University of Colombo School of Computing

SCS1308 - Foundations of Algorithms

Take Home Assignment 02

Instructions

- Try the following questions and upload your answer script as a zip file to the given link in the UGVLE on/before 8th of December at 6pm.
- Note: Rename your zip file with your index number and name. (i.e: indexNo_Name.zip)

1 Asymptotic Growth Rates

1.1 Questions on Asymptotic Notations

1. Find an upper bound for $f(n) = 3n + 8$.
2. Find a lower bound for $f(n) = n^2 - 4n + 7$.
3. Find a tight bound for $f(n) = 2n + 5$.
4. Find an upper bound for $f(n) = n \log_2 n + 3n$.
5. Find a tight bound for $f(n) = 4n^2 \log n + 2n \log n + 5n$.

1.2 Determine which relationship is correct and briefly explain why.

For each of the following $f(n)$ and $g(n)$ pairs, either $f(n)$ is in $O(g(n))$, $f(n)$ is in $\Omega(g(n))$ or $f(n)$ is in $\Theta(g(n))$.

1. $f(n) = 10$; $g(n) = \log(10)$
2. $f(n) = \log n^2$; $g(n) = \log n + 5$
3. $f(n) = 2n^4 - 3n^2 + 7$; $g(n) = n^5$
4. $f(n) = \log n$; $g(n) = \log n + \frac{1}{n}$

1.3 Prove or disprove the following

1. $n^2 = O(2^n)$
2. $n^3 - 3n^2 - n + 1 = O\left(n^3\right)$
3. $\Theta\left(n^2\right) = \Theta\left(n^2 + 1\right)$

1.4 Reason the following claims

Note: State 'Yes' if you agree, 'No' if you disagree. Provide reasons for your claim

1. Is $3^n = O(2^n)$?
2. Is $\log 3^n = O(\log 2^n)$?
3. Is $3^n = \Omega(2^n)$?
4. Is $\log 3^n = \Omega(\log 2^n)$?

2 Recursion and Recurrence Relations

Find the time complexity of the following algorithms using recursion-tree method

Question 01

```
int fact_helper(int n, int accumulator) {
    if (n <= 1)
        return accumulator;   // Base case: Return the accumulated result
    else
        return fact_helper(n - 1, n * accumulator);  // Recursive call
            with updated accumulator
}

int fact(int n) {
    return fact_helper(n, 1);  // Initial call with accumulator set to 1
}
```

This algorithm modifies the standard recursive factorial by introducing an accumulator to carry the computation, enabling tail-recursive optimization.
Part A: Derive the recurrence relation for the time complexity $T(n)$ of the `fact(n)` algorithm. Clearly explain each term in the recurrence relation.
Part B: Build a recursion tree for the algorithm fact(n). For each level of the tree, write down the number of nodes and the non-recursive work.
Part C: Using the recursion tree method, calculate the total number of operations performed by `fact(n)` and explain why it has $\Theta(n)$ complexity.

Question 02

Consider a sorted array A of size n containing distinct integers between 1 and $n+1$, with exactly one missing element (assume the arrays use 0-based indexing).
Part A: Design an algorithm $O(\log n)$ to find the missing integer, without using any extra space. Provide a pseudocode for your algorithm and briefly explain how it works.
Part B: Derive the recurrence relation for the time complexity $T(n)$ of your algorithm. Then, using the recursion tree method, prove that the run time is $\Theta(\log n)$. Include a sketch of the recursion tree and calculate the total cost.

Question 03

Given two sorted arrays A and B of size n and m, respectively, find the median of the elements $m + n$. The overall run time complexity should be $O(\log(n + m))$. Derive the recurrence relation for the time complexity $T(n)$ of your algorithm. Then, using the recursion tree method, prove that the runtime.

Question 04

Consider the quicksort algorithm for sorting an array of size n. Assume the pivot is always chosen as the last element, and analyze the average-case time complexity assuming random input (balanced partitions on average).

Part A Provide clear pseudocode (recursive) for the quicksort algorithm and briefly explain why it is correct.

Part B Derive the recurrence relation $T(n)$ for the average-case running time of quicksort.

Part C Using the **recursion tree method**, prove that the average-case running time is $\Theta(n \log n)$. Draw a sketch of the recursion tree and compute the total cost across all levels.

Question 05

Consider following algorithm of the Fast Fourier Transform (FFT) for computing the Discrete Fourier Transform (DFT) of a sequence of length n (assuming n is a power of 2 for simplicity), using a divide-and-conquer approach.

```
FFT(a):
    n = len(a)
    if n == 1:
        return a  // Base case: DFT of single element is itself

    omega = exp(2 * pi * i / n)  // Primitive nth root of unity

    a_even = [a[2*k] for k in 0 to n/2 - 1]
    a_odd = [a[2*k + 1] for k in 0 to n/2 - 1]

    even_dft = FFT(a_even)   // DFT of even indices
    odd_dft = FFT(a_odd)     // DFT of odd indices

    result = [0] * n
    for k in 0 to n/2 - 1:
        wk = omega ^ k
        result[k] = even_dft[k] + wk * odd_dft[k]
        result[k + n/2] = even_dft[k] - wk * odd_dft[k]

    return result
```

Part A Derive the recurrence relation $T(n)$ for the running time of the algorithm.

Part B Using the **recursion tree method**, prove that the running time is $\Theta(n \log n)$. Draw a sketch of the recursion tree and compute the total cost across all levels.

Definitions

Asymptotic Notation: Big-O, Omega, and Theta Bounds

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus, there exists some constant c such that $f(n) \leq c \cdot g(n)$ for every large enough n (that is, for all $n \geq n_0$, for some constant n_0).

- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus, there exists some constant c such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an *upper bound* on $f(n)$ *and* $c_2 \cdot g(n)$ is a *lower bound* on $f(n)$, for all $n \geq n_0$. Thus, there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$ for all $n \geq n_0$. This means that $g(n)$ provides a *nice, tight bound* on $f(n)$.

- $f(n) = o(g(n))$ means $g(n)$ *strictly dominates* $f(n)$ asymptotically. For *every* positive constant $c > 0$, there exists n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$. In other words, $f(n)$ grows *slower* than any positive multiple of $g(n)$. Equivalently, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

- $f(n) = \omega(g(n))$ means $f(n)$ *strictly dominates* $g(n)$ asymptotically. For *every* positive constant $c > 0$, there exists n_0 such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$. Thus, $f(n)$ grows *faster* than any positive multiple of $g(n)$. Equivalently, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.