Chapter 3: Recurrence Relation

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n > 1 \end{cases}$$

Prof. Anand Singh Jalal

Department of Computer Engineering & Applications

Recurrence: Introduction

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- An algorithm contains a recursive call to itself,
- We can often describe its running time by a recurrence equation or recurrence.
- Example:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + cn & n > 1 \end{cases}$$

Recurrence: Introduction …

Solving Methods

- Iterative Method (back-substitution)
- Master Method
- Recursion Tree
- Substitution Method

Recurrence: Iterative Method

- Convert the recurrence into a summation and try to bound it using known series
  - Iterate the recurrence until the initial condition is reached.
  - Expand the recurrence k times
  - Work some algebra to express as a summation
  - Use *back-substitution* (unrolling and summing) to express the recurrence in terms of *n* and the initial (boundary) condition.

Recurrence: Iterative Method

Analysis of Recursive Factorial method

Example1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

T(0) = c (1)
T(n) = b + T(n - 1) (2)
= b + b + T(n - 2) by subtituting T(n – 1) in (2)
= b +b +b + T(n - 3) by substituting T(n – 2) in (2)

= kb + T(n - k)

The base case is reached when n – k = 0 → k = n, we then have:

T(n) = nb + T(n - n)
= bn + T(0)
= bn + c

Therefore the method factorial is O(n)

```
long factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial (n – 1);
}
```

Recurrence: Iterative Method

Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                                          int low, int high) {
    if (low > high)
        return -1;
    else {
        int middle = (low + high)/2;
        if (array[middle] == target)
            return middle;
        else if(array[middle] < target)
            return binarySearch(target, array, middle + 1, high);
        else
            return binarySearch(target, array, low, middle - 1);
      }
}
```

The recurrence relation for the running time of the method is:

$T(1) = 1$ if $n = 1$ (one element array)

$T(n) = T(n / 2) + b$ if $n > 1$

Recurrence: Iterative Method

Without loss of generality, assume n, the problem size, is a multiple of 2, i.e., $n = 2^k$

Expanding:

$T(1) = 1 \qquad (1)$

$T(n) = T(n / 2) + b \qquad (2)$

$= [T(n / 2^2) + b] + b = T(n / 2^2) + 2b$ by substituting T(n/2) in (2)

$= [T(n / 2^3) + b] + 2b = T(n / 2^3) + 3b$ by substituting $T(n/2^2)$ in (2)

= ……..

$= T(n / 2^k) + kb$

The base case is reached when $n / 2^k = 1 \rightarrow n = 2^k \rightarrow k = \log_2 n$,

we then have:

$T(n) = T(1) + b \log_2 n$

$= 1 + b \log_2 n$

Therefore, Recursive Binary Search is **O(log n)**

Recurrence: Iterative Method

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

- $T(n) =$ $c + T(n\text{-}1)$ $=$ $c + c + T(n\text{-}2)$
  $=$ $2c + T(n\text{-}2)$ $=$ $2c + c + T(n\text{-}3)$
  $=$ $3c + T(n\text{-}3)$ … $kc + T(n\text{-}k)$
  $=$ $ck + T(n\text{-}k)$
- So far for $n \geq k$ we have
  - $\boldsymbol{T(n) = ck + T(n\text{-}k)}$
- To stop the recursion, we should have
  - $n - k = 0 \rightarrow k = n$
  - $\boldsymbol{T(n) = cn + T(0) = cn}$ Thus in general $\boldsymbol{T(n) = O(n)}$

Recurrence: Iterative Method

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

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

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

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

  $= n + n-1 + n-2 + n-3 + T(n-4)$

  $= \ldots$

  $= n + n-1 + n-2 + n-3 + \ldots + (n-k+1) + T(n-k) = \sum_{i=n-k+1}^{n} i + T(n-k)$

  for $n \geq k$

- To stop the recursion, we should have $n - k = 0$ ➔ $k = n$

$$\sum_{i=1}^{n} i + T(0) = \sum_{i=1}^{n} i + 0 = \frac{n(n+1)}{2} \qquad T(n) = \frac{n(n+1)}{2} = O(n^2)$$

Recurrence: Master Method

T(n)=a T(n/b) + f(n) **where a>=1 and b >1**

f(n) is asymptotically positive function

1. **If f(n)=O($n^{\log_b a-\varepsilon}$) $for\ some\ constant\ \varepsilon > 0$**

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

2. **If f(n)= $\theta(n^{log_b\ a})$**

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

3. **If f(n)=$\Omega(n^{log_b\ a+\varepsilon})$ for some constant $\varepsilon > 0\ and$**

   $if\ a\ f\left(\frac{n}{b}\right) \leq c\ f(n)\ for\ some\ constant\ c < 1$

   $and\ all\ sufficiently\ large\ n,$

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

Recurrence: Master Method ...

Ex1. T(n) = 9 T(n/3) + n

Compare with standard formula T(n) = a T(n/b) + f(n)

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

Compute $n^{\log_b a} = n^{\log_3 9} = n^{2-\varepsilon}$

It is the form of Case I then T(n)=ɵ(n²)

1. If f(n)=O$(n^{\log_b a-\varepsilon})$

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

2. If f(n)= $\theta(n^{\log_b a})$

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

3. If f(n)=$\Omega(n^{\log_b a+\varepsilon})$

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

Recurrence: Master Method ...

Ex2: $T(n)=4T(n/2) + n^2$ $[T(n)=aT(n/b) + f(n)]$

$a=4,\ b=2,\ f(n)= n^2$

$n^{\log_b a} = n^{\log_2 4} = n^2 = f(n)$

1. If $f(n)=O(n^{\log_b a-\varepsilon})$

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

2. If $f(n)= \theta(n^{\log_b a})$

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

3. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$

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

It is the form of Case II then

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

$T(n) = \theta\ (n^{\log_2 4} \lg n)$

$\mathbf{T(n) = \theta\ (n^2 \lg n)}$

Recurrence: Master Method ...

Ex3: $T(n)=3T(n/2) + n^2$ $[T(n)=aT(n/b) + f(n)]$

$a=3, b=2, f(n)= n^2$

$n^{\log_b a} = n^{\log_2 3} = n^{1.5+\varepsilon} = f(n)$

where $\varepsilon = 0.5$

It is the form of Case III then

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

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

1. **If $f(n)=O(n^{\log_b a-\varepsilon})$**

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

2. **If $f(n)= \theta(n^{\log_b a})$**

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

3. **If $f(n)=\Omega(n^{\log_b a+\varepsilon})$**

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

Recurrence: Master Method ...

Special Case:

If f(n)=ɵ$(n^{\log_b a} \log^k n)$, where k>=0, then the master recurrence has solution:

$$T(n)=ɵ(n^{\log_b a} \log^{k+1} n)$$

Ex. T(n) = 2T(n/2)+n logn

a=2, b=2, k=1,

f(n)= $n^{\log_b a} \log^k n = n^{\log_2 2} \log^1 n$

=nlogn

So, T(n)= $ɵ(n^{\log_b a} \log^{k+1} n) = ɵ(n^{\log_2 2} \log^{1+1} n) = ɵ(n \log^2 n)$

Recurrence-Recursion Tree

Steps

- Convert the recurrence into a tree.
- Each node represents the cost of a single sub problem somewhere in the set of recursive function invocations.
- Sum the costs within each level of the tree to obtain a set of per-level costs.
- Sum all the per-level costs to determine the total cost of all levels of the recursion.

Recurrence-Recursion Tree …

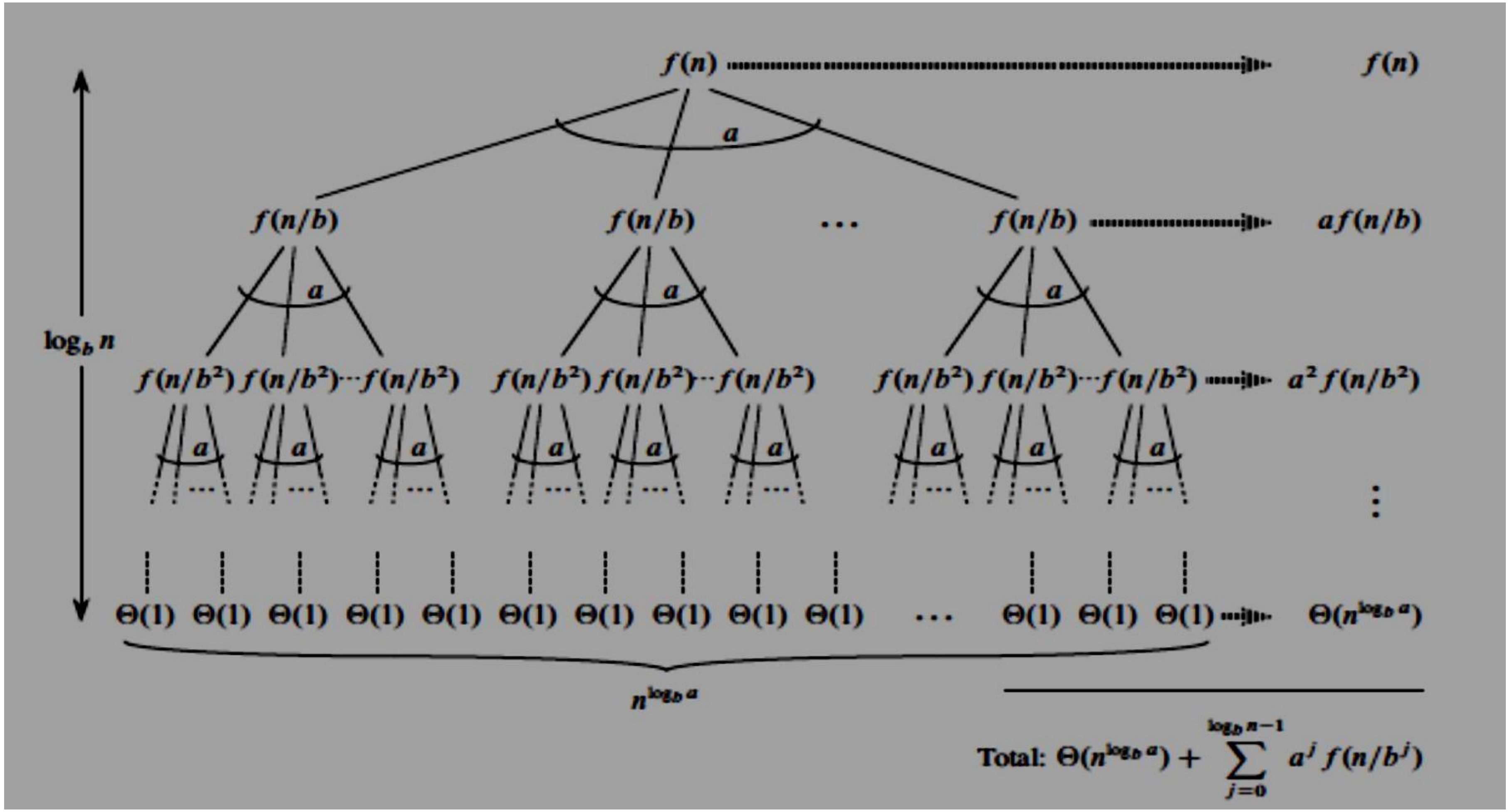
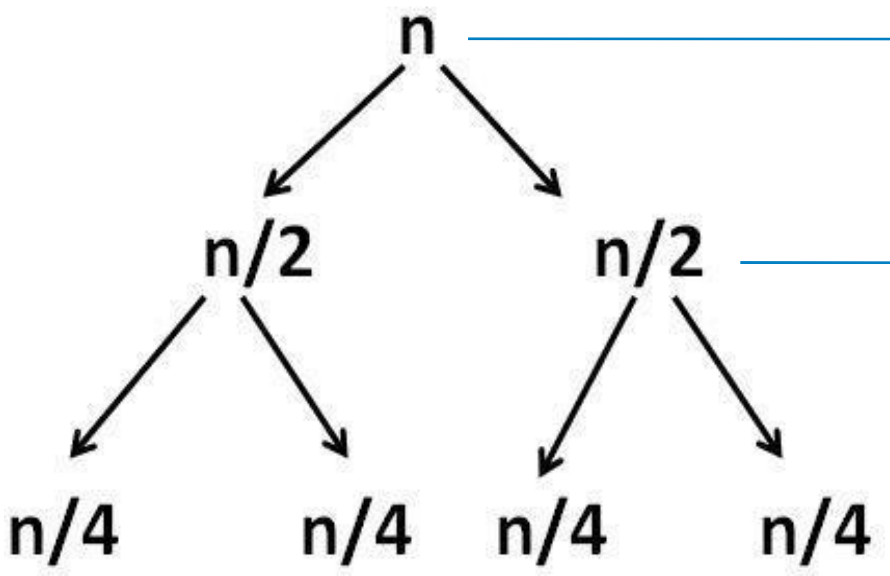


Figure 1: The recursion tree is generated by T(n)=aT(n/b)+f(n). The tree is complete a-ary tree with $n^{log_b a}$ leaves and height $\log_b n$

Recurrence-Recursion Tree …

$$T(n) = 2T(n/2) + c \qquad n>1$$
$$= 1 \qquad n=1$$



c

2c

4c

$T(n)= c + 2c + 2^2c + \ldots\ldots\ldots\ldots\ldots\ldots\ldots 2^kc$

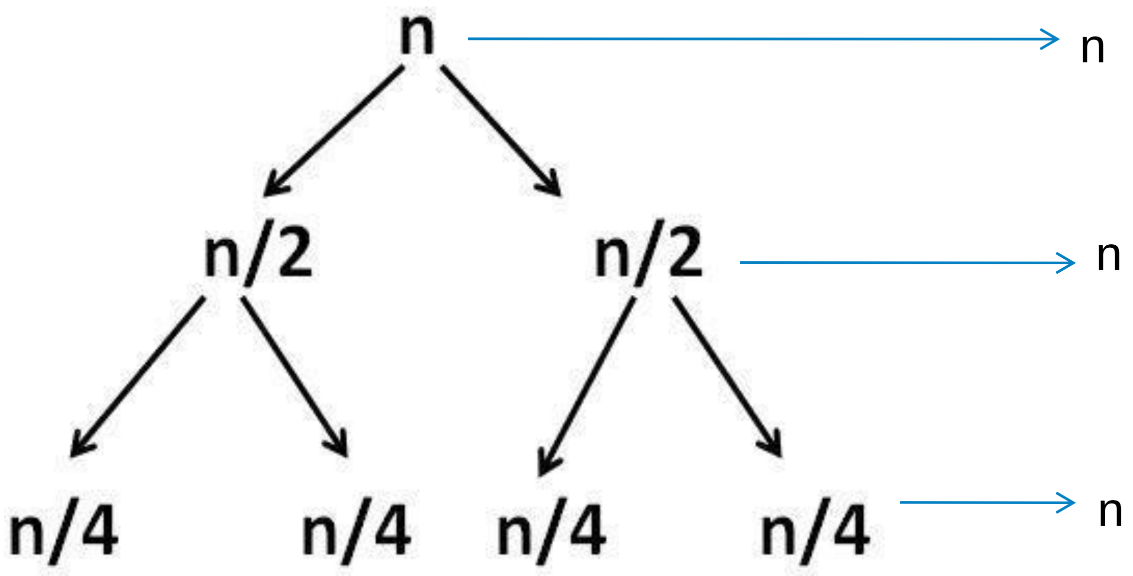
The recursion will end when $n/2^k=1$

So $T(n)= c(1+2+2^2+2^3\ldots\ldots\ldots 2^{k=}$

$$c\frac{(2^{k+1}-1)}{2-1} = c(2^{k+1}-1) = c(2n-1) = O(n)$$

Recurrence-Recursion Tree ...

$$T(n) = 2T(n/2) + n \qquad n>1$$
$$= 1 \qquad n=1$$



$T(n)= n + n + n + n$ …………………….upto k

where base case $n/2^k = 1$ => $n=2^k$ => $k=\log_2 n$

$T(n)= n*k=n*\log n$

$T(n) = O(n \lg n)$

Recurrence: Substitution Method

- Guess a bound and then use mathematical induction to prove our guess correct.
- The substitution method for solving recurrences comprises two steps:
  - ✓ Guess the form of the solution.
  - ✓ Use mathematical induction to find the constants and show that the solution works.

Recurrence: Substitution Method

- Substitution Method

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

Guess the solution is: $O(n \log n)$

Prove that $T(n) <= c. (n \log n)$ for $c>0$

$T(n)=2T(n/2)+n$……………………………….(1)

Compute for $T(n/2) <= c. (n/2).\log(n/2)$ and put into (1)

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

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

$T(n) <= c.n. \log n - c.n.\log 2 +n$

$T(n) <= c.n.\log n - c.n + n$ for $c>=1$

Recurrence: Substitution Method

- Substitution Method

$T(n) <= c.n.\log n - c.n + n$ for $c=1$

$T(n) <= 1.n.\log n - 1.n + n = n.\log n$

$T(n) <= 2.\ n.\ \log n - 2.\ n + n = n.\log n + n.\log n - n$

.

.

.

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

Any Questions ?

Dr. Anand Singh Jalal
Professor
Email: asjalal@gla.ac.in