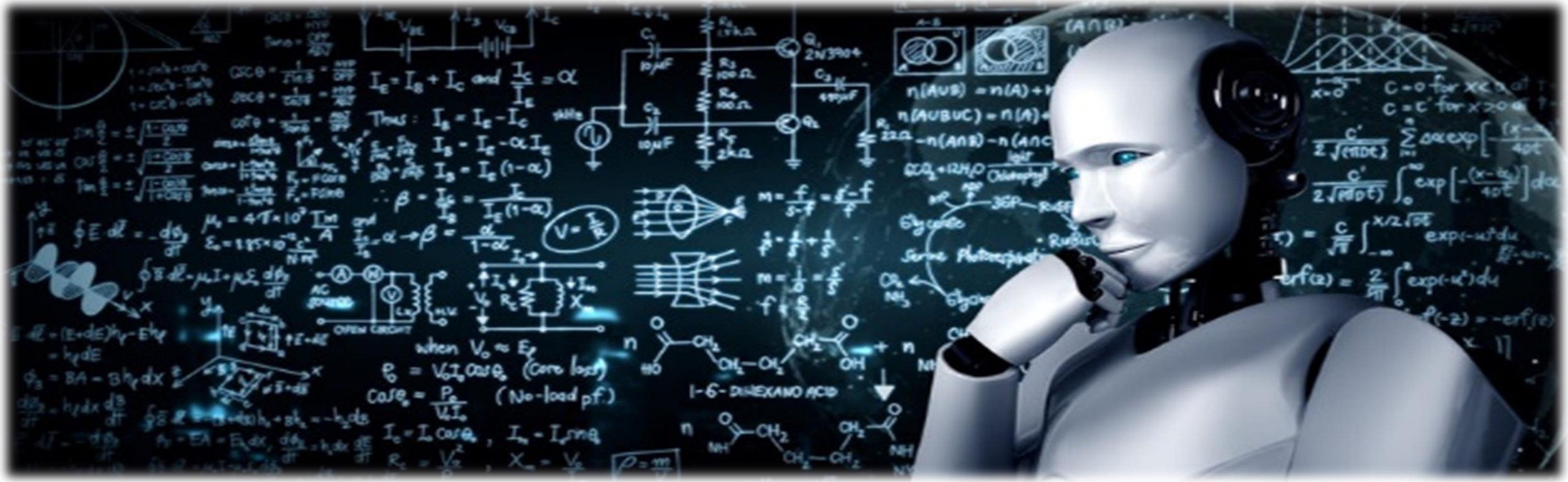


Lecture 03 - Introduction to Algorithms Recursion

Design and Analysis of Algorithms – IT1205

Year 02 Semester 02

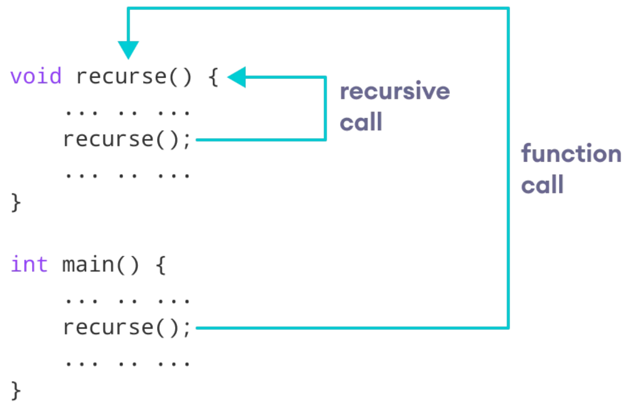
SCU
SLIIT CITY UNI

Contents for today

- Recursion & Recurrences
  - Recurrence equation
  - Factorial Series
- Finding a solution to a recurrence
  - Repeated Substitution method.
  - Recursion tree.
  - Master Theorem.

What is Recursion?

- Recursion is a function calls itself again and again until it reach to the base condition.



- Properties of Recursion:
  - Performing the same operations multiple times with different inputs.
  - In every step, try with smaller inputs to make the problem smaller.
  - Base condition is needed to stop the recursion otherwise infinite loop will occur.

Factorial

- We read n! as "n **factorial**"
- $n! = n*(n-1)*(n-2)*...*2*1$, and that $0! = 1$.
- A recursive definition is

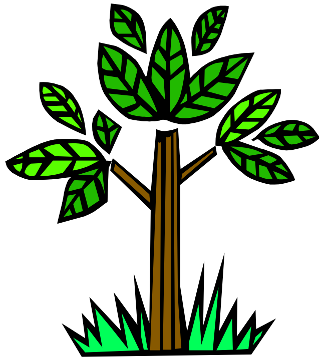
Recursive Call

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

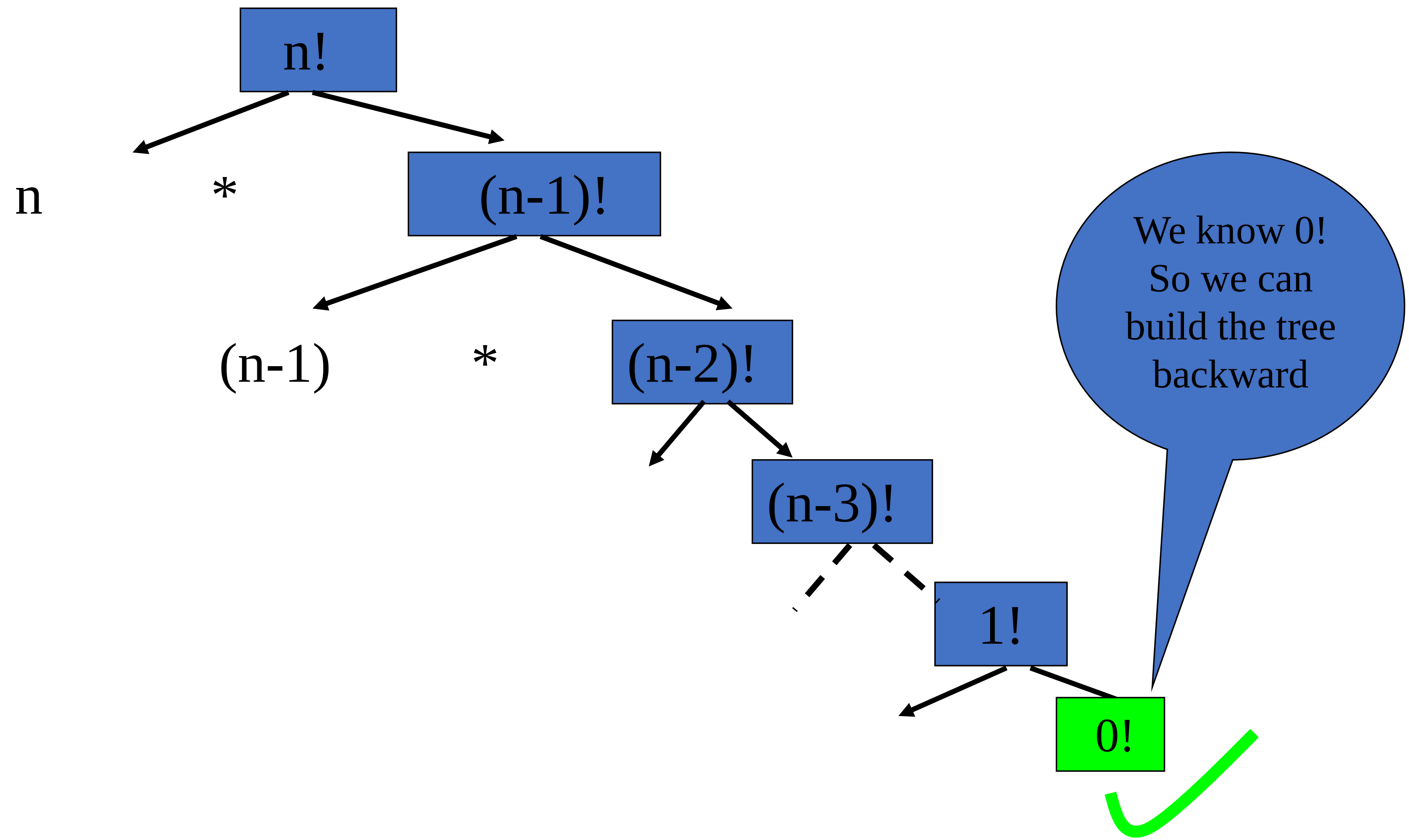
Initial Condition

Recursion tree

- We can draw a recursion tree for any recursive function.
- Drawing a recursion tree will help us to graphically visualize the recursive relation.
- Let's draw the recursion tree for the factorial function



Factorial -A graphical view



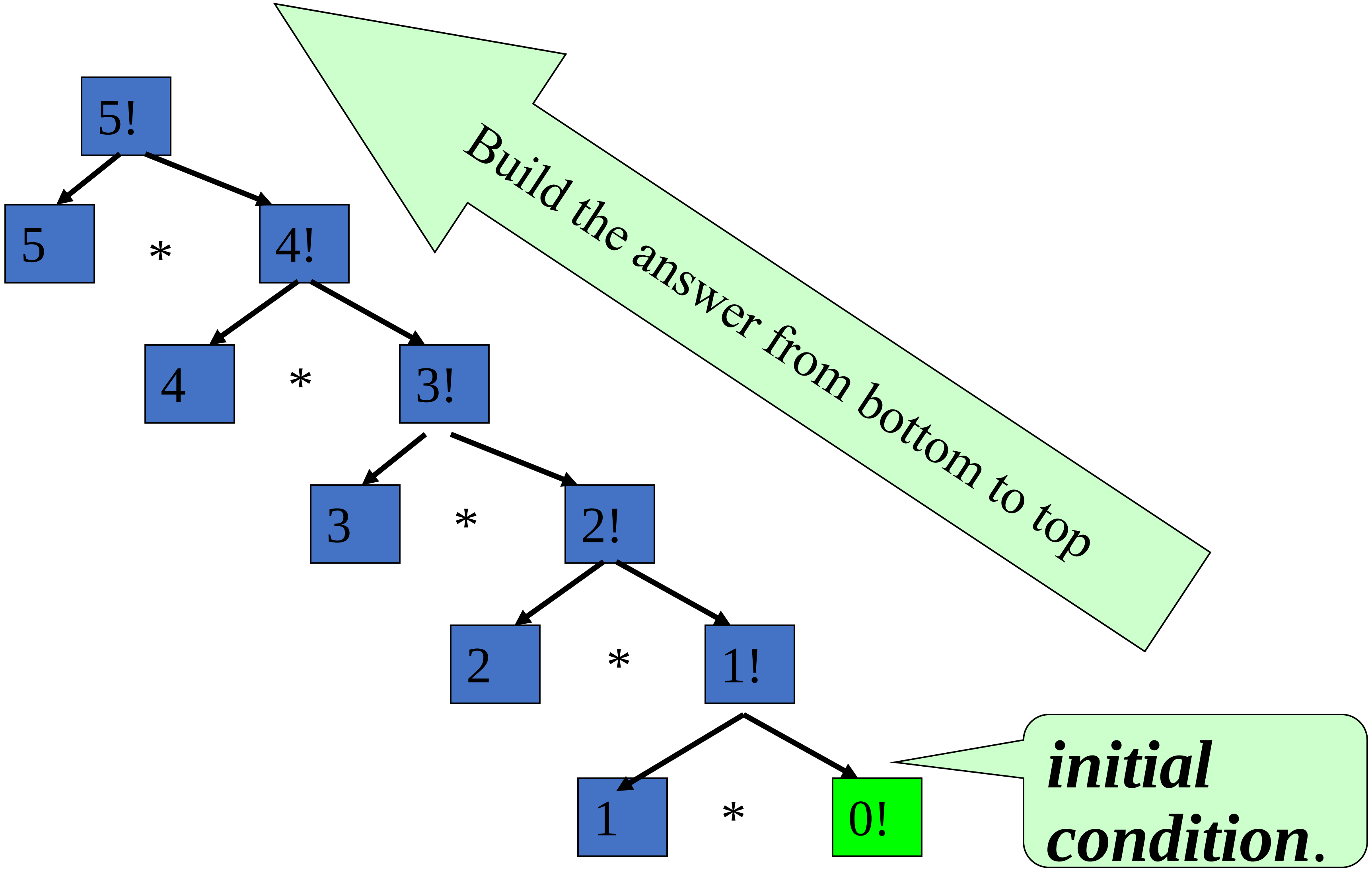
Exercise

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

  Bottom to top calculation or

  Top to bottom calculation

Solution



5!
5
*
4!
4
*
3!
3
*
2!
2
*
1!
1
*
0!
Build the answer from bottom to top
initial condition.

Factorial(contd.)

- Now, we want to build a procedure

- Let's try to devise an algorithm straight from the mathematical definition.

Factorial(contd.)

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

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

Compare

Recursive Function

- What is recursive Function?

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.

Understanding recursive algorithms can be done using ***recursive relations***

Definition of Recursive Relation

- A ***recursive relation*** for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms of the sequence, for all integers $n >= n_0$, where n_0 is a non-negative integer.

- The condition $n = n_0$ is called the ***initial condition***.

- NOTE: some cases may contain more than one initial condition.

  e.g. Fibonacci numbers

Definition of Recursive Relation

- Base case(s).
  - Values of the input variables for which we perform **no recursive calls** are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls.
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Recursion –Example 1

- The number of bacteria in a colony doubles every hour. If the colony begins with 05 bacteria, how many will be present in 2 hours?

- Recursive relation:

  Let a_n be the number of bacteria after n hours.

  $\mathbf{a_n = 2.a_{n-1}}$

  Initial condition $a_0 = 5$.

- Solution: Solve for a_2 given this relation.

  $a_2 = 2*a_1 = 2*2*a_0 = 2*2*5 = \mathbf{20}$

Recursion –Exercise

- Suppose Sunil deposits Rs.10,000 in a savings account at a bank, yielding 11% interest per year with interest compounded annually. How much will be in the account after 30 years?

Solution

- <u>Solution</u>:

Let P_n denote the amount in the account after n years. Then the sequence $\{P_n\}$ satisfies the recursive relation:

$P_n = P_{n-1} + 0.11\ P_{n-1} = (1.11)P_{n-1}$

The initial condition is $P_0 = 10{,}000$.

Note that:

$P_1 = (1.11)P_0$

$P_2 = (1.11)P_1$ *(That means $1.11*1.11P_0$)* $= (1.11)^2\ P_0$

$P_3 = (1.11)P_2 = (1.11)^3\ P_0$

We see a pattern! In general,

$P_n = (1.11)P_{n-1} = (1.11)^n\ P_0$. For $n = 30$, $P_{30} = (1.11)^{30}10{,}000$.

Recurrence equation

- *Mathematical function* that *define 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.
- when an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Definition of Recurrence Relation

- A recurrence relation for $T(N)$ is simply a recursive definition of $T(N)$.
  - This means $T(N)$ is written as a function of $T(k)$ where $k < N$.
- Two common types are:
  - $\boldsymbol{T(N) = T(N-1) + b}$
  - $\boldsymbol{T(N) = T(N/2) + c}$

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 "*c*" for the conditional evaluation
- Statement B takes time ***"d"*** for the return assignment
- Statement C takes time:

"e" - for the operations(multipl. & return)

+

T(n-1) – to determine (n-1)!

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

Finding a solution to a recurrence.

- Other methods
  - Repeated Substitution method.
  - Recursion tree.
  - Master Theorem.

Repeated substitution method

- The technique of Repeated Substitution can be used to solve “simple” recurrence relations.

- The idea is very straight-forward. We start with the recurrence relation given to us. We use the recurrence relation to expand the right hand side of the equation. We do so a few times with the goal of ***finding a pattern*** on the right hand side as the <u>argument becomes smaller</u>.
- Once we find a pattern, we write a general expression on the right hand side. When we have a general expression, we can go ahead and solve the recurrence and obtain a closed form solution.
- We illustrate the technique by solving a number of recurrences. In the course of performing this method, it is common practice to make assumptions regarding the values the argument n can take, in order to be able to solve a recurrence relation.

Method

1. Determine T(n) for the general case
2. Determine T(0) or T(1) i.e. base case
3. Expand T(n) determined in step 1 in T(n-1), T(n-2), etc.
4. Solve it to determine: T(n) = Polynomial
5. Apply Big- **O** to determine the order of **O**(T(n)).

Repeated substitution method(Example 01)

Example 1:

$\mathbf{T(n) = T(n-1) + c}$, $n > 1$ & c is a small positive constant. :$T(1) = d$

$$\begin{aligned}
T(n) &= T(n-1) + c \\
&= (T(n-2) + c) \quad + c \\
&= (T(n-2) \quad + 2c \\
&= (T(n-3) + c) \quad + 2c \\
&= (T(n-3) \quad + 3c
\end{aligned}$$

....after k times

$$= (T(n-k) \quad + k\,c$$

If $k=n-1$

$$\begin{aligned}
&= T(n-(n-1)) + (n-1)c \\
&= T(1) + (n-1)c \\
&= d + (n-1)c
\end{aligned}$$

$T(n)= nc + (d - c)$

Running Time : $O(n)$

Repeated substitution method (Example 02)

Example 2:

$T(n) = T(n/2) + c$, $n > 1$ & c is a small positive constant. :$T(1) = d$

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

$= (T(n/4) + c) + c$

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

$= (T(n/8) + c+c) + c$

$= (T(n/2^3)) + 3c$

....after k times

$= (T(n/2^k)) + k.c$

If $k=\log_2 n$

$= (T(n/2^{\log 2\ n})) + (\log_2 n).c$

$= T(n/n) + c \cdot \log_2 n$

$= T(1) + c \cdot \log_2 n$

$= d + c \cdot \log_2 n$

$T(n)= c \cdot \log_2 n + d$

Running Time : $O(\log_2 n)$


SEE YOU
next
week!