1. Time and Space Analysis of Algorithms

1.1 Asymptotic Notation

1.1.1 Big (oh)

→ used to represent exact or upper bound of growth i.e Worst Time

→ <u>Definition</u>:

$f(n) = O(g(n))$ iff

There exist constants c and n_0

$\because f(n) \le c\,g(n)$ for all $n \ge n_0$

This term comes from having one value above the highest term $\Rightarrow 2n + 1n = 3n$

<u>Example</u> $f(n) = 2n + 3$

<u>Sol</u> This can be written as $2n + 3 \le 3n$

where $c = 3$ $g(n) = n$

now $2n + 3 \le 3n$

$\Rightarrow 3 \le n$

$\therefore$ we get $n_0 = 3$

here $f(n)$ = Actual Time and space used by the Algorithm

$g(n)$ = A simplified function representing growth rate

c = A constant multiplier

n_0 = A threshold input size for which the inequality holds

Direct way-

1. Ignore lower order terms
2. Ignore leading term constant

eg:

$3n^2 + 5n + 6 \Rightarrow O(n^2)$

$3n + 10n\log n + 3 \Rightarrow O(n\log n)$

$10n^3 + 40n + 10 \Rightarrow O(n^3)$

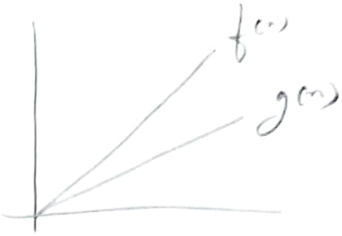
Big O notation for multiple variable

$100n^2 + 1000m + n \Rightarrow O(n^2+m)$

$1000m^2 + 200mn + 30m + 20n \Rightarrow O(m^2+mn)$

1.1.2 Omega Notation (Ω)

→ used to describe best case or lower bound performance of an Algorithm
or min time



Definition

A function $f(n)$ is said to be

$$f(n) = \Omega(g(n))$$

if there exist constants $c > 0$ and n_0

$\because$ for all $n > n_0$, $f(n) \geq c \cdot g(n)$

where $f(n)$: Actual time or space complexity

$g(n)$: function representing the lower bound

c : constant multiplier

n_0 : Threshold input size from which the inequality holds

Examples : $f(n) = 2n + 3$

$c \cdot g(n)$
$c = 1 \quad n = 0$

$\Rightarrow \quad 2n + 3 \geq 1n$

derived from $2 - 1 = 1$
$\Rightarrow 1$ less

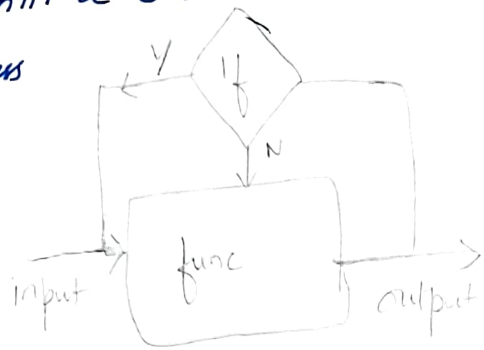
$\Rightarrow \quad -3 \leq n$

or $\quad n_0 \geq 0$

$\therefore \quad c = 1 \quad n_0 = 0$

4. Recursion

Definition: A function that calls itself, usually with a simpler input each time until a base case is reach which stops the process



\# Recursive function rely on call STACK to store info on active function calls

→ each time a function calls itself, a new frame is added to the stack

→ when the base case is reached, recursive calls starts returning and the stack begins to unwind

→ if a proper base case isn't provided, we get a stack overflow error

\# A good way to visualize Recursion is by using a stack trace

eg: a func to print numbers N to 1

sol

∵ n = 5

func(5)
├ 5
func(4)
├ 4
func(3)
├ 3
func(2)
├ 2
func(1)
├ 1
—X—

here func(n−1)
├ Print(n)
func(n−2)

Application of Recursion

1. Many Algorithms are based on Recussion
   eg: Dynamic Programming
   Back tracking
   Divide & Conquer

2. Many Problem inherently require Recursion
   eg: Tower of hanoi
   Graph Traversals

3. Real world Examples
   1. Traversing file Systems
   2. Parsing Nested Data Structures

Iteration vs Recursion

Definition

Concept	Recursion	Iteration
Method	func calls itself	uses loops
flow Control	func stack	loop ctrl var
End point	Base case	loop termination Condition

→ Recursion Breaks the problem into Sub problems
→ Iteration solves the problem by repeatly executing a block of code

every Recursive func can be converted into Iterative but the Reverse ~~it~~ can be done, but may not always be useful
(Recursion adds Function Call Overhead, which increases CPU time & memory)

When to Use Which?

Use Recursion When

1) Problem has recursive nature
   eg: Trees
2) You need cleaner, simpler Logic
3) Divide & conquer Algo

Use Iteration When

1) task involves repeated Action
2) Performance is critical
3) Liniear process like summing a list

Mnemonic

Recursion thinks like a Mathematician
Iteration works like a Machine

Tail Recursion

tail Recursion is a special kind of Recursion where:

recursive call is the Last Operation before returning the result, with no further computation needed Afterwards

eg:

```
func (int n)
{ if(n==0)
        return ;
  func (n-1);
  print (n);

}
```

Not Tail Recursive

```
func( int n, int k)
{  if (n==0)
       return
   print (k)
   func (n-1, k+1)
}
```

Tail Recursive

Some languages support TCO or Tail call optimization where the compiler:

1) Reuses current func stack frame instead of creating a new one
2) Avoids stack overflow

eg: ✓ JS, Scala, Java (Not supported but can be simulated)

X Python