Indian Institute of Technology Kharagpur
Department of Computer Science and Engineering

Algorithms-I (CS21003) **Class Test – 1 Solutions** **[Maximum Marks: 20]** **Spring Semester, 2019-2020**

Date: 03-Feb-2020 (Monday) | Time: 7:00pm – 8:00pm | Venue: F-116 / F-142

Name: ______________________________ **Roll No:** ______________

[Instructions: ***Write your answers in proper places mentioned in the question paper itself. Answer ALL questions. Be brief and precise. If you use any algorithm/result/formula covered in class, just mention it, do not elaborate.*** **]**

Q1. Let two recursive algorithms satisfy the following two recurrence relations:

(i) $T(n) = \begin{cases} 3.T(\frac{n}{3}) + n, & \text{if } n > 1 \\ 1 \quad , & \text{if } n = 1 \end{cases}$ and (ii) $T(n) = \begin{cases} \sqrt{n}.T(\sqrt{n}\,) + n.\log_2^d n, & \text{if } n > 2 \\ 2 \quad , & \text{if } n = 2 \end{cases}$ $(d \geq 0)$

Deduce the running time $T(n)$ in asymptotic Θ-notation for both of these cases separately. **[Marks: 4 + 6 = 10]**

Solution:

(i) Given that, $T(n) = 3.T(\frac{n}{3}) + n$ and $T(1) = 1$.

$$\implies T(n) = 3^2.T(\frac{n}{3^2}) + 3.\frac{n}{3} + n = 3^3.T(\frac{n}{3^3}) + 3^2.\frac{n}{3^2} + 3.\frac{n}{3} + n = \cdots = 3^k.T(\frac{n}{3^k}) + n.k$$

$$\implies T(n) = 3^k.T(1) + n.k = n + n.\log_3 n = \Theta(n \log_2 n) \qquad \ldots\ldots [\, assuming,\ n = 3^k \,]$$

Marking Scheme:

- Expansions / Calculations shown = 2-marks
- Final closed-form computed = 1-mark
- Θ-notation provided = 1-mark

(ii) Given that, $T(n) = \sqrt{n}.T(\sqrt{n}\,) + n.\log_2^d n$ (where $d \geq 0$) and $T(2) = 2$.

$$\implies \frac{T(n)}{n} = \frac{T(\sqrt{n}\,)}{\sqrt{n}} + \log_2^d n \qquad \ldots\ldots [\, diving\ both\ sides\ by\ n \,]$$

$$\implies S(n) = S(\sqrt{n}\,) + \log_2^d n \qquad \ldots\ldots [\, assuming,\ S(n) = \frac{T(n)}{n} \,]$$

$$\implies S(2^{2^k}) = S(2^{2^{(k-1)}}) + (2^k)^d \qquad \ldots\ldots [\, substituting,\ n = 2^{2^k}]$$

$$\implies R(k) = R(k-1) + (2^k)^d \qquad \ldots\ldots [\, let,\ R(k) = S(2^{2^k}) \,]$$

$$\implies R(k) = R(0) + (2^d)^1 + (2^d)^2 + \cdots + (2^d)^{k-1} + (2^d)^k \qquad \ldots\ldots [\, because,\ (2^k)^d = 2^{kd} = (2^d)^k]$$

$$\implies R(k) = 1 + \sum_{i=1}^{k} (2^d)^i \qquad \ldots\ldots [\, S(2) = \frac{T(2)}{2} = 1\, , implying\ R(0) = S(2^{2^0}) = 1 \,]$$

$$\implies R(k) = \begin{cases} \frac{(2^d)^{(k+1)} - 1}{2^d - 1}, & \text{if } d > 0 \\ 1 + k, & \text{if } d = 0 \end{cases}$$

Hence, $R(k) = S(2^{2^k}) = \begin{cases} \frac{(2^d)^{(k+1)} - 1}{2^d - 1} = \Theta(2^{kd}), & \text{if } d > 0 \\ 1 + k \quad = \Theta(k), & \text{if } d = 0 \end{cases}$

which means, $S(n) = \begin{cases} \Theta(\log_2^d n), & \text{if } d > 0 \\ \Theta(\log_2 \log_2 n), & \text{if } d = 0 \end{cases}$ implying, $T(n) = \begin{cases} \Theta(n.\log_2^d n), & \text{if } d > 0 \\ \Theta(n.\log_2 \log_2 n), & \text{if } d = 0 \end{cases}$

$\ldots\ldots [\, as,\ n = 2^{2^k}\ and\ S(n) = \frac{T(n)}{n} \,]$

Marking Scheme:

- Two substitutions made and Expansions shown = 2-marks
- Final closed-form obtained = 2-marks (deduct 1-mark if $d = 0$ case not shown)
- Θ-notation provided = 2-marks (deduct 1-mark if $d = 0$ case not shown)

Q2. Let $\mathcal{A}$ be an $n \times n$ two-dimensional array with all distinct elements, in which all rows and all columns are sorted in ascending order from smaller to larger indices. Given a key x, your task is to find out whether x is present in $\mathcal{A}$.

(i) Propose a recursive formulation to solve this, from which you can design a $\Theta(n \log_2 n)$-time algorithm.
(ii) Propose an *efficient* recursive formulation to solve this, from which you can design a $O(n)$-time algorithm.

In both the above cases (separately), develop the recurrence relations from your recursive formulations and finally solve these to deduce the above-mentioned time-complexity of the algorithms. **[Marks: 4 + 6 = 10]**

Solution:

(i) We can perform binary search in each row (or column). The *recursive formulation* of the solution will be as follows:

```
found = array_search (A[][], row, col, x)
{   if (row == 0), then return FALSE.
    call Binary_Search over A[row][] elements to find x.
    if (x is present inside A[row][] elements), then return TRUE.
    else,  return array_search (A[][], row-1, col, x).          }
```

Initially, we call this recursive definition as follows: `array_search (A, n, n, x).`

In general, the *recurrence relation* for an $n \times m$ two-dimensional array gives, $T(n, m) = T(n-1, m) + \Theta(\log_2 m)$ and $T(0, m) = \Theta(1)$. $\therefore T(n, m) = \Theta(n \log_2 m)$. Here, since $m = n$, so $T(n, n) = \Theta(n \log_2 n)$.

This observation also leads to the following simple *iterative algorithm*:

```
Initialize flag = 0  (flag indicates whether element x is found or not).
loop over each row r from 1 to n {
    flag = Binary_Search (A[r][], 1, n, x).
    if (flag == 1), return TRUE.  else, increment r by 1.
}
if (flag == 0), return FALSE.
```

In the worst case (when the element is not found in $\mathcal{A}$), n number of binary-search operations are required and each binary-search operation takes $\Theta(\log_2 n)$ time. So, the *time-complexity* of the proposed algorithm is $\Theta(n \log_2 n)$.

(ii) Let $\langle row, col \rangle$ be an index in $\mathcal{A}$. If $x = \mathcal{A}[row][col]$, the search succeeds. If $x > \mathcal{A}[row][col]$, we can discard the left of the current row. Finally, if $x < \mathcal{A}[row][col]$, we can discard the lower part of the current column. The *recursive formulation* of the solution will be as follows:

```
found = array_search (A[][], n, row, col, x)
{   if (row > n) or (col < 1), then return FALSE.
    if (x == A[row][col]), then return TRUE.
    if (x > A[row][col]), then return array_search (A, n, row+1, col, x).
    else,  return array_search (A, n, row, col-1, x).                      }
```

Initially, we call this recursive definition as follows: `array_search (A, n, 1, n, x).`

In general, the *recurrence relation* for an $n \times m$ two-dimensional array gives,

$T(n, m) = \texttt{MAX}\left[\{T(n-1, m) + O(1)\}, \{T(n, m-1) + O(1)\}\right]$ and $T(0, i) = T(i, 0) = O(1)$ $(\forall i, 1 \le i \le n)$.

$\therefore T(n, m) = O(n + m)$. Here, since $m = n$, so $T(n, n) = O(n)$.

This observation also leads to the following simple *iterative algorithm*:

```
Initialize row = 1 and col = n  (top-right corner of the matrix).
loop forever {
    if (row > n) or (col < 1), return FALSE.
    if (x == A[row][col]), return TRUE.
    if (x > A[row][col]), increment row by 1.  else, decrement col by 1.
}
```

In the worst case (when the element is not found in $\mathcal{A}$), the number of comparisons required is $2n$. So, the time complexity of the proposed algorithm is $O(n)$.

Marking Scheme:

- Recursive formulation shown = 2-marks (for Solution-i) and + 4-marks (for Solution-ii)
- Time-complexity derived from recurrences = 2-marks (for Solution-i) and + 2-marks (for Solution-ii)