1. Merge Sort

```
l3 = [0, 0, 0, 0, 0, 0, 0, 0]
l1 = [1, 16, 25, 31]
l2 = [-3, 0, 16, 27]
indexl1 = 0
indexl2 = 0;
i = 0;
Pass 1: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], false
l3[i] = l2[indexl2]
indexl2++;
i++;
//l3 = [-3, 0, 0, 0, 0, 0, 0, 0]

Pass 2: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], false
l3[i] = l2[indexl2]
indexl2++;
i++;
//l3 = [-3, 0, 0, 0, 0, 0, 0, 0]


Pass 3: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], true
l3[i] = l1[indexl1]
indexl1++;
i++;
//l3 = [-3, 0, 1, 0, 0, 0, 0, 0]


Pass 4: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], false
l3[i] = l2[indexl2]
indexl2++;
i++;
//l3 = [-3, 0, 1, 16, 0, 0, 0, 0]

Pass 5: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], true
l3[i] = l1[indexl1]
indexl1++;
i++;
```

//l3 = [-3, 0, 1, 16, 16, 0, 0, 0]

Pass 6: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], true
l3[i] = l1[indexl1]
indexl1++;
i++;
//l3 = [-3, 0, 1, 16, 16, 25, 0, 0]

Pass 7: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], false
l3[i] = l2[indexl2]
indexl2++;
i++;
//l3 = [-3, 0, 1, 16, 16, 25, 27, 0]

Pass 8: [1, 16, 25, 31] [-3, 0, 16, 27]
l1[indexl1] < l2[indexl2], true
l3[i] = l1[indexl1]
indexl1++;
i++;
//l3 = [-3, 0, 1, 16, 16, 25, 27, 31]

The merged Array is [-3, 0, 1, 16, 16, 25, 27, 31]

2. Insertion Sort

Array a = [-1, -5, 67, -10, 21, 8, 4, 1]

Pass 1
Sorted Elements [-1]
-5 < -1
Array a = [-5, -1, 67, -10, 21, 8, 4, 1]

Pass 2
Sorted Elements [-5, -1]
67 > -1
Array a = [-5, -1, 67, -10, 21, 8, 4, 1]

Pass 3
Sorted Elements [-5, -1, 67]
-10 < -5
Array a = [-10, -5, -1, 67, 21, 8, 4, 1]

Pass 4
Sorted Elements [-10, -5, -1, 67]
21 < 67
Array a = [-10, -5, -1, 21, 67, 8, 4, 1]

Pass 5
Sorted Elements [-10, -5, -1, 21, 67]
8 < 21
Array a = [-10, -5, -1, 8, 21, 67, 4, 1]

Pass 6
Sorted Elements [-10, -5, -1, 8, 21, 67]
4 < 8
Array a = [-10, -5, -1, 4, 8, 21, 67, 1]

Pass 7
Sorted Elements [-10, -5, -1, 4, 8, 21, 67]
1 < 4
Array a = [-10, -5, -1, 1, 4, 8, 21, 67]

3. Quick Sort

Pivot is the median of three, so the median of the low element, middle element, and last element.
The middle element is input[input.length/2 - 1]
The top element is input[input.length - 1]
The low element is input[0]
Array input = [-5, 42, 6,19, 11, 25, 26, -3]

Pass 1
Array input = [-5, 42, 6,19, 11, 25, 26, -3]
Int low = -5
Int mid = 19
Int high = -3
Int pivot = -3
Array partition1 = [-5]
Array partition2 = [42, 6,19, 11, 25, 26]

Partition1 recursive call where input is parrition1
Array input = [-5]

Base case is reached, so the program goes up the call stack

Partition2 recursive call
Array input = [42, 6,19, 11, 25, 26]
Int low = 42
Int mid = 19
Int high = 26
Int pivot = 26
Array partition1 = [6, 19, 11, 25]
Array partition 2 = [42, 26]

Partition1 recursive call
Array input = [6, 19, 11, 25]
Int low = 6
Int mid = 19
Int high = 25
Int pivot = 19
Array partition1 = [6, 11]
Array partition 2 = [25]

Partition1 recursive call
Array input = [6, 11]
Int low = 6
Int mid = 11
Int high = 11
Int pivot = 11
Array partition1 = [6]
Array partition 2 = []

Partition1 recursive call
Array input = [6]
Base case is reached, so the program goes up the call stack

Partition2 recursive call
Array input = []
Base case is reached, so the program goes up the call stack

Partition2 recursive call
Array input = [25]
Base case is reached, so the program goes up the call stack

Partition2 recursive call
Array input = [42, 26]
Int low = 42
Int mid = 26
Int high = 26
Int pivot = 26
Array partition1 = []
Array partition 2 = [42]

Partition1 recursive call
Array input = []
Base case is reached, so the program goes up the call stack

Partition2 recursive call
Array input = [42]
Base case is reached, so the program goes up the call stack

Now concatenating the arrays and pivots
Array input = [-5, -3, 6, 11, 19, 25, 26, 42]

4. Shell Sort

Array input = [15, 14, -6, 10, 1, 15, -6, 0]
Int n = input.length / 2

Pass1
n = 4
[15, 14, -6, 10, 1, 15, -6, 0]
15 > 1, so a swap is made
[1, 14, -6, 10, 15, 15, -6, 0]
14 < 15, so no swap is made
[1, 14, -6, 10, 15, 15, -6, 0]
-6 == -6, so no swap is made
[1, 14, -6, 10, 15, 15, -6, 0]
0 < 10, so a swap is made

List after Pass1
[1, 14, -6, 10, 15, 15, -6, 0]

Pass 2
n = 2
[1, 14, -6, 10, 15, 15, -6, 0]
-6 < 1, so a swap is made
15 > -6 so no swap is made

-6 < 15 so a swap is made
[-6, 14, -6, 10, 1, 15, 15, 0]
10 < 14, so a swap is made
15 > 14, so no swap is made
0 < 10 so a swap is made

List after pass1
[-6, 0, -6, 10, 1, 14, 15, 15]

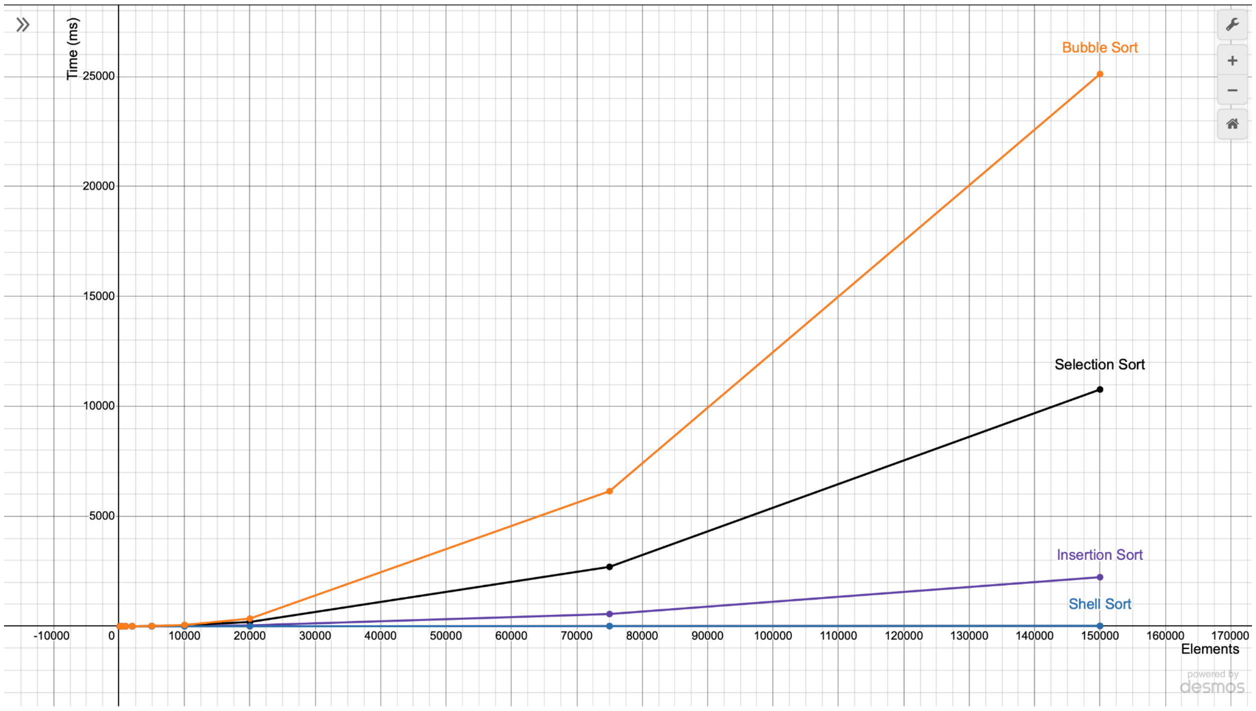
Pass3
n = 1
[-6, 0, -6, 10, 1, 14, 15, 15]
-6 < 0, so no swap is made
[-6, 0, -6, 10, 1, 14, 15, 15]
-6 < 0 so -6 is inserted before 0
[-6, -6, 0, 10, 1, 14, 15, 15]
10 > 0, so no insertion is made
[-6, -6, 0, 10, 1, 14, 15, 15]
1 < 10, so 1 is inserted before 10
[-6, -6, 0, 1, 10, 14, 15, 15]
14 > 10, so no insertion is made
[-6, -6, 0, 1, 10, 14, 15, 15]
15 > 14, so no insertion is made
[-6, -6, 0, 1, 10, 14, 15, 15]
15 == 15, so no insertion is made

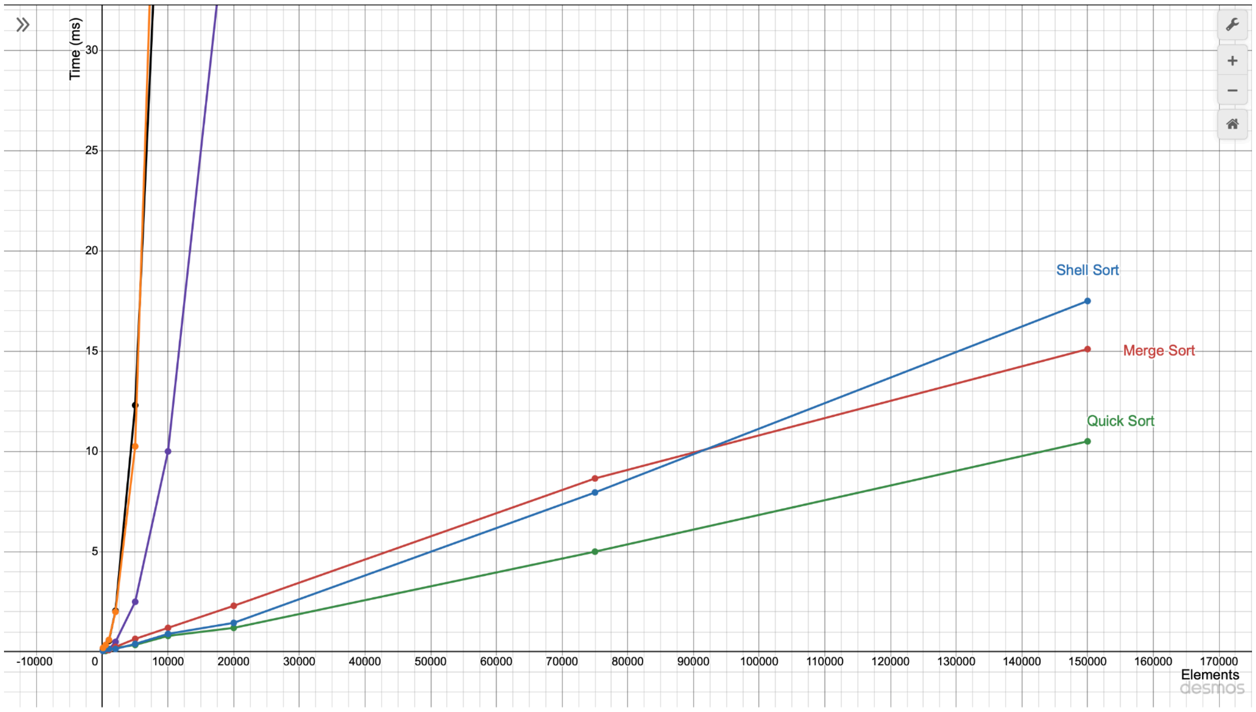
Final sorted list: [-6, -6, 0, 1, 10, 14, 15, 15]

5. Predictions of runtime

Based on the worst case time complexity of the algorithms I am going to predict the speeds of the algorithms from fastest to slowest will be merge sort, quick sort, shell sort, insertion sort, selection sort, then bubble sort. I think merge sort will be the fastest because it has the best possible worst case time complexity of O(nlogn) compared to all the other algorithms which have worst case time complexity of O(n^2). I think quick sort will be the next fastest because worst case time complexity comes from calling the algorithm on an already sorted list. Since I know the list is not already sorted and the quick sort algorithm uses a median of three as the pivot element, quick sort will be closer to the average time complexity of nlogn. Shell sort I predict to be the third fastest because shell sort can be thought of as a version of insertion sort that leads to a better average time complexity. Then I predict insertion sort as the fourth fastest because when an element is being inserted into the sorted position of the list the insertion does not need to loop through every element while the other two algorithms selection and bubble sort do. The fifth fastest algorithm I predict to be selection sort because selection sort swaps the elements after the nested loop while bubble sort makes the swaps in the nested loops, so even though the constant before O(n^2) is chopped off bubble sort's constant would be larger than selection sort's constant since the nested loop has more atomic operations.

9. Plot(Next Page)





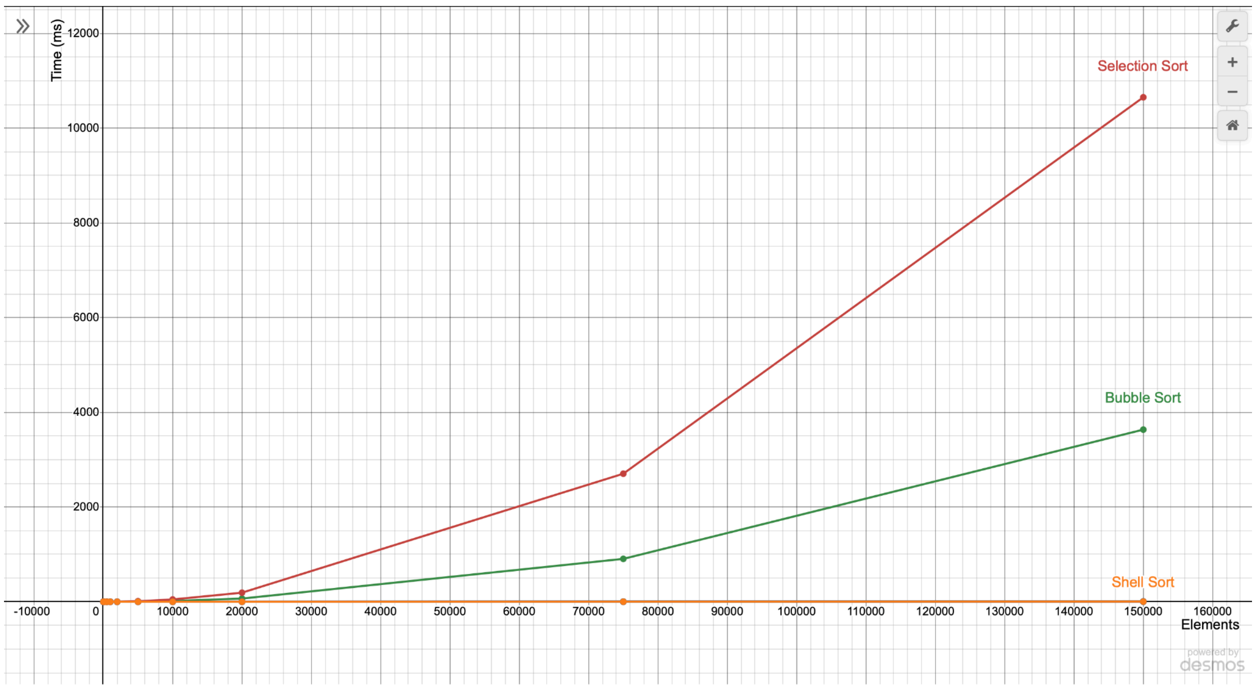
The issue that happens as the number of elements gets bigger is that the bubble sort, selection sort, and insertion sort algorithms are growing in quadratic time, so time is significantly longer than the other algorithms. This leads to the plot only showing shell sort, bubble sort, selection sort, and insertion sort because merge sort and quicksort appear under shell sort, so I have changed the bounds of the plot and provided two images one which shows shell sort, bubble sort, selection sort, and insertion sort and another that shows shell sort, merge sort, and quick sort.

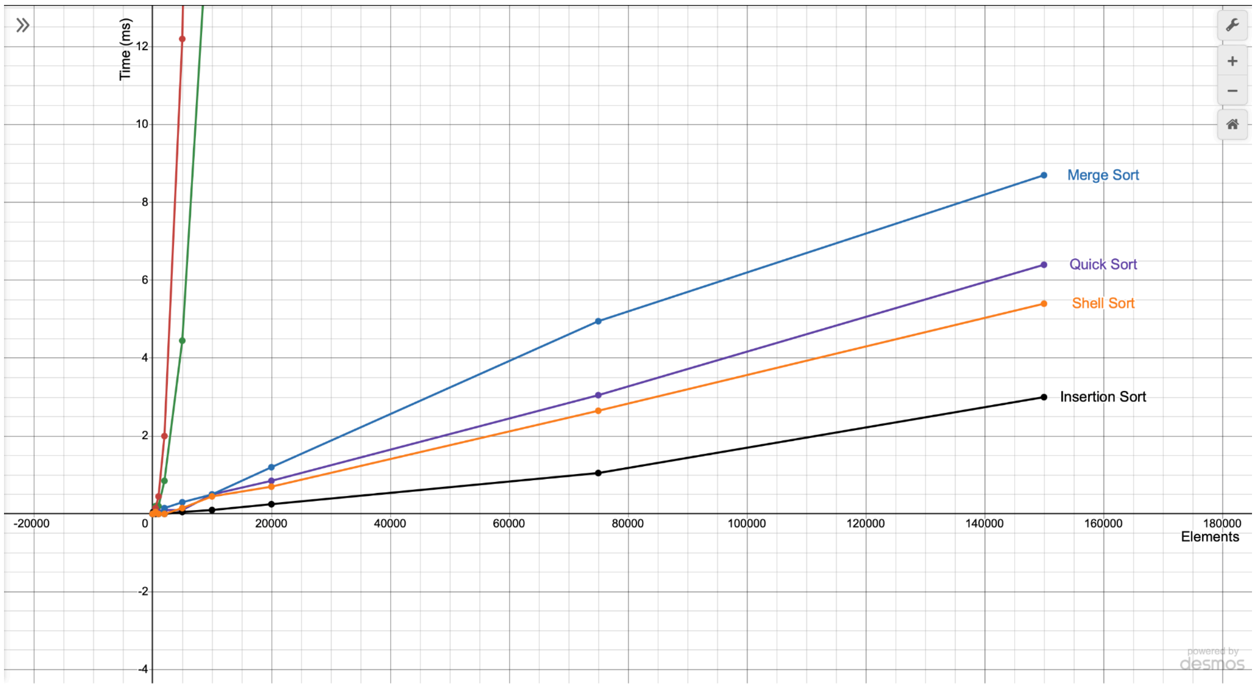
10. Analysis Of Random Data

The one difference between my prediction and the results was quicksort being faster than merge sort; however, merge sort is growing less rapidly as the number of elements increases while quicksort's time is increasing more linearly, so as the number of elements continues to increase past 150,000, I would predict merge sort to be faster on average. Shell sort and quicksort do not match the asymptotic complexities of the worst case. They both appear to be growing more linearly than quadratically. Bubble sort, selection sort, and insertion sort definitely match the predictions I made based on asymptotic complexity. All of those algorithms appear to be growing quadratically, and bubble sort which I predicted to be the slowest due to the constant for n^2 that is chopped off was the slowest.

CONT. On Next Page

12.





A. Plot issue

The plot encounters the same issue that the plot of random data encountered where certain algorithms average time is significantly longer. However, only bubble sort and selection sort average time for larger elements is significantly longer than the other 4 elements for this plot, so under shell sort is the plot for insertion sort, quick sort, and merge sort.

B. Analysis

The algorithms had significantly different rankings for k-sort data. Bubble Sort was no longer the slowest and insertion sort and shell sort were the two fastest. Bubble sort's time decreases because elements being 10-sorted means that for one element bubble sort will only need to make a maximum of ten swaps for the element to be put in its final position. This prevents the worst case scenario of bubble sort where the list is reverse sorted and the maximum number of swaps have to be made. Next shell sort and insertion sort got faster for the same reason which is shell sort and insertion sort's best possible scenario is a partially sorted list. Shell sort is faster than insertion sort though because shell sort still does the extra work to partially sort the list which takes time.