Performance Analysis of Heap, Merge and Insertion Sort

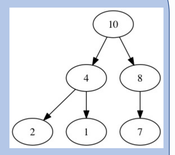
Siddhant Grover
Advisor: Dr. Sudharsan Iyengar
Department of Computer Science, Winona State University

Introduction

Numerical Data often has to be sorted for its application in different contexts and different fields. The data available is sorted through different algorithms based on different factors like time, efficiency, complexity, etc. This project focusses on taking large size arrays and comparing the sorting through Heap, Merge and Insertion sort algorithms in two high level languages: Java which is a compiled language and Python which is an interpreted language.

Heap Sort

Heapsort is a comparison-based sorting algorithm. Heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.



1) Convert the array into a maxheap → the max value element become the root of the new heap.
2) After getting the max value, it places the element at the last position of the array, then reduces the size of the heap by one → unsorted array size is also reduced by one. Repeat step 1 and 2
3) It recursively does this till the array is sorted.
   Its time complexity is $O(n \log n)$, in all cases.

Merge Sort

Merge Sort is a Divide and Conquer based sorting algorithm. The algorithm's motive is to divide the array into n subarrays of one element each where n is the size of the initial arrays. All these sub arrays are merged together to create new sorted sub arrays . This process is done till there will only be one sub array of size 1 left.

MergeSort(arr[], l, r) If r > l

1) Divide the array into two halves by arr$\left[\frac{l+r}{2}\right]$

2)Call mergeSort for first half: Call mergeSort(arr, l, m)

3)Call mergeSort for second half: Call mergeSort(arr, m+1, r)

4)Merge the two halves sorted in step 2 and 3: Call merge(arr, l, m, r)

The time complexity of Merge sort is $O(n \log n)$,in all cases.

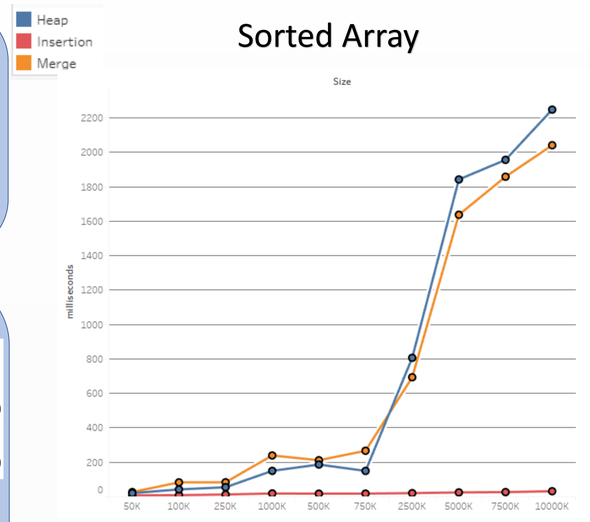
Insertion Sort

Insertion sort iterates, comparing one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it.

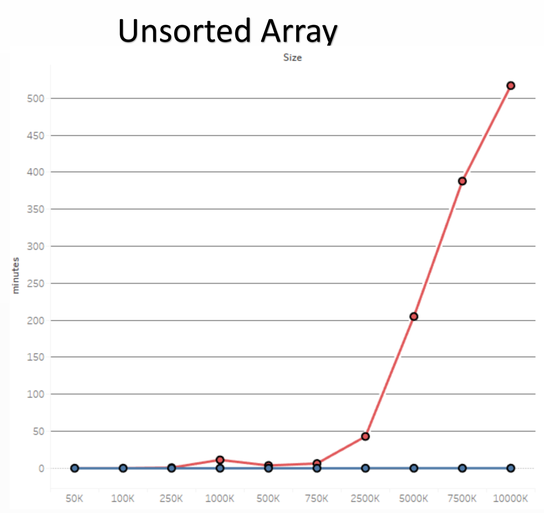
1)At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked).

2)If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

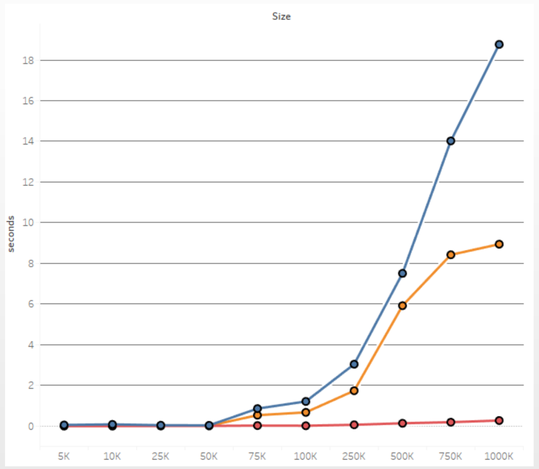
The time complexity of insertion sort is $O(n^2)$ for the worst case and $O(n)$ for the best case.



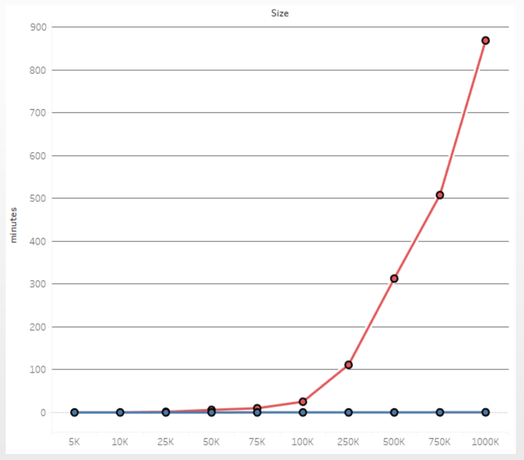
Graph 1: time taken to sort a sorted array in Java



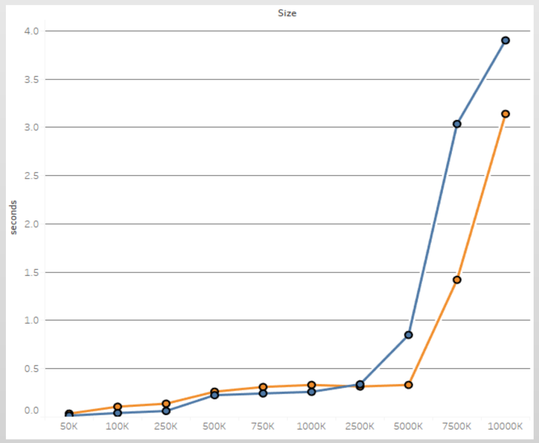
Graph 2: time taken to sort an unsorted array in Java



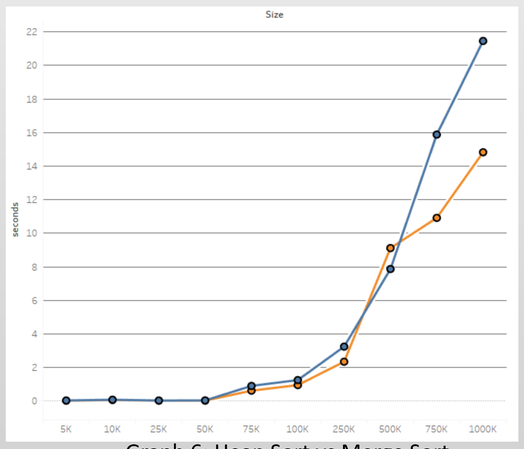
Graph 3: time taken to sort a sorted array in Python



Graph 4: time taken to sort an unsorted array in Python



Graph 5: Heap Sort vs Merge Sort
For an unsorted array in Java



Graph 6: Heap Sort vs Merge Sort
For an unsorted array in Python

Data

The data used for this project was purely Numerical.

Java: Large size sorted and unsorted arrays consisting of 1-n distinct elements. Where n varied from 50K to 10000K.

Python: Large size sorted and unsorted arrays consisting of 1-n distinct elements. Where n varied from 5K to 1000K.

*IDE used were **Eclipse** for Java and **Pycharm** for Python*

Methods

1) Every individual array is shuffled randomly between 4 and 7 times to create a randomly distributed unsorted array. The array is not shuffled more than 7 times to prevent over shuffling the array and, in some way, leading to a more sorted array.
2) The three sorting algorithms are implemented in Java and Python as functions. The functions are called on every individual array in both languages. The time taken to sort the array is noted.
3) This whole process is repeated for 10 iterations on different randomized arrays. The average values obtained after performing the sorts is noted. The time values for the different size arrays are then put together and visualized through Tableau.

Observations

1) The sort trends on the graph remain similar in both languages.
2) The time taken to run remains under a minute for all sorts In Java up to 250 K elements.
3) The time taken to sort an array in Java using merge and heap sorts remains under a minute for both the is $O(n \log n)$ algorithms(Merge and Heap sort).
4)At around 370K elements is when the time taken for insertion sort increases to more than a minute in Java.
5)Heap Sort runs faster in randomized arrays up to 2500K elements after which Merge sort works faster than heap sort in both sorted and unsorted arrays in Java.
6) Insertion Sort takes the highest time to run for an unsorted array.
7)Insertion Sort takes the least time to run for a sorted array.
8)Heap Sort works faster than Merge sort for up to 50K elements after which Mergesort runs faster in general, in Python.
9)Python: sorting take longer to run all the algorithms.

Acknowledgments

Special Thanks to the Computer Science Department of Winona State University, my academic advisor – Dr Tim Gegg-Harrison, my professors and mentors- Dr. Sudharsan Iyengar, Dr. Shimin Li, Dr. Joan Francioni, Dr. Yogesh Grover , and my peers.