CS325: Analysis of Algorithms, Fall 2022

Practice Assignment 1 Solution

Problem 1. For each of the following, indicate whether $f = O(g)$, $f = \Omega(g)$ or $f = \Theta(g)$.

(a) $f(n) = 12n - 5$, $g(n) = 1235813n + 2017$. $f = \Theta(g)$

(b) $f(n) = n \log n$, $g(n) = 0.00000001n$. $f = \Omega(g)$

(c) $f(n) = n^{2/3}$, $g(n) = 7n^{3/4} + n^{1/10}$. $f = O(g)$

(d) $f(n) = n^{1.0001}$, $g(n) = n \log n$. $f = \Omega(g)$

(e) $f(n) = n6^n$, $g(n) = (3^n)^2$. $f = O(g)$

Problem 2. Prove that $\log(n!) = \Theta(n \log n)$. (Logarithms are based 2)

Solution. We prove two facts: (1) $\log(n!) = O(n \log n)$, and (2) $\log(n!) = \Omega(n \log n)$.

(1) $\log(n!) = O(n \log n)$**:**

$$\begin{aligned}
log(n!) &= \log\left(n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1\right) \\
&\leq \log\left(n \times n \times n \times \cdots \times n \times n\right) \\
&\leq \log(n^n) \\
&\leq n * log(n)
\end{aligned}$$

All the inequalities hold for all $n \geq 1$. Therefore, by setting $c = n_0 = 1$ in the definition of big-O, we obtain $\log(n!) = O(n \log n)$.

(2) $\log(n!) = \Omega(n \log n)$**:**

$$\begin{aligned}
log(n!) &= log\left(n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1\right) \\
&\geq \log\left(n \times (n-1) \times (n-2) \times \cdots \times \frac{n}{2}\right) \\
&\geq \log\left(\frac{n}{2} \times \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2}\right) \\
&\geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \\
&\geq \frac{n}{2} \cdot (\log n - 1) \\
&\geq \frac{1}{4} \cdot n \log n
\end{aligned}$$

The last inequality holds assuming that $n \geq 4$ (as $n \geq 4$ implies that $\log n - 1 \geq (1/2)\log n$. Therefore, by setting $c = 1/4$ and $n_0 = 4$, we obtain $\log(n!) = \Omega(n \log n)$.

Problem 3. Write a recursive algorithm to print the binary representation of a non-negative integer. Try to make your algorithm as simple as possible. Your input is a non-negative integer n. Your output would be the binary representation of n. For example, on input 5, your program would print '101'.

Solution. Let n be the input. Note that $n = 2 \times \lfloor n/2 \rfloor + (n \textit{ mod } 2)$, which readily implies the following recursive algorithm,

```
BinaryPrint(n)
    if (n = 0 or n = 1)
        print n
    else
        BinaryPrint(⌊n/2⌋)
        print (n mod 2)
```

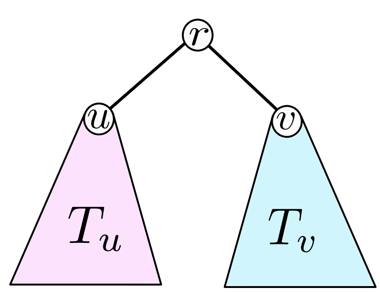
Note that the running time of the algorithm is proportional to the length of the binary representation of n, which is $\log(n)$. Also, note that what this algorithm prints is technically the reverse of the binary representation of n (how do you fix it?).

Problem 4.

(a) Read tree traversal from wikipedia: `https://en.wikipedia.org/wiki/Tree_traversal`, the first section, Types.

(b) Recall that a binary tree is *full* if every non-leaf node has exactly two children. Describe and analyze a recursive algorithm to reconstruct an arbitrary full binary tree, given its preorder and postorder node sequences as input. (Assume all keys are distinct in the binary tree)

Solution. Let a full binary tree $T = BinaryTree(r, T_u, T_v)$ be defined recursively by a root r, a left binary tree T_u, and a right binary tree T_v (since T is full either both of T_u and T_v are null, or none of them is null). The pre-order F of T is composed of the following three sequences in order (1) r, (2) F_u, the preorder of T_u, and (3) F_v, the preorder of T_v. Similarly, the postorder L of T is composed of the following three sequences in order (1) L_u, the postorder of T_u, (2) L_v, the postorder of T_v, and (3) r.



Moreover, the first element of F_u is u, and the last element of L_u is u. Hence, we can find the decomposition $L = L_u, L_v, r$ by locating u in the postorder traversal) (recall that the elements are distinct). Similarly, the last element of L_v is v, which is also the first element of F_v. Therefore, we can find the decomposition $F = r, F_u, F_v$ by locating v in the preorder traversal. After knowing the decompositions, it remains to recurse ☺. Note that, if the preorder and postorder traversals have length larger than one both u and v exist (as the tree is a full binary tree).

```
ReconstructTree(F[1...n], L[1...n])
    if len(F) = 1
        return BinaryTree(F[1], null, null)
    else
        Decompose F into r, F_u, F_v        # as explained above
        Decompose L into L_u, L_v, r        # as explained above
        return BinaryTree(r,ReconstructTree(F_u, L_u), ReconstructTree(F_v, L_v))
```

Running time analysis: You can look at the recursion tree to analyze the running time of this algorithm. The total sizes of all problems at each level is $O(n)$ (Why?). Therefore, non-recursive work at each level is at most $O(n)$. Note, all non-recursive work is to find the decompostions $F = r, F_u, F_v$ and $L = L_u, L_v, r$. Moreover, the tree has at most n levels (Why?). Adding everything together we can bound the running time of the algorithm by $O(n^2)$. In fact, a more careful analysis can show that this algorithm has running time $O(n)$ (Try to do it.)