Abstract:

Project #1 requires us to implement various sorting and searching algorithms. Amongst these implementations, they will result in different running time. Some implementations may cause the algorithm to run faster/slower and perform differently according to the data that is being inputted. We are required to analyze the different cases of these algorithms and find the Big-O analysis of each. We look for the best case, the worst case, and the average case to see how these algorithms perform and to see if they are efficient for n input size they are given.

Sequential search was implemented in a way where if the list was stored as an array, we start at the first element of the array and compare it to the target. If the result concluded in that it matched it will return that index. If the target was not found, it will go to the next element and continue on until the element we are looking for is found. If at the end of the list, the element was not found the method would return -1 stating that it has not been found. We should come with results:

Worst case - O(n)

Average case - O(n)

Best case - O(1)

In the worst case, we would be looking through all the list and find the element in the end of the list or not even find it at all. In the best case, we might find the element in the beginning of the array if we happen to be lucky.

Binary search was implemented so that it would accept an array, target, low and high. The binary search will look at the middle element of the list. If the middle element is the target, then it will return the middle element. If the element is greater than the middle element, it

will look in the direction to the right where elements are greater. The same goes for when it is less but they will be looking in the direction to the left. We should come up with results as:

Worst case - O(log n)

Average case - O(log n)

Best case - O(1)

As the sequential search, the worst case is when it does not find the element in the list. The best case will occur when the element is found as the first element.

Selection sort was implemented so that two arrays were created, array[i] and array[j]. One would be empty and the other would be a whole array. In each step the algorithm will find the least element in the unsorted array and add it to the empty array. It will keep running until the unsorted array is empty. We should come up with results as:

Worst case - O(n^2)

Average case - O(n^2)

Best case - O(n^2)

The worst case will be when the array is in descending order so it will take n times to compare and then n times again to place it in the right order ascending. The best case is also O(n^2) because we always iterate forward n and swap with the smallest element.

Insertion sort was implemented so that it consumes an element finds the location that it belongs to and inserts the element there. Then the other elements would be shifted to the right.

We should get results as:

Worst case - $O(n^2)$

Average case - $O(n^2)$

Best case - O(n)

As stated with the selection sort, the worst case is when the array is sorted in a reverse order. The best case input is when an array is already sorted

Merge sort is a divide and conquer algorithm and was implemented so that it is recursively splitting the array into two parts from the midpoint. Note that if elements are odd then the right half would have an extra one. Once they are split into two arrays, they are compared to one another and finding the lesser element and merging the two adjacent lists.

We should get results as:

Worst case - O(n log n)

Average case - O(n log n)

Best case - O(n)

The worst case occurs when the input is halved in recursion as it needs to keep sorting the halves and them merge them into one sorted output. As for best case, if the arrays are sorted we only need to go through it n number of iterations.

Quick sort is also a divide and conquer algorithm and was implemented to divide the large array into two smaller sub-arrays with the low elements and high elements. It would pick an

element called the pivot and reorder the array so that all the elements lesser than the pivot would come before the pivot and all the elements that are greater would come after the pivot. It will recursively call the steps until there are no more elements to compare to.

We should get results as:

Worst case - $O(n^2)$

Average case - O(n log n)

Best case - O(n)

Getting the pivot equal to the lowest number or highest number will result in the worst case which would split the elements lower/higher onto another side and then recursively sort the two sub group. Best case is when the elements are already sorted.

Analysis:

Sequential Search:

$$\begin{cases} T(n-1)+a & \text{if } n > 1 \\ b & \text{if } n = 1 \end{cases}$$

T(n) = a(n - 1) + b = an + (b - a) = O(n)

Upper Bound:
Lower Bound:

Binary Search

Initial condition: T(1) = O(1) = 1
Recurrence relation: T(N) = T(N/2) + 1
T(N) = T(N/4) + 2
T(N) = T(N/8) + 3
T(N) = $T(N/2^{k)}$ + k
Set $N \leq 2^m$
$T(N) \leq T(2^m/2^k) + k$

Let k = m

$T(N) \leq T(2^m/2^m) + m = T(1) + m = 1 + m = O(m)$
If $N = 2^m$, then $m = \log N$
$T(N) = O(\log N)$

Selection Sort

Initial condition:
1) T(0) = a
2) T(n) = T(n - 1) + n + c if n > 0

T(n) = [T(n - 2) + (n - 1) + c] + n + c = T(n - 2) + (n - 1) +n + 2c
T(n) = [T(n - 3) + (n - 2) + c] + (n - 1) + n + 2c = T(n - 3) + (n - 2) + (n - 1) +n + 3c
T(n) = T(n - 4) + (n - 3) + (n - 2) + (n - 1) + n + 4c
T(n) = T(n - k) + (n - k + 1) + (n - k + 2) + ... + n + kc

When n - k = 0 --> k = n
T(n) = T(0) + 1 + 2 + ... + n + nc

$T(n) = a + \sum_{i=1}^{n} i + cn$

$$T(n) = a + \frac{1}{2}(n(n+1)) + c\,n$$

$$T(n) = a + c\,n + \frac{n^2}{2} + \frac{n}{2}$$

$T(N) = O(n^2)$

Upper Bound: O(n) for some element: min, max
Lower Bound: Ω (n) must examine every element

Insertion Sort

Initial condition
1) T(1) = 1 n = 1
2) T(n - 1) + n n > 1

T(n) = T(n - 1) + n
T(n) = T(n - 2) + (n - 1) + n
T(n) = T(n - 3) + (n - 2) + (n - 1) + n

$T(n) = \sum_{i=1}^{n} i$

$T(N) = O(n^2)$

Upper Bound: $O(n^2)$

Lower Bound: $\Omega(n^2)$

Merge Sort

Initial condition:
1) $T(1) = 1 \qquad n = 1$
2) $2T(n/2) + \Theta(n) \quad n > 1$

$n = 2^k$
$T(n) = \log(n) + 1$

$T(1) = 1$
$T(2^{k+1}) = T(2^k) + 1$
$T(2^{k+1}) = \log(2^k) + 1 + 1$
$T(2^{k+1}) = k + 2$
$T(2^{k+1}) = \log(2^{k+1}) + 1$
$T(N) = O(n \log n)$

Upper Bound: $O(n \log n)$
Lower Bound: $\Omega(n \log n)$

Quick Sort

Best case when the pivot is the middle:
$T(N) = 2T(N/2) + cN$
$T(N) / N = T(N/2) / (N/2) + c$
$T(N/2) / (N/2) = T(N/4) / (N/4) + c$
$T(N/4) / (N/4) = T(N/8) / (N/8) + c$
Adding all the terms

$T(N) / N = T(1) + cLogN = 1 + cLogN$
$T(N) = N + NcLogN$
$T(N) = O(NlogN)$

Worst case when the pivot is the smallest element:
Base Case: $T(1) = 1$
$T(N) = T(N - 1) + cN, N > 1$
$T(N) = T(N - 2) + (N - 1) + N$
$T(N) = T(N - 3) + (N - 2) + (N - 1) + N$
$T(N) = T(1) + 2 + 3 + ... + (N - 1) + N$
$T(N) = \sum_{i=1}^{n} i$
$T(N) = O(n^2)$

Upper Bound: $O(n^2)$
Lower Bound: $\Omega(n)$

Test Case:

The test cases for each algorithm were very similar. For the search algorithms (sequential and binary) to find the worst case values, I made the elements either in the end of the list or got rid of them, so the algorithm would iterate towards the end. For the best case, I made it so that the elements were first in the list so when the it would iterate it would just find the first value and stop. For the sorts worst case, I made elements in the array reversed, so that it would go through all the iterations to compare and sort the values. Quick sort was different in the fact that the pivot point had to be a value of the smallest element as it is a divide and conquer method. The best case for the sorts was when the arrays were already sorted out and the algorithms would just iterate over them to see if they were in order.

Experimental Results:

Sequential Search - Worst Case

Size (n)	Time (nano seconds)
256	11879
512	19251
1024	27852
2048	55705
4096	109363
8192	209304
16384	453425
32768	968701
65536	1937401
131072	3369766

Sequential Search - Average Case

Size (n)	Time (nano seconds)
256	9420

512	11469
1024	15975
2048	49152
4096	105267
8192	69222
16384	57344
32768	55295
65536	1098543
131072	3462335

Binary Search - Worst Case

Size (n)	Time (nano seconds)
256	6144
512	7373
1024	6964
2048	11879
4096	12288
8192	9012
16384	15155
32768	43008
65536	41779
131072	52428

Binary Search - Average Case

Size (n)	Time (nano seconds)
256	6144
512	6554
1024	6963
2048	12288
4096	7373
8192	10240
16384	11469
32768	41779
65536	61031
131072	65023

Selection Sort - Worst Case

Size (n)	Time (seconds)
256	0.002
512	0.006

1024	0.007
2048	0.009
4096	0.019
8192	0.062
16384	0.222
32768	0.764
65536	3.412
131072	13.156

Selection Sort - Average Case

Size (n)	Time (seconds)
256	0.002
512	0.006
1024	0.006
2048	0.009
4096	0.019
8192	0.058
16384	0.212
32768	0.836
65536	3.305
131072	13.103

Insertion Sort - Worst Case

Size (n)	Time (seconds)
256	0.002
512	0.005
1024	0.009
2048	0.013
4096	0.031
8192	0.106
16384	0.365
32768	1.465
65536	5.792
131072	22.938

Insertion Sort - Average Case

Size (n)	Time (seconds)
256	0

512	0.003
1024	0.004
2048	0.007
4096	0.015
8192	0.049
16384	0.189
32768	0.745
65536	2.936
131072	11.525

Merge Sort - Worst Case

Size (n)	Time (seconds)
256	0.001
512	0.001
1024	0.001
2048	0.003
4096	0.008
8192	0.018
16384	0.021
32768	0.022
65536	0.030
131072	0.043

Merge Sort - Average Case

Size (n)	Time (seconds)
256	0
512	0
1024	0.001
2048	0.003
4096	0.008
8192	0.020
16384	0.020
32768	0.020
65536	0.030
131072	0.043

Quick Sort - Worst Case

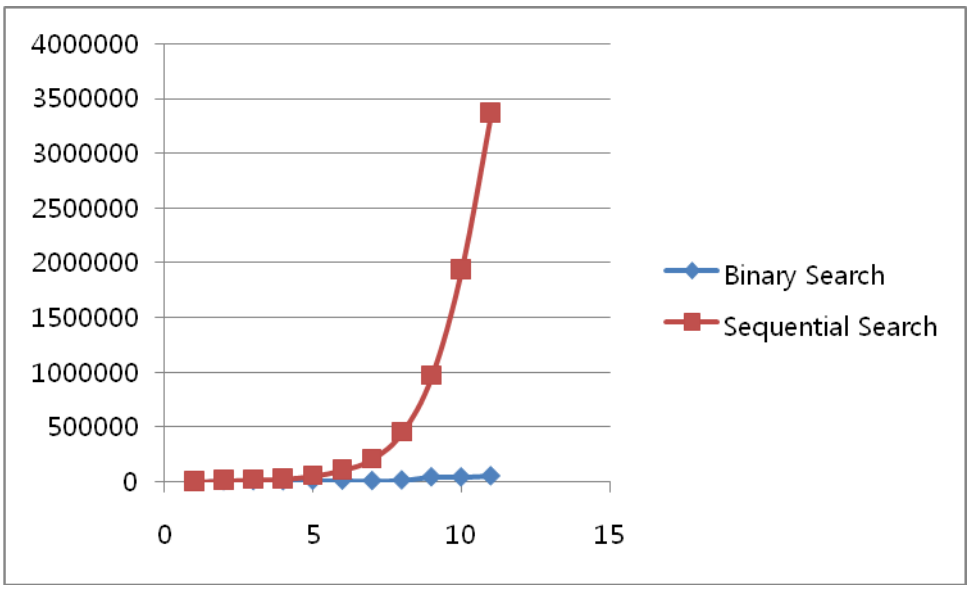
Size (n)	Time (seconds)
256	0
512	0
1024	0.001

2048	0.001
4096	0.006
8192	0.007
16384	0.009
32768	0.025
65536	0.033
131072	0.039

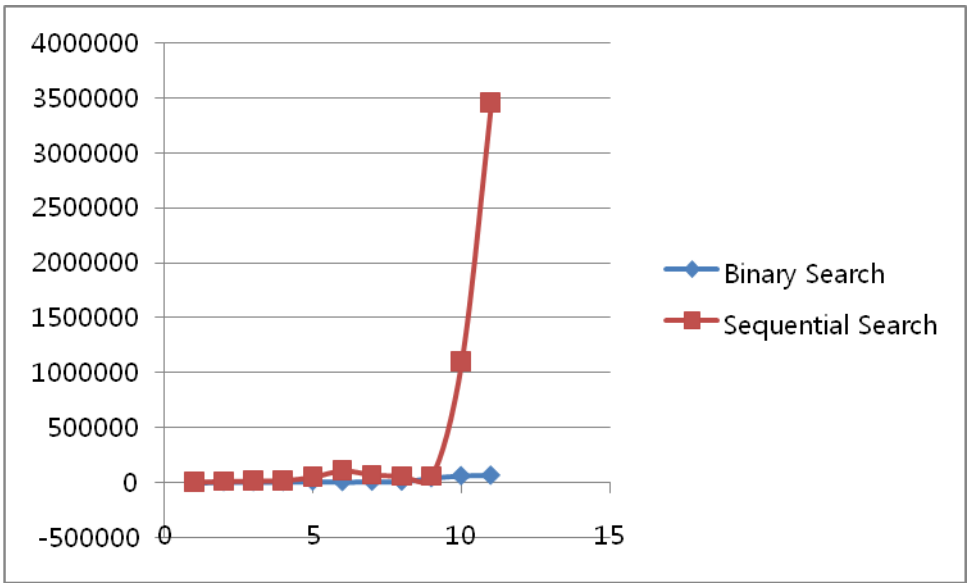
Quick Sort - Average Case

Size (n)	Time (seconds)
256	0
512	0
1024	0.001
2048	0.001
4096	0.005
8192	0.005
16384	0.007
32768	0.025
65536	0.031
131072	0.037

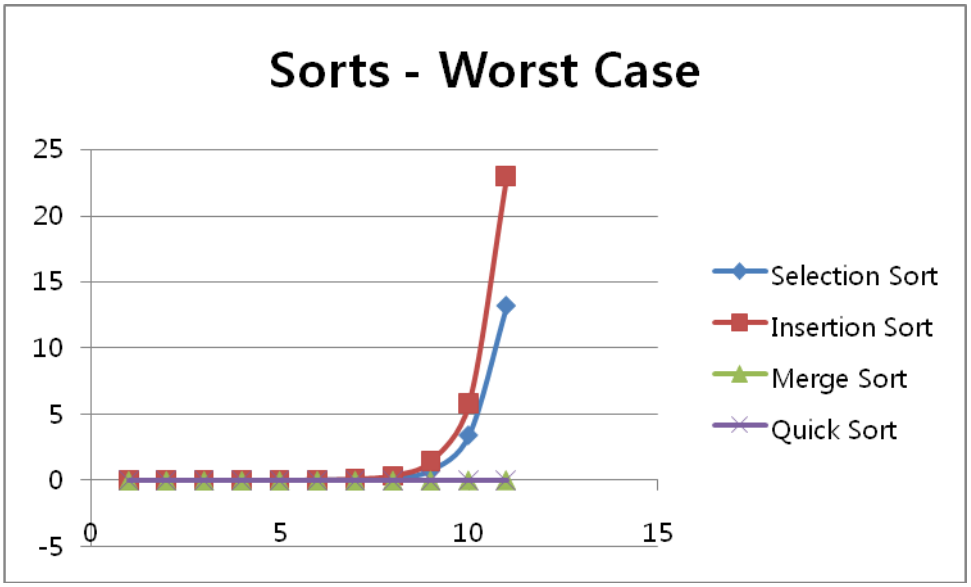
Sequential vs Binary Search (Worst Case) (Time (nano-second) vs. Size (n))



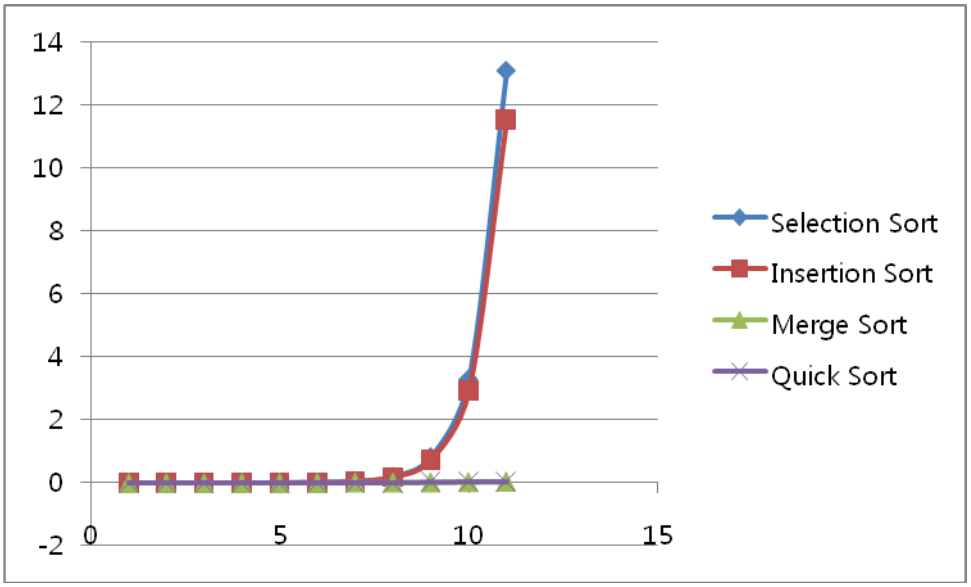
Sequential vs Binary Search (Average Case) (Time (nano-second) vs. Size (n))



Selection, Insertion, Merge, Quick Sort - Worst Case (Time (s) vs. Size (n))



Selection, Insertion, Merge, Quick Sort - Average Case (Time (s) vs. Size (n))



Conclusion:

There were difficulties trying to test the different run times with the searches. I had trouble getting an exact time especially with the binary search because it kept printing as 0 seconds when done using the System.currentMillis() timer. I had to switch it to nano seconds. Overall, I think it matched my experimental results and did exactly what I had planned it to do.