Parker Gray

CSC340 HW #6.2

2.1 Complexity analysis: Represent the time complexity of the following recursive algorithm, T(n), as a recurrence equation: **(5 points)**

```
int pow_7( int n ){
  if ( n==1)
        return 7;
 if ( n > 1)
        return ( 7 * pow_7( n-1 ) );
}
```

Line No.	Time taken to run this line of code.	Total number of times needed to run this line of code.
1 *(if n==1)*	C1	**1 time**, simple variable check.
2	C2	**1 time**.
3	C3	**1 time**.
4 *(return(7*pow_7(n-1));)*	C4	$\boldsymbol{n-1}$ **times**. To reach the base case of n == 1, pow_7 needs to be ran approximately n-1 times. Eg, if n is equal to 3, pow_7 must be ran twice before the base case of == 1 is found.
Total time needed to finish this loop:	$\boldsymbol{C1 + C2 + C3 + C4}$ **if n = 1.** *(the base case)* $\boldsymbol{C1 + C2 + C3 + T(n-1)}$ **if n > 1**	

2.2 Complexity analysis: analyze the time complexity of the Top-Down implementation of the MergeSort algorithm on the following wikipedia webpage
http://en.wikipedia.org/wiki/Merge_sort

You are required to represent the time complexity of TopDownMerge() as a polynomial function of the input size. Then represent the time complexity of TopDownSplitMerge(A[], iBegin, iEnd, B[]) as a recurrence equation. You don't need to solve this equation. **(15 points)**

```
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}
```

Line No.	Time taken to run this line of code.	Total number of times needed to run this line of code.
1 *(i = iBegin, j = iMiddle;)*	C1	$\mathbf{1}\ \boldsymbol{time}$, simple variable initialization
2 *(for (k = iBegin; k < iEnd; k++)*	C2	$\boldsymbol{iEnd\ times}$.
3	C3	$\boldsymbol{iEnd}(\boldsymbol{iMiddle})\ \boldsymbol{times\ OR\ iEnd}(\boldsymbol{j})$. This if statement runs exactly as many times as it takes for one of its conditions to become invalid. Ie, it will run until i = iMiddle, or until j is less than iEnd, and until A[i] is greater than A[j]. Therefore, the number of times needed to run this code needs to incorporate either possibility- either i becomes equal to iMiddle, or both of the conditions in the parens need to be false. The $\boldsymbol{iEnd}$ at the beginning of each possibility is to take into account the for statement preceding this if statement. $\boldsymbol{iEnd}(\boldsymbol{iMiddle})$ is derived from the fact that i will eventually be incremented to pass iMiddle, and $\boldsymbol{iEnd}(\boldsymbol{j})$ is derived from the fact that the value of A[i] will eventually be greater than A[j] assuming a not completely sorted array.
4	C4	$\boldsymbol{iEnd}(\boldsymbol{iMiddle})\ \boldsymbol{times\ OR\ iEnd}(\boldsymbol{j})\boldsymbol{times}$. Has the same amount of run time as the previous statement.
5	C5	$\boldsymbol{iEnd}(\boldsymbol{iMiddle})\ \boldsymbol{times\ OR\ iEnd}(\boldsymbol{j})\boldsymbol{times}$. Has the same amount of run time as the statement 3.
6	C6	$\boldsymbol{iEnd} - \mathbf{1}\ \boldsymbol{times\ OR\ iEnd}(???)$. The else statement does not run until one of the conditions in the if statement no longer is true. Once that occurs, the else statement runs until the for loop reaches iEnd, or until its iteration on j makes A[i] <= A[j] true again. I have to say, I'm not quite certain how to represent this latter case, ie the case where i is still less than iMiddle and j's iteration makes A[i] <= A[j] true. Assuming a not completely sorted array, the chance that j's iteration will make A[i] <= A[j] true is totally random. Who knows what the next number might be! As a result, I can only list the first possibility related to the for loop on line 2 with certainty.
7	C7	$\boldsymbol{iEnd} - \mathbf{1}\ \boldsymbol{times}$

8 (*j = j +1)*	C8	$\boldsymbol{iEnd - 1\ times}$
Total time needed to finish this loop:	$\boldsymbol{C1 + C2(iEnd) + C3(iEnd(iMiddle)\ or\ iEnd(j))}$ $\boldsymbol{+ C4(iEnd(iMiddle)\ or\ iEnd(j))}$ $\boldsymbol{+ C5(iEnd(iMiddle)\ or\ iEnd(j))}$ $\boldsymbol{+ C6(iEnd - 1\ times) + C7(iEnd - 1\ times)}$ $\boldsymbol{+ C8(iEnd - 1\ times)}$	

```
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if(iEnd - iBegin < 2)                       // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

//  Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is            B[ iBegin:iEnd-1   ].
```

Line no.	Time taken to run this line of code.	Total number of times needed to run this line of code.
1 *(if(iEnd – iBegin < 2))*	C1	$\boldsymbol{(iEnd - iBegin = 1)}$ **times.** This line does not return anything until iMiddle has become progressively small enough to meet the condition of the if statement. For iMiddle to reach this this condition, the function must be ran until the if statement is true.
2	C2	$\boldsymbol{(iEnd - iBegin = 1)}$ **times.**
3	C3	$\boldsymbol{(iEnd - iBegin = 1)}$ **times.**
4	C4	$\frac{\boldsymbol{n}}{\boldsymbol{2}}$ **times.** This line and the line following it split the array into two sides, the 'left run' and the 'right run'. It therefore splits an array of n objects into ½ of its initial size, and continues to do so until the arrays are considered

		sorted. Both line 4 and line 5 perform this $\frac{n}{2}$ operation, so the contribution appears as $2T\left(\frac{n}{2}\right)$ in the final calculation.
5	C5	$\frac{n}{2}$ **times.**
6 (*TopDownMerge(B, iBegin, iMiddle, iEnd, A))*	C6	**n times.** TopDownMerge runs until it sorts all elements in array of size n.
Total time needed to finish this loop:	= $C1(iEnd - iBegin) + C2(iEnd - iBegin) + C3(iEnd - iBegin) + 2T\left(\frac{n}{2}\right) + T(n)$ (for best case) $C1(iEnd - iBegin) + C2(iEnd - iBegin) + C3(iEnd - iBegin) + 2T\left(\frac{n}{2}\right) + T(n)$ (I don't know how to calculate the worst case, though Wikipedia claims the worst case's T(n) is $(n\lceil lg\ n\rceil - 2^{\lceil lg\ n\rceil} + 1)$.)	