TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE
CPSC 215: Data Structures and Algorithms
Instructor: Dr. Chandranil Chakraborttii
Spring 2025

Laboratory 9

Academic Honesty Policy

In working on programming assignments, you may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. Programming assignments are expected to be the work of the individual student, designed, and coded by him or her alone. Violations are easy to identify and will be dealt with promptly according to the Academic Integrity and Intellectual Dishonesty outlined in the Student Handbook.

- Copying another person's programs or encouraging or assisting another person to commit plagiarism is cheating. In particular, the following activities are strictly prohibited:
- Giving and receiving help in the actual development of code or writing of an assignment.
- Looking at another person's code or showing your code to another person.
- Sharing a copy of all or part of your code regardless of whether that copy is on paper or in a computer file.
- Turning in the work of any other person(s) (former students, friends, textbook authors, people on the Internet, etc.) and representing it as your own work.
- Fabricating compilation or execution results.
- Use of AI ChatBots for any help regarding the assignment is **strictly prohibited.**

The penalty for cheating is a failing grade (F) for the course, and the student is asked to appear at an Academic Dishonesty Hearing. In addition, the College also places a record of the incident in the student's permanent record. It is your responsibility to protect your work from unauthorized access. Do not discard copies of your programs in public places. Do not leave computers unattended and copies of output lying around.

An Empirical Comparison of Sorting Algorithms

Objectives

- To get familiar with various sorting methods
- To conduct performance analysis on different sorting methods

---

Part I: Which sorting algorithm is fastest?

Asymptotic complexity analysis lets us distinguish between O(n^2) and O (n log n) algorithms, but it does not help distinguish between algorithms with the same asymptotic complexity. Nor does asymptotic analysis say anything about which algorithm is best for sorting small lists. For answers to these questions, we can turn to empirical testing.

Review the Java program (Sorting.java) which sorts an array A of *n* random integers using the following sorting methods:

1. bubbleSort(A)
2. insertionSort(A)
3. shellSort(A)
4. mergeSort(A, T, 0, n-1)
5. quickSort(A, 0, n-1)
6. Heapsort(A)

where T is a temporary array of the same size as A. All these sorting methods must be defined in Sorting.java. All sorting methods are completed for you in this assignment.

Input Files (Starter Code)

- Sorting.java ***(No Need to change)***
- SortTest.java – Junit Test file ***(Update as directed)***

Tasks:

In the SortTest.java file, you will be testing the sorting algorithms defined in the Sorting class and recording their timings. Here's what the SortTest class does:

- **Main Method**: The main method is the entry point of the program. Inside the main method, the following steps are performed for each specified array size:

    - Generate a random array of N integers using the generateRandomArray method.
    - Copy the generated array using the copyArray method. This is done to ensure that each sorting algorithm sorts the same input array.
    - Record the start time using System.nanoTime().
    - Call each sorting algorithm with the copied array and record the execution time.
    - Output the execution time for each sorting algorithm.
    - Repeat these steps for each specified array size.

- **Helper Methods**:
    - **generateRandomArray:** Generates a random array of integers of the specified size.
    - **copyArray:** Creates a copy of the given array.

In the main method (defined in SortTest.java), you should complete the code to measure running time (in nanoseconds) as follows:

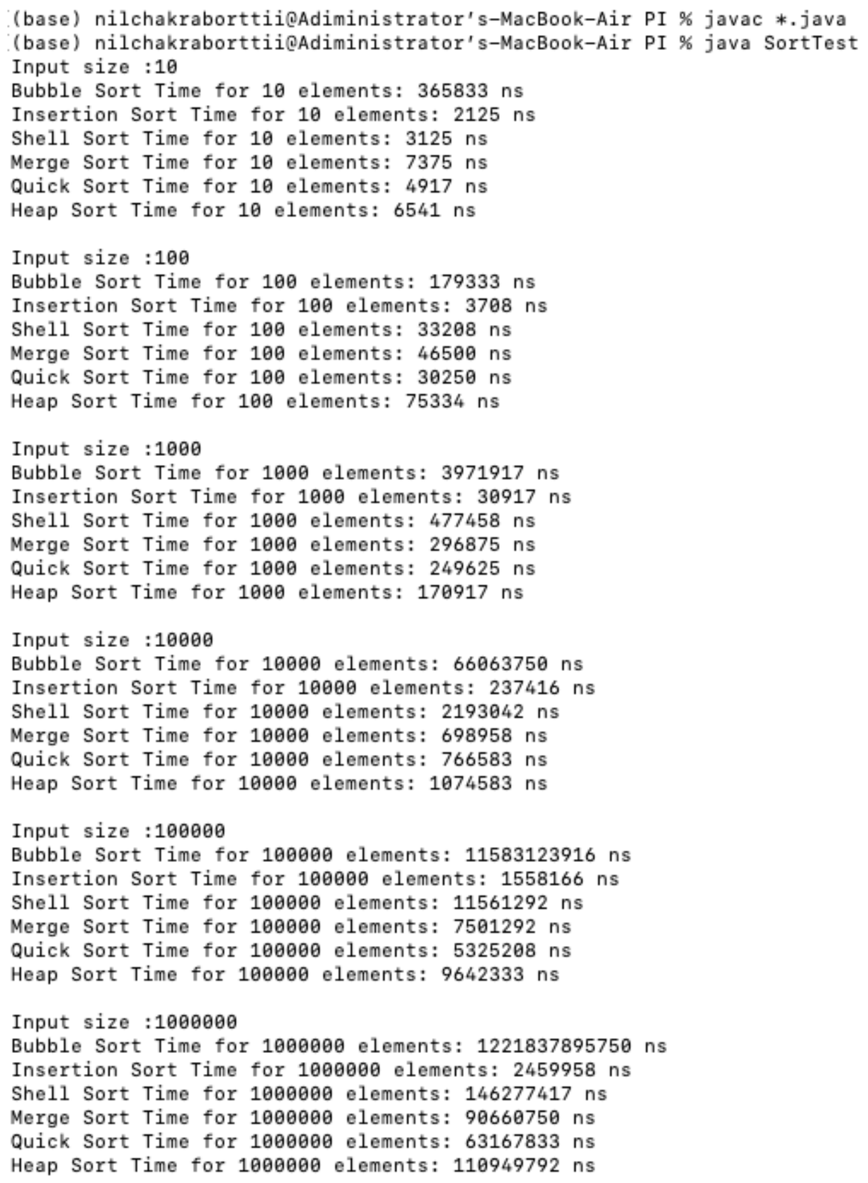
```
... create an array of n random numbers ...
long start = System.nanoTime();
... sort ...
long end = System.nanoTime();
System.out.println("Time: " + (end - start)/ " ns.");


...
```

Record running times for different values of *n* in the following table:

Method	**N=100**	**N=1000**	**N=10000**	**N=100000**	**N=1000000**
InsertionSort					
BubbleSort					
ShellSort					
MergeSort					
QuickSort					
HeapSort					

Expected Output:



```
(base) nilchakraborttii@Adiministrator's-MacBook-Air PI % javac *.java
(base) nilchakraborttii@Adiministrator's-MacBook-Air PI % java SortTest
Input size :10
Bubble Sort Time for 10 elements: 365833 ns
Insertion Sort Time for 10 elements: 2125 ns
Shell Sort Time for 10 elements: 3125 ns
Merge Sort Time for 10 elements: 7375 ns
Quick Sort Time for 10 elements: 4917 ns
Heap Sort Time for 10 elements: 6541 ns

Input size :100
Bubble Sort Time for 100 elements: 179333 ns
Insertion Sort Time for 100 elements: 3708 ns
Shell Sort Time for 100 elements: 33208 ns
Merge Sort Time for 100 elements: 46500 ns
Quick Sort Time for 100 elements: 30250 ns
Heap Sort Time for 100 elements: 75334 ns

Input size :1000
Bubble Sort Time for 1000 elements: 3971917 ns
Insertion Sort Time for 1000 elements: 30917 ns
Shell Sort Time for 1000 elements: 477458 ns
Merge Sort Time for 1000 elements: 296875 ns
Quick Sort Time for 1000 elements: 249625 ns
Heap Sort Time for 1000 elements: 170917 ns

Input size :10000
Bubble Sort Time for 10000 elements: 66063750 ns
Insertion Sort Time for 10000 elements: 237416 ns
Shell Sort Time for 10000 elements: 2193042 ns
Merge Sort Time for 10000 elements: 698958 ns
Quick Sort Time for 10000 elements: 766583 ns
Heap Sort Time for 10000 elements: 1074583 ns

Input size :100000
Bubble Sort Time for 100000 elements: 11583123916 ns
Insertion Sort Time for 100000 elements: 1558166 ns
Shell Sort Time for 100000 elements: 11561292 ns
Merge Sort Time for 100000 elements: 7501292 ns
Quick Sort Time for 100000 elements: 5325208 ns
Heap Sort Time for 100000 elements: 9642333 ns

Input size :1000000
Bubble Sort Time for 1000000 elements: 1221837895750 ns
Insertion Sort Time for 1000000 elements: 2459958 ns
Shell Sort Time for 1000000 elements: 146277417 ns
Merge Sort Time for 1000000 elements: 90660750 ns
Quick Sort Time for 1000000 elements: 63167833 ns
Heap Sort Time for 1000000 elements: 110949792 ns
```

Part II: Counting Sort

You can sort a large array of integers that are in the range 1 to *n* by using an array count of *n* entries to count the number of occurrences of each integer in the array. For example, consider the following array of 14 integers that range from 1 to 9:

```
9 2 4 8 9 4 3 2 8 1 2 7 2 5
```

Form an array count of 9 elements such that count[i-1] contains the number of times that i occurs in the array to be sorted. Thus, count is

```
1 4 1 2 1 0 1 2 2
```

We now know that 1 occurs once in the original array, 2 occurs four times, and so on. Thus, the sorted array is:

```
1 2 2 2 2 3 4 4 5 7 8 8 9 9
```

Input Files (Starter Code)

- Lab8P2.java – Junit Test file ***(Need to change)***

Tasks:

- Review the class (Lab8P2.java) containing methods for implementing the counting sort algorithm as directed. Description of each method is provided.
- If needed, you may take help of any code above.
- Using Big-Oh notation, describe the efficiency of this algorithm.
- Is this algorithm useful as a general sorting algorithm? Explain.

Here's a description of each method in countingSort.java:

- **countingSort(int[] arr)**:
  - This method performs the counting sort algorithm on the input array.
  - It takes an integer array arr as input and returns the sorted array.
  - Within this method, you need to find the maximum element in the input array to determine the size of the count array.
  - Then, create a count array to store the frequency of each element.
  - Iterate through the input array and increment the count for each element in the count array.
  - Finally, populate the sorted array based on the count array.

- **getCountArray(int[] arr)**:
  - This method generates the count array required by the counting sort algorithm.
  - It takes an integer array arr as input and returns the count array.

  - Similar to countingSort method, find the maximum element in the input array.
  - Create a count array to store the frequency of each element.
  - Iterate through the input array and increment the count for each element in the count array.
  - Return the count array.

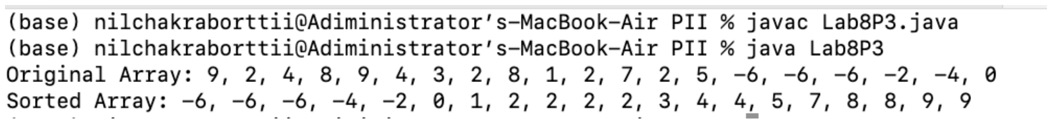
- **findMax(int[] arr)**:
  - This method finds the maximum element in the input array.
  - It takes an integer array arr as input and returns the maximum element.
  - Initialize a variable max to the minimum integer value.
  - Iterate through the input array and update max if a larger element is found.
  - Return the maximum element.

- **printArray(int[] arr)**:
  - This method prints the elements of an integer array.
  - It takes an integer array arr as input and does not return any value.
  - Iterate through the array and print each element separated by a comma.

Extra Credit (10 points)

- Modify code so that the program can accept both positive and negative numbers.

Expected Output:



```
(base) nilchakraborttii@Adiministrator's-MacBook-Air PII % javac Lab8P3.java
(base) nilchakraborttii@Adiministrator's-MacBook-Air PII % java Lab8P3
Original Array: 9, 2, 4, 8, 9, 4, 3, 2, 8, 1, 2, 7, 2, 5, -6, -6, -6, -2, -4, 0
Sorted Array: -6, -6, -6, -4, -2, 0, 1, 2, 2, 2, 2, 3, 4, 4, 5, 7, 8, 8, 9, 9
```

——

Submission Guidelines:

- Make sure you completed all sections with "To do" statements.
- Make sure you have added JavaDoc in your code for each class you need to complete.
- Upon the completion of your lab, upload your submission to the Moodle course website as a single Zip file.
- See your TA and get your Assignment graded before leaving the lab. Otherwise, inform the instructor.

——