j] 交换位置。

```
private int partition(T[] nums, int l, int h) {
    int i = l, j = h + 1;
    T v = nums[l];
    while (true) {
        while (less(nums[++i], v) && i != h) ;
        while (less(v, nums[--j]) && j != l) ;
        if (i >= j)
            break;
        swap(nums, i, j);
    }
    swap(nums, l, j);
    return j;
}
```

3. 性能分析

快速排序是原地排序，不需要辅助数组，但是递归调用需要辅助栈。

快速排序最好的情况下是每次都正好将数组对半分，这样递归调用次数才是最少的。这种情况下比较次数为 CN=2CN/2+N，复杂度为 O(NlogN)。

最坏的情况下，第一次从最小的元素切分，第二次从第二小的元素切分，如此这般。因此最坏的情况下需要比较 N2/2。为了防止数组最开始就是有序的，在进行快速排序时需要随机打乱数组。

4. 算法改进

4.1 切换到插入排序

因为快速排序在小数组中也会递归调用自己，对于小数组，插入排序比快速排序的性能更好，因此在小数组中可以切换到插入排序。

4.2 三数取中

最好的情况下是每次都能取数组的中位数作为切分元素，但是计算中位数的代价很高。一种折中方法是取 3 个元素，并将大小居中的元素作为切分元素。

4.3 三向切分

对于有大量重复元素的数组，可以将数组切分为三部分，分别对应小于、等于和大于切分元素。

三向切分快速排序对于有大量重复元素的随机数组可以在线性时间内完成排序。

```
public class ThreeWayQuickSort<T extends Comparable<T>> extends QuickSort<T> {

    @Override
    protected void sort(T[] nums, int l, int h) {
        if (h <= l) {
            return;
        }
        int lt = l, i = l + 1, gt = h;
        T v = nums[l];
        while (i <= gt) {
            int cmp = nums[i].compareTo(v);
            if (cmp < 0) {
                swap(nums, lt++, i++);
            } else if (cmp > 0) {
                swap(nums, i, gt--);
            } else {
                i++;
            }
        }
        sort(nums, l, lt - 1);
        sort(nums, gt + 1, h);
    }
}
```

5. 基于切分的快速选择算法

快速排序的 partition() 方法，会返回一个整数 j 使得 a[l..j-1] 小于等于 a[j]，且 a[j+1..h] 大于等于 a[j]，此时 a[j] 就是数组的第 j 大元素。

可以利用这个特性找出数组的第 k 个元素。

该算法是线性级别的，假设每次能将数组二分，那么比较的总次数为 (N+N/2+N/4+..)，直到找到第 k 个元素，这个和显然小于 2N。

```
public T select(T[] nums, int k) {
    int l = 0, h = nums.length - 1;
    while (h > l) {
        int j = partition(nums, l, h);

        if (j == k) {
            return nums[k];

        } else if (j > k) {
            h = j - 1;
```

```
        } else {
            l = j + 1;
        }
    }
    return nums[k];
}
```

堆排序

4. 优先队列

在某些情况下，我们并没要求知道所有数据的排序情况，可能只需要知道最大元素，这时候要求两种基本操作：删除最大元素和插入元素，这种数据类型就叫做优先队列。

4.1 初级实现

1. 数组实现(无序)

   insert方法与栈push方法完全一样，要删除最大元素，添加一段类似于**选择排序**内循环的代码，将最大元素与边界元素交换然后删除它。

2. 数组实现(有序)

   另一种方法就是在insert方法中添加代码，将所有较大的元素向右移动一格使得数组保持有序(类似于**插入排序**)，那么删除操作就和pop方法一样了。

3. 链表表示法

4.2 堆的定义

定义：当一颗二叉树的每个节点都大于等于它的两个子节点时，他被称为堆有序。

4.2.1 二叉堆表示法

- **指针表示法**：父节点以及两个子节点，需要三个指针
- **数组**：若为完全二叉树，仅用数组便可以实现，父节点为k。则两个子节点位置分别为2k和2k+1，

4.3 由下至上的堆有序化(上浮)

某个节点变得比它的父节点更大

```
private void swim (int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```

3.4 由上至下的堆有序化(下沉)

某个节点变得比他的两个子节点更小了

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

4.5 多叉堆

4.6 堆排序

堆排序分为两个阶段：堆的构造阶段& 下沉排序阶段。

代码实现

```
public static void sort (int[] a)
{
    int N = a.length;
    for (k = N/2; k >= 1; k--) {
        sink(a, k, N);
    } // 堆有序
    while (N > 1) {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
} // 下沉排序
```