WIA2005: Algorithm Design and Analysis

Lecture 3: Introduction to Algorithm Design & Analysis Fundamentals (Pt2)

Asmiza A. Sani

Semester 2, Session 2024/25

Learning Objectives

- The students will:
    - Understand and analyse time complexity of recursive algorithm using the following methods:
        - Back Substitution
        - Recursion Trees
        - Master Methods

Recursive Algorithm

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- The repeated application of a recursive function within itself to solve a problem.

- Recurrences go hand in hand with the divide-and-conquer paradigm
  - because they give a natural way to characterize the running times of divide-and-conquer algorithms.

Properties of Recursive Function

- To analyse running time complexity of a recursive function, we need to know (especially for back substitution and recursion tree):
  - Base Case - terminates scenario that does not use recursion to produce an answer
  - A Set of Conditions - reduces all other cases towards the Base Case

Example of recursive function and it’s recurrence relation

```
function1(n)
{
       if ( n > 0 )
           print n
           return function1(n – 1)
}
```

This function calls itself with a new n value (n-1).

Example of recursive function and it’s recurrence relation

EXAMPLE

- Factorial (denoted by !) is a function that calculate the product of all positive integers less than or equal to n.
  - n! = n x (n-1) x (n-2) x (n-3) x … x 3 x 2 x 1
  - 5! = 5 x x 4 x 3 x 2 x 1

```
factorial(n) {
     if n == 0
       return 1;
     else
       return n * factorial(n-1);
}
```

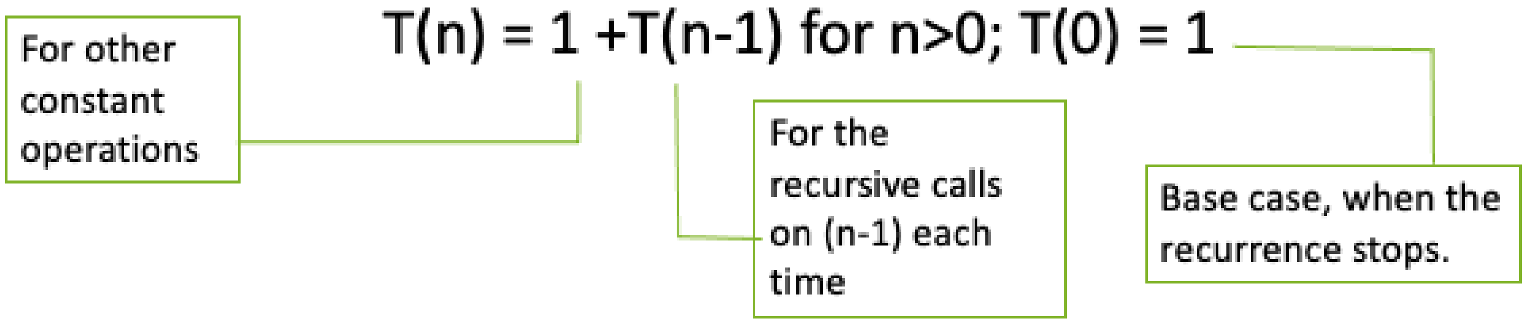
What is the recurrence relation for factorial(n) function?

Recurrence Relation

EXAMPLE

- Factorial (denoted by !) is a function that calculate the product of all positive integers less than or equal to n.
    - n! = n x (n-1) x (n-2) x (n-3) x … x 3 x 2 x 1
    - 5! = 5 x x 4 x 3 x 2 x 1

```
factorial(n) {
     if n == 0
       return 1;
     else
       return n * factorial(n-1);
}
```



T(n) = 1 +T(n-1) for n>0; T(0) = 1

For other constant operations

For the recursive calls on (n-1) each time

Base case, when the recurrence stops.

Analysing recursive algorithm

Methods for solving recurrences

- There are three methods for solving recurrences:
  - *Substitution method*, we guess a bound and then use mathematical induction to prove our guess correct.
  - *Recursion-tree method* converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
  - *Master method* provides bounds for recurrences of the form

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

    - where a ≥ 1, b > 1, and f (n) is a given function

Analysis of Recursive Algorithm: (1) Back Substitution Method

- The *back substitution method* for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.

Example 1: Back Substitution Method

T(n) = 1 +T(n-1) for n>0; T(0) = 1

Step 1: How can we write this in terms of summation?

- T(n) = 1 + T(n-1) – (1)

Step 2: Find the equation for the next call, (n-1)

- T(n-1) = 1 + T([n-1] -1)
  = 1 +T(n-2) – (2)

What happened here?
We are trying to find how T(n) is reduced to the base case.

Step 3: Repeat step 2 for (n-2)

- T(n-2) = 1 + T([n-2]-1)
  = 1 +T(n-3) – (3)

Example 1: Back Substitution Method

T(n) = 1 +T(n-1) for n>0; T(0) = 1

Step 4: Substitute (2) into (1)

- T(n)= 1 + [1 + T(n - 2)] – (4)
  = 2 + T(n-2)

Step 5: Substitute (3) into (4)

- T(n)= 2 + [1 + T(n-3)]
  = 3 + T(n-3)

Step 6: T(n) gradually decrease as the loop A(n) recurs. Base case T(0) = 1

- T(n) = k + T(n-k) – (5)

Example 1: Back Substitution Method

$T(n) = 1 + T(n-1)$ for $n>0$; $T(0) = 1$

Step 7: We know that the base case (where the recursion stops) is 1, therefore $T(n-k)$ $^{(\text{from } (5))}$ = 1 Find k.

- $n - k = 0$

  $k = n$

Step 8: Insert k into (5)

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

  $= (n) + T(0)$

  $= n$

We already know this is 1.

Step 9: Running time complexity.

- $T(n) = O(n)$

Example 2: Back Substitution Method

- Given the recursive function B is:

$$T(n) = n + T(n-1)\ ;\ n > 0 \text{ and } T(1) = 1$$

- Using substitution method, find the order of function B.

Example 2: Back Substitution Method

Step 1: How can we write this in terms of summation?

- $T(n) = n + T(n-1)$ – (1)

Step 2: Find the equation for the next call, (n-1)

- $T(n-1) = [n-1] + T([n-1] -1)$
  $= (n-1) + T(n-2)$ – (2)

Step 3: Repeat step 2 for (n-2)

- $T(n-2) =[n-2] + T([n-2]-1)$
  $= (n-2) +T(n-3)$ – (3)

Example 2: Back Substitution Method

Step 4: Substitute (2) into (1)

- T(n) = n + [(n-1) + T(n-2)] – (4)

Step 5: Substitute (3) into (4)

- T(n) = n + (n-1) + [(n-2) + T(n-3)]

Step 6: T(n) gradually decrease as the loop A(n) recurs. Base case T(1) = 1

- T(n) = n + (n-1) + (n-2) + (n-3) + … [n-(k-1) + T(n-k)] – (5)

Example 2: Back Substitution Method

Step 7: We know that the base case (where the recursion stops) is 1, therefore T(n-k) $^{\text{(from (5))}}$ *= 1 Find k.*

- $n - k = 1$

  $k = n - 1$

We already know this is 1.

Step 8: Insert k into (5)

$T(n) = n + (n-1) + (n-2) + (n-3) + \ldots + [n-([n-1]-1) + T(n-[n-1])]$

$= n + (n-1) + (n-2) + (n-3) + \ldots + 2 + T(1)]$

$= n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$

$= [n(n+1)]/2$

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

Arithmetic series (1)

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

Step 9: Running time complexity.

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

Analysis of Recursive Algorithm:
(2) Recursion Tree Method

- In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- We need to determine the total cost, then sum the costs within each level of the tree to obtain a set of per-level costs is calculated, and then we sum all the per-level costs.
- A recursion tree is best used to generate a ***good guess***, which you can then verify by the substitution method (or master method).

Example of Recursion-Tree Method (1)

- Let say, we have the following function:

```
function2(n)
    if n == 1
        return 1
    else
        return function2(n/2) + function2(n/2)
```

- We can write the recursive function relation as:

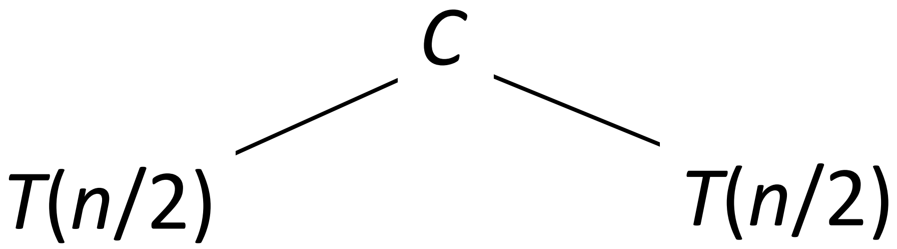
$$T(n) = 2T(n/2) + C$$

Example of Recursion-Tree Method (1)

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

Example of Recursion-Tree Method (1)

Step 1: Recursive call number 1

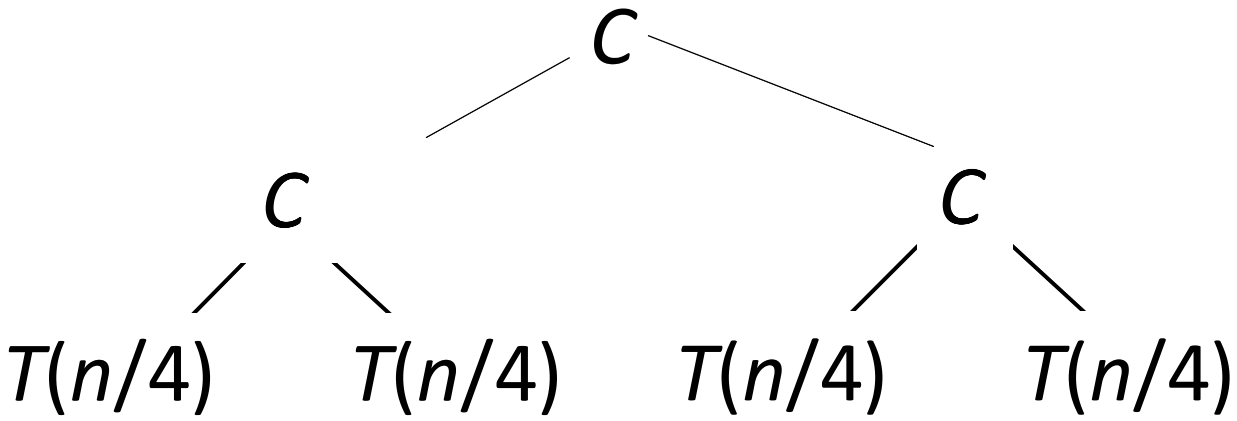


C

$T(n/2)$ $T(n/2)$

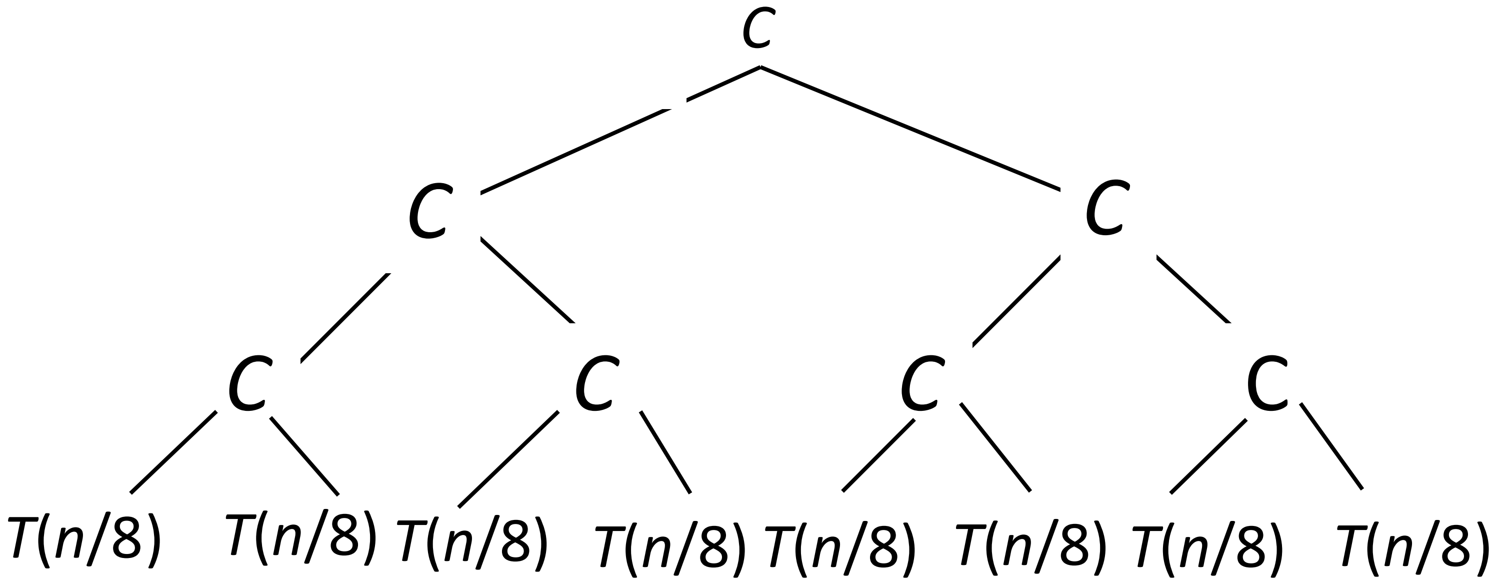
Example of Recursion-Tree Method (1)

Step 2: Recursive call number 2



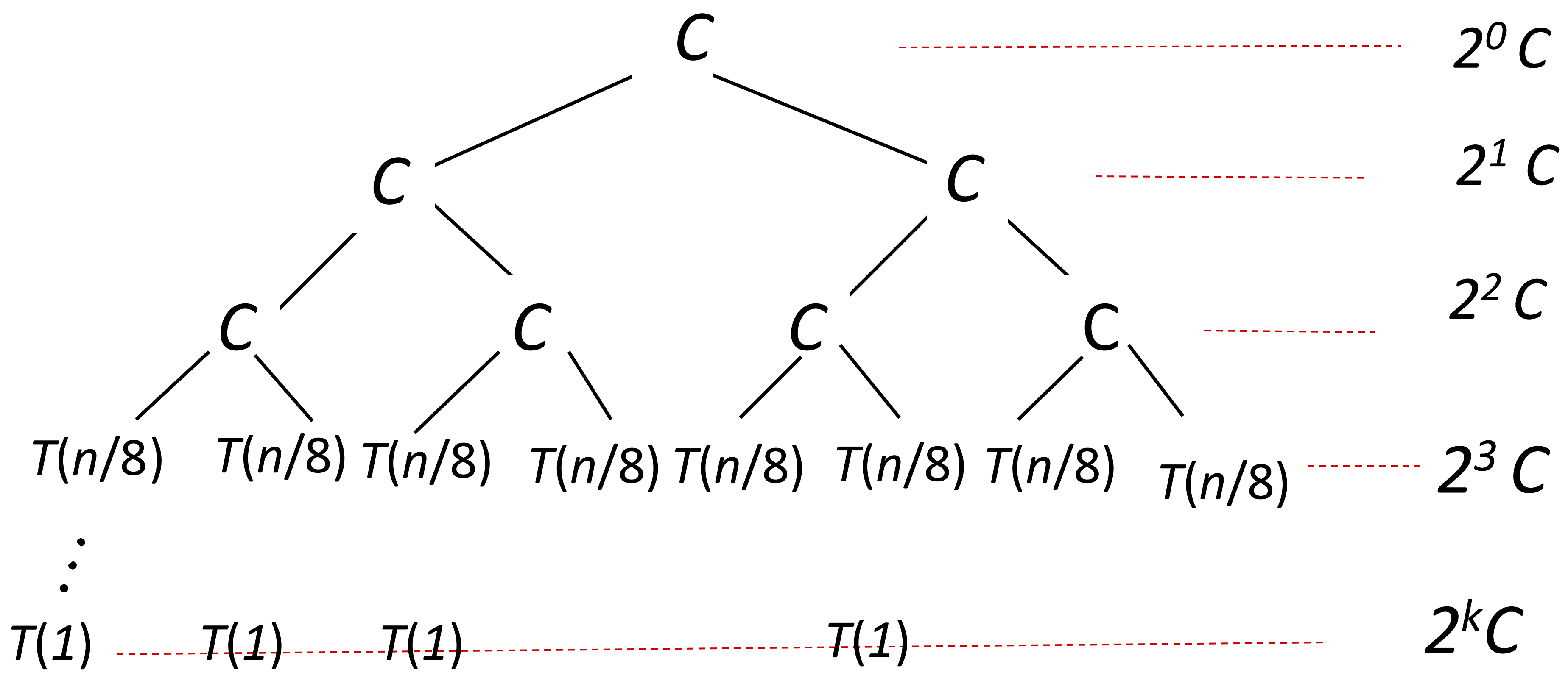
Example of Recursion-Tree Method (1)

Step 3: Recursive call number 3



Example of Recursion-Tree Method (1)

Step 4: Recursive call number n



Example of Recursion-Tree Method (1)

Step 5: Calculate the total cost for each call.

- $T(n) = c(2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^k)$

$k = \log_2 n$

Geometric series (6)

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a-1} \quad \text{for } a \neq 1$$

Step 6: Running time complexity

- $T(n) = O(n)$

$(2^{\log_2 n} - 1)$

Example of Recursion-Tree Method (2)

- Given the recursive function relation:

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

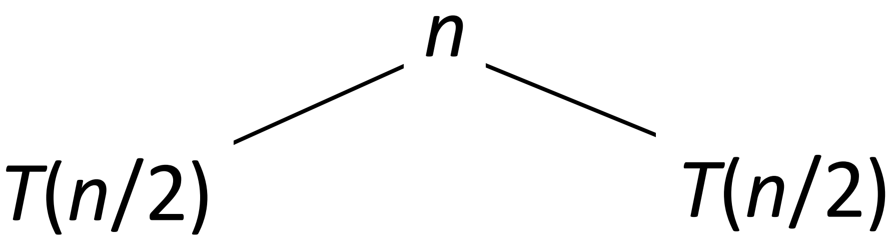
- Using recursion tree method, find the order of function.

Example of Recursion-Tree Method (2)

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

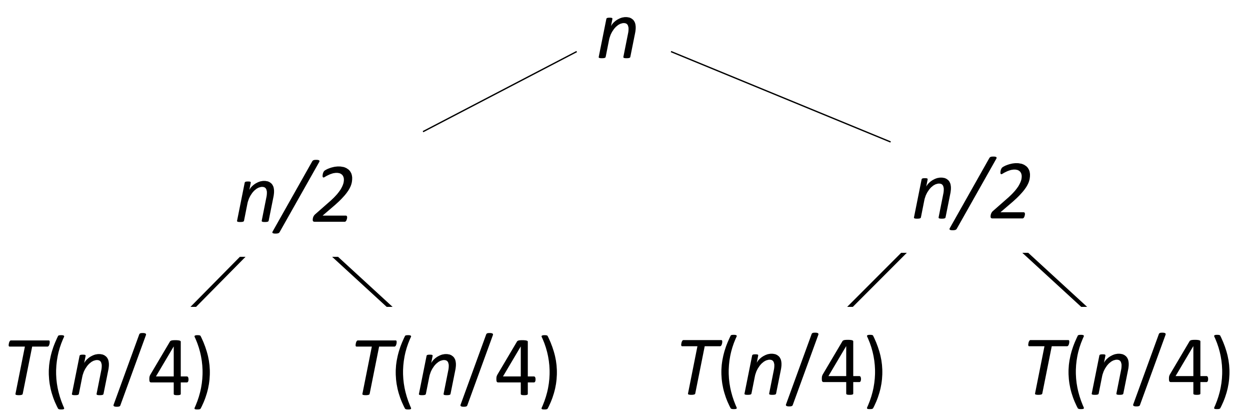
Example of Recursion-Tree Method (2)

Step 1: Recursive call number 1



Example of Recursion-Tree Method (2)

Step 2: Recursive call number 2



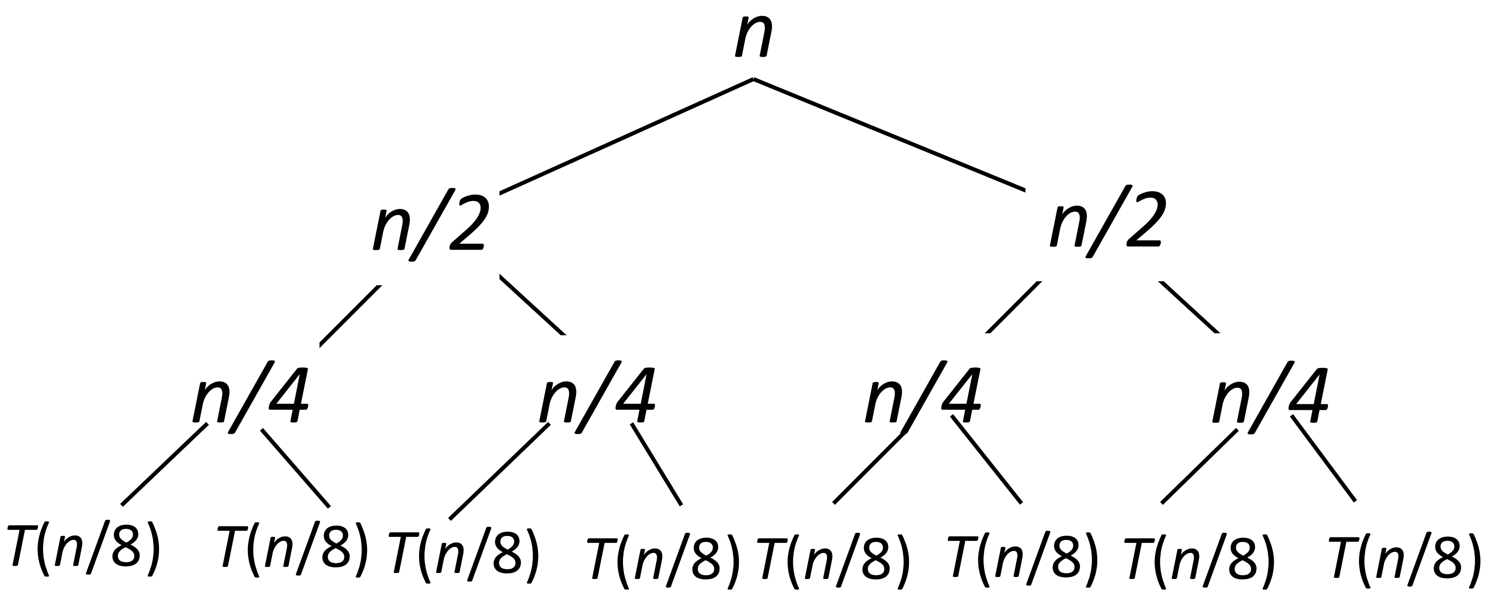
n

$n/2$ $n/2$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

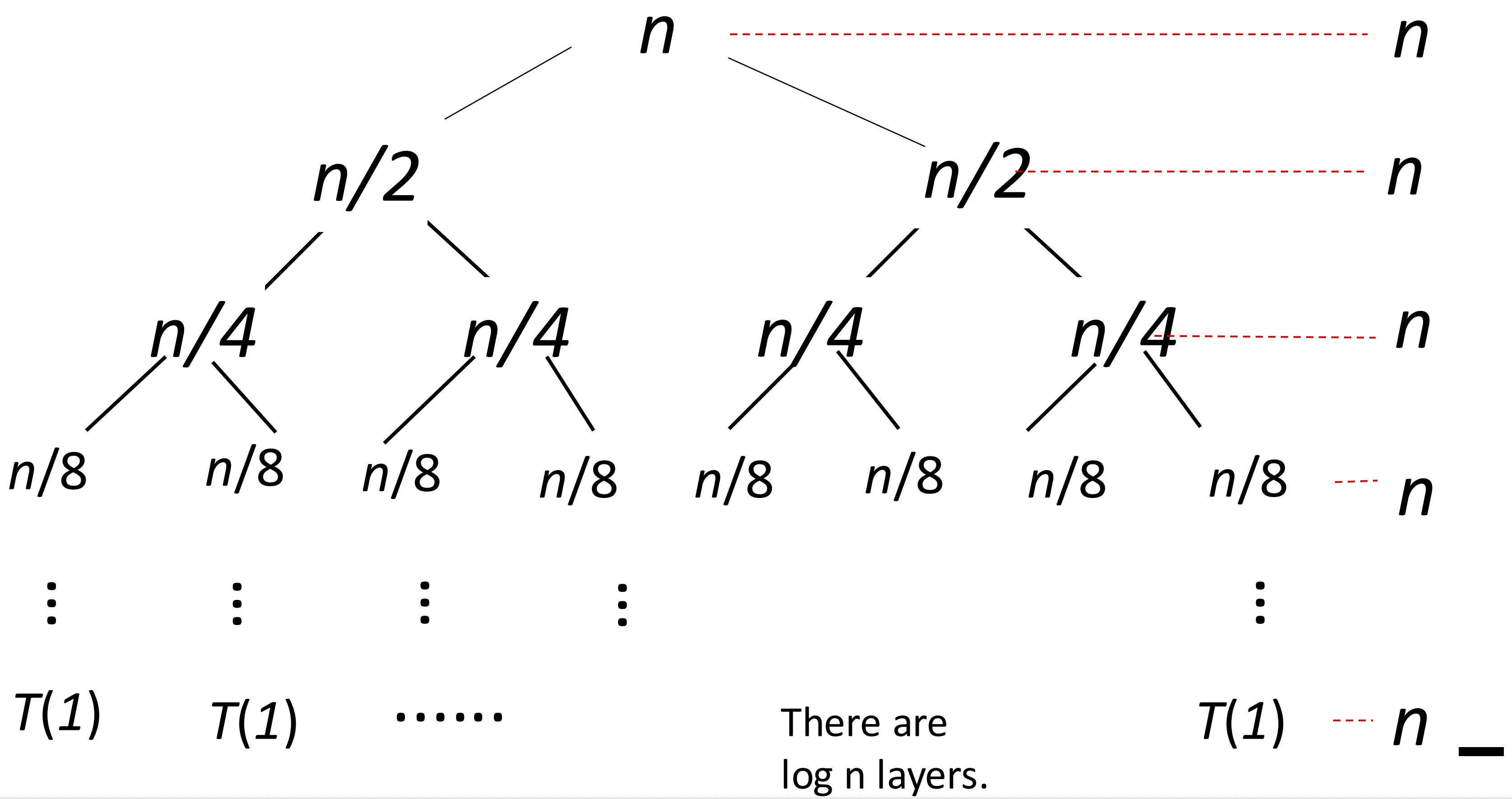
Example of Recursion-Tree Method (2)

Step 3: Recursive call number 3



Example of Recursion-Tree Method (2)

Step 4: Recursive call number n



Example of Recursion-Tree Method (2)

Step 6: Calculate the cost for each call.

- T(n) = n + n + n + n + … + n – for k times (layers)
  - Since there are (log n) layers, the number of k is log n.

Step 7: Running time complexity

- T(n) = n x (log n)

  = O(n log n)

Analysis of Recursive Algorithm:
(3) Master Method

- The master method provides a “cookbook” method for solving recurrences of the form

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

  - where a ≥ 1 and b > 1 are constants and f(n) is an asymptotically positive function.

Conditions for Master Method

Main pattern of recurrence relations:

$$\mathrm{T(n)} = \mathrm{aT}\left({}^{\mathrm{n}}/_{\mathrm{b}}\right) + \underbrace{\Theta\left(\mathrm{n}^{\mathrm{k}}\log^{\mathrm{p}}\mathrm{n}\right)}_{\mathrm{f(n)}}$$

Where, $\mathrm{a} \geq 1\,,\mathrm{b} > 1, \mathrm{k} \geq 0, \mathrm{p} =$ real number

Conditions for Master Method

$$\mathrm{T(n) = aT\left({}^{n}/_{b}\right) + \Theta\left(n^{k}log^{p}n\right)}$$

where $,\mathrm{a} \geq 1\ ,\mathrm{b} > 1, \mathrm{k} \geq 0, \mathrm{p} =$ real number

Case 1: If $\log_{\mathrm{b}} \mathrm{a} > k$ then, $T\ (n) = \Theta(n^{\log_b a})$

Case 2: If $\log_{\mathrm{b}} \mathrm{a} =\ k$ then

a) If $p > -1$, then, $T\ (n) = \Theta\left(n^{k}\ log^{p+1}\ n\right)$

b) If $p = -1$, then, $T\ (n) = \Theta\left(n^{k}\ log\ \log n\right)$

c) If $p < -1$, then, $T\ (n) = \Theta\left(n^{k}\right)$

Case 3: If $\log_{\mathrm{b}} \mathrm{a} < k$ then

a) If $p \geq 0$, then, $T\ (n) = \Theta\left(n^{k}\ log^{p}\ n\right)$

b) If p <0, then, $T\ (n) = \mathrm{O}\left(n^{k}\right)$

Example Case 1

a) $T(n) = 16T(^{n}/_{4}) + \mathrm{n}$

b) $T(n) = 3T(^{n}/_{2}) + \mathrm{n}$

Example Case 1

a) $T(n) = 16T(^{n}/_{4}) + \mathrm{n}$

Solution: a=16 , b=4 , f(n) = n, k = 1, p = 0
$\log_4 16 >$ 1 – Case 1: $\mathbf{log_b\ a} > \boldsymbol{k}$

Therefore:
T(n) =Θ(n^2)

b) $T(n) = 3T(^{n}/_{2}) + \mathrm{n}$

Solution: a=3 , b=2 , f(n) = n, k = 1, p = 0
$\log_2 3 >$ 1 – Case 1: $\mathbf{log_b\ a} > \boldsymbol{k}$

Therefore:
T(n) =Θ$\left(\mathrm{n}^{\log_2 3}\right)$

Example Case 2

a) $T(n) = 4T(^{n}/_{2}) + n^2$

b) $T(n) = 2T(^{n}/_{2}) + n \log n$

c) $T(n) = 2T(^{n}/_{2}) + \frac{n}{\log n}$

d) $T(n) = 2T(^{n}/_{2}) + n \log^{-2} n$

Example Case 2

a) $T(n) = 4T(^n/_2) + n^2$

Solution: a=4 , b=2 , f(n) = n^2, k = 2, p = 0
$\log_2 4$ = 2 and p > -1 – Case 2: $\mathbf{log_b\, a} = \boldsymbol{k}$

Therefore:
T(n) = Θ$(n^2\, log\, n)$

b) $T(n) = 2T(^n/_2) + n\, log\, n$

Solution: a=2 , b=2 , f(n) = n log n, k = 1, p = 1
$\log_2 2$ = 1 and p > -1 – Case 2: $\mathbf{log_b\, a} = \boldsymbol{k}$

Therefore:
T(n) = Θ(n $\log^2$n)

Example Case 2

c) $T(n) = 2T({}^{n}/_{2}) + \frac{n}{\log n}$

Solution: a=2 , b=2 , f(n) = n log n, k = 1, p = -1

$\log_2 2 =$ 1 and p = -1 – Case 2: $\mathbf{log_b\ a} = \boldsymbol{k}$

Therefore:

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

d) $T(n) = 2T({}^{n}/_{2}) + n \log^{-2} n$

Solution: a=2 , b=2 , f(n) = n log n, k = 1, p = -2

$\log_2 2 =$ 1 and p < -1 – Case 2: $\mathbf{log_b\ a} = \boldsymbol{k}$

Therefore:

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

Example Case 3

a)$\mathrm{T(n)} = 6\mathrm{T}(^{\mathrm{n}}/_{3}) + \mathrm{n}^2\log \mathrm{n}$

Solution: a=6 , b=3 , f(n) = n^2 log n, k = 2, p = 1

$\log_3 6 <$ 2 and p ≥ 0 – Case 3: $\mathbf{log_b\, a < \boldsymbol{k}}$

Therefore:

$\mathrm{T(n)}$ = Θ(n^2 log n)

b) $\mathrm{T(n)} = 7\mathrm{T}(^{\mathrm{n}}/_{49}) + \mathrm{n}^2 \log \mathrm{n}$

Solution: a=7 , b=49 , f(n) = n^2 log n, k = 2, p = 1

$\log_{49} 7 <$ 2 and p ≥ 0 – Case 3: $\mathbf{log_b\, a < \boldsymbol{k}}$

Therefore:

$\mathrm{T(n)}$ = Θ(n^2 log n)

Example (Others)

- Using the masters theorem, solve the following problem:

$$T(n) = 0.5T(n/2) + 1/n$$

$a <= 1$, therefore masters theorem cannot be applied to solve the problem.

Inadmissible equation

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

  a is not a constant; the number of subproblems should be fixed

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

  non-polynomial difference between f(n) and $n^{\log_b a}$ (see below)

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

  a<1 cannot have less than one sub problem

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

  f(n) which is the combination time is not positive

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

  case 3 but regularity violation.

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{\frac{n}{\log n}}{n^{log_2 2}} = \frac{n}{n \log n} = \frac{1}{\log n}$. It is clear that $\frac{1}{\log n} < n^{\epsilon}$ for any constant $\epsilon > 0$. Therefore, the difference is not polynomial and the Master Theorem does not apply.

These equation cannot be solve using Masters Theorem!

Class activity

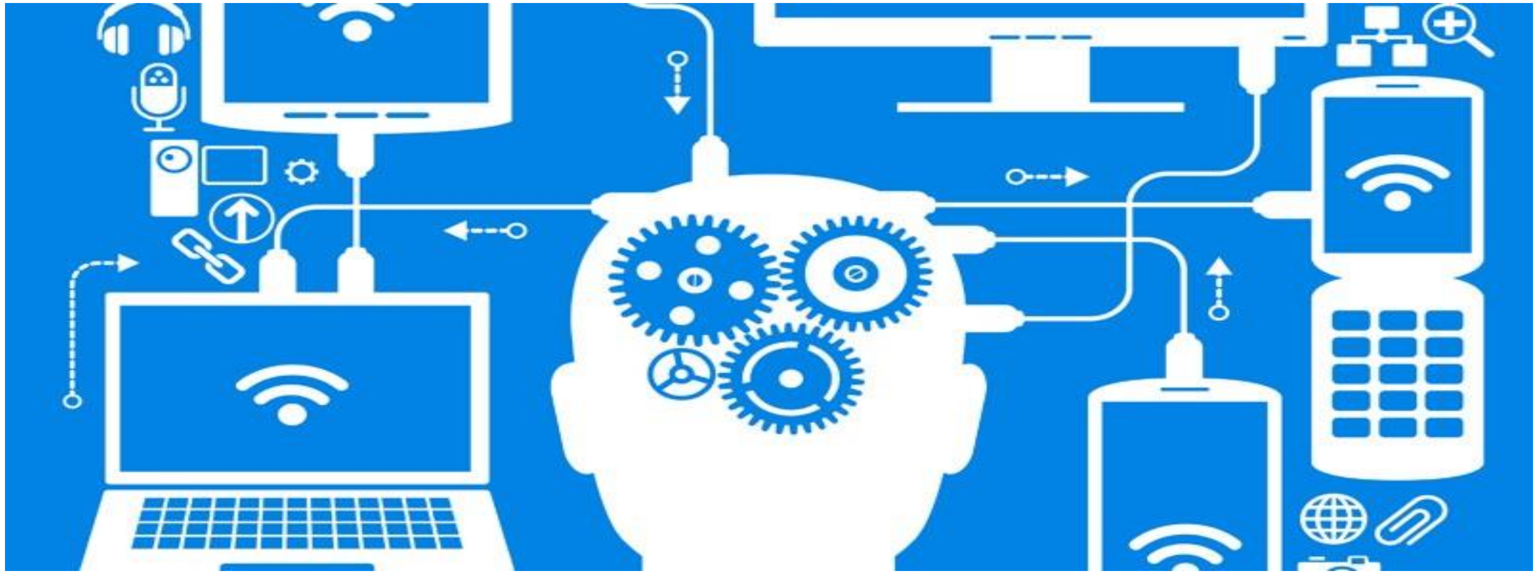
- From our lecture, we have seen that the running time of:

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

is $O(n \log n)$ using the recursion tree.

- Apply the back substitution method and master theorem to find the running time complexity of this relation.

In the next lecture..



Lecture 3: Sorting Algorithm

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. Introduction to Algorithms, 3rd edition. MIT Press.
- Robert Sedgewick and Kevin Wayne. 2011. Algorithm. 5th Edition. Addison-Wesley.
- Time complexity analysis of recursive program
  - https://www.youtube.com/watch?v=gCsfk2ei2R8&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=3
- Masters Theorem
  - https://www.youtube.com/watch?v=IPUhHmgrpik&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=5