BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design ZG519

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

SESSION 3 -PLAN

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
3	Analyzing Recursive Algorithms: Recurrence relations, Iteration Method, Substitution Method, Recursion Tree, Master Method.	T1: 1.4, 2.1

Analyzing Recursive Algorithms

- Recursive calls:-A procedure P calling itself-calls to P are for solving sub problems of smaller size.
- Recursive procedure call should always define a ***base case***.
- Base case-small enough that it can be solved directly without using recursion.

10

5

3

2

1

Analyzing Recursive Algorithms

rM (A, 5)

n = 5

- Algorithm recursiveMax(A,n)
  // Input : An array A storing n>=1 integers
  //Output: The maximum element in A
  if n = 1 then
  return A[0]
  return max{ recursiveMax(A,n-1),A[n-1]}

A[0] A[1] A[2] A[3] A[4]

A[5] = {3, 4, 1, 5, 2}

1 1 1 1

return max (5, 2) ⇒ 5

return max {rM (A, 4), 2}

return max {rM (A, 3), 5}

return max(4, 5)

return max {rM(A, 2), 1}

return max(4, 1)

return max(rM(A, 1), 4)

return (max(3, 4))

Analyzing Recursive Algorithms

- A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- ***Recurrence equation***: defines mathematical statements that the running time of a recursive algorithm must satisfy
- Analysis of ***recursiveMax***
  - T(n)-Running time of algorithm on an input size n

$$T(n) = \begin{cases} 3 \text{ Const} & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

base case

$$T(n) = T(n-1) + \text{Const}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Solving Recurrences

Solving recurrences : Iterative Method

Analyzing Recursive Algorithms-Iterative method

- **General Plan-Iterative Method**
  - Identify the parameter to be considered based on the size of the input.
  - Identify the basic operation in the algorithm
  - Obtain the number of times the basic operation is executed.
  - Obtain an initial condition-base case
  - Obtain a recurrence relation
  - Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Analyzing Recursive Algorithms-RecursiveMax

- Analysis of ***recursiveMax***

  T(n)-Running time of algorithm on an input size n

$$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

```
Algorithm recursiveMax(A,n)
// Input : An array A storing  n>=1 integers
 //Output: The maximum element in A
 if  n = 1 then
 return A[0]
return max{ recursiveMax(A,n-1),A[n-1]}
```

$$\begin{aligned} T(n) &= T(n-1) + 7 \\ &= T(n-2) + 7 + 7 \\ &= T(n-2) + 14 \\ &= T(n-3) + 21 \\ &\vdots \\ &= T(n-i) + 7i \end{aligned}$$

$\underline{T(1)}$

$n - i = 1$

$i = n - 1$

when $i = n-1$

$$\begin{aligned} T(n) &= T(n-(n-1)) + 7(n-1) \\ &= T(1) + 7n - 7 \\ &= 3 + 7n - 7 \\ &= \underline{7n-4} \text{ is } O(n) \end{aligned}$$

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

– ***Algorithm fact(n)***

//Purpose: Computes factorial of n

//Input: A positive integer n

//Output: factorial of n

If(n=0)

return 1

return n*fact(n-1)

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

- **Analysis**
  - Parameter to be considered –n
  - Basic operation –Multiplication
  - T(n) = 0 if n=0

    1+T(n-1) Otherwise

Time taken to compute fact(n-1)

Time to multiply n*fact(n-1)

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

- **Solve the recurrence**

$T(n) = T(n\text{-}1) + 1$

$[T(n\text{-}2) + 1] + 1 = T(n\text{-}2) + 2$ substituted $T(n\text{-}2)$ for $T(n\text{-}1)$

$[T(n\text{-}3) + 1] + 2 = T(n\text{-}3) + 3$ substituted $T(n\text{-}3)$ for $T(n\text{-}2)$

.. a pattern evolves

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

$=2+T(n-2)$

$=3+T(n-3)$

$=\ldots.$

$=i+T(n-i)$

When n=0 T(0)=0,No multiplications

When i=n, T(n) $=n+T(n-n)$

$=n+0$

$=n$ **$T(n) \in \Theta(n)$**

$T(0) = 0$

$n - i = 0$

$i = n$

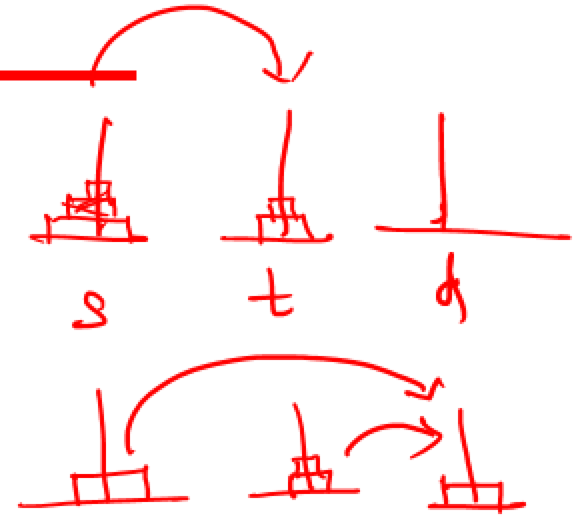
$= O(n)$

Analyzing Recursive Algorithms- Example **2:-Tower of hanoi**

Step 1 – Move n-1 disks from **source** to **temp**

Step 2 – Move n^{th} disk from **source** to **dest**

Step 3 – Move n-1 disks from **temp** to **dest**



Algorithm Hanoi(n, source, dest, temp)

//Input: n :number of disks

//Output :All n disks on dest

If disk = 1

move disk from source to dest

Hanoi(n - 1, source, temp, dest) // Step 1

move nth disk from source to dest // Step 2

Hanoi(n - 1, temp, dest, source) // Step 3

$T(1) = 1$

Analyzing Recursive Algorithms- Example **2:-Tower of hanoi**

1.Problem size is *n*, the number of discs

2. The basic operation is moving a disc from rod to another

3. Base case $M(1) = 1$

4. Recursive relation for moving n discs

$$\underbrace{M(n)}_{s \to d} = \underbrace{M(n\text{-}1)}_{s \to t} + \underbrace{1}_{s \to d} + \underbrace{M(n\text{-}1)}_{t \to d} = 2M(n\text{-}1) + 1$$

$$T(n) = \begin{cases} 1 & n = 1 \\ 2M(n-1)+1 & \text{otherwise} \end{cases}$$

Analyzing Recursive Algorithms-Example **2: Tower of hanoi**

Solve using backward substitution

$M(n) = 2M(n\text{-}1) + 1$

$= 2[2M(n\text{-}2) + 1] + 1 = 2^2M(n\text{-}2) + 2+1$

$=2^2[2M(n\text{-}3) + 1] + 2+1$

$= 2^3M(n\text{-}3) + 2^2 + 2 + 1$

...

$M(n) = 2^iM(n\text{-}i) + \underbrace{2\text{^}i\text{-}1+2\text{^}i\text{-}2+....+2\text{^}3+2\text{^}2+2\text{^}1+2\text{^}0}_{\text{It's a GP with } a=1, r=2, n=i}$

$M(n) = 2^iM(n\text{-}i) + (2\text{^}i\text{-}1)/(2\text{-}1)$

$= 2^iM(n\text{-}i) + 2\text{^}i\text{-}1$

Analyzing Recursive Algorithms- Example 2:- Tower of hanoi

- When i=n-1

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1$$
$$= 2^{n-1} + 2^{n-1} - 1$$
$$= 2*2^{n-1} - 1$$
$$= 2*(2^{n} / 2) - 1$$
$$= 2^{n} - 1$$

$\underline{\boldsymbol{M(n) \in O(2^n)}}$

- Time complexity is exponential
- More computattions even for smaller value of n
- Doesnt necessarily mean algorithm is poor
- Nature of the problem itself is computationally expensive.

Analyzing Recursive Algorithms-Example **3:Exercise**

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer *n*

//Output: The number of binary digits in *n*'s binary representation

if $n = 1$ **return** 1

else return *BinRec(n/2)* + 1

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing *BinRec(n/2)* is $A(n/2)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$A(n) = A(n/2) + 1$ for $n > 1$.

$A(1)=0$

Analyzing Recursive Algorithms-Example **3:Exercise**

Base condition A(1)= 0

$$A(n)=A(n/2)+1$$

The presence of $n/2$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the ***smoothness rule,*** which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of *n.*

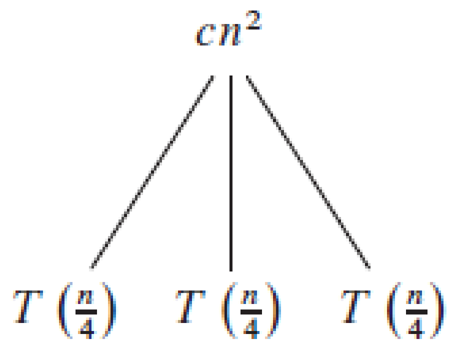
Solving recurrences : Recursion Tree

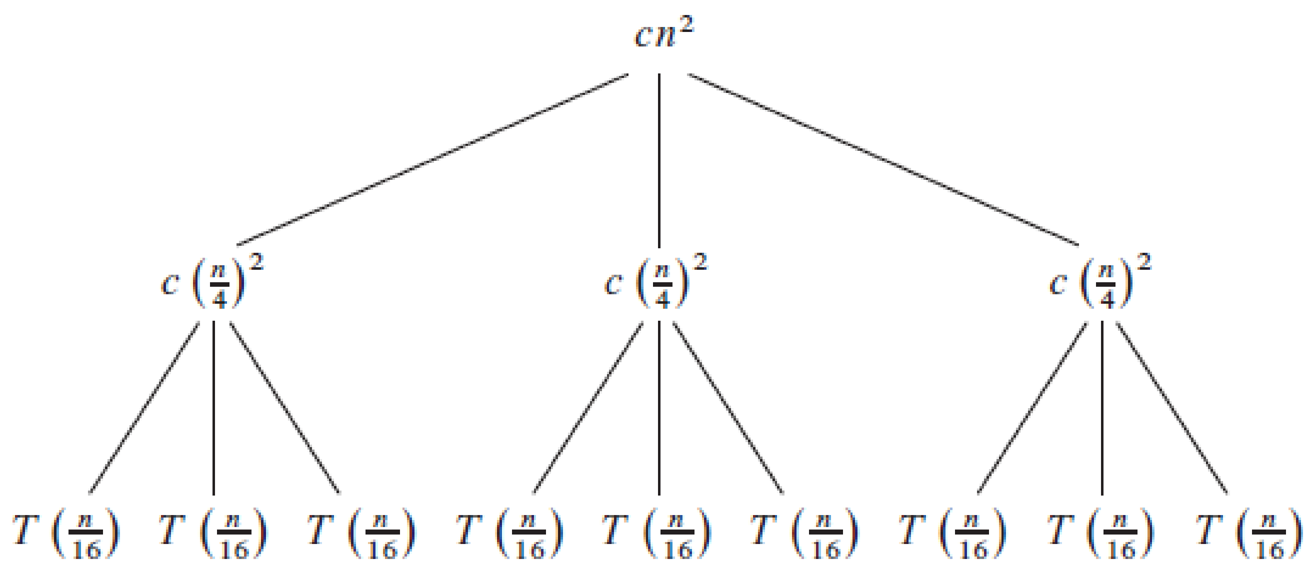
```
Void Test (int n)
{
        if(n>1)
        {
                for (i=1;i<n;i++)
                {
                        stmt;
                }
                Test(n/2);
                Test(n/2);
        }
}
```

$$T(n) = \begin{cases} 0, & n=1 \\ 2T(n/2)+n & n>1 \end{cases}$$

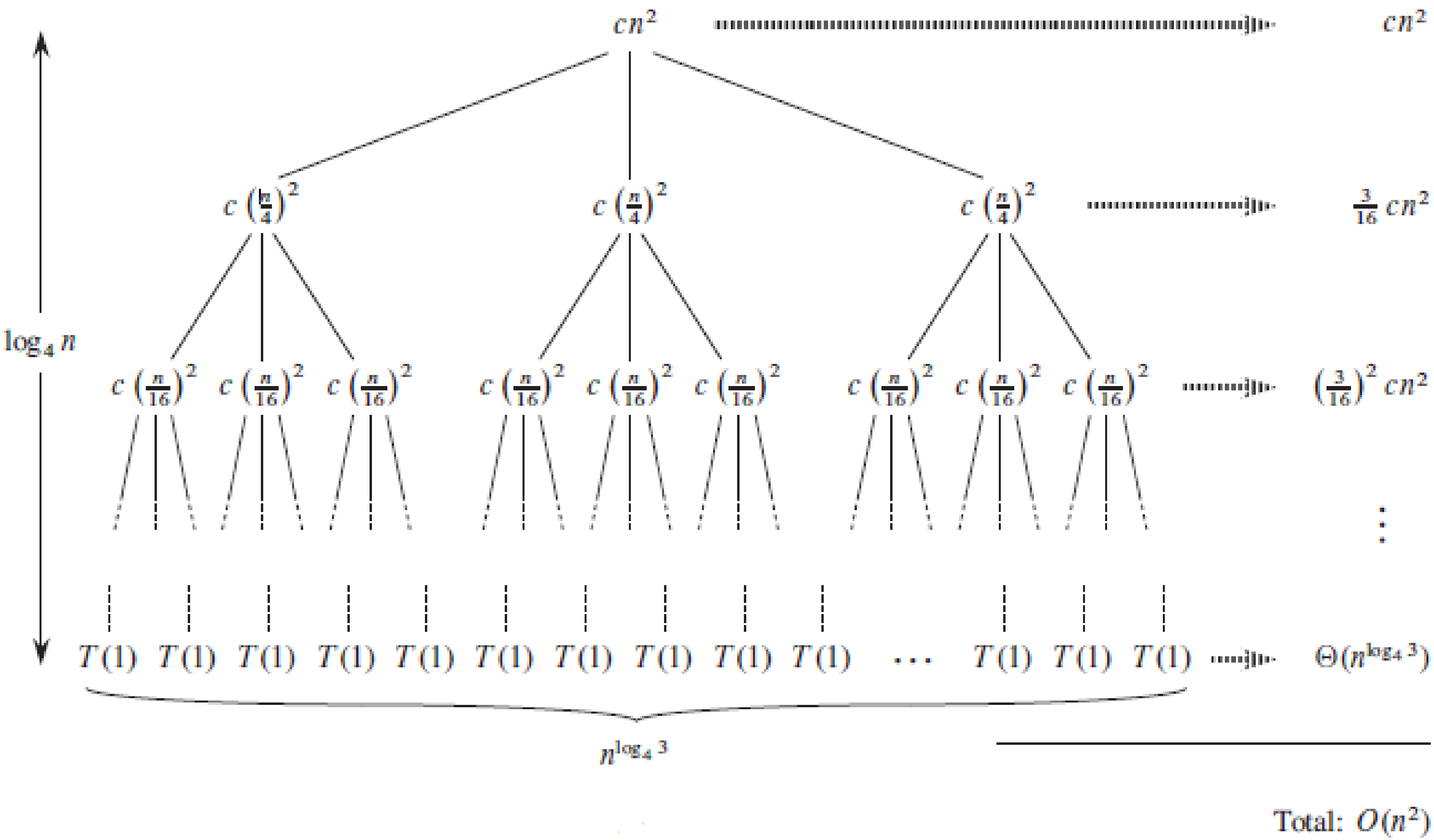
Solving Recurrences-Recursion Tree

- Solve $T(n) = 3T(n/4) + cn^2$.





Solving Recurrences-Recursion Tree



Solving Recurrences-Recursion Tree

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

Geometric or exponential series

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

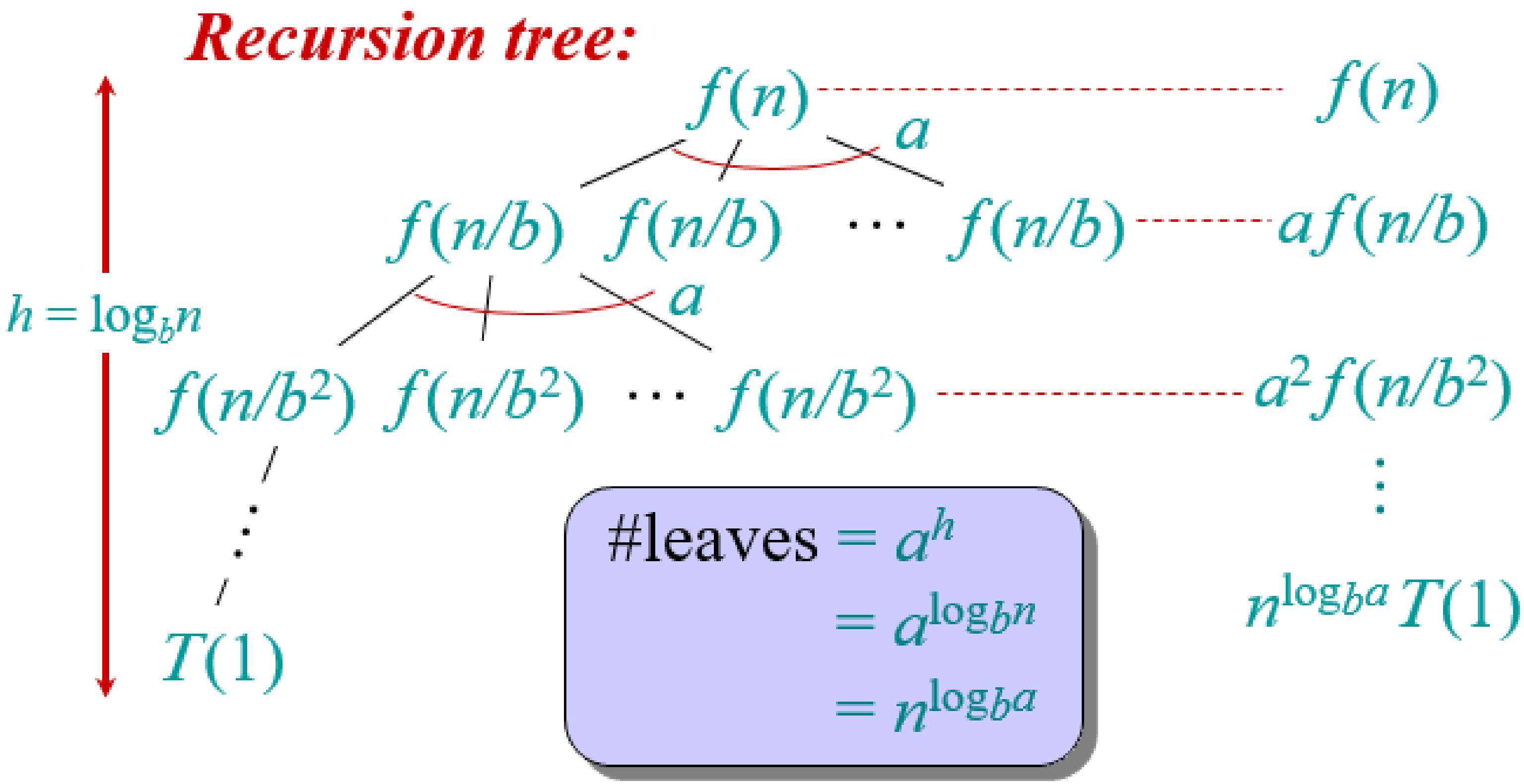
Solving recurrences: Master method Ref: Textbook R2

- The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

- where $a \geq 1$, $b > 1$, and f is asymptotically positive.
- (f(n)>0 for n>=n0)

Idea of Master theorem



Solving recurrences: Master method
Ref: Textbook R2

Case 1:

If $f(n)=O(n^{\log_b a-\varepsilon})$ *, for some constant* $\varepsilon > 0$*, then* $T(n)=\Theta(n^{\log_b a})$

f(n) grows polynomially slower than $n^{\log_b a}$

Case 2:

If $f(n)=\Theta(n^{\log_b a})$*, then* $T(n)=\Theta(n^{\log_b a} \log n)$

f(n) and $n^{\log_b a}$ **_grows at similar rates_**

Case 3:

If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ *for some constant* $\varepsilon > 0$*, and if* **_af(n/b)<=cf(n)_** *for some constant* $c < 1$ *and all sufficiently large n, then* $T(n)=\Theta(f(n))$

f(n) grows polynomially faster than $n^{\log_b a}$

Solving recurrences: Master method Ref: Textbook R2

<u>Case 2 : (Generalisation):</u>

If there is a constant k >= 0, such that f(n) is $\Theta(n^{\log_b a} \log^k n)$, then T (n) is $\Theta(n^{\log_b a} \log^{k+1} n)$

Example:

$T(n)= 2T(n/2) + n \log n$

$a=2, b=2$ $f(n)=n\log n$

$n^{\log_b a} = n$

f(n) is asymptotically larger than $n^{\log_b a}$,B ut it is not polynomially larger.

So no standard case of master theorem applies.

It belongs to case 2 general case.

$f(n) = \Theta(n^{\log_b a)} \log^k n) = \Theta(n^{\log_b a} \log^1 n)$

So $T(n)= \Theta(n \log^2 n)$

Solving recurrences: Master method

Example 1 : $T(n) = 2T(n/2) + n$

Sol:

Extract a=2, b=2 and f(n) =n

Determine $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Compare $n^{\log_b a} = n$

$f(n) = n$

Thus case 2: evenly distributed because

$$f(n) = \theta(n)$$

$$T(n) = \theta\,(n^{\log_b a}\log(n))$$

$$= \theta\,(n^1 \log(n))$$

$$= \theta\,(n\log n)$$

Solving recurrences: Master method

Example 2 : $T(n)=9T(n/3)+n$

$a=9$ $b=3$ and $f(n)=n$

Determine $n^{\log_b a} = n^{\log_3 9} = n^2$

Compare: $n^{\log_b a} = n^2$

$f(n) = n$

Thus case1; (express $f(n)$ in terms of $n^{\log_b a}$) because $f(n)= O(n^{2-\varepsilon})$

$T(n)= \theta\ (n^{\log_b a}) = \theta(n^2)$

Solving recurrences: Master method

Example 3 : $T(n) = 3T(n/4) + n\log n$

a= 3,

b=4,

f(n) = nlogn

Determine; $n^{\log_b a} = n^{\log_4 3}$ $\log_4 3 < 1$

Compare: $n^{\log_b a}$ and f(n)

$n^{\log_4 3} <= n\log n$ f(n) is asymptotically and polynomially larger

Thus case 3, but we have to check the reqularity condition!

The following should be true:

af(n/b) < = cf(n) where c<1

a(n/b) log (n/b) < = cf(n)

=> 3(n/4) log(n/4) <=c n log n

3/4nlog(n/4) <= c .n log n,

this is true for c=3/4 Hence.T(n) = $\theta\ (n log(n))$

Case Study: Analyzing Algorithms

- **Computing the prefix averages of a sequence of numbers**.

  The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{X}$ is average of the first

  ($\boldsymbol{i}$ + 1) elements of $\boldsymbol{X}$**:**

  $\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}])/(i+1)$
- **Applications**
  - **Runtime analysis example:**

    Two algorithms for prefix averages

Case Study: Analyzing Algorithms

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm ***prefixAverages1***(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Algorithm ***prefixAverages1*** runs in $O(n^2)$ time

Case Study: Analyzing Algorithms

- The following algorithm computes prefix averages in linear time by keeping a running sum

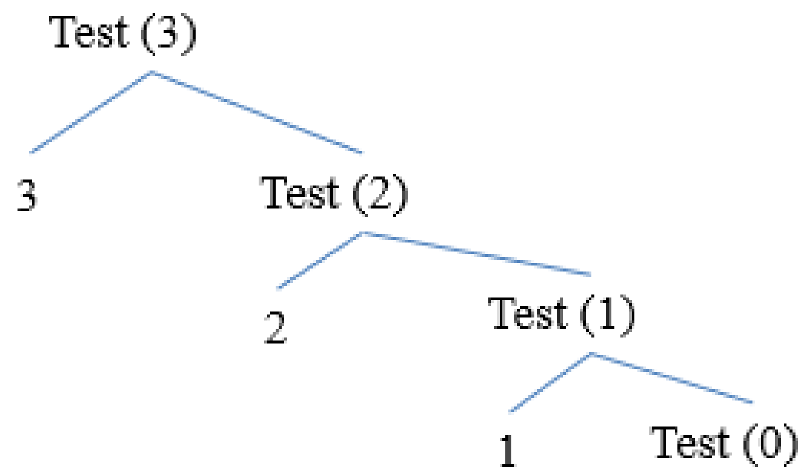
Algorithm *prefixAverages2*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Algorithm ***prefixAverages2*** runs in $O(n)$ time

Homework Problems

Solving recurrences : Iterative Method

```
void test(int n)
{
        if(n>0)
        {
        printf("%d",n);
        test(n-1);
        }
}
```



$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

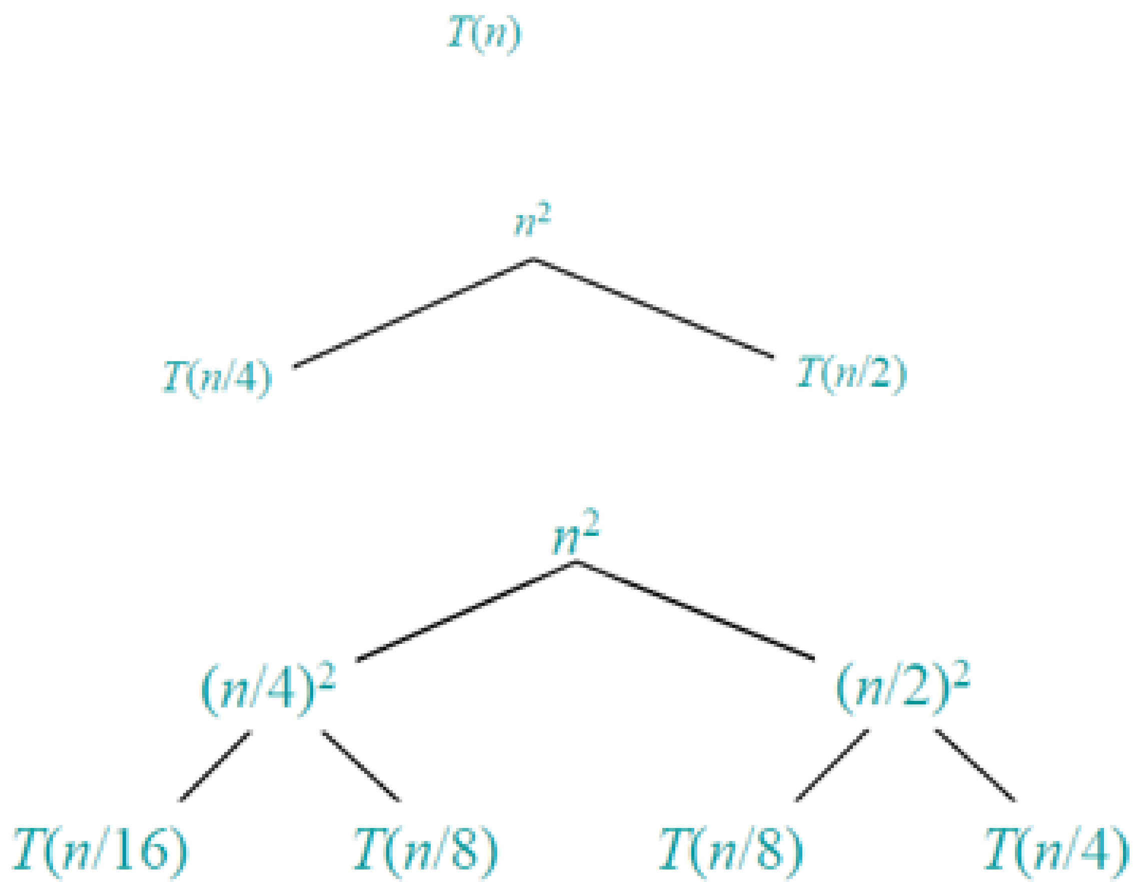
Solving recurrences : Iterative Method

```
Void Test (int n)
{
        if(n>1)
        {
                for (i=1;i<n;i++)
                {
                        stmt;
                }
                Test(n/2);
                Test(n/2);
        }
}
```

$$T(n) = \begin{cases} 0, & n=1 \\ 2T(n/2)+n & n>1 \end{cases}$$

Solving Recurrences-Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Solving Recurrences-Recursion Tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

n^2 .... n^2

$(n/4)^2$ $(n/2)^2$ .... $\frac{5}{16}n^2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$ .... $\frac{25}{256}n^2$

$\vdots$

$O(1)$

$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$$= O(n^2)$$ *geometric series*

Master method Problems

- $T(n)=9T(n/3)+n$
- $T(n)=T(2n/3)+1$
- $T(n)=3T(n/4)+n\log n$
- $T(n)=2T(n/2)+ n \lg n$
- $T(n)=8T(n/2)+\Theta(n^2)$

THANK YOU!

BITS Pilani
Hyderabad Campus