Algorithms

Quiz 1

Question 1:

a. Consider the modified binary search algorithm so that it splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write down the recurrence for this ternary search algorithm and find the asymptotic complexity of this algorithm?

Answer :

Since binary search tends to divide the elements into two subsets every iteration and recursively will do the same for the right subset of elements. In addition the division takes only one operation. Therefore, binary search complexity could be computed as follows:

$$T(n) = T(n/2) + 1 \qquad (1)$$

By analogy, we divide the elements instead of two subsets , we divide them into three subsets. Also, only two operations are required for the devision. Therefore, the complexity of the new problem could be computed as follows:

$$T(n) = T(n/3) + 2 \qquad (2)$$

Using the master theory, the overall complexity is $\theta(\log n)$

b. Consider another variation of the binary search algorithm so that it splits the input not only into two sets of almost equal sizes, but into two sets of sizes approximately one-third and two-thirds. Write down the recurrence for this search algorithm and find the asymptotic complexity of this algorithm?

Again, instead of dividing the elements into equal halves, we divide them into (n/3) and (2n/3). We need only one operation for such division. Therefore, the complexity of the algorithm will be based on the (2n/3) sub-elements. Therefore, we could compute the complexity as follows:

$$T(n) = T(2n/3) + 1 \qquad (3)$$

Using master theory, the complexity amazingly turns to be

$$T(n) = \text{is } \theta(\log n) \qquad (4)$$

Question 2:

Consider the following variation on Mergesort for large values of *n*. Instead of recursing until *n* is sufficiently small, recur at most a constant r times, and then use insertion sort to solve the 2^r resulting subproblems. What is the (asymptotic) running time of this variation as a function of *n*?

Answer:

The problem looks hard; however, by logical analysis, its trivial. So, let's dig into it.

The complexity of Merge sort in general is computed as follows:

$T(n) \leq 2T\left(\frac{n}{2}\right) + Cn$, where *Cn* is the merging cost ➔ O(nlogn)

In our modified problem, we recur at most a constant *r* times; then we start using insertion/selection sort to solve the 2^r resulting subproblems. The question now, what is the size of each (2^r) subproblem ?

The answer is simple each subproblem is of size $(n/ 2^r)$ since I divide *n* by 2 every time. E.g. if $r = 2$, therefore , I have left with $4\ (2^2)$ subproblems and each subproblem is of size $(n/4)= (n/2^2)$. Hum!!!

Okay, now how much it takes to solve 2^2 subproblems? Well Insertion/Selection sort takes $O(n^2)$. Therefore, it requires $(n/2^r)^2$ for each subproblem and $(n/ 2^r)^2 * 2^r$ for all of the (2^r) subproblems. Add to that the merging cost of the *r* levels, Right!.

Well, we forgot that *r* is constant; the (2^r) is also constant. What does this mean? It means that in terms of the worst, best, or average complexity, the constants do not have that much impact on the overall complexity. Then, the overall complexity is in a range of $\Omega(n^2)$. I can hear you saying, what about the merge operations? Well, can it be larger than n^2!

The previous case is not true in all cases as follows:

Using the original formula for the merge sort, we can get a general formula as follows:

$T(n) = 2T(n/2) + n$

$T(n) = 2T(2T(n/4)+n/2)+n$

$T(n) = 2(2(2T(n/8)+n/4)+n/2)+n$

...

$T(n)=2^r\ T(n/2^r) + nr$

Since $T(n/2^r) = n^2/(2^2r)$ [Selection sort]

Then,

$T(n) = n^2/(2^r) + nr$

For a shallow recursion ($r \sim 1$), the quadratic term becomes dominant and the merge sort cost becomes linear. For a deep recursion ($r \sim \log n$), the quadratic term reduces to $O(1)$ and the merge sort cost becomes $n \log n$.