# Data structure and Algorithms

# Course

Created by : Israa Abdelghany
LinkedIn : www.linkedin.com/in/israa-abdelghany-4872b0222
GitHub : https://github.com/IsraaAbdelghany9

# Algorithm Analysis:-

→ the process of finding the computational complexity of algorithms (which are :- time, storage and other resources needed to excute them)

→ used to test efficiency of an algorithm also used to compare between 2 or more algorithms

there are many factors :.

the most important 2 factors are:-

1. time "time Complexity"
2. storage "space Complexity"

⇒ best way to detect algorithm efficiency is to implement the algorithm and find the excution time. But it's not significant because this method depends on the machine itself.

"Bad code running on good computer is better than Good code on bad Computer"

⇒ even running code twice will give different excution time

⇒ there're many factors related to the machine that will make it insignificant

- Single processor / Multi processor
- Core speed
- Computer Architecture 32/64 bit
- Ram speed (Read / write)
- Input (Not related to machine)

- So we must separate the algorithm from the machine

⇒ if the time = f(input) : it will give approximate t to predict

⇒ Assumptions of time Complexity

- → Each read & write operations takes a unit time
- → Each Assignment " " " " "
- → " Arithmatic " " " " "
- → " Comparision " " " " "
- ⇒ Algorithm can carry 1 operation in single step
- → the only factor that is used to check complexity is the input as it's not related to the machine
- → our goal is to find a measure that will take into consideration all inputs
- → This measure that we need to find a function to calculate the time with a relation of the time (i.e. maps each input to time)

$t = f(\text{inputs})$

→ Since it's very hard to describe exactly the function, we will work with the approximation

Side Note:

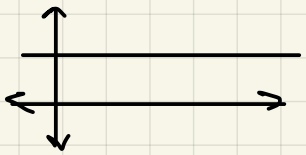
1. $n^2$ → 2. $6n^2$ / 3. $12n^2$ /

# Complexity Asymptotic Notations :-

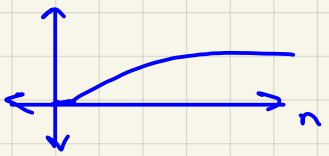
1. Big-O Notation :- O()
   - defines the upper bound of an algorithm "worst case"
2. Omega Notation :- Ω
   - it defines the lower bound of an algorithm " best case "
3. theta Notations :- θ
   - it defines the inner "middle" bound of an algorithm " between O and Ω "

## Equation properties :-

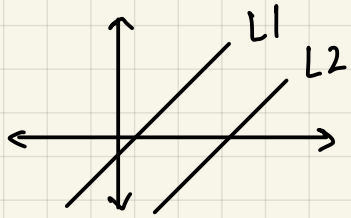
1. Constant Equation (y=a)
   this function is of order O(1)



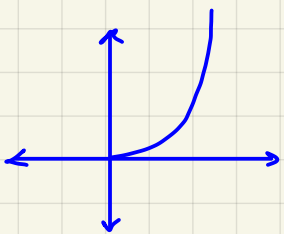
2. Logarithmic Equation ($y = a \log x + c$)
   this function is of order
   O(log n)

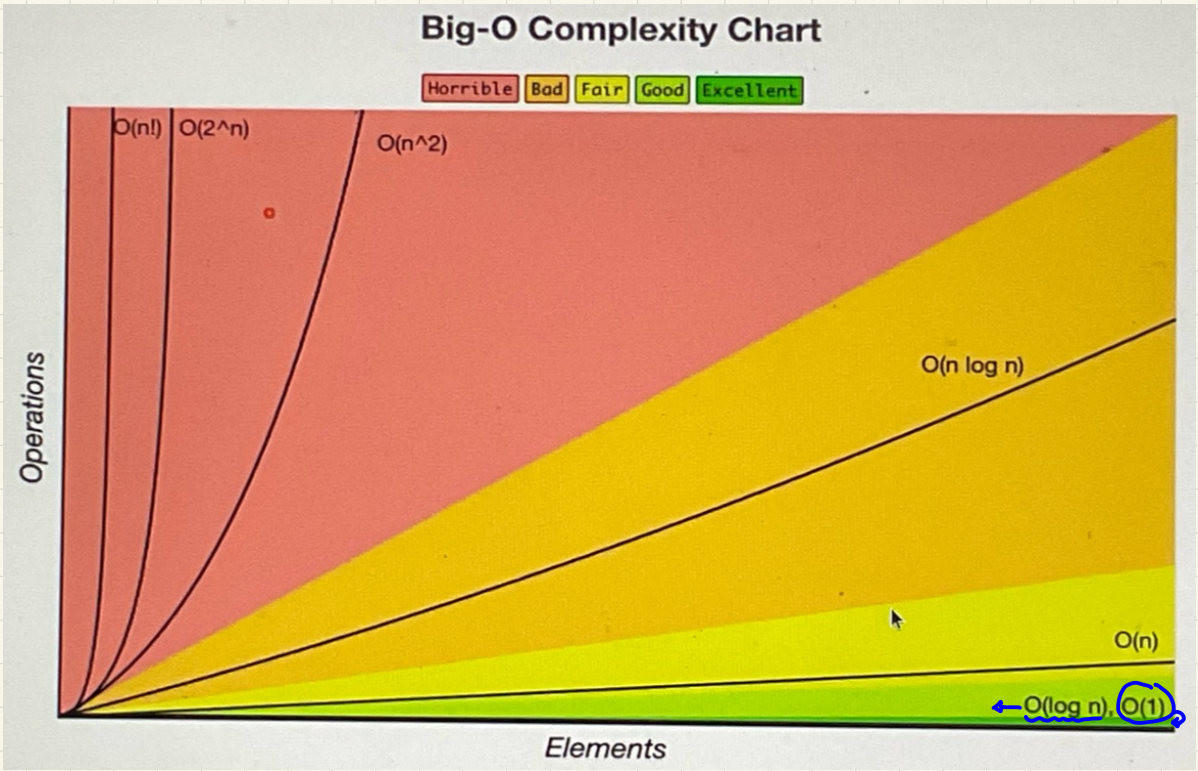



3. Linear equation ($y = ax + b$)
   for all values of x, the y will increase with the same value
   this function is of order O(n)




4. Quadratic equation ($y = ax^2 + bx + c$)
   for all x values, y will increase by $x^2$
   this function is of order $O(n^2)$






| | constant | logarithmic | linear | N-log-N | quadratic | cubic | exponential |
|---|---|---|---|---|---|---|---|
| $n$ | $O(1)$ | $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n^3)$ | $O(2^n)$ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 2 | 1 | 1 | 2 | 2 | 4 | 8 | 4 |
| 4 | 1 | 2 | 4 | 8 | 16 | 64 | 16 |
| 8 | 1 | 3 | 8 | 24 | 64 | 512 | 256 |
| 16 | 1 | 4 | 16 | 64 | 256 | 4,096 | 65536 |
| 32 | 1 | 5 | 32 | 160 | 1,024 | 32,768 | 4,294,967,296 |
| 64 | 1 | 6 | 64 | 384 | 4,069 | 262,144 | $1.84 \times 10^{19}$ |

I'm more concerned by the trend of change

approximately by the n=16 the time increased alot

. in for loops ⇒ if $i*2$ ⇒ $O(\log n)$

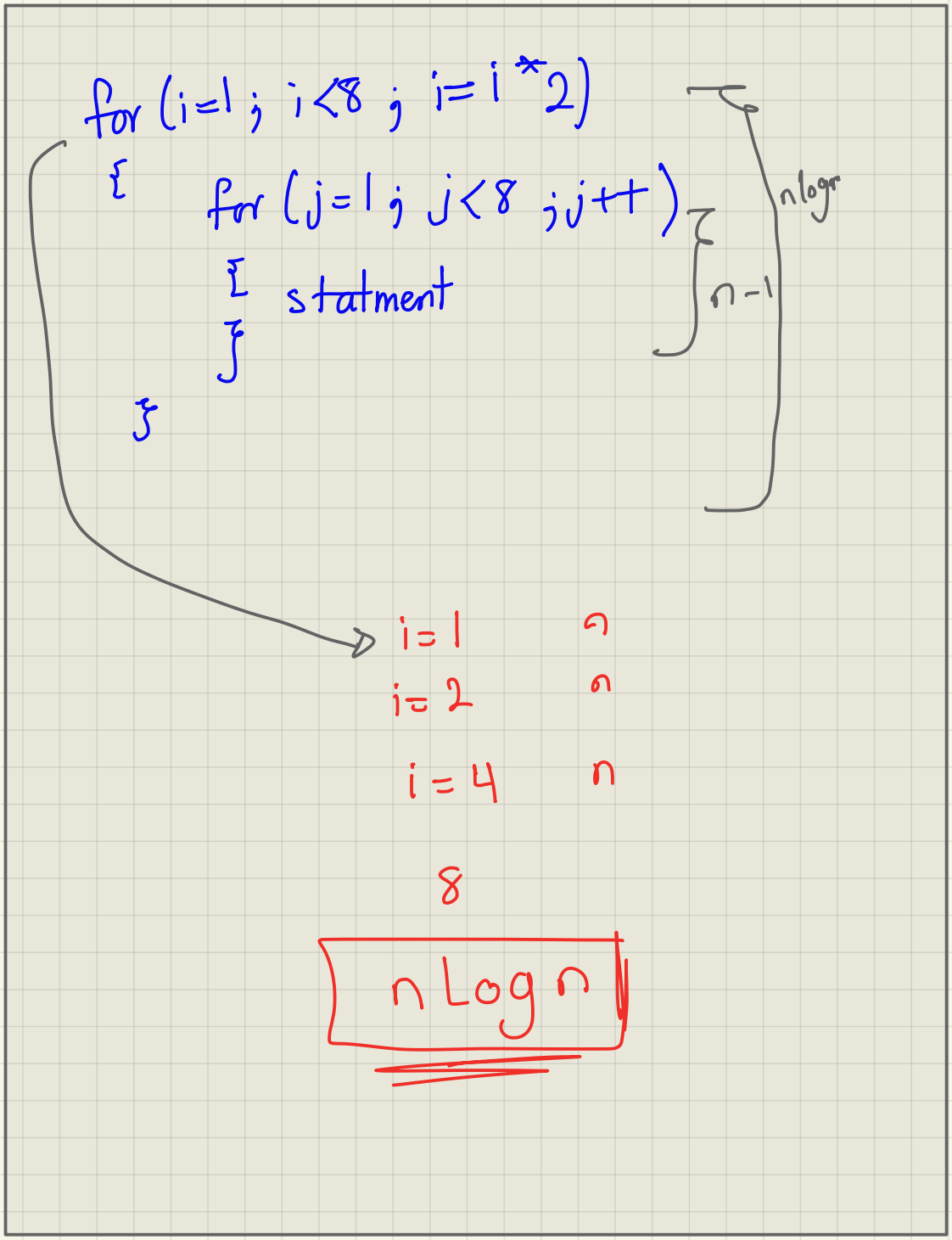
- if $i++$ and inner loop $j++$ ⇒ $O(n^2)$
  $(n-1)^2 * T1^2$
- " $i++$ and inner loop $j*2$ ⇒ $O(n \log n)$
  $(n-1) \log n$
- " $i*2$ and inner loop $j++$ ⇒ $O($

```
for (i=1 ; i<8 ; i = i*2)
{
  for (j = 1, j<i ; j++)
  { statement
  }
}
```

| | |
|---|---|
| i = 1 | 0 |
| i = 2 | 1 |
| i = 4 | 3 |
| i = 8 | - |

$O(n)$

not all the time I multiply outer & inner loops
Sometimes I add like this example



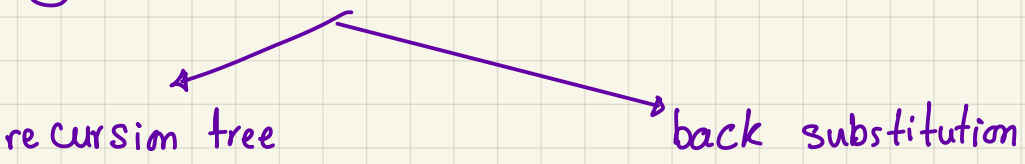

other examples in the Lec

# Recurrence Relation

- is the process of finding the terms of a sequence in a recursive manner

- finding the complexity of recursive function, lead to find the recursive relation of that function.

- after finding the recurrance relation we can use




to find the big O() notation.

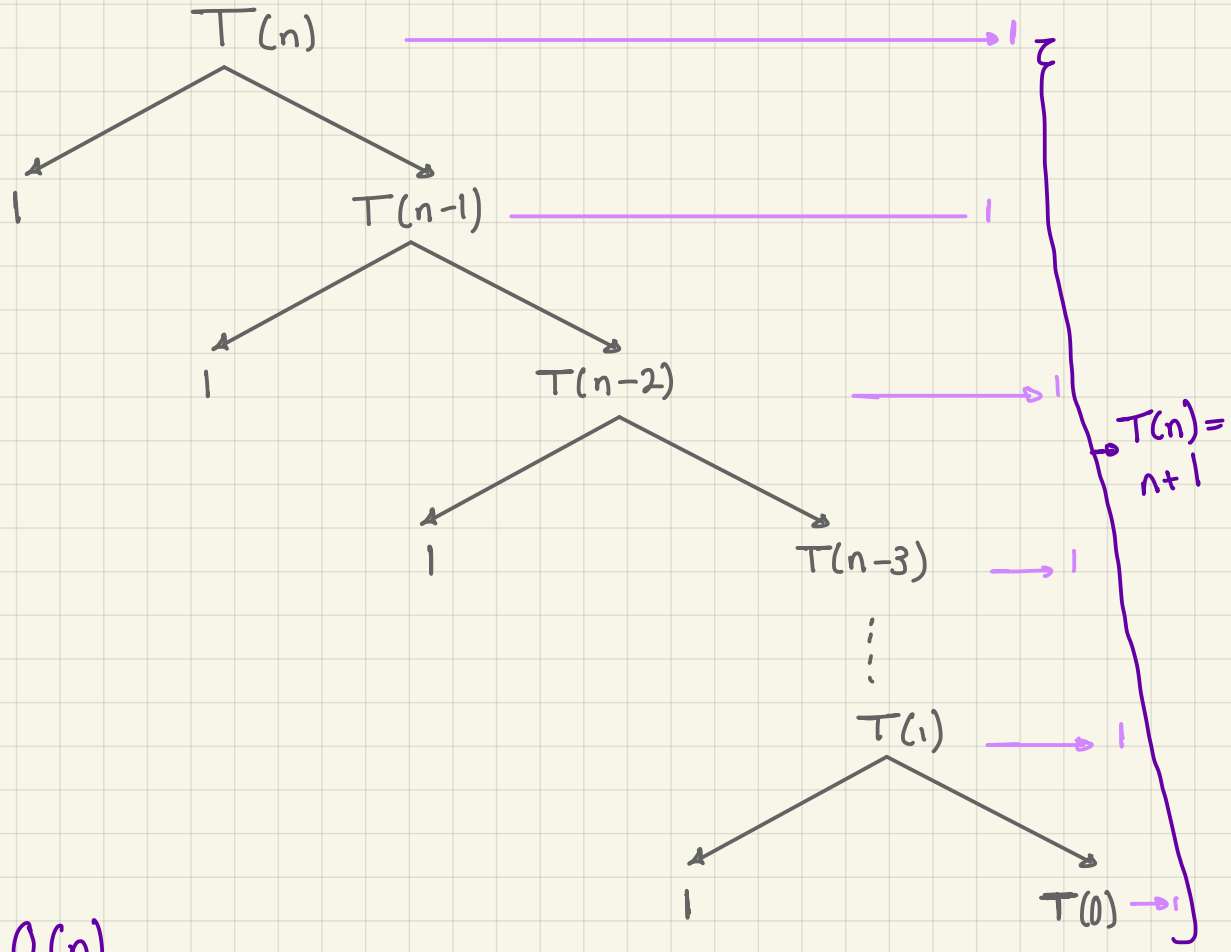
Example:-

```
void f1(int n)          ——————→ T(n)
{
    if(n>0)  ⇒ the stopping condition
    {
        print(n)    ——————→ 1
        f1(n-1)     ——————→ T(n-1)
    }
}
```

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

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

solution using Recursion Tree :-




∴ O(n)

solution using Back substitution :-

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

$$T(n-1) = T(n-2) + 1 \quad \text{II}$$

$$T(n-2) = T(n-3) + 1 \quad \text{III}$$

Substitute II in I :-

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

$$= T(n-2) + 2 \quad \text{IV}$$

Substitute III in IV

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

$$= T(n-3) + 3 \quad \rightarrow \text{will continue tok}$$

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

This will be repeated till $T(0)$

So

$$n - k = 0$$

$$n = k$$

Substitute in the general form:-

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

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

$$T(n) = T(0) + n$$

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

$\therefore O(n)$

Example 2 :-

```
void f2 (int n)                                  ----------> T(n)
{
    if (n >0)
    {
:       for (I =0 ; I<n; I++)                    ----------> n+1
        {
            print (n)                            ----------> n
        }
        f(n-1)                                   ----------> T(n-1)
    }
}
```

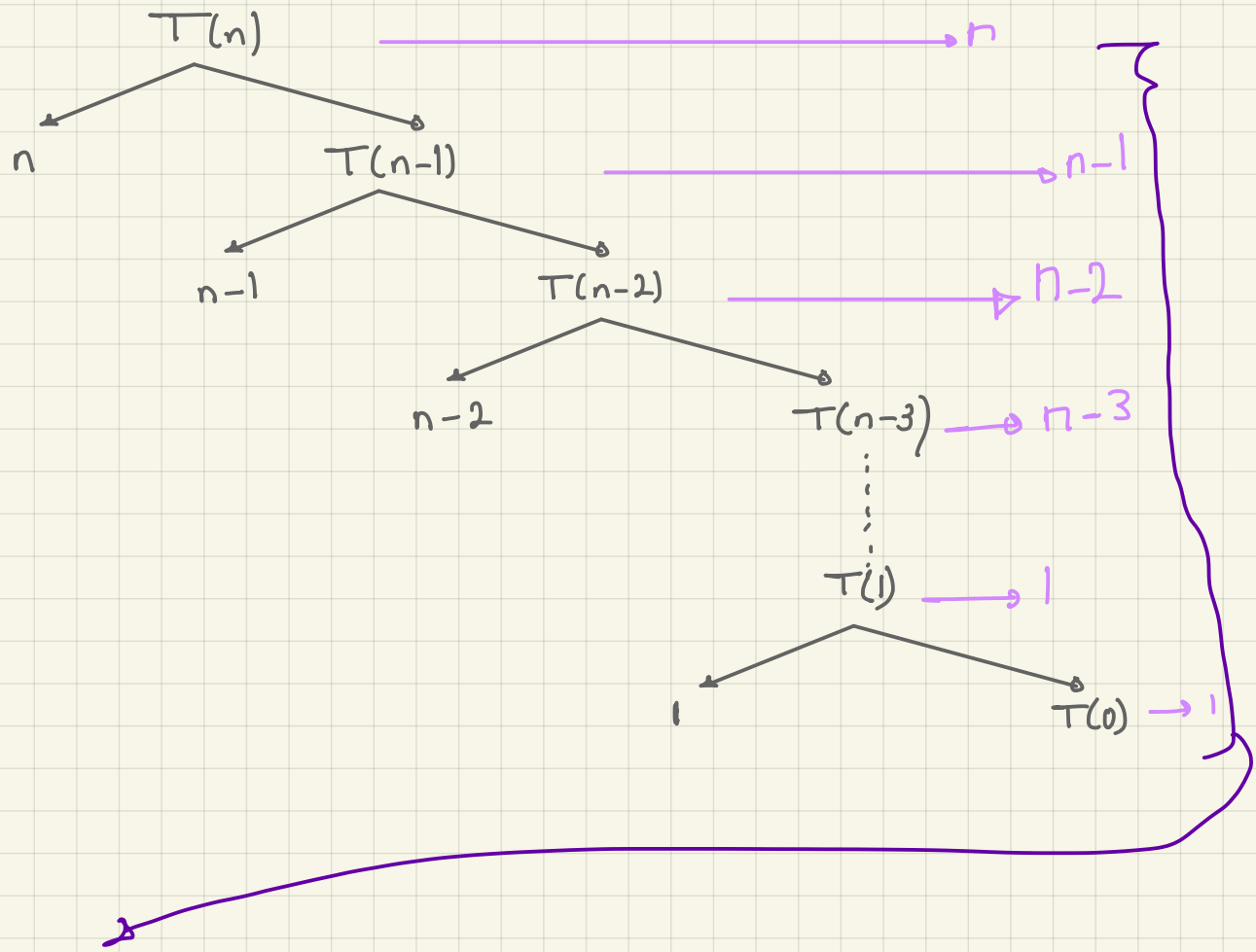
$T(n) = n+1 + n + T(n-1)$

$T(n) = T(n-1) + 2n + 1$ ~> Can be written as n

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

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

Solution using Recursion tree:-




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

$$= 1 + \frac{n(n+1)}{2}$$

$$\therefore O(n^2)$$

Solution using back substitute:-

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

$$T(n-1) = T(n-2) + (n-1) \quad \text{II}$$

$$T(n-2) = T(n-3) + (n-2) \quad \text{III}$$

Substitute II in I

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

$$= T(n-2) + (n-1) + n \qquad \text{IV}$$

Substitute III in IV

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

will continue to k

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

this will be repeated till $T(0)$

$$n-k=0$$

$$n=k$$

Substitute in the general form:

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

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

$$T(n) = 1 + \frac{n(n+1)}{2}$$

$$\therefore O(n^2)$$

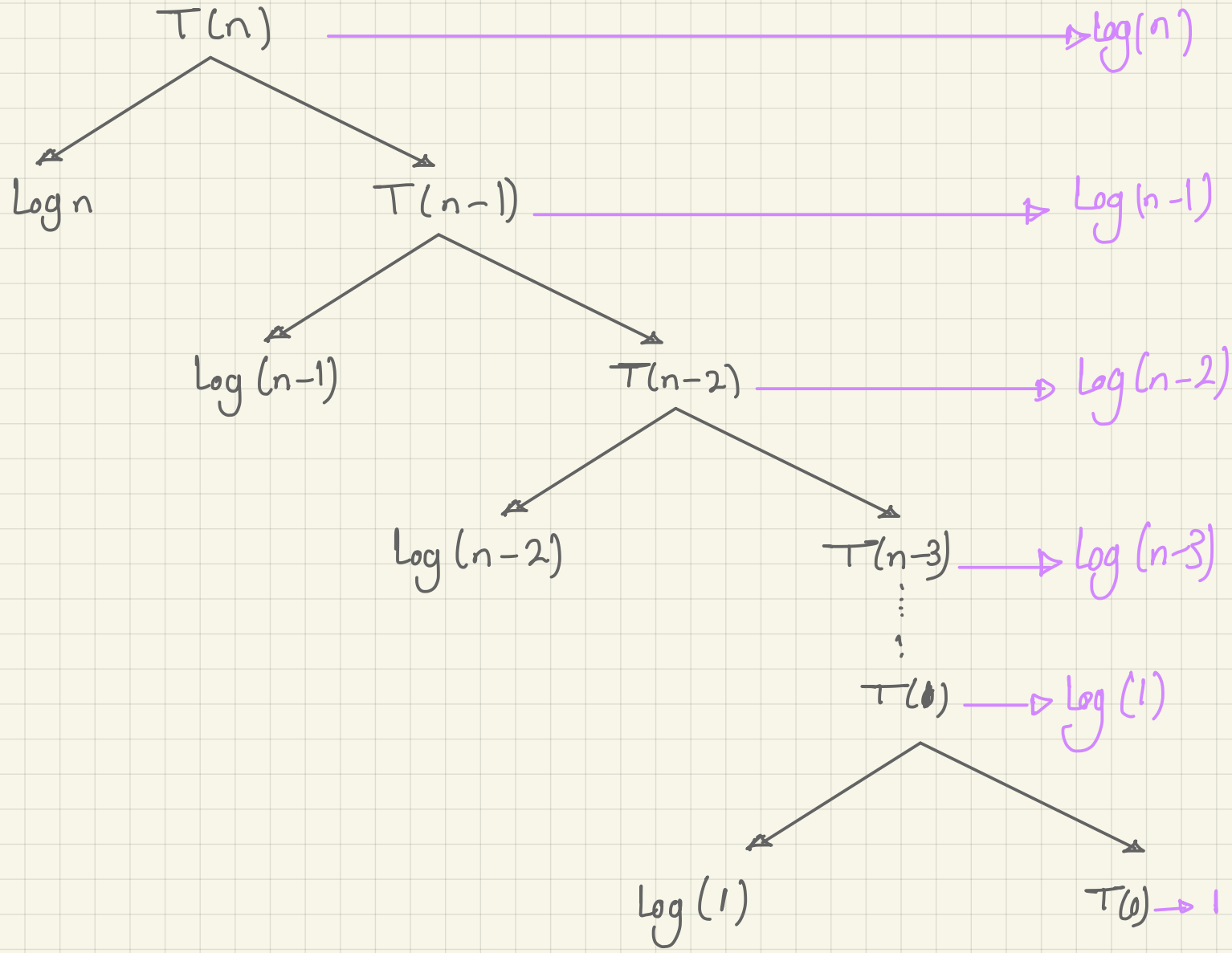
Example 3:-

```
void f3(int n)                                  ----> T(n)
{
  if (n>0)
  {
    for (I=1; I<n; I=I*2)
    {
      print (n)                                 ----> log n
    }
    f3(n-1)                                     ----> T(n-1)
  }
}
```

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

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

solution using Recursion tree:-




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

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

$$= \log(n!) + 1$$

$$O(n \log n)$$

Solution using back substitute :-

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

$$T(n-1) = T(n-2) + \log(n-1) \quad \text{II}$$

$$T(n-2) = T(n-3) + \log(n-2) \quad \text{III}$$

Substitute II in I

$$T(n) = T(n-2) + \log(n-1) + \log(n) \quad \text{IV}$$

Substitute III in IV:-

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

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

will continue till $k = 0$

$$n - k = 0$$

$$n = k$$

$$\therefore T(n) = T(0) + \log(n-n+1) + \log(n-n+2) \ldots + \log(n-1) + \log(n)$$

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

$$\therefore T(n) = 1 + \log(n!)$$

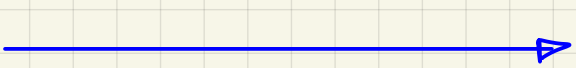
$\Rightarrow$ Note $\log(n!) = \frac{1}{2}\log(2\pi n) + n\log n - n$

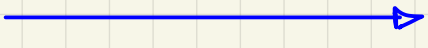
$\therefore$ same as $O(n \log n)$

$\therefore O(n \log n)$ #

Example 4 :-

```
void f4 (int n)
{
    if (n > 0)
    {
        print (n)
        f4(n-1)
        f4(n-1)
    }
}
```



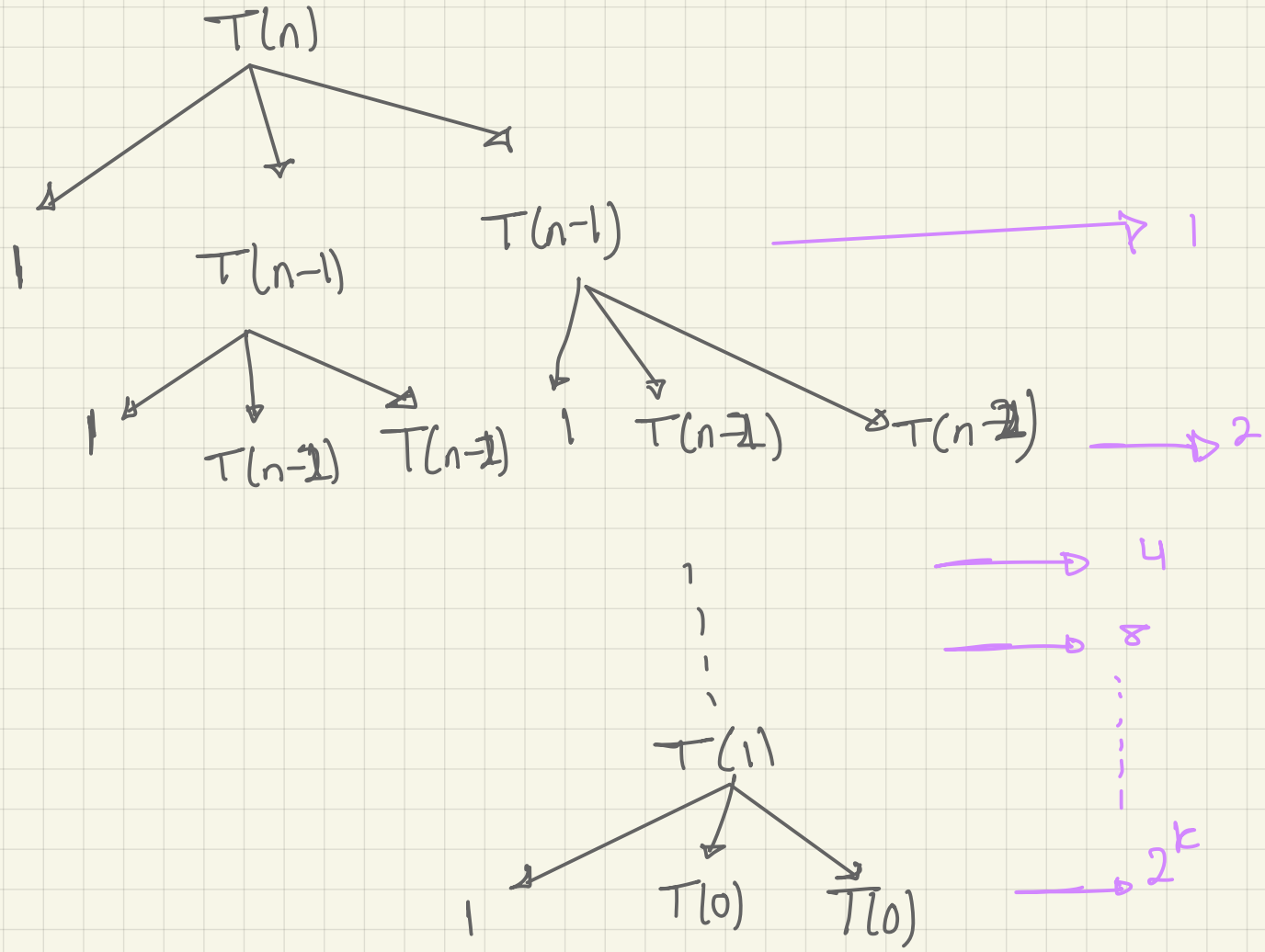



$$T(n) = 1 + T(n-1) + T(n-1)$$

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

$$T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1) + 1 & n > 0 \end{cases}$$

Solution using Recursion Tree :-




$T(n) = 1 + 2 + 4 + 8 + 16 \ldots\ldots + 2^k$

$\therefore T(n) = 2^{k+1} - 1$

Note

$k = n$

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

$O(2^n)$

Solution using backsubstitute :-

$$T(n) = 2T(n-1) + 1 \quad \text{I}$$
$$T(n-1) = 2T(n-2) + 1 \quad \text{II}$$
$$T(n-2) = 2T(n-3) + 1 \quad \text{III}$$

Substitute II in I

$$T(n) = 2\left[2T(n-2)+1\right] + 1$$

$$= 4T(n-2) + 2 + 1 \quad \text{IV}$$

Substitute III in IV :-

$$T(n) = 4\left[2T(n-3)+1\right] + 2 + 1$$

$$T(n) = 8T(n-3) + 4 + 2 + 1$$

$$\Rightarrow T(n) = 2^k T(n-k) + 2^2 + 2^1 + 2^0$$

This will continue till $T(0)$

$$n - k = 0$$
$$n = k$$

$$T(n) = 2^n T(0) + 2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2^1 + 2^0$$

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

$$\therefore O(2^n)$$

# Master theorem for decreasing functions

$T(n) = T(n-1) + 1$ ⟶ $O(n) = O(n*1)$

$T(n) = T(n-1) + n$ ⟶ $O(n^2) = O(n*n)$

$T(n) = T(n-1) + \log n$ ⟶ $O(n \log n) = O(n*\log n)$

$T(n) = 2T(n-1) + 1$ ⟶ $O(2^n) = O(1*2^n)$

$T(n) = 3T(n-1) + 1$ ⟶ $O(3^n) = O(1*3^n)$

$T(n) = 2T(n-1) + n$ ⟶ $O(n2^n) = O(n*2^n)$

$T(n) = 2T(n-2) + 1$ ⟶ $O(2^{n/2}) = O(1*2^{n/2})$

## General form:-

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

where $a > 0$, $b > 0$, $f(n) = O(n^k)$ where $k >= 0$

$O(f(n)) = O(n^k)$ ⟶ $a < 1$

$O(n * f(n)) = O(n^{k+1})$ ⟶ $a = 1$

$O(a^{n/b} * f(n)) = O(a^{n/b} * n^k)$ ⟶ $a > 1$

# Application on General theorem of decreasing fun.

## General form

$a\,T(n-b) + f(n)$

$f(n) \rightarrow f(n^k)$

| | |
|---|---|
| $a < 1$ | $\Rightarrow O(f(n)) = O(n^k)$ |
| $a = 1$ | $\Rightarrow O(n * f(n)) = O(n^{k+1})$ |
| $a > 1$ | $\Rightarrow O(a^{n/b} * f(n)) = O(a^{n/b} * n^k)$ |

$T(n) = T(n-1) + 1 \xrightarrow{a=1 \,\&\, f(n)=n^0} O(n * f(n)) = O(n \times n^0) = O(n)$

$T(n) = T(n-1) + n \xrightarrow{a=1,\ f(n)=n^1} O(n^{k+1}) = O(n^2)$

$T(n) = T(n-1) + \log n \xrightarrow{a=1,\ f(n)=\log n} O(n * f(n)) = O(n \log n)$

$T(n) = 2T(n-1) + 1 \xrightarrow{a>1} O(a^{n/b} * f(n)) = O(2^{n/1} * 1) = O(2^n)$

$T(n) = 3T(n-1) + 1 \xrightarrow{a>1} O(a^{n/b} * f(n)) = O(3^{n/1} * 1) = O(3^n)$

$T(n) = 2T(n-1) + n \xrightarrow{a>1} O(a^{n/b} * f(n)) = O(2^{n/1} * n^1) = O(n\,2^n)$

$T(n) = 2T(n-2) + 1 \xrightarrow{a>1} O(a^{n/b} * f(n)) = O(2^{n/2} * 1) = O(2^{n/2})$

# Master theorem for dividing functions

The general form

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

where $a \geq 0$, $b > 0$, $f(n) = O(n^k (\log n)^p)$

where $k >= 0$

→ Calculate $\log_b a$

Case 1 :-

if $\log_b a > k$ → $O(n^{\log_b a})$

Case 2 :-

if $\log_b a = k$ find p

if $p > -1$ → $O(n^k (\log n)^{p+1})$

if $p = -1$ → $O(n^k \log \log n)$

if $p < -1$ → $O(n^k)$

Case 3 :-

if $\log_b a < k$ find p

if $p \geq 0$ → $O(n^k (\log n)^p)$

if $p < 0$ → $O(n^k)$

Example 1 :-

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

$O(n^k (\log n)^p)$

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

$a = 4$ , $b = 2$ , $f(n) = n^1 (\log n)^0$

- $k = 1$
- $p = 0$

$$\log_b a = \log_2 4 = 2 > k$$

Case 1 $\Rightarrow$ $O(n^{\log_b a}) = O(n^2)$

Example 2 :-

$$T(n) = 9T(n/3) + 1$$

$a = 9$ , $b = 3$ , $f(n) = 1 = n^k (\log n)^p$

- $n^0 (\log n)^0$
- $k = 0$
- $p = 0$

$$\log_b a = \log_3 9 = 2 > k$$

So Case 1 $\Rightarrow$ $O(n^{\log_b a}) = O(n^2)$

Example 3:-

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

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

- $n^1 (\log n)^0$
  - $k = 1$
  - $p = 0$

$\log_b a = \log_2 2 = 1 = k$ & $p > -1$

So case 2-1 :- $O(n^k (\log n)^{p+1}) = O(n \log n)$

Example 4:-

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

$a = 4$ , $b = 2$ , $f(n) = n^2 (\log n)^1$

- $k = 2$
- $p = 1$

$\log_b a = \log_2 4 = 2 = k$ & $p > -1$

So case 2-2 :- $O(n^k (\log n)^{p+1}) = O(n^2 (\log n)^2)$

Example 5:-

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

$a = 2$ , $b = 2$ , $f(n) = n^1 (\log n)^{-1}$

- $k = 1$
- $p = -1$

$\log_b a = \log_2 2 = 1 = k$ & $p = -1$

So Case 2-2 :- $O(n^k \log\log n) = O(n \log\log n)$

Example 6 :-

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

$a = 2$ , $b = 2$ , $f(n) = n^1 (\log n)^{-2}$

- $k = 1$
- $p = -2$

$\log_b a = \log_2 2 = 1 = k$ & $p < -1$

So Case 2-3 :- $O(n^k) = O(n)$

Example 7:-

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

$a = 1$ , $b = 2$ , $f(n) = n^2 (\log n)^0$

$\rightarrow k = 2$

$\rightarrow p = 0$

$\log_b a = \log_2 1 = 0 < k$ , $p \geq 0$

$\therefore$ Case 3-1 :- $O(n^k \log^p n) = O(n^2 (\log n)^0)$

$= O(n^2)$

Example 8:-

$$T(n) = 4T(n/2) + n^3 / \log n$$

$a = 4$ , $b = 2$ , $f(n) = n^3 (\log n)^{-1}$

$\rightarrow k = 3$

$\rightarrow p = -1$

$\log_b a = \log_2 4 = 2 < k$ , $p < 0$

$\therefore$ Case 3-2 :- $O(n^k) = O(n^3)$

# Lecture 4

# Algorithm Design

Algorithm design is the branch of discrete mathematics and computer science that deals with research, development and implementation of sequential and a synchronous algorithms

Algorithm design refers to a method or a mathematical process for problem-solving and engineering algorithms

An algorithm is a set of self - contained sequence of instructions or actions that contains finite space or sequence and that will give us a result to a specific problem in a finite amount of time

It is a logical and mathematical approach to solve or crack a problem using any possible methods

Algorithm design follow a strategy which is a general concept (a way of thinking, and is independent of a special problem) :

1 - divide and conquer

2 - greedy techniques

3- dynamic programming

4- NP completeness

5 - backtracking algorithm

6- brute force algorithm

## Divide and Conquer

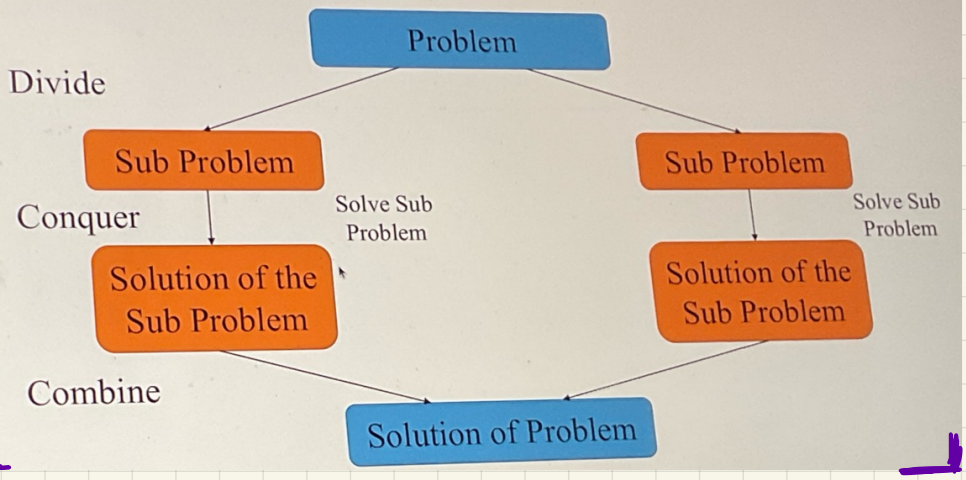
The divide-and-conquer strategy, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.

Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems.

The Divide and Conquer strategy path through 3 steps

1) Divide / Break
2) Conquer / Solve
3) Combine / Merge

## Divide and Conquer

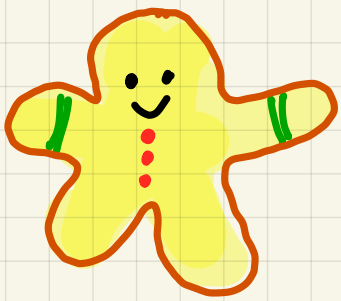



## Divide and Conquer

*Pros and Cons*

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously. (Advantage)

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored. (Disadvantage)



## Divide and Conquer

*Application that use Divide and Conquer*

Merge Sort
Quick Sort
Binary Search
Finding Maximum and Minimum
Strassen's Matrix Multiplication
Closest pair (points)
Karatsuba Algorithm