Project I: Sorting and Heaps

CS 456
Design and Analysis of Algorithms
SIUE

VENKATESH BOLLINENI - 800792618
vbollin@siue.edu

1.

InsertionSort Expected Asymptotic Behaviour

a. In the worst case: For insertion sort in the wort case scenario the time complexity is *O(n^2),* this is because the array is sorted in reverse order. Every element needs to compare with all the previous elements and need to move to the front, eventually this leads to maximum number of comparisons and shifts as well. For an array of size n, the first element doesn't need any shift, second element need to do one shift and so on, up to (n-1) shifts for the last element. This combined operations leads to quadratic number of operations, the total number of shifts and comparisons is proportional to the sum of the first n-1 elements, which is (n*(n-1))/2, asymptotically *O(n^2)*.

b. In the best case: Best case time complexity for Insertion sort is *O(n),* this main reason for this when the array is already sorted then each element is compared only once with the previous element and yields in n-1 comparisons without any shifts. The InsertionSort algorithm traverses the array confirming the correct order of elements without any extra operations. This leads the algorithm to run in linear time because every element needs only single operation to check the position so the time complexity is *O(n)* for n elements.

c. In the average case: Insertion Sort average case scenario time complexity is *O(n^2),* this main reason for this is on average each element need to compare and move almost for the half of the length of array. While the quantity of elements increases then number of comparisons and shifts also increases quadratically. Even though not every element need n comparisons, the average still outcomes a significant number of operations where number of elements grows.

MergeSort Expected Asymptotic Behaviour

a. In the worst case: Worst case time complexity for MergeSort is *O(nlogn).* This is because MergeSort divides the array into two halves until the array reaches to a single element of size and merges them back together. Mathematically this dividing happens in *O(logn)* time and the merging process needs comparisons and shifts so this happens in *O(n)* time. Despite the initial order of the array, these division and merging back the elements in sorted order takes *O(nlogn)* time complexity in worst case.

b. In the best case: Best case time complexity for merge sort is also *O(nlogn),* this is because mergesort's performance doesn't improve even if the array is sorted. This is because, irrespective of array during the operation of MergeSort it still need to divide the array until it reaches to a single element and merges back with comparisons and shift. These two operations (dividing and merging back) is like worst case so the time complexity is *O(nlogn)*. We can say MergeSort doesn't benefit from already sorted files and maintains same time complexity for best and worst case scenarios.

c. In the average case: MergeSort's average case time complexity is same as best and worst cases time complexity *O(nlogn)*. The algorithm still need to divide and merge all the array elements regardless of the initial order of the elements. So the number of

operations doesn't change and making the average performance same as best and worst cases.

HeapSort Expected Asymptotic Behaviour

a. In the worst case: Worst case time complexity of HeapSort is *O(nlogn)*. This is because during the operation of HeapSort, it first builds a heap out of the array, which takes *O(n)* time, and continually pulls out the maximum element from the heap and rebuilds the heap, this operation(extraction and rebuild) takes *O(logn)* time. The same operations are need for every element in the array despite the original sequence of the elements so the overall worst case time complexity is *O(nlogn).*

b. In the best case: Best case time complexity is *O(nlogn)*. Like mentioned in the worst case scenario, the HeapSort will always build the heap and performs all the steps of sorting operations regardless of whether the beginning order of the array is sorted or not. So these steps takes *O(nlogn)* time complexity same as worst case. The Heapsort algorithm takes same time for already sorted and unsorted data.

c. In the average case: In the average case, time complexity for heapsort is *O(nlogn)*. The HeapSort is always efficient because it uses the same steps to turn the array into a heap and then sort it. Without considering the initial arrangement of array, the HeapSort performs all the steps same as best and worst cases, so it maintains the time complexity of *O(nlogn)*.

Runtime Table

Timings for sorting of word files (milliseconds)						
Words count	**Permuted files timings**			**Sorted files timings**		
	IS	**MS**	**HS**	**IS**	**MS**	**HS**
15,000	8839.598	72.088	98.48213	2.887	55.233	102.149
30,000	35733.76	149.697	213.75847	5.777	117.069	220.546
45,000	79809.69	234.08	329.11944	8.677	182.598	343.487
60,000	143005.2	321.023	455.36947	11.673	250.2	467.918
75,000	221708.7	408.995	579.28753	14.373	318.893	602.42
90,000	323197.1	499.364	714.75124	17.5	386.328	738.773
105,000	447405.2	592.306	840.21258	20.419	450.589	875.082
120,000	570050.8	684.369	972.30959	22.86	525.599	1010.252
135,000	730787.5	782.552	1112.16092	25.862	600.667	1141.344
150,000	902114.2	873.963	1258.79478	29.217	667.449	1293.703

Table.1 Timings for sorting of permuted and sorted word files on InsertionSort (IS), MergeSort (MS) and HeapSort (HS) in milliseconds.

2.

InsertionSort plots behaviour

Permuted Inputs: The Fig.1 shows InsertionSort timings plot on permuted inputs, which is a clear quadratic curve which indicates the time complexity of $O(n^2)$.

Referring to Table.1 timings of Permuted Inputs of InerstionSort, the runtime 15,000 words files is around 8,839.6 milliseconds, increased sharply to nearly 902,114.2 milliseconds for the file with 150,000 words. This drastic increase represents the steep curve expected in quadratic growth. This behaviour is already expected because, for InsertionSort in the worst case scenario every element need to compare and shift almost for every insertion. These steps of operations results in leading to a steep increase in runtime as the input size grows.

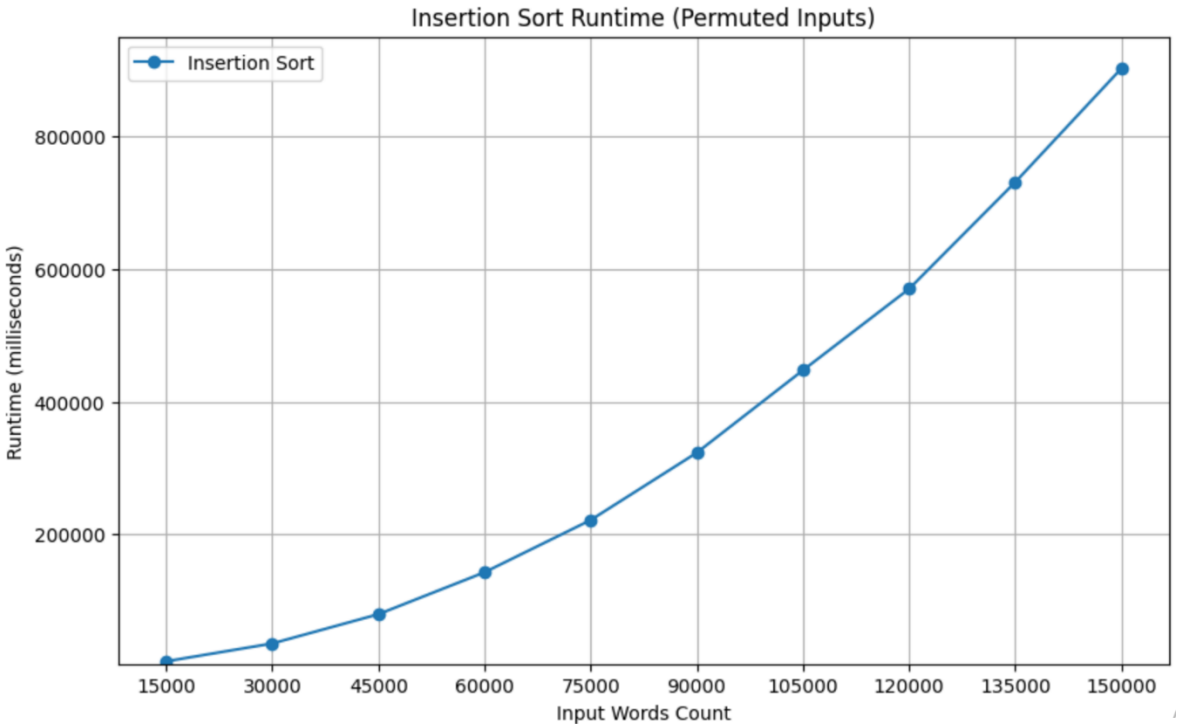

Fig.1 Insertion sort timings plot on permuted inputs

Sorted Inputs: The Fig. 2 shows InsertionSort timings plot on sorted inputs, which a clear cut linear trend curve indicates *O(n)* time complexity in the best case scenario.

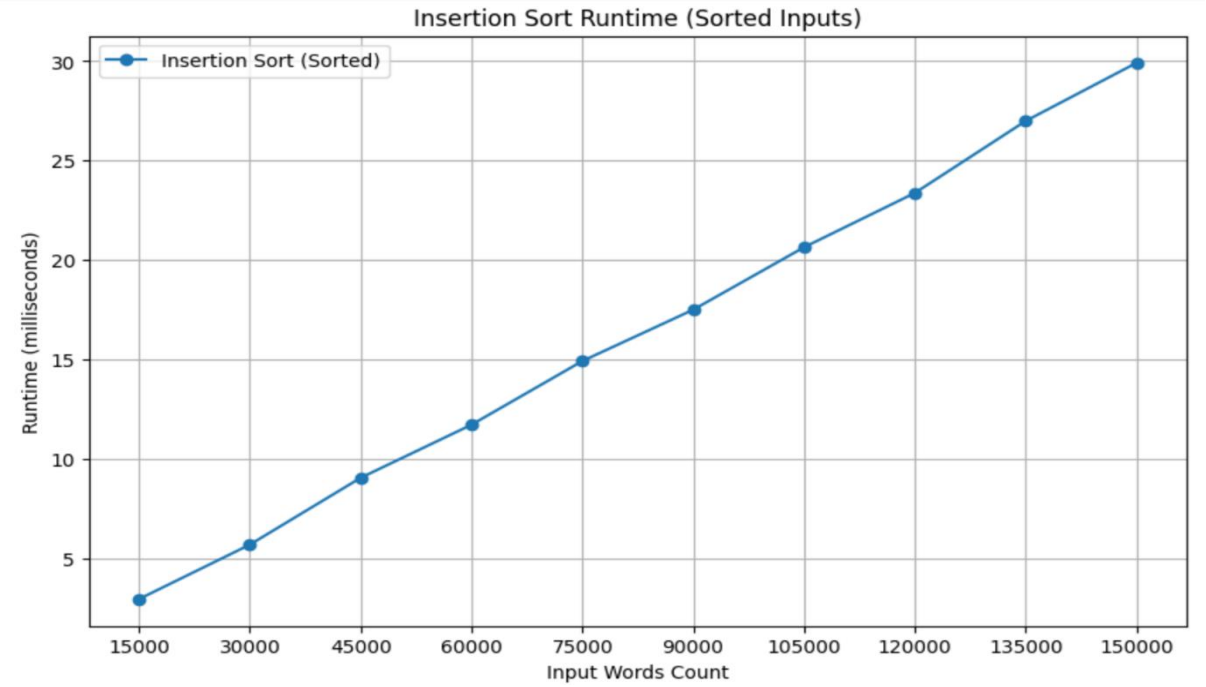

Fig.2 Insertion sort timings plot on sorted inputs

Referring to Table. 1 timings of Sorted Inputs of InsertionSort, the runtime start at around 2.887 milliseconds for 15,000 words and increases steadily to closely 29.217 milliseconds for 150,000 words. The runtime increases progressively and proportional to the input size, which completely aligns with the expected best case performance. When the file is already sorted then the each element is compared only once with the previous element and no shifts are required at all. This results in a direct linear relationship between the input size and the runtime, shows the best case scenario.

MergeSort plots behaviour

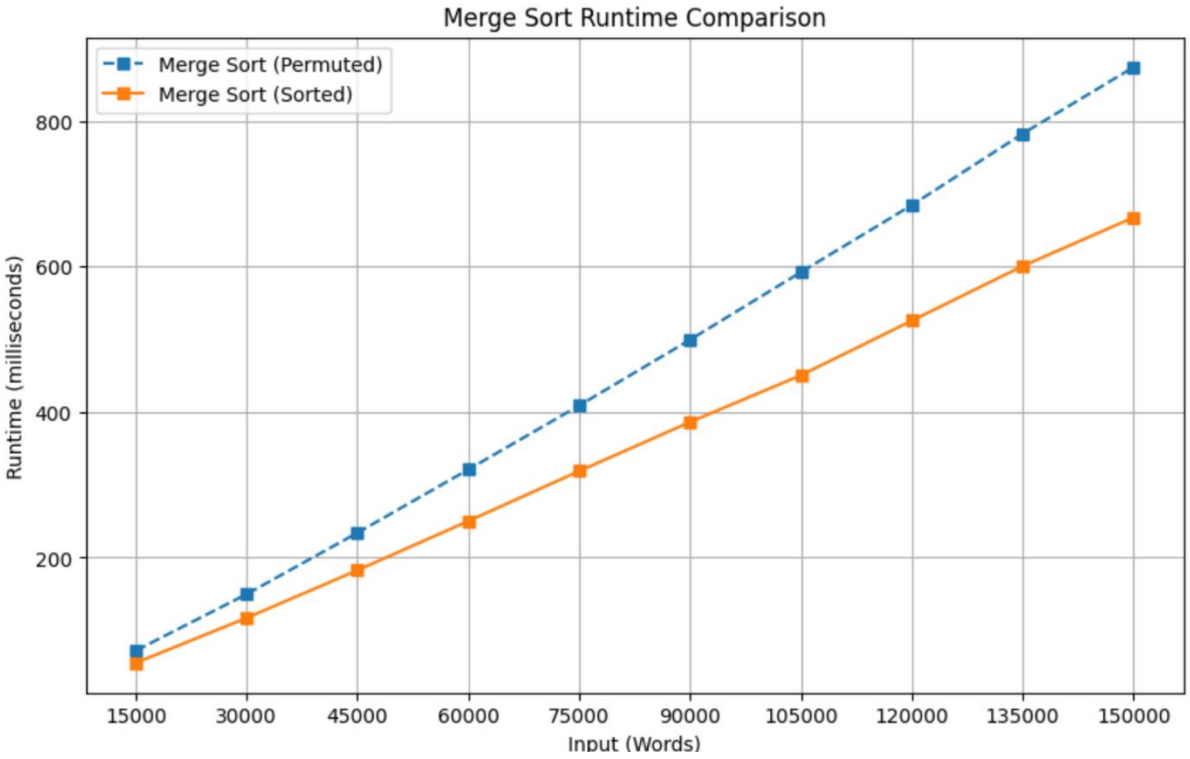

Fig.3 MergeSort timings plot on permuted and sorted inputs

Permuted Inputs: The Fig.3 shows Mergesort timings plot of permuted files shows a linearithmic *O(nlogn)* increase in runtime. The increase in timings are steady which is perfectly aligns with the expected complexity of MergeSort. Referring to Table. 1, timings of Permuted Inputs of MergeSort the runtime for sorting a file with 15,000 words is approximately 72.088 milliseconds, which increases to around 873.963 milliseconds for a file with 150,000 words. This shows that how well the MergeSort algorithm divides and merges the array as the data increases and so the time complexity is *O(nlogn*) regardless of the initial order of the array.

Sorted Inputs: The Fig.3 shows Mergesort timings plot of sorted files shows a linearithmic O(nlogn) increase in runtime. MergeSort on sorted inputs is almost same to that for permuted inputs, which is expected since MergeSort always processes the array in the same way regardless of its initial order. It always divides the array into halves, sorts, and merges them, leading to a time complexity of *O(nlogn).* The *O(nlogn)*

complexity is indicating accurately, showing that the runtime grows in a linearithmic way as the input data size increases. Referring to Table.1, timings of Sorted Inputs of MergeSort, the runtime starts at around 55.233 milliseconds for 15,000 words file and grows to approximately 667.449 milliseconds for 150,000 words file. The runtime increase is consistent with the expected linearithmic behaviour.

HeapSort plots behaviour

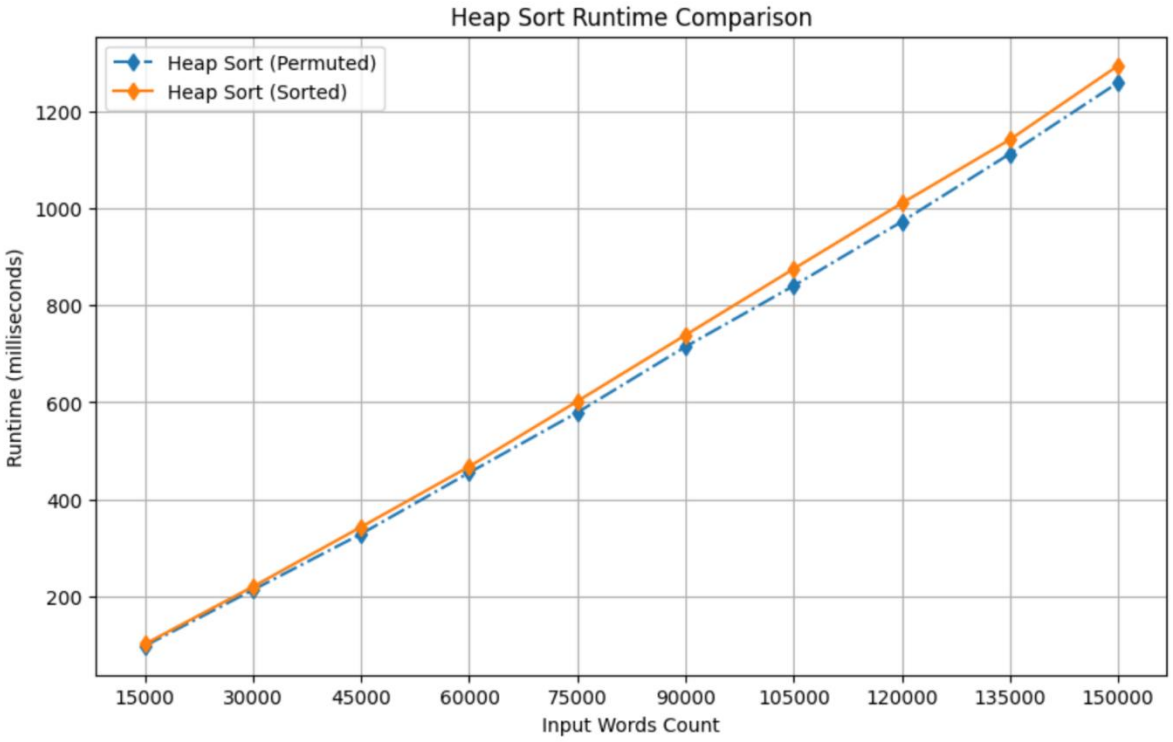

Fig.4 HeapSort timings plot on permuted and sorted inputs

Permuted Inputs: The Fig.4 shows HeapSort timings plot of permuted files shows a linearithmic *O(nlogn)* increase in runtime, confirms the time complexity as explained earlier. The increase in timings is steady and matches with the expected behaviour of HeapSort on unsorted data. Referring to Table.1, timings for Permuted Inputs of HeapSort, the runtime for sorting a file with 15,000 words is approximately 98.482 milliseconds, and it increases to around 1,258.794 milliseconds for a file with 150,000 words. The behaviour of plot shows the how the timings increases consistently as the data increases by building a heap and extracting the elements. These operations results in a overall complexity of *O(nlogn)* despite the order of elements.

Sorted Inputs: The Fig.4 shows HeapSort timings plot of sorted files shows a linearithmic *O(nlogn)* increase in runtime, confirms the time complexity as explained earlier. Referring to Table.1, timings for Sortted Inputs of HeapSort the runtime starts at around 102.149 milliseconds for 15,000 words and grows to approximately 1,293.703 milliseconds for 150,000 words.The performance is almost same as on permuted files, because even though the array is already sorted then also HeapSort build the heap and extracts the elements and continuously repeats the same operations.

The steps it follows are consistent, making the best-case time complexity *O(nlogn)* as well.

In conclusion, the plots and timings results confirm that the implementations and their behaviours are correct and aligns well with the theoretical asymptotic analysis of each algorithm.

3. Comparative Analysis of the Algorithms

a. Which algorithm had the best runtime on Sorted inputs? Why?

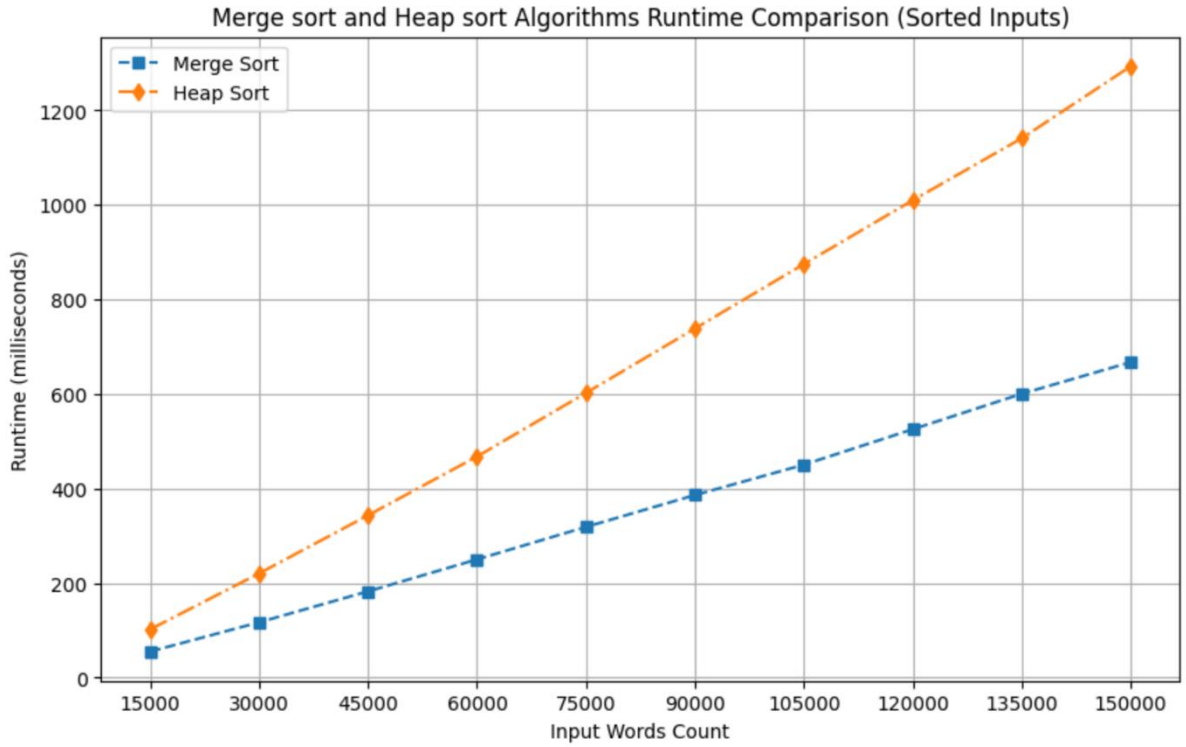

Fig 5. Mergesort and Heapsort timings plot on sorted inputs

Among the InsertionSort, MergeSort and HeapSort algorithms, referring to Fig. 2 and Fig. 5 plots**,** InsertionSort had the best runtime on sorted inputs. The reason for this is that InsertionSort's best case complexity is *O(n).* When the input is already sorted, InsertionSort simply traverses the array confirming the correct order of elements and now shifts are needed. On the other hand, even when the data is already sorted, HeapSort still need to build the heap and performs all the required heap operations. This consists of number of comparisons and swaps to maintain the heap property, resulting in *O(nlogn)* time complexity. Similarly, MergeSort's time complexity remains *O(nlogn)* regardless of the beginning order. MergeSort divides the array into subarrays and merges them back together, which involves high number of comparisons and shifts.

These are the main reasons for InsertionSort gives the best performance on Sorted Inputs.

b. Which algorithm had the best runtime on Permuted inputs? Why?

Among the three algorithms, referring to Fig. 1 and Fig. 6 plots, MergeSort has the best runtime on permuted inputs.

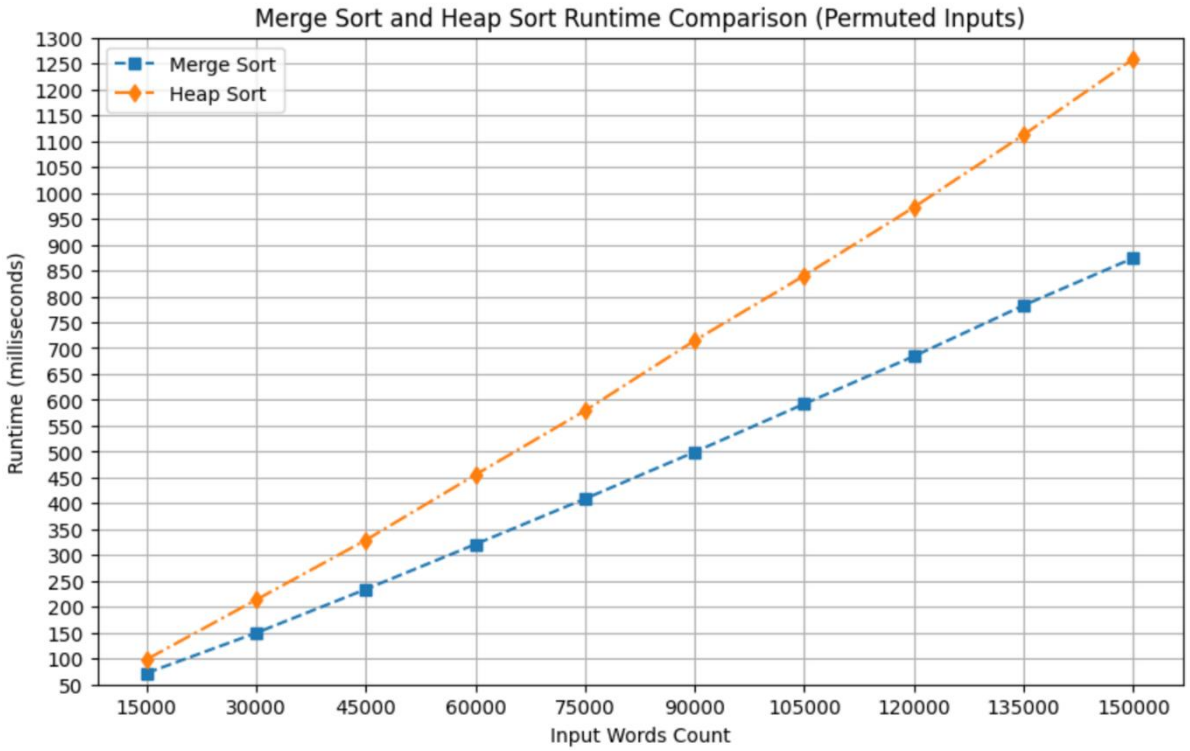

Fig.6 MergeSort and HeapSort timings plot on permuted inputs

Mergesort steadily performs with a complexity of *O(nlogn)* regardless of beginning arrangement of elements. MergeSort utilizes divide and conquer technique, which splits the arrays into subarrays till the array size reaches to single element and merges them back, this process is highly efficient and predictable and handles the datasets effectively. Meanwhile, HeapSort has a more complicated process to build the heap, which adds extra work and slows it down.

MergeSort might have better performance because it may reads memory in a straight line. It may processes large pieces of data at once, which uses the memory more efficiently. Meanwhile, HeapSort moves around memory a lot, especially when it is building the heap, which can slow the operations. On the other hand, InsertionSort takes longer because it handles data one element at a time. When sorting, it compares each item with all the previous items and moves it to the right place. If the data is not already mostly sorted, this can mean a lot of comparisons and movements for each item, especially with larger datasets.

MergeSort also does fewer comparisons and swaps than HeapSort. When merging, it simply checks through the data and combines them, making the process more smoother and faster. Whereas HeapSort needs to do many comparisons and swaps to keep the heap structure, which takes more time.

These are the main reasons why MergeSort usually runs faster than HeapSort and Insertion sort. The graph clearly show that MergeSort outperforms both InsertionSort and HeapSort on permuted inputs due to its stable and optimal time complexity.

c. Which algorithm had the worst runtime on Permuted inputs? Why?

Referring to Fig.1 and Fig, 6 plots, InsertionSort had the worst runtime on permuted inputs. The reason is that InsertionSort's worst-case complexity is *O(n^2).* When the input data is unsorted then InsertionSort has to perform higher number of comparisons and need to shifts (performs in reverse order too). This results in significantly much higher time than MergeSort and HeapSort.

d. Which algorithm had almost the same runtime for its Sorted and Permuted input files of the same size? Why?

Referring to Fig. 4 plot, HeapSort had almost the same runtime for its sorted and permuted input files of the same size. The main reason for this is HeapSort is influenced by the order of elements in the input file. Whether the input data is sorted or not still HeapSort algorithm constructs a heap and then performs heapification and results in a runtime of O(nlogn) for Permuted and Sorted input files.

e. If you're not seeing expected behavior for (a) through (d) above, you can also discuss programming language and implementation-related details that you might find explanatory.

The observed behaviours for each sorting algorithm timings and plots are same as with theoretical expectations.