023 排序

作者：pzs233
原文链接：https://ld246.com/article/1537288591840
来源网站：链滴

23.1 选择排序

选择排序：假设要按升序排列一个数列。先找到数列中最小的数，然后将它第一个元素交换。接下来在剩下的数中找到最小数，将它和第二个元素交换，以此类推，直到数列中仅剩一个数为止。

```
public class SelectionSort {
    /** The method for sorting the numbers */
    public static double[] selectionSort(double[] list) {
        for (int i = 0; i < list.length - 1; i++) {
            // Find the minimum in the list[i...list.length-1]
            double currentMin = list[i];
            int currentMinIndex = i;

            for (int j = i + 1; j < list.length; j++) {
                if (currentMin > list[j]) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }

            // Swap list[i] with list[currentMinIndex] if necessary
            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
        return list;
    }
}
```

23.2 插入排序

为了将 元素 list[i] 插入 数组 list[0..i-1] 中，需要将 list[i] 存储在一个名为 currentElement 的临时变中。

如果 list[i-1]>currentElement，就将 list[i-1] 移到 list[i] ;如果 list[i-2]>currentElement,就将 list[i-2 移到 list[i-1]，依次类推，直到 list[i-k]<=currentElement 或者 k>i （传递的是排好序的数列的第个元素）。将 currentElement 赋值给 list[i-k+1]。

```
public class InsertionSort {
    public static int[] insertionSort(int[] list){
        for (int i = 1; i < list.length; i++){
            int currentElement = list[i];
            int k;

            for (k = i - 1; k > 0 && list[k] > currentElement; k--){
            list[k + 1] = list[k];
        }

        list[k + 1] = currentElement;
        }
```

```
        return list;
    }
}
```

插入排序算法的复杂度为 O(n^2)。

23.3 冒泡排序

冒泡排序算法需要历遍几次数组。

在每次历遍中，连续比较相邻的元素。如果某一对元素时降序，则互换它们的值；否则，保持不变。

由于较小的值像“气泡”一行逐渐浮向顶部，而较大的值沉向底部，所以称为冒泡排序（bubble sor）或下沉排序（sinking sort）。

第一次历遍之后，最后一个元素称为数组中最大数。第二遍历遍之后，倒数第二个元素称为数组中的二大数。整个过程持续到所有元素都已排好序。

如果某次历遍中没有发生交换，那么就不必进行下一次历遍，因为所有元素都已经排好序了。

```
public class BubbleSort {
    public static void bubbleSort(int[] list){
        boolean needNextPass = true;

        for (int k = 1; k < list.length && needNextPass; k++){
            needNextPass = false;
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]){
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;

                    needNextPass = true;
                }
            }
        }
    }

    public static void main(String[] args){
        int[] list = {2,4,5,3,5,62,4,5,2,99,48,47,458,1290};
        bubbleSort(list);
        for (int c: list){
            System.out.print(c);
        }
    }
}
```

最佳的情况下，算法只需历遍一次，冒泡排序的时间复杂度为O(n)；最差的需要进行 n-1 次，时间复度为 O(n^2)。

23.4 归并排序

归并排序算法将数组分为两半，对每部分递归地应用归并排序。在两部分都排好序后，对它们进行归

。

```
public class MergeSort {
    /** The method for sorting the numbers */
    public static void mergeSort(int[] list) {
        if (list.length > 1) {

            // Merge sort the first half
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);

            //Merge sort the second half
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);
            mergeSort(secondHalf);

            //Merge firstHalf with secondHalf into list
            merge(firstHalf, secondHalf, list);
        }
    }

    /** Merge two sorted lists */
    public static void merge(int[] list1, int[] list2, int[] temp){
        int current1 = 0;//Current index in list1;
        int current2 = 0;//Current index in list2;
        int current3 = 0;//Current index in temp;

        while (current1 < list1.length && current2 < list2.length){
            if (list1[current1] <list2[current2]){
                temp[current3++] = list1[current1++];
            } else {
                temp[current3++] = list2[current2++];
            }

            while (current1 < list1.length) {
                temp[current3++] = list1[current1++];
            }

            while (current2 < list2.length) {
                temp[current3++] = list2[current2++];
            }
        }
    }

    /** A test method */
    public static void main(String[] args) {
        int[] list = {3,4,5,65,67,7,34,3,345,88,99,766,553};
        mergeSort(list);
        for (int e: list){
            System.out.print(e + " ");
        }
    }
```

```
}
```

归并排序的复杂度为 O(nlogn)，由于 选择排序、插入排序和冒泡排序，因为这些排序算法的复杂度为 O n^2)。

23.5 快速排序

快速排序算法在数组中选择一个称为主元（pivot）的元素，将数组分为两部分，使得第一部分中的有元素都小于或等于主元，而第二部分中的所有元素都大于主元。

对第一部分递归地应用快速排序算法，然后对第二部分递归地应用快速排序算法。

```
public class QuickSort {
    public static void  quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
    }

    public static void quickSort(int[] list, int first, int last) {
        if (last > first) {
            int pivotIndex = partition(list, first, last);
            quickSort(list, first, pivotIndex - 1);
            quickSort(list, pivotIndex + 1, last);
        }
    }

    /** Partition the array list[first...last] */
    public static int partition(int[] list, int first, int last) {
        int pivot = list[first];
        int low = first + 1;
        int high =  last;

        while (high > low) {
            //Searching forward from left find the first number that is bigger than pivot
            while(low <= high && list[low] <= pivot){
                low++;
            }

            //Search backward from right find the first number that is smaller than pivot
            while (low <= high && list[high] > pivot){
                high--;
            }

            //Swap two elements in the list
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }

        while (high > first && list[high] >= pivot) {
            high--;
        }
```

```
        //Swap pivot with list[high]
        if (pivot > list[high]){
            list[first] = list[high];
            list[high] = pivot;;
            return high;
        } else {
            return first;
        }
    }

    /**A test method */
    public static void main(String[] args) {
        int[] list = {2,3,2,5,6,1,-2,3,14,12};
        quickSort(list);
        for (int e: list){
            System.out.print(e + " ");
        }
    }
}
```

23.6 堆排序

实现 Heap 类：

```
public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

    /** Create a default heap */
    public Heap() {
    }

    /** Create a heap from an array of objects */
    public Heap(E[] objects) {
      for (int i = 0; i < objects.length; i++)
        add(objects[i]);
    }

    /** Add a new object into the heap */
    public void add(E newObject) {
      list.add(newObject); // Append to the heap
      int currentIndex = list.size() - 1; // The index of the last node

      while (currentIndex > 0) {
        int parentIndex = (currentIndex - 1) / 2;
        // Swap if the current object is greater than its parent
        if (list.get(currentIndex).compareTo(
            list.get(parentIndex)) > 0) {
          E temp = list.get(currentIndex);
          list.set(currentIndex, list.get(parentIndex));
          list.set(parentIndex, temp);
        }
        else
          break; // the tree is a heap now
```

```
      currentIndex = parentIndex;
    }
  }

  /** Remove the root from the heap */
  public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
      int leftChildIndex = 2 * currentIndex + 1;
      int rightChildIndex = 2 * currentIndex + 2;

      // Find the maximum between two children
      if (leftChildIndex >= list.size()) break; // The tree is a heap
      int maxIndex = leftChildIndex;
      if (rightChildIndex < list.size()) {
        if (list.get(maxIndex).compareTo(
            list.get(rightChildIndex)) < 0) {
          maxIndex = rightChildIndex;
        }
      }

      // Swap if the current node is less than the maximum
      if (list.get(currentIndex).compareTo(
          list.get(maxIndex)) < 0) {
        E temp = list.get(maxIndex);
        list.set(maxIndex, list.get(currentIndex));
        list.set(currentIndex, temp);
        currentIndex = maxIndex;
      }
      else
        break; // The tree is a heap
    }

    return removedObject;
  }

  /** Get the number of nodes in the tree */
  public int getSize() {
    return list.size();
  }
}
```

实现堆排序：

```
public class HeapSort {
  /** Heap sosrt method */
  public static <E extends Comparable<E>> void heapSort(E[] list) {
      //Create a Heap of integers
```

```
        Heap<E> heap = new Heap<>();

        //Add elements to the heap
        for (int i = 0; i < list.length; i++) {
            heap.add(list[i]);
        }

        //Remove elements from the heap
        for (int i = list.length - 1; i >= 0; i--) {
            list[i] = heap.remove();
        }
    }

    /**A test method */
    public static void main(String[] args) {
        Integer[] list = {-44,-5,-3,3,3,33,1,-4,0,1,23,99};
        heapSort(list);
        for (int e: list) {
            System.out.print(e + " ");
        }
    }
}
```

堆排序的时间复杂度为 O(nlogn)。

23.7 桶排序和基数排序

桶排序和基数排序是 **对整数** 进行的排序的高效算法。

桶排序的时间复杂度为 O(n+t)，基数排序的时间复杂度为 O(dn)。

23.8 外部排序

时间复杂度为 O(nlogn)。

```
import java.io.*;

public class SortLargeFile {
  public static final int MAX_ARRAY_SIZE = 43;
  public static final int BUFFER_SIZE = 100000;

  public static void main(String[] args) throws Exception {
    // Sort largedata.dat to sortedfile.dat
    sort("largedata.dat", "sortedfile.dat");

    // Display the first 100 numbers in the sorted file
    displayFile("sortedfile.dat");
  }

  /** Sort data in source file and into target file */
  public static void sort(String sourcefile, String targetfile)
      throws Exception {
    // Implement Phase 1: Create initial segments
```

```
  int numberOfSegments =
    initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");

  // Implement Phase 2: Merge segments recursively
  merge(numberOfSegments, MAX_ARRAY_SIZE,
    "f1.dat", "f2.dat", "f3.dat", targetfile);
}

/** Sort original file into sorted segments */
private static int initializeSegments
    (int segmentSize, String originalFile, String f1)
    throws Exception {
  int[] list = new int[segmentSize];
  DataInputStream input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(originalFile)));
  DataOutputStream output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f1)));

  int numberOfSegments = 0;
  while (input.available() > 0) {
    numberOfSegments++;
    int i = 0;
    for ( ; input.available() > 0 && i < segmentSize; i++) {
      list[i] = input.readInt();
    }

    // Sort an array list[0..i-1]
    java.util.Arrays.sort(list, 0, i);

    // Write the array to f1.dat
    for (int j = 0; j < i; j++) {
      output.writeInt(list[j]);
    }
  }

  input.close();
  output.close();
  return numberOfSegments;
}

private static void merge(int numberOfSegments, int segmentSize,
    String f1, String f2, String f3, String targetfile)
    throws Exception {
  if (numberOfSegments > 1) {
    mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
    merge((numberOfSegments + 1) / 2, segmentSize * 2,
      f3, f1, f2, targetfile);
  }
  else { // Rename f1 as the final sorted file
    File sortedFile = new File(targetfile);
    if (sortedFile.exists()) sortedFile.delete();
    new File(f1).renameTo(sortedFile);
  }
}
```

```
private static void mergeOneStep(int numberOfSegments,
    int segmentSize, String f1, String f2, String f3)
    throws Exception {
  DataInputStream f1Input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
  DataOutputStream f2Output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));

  // Copy half number of segments from f1.dat to f2.dat
  copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
  f2Output.close();

  // Merge remaining segments in f1 with segments in f2 into f3
  DataInputStream f2Input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
  DataOutputStream f3Output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));

  mergeSegments(numberOfSegments / 2,
    segmentSize, f1Input, f2Input, f3Output);

  f1Input.close();
  f2Input.close();
  f3Output.close();
}

/** Copy first half number of segments from f1.dat to f2.dat */
private static void copyHalfToF2(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataOutputStream f2)
    throws Exception {
  for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
    f2.writeInt(f1.readInt());
  }
}

/** Merge all segments */
private static void mergeSegments(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataInputStream f2,
    DataOutputStream f3) throws Exception {
  for (int i = 0; i < numberOfSegments; i++) {
    mergeTwoSegments(segmentSize, f1, f2, f3);
  }

  // f1 may have one extra segment, copy it to f3
  while (f1.available() > 0) {
    f3.writeInt(f1.readInt());
  }
}

/** Merges two segments */
private static void mergeTwoSegments(int segmentSize,
  DataInputStream f1, DataInputStream f2,
  DataOutputStream f3) throws Exception {
```

```
    int intFromF1 = f1.readInt();
    int intFromF2 = f2.readInt();
    int f1Count = 1;
    int f2Count = 1;

    while (true) {
      if (intFromF1 < intFromF2) {
        f3.writeInt(intFromF1);
        if (f1.available() == 0 || f1Count++ >= segmentSize) {
          f3.writeInt(intFromF2);
          break;
        }
        else {
          intFromF1 = f1.readInt();
        }
      }
      else {
        f3.writeInt(intFromF2);
        if (f2.available() == 0 || f2Count++ >= segmentSize) {
          f3.writeInt(intFromF1);
          break;
        }
        else {
          intFromF2 = f2.readInt();
        }
      }
    }

    while (f1.available() > 0 && f1Count++ < segmentSize) {
      f3.writeInt(f1.readInt());
    }

    while (f2.available() > 0 && f2Count++ < segmentSize) {
      f3.writeInt(f2.readInt());
    }
  }

  /** Display the first 100 numbers in the specified file */
  public static void displayFile(String filename) {
    try {
      DataInputStream input =
        new DataInputStream(new FileInputStream(filename));
      for (int i = 0; i < 100; i++)
        System.out.print(input.readInt() + " ");
      input.close();
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```