Benjamin Carlson
9/20/20
CS 253

Problem 1:

	Best Case	**Worst Case**	**Average Case**		
	1,2,3,4,5,6,7,8,9,10	10,9,8,7,6,5,4,3,2,1	1,3,2,5,47,9,6,8,10		
Selection Sort	9 Exchanges 45 Comparisons	9 Exchanges 45 Comparisons	9 Exchanges 45 Comparisons		
Bubble Sort	0 Exchanges 45 Comparisons	45 Exchanges 45 Comparisons	5 Exchanges 45 Comparisons		
Insertion Sort	0 Exchanges 9 Comparisons	45 Exchanges 54 Comparisons	5 Exchanges 14 Comparisons		
	1,2,3,...,1999,2000	2000,1999,...,3,2,1	Random ints 1-2000 (1)	Random ints 1-2000 (2)	Random ints 1-2000 (3)
Selection Sort	1999 Exchanges 1999000 Comparisons	1999 Exchanges 1999000 Comparisons	1999 Exchanges 1999000 Comparisons	1999 Exchanges 1999000 Comparisons	1999 Exchanges 1999000 Comparisons
Bubble Sort	0 Exchanges 1999000 Comparisons	1999000 Exchanges 1999000 Comparisons	982053 Exchanges 1999000 Comparisons	1028481 Exchanges 1999000 Comparisons	992365 Exchanges 1999000 Comparisons
Insertion Sort	0 Exchanges 1999 Comparisons	1999000 Exchanges 2000999 Comparisons	982053 Exchanges 984052 Comparisons	1028481 Exchanges 1030480 Comparisons	992365 Exchanges 994364 Comparisons

Selection Sort Explanation: In the best case scenario (1,2,3,4,5,6,7,8,9,10) we will start with the first element. We then iterate through the entire list to search for the smallest. This will result in n-1 comparisons, in this case 9. 1 will remain in its place with 1 exchange (we exchange one with itself). Next, we start with 2 and do the same thing. In this case there will be 8 comparisons and one more exchange, with 2 staying in its place. More broadly, during the first iteration of the outer loop there will be n-1 comparisons, during the second iteration of the outer loop there will be n-2 and so on. Each loop there will be one exchange (n-1 total exchanges). In this case 9 exchanges. Selection sort is not sensitive to the input because even if the array is sorted, it still compares each element to each other. Because of this, selection sort will always be quadratic - O(N^2) for the best, worst, and average case. We see the explanation above hold true when we run this sort with 2000 items. In the best case scenario (1,2,3,...,1999,2000) we get 1999 Exchanges and 1999000 comparisons. We get the same for the worst and average case. This is to be expected because we have n-1 total exchanges, in this case 1999, and n-1 comparisons for the first iteration, n-2 for the next and so on which is 1999000 (1999+1998+1997+...+1).

Bubble Sort Explanation: In the worst case scenario (10,9,8,7,6,5,4,3,2,1) the 10 will be pushed into its proper place using 9 comparisons and 9 exchanges. Next, the 9 will be pushed into its proper place using 8 comparisons and 8 exchanges. This will continue until the array is sorted. The total number of comparisons and exchanges will be 9 + 8 + 7 + … + 1 or 9!. The total number of comparisons and exchanges will be 45. In the best case scenario (1,2,3,4,5,6,7,8,9,10) we will loop through and compare each item to the next via the inner loop but there will be no exchanges. Therefore, there will still be 45 comparisons and no exchanges. As we can see, bubble sort is sensitive to the input. In this best case scenario the runtime will be O(N). We can conclude that bubble sort is O(N^2). When we run bubble sort with 2000 items we see the results we would expect. In the best case scenario there are 0 exchanges and in the worst case there are 1999000 (1999+1998+1997+...+1). There are the same number of comparisons: 1999000 just like we saw in the 10 int array example. Additionally, the average cases are 982053, 1028481, and 992365 exchanges which all fall into the expected theoretical range.

Insertion Sort Explanation: In the best case scenario (1,2,3,4,5,6,7,8,9,10) we start by comparing 2 to 1. This is one comparison but no exchange. Next we compare 3 to 2. Again, one comparison but no exchange. The outer loop iterates n-1 times and the inner loop fails right away (no exchanges or comparisons) so the best case runtime is O(N). Insertion sort is linear with respect to the number of comparisons in the best case. We can conclude that for the best case there are 0 exchanges and 9 comparisons. In the worst case (10,9,8,7,6,5,4,3,2,1), we will have n-1 exchanges in the first iteration, then n-2 and so on giving us a total of 45 exchanges. We will have the same number of comparisons but because I added a counter for comparisons in the outer loop, we will be over estimating the number of comparisons by n-1. That is why in the table for the worst case I have 54 comparisons. The same goes for the average case. Every time

we enter the inner while loop we are overestimating the comparisons by 1.We can conclude that insertion sort is O(N^2) in the average and worst case with respect to the number of exchanges. Insertion sort is also sensitive to the input. From the table, we can see that insertion sort is in most cases a better sort than bubble or selection. In the best case scenario (1,2,3,...,1999,2000) we get 0 exchanges and 1999 comparisons using the same logic used in the best case scenario insertion sort 10 int array. Similarly, using the same logic for the worst case scenario, we get 1999000 exchanges and 2000999 comparisons. The average cases all fall into the expected theoretical range. Keep in mind, we are overestimating the number of comparisons for the worst and average case.

Problem 2:

Recursive binary search:
Run time: O(log(N))
Explanation: My recursive factorial algorithm runs in logarithmic execution time - O(log(N)). In the first call, the entire array is passed into the method. On the next call, the array is split in half. This process of dividing by 2 repeats until there is only one item in the array left. At this time, the item we are searching for is either found (the last item) or not found (not the last item). Therefore, because we are returning half of the array each time we recursively call the method, the execution time is logarithmic time at its worst case.

Factorial:
Run time: O(N)
Explanation: My recursive factorial algorithm runs in linear execution time - O(N). The recursive equation for factorial is n! = n * (n-1) * (n-2) * … * 1. The recursive definition can be further written as n! = n * (n-1)! Every time the method calls itself, it decrements the value of n by 1 until it reaches 0. This means the method is called recursively n times, making it run in linear execution time at its worst case.

Fibonacci:
Run time: O(2^N)
Explanation: My recursive fibonacci algorithm runs in exponential execution time. In the first call, n is passed into the method. Next we return 2 calls with n-1 and n-2 passed into the method. This continues until n == 1. This means the method is called recursively n^2 times, making it run in exponential execution time at its worst case, the worst running time.