Life was never meant to turn into this!

Why algorithms are important?

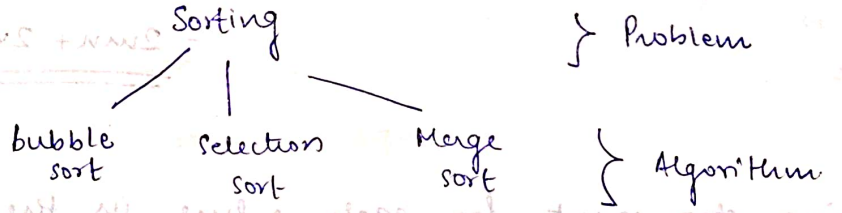
↳ many uses in different apps (google map, insta, etc.)

↳ they tell us how to build logic.

Algorithm → set of steps for a task.

Problem → generic

Algorithm → solution to the problem.



Posteriori analysis

hardware dependent.

↳ can the algo be run on different computers

↳ " " " " " with different clock speeds

↳ " " " " " with different instruction set

↳ " " " " " with different memory access speed.

if yes, then which system to choose?

analysis in this case depends on hardware on which algo runs.

∴ not practical.

Priori analysis

Instead of measuring the time taken by algorithm to run on a system, we should measure something like the number of steps it takes.

This is called priori analysis

Algorithm Add(a, b, c, m, n)

```
{
    for i=1 to m do                    ---------- m+1
        for j=1 to n do                ---------- m(n+1)
            c[i,j] = a[i,j] + b[i,j]   ---------- m.n.
}
```

Step count.

∴ Total no. of steps = $m+1 + m(n+1) + mn$

$= m+1 + mn + m + mn$

$= 2mn + 2m + 1$

Finding step count for each line in the algorithm is difficult and confusing as the program size grows.

Therefore, we define some bounds on time complexity instead of finding the exact step count.

~~This leads to find~~ improved version of the step count - Asymptotic analysis.

Asymptotic analysis

- Goal: to simplify analysis of running time by getting rid of 'details' (constants and lower order terms).
  - ↳ rounding off
- It is a technique that focuses on the significant term
- It tells us how the running time of an algorithm increases with the size of input.

Asymptotic notations

- Big-oh O notation
- Big Omega Ω notation
- θ Notation
- o little-oh
- ω little omega

} mathematical notations to represent time complexity or space complexity as a function of input size.

1. Big-oh O notation

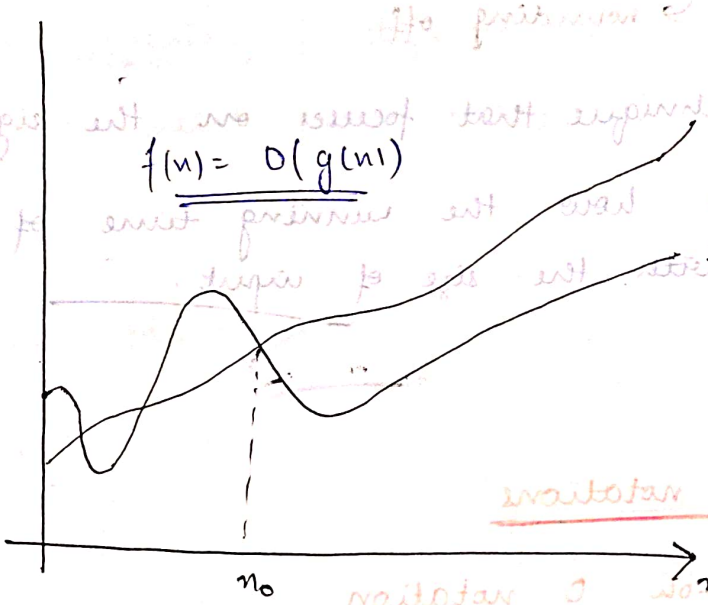
- $T(n)$ is $O(g(n))$ if there exist constant $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$

$$T(n) \leq c\,g(n)$$

- provides asymptotic upper bound.

Suppose $T(n) = O(n^2)$

Time complexity of my algorithm can be atmost n^2 for any input.



⊙ Asymptotic analysis is done for large inputs.
We do not care about small steps

2. Big Omega Ω Notation

⊙ $T(n)$ is $\Omega(g(n))$ if there exists constant $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$

$$T(n) \geq c \cdot g(n)$$

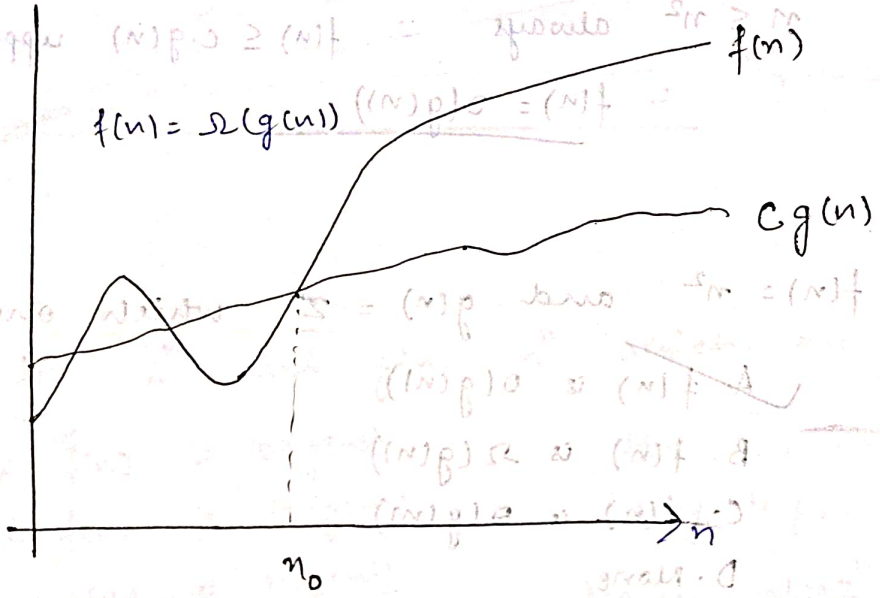
⊙ provides asymptotically lower bound

$T(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(T(n))$

i.e. $T(n) = \Omega(g(n)) \approx g(n) = O(T(n))$

$f(n) \leq c \cdot g(n) \Rightarrow g(n) \geq \frac{1}{c} \cdot f(n)$

$\Rightarrow g(n) \geq c' f(n)$ i.e. $g(n) = \Omega(f(n))$



3. Theta Θ Notation

$T(n) = \Omega(g(n))$ and $T(n) = O(g(n))$

$\Updownarrow$

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

i.e. There exist constants c_1, c_2 and n_0 such that $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Question :- $f(n) = n$ and $g(n) = n^2$ which one is true?

A. $f(n)$ is $O(g(n))$

B. $f(n)$ is $\Omega(g(n))$

C. $f(n)$ is $\Theta(g(n))$

D. None.

$n \leq n^2$ always $\therefore f(n) \leq c \cdot g(n)$ upper bound

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

Question :- $f(n) = n^2$ and $g(n) = 2^n$ which one is true?

A. $f(n)$ is $O(g(n))$

B. $f(n)$ is $\Omega(g(n))$

C. $f(n)$ is $\Theta(g(n))$

D. None.

$n^2 \leq 2^n$ always $\therefore f(n) = O(g(n))$

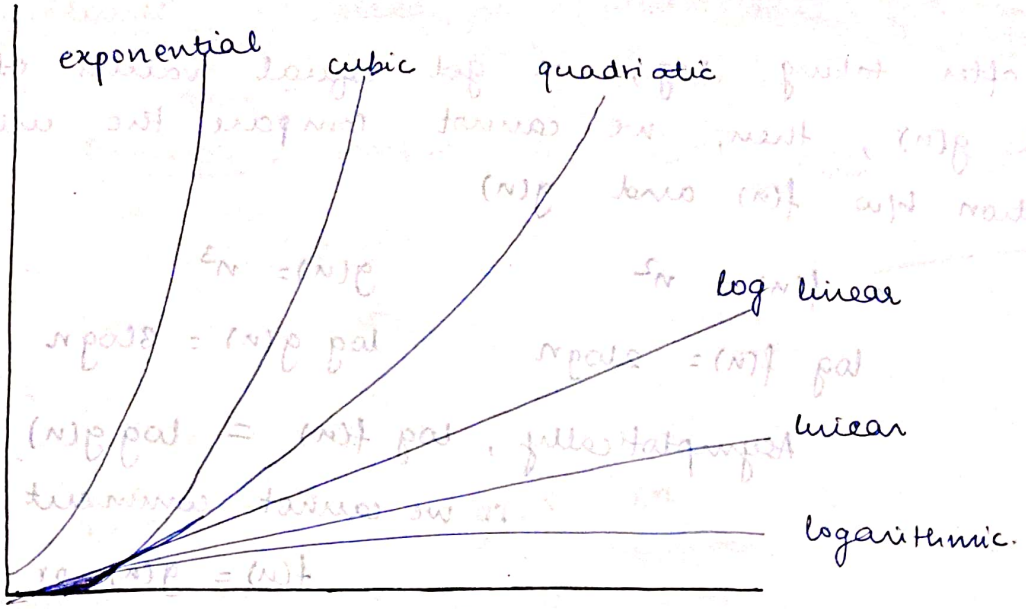
Remember

$$\log n < n < n \log n < n^2 < n^3 < 2^n < n! < n^n$$

When is it safe to ignore constants?

We can ignore when function can be reduced / expanded to the form

$c \cdot f(n) + d$ c & d can be ignored



Question :- $f(n) = 2^n$ and $g(n) = n^n$ which one is true?

A. $f(n)$ is $O(g(n))$
B. $f(n)$ is $\Omega(g(n))$
C. $f(n)$ is $\Theta(g(n))$
D. None.

$f(n) = 2^n$ $\quad g(n) = n^n$

Taking log,

$\log f(n) = n \log_2 2$ $\quad \log g(n) = n \log_2 n$

$= n$ $\quad = n \log_2 n$

$\therefore 1 < \log_2 n$

i.e. $2^n = O(n^n)$

Question :- $f(n) = n^2 \log n$ and $g(n) = n(\log n)^{10}$ which one is true?

A. $f(n)$ is $O(g(n))$
B. $f(n)$ is $\Omega(g(n))$
C. $f(n)$ is $\Theta(g(n))$
D. None

$f(n) = n^2 \log n$ $\quad g(n) = n(\log n)^{10}$

$n \log n$ $\quad (\log n)^{10}$

$n > (\log n)^9$

$\therefore f(n) = \Omega((\log n)^9)$

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

★ ⊙ If after taking log, we get equal values of $f(n)$ and $g(n)$, then, we cannot compare the initial relation b/w $f(n)$ and $g(n)$

e.g. $f(n) = n^2$ $\quad g(n) = n^3$

$\log f(n) = 2\log n$ $\quad \log g(n) = 3\log n$

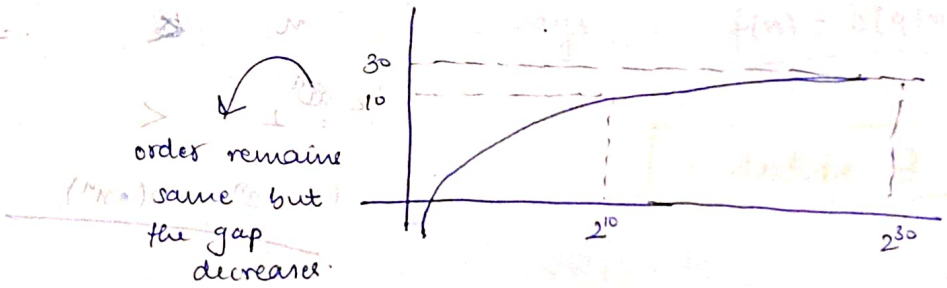
Asymptotically, $\log f(n) = \log g(n)$

∴ we cannot comment whether $f(n) = g(n)$ or $f(n) > g(n)$ or $f(n) < g(n)$

use some other method in this case.

⊙ for real numbers, log is order preserving.

ie: if $a > b$ then, $\log a > \log b$



log is increasing function

★ Workflow

⊙ Cancel out the common terms (multiply or divide)

⊙ Ignore constant only when $cf(n)+d$ form ie never ignore.

⊙ Take log and if asymptotically larger (or smaller)

⇒ original functions are also larger (or smaller) otherwise we can't say

all comparisons are asymptotic

$\log(f(n)) > \log(g(n)) \Rightarrow f(n) > g(n)$

$\log(f(n)) < \log(g(n)) \Rightarrow f(n) < g(n)$

$\log(f(n)) = \log(g(n)) \Rightarrow$ can't say anything

⊙ Sometimes, we need to assume some form of n
(ex 2^k, 2^{k^2}, 2^{100k}, 2^{400k^2}, 2^{2^k} etc.)

Example → compare b/w $n^{\frac{1}{\sqrt{\log n}}}$, $n^{1/100}$, $\sqrt{\log n}$.

Question $f(n) = n2^n$ $g(n) = 4^n$

$f(n)$		$g(n)$
$n2^n$		$(2^2)^n$
$n2^n$		2^{2n}
n	$<$	2^n

$\therefore$ $f(n) = O(g(n))$ $\quad n2^n = O(4^n)$

Question $f(n) = n^2 \log n$ $g(n) = n^{100}$

$f(n)$		$g(n)$
$n^2 \log n$		n^{100}
$\log n$	$<$	n^{99}

$\therefore$ $n^2 \log n = O(n^{100})$

i.e. $f(n) = O(g(n))$

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

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

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

$n^{\log n} = O(2^n)$

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

$f(n)$	$g(n)$
$n^{\log n}$	2^n
$\log n \log_2 n$	$n \log_2 2 \to 1$
$\log n \log_2 n$	n
$(\log n)^2$ $<$	n

$\therefore \quad n^{\log n} = O(2^n)$

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

$(\log n)^k \; \forall \; n^\epsilon$

$k >>>> \epsilon$

↳ no matter how big k is or how small ϵ is $(\log n)^k < n^\epsilon \quad \epsilon > 0$

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

$f(n)$	$g(n)$
2^n	$n^{\sqrt{n}}$
$n \log_2 2 \to 1$	$\sqrt{n} \log_2 n$
n	$\sqrt{n} \log_2 n$
$\sqrt{n}$ $>$	$\log_2 n$

(coz. $n^\epsilon > \log n \quad \forall \; \epsilon > 0$)

$\therefore \quad 2^n = \Omega(n^{\sqrt{n}})$

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

Question $f(n) = \log_2 n$ and $g(n) = \log_{10} n$

$f(n)$	$g(n)$
$\log_2 n$	$\log_{10} n$
$\frac{\log n}{\log 2}$	$\frac{\log n}{\log 10}$
$\log n$ $=$	$\log n$

$\log_2 n = \Theta(\log_{10} n)$

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

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

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

Question $f(n) = n^{\log_2 n}$ and $g(n) = n^{\log_{10} n}$

$f(n)$ $\quad$ $g(n)$

$n^{\log_2 n}$ $\quad$ $n^{\log_{10} n}$

$\log_2 n \; \log_2 n$ $\quad$ $\log_{10} n \; \log_2 n$

after taking log they are equal ∴ can't say anything

$\log_2 n = \log_{10} n$

wrong kar diye ना गलती

$\therefore n^{\log_2 n} = \Theta(n^{\log_{10} n})$

$\Rightarrow f(n) = \Theta(g(n))$

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

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

whichever power is greater, that is bigger function

∴ Now we compare

$\log_2 n$ $\quad$ $\log_{10} n$

$\frac{\log n}{\log 2}$ $\quad$ $\frac{\log n}{\log 10}$

$\because \log 10 > \log 2$

$\therefore \frac{\log n}{\log 10} < \frac{\log n}{\log 2}$

i.e. $\log_2 n > \log_{10} n$

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

$n^{\log_2 n} = \Omega(n^{\log_{10} n})$ or $g(n) = O(f(n))$

Question:- Arrange following functions in order of their asymptotic growth rate

$\log n$ $\quad$ $(\log n)^{10}$ $\quad$ $\log \log n$ $\quad$ $(\log(\log n))^{10}$

Let $n = 2^{2^k}$

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

$(\log n)^{10} = (\log 2^{2^k})^{10} = (2^k)^{10} = 2^{10k}$

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

$(\log(\log n))^{10} = k^{10}$ ...

$$k < k^{10} < 2^k < 2^{10k}$$

$$\therefore \log(\log n) < (\log(\log n))^{10} < \log n < (\log n)^{10}$$

Question Arrange following function in order by their asymptotic growth rate

$$2^{2^n}, \quad n!, \quad 4^n, \quad 2^n$$

Point to remember

$\log n! = n \log n$

$n! < n^n$

↳ Taking log both are $n \log n$. So, remember this relation (coz can't be derived by taking log)

Taking log,

A	B	C	D
$2^n \log_2 2$	$\log n!$	$n \log 4$	$n \log 2$
2^n	$n \log n$	n	n

$\therefore A > B > C, D$

For C and D —

C	D
4^n	2^n
$2^n \cdot 2^n$	2^n
$2^n > 1$	

$\therefore$ Relation is $A > B > C > D$.

i.e. $2^n < 4^n < n! < 2^{2^n}$

Question - Arrange following functions in increasing order of growth rate

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

Let $\log n = k$ i.e. $n = 2^k$

2^k $\quad k^2$ $\quad \sqrt{k}$ $\quad \log k$

In ascending order,

$$\log k < \sqrt{k} < k^2 < 2^k$$

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

4. Little-oh o Notation

$T(n)$ is $o(g(n))$ for any constant $c > 0$ and $n_0 > 0$ so that for all $n \geq n_0$

$$T_n < c\,g(n)$$

In big oh O, there exist constant c such that

$T(n) \leq c\,g(n)$

In little oh o, for all constants $c > 0$

$T(n) < c\,g(n)$

★ If there is function difference b/w $f(n)$ and $g(n)$ then, the relation is small oh

Question - $f(n) = 2^n$ $\quad g(n) = n!$ $\quad f(n) = o(g(n))$?

$2^n < cn!$ $\quad \forall c > 0$ $\quad \therefore$ True

5. Little omega ω Notation

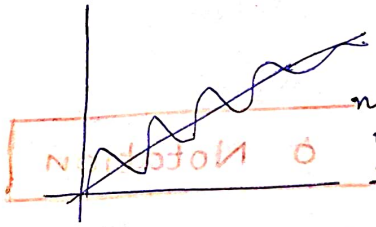
$T(n) \in \omega(g(n))$ for any constant $c > 0$ there is $n_0 \geq 0$ such that for all $n \geq n_0$

$$T(n) > cg(n)$$

$T(n) = \omega(g(n))$ if and only if $g(n) = o(T(n))$

Incomparibility

$n^{1+\sin n}$ is incomparable to n since $\sin n$ oscillates b/w -1 and 1.



no value for n_0 such that $\forall n \geq n_0$ $T(n)$ & $g(n)$ are greater or less than each other.

Properties of asymptotic notations

Transitivity

- $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$
- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \rightarrow f(n) = \Omega(h(n))$
- $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \rightarrow f(n) = \omega(h(n))$

Reflexivity

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

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

Symmetry

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Stirling's approximation

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Comparison b/w $n!$ and n^n —

$n!$: n^n

$\sqrt{2\pi n}\,\frac{n^n}{e^n}$: n^n

$\sqrt{2\pi}\cdot\frac{n^{1/2}}{e^n}$: 1

$\sqrt{2\pi}\cdot n^{1/2} < e^n$ [Multiplying by e^n]

$\therefore n^n > n!$ asymptotically.

Similarly it can be proved that

$\log n! = n \log n$ asymptotically.

$n! = o(n^n)$ $\quad$ $n! = O(n^n)$

$\log n! = \theta(n \log n)$

Question :- Compare the following functions

$n^{\sqrt{n}}$ (A) $\quad$ 2^n (B) $\quad$ $n^{\log n}$ (C) $\quad$ $n!$ (D)

Taking log,

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

~~let $\log n = k$ i.e. $n = 2^k$~~

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

$\therefore n^{\log n} < n^{\sqrt{n}} < 2^n < n!$

$A = n^{\sqrt{n}} \longrightarrow n^{1/2} \log n$

$B = 2^n \longrightarrow n$

$C \rightarrow n^{\log n} \longrightarrow (\log n)^2$

$D \longrightarrow n! \longrightarrow n \log n$

b/w A — B

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

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

$B > A$

$C < B$

$D > A, B, C$

$A > C$

$n^2 = O(n^3)$

n^2 ↓ one elem of set

$=$ ↓ $\in$

$O(n^3)$ → set of functions with comp n^3

$$O(n^3) = \{f(n) \mid f(n) \le c \cdot n^3\}$$

ideally, we should write

$n^2 \in O(n^3)$

but we write $n^2 = O(n^3)$

- abuse of notation
- mathematical convenience.

GATE CSE 1996

Which of the following is false?

A. $100n\log n = O\left(\frac{n\log n}{100}\right)$

B. $\sqrt{\log n} = O(\log\log n)$

C. if $0 < x < y$ then $n^x = O(n^y)$

D. $2^n \neq O(n^k)$

A.

$f(n) = 100n\log n$

$g(n) = \frac{n\log n}{100}$

Asymptotically,

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

$\therefore f(n) = O(g(n))$ ✓

B.

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

$g(n) = \log\log n$

let $\log n = k$

$f(n) \quad g(n)$

$k^{1/2} > \log k$

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

C.

$x < y$

$\therefore n^x < n^y$

$\therefore n^x = O(n^y)$ True ✓

D.

$f(n) = 2^n \quad g(n) = n^k$

$f(n) > g(n)$

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

GATE CSE 2000

Consider following functions

$f(n) = 3n^{\sqrt{n}}$

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

$h(n) = n!$

Tally log →

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

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

$f(n)$	$g(n)$	$h(n)$
$\sqrt{n}\log n$	$\sqrt{n}\log_2 n$	$n\log n$
$n^{\sqrt{n}}$	$2^{\sqrt{n}\log n}$	$h(n) > f(n)$
$n^{\sqrt{n}}$	$2^{\log n^{\sqrt{n}}} = n^{\sqrt{n}}$	$h(n) > g(n)$

GATE CSE 2001

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

$g(n) = n(\log n)^{10}$

$f(n)$	$g(n)$
$n^2 \log n$	$n(\log n)^{10}$
$n \log n$	$(\log n)^{10}$

$\therefore f(n) > g(n)$

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

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

$f(n) \neq O(g(n))$

GATE CSE 2003

I. $(n+k)^m = \Theta(n^m)$ — k & m are constants

II. $2^{n+1} = O(2^n)$

III. $2^{2n+1} = O(2^n)$

I. is true.

II. is true.

III. $\rightarrow 2^{2n+1} = 2 \cdot 2^{2n}$

$= 2 \cdot 4^n \neq O(2^n)$

false.

GATE CSE 2004

$f(n) = O(g(n))$ $\quad f(n) \leq c_1 g(n)$

$g(n) \neq O(f(n))$

$g(n) = O(h(n))$ $\quad g(n) \leq c_2 \cdot h(n)$

$h(n) = O(g(n))$ $\quad h(n) \leq c_3 \cdot g(n)$

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

$f(n) \leq$

$f(n) + g(n) = O(h(n) + h(n)) \rightarrow$ True.

$f(n) = O(h(n)) \rightarrow$ True.

$h(n) \neq O(f(n)) \rightarrow$ True.

$f(n) \cdot h(n) > g(n) \cdot h(n)$

$f(n) > g(n)$ — false.

GATE CSE 2008

$f(n) = 2^n$

$g(n) = n!$

$h(n) = n^{\log n}$

$f(n)$	$g(n)$	$h(n)$
2^n	$n!$	$n^{\log n}$
n	$n \log n$	$(\log n)^2$

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

$$h(n) < f(n) < g(n)$$

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

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

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

GATE CSE 2008

Arrange —

a. $n^{1/3} = n^{0.33}$

b. $e^n = e^n$

c. $n^{7/4} = n^{1.75}$

d. $n \log^9 n = n \log^n n$

e. $1.0000001^n = 1.000001^n$

$b > e$

$a < c$

$n(\log n)^9 \cdot n^{7/4}$

$(\log n)^9 < n^{3/4}$

$d < c$

~~$n(\log n)^9$~~ ~~$n^{1/3}$~~

$a < d < c$.

Ans → a d c e b

GATE CSE 2014

$f_1(n) = 2^n$

$f_2(n) = n^{3/2}$

$f_3(n) = n \log_2 n$

$f_4(n) = n^{\log_2 n}$

$f_2(n)$		$f_3(n)$
$n^{3/2}$		$n \log_2 n$
$n^{1/2}$	$>$	$\log_2 n$

$f_2 > f_3$.

f_2		f_4
$n^{3/2}$		$n^{\log_2 n}$
$\frac{3}{2} \log n$	$<$	$\log n \cdot \log n$

$f_1 > f_2$

$\therefore f_1 > f_2 > f_3$

f_1		f_4
2^n		$n^{\log_2 n}$
n	$>$	$(\log_2 n)^2$

$f_1 > f_4$

$f_4 > f_2$

$\therefore$ order is

$f_1 > f_4 > f_2 > f_3$

Analyzing loop complexity

Important summation formulae –

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

2) $1^2+2^2+3^2+\ldots\ldots+n^2 = \frac{n(n+1)(2n+1)}{6}$

Harmonic series.

3) $\sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots\ldots+\frac{1}{n} = \log n$

4) $\sum_{k=1}^{n} \log k = \log 1+\log 2+\log 3+\ldots\ldots+\log n = n\log n.$

★ `for(i=2; i<=n; i=i^2)`

`x=y+z.`

Suppose loop runs for k iterations.

Then time complexity = $O(k)$

	$i = 2$
1st	2^2
2nd	$(2^2)^2 = 2^4 = 2^{2^2}$
3rd	$(2^4)^2 = 2^8 = 2^{2^3}$
4th	$(2^8)^2 = 2^{16} = 2^{2^4}$

At k^{th} iteration, $i = 2^{2^k}$

$2^{2^k} = n$

$2^k = \log n$

$k = \log\log n.$

$\therefore$ TC = $O(\log_2 \log_2 n)$.

(2)

```
for (i = n; i >= 2; i = √i)
    x = y + z
```

$O(\log_2 \log_2 n)$

(3)

```
for (i = n^2; i >= 2; i = √i)
    x = y + z
```

$O(\log_2 \log_2 n^2)$

$= O(\log_2 2 \log_2 n)$

$= O(\log_2 \log_2 n)$

(4) ★

```
for (i = n/2; i <= n; i = i*2)
    x = y + z
```

$i = n/2$

$= \frac{n}{2} \times 2 = n$

$\therefore$ TC = $O(2) = O(1)$

3 Templates —

`i<n; i += a`	TC = $O(n/a)$
`i<n; i = i*a`	TC = $O(\log n)$
`i<n; i = i^2`	TC = $O(\log \log n)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=N; j=j+1)
        sum++
```

(1) $O(N^2)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=i; j=j+i)
```

(2) because of this loop sum divided by i times

$\therefore$ no of times inner loop run $= i/i = 1$

$\therefore O(N)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=i^2; j=j+i)
        sum++;
```

(3) $O(i^2/i) = O(i)$

For i=1	1
For i=2	2
⋮	
For i=N	N

$\therefore TC = O(1+2+\dots N)$

$= O(N^2)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=N; j=j+i)
        sum++
```

(4) Inner loop $\rightarrow N/i$ ★

$TC = O\left(\frac{N}{1} + \frac{N}{2} + \dots 1\right)$

$= O\left(N\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{N}\right)\right)$

$= O(N \log N)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=N; j=j*2)
        sum++;
```

(5) $\rightarrow O(\log N)$

$TC = O(N \log N)$

```
for (i=1; i<=N; i++)
    for (j=1; j<=i; j=j*2)
        sum++
```

(6) $\rightarrow \log i$

$TC = O(\log 1 + \log 2 + \log 3 + \dots \log$

$= O(N \log N)$

```
for (i=1; i<=N; i=i*2)      — logN
    for (j=1; j<=N; j++)    — N
        sum++;
```

(7) $TC = O(N \log N)$

```
for (i=1; i<=N; i=i*2)      — logN
    for (j=1; j<=i; j++)    — O(i)
        sum++;
```

(8)

$i=1$ — 1 time

$i=2$ — 2 time

$i=4$ — 4 time

~~$i=\log n$ — $\log n$ time~~

$i=8$ — 8 time

⋮

$i=k \rightarrow 2^k$

$1+2+4+8+\dots N^k = \frac{2^{k+1}-1}{2-1} = 2$

$TC = O(2^k)$

$= O(2^{\log N})$

$= O(N)$

(9)

```
for (i=1; i<=N; i=i*2)
    for (j=1; j<=i^2; j++)
        sum++
```

$\hookrightarrow O(i^2)$

$i=1$	—	1
$i=2$	—	$4 = 2^2$
$i=4$	—	$16 = 2^{2^2}$
$i=8$	—	$64 = 2^{2^3}$
$i=2^k$	—	2^{2k}

$\therefore TC = 1 + 2^2 + 2^4 + 2^6 + \dots + 2^{2k}$

$= \frac{1\cdot[(2^2)^k - 1]}{2^2-1} = \frac{2^{2k}-1}{3} = O(2^{2k})$

$k = \log N$

$\therefore TC = O(2^{2\log N}) = O(N^2)$

(10)

```
for (i=2; i<=n; i=i^2)        —— log log N times
    for (j=1; j<=n; j++)      —— N times
        for (k=1; k<=n; k=k+j)  —— N/j times.
            x=y+z
```

$j=1$	N times
$j=2$	$N/2$ times
$j=3$	$N/3$ times
⋮	
$j=N$	1 time

$\therefore N + \frac{N}{2} + \frac{N}{3} + \dots + 1$

$= N\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}\right)$

$= N \log N$

overall complexity $= \log\log N \cdot N \log N$

$= O(N \log N \log\log N)$

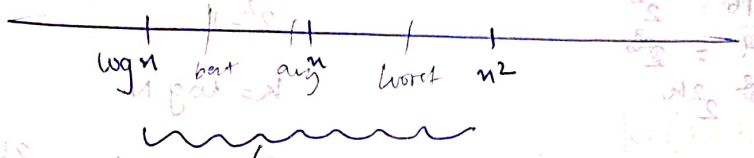
(11)

```
for (i=2; i<=n; i++){
    if (x%i==0)
        break;
    sum++;
}
```

$O(n)$ in worst case

$O(1)$ in best case

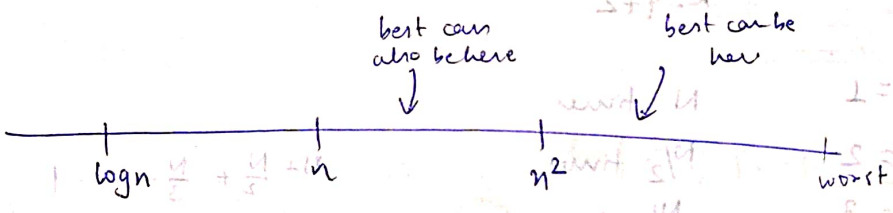
⊙ In worst case, if an algorithm takes $O(n^2)$ time, it means the algorithm is having time complexity of $O(n^2)$
↳ in all cases.



↳ worst case lies here
so best case will also lie $< n^2$.

∴ overall TC = $O(n^2)$.

⊙ If in worst case algorithm takes $\Omega(n^2)$, it does not mean that algorithm is having the time complexity of $\Omega(n^2)$



worst case

∴ No comments!

Time Complexity of Recursive Programs —

→ First we need to form Recurrence relation.

```
fun(n){
    if(n<=1) return 1;
    else return fun(n/2);
}
```

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

```
A(n){
    if(n<=1) return 1;
    else return A(n-1)+n;
}
```

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

```
fun(n){
    2fun(n/2) + n
}
```

$T(n) = T(n/2) + 1.$

⊙ The recurrence relation can be solved by using 3 methods

- ↳ Iteration method or Repeated substitution method.
- ↳ Recursive method
- ↳ Masters method.

1. Iterative / Substitution method —

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

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

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

$\vdots$

$= T(n-k) + k$

$n - k = 1 \quad \Rightarrow \quad k = n-1$

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

② $T(n) = T(n-1) + n \qquad T(1) = 1$

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

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

$\vdots$

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

$n - k = 1 \Rightarrow k = n-1$

$\therefore T(n) = T(1) + (2 + 3 + \dots n)$

$= 1 + 2 + 3 + \dots n = \frac{n(n+1)}{2} \qquad \underline{\theta(n^2)}$

③ $T(n) = \begin{cases} T(n/2) + n & n > 1 \\ 1 & n \le 1 \end{cases}$

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

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

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

$\vdots$

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

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

$\Rightarrow 2^k = n$

$\Rightarrow k = \log_2 n.$

$= T(1) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \dots \frac{1}{2^{k-1}}\right]$

decreasing GP. can be treated as constant.

$= 1 + n\left[1 \cdot \frac{1 - (1/2)^k}{1/2}\right] = 1 + 2n\left(1 - \frac{1}{2^k}\right)$

$= 1 + 2n\left(1 - \frac{1}{2^{\log_2 n}}\right) = 1 + 2n\left(1 - \frac{1}{n}\right)$

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

$T(1) + cn$

$1 + cn$

$\underline{\theta(n)}$

④ $T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n\log n & n>1 \\ 1 & n=1 \end{cases}$

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

$= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\log\frac{n}{2}\right] + n\log n$ $\quad \frac{n}{2^k} = 1$, $k = \log_2 n$

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

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

$= 2^3 T\left(\frac{n}{2^3}\right) + n\log\frac{n}{4} + n\log\frac{n}{2} + n\log n$

$\vdots$

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

~~$= 2^{\log_2 n} T(1) + cn$~~

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

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

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

$= 2^{\log_2 n} T(1) + n\left[\log_2 n \log_2 n - \frac{(\log_2 n)^2 - \log_2 n}{2}\right]$

$= n + n\left[(\log_2 n)^2 - \frac{(\log_2 n)^2}{2} + \frac{\log_2 n}{2}\right]$

$= n + n\left[\frac{(\log_2 n)^2}{2} + \frac{\log_2 n}{2}\right]$

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

⑤ $T(n) = \begin{cases} 2T(\frac{n}{2}) + n & n>1 \\ 1 & n=1 \end{cases}$

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

$= 2\left[2T(\frac{n}{2^2}) + \frac{n}{2}\right] + n$ $\quad k = \log_2 n$

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

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

$\vdots$

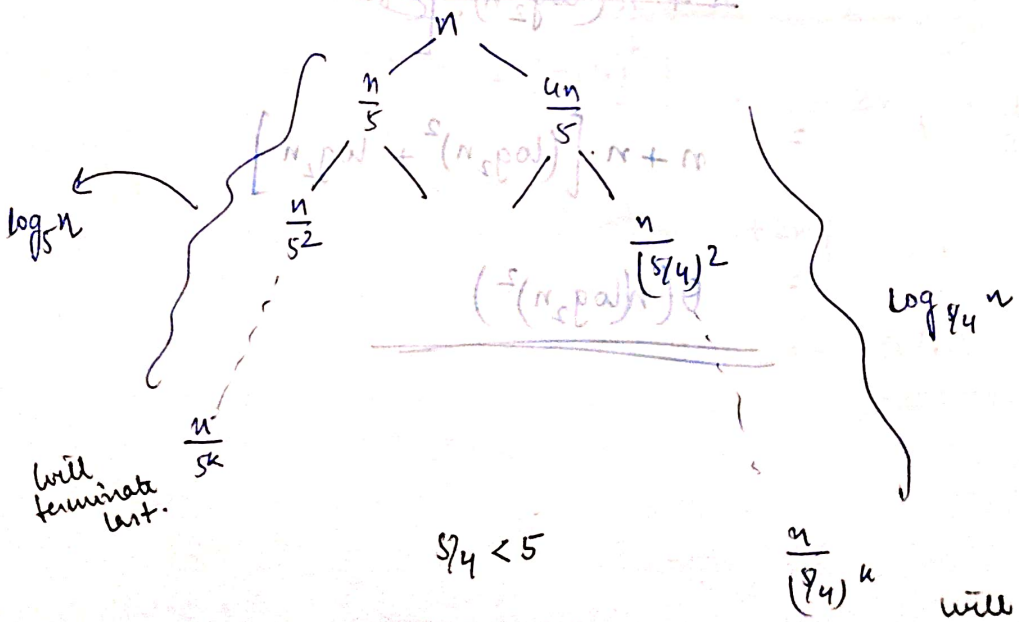
$= k\, 2^k T(\frac{n}{2^k}) + kn$

$= 2^{\log_2 n} T(1) + \log_2 n \cdot n = n + n\log_2 n$

$= \Theta(n\log_2 n)$

2. Tree method

① $T(n) = T(\frac{n}{5}) + T(\frac{4n}{5}) + n$



lower bound = $n \log_5 n$

upper bound = $n \log_{5/4} n$

$T(n) \leq n \log_{5/4} n$

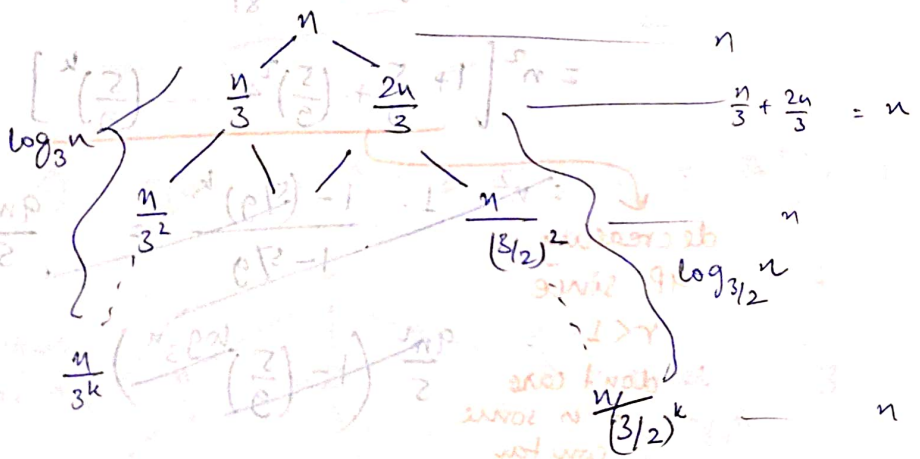
$T(n) = O(n \log_{5/4} n)$

$T(n) \geq n \log_5 n$

$T(n) = \Omega(n \log_5 n)$.

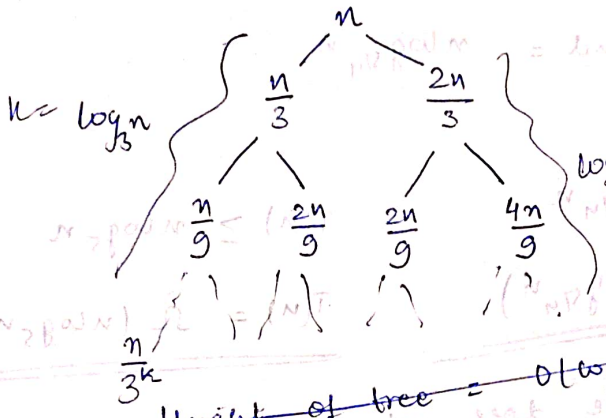
Since base does not matter in asymptotic comparisons, $T(n) = \Theta(n \log n)$

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



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

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



n^2

$\left(\frac{n}{3}\right)^2 + \left(\frac{2n}{3}\right)^2 = \frac{5n^2}{9}$

$\left(\frac{n}{9}\right)^2 + 2\left(\frac{2n}{9}\right)^2 + \left(\frac{4n}{9}\right)^2 = \frac{25n^2}{81}$

~~Height of tree = $\theta(\log n)$~~

~~$\therefore$ Tot. $T(n) = \log n \left[n^2 + \frac{5n^2}{9} + \frac{25n^2}{81} + \cdots \right]$~~

Lower bound —

$$T(n) = n^2 + \frac{5}{9}n^2 + \frac{25}{81}n^2 + \cdots \left(\frac{5}{9}\right)^k n^2$$

$$= n^2 \left[1 + \frac{5}{9} + \left(\frac{5}{9}\right)^2 + \cdots \left(\frac{5}{9}\right)^k \right]$$

$$= n^2 \cdot 1 \cdot \frac{1 - (5/9)^k}{1 - 5/9} = \frac{9n^2}{5}\left(1 - \left(\frac{5}{9}\right)^k\right)$$

decreasing GP since $r < 1$ $\therefore$ don't care it is some constant

$$= \frac{9n^2}{5}\left(1 - \left(\frac{5}{9}\right)^{\log_3 n}\right)$$

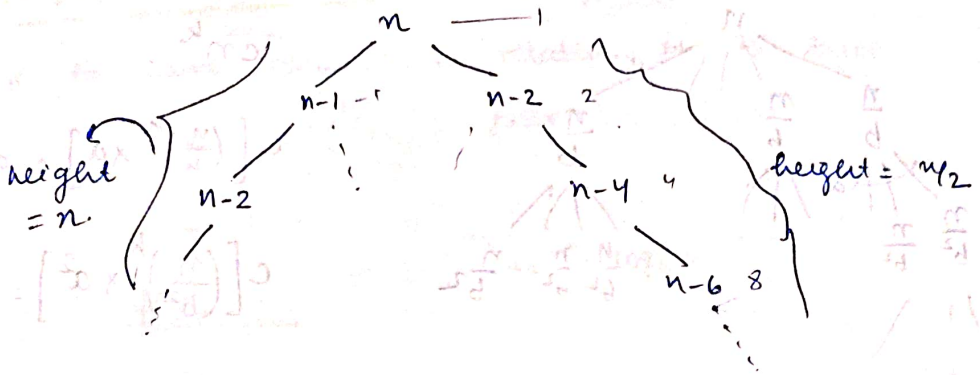
$$T(n) = n^2 c \cdot 0 = \Omega(n^2)$$

Remember this.

GP in ~~ord~~ asymptotic analysis —

$$1 + c + c^2 + \cdots + c^n = \sum_{i=1}^{n} c^i = \begin{cases} \theta(1) & \text{if } c < 1 \\ \theta(n) & \text{if } c = 1 \\ \theta(c^n) & \text{if } c > 1 \end{cases}$$

④ $T(n) = T(n-1) + T(n-2) + 1$



upper bound

$T(n) = 1 + 2 + 2^2 + 2^3 + \dots 2^n$

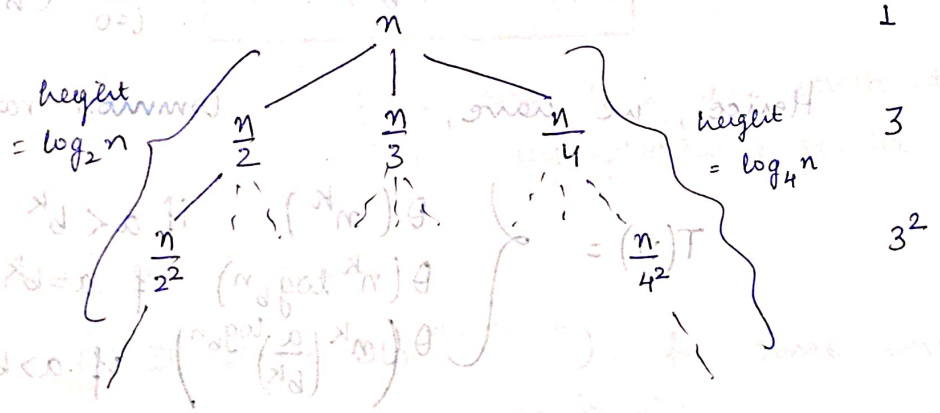
$\approx O(2^n)$ common ratio > 1 increasing G.P.

lower bound

$T(n) = 1 + 2 + 2^2 + 2^3 + \dots 2^{n/2}$

$= \Omega(2^{n/2})$ increasing G.P.

⑤ $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + 1$



upper bound —

$T(n) = 1 + 3 + 3^2 + \dots 3^{\log_2 n}$

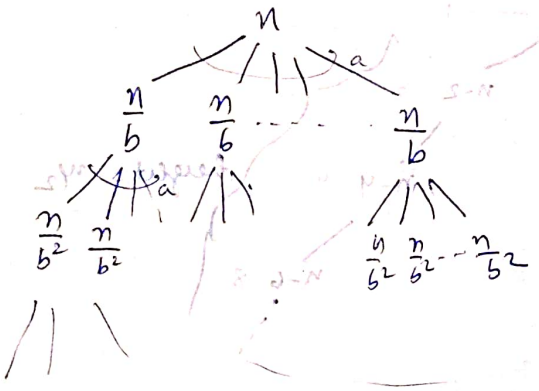
$= O(3^{\log_2 n}) = O(n^{\log_2 3})$ increasing G.P.

lower bound —

$T(n) = 1 + 3 + 3^2 + \dots 3^{\log_4 n}$

$= \Omega(3^{\log_4 n}) = \Omega(n^{\log_4 3})$

⑥ $T(n) = aT\left(\frac{n}{b}\right) + cn^k$



cn^k

$c\left[\left(\frac{n}{b}\right)^k \times a\right] = ac\left(\frac{n}{b}\right)^k$

$c\left[\left(\frac{n}{b^2}\right)^k \times a^2\right] = a^2c\left(\frac{n}{b^2}\right)^k$

At i^{th} level,

cost at that level $= a^i c\left(\frac{n}{b^i}\right)^k$

No. of levels $= \log_b n + 1$

$\therefore T(n) = \sum_{i=0}^{\log_b n} a^i c\left(\frac{n}{b^i}\right)^k$

$= n^k c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

Common ratio $= \frac{a}{b^k}$

Hence, we have,

$$T(n) = \begin{cases} \theta(n^k) & \text{if } a < b^k \\ \theta(n^k \log_b n) & \text{if } a = b^k \\ \theta\left(n^k \left(\frac{a}{b^k}\right)^{\log_b n}\right) & \text{if } a > b^k \end{cases}$$

Master Theorem

used to solve recurrence relation of the type

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

$f(n)$ vs $n^{\log_b a}$

$<$ polynomially	$\theta(n^{\log_b a})$
$=$ polynomially	$\theta(n^{\log_b a} \log n)$
$>$ polynomially	$\theta(f(n))$

⊙ The Master Theorem applies to recurrence relations of the following form –

$$\boxed{T(n) = aT(n/b) + f(n)} \quad \text{where}$$

$a \geq 1$ and $b > 1$ are constants and $f(n)$ is asymptotically positive function

There are 3 cases –

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then, $T(n) = \theta(n^{\log_b a})$

2. If $f(n) = \theta(n^{\log_b a})$ then $T(n) = \theta(n^{\log_b a} \log n)$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, then, $T(n) = \theta(f(n))$

Master Theorem

① $T(n) = 9T\left(\frac{n}{3}\right) + n$

$f(n) = n$ $\quad n^{\log_b a} = n^{\log_3 9} = n^2$

i.e. $f(n) = \cancel{\Theta(1)}\ O(n^{\log_b a - t})$

$\therefore T(n) = \Theta\left(n^{\log_b a}\right) = \underline{\Theta(n^2)}$

② $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$f(n) = n\log n$ $\quad n^{\log_b a} = n^{\log_4 3}$ (less than 1)

$\therefore T(n) = \underline{\Theta(n\log n)}$

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

$f(n) = n\log n$ $\quad n^{\log_b a} = n^{\log_2 2} = n$

$n\log n$ is not polynomially greater than n

$\therefore$ cannot apply master theorem.

④ $T(n) = 2T\left(\frac{n}{3}\right) + (\log(n))^2$

$f(n) = (\log(n))^2$ $\quad n^{\log_b a} = n^{\log_3 2}$ (>0 but <1)

$n^{\log_b a}$ is asymptotically greater than $(\log n)^2$

$\therefore T(n) = \underline{\Theta\left(n^{\log_3 2}\right)}$

Extended Master's Theorem

Recurrence relation of the form

$$T(n) = aT\left(\frac{n}{b}\right) + \theta\left(n^k \log^p n\right)$$

n = size of the problem

a = no. of subproblems in the recursion $a \geq 1$

n/b = size of each subproblem.

$b > 1$, $k >= 0$ and p is real number.

Case I:- If $k < \log_b a$ then $T(n) = \theta\left(n^{\log_b a}\right)$

Case II:- If $k = \log_b a$, then,

(a) if $p > -1$, then, $T(n) = \theta\left(n^{\log_b a} \cdot \log^{p+1} n\right)$.

(b) if $p = -1$, then, $T(n) = \theta\left(n^{\log_b a} \log\log n\right)$

(c) if $p < -1$, then, $T(n) = \theta\left(n^{\log_b a}\right)$.

Case III:- If $k > \log_b a$

(a) if $p >= 0$ then $T(n) = \theta\left(n^k \log^p n\right)$

(b) if $p < 0$ then $T(n) = \theta\left(n^k\right)$.

① $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{(\log n)^2}$

$\log_b a = \log_2 4 = 2 \qquad k = 2 \qquad p = -2$

$k = \log_b a$ and $p < -1$

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

② $T(n) = 9T\left(\frac{n}{3}\right) + (n \log n)^2$

$\log_b a = \log_3 9 = 2 \qquad k = 2 \qquad p = 2$

$\log_b a = k$ and $p > -1$

$\therefore T(n) = \theta(n^2 \log^3 n)$

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

$\log_b a = \log_2 2 = 1 \qquad k = 1 \qquad p = -3$

$\log_b a = k$ and $p < -1$

$\therefore T(n) = \theta(n)$

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

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

$\log_b a > k$

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

Recurrences not solvable by Master Theorem -

① $T(n) = \sqrt{n}\, T\left(\frac{n}{2}\right) + n$

$a = \sqrt{n}$ is not constant.

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

$b = \log n$ is not a constant

③ $T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + n^2$

$a = 1/2$ is not ≥ 1

④ $T(n) = 2T\left(\frac{4n}{3}\right) + n$

$b = 3/4$ is not > 1.

⑤ $T(n) = 3T\left(\frac{n}{2}\right) - n$

$f(n) = -n$ is not positive

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

violates regularity condition

⑦ $T(n) = T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + n$

a and b are not fixed.

Change of variable

① $T(n) = T(\sqrt{n}) + \log n$

↳ if it is of the form n^k then, use change of variable method by putting $n = 2^m$.

$T(n) = T(\sqrt{n}) + \log n$

Let $n = 2^m$. Then,

$T(2^m) = T(2^{m/2}) + \log 2^m$

$\Rightarrow T(2^m) = T(2^{m/2}) + m$

Let $S(m) = T(2^m)$. Then,

$S(m) = S(m/2) + m$

$\log_b a = \log_2 1 = 0$

$\therefore S(m) = \theta(m)$

i.e. $T(2^m) = \theta(m)$

$\Rightarrow T(n) = \theta(\log n)$ $\left[\because n = 2^m, m = \log n\right]$

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

Let $n = 2^m$. Then,

$$T(2^m) = T(2^{m/2}) + 2^{m/2}$$

Let $S(m) = T(2^m)$.

$$S(m) = S(m/2) + 2^{m/2}$$

$$\log_b a = \log_2 1 = 0$$

$$\therefore S(m) = \Theta(2^{m/2})$$

$$\text{Now } T(2^m) = S(m) = \Theta(2^{m/2})$$

$$\therefore T(n) = \Theta(2^{\log_2 \sqrt{n}}) = \Theta(2^{\log \sqrt{n}})$$

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

(3) $T(n) = T(\sqrt{n}) + \log \log n$

Let $n = 2^m$ $\quad m = \log_2 n$

$$T(2^m) = T(2^{m/2}) + \log \log 2^m \Rightarrow T(2^m) = T(2^{m/2}) + \log m$$

$\log_b a = \log$. Let $S(m) = T(2^m)$

$$\therefore S(m) = S(m/2) + \log m$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 < \log m$$

$$\therefore S(m) = \Theta((\log m)^2) \qquad p > -1$$

$$T(2^m) = S(m) = \Theta((\log m)^2)$$

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

(3) $T(n) = T(\sqrt{n}) + n^2$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = T(2^{m/2}) + (2^m)^2$

$\Rightarrow T(2^m) = T(2^{m/2}) + 2^{2m}$

Let $S(m) = T(2^m)$

$S(m) = S(m/2) + 2^{2m}$

$m^{\log_b a} = m^{\log_2 1} = n^0 < 2^{2m}$

$\therefore S(m) = \theta(2^{2m}) = \theta(4^m)$

$T(2^m) = \theta(4^m)$

~~$\Rightarrow T(n) = \theta(\log_2 4^m) = \theta(m \log_2 4) = \theta(2m) =$~~

~~$T(2^m) =$~~ $T(n) = \theta(4^{\log_2 n}) = \theta(2^{2\log_2 n})$.

$= \theta(2^{\log_2 n^2}) = \theta(n^2)$

(4) $T(n) = 2T(\sqrt{n}) + (\log n)^3$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 2T(2^{m/2}) + m^3$

Let ~~$f(m)$~~ $S(m) = T(2^m)$

$\therefore S(m) = 2S(m/2) + m^3$

$m^{\log_b a} = m^{\log_2 2} = m < m^3$

$\therefore S(m) = \theta(m^3)$

$T(n) = \theta((\log_2 n)^3)$

(5) $T(n) = 4T(\sqrt{n}) + (\log n)$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 4T(2^{m/2}) + m^2$

Let ~~$T(m)$~~ $T(2^m) = S(m)$. Then,

$S(m) = 4S(m/2) + m^2$

$m^{\log_b a} = m^{\log_2 4} = m^2 = m^2$

$\therefore S(m) = \theta(m^2 \log m)$ $[\because p \geq -1]$

$\therefore T(n) = \theta((\log_2 n)^2 \log\log n)$

(6) $T(n) = 12T(n^{1/3}) + (\log n)^2$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 12T(2^{m/3}) + m^2$

Let $S(m) = T(2^m)$

$S(m) = 12S(m/3) + m^2$

$m^{\log_b a} = m^{\log_3 12} > m^2$

$\therefore S(m) = \theta(m^{\log_3 12})$

$T(n) = \theta((\log n)^{\log_3 12})$

GATE 2006

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

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 2T(2^{m/2}) + 1$

Let $T(2^m) = S(m)$. Then,

$S(m) = 2S(m/2) + 1$

$m^{\log_b a} = m^{\log_2 2} = m^1 > 1$

$\therefore S(m) = \Theta(m)$

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

(7) $T(n) = \sqrt{n}\, T(\sqrt{n}) + 100n$

$\Rightarrow \frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 100$

Let $G(n) = \frac{T(n)}{n}$

GATE 2024

$G(n) = G(\sqrt{n}) + 100$

Let $n = 2^m \qquad m = \log_2 n$

$G(2^m) = G(2^{m/2}) + 100$

Let $G(2^m) = S(m)$

$S(m) = S(m/2) + 100$

$m^{\log_b a} = m^{\log_2 1} = 1 = \Theta(1)$

$\therefore S(m) = \Theta(\log m)$

$G(2^m) = \Theta(\log m)$

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

$\Rightarrow T(n) = n\, G(n) = \Theta(n \log \log n)$

Divide and Conquer Algorithms

1. Maximum element in an array —

```
maximum (a, l, r) {
    if (r == l)
        return a[l]
    m = (l+r)/2;
    u = maximum (a, l, m);
    v = maximum (a, m+1, r);
    return (max(u, v));
}
```

Base case

If there is only one element then maximum is element itself.

} Base case

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

$$= \theta(n).$$

2. Search in an array —

```
search (a, i, j, value) {
    if (i == j) {
        if (a[i] == value)
            return true;
        else
            return false;
    }
    mid = (i+j)/2;
    return search (a, i, mid, value) ||
           search (a, mid+1, j, value);
}
```

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

$$= \theta(n).$$

3. Dumb sort —

```
mysort (a, i, j) {
    if (i == j)
        return;
    mysort (a, i+1, j);
    for (k = i+1 to j) {
        if (a[k-1] > a[k]) {
            swap (a[k-1], a[k]);
        }
    }
}
```

Base case: array with 1 element is always sorted

sort elements from i+1, j

Put a[i] in the correct position in sorted array (insertion sort method).

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

$= O(n^2)$

4. Merge sort —

```
mergesort (a, i, j) {
    if (i == j)
        return;
    mid = (i+j)/2
    mergesort (a, i, mid)
    mergesort (a, mid+1, j)
    merge (a, i, mid, j)
}
```

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

$= \theta(n \log n)$

```
merge (a, i, mid, j) {
    p = i, q = mid+1, k = 0
    while (p <= mid && q <= j) {
        if (a[p] < a[q]) {
            b[k] = a[p]; p++;
        }
        else {
            b[k] = a[q]; q++;
        }
        k++;
    }
    if (p > mid) copy remaining elements of 2nd half.
    if (q > j) copy remaining elements of 1st half.
    <copy b to a>.
}
```

Number of comparisons being done in merge sort —

In worst case $(m+n-1)$ comparisons are done

In best case, $\min(m,n)$ comparisons are done.

Modified merge sort divides the array into 3 subarrays of equal size. What is the no. of comparisons required ~~for~~ in merging the 3 subarrays.

3 subarrays of size $\frac{n}{3}$, $\frac{n}{3}$, $\frac{n}{3}$

worst case:

Merging 1st & 2nd subarray —

comparisons required = $\frac{n}{3} + \frac{n}{3} - 1 = \frac{2n}{3} - 1$

Merging the array of size $\frac{2n}{3}$ with 3rd subarray —

Comparisons required = $\frac{2n}{3} + \frac{n}{3} - 1 = n - 1$

∴ Total no of comparisons required = $\frac{2n}{3} - 1 + n - 1$

$= \frac{5n}{3} - 2 \approx O(n)$.

Recurrence Relation:

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

Q. An array $A[1 \ldots n]$ is bitonic if there exists a t such that $A[1 \ldots t]$ is increasing and $A[t+1 \ldots n]$ is decreasing. Give an efficient algorithm to sort a bitonic array A.

1. Reverse $A[t+1 \ldots n]$ $\quad O(n)$
2. Apply merge algorithm.

$O(n)$ time

Q. An integer array A is k even-mixed if there are exactly k even integers in A and the odd integers in A appear in sorted order. Given a k even-mixed array A containing n distinct integers for $k = \frac{n}{\log n}$.

$\theta(n)$ { 1. Copy all even numbers in array1 (unsorted)
2. Copy all odd integers in array2 (sorted order)

$\theta(k \log k)$ { 3. Sort array1

$\theta(n)$ { 4. Merge array1 and array2.

$\therefore TC = n + k\log k + n$

If $k = \frac{n}{\log n}$, $\quad TC = n + \frac{n}{\log n}\log\left(\frac{n}{\log n}\right)$

$\because n + \frac{n}{\log n} \cdot \log n - \frac{n}{\log n} \cdot \log\log n$

$= \theta(n)$

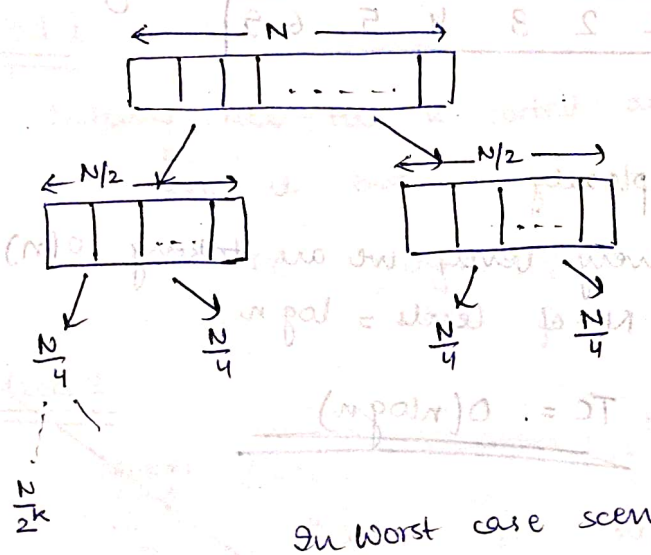
Q. Suppose that you have an integer of length N consisting of alternating B's and A's starting with B.

Ex → BABABABA

How many comparisons does merge sort take to sort the array as a function of N? You may assume N is a power of 2.

N elements in array

Let $N = 2^k$.



At k^{th} level —

In worst case scenario, no. of comparisons of merge sort

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

AAAA BBBB AAAA BBBB

AAAAAAAABBBBBBBB

only $\frac{3n}{4}$ comparisons are needed to merge

Last 4 Bs are directly merged without comparisons

$$\therefore T(n) = 2T\left(\frac{n}{2}\right) + \frac{3n}{4}$$

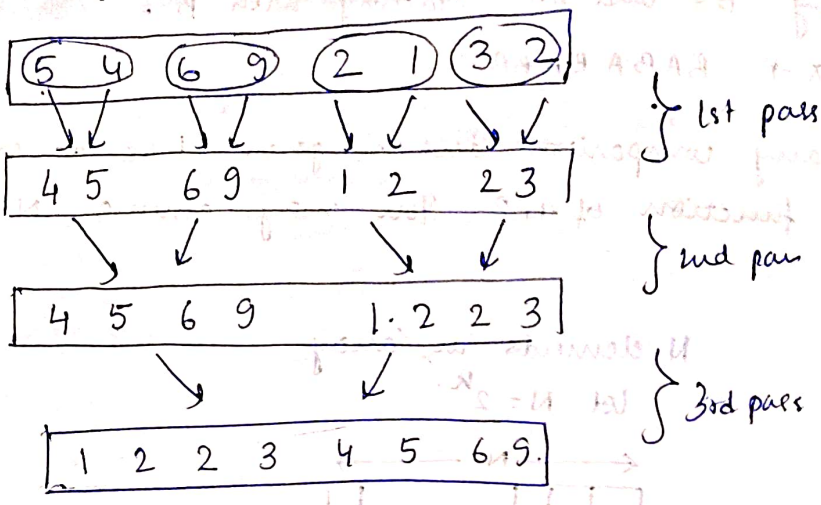
$T(n)$ → no. of comparisons for array of length N

$$= \frac{3}{4} N \log_2 N$$

Iterative merge sort –

Bottom up merge sort.

Straight Two way merge sort.



Time complexity –

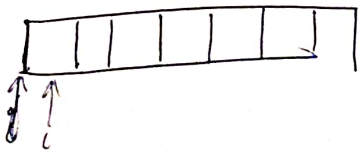
At every level, we are taking $O(n)$

No. of levels = $\log n$

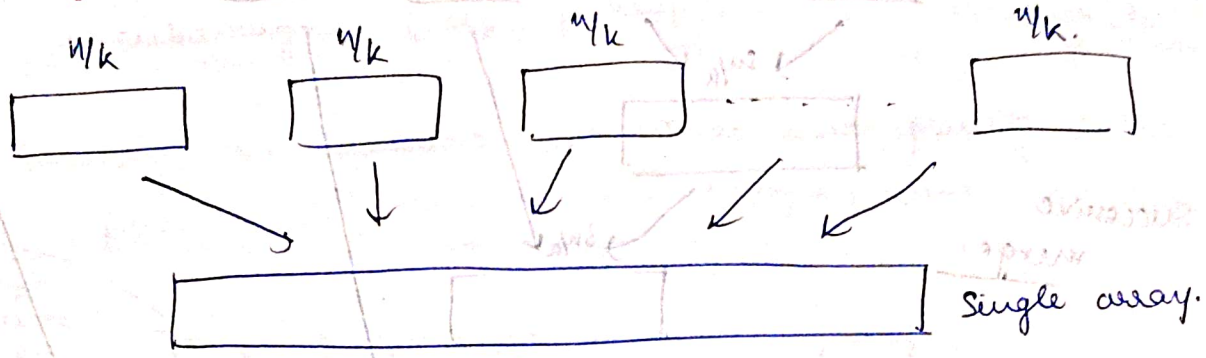
$\therefore$ TC = $O(n \log n)$

Bottom up merge sort code

```
for( width=1; width < a.length; width = 2*width){
    for( i=0; i< a.length; i = i+ 2*width) {
        int left, right, middle;
        left = i;
        middle = i+width-1;
        right = i+2*width-1
        merge (a, left, right, middle);
    }
}
```



5. Merge k sorted arrays into single array



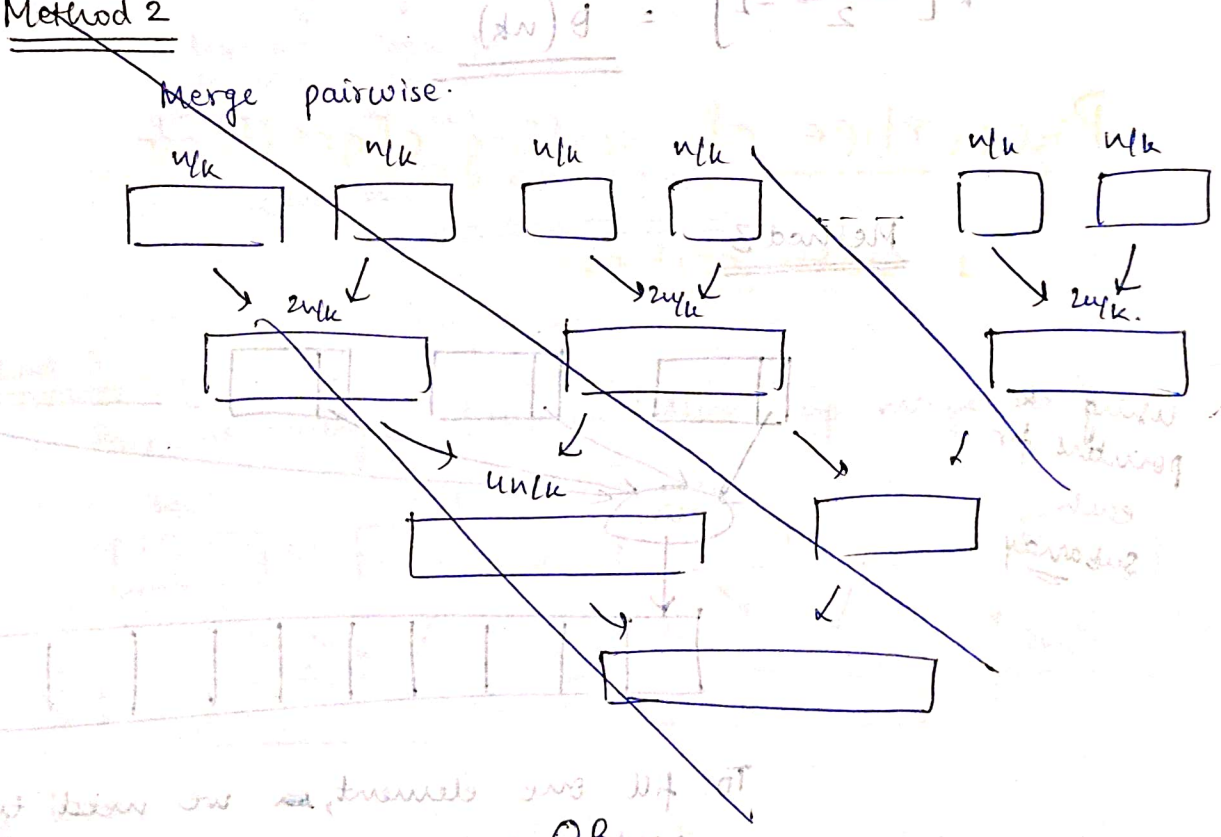
Method 1

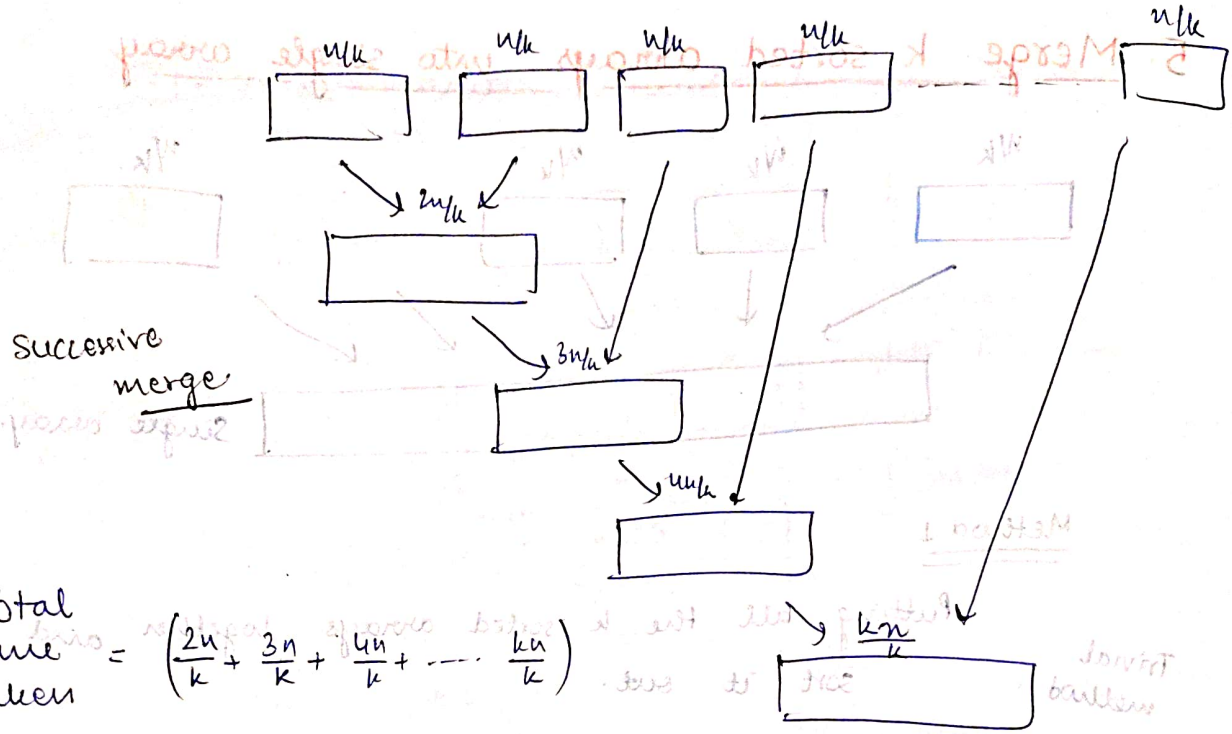
Trivial method

Putting all the k sorted arrays together and sort it out.

TC = nlogn

Method 2



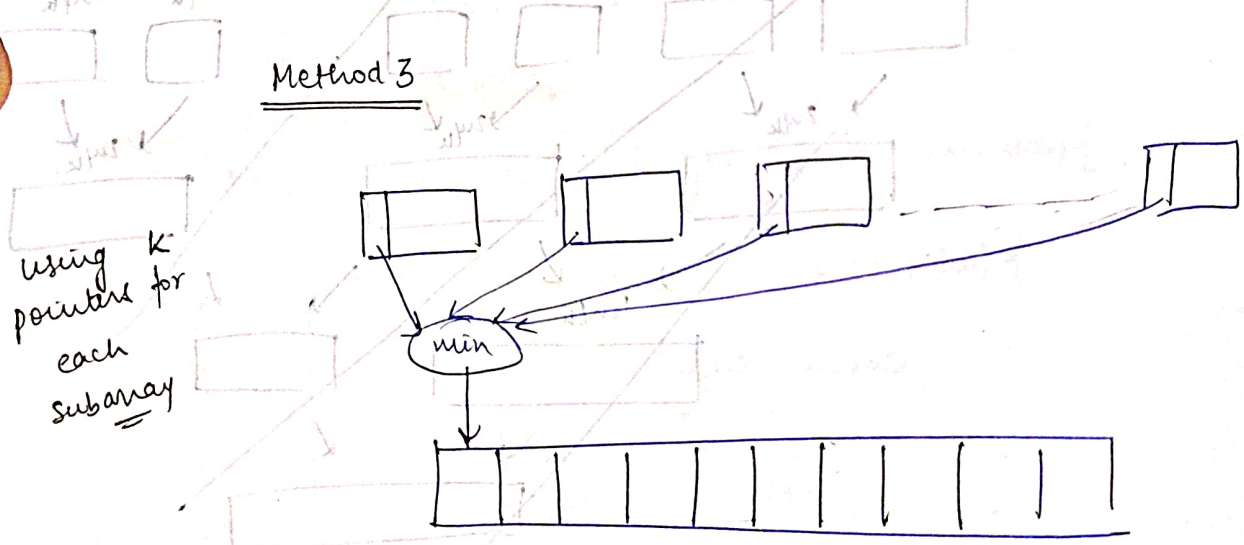


Total time taken $= \left(\frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \cdots \frac{kn}{k}\right)$

$= \frac{n}{k}\left[2+3+4+5+\ldots k\right]$

$= \frac{n}{k}\left[\frac{k(k+1)}{2} - 1\right] = \underline{\underline{\Theta(nk)}}$

Method 3



To fill one element, we need to ~~compare~~ find minimum among the k arrays.
This is done in $O(k)$ time

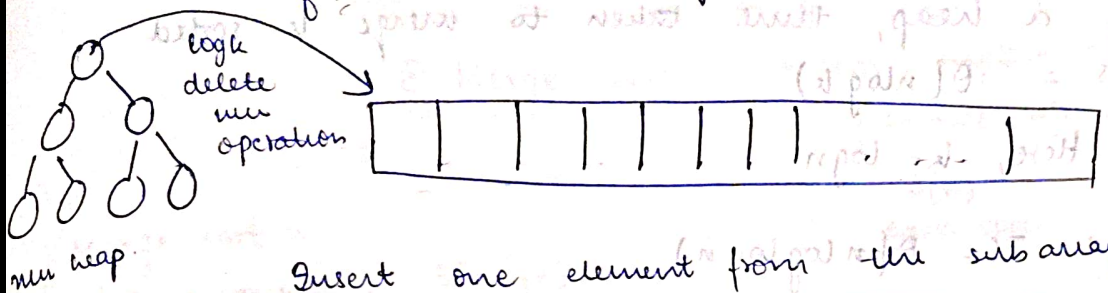
∴ To merge all the subarrays (n elements),

$O(nk)$ time req.

Method 4

Build min heap of minimum elements from each subarray. (size of heap = k) —— O(k).

Deleting an element from heap and placing it in final sorted array —— O(log k) time



Insert one element from the subarray to the heap (log k time)

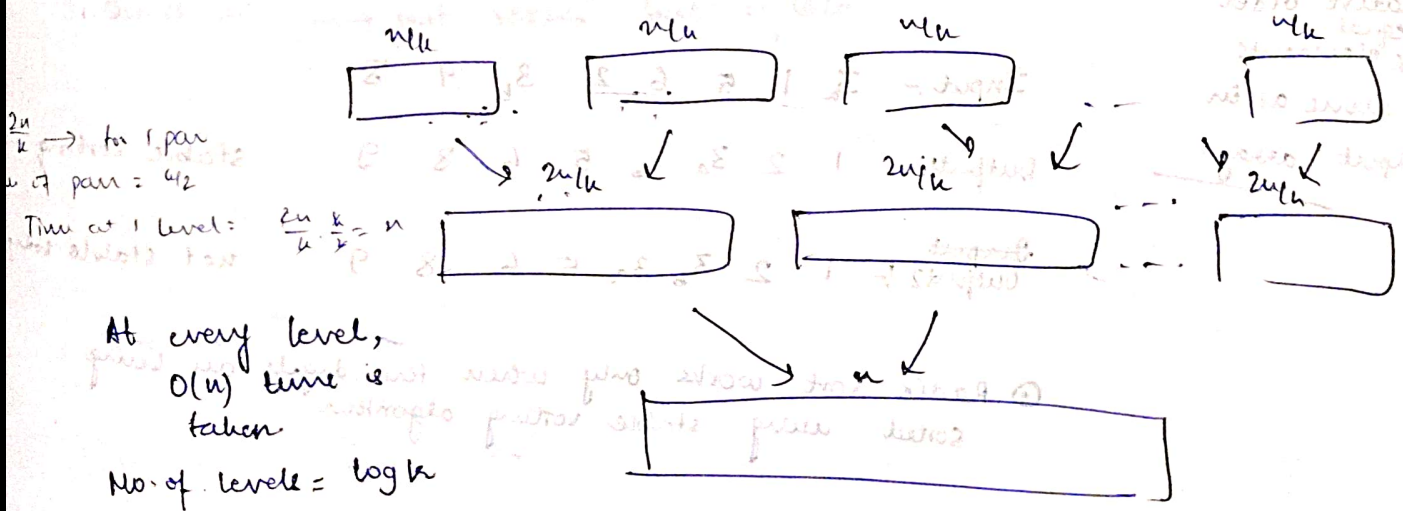
∴ To place 1 element in final sorted array, 2 log k time required

- log k for deletion from heap
- log k for insertion in heap

∴ Total time = $O(n \log k)$

Method 5

Pairwise merge (as in bottom up merge sort)



$\frac{2n}{k}$ → for 1 pair

no of pair = $n/2$

Time at 1 level: $\frac{2n}{k} \cdot \frac{k}{2} = n$

At every level, O(n) time is taken

No. of levels = log k

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

Question

Suppose we have $\log n$ sorted lists each of size $\frac{n}{\log n}$

Time taken to merge all the sorted lists into a single list using min heap.

Given a heap, time taken to merge k sorted lists = $O(n \log k)$

Here, $k = \log n$

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

Properties of sorting algorithms

1. Stable sorting –

⊙ A sorting algorithm is stable if elements with the same key appear in the o/p array in the same order as they do in the input array.

Relative order of equal elements is same as in input array.

Input :- 3_a 1 5 6 2 3_b 9 8

Output1 :- 1 2 3_a 3_b 5 6 8 9 stable sorting

~~Input~~ Output2 :- 1 2 3_b 3_a 5 6 8 9 not stable sorting

⊙ Radix sort works only when the digits are being sorted using stable sorting algorithm

⊙ Merge sort is stable sorting algorithm

Quick sort is not stable → nai.

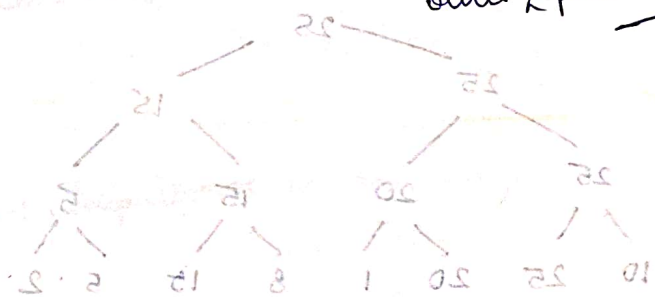
2. In Place algorithm –

A sorting algorithm is said to be an inplace sorting algorithm if the amount of extra space required by the algorithm is $\Theta(1)$. i.e. the amount of extra space is independent of the size of array.

⊙ Merge sort ⟶ not inplace algorithm.
(because of merge step)
↳ we create a new array
$O(n)$ extra space.

Merge sort
- ↳ Time complexity = $\Theta(n \log n)$
- ↳ Stable algorithm
- ↳ Not in place algorithm
- ↳ Space complexity = $O(n)$.

⟶ we don't care about the stack depth in recursion.
(even if algo uses recursion.
i.e. even if stack depth is present, algo can be inplace if no extra space required).



Merge sort ⟶ stable but not inplace

Quick sort ⟶ not stable but inplace.

Maximum and minimum of Numbers

1. Maximum in an array

```
max = a[0]
for (i = 1 to n-1) {
    if (a[i] > max) {
        max = a[i]
    }
}
```

Skewed Tournament

No. of comparisons = $n-1$

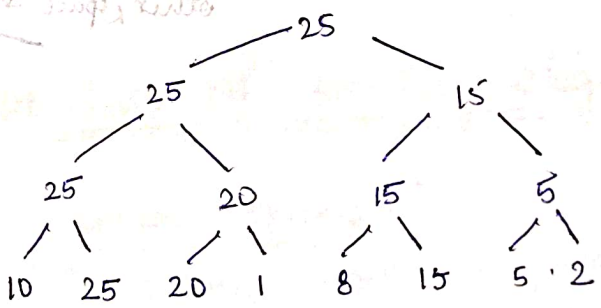
Minimum in an array

```
min = a[0];
for (i = 1 to n-1) {
    if (a[i] < min) {
        min = a[i];
    }
}
```

No. of comparisons = $n-1$

Method 2: Tournament method (Balanced tournament)

pairwise comparison is done.



∴ No. of comparisons required to find max. using tournament method $= n-1$

No. of comparisons

$= \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots \frac{n}{2^k}$

(where $2^k = n$)

$= \frac{n}{2}\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^{k-1}}\right]$

$= \frac{n}{2} \cdot \frac{1-\left(\frac{1}{2}\right)^k}{1-\frac{1}{2}}$

$= n\left(1 - \frac{1}{2^k}\right) = n\left(1 - \frac{1}{n}\right)$

$= n-1$ comparisons

2. Find maximum and second maximum in an array

```
if (a[0] > a[1]){
    max = a[0];
    secondmax = a[1];
}
else{
    max = a[1];
    secondmax = a[0];
}
```

} 1 comparisons

```
for(i=2 to n-1){
    if (a[i] > max){
        secondmax = max;
        max = a[i];
    }
    else if (a[i] > secondmax){
        secondmax = a[i];
    }
}
```

} 2(n-2) + 1 comparisons.
= 2n-3 comparisons

Best case

↳ 1 + n-2 = n-1 comparisons

Worst case

↳ 1 + 2(n-2) = 2n-3 comparisons

Method 2:- using tournament method

Key observation:- 2nd largest can only be defeated by the largest element.

2nd maximum is a sibling of maximum somewhere.

Maximum element has $\log_2 n$ siblings

2nd largest element is one of those $\log_2 n$ siblings of largest no.

Step 1:- Find maximum using tournament method
= $n-1$ comparisons.

Step 2:- Find 2nd maximum among $\log_2 n$ siblings
= $\log_2 n - 1$ comparisons

Total no. of comparisons = $n-1+\log_2 n - 1 = n+\log_2 n - 2$

3. Finding minimum and maximum in an array —

```
max = min = a[1];
for( int i=2 ; i<n ; i++){
    if (a[i] > max) max = a[i];
    else if (a[i] < min) min = a[i];
}
```

2(n-1) comparisons.

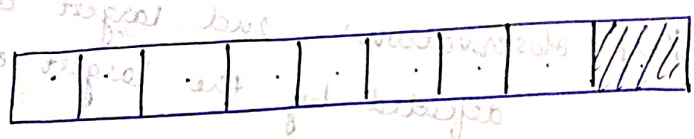
Total no. of comparisons —

Best case = $n-1$

Worst case = $2(n-1)$

Avg. case = $\left[\frac{1}{2} \times (1) + \frac{1}{2}(2)\right]_n = \frac{3n}{2}$.

Method 2:



```
if (a[0] > a[1])
    max = a[0]
    min = a[1]
else
    max = a[1]
    min = a[0]
```

~~for every pair~~
Pairwise compare the remaining $n-2$ elements.
No. of pairs formed = $\frac{n-2}{2}$

Largest element in pair is compared against max
Smaller element in pair is compared against min.

∴ For initialization, 1 comparison is required

For remaining $\frac{n-2}{2}$ pairs — 3 comparisons are done

- 1 for finding smaller and larger of the 2
- 1 for comparing larger with max
- 1 for comparing smaller with min.

Total number of comparisons required $= 1 + \frac{n-2}{2} * 3$

$= 1 + 3\left(\frac{n}{2} - 1\right)$

$= \frac{3n}{2} - 2$ if n is even.

When n is odd, min and max are initialized to $a[0]$.

$\frac{n-1}{2}$ pairs are formed and for each pair, 3 comparisons are made.

∴ No. of comparisons $= 3.\left(\frac{n-1}{2}\right) = \frac{3n}{2} - \frac{3}{2}$; if n is odd.

∴ No. of comparisons $= \left\lceil \frac{3n}{2} \right\rceil - 2.$

Question

Let p be the number of comparisons to find min and max using best algorithm.

(a) $p \leq \left\lceil \frac{3n}{2} \right\rceil - 2$

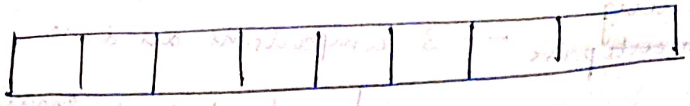
(c) $p \leq \left\lfloor \frac{3n}{2} \right\rfloor$

(b) $p \leq \left\lceil \frac{3n}{2} \right\rceil$

(d) $p \leq \log n$

$p = \left\lceil \frac{3n}{2} \right\rceil - 2.$

Method 3 : Tournament method



Compare elements pairwise

($\frac{n}{2}$ elements ⟶) Greater element in each pair is placed in List1

($\frac{n}{2}$ elements ⟶) Smaller element in each pair is placed in List2

Minimum from List2 ⟶ $\frac{n}{2} - 1$ comparisons

Maximum from List1 ⟶ $\frac{n}{2} - 1$ comparisons.

Total no of comparisons = $\frac{n}{2} + \frac{n}{2} - 1 + \frac{n}{2} - 1 = \frac{3n}{2} - 2$ if n is even

if n is ~~even~~ odd, max and min are found among $(n-1)$ elements and 2 comparisons are done to compare last element with min & max

$\therefore$ No. of comparisons = $\frac{3}{2}(n-1) - 2 + 2 = \frac{3n}{2} - 1.5$ if n is odd

$\therefore$ No. of comparisons by tournament method = $\left\lceil \frac{3n}{2} \right\rceil - 2$.

Summary

No. of comparisons required to find largest & second largest element = $n + \log_2 n - 2$

No. of comparisons required to find largest and smallest element = $\left\lceil \frac{3n}{2} \right\rceil - 2$

Counting inversions

We say that two indices $i<j$ form an inversion if $a[i] > a[j]$

0	1	2	3	4
2	4	1	3	5

Inversions –

2 – 1
4 – 1
4 – 3

Ques – How many inversions are there in 3, 2, 5, 7, 6, 4, 2 ?

(0 1 2 3 4 5 6)

3-2, 3-2
5-4, 5-2
7-6, 7-4, 7-2
6-4, 6-2
4-2

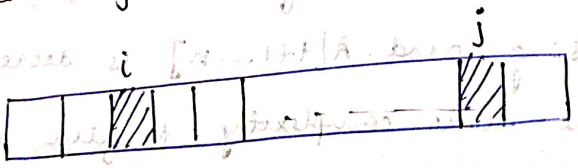
Ques

Largest possible number of inversions that a 6 element array can have:

$${}^6C_2 = \frac{3\cancel{6} \times 5}{\cancel{2}} = 15$$

Ques

$T[0 \ldots n-1]$ is a vector of integers.
If there is an inversion (i, j) in T, then, T has atleast $j-i$ inversions.



$(i, j) \rightarrow$ inversion
$\Rightarrow a[i] > a[j]$.

For indices k b/w i and j –

if $a[k] < a[j]$
i, k, j is inversion

if $a[k] > a[j]$ & $a[k] < a[i]$
k, i is inversion

if $a[k] > a[i]$
k, i is inversion

∴ In every case inversion exists
i.e. atleast $j-i$ inversions

Counting inversions

Brute force method –

Compare every pair of elements

$O(n^2)$ complexity.

Using Merge sort (Divide and Conquer)

$O(n \log n)$ time

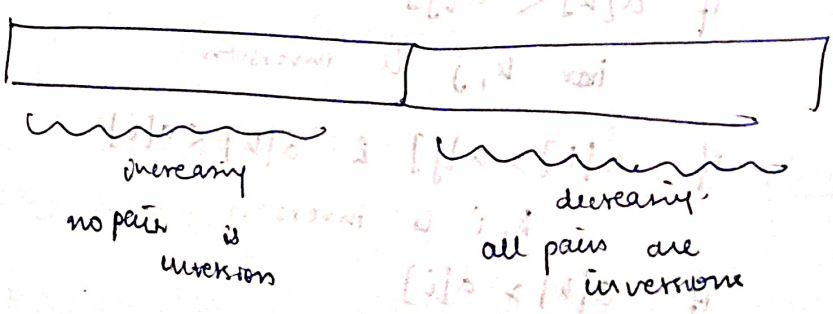
```
merge ( l, mid, r, ar)
    int count = 0
    L1 = ar[l ... mid]
    L2 = ar[mid+1 .... r]
    int p=0, q=0; int r=0
    while ( p<= mid-l && q<= r-mid-1)
        if (L1[p] <= L2[q])
            ar[r++] = L1[p++];
        else
            count += m-l-p;
            ar[r++] = aL2[q++];
```

Question

An array A[1...n] is bitonic if there exists a t such that A[1...t] is increasing and A[t+1...n] is decreasing.

What will be the time complexity to find all the inversions in bitonic array.

Find t — $O(n)$ time.

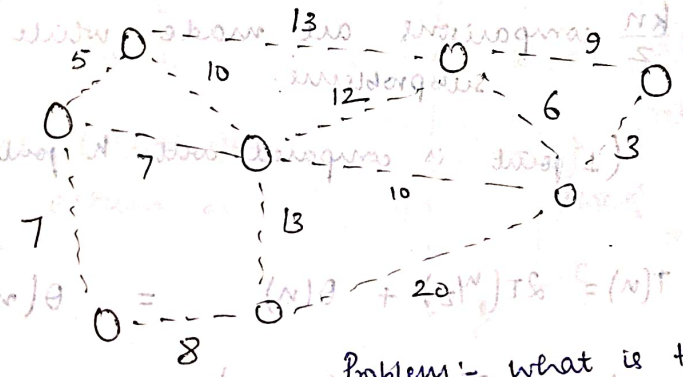


Reverse the second half and then count inversions while merging.
using merge procedure.

Total no. of inversions = $0 + {}^{n-t}C_2 +$ merge( ).

ver asked in Gate but
entioned in all standard books.

Closest Pair in 2D



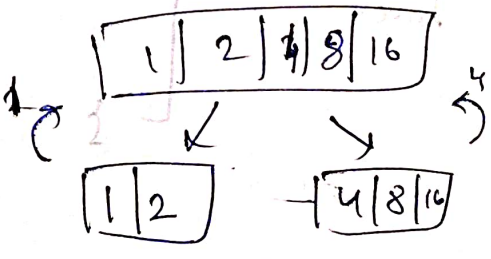
Problem :- what is the minimum distance b/w any 2 points. Ans - 3.

Input :- n points with (x, y) coordinates
Output :- a pair having minimum distance.

rute force — ${}^{n}C_2$ pairs → minimum distance b/w any pair — $O(n^2)$

for 1D points —

1. sort all the points in ascending order (nlogn)
2. use divide and conquer and solve subproblem
   $T(n) = 2T(n/2) + O(1)$
3. Ans — min(δ_L, δ_R, δ)



TC = nlogn + T(n) = nlogn + n = nlogn

For points in 2D —

→ First find minimum distance in left and right subarray.

$$\delta = \min(\delta_L, \delta_R)$$

→ for points near the boundary, find minimum distance in the radius δ.

→ There will always be fixed number of points in radius δ around a point.

Closest Points in 2D

$\therefore$ $\frac{kn}{2}$ comparisons are made while combining subproblems.

(1 point is compared with k points)

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

$T(n) = n\log n + T(n) = \theta(n\log n)$

Exponent of a number

$a^n = a \times a \times a \times a \_\_\_\_\_ \times a$ $\quad O(n)$ time

brute force

```
int exponent(a, n){
    mul = a
    for(int i=2 to n)        O(n) time
        mul = mul * a;
    return mul
}
```

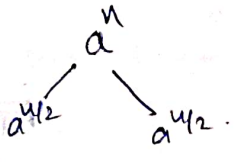
Divide and conquer – 1.

$a = a^{n-1} \cdot a$

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

$\Theta(n)$

Divide and conquer – 2



$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & a \text{ is even} \\ a^{n/2} \cdot a^{n/2} \cdot a & a \text{ is odd.} \end{cases}$$

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

$\Theta(\log n)$

```
int exponent (a, n){
    if(n == 0){
        return 1;
    }
    else if (n == 1){
        return a;
    }
    x = exponent(a, n/2);
    if(n%2 == 0){
        return x*x;
    }
    else{
        return x*x*a;
    }
}
```

Matrix multiplication

```
Multiply (A, B){
    for(int i=0; i<n; i++){
        for(j=0; j<n; j++){
            Cij = 0
            for(k=0; k<n; k++){
                Cij = Cij + aik * bkj
            }
        }
    }
    return C;
}
```

Brute force

$O(n^3)$

Divide and conquer approach.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

A_{11} A_{12} A_{21} A_{22}

↳ Blocks

Matrix A is divided into blocks of size $n/2 \times n/2$.

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$.

8 multiplications are required
i.e. 8 subproblems.

For addition, $\frac{n^2}{4}$ time is req. $\rightarrow O(n^2)$

$\therefore T(n) = 8T(\frac{n}{2}) + n^2$

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

same as naive approach.

Strassen's method of matrix multiplication

$$T(n) = 7T(n/2) + n^2 = \Theta(n^{\log_2 7}) = \Theta(n^{2.80})$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$P = (A_{11} + A_{22})(B_{11} + B_{22})$
$Q = (A_{21} + A_{22})B_{11}$
$R = A_{11}(B_{12} - B_{22})$
$S = A_{22}(B_{21} - B_{11})$
$T = (A_{11} + A_{12})B_{22}$
$U = (A_{21} - A_{11})(B_{11} + B_{12})$
$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

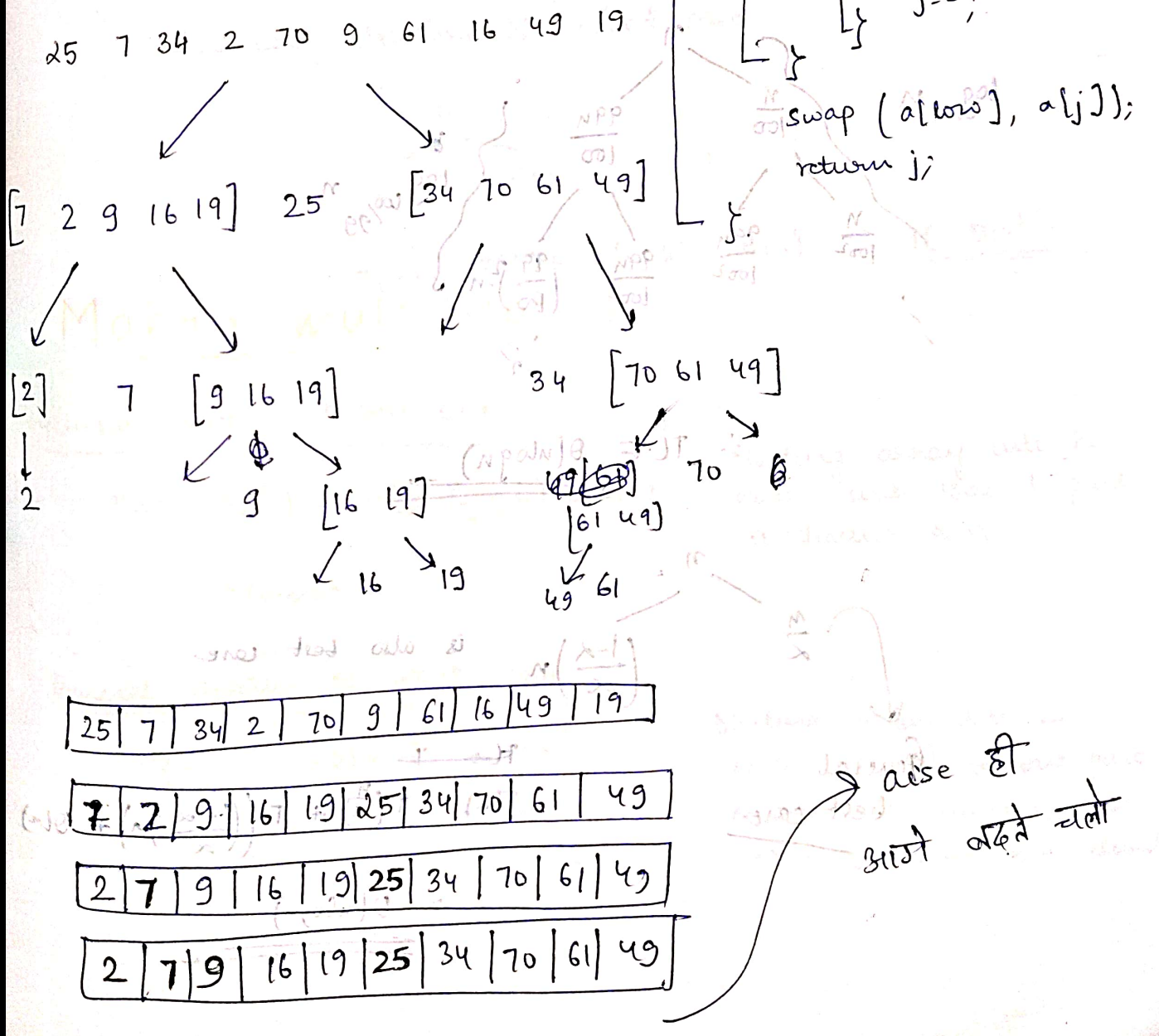
$C_{22} = P + R - Q + U$.

Quick sort

```
quicksort(a, i, j) {
    if (i == j)
        return;
    else {
        q = partition (A, i, j)
        quicksort (a, i, q-1)
        quicksort (A, q+1, j)
    }
}
```

Hoare implementation

```
partition (a, low, high) {
    i = low+1, j = high;
    pivot = a[low]
    while (i <= j) {
        while (i<=j and
            a[i] <= pivot)
            i++;
        while (i<=j and
            a[j] > pivot)
            j--
        if (i<=j) {
            swap (a[i], a[j]);
            i++;
            j--;
        }
    }
    swap (a[low], a[j]);
    return j;
}
```



25	7	34	2	70	9	61	16	49	19
7	2	9	16	19	25	34	70	61	49
2	7	9	16	19	25	34	70	61	49
2	7	9	16	19	25	34	70	61	49

else ही आगे बढ़ते चलो

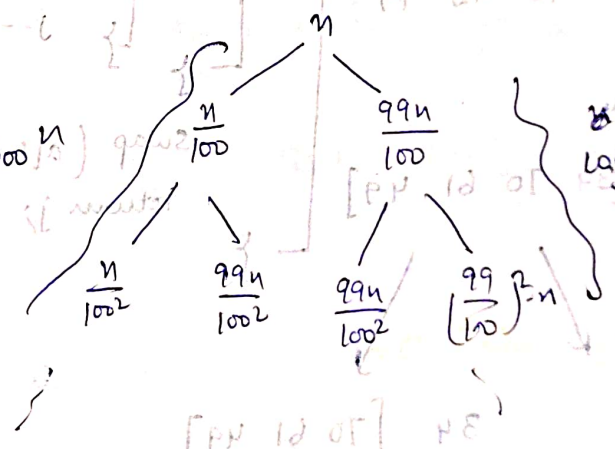
⊙ Quicksort is ~~stable~~ not stable

Best case

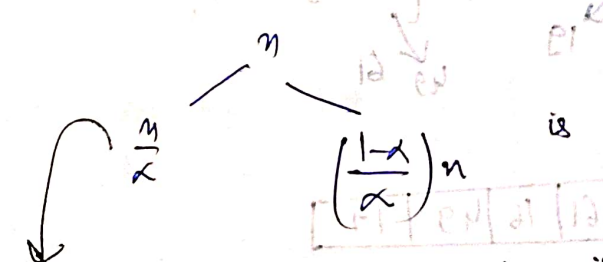
$$T(n) = 2T(n/2) + \underbrace{\Theta(n)}_{\text{divide cost}} = \Theta(n \log n)$$

If partition algo divides array of size $n/100$ into 2 subarrays of size $\frac{n}{100}$ and $\frac{99n}{100}$,

$$T(n) = T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right) + \Theta(n)$$



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



is also best case

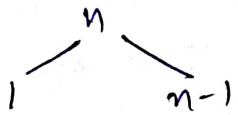
General best case

$$T(n) = T\left(\frac{n}{\alpha}\right) + T\left(\left(\frac{1-\alpha}{\alpha}\right)n\right) + \Theta(n)$$

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

Worst case

One side of partition has only 1 element.



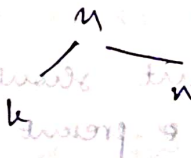
$T(n) = T(1) + T(n-1) + \theta(n)$ $\quad T(c) = \theta(n^2)$

$T(n)$
$T(n-1)$ $\quad n$
$T(n-2)$ $\quad n-1$
$\vdots$
$T(n-k)$ $\quad n-k+1$

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

$= \theta(n^2)$.

General structure of worst case.



$T(n) = T(k) + T(n-k) + \theta(n)$ $\quad T(c) = \theta(n^2)$.

General structure of best case

$T(n) = T\left(\frac{n}{x}\right) + T\left(\left(\frac{x-1}{x}\right)n\right) + \theta(n)$

$= \theta(n \log n)$.

Dividing array into 2 parts such that 1 part is fraction of n.

General structure of worst case

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

$= \theta(n^2)$

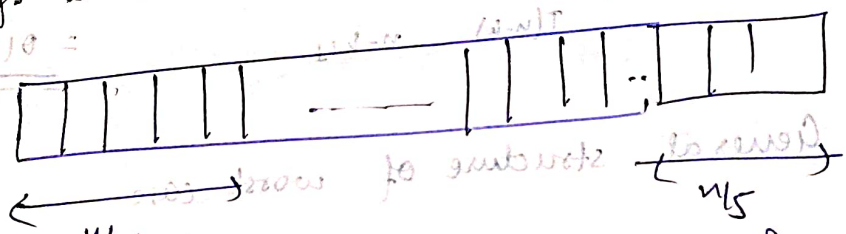
Dividing array into 2 parts such that one part contains k elements and other contains n-k elements.

Quicksort running time —

- ↳ Worst case — $\theta(n^2)$
- ↳ Best case — $\theta(n \log n)$
- ↳ Average case — $\theta(n \log n)$.

Question

What is the probability that, with a randomly chosen pivot element, the partition subroutine produces a split in which the size of the smaller of the 2 subarrays is $\geq n/5$?



↳ pivot element should not be ~~@~~ be present in one of these $\frac{2n}{5}$ positions in final sorted array

$\therefore$ favourable position = $n - \frac{2n}{5}$

$$\text{Probability} = \frac{n - \frac{2n}{5}}{n} = 1 - \frac{2}{5} = \frac{3}{5}$$

$\alpha = 1/5$

Ans → $1 - 2\alpha$

ction

Suppose we choose the median of five items as pivot.
If we have a N element array, we find median of the elements located at positions —

- left (= 0)
- right (= n-1)
- center ($\lfloor (left + right)/2 \rfloor$)
- left of center ($\lfloor (left + center)/2 \rfloor$)
- right of center. ($\lfloor (right + center)/2 \rfloor$)

Worst case running time

In the worst case, only 2 elements are guaranteed to be on one side after partition

It is of the form $T(n) = T(n-2) + T(2) + \Theta(n)$

in worst case

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

The Select Problem

Input:- unsorted array A of size n elements & k.

7	2	6	9	1	5	4	11

Output :- ~~Selk~~ Select(A, k) : k^{th} smallest element of A.

SELECT(A, 1) = 1
SELECT(A, 2) = 2
SELECT(A, 3) = 4

SELECT(A, 8) = 11

SELECT(A, 1) = Minimum element

SELECT(A, n/2) = Median

SELECT(A, n) = Maximum element

* The partition method in quick sort returns the k^{th} smallest element in array.

If partition returns k, pivot element is k^{th} smallest

Naive solution

Sort the array and return the element at k^{th} ~~index~~ index.

→ $\theta(n \log n)$

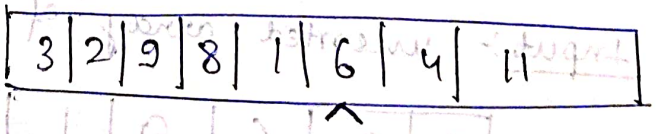
- Divide and conquer.

1. Select a pivot
2. Partition around it
3. Recurse

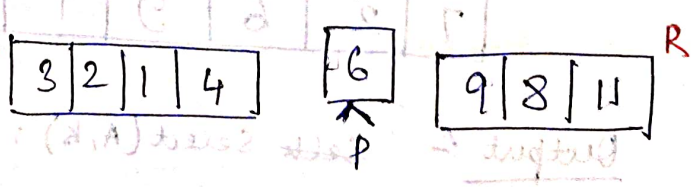
} kind of like binary search for the k^{th} smallest element (except that the array isn't sorted)

The Select Problem

Select a pivot :-



Partition around it



Repeat in subarray L or R depending upon value of k & p.

```
SELECT (A, p, q, k){
    if (p == q){
        return A[0];
    }
    m = partition (A, p, q)
    if (m == k)
        return A[m];
    else if (k < m)
        return SELECT (A, p, m-1, k)
    else
        return SELECT (A, m+1, q, k-m)
}
```

Worst case-

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

$\theta(n^2)$

↳ ∴ not a good algorithm

* This problem can be solved by median of median algorithm.

↳ out of scope.

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

$$= \theta(n).$$

Point to remember:- The SELECT(k) problem can be solved in $\theta(n)$ time complexity.

Binary Search

Given a sorted array, search for an element.

```
BS(A, i, j, key) {
    if (i == j) {
        if (a[i] == key) {
            return i;
        }
        else {
            return -1;
        }
    }
    mid = (i+j)/2;
    if (a[mid] == key)
        return mid;
    else if (key > a[mid]) {
        return BS(A, mid+1, j, key);
    }
    else {
        return BS(A, i, mid-1, key);
    }
}
```

Best case :- $\Theta(1)$. $T(n) = 1$.

Worst case :- $T(n) = T(\frac{n}{2}) + 1$

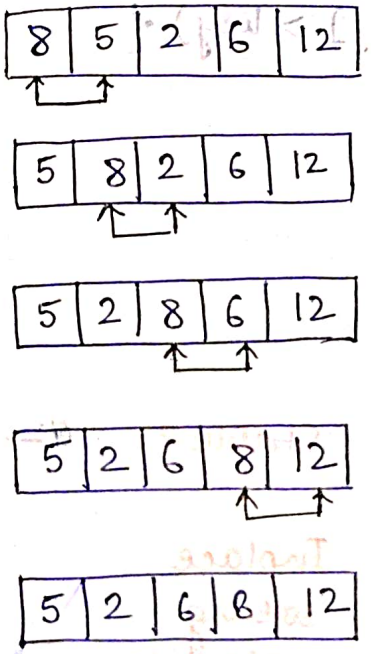
$= \Theta(\log n)$

Average case :- $\Theta(\log n)$.

SORTING ALGORITHMS

1. Bubble sort

- Compare pair of adjacent items.
- Swap if the items are out of order.



1 pass of bubble sort

→ largest element at correct position.

- Sorting from end.

```
for (i=0; i<n-1; i++) {
    for (j=0; j<n-i-1; j++) {
        if (a[j] > a[j+1]) {
            swap (a[j], a[j+1]);
        }
    }
}
```

If no swapping, then stop!

```
for (i=0; i<n-1; i++) {
    bool swapped = false;
    for (j=0; j<n-i-1; j++) {
        if (a[j] > a[j+1]) {
            swapped = true;
            swap (a[j], a[j+1]);
        }
    }
    if (swapped == false) {
        break;
    }
}
```

Time complexity

- Best case – $\theta(n)$
- Worst case – $\theta(n^2)$
- Average case – $\theta(n^2)$.

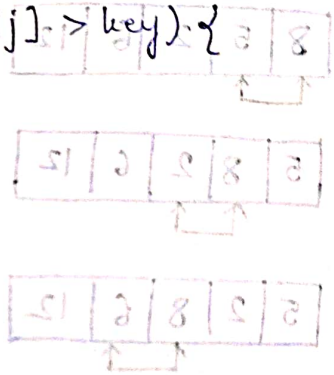
Inplace sorting algorithm? — YES!

Stable? — YES!

2. Insertion sort

At i^{th} iteration / pass, the subarray $A[0 \ldots i-1]$ is already sorted.

```
for( i=1; i<n-1; i++) {
    key = a[i]
    j = i-1
    while( j>=0 && a[j] > key) {
        a[j+1] = a[j];
        j = j-1;
    }
    a[j+1] = key;
}
```



Stable? Q:— YES!

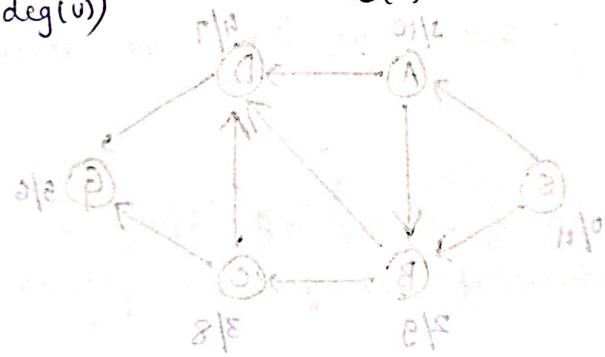
Inplace sorting algorithm YES!

Time complexity

↳ Worst case — $O(n^2)$

GRAPHS

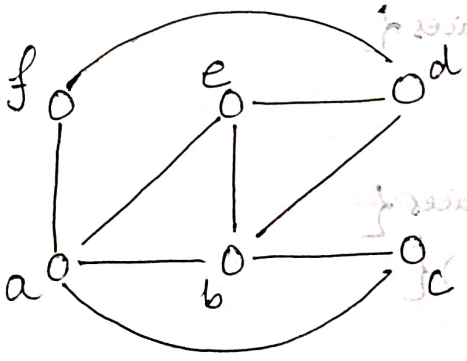
Operation	Adjacency list	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$\Theta(1+\deg(u))$	$\Theta(1)$
List v's neighbours	$\Theta(1+\deg(v))$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$\Theta(1)$	$\Theta(1)$
Delete edge uv	$\Theta(1+\deg(u)+\deg(v))$	$\Theta(1)$



Dense graph ⟶ $|E| \sim |V|^2$

$|E| = \begin{cases} V^2 & \text{in worst case} \\ V & \text{in best case.} \end{cases}$

Depth First Search



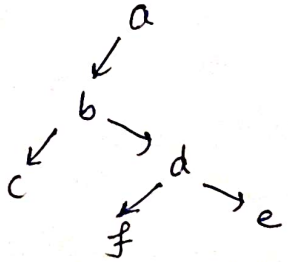
DFS traversal –

a b c d e f

a b c d f e

a c b d e f

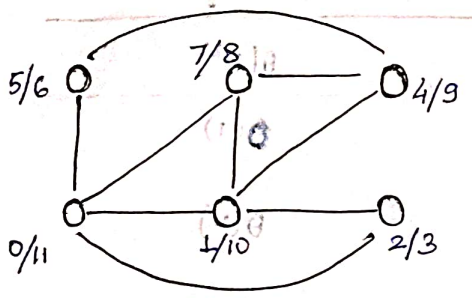
a c b d f e



Start time — the first time we visit a vertex

Finish time — the last time we visit a vertex

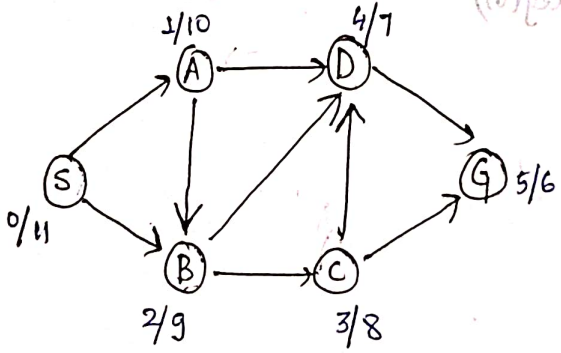
For every node — Start time / Finish time



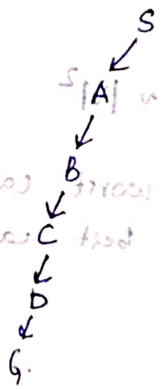
discovery time = d[v]

finish time = f[v]

★ Depth first tree can be directly drawn if the discovery time and finish time of every node in graph is known.

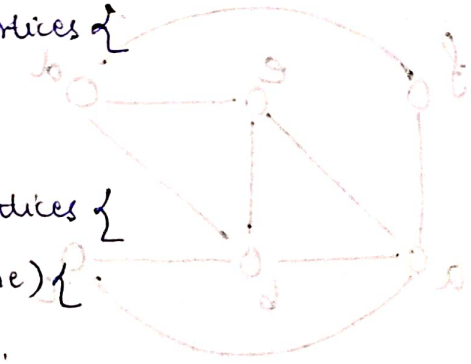


DFS on the graph with starting node s.



Implementing DFS

```
DFS(G) {
    for each vertex u in G.vertices {
        visited[u] = false
    }
    for each u in G.vertices {
        if (visited[u] == false) {
            DFS_VISIT(u);
        }
    }
}
```



```
DFS_VISIT(u){
    visited[u] = true;
    for(each v ; adjacent to u){
        if(not visited(v)){
            DFS_VISIT(v)
        }
    }
}
```

DFS Parenthesis Theorem.

[( )] → balanced ( ){ } balanced

[( ]) → not balanced ([ ]) balanced.

DFS traversal always produces balanced paranthesis.

⊙ In any depth first search of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the following conditions hold:

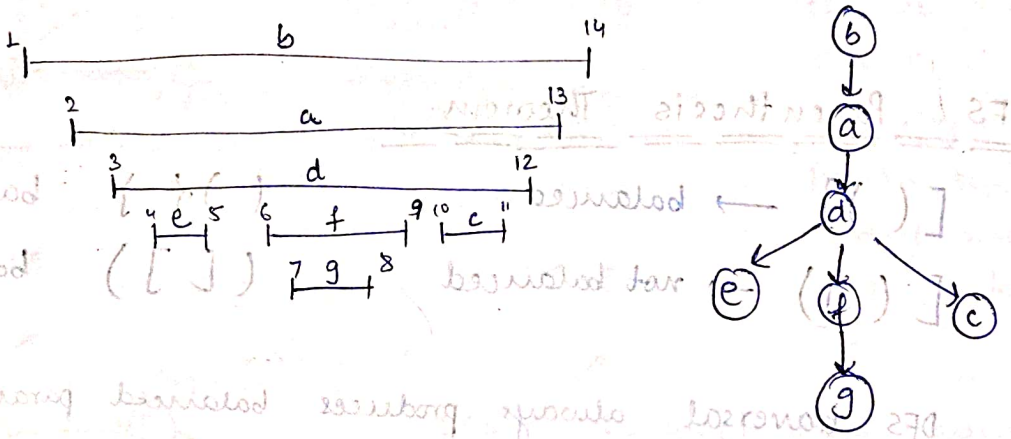
↦ The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth first tree.

↦ The interval $[u.d, u.f)$ is contained entirely within the interval $[v.d, v.f]$ and u is the descendent of v in depth first tree.

↦ The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$ and v is the descendent of u in depth first tree.

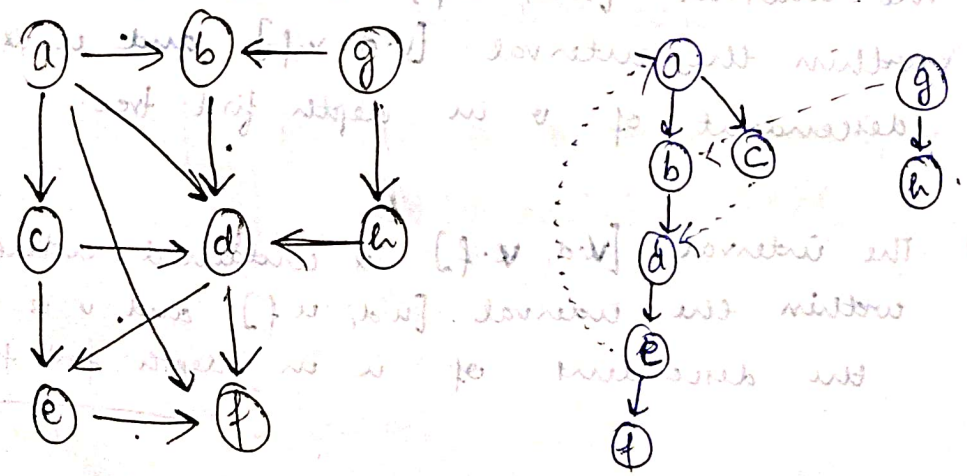
Ques Given the following data about DFS on a directed graph, reconstruct the DFS tree.

Vertex(v)	a	b	c	d	e	f	g
Entry(v)	2	1	10	3	4	6	7
Exit(v)	13	14	11	12	5	9	8



DFS Edge Classification

DFS on
- undirected graph
  - tree edge
  - back edge
- directed graph
  - tree edge
  - back edge
  - cross edge
  - forward edge

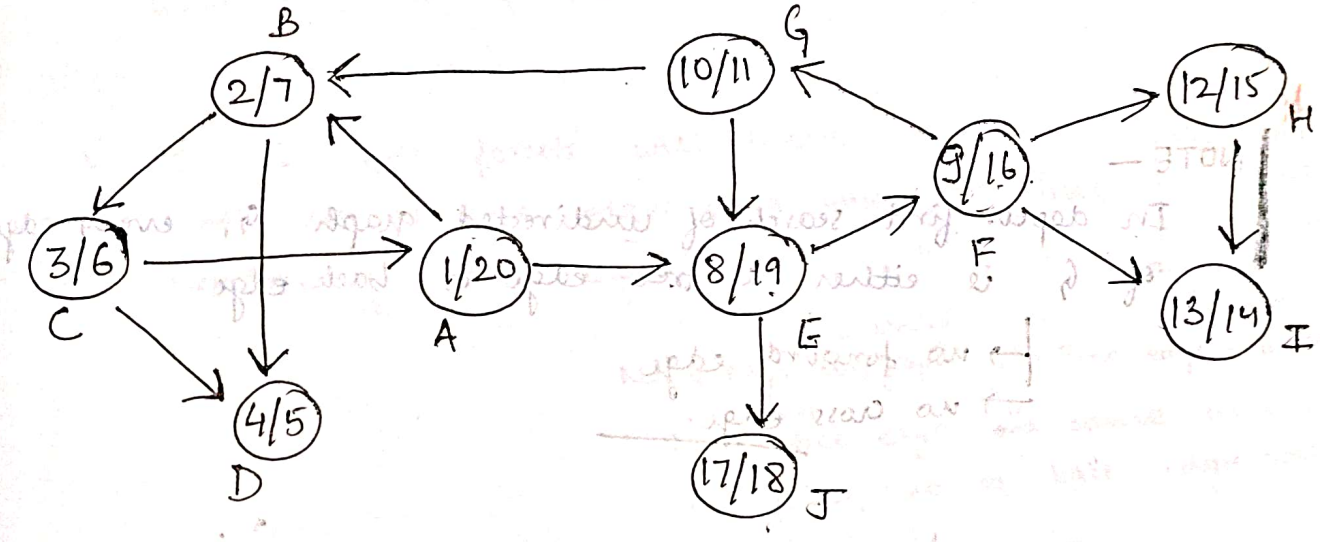


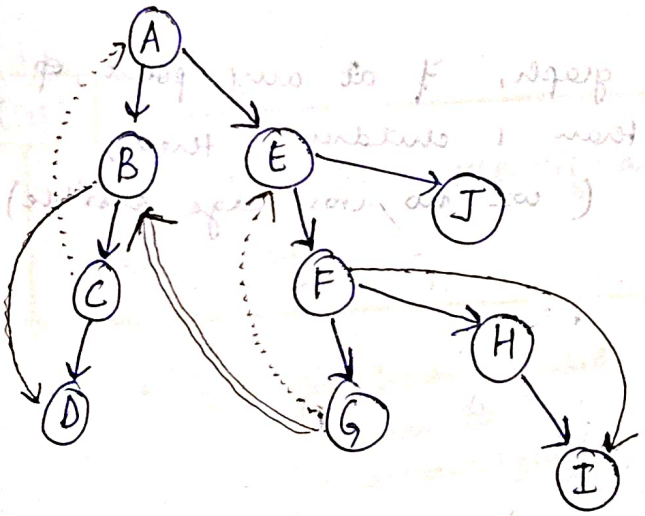
Tree edges — Edges that are part of depth first ~~tree~~ forest are tree edges.

Forward edge — edge which is not tree edge and goes from ancestor to descendant.

Back edge — edge which is not tree edge and goes from descendant to ancestor.

Cross edge — cross edge $c \to d$ iff c is neither ancestor nor descendant of d.





Forward edges — B → D, F → I

Back edges — C → A, G → E.

Cross edge — G → B.

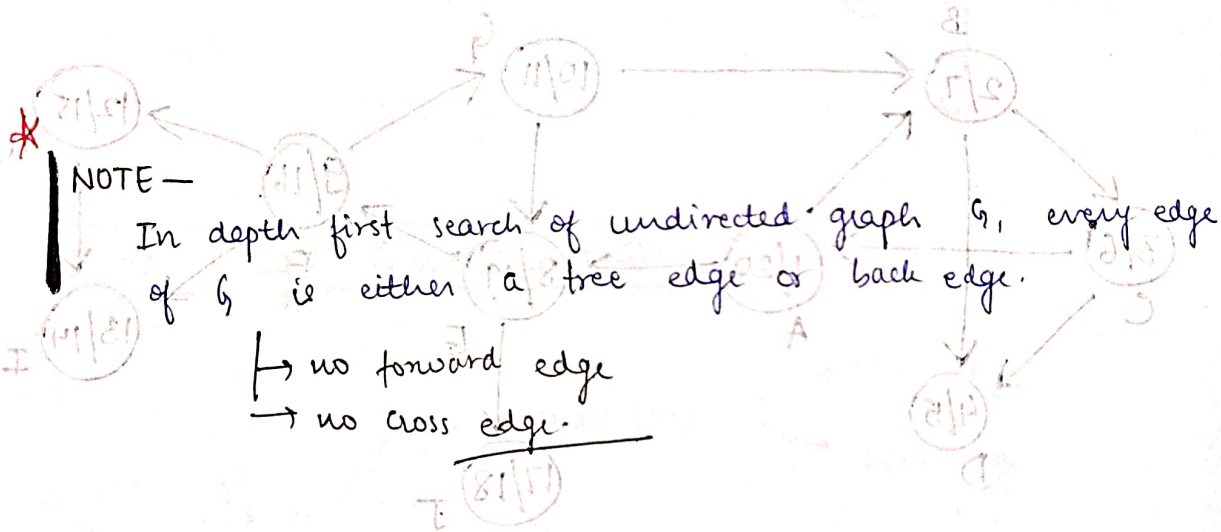
⊙ Forward edge is a non-tree edge (x, y) such that

$$d[x] < d[y] < f[y] < f[x].$$

⊙ Backward edge is a non tree edge (x, y) such that

$$d[y] < d[x] < f[x] < f[y].$$

⊙ Cross edge is a non tree edge (x, y) such that the intervals $(d[x], f[x])$ and $(d[y], f[y])$ are disjoint.



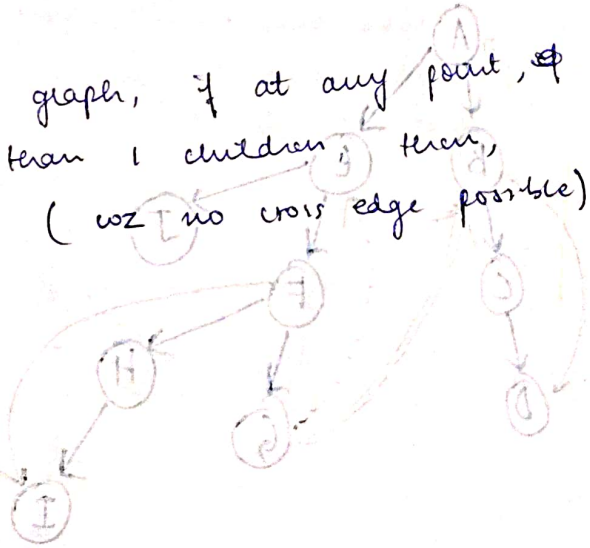
* NOTE —

In depth first search of undirected graph G, every edge of G is either a tree edge or back edge.

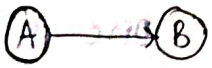
↳ no forward edge
→ no cross edge.

Imp.

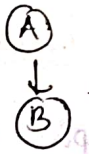
In DFT of undirected graph, if at any point, the root has more than 1 children then, it is a cut vertex (coz no cross edge possible).



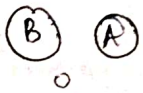
① A depth first search of a directed graph always produces same number of tree edges (ie. independent of the order in which the vertices are provided and independent of the order of adjacency lists.



DFS with source A



DFS with source B



→ False.

① Let G be undirected graph with n vertices and m edges

a) All its DFS forests (for traversals starting at different vertices) will have same no. of trees — True [1 tree will contain connected comp.]

b) All its DFS forests will have same no. of tree edges and same number of back edges. — True [No. of tree edges is same = $n_i - 1$ for each comp.]

No. of edges in undirected graph = Tree edge + Back edge

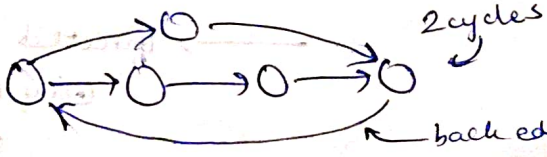
Since no. of tree edges is same in all DFS forests, no. of back edges will also be same.

IMP

A graph has cycle if and only if there is a back edge.

↳ True for both directed as well as undirected graph

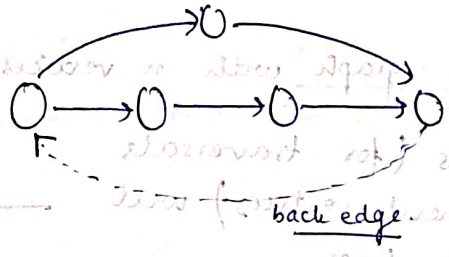
↳ One back edge can lead to multiple cycles (atleast one cycle)



One back edge ⇒ atleast one cycle

Two back edges ⇒ atleast two cycles.

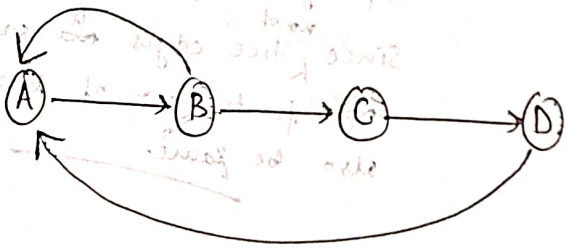
⊙ If a graph contains exactly one back edge, then removing that edge makes the graph acyclic.



On removing back edge, both the cycles are removed from the graph.

⊙ If a directed graph G is cyclic but can be made acyclic by removing one edge, then, depth first search in G will encounter exactly one back edge

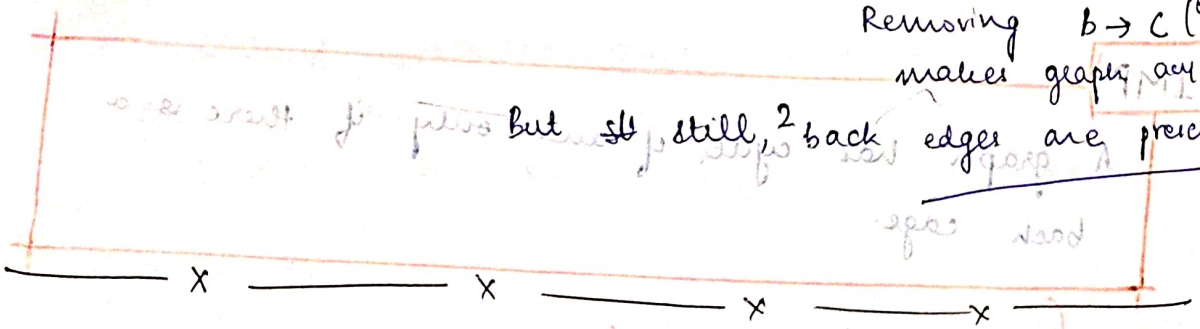
↳ False



2 back edges — B→A and D→A.

Removing B→C (only 1 edge) makes graph acyclic

But still, 2 back edges are present.



Applications of Depth First Search.

⟶ Undirected graph.

⊙ Connected components, articulation points, bridges, biconnected components.

⟶ Directed graph.

⊙ Cyclic / acyclic graphs, topological sort, strongly connected components.

Applications of DFS

- Cycle in the graph (directed & undirected)
- Finding topological sort of DAG (Directed)
- Cut vertex or articulation point (undirected)
- Cut edges or bridges (undirected)

} Can be solved in same complexity as DFS

1. Cycle in a graph –

A graph has cycle if and only if there is a back edge.

Undirected graph

```
public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj){
    boolean vis[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(vis[i] == false){
            return checkForCycle
            if(checkForCycle(i, -1, vis, adj)){
                return true;
            }
        }
    }
}
```

The graph contains a cycle if during DFS traversal, we reach a node such that one of its neighbours which is not parent is already visited.

```
public boolean checkForCycle(int node, int parent,
    boolean vis[], ArrayList<ArrayList<Integer>> adj){
    vis[node] = true;
    for(Integer it: adj.get(node)){
        if(vis[it] == false){
            if(checkForCycle(it, node, vis, adj)){
                return true;
            }
        }
        else if(it != parent){
            return true;
        }
    }
}
```

```
public static boolean isCyclic (int N, ArrayList<ArrayList<Integer>>
adj) {
    int vis[] = new int[N];
    int dfsvis[] = new int[N];

    for (int i=0; i<N; i++) {
        if (vis[i] == 0) {
            if (checkCycle (i, adj, vis, dfsvis) == true) {
                return true;
            }
        }
    }
    return false;
}
```

dfsvis tracks the vertices currently present in stack.

The directed graph contains a cycle if while performing DFS traversal, we reach a node such that one of its neighbours has been visited and is still present in the DFS call stack.

```
static boolean checkCycle (int node, ArrayList<ArrayList<
Integer>> adj, int[] vis, int[] dfsvis) {
    vis[node] = 1;
    dfsvis[node] = 1;
    for (Integer it: adj.get(node)) {
        if (vis[it] == 0) {
            if (checkCycle (it, adj, vis, dfsvis) == true) {
                return true;
            }
        }
        else if (dfsvis[it] == 1) {
            return true;
        }
    }
    dfsvis[node] = 0;
    return false;
}
```

For undirected graph — we need to track parents of nodes to check cycle

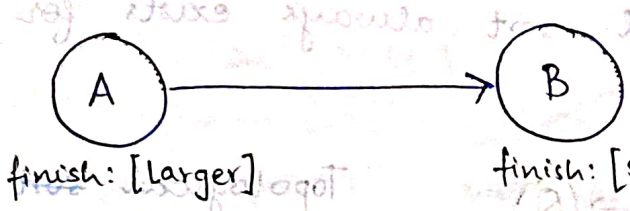
For directed graph — we need to track the nodes which are present in stack.

Directed acyclic graph –

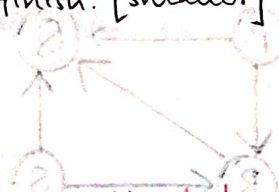
A directed graph which does not contain a cycle.

- In a DAG, we always have.

for any pair of vertices connected by a directed edge in DAG



true for all edges in DAG.



GATE 2007

A depth first search is performed on a directed acyclic graph. Let $d[u]$ denote the time at which vertex u is visited for the first time and $f[u]$ the time at which the DFS call to the vertex u terminates. Which of the following statements is always true for all edges (u, v) in the graph

A. $d[u] < d[v]$

B. $d[u] < f[v]$

C. $f[u] < f[v]$

D. $f[u] > f[v]$.

Directed acyclic graph.

d ∅

$f[v] < f[u]$ ✗

$d[u] < f[u]$ and $d[v] < f[v]$ ✓

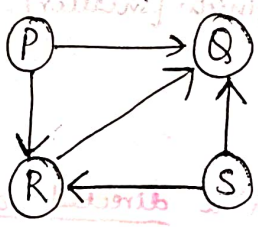
$\therefore d[u] < d[v]$ ✓ also true

2. Finding Topological sort of DAG

⊙ For every edge (u) → (v) u comes before v in topological sort.

⊙ Topological sort always exists for directed acyclic graph.

GATE 2014



Topological ~~sort~~ ordering —

~~PRQS~~ ~~PSQR~~

PSRQ SPRQ.

Kahn's algorithm

→ Go to a vertex with indegree 0. (u)

→ For all neighbours of u decrease indegree by 1

Topo sort using DFS —

1. Call DFS to compute finishing time f[v] for every vertex.
2. As every vertex is finished, insert it onto the front of linked list / stack.
3. Return the list.

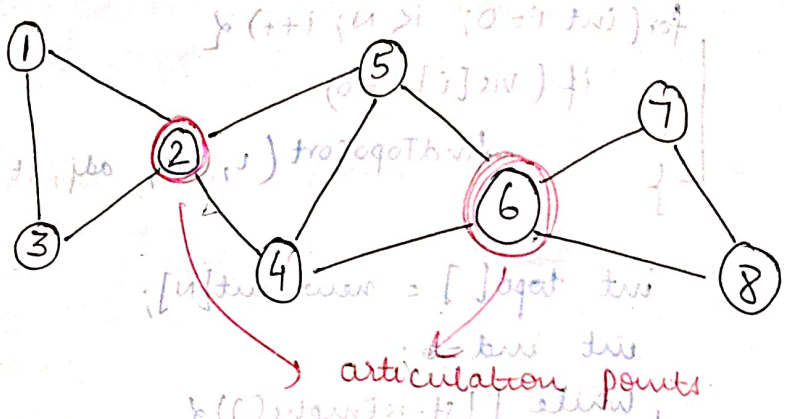
```
int[] toposort(int N, ArrayList<ArrayList<Integer>> adj) {
    Stack<Integer> st = new Stack<Integer>();
    int[] vis = new int[N];
    for(int i=0; i<N; i++) {
        if(vis[i]==0)
            findToposort(i, vis, adj, st);
    }
    int topo[] = new int[N];
    int ind=0;
    while(!st.isEmpty()) {
        topo[ind++] = st.pop();
    }
    return topo;
}

void findtoposort(int node, int[] vis, ArrayList<ArrayList
<Integer>> adj, Stack<Integer> st) {
    vis[node] = 1;
    for(Integer it: adj.get(node)) {
        if(vis[it]==0) {
            findToposort(it, vis, adj, st);
        }
    }
    st.push(node);
}
```

3. Articulation point / Cut vertex in undirected graph.

An articulation point of a graph G is a vertex whose removal disconnects G.

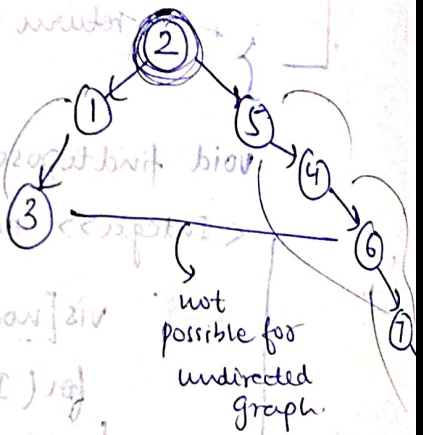
DFS tree of undirected graph contains
- tree edges
- back edges.

⊗ No cross edge.



⊙ Root is an articulation point iff there are atleast 2 children of the root.

[∵ there can't be cross edges in undirected graph].

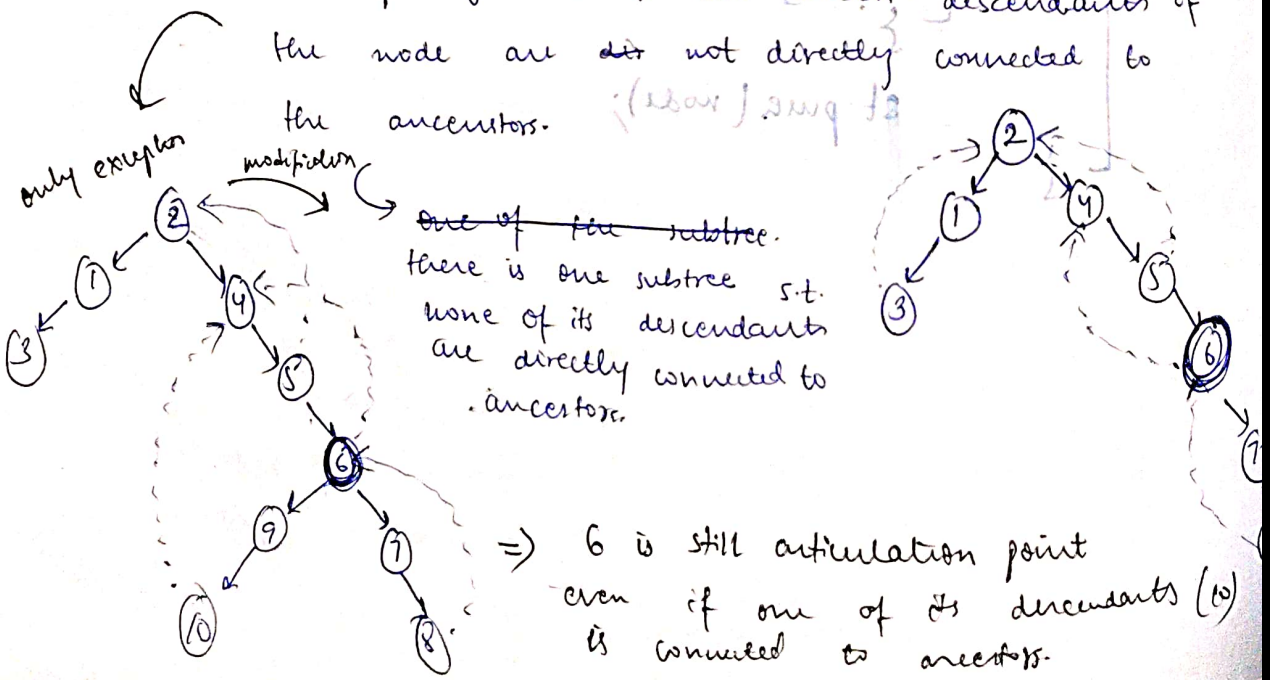


⊙ Non root can be an articulation point iff in the depth first tree, the ~~atleast~~ descendants of the node are ~~dir~~ not directly connected to the ancestors.

only exception

modification

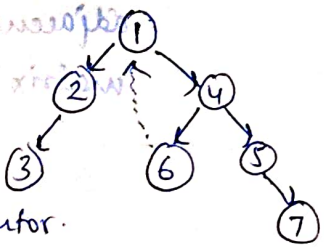
~~One of the subtree.~~ there is one subtree s.t. none of its descendants are directly connected to ancestors.



⇒ 6 is still articulation point even if one of its descendants (10) is connected to ancestors.

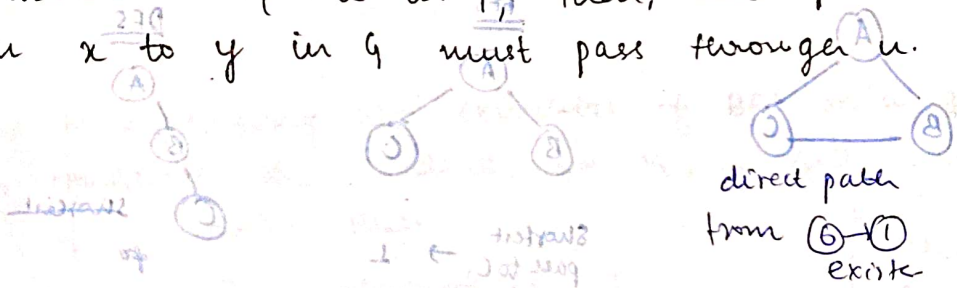
Theorem - An internal node x is articulation point if & only if it has a child y such that there is no back edge from subtree (y) to any ancestor of x.

u is an articulation point if and only if in DFS tree, all of its descendants should connect to its ancestors through u only. → false.



Here 4 is articulation point even if its decendant 6 is connected to ancestor.

If u is an articulation point in G such that x is an ancestor of u in T and y is a descendant of u in T, then, all paths from x to y in G must pass through u. → false.



(same example as above)

$u \rightarrow 4$
$x \rightarrow 1$
$y \rightarrow 6$

Run DFS ⇒ Make depth first tree ⇒ pick leaf node say v
Run DFS again but this time using v as root.
How many children of v will be there in 2nd run of DFS on v as root → Ans – 1

coz. leaf node cannot be articulation point.

Articulation point as root of DFT has 2 children →

Breadth First Search

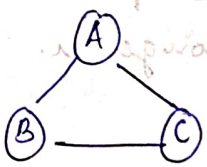
TC using adjacency list = $O(V+E)$

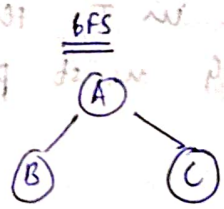
TC using adjacency matrix = $O(V^2)$.

```
BFS ( G, v) {
    visited[v] = true;
    add (v, Q)
    while (Queue is not empty) {
        u = delete (Q)
        for each adjacent x to u {
            if (visited[x] = false) {
                visited[x] = true
                add (x, Q)
            }
        }
    }
}
```

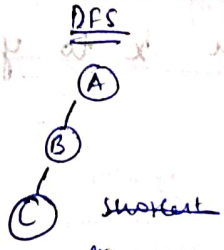
BFS for shortest path in unweighted graph –

BFS always give shortest path from source vertex to destination vertex in ~~undire~~ unweighted graph.





Shortest path to C → 1



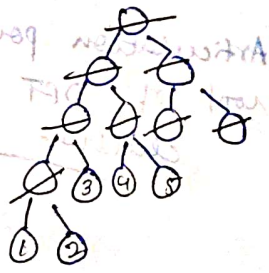
shortest

fo

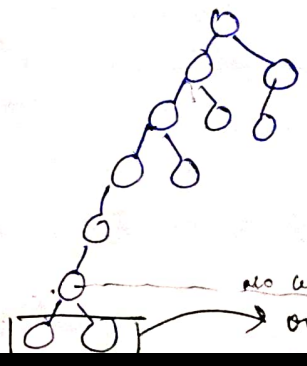
- Just by looking at the partial BFS tree, we can say about the queue state ⇒ internal nodes can never be in the queue.

given partial BFS tree, we can guess the queue state.

Only last 2 level leaves can be present in the queue



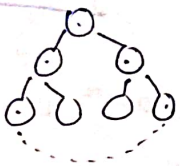
4, 3, 4, 5 → may be present in queue.



no leaf node at two levels → only these 2

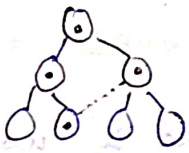
Undirected graph cycles due to cross edges

↳ within level — always forms odd length cycle.



→ 5 nodes cycle.

↳ across (continuous) levels — always forms even length cycle.



→ 4 nodes cycle.

BFS → Directed graph → tree edge → cross edge → back edge

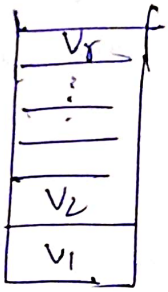
BFS → Undirected graph → tree edge → cross edge.

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$ where v_1 is head of Q and v_r is tail. Then,

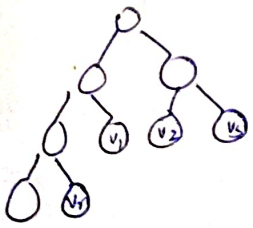
$$d[v_i] \leq d[v_{i+1}]$$

$$d[v_r] \leq d[v_1] + 1$$

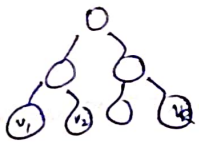
↙ distance



all these vertices are present in continuous level (at most level difference is 1)



or



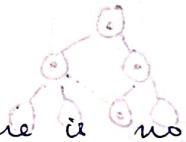
o 2 possible orderings — v_1 & v_R are at same level

v_1 & v_R are at different levels.

Suppose that vertices v_i and v_j are enqueued during the execution of BFS and that v_i is enqueued before v_j.

Then, $d[v_i] \leq d[v_j]$ — True.

Applications of BFS –

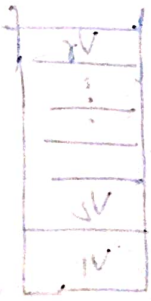


1. Bipartite graph

A graph is bipartite iff there is no odd length cycle.

Checking if a graph is bipartite or not using BFS –

- Undirected graph
  - ↳ if no cross edge, then, bipartite graph.
  - ↳ if cross edge present b/w vertices of same level, then, odd length cycle ∴ not bipartite
  - ↳ if cross edge present b/w vertices of different levels, then even length cycle ∴ bipartite

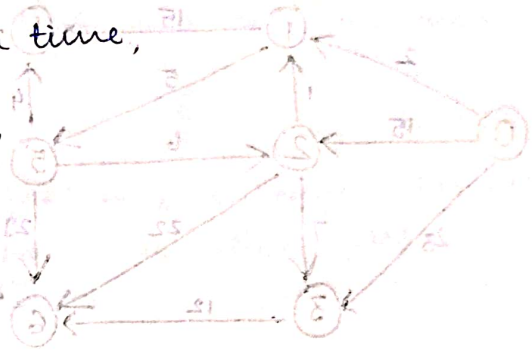


or

Greedy algorithms —

Whatever seems best at the moment, that strategy is applied. This is called greedy strategy

"Commit to choices, one at a time,
never look back,
and hope for the best"



Properties of greedy algorithm —

⊙ Optimal substructure Property —

The optimal solution contains optimal solutions to subproblems.

⊙ Greedy choice property —

The global optimum can be arrived at by selecting a local optimum.

1. Single source shortest Path

Find shortest path among all possible paths.

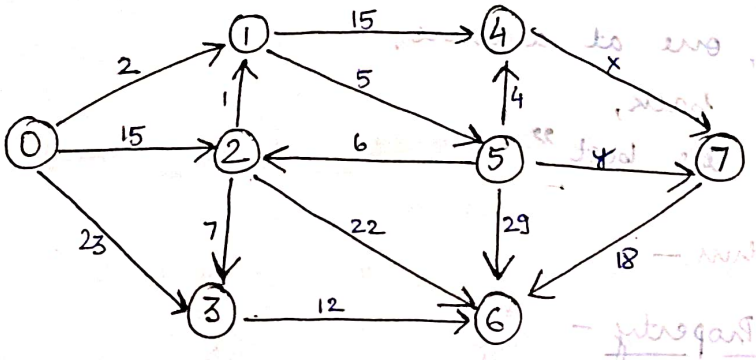
BFS works for unweighted graph. It fails for weighted graphs because it does not care about weights.

DIJKSTRA ALGORITHM

```
DIJKSTRA (G, s):
    key[v] = ∞ for all v in V
    key[s] = 0
    S = ϕ
    Initialize priority queue Q to all vertices
    while Q is not empty
        u = EXTRACT_MIN (Q)
        S = S ∪ {u}
        for each adjacent v of u
            RELAX (u, v, w)
```

```
RELAX(u, v, w)
    if d[v] > d[u] + w
        d[v] = d[u] + w
        Parent[v] = u.
```

Ques Suppose that you are running Dijkstra's algorithm on the edge weighted graph below, starting from vertex 0.



v	Priority Queue	Parent
0	0.0	null
1	2.0	0 → 1
2	13.0	5 → 2
3	23.0	0 → 3
4	11.0	5 → 4
5	7.0	1 → 5
6	36.0	5 → 6
7	19.0	4 → 7

The table gives the priority queue and parent values immediately after vertex 4 has been deleted from the priority queue and relaxed. What are the conditions x & y must satisfy

A. $x = 8.0$ and $y \geq 12.0$

B. $x > 8.0$ and $y = 11.0$

C. $x = 7.0$ and $y = 11.0$

D. $x > 8.0$ and $y = 12.0$

v	Priority queue	Parent
0	0.0	null
1	2.0	0 → 1
2	~~15.0~~ 13.0	0 → 2 ~~5 → 2~~
3	23.0	0 → 3
4	~~16~~ 11	~~1 → 4~~ 5 → 4
5	~~7~~	1 → 5
6	~~29~~	5 → 6
7	~~7+y~~	5 → 7

$7+y > 11+x$

$\Rightarrow 7+y > 19$

$\Rightarrow y > 12$

$\therefore x = 8.0$ and $y \geq 12$.

Now vertex 4 is popped from PQ and outgoing edges are relaxed.

v	Priority Queue	Parent
0	0.0	null
1	2.0	0 → 1
2	13.0	5 → 2
3	23.0	0 → 3
4	11.0	5 → 4
5	7.0	1 → 5
6	29.0	5 → 6
7	7+y	5 → 7

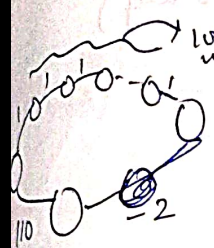
$\min(7+y, 11+x) = 19$

$11+x = 19$ [∵ parent is 4]

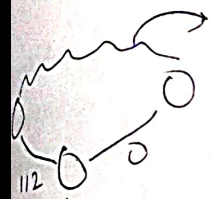
GOOD QUESTION

① Dijkstra does not work for negative edge weights.

① Add some weight to make every weight non negative and then, apply dijkstra



↳ this will not work because adding a weight to every edge adds more weight to long paths than short paths.



↳ this becomes shortest path.

② Dijkstra ~~does~~ may/may not work with -ve edges.

③ Dijkstra does not work with -ve weight cycle.

Time complexity of Dijkstra —

$$E \cdot T(\text{decrease-key}) + V \cdot T(\text{remove min})$$

↓ every edge is relaxed atmost once.

If heap is used as PQ,

$T(\text{decrease key}) = T(\text{remove min}) = O(\log V)$

$\therefore T.C = O(E \log V + V \log V) = O((E+V)\log V)$

$$\sum_{u \in V} \left(T(\text{remove min}) + \sum_{v \in Adj(u)} T(\text{decrease key}) \right)$$

↳ for all vertices. ↳ neighbours of u

Time complexity –

$$TC = \sum_{u \in V} \left[T(\text{removemin}) + \sum_{v \in u\text{-neighbours}} T(\text{decreasekey}) \right]$$

$$= \sum_{u \in V} T(\text{remove-min}) + \sum_{u \in V} \sum_{v \in adj(u)} T(\text{decreasekey})$$

↳ always $\log V$ (coz size of PQ will always be V).

$$= V \log V + \sum_{u \in V} \sum_{v \in adj(u)} T(\text{decreasekey})$$

	Adjacency list	Adjacency matrix
V_1	$deg(V_1) \cdot T(\text{decreasekey})$	$deg(V_1) \cdot T(\text{decreasekey}) + (V - deg(V_1)) \cdot O(1)$
V_2	$deg(V_2) \cdot T(\text{decreasekey})$	$deg(V_2) \cdot T(\text{decreasekey}) + (V - deg(V_2)) \cdot O(1)$
V_3	$deg(V_3) \cdot T(\text{decreasekey})$	$deg(V_3) \cdot T(\text{decreasekey}) + (V - deg(V_3)) \cdot O(1)$
V_4	$deg(V_4) \cdot T(\text{decreasekey})$	$deg(V_4) \cdot T(\text{decreasekey}) + (V - deg(V_4)) \cdot O(1)$
⋮		
V_n	$deg(V_5) \cdot T(\text{decreasekey})$	$deg(V_n) \cdot T(\text{decreasekey}) + (V - deg(V_n)) \cdot O(1)$
Total	$T(\text{decreasekey}) \cdot \sum_i^n deg(V_i)$	$T(\text{decreasekey}) \sum_i^n deg(V_i) + V \cdot \sum_i^n 1 - \sum_i^n deg(V_i)$
	$= \log V \cdot E$	$= \log V \cdot E + V^2 - E$
	$= E \log V$	$= (\log V - 1) E + V^2 \approx O(E \log V + V^2)$

* 1. for adjacency ~~matrix~~ list + heap as PQ,

$$TC = O((E+V) \log V)$$

for adjacency matrix + heap as PQ,

$$TC = O((E+V) \log V + V^2)$$

Shortest path in Directed Acyclic Graph (DAG)

Pick vertices in topological order and relax outgoing edges.

[Dijkstra – pick vertexes at min dist. and relax outgoing edges).

$d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$

DFS $O(V+E)$ ← Topologically sort the vertices of G

For each vertex u, taken in topologically sorted order, do

  for every edge (u, v) do

    Relax (u, v)

  endfor

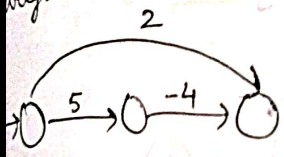
end for.

V_1	$deg(V_1)+1$
V_2	$deg(V_2)+1$
⋮	
V_n	$deg(V_n)+1$

$= V+E$.

$\therefore TC = O(V+E)$

works for negative weights as well



Dijkstra fails here.
This algorithm passes.

2. Bellman Ford Algorithm

- Shortest path algorithm.
- Relaxes edges in any order (V-1) times.
- works for -ve edges as well.

Based on Path Relaxation property

If we relax in the order of shortest path (along with intermixed other relaxations) then we will get shortest path cost.

BellmanFord {

Relax all edges V-1 times

Relax one more time to check cycle (if anything changes ⇒ there is a cycle)

}

```
BELLMANFORD (G, s) {
    d[v] = ∞   for all v in V
    d[s] = 0
    for i = 1 to V-1
        for each edge (u,v) in E
            RELAX (u, v, w)                    } O(VE)
    to check cycle
        for each edge (u,v) in E
            if d[v] > d[u] + w                 } O(E)
                return false
        return true.
}
```

$TC = O(VE + E) = O(VE)$

2 Bellman Ford Algorithm

* Dijkstra with heap $TC = O((E+V)\log V)$

  Bellman ford $TC = O(VE)$

Dijkstra is better than Bellman Ford in terms of time complexity

Adv. of bellman ford over Dijkstra–

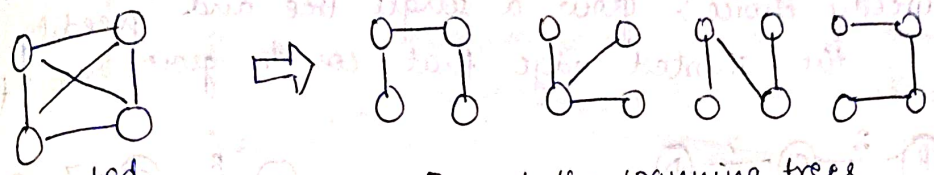
↳ Dijkstra does not work with -ve edge.

After k times relaxation in B.F., we have answer to all vertices which have atmost k shortest path length.

(atmost k edges in shortest path from sou

Minimum spanning Tree

A spanning tree of undirected graph is a connected subgraph which contains all the vertices and has no cycles.



A connected, undirected graph.

Four of the spanning trees of the graph.

Minimum spanning tree — The MST of a graph ~~contains~~ connects all the vertices together while minimizing the number of edges used (and their weights)

GATE 2024 1 mark question.

Number of possible spanning trees for complete graph (K_n) —

n^{n-2}.

Cayley formula.

16 MSTs

* Two Properties of MST —

1. Cut Property — The smallest edge crossing any cut must be part of all MSTs.

2. Cycle property - The largest edge on any ~~cycle~~ cycle is never part of any MST.

3. Kruskal algorithm (greedy)

Sort edges in increasing order of weights } $O(E \log E)$

$T = \phi$

for each edge e_i in the sorted order

if $e_i \cup T$ is not making a cycle → using DFS can be done in $O(V+E)$

$T = T \cup e_i$.

→ if union-find data structure is used, cycle can be detected in $O(1)$ time.

$O[E(V+E)]$

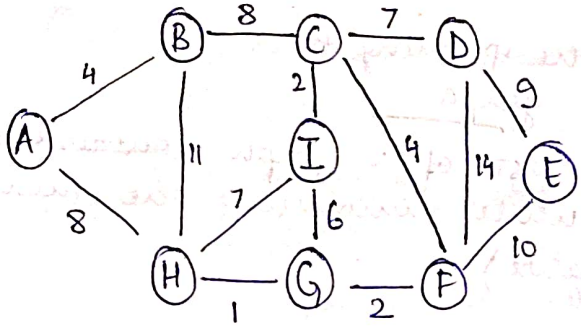
$\therefore TC = O(E \log E + (V+E)E)$

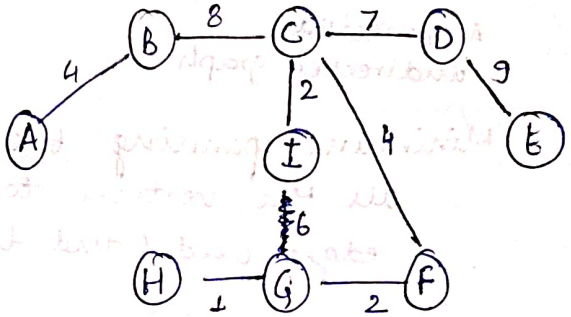
$TC = O(E \log E + E) = O(E \log E)$

4. Prim's algorithm (greedy)

Take any set of vertices, least weight edge in the cut is always part of MST.

Greedy choice :- grow a single tree and greedily add the shortest edge that could grow our tree.





MST with weight 37.

```
DIJKSTRA (G, s) {
    key[v] = ∞ for all v ∈ V
    key[s] = 0
    S = ϕ
    Initialize PQ Q to all
      vertices
    While Q is not empty {
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each adjacent v of u
            if v∈Q & d[v] > d[u] + w {
                d[v] = d[u] + w
                parent[v] = u
            }
    }
}
```

```
PRIMS (G, s) {
    key[v] = ∞ for all v ∈ V
    key[s] = 0
    S = ϕ
    Initialize priority queue Q
      to all vertices
    While Q is not empty {
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each adjacent v
          of u {
            If v∈Q and
              d[v] > w_uv {
                d[v] = w_uv
                parent[v] = u;
            }
    }
}
```

only difference

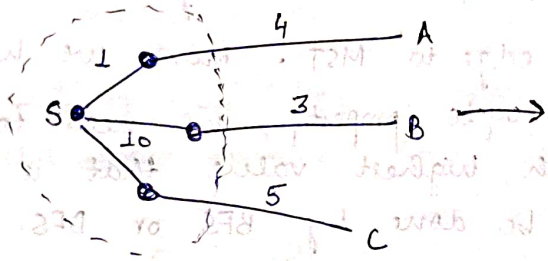
Prim's v/s Dijkstra

Both algorithms take one vertex at a time & relax outgoing edges.

Main distinction :-

Prim's :- closest vertex to tree is chosen.

Dijkstra :- closest vertex to source is chosen.



Prim's algorithm will choose edge as B with wt. 3.

Dijkstra algorithm will choose edge A with wt. 4. (coz distance from source u min (5)).

Prim's algorithm time complexity —

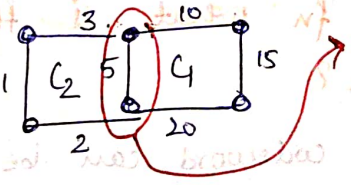
Adjacency list — $V \cdot T(\text{remove min}) + E \cdot T(\text{update key})$

Adjacency matrix — $V \cdot T(\text{remove min}) + E \cdot T(\text{update key}) + V^2$.

Ques

If G has a cycle and there is a unique edge e which has the minimum weight on this cycle, then, e must be part of MST ——— False

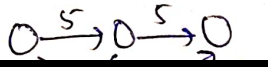
e may be maximum wt. in some other cycle.



min in C_1 but max in C_2 ∴ not included in MST.

⊙ If an edge e is not part of any MST of G, then, it must be maximum edge weight on some cycle in G.
——True.

⊙ Suppose the edge weights are non negative. Then, the shortest path b/w 2 vertices must be part of some MST
——False



Changing weight of an edge in a graph (on MST).

1. Edge is in MST and weight is decreased
   Current MST is still MST.

2. Edge is not in MST and weight is increased
   Current MST is still MST

3. Edge is not in MST and weight is decreased
   Add this edge to MST. Now, we have exactly 1 cycle
   Based on cycle property, we need to find and remove edge with highest value that is on the cycle.
   This can be done by BFS or DFS.

cycle property.
$O(V+E)$

4. Edge is in MST and weight is increased
   Remove this edge from MST. Now, we have 2 connected components. Find both the connected components using DFS / BFS.
   Now, find the edge with smallest weight that connects these components.

cut property.
to find components.
crossing edges
$O(V+E) + O(E) + O(E) + O(1)$
min. among crossing edges
compare if MST is changed.

Ques Under Huffman coding of n symbols with frequencies $f_1, f_2, \ldots f_n$ what is the longest codeword could possibly be?

longest codeword can be of length $n-1$

An encoding of n symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots 1/2^{n-2}$ and two of them having probabilities $1/2^{n-1}$.

Huffman encoding (greedy)

{Goal :- High frequency characters are encoded using least bits.

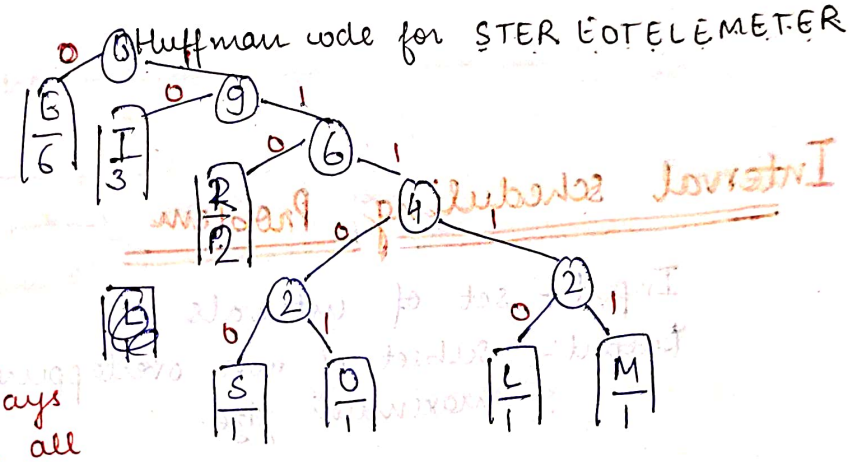
Input :- Some distribution on characters

Output :- A way to encode the characters as efficiently as possible.

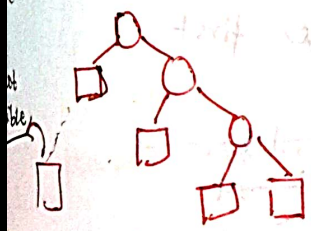
Prefix free code - A code is called prefix free code if any code is not a prefix of another code

Huffman coding - greedily builds subtrees by merging, starting with the two most infrequent letters.

11100	S – 1
11101	O – 1
11110	L – 1
11111	M – 1
110	R – 2
10	T – 3
0	E – 6



Huffman codes are always prefix-free code because all the characters are at the leaf (hence 2 or more character below a character).



```
HUFFMAN(C){
    n = |C|          → Priority queue.
    Q = C
    for i=1 to n-1 {
        x = EXTRACT-MIN(Q)        } log n
        y = EXTRACT-MIN(Q)        } log n
        z.freq. = x.freq + y.freq.
        INSERT(Q, z)              } log n
    }
}
```

n times

TC = $\theta(n \log n)$

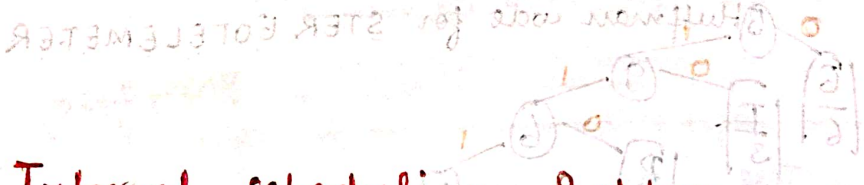
Optimal Merge Pattern

[Huffman coding extention]

Input :- Set of files of different lengths
Output :- an optimal ~~sub~~sequence of two way merges to obtain a single file.

* Distance from leaf node to root = No. of times the file is involved in merging

We want large files to be involved less => large files should have less distance.

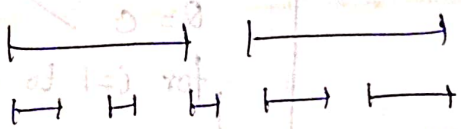


Interval scheduling Problem

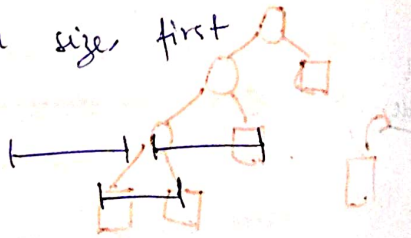
Input :- set of intervals
Output :- subset of non overlapping intervals of maximum size.

Algo1

Let's pick intervals of minimum size first



works for this case

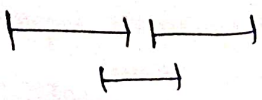


does not work for this case.

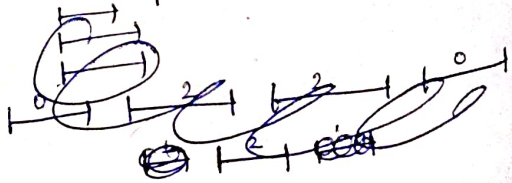
∴ ~~wrong algo~~ not working

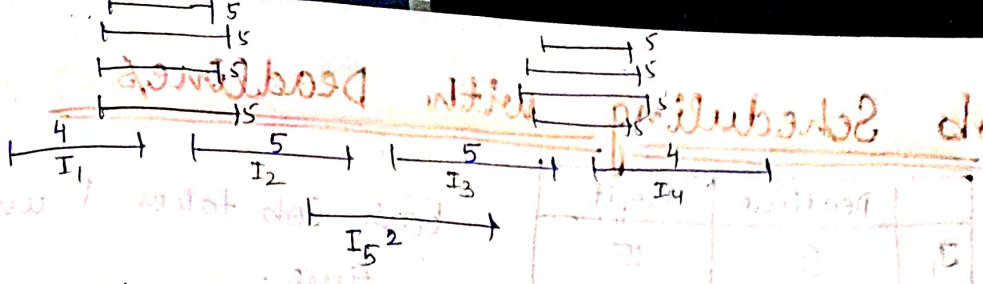
Algo2

Pick an interval that overlaps with minimum number of intervals.



works for this case





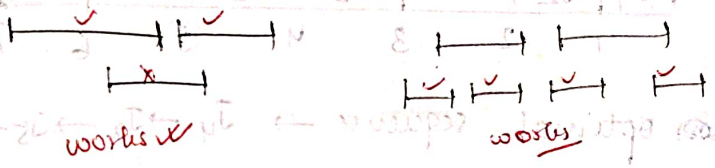
~~correct~~ optimal intervals — I_1 I_2 I_3 I_4 (4)

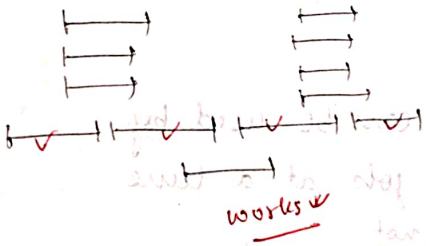
using the given algo — I_5 I_1 I_4 (3)

∴ not working

Algo 3

Pick the interval with earliest finish time





Optimal algorithm for interval scheduling problem —

↦ Choose the interval x that starts last, discard all classes that conflict with x and recurse.

↦ Choose the interval x that finishes first, discard all classes that conflict with x and recurse.

Other variants of this problem —

↦ Weighted Interval scheduling

↦ Interval Partitioning problem

↦ Scheduling to minimizing total lateness } req. ~~not~~ in Gate

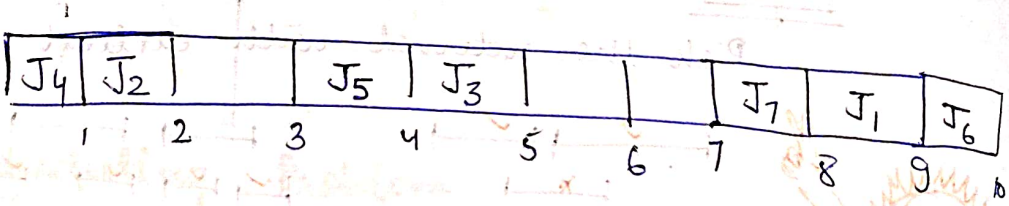
Job Scheduling with Deadlines

	Deadline	Profit
J_1	9	15
J_2	1	2
J_3	5	18
J_4	7	1
J_5	4	25
J_6	10	20
J_7	5	8

Each Job takes 1 unit of time.

Objective :-

schedule all jobs such that we get maximum profit.



∴ optimal sequence → $J_4 \rightarrow J_2 \rightarrow J_5 \rightarrow J_3 \rightarrow J_7 \rightarrow J_1 \rightarrow J_6$

Conditions —

- The resource can be used by almost one job at a time
- Jobs should not overlap.
- If job i has deadline D_i then, it has to be done before deadline
- Each job has profit g_i
- job has unit length.

Goal

Find a feasible schedule with maximum profit.

Algorithm —

1. Sort all the jobs in decreasing order of profit
2. Find maximum deadline and initialize array
3. Start from right side of the array and search for the slot @ i to find job which contains deadline <=i (linear search)
4. Repeat step 3 for all jobs.