

Data Structures and Algorithms – Week 2 Code Samples

Lecturer: Dr Anwarul Patwary

This zip file contains code samples we will refer to during week 2.

To understand the code samples well, you should make use of the textbook, which is *Data Structures and Problem Solving Using Java* (4th edition) by Mark Allen Weiss.

The outline for the Data Structures and Algorithms course contains suggested readings from the textbook that will help you understand the code here.

If there are sections of the textbook you particularly should read for some code listings, I will mention them here.

Four Sorting Algorithms

```
1
2
3 /**
4  * Utility class offering several sorting algorithms
5  */
6
7 public class SortingAlgorithms {
8     /**
9      * Execute the insertion sort algorithm sorting the argument
10     * array. There is no return since the parameter is mutated.
11     *
12     * @param arr the array of short integers to be sorted
13     */
14     public static void insertionSort(short[] arr) {
15         for (int j = 1; j < arr.length; j++) {
16             short key = arr[j];
17             int i = j - 1;
18             while (i >= 0 && arr[i] > key) {
19                 arr[(i + 1)] = arr[i];
20                 i = i - 1;
21             }
22             arr[i + 1] = key;
23         }
24     }
25
26     /**
27     * Execute the selection sort algorithm on an argument array.
28     * There is no return as the parameter is mutated.
29     * @param arr the array of short (numbers) to be sorted
30     */
31     public static void selectionSort(short[] arr) {
32         //build the sorted portion from 0 upwards
33         for (int i = 0; i < arr.length - 1; i++) {
34             //initialise the minimum value and its position
35             short min = arr[i];
36             int minpos = i;
37             //search the unsorted part for its smallest element
38             for (int j = i + 1; j < arr.length; j++) {
39                 if (arr[j] < min) {
40                     min = arr[j];
41                     minpos = j;
42                 }
43             }
44             if (i != minpos) { // swap min into place if necessary
45                 short temp = arr[i]; //copy
46                 arr[i] = arr[minpos]; //transfer
47                 arr[minpos] = temp; //replace
48             }
49         }
50     }
51 }
```

```

50     }
51
52
53
54
55
56
57     /**
58      * Execute the merge sort algorithm sorting the argument array.
59      * There is no return as the parameter is to be mutated.
60      * @param arr the array of short integers to be sorted
61      */
62     public static void mergeSort(short[] arr) {
63         mergeSort(arr, 0, arr.length - 1);
64     }
65
66     private static void mergeSort(short[] arr, int l, int r) {
67         if (l < r) {
68             int mid = (l + r) / 2;
69             mergeSort(arr, l, mid);
70             mergeSort(arr, mid + 1, r);
71             merge(arr, l, mid, r);
72         }
73     }
74
75     /**
76      * Merge two parts of array arr from l to mid and mid+1 to r inclusive
77      * @param arr the array containing the portions for merging
78      * @param l left index, used for portion 1 from l to mid
79      * @param mid mid index, user for portion 2 from mid+1 to r
80      * @param r right index
81      */
82     private static void merge(short[] arr, int l, int mid, int r) {
83         int lsize = mid - l + 1;
84         int rsize = r - mid;
85         short[] left = new short[lsize];
86         short[] right = new short[rsize];
87
88         for (int i = 0; i < lsize; i++) {
89             left[i] = arr[l + i];
90         }
91         for (int j = 0; j < rsize; j++) {
92             right[j] = arr[mid + 1 + j];
93         }
94         int i = 0;
95         int j = 0;
96         int k = l;
97         while (i < lsize && j < rsize) {
98             if (left[i] < right[j]) {
99                 arr[k++] = left[i++];
100             } else {

```

```

101         arr[k++] = right[j++];
102     }
103 }
104 while ( i < lsize ) { // Copy rest of first half
105     arr[k++] = left[i++];
106 }
107 while( j < rsize ) { // Copy rest of second half
108     arr[k++] = right[j++];
109 }
110 }
111
112 /**
113  * Execute the quicksort algorithm sorting the argument array.
114  * There is no return as the parameter is to be mutated.
115  * @param arr the array of short integers to be sorted
116  */
117 public static void quickSort(short[] arr) {
118     quickSort(arr, 0, arr.length - 1);
119 }
120
121 /**
122  * Overloads the quickSort method with parameters to set the range to be sorted
123  */
124 private static void quickSort(short[] arr, int p, int r) {
125     if (p < r) {
126         int q = partition(arr, p, r);
127         quickSort(arr, p, q - 1);
128         quickSort(arr, q + 1, r);
129     }
130 }
131
132 /**
133  * A private method to partition the array arr,
134  * between the indices start and finish inclusive
135  * inclusive. The method selects the element arr[r] as the pivot.
136  *
137  * @param arr the array to be sorted, which is mutated by the method
138  * @param p the lower index of the range to be partitioned
139  * @param r the upper index of the range to be partitioned
140  * @return the index of the point of partition
141  */
142 public static int partition(short[] arr, int start, int finish) {
143     short fence = arr[start]; //get the pivot value
144     int left = start+1;
145     int right = finish;
146     while (right >= left) {
147         while (left <= right && arr[left] <= fence)
148             left++;
149         while (right >= left && arr[right] >= fence)
150             right--;
151         if (right > left) {

```

```

152         short swap = arr[left];
153         arr[left] = arr[right];
154         arr[right] = swap;
155     }
156 }
157 arr[start] = arr[right];
158 arr[right] = fence;
159
160 return right; //return position of the fence
161 }
162
163
164
165
166 }

```

Three Search Algorithms

```

1  /**
2   * Three search algorithms
3   * Sequential and Step with
4   * Binary search from Weiss Fig 5.11
5   */
6  public class SearchAlgorithms {
7
8      public static int NOT_FOUND = -1;
9      public static boolean DEMO = true; //true to print debug statements
10
11     /**
12      * search for an item in an array this is the slowest search,
13      * O(N) but also the simplest
14      *
15      * @param a array to be searched, assumed to be sorted
16      * @param key item being searched for
17      * @return index of key in a if key is found, and NOT_FOUND otherwise
18      */
19     public static int SequentialSearch(int[] a, int key) {
20         for (int i = 0; i < a.length; i++) {
21             if (DEMO) { System.out.println("Testing□position□"+i); }
22             if (a[i] == key) { // found it!
23                 return i;
24             }
25         }
26         return NOT_FOUND; // if we get here, the item was not found
27     }
28
29     /**
30      * step search for an item in an array
31      * this is slightly faster than sequential search but still O(N)
32      *
33      * @param a array to be searched, assumed to be sorted

```

```

34  * @param key item being searched for
35  * @param step step size for moving through array
36  * @return index of key in a if key is found, and NOT_FOUND otherwise
37  */
38  public static int StepSearch(int[] a, int key, int step) {
39      int i = 0;
40      while (i < a.length) {
41          if (DEMO) { System.out.println("Testing□position□"+i); }
42          if (a[i] == key) { // found it
43              return i;
44          }
45          if (a[i] < key) { // flip forwards
46              i = Math.min(i + step, a.length - 1);
47          } else { // search back through the block
48              for (int j = i - 1; j > i - step; j--) {
49                  if (DEMO) { System.out.println("Testing□position□"+j); }
50                  if (a[j] == key) {
51                      return j;
52                  }
53              }
54              return NOT_FOUND;
55          }
56      }
57      return NOT_FOUND; // if we get here, the item was not found
58  }
59
60  /**
61   * Binary search to find key in array a is  $O(\log N)$ 
62   *
63   * @param a array to be searched, assumed to be sorted
64   * @param key item being searched for
65   * @return index of key in a if key is found, and NOT_FOUND otherwise
66   */
67  public static int BinarySearch(int[] a, int key) {
68      int low = 0;
69      int high = a.length - 1;
70      int mid;
71
72      while (low <= high) {
73          mid = (low + high) / 2;
74          // more generally ( a[ mid ].compareTo( x ) < 0 )
75          if (DEMO) { System.out.println("Testing□position□"+mid); }
76          if (a[mid] < key) { // continue to search lower part
77              low = mid + 1;
78          } else if (a[mid] > key) { // continue to search upper part
79              high = mid - 1;
80          } else { // we've found it
81              return mid;
82          }
83      }
84      // if we get here, the item was not found

```

```

85     return NOT_FOUND; // NOT_FOUND = -1
86 }
87
88 /**
89  * test code
90  */
91 public static void main(String[] args) {
92     //some test cases to see binary search at work
93     int[] a = new int[] { 11, 102, 223, 254, 265, 306, 367, 388, 399, 1000 };
94     int[] testkeys = new int[] { 265, 388, 1000, 200 };
95     for (int key : testkeys) {
96         if (DEMO) { System.out.println("Sequential_Search_for_" + key); }
97         int res1 = SequentialSearch(a, key);
98         if (DEMO) { System.out.println("Step_Search_for_" + key); }
99         int res2 = StepSearch(a, key, 3);
100        if (DEMO) { System.out.println("Binary_Search_for_" + key); }
101        int res3 = BinarySearch(a, key);
102        System.out.println("Found_key=" + key + "at_position" +
103            res1 + "," + res2 + "," + res3);
104    }
105 }
106
107 }

```