

# CME 2204 Assignment-1

## Comparison of Heapsort, dualPivotQuicksort and Shellsort

**Due : 05.04.2020, 23:59**

### Rules :

- The submissions will be checked for the code similarity. Plagiarism will be graded as zero.
- Research assistant of the course will check your assignments in their offices. You will have 10 minutes to present your assignments.
- You are required to bring your laptops and reports while coming to the assignment control.
- Control dates will be announced later.

**1.** In this assignment you will be implementing and testing all three sorting algorithms:

**Heapsort, dualPivotQuickSort and Shellsort.** You will write a class and own methods. Sorting algorithm methods will be invoked from this class;

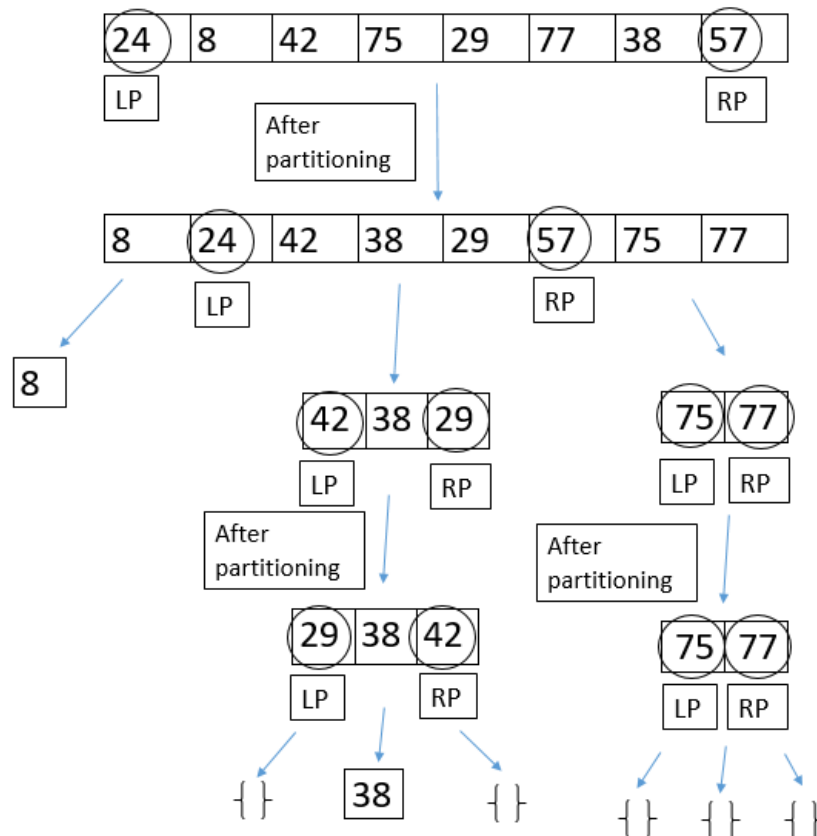
```
public class SortingClass {  
  
    heapSort (int[] arrayToSort) { .... }  
    dualPivotQuickSort (int[] arrayToSort) { .... }  
    shellSort (int[] arrayToSort) { .... }  
  
}
```

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

In the *dualPivotQuickSort* method, you must implement a different version of quicksort. This version has two pivots: one in the left end and one in the right end of the array. The left pivot (LP) must be less than or equal to the right pivot (RP).

< LP	LP	LP ≤ & ≤ RP	RP	RP <
------	----	-------------	----	------

Input array is divided into three parts. In the first part, all elements will be less than LP, in the second part all elements will be greater or equal to LP and also will be less than or equal to RP, and in the third part all elements will be greater than RP. Until the existing pivots have stopped their moving, sorting process continues recursively.



In the *shellSort* method, you must implement an improved version of the insertion sort algorithm. Unlike insertion sort, instead of comparing contiguous items, it 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.

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

The first pass, 5-sorting, performs insertion sort on five separate subarrays ( $a_1, a_6, a_{11}$ ), ( $a_2, a_7, a_{12}$ ), ( $a_3, a_8$ ), ( $a_4, a_9$ ), ( $a_5, a_{10}$ ). For instance, it changes the subarray ( $a_1, a_6, a_{11}$ ) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on three subarrays ( $a_1, a_4, a_7, a_{10}$ ), ( $a_2, a_5, a_8, a_{11}$ ), ( $a_3, a_6, a_9, a_{12}$ ). The last pass, 1-sorting, is an ordinary insertion sort of the entire array ( $a_1, \dots, a_{12}$ ).

Researchers have published different gap sequences approaches. You can use Shell's original sequence, which is  $\lfloor N/2 \rfloor, \lfloor N/4 \rfloor, \dots, 1$ .

**Pseudocode:**

```

shellSort(array)
// size:array length, n: general term ( $n \geq 1$ )
for interval  $i \leftarrow \text{size}/2^n$  down to 1
    for each interval "i" in array
        sort all the elements at interval "i"

end shellSort

```

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
<i>heapSort</i>												
<i>dualPivot QuickSort</i>												
<i>shellSort</i>												

- 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 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;

```

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  $\Theta$  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 preferences. If there are millions of students in the exam, which sorting algorithm would you use to do this placement task faster?

### Grading Policy

Item	%
heapSort	15
dualPivotQuickSort	25
shellSort	20
Running Time Computation	25
Report	15