```java
package com.bo.offer;

import java.nio.channels.NonWritableChannelException;
import java.util.Collection;
import java.util.Collections;
import java.util.LinkedList;

import com.bo.sort.Heap;

public class Sort {

    /**
     * @param data
     * @idea each iteration choose the min value and exchange to sorted head
     *       sequence
     * @compare not use the input data information, scan the whole sequence
     *          anyway. but without moving a lot only exchange in linear space
     */
    public static void SelectSort(int[] data) {
        for (int i = 0; i < data.length - 1; i++) {
            int min = data[i];
            // use k to store min index and check if need to exchange
            int k = i;
            for (int j = i + 1; j < data.length; j++) {
                if (data[j] < min) {
                    min = data[j];
                    k = j;
                }
            }
            // exchange
            if (k != i) {
                int temp = data[i];
                data[i] = min;
                data[k] = temp;
            }
        }
    }

    /**
     * @param data
     */
    public static void BubbleSort(int[] data) {
        boolean isswap;
        // i index control n-1 rounds not the data index
        for (int i = 1; i < data.length; i++) {
            isswap = false;
            for (int j = 0; j < data.length - i; j++) {
                if (data[j] < data[j + 1]) {
                    int temp = data[j];
                    data[j] = data[j + 1];
                    data[j + 1] = temp;
                    isswap = true;
```

```
                }

            }
            if (isswap == false) {
                break;
            }
        }
    }

    /**
     * @param data
     * @idea insert data[i] into proper position at data[0]...data[i-1],
     * @idea back trace and exchange while not sorted fit for partially sorted
     *       data and small number of dataset
     */
    public static void InsertSort(int[] data) {
        for (int i = 1; i < data.length; i++) {
            for (int j = i; j > 0 && data[j] < data[j - 1]; j--) {
                int temp = data[j - 1];
                data[j - 1] = data[j];
                data[j] = temp;
            }
        }
    }

    /**
     * @param data
     * @idea based on insert sort, exchange non adjacent data to sort the local
     *       sequence and sort those local later
     * @compare trade off the scale and effectivity, partition the large
     *          sequence small and partially sorted.. keep the h distance
     *          subsequence sorted suit for large scale of data, non extra
     *          storage
     */
    public static void ShellSort(int[] data) {
        int n = data.length;
        int h = 1;
        while (h < n / 3)
            h = 3 * h + 1; // 1,4,13,40. vary the distance h with input data
                           // length
        while (h >= 1) {
            // h sorted
            for (int i = h; i < data.length; i++) {
                for (int j = i; j >= h && data[j] < data[j - h]; j -= h) {
                    int temp = data[j];
                    data[j] = data[j - h];
                    data[j - h] = temp;
                }

            }
            // update step
            h = h / 3;
        }
    }
```

```
107.
108.     /**
109.      * @param data
110.      * @param low
111.      * @param high
112.      * @param temp
113.      *            recursive merge within nlogn use a global int[] temp to store
114.      *            sorted sequence during merging process, because in every
115.      *            recursive allocate a array would cause problem. running time
116.      *            is the depth of the tree saving time but waste storage
117.      */
118.     public static void MergeSort(int[] data, int low, int high, int[] temp) {
119.         if (data != null && low < high && temp.length == data.length) {
120.             int mid = (low + high) / 2;
121.             MergeSort(data, low, mid, temp);
122.             MergeSort(data, mid + 1, high, temp);
123.             MergeArray(data, low, mid, high, temp);
124.         }
125.     }
126.
127.     /**
128.      * @param data
129.      * @param low
130.      * @param mid
131.      * @param high
132.      * @param temp
133.      *            merge two sorted sequence data[low...mid] and
134.      *            data[mid+1...high] and keep them sorted top down merge method.
135.      *            there is also a bottom up one
136.      *
137.      */
138.     public static void MergeArray(int[] data, int low, int mid, int high, int[] te
    mp) {
139.         int i = low;
140.         int j = mid + 1;
141.         int k = low;
142.         while (i <= mid && j <= high) {
143.             if (data[i] < data[j])
144.                 temp[k++] = data[i++];
145.             else
146.                 temp[k++] = data[j++];
147.         }
148.         while (i <= mid)
149.             temp[k++] = data[i++];
150.         while (j <= high)
151.             temp[k++] = data[j++];
152.
153.         for (i = low; i <= high; i++) {
154.             data[i] = temp[i];
155.         }
156.     }
157.
158.     /**
159.      * @param data
```

```
160.          *            best explanation:
161.          *            http://developer.51cto.com/art/201403/430986.htm
162.          */
163.         public static void QuickSort(int[] data, int start, int end) {
164.             if (data != null && start < end) {
165.                 int q = Partition(data, start, end);
166.                 QuickSort(data, start, q - 1);
167.                 QuickSort(data, q + 1, end);
168.             }
169.         }
170.
171.         /**
172.          * @param data
173.          * @param start
174.          * @param end
175.          * @return
176.          */
177.         public static int Partition(int[] data, int start, int end) {
178.             int i = start;
179.             int j = end;
180.             int pivot = data[start];
181.             while (i < j) {
182.                 while (i < j && data[j] >= pivot)
183.                     j--;
184.                 if (i < j) {
185.                     // dig out the first bigger in right to left, do not worry
186.                     // data[i]
187.                     // it's stored in pivot from begin
188.                     data[i] = data[j];
189.                     i++;
190.                 }
191.                 while (i < j && data[i] <= pivot)
192.                     i++;
193.                 if (i < j) {
194.                     data[j] = data[i];
195.                     j--;
196.                 }
197.             }
198.             // iterate until i==j
199.             data[i] = pivot;
200.             return i;
201.         }
202.
203.         /**
204.          * @param data
205.          * @param start
206.          * @param end
207.          *            another version
208.          */
209.         public static void Quick_Sort(int[] data, int start, int end) {
210.             if (start < end && data != null) {
211.                 // partition
212.                 int i = start, j = end, pivot = data[start];
213.                 while (i < j) {
```

```
                while (i < j && data[j] >= pivot)
                    j--;
                if (i < j) {
                    data[i++] = data[j];
                }
                while (i < j && data[i] <= pivot)
                    i++;
                if (i < j) {
                    data[j--] = data[i];
                }
            }
            data[i] = pivot;
            Quick_Sort(data, start, i - 1);
            Quick_Sort(data, i + 1, end);
        }

    }

    /**
     * @param data
     *            we use array to store a Heap, if root index is 0 then it's
     *            left child index is 2*i+1 and right child index is 2*i+2 else
     *            if root index is 1, then left child index is 2*i and right
     *            child index is 2*i+1 check here:
     *            http://www.cnblogs.com/mengdd/archive/2012/11/30/2796845.html
     */
    public static void Heap_Sort(int[] data) {
        Build_Max_Heap(data);
        int len = data.length;
        int Heap_Size = data.length;
        for (int i = len - 1; i >= 0; i--) {
            // exchange A[0] with A[i] to swap large to end
            int temp = data[0];
            data[0] = data[i];
            data[i] = temp;
            Heap_Size--;
            Max_Heapfy(data, 0, Heap_Size);
        }
    }

    /**
     * @param data
     *            from the first non leaf down to root because leaf node always
     *            keep the heap properties
     */
    public static void Build_Max_Heap(int[] data) {
        for (int i = data.length / 2 - 1; i >= 0; i--) {
            Max_Heapfy(data, i, data.length);
        }
    }

    /**
     * @param data
     * @param i
```

```
268.         * @param Heap_Size
269.         *            use to control length of heap, used in Heap_Sort after
270.         *            Build_Heap. After swapping the root to last, last node is
271.         *            sorted as largest and don't need to Heapfy again. keep heap
272.         *            properties of node i, recursively select the largest node to i
273.         *            position
274.         */
275.        public static void Max_Heapfy(int[] data, int i, int Heap_Size) {
276.            int left = 2 * i + 1;
277.            int right = 2 * i + 2;
278.            int large;
279.            if (left < Heap_Size && data[left] > data[i])
280.                large = left;
281.            else
282.                large = i;
283.            if (right < Heap_Size && data[right] > data[large])
284.                large = right;
285.            // swap data[i] with data[large]
286.            if (large != i) {
287.                int temp = data[i];
288.                data[i] = data[large];
289.                data[large] = temp;
290.                Max_Heapfy(data, large, Heap_Size);
291.            }
292.        }
293.
294.        /**
295.         * @param A
296.         *            in put data where max num is k
297.         * @param k,
298.         *            max of input data
299.         * @return count sort first count those num's count and store the count at
300.         *         position num then change count array to store how many num little
301.         *         or equal than position index, by plusing count before current
302.         *         index all done http://blog.jobbole.com/74574/
303.         */
304.        public static int[] CountSort(int[] A, int k) {
305.            if (A == null || k < 1)
306.                return null;
307.            int[] B = new int[A.length];
308.            int[] C = new int[k + 1]; // counting array
309.            for (int i = 0; i < A.length; i++) {
310.                C[A[i]] = C[A[i]] + 1;
311.            }
312.            // how many small or equal i
313.            for (int i = 1; i < C.length; i++) {
314.                C[i] = C[i - 1] + C[i];
315.            }
316.            // how many small or equal i mean i stays at position how many - 1
317.            for (int i = 0; i < A.length; i++) {
318.                B[C[A[i]] - 1] = A[i];
319.                C[A[i]] = C[A[i]] - 1;
320.            }
321.            return B;
```

```
322.     }
323. 
324.     /**
325.      * @param data
326.      * list array to store is wonderful use
327.      */
328.     public static void BucketSort(double[] data) {
329.         if (data == null)
330.             System.out.println("Err with no input data");
331.         // a list array. jesus crist
332.         LinkedList[] arrList = new LinkedList[data.length];
333.         for (int i = 0; i < data.length; i++) {
334.             int floor = (int) Math.floor(10 * data[i]);
335.             if (arrList[floor] == null)
336.                 arrList[floor] = new LinkedList<Double>();
337.             arrList[floor].add(data[i]);
338.         }
339. 
340.         // in each bucket, we sort them, sort the linked list
341.         for (int i = 0; i < arrList.length; i++) {
342.             if (arrList[i] != null) {
343.                 Collections.sort(arrList[i]);
344.             }
345.         }
346.         // result
347.         int count = 0;
348.         for (int i = 0; i < arrList.length; i++) {
349.             if (arrList[i] != null) {
350.                 for (int j = 0; j < arrList[i].size(); j++) {
351.                     data[count++] = (double) arrList[i].get(j);
352.                 }
353.             }
354.         }
355.     }
356. 
357.     public static void main(String[] args) {
358.         int[] data = { 3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48 };
359.         double array[] = { 0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0
    .68 };
360.         // SelectSort(data);
361.         // InsertSort(data);
362.         // ShellSort(data);
363.         // int[] b = new int[data.length];
364.         // MergeSort(data, 0, data.length-1, b);
365.         // Quick_Sort(data, 0, data.length-1);
366.         // BubbleSort(data);
367.         // Heap_Sort(data);
368.          int[] result = CountSort(data, 50);
369.         //BucketSort(array);
370.         for (int i : result) {
371.             System.out.print(i + ",");
372.         }
373.     }
374. 
```

```
375.   }
```