Computer Engineering - Sem IV

NCMPC 41 : Design and Analysis of Algorithms

Module - 1 : Introduction to Design and Analysis of Algorithms (10 Hours)

Instructor : Mrs. Lifna C S

Department of Computer Engineering, VESIT, Mumbai

Topics to be covered

- **Introduction to Algorithm Analysis**
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Algorithm Vs Program

Algorithm	Vs.	Program
Design phase of S/w Development		- Implementation
Domain Knowledge		- Programmer
any lang/. { English, Mathematics		- Programming lang
Independent H/w & OS		- Dependent H/w & OS
Analysis		- Testing

Algorithm Vs Program

Aspect	Algorithm	Program
Definition	A step-by-step procedure to solve a problem logically.	A set of instructions written in a specific programming language to execute an algorithm.
Language	Represented in plain language, pseudocode, or flowcharts.	Written in a programming language like Python, Java, or C++.
Purpose	Focuses on *what to do* and *how to solve* a problem conceptually.	Focuses on *implementing* the solution on a computer.
Execution	Cannot be executed directly by a computer.	Can be executed directly by a computer after compilation or interpretation.
Dependency	Independent of programming languages and platforms.	Depends on specific programming languages and platforms for execution.

Priori Analysis Vs Posteriori Analysis

	Priori Analysis	Posteriori Testing
1.	Algorithms (analysis)	1. Programs
2.	Independent of lang.	2. lang. dependent
3.	H/w independent.	3. H/w dependent
4.	Time & Space fn	4. watch time & bytes

Priori Analysis Vs Posteriori Analysis

Key Concept	Priori Analysis	Posteriori Testing
Definition	Theoretical analysis conducted before testing, often based on mathematical or logical reasoning.	Empirical testing performed after implementation or execution to validate results.
Purpose	Predicts system behavior or performance without actual execution.	Verifies and validates outcomes based on observed data or results.
Approach	Deductive reasoning: focuses on theoretical predictions.	Inductive reasoning: relies on experimental or real-world evidence.
When Applied	Used in the design or planning phase of a system or algorithm.	Applied after implementation to test and validate the system or algorithm.
Examples	Analyzing time complexity of an algorithm (e.g., Big-O analysis).	Running test cases to measure actual runtime of an algorithm.

Characteristics of Algorithms

1. Input — 0/more
2. Output — atleast 1 o/p.
3. Definiteness — steps/stmts should be unambiguous/clear
4. Finiteness — must terminate @ some pt.

   eg Oracle Server / Web Server Runs until u/n stop.
5. Effectiveness — must be precise

   eg Cooking Receipe.
   Chemical Experiments

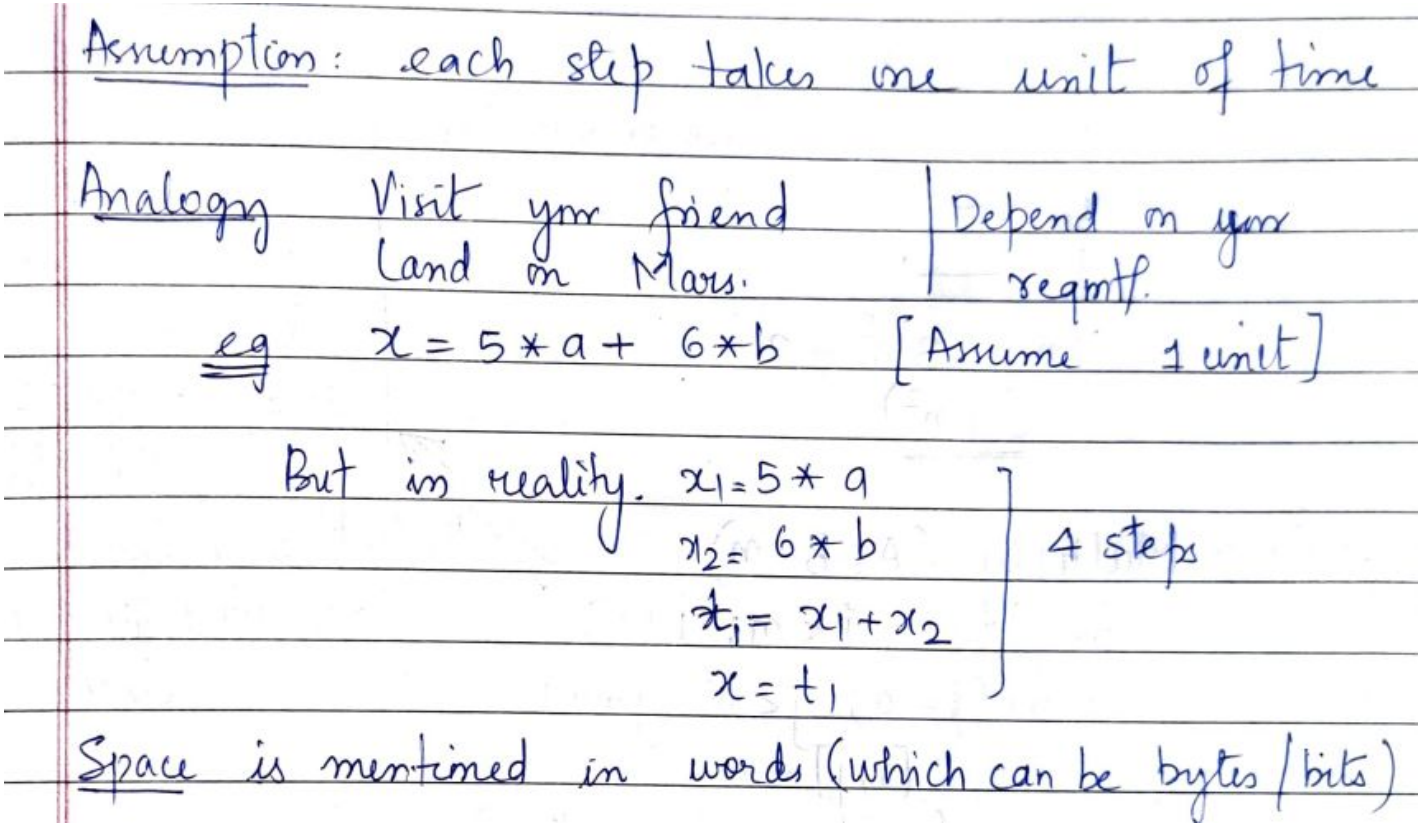
Topics to be covered

- Introduction to Algorithm Analysis
- **Performance analysis - Space and time complexity**
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

How to analyze an Algorithm

1. Time (time taken for execution)
2. Space (m/y reqd.)
3. N/w consumption As nowadays processing is done @ Cloud Computing
4. Power Consumptin Nowaday Handheld devices
5. CPU Registers Consumed

How to analyze an Algorithm



How to write an Algorithm

How to write an Algo.

Algorithm : Swap (a, b)

Begin

temp := a	—— 1
a := b	—— 1
b := temp	—— 1

End

$t(n) = 3$

As its is constant : $O(1)$

* No fixed syntax.
* No declarations on data types

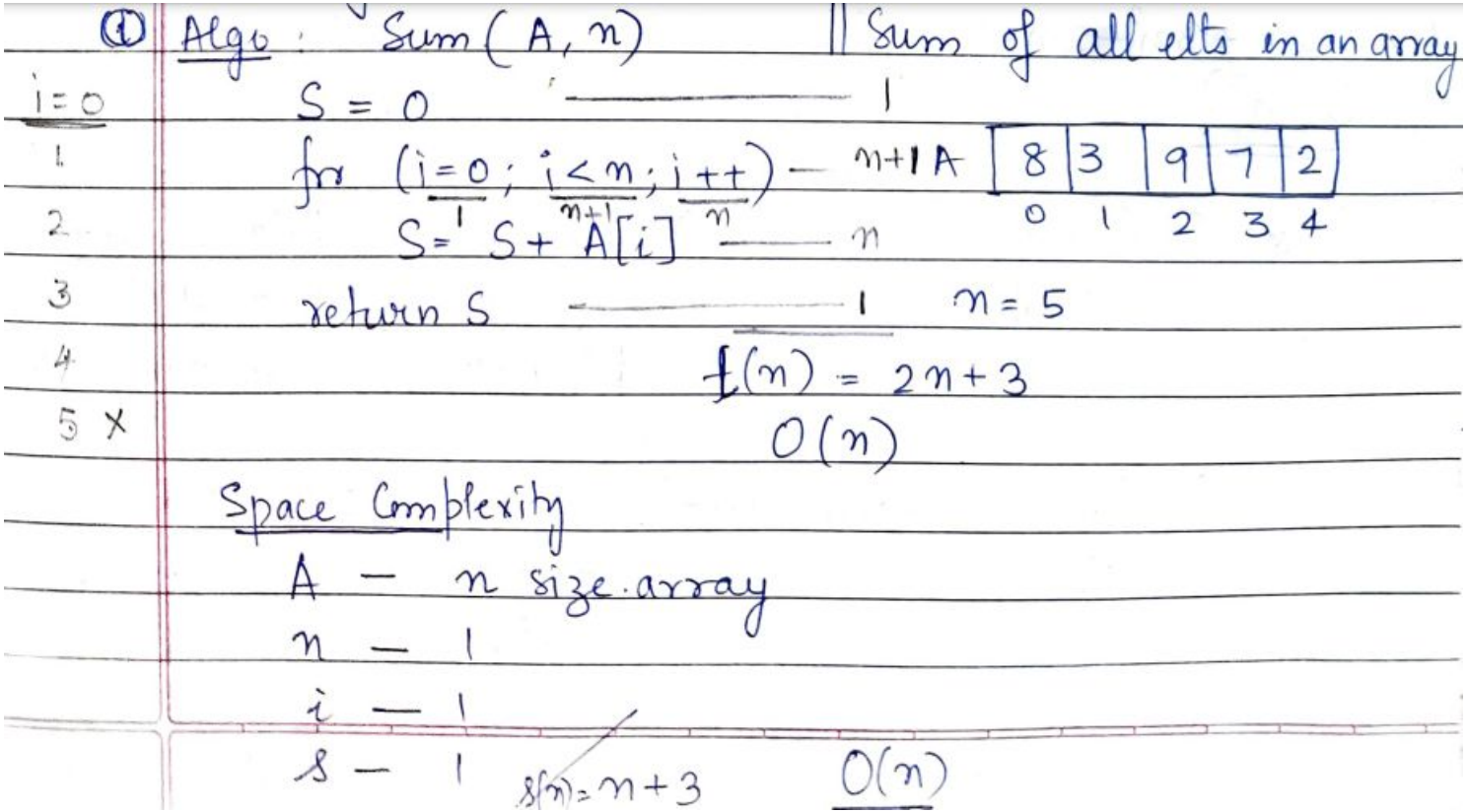
a — 1 m/y

b — 1 m/y

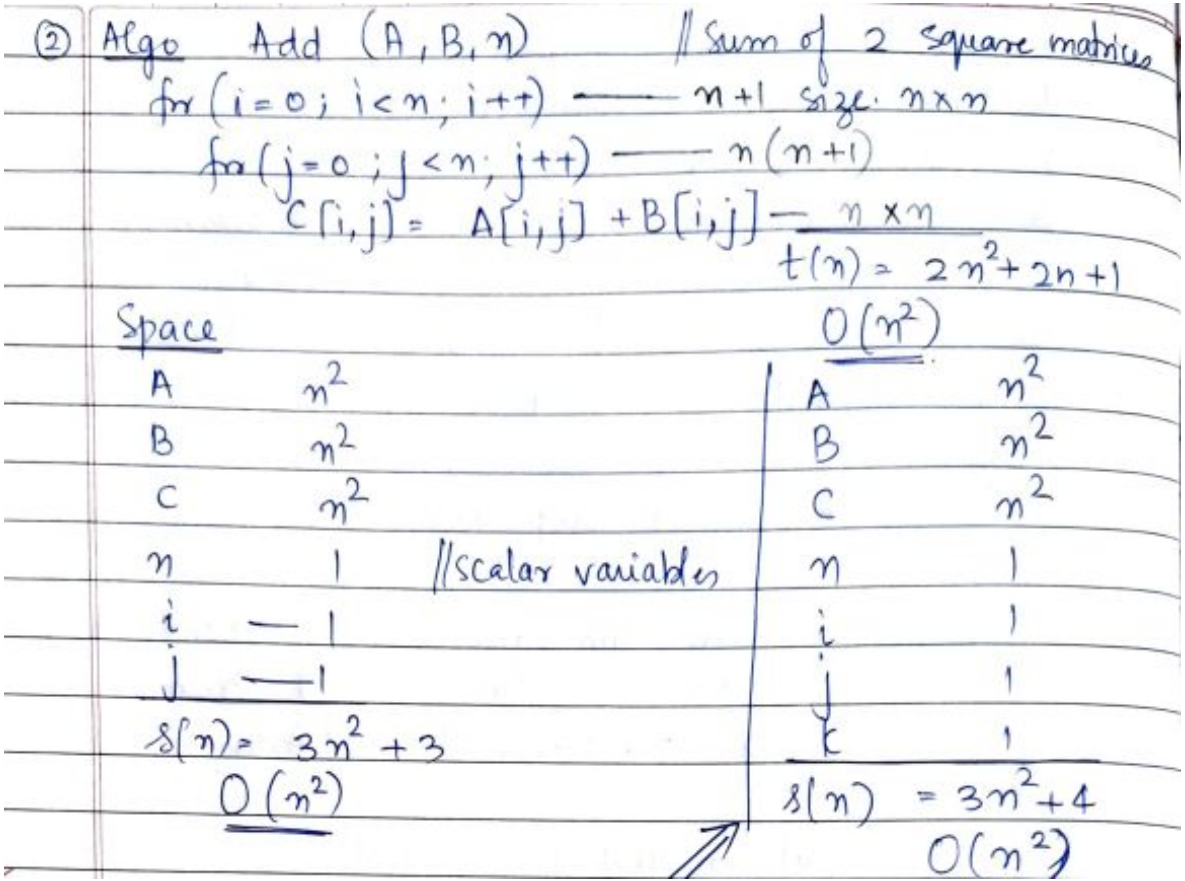
temp — 1 m/y

$S(n) = 3$ word

Frequency Count Method



Frequency Count Method



② Algo Add (A, B, n) // Sum of 2 square matrices

```
for (i=0; i<n; i++)        ---- n+1   size: n x n
    for(j=0; j<n; j++)     ---- n(n+1)
        C[i,j] = A[i,j] + B[i,j]  ---- n x n
```

$t(n) = 2n^2 + 2n + 1$

$O(n^2)$

Space

A	n^2	
B	n^2	
C	n^2	
n	1	//scalar variables
i	1	
j	1	

$s(n) = 3n^2 + 3$

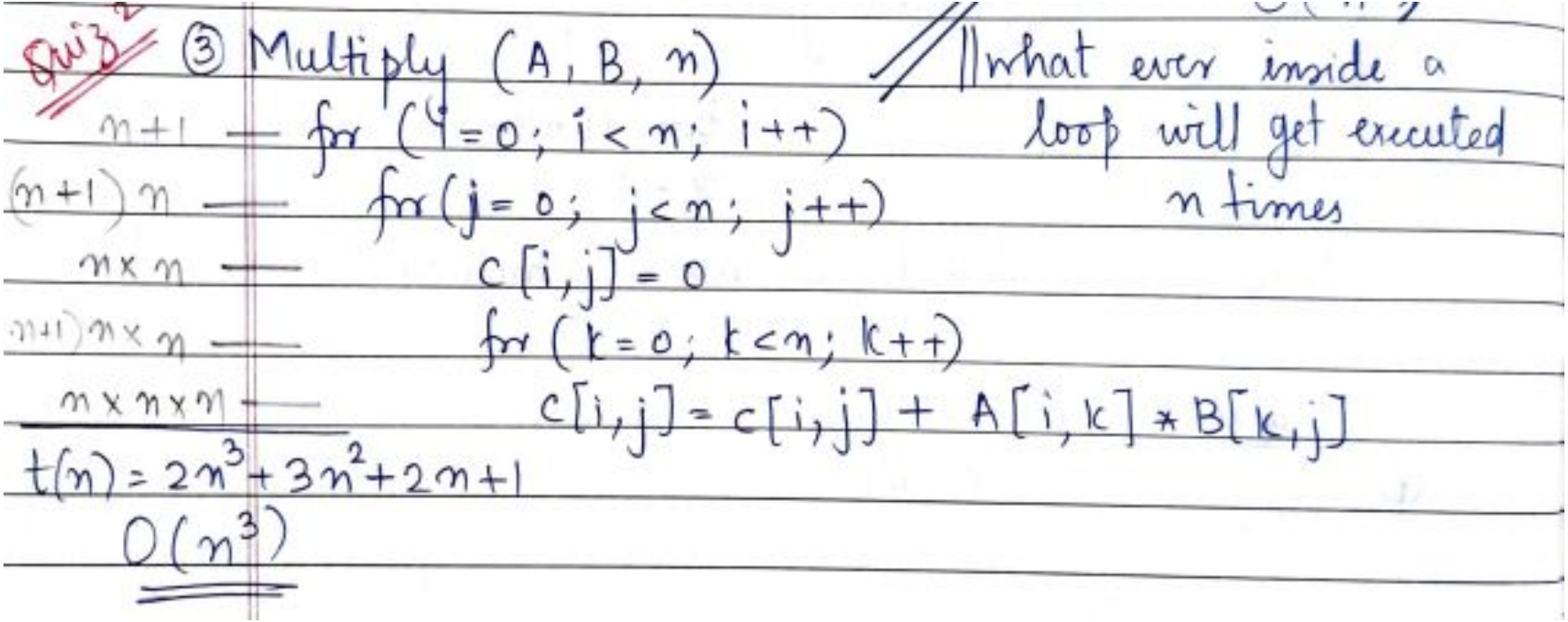
$O(n^2)$

A	n^2
B	n^2
C	n^2
n	1
i	1
j	1
t	1

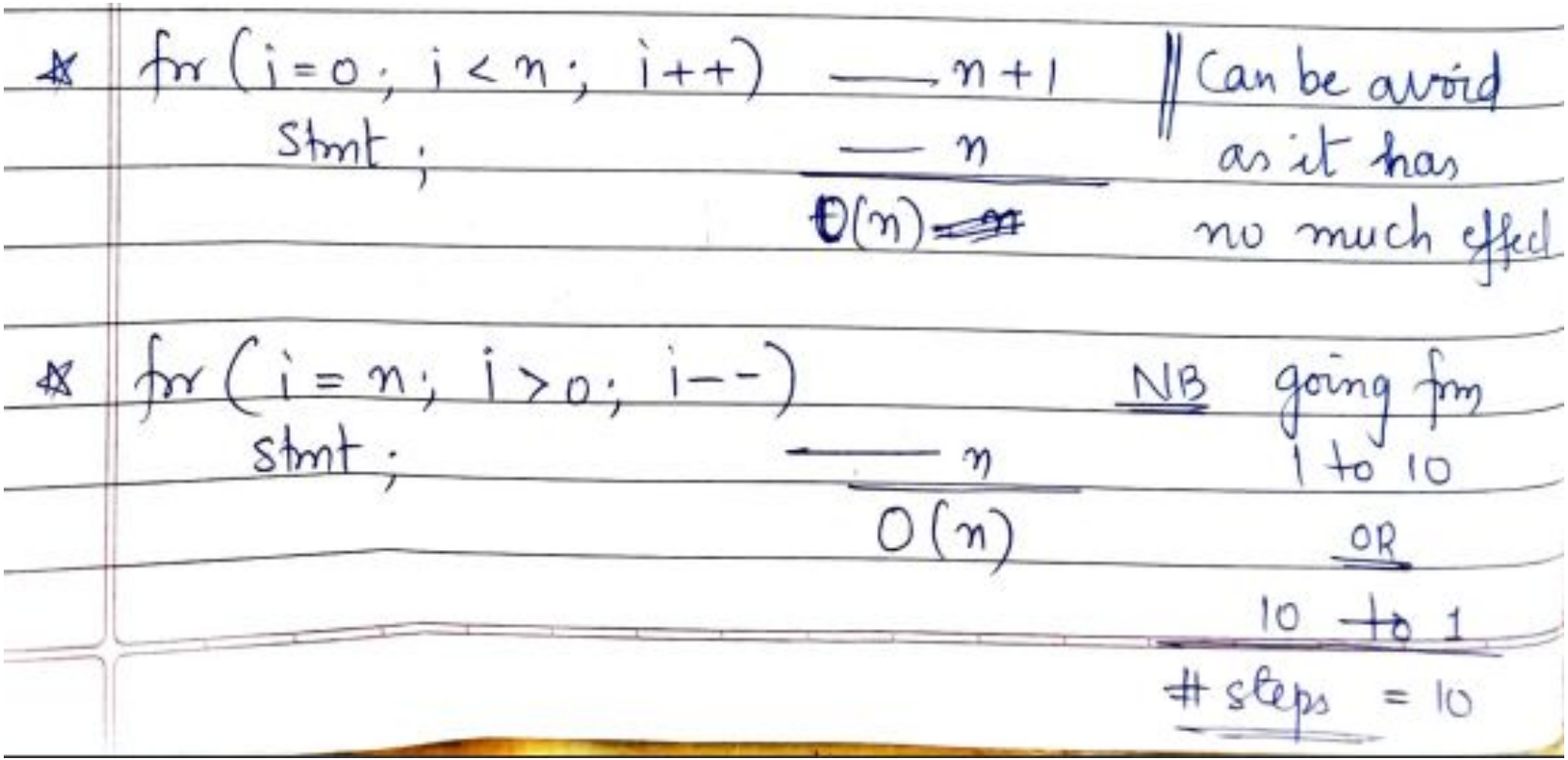
$s(n) = 3n^2 + 4$

$O(n^2)$

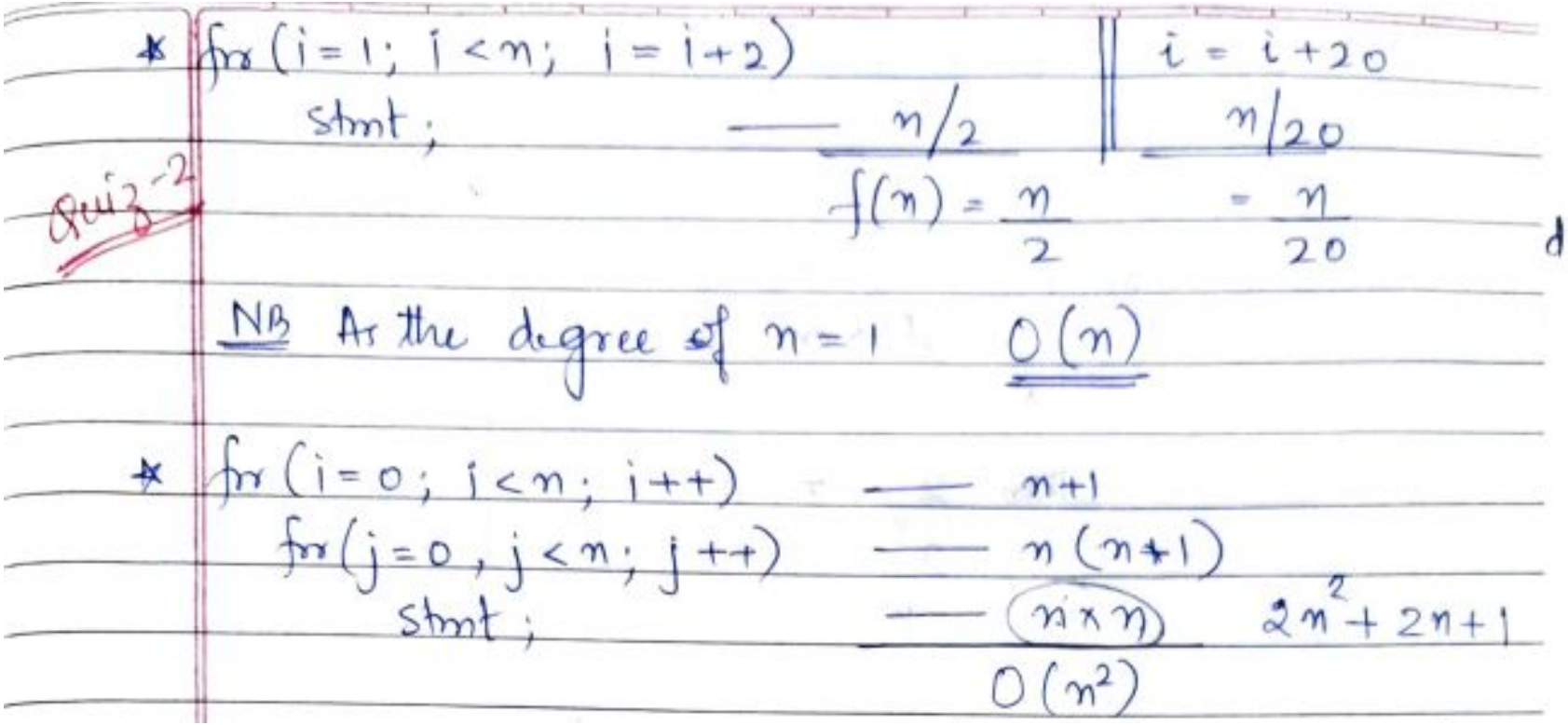
Frequency Count Method



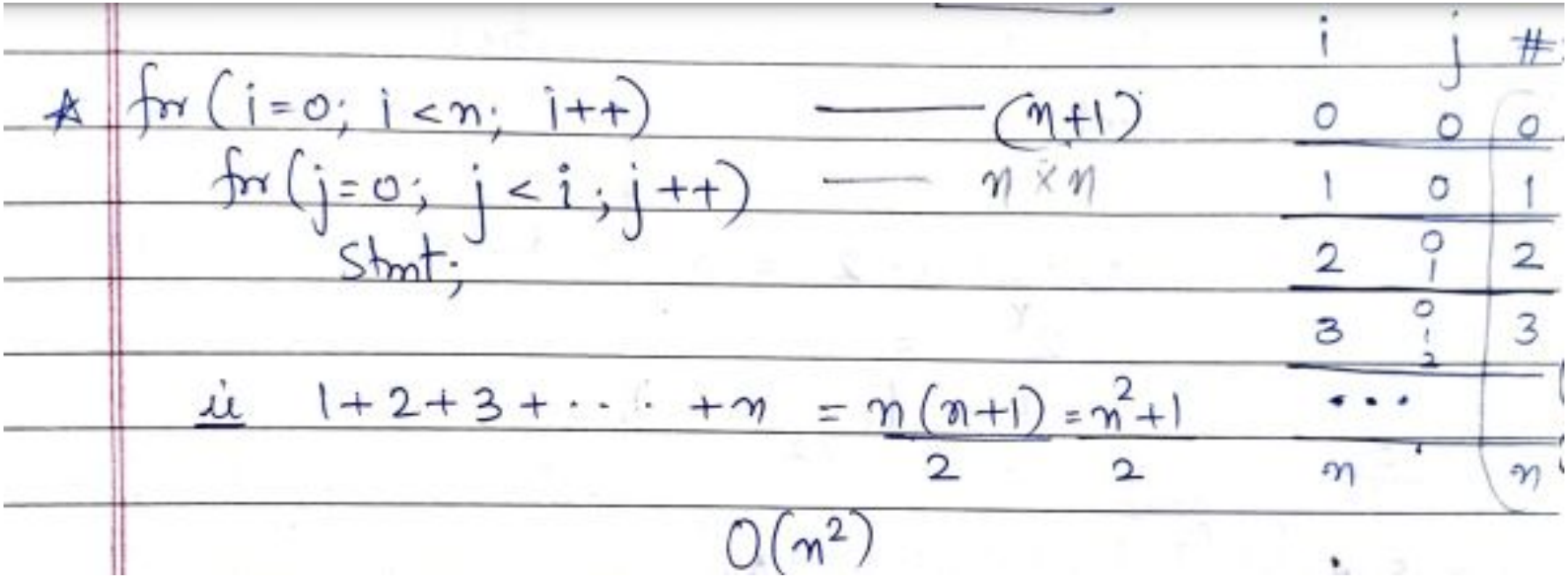
Frequency Count Method



Frequency Count Method



Frequency Count Method



Frequency Count Method

```
p = 0
for ( i = 1 ; p <= n ; i++)
    p = p + i
```

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3 = 6
. . .	
k	1+2+3+..+k

Assumption stops $p > n$

$\because p = \frac{k(k+1)}{2}$

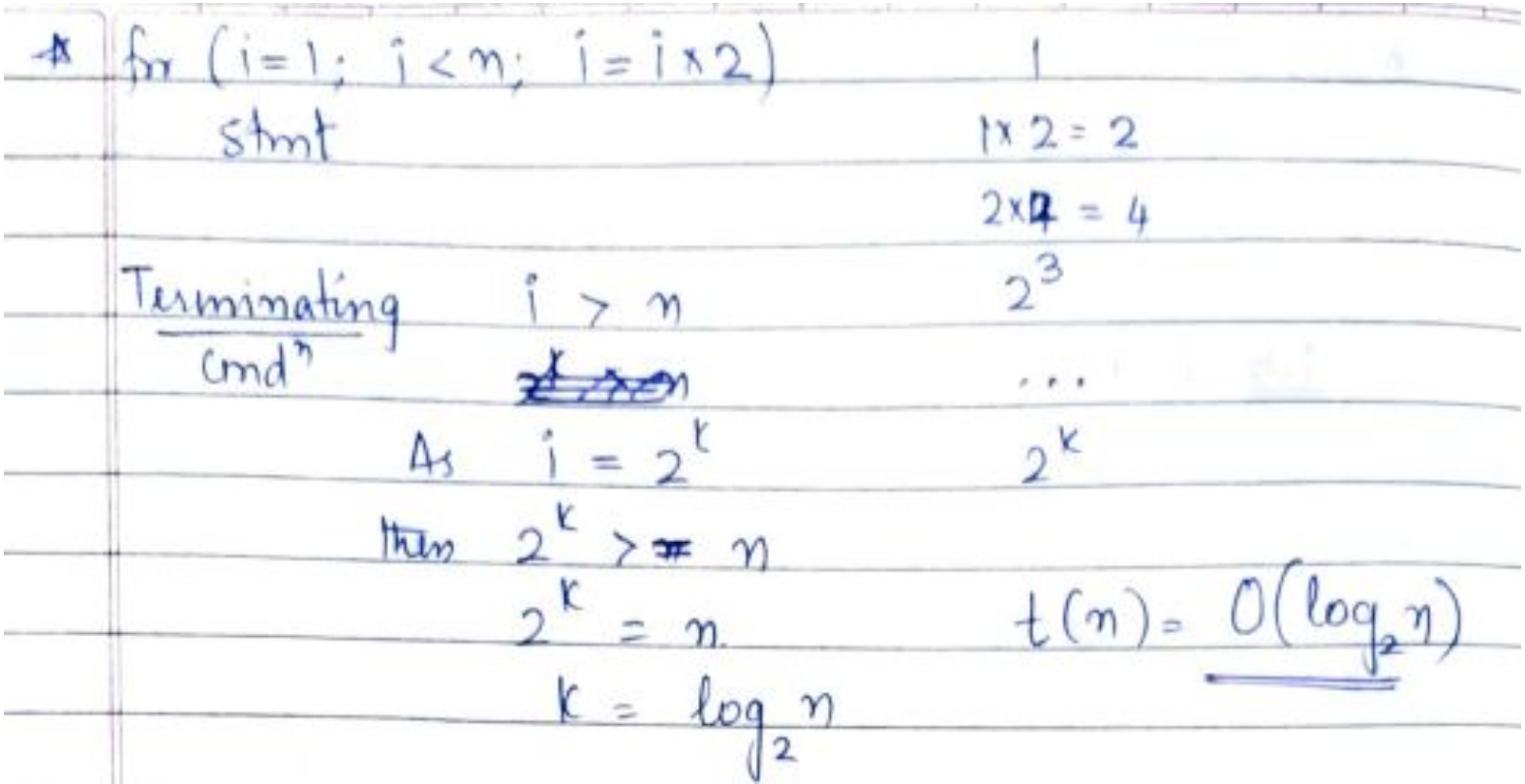
$\frac{k(k+1)}{2} > n$

$k^2 > n$

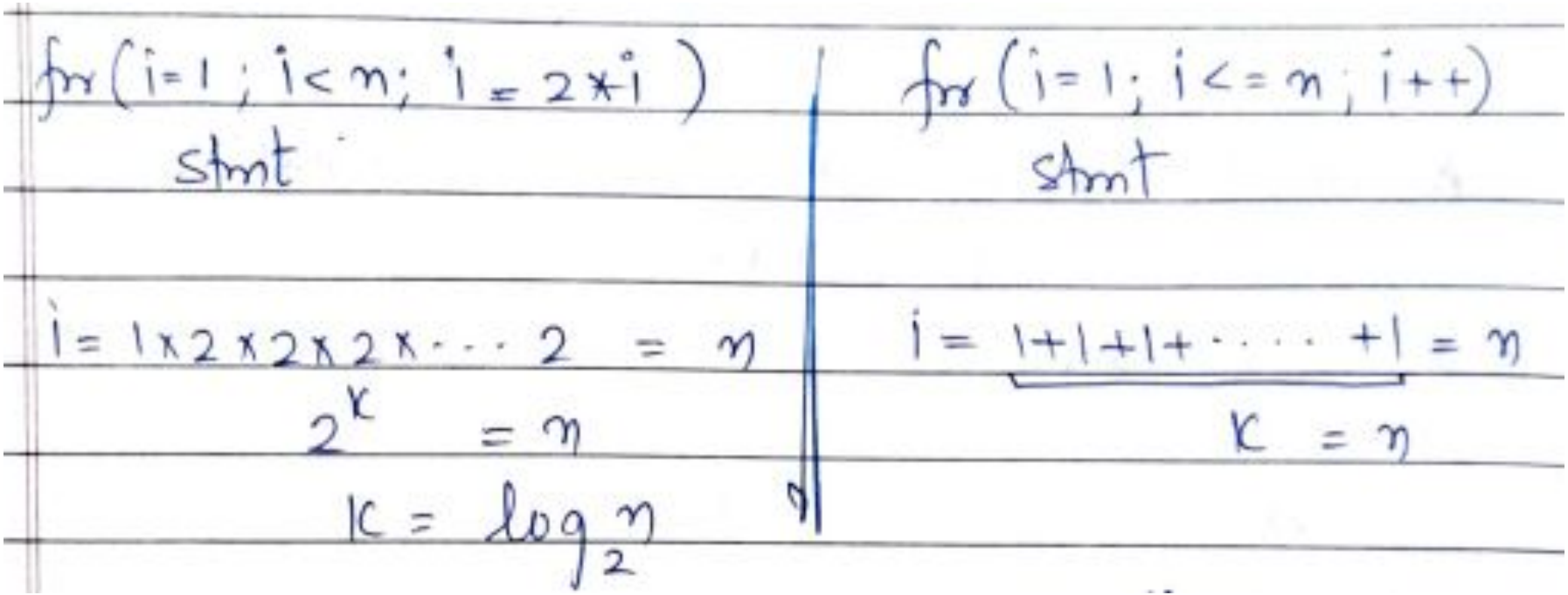
$k > \sqrt{n}$

$t(n) = O(\sqrt{n})$

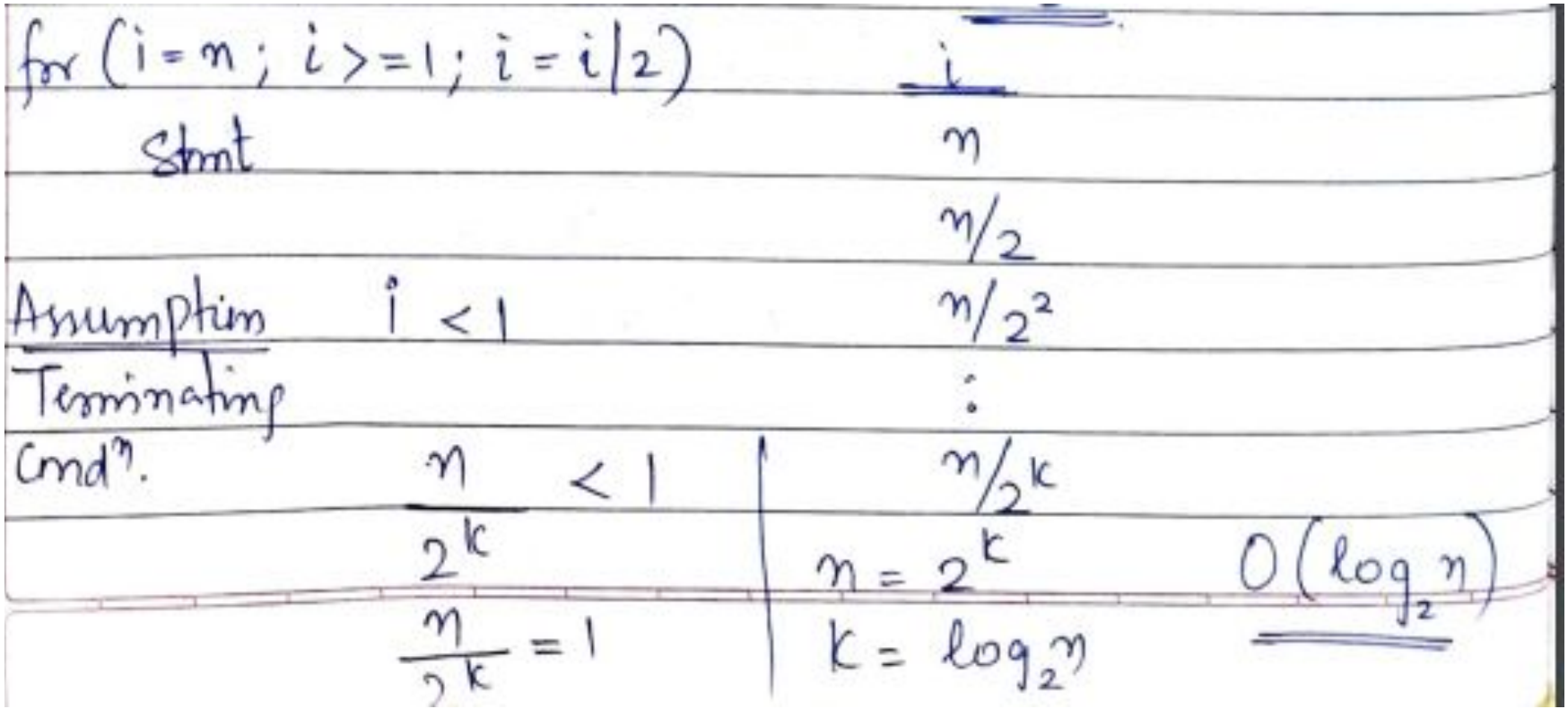
Frequency Count Method



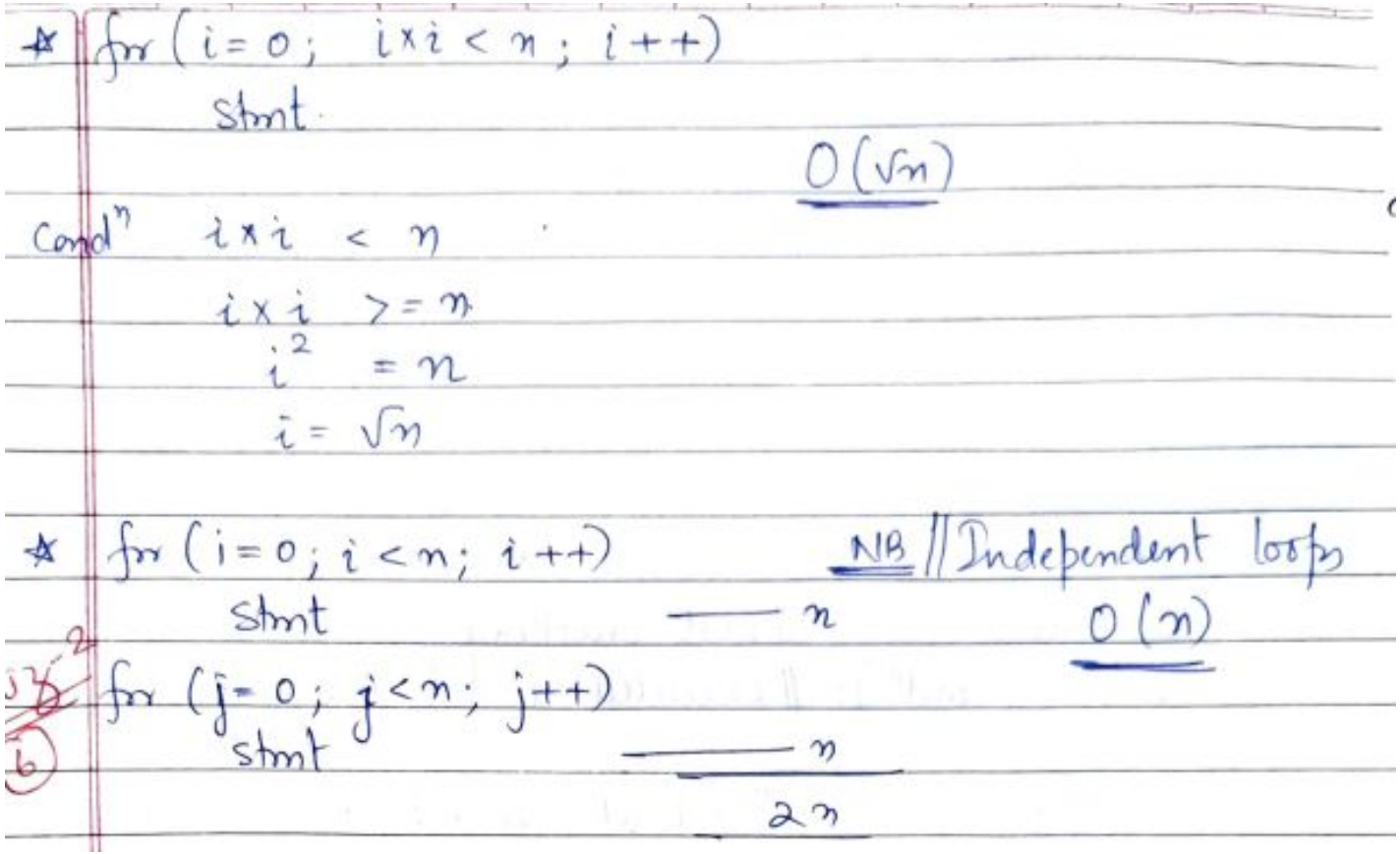
Frequency Count Method



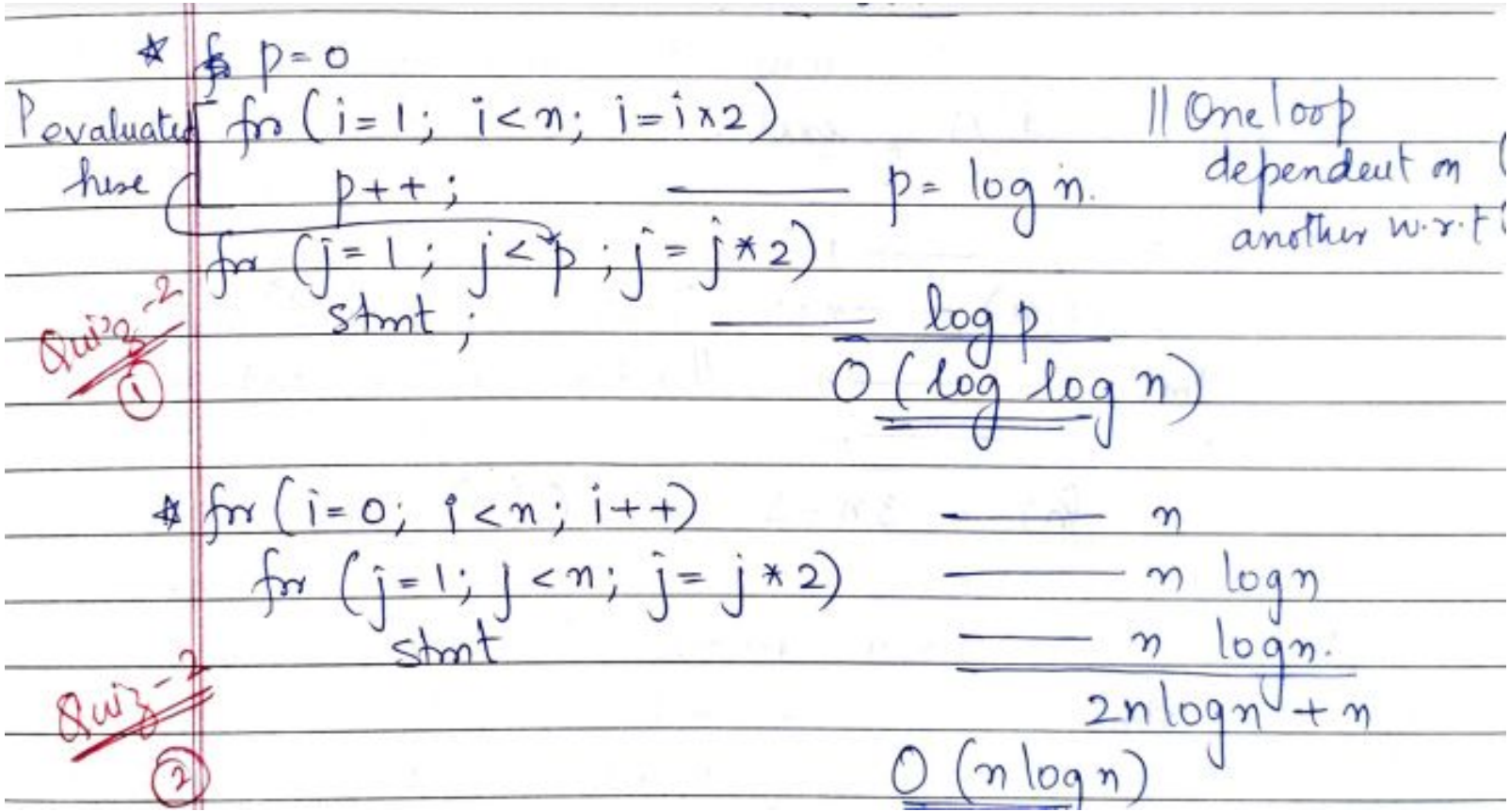
Frequency Count Method



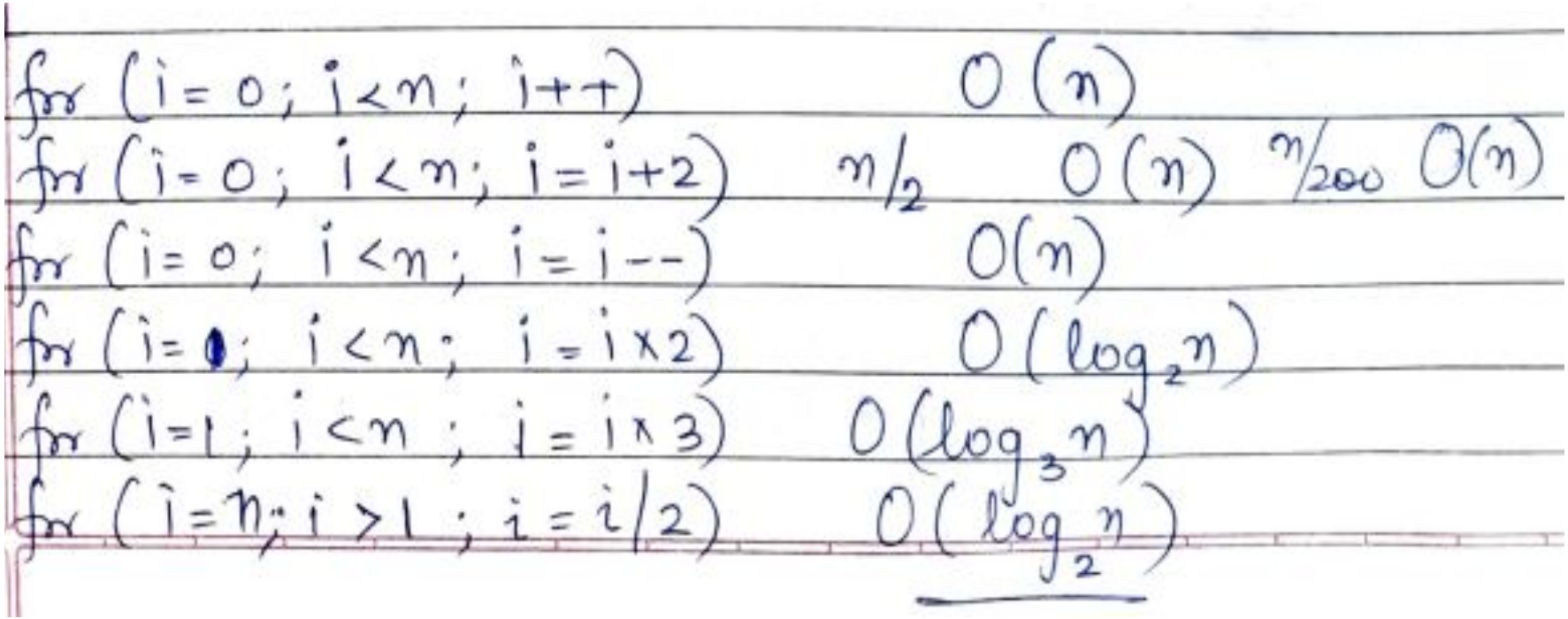
Frequency Count Method



Frequency Count Method



Frequency Count Method - Summary



Analysis of Loops - if , while

for (i=1 to n do stmt

step 1 — n+1 for loop.

— n

step 2

i = 1, 3, 5, ...

while (condn)
    stmt

// Needs to study the when the loop exit.

// pre checking

Analysis of Loops - if , while

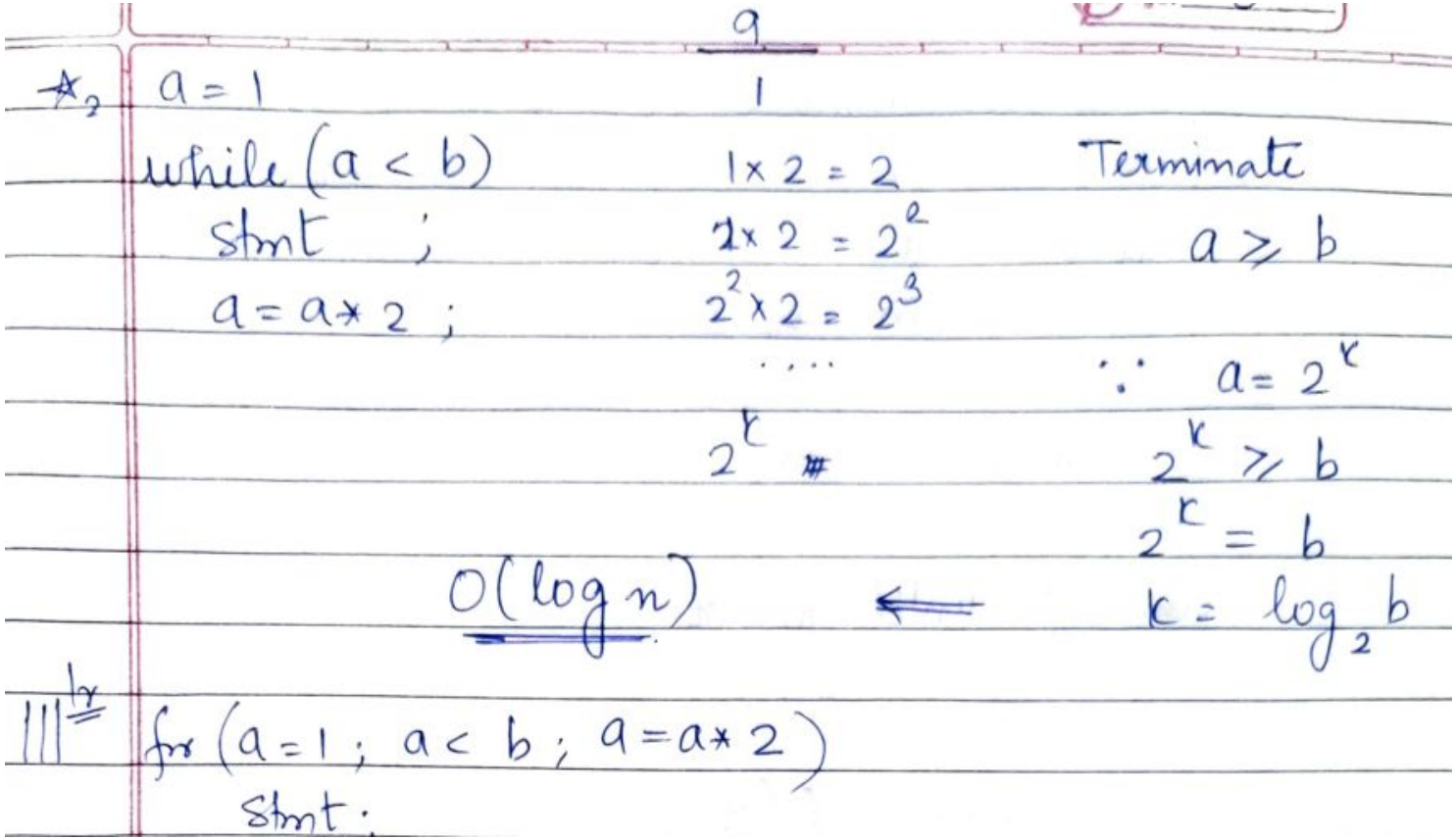
```
i = 0                          ---- 1
while (i < n)                  ---- n+1   //n times when cond^n is true
    stmt;                      ---- n     // 1 time when cond^n is false
    i++;                       ---- n
                                   ----
                                   2n+1

f(n) = 3n+2  ⟹  O(n)
```

```
for (i = 0;   i < n;   i++)        ---- n+1
     stmt     n+1      n           ---- n
                                   ----
                                   3n+2

                                   2n+1  ⟹  O(n)
```

Analysis of Loops - if , while



Analysis of Loops - if , while

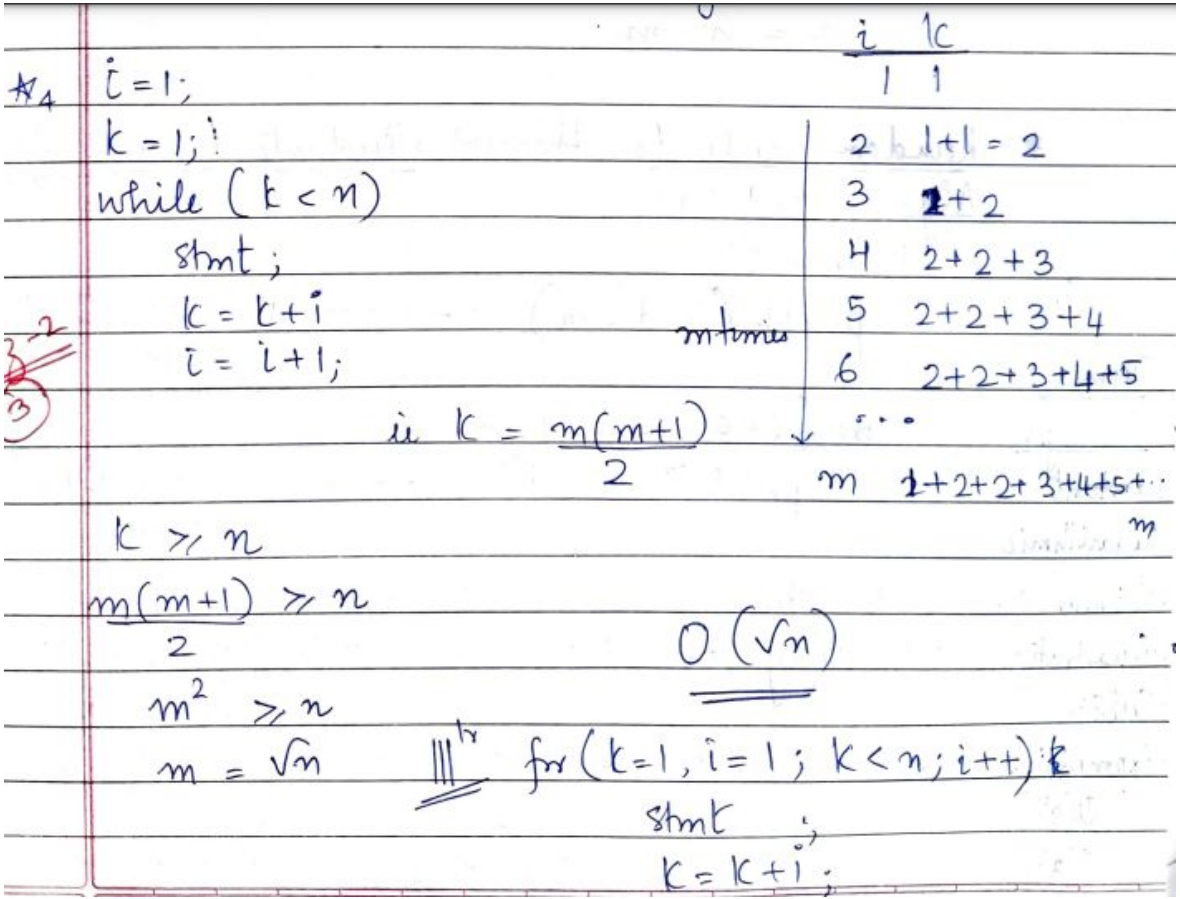
★ DAB 3

```
i = n;
while (i > 1)
    stmt;
    i = i/2;
```

```
for (i = n; i > 1; i = i/2)
    stmt
```

$\Longrightarrow O(\log_2 n)$

Analysis of Loops - if , while



Analysis of Loops - if , while

* GCD of two numbers m & n

```
while (m != n)
    if (m > n)
        m = m - n;
    else
        n = n - m;
```

Terminate: until m = n

min O(1)

max O(n)

```
for ( ; m != n; )
    if (m > n)
        m = m - n;
    else
        n = n - m;
```

let

m	n	
6	3	
3	3	1 time

m	n	
5	5	0 time

m	n	
16	2	
14	2	
12	2	7 times
10	2	
8	2	$16 \Rightarrow \frac{n}{2}\ \frac{n}{2}$
6	2	
4	2	
2	2	

Analysis of Loops - if , while

Algo Test (n)

```
if (n < 5)
    printf ("%d", n);        ---- 1   Best Case  O(1)
else
    for (i=0; i<n; i++)
        printf ("%d", i)     ---- n   Worst Case O(n)
```

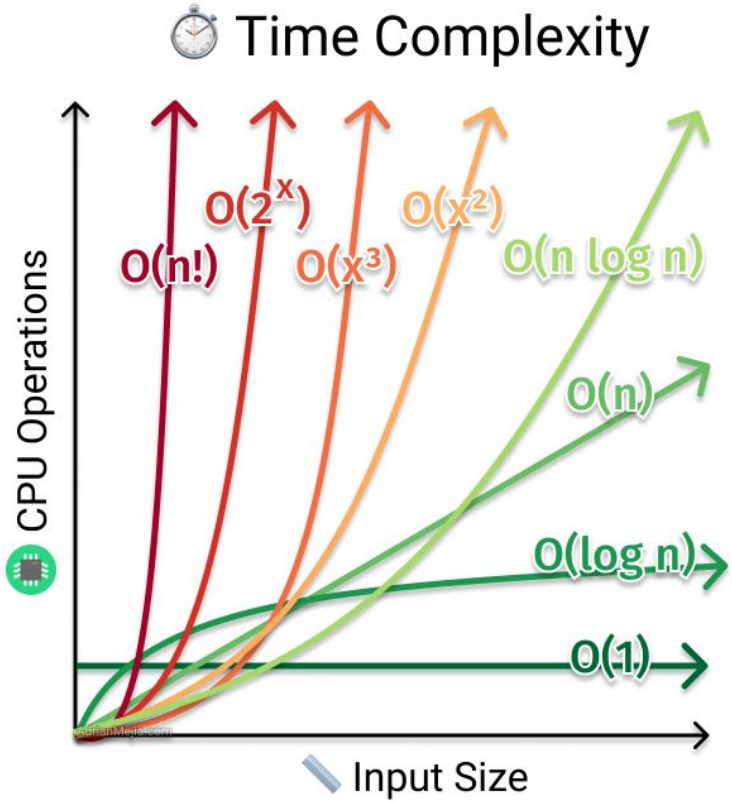
Cond^n stmts changes the execution time depending on the flow of the pgm snippet

Topics to be covered

- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- **Growth of function - Big-Oh, Omega, Theta notation**
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Types of Time Functions

Name	Time Complexity
Constant Time	$O(1)$
Logarthimc Time	$O(\log\ n)$
Linear Time	$O(n)$
Quasilinear Time	$O(n\ \log\ n)$
Quadratic Time	$O\left(n^2\right)$
Exponential Time	$O\left(2^n\right)$
Factorial Time	$O(n!)$



Courtesy : Kshitij Shah

Courtesy : Sumaiya Rinu

Types of Time Functions in ascending order

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots$$

$$< 2^n < 3^n < n^n \ldots < n^n$$

Asymptotic Notations

Asymptotic Notation { Method to describe the limiting behavior of the fn.

Big Oh	upper Bound	O
Big Omega	lower Bound	Ω
Theta	Average Bound	Θ ← Ideally Useful

if you cannot specify the exact time then we require Big O / Ω

Asymptotic Notations

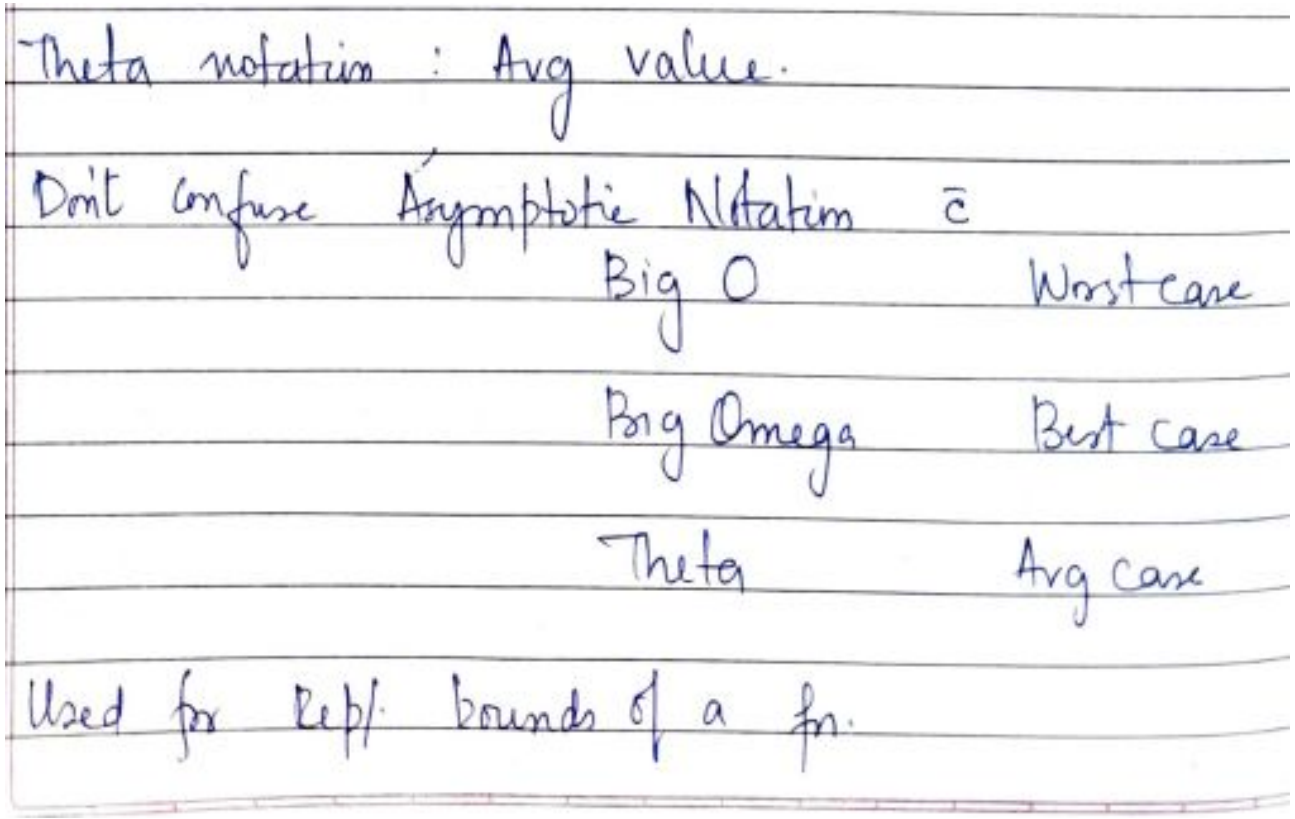
Analogy Best price for a Better phone

2000 ≤ ≤ 20,000 ≤ 1 lakh.

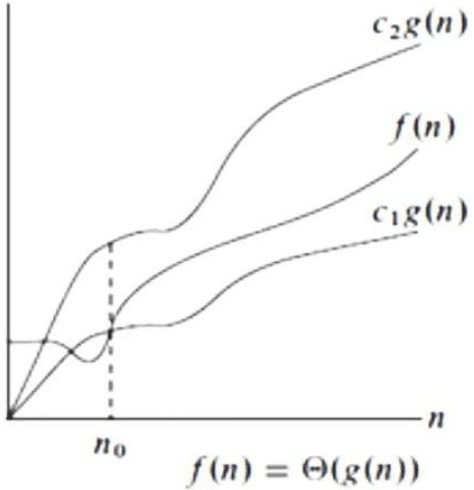
Θ — useful. extra fig/. exact time Complexity

Ω / O — { used when not clear abt the exact value

Asymptotic Notations

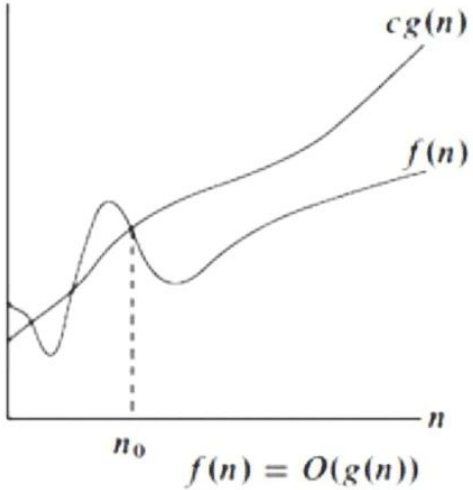


Asymptotic Notations



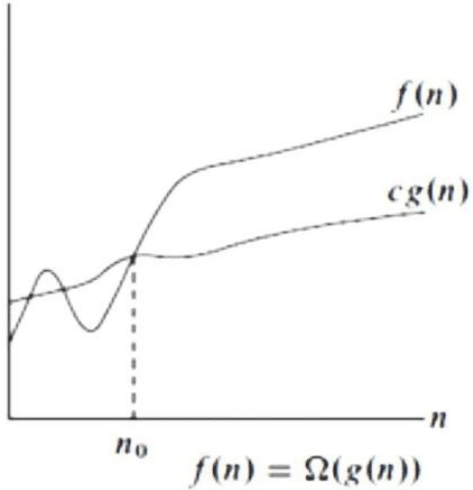
(a)

Theta



(b)

Big O



(c)

Omega

Courtesy : learnloner.com

Asymptotic Notations -- Big O

The fn $f(n) = O(g(n))$ iff $\exists$ +ve constants c and n_0.
such that, $f(n) \leq c\, g(n)$ $\forall$ $n \geq n_0$

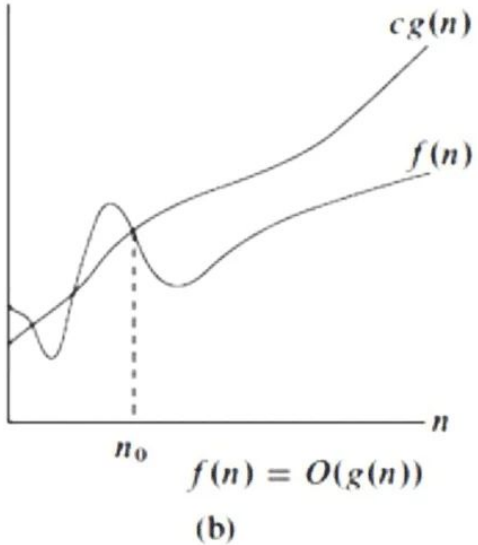
eg $f(n) = 2n + 3$ $\quad f(n) = O(n)$

$2n+3 \leq 10n \quad \forall n \geq 1$ closest fn
$f(n) \quad c\, g(n)$

OR

$2n+3 \leq 2n+3n$
$2n+3 \leq 5n \quad \forall n \geq 1$

OR $\quad 2n+3 \leq 2n^2+3n^2 \quad f(n) = O(n^2)$
$2n+3 \leq 5n^2 \quad \forall n \geq 1$



Big O

Courtesy : learnloner.com

Asymptotic Notations - Big Omega

The fn. $f(n) = \Omega(g(n))$ iff $\exists$ +ve constants c and n_0 such that $f(n) \geq c * g(n)$ $\forall$ $n \geq n_0$

eg $f(n) = 2n + 3$ → $f(n) = \Omega(n)$

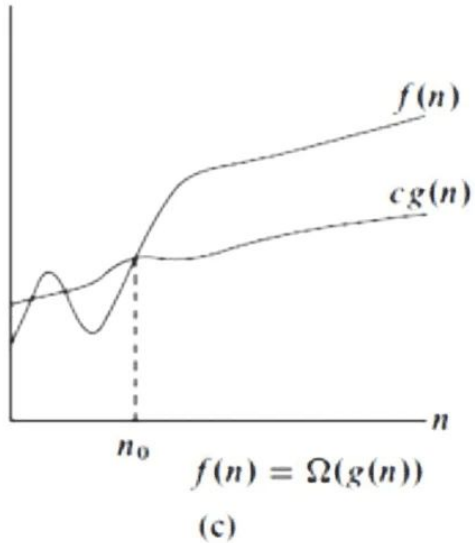
$2n+3 \geq 1 * n$ $\forall$ $n \geq 1$

$f(n)$ c $g(n)$

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

$f(n) = \Omega(\sqrt{n})$

True But not useful



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

(c)

Omega

Courtesy : learnloner.com

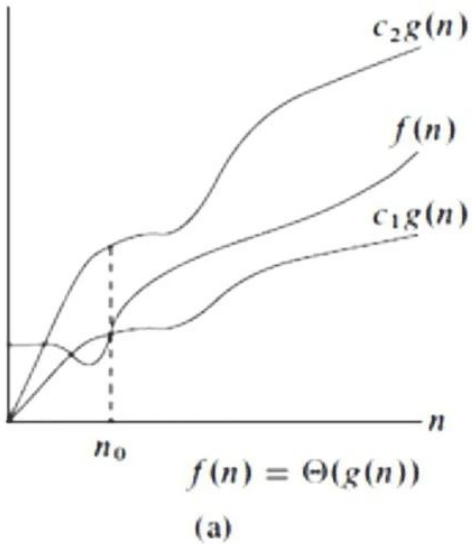
Asymptotic Notations - Big Theta

$f(n) = \Theta(g(n))$ iff $\exists$ +ve constants c_1 c_2 & n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$

eg $f(n) = 2n+3$

$1*n \leq 2n+3 \leq 5*n$ $\quad f(n) = \Theta(n)$

$c_1 g(n)$ $\quad f(n)$ $\quad c_2 g(n)$ $\quad f(n) \neq \Theta(n^2)$



(a)

Theta

Courtesy : learnloner.com

Asymptotic Notations - Examples

$f(n) = 2n^2 + 3n + 4$

$f(n) = O(n^2)$

$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$2n^2 + 3n + 4 \leq 9n^2 \quad \forall n \geq 1$

$c \uparrow \quad g(n) \uparrow$

$2n^2 + 3n + 4 \geq 1 * n^2$

$f(n) = \Omega(n^2)$

$1 * n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

ie $f(n) = \Theta(n^2)$

Asymptotic Notations - Examples

$f(n) = n^2 \log n + n$.

$1 * n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$.

$\Omega(n^2 \log n)$ $\qquad$ $O(n^2 \log n)$

$\Theta(n^2 \log(n))$

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

$n^2 \log n$

Asymptotic Notations - Examples

$f(n) = n! = n(n-1)(n-2)\cdots 3 \times 2 \times 1$

$1 \times 1 \times 1 \times 1 \times 1 \times 1 \leq 1 \times 2 \times \cdots \times n \leq n \times n \times n \times \cdots \times n$

$1 \leq n! \leq n^n$

$\Omega(1)$

$O(n^n)$

No tight bound

Θ

Asymptotic Notations - Examples

$f(n) = \log n!$

$\log(1 \times 1 \times \cdots \times 1) \leq \log(1 \times 2 \times 3 \times \cdots \times n) \leq \log(n \times n \times \cdots \times n)$

$1 \leq \log n! \leq \log n^n \Rightarrow [n \log n]$

$\Omega(1)$ $O(n \log n)$

No Θ. only upper bound & lower bound

Properties of Asymptotic Notations

Properties of Asymptotic Notations

General Property

1. if $f(n) = O(g(n))$ then $a * f(n) = O(g(n))$

eg $f(n) = 2n^2 + 5 = O(n^2)$

then

$7f(n) = 7(2n^2 + 5) = O(n^2)$

2. if $f(n) = \Omega(g(n))$ then $a * f(n) = \Omega(g(n))$

if $f(n) = \Theta(g(n))$ then $a * f(n) = \Theta(g(n))$

Properties of Asymptotic Notations

Reflexive ✱2

If $f(n)$ is given then $f(n) = O(f(n))$

NB $f(n)$ is upper / lower / tight bound of itself.

Transitive ✱3

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$

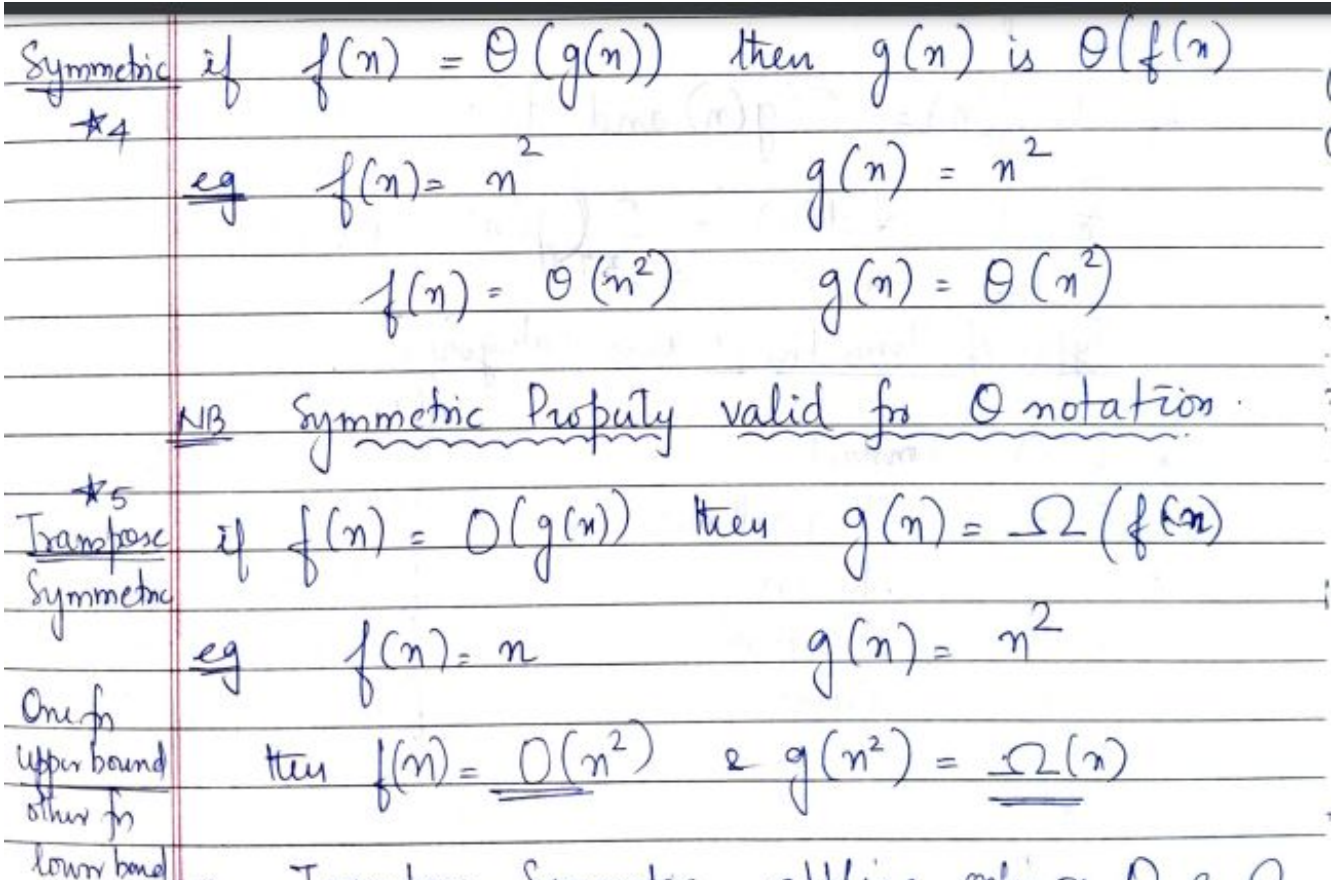
then $f(n) = O(h(n))$

eg $f(n) = n$ $g(n) = n^2$ $h(n) = n^3$

$f(n) = O(n^2)$ and $g(n) = O(n^3)$

ie $f(n) = O(n^3)$.

Properties of Asymptotic Notations



Properties of Asymptotic Notations - Summary

1. **Reflexive:**
   a. $f(n) = O(f(n))$,
   b. $f(n) = \Omega(f(n))$,
   c. $f(n) = \Theta(f(n))$
2. **Transitive:** if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
3. **Symmetric:** if $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$
4. **Addition:**
   a. $f(n) + g(n) = O(\max(f(n), g(n)))$,
   b. $f(n) + g(n) = \Theta(\max(f(n), g(n)))$
5. **Multiplication:**
   a. $f(n) * g(n) = O(f(n) * g(n))$,
   b. $f(n) * g(n) = \Omega(f(n) * g(n))$,
   c. $f(n) * g(n) = \Theta(f(n) * g(n))$

Properties of Asymptotic Notations - Summary

Property / Notation	Big O	Big Omega	Big Theta
General Property : If f(n) = O(g(n)) then a * f(n) = O(g(n))	✔	✔	✔
Reflexive Property : f(n) is upper / lower / tightly bound to itself.	✔	✔	✔
Transitive Property : If f(n) = O(g(n)) and g(n) = O(h(n)) then f(n) = O(h(n)	✔	✔	✔
Symmetric Property : If f(n) = Θ(g(n)) then g(n) = Θ(f(n))	✘	✘	✔
Transpose Symmetric : If f(n) = O(g(n) then g(n) = Ω(f(n))	✔	✔	✘
Addition: f(n) + g(n) = O(max(f(n), g(n))), f(n) + g(n) = Θ(max(f(n), g(n)))	✔	✔	✘
Multiplication: f(n) * g(n) = O(f(n) * g(n)), f(n) * g(n) = Ω(f(n) * g(n)), f(n) * g(n) = Θ(f(n) * g(n))	✔	✔	✔

Topics to be covered

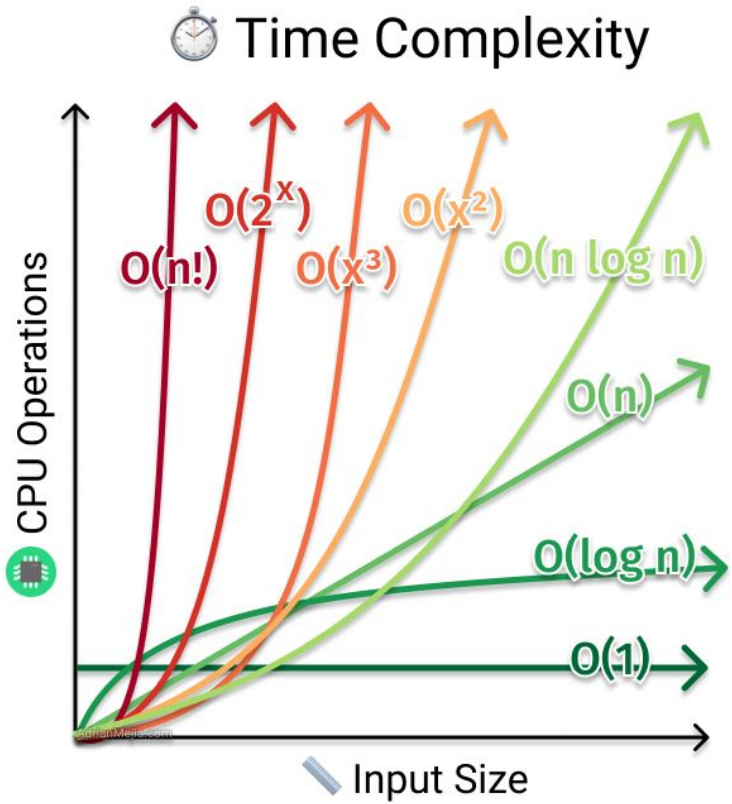
- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- **Mathematical background for algorithm analysis;**
- Analysis of selection sort, insertion sort.
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Comparing Time Functions

$\log n$	n	n^2	2^n	
0	1	1	2	
$\log_2 2 = 1$	2	4	4	
2	4	16	16	
3	8	64	256	$\leftarrow 2^n > n^2$
3.1	9	81	512	

Types of Time Functions

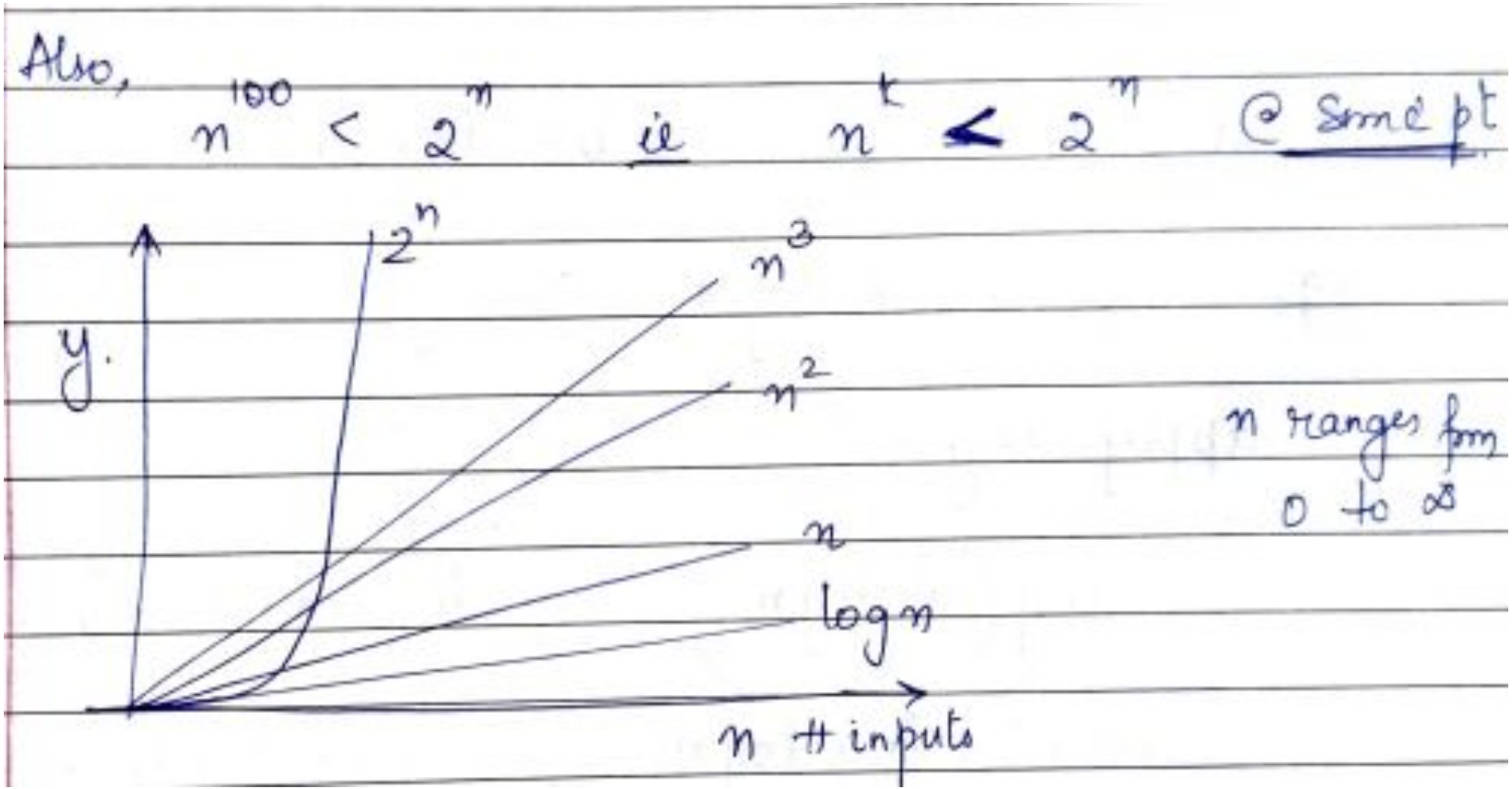
Name	Time Complexity
Constant Time	$O(1)$
Logarthimc Time	$O(\log\ n)$
Linear Time	$O(n)$
Quasilinear Time	$O(n\ \log\ n)$
Quadratic Time	$O\left(n^2\right)$
Exponential Time	$O\left(2^n\right)$
Factorial Time	$O(n!)$



Courtesy : Kshitij Shah

Courtesy : Sumaiya Rinu

Comparing Time Functions



Comparing Time Functions

Comparison of fns n^2 & n^3

Method 1: sample some values.

n	n^2	n^3
2	$2^2 = 4$	$2^3 = 8$
3	$3^2 = 9$	$3^3 = 27$
4	$4^2 = 16$	$4^3 = 64$.

$n^2 < n^3$

Method 2 : Apply log on both sides For easy comparison

$\log n^2 \quad \log n^3$

$2 \log n < 3 \log n.$

Formulas for logs

1. $\log ab = \log a + \log b$

2. $\log \frac{a}{b} = \log a - \log b$

3. $\log a^b = b \log a$

4. $a^{\log_c b} = b^{\log_c a}$

5. $a^b = n$ then $b = \log_a n$

Compare the following Time functions

No	f(n)	g(n)
1.	$n^2 \log n$	$n(\log n)^{10}$
2.	$3n^{\sqrt{n}}$	$2^{\sqrt{n} \log n}$
3.	$n^{\log n}$	$2^{\sqrt{n}}$
4.	$2^{\log n}$	$n^{\sqrt{n}}$
5.	$2n$	$3n$
6.	2^n	2^{2n}
7.	$g_1(n) = \begin{cases} n^3, & n < 100 \\ n^2, & n >= 100 \end{cases}$	$g_2(n) = \begin{cases} n^2 & n < 10000 \\ n^3 & n >= 10000 \end{cases}$

Comparing Time Functions

eg. $f(n) = n^2 \log n$ > $g(n) = n (\log n)^{10}$

apply log,

$\log(n^2 \log n)$	$\log(n (\log n)^{10})$
$\log n^2 + \log \log n$	$\log n + \log (\log n)^{10}$
$2 \log n + \log \log n$	$\log n + 10 \log \log n$

Comparing Time Functions

$f(n) = 3n^{\sqrt{n}} > g(n) = 2^{\sqrt{n}\log n}$

$3n^{\sqrt{n}}$ $\quad 2^{\sqrt{n}\log n} \xrightarrow[F_3]{} 2^{\log_2 n^{\sqrt{n}}} \xrightarrow[F_4]{} (n^{\sqrt{n}})^{\log_2 2}$

$3n^{\sqrt{n}}$ $\quad (n^{\sqrt{n}})^1$

$3n^{\sqrt{n}}$

NB Applied the formulas of log. as F_3 & F_4

NB Asymptotically $3n^{\sqrt{n}} \equiv n^{\sqrt{n}}$ || But, value wise unequal

Comparing Time Functions

$f(n) = n^{\log n}$ $<$ $g(n) = 2^{\sqrt{n}}$

apply log,

$\log(n^{\log n})$ $\log 2^{\sqrt{n}}$

$\log n * \log n$ $\sqrt{n} \log_2 2$

$(\log^2 n)$ $\sqrt{n} = n^{1/2}$

$\log(\log n)^2$ $\log n^{1/2}$

apply log,

$2 \log \log n$ $\frac{1}{2} \log n$

Comparing Time Functions

$f(n) = 2^{\log_2 n} < g(n) = n^{\sqrt{n}}$

apply log (F4)

$\log 2^{\log n}$ $\log(n^{\sqrt{n}})$

$\log n \times \log_2 2$ $\sqrt{n} \log n$

$\log n$ $\sqrt{n} \log n$

Comparing Time Functions

$f(n) = 2n$ $g(n) = 3n$ // Directly from fns we can cancel coeffic

Asymptotically equal.

$f(n) = 2^n$ $<$ $g(n) = 2^{2n}$ $= 4^n$

$\log 2^n$ $\log 2^{2n}$

$n \log_2 2$ $2n \log_2 2$

NB

Here dont cancel coefficient

Because already we have applied log

Comparing Time Functions

$$g_1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$$

$$g_2(n) = \begin{cases} n^2 & n < 10,000 \\ n^3 & n \geq 10,000 \end{cases}$$

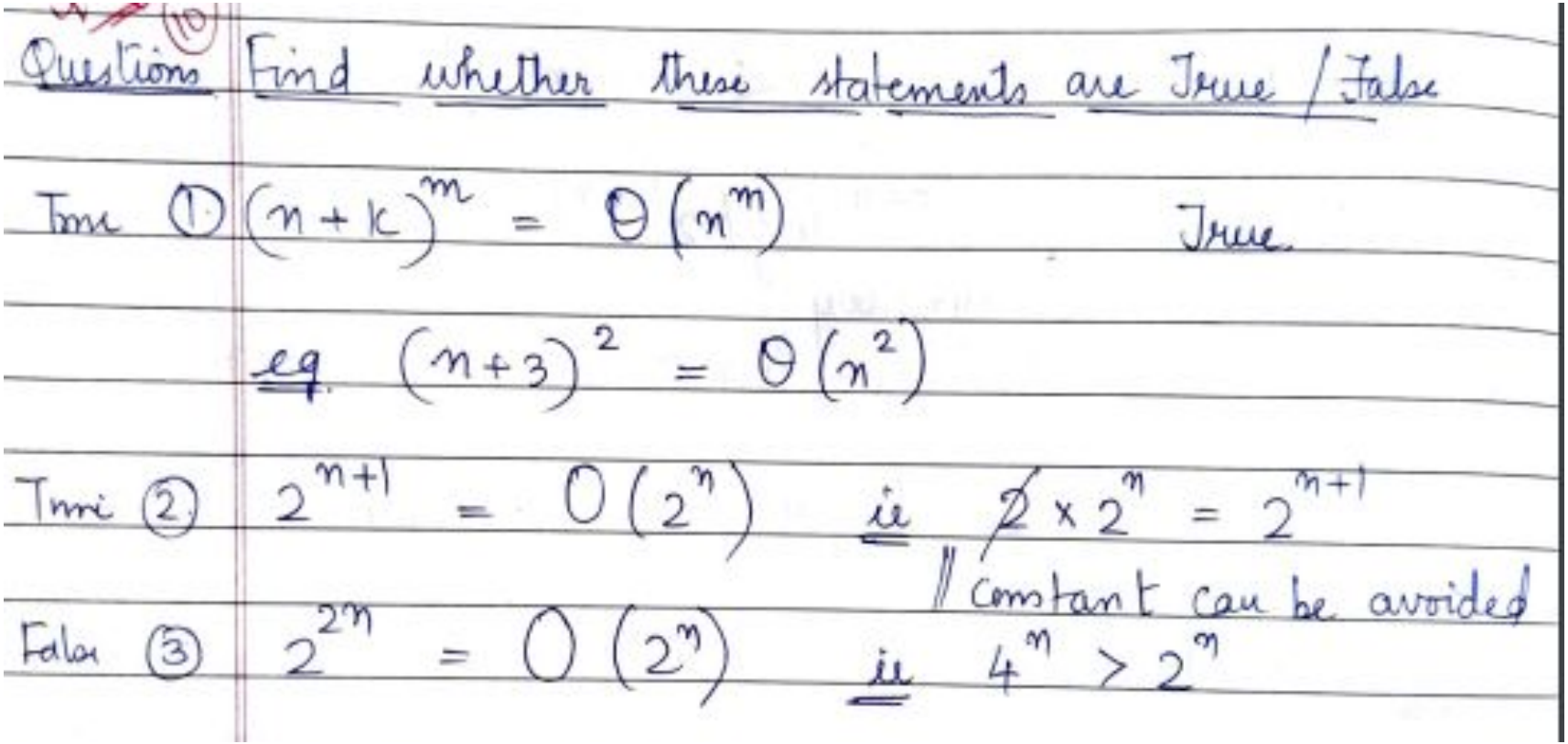
$n \longrightarrow$

100 10,000

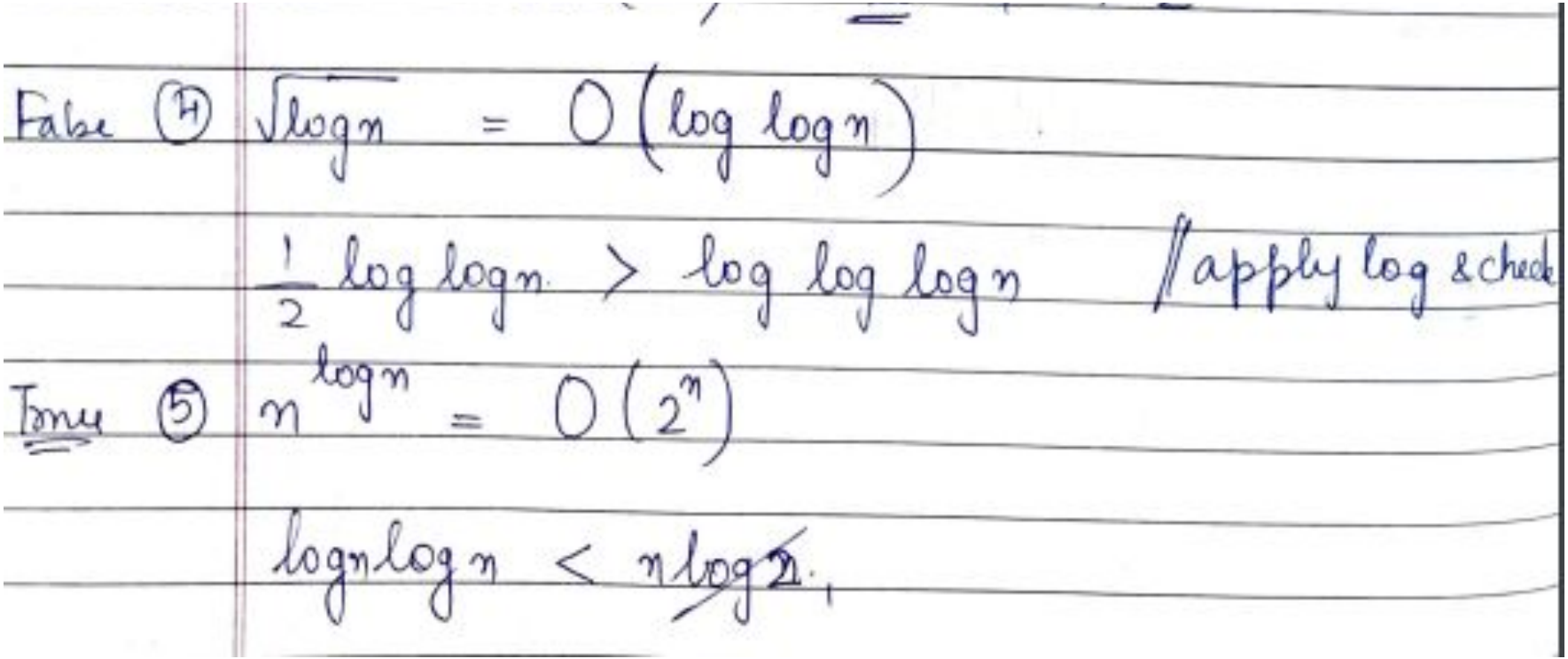
$g_1(n) > g_2$ $g_1 = g_2$ $g_2 > g_1$

From, $n = 10,000$ onwards $g_2 > g_1$

Comparing Time Functions



Comparing Time Functions



Topics to be covered

- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- **Analysis of selection sort, insertion sort, Linear Search, Binary search**
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Selection Sort Algorithm

```
fori← 1 to n-1 do
    min j ←i;
    min x ← A[i]
    for j ←i + 1 to n do
        if A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A [i]
    A[i] ← min x
```

Courtesy : shiksha.com

Selection Sort - Example

Step 0: Consider the Initial Array, arr[] = {17, 34, 25, 49, 09}

The entire array is currently unsorted. The sorted portion is empty.

Step 1: 1st Iteration

Find the smallest element in the entire array and swap it with the first position.

i	j	minindex	arr[]	Comparison (arr[i] < arr[j])	Swap?	Updated Array
0	1	0	[17, 34, 25, 49, 9]	17 < 34	No	[17, 34, 25, 49, 9]
0	2	0	[17, 34, 25, 49, 9]	17 < 25	No	[17, 34, 25, 49, 9]
0	3	0	[17, 34, 25, 49, 9]	17 < 49	No	[17, 34, 25, 49, 9]
0	4	4	[17, 34, 25, 49, 9]	17 > 9	Yes	[9, 34, 25, 49, 17]

- arr[] = {09, 34, 25, 49, 17}
- The first element, 09 is now in its correct position,
- The array is divided into a sorted portion {09} and an unsorted portion {34, 25, 49, 17}.

Selection Sort - Example

Step 2: 2nd Iteration

Find the smallest element in the remaining unsorted portion {34, 25, 49, 17} and swap it with the second position.

i	j	minindex	arr[]	Comparison (arr[i] < arr[j])	Swap?	Updated Array
1	2	1	[9, 34, 25, 49, 17]	34 > 25	Yes	[9, 25, 34, 49, 17]
1	3	2	[9, 25, 34, 49, 17]	25 < 49	No	[9, 25, 34, 49, 17]
1	4	4	[9, 25, 34, 49, 17]	25 > 17	Yes	[9, 17, 34, 49, 25]

- arr[] = {09, 17, 34, 49, 25}
- Here, sorted portion is {09,17} and unsorted portion is {25,49,34}

Selection Sort - Example

Step 3: 3rd Iteration

Find the smallest element in the remaining unsorted portion {25, 49, 34} and swap it with the third position.

i	j	minindex	arr[]	Comparison (arr[i] < arr[j])	Swap?	Updated Array
2	3	2	[9, 17, 34, 49, 25]	34 < 49	No	[9, 17, 34, 49, 25]
2	4	2	[9, 17, 34, 49, 25]	34 > 25	Yes	[9, 17, 25, 49, 34]

- At the end of this iteration, element 25 is already in its correct position.
- arr[] = {09, 17, 25, 49, 34}
- Here, sorted portion is {09,17,25} and unsorted portion is {49,34}

Selection Sort - Example

Step 4: 4th Iteration

Find the smallest element in the remaining unsorted portion {49, 34} and swap it with the fourth position.

i	j	minindex	arr[]	Comparison (arr[i] < arr[j])	Swap?	Updated Array
3	4	3	[9, 17, 25, 49, 34]	49 > 34	Yes	[9, 17, 25, 34, 49]

- Now, after the 4th iteration, only one element remains 49, which is inherently in its correct place as it's the last and largest element in the list.
- So, at the end of the 4th iteration, the entire array is sorted in ascending order:
- arr[] = {09, 17, 25, 34, 49}
- That completes the selection sort for the given array

Analysis Selection Sort

Time Complexity :

1. Best Case : Input array is already sorted.
   - Algorithm will still go through all its steps trying to find the smallest element in each iteration.
   - Time complexity : **$\Omega(n^2)$**
2. Average Case : For an average or random list
   - Algorithm will have to go through roughly half of the list for every number.
     - With a pattern : n + (n-1) + (n-2) + … + 2 + 1, ⇒ **n(n+1)/2.**
   - Time complexity : **$O(n^2)$ for large values of n.**
3. Worst Case : Input list is sorted in reverse.
   - Algorithm still goes through its entire process of finding the smallest element for each iteration.
   - Time complexity = **$O(n^2)$**

Space Complexity

- Selection Sort is an in-place sorting algorithm,
  - doesn't require any additional storage that grows with the input size.
  - It uses a constant amount of extra space for its variables (i, j, minIndex, temp).
  - Space complexity is **O(1)**

Courtesy : shiksha.com

Insertion Sort Algorithm

```
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i -1
    A[i + 1] = key
```

Insertion Sort Example

Step 0: Consider the Initial Array, arr[] = {7, 4, 5, 2}

Assumption :

- First element is sorted. (We have a sorted array of size = 1.
- Array Indexing in this example starts @ 1

Step 1: 1st Iteration

i	j	key = arr[j]	arr[]	i > 0 and (arr[i] > key)	Shift Required ?	Remark
1	2	4	[7, 4, 5, 2]	i > 0 and 7 > 4	Yes	Shift [7, 7, 5, 2] & Search location for key
0	2	4	[7, 7, 5, 2]	i = 0	No	arr[i+1] = key ==> [4, 7, 5, 2]

arr[] = {4, 7, 5, 2} and We got a sorted array of size 2

Insertion Sort Example

Step 2: 2nd Iteration

i	j	key = arr[j]	arr[]	i > 0 and (arr[i] > key)	Shift Required ?	Remark
2	3	5	[4, 7, 5, 2]	i > 0 and 7 > 5	Yes	Shift [4, 7, 7, 2] & Search location for key
1	3	5	[4, 7, 7, 2]	i > 0 and 4 < 5	No	arr[i+1] = key ==> [4, 5, 7, 2]

arr[] = {4, 5, 7, 2} and We got a sorted array of size 3

Insertion Sort Example

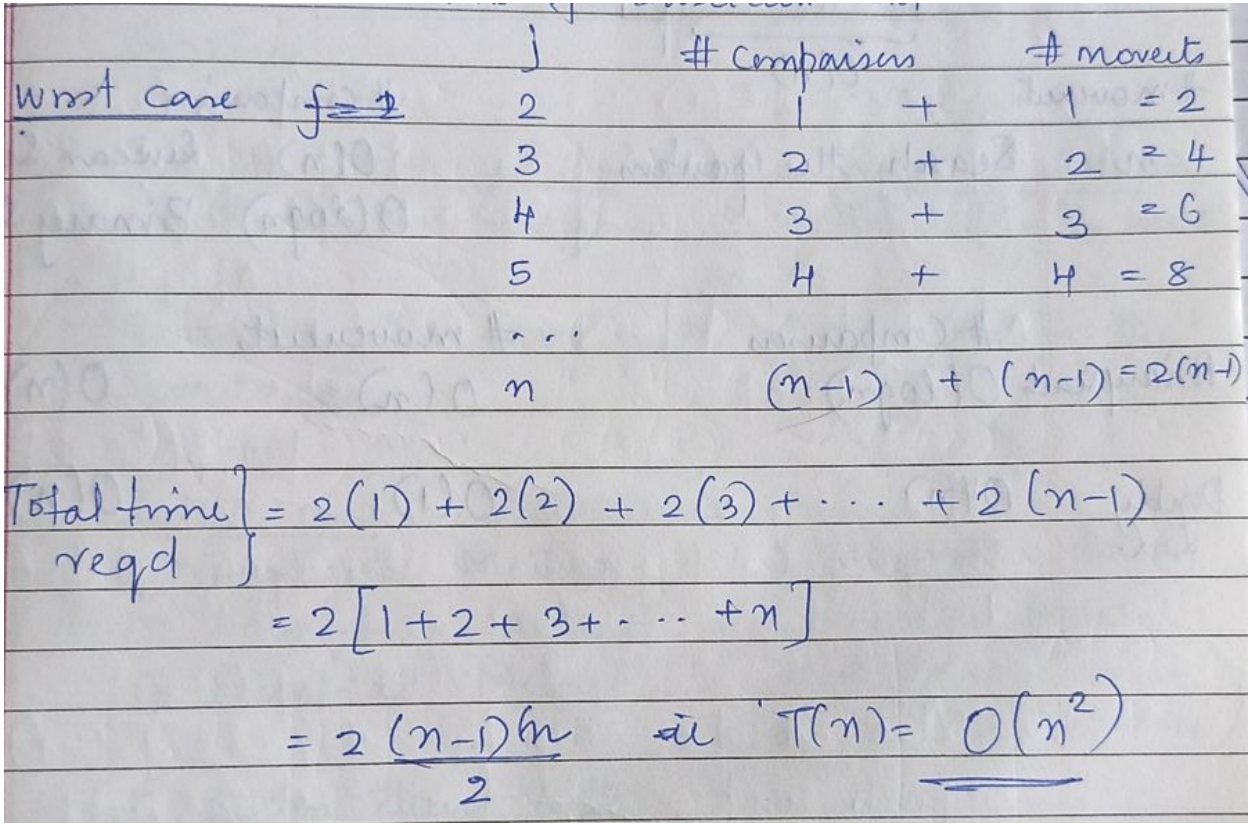
Step 3: 3rd Iteration

i	j	key = arr[j]	arr[]	i > 0 and (arr[i] > key)	Shift Required ?	Remark
3	4	2	[4, 5, 7, 2]	i > 0 and 7 > 2	Yes	Shift [4, 5, 7, 7] & Search location for key
2	4	2	[4, 5, 7, 7]	i > 0 and 5 > 2	Yes	Shift [4, 5, 5, 7] & Search location for key
1	4	2	[4, 5, 5, 7]	i > 0 and 4 > 2	Yes	Shift [4, 4, 5, 7] & Search location for key
0	4	2	[4, 4, 5, 7]	i = 0	No	arr[i+1] = key ==> [2, 4, 5, 7]

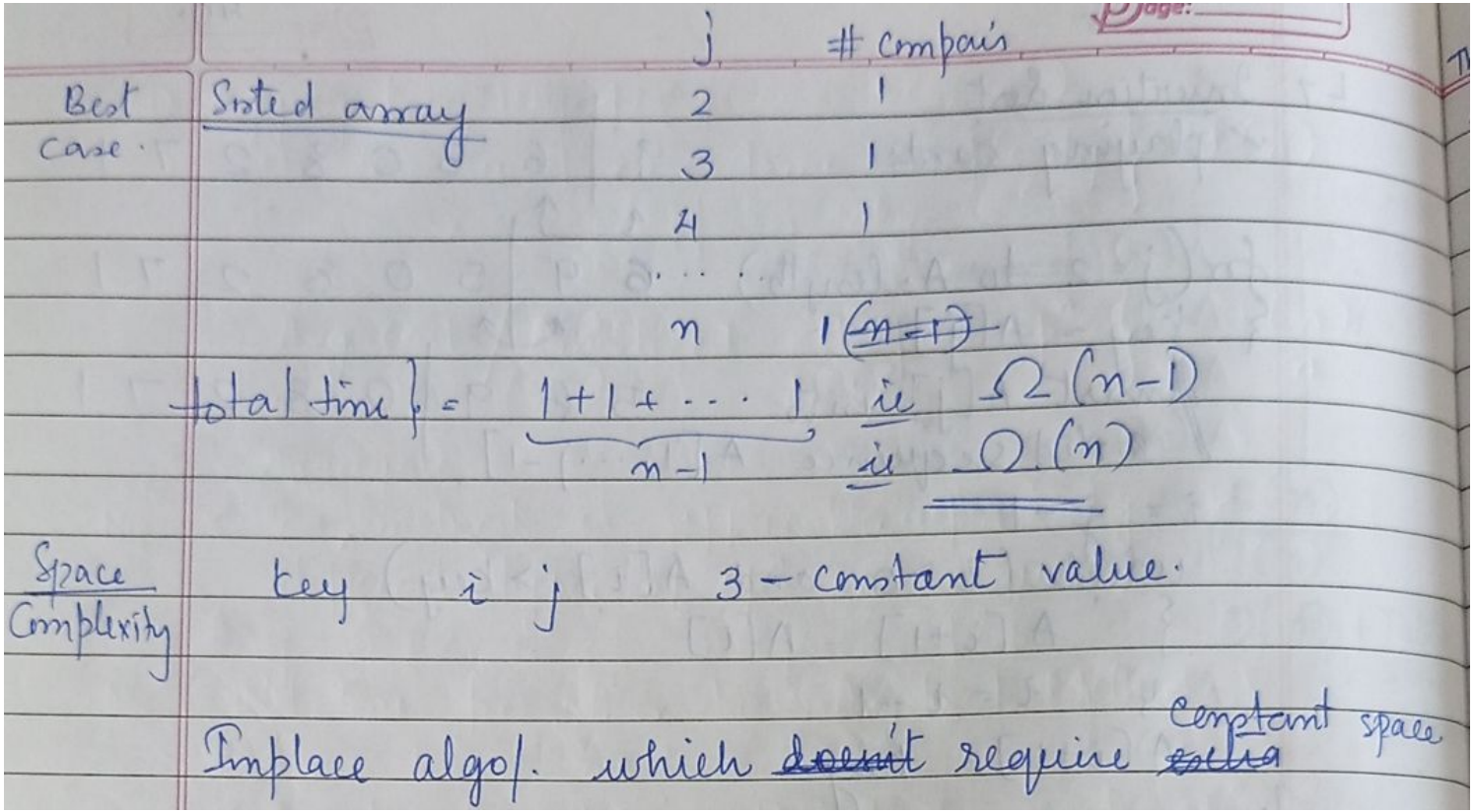
arr[] = {2, 4, 5, 7} and We got a sorted array of size 4

That completes the insertion sort for the given array

Analysis Insertion Sort



Analysis Insertion Sort



Best, Worst & Average Case Analysis - Linear Search

Linear Search

→ Sequentially search key elt frm the beginning of Array

→ search may be succesful / not.

Best case ✓ elt @ index first
✓ constant $O(1)$ $B(n) = 1$

Worst case ✓ elt @ last index $W(n) = n$

Avg Case ✓ $\frac{\text{all possible case time}}{\#\text{ cases}}$ $A(n) = \frac{n+1}{2}$

$$\frac{1+2+3+\ldots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

Best, Worst & Average Case Analysis - Linear Search

NB Case are no where related to ~~fns~~. notations

* Asymptotic notations are related to fns

* Case are the type of analysis done on the algo/.

$B(n) = O(1)$	$W(n) = O(n)$	$A(n) = O(n)$
$B(n) = \Omega(1)$	$W(n) = \Omega(n)$	$A(n) = \Omega(n)$
$B(n) = \Theta(1)$	$W(n) = \Theta(n)$	$A(n) = \Theta(n)$

NB Here we can write Best, Worst & Avg in terms ~~case~~ of all 3 Asymptotic Notation's. So, don't mix up.

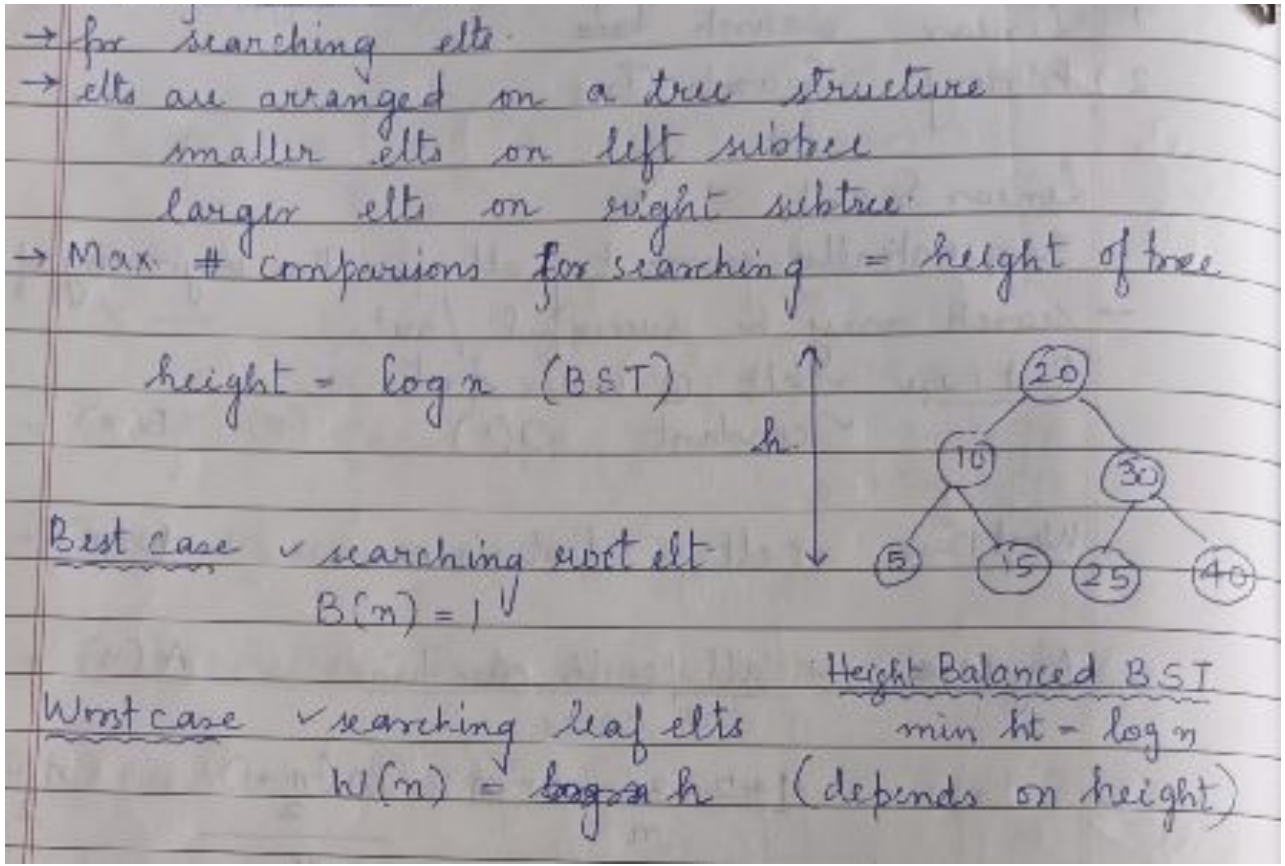
Big O – Worst case

Big Ω – Best case

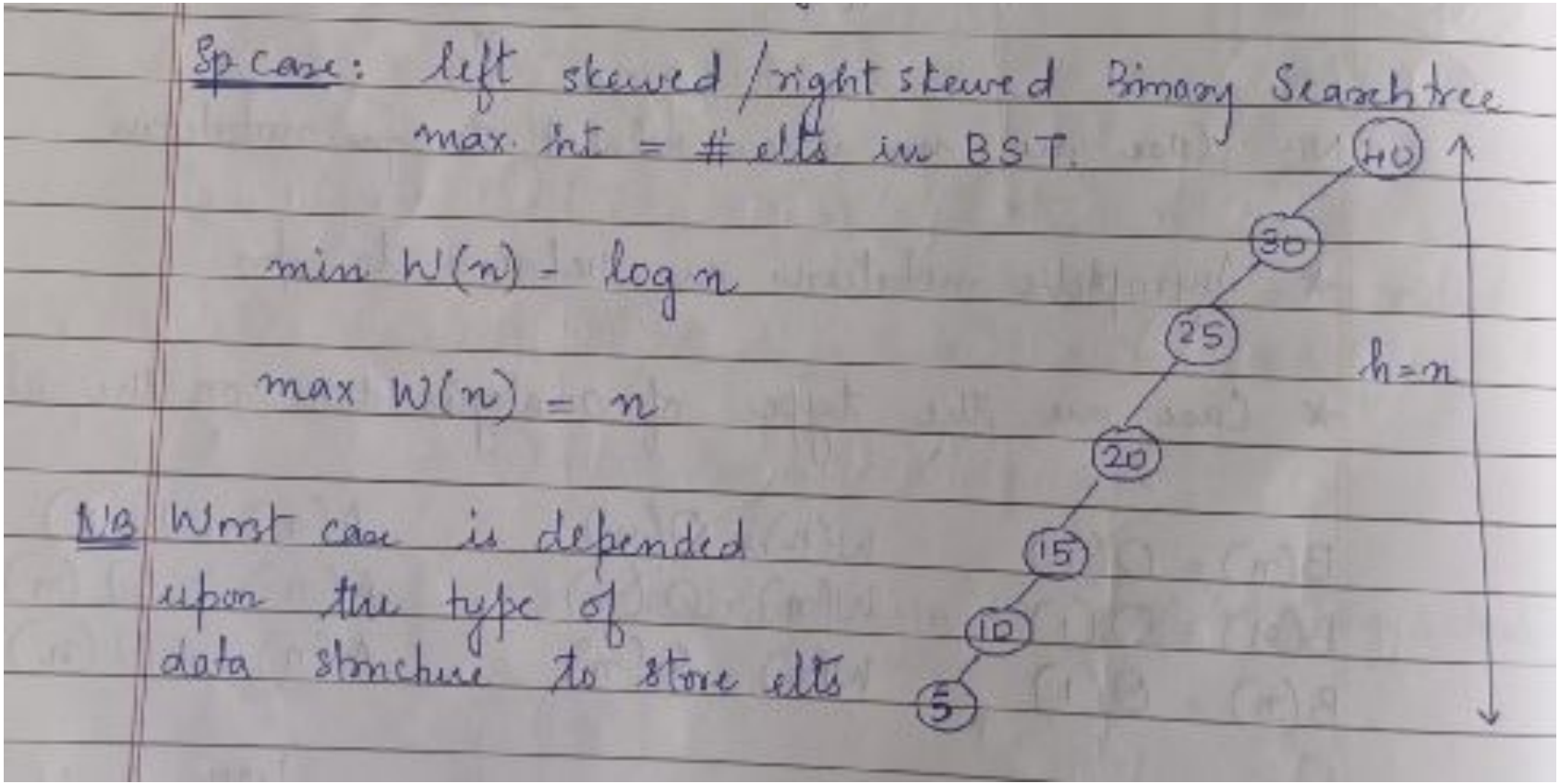
Θ – Avg Case.

Don't mix up cases

Best, Worst & Average Case Analysis - Binary Search



Best, Worst & Average Case Analysis - Binary Search



Topics to be covered

- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- **Recurrences:**
  - **The substitution method,**
  - **Recursion tree method,**
  - Master method
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Recurrence Relations / Recursive Algorithms : Example - 1

```
Test (n)                                    ——— T(n)
{ if (n > 0)                                ——— 1
  { printf (%d, n);                         ——— (1)
    Test (n-1);                             ——— T(n-1)
  }
}
```

Test (3) ——— 1

Test (2) ——— 1

Test (1) —

Test (0)

Main task: printing

times fn is called : 4 times

(3 + 1) calls

~~if n if~~. Work done = # calls

$T(n) = f(n) = (n+1)$ calls $\Rightarrow O(n)$

Recurrence Relations / Recursive Algorithms : Example - 1

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

$$T(n) = T(n-1) + 1 \qquad \because T(n-1) = T(n-2) + 1$$

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

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

$$\cdots \qquad \text{(k times)}$$

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

Recurrence Relations / Recursive Algorithms : Example - 1

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

Assume $(n-k) = 0 \Rightarrow n = k$

$$T(n) = T(n-n) + n$$
$$= T(0) + n$$
$$T(n) = 1 + n$$

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

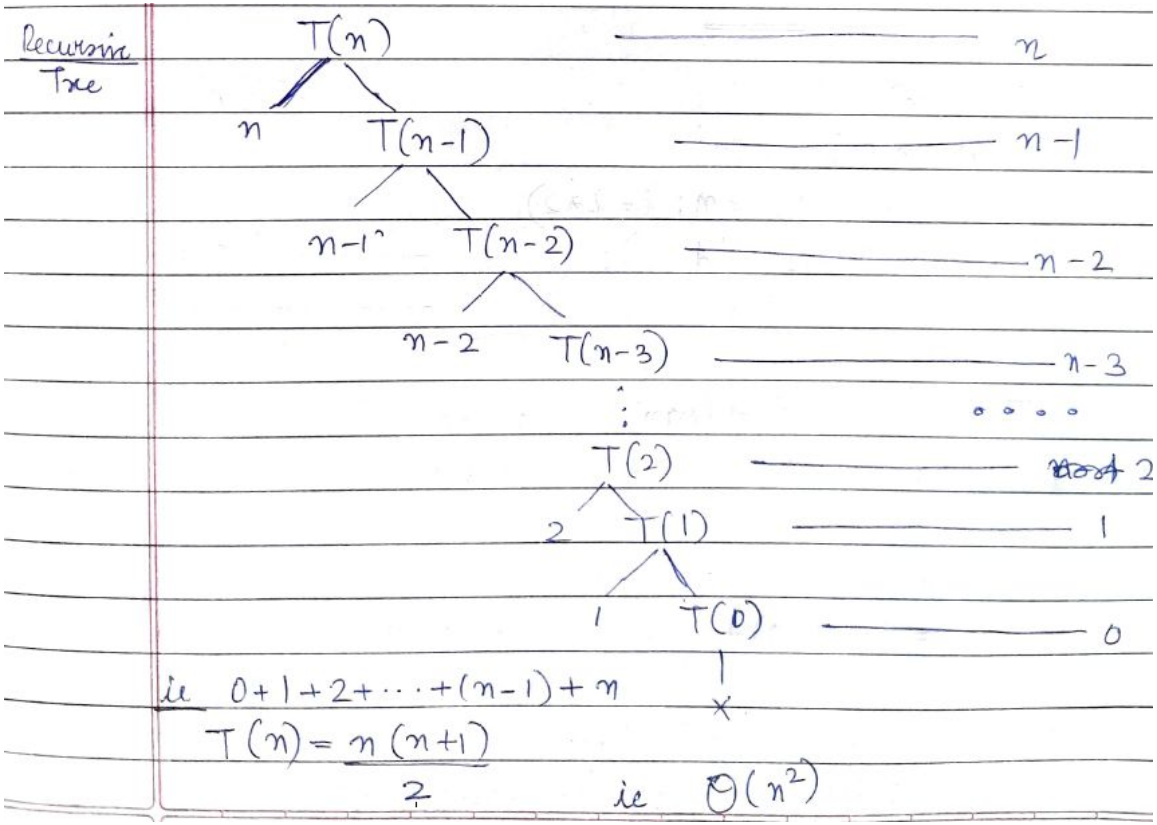
Recurrence Relations / Recursive Algorithms : Example - 2

```
Test (n)                                    T(n)
    if (n>0)                                 1
        for (i=0; i<n; i++)                 n+1
            printf ("%d", n)                n
        Test (n-1)                          T(n-1)
```

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

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

Recurrence Relations / Recursive Algorithms : Example - 2



Recurrence Relations / Recursive Algorithms : Example - 2

$T(n) = T(n-1) + n$ $\quad \because T(n-1) = T(n-2) + n-1$

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

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

... continue for k times

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

Recurrence Relations / Recursive Algorithms : Example - 2

Assume $n - k = 0$

$n = k$

$$T(n) = T(0) + (n - n + 1) + (n - n + 2) + \cdots + (n-1) + n$$

$$= T(0) + 1 + 2 + 3 + \cdots (n-1) + n$$

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

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

Recurrence Relations / Recursive Algorithms : Example - 3

```
Test (n)                                        T(n)
  if (n>0)
    for (i=1; i<n; i=i*2)
      printf ("%d", i);                         ------ log n
    Test (n-1);                                 ------ T(n-1)
```

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

Recurrence Relations / Recursive Algorithms : Example - 3

Substitution Method

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

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

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

... k times

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

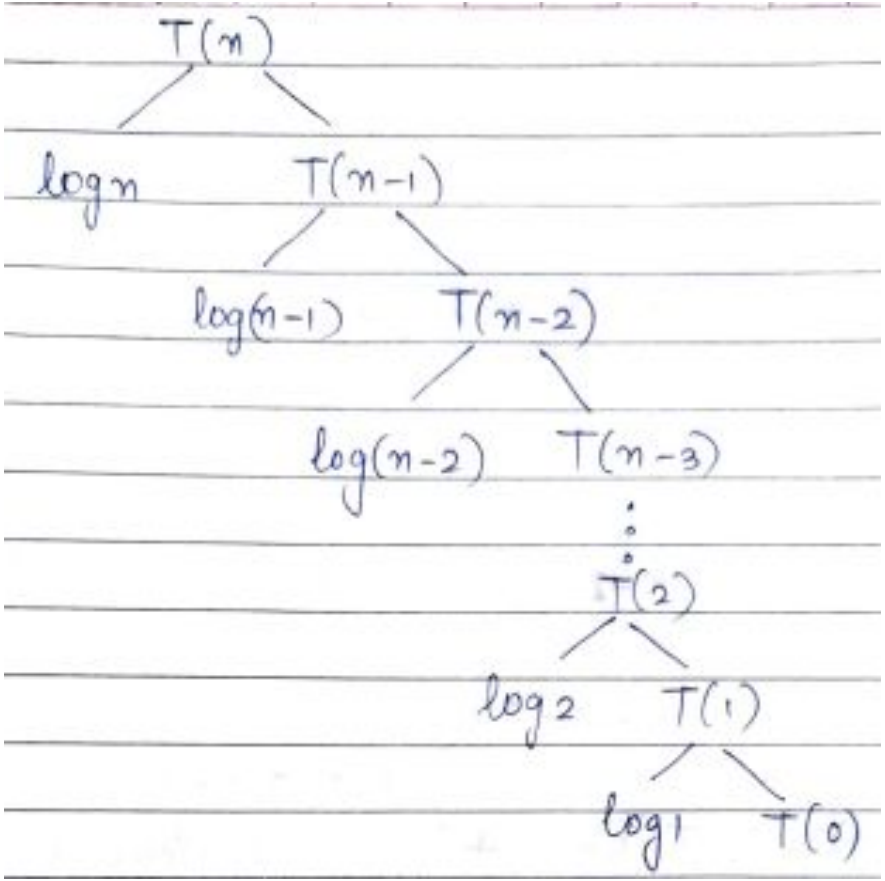
$\because n - k = 0 \Rightarrow T(n) = T(0) + \log n!$

$n = k.$

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

Recurrence Relations / Recursive Algorithms : Example - 3

Recursive Tree



Recurrence Relations / Recursive Algorithms : Example - 3

Recursive Tree

For $\log n + \log(n-1) + \log(n-2) + \cdots + \log 2 + \log 1$

i.e. $\log [n(n-1)(n-2)\{\cdots \times 2 \times 1]$

$\log n!$

NB There is no tight bound for $\log n!$

$\exists$ upper bound i.e. $O(n \log n)$

for $n! = O(n^n)$

$\log n! = O(\log n^n)$ i.e. $O(n \log n)$

Recurrence Relations - Summary

$T(n) = T(n-1) + 1$	$O(n)$
$T(n) = T(n-1) + n$	$O(n^2)$
$T(n) = T(n-1) + \log n$	$O(n \log n)$
$T(n) = T(n-1) + n^2$	$O(n^3)$
$T(n) = T(n-2) + 1 \Rightarrow \frac{n}{2}$	$O(n)$
$T(n) = T(n-100) + n$	$O(n^2)$

Recurrence Relations / Recursive Algorithms : Example - 4

```
Test (n)                                  ---------- T(n)
    if (n > 0)
        print (%d, n);                    ---------- 1
        Test (n-1);                       ---------- T(n-1)
        Test (n-1);                       ---------- T(n-1)
                                          T(n) = 2T(n-1) + 1
```

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

Recurrence Relations / Recursive Algorithms : Example - 4

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

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

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

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

$$= 2^3 T(n-3) + 4 + 2 + 1 \quad \text{——— (3)}$$

... k times

Recurrence Relations / Recursive Algorithms : Example - 4

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

Assume $(n-k) = 0$

$n = k.$

$T(n) = 2^n T(0) + \underbrace{1 + 2 + 2^2 + \cdots + 2^{k-1}}_{2^k - 1}$

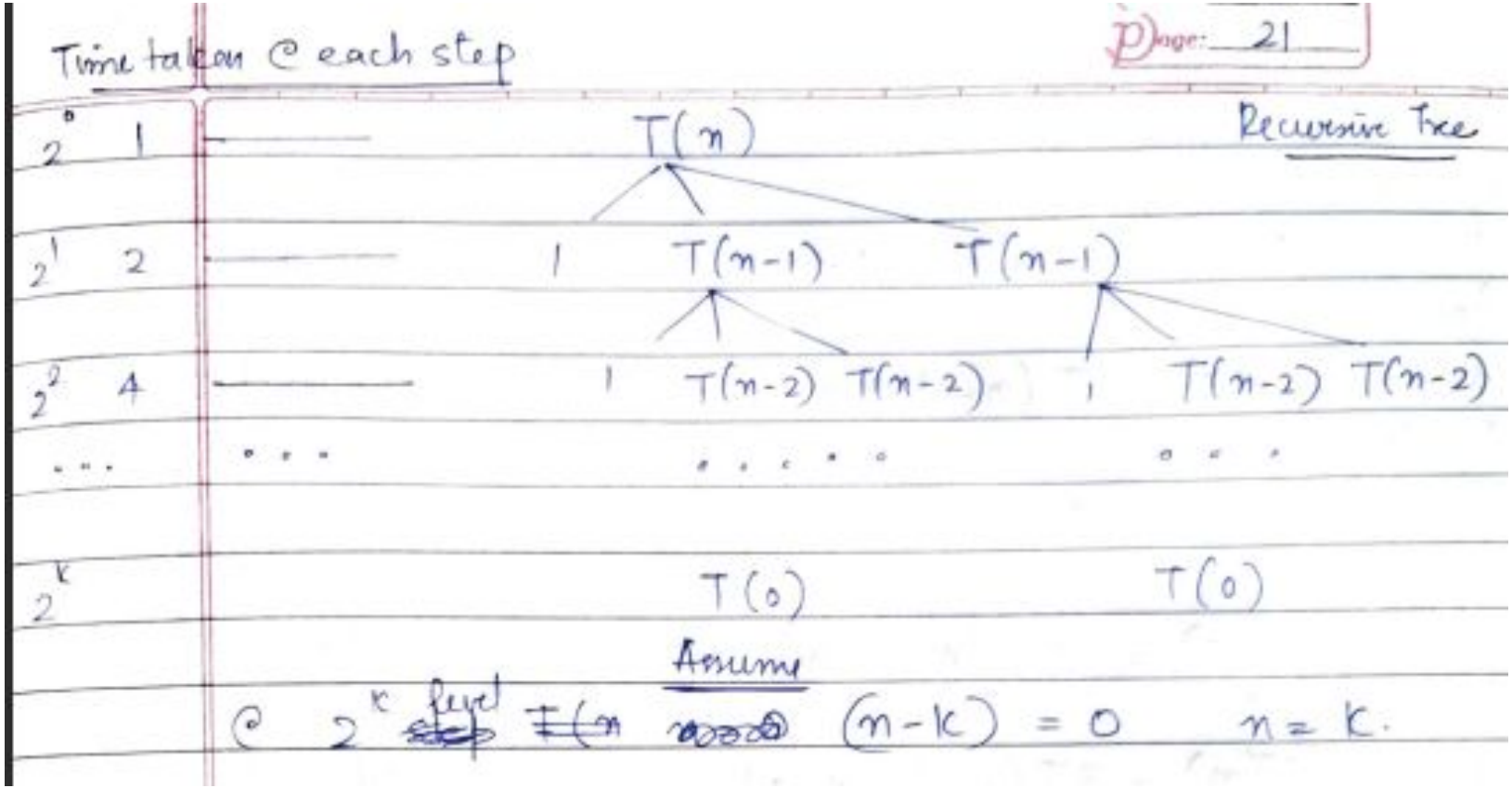
$= 2^n + 2^k - 1$

$= 2^n + 2^n - 1$

$T(n) = 2^{n+1} - 1$ i.e. $T(n) = O(2^n)$

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

Recurrence Relations / Recursive Algorithms : Example - 4



Recurrence Relations / Recursive Algorithms : Example - 4

NB $1^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

Geometric Series $a + ar + ar^2 + \cdots + ar^k = \frac{a(r^{k+1} - 1)}{(r-1)}$

Here. $a = 1 \quad r = 2 \quad = \frac{1(2^{k+1} - 1)}{2-1}$

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

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

Topics to be covered

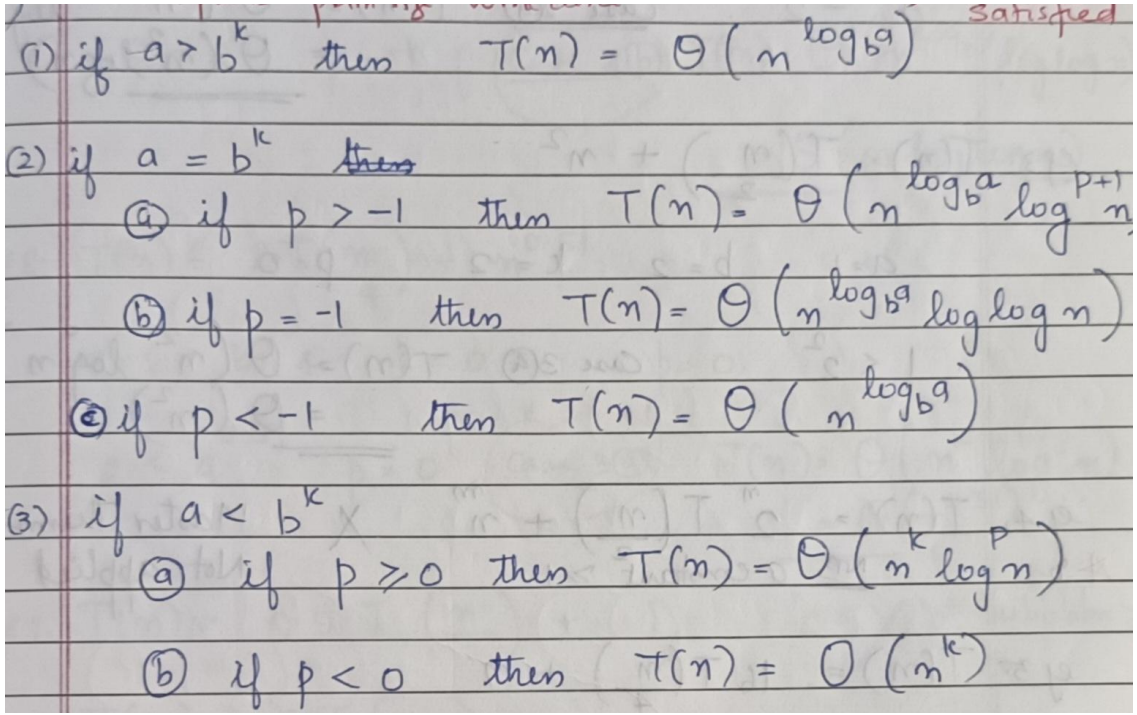
- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- **Recurrences:**
  - The substitution method,
  - Recursion tree method,
  - **Master method**
- Complexity Classes: P, NP, NP-Hard, NP-Complete

Master's theorem

Recursive equation should be in form below

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

Where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real no.



Master's theorem : Example - 1

$T(n) = 3T\left(\frac{n}{2}\right) + n^2$

$a = 3 \quad b = 2 \quad k = 2 \quad p = 0$

$a \quad b^k$

$3 < 2^2$

Case 3 (a) $T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$

Master's theorem : Example - 2

$T(n) = 4T\left(\frac{n}{2}\right) + n^2$

$a = 4 \quad b = 2 \quad k = 2 \quad p = 0$

$4 = 2^2$ Case 2(a) $T(n) = \Theta\left(n^{\log_2 4} \log^0 n\right)$

$= \Theta(n^2 \log n)$

Master's theorem : Example - 3

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

$a = 1 \quad b = 2 \quad k = 2 \quad p = 0$

$1 < 2^2$ Case 3(a) $T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$

Master's theorem : Example - 4 , 5

eg 4. $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$ ✗ Master Theorm Not applied

* NB a-constant ≥ 1

eg 5. $T(n) = 16\, T\left(\frac{n}{4}\right) + n$

$a = 16 \quad b = 4 \quad n = 1 \quad p = 0$

$16 > 4^1$ Case 1 $T(n) = \Theta(n^{\log_4 16})$

$= \Theta(n^2)$

Master's theorem : Example - 6

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

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$

$2 = 2^1$ Case 2(a) $T(n) = \Theta\left(n^{\log_b a} \log^{p+1} n\right)$

$= \Theta(n \log^2 n)$

Master's theorem : Example - 7

$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n} = 2T\left(\frac{n}{2}\right) + n \log^{-1} n$

$a = 2 \quad b = 2 \quad k = 1 \quad p = -1$

$2 = 2^1 \quad p = -1$ Case 2(b) $T(n) = \Theta\left(n^{\log_b a} \log\log n\right)$

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

Master's theorem : Example - 8

$T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

$a = 2 \quad b = 4 \quad k = 0.51 \quad p = 0$

$2 < 4^{0.51} \quad p = 0$ Case 3(a) $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^{0.51})$

Master's theorem : Example - 9, 10

$T(n) = 0.5\,T\left(\frac{n}{2}\right) + \frac{1}{n}$ ✗ # subproblems ≥ 1

$T(n) = 6\,T\left(\frac{n}{3}\right) + n^2 \log n$

$a = 6 \quad b = 3 \quad k = 2 \quad p = 1$

$6 < 3^2 \quad p = 1$ Case 3(a) $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log n)$

Master's theorem : Example - 11, 12

eg 11 $T(n) = 64\,T\left(\frac{n}{8}\right) - n^2 \log n$ ✗

work done < 0

eg 12 $T(n) = 7\,T\left(\frac{n}{3}\right) + n^2$

$a = 7 \quad b = 3 \quad k = 2 \quad p = 0$

$7 < 3^2 \quad p = 0$ Case 3(a)

$T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$

Master's theorem : Example - 13

$T(n) = 4T\left(\frac{n}{2}\right) + \log n$

$a = 4 \quad b = 2 \quad k = 0 \quad p = 1$

$4 > 2^0$ Case 1 $T(n) = \Theta(n^{\log_b a})$

$= \Theta(n^2)$

Master's theorem : Example - 14

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

$a = \sqrt{2} \quad b = 2 \quad k = 0 \quad p = 1$

$\sqrt{2} > 2^0$ (Case 1) $T(n) = \Theta\left(n^{\log_b a}\right)$

$= \Theta\left(n^{\log_2 \sqrt{2}}\right)$

$= \Theta(\sqrt{n})$

Master's theorem : Example - 15

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

$a = 2 \quad b = 2 \quad k = \frac{1}{2} \quad p = 0$

$2 > 2^{1/2}$ Case 1 $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

Master's theorem : Example - 16

$T(n) = 3T\left(\frac{n}{2}\right) + n$

$a = 3 \quad b = 2 \quad k = 1 \quad p = 0$

$3 > 2^1$ Case 1 $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 3}\right)$

Master's theorem : Example - 17

$T(n) = 3T\left(\frac{n}{2}\right) + \sqrt{n}$

$a = 3 \quad b = 2 \quad k = \frac{1}{2} \quad p = 0$

$3 > 3^{1/2}$ case 3 $T(n) = \Theta\left(1 n^{\log_3 3}\right) = \Theta(n)$

Master's theorem : Example - 18

$T(n) = 4T\left(\frac{n}{2}\right) + cn$

$a = 4 \quad b = 2 \quad k = 1 \quad p = 0$

$4 > 2^1$ Case 1 $T(n) = \theta(n^{\log_b a}) = \theta(n^2)$

Master's theorem : Example - 19

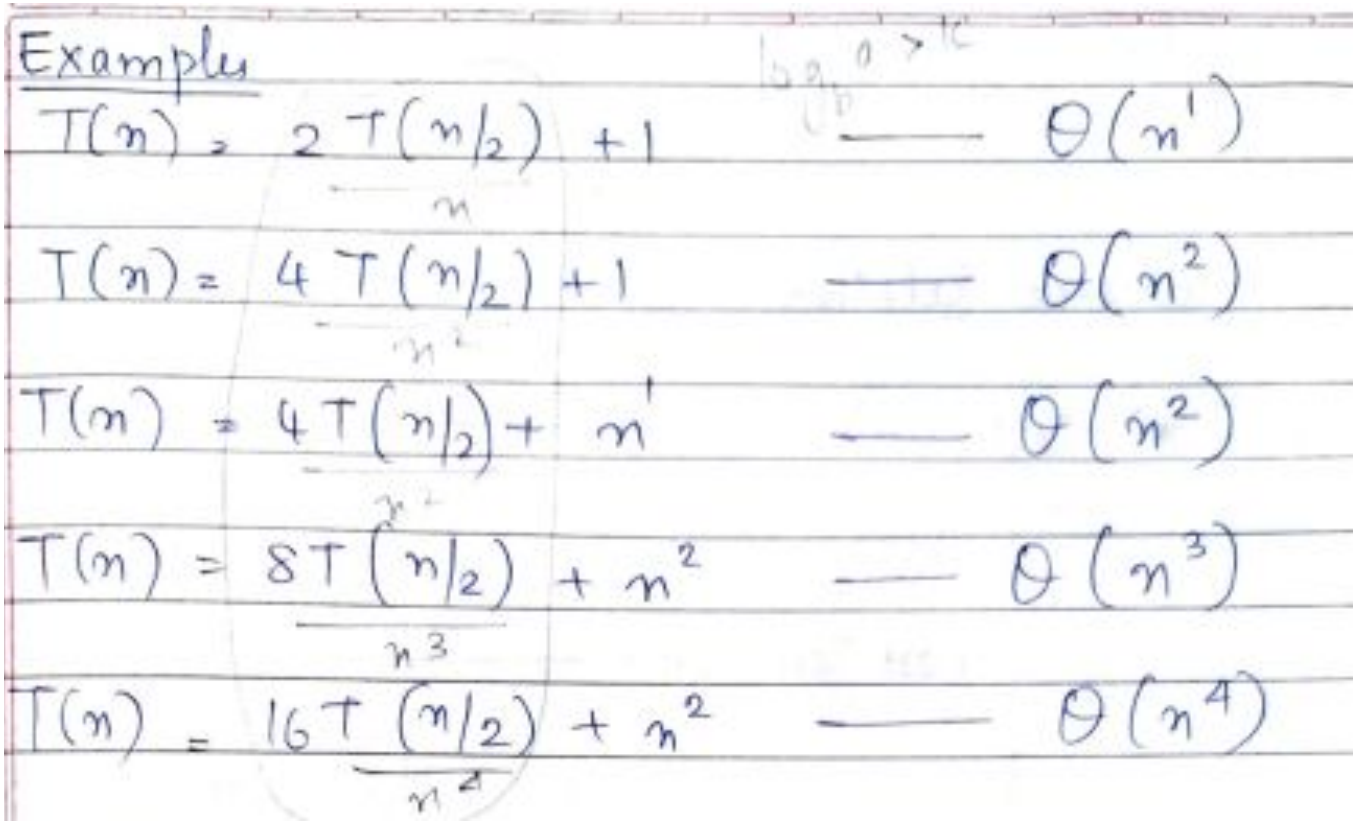
$T(n) = 3T\left(\frac{n}{4}\right) + n \log n.$

$a = 3 \quad b = 4 \quad k = 1 \quad p = 1$

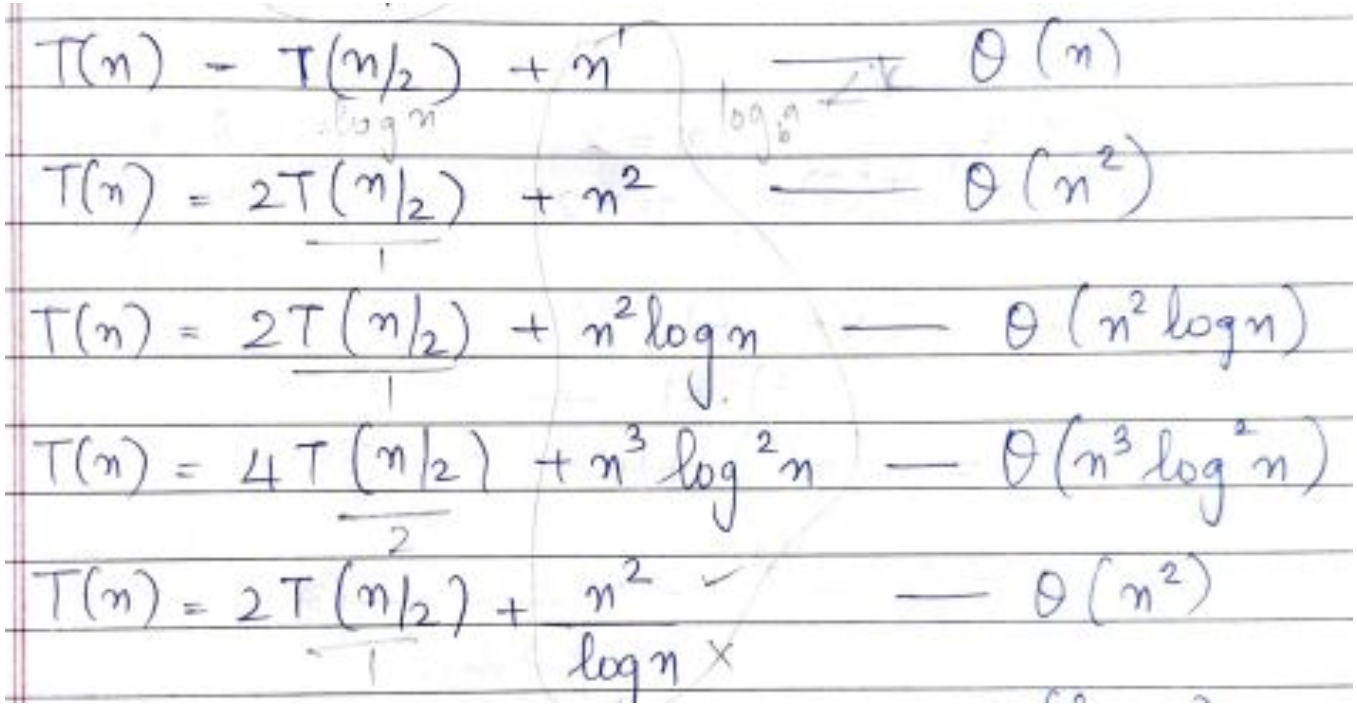
$3 < 4^1 \quad p \geq 0$ case 3(a) $T(n) = \Theta(n^k \log^p n)$

$= \Theta(n \log n)$

Master's theorem : Examples



Master's theorem : Examples



Master's theorem : Examples

$T(n) = T(n/2) + 1$ — $\log_b a = k$ — $\Theta(\log n)$

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

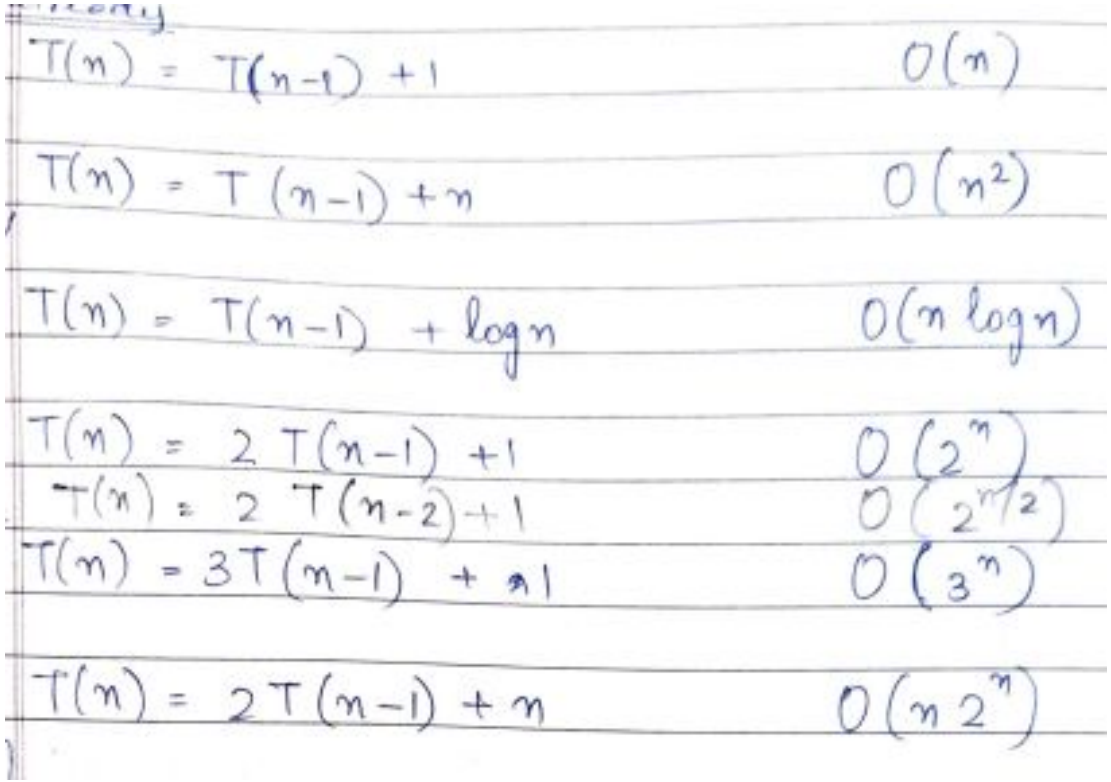
$T(n) = 2T(n/2) + n \log n$ — $\Theta(n \log^2 n)$

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

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

$T(n) = 2T(n/2) + \frac{n}{\log n}$ — $\Theta(n \log \log n)$

Master's theorem for Decreasing Functions



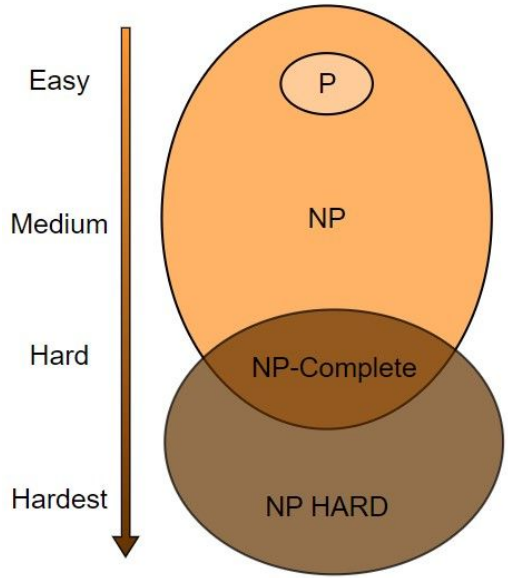
Topics to be covered

- Introduction to Algorithm Analysis
- Performance analysis - Space and time complexity
- Growth of function - Big-Oh, Omega, Theta notation
- Mathematical background for algorithm analysis;
- Analysis of selection sort, insertion sort.
- Recurrences:
  - The substitution method,
  - Recursion tree method,
  - Master method
- **Complexity Classes: P, NP, NP-Hard, NP-Complete**

Complexity Classes

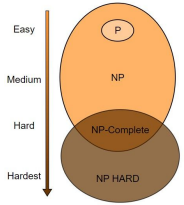
- Problems are divided into classes ⇒ **Complexity Classes**.
  - a set of problems with related complexity.
  - based on how much time and space they require to solve problems and verify the solutions.
  - useful in organising similar types of problems.

- Types of Complexity Classes
  1. **P Class**
  2. **NP Class**
  3. **NP-hard**
  4. **NP-complete**



Courtesy : compgeek.co.in

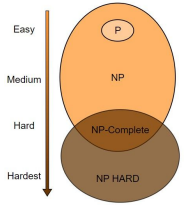
Complexity Classes



1. **Class P (Polynomial Time)**

- Class of decision problems that can be **solved by a deterministic Turing machine in polynomial time**.
- Characteristics:
  - Problems in P are considered **efficiently solvable.**
  - Algorithms exist with worst-case time complexity $O(n^k)$ for some constant k.
- Examples:
  - **Sorting**: Merge Sort, Quick Sort (O(n log n))
  - **Graph Algorithms**: Dijkstra's shortest path ($O(V^2)$ using adjacency matrix)

Complexity Classes

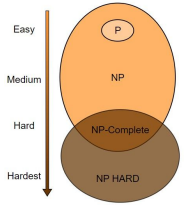


Examples of Polynomial Time Algorithms

Algorithm	Worst Case
Bubble Sort	$O(n^2)$
Merge Sort	$O(n \log n)$
Insertion Sort	$O(n^2)$
Quick Sort	$O(n^2)$
Counting Sort	$O(n + k)$
Dijkstra's Algorithm	$O(n^2)$
Binary Search	$O(\log n)$
Prim's Algorithm	$O(n^2)$
Kruskal's Algorithm	$O(E \log E)$
Breadth-First Search (BFS)	$O(V + E)$
Depth-First Search (DFS)	$O(V + E)$
Matrix Multiplication	$O(n^3)$
Strassen Matrix Multiplication	$O(n^{2.81})$
Floyd-Warshall Algorithm	$O(n^3)$
Topological Sorting	$O(V + E)$
Knapsack Problem by Greedy	$O(n \log n)$

Courtesy : compgeek.co.in

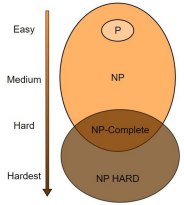
Complexity Classes



2. Class NP (Nondeterministic Polynomial Time)

- Class of decision problems for which a **given solution can be verified in polynomial time** by a deterministic Turing machine.
- Characteristics:
  - **If a solution is given,** we can verify its correctness in polynomial time.
  - It is unknown whether **P = NP**, meaning whether problems that can be verified quickly can also be solved quickly.
- Examples:
  - **Traveling Salesman Problem (TSP) (Decision Version)**:
    Given a graph and a target distance d, is there a Hamiltonian cycle with a total distance ≤ d?
  - **Boolean Satisfiability Problem (SAT):**
    Given a Boolean formula, does there exist an assignment of variables that makes the formula true?

Complexity Classes

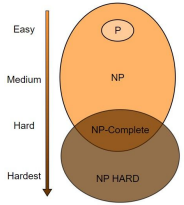


Examples of Non-Polynomial Time Algorithms

Algorithm	Worst Case
Traveling Salesman Problem (TSP)	O(n!)
Hamiltonian Path Problem	O(n!)
Subset Sum Problem by backtracking	$O(2^n)$
Boolean Satisfiability Problem (SAT)	$O(2^n)$
Graph Colouring Problem	$O(m^V)$ or $O(4^n)$
Sudoku Solver (Brute Force)	$O(2^n)$
Clique Problem	$O(2^n)$
Partition Problem	$O(2^n)$
0/1 Knapsack Problem	O(nW)

Courtesy : compgeek.co.in

Complexity Classes



3. Class NP-Hard (Nondeterministic Polynomial-Hard)

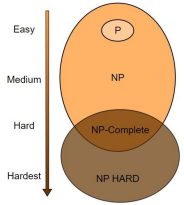
- At least as hard as the hardest problems in NP.
- They do not have to be decision problems (i.e., they can be **optimization problems**).
- Characteristics:
  - **A problem X is NP-Hard if every problem in NP can be reduced to X in polynomial time.**
  - NP-Hard problems may not be in NP (i.e., their solutions might not be verifiable in polynomial time).
- Examples:
  - **Traveling Salesman Problem (Optimization Version):**

    Find the shortest possible route visiting all cities and returning to the starting point.
  - **Knapsack Problem (Optimization Version):**

    Find the most valuable subset of items that fit into a knapsack of limited capacity.
  - **Graph Coloring Problem:**

    Assign the fewest colors to a graph such that no two adjacent vertices share the same color.

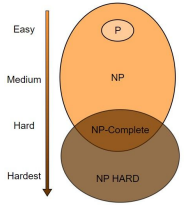
Complexity Classes



Example

1. If you're trying to solve a **traveling salesman problem** and you input 1000 cities, expecting to get a solution, it won't happen. You've essentially set the program to run indefinitely, and it will take about 2^{40} years to solve if you have a super computer.
2. **Scheduling a Family Gathering** is an NP-hard problem. When organizing a gathering for many family members, each with their own availability, it becomes increasingly difficult to find a time that works for everyone. While it's easy to accommodate a few people, as the number of participants grows, the combinations of schedules become exponentially complex, making it challenging to find the best solution.

Complexity Classes



4. Class NP-Complete

A problem A is NP-Complete if it satisfies <u>two properties</u>

- For every problem **B ϵ NP**, problem **B** can be reduced to problem **A** in polynomial time (**A is NP-Hard**).
- Problem **A** is in **NP** means a non-deterministic algorithm can be written for a problem **A**.

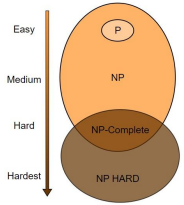
- <u>Characteristics:</u>
  - **If any NP-Complete problem can be solved in polynomial time, then P = NP.**
  - **NP-Complete problems are both intractable** and widely studied.
- <u>Examples</u>:
  - **Boolean Satisfiability Problem (SAT)**

    First NP-Complete problem, proved by Cook's Theorem.
  - **Hamiltonian Cycle Problem**:

    Given a graph, does there exist a cycle that visits every vertex exactly once?
  - **Clique Problem:**

    Given a graph and an integer k, is there a clique (fully connected subgraph) of size k?

Complexity Classes



Relationships Between These Classes

- **P ⊆ NP**:

  Every problem that can be solved in polynomial time can also be verified in polynomial time.

- **NP-Complete ⊆ NP** and **NP-Complete ⊆ NP-Hard**:

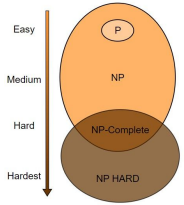
  Every NP-Complete problem is NP-Hard but not necessarily vice versa.

- **If any NP-Complete problem is solved in polynomial time, then P = NP**.

- **P ⊆ NP ⊆ NP-Hard**

- **NP-Complete = NP ∩ NP-Hard**

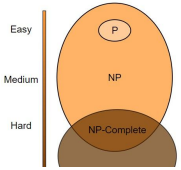
Complexity Classes



Summary

- P problems are efficiently solvable.
- NP problems are efficiently verifiable but not necessarily solvable in polynomial time.
- NP-Hard problems are at least as hard as the hardest NP problems but may not be decision problems.
- NP-Complete problems are the hardest problems in NP and are both NP and NP-Hard.

Reduction : Comparing Complexity Classes



- **The process of transforming one problem into another in such a way that a solution to the second problem can be used to solve the first.**
- If a problem A can be **reduced** to problem B, it means that solving problem B would allow us to solve problem A as well.

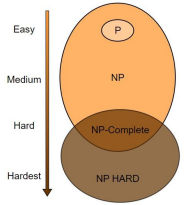
Polynomial Time Reductions

- There exists a **polynomial-time computable function f** that maps any instance of problem **A** to an instance of problem **B**.
- **$x \in A \Leftrightarrow f(x) \in B$**

  This means that if **x** is a "**yes**" instance of decision problem **A**, then **f(x)** is a "**yes**" instance of decision problem **B**, and if **x** is a "**no**" instance of **A**, then f(x) is a "**no**" instance of **B**.

Courtesy : compgeek.co.in

Reduction : Comparing Complexity Classes



Boolean Satisfiability Problem (SAT) ∝ Hamiltonian Path Problem

- Boolean Satisfiability Problem (SAT) is NP-Hard
- SAT is reducing to Hamiltonian Path Problem in polynomial time.
- So, Hamiltonian Path Problem is NP-Hard.

Similarly :

SAT ∝ Subset Sum Problem

SAT ∝ Travelling Salesman Problem

SAT ∝ Graph Colouring Problem

SAT ∝ Sudoku Solver

SAT ∝ Clique Problem

SAT ∝ Partition Problem

SAT ∝ 0/1 Knapsack Problem

Courtesy : compgeek.co.in