Program One(CSC505_P1)

Implementation and Comparison of Insertion, Quick and Merge sort through Linked List

Team Theta

Justin Kirschner(200365781)
jkirsch@ncsu.edu

Jonathan Nguyen(200003034)
jhnguye4@ncsu.edu

Aakriti Aakriti(200388633)
aaakrit@ncsu.edu

Naga Suryadevara(200325299)
nsuryad2@ncsu.edu

Implementation Description :

Insertion sort, Merge sort and Quick sort are the three sorting algorithms we implemented here on linked lists(Singly linked lists) in Java. We compared their performance of sorting on random number sequences of distinct numbers, generated using the program random_permutation.py (provided). This generator provides us with sequences in blocks of equal size for accurate comparison. Each of the three sorting algorithms and their implementation is discussed here in detail below:

Insertion sort implementation:

From its name, Insertion sort, in-place sorting algorithm inserts an element by comparing it to all the remaining elements in the array/list.This insertion is based on the three conditions, whether the new element in the unsorted list is less than, greater than or equal to the previous element.

In our implementation, we consider the first two nodes adjacent to each other and compare the values in them. If the value in the first node is greater than the second value, then the value of two nodes swap. If the first value is smaller than the second node, then the order remains the same. And the pointers are moved to the next two nodes. Now these two nodes are compared and swapped. This process is done repeatedly until we reach the end of the sequence. By now, we traversed through the input sequence and performed one set of swaps on the sequence. And, we repeat this process of comparison and swapping, iteratively until we obtain the sequence, which is sorted. Finally, we have our Insertion sort with time complexity O(n^2) and the space complexity is O(1).

Merge Sort Implementation:

Merge sort uses divide and conquer approach i.e. the list is iteratively split into two halves until it has only one element and merged back so as to form a new sorted list.

In our implementation, the input linked list with the sequence of numbers generated randomly. This unsorted list is given as an input to the Merge Sort algorithm, which iteratively breaks down the linked list into two equal parts. For the case in which the number of nodes is even, we divide the list into two halves. In the case of an odd number in the list, we find the node of the middle element and break the link after this middle number. This process is recursively called until we get a single node, that is until the list can no longer be broken down. Now, the value in each node is compared and linked to each other in the order such that the values in the nodes are sorted. The merging process continues until the entire list is sorted. Here, instead of copying the linked list into our memory, we use in-place merge sort with the help of pointers. In this implementation for merge sort, we expect the time complexity O(nlogn) and the space complexity to be O(n).

Quick-Sort implementation: Quick-sort, similar to Merge sort uses Divide and Conquer approach but with a variation. In Quick-sort, a pivot is used to partition the list so that the elements before the pivot are less than or equal to the pivot while the elements on the right half are greater than the pivot. There are many ways to choose pivot. We can either choose the start node, the end node or any random node of the input sequence as pivot for each iteration. In our implementation, we used the first element as the pivot.

After choosing the first node has pivot, we consider two pointers pointing to the next two nodes. To understand the process, let us consider the node immediately to the pivot as first and the node next to it as next. The first traverses through the sequence and links all the nodes for the values that are smaller than the pivot to one side of the pivot. The next pointer links all the nodes which have values that are larger than the pivot to one side.

Initially, we compare the value in the node ’first’ points, to the pivot value. If this value is lower than the pivot, then, the first pointer remains at this node. Then, the value in the next node is compared to the pivot. If this value is smaller than the pivot then the first pointer shifts to this node. If the value is larger, then the node next points to the element which lies next to this node. This process continues till we reach the end of the sequence. We then select the first element as pivot and call this process recursively.

Once we reach the end of the sequence, the pointers pivot and the small gets swapped. This leaves us with a sequence, where we are left with a sequence partitioned such that all the elements which are smaller than the pivot stays on the one side of the pivot and the elements which are larger than the pivot stays on the other side.

We repeat this process until the entire sequence is sorted. We expect the time complexity of this implementation to be O(nlogn) and the space complexity O(1) that is to perform it in place so that it doesn't require additional space.

Experimental Design

The following hypotheses were made for the given program

Hypothesis 1: For insertion sort, Number of comparisons grows quadratically with input size ~ N^2/4.

Experiment:A table is made to compare the observed number of comparisons with Predicted comparison as per hypothesis. As a result, the number of comparisons are matching closely.
7 test inputs were considered for this hypothesis and listed below in the table.

Conclusion: Prediction matches observation hence, agrees with the hypothesis!

Table for observations:

N(Input size)	Observed Comparisons	Predicted Comparisons
10	22	25
100	2,645	2500
1000	251,342	250 thousand
10000	25.129364 million	25 million
100000	2.500325388 billion	2.5 billion
500000	62.573548907 billion	62.5 billion

Hypothesis 2: The variation in the number of comparisons and runtime for the same length input file generated multiple times with random inputs.

Experiment: Generated 4 different files through random_permutation.py, resulting in random permutation of inputs.Ran the algorithm(Insertion sort) for the number of comparisons and

Runtime for the same length file but different inputs to check the variations .The first input in the table is used as base input and 3 more files with same input size is generated to validate the number of comparisons

fixed input size N=1000000

Conclusion: prediction matches observation hence, agrees with the hypothesis

N(Input size)	Observed Comparisons()	Runtime
1000000	249.923758763 billion	98923.1708678
1000000	250.076858922 billion	100060.42346532
1000000	250.345164285 billion	100296.59455266
1000000	249.816082382 billion	100171.23682878

Hypothesis 3: To check effectiveness of insertion sort for small input sets.

3.a The algorithm is excellent for small sequences of elements as compared to other two algorithms namely Merge sort and Quicksort.

Experiment: Input Files with size 14, 4 and 10 were created to check the effectiveness of insertion sort for small set of elements

Conclusion: Prediction matches observation hence, agrees with the hypothesis!

Table for comparison

Description	**Insertion**	**Quick**	**Merge**
Inputs: 14 Range: 0-1 Duplicates: Yes	13	56	49
Inputs: 4 Range: -5-8 Duplicates: No	6	12	11

Inputs: 10 Range: 1-10 Duplicates: No	22	35	32

3.b insertion-sort is quite effective for sorting sequences that are already "almost" sorted.

Experiment: Input file of size 8 is created where a single element is considered for input size=8 which is an already sorted case. Insertion sort took 0 comparison whereas Quick and Merge sort took certain comparison as mentioned below in table.

Conclusion: prediction matches observation hence, agrees with the hypothesis!

Table for comparison:

Description	**Insertion**	**Quick**	**Merge**
Inputs: 8 Range: 4 Duplicates: Yes	0	28	19

Hypothesis 4: Compare how the running time increases with the input size.Insertion sort has the largest runtime in all the three algorithms.

Experiment: Runtime of all 3 algorithms is calculated with different input files and observed, except for input size 10(which is the smallest input set and insertion sort works excellent for small sets(proved in hypothesis 3)), the insertion sort has the largest runtime compared to other two algorithms.
Least runtime
Most runtime

Conclusion: prediction matches observation hence, agrees with the hypothesis!

Table for observation

N(Input size)	Isort Observed Runtime	Msort Observed Runtime	Qsort Observed Runtime
10	1.0658E-4	1.5733E-4	1.256E-4
100	0.00244306	7.3295E-4	8.5819E-4
1000	0.08244335	0.01223925	0.01024177
10000	1.26937308	0.03892896	0.03183908
100000	69.58663046	0.29638668	0.16230185
500000	1628.04685325	2.09458001	0.57458916

1000000	39486.63958579	2.02501695	1.549790443

Experimental Results

Sorting Times (1/10th of Second)

N(Input size)	Insertion Sort	Quicksort	Merge Sort
10	1.0658E-4	1.256E-4	1.5733E-4
100	0.00244306	8.5819E-4	7.3295E-4
1000	0.08244335	0.01024177	0.01223925
10000	1.26937308	0.03183908	0.03892896
100000	69.58663046	0.16230185	0.29638668
500000	1628.04685325	0.57458916	2.09458001
1000000	39486.63958579	1.549790443	2.02501695

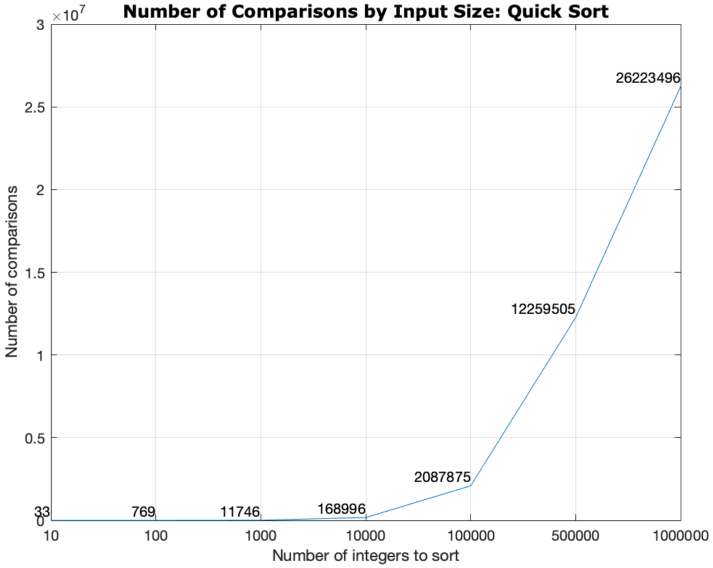
Number of Comparisons

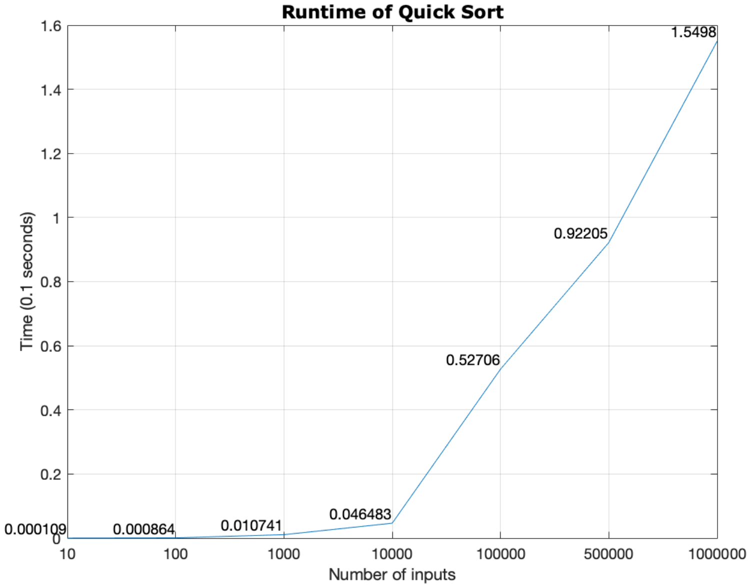
Input File	Insertion Sort	Quicksort	Merge Sort
10	22	35	32
100	2,645	742	637
1000	251,342	11,530	9,719
10000	25,129,364	172,228	130,445
100000	2,500,325,388	2,098,439	1,636,223
500000	62,573,548,907	12,583,672	9,337,018
1000000	499,749,801,846	24,484,677	14,930,879

The figures and tables below are the average over 3 runs using the py script (so, they are the average of 3 runs where each input is completely random). This will help us analyze the average case runtime. Note that the x-axis is not to scale (to make it more readable, otherwise the 500K and 1M data points would consume almost all of the x-axis).

Quicksort

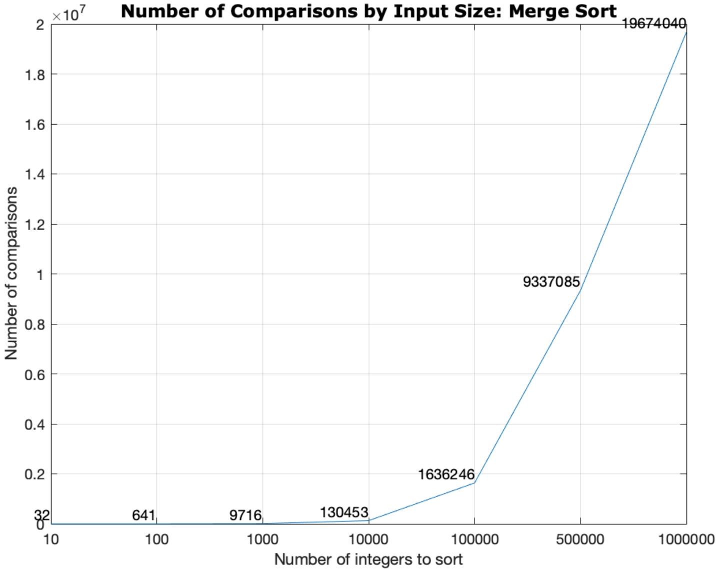
N	Runtime (1/10th s)	Comparisons
10	0.000109323333	33
100	0.000863993333	769
1,000	0.01074063667	11,746
10,000	0.04648298	168,995
100,000	0.5270603767	2,087,875
500,000	0.9220549433	12,259,505
1,000,000	1.549790443	26,223,496

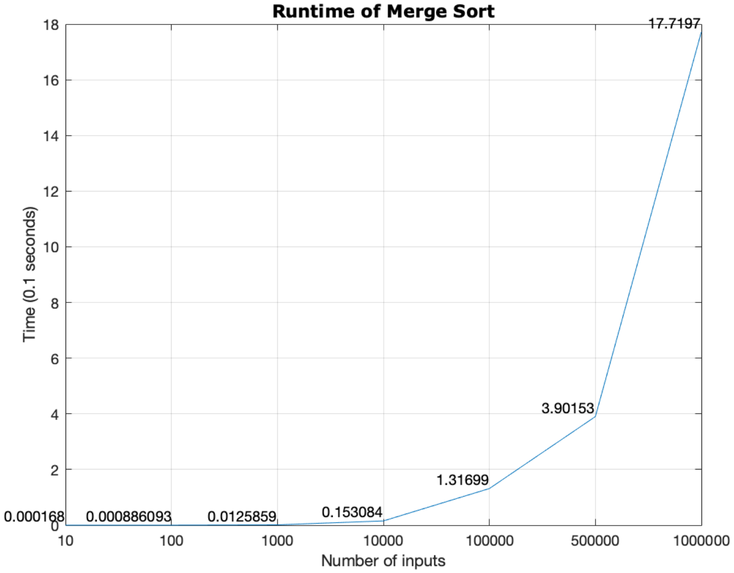




Merge Sort

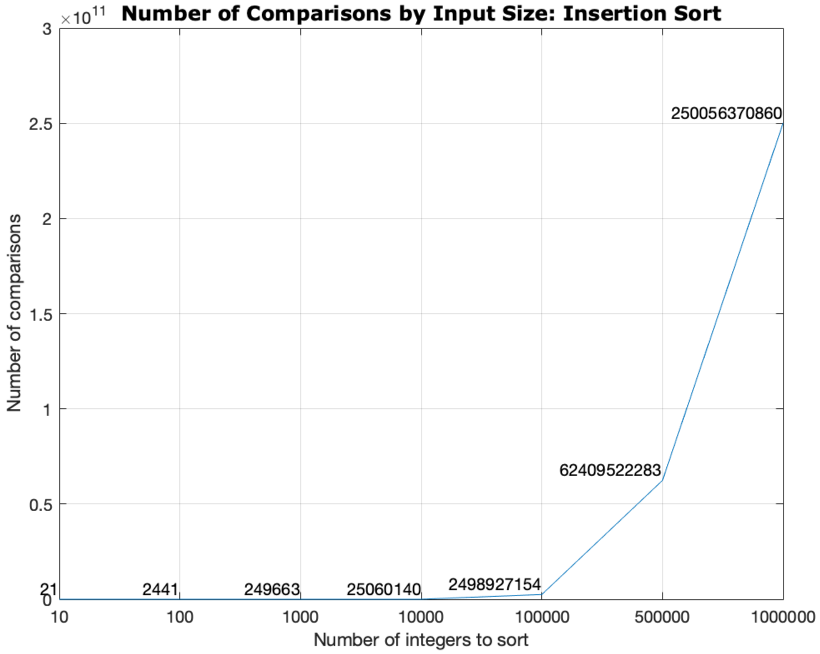
N	Runtime (1/10th s)	Comparisons
10	0.000168076666	33
100	0.000886093333	641
1,000	0.01258585333	9,716
10,000	0.1530838	130,453
100,000	1.316987103	1,636,246
500,000	3.901529413	9,337,085
1,000,000	17.71972116	19,674,040

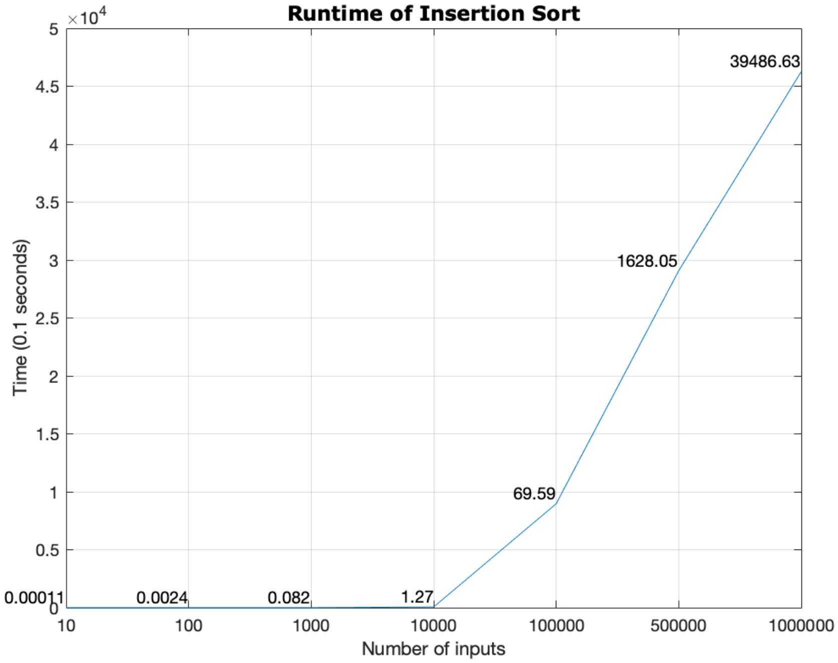




Insertion sort

N	Runtime (1/10th s)	Comparisons
10	0.0001655	21
100	0.00579997	2,441
1,000	0.7686512567	249,663
10,000	63.4654661	25,060,140
100,000	8969.728354	2,498,927,154
500,000	29081.47821	62,409,522,282
1,000,000	46262.32366	250,056,370,859





Analysis

As expected, we see that insertion sort does better on our small lists (on our input files that contain 1 - 14 elements), and that insertion sort performs much more poorly than quicksort and merge sort when the lists become substantial. Both of these observations come from the fact that insertion sort is a quadratic sorting algorithm (in the average case, in addition to the worst cast). Even with the list of only 100 elements, insertion sort has ~4x the amount of comparisons as the other two algorithms. From 100 elements to 1 million elements, the number of comparisons under insertion sort is at least one order of magnitude larger than for the other two algorithms, often multiple orders of magnitude larger. In other words, the number of comparisons is indeed growing quadratically.

This holds true for the runtime on these inputs, too, and the effect is much more pronounced in the runtime results than the comparison results. In terms of runtime, insertion sort almost always performed the worst (except on the smallest lists), and the runtime from our experiments did grow at a quadratic rate. Overall insertion sort compares poorly to the other algorithms on any meaningful list. For smaller lists, non-recursive sorting algorithms could be faster than insertion sort, anyway. So, insertion sort is not that useful in most cases. However, it is especially useful when lists have little variation or are close to sorted, and we saw this manifest in our test files/hypotheses.

We know that quicksort sort has an average case of O(nlgn) and a quadratic worst case. After averaging a few runtimes of quicksort, it is apparent that our implementation is exhibiting preliminarily O(nlgn) time complexity. It is also exhibiting linear growth regarding the number of comparisons. However, with respect to the larger set of inputs (500,000 or 1,000,000), it appears the algorithm *could* be running in O(n) time. In order to confirm the O(nlgn) runtime, one would need to average over a few dozen more runs, or create longer lists to further extrapolate the data and see if convergence does indeed occur. It is also possible that our averaging is experiencing growth higher than O(nlgn) due to bad luck with pivot selection. Our pivots could have failed to cause a 25%-75% or closer (e.g. 50%-50%) split in the partitions.

We know that merge sort has an average case and worst case of O(nlgn). Our results show this to be the case. Unlike quicksort, where randomness could affect the results, there is not much room for runtime variation in merge sort if it is implemented correctly. Similarly to quicksort, runtime growth with the largest input sizes isn't as convincing as it could be. More averaging or longer inputs are needed to confirm the O(nlgn) time complexity. Also similarly to quicksort, merge sort's number of comparisons grew linearly. This is why, given that insertion sort grows much faster as lists get longer, insertion sort originally has a lower number of comparisons that quickly blows up with input size. Also, given the definition of mergesort and quicksort, one would expect far fewer comparisons with mergesort, which our data shows.

Conclusion and Future Work

Overall, our experiments did not result in outcomes drastically unexpected. Almost all of the runtime behavior of the algorithms reflects their theoretical analyses. However, as mentioned above, more experiments or longer experiments (i.e. with larger input sizes) are needed to confirm the correct convergence behavior.

Additional experiments could analyze memory (space complexity), the other main factor in analyzing these algorithms aside from runtime (time complexity). Merge sort does not necessarily sort in place (depending on the implementation), so it could require double the amount of memory as the input size needs (to store the output with the same size as the input, in the scenario where it is not done in place). Likewise, the space complexity of *quicksort* also depends on its implementation.

Further work could investigate other divide-and-conquer algorithms such as heap sort, or non-comparison based sorting algorithms like bucket sort and radix sort. We could also investigate non-recursive algorithm such as insertion sort. Further work could also use different implementations of the algorithms to compare their space complexity across and within the algorithm types. For example, when quicksort is done in-place, we would only expect O(logn) complexity. However, if it is not done in-place, it is possible for quicksort to be stable, but it would need O(n) space.

This is worth pursuing because these algorithms might need to be used in machines with little memory such as embedded systems and/or used in circumstances needing stability.