CS EE World
https://cseeworld.wixsite.com/home
May 2021
28/34
A
Submitter Info:
Anonymous

Extended Essay in Computer Science

Examination Session – May 2021

Title – Investigating the Algorithmic Efficiency of Binary Search Tree and Binary Heap Based Sorting Algorithms

Research Question - How does the sorting efficiency of the Tree Sort compare to that of the Heap Sort in terms of time complexity for increasing sizes of randomized integer datasets?

Word Count - 3997

Table of Contents

1. **Introduction** **2**
2. **Theory** **2**
   2.1. Sorting Algorithms 2
   2.2. Tree Sort & Binary Search Trees 5
   2.3. Heap Sort & Binary Heaps 10
3. **Hypothesis** **14**
4. **Methodology** **15**
   4.1. Independent Variable 15
   4.2. Dependent Variables 16
   4.3. Controlled Variables 16
   4.4. Procedure 17
5. **Data Processing and Graphing** **17**
   5.1. Raw Data Collection 17
   5.2. Graphing and Curve Fitting 20
6. **Analysis** **23**
7. **Results Discussion & Evaluation** **26**
8. **Conclusion** **28**
9. **Further Scope** **29**
10. **Bibliography** **30**
11. **Appendices** **33**
   11.1. Appendix A 33
   11.2. Appendix B 34
   11.3. Appendix C 35
   11.4. Appendix D 36

1. Introduction

The primary focus of this essay is to investigate the computational complexities or the sorting efficiencies of binary-tree based sorting algorithms, a class of algorithms based on binary abstract data structures. Today, sorting is one of the most popular and useful computational processes, and hence, performing a comparative study between a specific set of these algorithms is crucial. Thus, this essay will look specifically into two sorting algorithms: Tree Sort, which is based on Binary Search Trees (BST) and the Heap Sort, which is based on Binary Heaps. These algorithms will be compared in terms of their time complexity: the time taken for algorithm execution based on the input dataset size. Hence, this gives rise to the research question: "**How does the sorting efficiency of the Tree Sort compare to that of the Heap Sort in terms of time complexity for increasing sizes of randomized integer datasets?**"

2. Theory

2.1 Sorting Algorithms

Sorting algorithms are one of the simplest but most unique classes of algorithms. A sorting algorithm performs a series of operations on a set of integers and outputs them, in sorted or ascending order. For example –

$$[\mathbf{5}, \mathbf{3}, \mathbf{2}, \mathbf{4}, \mathbf{1}] \qquad \rightarrow \qquad [\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}]$$

As shown above, the concept of sorting is straightforward. However, the approaches taken to sorting can be very diverse. Hence, sorting algorithms can further be classified into **Comparison Sorts** and **Integer Sorts**.[1] Comparison sorts are based on comparing two elements to determine if one should be before or after the other in the sorted list. A few

[1] "Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) Based Sorting Algorithms?," *Javarevisited*, accessed July 12, 2020, https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#axzz6nplsEjux.

examples are the Heap Sort and Merge Sort. On the contrary, Integer Sorts determine the number of elements which are lesser in value than a selected element, based on its integer key, to identify the correct position of this element in the list without requiring extensive comparisons.[2] A few examples are the Radix Sort and Bucket Sort.[3] An example of a comparison sorting algorithm is shown below –

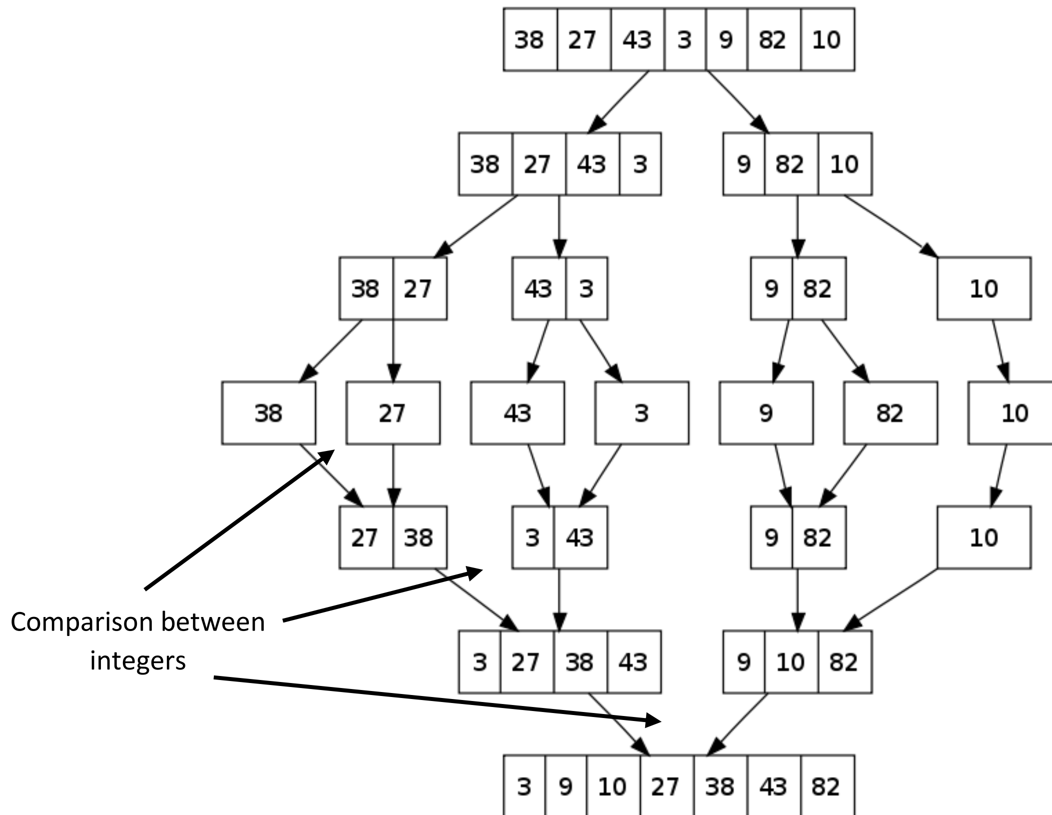


Figure 1 – Visualization of Merge Sort[4]

Figure 1 is a depiction of the merge sort which portrays single comparisons between pairs of integers as a means to sort an array. This brings into picture sorting algorithm design paradigms such as divide & conquer and recursion,[5] and also introduces time complexity as a means for algorithmic analysis.

---

[2] ibid.
[3] ibid.
[4] Nikhil Joshi, "Implementation and Analysis of Merge Sort," *Dotnetlovers* (Dotnetlovers, October 29, 2018), accessed July 12, 2020, https://www.dotnetlovers.com/article/128/implementation-and-analysis-of-merge-sort.
[5] TimTim 1, "Divide and Conquer and Recursion," *Stack Overflow,* January 1, 2009, accessed July 12, 2020, https://www.stackoverflow.com/questions/2249767/divide-and-conquer-and-recursion.

The primary method of measuring the efficiency of a sorting algorithms it to measure its time complexity. However, **asymptotic time complexity** – algorithm execution time as dataset size approaches infinity – can be used for a better understanding of algorithm efficiency. It can be divided into three parameters - $\boldsymbol{O(n)}$ the upper bound or worst-case complexity, $\boldsymbol{\Omega(n)}$ the lower bound or best-case complexity, and $\boldsymbol{\theta(n)}$ the average-case complexity.[6] These functions tell us the limits of, and the average running time of any algorithm as depicted below –

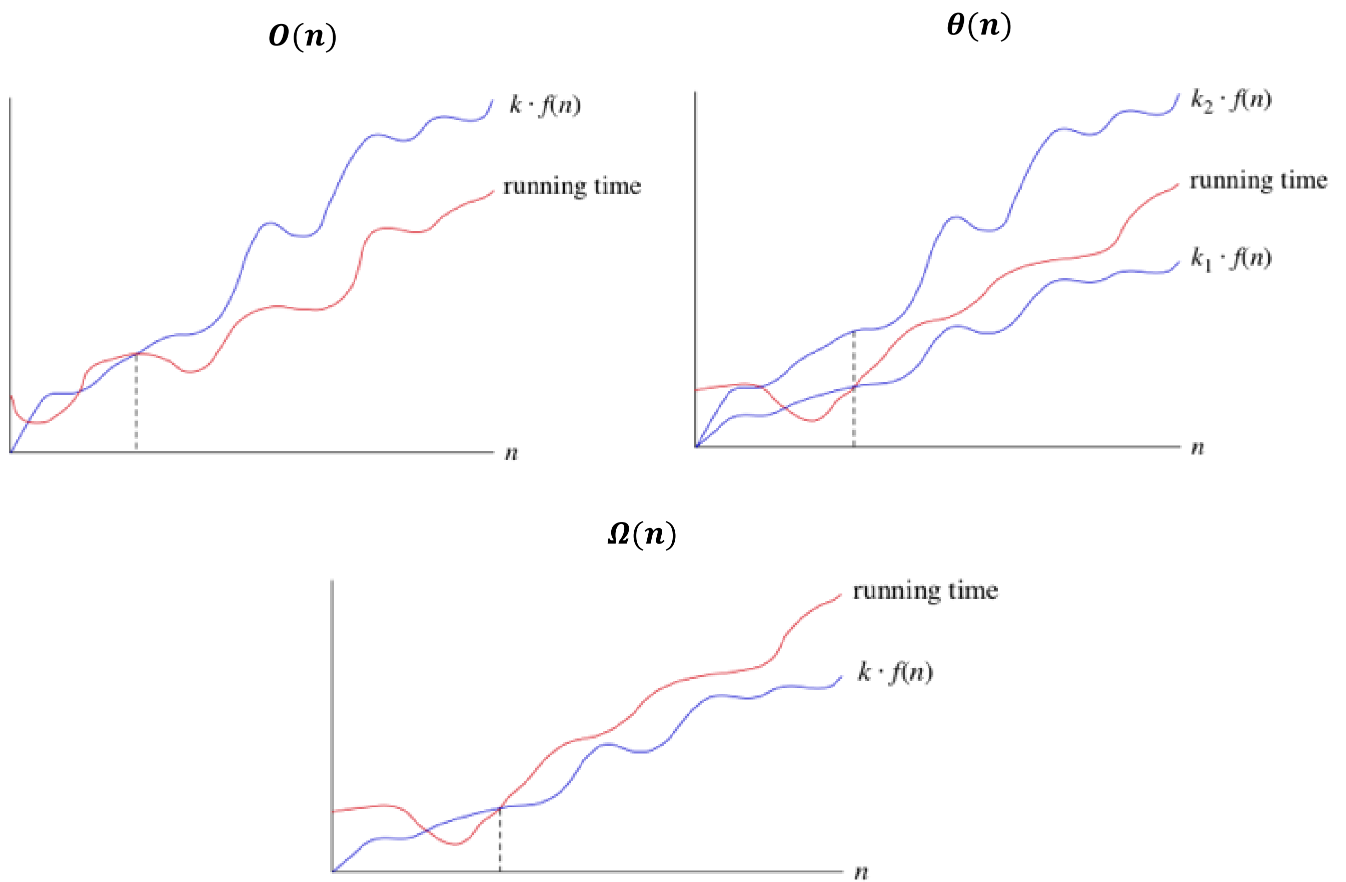


Figure 2 – Asymptotic Time Complexity Parameters[7]

Therefore, before experimentally determining the running-time of Tree Sort and Heap Sort, which are both comparison sorts, we can mathematically derive the best-case complexity to preordain a trend in running-time.

---

6 "Asymptotic Analysis: Big-O Notation and More," *Programiz,* accessed July 12, 2020, https://www.programiz.com/dsa/asymptotic-notations.

7 "Big-O Notation (Article) | Algorithms," *Khan Academy*, Khan Academy, accessed July 12, 2020, https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation.

Taking a decision tree, where the leaves are all possible permutations ($\boldsymbol{n}!$) of a set of integers and the comparison sort is modelled by the root-to-leaf path where each step is a comparison, then the number of comparisons is limited by the height of the tree.[8] As $\boldsymbol{2^h}$ is the number of leaves of the decision tree as a function of the height –

$$\boldsymbol{2^h \geq n!} \qquad \Rightarrow \qquad \boldsymbol{h \geq \log(n!)}$$

Equation 1 – Relationship Between Number of Comparisons and Height of a Binary Tree[9]

Using Stirling's Approximation –

$$\Rightarrow \boldsymbol{n! > \left(\frac{n}{e}\right)^n}$$

$$\boldsymbol{\therefore h \geq log\left(\frac{n}{e}\right)^n = n \cdot log\left(\frac{n}{e}\right)}$$

$$\boldsymbol{= n \cdot log(n) - n \cdot log(e)}$$

$$\boldsymbol{= \Omega(n \cdot log(n))}$$

Equation 2 – Deriving the Lower-Bound Time Complexity of Comparison Sorts[10]

Consequently, we know that the running time of both Tree Sort and Heap Sort will not be better that $\boldsymbol{n \cdot log(n)}$.

2.2 Tree Sort & Binary Search Trees

A binary tree is an abstract data structure composed of nodes. Each node has some data (integers in this case), and has pointers to a left and right child node. The topmost node is called the root, and a node with no child nodes is called a leaf. A Binary Search Tree (**BST**) is a special binary tree with certain properties. The value of any left child must always be **less than** the value of its parent node, and the value of any right child must always be **greater than** the value of its parent node.[11] An example is shown below –

[8] Karleigh Moore, "Sorting Algorithms," *Brilliant Math & Science Wiki*, accessed July 24, 2020, https://www.brilliant.org/wiki/sorting-algorithms/.
[9] ibid.
[10] ibid.
[11] "Data Structure - Binary Search Tree," *Tutorialspoint*, accessed July 24, 2020, https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm.

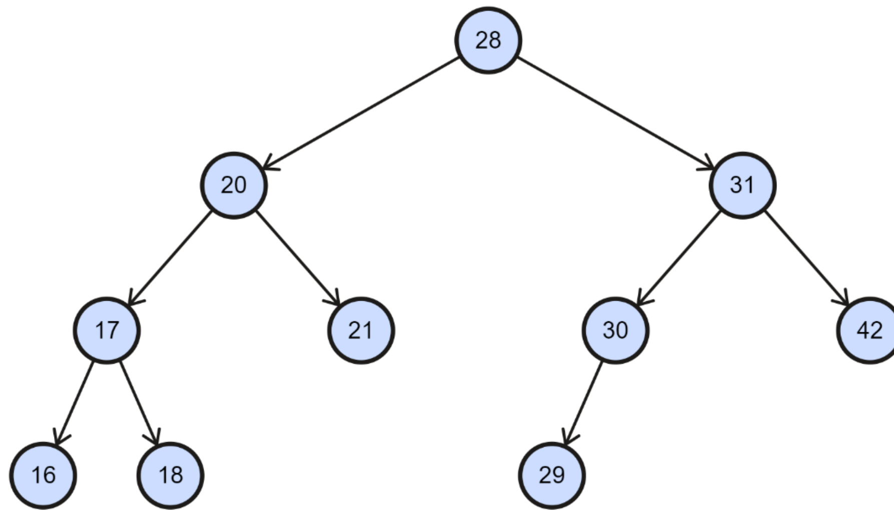


Figure 3 – Binary Search Tree Example

For sorting an integer dataset with tree sort, the integers must first be **inserted** into the BST through the following procedure –

1. If the root node is **null** i.e., the BST is empty, then the root node is set to this value.
2. If the root node is present, the value being inserted is **less than** the root, and the left child node is **null**, then the left child will be set to this value.
3. If the root node is present, the value being inserted is **greater than** the root, and the right child node is **null**, then the right child will be set to this value.
4. If the child nodes already exist, this logic will occur recursively until a **null** child is found.[12] This value will then be assigned to a new leaf node. A sample insertion is shown below.

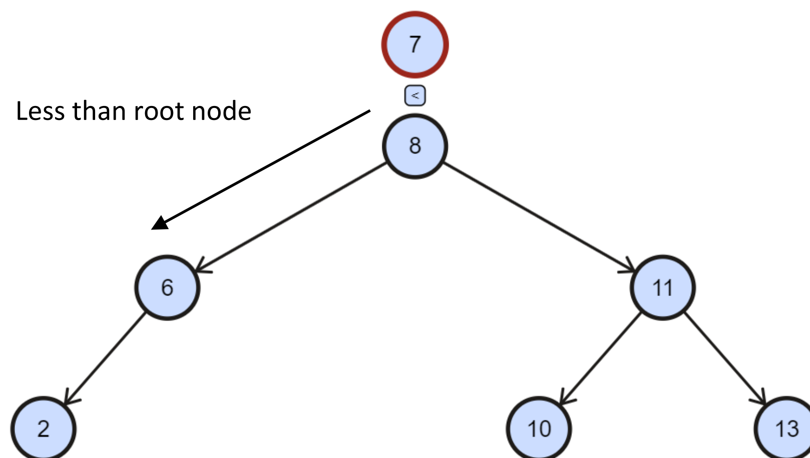


Figure 4 – Binary Search Tree Insertion (Comparison with Root Node)

[12] Robert Sedgewick, and Kevin Wayne, "Binary Search Trees," *Princeton University*, The Trustees of Princeton University, accessed July 24, 2020, https://algs4.cs.princeton.edu/32bst/,

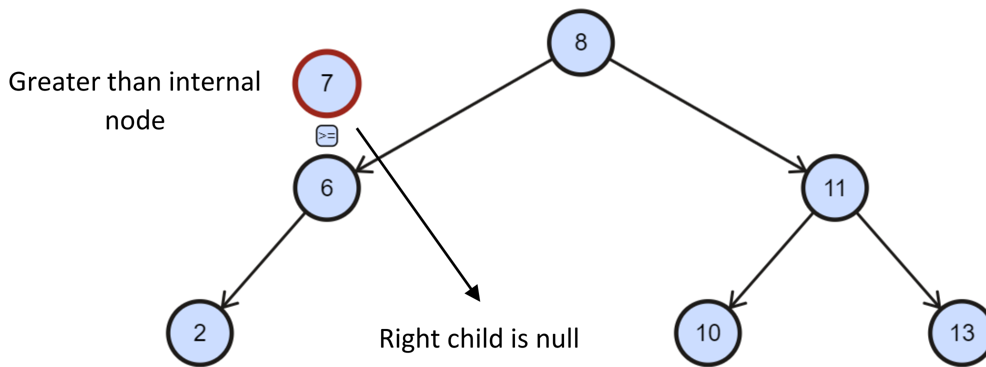


Figure 5 – Binary Search Tree Insertion (Comparison with Internal Node)

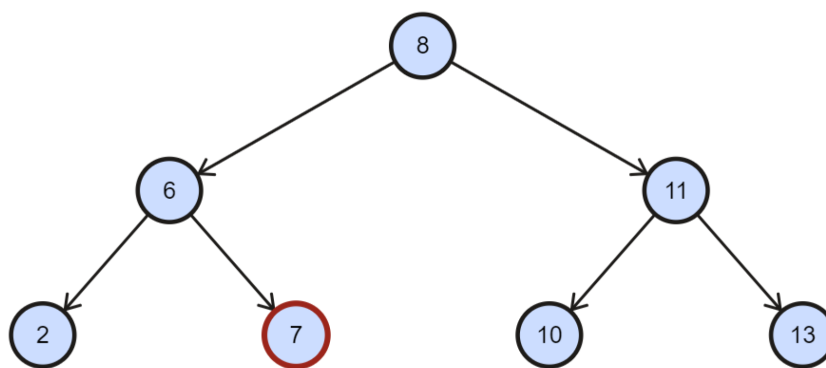


Figure 6 – Binary Search Tree Final Insertion

It is to be noted that BSTs are not naturally self-balancing. There is no restriction on tree height.

After insertion, the second half of tree sort entails performing a traversal on the BST. A **Depth-First Traversal** algorithm, which traverses a BST branch-wise rather than level-wise (Breadth-First Traversal) would be more appropriate in this case since we need to access the leaves of the BST (lowest and highest values) in lower time. Furthermore, an Inorder traversal, which first traverses the left sub-tree, visits the root, and then traverse the right sub-tree, would allow the BST values to be returned in sorted order.[13] This is shown below –

[13] Javinpaul, "How to Implement Inorder Traversal in a Binary Search Tree?," *DEV Community* (DEV Community, August 14, 2019), accessed July 24, 2020, https://www.dev.to/javinpaul/how-to-implement-inorder-traversal-in-a-binary-search-tree-1787.

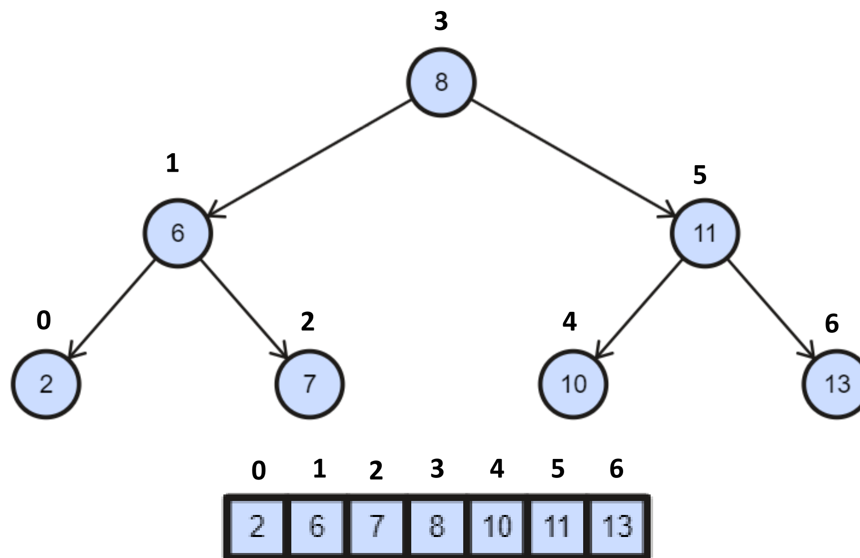


Figure 7 – Sorted Binary Search Tree

Hence the algorithm is divided into two methods, the *insert()* method and *dfs()* method. Each node is represented by an object with three instance variables. One being the integer value of the node, and the other two being pointers to the left and right child nodes. The *insert()* method has two parameters: the root node object, and the integer value to be inserted into the Binary Search Tree.

```
1. public Node insert(Node node, int key) {
2.         if (node == null) {
3.             node = new Node(key); // Creating a new tree
4.             return node;
5.         }
6.         if (key < node.key)
7.             node.left = insert(node.left, key);
8.
9.         else if (key > node.key)
10.             node.right = insert(node.right, key);
11.
12.         return node;
13.     }
```

Figure 8 – Tree Sort Insert Function (Appendix A)[14]

If a root node is not present, a new BST is created. However, if the value to be inserted is less than the value of the root node, then the *insert()* method is recursively called on the left sub-

[14] Vibin M, "Tree Sort," *GeeksforGeeks*, April 20, 2020, accessed August 1, 2020, https://www.geeksforgeeks.org/tree-sort/.

tree until a base case is reached where the left or right child nodes are empty, after which a new node is inserted as a leaf. If the value to be inserted is greater than the value of the root node, then the *insert()* method is recursively called on the right sub-tree instead and the process is repeated.

```
1. public void dfs(Node node) {
2.         if (node != null) {
3.             dfs(node.left); // Recursing down the left sub-tree
4.             System.out.print(node.key + ", ");
5.             dfs(node.right); // Recursing down the right sub-tree
6.         }
7.     }
```

Figure 9 – Inorder Traversal Function (Appendix A)[15]

In the *dfs()* method, the method recurses down the left sub-tree until the base case, a left child leaf is reached, in which case its value is printed, followed by the value of the parent node, followed by the value of the right child leaf. After the left sub-tree is recursively traversed, the root node is printed, and finally, the method recurses down the right-sub-tree. This would output the BST in sorted order.

The average time complexity of the Tree Sort $\boldsymbol{\theta(nlog(n))}$ can be broken down. The time complexity of both *insert()* and *dfs()* is $\boldsymbol{O(nlog(n))}$. For these functions, $\boldsymbol{n}$ integers must be inputted into and outputted from the trees respectively and the time taken to recurse down the tree to insert and traverse each node are both $\boldsymbol{O(log(n))}$ since the number of levels in a BST increases logarithmically with respect to the number of nodes. Therefore by adding the complexities, the constant can be ignored and the overall complexity comes to $\boldsymbol{O(nlog(n))}$.

[15] ibid.

2.3 Heap Sort and Binary Heaps

A Binary Heap, specifically a Max Heap, is a binary tree with properties different to those of a BST. The value of each node must be greater than or equal to the values of the child nodes. Hence unlike a BST, a values in a max heap increase from bottom to top instead of from left to right. This property of a Max Heap also allows it to be naturally self-balancing.[16] An example is shown below –

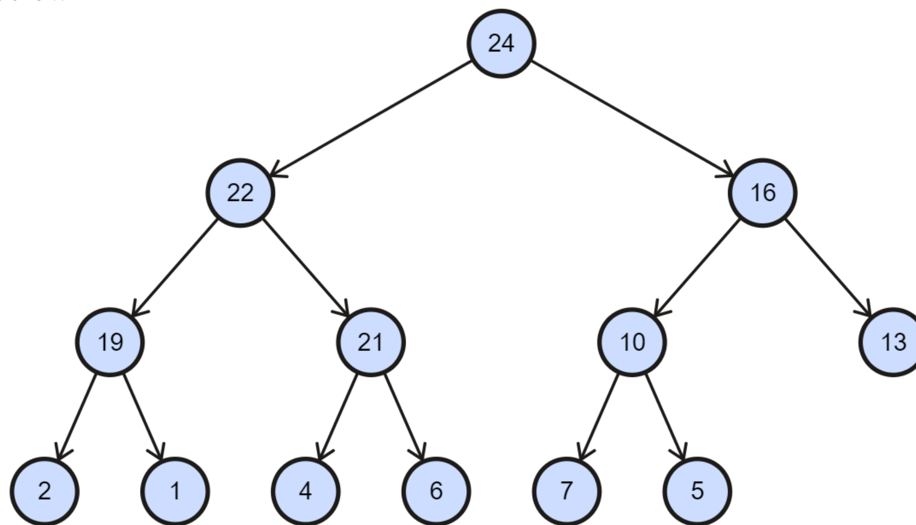


Figure 10 – Max Binary Heap Example

This naturally self-balancing property allows Max Heaps to be represented through arrays, where if a node is an index $\boldsymbol{i}$, then the left child is at index $\mathbf{2}\boldsymbol{i} + \mathbf{1}$, and the right child is at index $\mathbf{2}\boldsymbol{i} + \mathbf{2}$.[17] For Heap Sort, a Max Heap must first be built by rearranging the array using a reverse breadth first traversal –

1. Beginning from the last node in the $\boldsymbol{n} - \mathbf{1}$ level of the tree ($\boldsymbol{n}$ is the number of levels), if the node is **greater than** both child nodes, the sub-tree is already heapified.
2. However, if the node is **less than** either or both child nodes, it is swapped with the **greater** child node. Similarly all sub-trees on the $\boldsymbol{n} - \mathbf{1}$ level must be heapified.

[16] Navjot Singh, "Why Is Binary Heap Never Unbalanced?," *Computer Science Stack Exchange*, May 2, 2019, accessed August 18, 2020, https://cs.stackexchange.com/questions/108852/why-is-binary-heap-never-unbalanced.
[17] "Binary Heaps," *Heaps*, Andrew CMU, accessed August 18, 2020, http://www.andrew.cmu.edu/course/15-121/lectures/Binary%20Heaps/heaps.html.

3. Move to level $n-2$ and repeat the process from right to left. If nodes are swapped, the affected sub-trees must be recursively re-heapified.

4. Once the traversal reaches the root node, the binary tree has been heapified into a Max Heap as depicted below –

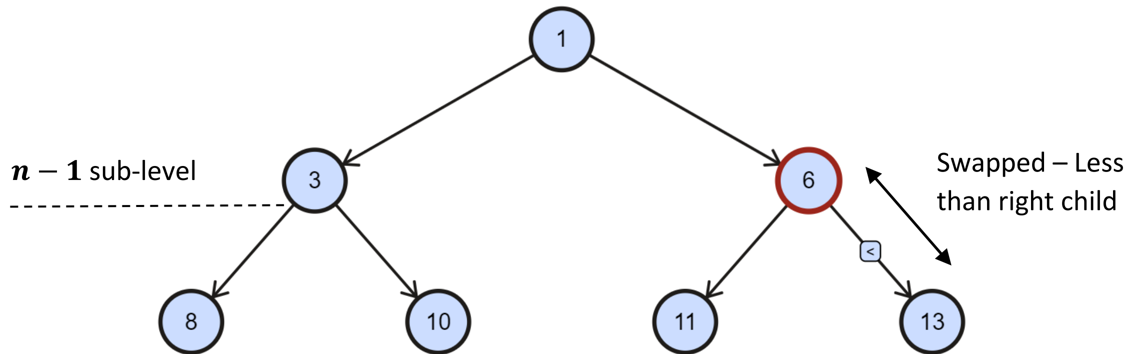


Figure 11 – Max Heap Heapification (Comparison and Swap Within Right Sub-Tree)

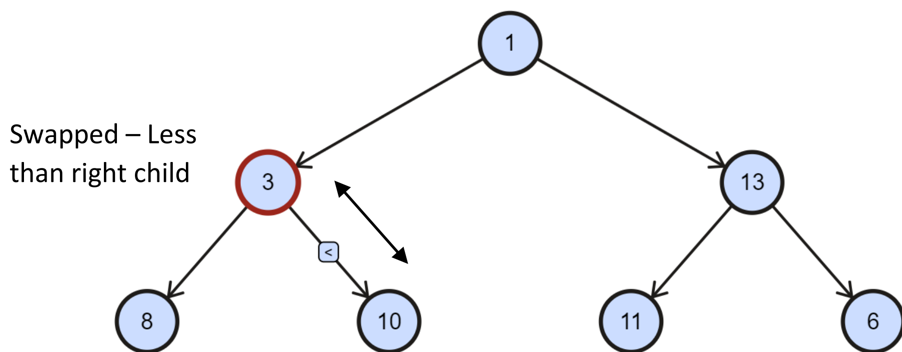


Figure 12 – Max Heap Heapification (Comparison and Swap Within Left Sub-Tree)

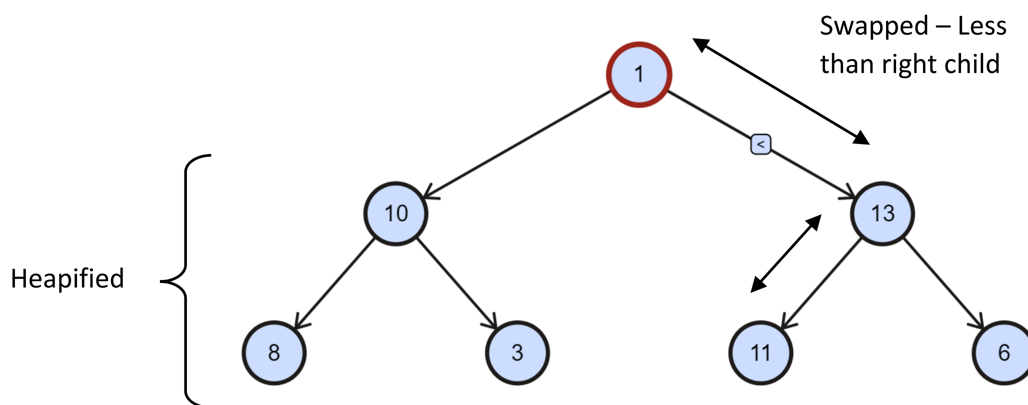


Figure 13 – Max Heap Heapification (Comparison with Root Node and Re-Heapification)

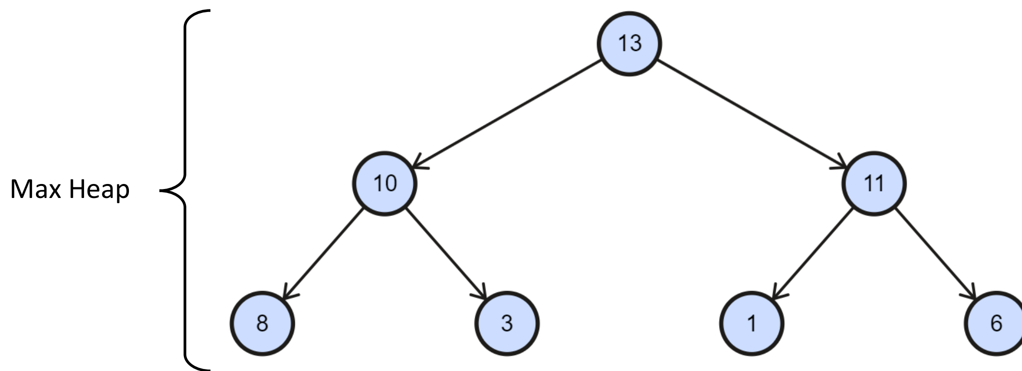


Figure 14 – Complete Max Heap after Heapification

After the tree has been heapified, the root node is swapped with the last leaf node and added to the end of the array. The reduced heap is then re-heapified. This process of swapping the root with the last leaf and re-heapifying is repeated until the array is sorted –

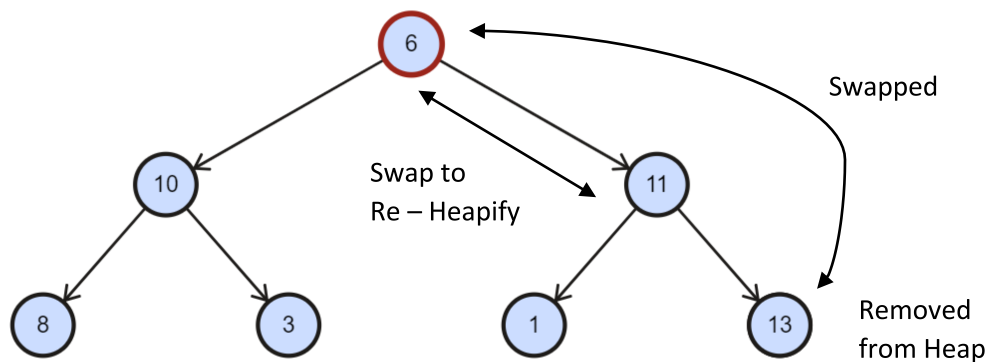


Figure 15 – Swapping of Root with Last Leaf Node and Re-Heapification

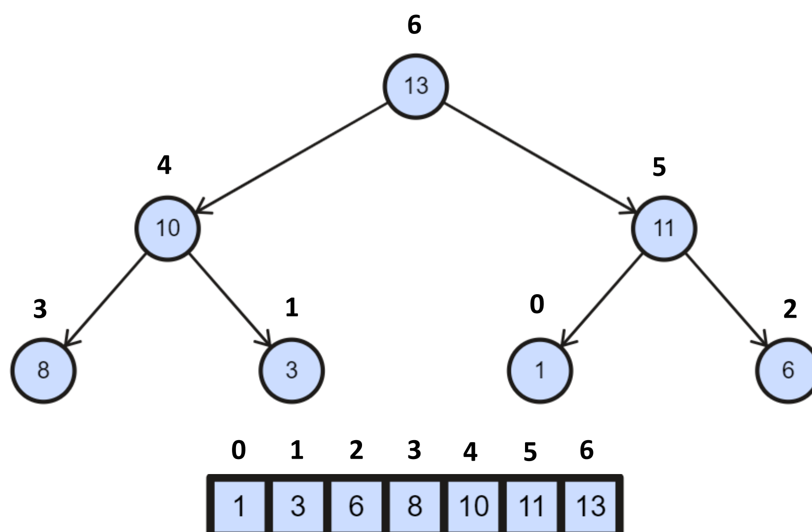


Figure 16 – Sorted Max Binary Heap

Hence, the primary method in Heap Sort is *heapify()* which restores the Max Heap structure of the Binary Heap. This method is iteratively called by two loops in the *sort()* method, one that builds the Max Heap, and one that repeatedly extracts the root node and sorts the array. *Heapify()* has three parameters: the array to be sorted, the array length, and the array index of the root of a sub-tree to be heapified.

```
1. void heapify(int arr[], int n, int i) {
2.         int largest = i; // Initializing largest as root
3.         int l = 2*i + 1; // Left Child = 2*i + 1
4.         int r = 2*i + 2; // Right Child = 2*i + 2
5.
6.          if (l < n && arr[l] > arr[largest])
7.              largest = l;
8.
9.          if (r < n && arr[r] > arr[largest])
10.             largest = r;
11.
12.         if (largest != i) {
13.             int swap = arr[i];
14.             arr[i] = arr[largest];
15.             arr[largest] = swap;
16.
17.              heapify(arr, n, largest); // Recursively heapifying
18.           }
19.     }
```

Figure 17 – Heap Sort Heapify Function (Appendix B)[18]

The indices of an internal node and its children and represented by the variables *largest, l,* and *r*. If either child node is larger than the other child and the parent node, then *largest* is reassigned to that node, the indices of the child and parent node are swapped, and the affected sub-tree[19] rooted at the *largest* node is recursively heapified until a base case is reached where both child nodes are lesser than the parent node.

```
1. public void sort(int arr[]) {
2.         int n = arr.length;
3.
4.         for (int i = n / 2 - 1; i >= 0; i--) // Building max heap
5.             heapify(arr, n, i);
```

[18] Shivi Aggarwal, "HeapSort," *GeeksforGeeks*, Last Modified November 16, 2020, accessed August 18, 2020, https://www.geeksforgeeks.org/heap-sort/.
[19] ibid.

```
6.
7.          for (int i=n-1; i>0; i--) {
8.              int temp = arr[0]; // Moving current root to end
9.              arr[0] = arr[i];
10.             arr[i] = temp;
11.
12.         heapify(arr, i, 0); // Heapifying reduced heap
13.         }
14.     }
```

Figure 18 – Sort Function: Building and Sorting Max Heap (Appendix B)[20]

The first loop performs a breadth first traversal by calling *heapify()* on all sub-trees rooted at nodes level-wise, from node $\boldsymbol{i} = \frac{\boldsymbol{n}}{\boldsymbol{2}} - \boldsymbol{1}$ (where $\boldsymbol{n}$ is the array length) to node $\boldsymbol{i} = \boldsymbol{0}$ in order to construct a Max Heap, i.e., the loop disregards the leaves of the heap. The second loop performs the actual Heap Sort by iteratively executing a simple swapping algorithm to swap the indices of the first and last nodes, and then calling *heapify()* on the root of the reduced heap whose number of nodes decrease from $\boldsymbol{n} - \boldsymbol{1}$ to $\boldsymbol{0}$ throughout loop execution as the array is sorted.

The average time complexity of Heap Sort is $\boldsymbol{\theta}(\boldsymbol{nlog}(\boldsymbol{n}))$. The *heapify()* function happens in $\boldsymbol{O}(\boldsymbol{log}(\boldsymbol{n}))$ time since the number of levels to be recursed down in a heap or a sub-tree increases logarithmically with respect to the number of nodes. Hence, when building and sorting the Max Heap, $\boldsymbol{n}$ integers must be inserted and outputted respectively and re-heapification takes place after each. Therefore, by adding both linearithmic complexities, the constant can be ignored and the overall complexity becomes $\boldsymbol{O}(\boldsymbol{nlog}(\boldsymbol{n}))$.

3. Hypothesis

It is evident that both sorting algorithms have an average time complexity of $\boldsymbol{\theta}(\boldsymbol{nlog}(\boldsymbol{n}))$. However, multiple stark contrasts are present in the properties of both data structures and the respective algorithm designs, such as the contrast between a Binary Heap's self-balancing[21] to

[20] ibid.

[21] "CS 312 Lecture 25: Priority Queues and Binary Heaps," *Lecture 25: Priority Queues and Binary Heaps*, accessed August 18, 2020, https://www.cs.cornell.edu/courses/cs312/2007sp/lectures/lec25.html.

a normal BST's unbalanced nature or the contrast between the Heap Sort's partially iterative logic to a Tree sort's purely recursive logic. As a result, while the trends in algorithm execution time might be similar between both algorithms, the actual execution times for sorting very large integer datasets could be substantially different, perhaps lower for Heap Sort due to its asymptotically balanced nature and space-efficient array implementation.[22]

Therefore, the aim of this experiment is to test the effect of increasing randomized dataset size $\boldsymbol{n}$ on the time taken $\boldsymbol{t}$ by both Tree Sort and Heap Sort to sort the dataset in increasing order. The relationships between the two variables will be comparatively analyzed between both algorithms. Moreover, for deeper analysis, the range $\boldsymbol{R}$ of the datasets will be changed as an auxiliary independent variable to determine any additional effect on sorting performance.

I hypothesize that the Heap Sort will sort the dataset in lower time than the Tree Sort. There will be a **linearthimic relationship** between $\boldsymbol{n}$ and $\boldsymbol{t}$.

4. Methodology

4.1 Independent Variable

The independent variable is the size of the integer datasets $\boldsymbol{n}$. The sizes will increase from $\mathbf{10000}$ integers to $\mathbf{100000}$ integers in increments of $\mathbf{10000}$ in order to acquire a significant number of data points to plot more accurate and precise graphs. Furthermore, for each size $\boldsymbol{n}$ three datasets will be generated with ranges of $\mathbf{2 \times 10^5}$, $\mathbf{4 \times 10^5}$, and $\mathbf{6 \times 10^5}$ respectively. All integer datasets will have completely randomized distribution (discrete uniform). Moreover, the datasets will also contain both positive and negative integers. An online random number generator will be used for the same.

[22] "OpenDSA Data Structures and Algorithms Modules Collection," 13.12. *Heapsort - OpenDSA Data Structures and Algorithms Modules Collection*, accessed August 20, 2020, https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Heapsort.html.

4.2 Dependent Variable

The dependent variable is the time taken ***t*** by each algorithm to sort integer datasets of increasing size and the three respective ranges in nanoseconds. The *nanoTime()* method in the *Stopwatch* class will be used in order to determine the difference in time before and after the sorting execution with high precision and reduced random and systematic errors.

4.3 Controlled Variables

Variable	Description	Specifications
Computer and Operating System	The algorithms will be run on a Dell G3 3500 with Windows 10 Home.	**Processor**: Intel Core i7-10750U @ 3.0 GHz **OS**: Windows 10 x64 **Memory**: 8GB RAM (DDR3 – 12800)
Integrated Development Environment	The IntelliJ IDEA IDE will be used under the Apache 2 license.	**Version:** Community Edition 2020.2.1 **JDK and JRE**: Java SE 8u261
Probability Distribution of Datasets	All datasets will have discrete uniform distribution within the given ranges.	**RNG:** PineTools
Mean of Datasets	The mean for all datasets will be within [-100, 100].	The mean will be fairly constant since the datasets will be randomly distributed on both sides of the mid-range.
Data types	Only the ***int*** (32 bit) primitive data type will be used to represent the numbers. ***long*** (64 bit) will be used to store the sorting times in nanoseconds.	

Table 1 – List of Controlled Variables

4.4 Procedure

1. Set up both *TreeSort.java* and *HeapSort.java* files (refer **Appendix A, B)** in an IntelliJ project folder.
2. Using the PineTools random number generator, generate 10 randomized integer datasets each for the ranges of $[-\mathbf{1} \times \mathbf{10^5}, \mathbf{1} \times \mathbf{10^5}]$, $[-\mathbf{2} \times \mathbf{10^5}, \mathbf{2} \times \mathbf{10^5}]$, and $[-\mathbf{3} \times \mathbf{10^5}, \mathbf{3} \times \mathbf{10^5}]$ with number of integers $\boldsymbol{n}$ ranging from $\mathbf{10000}$ to $\mathbf{100000}$ (30 datasets in total).
3. Transfer 30 datasets as properties (key-value pairs) in a *.properties* file (refer **Appendix D**).
4. Create a new file *SortLauncher.java* (refer **Appendix C)** and using the *Properties* and *FileInputStream* classes, load all 30 datasets into an instance of the *Properties* class.
5. Access the required dataset using the *getProperty()* method of the *Properties* class and convert it into a String array.
6. Use the *convertStringToIntegerArray()* method to parse the String array and convert it into an integer array.
7. Finally create instances of the *HeapSort* and *TreeSort* classes and run the *sort()* methods with the integer array (unsorted dataset) as the argument.
8. Refer to the terminal to record the sorting times for both the Heap Sort and the Tree Sort.
9. Re-run *SortLauncher.java* for all 30 datasets by changing the property being accessed in IntelliJ's debug configuration. Perform 3 trials for each dataset and take average times for both sorting algorithms.

5. Data Processing and Graphing

5.1 Raw Data Collection

It must noted that both the sorting algorithms chosen are reliable, efficient, and concisely follow the expected algorithm paradigms. All applications were closed during algorithm execution and startup programs were disabled to free up RAM.

Tree Sort ($R = 2 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**
10000	5180800	5552200	4887200	5206733
20000	6906800	8257100	6790100	7318000
30000	10384800	11157500	9796600	10446300
40000	13978400	13087600	13268100	13444700
50000	16832100	17881100	17600200	17437800
60000	20705700	19277100	20636100	20206300
70000	26992600	24162600	24371900	25175700
80000	30919700	28979400	27821400	29240167
90000	32581000	34838000	32718600	33379200
100000	36683100	37547000	38884900	37705000

Table 2 – Tree Sort Sorting Times for Dataset Range of 2×10^5

Tree Sort ($R = 4 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**
10000	5549400	5294000	5268000	5370467
20000	8860400	6638000	6802800	7433733
30000	10220600	10049500	10121800	10130633
40000	12858600	13255600	13372200	13162133
50000	18960000	16757900	16434400	17384100
60000	22185400	20818300	20075100	21026267
70000	24875900	24466000	29302900	26214933
80000	30017000	28094700	28928600	29013433
90000	34962100	30965200	33269600	33065633
100000	40465600	35123900	37169900	37586467

Table 3 – Tree Sort Sorting Times for Dataset Range of 4×10^5

Tree Sort ($R = 6 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**

10000	5091700	5427700	5304200	5274533
20000	8692600	6833000	7104800	7543467
30000	10113000	10327300	9529500	9989933
40000	13037200	13377000	13058100	13157433
50000	17986600	17751900	17198000	17645500
60000	20253300	22187500	21088400	21176400
70000	26975100	24872600	24872600	25573433
80000	28565400	27309300	29578600	28484433
90000	32314900	31227900	32442500	31995100
100000	35997400	37716000	36756000	36823133

Table 4 – Tree Sort Sorting Times for Dataset Range of 6×10^5

Heap Sort ($R = 2 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**
10000	3368400	3075900	3037200	3160500
20000	4787400	5692900	5325300	5268533
30000	7649300	6399900	6786700	6945300
40000	7996800	9042800	8338200	8459267
50000	11089600	9162200	10031800	10094533
60000	12513000	12316200	13213400	12680867
70000	14394300	14172500	15126500	14564433
80000	16780200	16651600	15549900	16327233
90000	18654000	20902600	19026000	19527533
100000	22983400	23409800	22394000	22929067

Table 5 – Heap Sort Sorting Times for Dataset Range of 2×10^5

Heap Sort ($R = 4 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**
10000	4263500	3449400	2873100	3528667
20000	4763900	5012500	5104200	4960200
30000	6562400	6802300	7014200	6792967
40000	8907500	9081900	8131600	8707000

50000	10824800	10007900	11792000	10874900
60000	13105800	12784400	12922700	12937633
70000	14211300	15064200	14191600	14489033
80000	16805600	15610600	15613800	16010000
90000	17475000	19076200	19892500	18814567
100000	21968100	20979500	22872300	21939967

Table 6 – Heap Sort Sorting Times for Dataset Range of 4×10^5

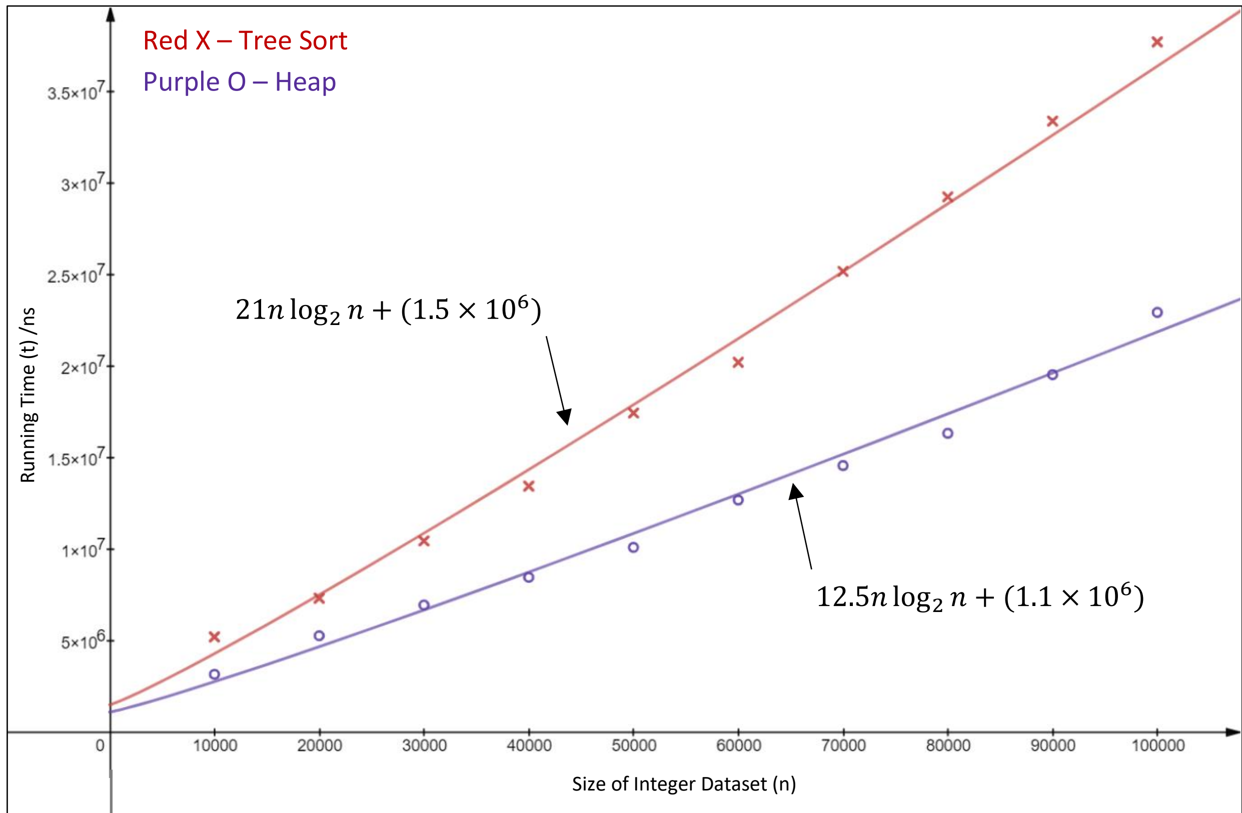
Heap Sort ($R = 6 \times 10^5$)				
Size of Integer Dataset (n)	**Time 1 /ns**	**Time 2 /ns**	**Time 3 /ns**	**Average Time (t) /ns**
10000	4030400	3227700	3234700	3497600
20000	4942000	5500100	4996900	5146333
30000	7501900	6442900	6890100	6944967
40000	8816200	8823800	9018200	8886067
50000	10778800	10092000	11111700	10660833
60000	11765900	12241500	12317200	12108200
70000	13450000	15232200	13602900	14095033
80000	16182800	17037500	15025700	16082000
90000	18873500	17315100	18102500	18097033
100000	21992000	21936400	21617100	21848500

Table 7 – Heap Sort Sorting Times for Dataset Range of 6×10^5

5.2 Graphs and Curve Fitting

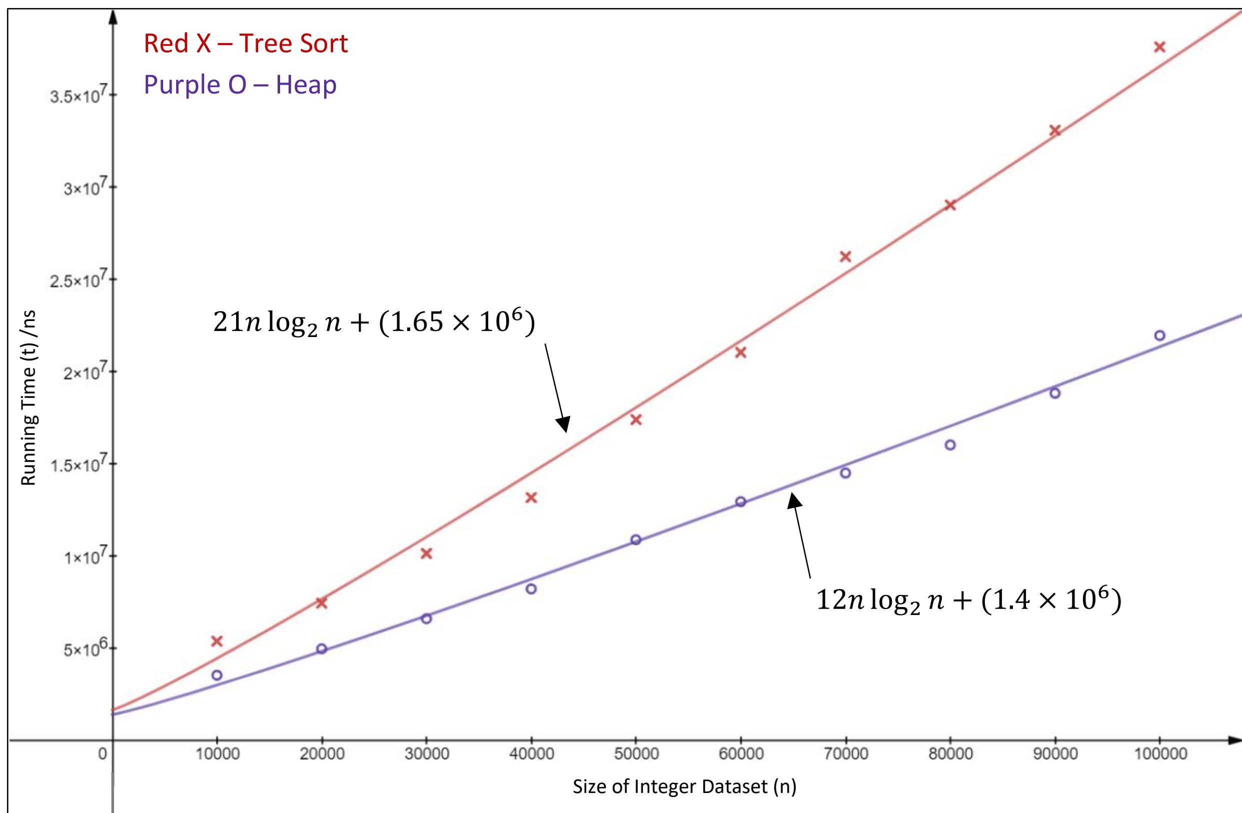
The above average times have been graphed first comparatively between the two algorithms for each range, and then for each algorithm individually with all three ranges. Since all trends followed a **linearithmic** pattern, only some minor transformations were required in order to curve fit $\boldsymbol{n \log_2 n}$ effectively. The two primary function transformations shown throughout are vertical dilation by a certain factor and vertical translation upwards by a certain number due to systematic error.

Tree Sort vs Heap Sort ($R = 2 \times 10^5$)



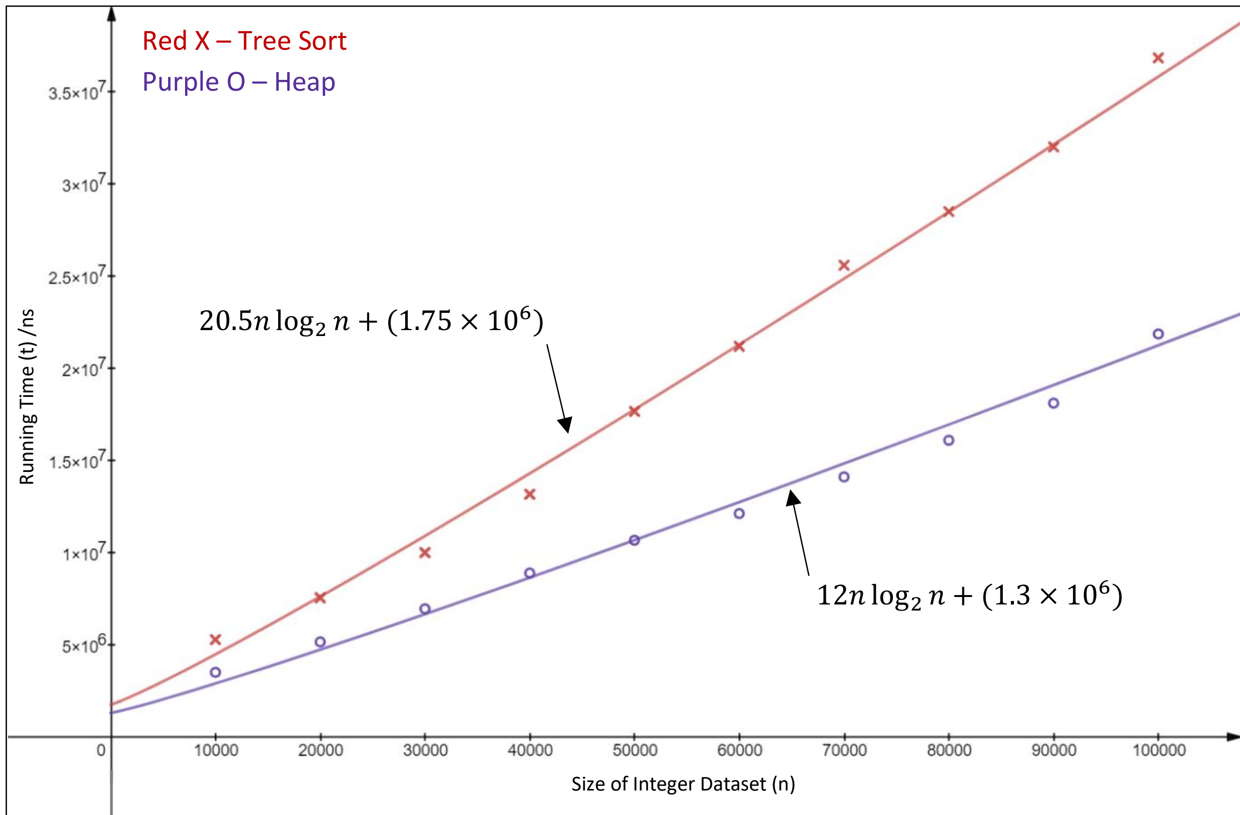
Graph 1 – Size of Dataset vs Sorting Time for Dataset Range of 2×10^5

Tree Sort vs Heap Sort ($R = 4 \times 10^5$)



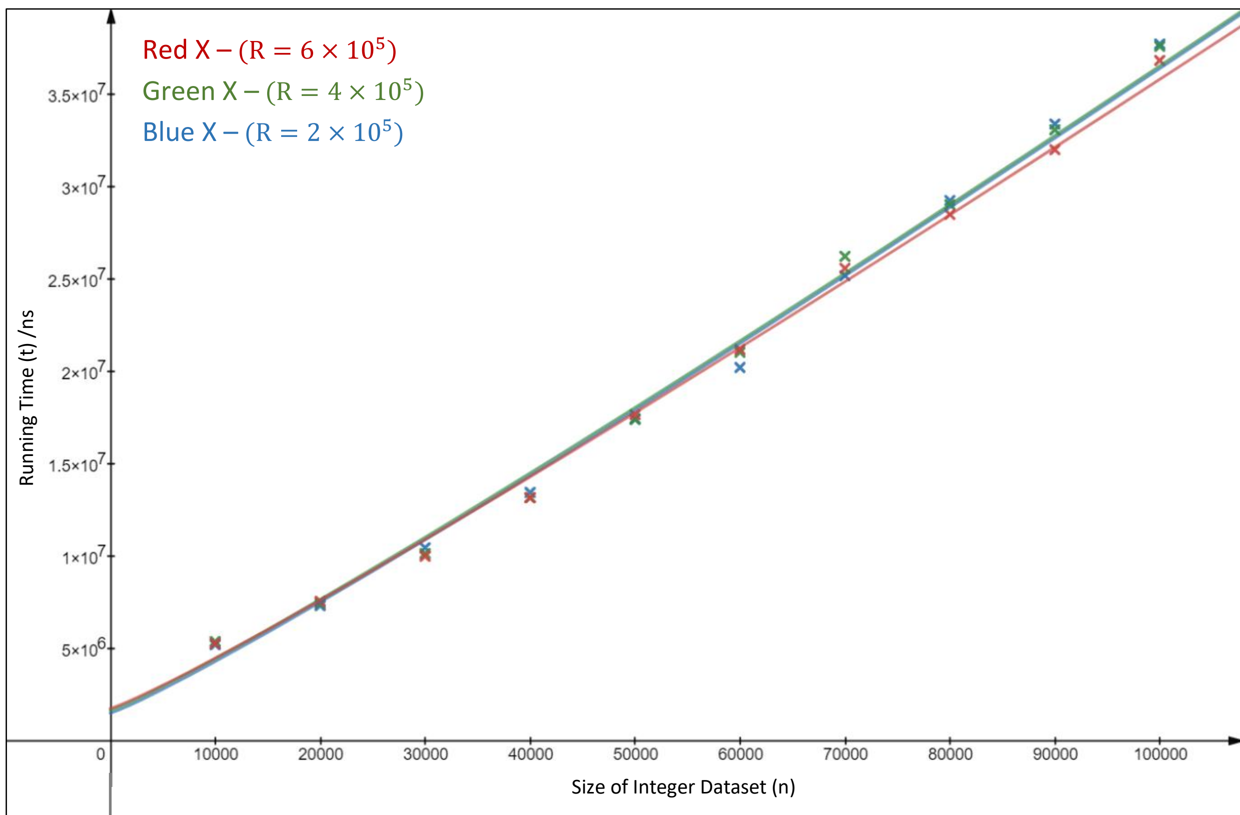
Graph 2 – Size of Dataset vs Sorting Time for Dataset Range of 4×10^5

Tree Sort vs Heap Sort ($R = 6 \times 10^5$)



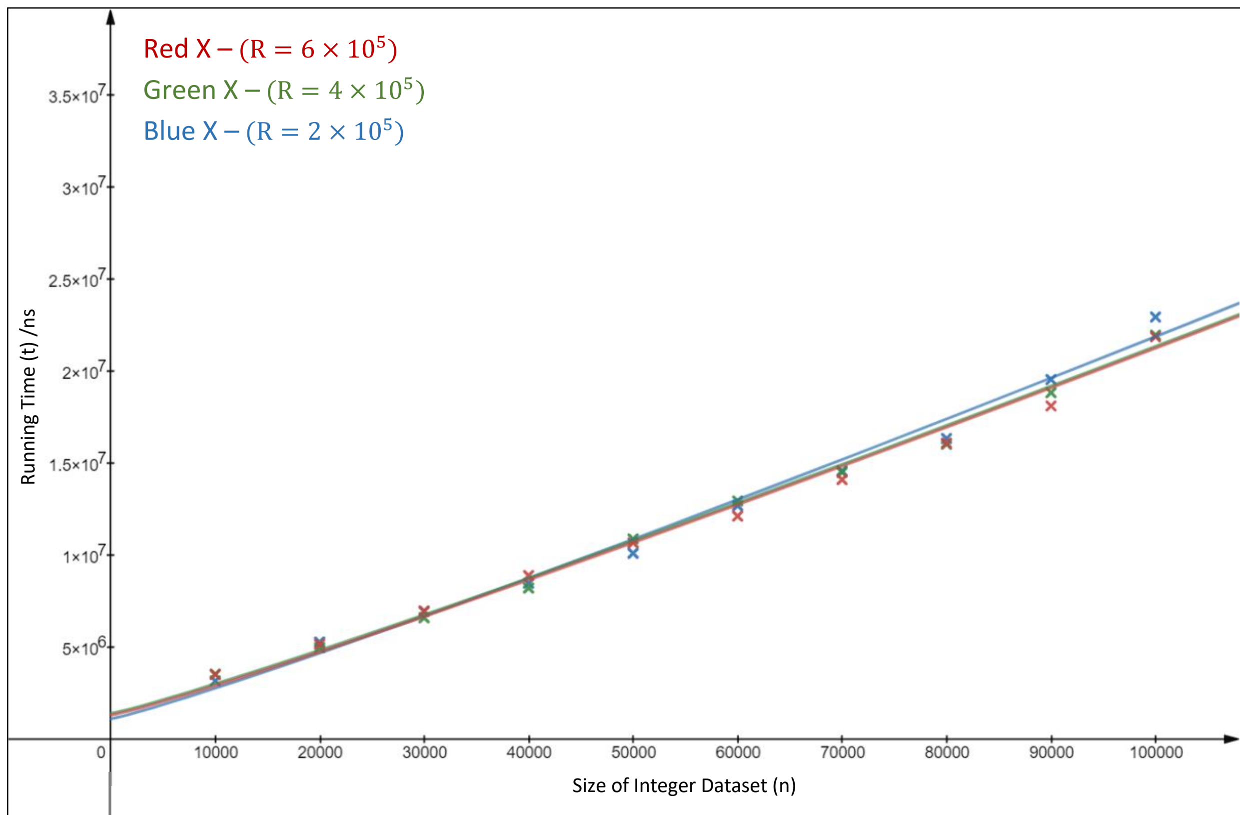
Graph 3 – Size of Dataset vs Sorting Time for Dataset Range of 6×10^5

Tree Sort All 3 Ranges



Graph 4 – Size of Dataset vs Tree Sort Sorting Time for All 3 Dataset Ranges

Heap Sort All 3 Ranges



Graph 5 – Size of Dataset vs Heap Sort Sorting Time for All 3 Dataset Ranges

6. Analysis

Therefore, in every trend present in the above graphs, it is evident that a linearthimic function is able to effectively model the data. However, to quantify the goodness of fit we must find the coefficient of determination, using the Pearson correlation coefficient, which has been computed and shown in the table below.

Algorithm Type	$R = 2 \times 10^5$	$R = 4 \times 10^5$	$R = 6 \times 10^5$
Tree Sort	0.9918	0.9921	0.9939
Heap Sort	0.9858	0.9924	0.9894

Table 8 – Coefficients of Determination for Best Fit Curves

This shows that there is a **very strong linearithmic relationship** between the input dataset size and the sorting time. However, due to the inconclusiveness of Pearson correlation for non-linear relationships, nonparametric Spearman rank-order correlation coefficients $\boldsymbol{\rho}$ must also be computed.

Algorithm Type	$R = 2 \times 10^5$	$R = 4 \times 10^5$	$R = 6 \times 10^5$
Tree Sort	1.0000	1.0000	1.0000
Heap Sort	1.0000	1.0000	1.0000

Table 9 – Spearman Rank-Order Correlation Coefficients for Best Fit Curves

This shows that along with goodness of fit, all the XY (size vs time) values can be perfectly modelled with a monotonically increasing function[23], which in this case is linearithmic, hence proving the efficacy of our model.

Finally, to determine the appropriacy of the linearithmic model, we must also use T-tests in order to find P-values for the data.

Algorithm Type	$R = 2 \times 10^5$	$R = 4 \times 10^5$	$R = 6 \times 10^5$
Tree Sort	1.23×10^{-9}	1.06×10^{-9}	3.71×10^{-10}
Heap Sort	1.13×10^{-8}	9.00×10^{-10}	3.50×10^{-9}

Table 10 – P-Values of Tree and Heap Sort Data

This shows that the data is highly statistically significant. Hence, assuming that the null hypothesis is that there is NOT a significant linearithmic relationship between $\boldsymbol{n}$ and $\boldsymbol{t}$, this low P-value < 0.05 ($\boldsymbol{\alpha} -$ significance level) indicates that there is extremely high probability that the alternate hypothesis is true i.e., the presence of a strong linearithmic relationship, which our best fit functions clearly support.

Contrarily, the Heap Sort evidently has lower sorting times than the Tree Sort for all dataset sizes and all three ranges. For example, for $\boldsymbol{R = 2 \times 10^5}$, the average time taken to sort 10000 integers by Heap Sort was 3160500 nanoseconds, around 39% lower than the 5206733 nanoseconds sorting time for the Heap Sort. In fact, as the size of the dataset increases, the average and instantaneous sorting time per integer for Heap Sort changes much slower than the

[23] A.W. Bowman, M. C. Jones, and I. Gijbels, "Testing Monotonicity of Regression," *Journal of Computational and Graphical Statistics* 7, no. 4 (1998): 489-500, accessed November 4, 2020, https://doi.org/10.2307/1390678.

Tree Sort making the difference between the performances of both algorithms significantly more pronounced. To examine this, we can compute the derivatives of the best fit functions –

Tree Sort $\left(\boldsymbol{R = 2 \times 10^5}\right)$ –

$$\boldsymbol{T(n) = 21n\log_2 n + (1.5 \times 10^6)}$$

$$\boldsymbol{\Rightarrow T'(n) = 21\left(\log_2 n + \frac{1}{\ln(2) \cdot n}\right)}$$

Using the Chain Rule

Equation 3 – Tree Sort Trend Line Derivative

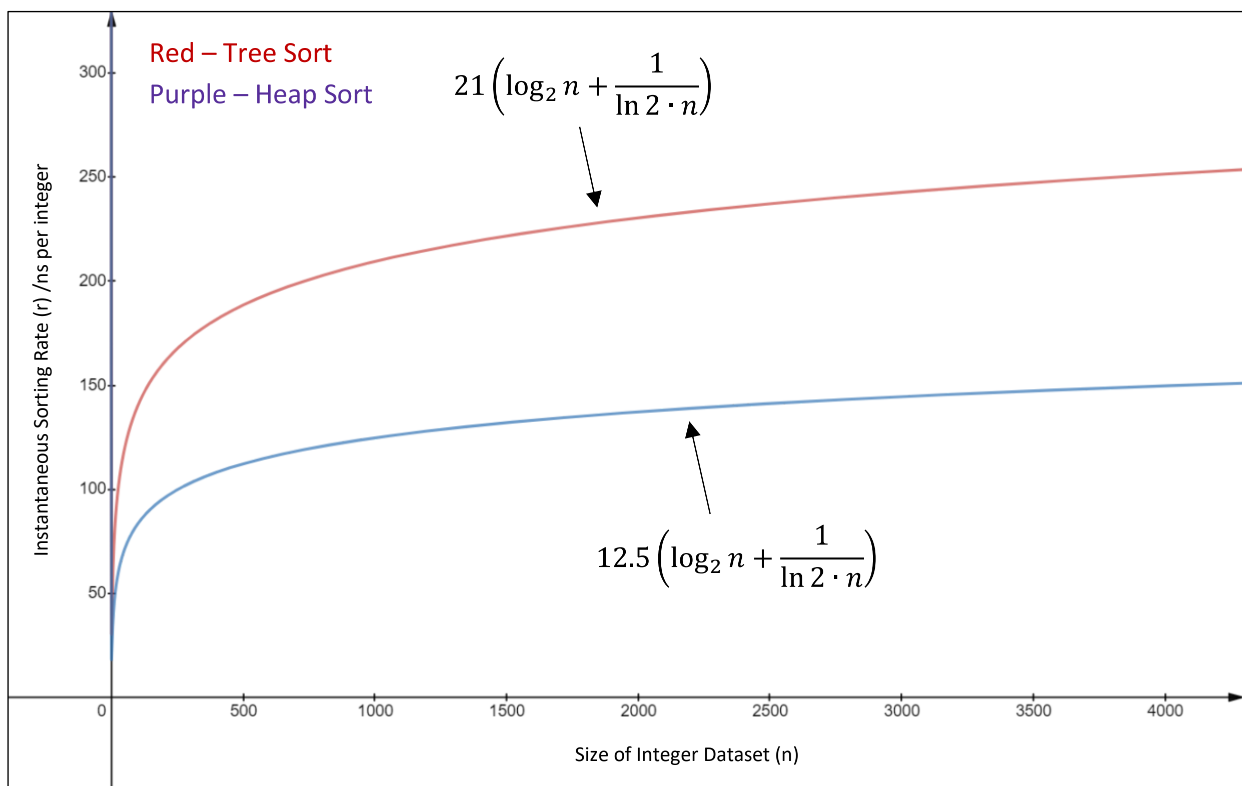
Heap Sort $\left(\boldsymbol{R = 2 \times 10^5}\right)$ –

$$\boldsymbol{H(n) = 12.5n\log_2 n + (1.1 \times 10^6)}$$

$$\boldsymbol{\Rightarrow H'(n) = 12.5\left(\log_2 n + \frac{1}{\ln(2) \cdot n}\right)}$$

Equation 4 – Heap Sort Trend Line Derivative

Tree Sort vs Heap Sort Derivatives $\boldsymbol{(R = 2 \times 10^5)}$



Graph 6 – Size of Dataset vs Tree and Heap Sort Instantaneous Sorting Times

For example, at $\boldsymbol{n}$ = 10000, Tree sort took 521 nanoseconds per integer on average and Heap Sort took 316 nanoseconds per integer on average $\left(\approx \frac{\boldsymbol{H(1000)}}{\boldsymbol{1000}}\right)$. However, at $\boldsymbol{n}$ = 100000, average times of 377 nanoseconds and 229 nanoseconds per integer were taken respectively.

According to Graph 6, at $\boldsymbol{n}$ = 10000, Tree sort had an instantaneous sorting rate of 279 nanoseconds per integer, 68% greater than the Heap Sort's rate of 166 nanoseconds per integer ($\boldsymbol{H}'(\mathbf{1000})$). However, at $\boldsymbol{n}$ = 100000, the instantaneous sorting rates were 349 and 207 nanoseconds per integer respectively. This proves that the Heap Sort is sorting in significantly lower time and also scaling up at a much lower rate than the Tree Sort as depicted in the graph.

Furthermore, it is also evident that for both Tree Sort and Heap Sort, the sorting time trends for the three ranges barely vary. Apart from a slightly increases and decreases in sorting times across ranges, we cannot conclusively say whether $\boldsymbol{t}$ is proportionally related to $\boldsymbol{R}$. In this case, a Kruskal-Wallis one-way ANOVA test can be done on the data collected for the three ranges for both algorithms. Tree Sort had a P-value of 0.9961 and Heap Sort had a P-value of 0.9885. Hence, we can say that the probability of the null hypothesis being true i.e., there is no relationship between $\boldsymbol{R}$ and $\boldsymbol{t}$, is very high.

Finally, some random and systematic error is present in the data. As seen with the graphs for $\boldsymbol{R} = \mathbf{2} \times \mathbf{10^5}$, the y-intercept of the Tree Sort best fit function of 1.5×10^6 is around 36% greater than the y-intercept of the Heap Sort best fit function of 1.1×10^6. Furthermore, points such as (60000, 20206300) for Tree Sort and (50000, 10094533) for Heap Sort show significant deviation from the best fit function. The reasons for these random and systematic errors will be discussed in Evaluation.

7. Results Discussion & Evaluation

Hence, as conclusive results have been obtained, numerous means can be used to justify the same. Firstly, it is apparent that the naturally self-balancing nature of the binary heap gives it an advantage. This is because for a similar number of inserted integers, a max-heap constructs a tree with the minimum number of levels required (since the comparisons between adjacent

nodes are only done **after** the integers are inserted) while a BST is not concerned with the same (since the comparisons between existing nodes and a to-be-inserted integers are done **before** they are inserted). The same is illustrated below –

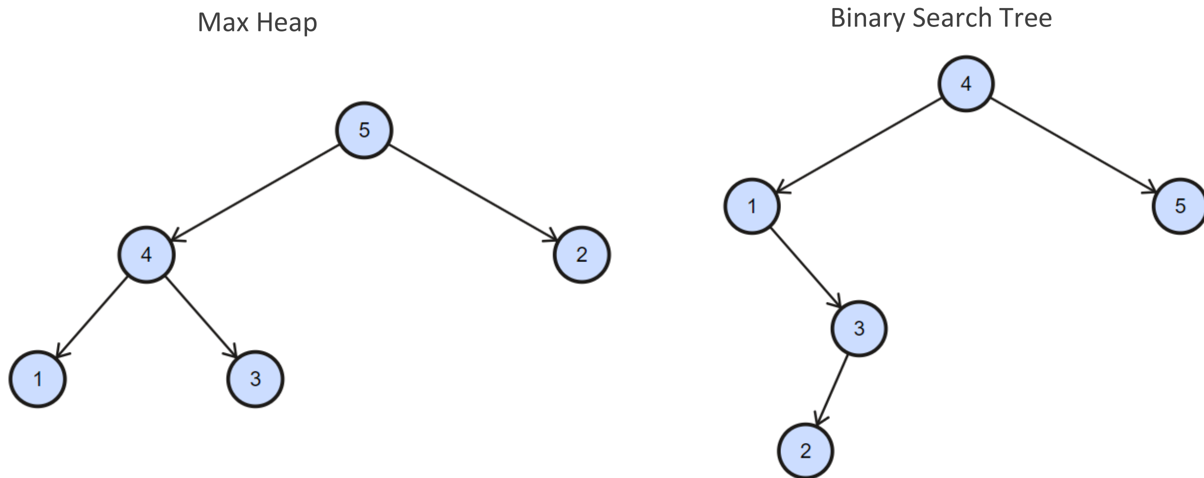


Figure 19 – Balanced vs Unbalanced Binary Search Tree Example

As shown, the max heap has the minimum of 3 levels as required by 5 nodes, whereas the BST has 4 levels due to its unbalanced nature therefore making it take longer to carry out a depth first traversal. This is also supported by the fact that the worst-case complexity of an unbalanced Tree Sort is $\boldsymbol{O(n^2)}$[24] (showing that a BST can be constructed as a straight chain: having as many levels as nodes) while the Heap Sort's is $\boldsymbol{O(n\log_2 n)}$.

Another perspective that must be considered is that of Recursion vs Iteration. The Tree Sort is a purely recursive algorithm with both the insertion and traversal of every node being done recursively. The Heap Sort, contrarily, does heapification/insertion recursively but conducts the traversal iteratively. The heapification is also optimized since the algorithm iteratively visits the roots of sub-trees that need to be heapified after which recursion takes over.

The reason recursion is slower than iteration is that, when considering depth-first traversals as

[24] Alexa Ryder, "Tree Sort Algorithm," *OpenGenus IQ: Learn Computer Science* (OpenGenus IQ: Learn Computer Science, March 18, 2018), accessed November 12, 2020, https://iq.opengenus.org/tree-sort/.

used by the BST, each successive recursive call to the *insert()* or *dfs()* (refer **Appendix** A) functions gets added as a stack frame to the top of a call stack (a linear data structure that follows the last in first out principle)[25] from which the recursive subroutines take place. This call stack thus necessitates the allocation of excess overhead time and memory (iteration does not require this) consequently explaining why the tree sort is more time-intensive.

By the same token, it must also be realized that that the $\boldsymbol{O(1)}$ space complexity[26] of the Heap Sort is also a massive advantage compared to the $\boldsymbol{O(n)}$ space complexity of the Tree Sort. The fact that the Heap Sort can use array indices as node pointers allows it to quickly sort the dataset within the array itself. Contrarily, integers in the Tree Sort must be assigned to an object along with two other pointers. Hence not only does a larger dataset require more time and memory to create more objects, the fact that an integer itself has a 12 byte overhead in an object[27] is a huge memory allocation time-waste for the Tree Sort.

Finally, the minimal systematic error can obviously be attributed to the *javac* compile time of the algorithms since at $\boldsymbol{n = 0}$, the runtime is negligible yet the y-intercept of the linearithmic functions is not 0. Virtual memory stored on the PC could have also contributed to compiler lag. The random errors could have been caused by algorithm runtime being affected by constantly changing CPU clock-speeds due to the varying processing consumption.

8. Conclusion

Therefore, with reference to my hypothesis "I hypothesize that the Heap Sort will sort the dataset in lower time than the Tree Sort. There will be a **linearthimic relationship** between $\boldsymbol{n}$

[25] "4.3. What Is a Stack?" 4.3. *What Is a Stack? - Problem Solving with Algorithms and Data Structures*, accessed November 12, 2020, https://www.runestone.academy/runestone/books/published/pythonds/BasicDS/WhatisaStack.html.
[26] Time Complexity and Space Complexity comparison of Sorting Algorithms, *Scanftree*, accessed November 12, 2020, https://www.scanftree.com/Data_Structure/time-complexity-and-space-complexity-comparison-of-sorting-algorithms.
[27] Java Tips By Vladimir Roubtsov and Vladimir Roubtsov, "Java Tip 130: Do You Know Your Data Size?," *InfoWorld* (JavaWorld, August 16, 2002), accessed November 12, 2020, https://www.infoworld.com/article/2077496/java-tip-130--do-you-know-your-data-size-.html.

and $\boldsymbol{t}$", this experiment was able to comparatively determine the time complexities and hence the sorting efficiency of both algorithms and provide conclusive evidence for the fact that the Heap Sort always sorts in lower time than the Tree Sort and that the relationship between size of the randomized integer dataset $(\boldsymbol{n})$ and time taken to sort $(\boldsymbol{t})$ is most closely linearithmic: proving my hypothesis correct.

9. Further Scope

Therefore, considering that the primary limitation of the BST is that it is unbalanced, the performance of the BST can be improved by using a self-balancing red-black tree for insertions in order to avoid skewed trees and consequently worst-case complexities. In addition to this, adaptive variants of both sorting algorithms (adaptive heap sorts and splay sorts)[28] could also reduce running time by exploiting any partially ordered input data.

Furthermore, the tree sort's space inefficient recursive logic can be solved using an iterative variant of the algorithm so that additional time required by the call stack can be avoided. Contrarily, utilizing a ternary instead of a binary heap could be useful since the height of the tree could now be decreased to $\boldsymbol{\log_3 n}$ from $\boldsymbol{\log_2 n}$.[29] So, while the comparisons per level would increase, the number of levels recursed through itself would be lower.

Ultimately, the differences between the running times for datasets of various ranges could have been made more significant if larger ranges of *long* data type integers were used. The type of integer distribution used such as Gaussian or Poisson distributions could also be added as another complex parameter.

---

[28] Alistair Moffat, *Splaysort: Fast, Versatile, Practical*, accessed November 12, 2020, https://people.eng.unimelb.edu.au/ammoffat/abstracts/spe.splay.html.

[29] Kosmopo, "set3solutions," University of Texas at Arlington, accessed November 12, 2020, http://ranger.uta.edu/~kosmopo/cse5311/homework/set3solution.pdf.

Bibliography

Aggarwal, Shivi. "HeapSort." *GeeksforGeeks*, Last Modified November 16, 2020. Accessed August 18, 2020. https://www.geeksforgeeks.org/heap-sort/.

"Asymptotic Analysis: Big-O Notation and More." *Programiz.* Accessed July 12, 2020. https://www.programiz.com/dsa/asymptotic-notations.

"Big-O Notation (Article) | Algorithms." *Khan Academy*, Khan Academy. Accessed July 12, 2020. https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation.

"Binary Heaps." *Heaps*, Andrew CMU. Accessed August 18, 2020. http://www.andrew.cmu.edu/course/15-121/lectures/Binary%20Heaps/heaps.html.

Bowman, A. W., M. C. Jones, and I. Gijbels. "Testing Monotonicity of Regression." *Journal of Computational and Graphical Statistics* 7, no. 4 (1998): 489-500. Accessed November 4, 2020. https://doi.org/10.2307/1390678.

"CS 312 Lecture 25: Priority Queues and Binary Heaps." *Lecture 25: Priority Queues and Binary Heaps*. Accessed August 18, 2020. https://www.cs.cornell.edu/courses/cs312/2007sp/lectures/lec25.html.

"Data Structure - Binary Search Tree." *Tutorialspoint.* Accessed July 24, 2020. https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm.

"Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) Based Sorting Algorithms?" *Javarevisited*. Accessed July 12, 2020. https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#axzz6nplsEjux.

Javinpaul. "How to Implement Inorder Traversal in a Binary Search Tree?" *DEV Community*, DEV Community, August 14, 2019. Accessed July 24, 2020. https://www.dev.to/javinpaul/how-to-implement-inorder-traversal-in-a-binary-search-tree-1787.

Joshi, Nikhil. "Implementation and Analysis of Merge Sort." *Dotnetlovers*, Dotnetlovers, October 29, 2018. Accessed July 12, 2020. https://www.dotnetlovers.com/article/128/implementation-and-analysis-of-merge-sort.

Kosmopo. *"set3solutions."* University of Texas at Arlington. Accessed November 12, 2020. http://ranger.uta.edu/~kosmopo/cse5311/homework/set3solution.pdf.

Moffat, Alistair. *Splaysort: Fast, Versatile, Practical*. Accessed November 12, 2020. https://people.eng.unimelb.edu.au/ammoffat/abstracts/spe.splay.html.

Moore, Karleigh. “Sorting Algorithms.” *Brilliant Math & Science Wiki*. Accessed July 24, 2020. https://www.brilliant.org/wiki/sorting-algorithms/.

M, Vibin. “Tree Sort.” *GeeksforGeeks*, April 20, 2020. Accessed August 1, 2020. https://www.geeksforgeeks.org/tree-sort/.

“OpenDSA Data Structures and Algorithms Modules Collection.” 13.12. *Heapsort - OpenDSA Data Structures and Algorithms Modules Collection*. Accessed August 20, 2020. https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Heapsort.html.

“Properties Class (Java Platform SE 8).” *Oracle Java Documentation*. Accessed October 24, 2020. https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html.

Roubtsov, Java Tips By Vladimir, and Vladimir Roubtsov. “Java Tip 130: Do You Know Your Data Size?” *InfoWorld*, JavaWorld, August 16, 2020. Accessed November 12, 2020. https://www.infoworld.com/article/2077496/java-tip-130--do-you-know-your-data-size-.html.

Ryder, Alexa. “Tree Sort Algorithm.” *OpenGenus IQ: Learn Computer Science*, OpenGenus IQ: Learn Computer Science, March 18, 2018. Accessed November 12, 2020. https://iq.opengenus.org/tree-sort/.

Sedgewick, Robert, and Kevin Wayne. “Binary Search Trees.” *Princeton University*, The Trustees of Princeton University. Accessed July 24, 2020. https://algs4.cs.princeton.edu/32bst/.

Singh, Navjot. “Why Is Binary Heap Never Unbalanced?” *Computer Science Stack Exchange*, May 2, 2019. Accessed August 18, 2020. https://cs.stackexchange.com/questions/108852/why-is-binary-heap-never-unbalanced.

Time Complexity and Space Complexity comparison of Sorting Algorithms. *Scanftree*. Accessed November 12, 2020. https://www.scanftree.com/Data_Structure/time-complexity-and-space-complexity-comparison-of-sorting-algorithms.

TimTim 1. “Divide and Conquer and Recursion.” *Stack Overflow,* January 1, 2009. Accessed July 12, 2020. https://www.stackoverflow.com/questions/2249767/divide-and-conquer-and-recursion.

“4.3. What Is a Stack?” 4.3. *What Is a Stack? - Problem Solving with Algorithms and Data Structures*. Accessed November 12, 2020. https://www.runestone.academy/runestone/books/published/pythonds/BasicDS/WhatisaStack.html.

<u>Software & Online Tools Used</u> –

2011. *Desmos Graphing Calculator*. San Francisco, California, United States: Desmos Inc. Accessed November 1, 2020. https://www.desmos.com/.

2001. *IntelliJ Idea Community Edition*. Prague, Czech Republic: JetBrains s.r.o. Accessed October 21, 2020. https://www.jetbrains.com/idea/.

Melezinek, J., n.d. *Binary Tree Visualizer*. CTU FIT Web and Multimedia. Accessed July 24, 2020, and August 4, 2020. http://www.btv.melezinek.cz/binary-search-tree.html.

Pinetools – Online Random Number Generator. Pinetools. Accessed October 23, 2020. https://www.pinetools.com/random-number-generator.

2011. *Prism Free Trial*. San Diego, California, United States: GraphPad Software Inc. Accessed November 2, 2020. https://www.graphpad.com/scientific-software/prism/.

Appendices

Appendix A – TreeSort.java[30]

```
1. package com.company;
2.
3. public class TreeSort {
4.
5.         public static class Node {
6.             int key; // Integer value of the node
7.             Node left, right; // Pointers to left and right child nodes
8.
9.             public Node(int item) {
10.                key = item;
11.                left = right = null;
12.            }
13.        }
14.
15.        Node root;
16.
17.        public TreeSort() {
18.            root = null;
19.        }
20.
21.        public Node insert(Node node, int key) {
22.            if (node == null) {
23.                node = new Node(key); // Creating a new tree
24.                return node;
25.            }
26.            if (key < node.key)
27.                node.left = insert(node.left, key);
28.
29.            else if (key > node.key)
30.                node.right = insert(node.right, key);
31.
32.            return node;
33.        }
34.
35.        public void dfs(Node node) {
36.            if (node != null) {
37.                dfs(node.left); // Recursing down the left sub-tree
38.                int nodeValue = node.key;
39.                dfs(node.right); // Recursing down the right sub-tree
40.            }
41.        }
42.
43.        public void sort(int[] arr) {
44.            long startTime = System.nanoTime(); // Stopwatch start
45.
46.            for (int j : arr) {
```

[30] Vibin M, "Tree Sort," *GeeksforGeeks*, April 20, 2020, accessed August 1, 2020, https://www.geeksforgeeks.org/tree-sort/.

```
47.                 root = insert(root, j);
48.             }
49.
50.             dfs(root);
51.
52.             long stopTime = System.nanoTime(); // Stopwatch stop
53.
54.             System.out.println("\n\nTree Sort Start Time: " +
   startTime);
55.             System.out.println("\nTree Sort Stop Time: " + stopTime);
56.             System.out.println("\nTime Taken To Tree Sort: " +
   (stopTime - startTime));
57.         }
58.     }
```

Appendix B – HeapSort.java[31]

```
1. package com.company;
2.
3. public class HeapSort {
4.
5.         public void sort(int[] arr) {
6.             int n = arr.length;
7.
8.             long startTime = System.nanoTime(); // Stopwatch start
9.
10.             for(int i = n / 2 - 1; i >= 0; i--) // Building max heap
11.                 heapify(arr, n, i);
12.
13.             for(int i = n - 1; i > 0; i--) {
14.                 int temp = arr[0]; // Moving current root to end
15.                 arr[0] = arr[i];
16.                 arr[i] = temp;
17.
18.                 heapify(arr, i, 0); // Heapifying reduced heap
19.             }
20.             long stopTime = System.nanoTime(); // Stopwatch stop
21.
22.             System.out.println("\nHeap Sort Start Time: " + startTime);
23.             System.out.println("\nHeap Sort Stop Time: " + stopTime);
24.             System.out.println("\nTime Taken To Heap sort: " +
   (stopTime - startTime));
25.
26.             System.out.println("\nHeap Sorted Array Is: ");
27.             printArray(arr);
28.         }
29.
30.         void heapify(int[] arr, int n, int i) {
```

[31] Shivi Aggarwal, "HeapSort," *GeeksforGeeks*, Last Modified November 16, 2020, accessed August 18, 2020, https://www.geeksforgeeks.org/heap-sort/.

```
31.             int largest = i; // Initializing largest as root
32.             int l = 2*i + 1; // Left Child = 2*i + 1
33.             int r = 2*i + 2; // Right Child = 2*i + 2
34.
35.             if(l < n && arr[l] > arr[largest])
36.                 largest = l;
37.
38.             if(r < n && arr[r] > arr[largest])
39.                 largest = r;
40.
41.             if(largest != i) {
42.                 int swap = arr[i];
43.                 arr[i] = arr[largest];
44.                 arr[largest] = swap;
45.
46.                 heapify(arr, n, largest); // Recursively heapifying
47.             }
48.         }
49.
50.         public void printArray(int[] arr) {
51.             for (int j : arr) {
52.                 System.out.print(j + ", ");
53.             }
54.             System.out.println();
55.         }
56.     }
```

Appendix C – SortLauncher.java

```
1. package com.company;
2.
3. import java.io.IOException;
4. import java.io.FileInputStream;
5. import java.util.Properties;[32]
6.
7. public class SortLauncher { // Program to run both sorting algorithms
8.
9.         public static int[] convertStringToIntegerArray(String[] string)
   {
10.             int[] arr = new int[string.length];
11.
12.             for(int i = 0; i < string.length; i++)
13.                 arr[i] = Integer.parseInt(string[i]);
14.
15.             return arr;
16.         }
17.
18.         public static void main(String[] args) throws IOException {
```

[32] "Properties Class (Java Platform SE 8)," *Oracle Java Documentation*, accessed October 24, 2020, https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html.

```
19.                FileInputStream fis = new
   FileInputStream("src/com/company/numbers.properties");
20.                Properties prop = new Properties();
21.                prop.load(fis); // Loading number.properties file
22.
23.                System.out.println(args[0] + " Input Array: " +
   prop.getProperty(args[0])); // This argument is the property's key
24.
25.                String[] inputNumber = prop.getProperty(args[0]).split(",
   ");
26.                int[] unsortedArray =
   convertStringToIntegerArray(inputNumber);
27.
28.                HeapSort heap = new HeapSort();
29.                heap.sort(unsortedArray); // Inputting preferred dataset
30.
31.                unsortedArray = convertStringToIntegerArray(inputNumber);
32.                TreeSort tree = new TreeSort();
33.                tree.sort(unsortedArray); // Inputting preferred dataset
34.            }
35.        }
```

Appendix D – number.properties (screenshot)