

Recurrence Relation

-MS. SWATI VARMA

Lecture Outcomes

Understand Recurrence Relation

Calculate complexity using following methods

- Recursive tree
- Substitution method
- Master's method

Recurrence Relation

A **recurrence** relation is an **equation** that recursively **defines a sequence** of values and each term of the sequence is defined as a function of the **preceding** terms.

Or

A **recurrence** relation is an **equation** that expresses **elements** of a sequence as a **function of the preceding ones**.

E.g.

1) $n! = n (n-1)!$

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

Solving Recurrence Relation

3 methods:

1. Substitution method
2. Recursive tree
3. Master's method

Recursive Tree Method

Recursive Tree Method

- Recursion tree is a method for solving the recurrence relations
- A recursion tree is a tree where each node represents the cost of a certain recursive sub problem
- We sum up the values in each node to get the cost of the entire algorithm
- A recursion tree is useful for visualizing what happens when a recurrence is iterated.
- They can be useful in gaining the intuition, but they are not a proof.

Recursive Tree Method

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

Total time: $T(n) = T(n-1) + 2$ or $T(n) = T(n-1) + 1$

Recursive Tree Method

```
Test(int n)
```

```
{
```

```
    if (n>0)
```

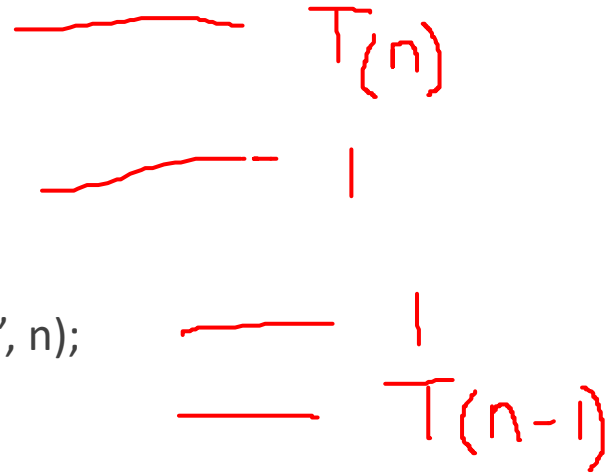
```
    {
```

```
        printf("%d", n);
```

```
        Test(n-1);
```

```
    }
```

```
}
```



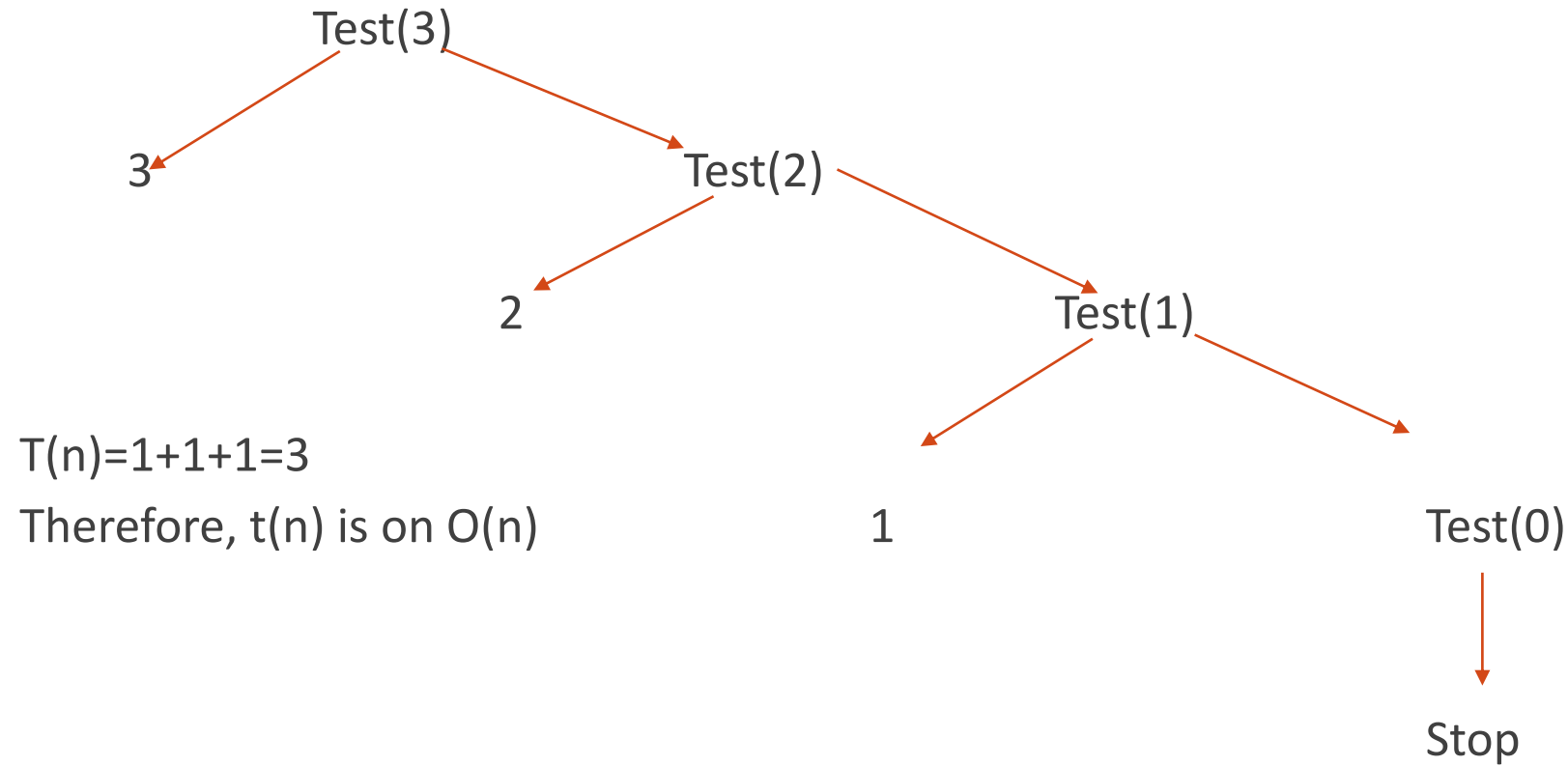
Total time: $T(n) = T(n-1) + 2$ or $T(n) = T(n-1) + 1$

Recursive Tree Method

Recurrence Relation

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

Recursive Tree Method



Recursive Tree Method

Test(int n)

```
{
    if (n>0)
    {
        for(i=0;i<n;i++)
        {
            printf("%d", i);
        }
        Test(n-1);
    }
}
```

Total time: $T(n) = T(n-1) + 2n + 2$ or $T(n) = T(n-1) + n$

Recursive Tree Method

Test(int n)	—	$T(n)$
{		
if (n>0)	—	1
{		
for(i=0;i<n;i++)	—	$n+1$
{		
printf("%d", i);	—	n
}		
Test(n-1);	—	$T(n-1)$
}		
}		

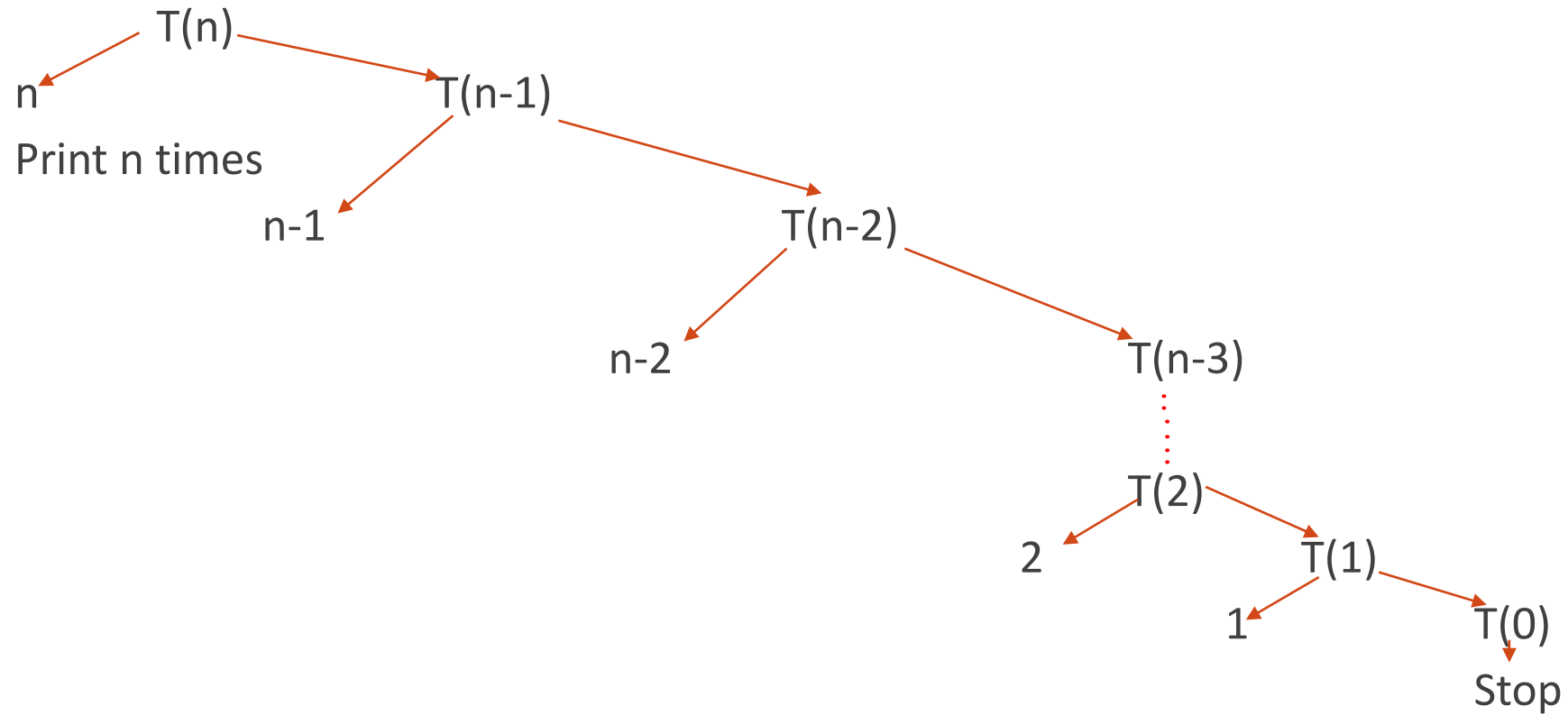
Total time: $T(n) = T(n-1) + 2n + 2$ or $T(n) = T(n-1) + n$

Recursive Tree Method

Recurrence Relation

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

Recursive Tree Method



Recursive Tree Method

Sum up:

$$T(n) = 0 + 1 + 2 + \dots + (n-2) + (n-1) + n$$
$$= n(n+1)/2$$

Therefore, $T(n)$ is in $O(n^2)$

Recursive Tree Method

Test(int n)

```
{
    if (n>0)
    {
        for(i=1;i<n;i=i*2)
        {
            printf("%d", i);
        }
        Test(n-1);
    }
}
```

Total time: $T(n) = T(n-1) + 2\log n + 2$ or $T(n) = T(n-1) + \log n$

Recursive Tree Method

```
Test(int n)
```

```
{
```

```
    if (n>0)
```

```
    {
```

```
        for(i=1;i<n;i=i*2)
```

```
        {
```

```
            printf("%d", i);
```

```
        }
```

```
        Test(n-1);
```

```
    }
```

```
}
```

— $T(n)$

— 1

— $\log n + 1$

— $\log n$

— $T(n-1)$

Total time: $T(n) = T(n-1) + 2\log n + 2$ or $T(n) = T(n-1) + \log n$

Recursive Tree Method

Iteration	i
1	1
2	2
3	4
4	8
5	16
6	32
...	...

So after **k iterations**:

$$i = 2^k$$

The loop stops when i becomes approximately n.

Therefore:

$$2^k \approx n$$

Taking log on both sides:

$$k = \log_2 n$$

So the loop executes approximately: $\log_2 n$ times

Recursive Tree Method

for(i=0;i<n;i=i*2)

i	
1*2= 2	2 ¹
2*2=4	2 ²
4*2=8	2 ³
8*2=16	2 ⁴
16*2=32	2 ⁵
32*2=64	2 ⁶
64*2=128	2 ⁷
128*2=256	2 ⁸
256*2=512	2 ⁹
512*2=1024	2 ¹⁰

For 1000, 10 times

So basically it runs (**log n**) times

$$2^x = 1024$$

$$X = \log_2 1024$$

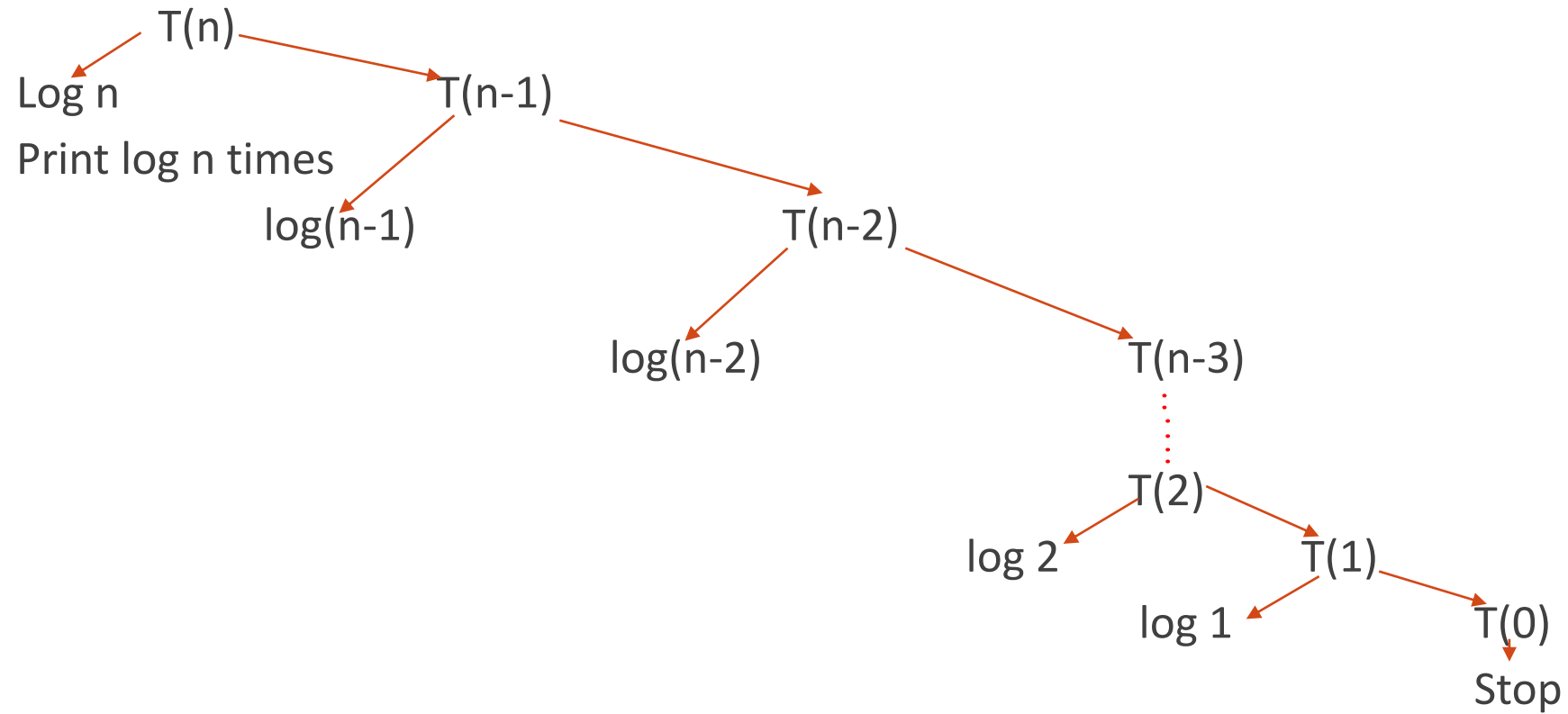
$$X = \log 1024 / \log 2$$

$$X = 10$$

Recursive Tree Method

$$\begin{array}{ll} T(n) = T(n-1) + \log n & n > 0 \\ = 1 & n = 0 \end{array}$$

Recursive Tree Method



Recursive Tree Method

Therefore,

$$T(n) = \log 1 + \log 2 + \dots + \log (n-2) + \log (n-1) + \log n$$

$$\begin{aligned} T(n) &= \log(1 * 2 * 3 * \dots * n-2 * n-1 * n) \\ &= \log(n!) \end{aligned}$$

Upper bound of $n!$ is n^n

Therefore, upper bound of $\log(n!)$ is $\log(n^n)$ i.e. $n \log n$

Therefore $T(n)$ is in $O(n \log n)$

Recurrence Relation

How to directly get the recurrence relation?

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

$$T(n) = T(n-1) + n \text{ ----- } O(n^2)$$

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

If $T(n-1) + n^2$ then

$$O(n^3)$$

Master theorem

- Used in analysis of divide and conquer algorithm
- Not all recurrence relations can be solved with the use of master theorem.

Consider the following problem which can be solved using recursive algorithm

Procedure $T(n$: size of problem) defined as:

If $n < 1$ then exit

Do work of amount $f(n)$

$T(n/b)$

$T(n/b)$

.....repeat for a total of a times.

$T(n/b)$

End procedure

Master theorem

- We divide the problem into no. of sub problems recursively
- Each sub problem is of size n/b
- Such algorithms can be represented as a recurrence relation

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

- Master theorem calculates complexity in θ notation

Generic form

Master theorem concerns recurrence relations of the form

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

where $a \geq 1$, $b > 1$

n - size of the problem

a - no. of sub problems

n/b - size of each sub problem

$f(n)$ – cost of work done outside the recursive calls, which includes cost of merging the solution.

It is possible to determine an asymptotic tight bound in 3 cases.

Master theorem

Case 1: Generic form

If $f(n) = \theta(n^c)$ where $c < \log_b a$ then $T(n)$ is in $\theta(n^{\log_b a})$

E.g.

$$T(n) = 8T(n/2) + 1000n^2$$

Compare with generic form: $T(n) = aT(n/b) + f(n)$

Therefore, $a=8$, $b=2$, $f(n) = 1000n^2$, $c=2$ because $f(n) = n^c$

$\log_b a = \log_2 8 = 3$, therefore $c < 3$

It follows from the first case of the master theorem that

$$T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_2 8}) = \theta(n^3)$$

Case 3: Generic form

If it is true that $f(n) = \theta(n^c)$ where $c > \log_b a$, then $T(n)$ is in $\theta(f(n))$

E.g. $T(n) = 2T(n/2) + n^2$

Compare with generic form: $T(n) = a T(n/b) + f(n)$

$A=2$, $b=2$, $f(n)=n^2$, $c=2$ because $f(n) = n^c$

Here $c > \log_b a$

Therefore case 3

Therefore $T(n) = \theta(n^2)$

Case 2: Generic form

If it is true for some constant $k \geq 0$ that $f(n) = \theta(n^c \log^k n)$ where $c = \log_b a$
then $t(n) = \theta(n^c \log^{k+1} n)$

E.g. $T(n) = 2T(n/2) + 10n$

$a=2, b=2, f(n)=10n$, therefore $c=1$

$\log_2 2 = 1 = c$ therefore case 2

Therefore $T(n) = \theta(n^1 \log^1 n)$ ($k=0$)
 $= \theta(n \log n)$

Examples

1. $T(n) = 3T(n/2) + n^2$

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

3. $T(n) = 16T(n/4) + n$

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

Solutions

1. $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Case 1)
4. $T(n) = 2T(n/2) + n \log n \Rightarrow T(n) = n \log^2 n$ (Case 2)