Importance of Algorithm Analysis

Analysis of Algorithm

⇒ The algorithm taking least time & memory space is preferred

⇒ Two ways to analyze an algorithm:

1. Priori Analysis
2. Posteriori Analysis

Priori Vs. Posteriori Analysis

Priori Analysis	Posteriori Analysis
1. Estimation of time and memory space is required by an alg. (before) executing it on the system	1. Calculation of time and memory space required by an alg. (after) exculing it on the system
2. Independent of the programming language	2. Dependent
3. Independent " " hardware.	3. Dependent

Importance of Algorithm Analysis

⇒ There are many algorithm to solve a problem.

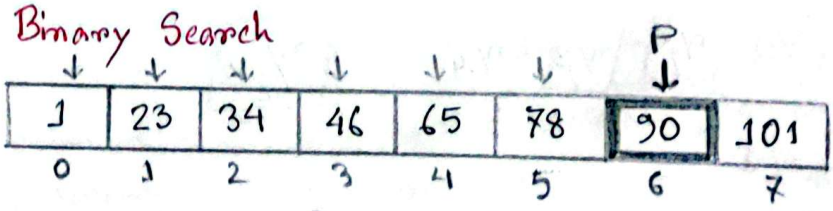
⇒ Requirement : to find the most efficient algorithm to solve a prob.

Efficiency (in terms of): time & memory space consumption.

Example finding the Key

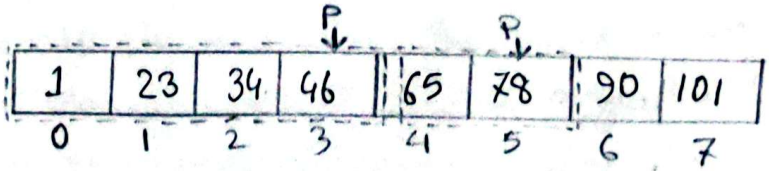
Linear Search Vs. Binary Search

Linear Search :



Key: 90
Total comparisons: 7

Binary Search :
↳ Faster



Key: 90
Total comparisons: 3

1st step – mid index = (0+7)/2 = 3.5 ; 46<90 → Exclude Left side

2nd step – mid index = (4+7)/2 = 5.5 ; 78<90 → Exclude Left side

3rd step – mid index = (6+7)/2 = 6.5 ; Found 90

Example - finding the Key

For 100 integers → Linear Search : 100 comparisons
Binary Search : 7 comparisons

Binary Search comparisons = $\log_2(n)$
$= \log_2(100)$
$= 6.64 \approx 7$

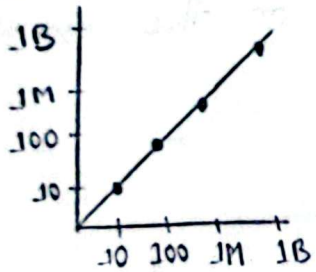
Growth Rate

↳ Rate at which running time increases as a function of input is called

The Big O notation

Linear Search

No. of comparison $\propto$ Input size ⇒ Running time : $O(n)$ ← no. of operations

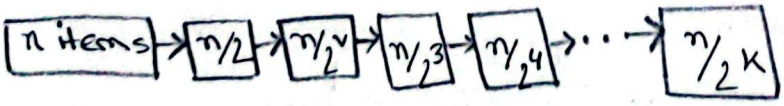


No. of Items	No. of comparisons
10	10
100	100
1 million	1 million
1 billion	1 billion

⇒ Big O notation doesn't tell speed in seconds
⇒ It helps in comparing th no. of operations.

Binary Search

Every time half of the list is eliminated Running time : $O(\log n)$



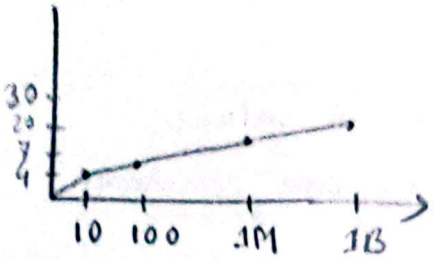
Now, $\frac{n}{2^k} = 1$

$\Rightarrow 2^k = n$

$\Rightarrow \log_2 2^k = \log_2 n$

$\Rightarrow k = \log_2 n$

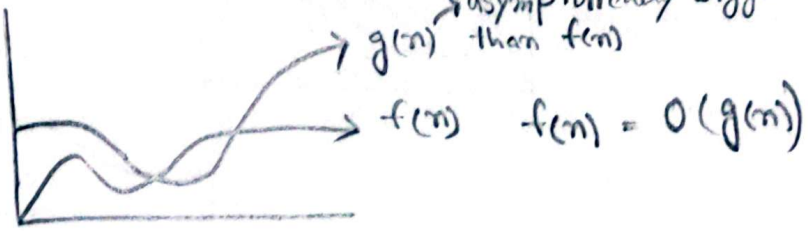
Growth Rate of Binary Search



No. of items	No. of Comparisons
10	$3.32 = 4$ (ceiling of $\log_2 n$)
100	$6.64 \approx 7$
1 M	$19.93 \approx 20$
1 B	$29.89 \approx 30$

Growth Rate Comparison → See Slide

Growth Rate Comparison of functions



Asymptotic Analysis

⇒ A mathematical way of describing how an algorithm performs as the input size (n) grows large.

⇒ It gives a mathematical tool to compare algorithms independent of hardware, compilers/programming language.

Main Asymptotic Notations

Notation	Meaning	Bounds
Big O (O)	Upper Bound	Worst-case
Big Ω (Omega)	Lower "	Best-case
Big Θ (Theta)	Tight "	Exact growth

Big O

⇒ A mathematical notation used in computer science to describe the upper bound/worst case scenario of an algorithm's runtime complexity (in terms of the input size)

⇒ It provides a standardized & concise way to express how an algorithm's performance scales as the i/p size grows.

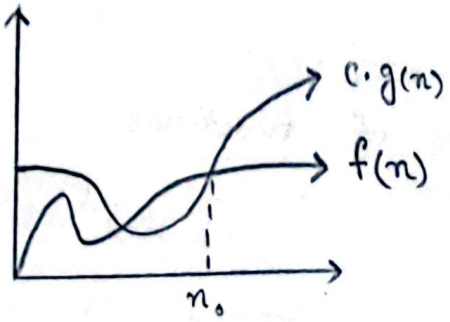
✓ Big O gives the worst case performance
✓ Defines the maximum amount of time an alg. could take.

Definition:

Assuming, $f(n)$ & $g(n)$ → non-negative functions.

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values of n where $n \geq n_0$ and c and n_0 are constants.

$g(n)$ is the tight upper bound of $f(n)$



Math

Problem 1: Assume that $f(n) = 5n + 50$ & $g(n) = n$. Is $f(n) = O(g(n))$?

Sol^n: According to the notation of Big O notation:

$f(n) = O((g(n))$ if $f(n) \leq c \cdot g(n)$ for all values of n where $n \geq n_0$ and c and n_0 are constants.

	5n+50 (Dominating Factor)	6n (c)
n = 10	100	60
n = 20	150	120
n = 30	200	180
n = 40	250	240
n = 50	300	300

$5n + 50 = O(n)$

for $c = 6$ and $n \geq 50$

tight upper bound

5n এর কাছাকাছি

6n

upper bound: ✗

tight " : 6

$c \cdot g(n)$

$\Rightarrow c \cdot n$ [$\because g(n) = n$]

$\Rightarrow 6n$

Problem 2: Assume that $f(n) = 5n + 50$ and $g(n) = ?$. Is $g(n)$ upper bound of $f(n)$?

Soln: According to the definition of Big O notation:
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values of n where $n \geq n_0$ and c and n_0 are constants.

	$5n+50$	$1000 \cdot \log_{10} n$
$n = 10$	100	1000
$n = 10^2$	550	2000
$n = 10^3$	5050	3000
$n = 10^4$	50050	4000

Here, Assuming $c = 1000$?

$g(n)$ is not upper bound of $f(n)$.

Problem 3: Find the upper bound of $f(n) = 3n + 8$.

Step 1: Identify the dominant term:

	$3n$ ← Dominant term	8
$n = 10$	30	8
$n = 20$	60	8
$n = 30$	90	8

Step 2: Choose $g(n)$ according to the dominant term

$g(n) \rightarrow n$, $n\log n$, n^2, n^3, ... [$g(n) = n$ = tightest upper bound]

Step 3: Apply the Big O definition:
According to the definition of Big O notation,
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values of n where $n \geq n_0$ and c and n_0 are constants.

Assuming $c = 4$

$3n + 8 \leq 4n$?
[$g(n)$ is the upper bound of $f(n)$?]

	$3n+8$	$4n$
$n = 7$	29	28
$n_0 \rightarrow n = 8$	32	32
$n = 9$	35	36
$n = 100$	308	400

$3n + 8 \leq$ is True

$\therefore 3n + 8 = O(n)$ for $c = 4$ and $n_0 = 8$

Prob 4: Find the upper bound of $f(n) = n^2 + 10$!

Solⁿ: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \le c \cdot g(n)$ . . . .

	n^2	10 (Dominant Term)
$n=10$	100	10
$n=20$	400	10
$n=30$	900	10

Assuming, $c = 2$

$n^2 + 10 \le 2n^2$

	n^2+10	$2n^2$
$n=1$	11	2
$n=2$	14	8
$n=3$	19	18
$n_0 \rightarrow$ $n=4$	26	32

$n^2 + 10 \le 2n^2$ is True

$\therefore n^2 + 10 = O(n^2)$ for $c = 2$ and $n_0 = 4$

Prob 5: Find the upper bound of $f(n) = 2n^3 - 2n^2$

Solⁿ: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \le c \cdot g(n)$ . . . .

	$2n^3$ (Dominant)	$2n^2$
$n=2$	16	8
$n=3$	54	18
$n=4$	128	32

Assuming, $g(n) = n^3 \rightarrow$ According to the Dominant Term

$c = 2$

	$2n^3 - 2n^2$	$2n^3$
$n_0 \rightarrow$ $n=1$	0	2
$n=2$	8	16
$n=3$	36	54

$2n^3 - 2n^2$
→ $3n^3$ নেই কি
→ $c_0 = 2$ এটা substeadia ultimately ছোট হবে. so $2n^3$ নিলেই হবে
* $3n^3 - 2n^2$ ছোট হয়ে যাবে

$2n^3 - 2n^2 \le 2n^3$ is True

$\therefore 2n^3 - 2n^2 = O(n^3)$ for $c = 2$ and $n_0 = 1$

Prob 6: Find the upper bound of $f(n) = n^4 + 100n^2 + 35$

Soln: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

Assuming, $g(n) = n^4$

$c = 2$

	$n^4 + 100n^2 + 35$	$2n^4$
$n = 1$	136	2
$n = 10$	20035	20000
$n_0 \rightarrow n = 11$	26776	29282

Dominant

	n^4	$100n^2$	35
$n=10$	10,000	10,000	35
$n=11$	14,641	12,100	35
$n=12$	20,736	14,400	35

$n^4 + 100n^2 + 35 \leq 2n^4$ is True

$\therefore n^4 + 100n^2 + 35 = O(n^4)$ for $c = 2$ and $n_0 = 11$

Prob 7: Find the upper bound of $f(n) = 2^n + 3n^3$

Soln: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

Assuming, $g(n) = 2^n$

$c = 2$

	$2^n + 3n^3$	2^{n+1}
$n = 1$	5	4
$n = 2$	28	8
$n = 12$	9280	8192
$n_0 \rightarrow n = 13$	14783	16384

Dominant

	2^n	$3n^3$
$n=12$	4096	5184
$n=13$	8192	6591
$n=14$	16384	8232

$2^n + 3n^3 \leq 2^{n+1}$ is True

$\therefore 2^n + 3n^3 = O(2^n)$ for $c = 2$ and $n_0 = 13$

Prob 8: Find the upper bound of $f(n) = 300$.

Sol": According to the Definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

Here, 300 is the Dominant term → coz only One Value left

Assuming, $g(n) = 1$, $c = 300$ (c: constant) ?

$n = 1$	300	$1 \cdot 300$
	300	300

$300 \leq 1 \cdot 300$ is True.

$\therefore 300 = O(1)$ for $c = 300$ and $n_0 = 1$

Common Big O Runtimes

$O(\log n)$	log time	Example: Binary Search
$O(n)$	linear time	Example: Linear Search
$O(n \log n)$		Example: Quicksort
$O(n^2)$	polynomial time	Example: Selection Sort
$O(n!)$	exponential time	Example: Travelling Salesman

Polynomial time

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0$

$x \rightarrow$ variable

$a_0, a_1, \ldots, a_n \rightarrow$ co-efficients

$n \rightarrow$ non-negative integers

$a_n \neq 0$

Linear Polynomial (degree 1)

$2x + 3 = 0$

Quadratic Polynomial (degree 2)

$x^2 - 5x + 6 = 0$

Visualization of Runtimes → See Slide

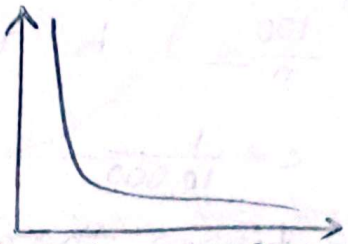
DCLPE

1. Decrement Function
2. Constant
3. Logarithm
4. Polynomial
5. Exponential

Decrement functions

Definition: A function in which the denominator is bigger than the numerator.

Examples: $\frac{c}{n}$, $\frac{n}{n^2}$, $\frac{n}{2^n}$, $\frac{n^2}{3^n}$ ...



Graph of a decrement function

As the size of i/p increases the decrement func. → decreases approaching to zero.

Problem: Arrange the following functions in the order of decrease from slowest to fastest.

$\frac{100}{n}$, $\frac{n}{2^n}$, $\frac{n}{n^2}$, $\frac{n^2}{3^n}$, $\frac{n^3}{3^n}$

Solution:

Step 1 : Simplify the fractions

$\frac{100}{n}$, $\frac{n}{2^n}$, $\frac{1}{n^2}$, $\frac{n^2}{3^n}$, $\frac{n^3}{3^n}$

Step 2 : Compare the denominators and arrange

$\frac{100}{n}$, $\frac{1}{n}$, $\frac{n}{2^n}$, $\frac{n^2}{3^n}$, $\frac{n^3}{3^n}$

$n < 2^n < 3^n$

Step 3 : Compare the numerators and arrange

$\frac{100}{n}$, $\frac{1}{n}$, $\frac{n}{2^n}$, $\frac{n^3}{3^n}$, $\frac{n^2}{3^n}$

$n=2$	50	0.5	0.5	0.8	0.4

Slowest ⟶ fastest

Comparison with decrement function

Constant Function → asyptotically bigger than the decrement function

Problem: $f(n) = \frac{100}{n}$ and $g(n) = 10,000$. Is $f(n) = O(g(n))$?

Soln: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

Assuming, $c = \frac{1}{10,000}$?

$\therefore \frac{100}{n} \leq \frac{1}{10,000} \cdot 10,000 = \frac{100}{n} \leq 1$ is True

$\frac{100}{n} = O(10,000)$ for $c = \frac{1}{10,000}$

$\therefore$ $g(n)$ is asymptotically bigger than $f(n)$.

Logarithm function

⇒ A fun. that grows positively, but with slower growth rate.

Example: $\log n$, $(\log n)^{10}$, $\log\log n$, $\log\log\log n$ etc.

$f(n) = \log n$

⇒ Asymptotically bigger than the Constant function

Problem: $f(n) = 100$ and $g(n) = \log_{10} n$. Is $f(n) = O(g(n))$?

Soln: According to the definition of Big O notation:

$f(n) = O(g(n))$ if $f(n) \leq c \cdot g(n)$ . . .

Assuming, $c = 1$

	100	$\log_{10} n$
$n = 10$	100	1
$n = 10^{100}$	100	100
$n = 10^{1000}$	100	1000

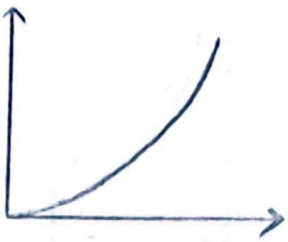
$100 \leq \log_{10} n$ is True.

$\therefore$ $g(n)$ is asymptotically bigger than the $f(n)$.

Polynomial functions

⇒ A fun. whose growth rate increases polynomially.

Example : $n^{0.1}$, $\sqrt{n}$, n^{v}, $n\log n$, n^{k} etc



$f(n) = n^{k}$ ← constant

⇒ asymptotically bigger than the logarithmic function.

Problem: $f(n) = \log_{10} n$ and $g(n) = \sqrt{n}$. Is $f(n) = O(g(n))$?

Sol^{n}: According to the definition of Big O notation,

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

Assuming, $C = 1$

	$\log_{10} n$	$\sqrt{n}$
$n = 10$	1	3.16
$n = 10^{v}$	2	10
$n = 10^{3}$	3	31.62
$n = 10^{4}$	4	100

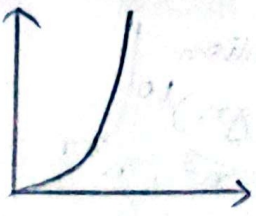
$\log_{10} n \leq \sqrt{n}$

∴ g(n) is asymptotically bigger than f(n)

Exponential Functions

⇒ A func. whose growth rate increases rapidly.

Example: 2^{n}, 3^{n}, $n!$, n^{n}, a^{n}



$f(n) = a^{n}$ ← constant

⇒ asymptotically bigger than the polynomial function.

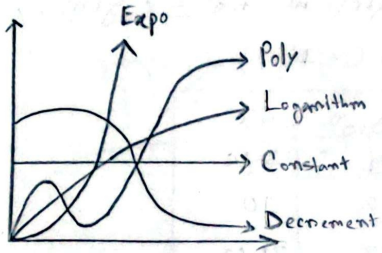
Problem: $f(n) = n^2$ and $g(n) = 2^n$. Is $f(n) = O(g(n))$?

Soln: According to the definition of Big O notation,
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . . .

Assuming, $c = 1$

	n^2	2^n
$n = 10$	100	1024
$n = 20$	400	1048576
$n = 30$	900	1073741824

$n^2 \leq 2^n$ is True
$\therefore$ $g(n)$ is asymptotically bigger than $f(n)$.



Dec Functions < Const Functions < Log Functions < Poly functions < Expo function

Functions in Asymptotic Notations

Problem: Write the following functions in asymptotically Increasing Order:
n^3, $n^{3.1}$, $n^2 \log n$, $\log n$, $(\log n)^{10}$, 100, 1 billion, $(0.3)^n$, $(1.5)^n$

Decrement Function: $(0.3)^n \rightarrow \frac{3^n}{10^n}$ → greater than 3^n
Constant " : 100, 1 billion
Logarithmic " : $\log n$, $(\log n)^{10}$
Polynomial " : $n^2 \log n$, n^3, $n^{3.1}$
Exponential " : $(1.5)^n \rightarrow \frac{15^n}{10^n}$ → not greater than 15^n

$(0.3)^n < 100 < 1 \text{ billion} < \log n < (\log)^{10} < n^2 \log n < n^3 < n^{3.1} < (1.5)^n$

Problem: Which of the following fun
f

Solution:

	n	$\log_{10}{}^n$	$\log_{10} \log_1$
$n_0 \rightarrow$	10^{10}	10	> 1
	10^{100}	100	> 2
	$10^{10^{10}}$	10^{10}	> 10
	$10^{10^{100}}$	10^{100}	> 100

Problem: Which of the following func
$f(n) = \log_{10} n$ and $g(n)$ =

n	$\log_{10}{}^n$	$(\log_{10} \log_1$
10^{10}	10	> 1
10^{100}	100	< 1024
$10^{10^{10}}$	10^{10}	= 10^{10}
$10^{10^{100}}$	10^{100}	> 100^{1}

Problem: Which of the following function is asymptotically biggeo?
$f(n) = \log_{10} n$ and $g(n) = \log_{10} \log_{10} n$.

Solution:

n	$\log_{10} n$	$\log_{10} \log_{10} n$
10^{10}	10	> 1
10^{100}	100	> 2
$10^{10^{10}}$	10^{10}	> 10
$10^{10^{100}}$	10^{100}	> 100

$n_0 \rightarrow$

$n_0 = 10^{10}$
$c = 1$

$\therefore g(n) = O(f(n))$
↑ is less than / equal to $f(n)$

Problem: Which of the following function is asymptotically biggeo?
$f(n) = \log_{10} n$ and $g(n) = (\log_{10} \log_{10} n)^{10}$

n	$\log_{10} n$	$(\log_{10} \log_{10} n)^{10}$
10^{10}	10	> 1
10^{100}	100	< 1024
$10^{10^{10}}$	10^{10}	$= 10^{10}$
$10^{10^{100}}$	10^{100}	$> 10^{10}$

$\log_{10}(10)^{10} = 1$; $\log_{10}(10^{10}) = 10$

$\therefore g(n) = O(f(n))$

The Big Omega (Ω) Notation

⇒ describes the Lower Bound of an algorithm.

⇒ Used to express the best case running time of an algorithm

Definition: Assuming $f(n)$ & $g(n) \rightarrow$ non-negative functions

$f(n) = \Omega(g(n))$ iff $f(n) \geq c \cdot g(n)$ for some $c > 0$ and for all values of n where $n \geq n_0$ and c and n_0 are constants.

$g(n)$ is the asymptotic lower bound of $f(n)$.

The Big Theta (Θ) Notation

Assuming $f(n)$ & $g(n) \rightarrow$ non negative functions

$f(n) = \Theta(g(n))$ iff $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all values of n where $n \geq n_0$ and c_1, c_2 and n_0 are constants.

$g(n)$ is the asymptotic tight bound of $f(n)$.

Math

Prob 1: Find the Lower bound for $f(n) = 10n^2 + 5$

Soln:

Step 1: Find the Dominant term = $10n^2$

Step 2: find $g(n)$. Assuming, $g(n) = n^2$

Step 3: find c. Assuming, $c = 10$

According to the definition of Big Omega (Ω) notation,

$f(n) = \Omega(g(n))$ iff $f(n) \geq c \cdot g(n)$ for all values of n where...

$10n^2 + 5 \geq 10n^2$ is True for all $n \geq 1$

$\therefore 10n^2 + 5 = \Omega(n^2)$ for $c = 10$ and $n_0 = 1$

n	$10n^2+5$	$10n^2$
1	15	10

Problem 2: Show that $f(n) = n^3 + 3n^2 = \Theta(n^3)$

Soln: Given, $g(n) = n^3$

According to the definition of Big Theta notation:
$f(n) = \Theta(g(n))$ iff $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all values of n where $n \geq n_0$ and c_1, c_2 and n_0 are constants.

n	n^3+3n^2	$2n^3$
1	4	2
2	20	16
$n_0 \rightarrow 3$	54	54

Prove: $c_1 g(n) \leq f(n)$

Assuming, $c_1 = 1$

$n^3 \leq n^3 + 3n^2$ is True

Prove: $f(n) \leq c_2 \cdot g(n)$

Assuming, $c_2 = 2$

$n^3 + 3n^2 \leq 2n^3$ is True for $n_0 = 3$

$\therefore n^3 + 3n^2 = \Theta(n^3)$ for $c_1 = 1$, $c_2 = 2$ and $n_0 = 3$.

Properties of Asymptotic Notations

If $f(n) = O(g(n))$, then $a * f(n) = O(g(n))$

Problem 1: $f(n) = 3n^2 + 9$ and $g(n) = n^2$. Is $f(n) = O(g(n))$?

Soln: Given, $g(n) = n^2$

According to the def. of Big O notation,
$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ . . .

$3n^2 + 9 \leq 4n^2$
$\Rightarrow 9 \leq 4n^2 - 3n^2$
$\Rightarrow 9 \leq n^2$
$\Rightarrow 3 \leq n$

$\therefore f(n) = O(g(n))$

Assuming, $c = 4$

n	$3n^2+9$	$4n^2$
1	12	4
2	21	16
$n_0 \rightarrow$ 3	36	36

$3n^2+9 \leq 4n^2$ is True

$\therefore 3n^2+9 = O(n^2)$ for $c = 4$ and $n_0 = 3$

Assuming, $a = 1000$

$a * f(n) = O(g(n))$
$\Rightarrow 1000(3n^2+9) = O(n^2)$
$\Rightarrow 3000n^2 + 9000 = O(n^2)$

Some C will greater than 3000 which $c \cdot g(n)$ will bigger than dominant term $3000n^2$

$\therefore a * f(n) = O(g(n))$ is True

Divide and Conquer Algorithm

Problem solving Strategies & Algorithms

1. Brute Force Strategy

↳ Tries all possible solutions until the correct one is found

Use: Effective for small problem spaces

Example: Checking all factory records (manually for inconsistencies)

2. Divide and Conquer

↳ Divides a problem into smaller sub problems, solves each recursively & combines solutions

Use: Solving large, complex problems like supply chain optimization

Example: Analyzing production stages separately before combining the results.

3. Greedy Algorithm

↳ Makes the locally optimal choice at each step with the hope of finding the global optimum.

Use: Useful for allocation problems.

Example: Assigning workers → machines (based on immediate delivery)

4. Dynamic Programming

↳ Solves problem by breaking them down into overlapping subproblems and storing their solutions.

Use: Optimizing costs, efficiency/production scheduling

Example: Finding the most efficient sequence of manufacturing tasks.

5. Backtracing

↳ Builds up solutions incrementally and abandons solutions that fails to satisfy constraints.

Use: Decision making problems where constraints (must be respected)

Example: Workers shift scheduling under labor law constraints.

6. Heuristic Methods

↳ Provides approximate solutions quickly when exact solutions are impractical.

Use: Complex, Real world problems like predicting demand in garment production.

Example: Estimating staffing needs based on historical data trends.

7. Optimization Algorithms

↳ Focus on finding the best solution according to some criteria.

Use: Maximizing production op / minimizing cost.

Example: Linear Programming, Integer Programming

8. Evolutionary Algorithm (Genetic Algorithm)

↳ Uses mechanisms inspired by biological evolution, like: mutation, crossover and selection

Use: Solving optimization problems where the solution space is very large.

Example: Designing the most efficient Family Layout.

9. Maching Learning Strategies

↳ Algorithms learn from data and improve over time.

Use: Predictive maintenance, quality control, demand forecasting in RMG.

Examples: Decision Trees, Neural Networks.

10. Game Theory Strategies

↳ Mathematical modeling of strategic interaction among rational decision-makers.

Use: Negotiating buyer contracts / labor union agreements

Example: Modeling Factory-buyer price negotiations.

Divide and Conquer

An algorithmic paradigm that solves a problem by:

1. Dividing the problem into smaller subproblems of the same type
2. Conquering each subproblem recursively
3. Combining their solutions to solve the original problem.

⇒ highly efficient for problems that can be naturally broken down into smaller, similar problems.

Three Key phases of Divide & Conquer Algorithm are:

(1) Divide: Split the problem into smaller parts

(2) Conquer: Solve the subproblems recursively. If the subpro

(3) Combine: Merge the solutions of subproblems into the solution of the original problem.

```
DAC(P)                                  Function DivideandConquer(Problem)
{
  if (small(P))                         if (Problem is small enough)
    {
      S(P);                             Solve Problem directly
    }
  else
    {
      divide P into P1,P2,P3,..Pk       divide problems into sub problems
      Apply DAC(P1), DAC(P2),....       solve each subproblem recursively
      Combine(DAC(P1), DAC(P2),....)    combine solution of subproblem
    }
}
```

Common Examples

Algorithm	Explanation
Merge Sort	Divides array into halves, sorts each half then merge them.
Quick Sort	Divides array based on Pivot, recursively sorts left and right parts.
Binary Search	Divides search space into halves to find an element.
Matrix Multiplication (Strassen's Algorithm)	Divides matrices into sub-matrices and multiplies recursively.

Recursive function

↳ A func. that calls itself directly/ indirectly to solve a problem.

⇒ Solves big problem by solving smaller and smaller pieces of it & it keeps calling itself until it finishes.

⇒ It breaks down a problem into smaller instances of the same problem until it reaches a → base case, which can be solved without further recursion.

Key components of Recursive function

1. **Base Case:**
   → condition that stops further Recursion
   → Without a Base case, recursion would continue indefinitely.

2. **Recursive Case:**
   → Part where the function calls itself to work on a smaller or simpler subproblem.

Simple Example

Ex: Factorial of a number n:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Recursive definition:

Base case – $0! = 1$

Recursive case – $n! = n \times (n-1)!$

Ex:
Factorial(3) → 3 × Factorial(2)
Factorial(2) → 2 × Factorial(1)
Factorial(1) → 1 × Factorial(0)
Factorial(0) → 1 (base case)

Final Result:
$3! = 3 \times 2 \times 1 = 6$

- Mahiy mere mahi
- Me dil ka Ras kehalu hu, Mitwa

Summary:

Term	Meaning
1. Recursive function	A function that calls itself to solve a prob
2. Base Case	Stops function
3. Recursive Case	Breaks problem into smaller parts

The Call Stack

⇒ A specific memory space used to manage function calls.

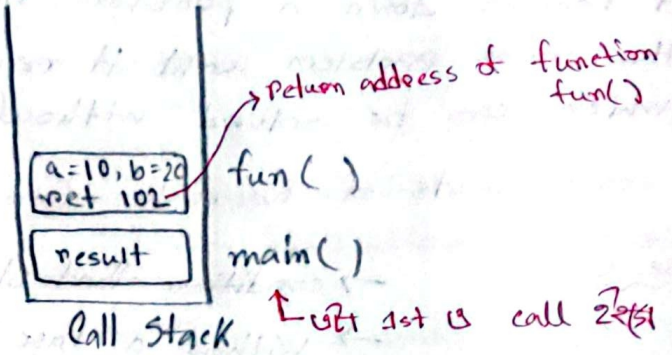
⇒ When a func. is called, the activation record goes inside the stack.

↳ Contains information
i) return address
ii) variable

Ex:

```
int fun(int a, int b){
    return a+b
}
main(){
101 int result;
102 result = fun(10,20);
103 return 0;
}
```

Return address of this instruction



⊛⊛ After completion of the function its → Activation Record goes outside of the stack. [It will be Removed from the Stack]

Concept	Meaning
Call Stack	Memory Structure to keep track of function calls.
Push	Adding a function call to the top of the stack
Pop	Removing the function call after it completes.
LIFO	Last In, First Out

Recurrence Relation

⇒ A mathematical equation $\xrightarrow{\text{defines}}$ a sequence recursively, where each term is expressed as a function of its preceding term.

⇒ A mathematical expression $\xrightarrow{\text{defines}}$ a sequence/a function in terms of its own earlier values.

Exba:

- ✱ It expresses the value of something (like the time it takes to solve a problem) based on smaller versions of the same problem.
- ✱ A foundational tool in mathematics & computer science for modeling recursive structures, algorithms & sequences.

```
Void Test(int n)
{
 if(n>0)
   {printf("%d", n);
    Test(n-1);
   }
}
```

Tracing Tree/Recursion Tree

↳ A visual tool used to map out the sequence of recursive function calls and their outcomes.

✱ It helps beginners understand how recursion "unfolds" step by step, *showing how a problem $\xrightarrow{\text{is}}$ broken into smaller subproblems $\xrightarrow{\text{until}}$ *reaching the base case.

Que: Trace the Recursive Factorial function `Fact(n)` using a Tree?

```
def fact(n)
{ if (n==0)
   {return 1
   }
  else
    return n* fact(n-1)
```

Step 1: Root node (Initial call)
Start with the 1st call: fact(3)

Step 2: Expand the node.
fact(3) call fact(2), creating a branch:

fact(3)
|
fact(2)

Step 3: Continue expanding
Repeat until the base case (n = 0)

fact(3)
|
fact(2)
|
fact(1)
|
fact(0) → return 1 (base case)

Step 4: Unwind the tree (Return Values)

fact(0) returns 1 → fact(1) computes 1 * 1 = 1
fact(1) returns 1 → fact(2) computes 2 * 1 = 2
fact(2) returns 2 → fact(3) " 3 * 2 = 6

∴ Final Result : fact(3) = 6

Ex 1:

```
void Test(int n) → Test(3)
{
 if(n>0)
 {
  printf("%d",n);
  Test(n-1);
 }
}
```

3 2 | 2 1 | 1 0 → 0 → ×

3 → print

Test(3)
Test(2) — call
2
Test(1)
1
Test(0)
|
×

∴ f(n) = n+1 calls. Time Complexity = O(n)

Que: Solve this problem by substitute method

```
void Test(int n)
{
 if(n>0)
 {
  printf("%d",n); → 1
  Test(n-1); → T(n-1)
 }
}
```

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

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

Step by Step solution of $T(n) = T(n-1) + 1$

Step 1: Expand the recurrence

Start expanding : $T(n) = T(n-1) + 1$ $[\because T(n-1) = T(n-1-1) + 1]$

Now expand : $T(n-1) = [T(n-1-1)+1] + 1 = T(n-2) + 2$ $[\because T(n-2) = T(n-3) + 1]$

Expand further : $T(n-2) = [T(n-2-1)+1] + 2 = T(n-3) + 3$

Similarly : $T(n-3) = [T(n-3-1)+1] + 3 = T(n-4) + 4$

Continue expanding . . .

After K steps : $T(n) = T(n-K) + K$

Step 2: Reach the base case

We keep expanding until $\boxed{n-K = 0}$.

Thus : $K = n$

Substitute $K=n$ into the expanded form:

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

$T(n) = T(0) + n$; where $T(0)$ = time to solve the problem of size = $1+n$ (a constant time = 1)

$= 1 + n$

$\therefore \boxed{T(n) = O(n)}$ because $T(0)$ is a constant.

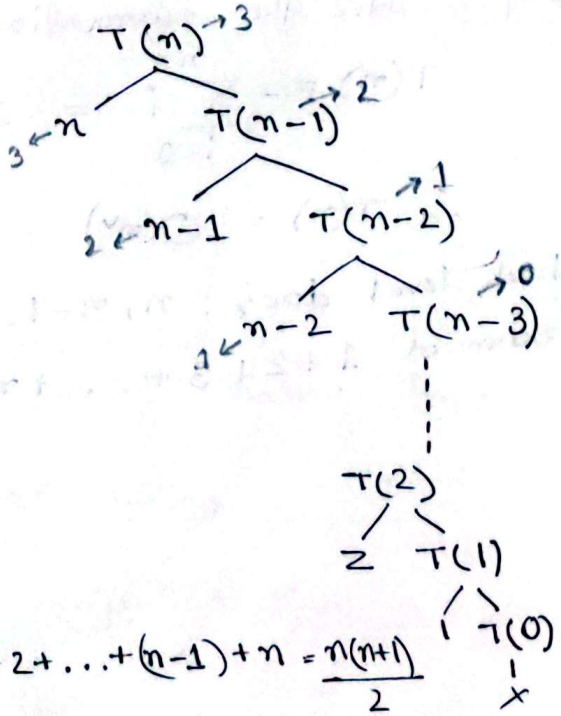
Ex 2:

```
void Test(int n) ⟶ T(n)
{
 if (n>0)  ⟶ 1
   {
    for(i=0 ; i<n ; i++) → n+1   (n times for true, 1 more time when false)
     {
      printf("%d",n); → n
     }
     Test(n-1); ⟶ T(n-1)
   }
}
```

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

$\approx T(n-1) + n$

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



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

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

$= O(n^2)$

... of the solution for $T(n) = T(n-1) + n$

Step 1: Expand the recurrence

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

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

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

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

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

By continuing this for K times,

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

Assume, $n-k = 0 \Rightarrow n = k$

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

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

$$= 1 + \frac{n(n+1)}{2} \qquad \left[\because 1+2+3+\ldots+n = \frac{n(n+1)}{2}\right]$$

Step 2: Solve the summation

$$T(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

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

Each level does : $n, n-1, n-2, .., 1$ units of work

Sum of $1+2+3+\ldots+n = O(n^2)$

Level 1 $(n) \rightarrow n$

Level 2 $(n-1) \rightarrow n-1$

Level 3 $(n-2) \rightarrow n-2$

....

Level $(n-1)$ $1 \rightarrow 1$

Each level removes 1 item

Exam ও মিলিয়ে পড়ো

Recurrence Comparisons

Recurrence	Expansion	Final time
$T(n) = T(n-1) + 1$	Adding 1 at each step	$O(n)$
$T(n) = T(n-1) + n$	Adding $1+2+3+\ldots+n = \frac{n(n+1)}{2}$	$O(n^2)$
$T(n) = T(n-1) + n^2$	Adding $1^2+2^2+3^2+\ldots+n^2$	$O(n^3)$

Ex 3:

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

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

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

1) Why $T(n) = T(n-1) + \log n$?
2) Find the Time Complexity/Final Time using Tracing Free?
3) Find the Time Complexity/Final Time using expanding the Recurrence.
   (Show Step by Step Solution for: $T(n) = T(n-1) + \log n$)
4) What if $i = i*3$ in the loop?

```
for(i=0; i<n; i=i*3)
```

1) This is a recursive function that run until $n \leq 0$ & there is a loop that iterates with $i = i * 2$ until $i < n$.

Now, let's find the runtime for loop.

- After 1st iteration, $i = 2$
- After 2nd iteration, $i = 2 \times 2 = 2^2$
- After 3rd iteration, $i = 2^2 \times 2 = 2^3$

→ After K iteration, $i = 2^k$

The termination condition for loop,

$$i \geq n$$

So, $2^k \geq n$

Assume, $2^k = n$

$\Rightarrow k = \log_2 n$ [∵ Take log (base 2) on both sides]

Now, The time for the recursive call Test $(n-1)$, is $T(n-1)$. Because after loop the function call itself with $n-1$ (means remaining work is exactly same as solving the problem for $n-1$)

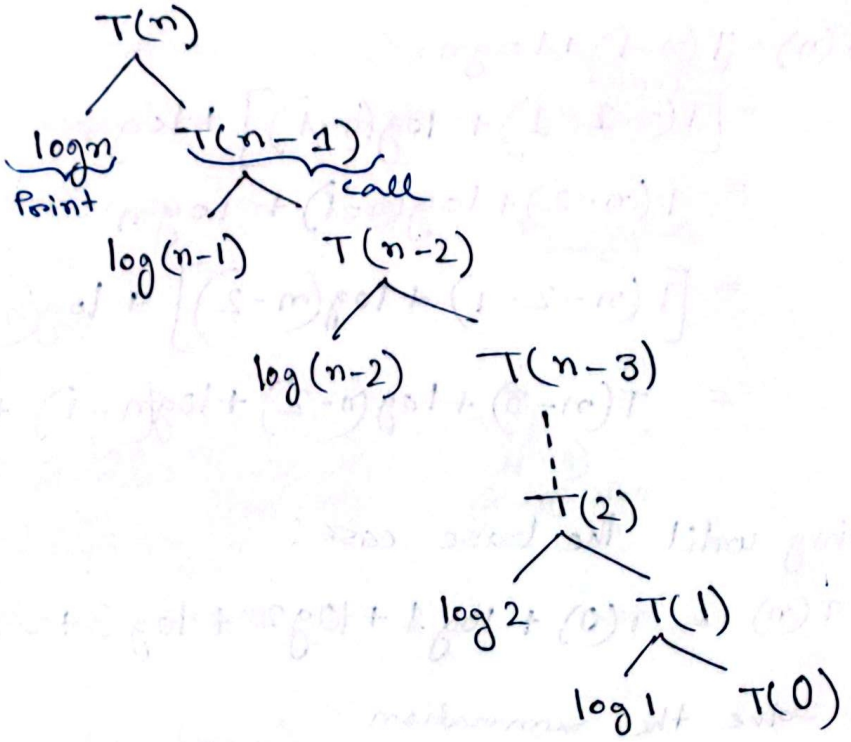
As, Test (int n) time is $T(n)$

Then, Test $(n-1)$ is $T(n-1)$

$\therefore T(n) = T(n-1) + \log_2 n$.

$\approx T(n-1) + \log n$.

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



So,

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

$= \log [n*(n-1)*(n-2)*\ldots 2*1]$ $\quad [\because \log a + \log b = \log a \times b]$

$= \log n!$

$\approx n \log n$

$\therefore$ Time complexity $= O(n \log n)$

3) Step by step solution for: $T(n) = T(n-1) + \log n$

Step 1: Expanding the recurrence

$$T(n) = \underbrace{T(n-1)}_{\text{Expand}} + \log n$$

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

$$= \underbrace{T(n-2)}_{\text{Expand}} + \log(n-1) + \log n$$

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

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

$(n-k)$ $(n-(k-1))$ $(n-(k-2))$

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

by expanding until the base case:

$$T(n) = T(0) + \log 1 + \log 2 + \log 3 + \ldots + \log n$$

Step 2: Solve the summation

$$T(n) = \log(1 \times 2 \times 3 \times \ldots \times n)$$

$$= \log(n!)$$

$$= n \log n$$

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

Level	1	(n)	$\rightarrow \log n$
Level	2	$(n-1)$	$\rightarrow \log(n-1)$
Level	3	$(n-2)$	$\rightarrow \log(n-2)$
...			
Level	$n-1$	1	$\rightarrow \log 1$

4) for loop $i = i * 3$, $i < n$

- First iteration, $i = 3$
- After 2nd iteration, $i = 3 \times 3 = 3^2$
- After 3rd iteration, $i = 3^2 \times 3 = 3^3$
- After K iteration, $i = 3^k$

The termination condition for loop,

$$i \geq n$$
$$\Rightarrow 3^k \geq n$$
$$\Rightarrow 3^k = n$$
$$\Rightarrow k = \log_3 n$$

$\therefore T(n) = T(n-1) + \log_3 n$
$\approx T(n-1) + \log n$

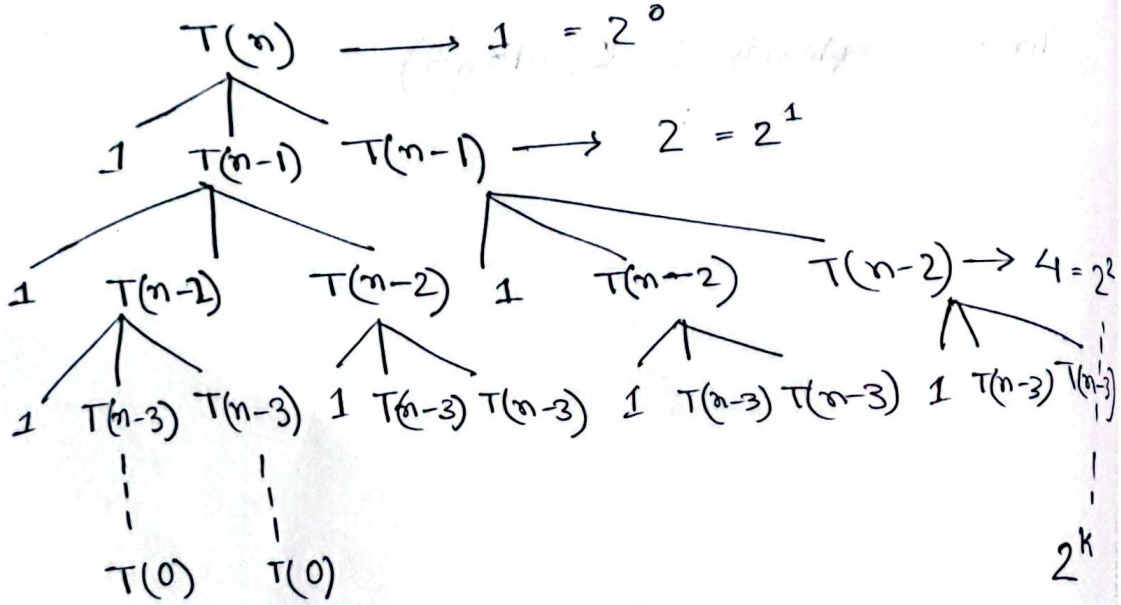
Time complexity : $O(n \log n)$

Ex 4

```
void Test (int n) ⟶ T(n)
{
  if (n>0)
    {
    printf("%d ",n); ⟶ 1
    Test (n-1) ⟶ T(n-1)
    Test (n-1) ⟶ T(n-1)
    }
}
```

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

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



Now,

$1 + 2 + 2^2 + 2^3 + \ldots + 2^k = 2^{k+1} - 1$

we know, $a + ar + ar^2 + ar^3 + \ldots + ar^k = \frac{a(r^{k+1}-1)}{r-1}$

$= \frac{1(2^{k+1}-1)}{2-1}$ [$\because a=1, r=2$]

$= 2^{k+1} - 1$

Assume, $n - k = 0 \Rightarrow n = k$

$\therefore \quad 2^{n+1} - 1 \approx 2^n \longrightarrow$ Time Complexity $= O(2^n)$

Step by step solution for $T(n) = 2T(n-1)+1$

Step 1: Expand the recurrence

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

Expand

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

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

Expand

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

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

Expand

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

$$= 2^4 T(n-4) + 2^3 \times 1 + 2^2 \times 1 + 2 \times 1 + 1$$

After K steps:

$$T(n) = 2^K T(n-K) + \sum_{i=0}^{K-1} 2^i$$

Step 2: Reach the Base case

Assume, $n-K=0 \Rightarrow n=K$

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

$$= 2^n \cdot 1 + 2^n - 1 = 2^{n+1} - 1$$

$\therefore$ Time complexity $= O(2^n)$

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{K-1} = \sum_{i=0}^{K-1} 2^i = 2^K - 1$$

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^K = \ldots = 2^{K+1} - 1$$

In Big O, we drop constants and lower order terms.

Masters Theorem Decreasing Function

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

Here,
a = No. of subproblems generated
b = Size decrease
f(n) = work done outside the recursion

- Applies when the problem size Reduces by subtraction $(n-b)$
- Used in problems like : factorial, Simple Recursion etc.

Expantion of the Recurrence :

$$T(n) = a\,T(n-b) + f(n)$$
$$= a\,[a\,T(n-b-b) + f(n-b)] + f(n)$$
$$= a^2 T(n-2b) + a\,f(n-b) + f(n)$$
$$= a^2\,[a\,T(n-2b-b) + f(n-2b)] + a\,f(n-b) + f(n)$$
$$= a^3 T(n-3b) + a^2 f(n-2b) + a\,f(n-b) + f(n)$$

After K steps:

$$T(n) = a^K T(n-Kb) + \sum_{i=0}^{K-1} a^i f(n-ib)$$

Assume, $n - kb = 0$

$$\Rightarrow n = kb$$
$$\Rightarrow k = \frac{n}{b}$$

Now, $T(n) = a^{\frac{n}{b}}\,T\left(n - \frac{n}{b}\cdot b\right) + \sum_{i=0}^{K-1} a^i f(n-ib)$

$$= a^{\frac{n}{b}}\,T(0) + \text{Summation of work } \sum_{i=0}^{K-1} a^i f(n-ib)$$

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

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

Three cases for a

$T(n) = T(n-1) + 1 \rightarrow O(n)$

$T(n) = T(n-1) + n \rightarrow O(n^2)$

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

$T(n) = 2T(n-1) + 1 \rightarrow O(2^n)$

$T(n) = 3T(n-1) + 1 \rightarrow O(3^n)$

$T(n) = 2T(n-1) + n \rightarrow O(n\,2^n)$

$T(n) = 2T(n-2) + 1 \rightarrow O(2^{\frac{n}{2}})$

Case (for a)

1) if $a < 1$, $O(n^k) = O(f(n))$

2) if $a = 1$, $O(n^k \cdot n) = O(f(n) * n)$

3) if $a > 1$, $O(n^k \cdot a^n) = O(n^k * a^{n/b})$ $[\because b > 1]$

$= O(f(n) * a^{\frac{n}{b}})$

Three cases of Master Theorem

↳ depending on how $f(n)$ reacts compared to a^n.

Case 1: $f(n)$ is smaller $= O(a^{n/b})$

Then $T(n) = O(a^{n/b})$ ✓ Recursion dominates

$f(n) = 1$ Ex: $T(n) = 2T(n-1) + 1$ $[\because a = 2, f(n) = 1]$

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

Case 2: $f(n)$ same as recursion $= O(a^{n/b} * n)$

$f(n) = n$ Ex: $T(n) = 2T(n-1) + n$

$\therefore T(n) = O(2^n * n)$

Case 3: $f(n)$ is bigger $= O(\text{sum of } f(n))$

Ex: $T(n) = T(n-1) + n^2$ (big compared to recursion, $a = 1$)

$f(n) = n^2$

$\therefore T(n) = O(n^3)$

General Form of Divide-and-Conquer Recurrence

```
int main(){
    int n = 16;
    int a = 2;
    int b = 2;
    int result = T(n, a, b)
    printf ("T(%d) = %d ", n, result);
    return 0;
}
int T(int n, int a, int b)
{
  if (n <= 1)
    {return 1;
    }
  else
    return a * T(n/b;
    }
```

* Base case : If n becomes smaller (<= 1), stop Recursion. and return 1.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$ → Classic Master's Theorem

Here,

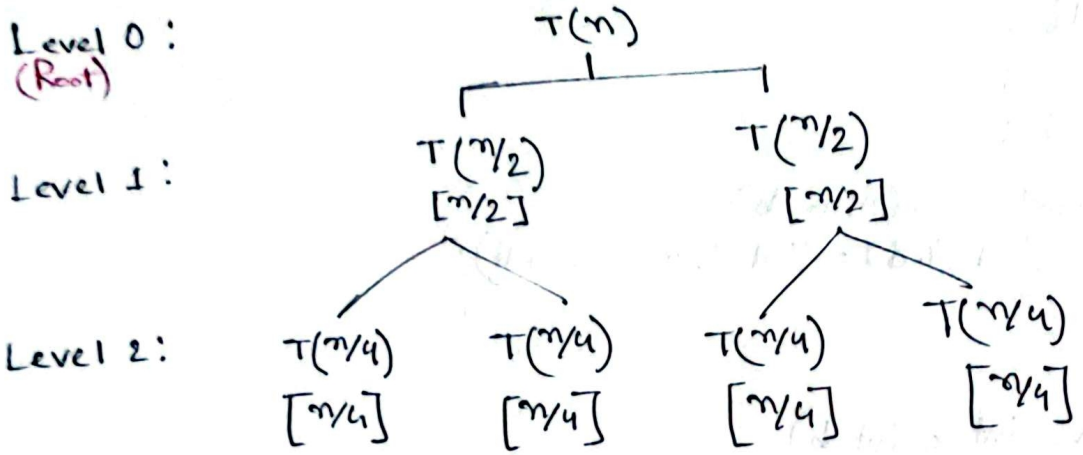
a = no. of subproblems

b = division factor

f(n) = work done outside the recursion

Deduce the recurrence Relation (using Tracing tree)

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$



Root = original problem of size n
work done at Level 0 : $f(n) = n$

Here, Each level has: 2^i nodes

Each node does work $= \frac{n}{2^i}$

Total work per level $= 2^i * \frac{n}{2^i} = n$

Now, Number of level (Tree Height) :

The recursion stops $\frac{n}{2^k} = 1$

$\Rightarrow k = \log_2 n$

$\therefore$ No. of levels $= \log_2 n + 1$ → for level 0

Now, Total Work (Sum over levels) :

Total cost per level = n units of work

$\therefore$ Total cost $= n * (\log_2 n + 1)$

$= n \log_2 n + n$

$T(n) = \Theta(n \log_2 n)$

Substitution method for Masters theorem

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \rightarrow \text{for } T(n)$$

$$T(n) = a\left[a\,T\left(\frac{n/b}{b}\right) + f\left(\frac{n}{b}\right)\right] + f(n) \rightarrow \text{for } T\left(\frac{n}{b}\right)$$

$$= a^2\,T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \quad \text{for } T\left(\frac{n}{b^2}\right)$$

$$= a^2\left[a\,T\left(\frac{n/b^2}{b}\right) + f\left(\frac{n}{b^2}\right)\right] + af\left(\frac{n}{b}\right) + f(n)$$

$$= a^3\,T\left(\frac{n}{b^3}\right) + a^2 f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n)$$

$\rightarrow \frac{n}{b^k} = 1$

After K times expanding :

$$T(n) = a^k\,T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$$

Recursion stops when :

$$\frac{n}{b^k} = 1 \quad **$$

$$\Rightarrow n = b^k$$

$$\Rightarrow k = \log_b n$$

$$\therefore T(n) = a^{\log_b n} \cdot \Theta(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$= n^{\log_b a} \cdot \Theta(1) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

$$a^{\log_b n} = n^{\log_b a}$$ Formula

Depending on $f(n)$, 3 cases (Masters Theorem)

Case	Condition	Result
1	$f(n) = O(n^{\log_b a - \varepsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
3	$f(n) = \Omega(n^{\log_b a + \varepsilon})$	$T(n) = \Theta(f(n))$

Master Theorem

→ Gives a shortcut to find time complexity of Divide & Conque recurrences without drawing Recursion Trees & substituting repeatedly.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Here, $a \geq 1$: no. of subproblems/recursive calls
$b > 1$: size of each subproblems (reduced to n/b)
$f(n)$: cost of dividing and combining

Step by Step Guide to Apply Master Theorem

Step 1: Identify a, b, $f(n)$

Step 2: Compute $n^{\log_b a}$. (This is comparison baseline)

Step 3: Compare $f(n)$ with $n^{\log_b a}$.
(This step determines → which case of MT to apply)

Step 4: Pick the case & write final ans.
(Use Case-Table to write Θ complexity)

Suppose, $b = 2$, $a = 8$

Level 0 : Start with size $\boxed{n}$

Here, work done = $f(n) = n^{\log_b a}$

Level 1 : Divide the problem into parts, each size $\boxed{\frac{n}{b}}$

Work done $= a * f\left(\frac{n}{b}\right)$

$= a * \left(\frac{n}{b}\right)^{\log_b a}$

$= a * \frac{n^{\log_b a}}{b^{\log_b a}}$

$= a * \frac{n^{\log_b a}}{a^{\log_b b = 1}} = n^{\log_b a}$

Level 2 : Each subproblems again splits into subproblems.

Now, a^2 subproblems of size $\boxed{\frac{n}{b^2}}$

Total work (at this level) $\approx n^{\log_b a}$

* The recursion goes until n becomes : $n \to \frac{n}{b} \to \frac{n}{b^2} \to \dots \to 1$

Here, Work at each level = $n^{\log_b a}$

Number of levels = $\log n$

$\therefore$ Total Work = Work per level $\times$ No. of levels

$= n^{\log_b a} \times \log n$

$$\boxed{T(n) = n^{\log_b a} \cdot \log n}$$

Ex 1 : Merg Sort : $T(n) = 2T(n/2) + n$

Step 1 : Here, $a = 2$, $b = 2$, $f(n) = n$

Step 2 : $n^{\log_b a} = n^{\log_2 2} = n$

Step 3 : Compare $f(n)$ with $n^{\log_b a}$

$\underbrace{n^{\log_b a} = f(n)}_{\text{Equal}}$ [$\because f(n) = n$]

Step 4 : Case 2

$$T(n) = \Theta(n^{\log_b a} \log n)$$
$$= \Theta(n^{\log_2 2} \log n)$$
$$= \Theta(n \log n)$$

Ex 2: Binary Search : $T(n) = T(n/2) + 1$

Here, $a = 1$, $b = 2$, $f(n) = 1$

$\therefore n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Now, $n^{\log_b a} = f(n)$

$\Rightarrow \underbrace{1 = 1}_{\text{Equal}}$

Case 2 $\Rightarrow$ $T(n) = \Theta(n^{\log_b a} \log n)$
$= \Theta(n^{\log_2 1} \log n)$
$= \Theta(n^0 \log n)$
$= \Theta(\log n)$

Ex 3: Strassen's Matrix multiplication : $7T(n/2) + n^2$

Here, $a = 7$ $b = 2$ $f(n) = n^2$

$\therefore n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

Now, $f(n) = n^2 = O(n^{2.81})$

Case 1 $\Rightarrow$ $T(n) = \Theta(n^{\log_b a})$
$= \Theta(n^{2.81})$

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

Here, $a = 1$, $b = 2$, $f(n) = n$

$\therefore n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Comparing: $n^{\log_b a} = 1$

$f(n) = n$

Shows, $f(n)$ is faster compared to $n^{\log_b a}$ $\left(n^k > n^{\log_b a}\right)$

Case 3 : $T(n) = \Theta(f(n))$

$= \Theta(n)$

Searching and Sorting

[1] Binary Search :

Recurrence Relation: $T(n) = T\left(\frac{n}{2}\right) + c$

⇒ Data গুলো sorted থাকতে হবে
⇒ Cuts the array size in half each time.
⇒ Time Complexity: $T(n) = T\left(\frac{n}{2}\right) + c \Rightarrow O(\log n)$

Que: What Binary Search does?

- Binary Search works on sorted arrays. At each step:
- It cheks the middle element of the current range
- If it matches the target, it's done.
- If the target is smaller → it searches the left half
- If the target is larger → " " " right "
- This reduces the size of the problem (from elements to n/2 elements with each step)

Example

Sorted Array: $A = [2, 4, 7, 10, 13, 18, 21, 27, 31]$

i) Find 13
Here, Low = 0, High = 8 ⇒ $\frac{0+8}{2}$ = 4th array (mid) ∴ A[4] = 13
Target = 13
∴ Index of 13 is 4.

ii) Find 21
mid = $\frac{0+8}{2}$ = 4 ∴ A[4] = 13 → too small go right.
Update ⇒ Low = 5, High = 8
mid = $\frac{5+8}{2}$ = 6.5 ∴ A[6] = 21

∴ Value at index [6] is 21.

What the Recurrence Means

Why this Recurrence Makes Sense: ⇒ See Slide

Merge Sort

Recurrence Relation : $T(n) = 2T(\frac{n}{2}) + cn$

⇒ Divide and Conquer Algorithm.

i) Dividing the array into two halves

ii) Recursively sorting two halves, each of size $n/2$:
gives us : $2T(\frac{n}{2})$

iii) Merging the two sorted halves into a single sorted array.

* Merging two arrays of total size n takes linear time ∴ cn

⇒ Total time for sorting array :

$$T(n) = \underbrace{2T(\frac{n}{2})}_{\text{Recursive Sorting}} + \underbrace{cn}_{\text{Merging}}$$

Solving the Recurrence (Time Complexity):

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

$$= 2\left[2T\left(\frac{\frac{n}{2}}{2}\right) + c\left(\frac{n}{2}\right)\right] + cn$$

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

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

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

$$= 2^k T\left(\frac{n}{2^k}\right) + kcn \quad [\text{Repeat until base case}]$$

when, $\frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

$\Rightarrow k = \log_2 n$

$\therefore T(n) = nT(1) + \log_2 n * cn$

Time complexity: $O(n \log n)$

Example

$A = [38, 27, 43, 3, 9, 82, 10]$

Quick Sort

Recurrence Relation: $T(n) = T(n-1) + cn$

⇒ Divide and conquer algorithm

i) Picks a pivot's element

ii) Partions the array into two parts:

- Elements less than the Pivot
- Elements greater than the Pivot

iii) Recursively sorts each part

* What happens in the worst case?

⇒ When the pivot is the smallest or largest element every time. This causes one of the partitions to be empty & the other to have $n-1$ elements.

⇒ Instead of dividing into two balanced parts, we get

- One side with 000 elements
- One side with $n-1$ elements

* Setting up the Reccurrence

cn : Time to partition the array

$n-1$: Recursive call on the $n-1$ sized subarray

* Expand the Reccurrence

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

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

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

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

$$\vdots$$

$$= T(1) + c(2+3+\ldots+n)$$

The sum $2+3+\ldots+n \longrightarrow$ arithmetic series

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

$$\therefore T(n) = T(1) + c \cdot \left(\frac{n(n+1)}{2} - 1\right)$$

$$= O(n^2)$$

Average Case:

i) Chooses a Pivot

ii) Partitions the array into two parts based on the pivot:
- Left: element < pivot
- Right: elements > pivot

iii) Recursively sorts both partitions

⇒ In the average case, the pivot divides the array into reasonably balanced parts – not perfectly in half, but not worst-case skewed either.

⇒ Assume (on average) the pivot splits the array into two parts of size approximately $n/2$ each

Recurrence Relation

Insertion Sort

Recurrence relation: $T(n) = T(n-1) + cn$

How insertion Sort works:

- Insertion Sort builds the sorted list one element at a time.
- Start with the 2nd element.
- Compare it with element to its left
- Shift larger element to the right
- Insert the current element into its connect position.

(*) Each time we insert the nth element:

$n-1$: Time to sort $n-1$ elements

cn : Time to insert the nth element.

Algorithm	Best Case	Average Case	Worst Case
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$