

Merge Sort & Bubble Sort

Data Structures & Algorithms

data structure & algorithms Merge sort & Bubble sort ndkv mdsj,v ds ks

Project Report



2021

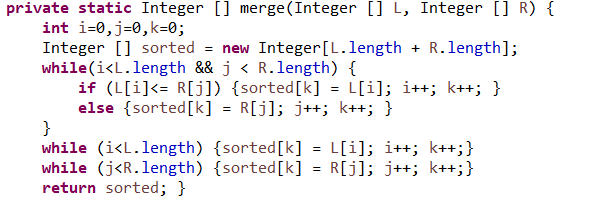
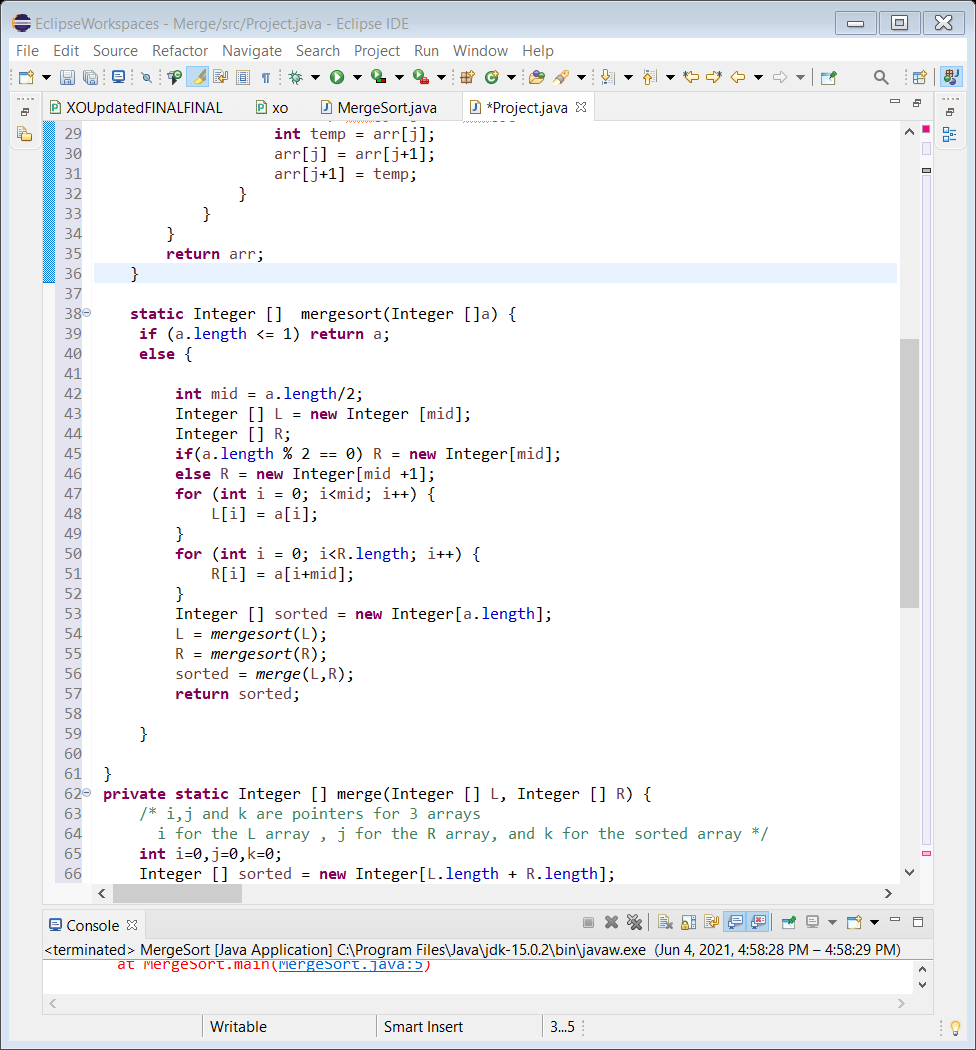
**Team Members:**

Andrew Botros Ayad | 20201322493 | Group 2 | Private depart. | even

Catherine Ashraf Emiel | 20201378779 | Group 2 | Private depart. | odd

Code Explanation

# First : merge sort

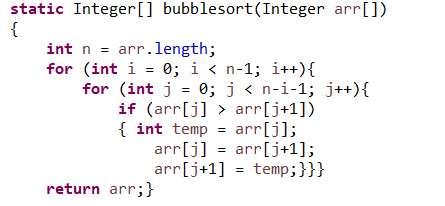


We implemented a method in java mergesort() that takes an array of type Integer to keep dividing the array into halves till there’s nothing but 1 element remaining in each divided array.  
 Division occurs by initializing two arrays of same type (L) and (R), but first, we made a condition to check whether the length of the required array is even or odd; in case of odd, we elongate the length of the other half with one place, knowing the elements won’t be evenly divided. The other way around in case of even.  
 We initialized a for loop to place the first half of the required array into (L) and the next half into (R).  
 Later, we let the method call itself (recursion) to (L) and (R) to keep on dividing themselves till they have 1 element left.  
 We initialized another array (sorted) to be assigned with the merge of both (L) and (R) by calling the method merge() and be the returned value.

# Second: bubble sort

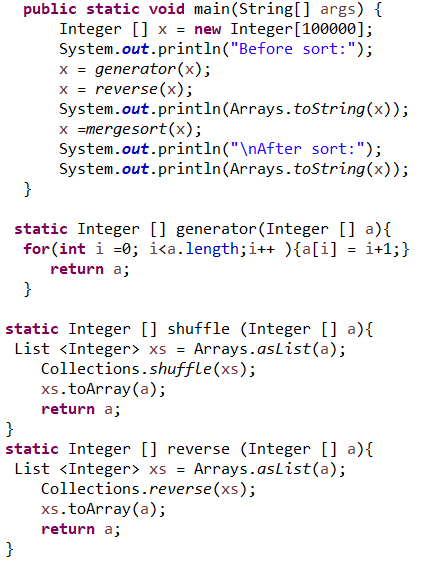
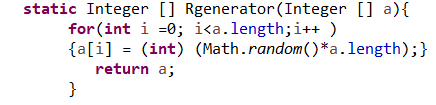
We initialize a while loop that stops when there’s no elements are being pointed at in either array (L) or (R).  
 The loop starts comparing the first element in both arrays and adds the smaller to the (sorted) array and the pointer is incremented of the used array and the (sorted) one.   
 When loop is exited, we start adding what’s left in either array and return the sorted array.

We implemented another method merge() that takes the two arrays (L) and (R) to be merged.  
 We initialize three pointer i, j and k. We use i to point in (L), j to point in (R) and k for the newly initialized array (sorted).



We made a method bubblesort() that takes an array and return it sorted.  
We make two nested for loops with a counter i and a pointer j.   
 j will point at the first element and compare it with one that follows it j+1, if found that it’s smaller than the first element, we swap them, else they remain as they are, and j will move to the next element till it finishes the array and then will start comparing again after i is incremented after each time j goes through all the elements of the array.

# Third: Generating the dataset



We created three methods to measure our average, best and worst cases of the sorting algorithms  
 The first method is the generator. It’s a for loop that iterates n times (n is the length of the array), and assign sequential numbers from 1 to n for the best case (sorted order).  
 The second method is the shuffle. We randomize the order of the element using the shuffle() method in the class Collection. But, in order to do that we convert the array to array-list then back to array when done. It’s for the average case.  
 The third method is the reverse. We reverse the order of the array using also the reverse() method in the class Collection after converting the array to array-list and back to array when done. It’s for the worst case (reversely sorted).

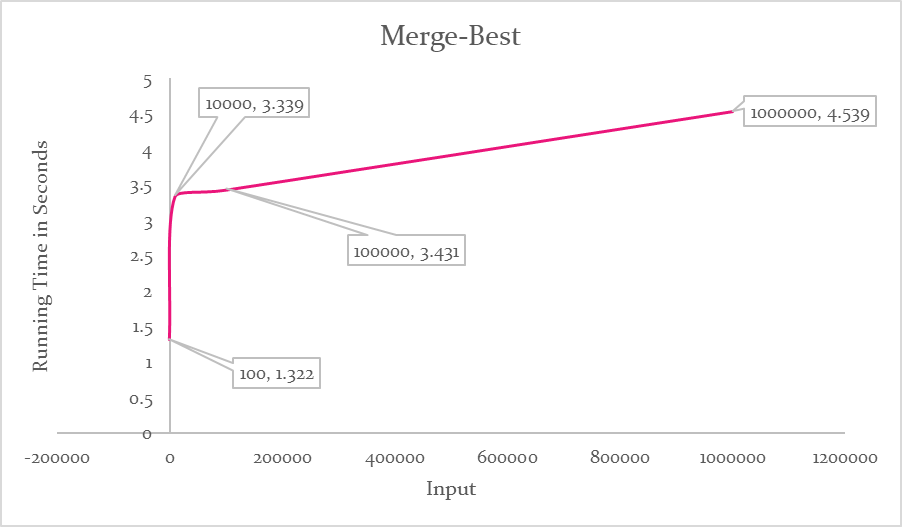
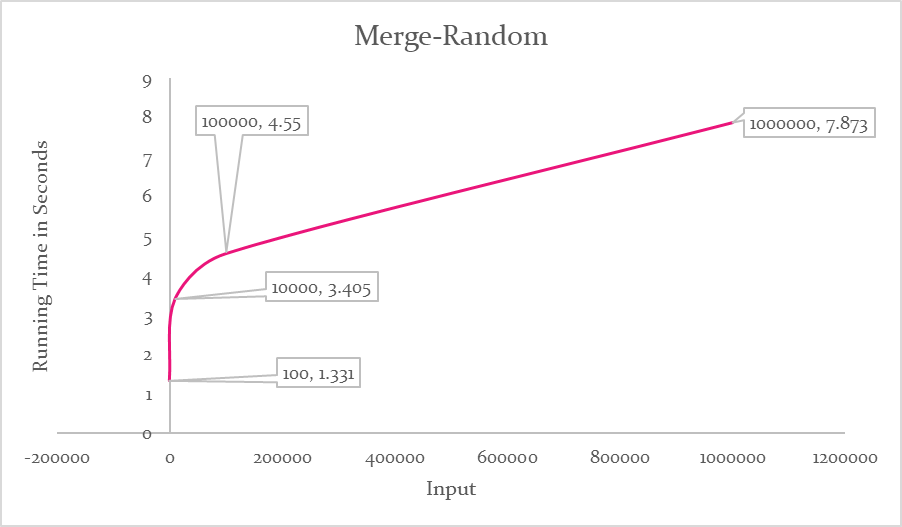
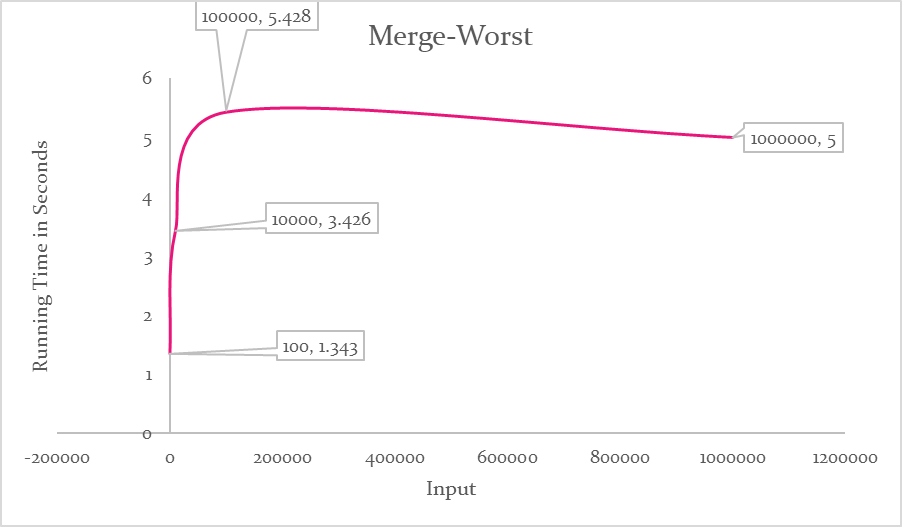
We created an optional fourth method to generate random dataset using random() method from Math class. It’s used to make the results of average case more accurate.

Comparisons

# First: Merge Sort (best/average/worst)

the following table shows comparisons of best-average-worst case of merge sort on different input sizes:

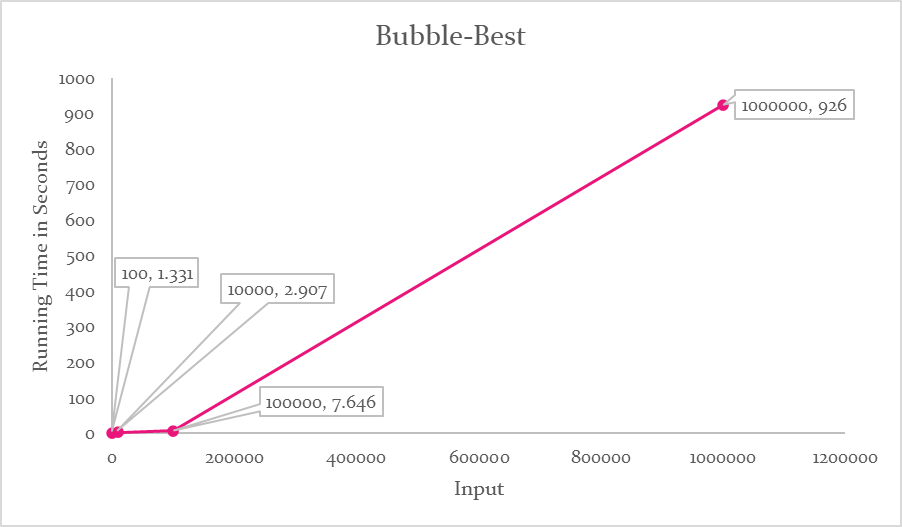
|  |  |  |  |
| --- | --- | --- | --- |
| Size of inputs | Best | Average | Worst |
| 100 |  |  |  |
| 10,000 |  |  |  |
| 100,000 |  |  |  |
| 1,000,000 | |  |  |  |

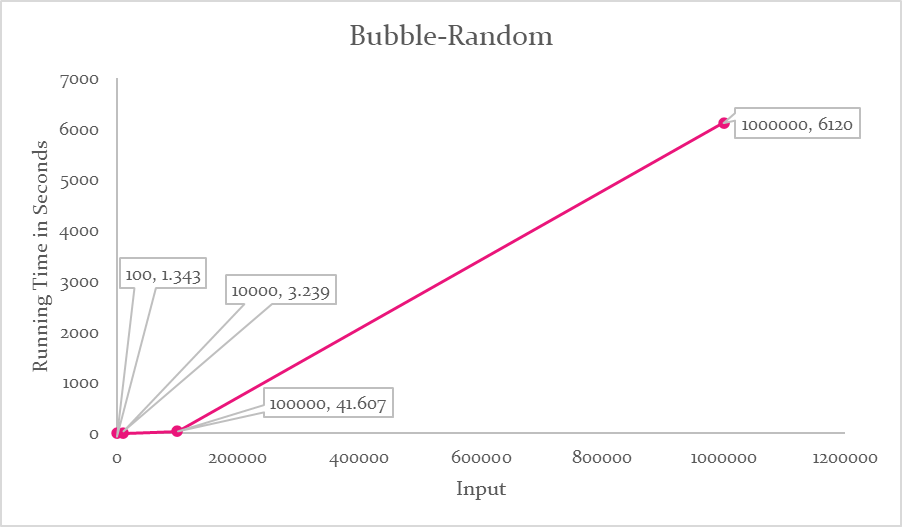
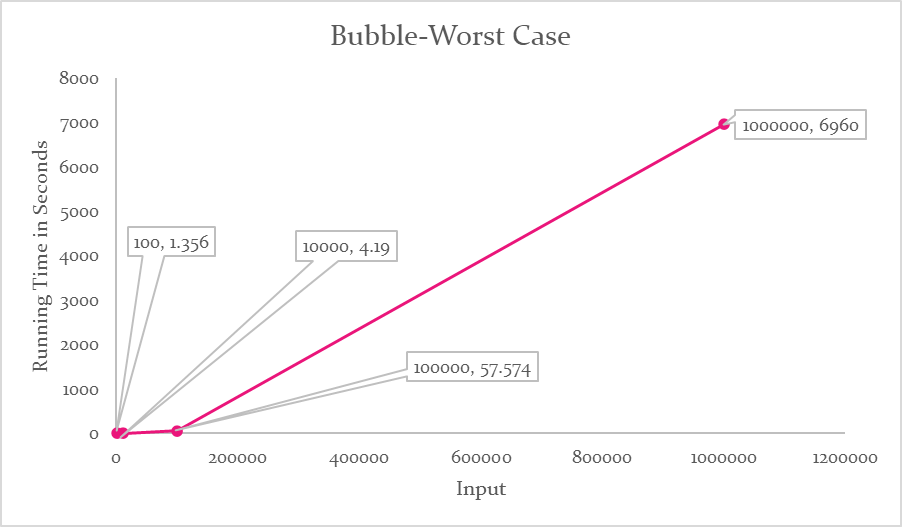
  
We can clearly see in the table that the running time increases on increasing input size and increases more when it’s reversely ordered, so the following graph shows the relation of time and input size in the three cases:

# Second: bubble sort

the following table shows comparisons of best-average-worst case of bubble sort on different input sizes:

|  |  |  |  |
| --- | --- | --- | --- |
| Size of inputs | Best | Average | Worst |
| 100 |  |  |  |
| 10,000 |  |  |  |
| 100,000 |  |  |  |
| 1,000,000 | |  |  |  |

  
We can clearly see in the table that the running time increases on increasing input size and increases more when it’s reversely ordered, so the following graph shows the relation of time and input size in the three cases:



# Merge sort vs. bubble sort

We notice that, bubble sort is quicker than merge sort in small inputs and best cases, otherwise, merge sort is a lot quicker.  
Their graphs are a lot equivalent to their order (Time Complexity)

|  |  |  |
| --- | --- | --- |
| POC | Merge Sort | Bubble Sort |
| Time Complexity (Best Case) | O(n log n) | O(n) Better! |
| Time Complexity (Average Case) | O(n log n)  Better! | O() |
| Time Complexity (Best Case) | O(n log n)  Better! | O( |
| Space Complexity (overall) | O(n) \*It uses memory to initialize its array elements when divided and merged again\* | O(1) \*Sorts itself in place by swapping\* |

# Device specifications used in measuring:

Processor: AMD Ryzen 7 2700 8-Core Processor, 3200 MHz, 8 Cores, 16 Logical Processors

Ram: 16 GB

Cache: L1 cache: 768 KB - L2 cache: 4 MB - L3 cache 16 MB

### Thank you