

Question 1)

Solve the following recurrence relations. For each one come up with a **precise** function of n in closed form (i.e., resolve all sigmas, recursive calls of function T , etc) using the substitution method. Note: An asymptotic answer is not acceptable for this question. Justify your solution and show all your work.

- a) $T(n)=T(n-1)+cn$, $T(0)=1$,
- b) $T(n)=4T(n/2)+n$, $T(1)=1$
- c) $T(n)=2T(n/2)+1$, $T(1)=1$

Answer:

a) $T(n)=T(n-1)+cn$, $T(0)=1$

Given, $T(0) = 1$ and $T(n) = T(n - 1) + cn$

$$T(n) = T(n - 1) + cn$$

Now, $T(n - 1) = T(n - 2) + c(n - 1)$ [Since, $T(m) = T(m - 1) + cm$ and $m = n - 1$ here]

Substitute $T(n - 1)$ in above equation.

$$T(n) = T(n - 2) + c(n - 1) + cn$$

Now, $T(n - 2) = T(n - 3) + c(n - 2)$ [Since, $T(m) = T(m - 1) + cm$ and $m = n - 2$ here]

Substitute $T(n - 2)$ in above equation.

$$T(n) = T(n - 3) + c(n - 2) + c(n - 1) + cn$$

And if we repeat it for k times, we get

$$T(n) = T(n - k - 1) + c(n - k) + \dots c(n - 1) + cn$$

We reach base case when $n - k - 1$ becomes 0

$$n - k - 1 = 0$$

$$\implies k = n - 1$$

$$\text{and } n - k = 1$$

At $k = n - 1$,

$$\text{Now, } T(n) = T(0) + c(1) + c(2) + c(3) + \dots c(n - 1) + cn$$

$$\text{Given, } T(0) = 1$$

$$\text{So, } T(n) = 1 + c[1 + 2 + 3 + \dots (n - 1) + n]$$

$$1 + 2 + \dots (n - 1) + n = n * (n + 1) / 2 \text{ according to mathematics.}$$

$$\text{So, } T(n) = c * n * (n + 1) / 2 + 1$$

$T(n) = (c / 2) * n^2 + (c / 2) * n + 1$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0 , C where $T(n) \leq C * g(n)$ for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $c * n^2$

Then, $(c / 2) * n^2 \leq c * n^2$ for all $n \geq 1$ since $c > c / 2$

$$(c / 2) * n \leq c * n^2 \text{ for all } n \geq 1 \text{ since } c > c / 2 \text{ and } n^2 > n$$

$$1 \leq c * n^2 \text{ for all } n \geq 1$$

If we add all three inequalities, then we get,

$$(c / 2) * n^2 + (c / 2) * n + 1 \leq c * n^2 + c * n^2 + c * n^2 \text{ for all } n \geq 1$$

$$\implies (c / 2) * n^2 + (c / 2) * n + 1 \leq 3 * c * n^2 \text{ for all } n \geq 1$$

$$\implies T(n) \leq 3 * c * n^2 \text{ for all } n \geq 1$$

Therefore, $T(n) = O(n^2)$ where $n_0 = 1$ and $C = 3 * c$

b) $T(n)=4T(n/2)+n$, $T(1)=1$

Given, $T(1)=1$ and $T(n)=4T(n/2)+n$

$$T(n) = 4T(n/2) + n$$

Now, $T(n/2) = 4T(n/4) + n/2$ [Since, $T(m) = 4T(m/2) + m$ and $m = n/2$ here]

Substitute $T(n/2)$ in above equation,

$$T(n) = 4 * [4 * T(n/4) + n/2] + n$$

$$T(n) = 4 * 4 * T(n/4) + 4 * n/2 + n$$

Now, $T(n/4) = 4T(n/8) + n/4$ [Since, $T(m) = 4T(m/2) + m$ and $m = n/4$ here]

Substitute $T(n/4)$ in above equation,

$$T(n) = 4 * 4 * [4 * T(n/8) + n/4] + 4 * n/2 + n$$

$$T(n) = 4 * 4 * 4 * T(n/8) + 4 * 4 * n/4 + 4 * n/2 + n$$

And if we repeat $k - 1$ times, we get

$$T(n) = 4^k * T(n/2^k) + 4^{(k-1)} * n/2^{(k-1)} + \dots n$$

We know, $4^{(k-1)} / 2^{(k-1)} = 2^{(k-1)} * 2^{(k-1)} / 2^{(k-1)} = 2^{(k-1)}$

So, $T(n) = 4^k * T(n/2^k) + 2^{(k-1)} * n + \dots n$

$$T(n) = 4^k * T(n/2^k) + n * [1 + 2 + 4 + \dots 2^{(k-1)}]$$

$1 + 2 + 4 + \dots 2^{(k-1)} = 2^k - 1$ according to mathematics

So, $T(n) = 4^k * T(n/2^k) + n * (2^k - 1)$

We reach base case when $n/2^k = 1$

$$n/2^k = 1$$

$$n = 2^k$$

$$k = \log_2(n)$$

At $k = \log_2(n)$

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

Substitute $T(1) = 1$

And $2^{\log_2 n} = n$ according to mathematics

And, $4^{\log_2 n} = 2^{2\log_2 n} = 2^{\log_2 n^2} = n^2$

So, $T(n) = n^2 + n * (n - 1)$

$$T(n) = 2n^2 - n$$

$T(n) = 2n^2 - n$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0 , C where $T(n) \leq C * g(n)$

for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $2 * n^2$

Then, $2 * n^2 = 2 * n^2$ for all $n \geq 1$ since both are equal

$-1 * n \leq 2 * n^2$ for all $n \geq 1$ since left value is negative and right one is positive

If we add both inequalities, then we get,

$$2 * n^2 - n \leq 2 * n^2 + 2 * n^2 \text{ for all } n \geq 1$$

$$\text{====} \quad 2 * n^2 - n \leq 4 * n^2 \text{ for all } n \geq 1$$

$$\text{====} \quad T(n) \leq 4 * n^2 \text{ for all } n \geq 1$$

Therefore, $T(n) = O(n^2)$ where $n_0 = 1$ and $C = 4$

c) $T(n) = 2T(n/2) + 1$, $T(1) = 1$

Given, $T(n) = 2T(n/2) + 1$, $T(1) = 1$

$$T(n) = 2 * T(n/2) + 1$$

Now, $T(n/2) = 2 * T(n/4) + 1$ [Since $T(m) = 2T(m/2) + 1$ and $m = n/2$ here]

Substitute $T(n/2)$ in above equation

$$T(n) = 2 * [2 * T(n/4) + 1] + 1$$

$$T(n) = 2 * 2 * T(n/4) + 2 + 1$$

Now, $T(n/4) = 2 * T(n/8) + 1$ [Since $T(m) = 2T(m/2) + 1$ and $m = n/4$ here]

Substitute $T(n/4)$ in above equation

$$T(n) = 2 * 2 * [2 * T(n / 8) + 1] + 2 + 1$$

$$T(n) = 2 * 2 * 2 * T(n / 8) + 2^2 + 2 + 1$$

And if we repeat k - 1 times, we get

$$T(n) = 2^k * T(n / 2^k) + 2^{(k-1)} + \dots + 2 + 1$$

$1 + 2 + 4 + \dots + 2^{(k-1)} = 2^k - 1$ according to mathematics

$$T(n) = 2^k * T(n / 2^k) + 2^k - 1$$

We reach base case when $n / 2^k = 1$

$$n / 2^k = 1$$

$$n = 2^k$$

$$k = \log_2(n)$$

At $k = \log_2(n)$

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

Substitute $T(1) = 1$

And $2^{\log_2 n} = n$ according to mathematics

$$T(n) = n * 1 + n - 1$$

$$T(n) = 2n - 1$$

$T(n) = 2n - 1$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0, C where $T(n) \leq C * g(n)$ for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $2 * n$

Then, $2 * n = 2 * n$ for all $n \geq 1$ since both are equal

$-1 \leq 2 * n$ for all $n \geq 1$ since left value is negative and right value is positive

If we add both inequalities, then we get,

$$2 * n - 1 \leq 2 * n + 2 * n \text{ for all } n \geq 1$$

$$\implies T(n) \leq 4 * n$$

Therefore, $T(n) = O(n)$ where $n_0 = 1$ and $C = 4$

Question 2)

Consider Question 1 again. Apply Master Theorem if applicable for each case. Bound the recurrence relation in Big-O.

Answer:

Master's Theorem is used to easily find Big-O Notation for functions with patterns in their recurrence relation.

We can apply Master's theorem only if the recurrence relation is of form:

$$T(n) = a * T(n / b) + f(n)$$

Where $a \geq 1$, $b \geq 2$ and $f(n)$ is an asymptotically positive function

It has 3 cases.

Case-I:

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Case-II:

If $f(n) = \Theta(n^d)$ where $d > \log_b a$

Then $f(n)$ grows asymptotically faster than $\log_b a$

Therefore, $T(n) = \Theta(n^d)$

Case-III:

If $f(n) = \Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = \Theta(n^d \log^{(k+1)} n)$

a) $T(n)=T(n-1)+cn, T(0)=1$

To Apply Master's Theorem, $T(n)$ should be of form

$$T(n) = a * T(n / b) + f(n)$$

Since, given function $T(n)=T(n-1)+cn$ is not in that form. We can't apply Master's theorem on this function.

b) $T(n)=4T(n/2)+n, T(1)=1$

To Apply Master's Theorem, $T(n)$ should be of form

$$T(n) = a * T(n / b) + f(n)$$

Given function is of the same form where $a = 4, b = 2$ and $f(n) = n$

$$f(n) = \Theta(n)$$

$$==== \Theta(n^1)$$

So, $d = 1$ in Master's Theorem

$$\log_b a = \log_2 4 = 2$$

Since, $d = 1 < \log_b a = 2$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Therefore, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Since, Θ represents a tight bound

$$C1 * n^2 \leq T(n) \leq C2 * n^2$$

If we consider, only $T(n) \leq C2 * n^2$ it represents Big-O Notation

Therefore, $T(n) = O(n^2)$

c) $T(n)= 2T(n/2)+1, T(1)=1$

To Apply Master's Theorem, $T(n)$ should be of form

$$T(n) = a * T(n / b) + f(n)$$

Given function is of the same form where $a = 2, b = 2$ and $f(n) = 1$

$$f(n) = \Theta(1)$$

$$==== \Theta(n^0)$$

So, $d = 0$ in Master's Theorem

$$\log_b a = \log_2 2 = 1$$

Since, $d = 0 < \log_b a = 1$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Therefore, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$

Since, Θ represents a tight bound

$$C1 * n \leq T(n) \leq C2 * n$$

If we consider, only $T(n) \leq C2 * n$ it represents Big-O Notation

Therefore, $T(n) = O(n)$

Question 3)

A binary tree's "maximum depth" is the number of nodes along the longest path from the root node down to the farthest leaf node. Given the root of a binary tree, write a complete program in C++/Java that returns tree's maximum depth. What is the time-complexity of your algorithm in the worst-case once you have n nodes in the tree. Analyze and clearly discuss your reasoning. Paste your complete program in the solution file.

Answer:

Java Code to Calculate maximum depth of a binary tree is

```
public static int maxDepth(Node root) {
    // Base Condition
    if(root == null) return 0;
    // Recursive Calls
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);

    return Math.max(left, right) + 1;
}
```

Let $T(n)$ is the time complexity of the above function.

Number of basic operations are:

Code	Cost	Number of times it runs
Base Condition		
if(root == null) return 0;	C1	1
Recursive Calls		
int left = maxDepth(root.left);	$T(n/2)$	1
int right = maxDepth(root.right);	$T(n/2)$	1
return Math.max(left, right) + 1;	C2	1

So, $T(n) = T(n/2) + T(n/2) + C2$ and $T(1) = C1$

$T(n) = 2T(n/2) + C2$ and $T(1) = C1$

Given, $T(n) = 2T(n/2) + C2$, $T(1) = C1$

$$T(n) = 2 * T(n/2) + C2$$

Now, $T(n/2) = 2 * T(n/4) + C2$ [Since $T(m) = 2T(m/2) + C2$ and $m = n/2$ here]

Substitute $T(n/2)$ in above equation

$$T(n) = 2 * [2 * T(n/4) + C2] + C2$$

$$T(n) = 2 * 2 * T(n/4) + 2C2 + C2$$

Now, $T(n/4) = 2 * T(n/8) + C2$ [Since $T(m) = 2T(m/2) + C2$ and $m = n/4$ here]

Substitute $T(n/4)$ in above equation

$$T(n) = 2 * 2 * [2 * T(n/8) + 1C2] + 2C2 + C2$$

$$T(n) = 2 * 2 * 2 * T(n/8) + 2^2C2 + 2C2 + 1C2$$

And if we repeat $k - 1$ times, we get

$$T(n) = 2^k * T(n/2^k) + [2^{(k-1)} + \dots + 2 + 1]C2$$

$1 + 2 + 4 + \dots + 2^{(k-1)} = 2^k - 1$ according to mathematics

$$T(n) = 2^k * T(n/2^k) + [2^k - 1]C2$$

We reach base case when $n/2^k = 1$

$$n/2^k = 1$$

$$n = 2^k$$

$$k = \log_2(n)$$

At $k = \log_2(n)$

$$T(n) = 2^{\log_2 n} * T(1) + [2^{\log_2 n} - 1]C2$$

Substitute $T(1) = C1$

And $2^{\log_2 n} = n$ according to mathematics

$$T(n) = n * C1 + [n - 1]C2$$

$$T(n) = [C1 + C2]n - C2$$

$T(n) = [C1 + C2]n - C2$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0, C where $T(n) \leq C * g(n)$ for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $[C1 + C2] * n$

Then, $[C1 + C2] * n = [C1 + C2] * n$ for all $n \geq 1$ since both are equal

$-C2 \leq [C1 + C2] * n$ for all $n \geq 1$ since left value is negative and right value is

positive

If we add both inequalities, then we get,

$$[C1 + C2]n - C2 \leq [C1 + C2]n + [C1 + C2]n \text{ for all } n \geq 1$$

$$\implies T(n) \leq 2[C1 + C2]n$$

Therefore, $T(n) = O(n)$ where $n_0 = 1$ and $C = 2[C1 + C2]$

We can also solve the recurrence relation Using Master's theorem.

Master's Theorem is used to easily find Big-O Notation for functions with patterns in their recurrence relation.

We can apply Master's theorem only if the recurrence relation is of form:

$$T(n) = a * T(n / b) + f(n)$$

Where $a \geq 1$, $b \geq 2$ and $f(n)$ is an asymptotically positive function

It has 3 cases.

Case-I:

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Case-II:

If $f(n) = \Theta(n^d)$ where $d > \log_b a$

Then $f(n)$ grows asymptotically faster than $\log_b a$

Therefore, $T(n) = \Theta(n^d)$

Case-III:

If $f(n) = \Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = \Theta(n^d \log^{(k+1)} n)$

Given function is of the same form where $a = 2$, $b = 2$ and $f(n) = C2$

$$f(n) = \Theta(1)$$

$$\implies \Theta(n^0)$$

So, $d = 0$ in Master's Theorem

$$\log_b a = \log_2 2 = 1$$

Since, $d = 0 < \log_b a = 1$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Therefore, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$

Since, Θ represents a tight bound

$$C1 * n \leq T(n) \leq C2 * n$$

If we consider, only $T(n) \leq C2 * n$ it represents Big-O Notation

Therefore, $T(n) = O(n)$

Question 4) Task given in video lectures

Answer:

Given, $T(n) = T(n-1) + T(n-2) + O(1)$, $T(0) = 1$ and $T(0) = T(1) = O(1)$

$$T(n) = T(n-1) + T(n-2) + O(1)$$

We know that $T(n-1) > T(n-2)$ since $T(n-1) = T(n-2) + T(n-3) + O(1)$ i.e; Time Complexity of $T(n-1)$ has $T(n-2)$ Time Complexity + Some more.

$$==== T(n-2) < T(n-1)$$

$$==== T(n-2) + T(n-1) < T(n-1) + T(n-1)$$

$$==== T(n-2) + T(n-1) + O(1) < T(n-1) + T(n-1) + O(1)$$

$$==== T(n) < 2 * T(n-1) + O(1)$$

$$\text{So, } T(n) < 2 * T(n-1) + O(1)$$

$$T(n) < 2 * [2 * T(n-2) + O(1)] + O(1)$$

$$T(n) < 4 * T(n-2) + 2 * O(1) + O(1)$$

$$T(n) < 4 * [2 * T(n-3) + O(1)] + 2 * O(1) + O(1)$$

$$T(n) < 8 * T(n-3) + 4 * O(1) + 2 * O(1) + O(1)$$

If we repeat for k steps

$$T(n) < 2^k * T(n-k) + 2^{(k-1)} * O(1) + 2^{(k-2)} * O(1) + \dots + 2 * O(1) + O(1)$$

$$T(n) < 2^k * T(n-k) + O(1) * [2^{(k-1)} + 2^{(k-2)} + \dots + 2 + 1]$$

$$2^{(k-1)} + 2^{(k-2)} + \dots + 2 + 1 = 2^k - 1 \text{ according to mathematics}$$

$$T(n) < 2^k * T(n-k) + O(1) * (2^k - 1)$$

We reach base case when $n - k = 0 \implies k = n$

At base case,

$$T(n) < 2^n * T(0) + O(1) * (2^n - 1)$$

$$T(n) < 2^n * O(1) + O(1) * (2^n - 1)$$

$$==== T(n) < O(1) * [2^n + 2^n - 1]$$

$$==== T(n) < 2O(1) * [2^n - 1]$$

$$==== T(n) < 2O(1) * 2^n - O(1)$$

We know that $-O(1) < 2O(1) * 2^n$ since left hand value is -ve and right hand value is +ve.

$$\text{So, } 2O(1) * 2^n - O(1) < 2O(1) * 2^n + 2O(1) * 2^n$$

$$==== T(n) < 4O(1) * 2^n$$

Therefore, $T(n) = O(2^n)$

Question 4)

Part (a) Write a **linear time divide and conquer algorithm** (i.e., $\theta(n)$) to calculate x^n (x is raised to the power n). Assume a and n are ≥ 0 .

Part (b) Analyze the time complexity of your algorithm in the worst-case by first writing its recurrence relation.

Part (c) Can you improve your algorithm to accomplish the end in $O(\log n)$ time complexity (we still look for a divide and conquer algorithm). If yes, write the corresponding algorithm, write the recurrence relation for its time complexity and analyze it. If no, justify your answer.

Answer:

a) Algorithm to find x^n in linear time is

```
public static int linearXpowN(int x, int n){
    if(n == 1) return x;
```

```

    int product = 1;
    if(n % 2 != 0){
        product = x;
        n -= 1;
    }
    return product * linearXpowN(x, n / 2) * linearXpowN(x, n / 2);
}

```

b) Let $T(n)$ is the time complexity of the above function. Number of basic operations are:

Code	Cost	Number of times it runs
Base Condition		
if(n == 1) return x;	C1	1
Recursive Calls		
int product = 1;	C2	1
if(n % 2 != 0)	C3	1
product = x;	C4	1
n -= 1;	C5	1
return product * linearXpowN(x, n / 2) * linearXpowN(x, n / 2);	$T(n/2)$	2

So, $T(n) = 2T(n/2) + (C2 + C3 + C4 + C5)$ and $T(1) = C1$

Let, $C = (C2 + C3 + C4 + C5)$

Then, $T(n) = 2T(n/2) + C$ and $T(1) = C1$

Given, $T(n) = 2T(n/2) + C$, $T(1) = C1$

$$T(n) = 2 * T(n/2) + C$$

Now, $T(n/2) = 2 * T(n/4) + C$ [Since $T(m) = 2T(m/2) + C$ and $m = n/2$ here]

Substitute $T(n/2)$ in above equation

$$T(n) = 2 * [2 * T(n/4) + C] + C$$

$$T(n) = 2 * 2 * T(n/4) + 2C + C$$

Now, $T(n/4) = 2 * T(n/8) + C$ [Since $T(m) = 2T(m/2) + C$ and $m = n/4$ here]

Substitute $T(n/4)$ in above equation

$$T(n) = 2 * 2 * [2 * T(n/8) + 1C] + 2C + C$$

$$T(n) = 2 * 2 * 2 * T(n/8) + 2^2C + 2C + 1C$$

And if we repeat $k - 1$ times, we get

$$T(n) = 2^k * T(n/2^k) + [2^{(k-1)} + \dots + 2 + 1]C$$

$1 + 2 + 4 + \dots + 2^{(k-1)} = 2^k - 1$ according to mathematics

$$T(n) = 2^k * T(n/2^k) + [2^k - 1]C$$

We reach base case when $n/2^k = 1$

$$n/2^k = 1$$

$$n = 2^k$$

$$k = \log_2(n)$$

At $k = \log_2(n)$

$$T(n) = 2^{\log_2 n} * T(1) + [2^{\log_2 n} - 1]C$$

Substitute $T(1) = C1$

And $2^{\log_2 n} = n$ according to mathematics

$$T(n) = n * C1 + [n - 1]C$$

$$T(n) = [C1 + C]n - C$$

$T(n) = [C1 + C]n - C$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0, C where $T(n) \leq C * g(n)$ for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $[C1 + C] * n$

Then, $[C1 + C] * n = [C1 + C] * n$ for all $n \geq 1$ since both are equal

$-C \leq [C1 + C] * n$ for all $n \geq 1$ since left value is negative and right value is

positive

If we add both inequalities, then we get,

$$[C1 + C]n - C \leq [C1 + C]n + [C1 + C]n \text{ for all } n \geq 1$$

$$\implies T(n) \leq 2[C1 + C]n$$

Therefore, $T(n) = O(n)$ where $n_0 = 1$ and $C = 2[C1 + C]$

We can also solve the recurrence relation Using Master's theorem.

We can apply Master's theorem only if the recurrence relation is of form:

$$T(n) = a * T(n/b) + f(n)$$

Where $a \geq 1, b \geq 2$ and $f(n)$ is an asymptotically positive function

It has 3 cases.

Case-I:

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Case-II:

If $f(n) = \Theta(n^d)$ where $d > \log_b a$

Then $f(n)$ grows asymptotically faster than $\log_b a$

Therefore, $T(n) = \Theta(n^d)$

Case-III:

If $f(n) = \Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = \Theta(n^d \log^{(k+1)} n)$

Given function is of the same form where $a = 2, b = 2$ and $f(n) = C$

$$f(n) = \Theta(1)$$

$$\implies \Theta(n^0)$$

So, $d = 0$ in Master's Theorem

$$\log_b a = \log_2 2 = 1$$

Since, $d = 0 < \log_b a = 1$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Therefore, $T(n) = \Theta(n^{\log_b a})$

$$\implies \Theta(n)$$

Therefore, $T(n) = \Theta(n)$

Since, Θ represents a tight bound

$$C1 * n \leq T(n) \leq C2 * n$$

If we consider, only $T(n) \leq C2 * n$ it represents Big-O Notation

Therefore, $T(n) = O(n)$

c) Algorithm to find x^n in logarithmic time is

```
public static int logarithmicXpowN(int x, int n){
    if(n == 1) return x;
    int product = 1;
```

```

if(n % 2 != 0){
    product = x;
    n -= 1;
}
int subProduct = logarithmicXpowN(x, n / 2);
return product * subProduct * subProduct;
}

```

Let $T(n)$ is the time complexity of the above function. Number of basic operations are:

Code	Cost	Number of times it runs
Base Condition		
if(n == 1) return x;	C1	1
Recursive Calls		
int product = 1;	C2	1
if(n % 2 != 0)	C3	1
product = x;	C4	1
n -= 1;	C5	1
int subProduct = logarithmicXpowN(x, n / 2);	$T(n / 2)$	1
return product * subProduct * subProduct;	C6	1

So, $T(n) = T(n / 2) + (C2 + C3 + C4 + C5 + C6)$ and $T(1) = C1$

Let, $C = (C2 + C3 + C4 + C5 + C6)$

Then, $T(n) = T(n / 2) + C$ and $T(1) = C1$

Given, $T(n) = T(n/2) + C$, $T(1) = C1$

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

Now, $T(n / 2) = T(n / 4) + C$ [Since $T(m) = T(m / 2) + C$ and $m = n / 2$ here]

Substitute $T(n / 2)$ in above equation

$$T(n) = T(n / 4) + C + C$$

$$T(n) = T(n / 4) + C + C$$

Now, $T(n / 4) = T(n / 8) + C$ [Since $T(m) = T(m / 2) + C$ and $m = n / 4$ here]

Substitute $T(n / 4)$ in above equation

$$T(n) = T(n / 8) + C + C + C$$

$$T(n) = T(n / 8) + C + C + C$$

And if we repeat $k - 1$ times, we get

$$T(n) = T(n / 2^k) + (k - 1)C$$

We reach base case when $n / 2^k = 1$

$$n / 2^k = 1$$

$$n = 2^k$$

$$k = \log_2(n)$$

At $k = \log_2(n)$

$$T(n) = T(1) + [\log_2(n) - 1]C$$

Substitute $T(1) = C1$

$$T(n) = C1 + [\log_2(n) - 1]C$$

$$T(n) = C * \log_2(n) + C1 - C$$

$T(n) = C * \log_2(n) + C1 - C$ is the function representing the time complexity of given function.

To Find the Big-O Notation of given function, we need to find n_0, C where $T(n) \leq C * g(n)$ for all $n \geq n_0$

Consider a function $G(n)$ by replacing each value in $T(n)$ with $[C + C1] * \log_2(n)$

Then, $C * \log_2(n) \leq [C1 + C] * \log_2(n)$ for all $n \geq 1$ since right equation has one $C1$ more

$C1 - C \leq [C1 + C] * \log_2(n)$ for all $n \geq 1$ since left value is constant and right

value is variable according to n .

If we add both inequalities, then we get,

$C * \log_2(n) + C1 - C \leq [C + C1] * \log_2(n) + [C + C1] * \log_2(n)$ for all $n \geq 1$

==== $T(n) \leq 2[C1 + C] * \log_2(n)$

Therefore, $T(n) = O(\log_2(n))$ where $n_0 = 1$ and $C = 2[C1 + C]$

We can also solve the recurrence relation using Master's Theorem

We can apply Master's theorem only if the recurrence relation is of form:

$T(n) = a * T(n/b) + f(n)$

Where $a \geq 1, b \geq 2$ and $f(n)$ is an asymptotically positive function

It has 3 cases.

Case-I:

If $f(n) = \Theta(n^d)$ where $d < \log_b a$

Then $f(n)$ grows asymptotically slower than $\log_b a$

Therefore, $T(n) = \Theta(n^{\log_b a})$

Case-II:

If $f(n) = \Theta(n^d)$ where $d > \log_b a$

Then $f(n)$ grows asymptotically faster than $\log_b a$

Therefore, $T(n) = \Theta(n^d)$

Case-III:

If $f(n) = \Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = \Theta(n^d \log^{(k+1)} n)$

Given function is of the same form where $a = 1, b = 2$ and $f(n) = C$

$f(n) = \Theta(1)$

==== $\Theta(n^0)$

So, $d = 0$ in Master's Theorem

$\log_b a = \log_2 1 = 0$

Since, $d = 0 \implies \log_b a = 0$, the given function follows Case-III pattern of Master's Theorem

According to Case-III of Master's Theorem,

If $f(n) = \Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = \Theta(n^d \log^{(k+1)} n)$

We can rewrite $f(n)$ as

$f(n) = \Theta(1)$

$\Theta(n^0 \log^0 n)$

So, $d = 0$ and $k = 0$ in Master's Theorem

Therefore, $T(n) = \Theta(n^d \log^{(k+1)} n)$

==== $\Theta(n^0 \log^{(0+1)} n)$

==== $\Theta(\log n)$

Therefore, $T(n) = \Theta(\log n)$

Since, Θ represents a tight bound

$C1 * \log n \leq T(n) \leq C2 * \log n$

If we consider, only $T(n) \leq C2 * \log n$ it represents Big-O Notation

Therefore, $T(n) = O(\log n)$