SLIIT
Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 05

Introduction to Recursion

Recursion –Example 1

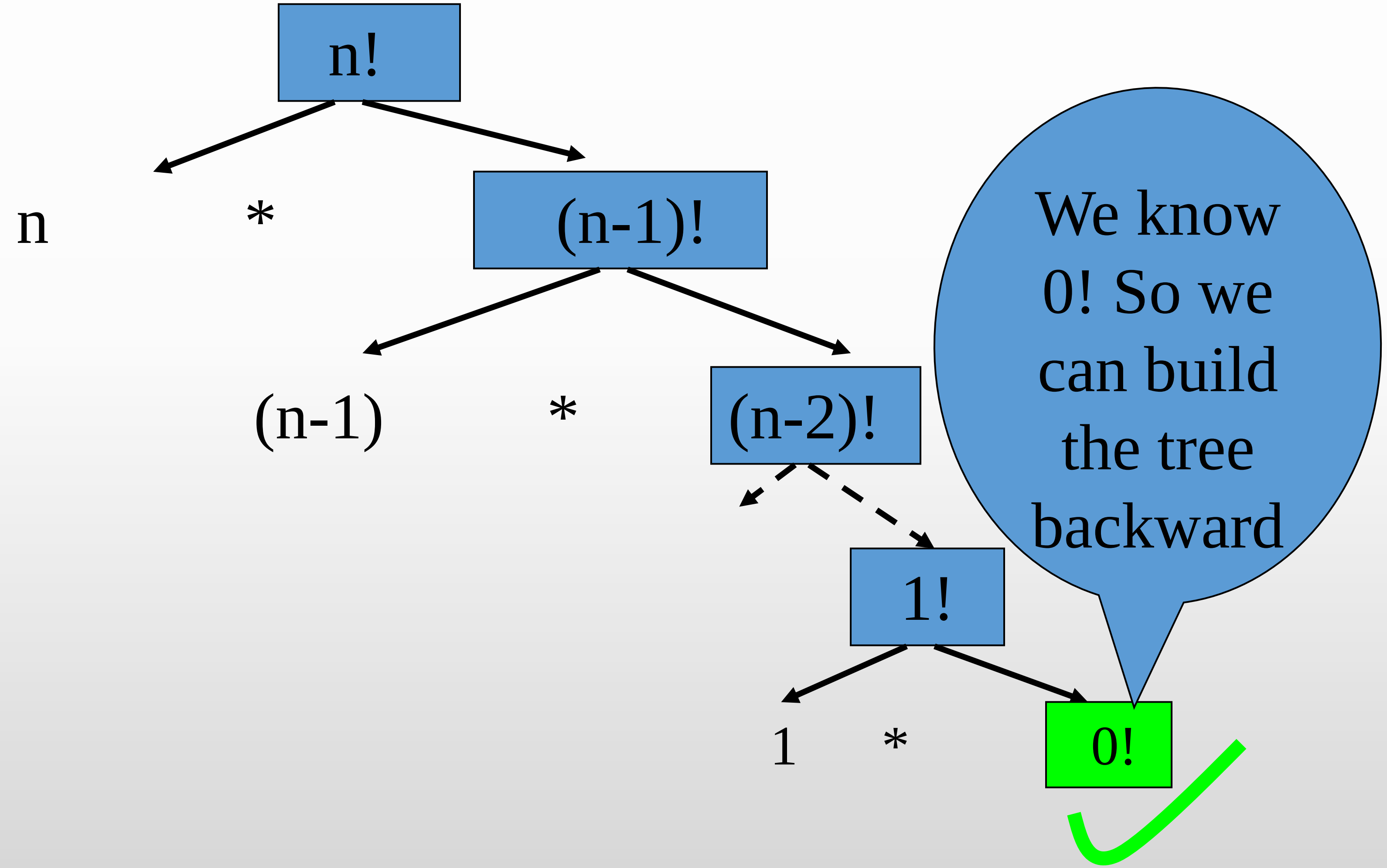
Factorial

$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$, and that $0! = 1$.

A recursive definition is

$$(n)! = \begin{cases} n * (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Factorial -A graphical view

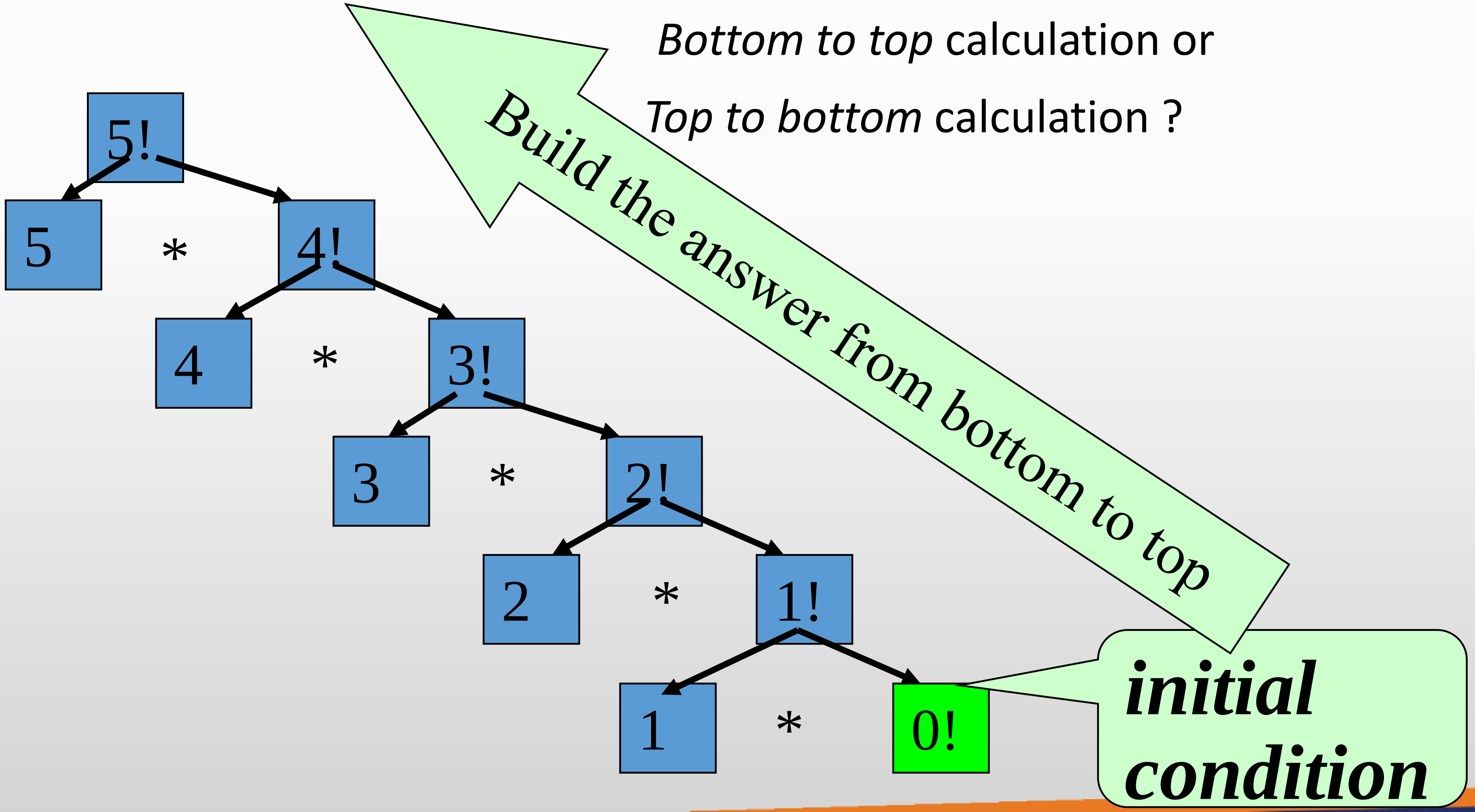


Exercise

- Draw the recursive tree for 5!
- How does it calculate 5! ?   Is it:

  Bottom to top calculation or

  Top to bottom calculation ?



Factorial(contd.)

$(n)! = \{n * (n-1)! \quad \text{if } n > 0\}$

$\{1 \quad \text{if } n = 0\}$

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

Compare

Recursion

What is recursion?

A function that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call is a ***recursive*** function.

Recurrence equation

- Mathematical function that defines the running time of recursive functions.

- This describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs.

Ex: $T(N) = T(N-1) + b$

$T(N) = T(N/2) + c$

Recurrence - Example1

Find the Running time of the following function

```
int factorial(int n) {
      if (n == 0)            //A
            return 1;          //B
      else
            return (n * factorial(n -1));      //C
}
```

Statement A takes time a → for the conditional evaluation

Statement B takes time b → for the return assignment

Statement C takes time:

- c → for the operations(multiplication & return)
- T(n-1) → to determine (n-1)!

Recurrence - Example1 (Contd.)

$$T(n) = \text{Time to execute } A \ \& \ C$$

$$T(n) = a + \overbrace{c + T(n-1)}$$

- This method is called iteration method (or repeated substitution)

Exercise

- Solve the recurrence

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

  You are given that

  $n = 16$ and

  $T(1) = 1$

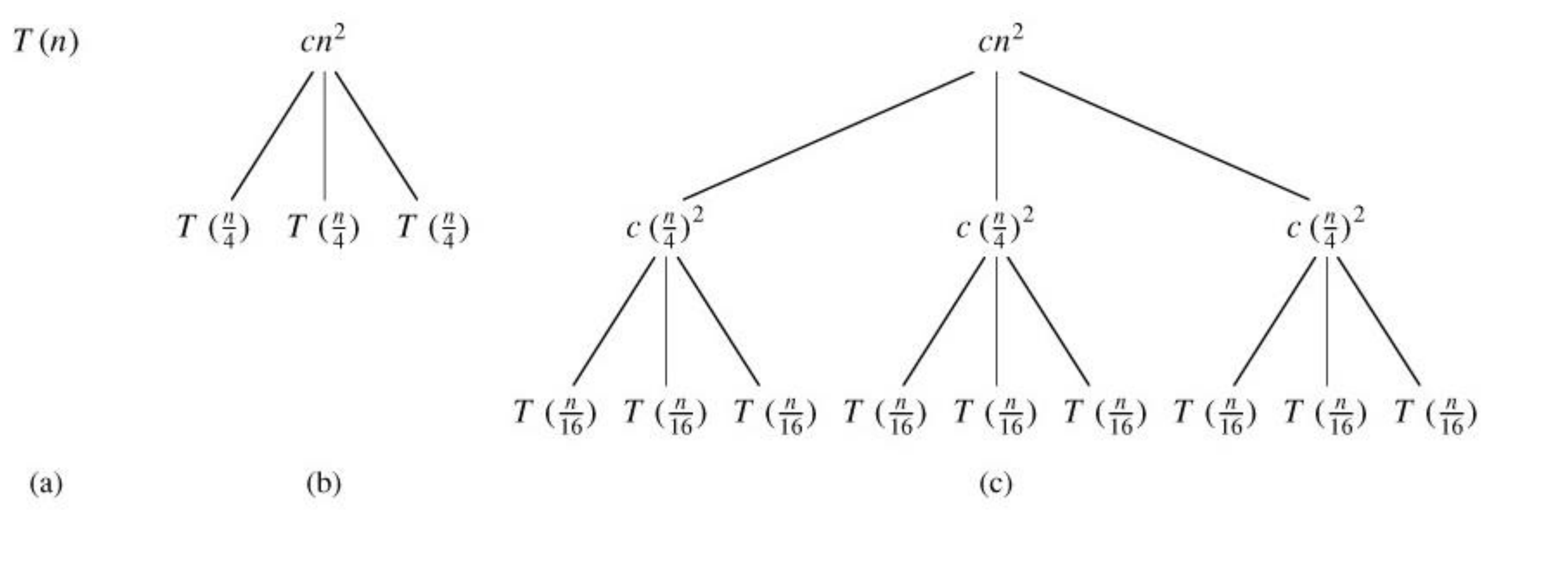
Finding a solution to a recurrence.

- Other methods
  - Recursion tree.
  - Master Theorem.

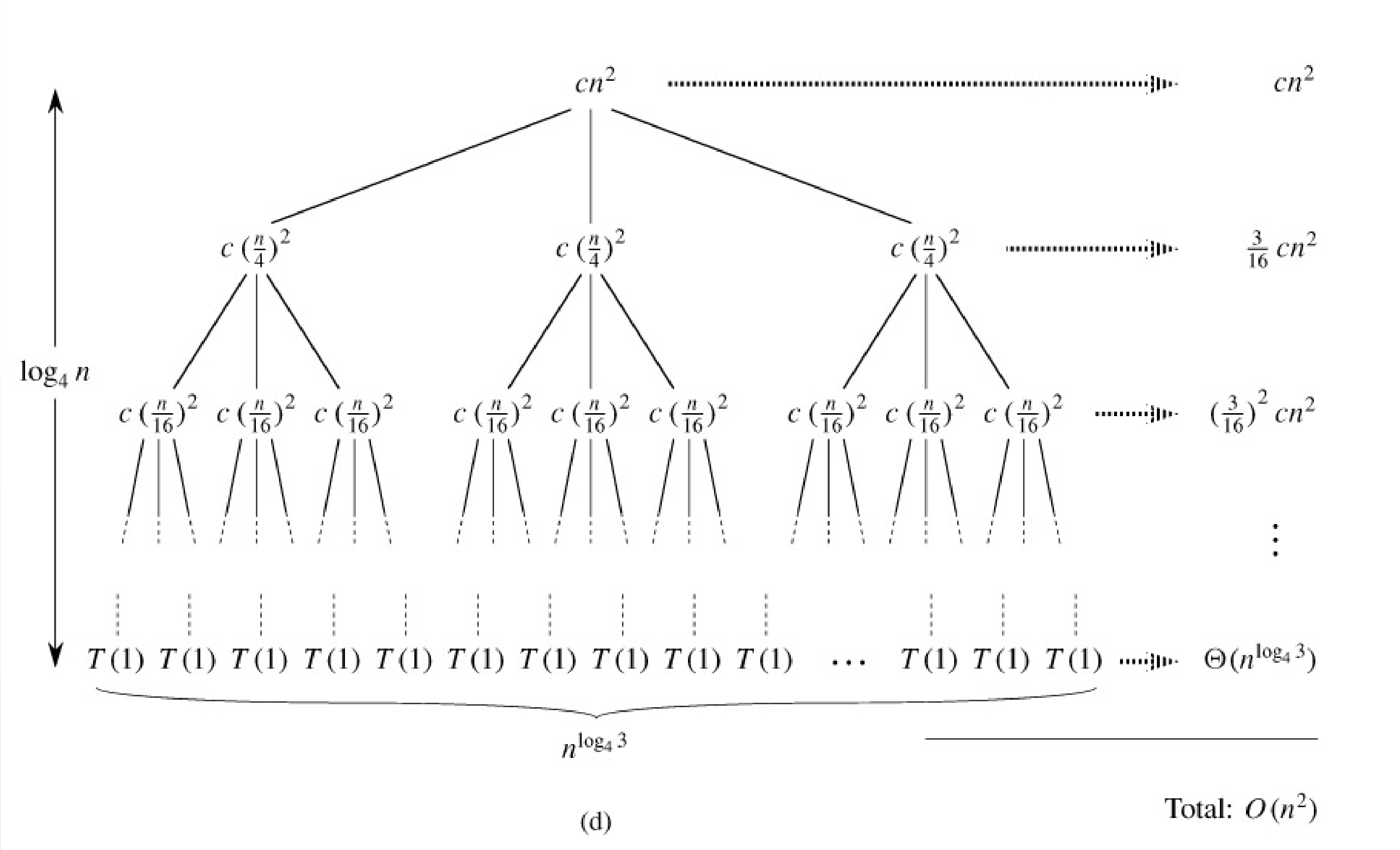
The recursion-tree method

- Although the substitution method can provide a succinct proof that a solution to a recurrence is correct, it is sometimes difficult to come up with a good guess. Drawing out a recursion tree,is a straightforward way to devise a good guess. In a **recursion tree**, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- A recursion tree is best used to generate a good guess, which is then verified by the substitution method.

Recursion tree for $T(n) = 3T(n/4) + cn^2$



Recursion tree for $T(n) = 3T(n/4) + cn^2$



The Master Method

- The Master method applies to recurrences of the form

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

where $a \geq 1$ and $b > 1$, and $f(n)$ is an asymptotically positive function.

The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time T (n/b). The cost of dividing the problem and combining the results of the subproblems is described by the function f (n).

The master theorem

- The master method depends on the following theorem.
- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

The master theorem

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

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

The master theorem

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \rightarrow f(n) < n^{\log_b a} \\ \Theta\left(n^{\log_b a} \lg n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \rightarrow f(n) = n^{\log_b a} \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \rightarrow f(n) > n^{\log_b a} \\ & \text{if } af(n/b) \le cf(n) \text{ for c} < 1 \text{ and large } n \end{cases}$$

Master Theorem – Case 1 example

Give tight asymptotic bound for

$$T(n) = 9T(n/3) + n$$

Solution:

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

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

$$f(n) = O(n^{\log_3 9 - \varepsilon}) \text{ for } \varepsilon = 1 \text{ or } f(n) < n^{\log_3 9} \rightarrow \text{case1}$$

$$\therefore T(n) = \Theta(n^2)$$

Master Theorem – Case 2 example

Give tight asymptotic bound for

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

Solution:

a=1, b=3/2, and $f(n)$ = 1.

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$$f(n) = \Theta(n^{\log_b a}) \text{ or } f(n) = n^{\log_b a} \rightarrow \text{case } 2$$

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

SLIIT
Discover Your Future

Master Theorem – Case 3 example

- Give tight asymptotic bound for
- $T(n) = 3T(n/4) + n \log n$
- **Solution:**
- a=3, b=4, and $f(n) = n \log n$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3+\varepsilon}), \text{for } \varepsilon \approx 0.2 \text{ or } f(n) > n^{\log_4 3} \rightarrow \text{case } 3$$

$$Note: n \lg n \geq c.n^{\log_4 3}.n^{0.2}$$

Exercises.

- Use the master method to give tight asymptotic bounds for the following recurrences.
  1. $T(n) = 4T(n/2) + n$.
  2. $T(n) = 4T(n/2) + n^2$.
  3. $T(n) = 4T(n/2) + n^3$.
- Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$.