David Vences 801045180

Project One

Background

The purpose of this project is to analyze the different comparison based sorting algorithms for their performance under different data input sizes. The main goal of these sorting algorithm is to sort a list of elements from either ascending or descending order by comparing elements and manipulating the list as necessary. In order to analyze these sorting algorithms we will be looking at their time complexity (Big O notation) and real-life performance. The specific algorithms this project covers are:

- Insertion Sort
- Merge Sort
- Heap Sort
- In-Place Quick Sort
- Modified Quick Sort

One major observation that can be made from this analysis is that the performance of the algorithm greatly depends on the input size and magnitude of order/disorder. For instance, as will be discussed further below, insertion sort struggles with larger unordered number lists but is very efficient the closer the data is to an ordered number list as well as with small data sets. **Algorithms**

The algorithms that were analyzed were insertion sort, merge sort, heap sort, in-place quick sort, and modified quick sort. Below you can see a brief background on the specified sorting algorithm, some pseudocode, and finally the actual code used for this project.

Insertion Sort

The first sorting algorithm we will look at is insertion sort. This is one of the more simple comparison-based sorting algorithms that was analyzed. Insertion sort is known for being efficient with small data sets and also data sets that are partially sorted. This aspect of insertion sort is less desirable since it is usually unknown the degree to which the data set is sorted. As the name suggests this algorithm involves

inserting an element into its correct position within an array. The pseudocode for this algorithm is seen below:

```
Algorithm insertionSort(A, l, r)
    input:
    A, array of numbers needing sorting.
    l, left most index
    r, right most index
    for i <- l to r
        c <- A[i]
        j <- i-1
        while j >= 0 and A[j] > c
            A[j+1] <- A[j]
            j <- j - 1
    A[j+1] <- c
```

We start off by considering the first element sorted and thus skipping to the second element. The second element is then considered the comparator. We compare the comparator with the elements to its left and move them to the right if they are greater when the value being checked is no longer larger than we insert the comparator at that index. This process is repeated until every position is checked and moved to its sorted position.

In Big O notation the time complexity is O(n^2) since in the worst case, reverse sorted data set, every value will need to be checked against every other value. Due its simplicity this algorithm may be an attractive option when dealing with smaller data sets but other sorting algorithms will be more efficient where insertion sort fails.

Merge Sort

The next sorting algorithm we will discuss is merge sort; this algorithm is widely used because of how efficient it is due to it's divide and conquer strategy. Merge sort divides the input array in smaller sub-

arrays and sorts them recursively. After it is finished sorting sub arrays it merges them together resulting in a fully sorted array. The pseudocode for merge sort is shown below:

Algorithm MergeSort(A, p, r)

Input A, array of elements.

p, pivot

r, right side index

Output: Sorted array A

if p< r

q = round((p+r)/2)

MergeSort(A, p, q)

MergeSort(A, q+1, r)

Merge(A, p, q, r)

The main steps of merge sort are divide, conquer, and merge. The input array is divided into two roughly equal halves. This process continues recursively until there is only one element. Pairs of sub-arrays are sorted individually until finally merged back together in the correct order.

Merge sort has a time complexity of n log n in all cases, this is due to the fact that the problem is divided at each iteration so it is not necessary to check against all other elements like it is in insertion sort.

Heap Sort

Another sorting algorithm that was analyzed was heap sort; the main feature of this algorithm is that it uses a binary heap data structure to sort an array. First, the input array is turned into a binary heap, also known to heapify the array. Next the largest element, or root, is swapped with the last element of the array and the heap size is reduced by one. This process is repeated until until all elements are in the right order. The time complexity for heap sort is O(n log n). The pseudocode is below:

Algorithm Heapify(A, n, i)

Input: A, n, i

Largest <- i

LeftChild <- 2i+1

RightChild <- 2i+2

If leftChild <= n and A[I] < A[leftchild]

largest = LeftChild

Else

largest = i

If rightChild <= n and A[largest] > A[rightChild]

largest = rightChild

If max != i

swap[A[I], A[largest])

Heapify(A, n, largest)

Algorithm HeapSort(A)

Input: A, array

N = length(A)

For i <- n/2 to 1

Heapify(A, n, i)

For I <- to 2

swap(A[1], A[I]

A.heapsize = A.heapsize - 1

Heapify(A, I, 0)

Quick Sort

Quick sort is the final sorting algorithm analyzed. Quick sort is a popular sorting algorithm for it's ability to sort large data sets. The two quick sorts that were looked at were in-place quick sort and modified quick sort. In-place quick sort refers to the method of sorting elements directly within the array rather than needing additional space for sub arrays.

The main feature of quick sort is the idea of partitioning the input array into two sub-arrays around a chosen pivot. Elements smaller than the pivot are then placed to the left of the pivot and elements are larger are placed to the right. The pivot in this analysis was chosen using the center-of-three method. Modified quick sort was also analyzed. This variation of quick sort makes an improvement to in-place quick sort by sorting smaller partitions using a different method, such as insertion sort. The big O time complexity of quick sort is O(n log n).

Time Complexity

The time complexity for the sorting algorithms varies on the input. This analysis was made using a random order list of elements, sorted order, and reverse sorted. Below is a figure that displays the time complexity for each sorting algorithm based on the input data.

	Insertion Sort	**Merge Sort**	**Heap Sort**	**In-place Quicksort**	**Modified Quicksort**
Random Order	O(n^2)	O(n log n)	O(n log n)	O(n log n)	O(n log n)
Sorted Order	O(n)	O(n log n)	O(n log n)	O(n^2)	O(n log n)
Reverse Order	O(n^2)	O(n log n)	O(n log n)	O(n^2)	O(n log n)

Results

Time complexity of sorting algorithms can vary a lot and their performance depends on several factors like input size and data distribution. Below you can see real life benchmarks of these algorithms and how they compare to theoretical performance.

n	**Insertion Sort**	**Merge Sort**	**Heap Sort**	**In-Place Quick Sort**	**Modified Quicksort**
1000	3.51E+03	3252.3252	3895.8961	2394.5919	774.8625
2000	9.07E+03	5174.04175	6299.85425	3911.80845	1247.21655
3000	1.65E+04	7110.063933	8840.6167	5537.819567	1764.404067
5000	3.46E+04	10184.85945	12829.5209	8338.596975	2566.021825
10000	9.89E+04	16376.84424	21155.36754	14582.89344	4218.95412
20000	3.17E+05	28230.45217	36723.92367	28758.81817	7624.486083
30000	7.30E+05	43442.33816	56606.23337	50045.62747	11977.29226
40000	1.35E+06	61071.77088	79473.43335	78383.32561	17106.89375
50000	2.19E+06	80337.30743	104439.4116	113154.5691	22837.54953
60000	3.25E+06	100792.1334	131168.9221	154567.1726	28918.0875

Figure 1: Random Order Raw Results

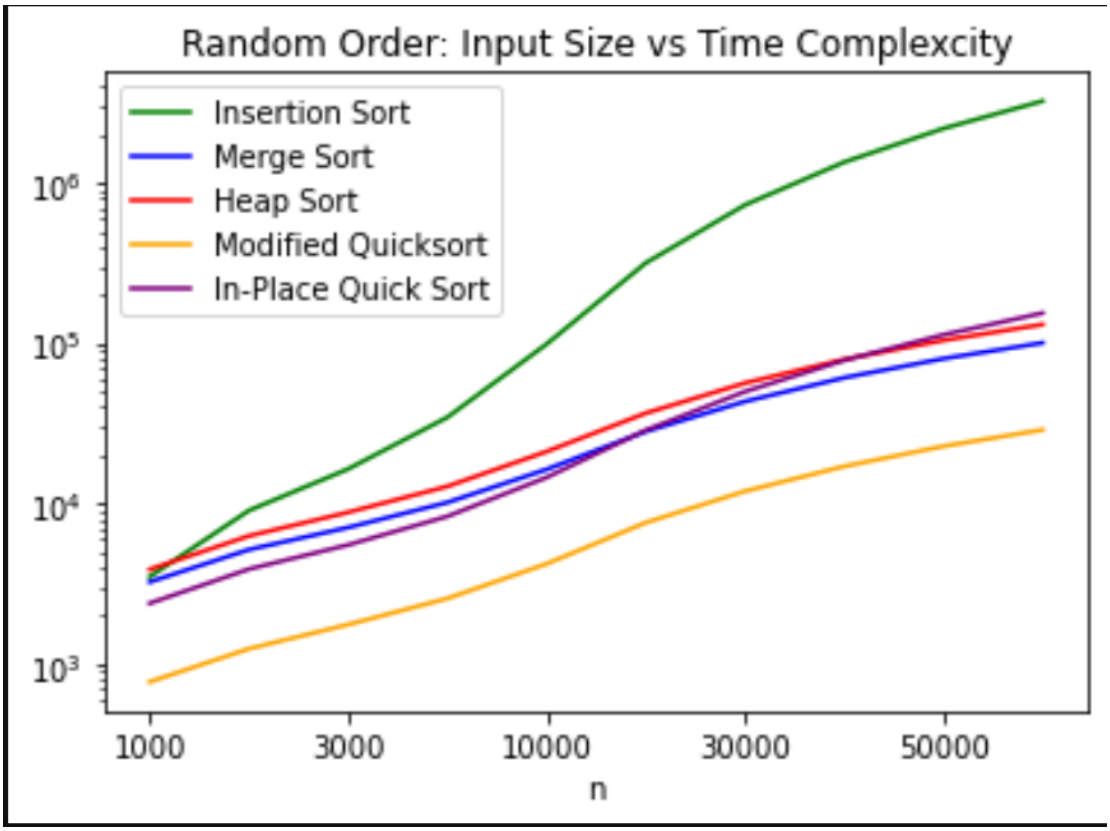

Figure2:Random order plotted results.
Figure 2 shows the results from the sorting algorithms with input of random order elements, and we see what we expected with insertion having the worst performance do to it's O(n^2) performance. Insertion sort is significantly worse than all other sorting algorithm and it's very clear that the time grows exponentially the larger the input size is while the other sorting algorithms have a less abrupt increase in performance.

n	Insertion Sort	Merge Sort	Heap Sort	In-Place Quick Sort	Modified Quicksort
1000	167.2001	3220.6415	3910.6793	2554.3501	721.9668
2000	255.9334	5094.63735	6337.9438	3976.8604	1158.225
3000	341.945867	7042.5735	8905.5348	5588.998567	1648.416733
5000	476.366675	10116.1301	12948.10728	8424.7427	2400.706325
10000	730.75418	16237.20906	21284.04496	14685.01834	3949.49756
20000	1178.985417	27987.89782	36876.94093	28873.85693	7219.391
30000	1734.778571	43027.988	56673.0065	50353.9464	11433.36549
40000	2358.664575	60400.21814	79346.50935	78462.92654	16393.22343
50000	3029.710189	79329.64297	104059.5764	113310.4287	21956.48843
60000	3746.09583	99432.18283	130476.5587	155145.8825	27853.115

Figure 3: Sorted order raw results.

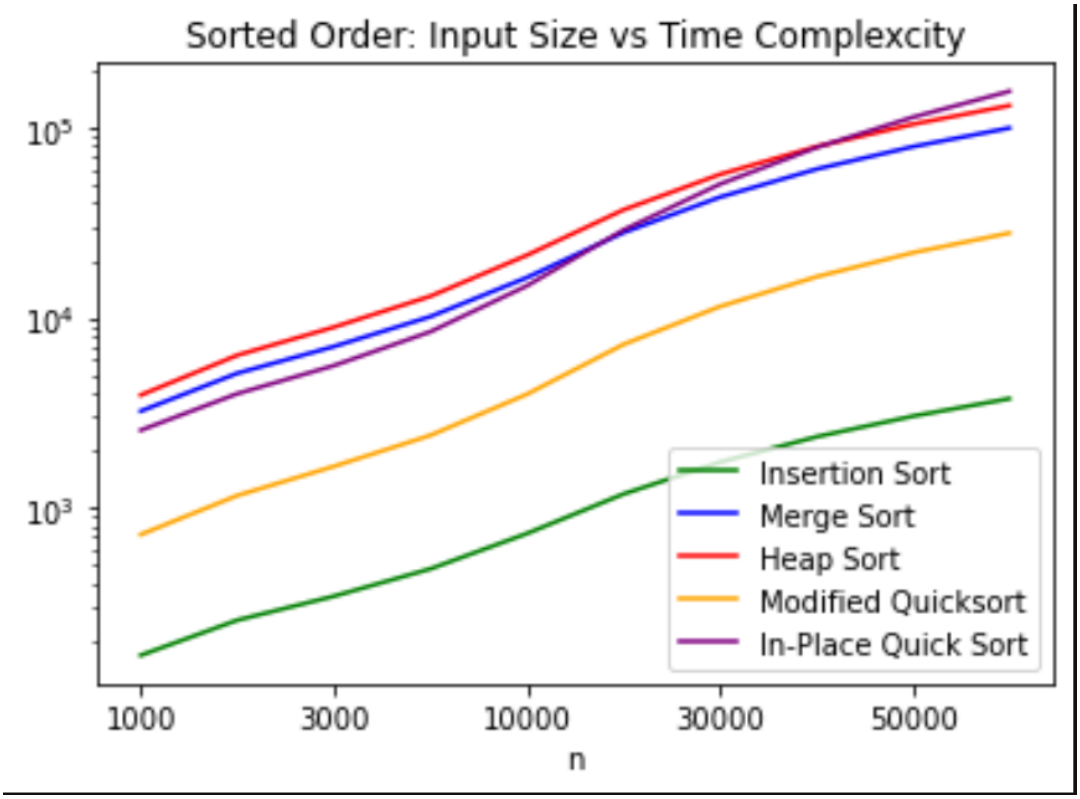

Figure 4: Sorted order plotted results.

Figure 4 shows the results from the analysis conducted using and input of sorted elements. What stands out from this is that insertion sort is now the best performing algorithm. The complexity for insertion sort with sorted input is O(n), this is due to the fact that the comparator does not need to be checked for every other element when the element is already in order.

n	Insertion Sort	Merge Sort	Heap Sort	In-Place Quick Sort	Modified Quicksort
1000	6.74E+03	3222.8956	3896.8957	2385.879	720.7877
2000	1.72E+04	5088.18115	6288.03125	3934.004	1152.6022
3000	3.24E+04	7043.493033	8813.280567	5581.655433	1644.4584
5000	6.82E+04	10187.6656	12851.75938	8391.9384	2415.986525
10000	1.96E+05	16362.97912	21130.04996	14677.74322	3982.97588
20000	6.35E+05	28019.65618	36610.99235	28823.87978	7267.083367
30000	1.45E+06	43039.17377	56364.25597	49905.85947	11498.00359
40000	2.69E+06	60489.7604	79258.40624	77940.4192	16458.24323
50000	4.40E+06	79510.23333	103964.0315	113028.7555	22035.33933
60000	6.52E+06	99856.91209	130278.6958	154260.9441	27955.31709

Figure 5: Reverse sorted raw results.

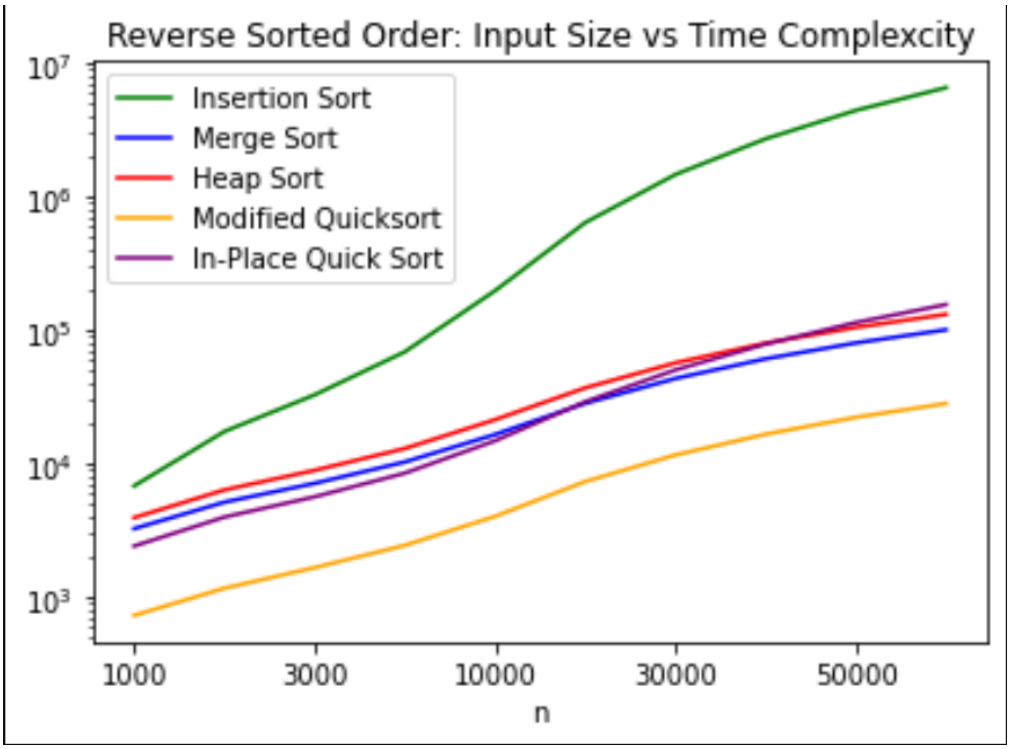

Figure 6: Reverse order plotted results.

In figure 6 we can see that once again insertion sort is the worst performing because of its O(n^2) time complexity. What stands out here is that modified quicksort has the best performance and is better than in-place quicksort due to its improvement over it.

Code

Insertion sort:

```
def insertionSort(A, l, r):

    for i in range(l, r):

        # Store comparator
        c = A[i]

        # While current value being checked is larger than comparator c
        # one position to the right and c
        j = i-1
        while j>= 0 and c < A[j]:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = c
```

Merge sort:

```
# Merge function
def merge(A, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # Create L and R arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to arrays L and R
    for i in range(0, n1):
        L[i] = A[l + i]

    for j in range(0, n2):
        R[j] = A[m + 1 + j]

    #Indices
    i = 0
    j = 0
    k = l

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        A[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        A[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(numbersList, p, r):
    if p < r:
        q = (p+r)//2
        mergeSort(numbersList, p, q)
        mergeSort(numbersList, q+1, r)
        merge(numbersList, p, q, r)
```

Heap sort:

```
def heapify(A, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and A[i] < A[l]:
        largest = l

    if r < n and A[largest] < A[r]:
        largest = r

 # Change root as requred
    if largest != i:
        (A[i], A[largest]) = (A[largest], A[i])
        heapify(A, n, largest)
```

```
def heapSort(A):
    n = len(A)

    for i in range(n // 2 - 1, -1, -1):
        heapify(A, n, i)

    for i in range(n - 1, 0, -1):
        (A[i], A[0]) = (A[0], A[i])
        heapify(A, i, 0)
```

In-place quicksort:

```python
#Partion function
def inPlacePartition(A, s, e, p):

    if not (s <= p <= e):
        raise ValueError('pivot error')

    A[s], A[p] = A[p], A[s]
    pivot = A[s]
    i = s + 1
    j = s + 1

    while j <= e:
        if A[j] <= pivot:
            A[j], A[i] = A[i], A[j]
            i += 1
        j += 1

    A[s], A[i - 1] = A[i - 1], A[s]
    return i - 1

# In place quick sort function
def inPlaceQuickSort(A, l=0, r=None):

    if r is None:
        r = len(A) - 1

    if r - l < 1:
        return

    idx_pivot = random.randint(l, r)
    i = inPlacePartition(A, l, r, idx_pivot)

    inPlaceQuickSort(A, l, i - 1)
    inPlaceQuickSort(A, i + 1, r)
```

Modified quicksort:

```
#Median of three function

def median_of_three(a, b,c):
    if ((b <= a and a <= c) or (c <= a and a <= b)):
        return a
    if ((a <= b and b <= c) or (c <= b and b <= a)):
        return b
    return c

def getPivot(A, left, right):
    mid = (left + right) // 2;
    return median_of_three(A[left], A[mid], A[right]);
```

```
def partition(arr,low,high,pivot):
    i = low-1
    j = high+1
    while 1:
        while 1:
            i+=1
            if(arr[i]>=pivot):
                break
        while 1:
            j-=1
            if(arr[j]<=pivot):
                break
        if i>=j:
            return j
        arr[i],arr[j] = arr[j],arr[i]
```

```
def swapPositions(list, pos1, pos2):

    list[pos1], list[pos2] = list[pos2], list[pos1]
```

```
def modifiedQuickSort(A, left, right):
    if (left + 10 <= right):
        pivot = getPivot(A, left, right)
        i = left
        j = right

        while (True):
            while (A[i] < pivot):
                i += 1
            while (pivot < A[j]):
                j -= 1
            if (i < j):
                swapPositions(A,i, j)
                i += 1
                j -= 1
            else:
                break

        modifiedQuickSort(A,left, i - 1)
        modifiedQuickSort(A,j + 1, right)
    else:
        insertionSort(A, left, right)
```