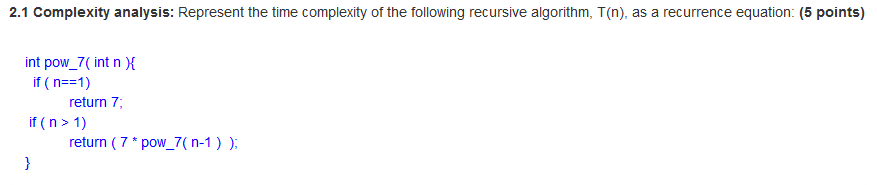
Parker Gray

CSC340 HW #6.2



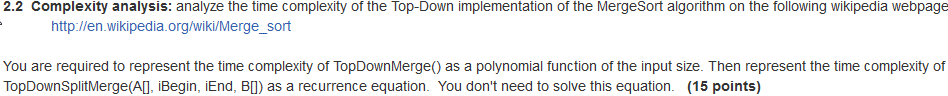
|  |  |  |
| --- | --- | --- |
| Line No. | Time taken to run this line of code. | Total number of times needed to run this line of code. |
| 1 *(if n==1)* | C1 | **1 time**, simple variable check. |
| 2 | C2 | **1 time**. |
| 3 | C3 | **1 time**. |
| 4 *(return(7\*pow\_7(n-1) ); )* | C4 | **times**. 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: | **if n = 1.** *(the base case)*  **if n > 1** | |

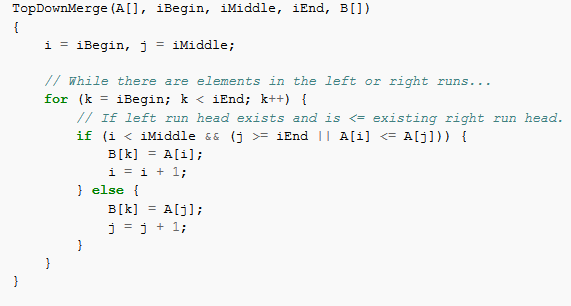
**A recurrence equation is not the typical time complexity table. It does not show the time taken to run a line line by line. It simply shows the different cases of the recursive algorithm, ie *if n = 0, 1, then the base case is used, and the algorithm completes in Csub0 amount of time, a constant amount of time.* *If n > 1, there are different cases depending on how large n is- ie, if n falls in a specific range, the recursive algorithm only needs to run one time. If n is greater than a certain value, then it needs to be run two times*, and so on.**

**This is typically represented as something like (partition algorithim) (C1 \* n + C0) + (ksmall algorithim) (T(n/2)) if n > 1 in a best case scenario. The worst case scenario would have a different T(n) value. Something like (C1 \* n + C0) + (T(n-1)).**

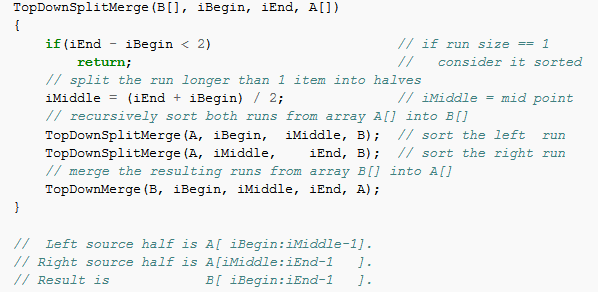
**Avg case: 70 30 split in where n ends up above or below pivot value has the following value: (C1\*n + C0) + T(.7n) || T(.3n) where the two parallel lines indicate or.**

**Next page contains second half of complexity analysis portion.**



****

|  |  |  |
| --- | --- | --- |
| 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 | , simple variable initialization |
| 2 *(for (k = iBegin; k < iEnd; k++)* | C2 |  |
| 3 | C3 | 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 at the beginning of each possibility is to take into account the for statement preceding this if statement. is derived from the fact that i will eventually be incremented to pass iMiddle, and 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 | Has the same amount of run time as the previous statement. |
| 5 | C5 | Has the same amount of run time as the statement 3. |
| 6 | C6 | 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 |  |
| 8 (*j = j +1)* | C8 |  |
| Total time needed to finish this loop: |  | |



The above function will run many times, until iEnd and iBegin are less than 2. Otherwise, the function repeatedly calls itself until using iMiddle as the new iBegin and iEnd points. iMiddle will progressively get smaller and smaller until it sorts both the beginning and ending of the array.

|  |  |  |
| --- | --- | --- |
| 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 | **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 | **times.** |
| 3 | C3 | **times.** |
| 4 | C4 | **times.** |
| 5 | C5 | **times.** |
| 6 (*TopDownMerge(B, iBegin, iMiddle, iEnd, A))* | C6 | **times.** |
| Total time needed to finish this loop: |  |  |

**Quick sort recurrence equation from a slide in the recursion ppt**

**Q sort**

**T(n) = C0 if n = 0, 1.**

**= C1 \* n + C0 + 2T(n/2) The /2 comes from the two resulting subproblems BC n > 1**

**= C1 \* n + C0 + T(n-1) WC n > 1**

**= C1 \* n + C0 + T(.8n) + T(.2n) Avg 80:20**

**For BC and Avg, the Big O is (n log n)**

**For WC, the Big O is n^2.**