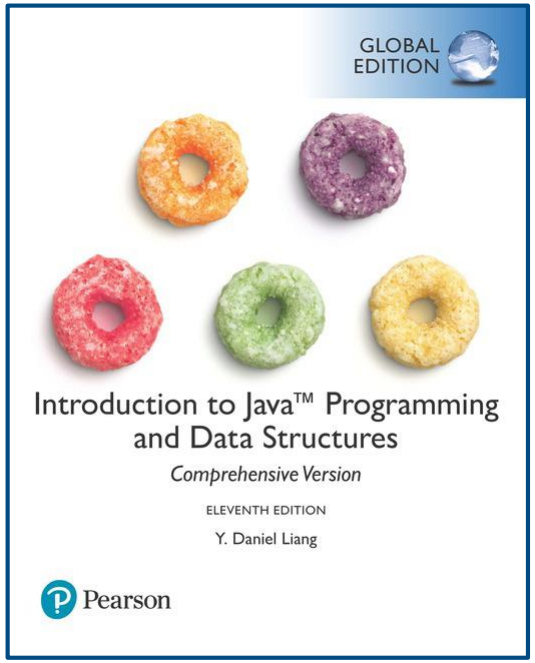
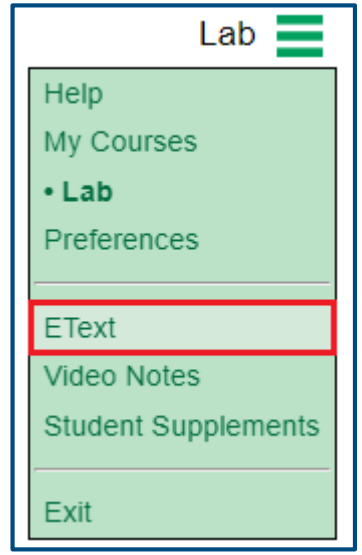
COMP 201 Data Structures and Algorithms
Lab8 – Comparing Execution Times for Sorting Algorithms

Sorting Algorithms



- Sorting is a common task in computer programming and many different algorithms have been developed for sorting.
- Sorting algorithms are good examples for studying algorithm design and analysis.
- For this lab work, sorting algorithm implementations are taken from the book "Introduction to Java Programming and Data Structures (11th Edition) by Y. Daniel Liang, Pearson.".
- You can access the book in Pearson MyProgrammingLab (EText) as shown on the right. In the book, the sorting algorithms are presented using illustrations, explanations, links to animations and time complexity analyses.
  - Selection sort → Section 11 of Chapter 7 in the book
  - Other sorting algorithms → Chapter 23 in the book
- Each implementation included in the provided code CompareSortingTimes.java sorts a given integer array in increasing order of the values.



Given Code

- CompareSortingTimes.java contains the methods that implement seven sorting algorithms and a method to test these implementations (shown on the right).

```java
// Method for testing different sorting algorithm implementations
// -----------------------------------------------------------------------
public static void testSortingImplementations(int[] list, int numDigits) {
   // create an integer array to preserve the contents of the given array for
   // using the same array with different sorting algorithm implementations
   int[] toSort = new int[list.length];
   // print the given array before sorting
   System.out.println("Array before sorting: " + Arrays.toString(list));
   // use each sorting algorithm implementation for sorting the array and
   // print the array after it is sorted
   String[] names = { "Selection", "Insertion", "Bubble", "Merge", "Quick", "Heap", "Radix" };
   for (int i = 0; i < 7; i++) {
      // copy the contents of the given array list to the array toSort
      System.arraycopy(list, 0, toSort, 0, list.length);
      // select the sorting algorithm to use based on the value of i
      switch (i) {
         case 0: selectionSort(toSort); break;
         case 1: insertionSort(toSort); break;
         case 2: bubbleSort(toSort); break;
         case 3: mergeSort(toSort); break;
         case 4: quickSort(toSort); break;
         case 5: heapSort(toSort); break;
         case 6: radixSort(toSort, numDigits);
      }
      // print the sorted array
      System.out.println(names[i] + " sort: " + Arrays.toString(toSort));
   }
}
```

Main Method

- The main method of the given code CompareSortingTimes.java and its output are shown below.

```java
public static void main(String[] args) {
    // test the implementations for different sorting algorithms
    // ---------------------------------------------------------------------
    // an array to be sorted
    int[] testArray = { 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9 };
    // the number of the digits of the largest element in the array
    int digits = 3; // needed for using radix sort
    // invoke the method that sorts the given array using different sorting
    // algorithm implementations
    testSortingImplementations(testArray, digits);
    System.out.println(); // print an additional new line

    // Write your code here.
}
```

Console Output:

```
Array before sorting: [331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9]
Selection sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Insertion sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Bubble sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Merge sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Quick sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Heap sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Radix sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
```

The Task for This Lab Work

- Print a table for comparing the execution times of the given sorting algorithm implementations in the code CompareSortingTimes.java. The details are in the next slide.
- Write your code in the main method of the CompareSortingTimes class to produce a similar console output as the one shown below.

```
public static void main(String[] args) {
    // test the implementations for different sorting algorithms
    // ------------------------------------------------------------------
    // an array to be sorted
    int[] testArray = { 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9 };
    // the number of the digits of the largest element in the array
    int digits = 3; // needed for using radix sort
    // invoke the method that sorts the given array using different sorting
    // algorithm implementations
    testSortingImplementations(testArray, digits);
    System.out.println(); // print an additional new line

    // Write your code here.
}
```

Console Output:

```
Array before sorting: [331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9]
Selection sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Insertion sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Bubble sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Merge sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Quick sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Heap sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
Radix sort: [9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454]
```

Array Size	Selection Sort	Insertion Sort	Bubble Sort	Merge Sort	Quick Sort	Heap Sort	Radix Sort
10000	18.00	16.20	123.80	1.70	0.90	7.90	5.20
20000	56.30	39.70	512.90	3.20	1.60	5.60	5.40
30000	138.20	89.00	1219.00	5.00	2.60	9.80	8.40
40000	300.60	173.40	2305.30	5.90	3.80	15.20	11.30
50000	392.90	248.70	3409.30	7.80	4.40	18.00	12.40
60000	551.80	364.90	4944.50	8.50	5.30	21.70	15.10
70000	754.90	482.80	6752.20	11.60	6.00	27.00	17.10
80000	965.00	629.30	8717.20	13.20	7.00	28.00	19.80
90000	1287.30	822.00	11082.70	13.90	8.00	33.10	24.60
100000	1533.80	1009.80	13812.30	17.00	9.10	40.20	26.00

The Task for This Lab Work

1. Generate a random integer array of size N with values in the range [0, 1.000.000).
2. Sort this array using each given sorting algorithm implementation.
   - **Hint:** You can use two arrays: one to store the generated random values and one for sorting. After each sorting operation, you can reset the array used for sorting with the values stored in the other array.
3. Measure the execution time for each sorting algorithm.
   - **Hint:** You can use System.currentTimeMillis() to get the time before and after sorting.
4. Repeat the steps above for 10 times, compute the average execution time for each sorting algorithm and print the computed average execution time on the console.
5. Repeat the steps above for different array sizes N = [10.000, 20.000, 30.000, ... , 90.000, 100.000].

Submission of Lab Work

- Submit your **Java code file(s)** (no report required) to Blackboard.
- You can work as a team of 2 students, if you like. Each team member should upload his/her code to Blackboard individually.

Grading
Your lab work will be graded on a scale of 0: Incorrect/NA, 1: Partially Correct and 2: Correct.