Muangkong Yang

HW1 CSCD320

Question 1

1. Read the following code and answer questions.

```
public static void one(int a[], int j, int m)
{
	int i, temp;
	if(j < m) {
		for( i = j; i <=m; i ++ ){
			if( a[i] < a[j] ) {
				temp=a[i];
				a[i]=a[j];
				a[j]=temp;
			}
		}//end of for

		j++;
		one(a, j, m);
	}//end of outer if
}//end of one method
public static void main(String args[]) {
	int a[] = {2, 5, 1, 7, 9, 3, 6, 8};
	one(a, 0, a.length - 1);
	System.out.println(Arrays.toString(a));
}
```

(a)Summarize what task(s) the method ***one*** perform? What will the **main()** method print on the screen after execution? (10 points)

The task that method one will be performing is sorting an array in ascending order recursively. The main method will be printing the following array: {1, 2, 3, 5, 6, 7, 8, 9}.

(b) Determine the Growth Rate Function(GRF) for the method ***one***, please justify how did you get your GRF. As a separate task, then, you show the time complexity using Big-Oh notation. (40 points)

Inside method *one*, ignoring the variable declarations and if statements, as they are constants at the end, it consists a for loop that will run *n* times, specifically j to m times because of the `if(j < m)`. However each loop will increment j by 1. So it starts with n, then n-1, n-2 and so forth. Likewise, the recursive call will also run *n* times. We then get the following GRF: `n + (n-1) + (n-2) + … + 2 + 1`, for both the for loop and the recursive call. We can also write that as `(n(n+1) / 2)` or `((n^2 + n) / 2)`. Dropping the constants we get a time complexity of `O(n^2)`.

Question 2. Programming is required.

2b)Please analyze whether the time complexity of your algorithm is (log n). **If you cannot provide a *O(log n)* algorithm, you lose a lot of points.**

```
62   public static int quickSearch(int array[], int value)
63   {
64      int mid = (array.length / 2);
65      int tempSize = mid;
66
67      while(mid < array.length && tempSize > 0)
68      {
69         if(value == array[mid-1]) // deals with middle element
70            return mid - 1;
71         else if(mid == 1 && value < array[mid-1]) // deals with #'s less than array elements
72         {
73            return 0;
74         }
75         else if(value > array[mid-1] && value <= array[mid]) // returns the greater element's index when positioned **1) has to be above 2)
76         {
77            return mid;
78         }
79         else if(value < array[mid-1]) // left half of array
80         {
81            // NOTE**This line of code will imitate a O(logn) time complexity by dividing the array by half every time we execute this part
82            mid = mid - (tempSize / 2);
83            tempSize = tempSize - (tempSize / 2);
84         }
85         else if(value > array[mid-1] && mid == array.length-1) // deals with #'s greater than array elements **2) has to e below 1)
86         {
87            return -1;
88         }
89         else //if(value > array[mid-1]) // right half of array
90         {
91            // NOTE**This line of code will imitate a O(logn) time complexity by dividing the array by half every time we execute this part
92            mid = mid + (tempSize / 2);
93            tempSize = tempSize - (tempSize / 2);
94         }
95      }// end while
96
97      return mid;
98   }// end quickSearch
99
100  |
```

My quickSearch method has been provided above in a screenshot. The while loop consists of if, else/if, and else statements in which only one statement occurs. There are only 2 statements that continue the loop and I've noted them in my screenshot. The others are returning the correct index. The array begins with 8 elements. We start off placing the index right at the half way point. We use zero-indexing (0-7) so 8/2 = 3. The while loop will only loop if the index hasn't been found. Assuming the worst case scenario, we then divide the index by half again, either moving to the left half of the array or right depending on the value passed in, which is 3/2 = 1 for left, or 5 for right. We will repeat until we reach both ends of the array.
Since we are halving our index every loop, our GRF is: 1 * 2 * 2 * 2 = 8. Simplifying we get 2^x = 8. 8 can be arbitrary, so we can replace it with *n*, so we get 2^x = n. Solving for x we get x = logn. Thus, the time complexity for my method is O(logn).