COMP 3711 – Design and Analysis of Algorithms
2024 Fall Semester – Written Assignment 1
Distributed: 9:00 on September 14, 2024
Due: 23:59 on September 27, 2024

Your solution should contain
(i) your name, (ii) your student ID #, and (iii) your email address
at the top of its first page.

Some Notes:

- Please write clearly and briefly. In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.
- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page. ***You must acknowledge individuals who assisted you, or sources where you found solutions.*** Failure to do so will be considered plagiarism.
- The term *Documented Pseudocode* means that your pseudocode must contain documentation, i.e., comments, inside the pseudocode, briefly explaining what each part does.
- Many questions ask you to explain things, e.g., what an algorithm is doing, why it is correct, etc. To receive full points, the explanation must also be *understandable* as well as correct.
- Submit a SOFTCOPY of your assignment to Canvas by the deadline. If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.

1. (25 points) For each pair of expressions (A, B) below, indicate whether A is O, Ω , or Θ of B. Note that zero, one, or more of these relations may hold for a given pair. **List all applicable relations. No explanation is needed.** If $A = \Theta(B)$ then write $A = \Theta(B)$ which already implies that $A = O(B)$ and $A = \Omega(B)$. If $A = \Theta(B)$ and you write $A = O(B)$ or $A = \Omega(B)$ only, you will only receive partial credits. It often happens that some students will get the directions wrong, so please write out the relation in full, i.e., $A = O(B)$, $A = \Omega(B)$, or $A = \Theta(B)$ and not just $O(B)$, $\Omega(B)$ or $\Theta(B)$.

$n^3(\log n)^5$	$n^{3.1}$
$2^{\sqrt{\log n}}$	n
$10n^2 - 2n + 5$	$100n^2$
n^2	$4^{\log_2 n}$
$n \log_2 n + n^{1.5}$	$n(\log_{10} n)^{1.5}$

Solution:

(a) $A = O(B)$
(b) $A = O(B)$
(c) $A = \Theta(B)$
(d) $A = \Theta(B)$
(e) $A = \Omega(B)$

2. (25 points) Derive asymptotic upper bounds for $T(n)$ in the following recurrences. Make your bounds as tight as possible. **Do not use the master theorem. Derive your solutions from scratch.** Show all your steps in order to gain full credits (either using the repeated expansion method or the recursion tree method). You may assume that n is a power of 2 for (b) and (c), n is a pwoer of 4 for (d), and n is a power of 5 for (e).

(a) $T(1) = 1$; $T(n) = T(n-1) + n^2$ for $n > 1$.
(b) $T(1) = 1$; $T(n) = 4T(n/2) + n^2$ for $n > 1$.
(c) $T(1) = 1$; $T(n) = 3T(n/2) + n$ for $n > 1$.
(d) $T(1) = 1$; $T(n) = 2T(n/4) + \sqrt{n}$ for $n > 1$.
(e) $T(1) = 1$; $T(n) = T(n/5) + T(3n/5) + n$ for $n > 1$. Hint: Draw the recursion tree. Examine of the sum of costs across a level. Guess a solution. Then, try to prove by induction.

Solution:

(a) Given:

$$T(1) = 1, \quad T(n) = T(n-1) + n^2 \textit{ for } n > 1$$

Using the repeated substitution method:

$$\begin{aligned}
T(n) &= T(n-1) + n^2 \\
&= (T(n-2) + (n-1)^2) + n^2 \\
&= ((T(n-3) + (n-2)^2) + (n-1)^2) + n^2 \\
&= \cdots = T(1) + \sum_{k=2}^{n} k^2 \\
&= 1 + \sum_{k=2}^{n} k^2
\end{aligned}$$

We know that:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

So,

$$\sum_{k=2}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} - 1^2 = \frac{n(n+1)(2n+1)}{6} - 1$$

Thus,

$$T(n) = 1 + \frac{n(n+1)(2n+1)}{6} - 1 = \frac{n(n+1)(2n+1)}{6}$$

The asymptotic upper bound is:

$$T(n) = O(n^3)$$

(b) Given:

$$T(1) = 1, \quad T(n) = 4T(n/2) + n^2 \text{ for } n > 1$$

Using the repeated substitution method:

$$\begin{aligned}
T(n) &= 4T(n/2) + n^2 \\
&= 4(4T(n/4) + (n/2)^2) + n^2 \\
&= 4^2T(n/4) + 4 \cdot \frac{n^2}{4} + n^2 \\
&= 4^2T(n/4) + n^2 + n^2 \\
&= 4^3T(n/8) + 3n^2 \\
&= \cdots \\
&= 4^kT(n/2^k) + kn^2
\end{aligned}$$

For $k = \log_2 n$:

$$T(n) = 4^{\log_2 n}T(1) + (\log_2 n)n^2 = n^2(\log_2 n + 1)$$

The asymptotic upper bound is:

$$T(n) = O(n^2 \log n)$$

(c) *Given:*

$$T(1) = 1, \quad T(n) = 3T(n/2) + n \textit{ for } n > 1$$

Using the repeated substitution method:

$$\begin{aligned}
T(n) &= 3T(n/2) + n \\
&= 3(3T(n/4) + n/2) + n \\
&= 3^2T(n/4) + 3 \cdot \frac{n}{2} + n \\
&= 3^2T(n/4) + \frac{3n}{2} + n \\
&= 3^3T(n/8) + 3^2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{2} + n \\
&= \cdots \\
&= 3^kT(n/2^k) + \sum_{i=0}^{k-1} 3^i \cdot \frac{n}{2^{k-i}}
\end{aligned}$$

For $k = \log_2 n$*:*

$$T(n) = 3^{\log_2 n}T(1) + n \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i = n^{\log_2 3} + n \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i$$

Since $\sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i$ *is a geometric series:*

$$\sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i = \frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = \frac{n^{\log_2 \frac{3}{2}} - 1}{\frac{1}{2}} = 2(n^{\log_2 \frac{3}{2}} - 1)$$

Thus,

$$T(n) = O(n^{\log_2 3})$$

(d) *Given:*

$$T(1) = 1, \quad T(n) = 2T(n/4) + \sqrt{n} \textit{ for } n > 1$$

Using the repeated substitution method:

$$\begin{aligned}
T(n) &= 2T(n/4) + \sqrt{n} \\
&= 2(2T(n/16) + \sqrt{n/4}) + \sqrt{n} \\
&= 2^2T(n/16) + 2\sqrt{n/4} + \sqrt{n} \\
&= 2^2T(n/16) + \sqrt{n} + \sqrt{n} \\
&= 2^3T(n/64) + 3\sqrt{n} \\
&= \cdots \\
&= 2^kT(n/4^k) + k\sqrt{n}
\end{aligned}$$

For $k = \log_4 n$*:*

$$T(n) = 2^{\log_4 n}T(1) + \sqrt{n}\log_4 n = n^{\log_4 2} + \sqrt{n}\log_4 n$$

Since $\log_4 2 = \frac{1}{2}$*:*

$$T(n) = O(\sqrt{n}\log n)$$

(e) *Given:*

$$T(1) = 1, \quad T(n) = T(n/5) + T(3n/5) + n \text{ for } n > 1$$

We will use induction to show that $(T(n) = O(n))$.

Base Case:

For $(n = 1)$, $(T(1) = 1)$, which is $(O(1))$.

Inductive Hypothesis:

Assume that for all $(k < n)$, $(T(k) \leq ck)$ for some constant $(c > 0)$.

Inductive Step:

We need to show $(T(n) \leq cn)$.

$$T(n) = T(n/5) + T(3n/5) + n$$

Using the inductive hypothesis:

$$T(n/5) \leq c(n/5), \quad T(3n/5) \leq c(3n/5)$$

Thus:

$$T(n) \leq c(n/5) + c(3n/5) + n$$

Simplifying:

$$T(n) \leq c(\frac{n}{5} + \frac{3n}{5}) + n$$

$$T(n) \leq (\frac{4c}{5} + 1)n$$

We can choose c such that $4c/5 + 1 \leq c$ or equivalently, $c \geq 5$ so that:

$$T(n) \leq cn$$

This completes the induction, proving that $(T(n) = O(n))$.

3. (25 points)

   (a) (15 points) Describe a *recursive* algorithm that returns all possible $n \times n$ binary arrays in which there is exactly one 1 in each row and each column. The following figure shows three examples of such 3×3 binary arrays (zeros are suppressed).

   Describe your algorithm using a **documented** pseudocode. Make sure that your algorithm is recursive; otherwise, you will lose most of the points for problem 3. Make sure that your description is understandable.

   (b) (10 points) Write down the recurrence for the running time of your recursive algorithm in (a) with the boundary condition(s). Explain your notations. Solve your recurrence **from scratch** to obtain the the running time of your algorithm.

Solution:

Answer: (a) *Here is the high-level idea. You can place "1" in any cell of the first row (second, third, ..., nth). After placing the 1 in the k-th cell, remove the first row and the k-th column, resulting in another $(n-1) \times (n-1)$ table. Then, recursively generate the smaller tables.*

Algorithm 1 GenerateBinaryArrays

```
 1: function RecursiveGenerate($n$)
 2:     if n=1 then
 3:         Return $\{A[1]\}$                    ▷ a $1 \times 1$ table that has 1 in its cell.
 4:     end if
 5:     NewTables $\leftarrow$ RecursiveGenerate($n-1$)
 6:     Tables $\leftarrow \emptyset$
 7:     row = 1
 8:     for $col$ from 1 to $n$ do
 9:         for each table $T$ in NewTables do
10:             form a new table $T'$ as follows
11:             make the $j$-th column of $T$ the $j$-th column of $T'$ after adding
                a leading 0 for $j \in [1, col-1]$
12:             make the $j$-th column of $T$ the $(j+1)$-th column of $T'$ after
                adding a leading 0 for $j \in [col, n-1]$
13:             set the column $col$ of $T'$ to have a leading 1 followed by 0s
14:         end for
15:         Insert $T'$ into Tables
16:     end for
17:     return Tables
18: end function
```

(b) *Write down the recurrence for the running time of your recursive algorithm in (a) with the boundary condition(s). Explain your notations. Solve your recurrence from scratch to obtain the running time of your algorithm.*

The recurrence for the running time $T(n)$ of the recursive algorithm is:

$$T(n) = n^2 \cdot T(n-1)$$

Boundary condition:

$$T(1) = O(1)$$

Solving the recurrence:

$$\begin{aligned} T(n) &= n^2 \times T(n-1) \\ &= n^2 \times (n-1)^2 \times T(n-2) \\ &= \cdots \\ &= n^2 \times (n-1)^2 \times \cdots \times T(1) \\ &= (n \times (n-1) \times \cdots \times 1) \cdot (n \times (n-1) \times \cdots \times 1) \\ &= O((n!)^2) \end{aligned}$$

Thus, the running time of the algorithm is $O(n!)$.

4. (25 points) Let $A[1..n]$ be an array of n elements. One can compare in $O(1)$ time two elements of A to see if they are equal or not; however, the order relations $<$ and $>$ do not make sense. That is, one can check whether $A[i] = A[j]$ in $O(1)$ time, but the relations $A[i] < A[j]$ and $A[i] > A[j]$ are undefined and cannot be determined.

   In the tutorial you developed an $O(n \log n)$-time divide-and-conquer algorithm for finding a *majority element* of A if one exists. In this assignment you need to generalize this problem.

   Let $k \in [1..n]$ be a fixed integer. An element of $A[1..n]$ is a k-major element if its number of occurrences in A is greater than n/k. For example, if $n = 30$, then a 10-major element should occur greater than 3 times (i.e., at least 4 times). Note that it is possible that no k-major exists for a particular k; it is also possible that there are multiple k-major elements for a particular k.

   This problem concerns with designing a divide-and-conquer algorithm for finding **all** 10-major elements in $A[1..n]$ in $O(n \log n)$ time; if there is no 10-major element, report so. Answer the following questions.

   (a) (2 points) What is the maximum number of 10-major elements in $A[1..n]$? Explain.

   (b) (15 points) Design a divide-and-conquer algorithm that finds all 10-major elements in $A[1..n]$ in $O(n \log n)$ time; if there is no 10-major element, your algorithm should report so. Recall that one can check whether $A[i] = A[j]$ in $O(1)$ time, but the relations $A[i] < A[j]$ and $A[i] > A[j]$ are undefined and cannot be computed.

   Write your algorithm in documented pseudocode. Also, explain in text what your pseudocode does. Explain the correctness of your algorithm.

   Since your algorithm uses the divide-and-conquer principle, it should be recursive in nature. That is, it should work on $A[1..n]$ in the first call to return all 10-major elements of $A[1..n]$, and in subsequent recursive calls, it may recurse on many subarrays $A[p..q]$ for some $p, q \in [1..n]$ to return all 10-major elements of $A[p..q]$.

   Given a particular subarray $A[p..q]$, a 10-major element of $A[p..q]$ is not necessarily a 10-major element of $A[1..n]$. Conversely, a 10-major element of $A[1..n]$ is not necessarily a 10-major element of $A[p..q]$.

   (c) (6 points) Derive a recurrence relation that describes the running time $T(n)$ of your algorithm. Explain your reasoning. State the boundary condition(s).

   (d) (2 points) Solve your recurrence **from scratch** to show that $T(n) = O(n \log n)$.

   Solution:

(a) *The maximum number of 10-major elements in ($A[1 :: n]$) is 9. This is because, for an element to be a 10-major element, it must appear more than ($\frac{n}{10}$) times. Therefore, there can be at most 9 such elements since ($10 \times \frac{n}{10} = n$).*

(b) *Here is a divide and conquer algorithm to find 10-major elements. The high-level idea is that if an element is 10-major element in an array, it has be a 10-major element in either left half or right half(it can be 10-major element in both as well). But its not possible to have a 10-major element that is not a 10-major element in left half and right half. So We recursively find the 10-major element of the left half and right half and then count them in the main array to make sure if they are 10-major elements for the whole array or not. Note that as the base case, if we have an array of length less than 10, every value is a 10-major element.*

Algorithm 2 Find10MajorElements

```
procedure FIND10MAJORELEMENTS(A, n)
    Input: $A-$ array of size $n$
    Output: All 10-major elements in $A$
    if $n < 10$ then
        return []
    end if
    Split $A$ into two halves $A_1$ and $A_2$
    $M_1 \leftarrow$ Find10MajorElements($A_1, n/2$)          ▷ This is $T(n/2)$
    $M_2 \leftarrow$ Find10MajorElements($A_2, n/2$)          ▷ This is also $T(n/2)$
    $M \leftarrow M_1 \cup M_2$
    Count occurrences of elements in $M$ in $A$ ▷ Note that we have at most
constant number of elements in M. so this step takes $O(n)$
    result $\leftarrow$ elements in $M$ with occurrences more than $n/10$
    return result
end procedure
```

(c) *The recurrence relation for the running time $T(n)$ is:*

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

Boundary condition:

$$T(1) = O(1)$$

(d) *Using the master theorem for divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$:*

Here, $a = 2, b = 2$, and $f(n) = O(n)$.

Since $f(n) = O(n)$ and $n^{\log_b a} = n^{\log_2 2} = n$, we have $f(n) = O(n^{\log_b a} \log^k n)$ with $k = 0$.

Thus, by the master theorem, $T(n) = O(n \log n)$.