Mahamed Ahmed

Sorting Algorithm

Introduction:

It is our human nature to think and picture things in a sequential order. For example when stacking boxes, we would want to arrange the boxes in such a way that the larger boxes go on the bottom and the smaller boxes go on top so that they won't fall. Also when taking a picture in large groups, we rearrange ourselves in such a way that the shorter people line up in front while the taller people line up in the back. This allows everyone to be seen in the picture. We always try to find ways to sort things in a fashionable way so that it makes our lives easier. One of the possibilities in computer science is to develop algorithms that help solve complex problems that are difficult for humans to solve by hand. Sorting algorithms work in such a way that they compare, swipe, and place elements in the order that we want. The goal for this paper is to determine which sorting algorithm will be efficient and which ones will be inefficient in a given array with three different order of implementation. We will compare the running times for all the sorting algorithms to determine which one would be best used in certain situations.

Methodology

In this paper I will be analyzing the time complexity and the performance of six well known sorting algorithms which include: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Heapsort, and Quicksort. I will be using the java version 7 language on the IDE eclipse to program these algorithms. In addition, I will be testing the sorts by using the data structure arrays in the data type Int. The sizes of the arrays will range from 100, 1000, 10000, 100,000 and 1,000,000. The arrays are ordered from to low to high, high to low, and randomly distributed. The method from the system class called nanoTime will be used to keep track of the runtimes for each sorting algorithm in nanoseconds. After testing each sorting algorithm, the result will be saved in an excel file which will be used to graph different run times. In each array, there will not be any duplication, the same array will be used for each sorting algorithm to ensure accurate measurement of the run times.

BubbleSort

8	1	3	9	11
1	8	3	9	11
1	3	8	9	11
1	3	8	9	11

Figure 1: An example of how the bubble sort works

The Bubble Sort algorithm works by sorting elements in the array in increasing order. This algorithm repeatedly compares adjacent elements. It sorts from low to high searching the smallest element in the array. The first smallest element it finds goes to the first position of the array. After the first iteration, it searches for the second smallest element in the array and it puts it in the second position of the array. It does this until all elements are sorted. This is a well known algorithm and performs its best when sorting small numbers of elements and it is inefficient when it comes to a large list of elements and is considered one of the slowest sorting algorithms which has a best and a worst case running time of Big O(n^2).

BUBBLE SORT (A)

```
1 for i = 1 to A.length - 1
2         for j = A.length downto i + 1
3             if A[j] < A [j - 1]
4         exchange A[j] with A [j - 1]
```

Based on the pseudo code, we can see that this algorithm has an embedded for loops. In each loop we get a runtime of O(n), so each time we run it, it performs a runtime O(n^2). Depending on the way we set up the arrays, the algorithm is a little faster when it is sorting an array that's from low to high. This is because the elements are already sorted and it doesn't have to swipe an element. However when it's sorting an array that's from high to low or random orders, it will run slower because it has to swipe elements in each iteration. In figure 1, we see that the bubble sort swipes adjacent elements in the front and bubbles the largest element to the top.

Insertion Sort:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Insertion Sort is efficient when sorting small amounts of data. The elements are sorted from the left of the array. It works in such a way that everything to the left of the key element is sorted. Based on the pseudo code, the key is at position 2 of the array. So everything to the left of the key element is sorted. The key element is compared to the elements to its left. If the key element is less than the element to its left, then they swap position. These steps are repeated until the key element is at the last position of the array.This sorting algorithm has the best case running time O(n) and the worst case running time of O(n^2).

The InsertionSort works best when the elements are closely sorted. When the elements are closely sorted, we do fewer comparisons to insert elements. which gives us a runtime of O(n) for the best case.. However when the list is not closely sorted, we do more comparison as we iterate through the list. For example, to insert the first element, we need 0 comparison, the second element needs 2 comparison and ect. This would give us 1 +2+3…(n-1)=O(n^2)

SelectionSort:

The Selection Sort works by going through the list of elements in search of the smallest element in the array. Once it finds it, it swaps it with the element in the first position. Then it searches for the second smallest element of the array and swaps it with the element in the second position. It does this until all elements in the array are sorted. This algorithm is also not a fast algorithm and it has a running time of O(n^2).

```
SELECTIONSORT (A)

1 for i = 1 to length(A) -1
2   smallest = i
3           for k = i +1 to length(A)
4           do if smallest > A[k]
5                   then smallest = k
6                   swap A[i] with smallest
```

The Selection Sort has a similar structure as the Bubble Sort in the sense that it has embedded loops. Both the best and the worst case is O(n^2). Given a million elements, the Selection Sort would go through all of the elements searching for the smallest elements, and it would take O(n^2). With small amount of elements to sort, the Selection Sort would also take O(n^2)

Heap Sort:

The HeapSort is a comparison algorithm. The first thing that the algorithm does is build HEAP. Once it does that, then it builds MAXHEAP. In this process, the elements are in ascending order and each node is greater than or equal to its child. After this process, the algorithm does the sorting process. It starts this process by swapping the first and the last node. After each swap, the algorithm does the MAXHEAP. It does this until all elements are sorted. The best and the worst case running time of this algorithm is O(n lgn)

Example:

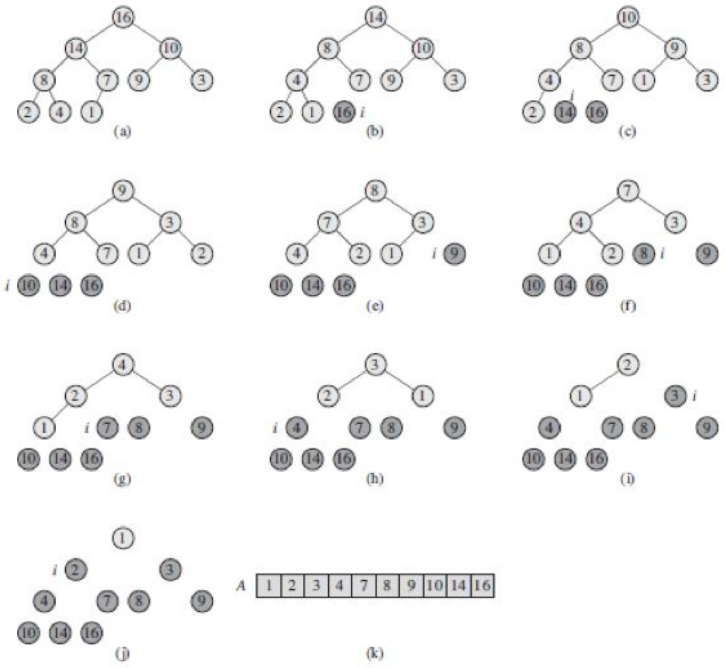


Figure 8: The operation of HEAPSORT. ***(a)*** *The max-heap data structure just after BUILD-MAXHEAP has built it in line 1.* ***(b)–(j)*** *The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap.* ***(k)*** *The resulting sorted array A.*

The Big O for the HeapSort is O(nlgn). The Build-MaxHeap operation runs on O(n). The Max-Heapify operation runs on O(lgn) since we have a binary search tree. Therefore we get a best and worst case runtime of O(nlgn) when the HeapSort algorithm is called.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

MergeSort:

The MergeSort closely follows the divide and conquer process.The algorithm divides the n-element sequence to be sorted into two subsequences of n/2 elements each.Then it sorts the subsequences recursively by calling merge sort method. And then it merges the two sorted subarrays to output the answer. This algorithm has the best and the worst case running time of O(nlgn)

Examples:

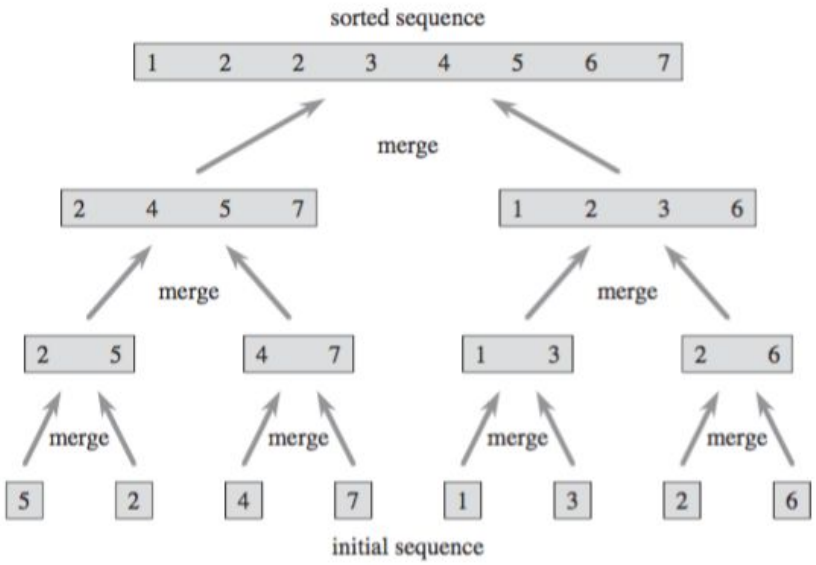


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

The Mergesort runs on an order of O(nlgn). It is a similar tree structure where the number of comparisons that are needed to sort an array of n elements will be narrowed down to smaller parts, n/2. Although the best and the worst case running time are the same, the best case in general for this algorithm is when the elements are sorted from low to high because there are fewer comparisons and the worst case is when the elements are in high to low because there are more comparison.

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Quick Sort:

This sorting algorithm is efficient when sorting small amounts of elements. It works in such a way that when all of the elements are randomly sorted, it performs its best. In contrast, when the elements are nearly sorted, it performs its worst. It uses the notion of divide and conquer. Based on the book that we are using in class, we pick a pivot and assign it to the last element. Then the list is divided into two parts, where the left side of the list is less than or equal to the pivot and the right side greater than or equal to the pivot. The algorithm does this until all of the elements are sorted. This sorting algorithm has a best case running time of O(n log n) and a worst case of O(n^2).

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

For the QuickSort, we get the worst case running time of O(n^2) when the elements are sorted from low to high or high to low. This is because once we call the partition function with nearly sorted array. The pivot would be compared a bigger or smaller elements and put that element in its correction position. By doing this, we would get a runtime of O(n^2).

Based on the figure below, we get the best case running time of O(nlgn) when we are able to divide the number of elements in half. Once we call the two calls for QuickSort, they will

have n/2 elements to sort and which gives a runtime of O(nlgn).

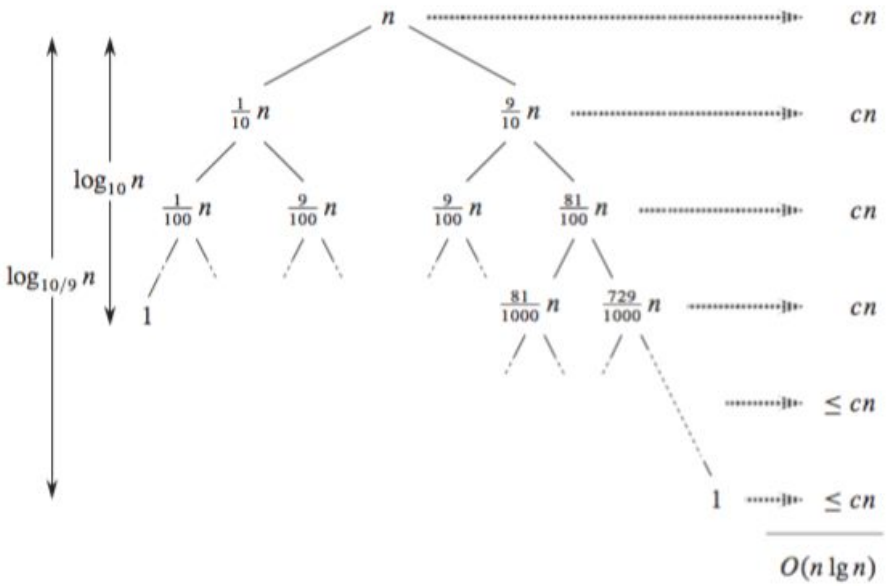


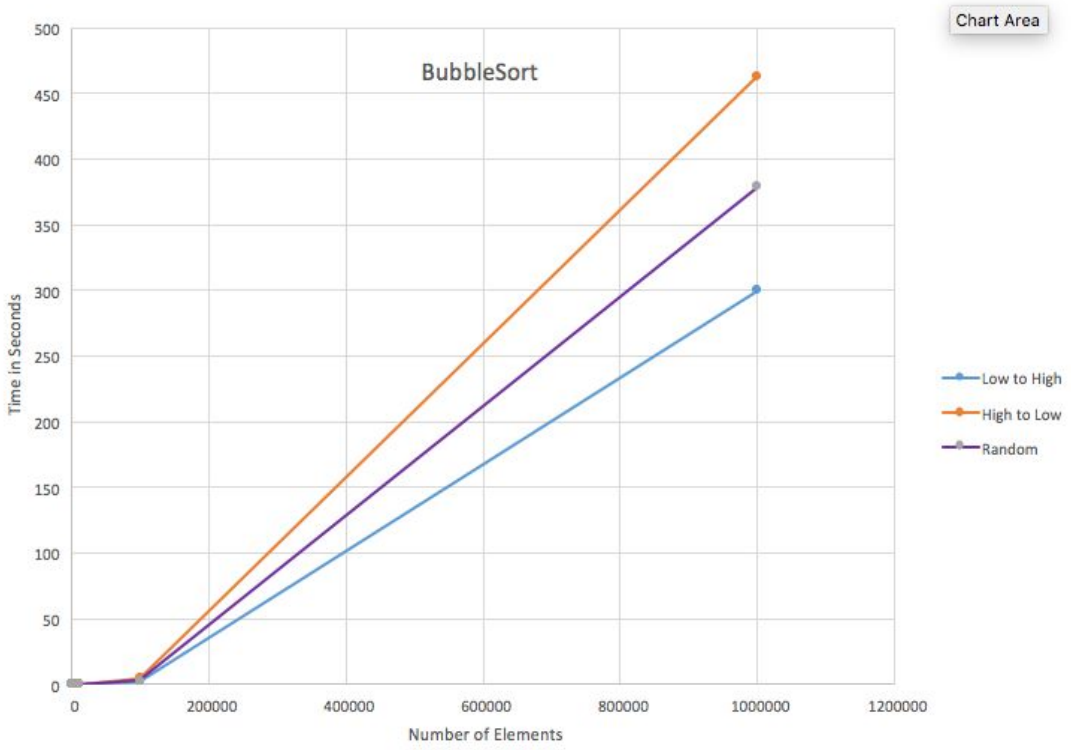
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Hypothesis:

	Low to High	High to Low	Randomly Distributed
Fastest	InsertionSort	MergeSort	Heapsort
Slowest	SelectionSort	BubbleSort	BubbleSort

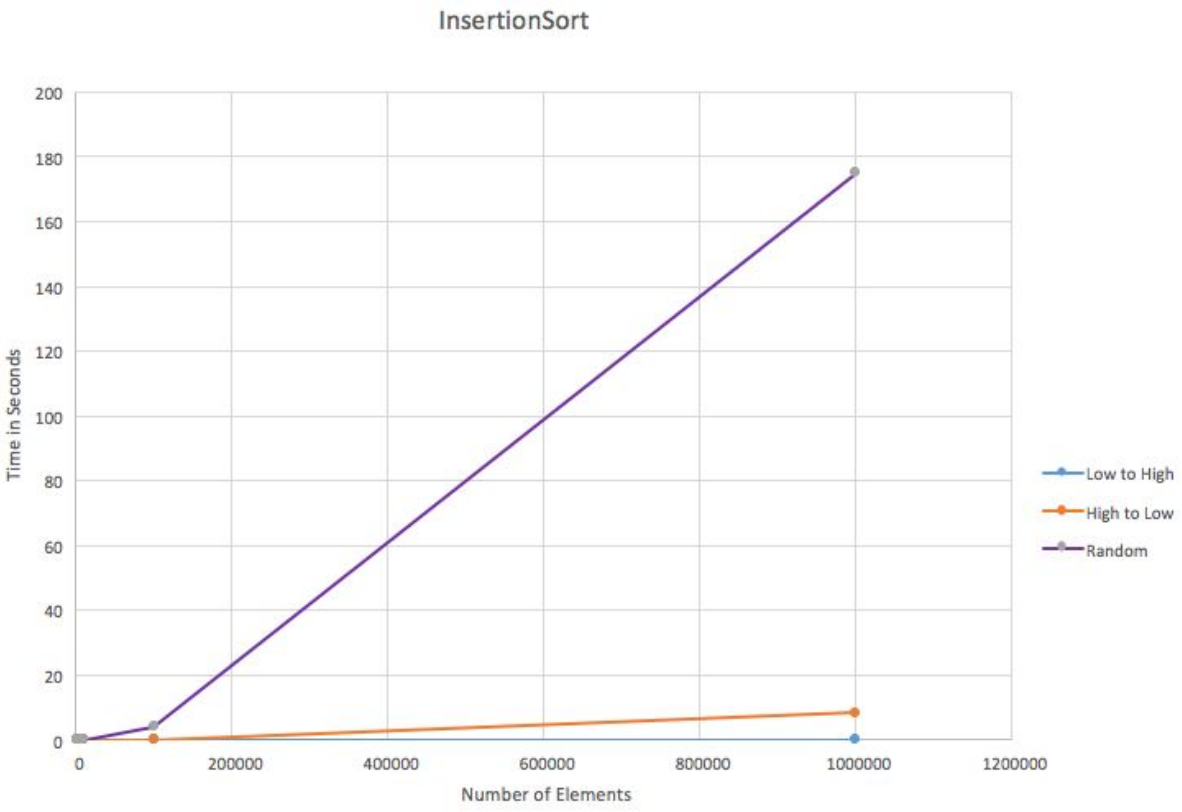
I hypothesize that from low to high, the InsertionSort will be the fastest because the elements are already sorted and it will only move through the elements and compare values. The slowest sorting algorithm from low to high would be Selectionsort. When sorting from high to low, I think the fastest algorithm would be MergeSort and the slowest would be Bubble because of the O(n^2). And when sorting an array that's randomly distributed, I think the fastest sorting algorithm would be HeapSort because of the way HeapSort works and its structure and the O(nlgn). And the slowest would be the BubbleSort because of the embedded for loop and O(n^2).

Result:



# of Elements	Low to High	High to Low	Randomly Distributed
100	0.000147031	0.000213413	0.00014381
1000	0.005852665	0.007197138	0.00054816
10000	0.032008632	0.042306008	0.030833037
100000	2.539723201	4.472922281	3.216017553
1000000	299.868016	463.3045413	379.1030126

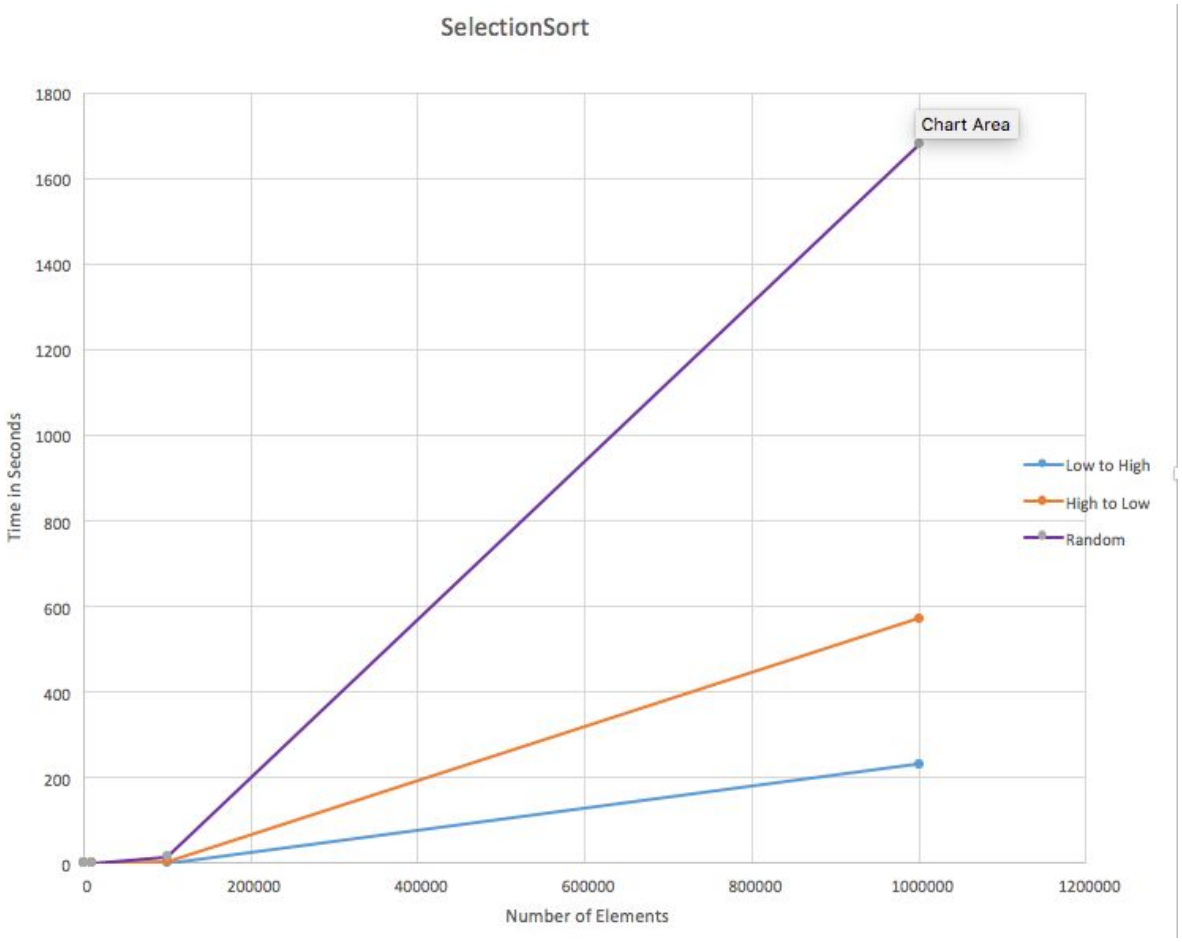
Based on the BubbleSort graph, we can see that the array from low to high ran the slowest and the array from high to low ran the fastest, and the random order is in the middle. It's interesting to see that the array from high to to ran faster than the low to high. I assumed it would be the opposite. We can also see from the graph that as the size of the arrays get larger, the times also get larger as well



# of Elements	Low to High	High to Low	Randomly Distributed

100	0.00001653	0.000168756	0.000075664
1000	0.000049178	0.005979371	0.000753544
10000	0.000450194	0.091107102	0.013832074
100000	0.00427186	0.082061315	4.025936698
1000000	0.008463304	8.443186439	175.1286258

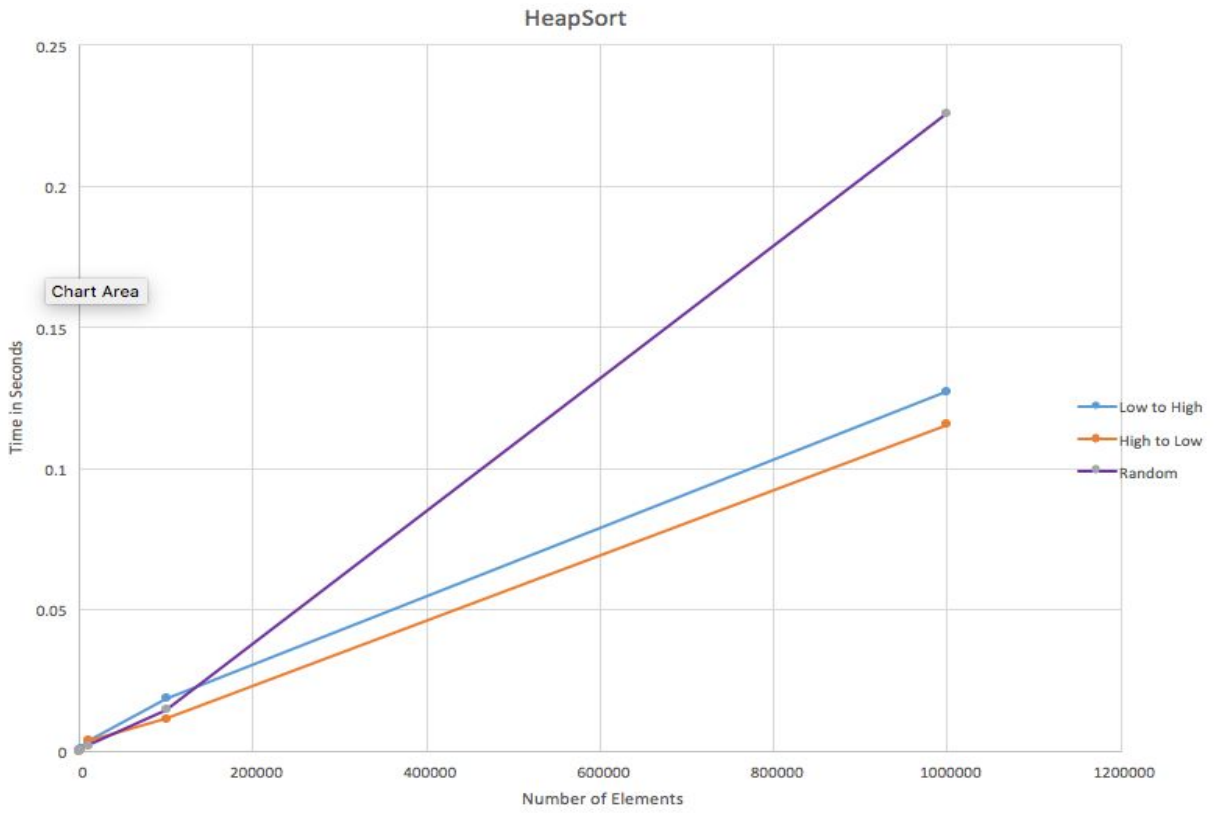
From the InsertionSort graph, the array from low to high ran the fastest, which totally makes sense because the elements are already sorted. The high toow array ran slightly slower than the low to high, and the random order array took a long longer and ran the slowest out of all three. This is strange because I was not expecting the random order would take that long to finish sorting.



# of Elements	Low to High	High to Low	Randomly Distributed
100	0.000232333	0.000176736	0.000325234
1000	0.007895939	0.009285485	0.001507405
10000	0.023007487	0.050517335	0.16042594

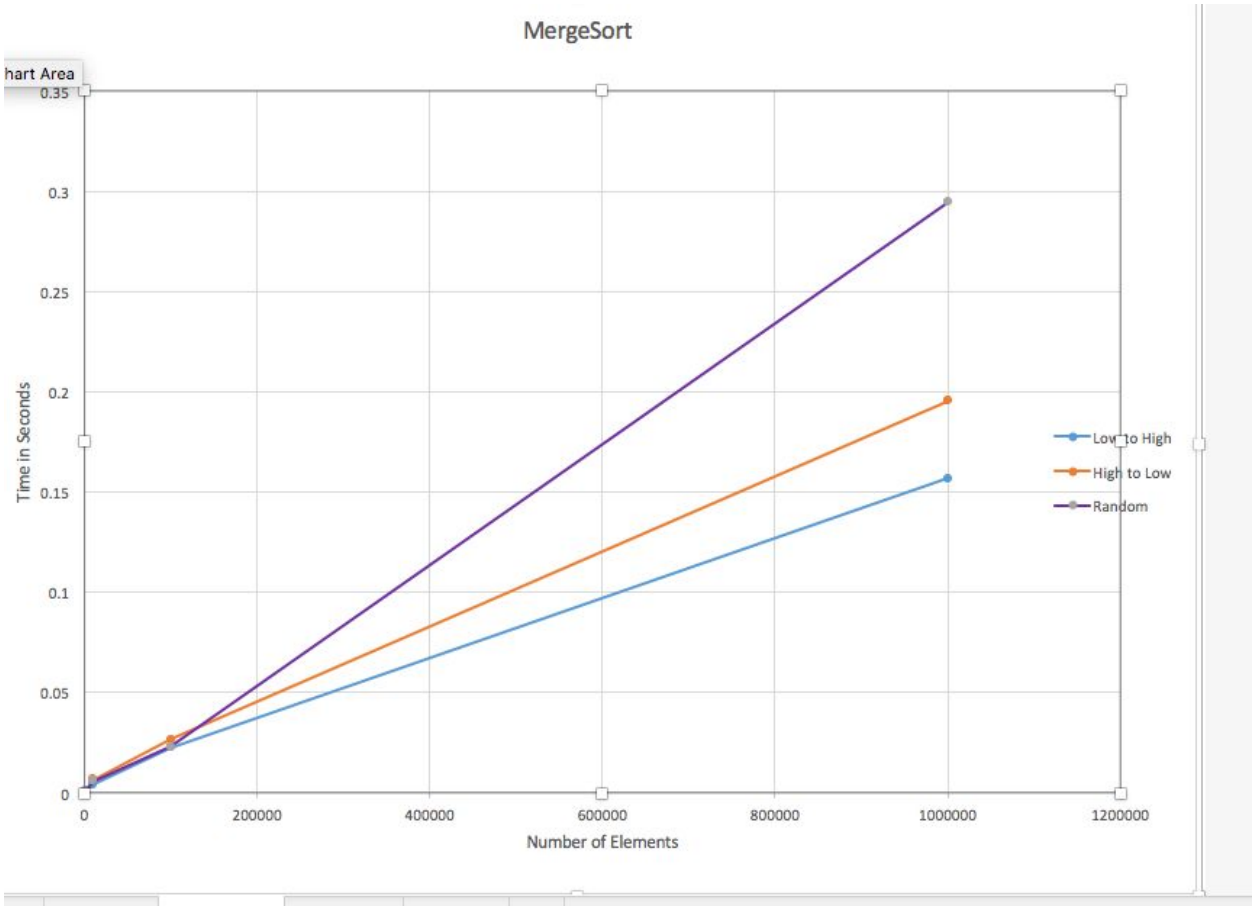
100000	1.648442615	4.56514535	15.51262978
1000000	231.7555201	572.9506599	1680.648966

In the SelectionSort graph, there is a little variation of the different orders of the arrays. Since the SelectionSort goes through the entire array despite the size of the array, I expected all of the order to produce O(n^2), which means they should all be close to each other. But it shows in the graph that the orders of the arrays from low to high, high to low, and random order are far from apart. The random array finished last while the low to high array finished first and the random order array finished second.



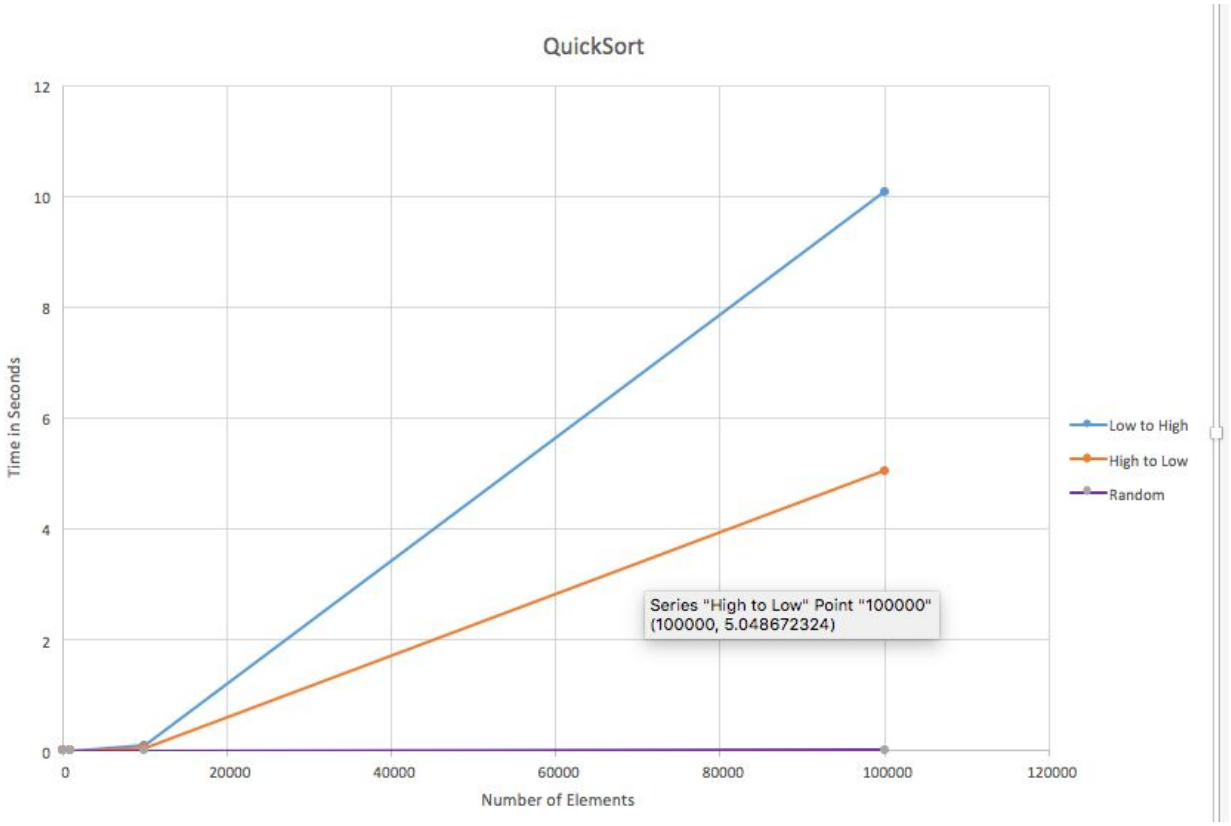
# of Elements	Low to High	High to Low	Randomly Distributed
100	0.000396894	0.000025541	0.00002145
1000	0.000853017	0.000275677	0.000386712
10000	0.003436716	0.003799431	0.002053161
100000	0.018756221	0.011314167	0.014763054
1000000	0.127445393	0.115639983	0.225861321

In the HeapSort graph, we can see that the random array took a lot longer to finish while high to low beat the low to high in small margin.



# of Elements	Low to High	High to Low	Randomly Distributed
100	0.000115329	0.000284735	0.00016497
1000	0.001142319	0.000441509	0.000407361
10000	0.004012642	0.006795824	0.005836105
100000	0.022507587	0.026430579	0.022918376
1000000	0.156820312	0.195315271	0.294467237

The MergeSort seems to be slightly slower than the HeapSort. Just like the HeapSort, the random order array finished last for MergeSort. However the high to low array ran slower in MergeSort than the low to high, which is reversed in HeapSort.



# of Elements	Low to High	High to Low	Randomly Distributed
100	0.000366008	0.00018697	0.000058458
1000	0.00285722	0.001576438	0.00064262
10000	0.090966073	0.050997742	0.00243086
100000	10.09085962	5.048672324	0.015900984

1000000	0.000366008	0.00018697	0.000058458

When I ran the QuickSort, it worked for all sizes except for the array size 1 million. The error that I got said “Your computer does not have enough free memory….” So I was only able to run up to size 100,000. With that being said, the QuickSort ran slower than both the Heap and the MergeSort. The random array is almost an instant finishing up in nearly zero seconds. On the other hand, low to high came last and the high to low finished last.

Conclusion:

In my hypothesis, I predicted that the InsertionSort would win from low to high and I was right. It finished sorting in .086 seconds. In high to low, I predicted that the MergeSort would win, but I was wrong. The HeapSort actually won and finished in 0.1156 seconds, which is .08 seconds faster than MergeSort. In random order, I predicted that the HeapSort would finish first, but I was wrong. The QuickSort won which finished sorting in 0.0159 seconds which is 0.278 seconds faster than the HeapSort. In terms of which sort ran the slowest, I assumed that in low to high, the SelectionSort would run the slowest. However, in the data it showed that the BubbleSort ran the slowest in low to high orders which finished in 299 seconds while the SelectionSort finished in 231 seconds. Also I predicted that in high to low the Bubble Sort would run the slowest. I was wrong and in the data, it showed that the SelectionSort ran the slowest from high to low in 572 seconds while the BubbleSort ran in 463 seconds. In addition, I predicted the BubbleSort would also run the slowest in random order. Once again I was wrong and as the data showed the SelectionSort ran the slowest in random order finishing in 1608 seconds while the BubbleSort ran in 379 seconds. From this experiment, we can conclude that when it comes to a large sets of data, the best sorting algorithm to use is Heapsort and the worst sorting algorithm to use is the SelectionSort