Algorithm\_prob.1

1412867 이주영

1. Write a merge sort algorithm that sorts a list of n integers by dividing it into three sublists of about n/3 items. Analyze your algorithm, and give the results using the order notations.(big O, Omega, theta notations)

MERGESORT3 (low, high) {

//initial low is 1

//initial high is n

If (low < high) {

third <- floor(low+high/3);

call MERGESORT3(low,third);

call MERGESORT3(third+1, 2\*third);

call MERGESORT3(2\*third-1, high);

call MERGE(low, third, 2\*third);

call MERGE(low, 2\*third, high);

}

MERGE(low, mid, high)

{

h <- low //lower half index

i <- low //global s’s index

j <- mid+1 //upper half index

while (h <= mid && j <= high) {

if (s(h) <= s(j))

{A(i) <- s(h); h <- h+1;}

Else {A(i) <- s(j); j <- j+1; }

i <- i+1;

}

// when one of 2 parts’ elements are all in A, the rest’s elements can just go in A in order since it has already been sorted recursively.

If ( h > mid)

For (k=j;k<=high;k++)

{A(i) <- s(k); i<- i+1;}

Else{

For(k=h;k<=mid;k++)

{A(i) <- s(k); i <- i+1;}

}

}

In terms of time complexity, it would be same as the time complexity when we are dividing the problem in two halves every time, only the base of log changes. But when we divide array into two sub-arrays then two-way merge needs one comparison but 3-way merge needs 2 comparisons. Therefore, although by splitting array into 3 sub-arrays, we are decreasing fewer number of passes, we are actually increasing the cost of each pass by doing more number of comparisons.

Here are the steps of 3-way Merge Sort:

1) Merge Sort the first half of the list.

2) Merge Sort the second half of the list.

3) Merge Sort the third half of the list.

3) Merge all of them together.

Noticing that step 1 and step 2 and step3 are sorting problems also, but of size n/3, and that the last step runs in O(n) time, we get the following equation for T(n):

This is known as a recurrence relation since the function T(n) is defined in terms of another value of the function T. Now, let's see if we can try to figure out what T(n) is, just in terms of n, (for the time being, let's simplify O(n) to n):

T(n)=3T(n/3)+ O(n)

T(n) = 3T(n/3) + c\*n (c: constant)

T(n) = 3[3T(n/9)+c\*n/3] + c\*n

= 9T(n/9) + 2c\*n

= 9[3T(n/27)+c\*n/9] + 2c\*n

= 27T(n/27) + 3c\*n

…

T(n) = 3kT(n/3k) + kcn

Eventually, when applying this recurrence, we should stop. In particular, we can assume that T(1) = 1. Then, we can solve for T(n) directly by plugging in k = log3n. Also, we have T(n/3k) in our formula. So it would be nice if n = 3k. But this occurs when k = log3n. Plugging in the value for k we find:

T(n) = nT(1) + nlog3n

= O(nlog3n) = Ω (nlog3n) =  θ(nlog3n)