

CMPE 211: Data Structures and Algorithms  
Project

SELÇUK ARDA ÖZCAN

122200072

CMPE 211.01/0102

## Part 1: Sorting Algorithms

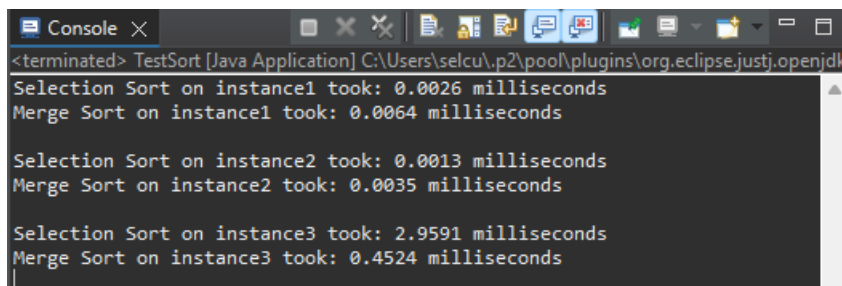
Algorithms that I will use for comparing between each other are Selection Sort and Merge Sort.

- **selectionSort()** : It's a simple algorithm . It searches an array to find the smallest value and when the algorithm finds it, replaces it with the first element. Its time complexity is  **$O(n^2)$**  .Outer loop's is  $O(n)$  and Inner loop as well. Swap operation's  $O(1)$ .
- **mergeSort()**: It's algorithm with Divide & Conquer tactic.Its time complexity is  **$O(n \log n)$** , which is more efficient than selectionSort in the larger data sets. But since it uses mergeSort() recursively it leads to an expensive space complexity. This is the 'Divide' part and its time complexity is constant time  $O(1)$ . Recursively calling mergeSort() causes time complexity of  $O(\log n)$ . merge() combines two sorted subarrays to one array and takes times as  $O(n)$ .

Here its test results:

instance1 and instance2 are small arrays that were given already.

To analyze and compare deeply I've used an array named **instance3** which its **length** is **1000** and contains **random numbers** .



```
<terminated> TestSort [Java Application] C:\Users\sclcu\p2\pool\plugins\org.eclipse.justj.openjdk
Selection Sort on instance1 took: 0.0026 milliseconds
Merge Sort on instance1 took: 0.0064 milliseconds

Selection Sort on instance2 took: 0.0013 milliseconds
Merge Sort on instance2 took: 0.0035 milliseconds

Selection Sort on instance3 took: 2.9591 milliseconds
Merge Sort on instance3 took: 0.4524 milliseconds
```

As it shows , selectionSort is handy at taking care of little arrays which are instance1 and instance2 . Instance2 is slightly sorted so it takes much less time compared to instance1. However , mergeSort is much more efficient as long as the array size increases.

Additionally I **increased** the size of **instance3** to **5000** ,**10.000** , **100.000** and **1.000.000** to demonstrate the performance differences between two algorithms.

instance3's size is 5000 . **mergeSort** is approximately 12 times better than **selectionSort**.

```
Selection Sort on instance3 took: 11.1051 milliseconds  
Merge Sort on instance3 took: 0.9044 milliseconds
```

the size is 10.000 . **mergeSort** is 20 times better.

```
Selection Sort on instance3 took: 26.8816 milliseconds  
Merge Sort on instance3 took: 1.4289 milliseconds
```

the size 100.000 . **mergeSort** is 170 times better.

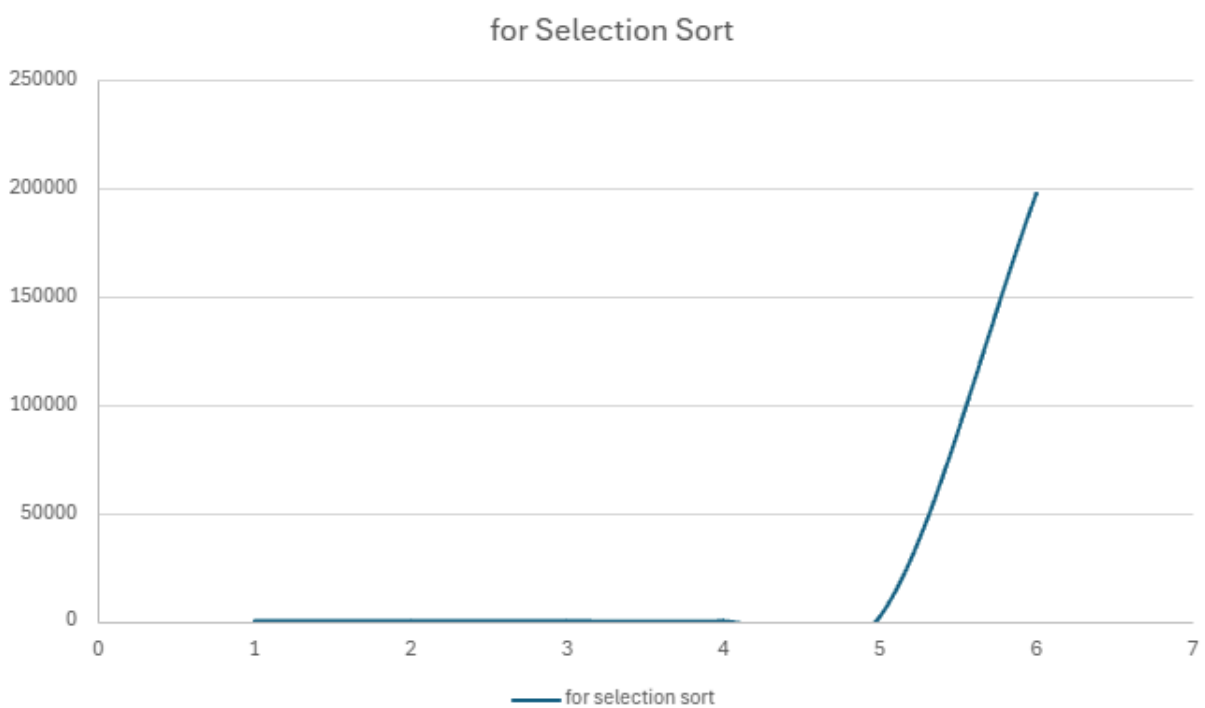
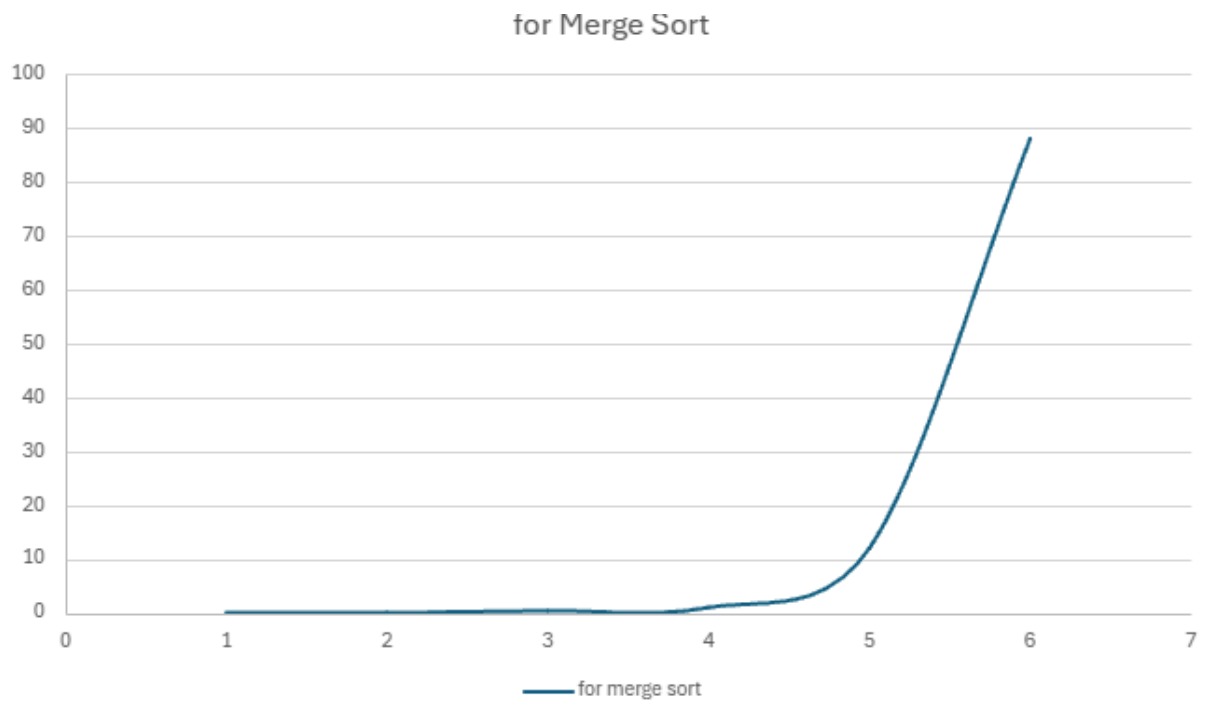
```
Selection Sort on instance3 took: 2087.44 milliseconds  
Merge Sort on instance3 took: 12.5609 milliseconds
```

Finally the size is **1.000.000** . mergeSort is 2.245 times better.

```
Selection Sort on instance3 took: 197552.1736 milliseconds  
Merge Sort on instance3 took: 88.3994 milliseconds
```

As a consequence, If you are working with small datasets like instance1 and instance2 you ought to use Selection Sort. But in real life mostly there will be large datasets to work on . In that case , Merge Sort will be useful and quicker.

And here it is the corresponding output time(Y-AXIS in milliseconds) graphics in terms of changes the array size for both algorithms.



## Part 2: Search Algorithms

I've chosen Binary Search Tree and Hash Table for comparison. And the sequences of operations are taken from the instruction sheet that is given.

- **BST:** Built by Nodes . Each node of the tree has a key, and for any given node, its left subtree contains only nodes with keys less than the node's key, and its right subtree contains only nodes with keys greater than the node's key. Its time complexity is  $O(\log n)$  .
  - **put(int key):** Inserts a key into the BST, ensuring that the tree remains structured based on the key's value. If the tree is balanced then time complexity is  $O(\log n)$ , if it's not then time complexity will be  $O(n)$ .
  - **get():** Similarly, the get operation has an average time complexity of  $O(\log n)$  in a balanced tree and  $O(n)$  in the worst-case scenario.
- **HashTable:** The critical difference is , it stores the data in the calculated hash code in order to refer it and call .
  - **Usually** its operations and methods have  $O(1)$  or  $O(n)$ .
  - **hashFunction():** It's the vital method of the HashTable . Creating a index for a given key. It holds the key with hashcode. It takes modulo with size of the created LinkedList.
  - The rest is the same with BST.

Test Results for sequence1.

```
<terminated> Main (1) [Java Application] C:\Users\selscu\.p2\pool\plugin
Key 10 exists: true
Key 5 exists: true
Key 7 exists: false
Key: 3
Key: 5
Key: 10
BST Search on sequence1 took: 8.8381 milliseconds

Key 10 exists: true
Key 5 exists: true
Key 7 exists: false
Index 0: {10}
Index 1:
Index 2:
Index 3: {3}
Index 4:
Index 5: {5}
Index 6:
Index 7:
Index 8:
Index 9:
HashTable on sequence1 took: 2.6401 milliseconds
```

**HashTable:** The keys in sequence1 result in fewer collisions, meaning the hash function distributes the keys well across the table. This efficient distribution ensures that each bucket contains a minimal number of entries, leading to  $O(1)$  average time complexity for 'put' and 'get'.

**BST:** While the BST handles the 'put' and 'get' efficiently, it still has an average time complexity of  $O(\log n)$  for each operation. So, although the BST performs well, the constant time complexity of the HashTable gives it a performance edge in this sequence.

Test Results for sequence2:

```
Key 10 exists: true
Key 5 exists: true
Key 5 exists: true
Key: 1
Key: 2
Key: 3
Key: 4
Key: 5
Key: 6
Key: 10
HashTable on sequence2 took: 0.4533 milliseconds|

Key 10 exists: true
Key 5 exists: true
Key 5 exists: true
Index 0: {10}
Index 1: {1}
Index 2: {2}
Index 3: {3}
Index 4: {4}
Index 5: {5}
Index 6: {6}
Index 7:
Index 8:
Index 9:
HashTable on sequence2 took: 0.8181 milliseconds
```

**HashTable:** The keys in sequence2 may have caused multiple keys to be placed in the same bucket. When collisions occur, the linked list at each bucket becomes longer, resulting in  $O(n)$  worst-case time complexity for insertions and retrievals, as each operation may need to traverse the entire list.

**BST:** The BST can handle these operations efficiently, particularly when the tree remains relatively balanced. The keys in sequence2 are such that the tree stays balanced, ensuring  $O(\log n)$  time complexity for insertions and searches. This balanced structure allows the BST to outperform the Hash Table in this sequence.

Consequently, The HashTable is highly effective when keys are well-distributed and collisions are minimal, while the BST excels when the tree remains balanced, providing efficient operations.