String Processing Algorithms 2015 - Week 1 Exercises

Rodion Efremov

October 23, 2015

Exercise 1

The worst case time complexity of the standard quicksort algorithm is $\Omega(n^2)$, but by a suitable pivot selection one can achieve $\mathcal{O}(n \log n)$ time. Explain how to achieve

(a) average time complexity $\mathcal{O}(n \log n)$

(b) expected time complexity $\mathcal{O}(n \log n)$

(c) worst case time complexity $\mathcal{O}(n \log n)$.

Solution

(a)

Whenever we are dealing with an **average** time complexity, the assumption is that the algorithm is **not** random. Basically, we assume that every input is equally likely and we compute the average time complexity simply by summing all running times over all inputs and by dividing by the number of inputs; the same way as an average is computed.

In order to achieve an average time complexity $\mathcal{O}(n \log n)$ for the quicksort, whenever calling the actual quicksort routine with a particular range to sort, we could program it to choose, say, three pivots: one from the beginning of the input range, one from the end of the input range and one from the middle of the input range. After that, use the pivot that is second largest.

(b)

Whenever talking about **expected** time complexity, the assumption is that the algorithm is **randomized**, i.e, it relies on a pseudorandom number generator in order to make decisions.

We can guarantee an expected time complexity $\mathcal{O}(n \log n)$ if we choose pivot/pivots randomly.

(c)

In order to quarantee the **worst case** time complexity $\mathcal{O}(n \log n)$ for a quicksort implementation, I see no other possibility but using **Introsort**: use a quicksort with arbitrary pivot selection scheme, but keep track how deep in the recursion we get; if we exceed some precomputed depth, sort the range using **heapsort**, which guarantees the worst case time complexity of $\mathcal{O}(n \log n)$.

Exercise 2

A full binary tree is a binary tree where every node is either a leaf or has two children. Show that every full binary tree with n leaves has exactly $2n-1$ nodes. *Hint: Use induction.*

Solution

The base case: Suppose $n = 1$. The only possibility is that the leaf node is, in fact, the only node of the tree: $2n - 1 = 2 - 1 = 1$.
Now, the formula tells us that adding a new node (call it u) to the tree (such that the tree remains a full binary tree) introduces two nodes: u itself and some node (call it v) that makes sure that the binary tree remains full. We have three cases:

(I) Adding the new node to a leaf of the tree.

(II) Replacing an internal (that is, non-leaf) node with a new node.

Case I

Choose a leaf node l of the tree. Attach u to it. Now the tree is not full because l has only one child (u). In order to fix the full binary tree invariant, we add v as the other child of l. Now the tree is a full binary tree, and its number of nodes increased by 2.

Case II

Choose a non-leaf node (call it x). Substitute x with u and attach x as one of the children of u. At this point u has only one child; add v as the other child, which makes the binary tree full.