CME 2204 Assignment-1

Comparison of Heapsort, Shellsort and Introsort

Due: 09.04.2022, 23:59

Rules:

- The submissions will be checked for the code similarity. Plagiarism will be graded as zero. You must declare if you are using a code directly copied from a public web site (e.g., Wikipedia, GitHub etc.)
- You are required to upload two different files: the source code you have written in Java programming language and the report.
- The files you are required to upload are given below with explanations:

(STUDENT_NUMBER)_(STUDENT_NAME)_HW1_Code
(Source code you have written for the HW1)

Example = 2043901815_Augusta_Ada_HW1_Code

(STUDENT_NUMBER)_(STUDENT_NAME)_HW1_Report.pdf

Example = 2043901815_Augusta_Ada_HW1_Report.pdf

Part 1.
In this assignment you will be implementing and testing all three sorting algorithms:
Heapsort, **Shellsort and Introsort.** You will write a class and own methods. Sorting algorithm methods will be invoked from this class;

```
public class SortingClass {

    heapSort (int[] arrayToSort) { .... }
    introSort (int[] arrayToSort) { .... }
    shellSort (int[] arrayToSort) { .... }

}
```

In the *heapSort* method, you must implement a **maximum heap** to sort the given array.

In the *shellSort* method, you must implement an improved version of the insertion sort algorithm. Unlike insertion sort, instead of comparing contiguous items, the Shellsort breaks the master array into several subarrays using an interval "i" (which is known as the gap), then sorts the subarrays with insertion sort. The above steps are repeated until the gap is 1.

An example run of Shellsort with gaps 5, 3 and 1 is shown below.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on five separate subarrays (*a1, a6, a11*), (*a2, a7, a12*), (*a3, a8*), (*a4, a9*), (*a5, a10*). For instance, it changes the subarray (*a1, a6, a11*) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on three subarrays (*a1, a4, a7, a10*), (*a2, a5, a8, a11*), (*a3, a6, a9, a12*). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (*a1 ,..., a12*).

Researchers have published different approaches for gap sequences. You can use Shell's original sequence, which is ⌊N/2 , N/4 , …, 1⌋.

Pseudocode:

```
shellSort(array)
// size:array length,  n: general term (n ≥ 1)
  for interval i <- size/2^n down to 1
     for each interval "i" in array
        sort all the elements at interval "i"
end shellSort
```

In the *introSort* method, you must implement a hybrid sorting algorithm. Heap sort and quick sort algorithms should be used while developing this sorting method. It is called as *introsort* or *introspective* sort, which begins quicksort and switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted. The maximum recursion depth is defined as follows.

$$maxdepth = [\ log(length(Input))] \times 2$$

Introsort combines the good parts of both algorithms. C++ standard libraries, Microsoft .Net platform and Swift are using this algorithm.

Introsort steps:

a) Calculate the *maxdepth*
b) Select the pivot (choose the last item as a pivot)
c) Partition the input array
c) Repeat step *b* and *c* recursively until *maxdepth* = 0
d) If *maxdepth* = 0 invoke heapsort

Pseudocode:

```
procedure sort(A : array):
    maxdepth = [log(length(A))] × 2
    introsort(A, maxdepth)

procedure introsort(A, maxdepth):
    n ← length(A)
    p ← partition(A) // pivot selection, p final pivot position

    if n ≤ 1:
        return // base case
    else if maxdepth = 0:
        heapsort(A)
    else:
        introsort(A[0:p], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

Part 2.

You are expected to compare each sorting algorithm according to their running times (in milliseconds) for different inputs and fill the following table accordingly.

	EQUAL INTEGERS			RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000
heapSort												
shellSort												
introsort												

- You must create the input arrays by yourself in the Test class according to the specified rules (1000, 10000, 100000 of each equal, random, increasing, and decreasing order).

- Note that, the elapsed time of creating the arrays should not be considered during the calculation of the total running time of sorting algorithms. You only have to check how long the actual sorting algorithm is running.

- One of the methods you can use to measure the time elapsed in Java is given below:

```
n=1000     // array size
int arrayToSort = new int[n];
for (int i = 0 ; i < n ; i++)

        arrayToSort[i] = i;   // generate numbers in increasing order

long startTime = System. currentTimeMillis();
heapSort(arrayToSort);          // run one of the sorting methods
long estimatedTime = System. currentTimeMillis() - startTime;
```

Part 3.

Prepare a scientific report in the light of the results obtained from your program and the information you learned in the class. Establish a connection between the asymptotic running time complexity (in Θ notation) and the results (in milliseconds) of your experiments. Your report should include a comparison table (as shown in the previous page). Additionally, you are expected to determine the appropriate sorting algorithm for the following scenario. Which algorithm would you choose and why? Explain your thoughts and reasons clearly and broadly.

Scenario: We aim to place students at universities according to their central exam grades and department preferences. If there are millions of students in the exam, which sorting algorithm would you use to do this placement task faster?

Grading Policy

Part	**Grade %**
HeapSort	15
Introsort	25
ShellSort	20
Running Time Computation	25
Report	15