Erik Ayavaca-Tirado
4/24/17
CSC 320 Algorithms
Shana Watters

Algorithms Sort Analysis paper

Introduction:

In today's world the ability to read and understand data and how it plays a crucial role in everyday activities is what can make or break a company. People are always looking for patterns or how something is ordered,which makes sense we do it in our everyday lives like when we arrange people by last name or sorting mail by the date they were sent. A person can easily perform these tasks when the size of the problem is small. When the size of the problem is increased to a size that makes it hard for humans to do, then the need for sorting algorithms arises. So knowing which algorithm to use for certain problems is very important as person or company would like to use their time or money wisely.

Data analysis is used by everyone in the world in order to solve problems in the most efficient way possible. In order to understand this process more I am going to be analyzing six common sorting algorithms and analyze which sort is better for a certain situation. These sorts are the Bubble sort, Insertion sort, Selection sort, Heap sort, Merge sort and Quick sort. These sorts will be implemented using the java language and analyzed in order to find their running time and effectiveness in certain situations to find which sort is better than the other in certain problems.

Methodology:

For our analysis we are using six different common sorting algorithms. These sorting algorithms are the Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Merge Sort, and Quick Sort. I used the pseudo code from the third edition of "Introduction to Algorithms". Using the provided pseudo code I implemented the different sorting algorithms using the Java programming language in the Eclipse IDE. In order to solve a sorting problem in the most efficient way. One must test and examine the performance and run time of each sorting algorithm. In order to analyze the run times of all six sorts the use of Big O notation will be necessary. I have written my own hypotheses on the run time of each sorting algorithm using the Big O notation.

In order to test my hypotheses I have created 5 arrays varying in size. The array sizes go from 100,1000,10000,100000 and 1000000. The arrays are ordered from high to low, low to high and randomly distributed. In order to ensure an accurate measure of efficiency and time for all the sorting algorithms the following conditions are applied:

1. The Arrays can not contain any duplication of elements.
2. The same arrays will be used for each sorting algorithm.
3. The use of System.nanoTime() will be used to keep track of the run time of each sort in nano seconds. This shall later be converted to minutes for easier understanding.

Bubble Sort:

The Bubble Sort is a well known comparison sort. The bubble sort is very inefficient when it comes to a large list of elements within an array. In this sort one can either "bubble up" or "bubble down". In the case of our pseudocode we will be bubbling down. This means that the

sort will start at the end of the list. It starts by comparing two adjacent elements.The smaller element is swapped to the left.This comparison continues until the sort reaches the front of the list. It is at this time that the smallest value has been sorted. This process starts over, this time not including the sorted element, until eventually the whole list is sorted. Below is the pseudocode for the bubble sort:

BUBBLESORT (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]

This algorithm has a best and worst case runtime of O(n^2). This occurs because the sort is composed of two for loops, where one is inside the other. This is called an embedded loop . We know that each for loop has a run cost of n, so we have a total cost of n^2 because of n*n. Due to the algorithm constraints the sort will always have a runtime of O(n^2). However, the order of the element in an arrays will have an effect in the time that bubble sort sorts the array. Thus, bubble sort will run faster when the elements in the array are in a Low-High order since it will only compare the values and no swap is needed. Bubble sort will be slower when the element in the array are in a High-Low order since the values would need to be swapped every iteration of the algorithm.

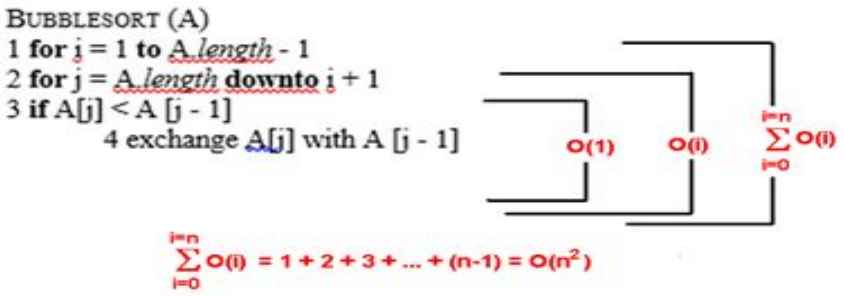


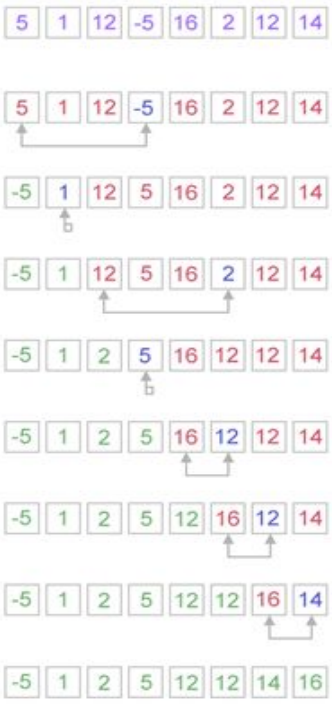
Figure 1: Running time of Bubble sort with respect to the number of steps that the algorithm takes to sort an array of n elements.

Selection Sort:

The Selection Sort is another comparison sort. This sort starts at the beginning of the array. It first chooses the first value in the array as the smallest value called *Smallest* and keeps track of this *Smallest*. It starts by comparing values of the list with this *Smallest* value; if it finds a value smaller than the current value in *Smallest*, then *Smallest* is updated to the smaller value. This comparison continues until the end of list is reached. Now *Smallest* contains the smallest value. *"Smallest"* is sorted to the first position in the array. Now position one is sorted. Next *"Smallest"* is updated to the second value of the array and goes through the list to find the second smallest value. This process continues until the list is sorted. The pseudocode for the Selection sort is below:

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
```

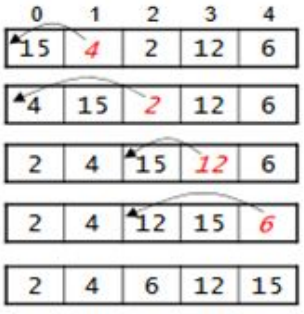


To the right is an example of how the selection sort works by being performed on an array of size eight .This algorithm has a best and worst case runtime of O(n^2). The selection sort has a similar structure as the bubble sort. Since it has an embedded loop it has a runtime of O(n^2)The Selection sort makes n-1 passes, making it fall in the order of $O(n^2)$ as well. Every step of the outer loop requires finding the minimum in the unsorted part. Adding up all the steps the result is n+(n-1)+(n-2)+…+1, which gives us a result of $O(n^2)$. The selection sort algorithm has two for loops and an if statement, making its structure similar to bubble sort. The same approach from figure 1 can be applied to the selection algorithm.

Insertion Sort:

The Insertion Sort like the other two sorts is a comparison sort. This sort is very efficient when dealing with small numbers of elements. This sort can be described as a player drawing cards from a deck. First a player starts off with an empty hand. The player draws a card and places the card in their hand. Next the player draws a card and compares the drawn card to the card in their hand. If the value is smaller then it goes to the left or if the card is bigger it goes to the right of the current card. This process of comparing the card drawn to the current hand continues until the deck has no more cards. Thus the deck/list is sorted. Below is the pseudocode for this sorting algorithm. Also there is an example of how the insertion sort works.

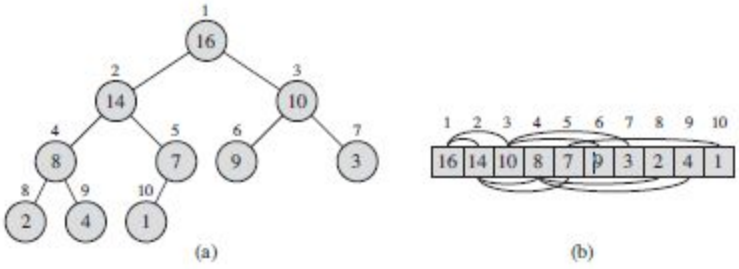
```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



This algorithm has a best case runtime of O(n) and worst case runtime of O(n^2). The best case occurs when the list is listed in ascending order as the number of comparisons and number of elements are the same making it a runtime of O(n). The worst case happens because 0 comparison are needed to insert the first element. 1 comparison to insert the second element, 2 comparison to insert the third element. So (n-1) comparisons at most are needed to insert n elements. 1+2+3+4+…+(n-1) = O(n^2).

Heap Sort:

The Heap Sort uses a total of three methods to organize and sort a list by placing the largest element in its proper place. When given a list of elements , the method Build Max-Heap is called and within this method Max-Heapify is called. Max-Heapify moves each of the elements around until each value is smaller than the parent value. This means the largest value is at the start of the list, creating a max heap. Heap sort then starts to place the largest element in its proper place; this done by swapping the largest value with the value in the last position of the array. Then the array length is decreased by one since the largest has been sorted. This process continues until the whole array is sorted. For this sort you can imagine the input array as an binary tree as shown below:



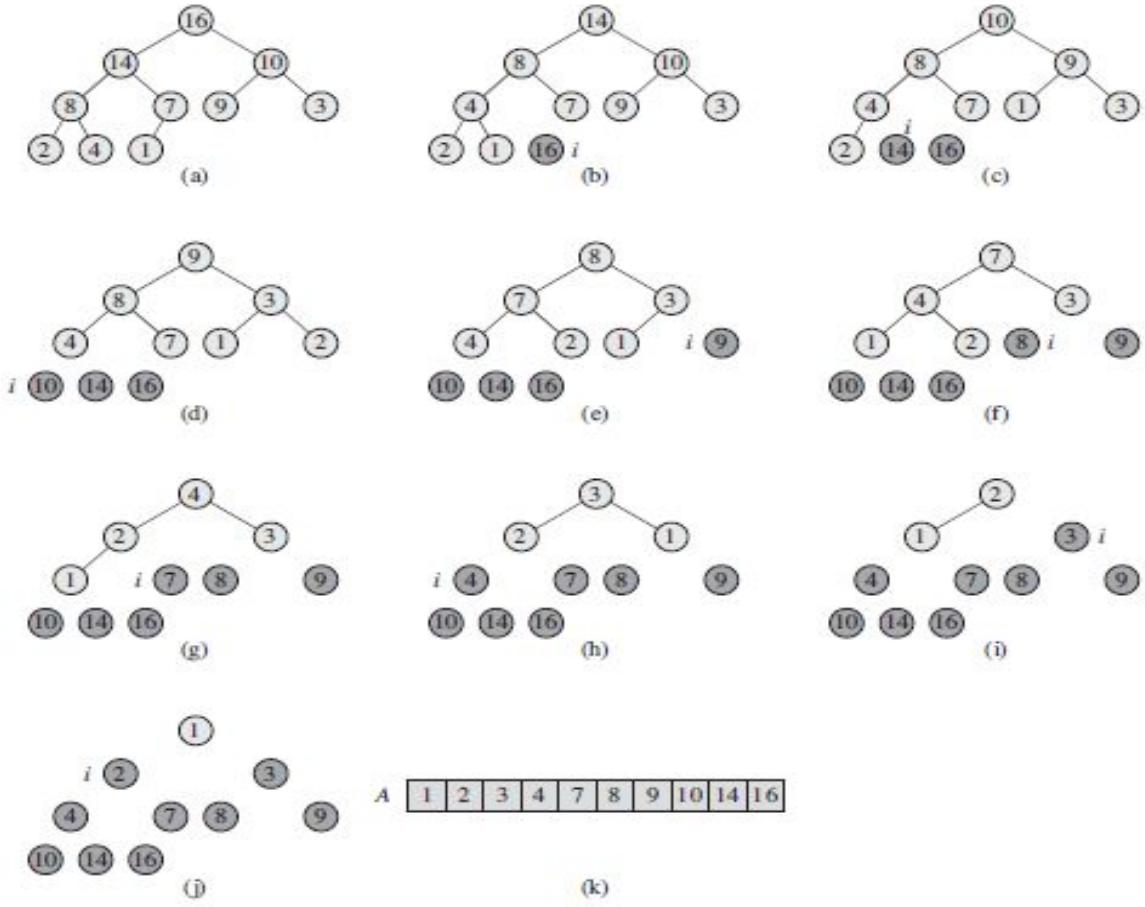
Below is what the pseudocode for the Heap sort looks like as well as an example to show how the heap sort works.

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

```
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)
```

```
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)
```

The Heap Sort has a best case and worst case runtime of O(n*lg(n)). The build max-heap operation is run once, and it performs a O(n). The max-heapify function runs in O(lg n) since the tree structure is binary. When the heap-sort algorithm is called this two functions add up to (n+ lg n) which makes the running time performs a O(n*lg(n)) in the best, average and worst case. Below is an example of this sort works:



Quick Sort:

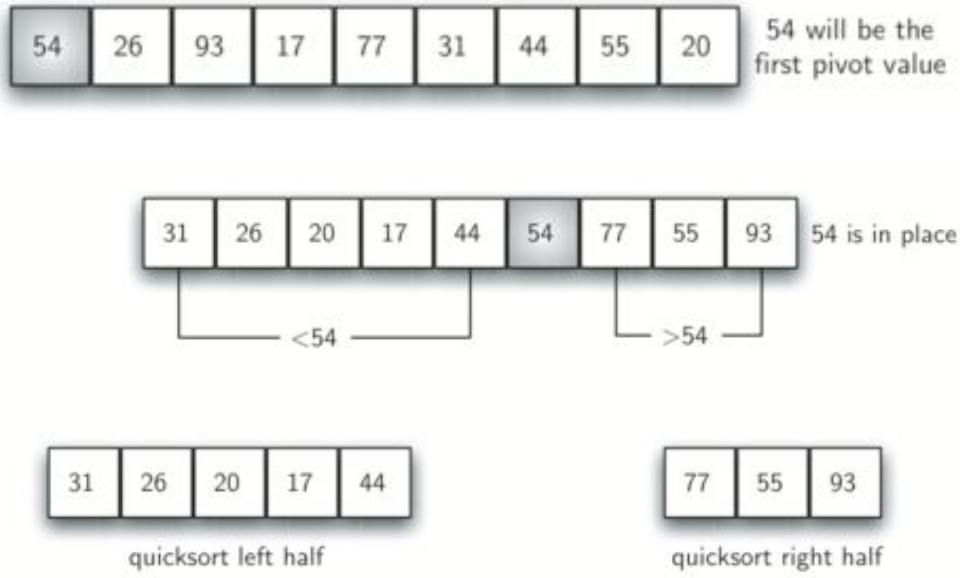
The Quick Sort is a divide and conquer algorithm. The Quick Sort starts by choosing a pivot. The pivot is usually the last element of the unsorted array. Everything in the unsorted array

is going to be compared to the pivot. It starts by comparing all values left of pivot and if are less than value in the pivot then they are placed left of the pivot. If the values are greater than the pivot then they are placed to the right. This process creates two sublists where everything left of pivot is smaller than the pivot and everything right of the pivot is larger. After the whole array is sorted into these two sublists, The pivot is placed in its correct position. This process continues until the whole array becomes sorted. Below is the pseudocode for the Quick Sort.

```
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
```

Below is an example of Quick sort:



The Quick Sort has a best case runtime of O(n*log(n)) and worst case runtime of O(n^2).. The worst case would occurs when the pivot is the smallest value in the array, dividing the list in 0 elements in the left and n-1 elements in the right. The other worst case scenario would be to encounter values that are equal, however it has been established at the beginning that the array will not have duplicates, which makes the previous scenario impossible.

Merge Sort:

The Merge Sort is a divide and conquer algorithm. It starts by finding the midpoint of an array. Then it divides the original array into two sublists. The process of dividing each sublist in half continues until it reaches the base case, which means that each sublist only contains one

element. Now, with the sublists created, the sort starts merging the two sorted sublists using comparisons and then is sorted. The process of merging, comparing, and sorting the elements continues until it reaches the original array size. Below is the pseudocode for this algorithm.

```
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)
```

Below is an example of how the merge sort fully works:

```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 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
```

Example. Consider the following array of numbers

```
27  10  12  25  34  16  15  31
```

divide it into two parts

```
27  10  12  25          34  16  15  31
```

divide each part into two parts

```
27  10          12  25          34  16          15  31
```

divide each part into two parts

```
27      10      12      25      34      16      15      31
```

merge (cleverly-!) parts

```
10 27       12 25       16 34       15 31
```

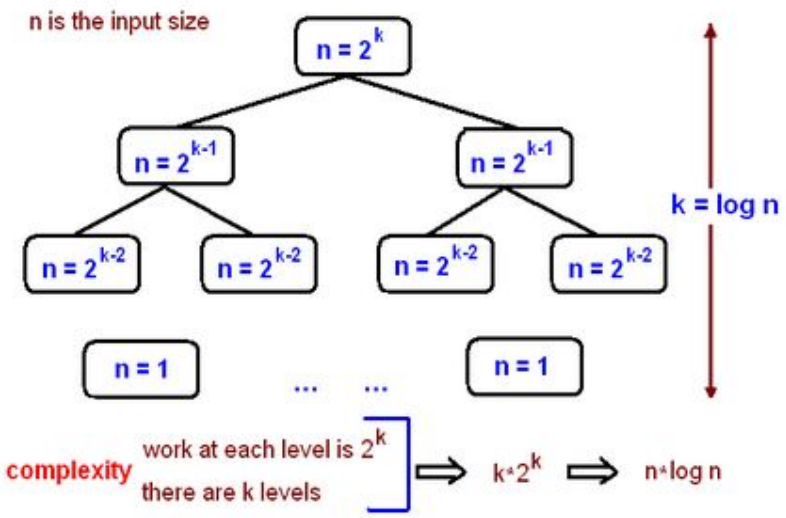
merge parts

```
10  12  25  27                15  16  31  34
```

merge parts into one

```
10  12  15  16  25  27  31  34
```

The Merge Sort has a best case and worst case runtime of O(n*log(n)).Sine merge sort behaves similarly to a tree structure the number of comparison that are needed to sort an array of n elements will be reduced into smaller parts, n/2. And at the end there is still an n-1 comparison to merge two parts in one. The figure below will show how the worst scenario occurs



Hypothesis:

Each of the six algorithms have different runtimes depending on the arrays which need to be sorted. The form in which the elements of an unsorted array can cause a sort to produce the

algorithms best case scenario over it's worst case or vise versa. Below is a table of run times for each of the sorting algorithms and then a table on my predictions on what sorts are better in sorting depending on the situations of the input array.

Table 1: Best and Worst Case runtimes for the six sorting algorithms:

Sorting Algorithm	Data Type/Structure	Best and Worst Case runtime
Bubble Sort	Array of Numbers(Integers)	O(n^2)
Selection Sort	Array of Numbers(Integers)	O(n^2)
Insertion Sort	Array of Numbers(Integers)	O(n) and O(n^2)
Heap Sort	Array of Numbers(Integers)	O(n*lg(n))
Quick Sort	Array of Numbers(Integers)	O(n*lg(n)) and O(n^2)
Merge Sort	Array of Numbers(Integers)	O(n*lg(n))

These times are based on my observations for different characteristics each of the six different sort algorithms have. As well the fact that each of them behave specifically.

Table 2: My predictions on the fastest and slowest algorithm based on the array size and order. Used the best case runtimes

Size of arrays	100	1,000	10,000	100,000	1,000,000
Array order					
Low to High					
Fastest	Insertion O(n)	Insertion O(n)	Insertion O(n)	Insertion O(n)	Insertion O(n)
Slowest	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)
High to Low					

fastest	Merge O(n*log(n)	Merge O(n*log(n)	Merge O(n*log(n)	Merge O(n*log(n)	Merge O(n*log(n)
Slowest	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)
Random					
Fastest	Heap O(n*log(n))	Heap O(n*log(n))	Heap O(n*log(n))	Heap O(n*log(n))	Heap O(n*log(n))
Slowest	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)	Bubble O(n^2)

Predictions Reasoning:

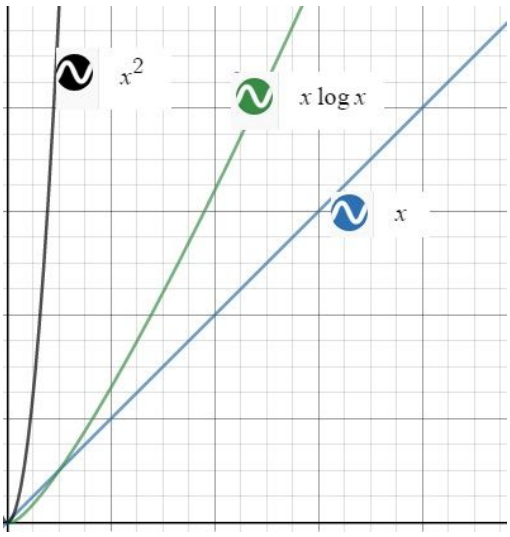
Reasoning for slowest in all arrays. The bubble sort is one the most inefficient sorting algorithms. This happens due the constraint of having embedded for loops. This forces the input array to go thru each for loop. Even though the list may already be in sorted order. I believe that for they reasons the bubble sort will take the longest time for an array sorted high to low and one of random distribution. Now when the elements are arranged from Low to High, the bubble sort will still be the slowest even if the selection sort has a similar run time and structure.

Reasoning for the fastest sort in terms of low to high. I believe the insertion sort will be the fastest because with this array you a practically producing its best case which is O(n) and nothing is faster than O(n).

Now I listed sorted from high to low. I assume the fastest will be merge sort due to the O(n lg n) run time .this was a difficult choice since the merge sort also has O(n lg n) running time. In the end I decided with merge because I feel the process of dividing the array and then merging the sublists is faster than creating creating a max heap and sorting it in it's rightful position.

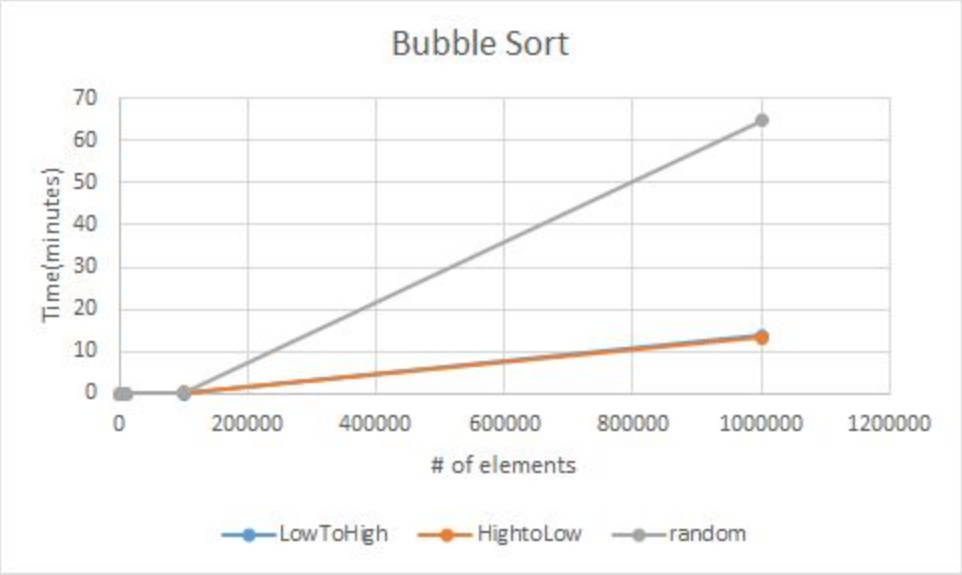
Now unto the list sorted in random order. I assume that the fastest will be the heap sort. I believe this because It has a running time of O(n lg n), due to the tree like structure I'm assuming that the heap sort will perform the fastest thank merge sort in this case.

Test results:



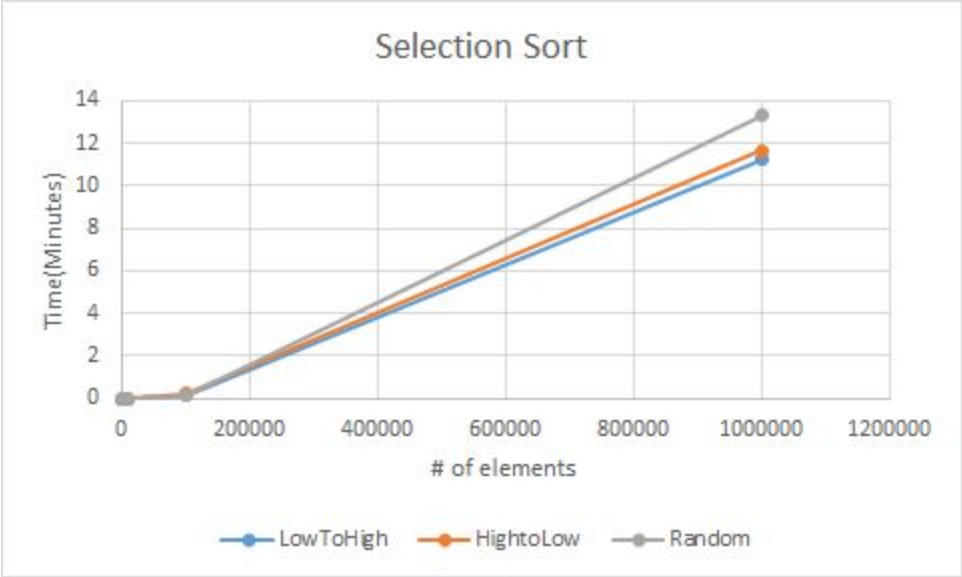
Before I talk about my test results I would like to go over what the graphs could possibly look like to get a better idea of why runtime is very important. Below is a picture of the functions x^2, x and xlog(x). The figure shows us that in long run the function F(x)= x is faster than both F(x)= x*log(x), which is faster than F(x)= x^2.

Bubble Sort Graph:



This data set matches the idea that the bubble sort sorts at about a O(n^2) . It looks like the random distribution array took way longer than an array sorted from Low to High or High to Low. This is expected because since the unsorted array is random we have no clue on the order. I believe this forces the bubble sort to take longer as there can be many swaps that occur. The other two unsorted arrays are sorted at similar times. It looks like the High to Low unsorted array sorts faster than an unsorted array from Low to High. This is kinda unexpected because I would think the array from Low to High would be sorted faster. Based on these test results we can conclude that the bubble sort while a common sorting algorithms is pretty inefficient in terms of run time. It turns out that one could use the bubble sort for a small number of elements but it would be bad to use this sort on a larger amount of element especially if the unsorted array is in random distribution. There are other sorting algorithms that are faster and more efficient when compared to this sort.

Selection Sort Graph:

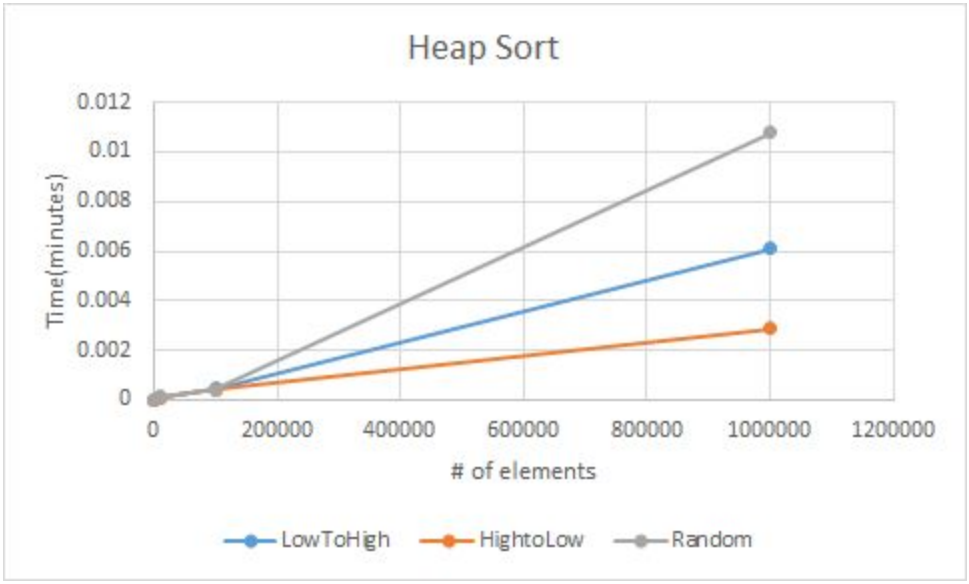


Another sort similar in structure to the bubble sort is the selection sort. So this means that they are going to have a similar best and worst case run time. This algorithm has a best and worst case run time of O(n^2). From this graph you can see that no matter the unsorted array arrangement you will always get a similar run time.

Insertion Sort graph:

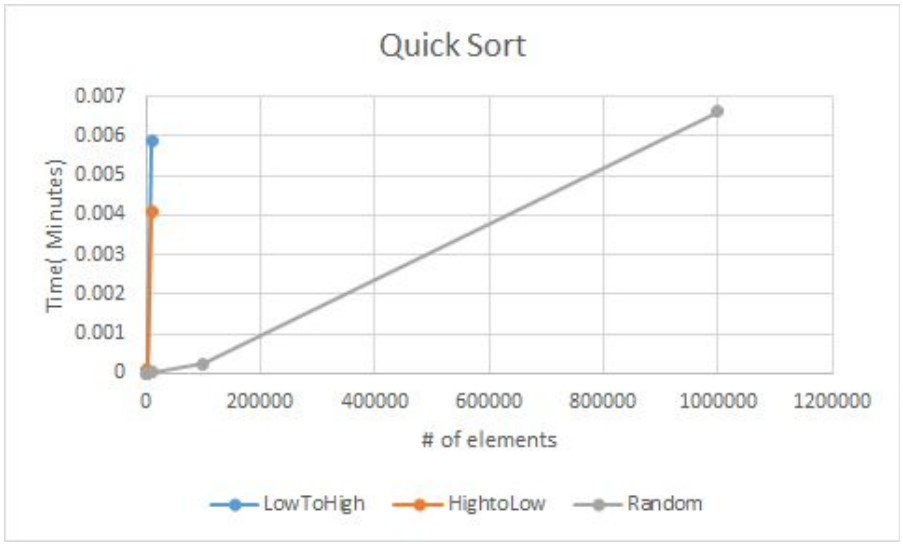
The insertion sort has a best case of O(n) and a worst case of O(n^2). Using these results can see the insertion sort best case occurs when the unsorted array is arranged from Low to High. This makes sense because the array is already in sorted order so the insertion sort does not work as hard. This of course is the opposite when the unsorted array is arranged from High to Low. This unsorted array produces the algorithms worst run time which is O(n^2).

Heap Sort Graph:



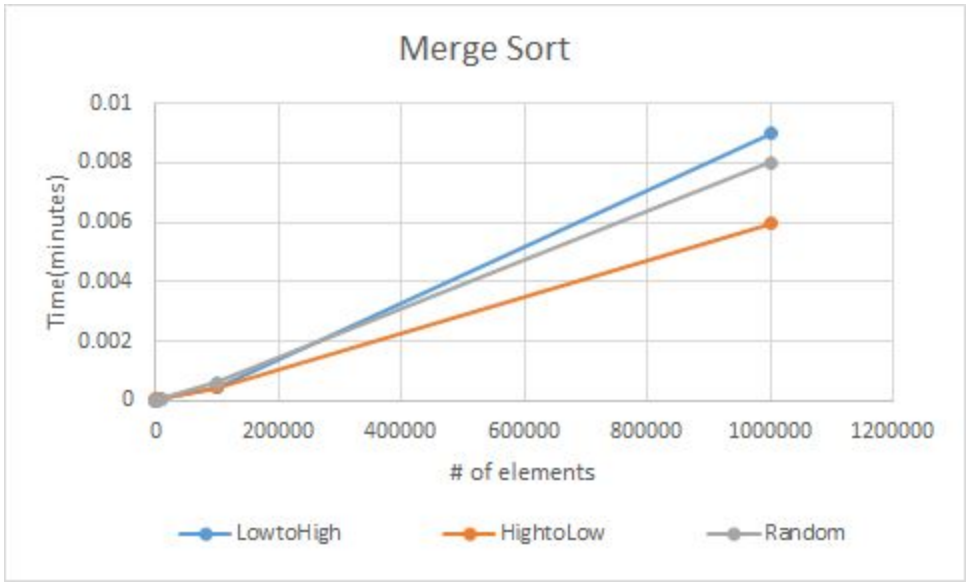
The Heap sort has a best and worst case run time of O(n*lg(n)). This is clearly shown by the data plotted above. It shows how fast the heap can run. The array which was sorted the fastest is the array going from High to Low, this occurs because one heap sort properties called Max-Heapfy and build a Max Heap. This takes time to due but since an array sorted High to Low is already in a max heap, it reduces the run time. When the unsorted array is randomly generated it takes longer to create the max heap.

Quick Sort Graph:



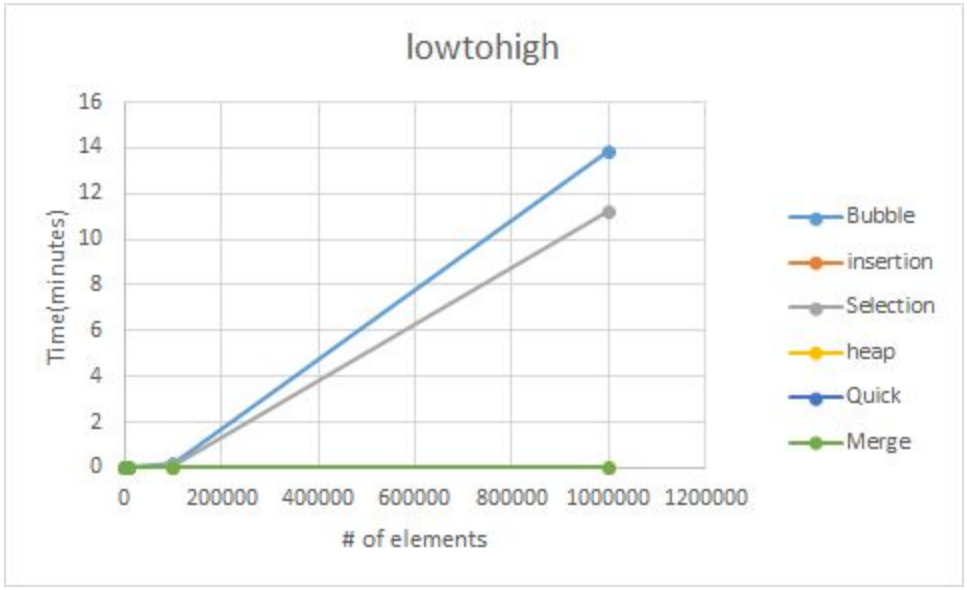
The quick sort is a recursive sort. While have a best case of O(n*lg(n)) and a worst case of O(n^2). Due to its recursive nature the quick sort uses a lot of memory. I learned this the hard way. While testing this sort at 100,000 elements arranged from Low to High and High to Low , computer ran out of memory and made the IDE Eclipse to crash. However the quicksort was able to run for unsorted array of sizes 100,000 and 1,000,000.

Merge sort Graph:

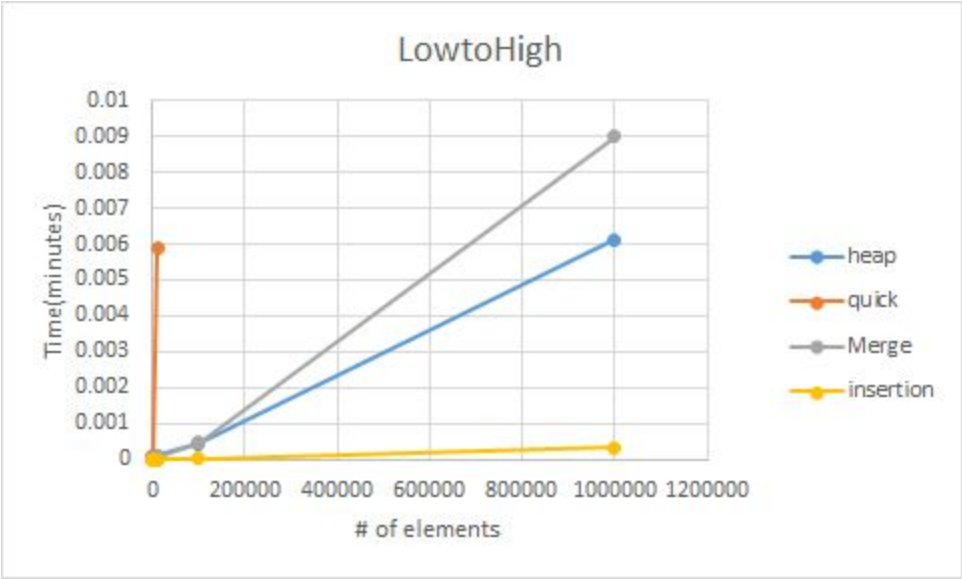


The Merge sort has a run time of O(n*lg(n) . The merge sort is a recursive algorithm. As you can see the merge sort is very fast. All three unsorted arrays were sorted at similar times. I found the results surprising in because I would have thought that the array sorted from Low to high would run faster because it is already in sorted order but in this case it looks High to Low ran faster. The unsorted random array even ran faster than the unsorted Low to High.

Low to High Graph:

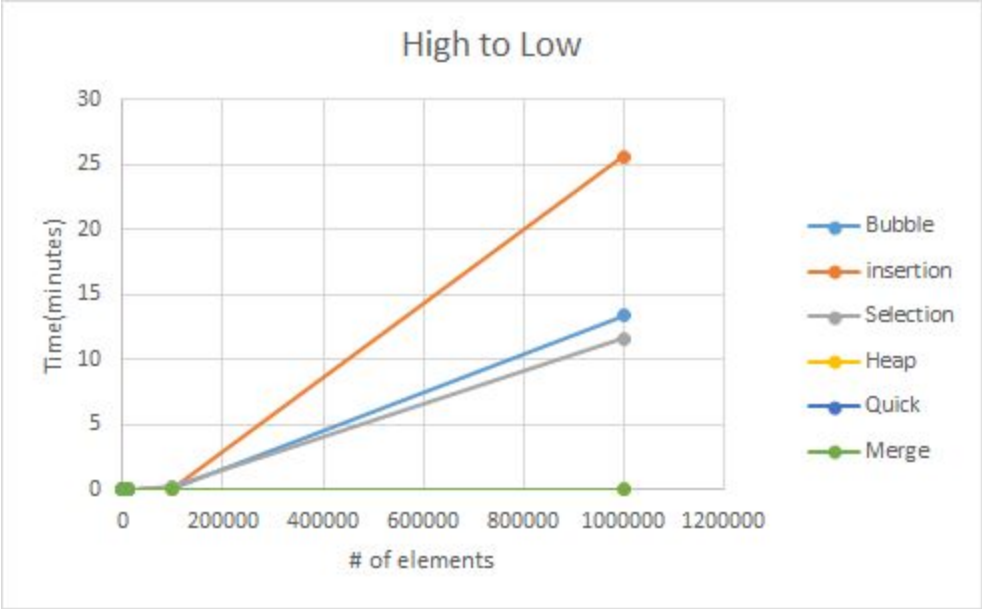


When comparing all six sorts together we can see that when the elements within an array are sorted from Low to High, the slowest sort is that of the bubble sort. Which proves my hypothesis from earlier. This makes sense due to the fact the bubble sort structure forces it to run through both of the for loops. Now to see which is a faster sort we must compare heap,merge,insertion and quick. Below is a graph which shows this.

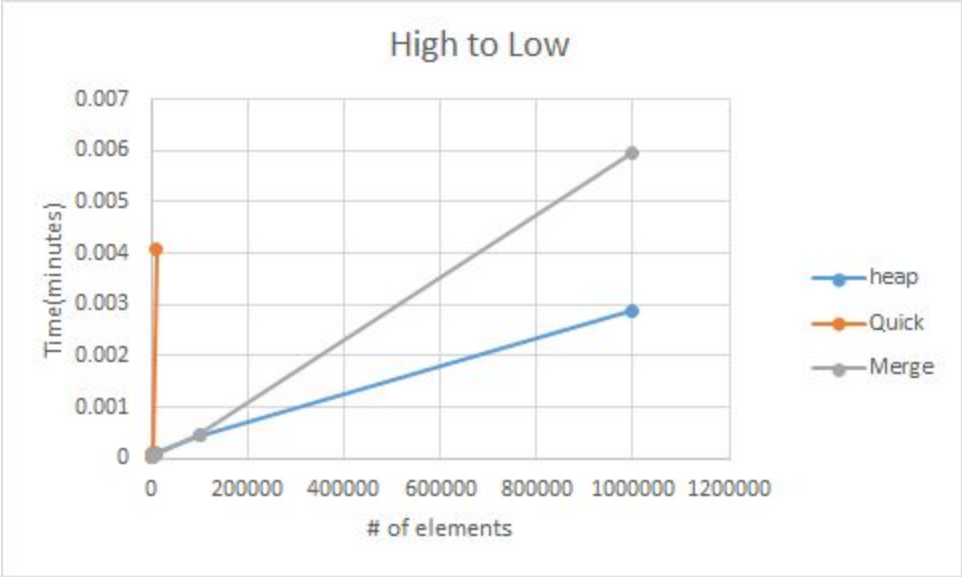


Based on the graph we can see that the insertion sort has the fastest running time compared to the other sorts. This occur because the unsorted are replicates the insertion sort best possible run time which is O(n). This proves my hypothesis of the insertion sort running the fastest when the unsorted array is sorted from Low to high.

High to Low Graph:

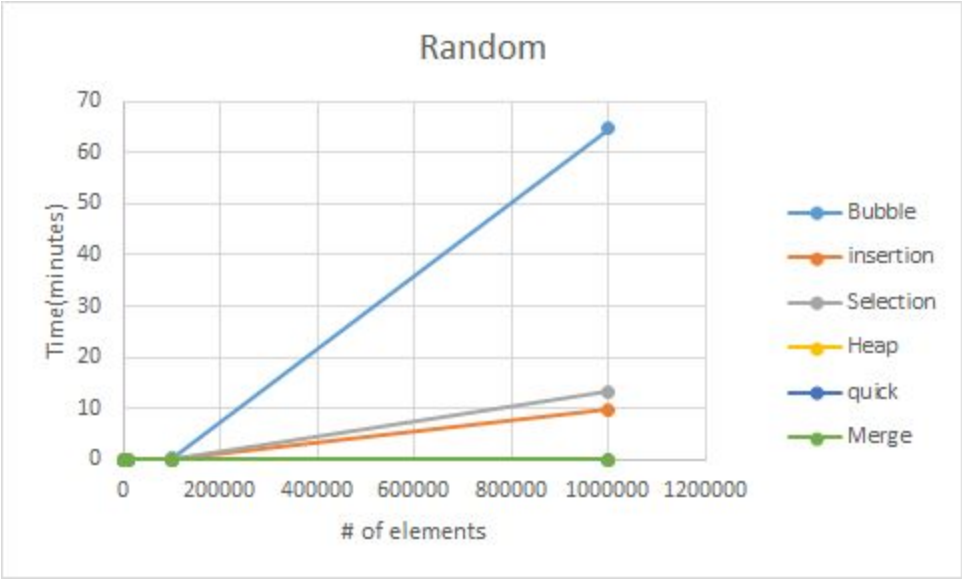


Based on these results we can see that the slowest sort was the insertion sort This disproves my thought of the bubble sort being slower. This result makes a lot of sense because of the insertions sort nature it would force it to take a longer time. Plus with the bubble sort since it is a bubble down then no swaps will occur but it still has O(n^2) run time because of embedded for loops. Below is a graph comparing the faster sorts to see which one is truly the fastest in this case.

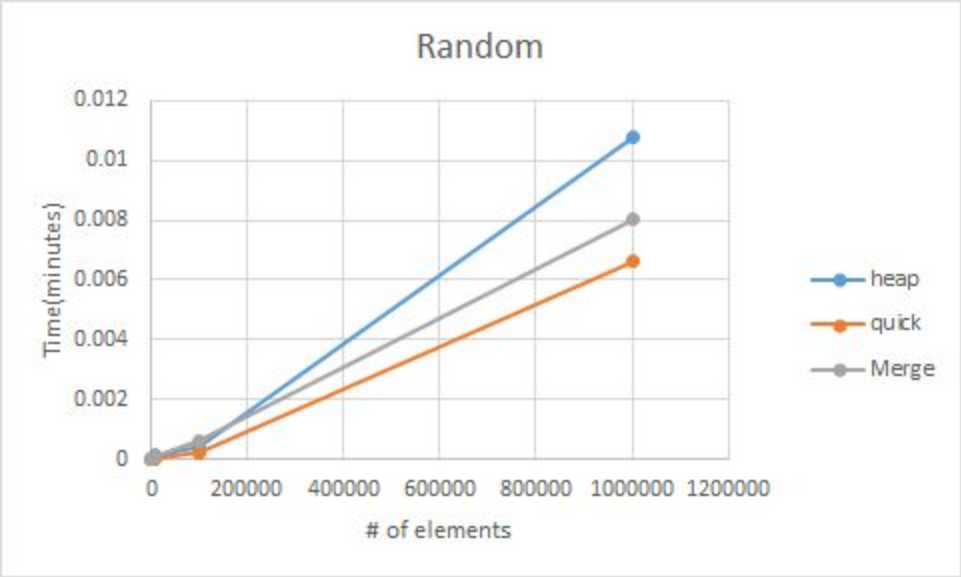


Based on this graph we can see that the heap sort performs the fastest. This proves my hypothesis of the heap being the fastest when the unsorted array is arranged for High to Low. This happens because the heap sort property of building a max heap. Since the array is already in a max heap it makes the process a lot faster.

Random Graph:



In case of the random arrangement of elements we see that the bubble sort really takes a long time. This happens since the elements are random so there are more swaps that occur making it run longer. This proves my initial thought of the bubble sort is very slow. Now in order to see which is the most efficient we must compare the sort that we can really tell how fast they ran



Above is a graph of heap,quick and merge sort to see which sorts the fastest. Based on the results we can see that the quick sort sorted the fastest when the elements are arranged in an random order, This disproves my initial thought of the merge being the fastest. The merge sort came second in terms of run time. But in terms of being the most efficient I would pick the merge due to the fact the quick sort is very memory intensive.

Conclusion:

The truth is knowing when a algorithm is going to perform its best or worst is very important because in today's real world time is money. Knowing this what sets amateurs from professionals in trying program the most efficient way. I believe that having an basic understanding of how algorithms work and of their time complexity is most beneficial for all computer programmers as we are getting paid to do these kinds of things. Based on the results all six sorts can be used when dealing with smaller amounts of data. As the number of elements grow it would be inefficient to use the bubble, selection or the insertion sort, One would most likely choose one between the merge, heap or quick sort. When deciding of sorting many elements of unsorted array that goes from low to high, one should choose the insertion sort as in this case it has a run time of O(n). Now if the elements are in the form of high to low , you probably use the heap sort to sort the elements because it sorted the fastest between all of them. Should the elements be in a random order, one should use the quicksort to sort the elements but it would be inefficient because of how memory intensive it is but the merge sort is another option.

Works Cited:.

Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to algorithms, 3rd edition*.

- Used for the pseudo code and some of the sort examples explanations,

Selection Sort. (n.d.). Retrieved April 10, 2017, from http://www.algolist.net/Algorithms/Sorting/Selection_sor

- Used for the selection sort example.

(n.d.). Retrieved April 11, 2017, from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html

- Used for sorting examples