# CME 2204 Assignment-1

## Comparison of Heapsort, dualPivotQuicksort and Bucketsort

## Due to: 07.05.2021, 23:55

**Rules**

- The submissions will be checked for the code similarity. Plagiarism will be graded as zero.
- You are required to upload two different files. One is 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**

**1.** In this assignment you will be implementing and testing all three sorting algorithms: **Heapsort**, **dualPivotQuickSort** and **Bucketsort**. 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) { .... }
        bucketSort (int[] arrayToSort) { .... }

}
```

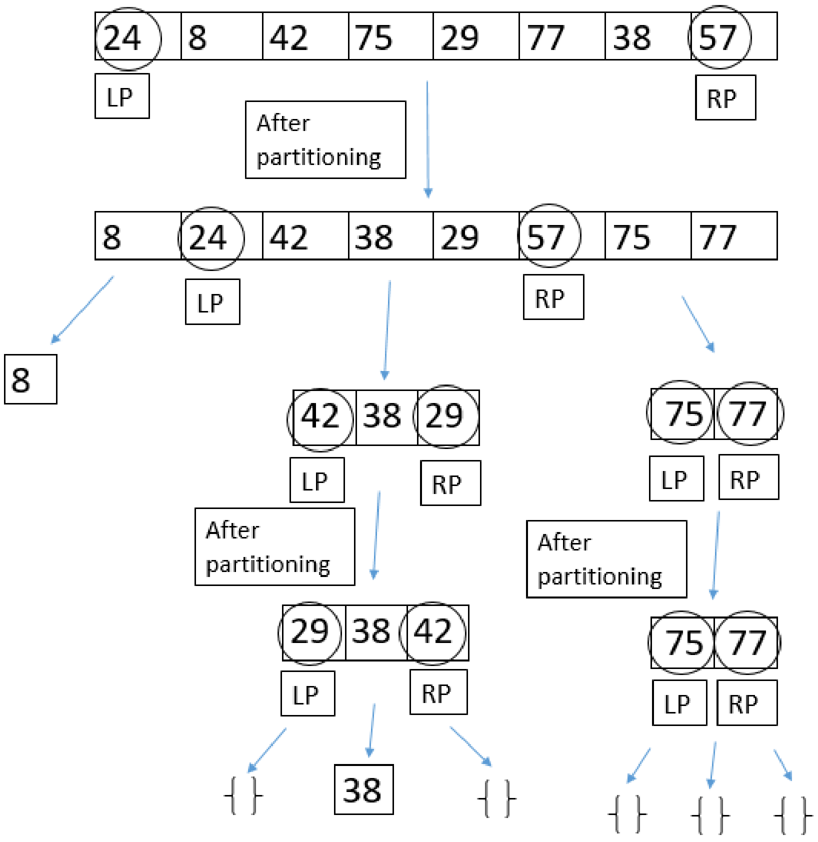
In the *heapSort* method, you must implement a **maximum heap** to sort the given array.

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).

| | | | | |
|---|---|---|---|---|
| $< \text{LP}$ | LP | $\text{LP} \leq \ \& \ \leq \text{RP}$ | RP | $\text{RP} <$ |

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

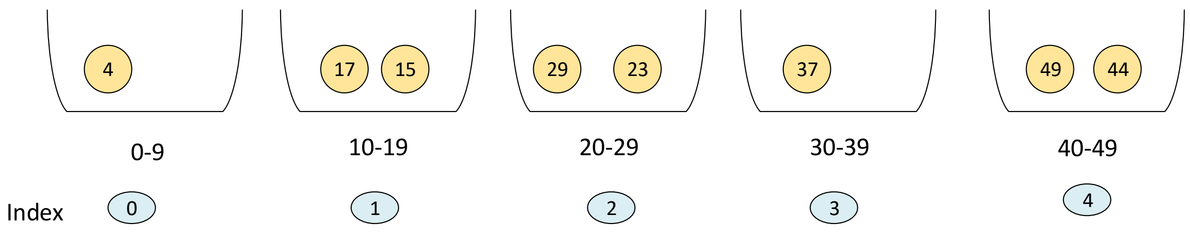
RP, and in the third part all elements will be greater than RP. Until the existing pivots have stopped their moving, sorting process continues recursively.



In the *bucketSort* method, you must create ***n*** empty buckets. This sorting works by distributing the elements into appropriate buckets of fixed interval. And then, you should apply individually insertion sort to sort elements in each bucket. At the last step, to create a final sorted list, you should concatenate the sorted buckets together.

An example run of bucket sort with these numbers, {**4, 17, 37, 29, 23, 49, 44, 15}**, is shown below.

1) Suppose we want to do bucket sort with five buckets (*n=5*)
2) Largest value is close to 50
3) Each bucket should have an interval of (*the largest number / n) = 50/5 = 10*



Now, we have five buckets, each of which has an interval of ten. So, we have five buckets with indices zero to four. And then, let us see how we can put the given numbers into each bucket.

**Input number / Interval size = Int (4/10) = 0**

We want to handle the first element in order to determine the index of the bucket, we can do the division. We divide the element by the interval size. And then, we just take the integer part of the result. So, in this case “4” divided by “10” gives us “0.4". When the integer part is just taken, we would just take zero.  So the input number “4” goes to index zero, it should go to the first bucket.

**Int (17/10) = 1**

Then we try to handle another value, which is “17”. So, 17 divided by ten is equal to “1.7”. We just take the integer part of it, we would get "1". So number “17” goes to index “1”.

**Int (37/10) = 3**

Now we handle “37”, whose division by ten is “3.7”. When we just take the integer part, we will simply get "3". So, 37 goes to bucket index “3”.

The rest of inputs place into appropriate buckets with the same calculations.

**Int (23/10) = 2**

**Int (29/10) = 2**

**Int (49/10) = 4**

**Int (44/10) = 4**

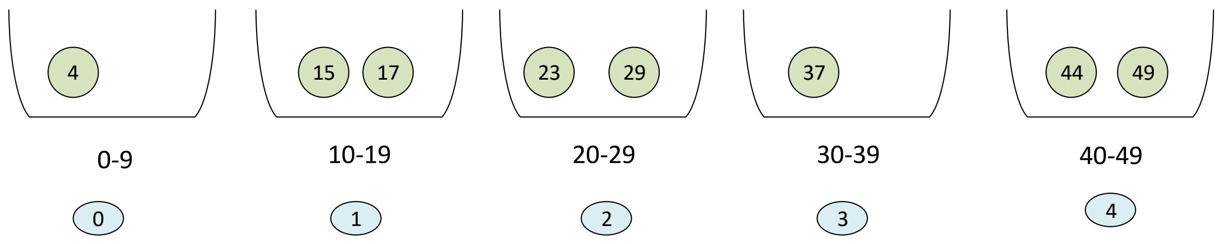
**Int (15/10) = 1**

When distribution of elements inside the right buckets is completed, sorting is applied for all elements available in each bucket. If a bucket contains only one element, this element will be already sorted.

For buckets zero and three, there is only one element in these buckets. So there is no need an operation for these buckets.

For index 1, we see that the bucket has two elements that are “17”and “15”. We can apply the insertion sort to order numbers inside bucket index “1”.

Insertion sort is applied to remaining buckets (1, 2, 4) to sort elements in each bucket as follows.



After all elements in the buckets are fully sorted, the sorted buckets are concatenated and listed as follows as the final sorting result.

**4, 15, 17, 23, 29, 37, 44, 49**

**Pseudocode:**

```
bucketSort(array, n) is
    buckets ← new array of n empty lists
    M ← the maximum key value in the array
    for i = 1 to length(array) do
        insert array[i] into buckets[floor(array[i] / ((M+1) / n))]
    for i = 1 to n do
        sort(buckets[i])
    return the concatenation of buckets[1], ...., buckets[k]
```

**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* | | | | | | | | | | | | |
| *dualPivot QuickSort* | | | | | | | | | | | | |
| *BucketSort* | | | | | | | | | | | | |

- 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 each sorting algorithm runs.

- 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.** You should 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 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 |
| bucketSort | 20 |
| Running Time Computation | 25 |
| Report | 15 |