omsairam omsriventatesvayanamaha ommahaganadhipataye namaha omnamaha shivayaha

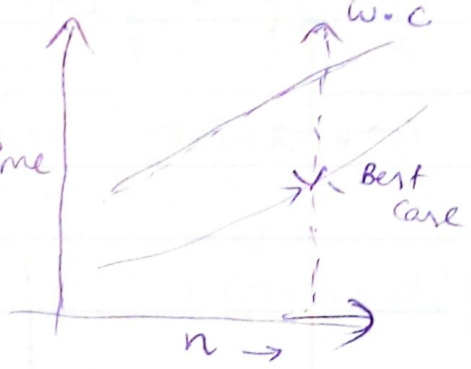
Introduction to Algorithms

1) Worst - Best - Average Case Complexities

Performance: Trend of "n"

→ ignore constant factors size of input encoding

→ Mathematicians came with symbols, to compare functions Time



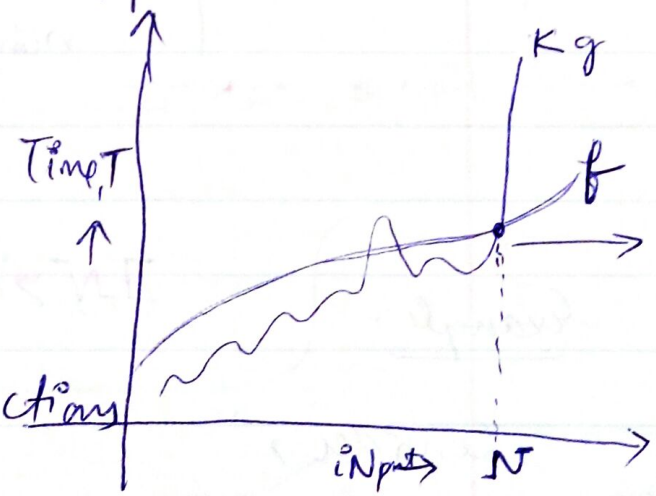
$\overset{\leq}{O}$, $\overset{\geq}{\Omega}$, $\overset{\approx}{\Theta}$

$f = O(g)$

$f \overset{\leq}{} O(g)$ "is upper bounded by"

⇓

→ It means function, "f" is upperbounded by the function, "g"



Explanation:-

~~It means~~.

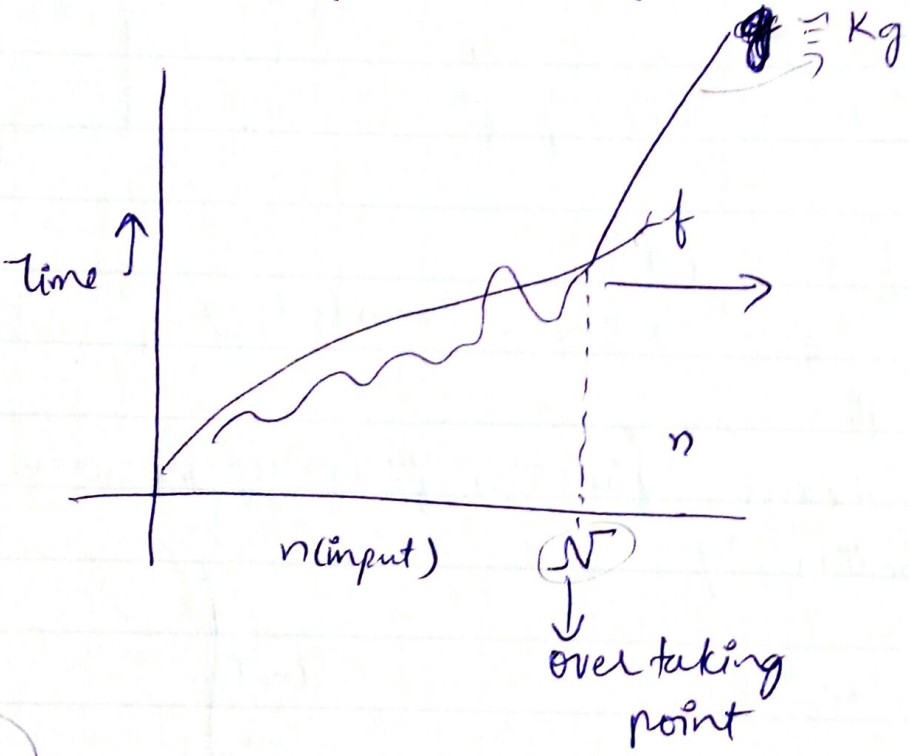
Let us take two functions "f" and "g"

$f = n$

$g = n^2$

However, great your implementation might be for $g = n^2$ & and however worst the implementation for $f = n$

there will be always a overtaking point for function 'g' in time taken to complete. In comparison with function 'f':



Example:-

$$\exists N > 0, \exists k > 0, \forall n > N$$

$$f(n) \leq k\, g(n)$$

Consider;

$f =$ ~~2000~~ (2 trillion) $\times n$; $g = 0.000000002\, n^2$

In the above example, for small values of "n" the function "g" might be performing better. but there will be always an overtaking value of input size, where function "f" performs better than "g"

→ We can level function "g" (or) bring "g" down to "f", by multiplying with a constant "k"

→ So, in some algorithms, (or) coding, we use certain functions till certain input size and later use another after reaching the overtaking value

Overhead:

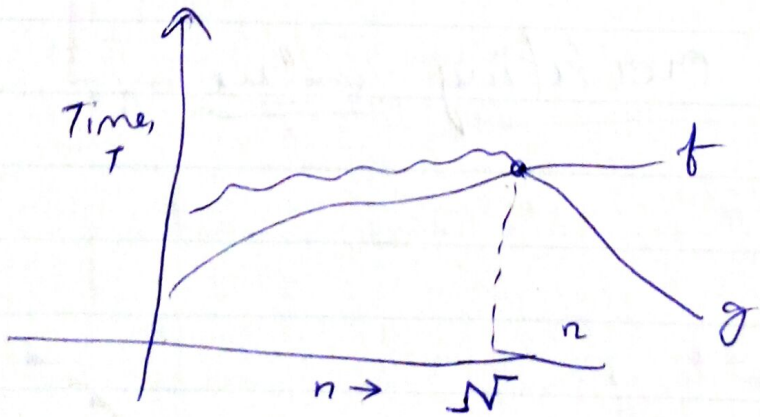
In the above example,

$f = 2000000000n$; & $g = 0.000000002n^2$

It is called overhead, the function, "f" algorithm will not do good for small input but will do better for large ~~data~~ inputs after reaching the "overtaking" point

Big Omega (Ω):-

It is just the same as "g" big 'O' but in the opposite direction



$f = \Omega(g)$

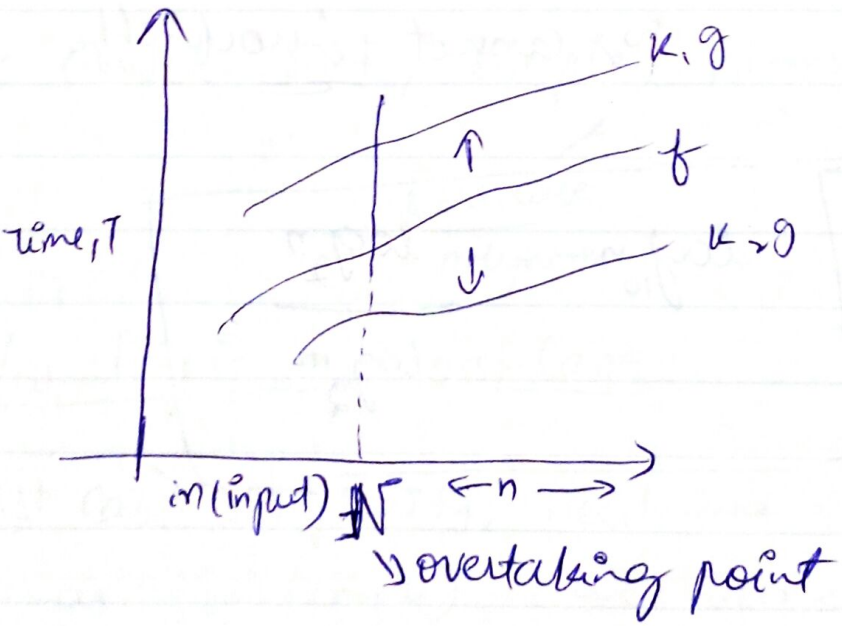
$\geq$; & $f(n) \geq kg(n)$

Big Theta (Θ):

$$f = \Theta(g) \text{ iff } f = O(g) \text{ and } f = \Omega(g)$$

In this case, "after" reaching the "overtaking point", "function, f" remains sandwitched b/w "$k_1 g$" and "$k_2 g$"

(Before reaching overtaking point anything can happen).



$\forall n \geq N \quad f \in [k_2 g, k_1 g]$;

$\exists N, \exists k_1 > 0 \quad \exists k_2 > 0$

While comparing functions in "Θ" (Big Theta)

$f = n^2 + \sin^2(n)$ $\quad$ $g = 20n^2 + 3n + 14\sqrt{n}$

Only keep functions which are dominating

Ex: $f = n^2 + \sin(n)$; $g = 20n^2 + 3n + 14\sqrt{n}$

$\downarrow\downarrow$ $\quad$ $\downarrow\downarrow$

$f = n^2$; $g = n^2$

BUT,

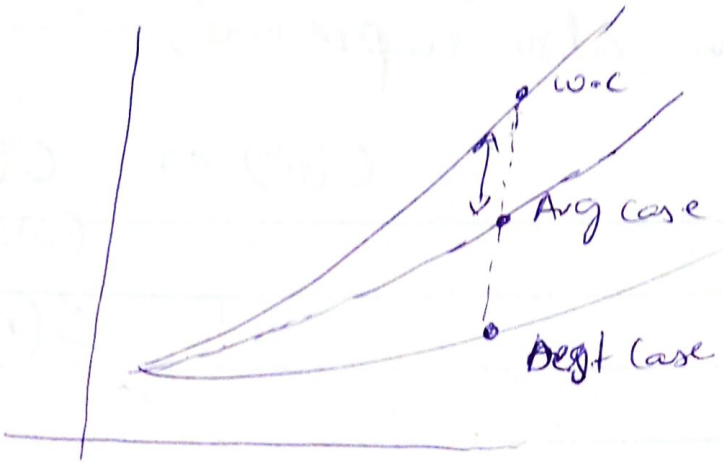
we cannot knock-off powers in exponents

$f = 2^n$; $g = 2^{3n} \Rightarrow (2^n)^3 \Rightarrow f^3$

so, cannot knock-off

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

Average Case :-



→ Average of all possible inputs b/w the best case and the worst case, and we get Average case.

→ Certain algorithms will have huge difference b/w "Avg case" & "worst case"

ie, Ex:- Quick Sort, Simplex Algorithm.

Quick Sort: best case polynomial, worst case exponential

Confusion b/w Big Θ & Worst Case :-

→ worst case complexity, need not to be always represented as "O" (Big O), we can even use Big Theta, Θ ⇒ W.C = $\Theta(n^2)$ ⇒ it means, worst case complexity, is sandwiched b/w $k_1 n^2$ & $k_2 n^2$

→ we can also represent,

$\Theta(n^2)$ as $O(n^3)$

(or)

$O(n^2 \log n)$

Euclid's Algorithm

gcd (35, 20)

↳ gcd (20, 15)

↳ gcd (15, 5)

↳ gcd (5, 0)

↳ 5

gcd (m, n) $\quad m \geq n \geq 0$

if m == n : return m

if n == 0 : return m

if n == 1 return 1

gcd (n, m mod n)

Running time of gcd Algo:

If gcd (m, n) runs k times $m \geq F_{k+1}$, $n \geq F_k$

$F_0 = 1 \quad F_1 = 1 \; ; \; F_2 = 2 \; .. \; F_3 = 3 \; ...$

→

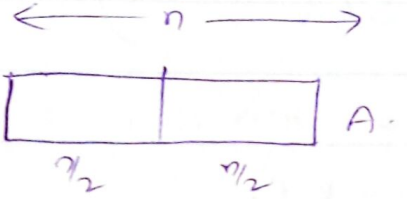
The input size of a number

For ex: 225

$= \lceil \log_2 225 \rceil$

Divide and Conquer Algorithms

Merge Sort:




```
mergeSort(A)
    if (n == 1):          } Θ(1)
        return A
    B = mergesort(A(0) ... A(n/2))
    C = mergesort(A(n/2+1) ... A(n))
    return merge(B, C) → Θ(n)
```

$$T(n) = \begin{cases} 16 & \text{if } n \le 1 \\ 2T(n/2) + C_1 n & \text{otherwise} \end{cases}$$

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

Expansion Method:

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

$$= 2\left[2T\left(\frac{n}{4}\right) + C\frac{n}{2}\right] + Cn$$

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

$$= 4\left[2T\left(\frac{n}{8}\right) + C\left(\frac{n}{4}\right)\right] + Cn$$

$$= 8T\left(\frac{n}{8}\right) + cn + cn + cn$$

$$= 8T\left(\frac{n}{8}\right) + 3cn$$

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

For the base case,

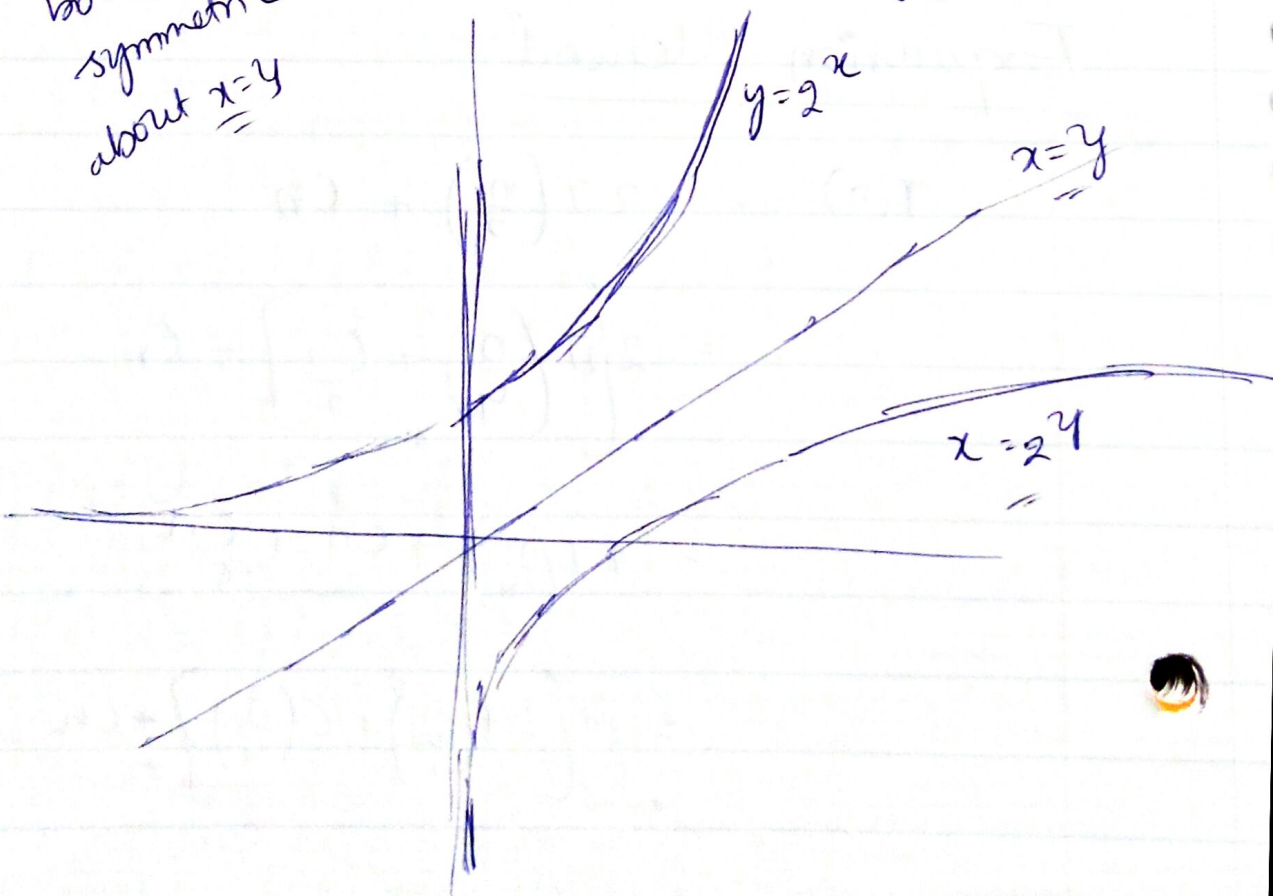
$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \boxed{k = \log_2 n}$$

Logarithms

~~$x = \log$~~

both are symmetric about $x = y$

$$\begin{cases} x = 2^y \Rightarrow y = \log_2 x \\ y = 2^x \Rightarrow x = \log_2 y \end{cases}$$



substitute k in $2^k T\left(\frac{n}{2^k}\right) + kcn$

$$= \left(2^{\log_2 n}\right) c_0 + cn\log\left(\frac{n}{2}\right)$$

$$= cn\log n + c_0 n$$

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

MASTER'S THEOREM :-

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$$

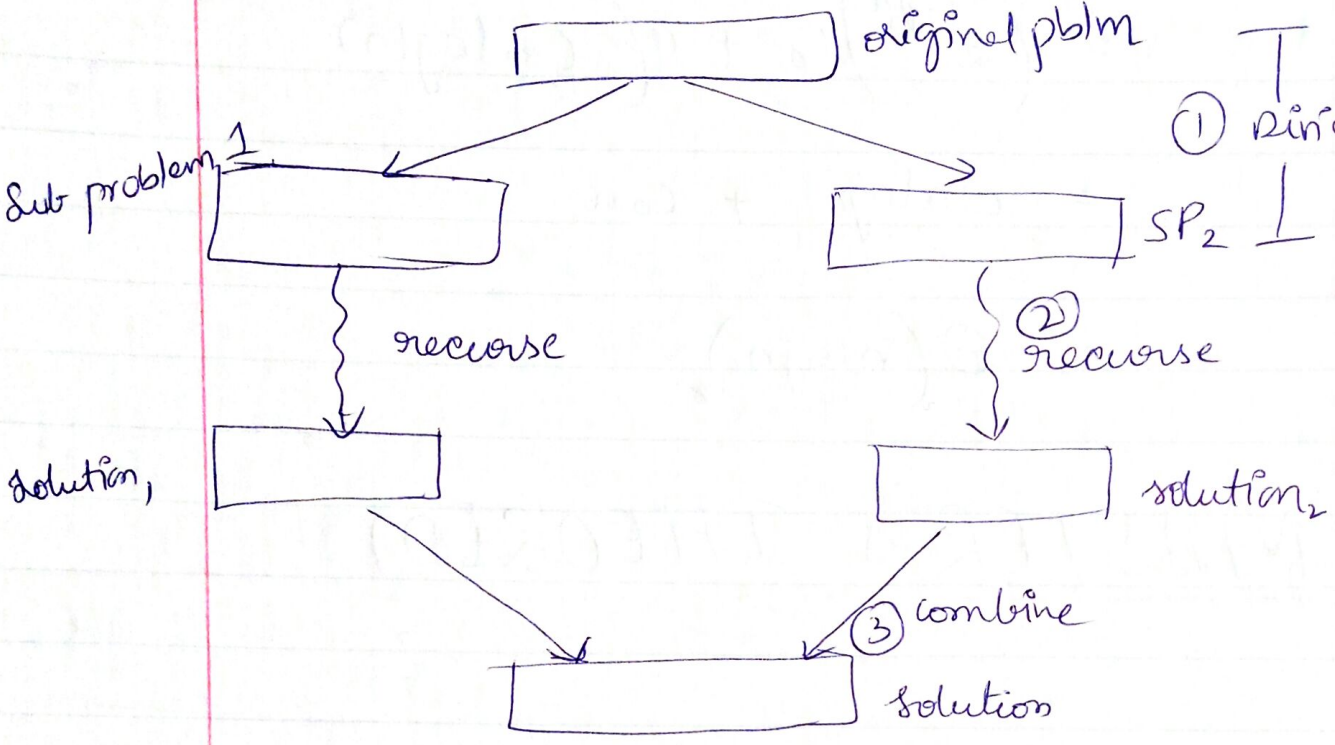
$\varepsilon = \log_b^a$ compare 'c'

$\varepsilon > c \quad T(n) = \Theta(n^\varepsilon)$

$\varepsilon = c \quad T(n) = \Theta(n^\varepsilon \log_2 n)$

$\varepsilon < c \quad T(n) = \Theta(n^c)$

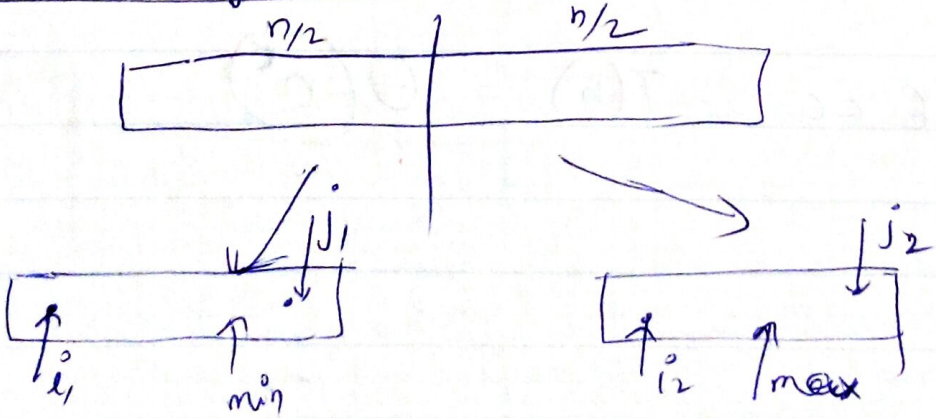
Divide and Conquer Algorithms :-



Algo's

1) Max-Subarray Pblm
2) Karatsuba multiplication
3) Finding closest pair
4) F F T.

Max-Subarray Pblm :-



1) ~~Divide~~

Goal:- To find the two elements in the array such that, $j > i$, and $a[j] - a[i]$ is the maximum difference (index) in the array.

Steps:-

1) Divide the array into two halves
2) find the ① max diff elements in first half
   ② max diff elements in second half
   ③ min in first half & max in second half & check the difference
3) Choose the best among above "3" options

$$\max \begin{cases} A[j_1] - A[i_1] \\ A[max] - A[min] \\ A[j_2] - A[i_2] \end{cases}$$

$$\rightarrow T(n) = 2T(n/2) + \Theta(1) + \Theta(n)$$

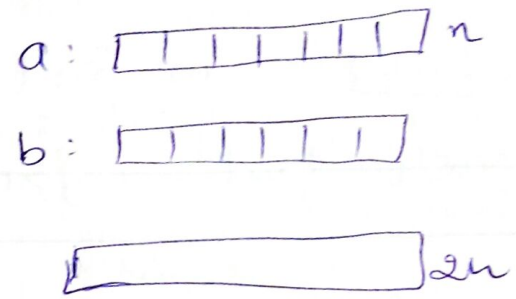
$$= 2T(n/2) + \Theta(n)$$

$$= n \log n //$$

Karatsuba Multiplication :-

9178432171
2138123492 ×

a × b

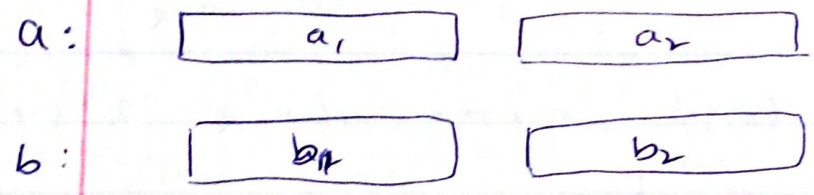


Adding no's is easy ⇒ In $\Theta(n)$ can be done

→ Multiplying two numbers in 'Grade School multiplication'
$= \Theta(n^2)$,

Divide and Conquer

1) Divide a number into two halves



a : 794 321
a_1 a_2

$a = a_1 \times 10^{n/2} + a_2$

$b = b_1 \times 10^{n/2} + b_2$

→ $a \times b = a_1 b_1 \times 10^{n} + a_1 b_2 \times 10^{n/2} + a_2 b_1 \times 10^{n/2} + a_2 b_2$

2n digit multiplication

① 4 n/2 digits
② Pad + shift
③ 4 adds

⇒

$$\Rightarrow T(n) = 4T(n/2) + \Theta(n) + \Theta(n)$$

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

$$\boxed{T(n) = \Theta(n^2)} \Rightarrow$$

But above method of multiplication is same as "Grade School Multiplication"

BUT ||Karatsuba|| thought in a different Manner:

② He ~~found, multiplying~~

$$\cancel{(a_1+a_2)\cdot(b_1+b_2) = a_1b_1}$$

① He reduced four multiplications to three,

② He made an observation that, middle term in the multiplication can be obtained by,

$$(a_1+a_2)\cdot(b_1+b_2) - a_1b_1 - a_2b_2$$

③ Therefore no need for "4" multiplications, i.e, three multiplications are enough

(recursive call to multiply $a_1 \cdot b_1$)

$$r_1 = a_1 \times b_1$$

$$r_2 = a_2 \times b_2$$

$$r_3 = (a_1 + a_2) \cdot (b_1 + b_2)$$

$$\Rightarrow a \times b = \underbrace{a_1 \cdot b_1}_{} + (a_1 + a_2) \cdot (b_1 + b_2) - a_1 b_1 - a_2 b_2 + a_2 b_2$$

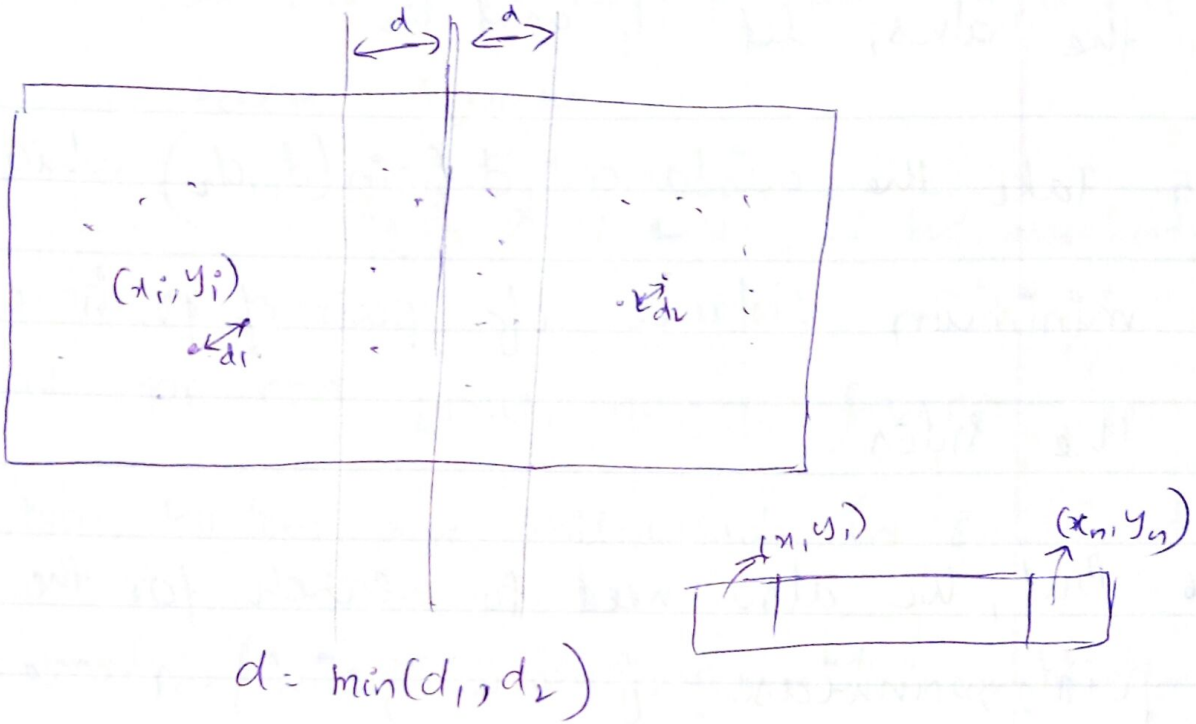
$$\Rightarrow T(n) = 3\,T(n/2) + \Theta(n)$$

$\Rightarrow$ Master's Theorem,

$$T(n) = \Theta\left(n^{\log_2^3}\right) \qquad 1 < \log_2^3 < 2$$

$$= \Theta\left(n^{1.55\ldots}\right)$$

Find Closest Pair of Points :-



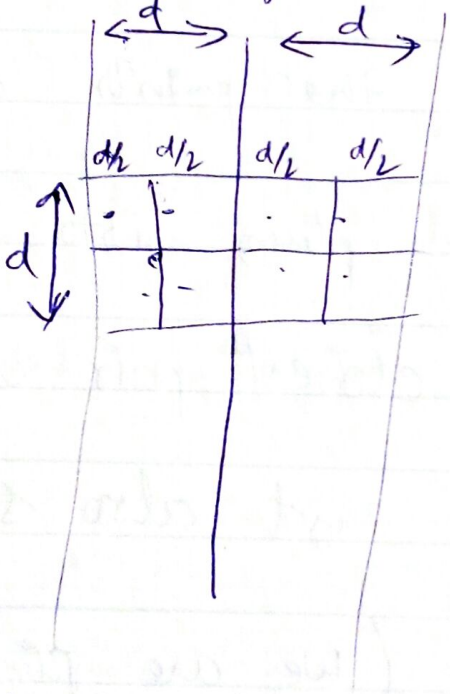
Steps:

1. First sort the points along the x-axis
2. Divide the plain into two halves
3. Find the closest pair of points in both the halves and also sort the points along the Y-axis (We are piggy-backing sorting on the recursion of finding closest pair of points).

4. Find the ~~pairing~~ closest pair of points on ~~the~~ both the halves, let be "d_1" and "d_2"

5. Take the distance, $d = \min(d_1, d_2)$ which is the minimum distance of pair of points of both the sides

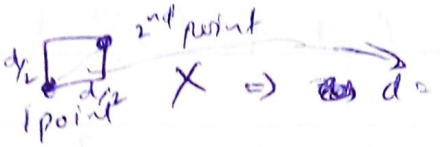
6. But, We also need to search for the pair of points, in the case of one point on one ~~side~~ of half and the other point on the other half



7) Now, you get the strip, which is formed by expanding a net of 'd' on one side & 'd' on the other side.

8 Divide the strip into $\frac{d}{2} \times \frac{d}{2}$ squares of side $d/2$. and

We know that, there cannot be more than one point, in each square.



$\Rightarrow$ ~~as~~ $d = 0.7d$ but, ~~d~~ already $\min, d = d$

9) Now, for each point, search for point, in the top, bottom and sidewards in 8-squares

10) Likewise, find the rolling minimum in the strip

11). ~~The~~ If we find a minimum, less than 'd' in the strip, then those points are the closest pair of points

FAST FOURIER TRANSFORM

$$A(x) = A_0 + A_1 x + A_2 x^2 + A_3 x^3 + \dots + A_n x^n$$

$$* \quad B(x) = B_0 + B_1 x + B_2 x^2 + B_3 x^3 + \dots + B_n x^n$$

$$C(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + \dots + C_{2n} x^{2n}$$

Ex:-

$$\begin{array}{r} 219 \\ * \quad 64 \\ \hline \end{array}$$

$$2 \times 10^2 + 1 \times 10^1 + 9 \times 10^0$$

$$* \quad 0 \times 10^2 + 6 \times 10^1 + 4 \times 10^0$$

$$8 \times 10^2 + 4 \times 10^1 + 36 \times 10^0 + 6 \times 10^2 + 54 \times 10^1 + 12 \times 10^3$$

$$\Rightarrow 14016$$

$\Rightarrow$ The above method is not efficient as, every digit has to ~~get~~ multiplied with 'n' other digits

$$\Rightarrow O(n \times n) = O(n^2)$$

$\rightarrow$ Instead of the above method, we can try a different method,

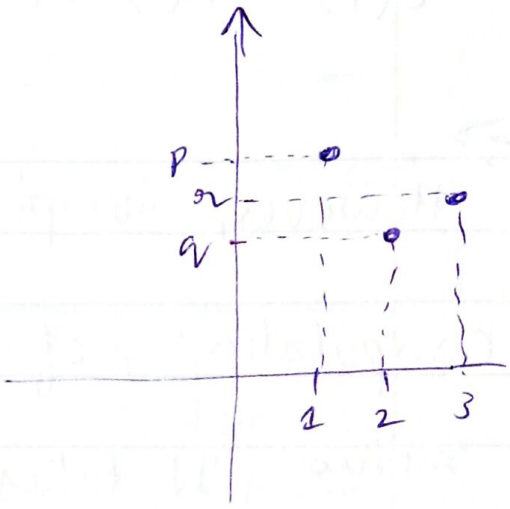
We can convert a entire polynomial into a graph

For ex:

$B(1) = p$

$B(2) = q$

$B(3) = r$

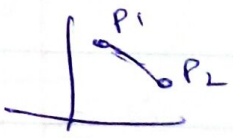


→ multiply, two graphs, 'B' and 'A' and then get 'C' graph "C"

→ From graph 'C', recover the co-efficients of 'polynomial 'C'.

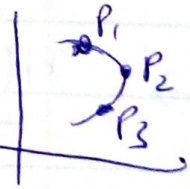
Ex:-

→ You need 2 points on graph, to recover a polynomial of degree 1 ; $y = mx + C$,



→ need 3 points on graph, to recover a polynomial of degree '2' ;

$y = B_0 + B_1 x + B_2 x^2$



→ need "2n" pts, to recover a poly of degree "2n-1"

$C(i) = A(i) \cdot B(i) \quad - O(1)$

⇒

However, to plot a point, we need to do the computation, of Polynomial using HORNER's method, It takes $O(n)$, complexity.

→ We need "$2n+1$" pts to compute a polynomial of degree "$2n$".

⇒ Complexity : $(2n+1)\,O(n)$

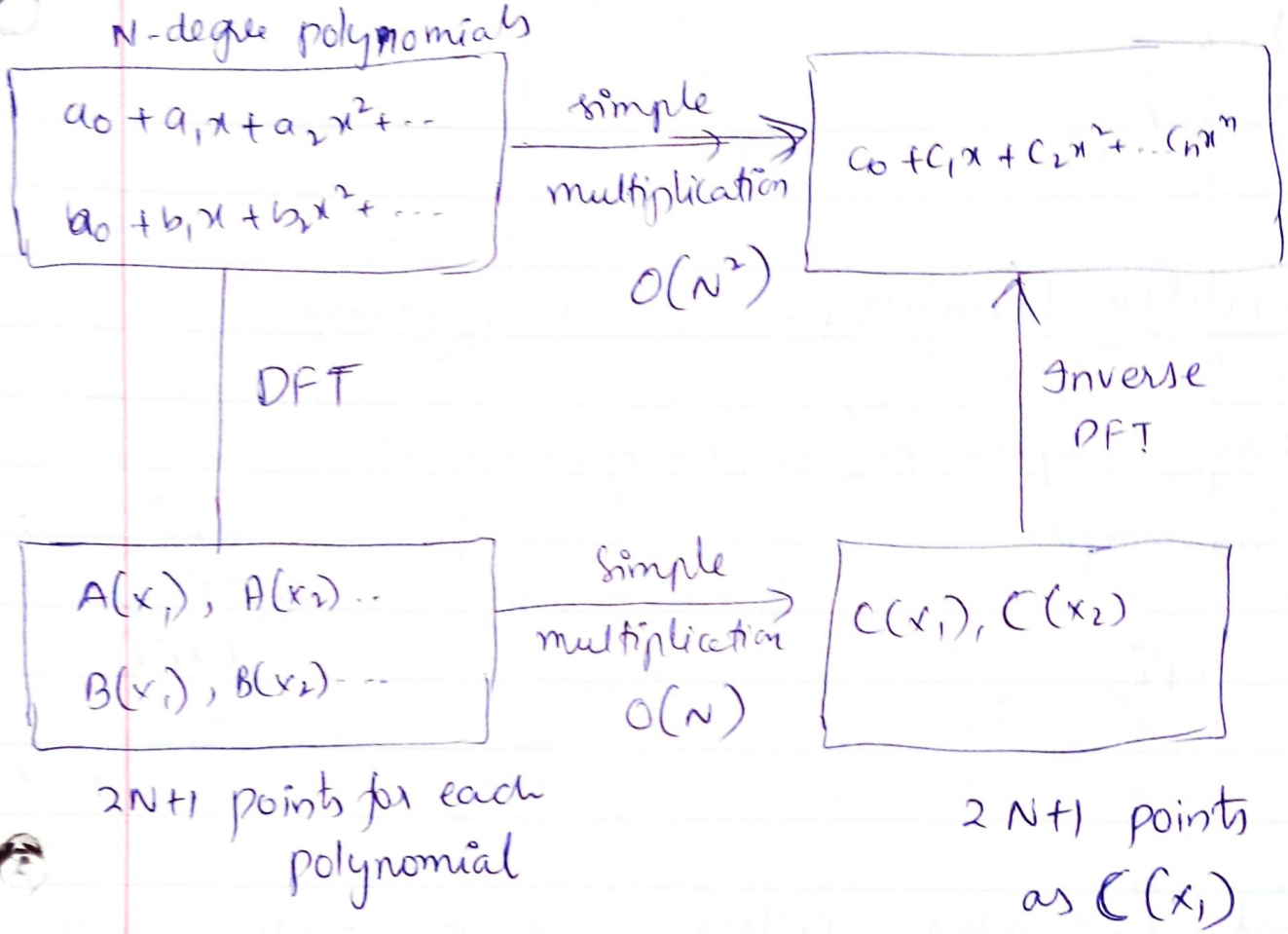
$= O(n^2)$

→ We used, Fourier Transform ~~from~~ for going from

• "CO-EFFICIENT FORM" TO "POINT-FORM"

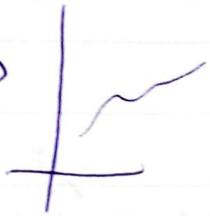
⇒ THERE COMES, "FAST FOURIER TRANSFORM"

FFT:-



$Y = A(x_1)$

$+ \; y = B(x_1)$ →



$C(x_1) = A(x_1) \cdot B(x_1)$

↓

→ For example,

$$A(x) = 30 + 21x + 0 \cdot x^2$$

$$\times \; B(x) = 1 + 3x + 2x^2$$

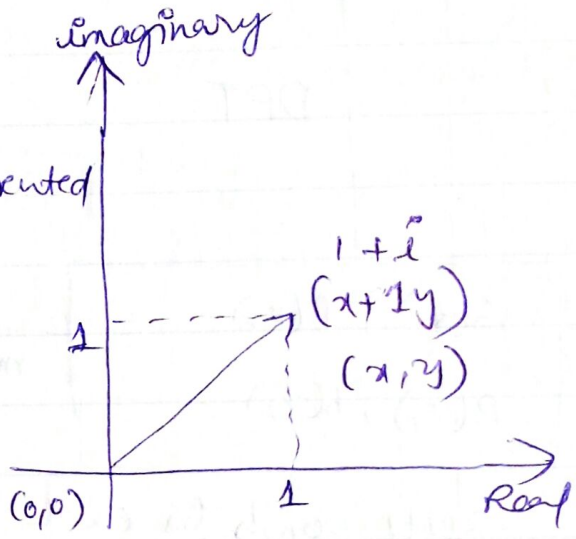
$$C(x) = 42x^3 + 123x^2 + 111x + 30$$

A(1) = 51

B(1) = 6

C(1) = 306 ;

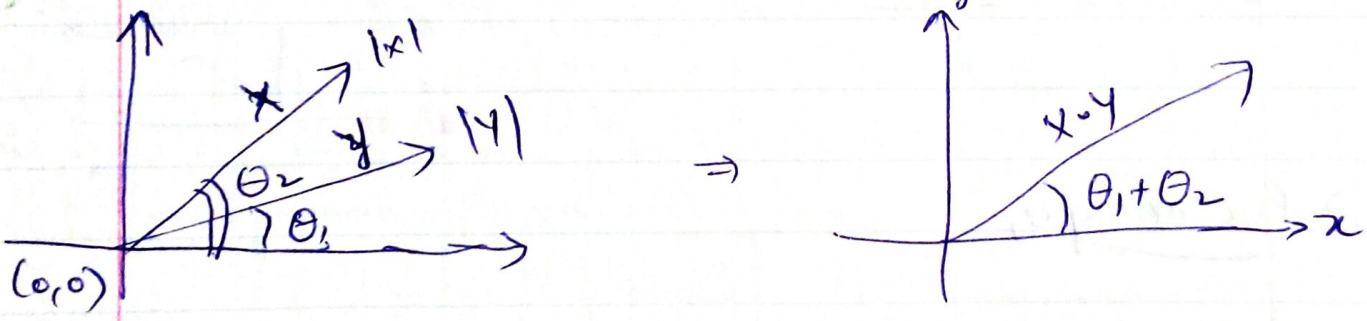
Complex Numbers:-

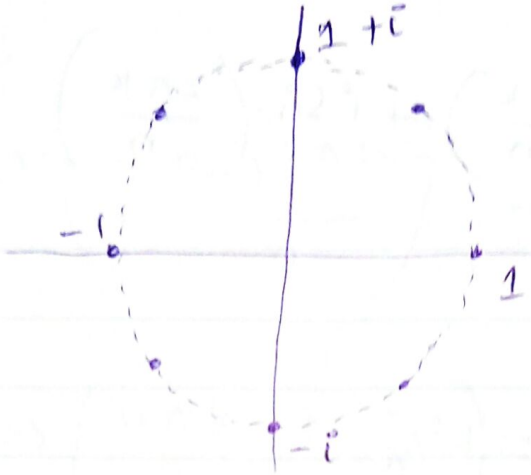


→ every complex no, can be represented on a graph

Ex:- $1+i$

what happens when we multiply two Complex Numbers?





$\theta = \theta_1 + \theta_2$

$\sqrt[n]{1}$ = n^{th} roots of unity

$\Downarrow$ $r e^{i\theta}$ → $r = 1$, unity circle $\Rightarrow 1 \cdot e^{i\theta}$

$e^{i\theta}$

4th roots of '1' (unity)

$\sqrt[4]{1}$ ⇒ $+1, -1, +i, -i$

$\sqrt[2]{1}$ → $+1, -1$

$\sqrt[2]{-1}$ → $+i, -i$

→

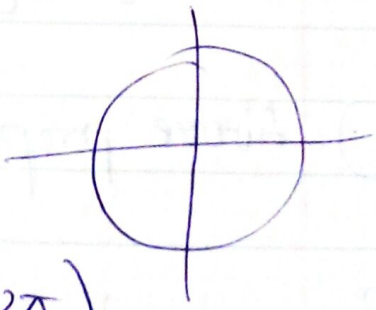
$A(x)$, $B(x)$

↓

'x' will be n^{th} roots of unity, complex no's

→

$$\omega_n = e^{2\pi i/n}$$



$$\omega_n = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right)$$

$k \rightarrow 0$ to $(n-1)$

$$\omega_n^k = \cos\left(\frac{2\pi k}{n}\right) + i\sin\left(\frac{2\pi k}{n}\right).$$

$\longrightarrow$

we have,

$$\omega_n^k = e^{2\pi k i/n}$$

$$\omega_{2n}^{2k} = \omega_n^k$$

proof $\Downarrow$

$$\cos\left(\frac{2\pi \cdot 2ik}{2n}\right) + i\sin\left(\frac{2\pi \cdot 2ik}{2n}\right)$$

$$= \cos\left(\frac{2\pi ik}{n}\right) + i\sin\left(\frac{2\pi ik}{n}\right)$$

$$= e^{\frac{i2\pi k}{n}}$$

(1) Above property is called "cancellation Lemma"

Property 2:

$$\left(\omega_n^{k+n/2}\right)^2 = \left(\omega_n^k\right)^2 \quad [\because n \text{ is even}]$$

L.H.S

$$\cos\left(\frac{(k+n/2)\,2\pi i\times 2}{n}\right) + i\sin(\theta)$$

$$= \cos\left(\frac{4\pi k + \frac{4\pi n}{2}}{n}\right) + i\sin(\theta)$$

$0, 2\pi, 4\pi$

$$= \cos\left(\frac{4\pi k}{n} + (2\pi)\right) + i\sin(\theta)$$

$$= \cos\left(\frac{4\pi k}{n}\right) + i\sin(\theta)$$

$$= e^{\frac{i4\pi k}{n}} = \left(e^{\frac{2\pi ik}{n}}\right)^2 = \left(\omega_n^k\right)^2$$

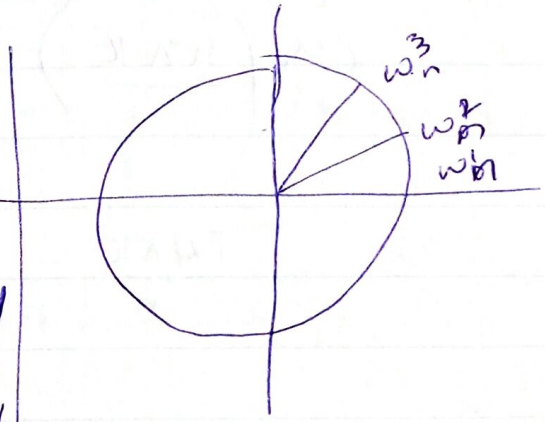
$$A(x) = \sum_{i=0}^{n-1} a_i x^i \quad \rightarrow n \text{ points}$$

$$2 + 0x + 3x^2 + 4x^3 + 0x^4 - 20x^5$$

$$\omega_n = e^{\frac{i2\pi}{n}}$$

① $$\left(\omega_n^{k+n/2}\right)^2 = \left(\omega_n^k\right)^2$$

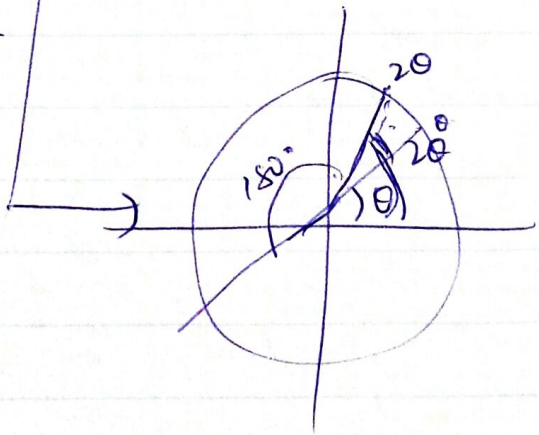
② $$\omega_{dn}^{dk} = \omega_n^k$$



It says, you are at angle, θ & move "$n/2$" parts ($1/2$ part) ahead and squared it.

$$(180+\theta) * 2 = (360+2\theta)$$

⇒ It is same as, you are at angle "θ", and, adding (or) moving to angle "2θ"



$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + \dots\dots + a_nx^n$$

Assume:

$n = 2^q$; perfect power of "2".

$$= \underbrace{\left(a_0x^0 + a_2x^2 + a_4x^4 + \dots a_{n-1}x^{n-1}\right)}_{\text{even}} +$$

$$\left(a_1x^1 + a_3x^3 + a_5x^5 + \dots a_nx^n\right)$$

$\downarrow$ odd

$$= \left(a_0x^0 + a_2x^2 + \dots a_nx^{n-1}\right) +$$

$$x\left(a_1x^0 + a_3x^2 + \dots a_nx^{n-1}\right)$$

$$A(x) = \underbrace{\text{even } A(x^2)}_{n/2} + x \cdot \underbrace{\text{odd } A(x^2)}_{n/2}$$

$A(\omega_n^0)$

$A(\omega_n^1)$

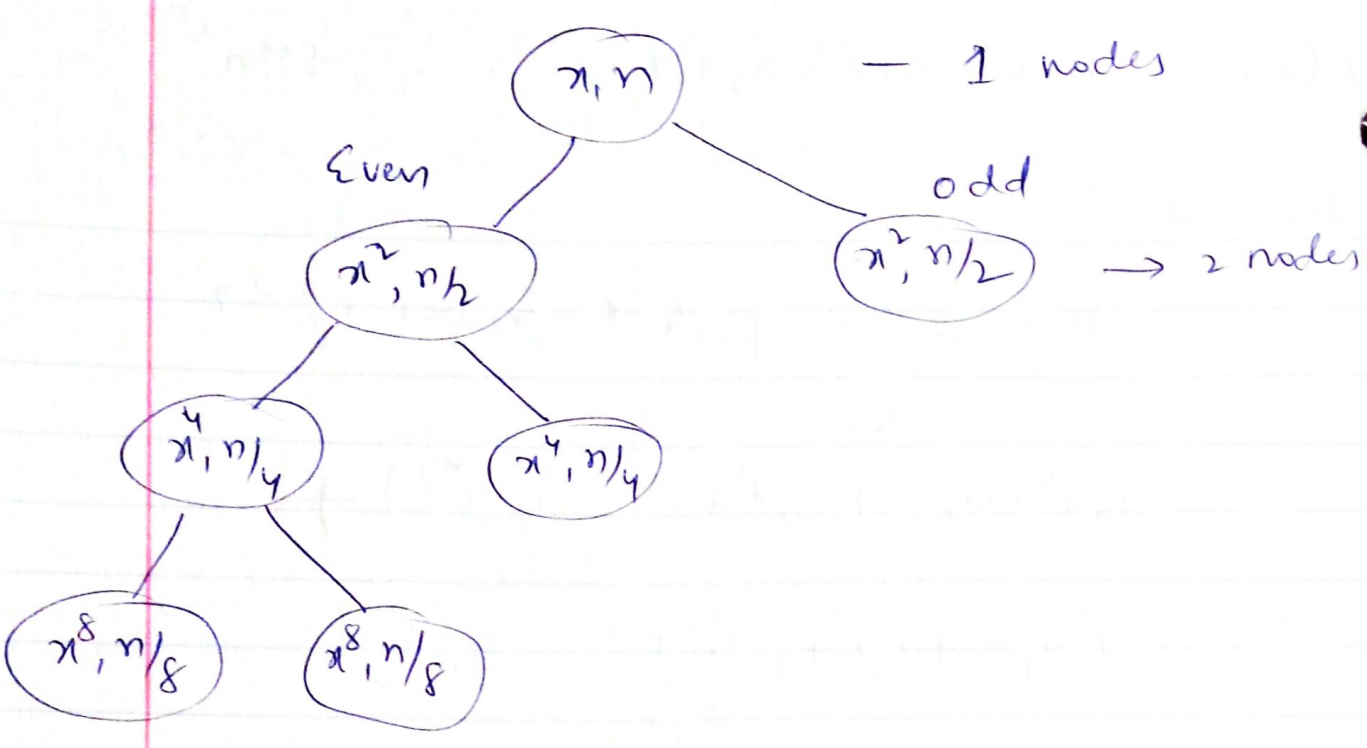
$A(\omega_n^2) \rightarrow \text{even } A\left((\omega_n^2)^2\right) + (\omega_n^2)\ \text{Odd } A\left((\omega_n^2)^2\right)$

$\rightarrow$ even even $A\left((\omega_n^2)^2\right)^2$ + $(\omega_n^2)^2$ even odd $A\left((\omega_n^2)^2\right)^2$

$A(\omega_n^3)$

$\vdots$

$A(\omega_n^n)$



$$A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$$

$$A(\omega_n^k) = A_{even}\left((\omega_n^k)^2\right) + \omega_n^k A_{odd}\left((\omega_n^k)^2\right)$$

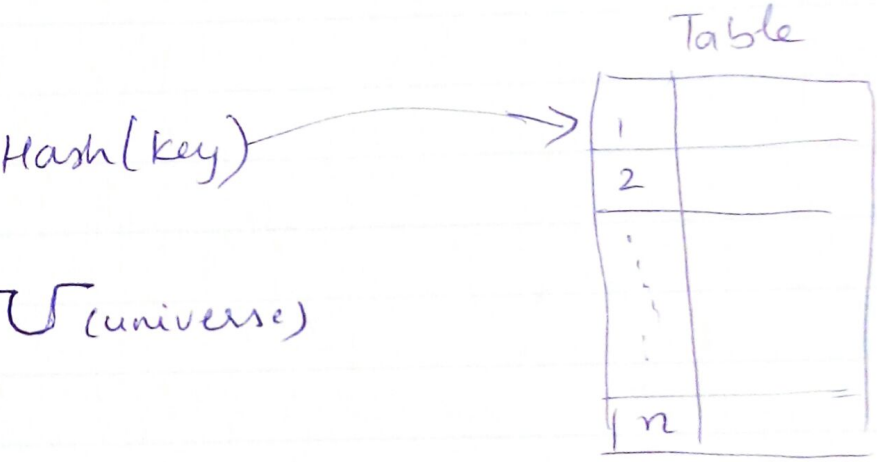
$$\left(\omega_n^k\right)^2 = \left(\omega_n^{k+n/2}\right)^2$$

	0°	30°	45°	60°	90°
sin	0	$1/2$	$\frac{1}{\sqrt{2}}$	$\sqrt{3}/2$	1
cos	1	$\sqrt{3}/2$	$1/\sqrt{2}$	$1/2$	0
Tan	0	$1/\sqrt{3}$	1	$\sqrt{3}$	∞

Hash Tables

→ Combination of <key, value> Pair

→ It has a hash function, when we hash a key using hash function, we get index in the table, where we store the data.



Operations

1) Search / look up
2) Insert
3) Delete

} Iterator, diff to implement

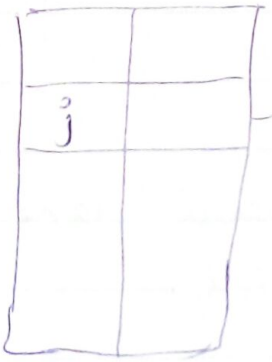
Ex:- Python dictionaries, Java Hash data struct
routers

How to deal with collisions?

1) a Live with it:- (Chaining)

⇒ When a collision happens, append the element at the end of the list.

2) It is called "Chaining"



chaining

→ Instead of a list, I can use a B.S.T (or Red. Black Tree etc:

(or)

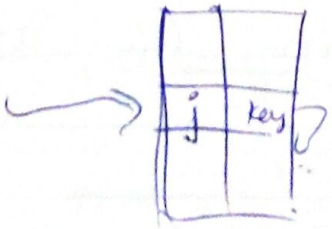
another Hash Table

↳ Multi-level Hash Table.

1 b) Open Addressing: (Python dictionaries)

⇒ If a collision happens, go to the next cell and insert the key

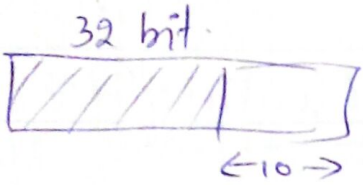
2) If you can't find a empty cell, keep on looking for empty cell, till you find one.

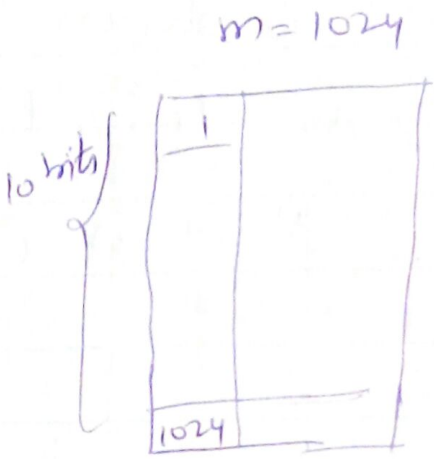


Second Strategy:

Be Smart about Hash function:

Key : 32 bit integers





→ Hash function should make collisions as minimum as possible

→ Good Hash functions: SHA, MD5, expensive, cryptographic algorithms

(OR)

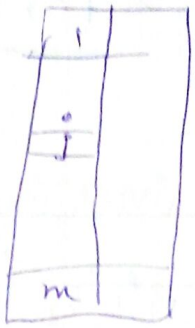
Randomise Hash functions:

→ In this, instead of single hash function choose a hash function from a family of hash functions, $H : \{h_1, h_2 \dots h_j \; h_k\}$

→ BUT after choosing One, you have to stick to it

Good Hash function:-

$\forall x, y \xrightarrow{x \neq y} Pr_H(h(x) = h(y)) = 1/m$

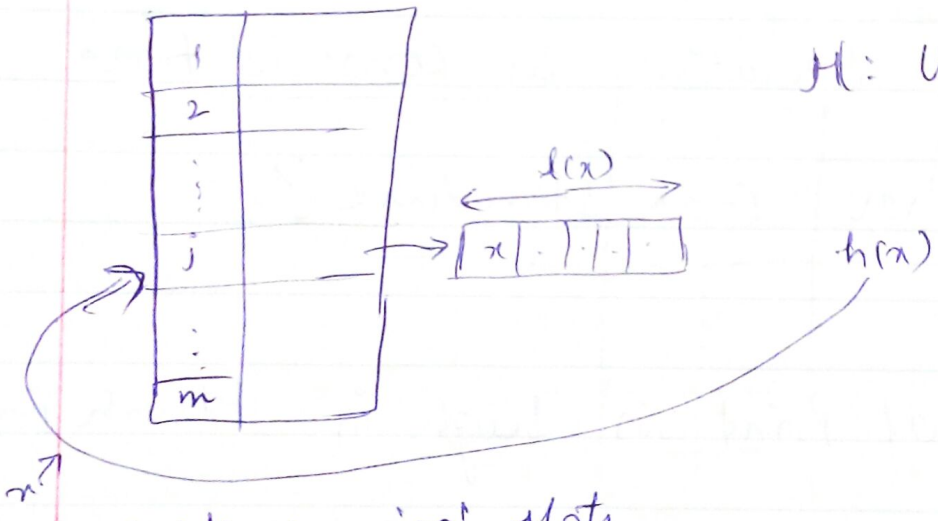


$\Rightarrow$ Universal Hash function

Near Universal

$\forall x, y \;\&\; x \neq y \quad Pr(h(x) = h(y)) \leq \frac{c}{m}$

Average Case Analysis:



H: Universal hash function

'n' keys, 'm' - slots

$\underbrace{x_1, x_2 \ldots x_n}_{n \text{ distinct keys}}$

$$E(l(x)) = \sum_{i=1}^{n} Pr(x_i \text{ collides with } x)$$

$$= \sum_{i=1}^{n} Pr(h(x_i) = h(x))$$

$$= \sum_{i=1}^{n} \frac{1}{m} \Rightarrow \boxed{\frac{n}{m} = \alpha, \text{ load factor}}$$

→ Load Factor:

How many keys "n" (or ratio) at which you are inserting into the hash table compared to slots, "m" available.

→ Ex: $n = 2m \Rightarrow \alpha$, load factor $= 2$

∴ So, Cost of lookup, delete, insert = $O(1) + \alpha$

* IF HASHTABLES, are efficient & we can perform any operation in constant time,

why Other Data structure?

⇓

* Memory foot Print is high in Hash Table
* Certain Operations are not easy on Hash Table like, finding minimum element, Second smallest element, Iterating Over Elements

Designing Universal Hash Function?

→ Classic Method :-

1) choose a Prime numbers, P greater than any key, you may encounter.

U : 32 bit numbers, then

$P = 2^{32} + 1$.

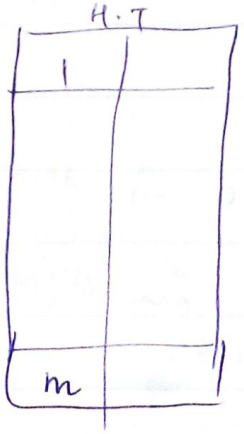
$h_a(x) : ((a \cdot x \bmod p) \bmod m)$ → To fit it into the hash table

a : salt

$a = \{1, \ldots, p-1\}$

↓

random number, choosen b/w 1 & (p-1)



why Universal hash function?

$h_a(x) = h_a(y)$?

Consider equation,

$a \cdot x \bmod p = z$

Given x, z what is a?

Ex:

$? \xrightarrow{3} * 3 \bmod 5 = 4$

for a fixed x, z, there is exactly one 'a'

a = modular inverse of $x \bmod p$

$a = x^{-1} z \bmod p$

$h_a(x) = (a x \bmod p) \bmod m$

$h_a(y) = (a y \bmod p) \bmod m$

$h_a(x) - h_a(y) = 0 \Rightarrow a(x-y) \bmod p) \bmod m = 0$

if $\bmod m = 0 \Rightarrow$ then,

$a(x-y) \bmod p \Rightarrow$ { 0, m, 2m, 3m, ..., km }

$km = \rightarrow$ should be less than 'p', as if becomes greater than '1' it wraps around

$ik = \left\lfloor \frac{p}{m} \right\rfloor$

$\Rightarrow$ &c $\{0,$

$\Rightarrow$ If $a(x-y) \bmod p = 0$, then

$a = \{1, 2, \ldots p-1\}$ ✗ [as 'p' prime number, so cannot be '0']

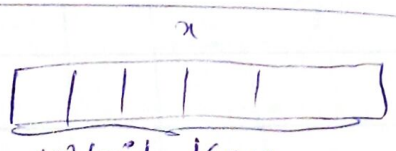
$\Rightarrow a(\text{blah}) \bmod p = m$

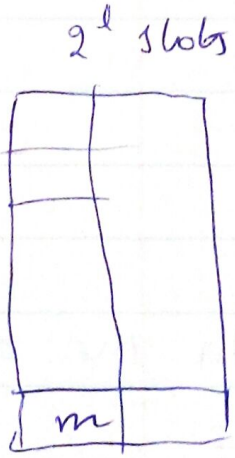
$\Rightarrow a(x-y) \bmod p = \{$ 0 ✗ — $x \neq y$; $m, 2m, \ldots km$ } k choices

$a = \{1, 2, 3 \ldots p-1\}$

Pr of collision $= \frac{k}{p} = \left\lfloor \frac{p}{m} \right\rfloor \cdot \frac{1}{p} \leq 1/m$

Near Universal Hash Function:-

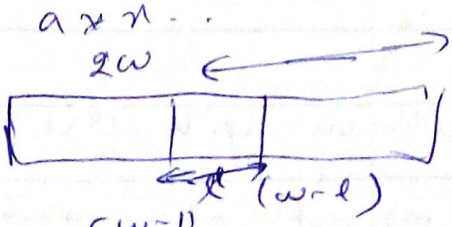




odd number

$a =$ 

$= \left\lfloor \frac{a\, w \bmod 2^w}{2^{w-l}} \right\rfloor$



If, we use above function, Pr of collisions

$$Pr\left(h_a(x) = h_a(y)\right) \leq 2/m$$

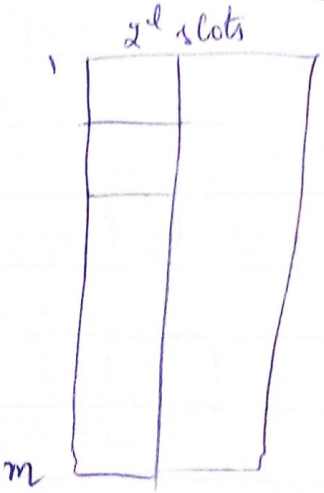
Explanation:-

① Take a key, x of size 'w' bits

② Choose a salt, 'a', (odd number) also of size 'w' bit

③ Consider of a hash Table of size = 2^l slots

$$h_a(x) = \left\lfloor \frac{a \cdot x \bmod 2^w}{2^{w-l}} \right\rfloor$$



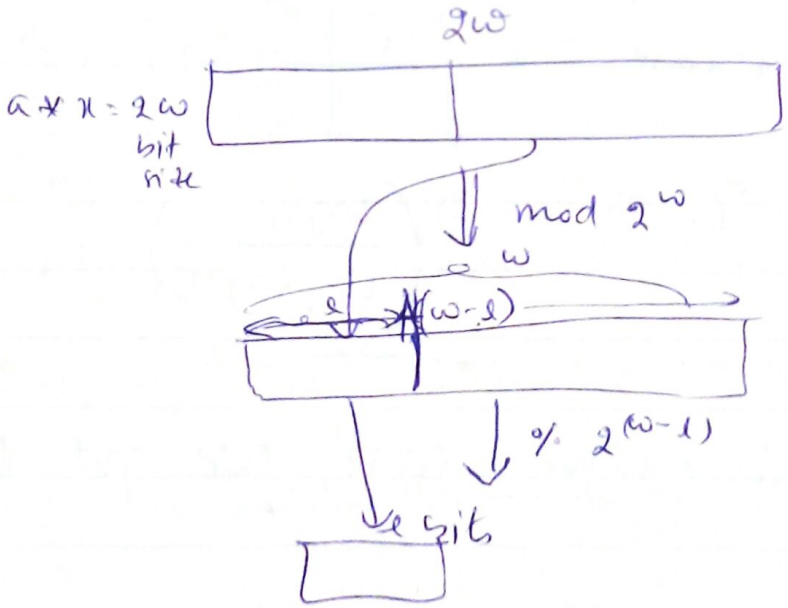
④ In the above function, we multiply 'a' with 'x' to get '2w' bit size word.

⑤ when you do "mod 2^w", you ~~Consider only~~ chop of "w" most significant bits and

6) ~~Consider only~~ by doing a right shift (or) bit masking

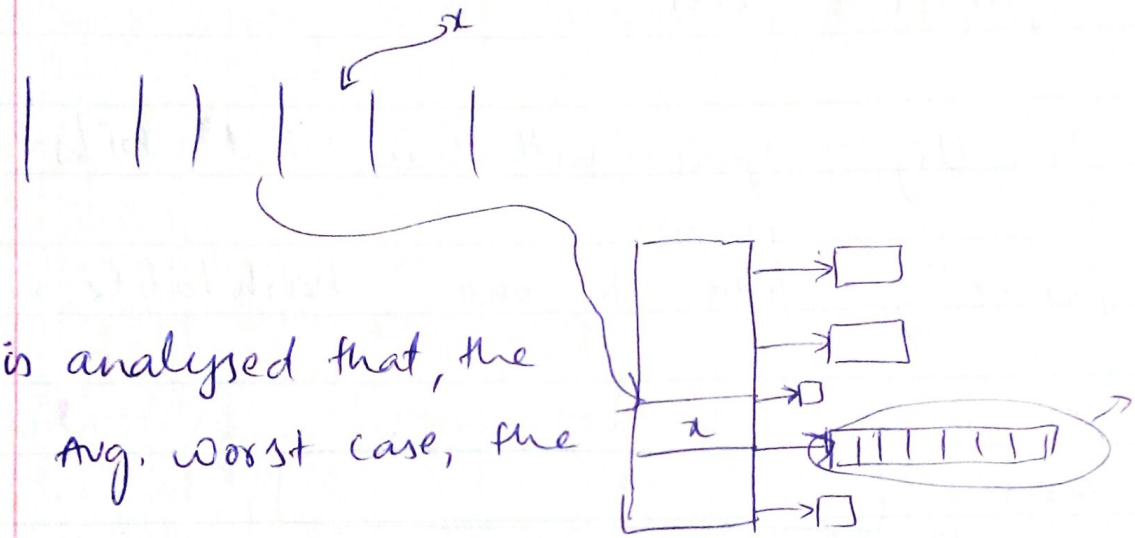
7) In the remaining 'w' bits by dividing with '2^{w-s}', you chop of $(w-s)$ least significant bits

(7) Finally you will have 's' bits which perfectly fits (maps) to the hashtable, slot



Perfect Hashing :-

In the hashing mechanism, when we use chaining because of collisions, the list for a particular key might grow much bigger.

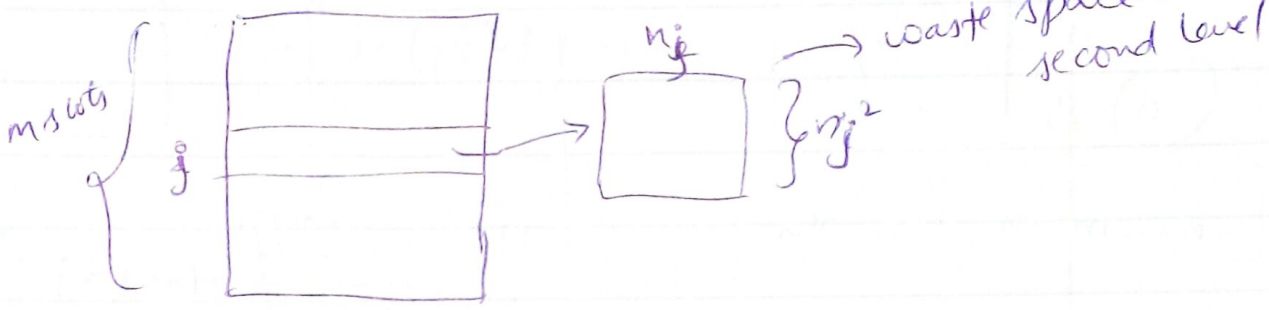


→ It is analysed that, the in the Avg. worst case, the list can be big of size = $\Theta\left(\frac{\log n}{\log \log n}\right)$

→ TO HANDLE Above Case, We got Perfect HASHING

→ It is based on following observation, if, I have 'n' items, & "$n^2+\varepsilon$" slots, it is Very Highly Probable that '0' collisions happen.

→ with the above observation, we design a second level Hash Table



→ whether we get savings if we do at second level?

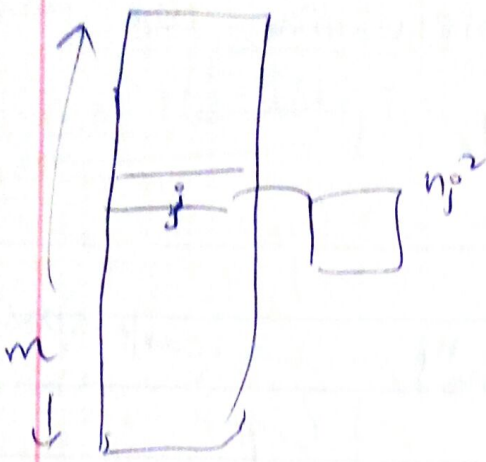
secondary level space = $(n_1^2 + n_2^2 + \dots + n_m^2)$.

→ Process:

At first we have secondary table of fixed size, n, then when the threshold is reached we increase the size of the table.

Ex:- First, we have a table of size, 64, (8^2).

We are good till 'q' collisions, but when the q^{th} collision happens, we increase the table size to "16^2" 256"



Expected summation =

$$\notin \left(\sum_{j=1}^{m} n_j^2 \right)$$

$$\left[h(x_i) = j \right]$$

$$n_j = \sum_{i=1}^{n} \left\{ \left[h(x_i) = j \right] \right\}$$

↳ Indicator variable, it becomes "1" when cond^n is satisfied

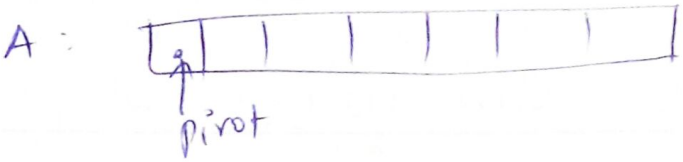
$$\notin = \left(\sum_{j=1}^{m} n_j^2 \right)$$

$$(n_j)^2 = \left(\sum_{i=1}^{n} \left[h(x_i) = j \right] \right)^2$$

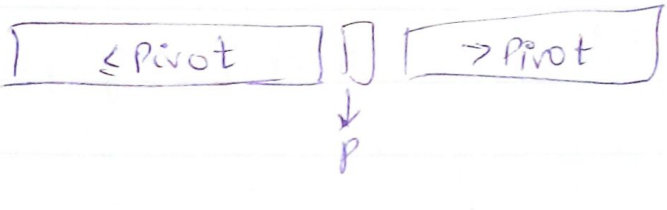
$$= \sum_{j=1}^{m} \left(\sum_{i=1}^{n} \left[h(x_i) = j \right] \right)^2$$

$$= n + 2\alpha (n-1)/2$$

Quick Sort: -> Sir Tony Hoare



Inplace



Complexity

w.c = $\Theta(n^2)$

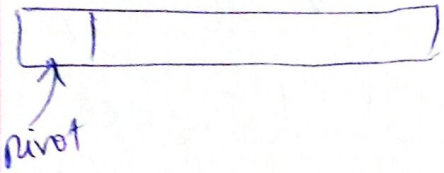
Trick in Selecting Pivot:-

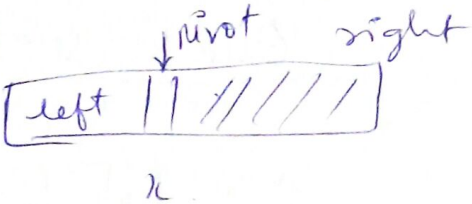
1. Randomisation → Expected $\Theta(n \log_2 n)$
2. Median of Medians → w.c => $\Theta(n \log n)$ → deterministic

Quick Select:

k^{th} smallest element

1. Take a pivot and place it in its final position
2. If the index of the pivot $\leq k$, then look in the left sub-array
3. If the index of the pivot $\geq k$, then look in the right sub-array





Deterministic Pivot:- A[i] is my pivot, fixed pivot

Randomisation:

Treaps

Deterministic:

median of medians.

Deterministically select my almost magic Pivot

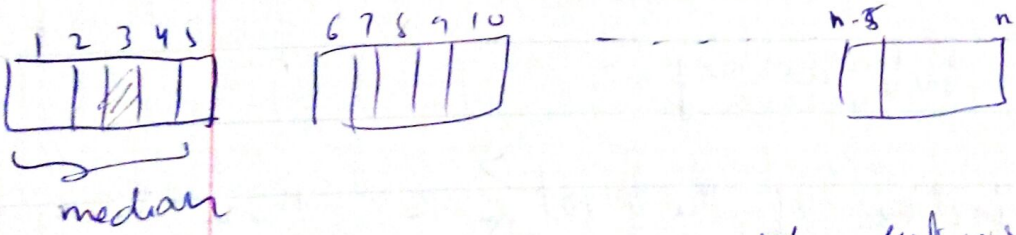
Fractional guarantee:-

at least 1% of elements on left | at most 99% of elements on right,

Median of Medians:-

Algorithm for selecting a pivot

* select a pivot → $\theta(n)$
* Partition - $\theta(n)$
* Recurse



n/5 sub arrays

Steps to Select Pivot:

(1) Divide the array into sub-arrays of size = 5

(2) Compute median of each sub-array: $25 \times n/5$ (5^2) → $\Theta(n)$

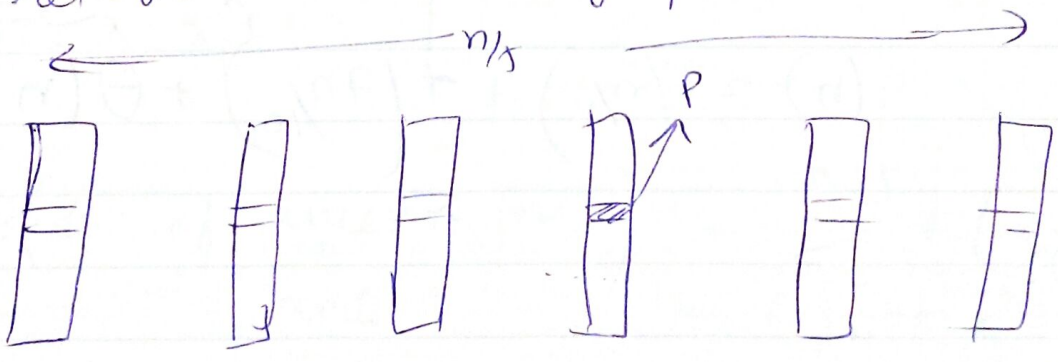
(3) ~~Recus~~ Collect the "$n/5$" ~~array~~ medians into a new array, M - $\Theta(n)$

4) Recursively call quick select to find:

Median of M ⇒ "p"

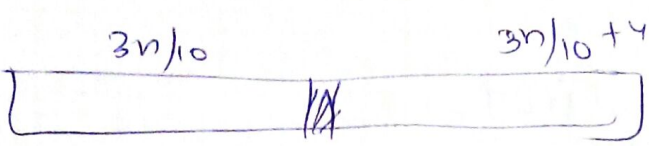
(This steps also uses median of median trick to select pivot)

(5) return "p" as my pivot



Count: how many elts are for sure $> p$: $3n/10$

for sure $< p$: $3n/10$



P : almost magical

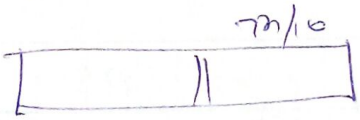
at least 30%.
at most 70%.

Quick Select using mofm Pivot Selection

1) Select a pivot 'p'
   - ① Find median array
   - ② Quick select on median array
   - ③ Return pivot

② Partition acc to p : $\Theta(n)$

③ Recurse on the appropriate part



$$T(n) = T(n/5) + T\left(\frac{7n}{10}\right) + \Theta(n)$$

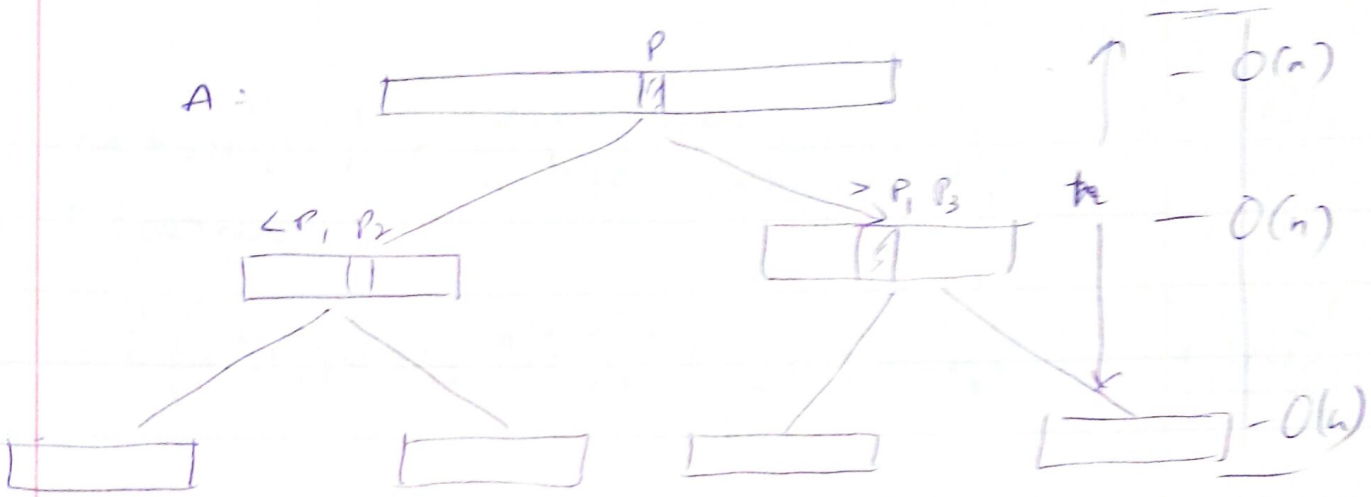
all $\Theta(n)$'s

BAD:

① we are copying a array into multiple small arrays

②

Randomised Pivot Selection :-



O(nh) → on avg, what is the height of a random BST

Height of a Random Binary Search Tree :-

{ 1, 2, 3, 4, 5, 6, 7 } --- n }
k
P1 P2 P3 P4 ..... Pn

$$X_{i,k} = \begin{cases} 1 & \text{if } i \text{ is an ancestor} \\ 0 & \text{o.w} \end{cases}$$

$$E(\#\text{ of ancestor for node } k) = \sum_{i=1, i\neq k}^{n} E(X_{i,k})$$

Pr(i is an ancestor of k)

{ i, i+1 ... i+k-1, k }
Pi Pi+1 Pi+k-1 Pk

P_i must be the lowest priority.

Case 1: 'k' is the root, ⇒ 'i' is not an ancestor

Case 2: 'i' is the root ⇒ 'i' is an ancestor

Case 3: 'i' & 'k' are in diff. sub tree
⇒ 'i' is not an ancestor

Case 4: 'i' & 'k' in same sub-tree

If $P_i < P_k$ then; 'i' is the ancestor of 'k'

Take, set

$\{ i, i+1, i+2 \ldots i+k-1, k \}$

$P_i \; P_{i+1} \; P_{i+2} \quad P_{i+k-1} \; P_k$

Pr(i is an ancestor of 'k') =

= 'i' should be minimum & permute remaining elements

$$= \frac{(k-i)!}{(k-i+1)!} = \frac{1}{k-i+1} \quad \text{if } k > i$$

if $k > i$, then $\frac{1}{k-i+1}$

$k < i$ then $\frac{1}{i-k+1}$

$\therefore \#(\#\text{ of ancestors of } k) = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1}$

$= H_k + H_{n-k+1} - 1$

$= \Theta(\log k) + \Theta(\log(n-k+1))$

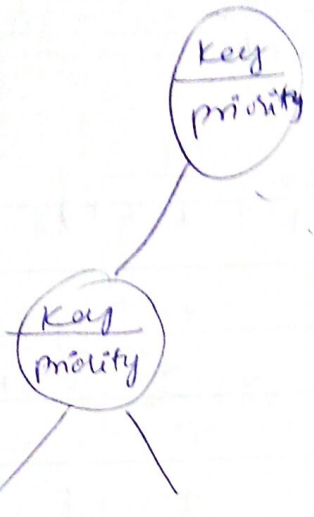
$\leq \log n + \log n$

$\leq 2\log n$

$\longrightarrow$

"i" is ancestor "k" iff "i" is the least common ancestor of "k".

Treap:-



→ B.S.T on the keys

→ Min Heaps on priorities

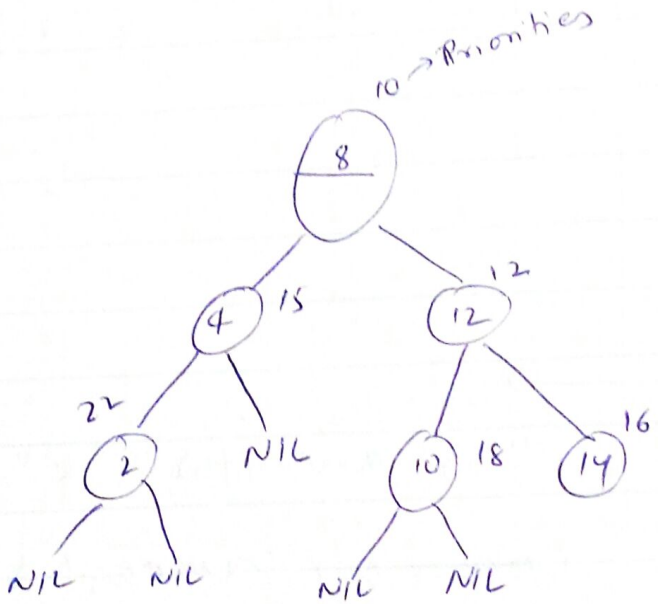
→ It is a B.S.T and if priorities are random, then it is random B.S.T

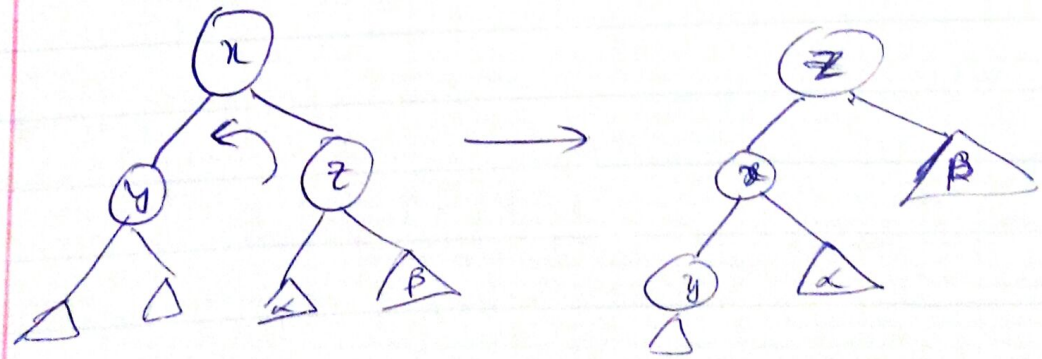
insert (key)

insert (14)

1) generate a random priority

2) BST insert

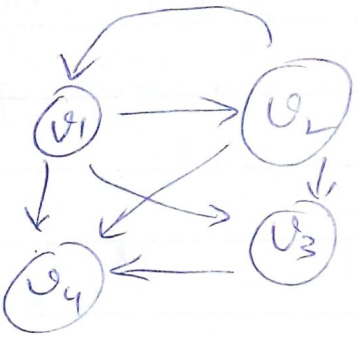




Introduction to Graphs

Graphs:

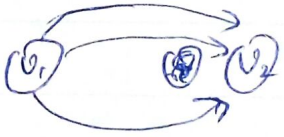
Binary relation over a finite set of nodes / vertices



$V = \{v_1, v_2, v_3, v_4\}$

$E = \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)\}$

Not considered in classes

1. loop over a vertex –  ×
2. Multiedges  ×

Other concepts

1. Directed 
2. Undirected  (bi-directional)
   ↳ symmetric relation.

where do graphs come from?

1. Computer network (vertices - servers
   Edges - Links)
2. Social networks (Vertices - People
   Edges - relationship
3. Ecological networks (Prey, Predators)
4. Electrical circuits (Nodes, - junctions
   edges - connections)
5. Programs (Nodes - control locations
   edges - control flow
   etc)
6. Roads. (Maps)

Problems:

1) Traversal
2) Identifying cycles
3) Ranking nodes
4) finding shortest paths
5) Finding spanning trees
6) Flows.

Flow

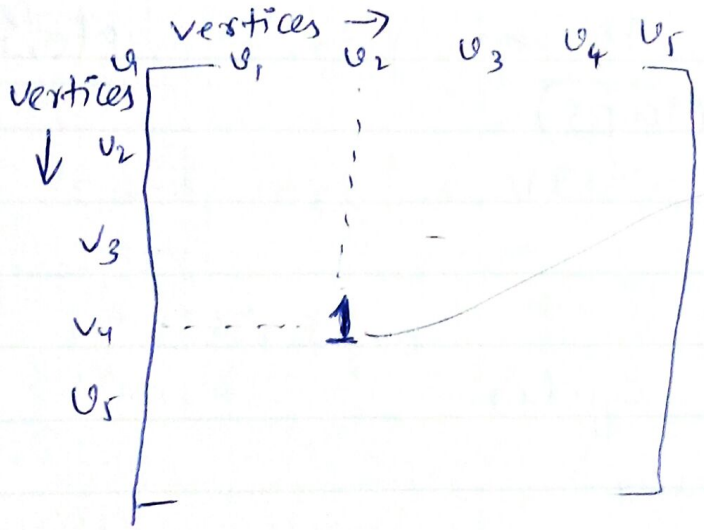
How to represent graphs & run Algorithms on graphs :-

1) How do represent graphs on a Computer?

: Many representations
- complexity of algorithms depend on representation

- Types:

① Adjacency Matrix:



→ matrices of 0's & 1's

→ edge from $(v_4 \to v_2)$

→ If there are k, vertices then size of representation is $|k|^2$

→ Graphs have many vertices, less edges.

Ex:- facebook, 1 billion ppl, how many friends per each?

Comp. Networks, many services, but all are not connected

Ecological n/w: thousands of species, but everyone will not have a prey, predator relationship

→ Graphs in the world have "sparsity" structure

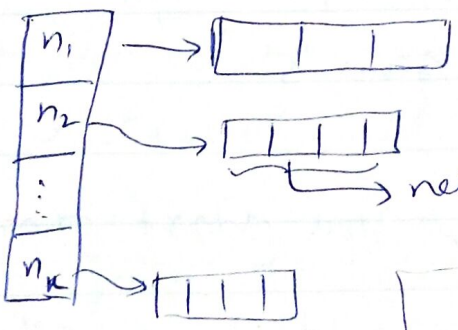
→ Max connections (edges) a graph of 'k' nodes can have is "$k(k-1)$" ~~Edges~~

→ So, in adjacency matrix, lots of elements will be zeroes and will be wasting space.

→ So, we use a Diff representation called

Adjacency List Representation:-

1) Every node in set of nodes, points to its neighbours



'k' nodes, $V = \{v_1, v_2, v_3 \dots v_4\}$

'm' edges, $E = \{(v_1, v_2), (v_2, v_1), (v_1, v_3)\}$

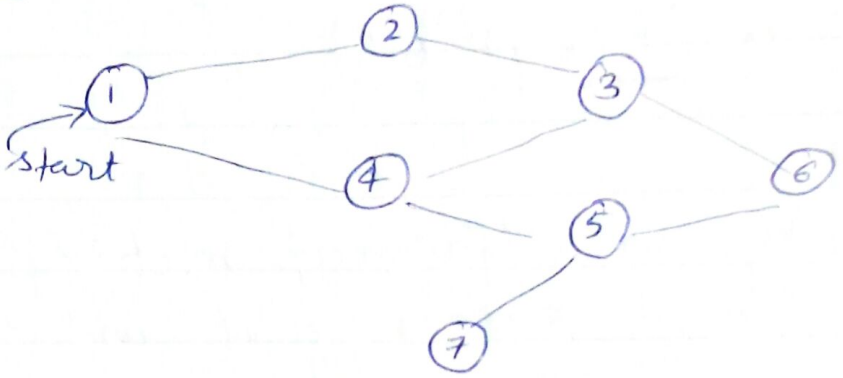
∴ Total no of cells $\{K + M\}$ (θ) $|V| + |E|$.

Other Representations:

3) Incedency Matrix Representation

Breadth First Search:-

Graph Traversal:



Traversal:

① Visits nodes in the graph in some order but can visit a node ~~only~~ atmost once.

② If I visit a new node, it must be along an edge starting from an already visited node

Ex:- In the above ex, if I'm visiting '6', I should have already seen '5'

Ex:- You are Marco Polo, you start from one port & traverse to other ports and add it to the known world

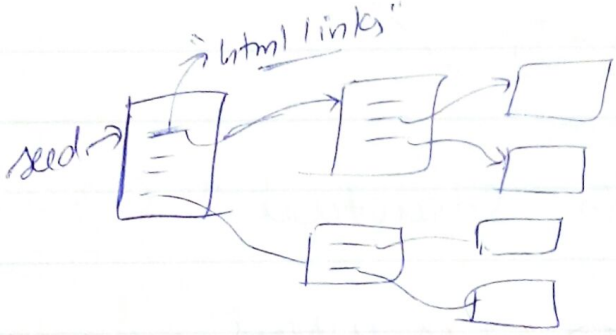
Why Traversals?

Google:-

→ crawling the web

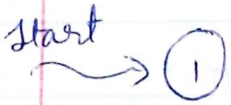
→ It has some "seed" web pages, and then traverses to the other web pages but not traverses more than once.

(Imagine you are searching for a web-page & it will not show up more than once in your search)



BFS:-

Use a FIFO Queue (FQ)



① Start from node ①

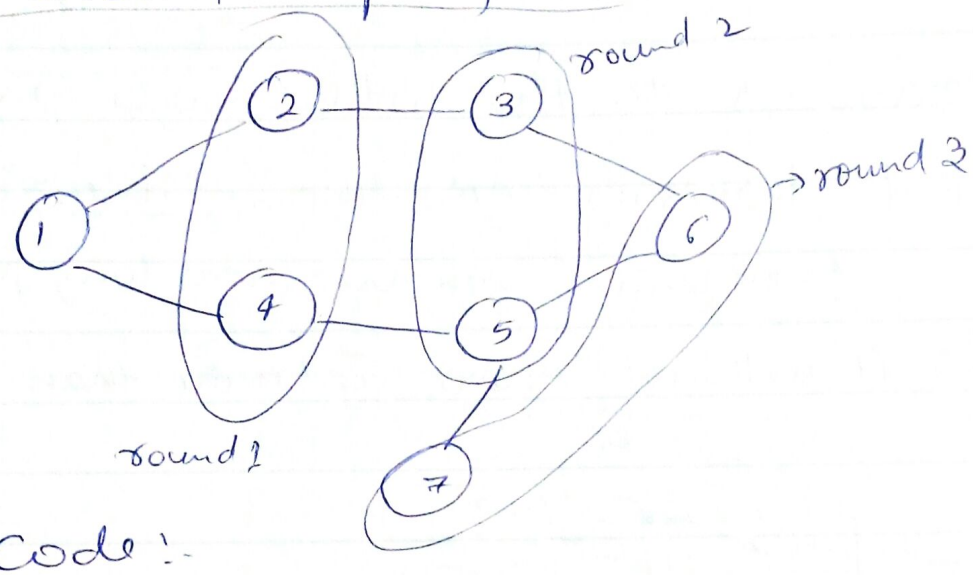
② Add successors of node ① to FIFO Queue (FQ)

③ then pick the element (e,) from FQ, according FQ, property

④ Add the successors of (e,) to FQ, according FIFO Property

5. Then record (or) store the fact that you already visited, i.e.,

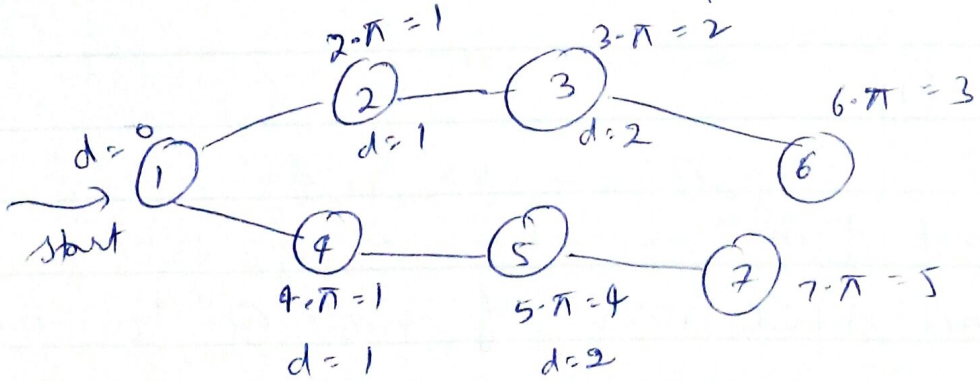
→ Traversal Graph for BFS



→ Code :-

⇒ "Queue Data Structure" is used.

→ v. $\begin{cases} \pi \rightsquigarrow \text{bfs parent} \\ d \rightsquigarrow \text{depth of the node} \\ \text{visited} \rightsquigarrow \text{True / False} \end{cases}$



Pseudo Code:

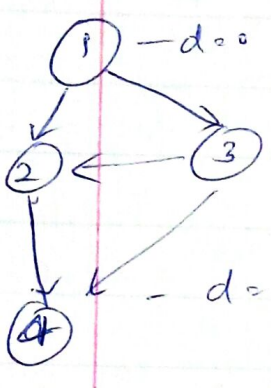
Graph ↑ start node →

FIFO Queue

```
def bfs(G, s):
    Q ← {s}
    s.d ← 0
    s.seen ← True
    s.π ← NIL
    while (Q ≠ ϕ)
        u ← dequeue(Q)
        for all v ∈ Adj(u, G)
            if (!v.visited) → Or v.seen,
                v.d ← u.d + 1
                v.π ← u
                v.seen ← True
                enqueue(v, Q)
```

Depth First Search (DFS):-

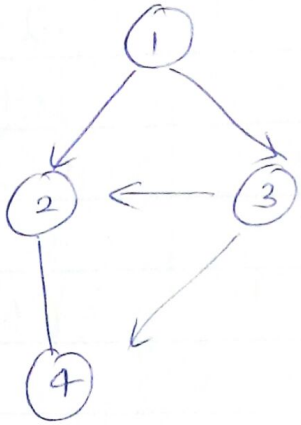
BFS:



Start: 1
2
3
4

Queue : FIFO

DFS:

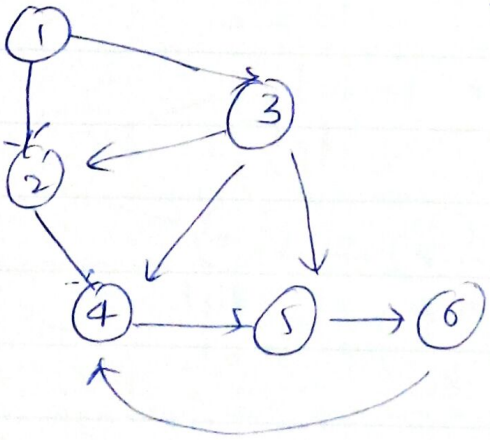


Start: 1
2
4
3

→ It seeks deeper & deeper nodes & then backtracks to the other nodes of same level, where it started backtracking

LIFO

Ex:-



Time stamp

dfs visit

1 visit(1)
2 visit(2)
3 visit(4)
4 visit(5)
5 visit(6)
6 visit(3)

attributes of node

v { π := parent UNDEF
d := discovery time UNDEF
f := finish Time UNDEF
seen := FALSE UNDEF

all the above "attributes" are useful for book keeping

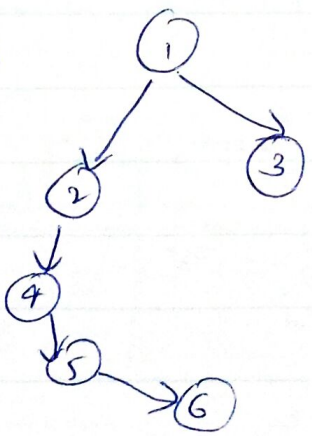
global time := 1

```
dfsVisit (G, u):
    if (u.seen):
        return
    time := time + 1;
    u.d := time
    for all { v ∈ Adj(u)
        if (not v.seen)
            v.seen := TRUE
            v.π := u
            dfsVisit(G, v)
    time := time + 1
    u.f := time
```

} record when I finished

→ When we do the DF-traversal, it creates a Depth first Search Tree.

For above ex:

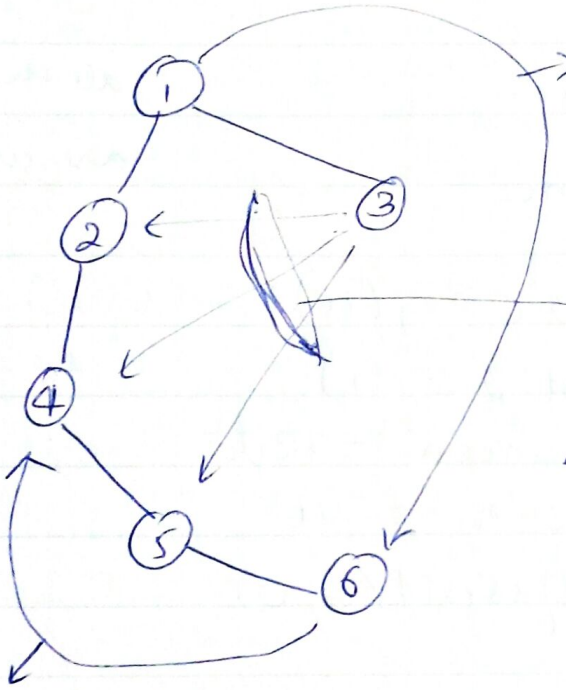


→ Every edge in DFS Tree is a edge of DFS graph

→ This edges cannot have cycles.

→ There are some names to the edges which are present in original graph, but not in DFS Tree:

Ex:-



→ forward edge: from ancestor to the descendant

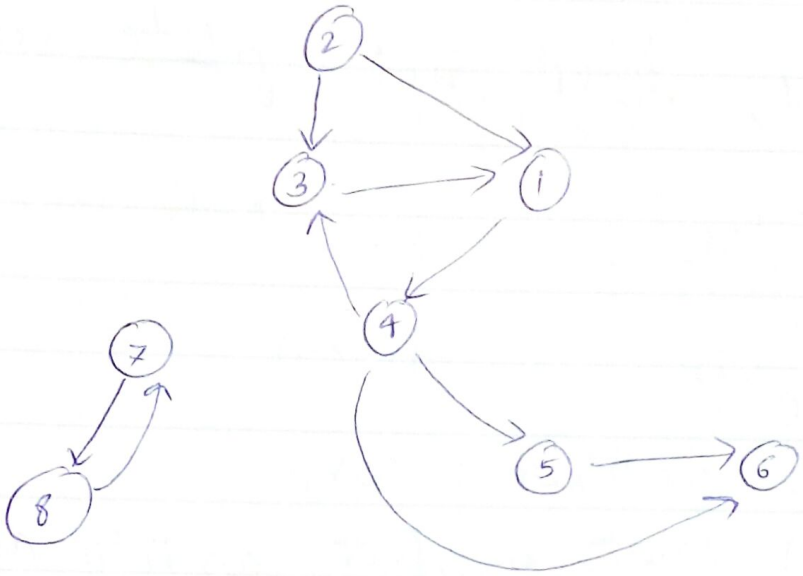
→ cross edges: either from siblings (or) ~~direct~~ immediate ~~ancestors~~. ?

backward edge:-
from descendant to ancestor

Example of DFS:

Graph

DFS on a unconnected graph



Outer loop on Nodes

for i=1 to n ← # of nodes

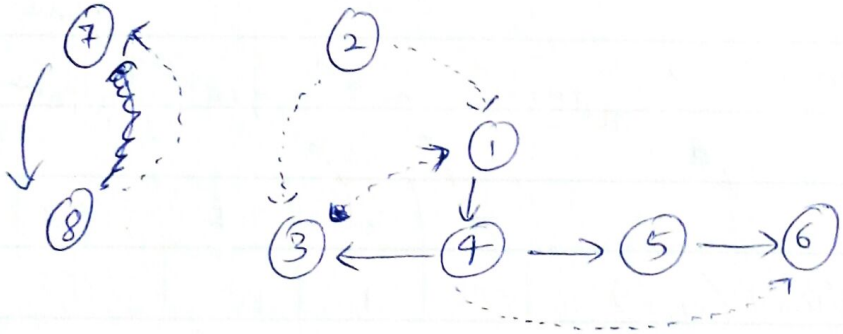
dfsVisit(G, i)

	π (parent)	d (discovery time)	f (finished time)	seen
①	NIL	1	10	T
2	NIL	11	12	T
3	④	3	4	T
4	①	2	9	T
5	④	5	8	T
6	⑤	6	7	T
7	NIL	13	16	T
8	⑦	14	15	T

→ when you run a outer loop, it runs on all the nodes, but as the seen will be updated to "TRUE" for all nodes which got visited, it'll not ~~update~~ run the function.

DFS Tree ?

→ It's no longer a tree, as it'll not be connected



? From table, check the parents and create the ~~graph~~, tree.

→ Edges, which are there in original graph but not in DFS Tree are shown using "....." (Dotted lines)

Back Edges: Desc to Ancestor in a Tree

Ex:- (3,1) (8,7)

Fwd Edges: Ancestor to desc, which is not a child

(4,6)

Cross Edges: Edges b/w different Trees in the Forest (or) b/w nodes inside the same tree, neither a ancestor nor a descendant of the target node

(2,3), (2,1)

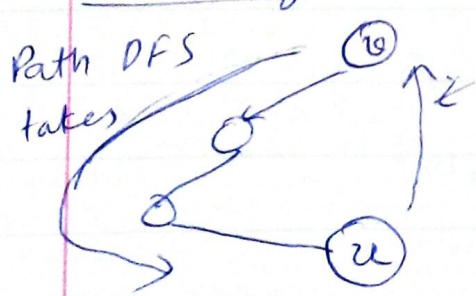
Tree Edges: Edges along DFS proceeds

Forest:-

Collection of different Trees.

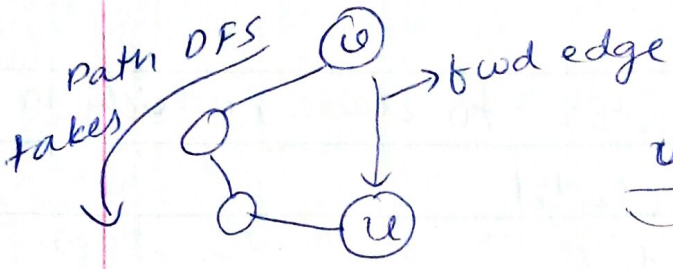
Classification of edges based on Discovery and Finish Time:

→ Back edge:-



$v.d < u.d < u.f < v.f$

forward Edge:



v's interval

$u.d < v.d < v.f < u.f$

u's interval

Cross Edge :-

$[u.d, u.f]$ is disjoint $[v.d, v.f]$

↓ u's interval — v's interval.

Ex:- (2,1)

2's interval = [11,12]

1's interval = [1,10]

on the other hand,

fwd edge :- (4,6)

4's interval : [2,9] → big interval

6's interval : [6,7] → small interval

Back edge: (3,1)

3's interval : [3,4]

1's interval : [1,10]

Does a graph have a Cycle?

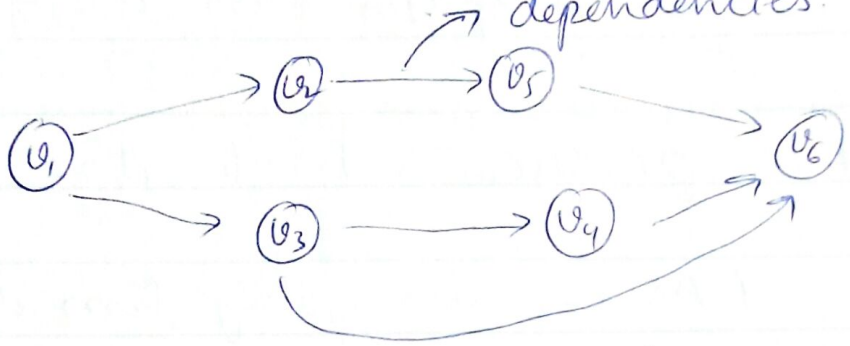
Theorem:- A graph has a cycle iff a DFS produces a "Back edge".

(As we are going from descendant to ancestor),

DFS Notes:-

Topological Sort :-

Directed Acyclic Graph (DAG)

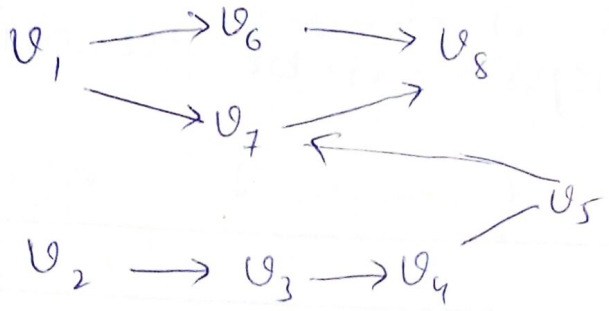


v_1
⋮
v_6
} Tasks

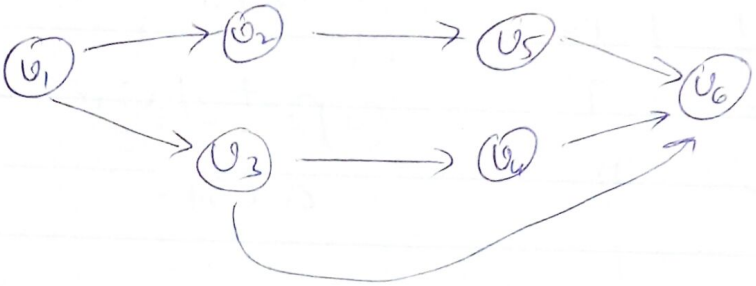
v_2 can only be started after v_1 is finished

Making a Tea:

- Take a cup ..... v_1
- Take a kettle ..... v_2
- Place kettle on boiler ..... v_3
- Add water to kettle ..... v_4
- Bring water to boil ..... v_5
- Place Teabag ..... v_6
- Pour water in cup ..... v_7
- Add ~~sugar~~ milk ..... v_8
- ~~Add~~



Ex: (1)



v_1 v_2 v_5 v_3 v_4 v_6

Topological sort

Place the vertices in ascending order such that every edges goes from lower ranked vertex to higher ranked vertex.

(*) Can be done only on DAG, graphs.

STEPS:

(1) DFS

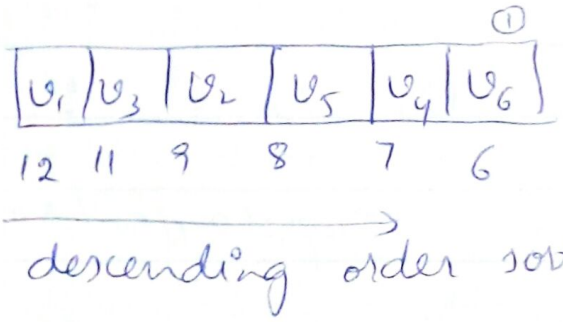
(2) Sort the nodes in descending order according to their finish times

→ The above steps can be done in a single fold.

Above Ex (1) :-

	d	f
v_1	1	12
v_2	2	9
v_3	10	11
v_4	4	7
v_5	3	8
v_6	5	6

① As a node is finishing, put them at the end of a list.

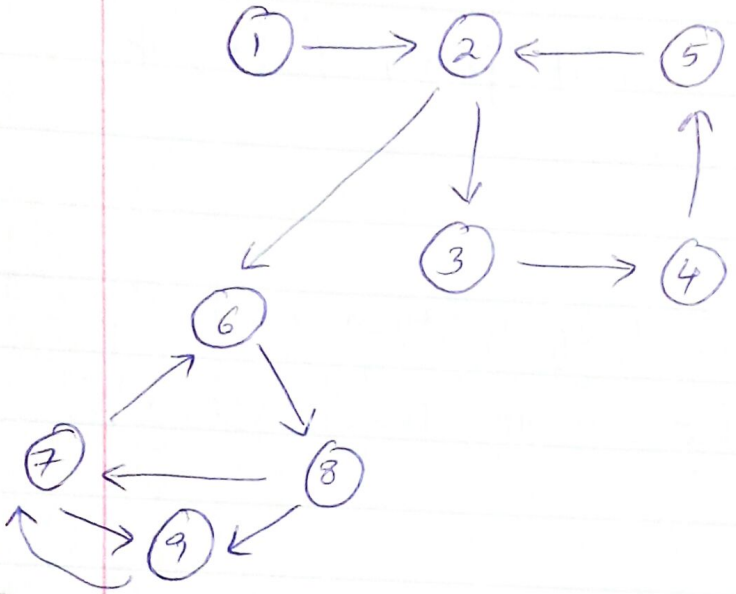


→ Total Time : $\Theta(|V| + |E|)$

Strongly ~~Compone~~ Connected Components :-
(SCC)

graph :- Directed graph, G.

Graph 1 :



SCC :- It is a subset of vertices

$S \subseteq V$

Ex: $\{2, 3, 4, 5\}$

i) $\forall v_i, v_j \in S$, $\exists$ a path from v_i to v_j

ii) This path must entirely lie inside (S)

Ex: $\{2, 3, 4, 5\}$

① Path from (2,5) & (5,2) ✓ (3,4) (3,5), (2,9)

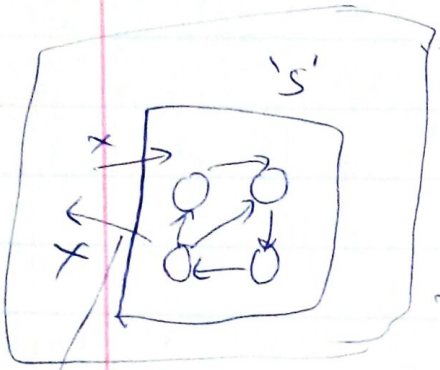
② Path must entirely lie in S ✓

Ex: {4,6}

① Path exists from (4,6)

② But it doesn't lie in the set, we have to go to other nodes

① Path doesn't exist from (6,4) ✱



→ subset of vertices, V, & you are not allowed to leave, set, 's'.

not allowed to leave set 's'

② Maximal Strongly Connected Graph:

's' is a MSCC,

iff → S is a SSC

→ $\forall \hat{s} \subseteq V$, $\hat{s} \not\subset S$,

$S \cup \hat{s}$ is not a SCC

(OR)

→ $\forall \hat{s} \supset S$, $\hat{s}$ is not an SCC.

Ex:-

In the Graph 1, in previous pages

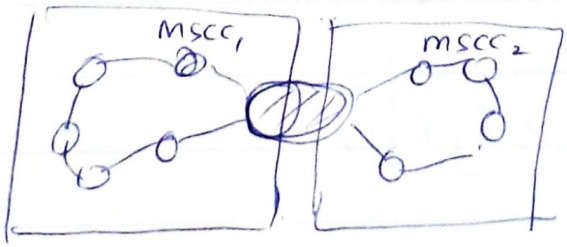
Set, {i} → trivial mscc.
{2, 3, 4, 5}
{6, 7, 8, 9} are mscc.

→ mscc decomposition of a graph.

Properties of mscc:-

1. 'S₁' and 'S₂' are two diff mscc's of a graph, then $S_1 \cap S_2 = \phi$

(or) else



Take $S_1 \cup S_2$

$\exists$ path from v_1 to v_2 and vice versa path only involves nodes in $S_1 \cup S_2$

$\therefore S_1 \cup S_2$ is a large SSC containing S_1 & S_2.

$\therefore$ 'S₁' cannot be maximal (or) 'S₂' cannot be maximal.

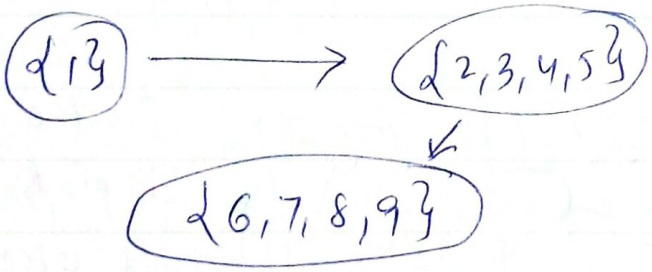
If S_1, S_2, S_3 are maximal SSC, then,

$$S_1 \cup S_2 \cup S_3 = V \text{ (vertices of graphs)}$$

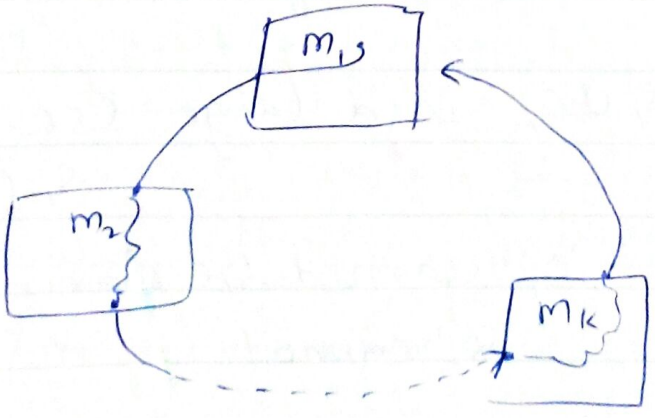
→ So, mSCCs partition the graphs into many disjoint subsets.

Property (2) mSCC supergraph is a directed acyclic graph.

mSCC. Supergraph: Compress each maximal SCC into a single node and form a graph.



Else.



then,

$m_1 \cup \ldots m_k$ is a mscc

Property:

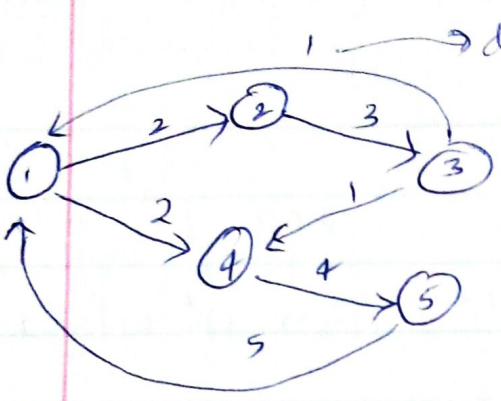
③ : the reverse graph, G^T has the same MSCC as the orig.

Reverse graph:-

Reverse the direction of edges in the original graph. G.

→ Adjacency matrix Rep: Transpose the matrix to make it reverse graph.

Graph - Shortest Path Algorithms



1 → distance, toll, time.

(1) Types of shortest path problems :-

1. Single Source - sing destination - simplest (minimum cost path from A to B)

2. All-pair Shortest paths: from all ~~pairs~~ Src to all Dst.

↳ mostly we end up computing shortest path from one source to all destinations & take required src-dest

→ Non-negative edge weights - natural

→ There are scenarios where negative & +ve edges are included

Ex: ① Systems where pressure, & temp are included

② In a car, due to its characteristic, it may build up energy when you go uphill & spend energy while coming ~~up~~ down or, so

⑤ Airline fkt prices,

ii) Physical systems, involving pressure.

PATH :-

The notion of a path is, we are not allowed to repeat a node in the traversal

Walk:

In walk, you can arbitarily repeat a node that you have been to before, and cannot repeat an edge, you have been to before.

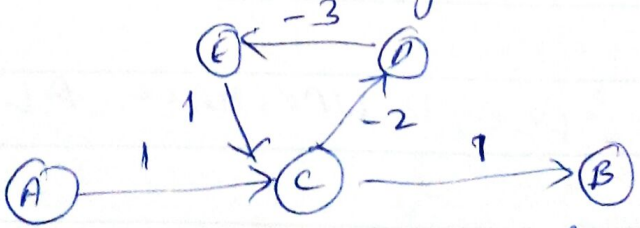
Tour :-

Can repeat a node, but not an edge.

In shortest Path Algos, we are talking about Path (or) Walk ?

Ok, whether repeating a ~~path~~ edge will ever give the shortest path?

Yes, it gives in the negative weight graphs,



Ex:

→ Here, going & repeating through loop (C,D,E) reduces my cost, every time I repeat

1. We are solving pblms with
   1) No negative wt edges
   2) No negative wt cycles

* → If there are no negative wt cycle, the shortest walk becomes shortest path.

↳ If a graph has neg wt cycles, the algorithms we are currently learning will fail, as it returning '-∞' as cost.

→ We are learning shortest walk, in graphs, which donot have negative wt cycles

→ we find shortest walk (or) Walk, in problems where we are asked to find path b/w 'A' & 'B' with exactly "x" stops.

↓
FLOYD WARSHALL ALGO - $\log k$ iterations

→ ~~In the algo~~

→ For the below algo's, there can be negative edges, but there cannot be negative edges in a cycle, whose overall weight is "-ve".

⇒ If there are such negative cycles, it will be like a gift which keeps on giving and we don't want to leave the cycle.

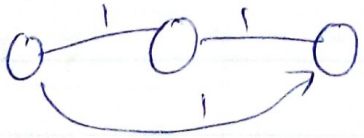
→ "Asbitrage" Trading → Wall street → free money without risk; Trading → $ → ₹ → ¥.

Algorithms:-

Unit weight edges:

If all the edges have "unit" weight, by doing BFS on a graph, we can find the shortest path.

'G'



BFS

running time: $\Theta(|V|+|E|)$

↓

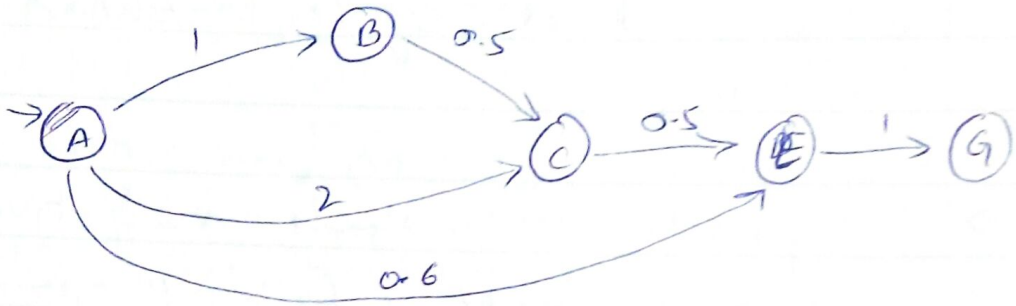
linear Algorithm
extremely fast

~~DAG~~:

Case 2:- DAG,

1) Perform Topological sort and relax Operations

Ex:-



→ Shortest path from source 'A'.

Steps:-

① Sort the graph in a Topological manner. (A B C E G)

② from 'A'

destination	A	B	C	E	G
distance	0				

- first assign '∞' to all nodes, distance estimates

dest	A	B	C	E	G
distance	0	∞	∞	∞	∞

- Traverse the nodes in the topological order
- For each 'node', process the outgoing edges of the node and update the estimated-distance of those edge nodes.

"x"

- If (current_node.est_dist + outgoing_edge_weight) is less than (est_dist_of_edge_node), then update the est_dist to "x" else, it remains unchanged.
- The above operation is called "Relax Operation"

Ex: Step 1:

dist	A ✓	B	C	E	G
distance	0	1	2	0.6	∞

Step 2:

	A ✓	B ✓	C	E	G
	0	1	1.5	0.6	∞
			=		

Step 3:

	A ✓	B ✓	C ✓	E	G
	0	1	1.5	0.6	∞

Step 4:

	A ✓	B ✓	C ✓	E ✓	G ✓
	0	1	1.5	0.6	1.6

* To find the shortest You also need Parent Info ← (The node from which I reached current node)

dist	A	B	C	E	G
dist	0	1	1.5	0.6	1.6
π	NIL	A	B	A	E

A ⟶ E ⟶ G.

Topological sort : $(|V| + |E|)$

Shortest Path computation : $(|V| + |E|)$

Case 3: Any Weighted Directed Graph:

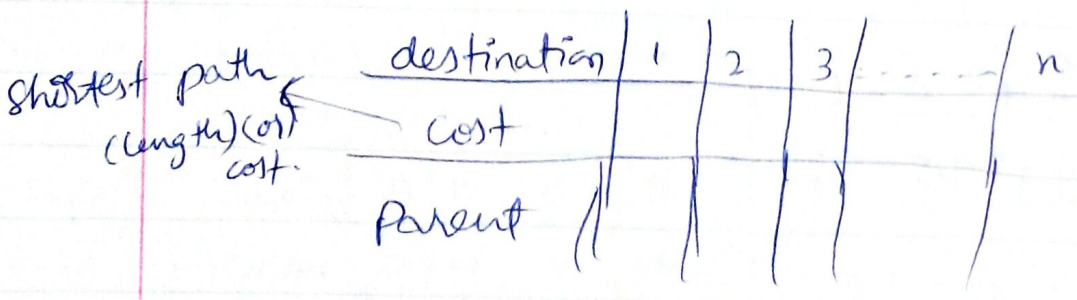
* Bellman-Ford Algorithm

Given;

* weighted, directed graph : 'n' nodes, 'm' edges

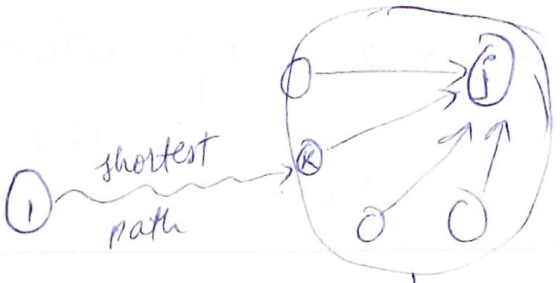
→ Starting Node

→ Output Table,



* Shortest Past Algorithms are "Dynamic Programming" Algorithms on graphs.

* Optimal Sub structure:



* Back View, D.P

who comes before "j"?

⇒ Optimal sub structure

* Other Method: (forward View):-

1) Suppose, I know already, shortest path from '1' to 'k', then how do I extend it to "j"

$d_k + w_{kj} \geq$ shortest path from '1' to "j"

STEPS

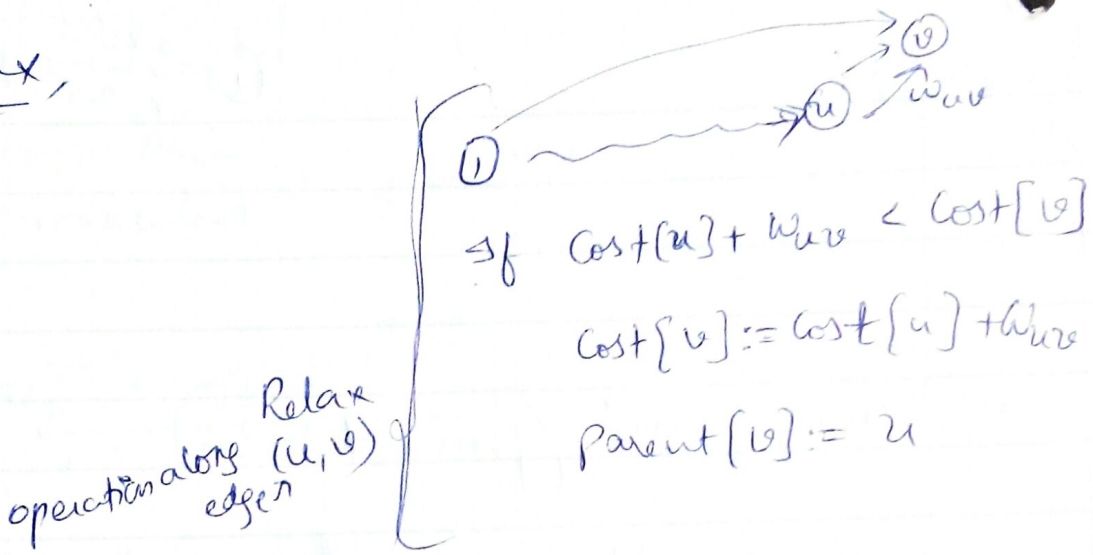
direct to table (memo table).

Step 1) Work on Table Memoization.

	1	2	3	. . . . .	n
dest	0	∞	∞	. . . . .	∞
parent	NIL	NIL	NIL	. . . . .	NIL

Now, main operation,

Relax,

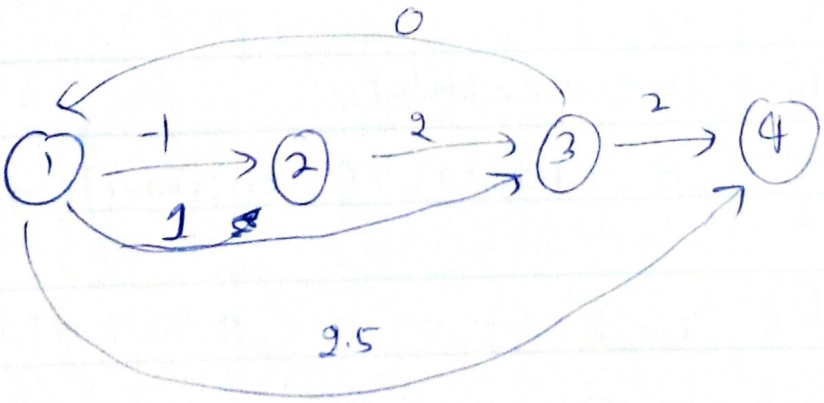


Bellman-Ford Algo:

```
for i=1 to n.
    for every edge e ∈ G
        Relax (e).
```

→ Bellman Ford, can handle graphs, even with the negative weight cycles

Ex:-



edges:

- (1 → 4)
- (1 → 3)
- (1 → 2)
- (2 → 3)
- (3 → 1)
- (3 → 4)

for i=1 to 4

for each edge e ∈ G

Relax(e)

Initial:

	1	2	3	4
Cost	0	∞	∞	∞
Parent	NIL	NIL	NIL	NIL

- Step (1):

(3,4)

'3' is at '∞', & '4' is at '∞'

so nothing happens

(2) (3,1)

nothing happens

(3) (2,3) nothing happens

4) (1,2)

'1' is at '0' & '2' is at '∞'

0 − 1 = −1 < ∞ ⇒ update the est-dist of "2" to "−1".

and parent(2) = 1

	1	2	3	4
c	0	−1	∞	∞
p	NIL	1	NIL	NIL

3) Now $1 \to 3$ $4 \mid 1 \to 4$

1	2	3	4
0	-1	1	2.5
NIL	1	1	1

Round 2:

AGAIN, Now again, go through edges in this order and relax again;

$1 \to 4$
$1 \to 3$
$1 \to 2$
$2 \to 3$
$3 \to 1$
$3 \to 4$

① ③ $\xrightarrow{1}$ ④ → no update
(1) (2.5)

② ③ $\xrightarrow{0}$ ① → no update

$\vdots$

① → ④

Do it for "Round 3" and "Round 4".

Finally you'll get table

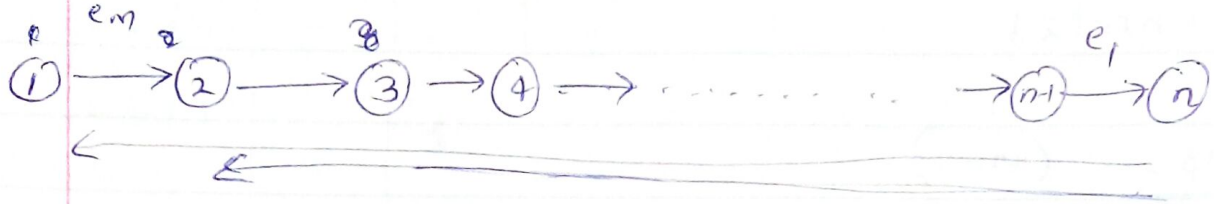
	1	2	3	4
cost	0	-1	1	2.5
parent	NIL	1	1	1

* for i=1 to 4
  for each edge $e \in G$
    Relax(e)

If nothing in Table changes after this for loop, you can stop execution

Why does it work & why 'n' times?

→ In a straight line graph, like below



→ when you come the edges at last to first, only the "2" node gets updated and remaining nodes estimated-dist remains unchanged.

→ So, In this case, we need to run algorithm for "n" times.

→ until (n-1) iteration, we'll have changes & in last iteration, there will not be any changes, I/f graph doesn't have any negative weight cycles.

If I have, a shortest path from A → B, how many edges, can it have in the worst case of a graph, 'n' nodes

= Ans:- (n-1)

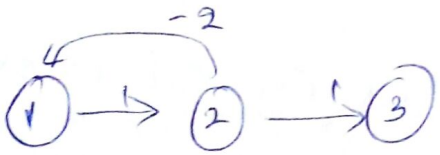
then (n-1) relaxations are enough to find the shortest path, but why the last nth round is done?

Ans→ It is done to find, whether graph has a negative weight cycle (or) not.

→ If there are no negative weight cycles, then table would stabilize for (n-1) rounds and will not change in nth round.

→ But, if on nth round, if the table changes then we can conclude that, graph has negative weight cycles.

Ex:



order of relaxation

$1 \to 2$
$2 \to 1$
$2 \to 3$

	1	2	3
Round 0	0	∞	∞
Round 1	-1	1	2
Round 2	-2	0	1
Round 3	-3	-1	0

(6) Round 3 → Again changed

→ So, run for $(n-1)$ times in a loop and then, run n^{th} round outside the for loop and if the table changes, then we can conclude that graph has a "negative weight cycle"

Home Work: Find the negative weight cycle, where the negative wt cycle is there, + -ve, wt cycle. (cost table should give you that)

Complexity:

$(n * m)$ ← nodes, edges

↳ if $m \simeq n^2$ ⇒ then $\Theta(n^3)$

→ Bellman Ford Algo. works in all cases including -ve wt cycles, but if edge weight ≥ 0, then it'll be taking huge time

→ If we can derive some order like in DAG (topological sort) where, we did only one iteration of relaxation, then we can reduce the complexity.

→ DIJKSTRA'S ALGO: works if edge weights ≥ 0

1) Relax. Operation exactly once on each edge

2) Greedy Algorithm,

Idea:

① Same Table

	1	2	3	...	n
cost ↑	0	∞	∞	...	∞

∗ At every step; (On)

- Select the node with smallest cost from the table, that has not yet been considered
- Relax all of its outgoing edges.
- Remove node it for further consideration

(relax operations: m + m log n)

∗ ① It takes 'n' steps.

② How many a edge gets relaxed in Dijkstra's algo?
exactly once, when src of that edge is taken out for consideration

③ Min Heap is used to maintain the minimum and get the minimum rapidly

∗) when you are relaxing, you have to adjust the heap, ~~which~~ using a bubble up (or) bubble down operation

∗ size of heap in worst case ⇒ 'n'

∗ Cost of extraction of min from Heap ⇒ O(1)
but we have to replace it so log(n)

Run Time :- relax operations

$m + n \log n$

Why does this Work ?

→ Any point think of the world in two ways,

⇒ Part of the world (or) graph which contains nodes taken out from the priority queue, whose outgoing edges are relaxed and have been taken out of consideration for future

→ The key thing is cost_estimates of these nodes, is the actual shortest path cost.

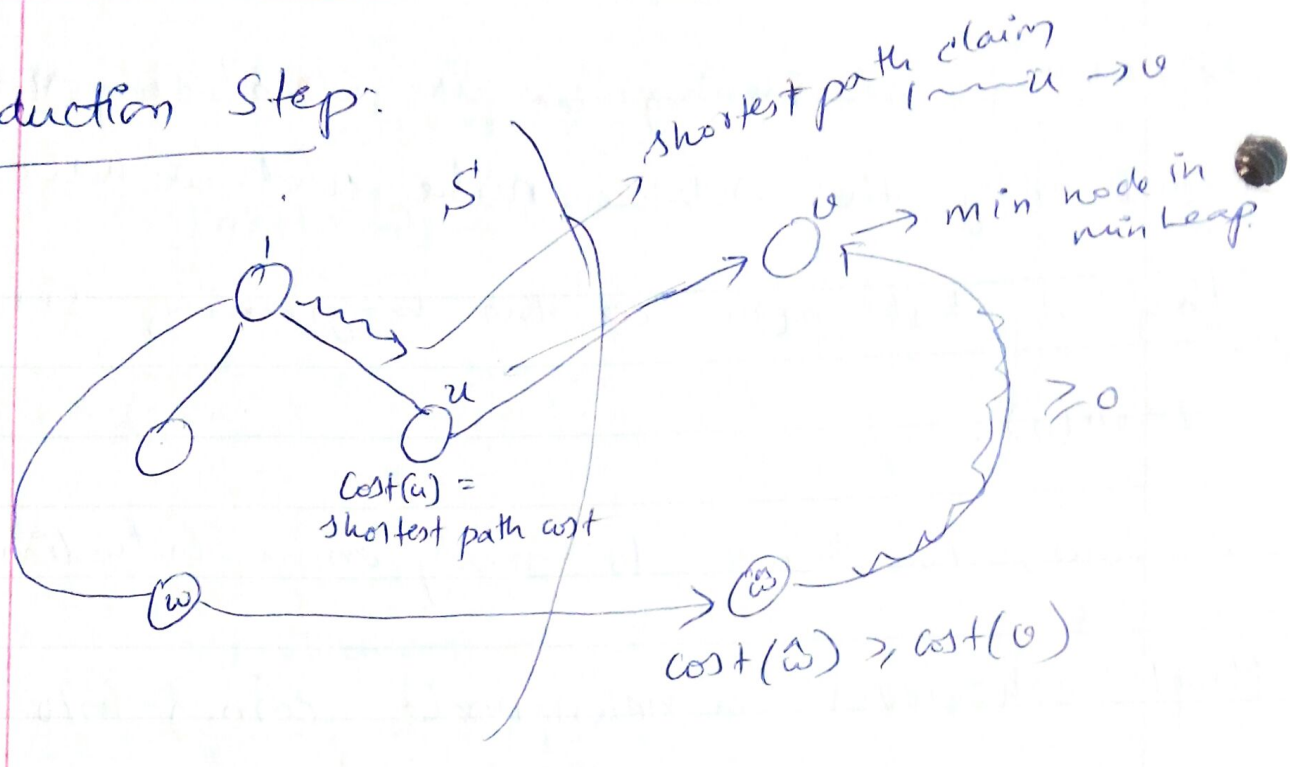
→ whenever, I'm taking smallest cost 1 node of out of priority queue, it's cost is already became shortest path cost and not going to decrease anymore. further (or) change further

→ At the beginning, we are considering & taking out only the source node, and we have found the shortest path at the beginning as it is zero (0).

→ Now, we have to carry on a induction argument that whenever a new node comes into set, S from outside, that new node shortest path cost has already been found and that's why it can come into 'S'.

→ If that continues, then algorithm runs until the finish, every node comes into the 'S' shortest path is already found.

Induction Step:



① Let us consider a node, v, smallest cost node among all possible nodes, then there has to be a parent of v, which is 'u', already in the nodes considered. which means shortest path for 'v', came along 'u'.

② My claim is that, path from '$1 \rightsquigarrow u \rightarrow v$' is the shortest path to 'v', but suppose that claim weren't true, there was a shorter path, where we went from

$1 \rightarrow w \rightarrow \hat{w} \rightarrow v$

→ I have to claim there is a contradiction

→ As 'v' is the min. cost node; $cost(\hat{w}) \geq cost(v)$.

→ Cost of edge $(\hat{w}, v) \geq 0$. So, if I went from ① → ⓦ → ⓦ̂ → ⓥ, then my shortest path is going to become ~~(cost(ŵ) + Path(ŵ, v))~~

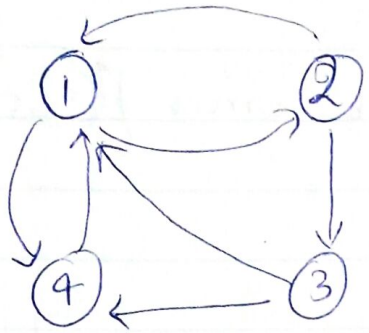
$$\left[cost(\hat{w}) + Path(\hat{w}, v) \right]$$

$$\geq$$

$$cost(v) + \geq 0$$

FLOYD-WARSHALL ALGORITHM (ALL-PAIRS SHORTEST PATH) :-

① Consider a following graph.



matrix Representation

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

1. Consider the above matrix as,

$$A^0 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

2. ~~Next compute A^1 the~~

2. Consider two vertices, (1, 2), there might be shortest path existing b/w $1 \rightarrow 2$ (or) there might be shortest path existing b/w other nodes also like.

$(1 \rightarrow 3 \rightarrow 2)$ (or) $(1 \rightarrow 4 \rightarrow 2)$

→ So, to consider all possibilities, we check paths b/w all vertices through all vertices in the graph.

→ In the final iteration, we'll be getting the shortest path b/w all vertices

(*) So, At first consider vertex, 1 as the intermediate vertex.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array}$$

(*) As, we are considering Vertex 1, all direct paths from it will remain unaltered

(*) src-src paths will remain '0'

$$\Rightarrow A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix}$$

$A^0[2,3]$ $\quad$ $A^0[2,1] + A^0[1,3]$

$2 \quad < \quad 8 + \infty$

$A^1[2,3] = 2$

$A^1[2,4] = \min\left[A^0[2,4] , A^0[2,1] + A^0[1,4] \right]$

So, $A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix}$

→ Considering A^1 as Base matrix,

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^2[3,1] = \min\left[\left(A^1[3,2] + A^1[2,1]\right), A^1[3,1]\right]$$

$$A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \qquad A^4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

So, finally A^4, gives the shortest path b/w all the pair of vertices.

$$A^k[i,j] = \min\left\{ A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j] \right\}$$

```
for (k=1 ; k<=n ; k++)        → no. of vertices
    for (i=1 ; i<=n ; i++)
        for (j=1 ; j<=n ; j++)
            A[i,j] = min (A[i,j],
                          A[i,k] + A[k,j])
```

Using the single matrix.

$O(n^3)$.

Linear & Integer Programming

→ Optimisation Problems:

→ maximise/minimise a cost function/ objective function subject to constraints

Linear Programming:

World war - II → massive logistic problems,
→ needed optimisation
→ moving people, resources etc.
(logistics)

Linear programming pblm:- (simplest)
max/minimise a linear function

Ex:- $2x_1 + 3x_2 + 7x_3 - x_4$ - max/min → objective fn.

subject to,

$x_1 \geq 0$
$x_1 \leq 10$
$x_2 - x_1 \geq 10$
$x_2 - 2x_1 + 3x_2 \leq 15$ } constraints

Variables → Decision variables

x_1, x_2, x_3, x_4

Ex:- Beer Manufacturing , A, B.

Beer A, B

Classic Linear Programming Pblm

	Hops	Barley	Water	Malt	Person Hrs	Price
A 1unit	0.2	0.3	0.4	0.7	3.2	1.2
B 1unit	0.3	0.4	0.2	0.6	1.4	0.7

constraints: 10 units of hops
15 units of Barley
15 gallons of H_2O
20 units of malt
10 person hours.
} limitations

→ How much of Beer 'A', & Beer 'B' should we manufacture to max/min the profit. (Revenue)

Decision Variables:

x_1, x_2 → How much of 'A' & 'B' should be produced.

→ max : $1.2x_1 + 0.4x_2$

s.t ; $x_1 \geq 0$; $x_2 \geq 0$

$0.2x_1 + 0.3x_2 \leq 10$ → hop constraint

$0.3x_1 + 0.4x_2 \leq 15$ → barley "

$0.4x_1 + 0.2x_2 \leq 15$ → H_2O "

$0.7x_1 + 0.6x_2 \leq 20$ → malt "

$3.2x_1 + 1.4x_2 \leq 10$ → person "

→ We can solve extremely fast for large LPs with millions of vars + constraints

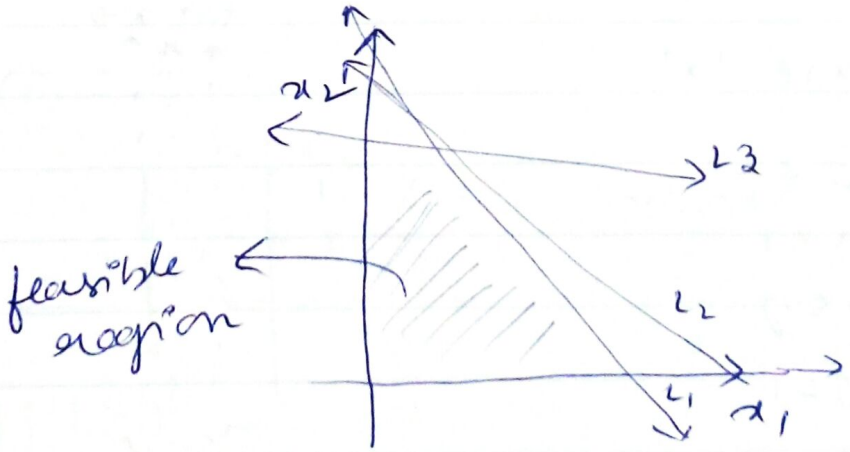
Algo's

① Simplex Algo → George Dantzig
↳ Greedy

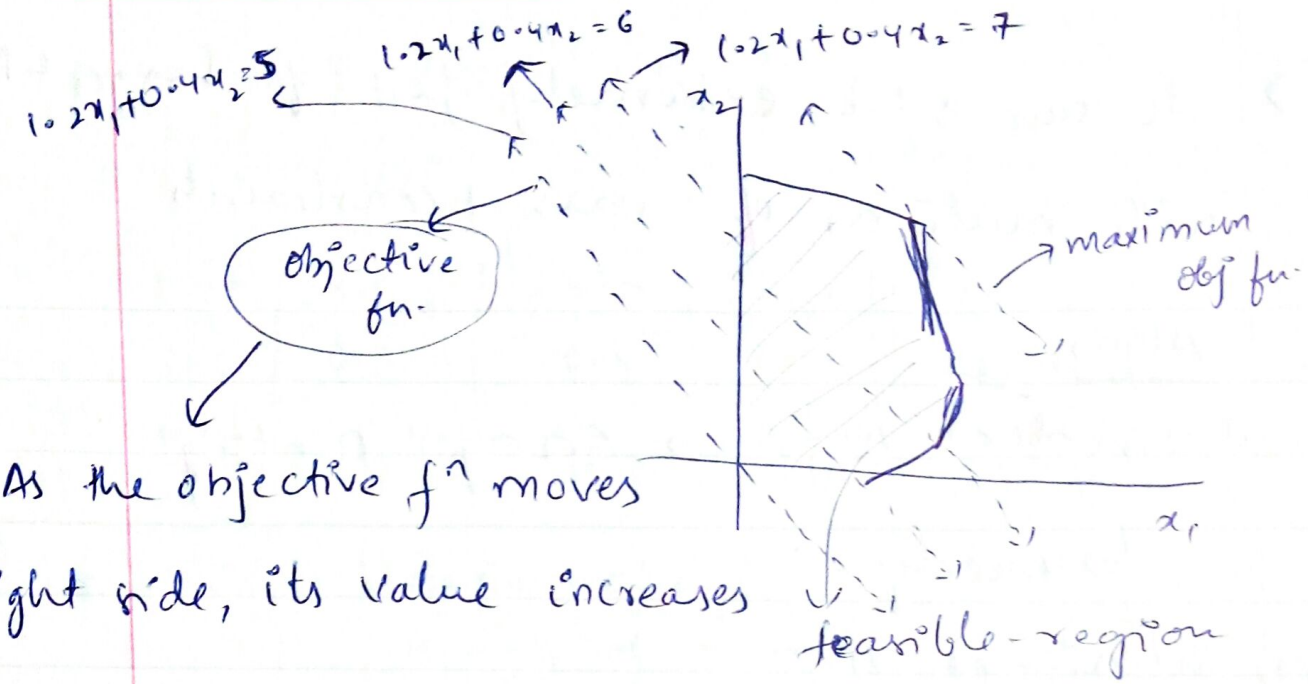
② Interior pt Algo

Geometry of LP :-

→ when you draw all the equations on a graph, you get the feasible region



→ we get a convex polygon in 2-d and convex polyhedron in n-dimensions



→ As the objective fⁿ moves right side, its value increases

→ when it barely touches the vertex, it's value is maximised.

→ Condⁿ where max fⁿ is not a Vertex:

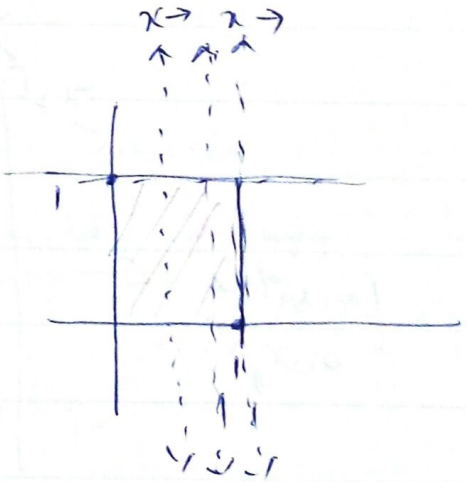
Ex: maximise 'x'

Sot $0 \leq x$

$x \leq 1$

$0 \leq y$

$y \leq 1$



Positions: $x = 1, y = 0$

$x = 1, y = 1$

$x = 1, y = 0.5$

If the L·P has an optimal solution, one of the optimal solution is vertex

L·P problems, with no solution ?

Yes, if the ① "feasible region is empty"

② Region is unbounded

Ex. for ①:

$\max x;$
$x \geq 1 \; ; \; x \leq -1$ } infeasible

Ex for ②

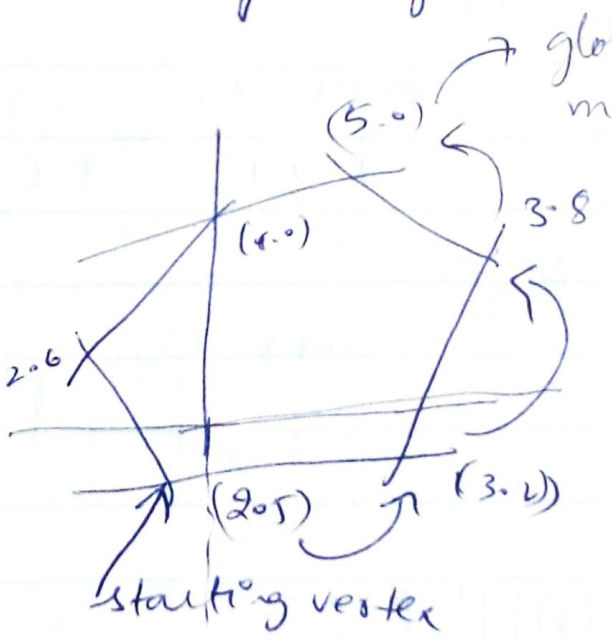
$\max x;$
$x \geq 1.$ } unbounded.

SIMPLEX :-

* In this algorithm, we start from one vertex, and check the neighbouring vertices. if it goes to the neighbouring

vertex, which has maximum objective function, ~~likewise~~ if all its ~~vert~~ neighbouring vertices are less than its objective function (at that vertex) it declares, it is the global maximum and it cannot maximise more than that.

→ It is a hill climbing algorithm



What is a Vertex? (In computers)

2 Dimension

- Intersection of 2 sides
- must lie in feasible region

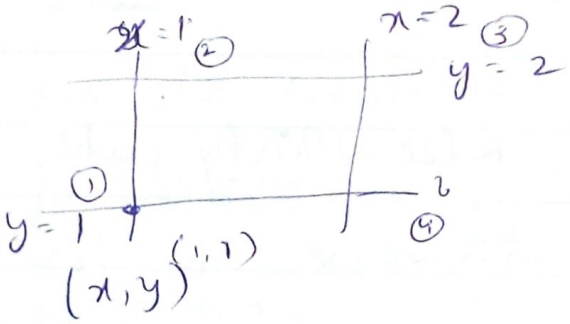
- n-dimension. 'n'
  - intersection of ~~faces~~ sides

Steps in Simplex:

1. Find a starting vertex
2. Pivot to find a "better vertex"

↓

In a 2-d rect:



PIVOTING:-

→ At first you get a pt where $x=1; y=1$ will intersect, now if you want to go to next vertex, fix $x=1$, throw out $y=1$, and bring in $y=2$, $(x=1, y=2)$ will be its neighbouring vertex.

→ So, in n-dimensional space, you fix one side, and change other sides one at a time, to get to neighbouring vertex

* Worst-Case:

In the w.c, you can have exponential # of vertices, ex:

4 vertices - 4 sides → square

8 v - 6 s → cube

16 v - 8 s →

⋮

2^n vertices - $2n$ sides

→ Klee-Minty cube, 2^n vertices, simplex has to traverse to find ↓ solution.
optimal

Simplex :- w.c → exponential

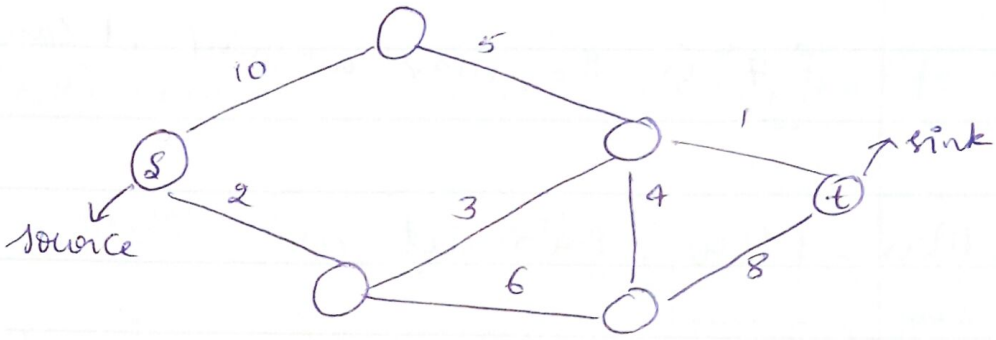
Avg → polynomial time

Combinatorial Pblms:

- Optimisation Pblm

→ **Max-flow Pblm:**

Ex:-



① Consider nodes as junctions and edges as pipelines, weights as the capacities, that how much oil, the pipelines can carry.

(or) nodes → as servers
edges → connections
weights → bandwidth capacity.

→ Now, maximise the amt of oil, if every edge can carry only upto its capacity

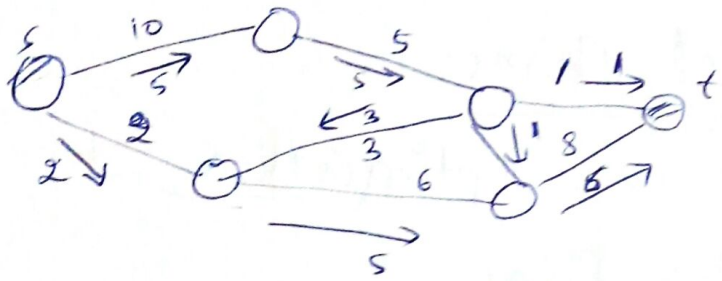
* Apply Kirchoff's law on every node

∴ incoming flow at junction = outgoing flow

(flow must be conserved at every node).

? What is the max amount of flow that can be sent. How much is it?

Ex:



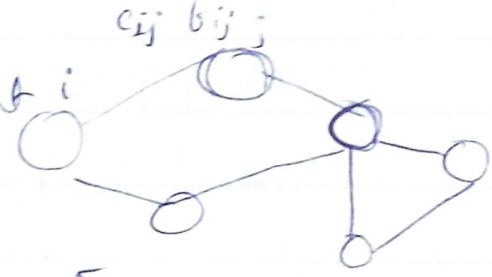
∴ 7 is the max. amount, I can send from 's' to 't'

=

Max-Flow, Min cut

Convert it to LP problem:

maximise $\sum_{s \to j} f_{sj}$; all edges incident on 's'



$$f_{ij} \leq C_{ij} \quad \forall (i,j) \in E$$

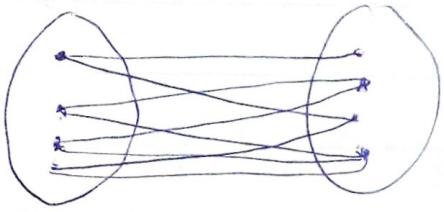
$$\sum_{i \to j} f_{ij} = 0 \quad \forall i \in V$$

Flow Prblm are mostly used for Assignment Pblm:-

Consider the example of assigning people to chores. Edge weights are their priority (or) affinity towards a chore. (more the edge weight higher the priority)

Bipartite Graph:-

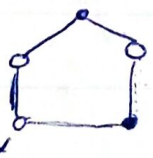
- It is graph, whose vertices can be divided into two independent sets U and V such that every edge (u, v) either connects a vertex from U to V (or) a vertex from V to U.
- There is no edge that connects vertices of same set.



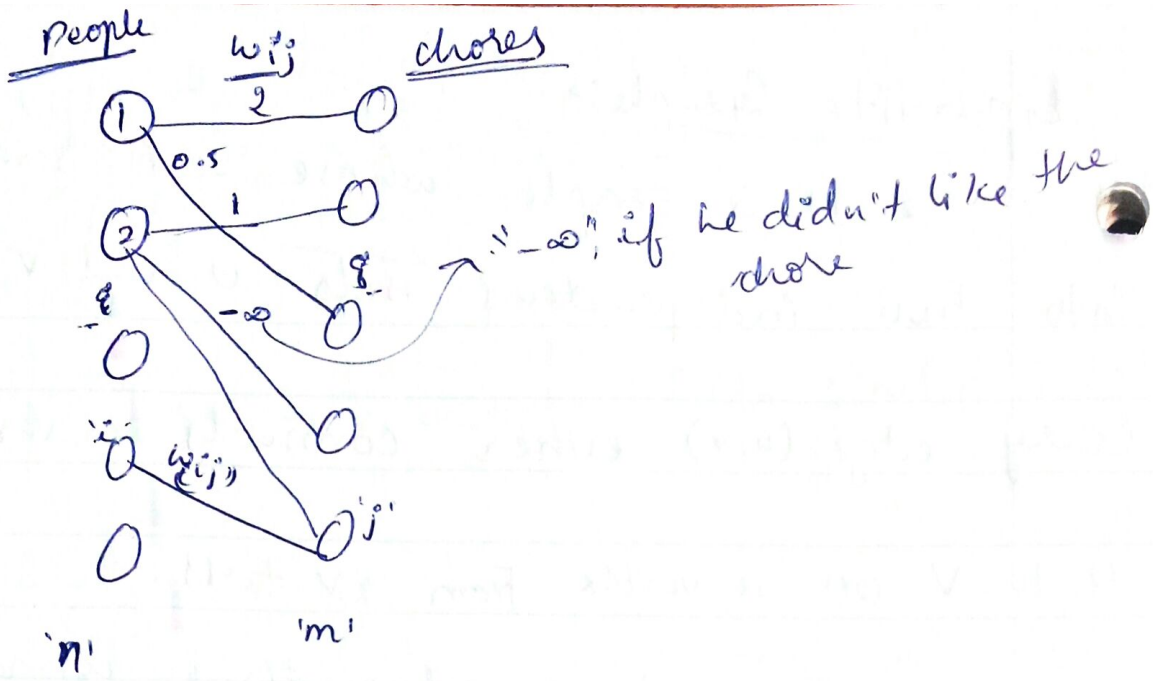
- A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.

> - It is not possible to color a cycle graph with odd cycle using two colors

Ex:



→ odd cycle



Now, maximise the sum of edge weights,

$$\sum w_{i,j}$$

- Every one has to be assigned a chore
- Every chore has to be assigned to one person.

→ Welfare function :- When a person 'i' is assigned to a chore 'j', then the edge weight $w_{i,j}$ is added to the welfare.

→ Maximise welfare function :

How to convert to L.P?

Decision Variable:-

$$w_{ij} : \begin{cases} 1 & \text{if } 'i' \text{ is assigned to } 'j' \\ 0 & \text{o.w.} \end{cases}$$

$w_{i1} + w_{i2} + w_{i3} + \cdots w_{in} = 1$ → every person has to do only one chore

person

$w_{1j} + w_{2j} + w_{3j} + \cdots w_{nj} = 1$

chore

} constraints

$w_{ij} \in \{0, 1\}$

Binary constraint; The pblm is no longer L.P. In L.P, we can write constraints like $0 < w_{ij} \leq 1$ but not $w_{ij}^2 = w_{ij}$.

⊙ But if you make above constraint as, $0 \leq w_{ij} < 1$; then, fraction of a person is assigned to a fraction of a task. which is not a desired solution.

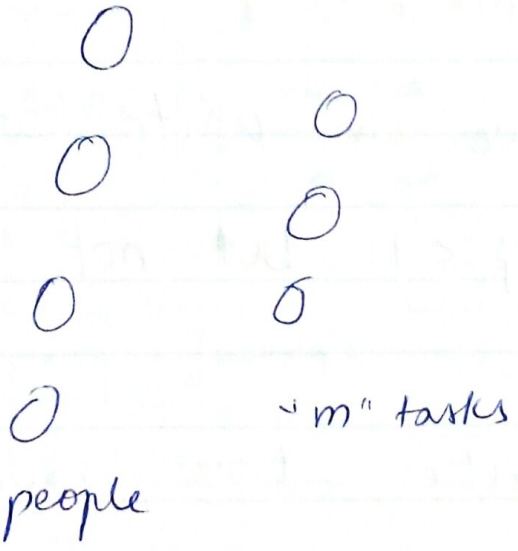
⊙ We want "Decision Variables" to be integers not fractions.

Ex: we can't assign 1½ aeroplane (or) 2½ objects to anything.

↳ Meaningless.

→ Now, these kind of assignment problems can be solved using flow problems.

Ex:



→ We have 'n' people, "m" tasks, In this pblm, the person says, "yes" or "no" to task, but no weights.

- If n > m, there'll be jobless people
- if n < m ; there'll be ~~more~~ tasks unsolved

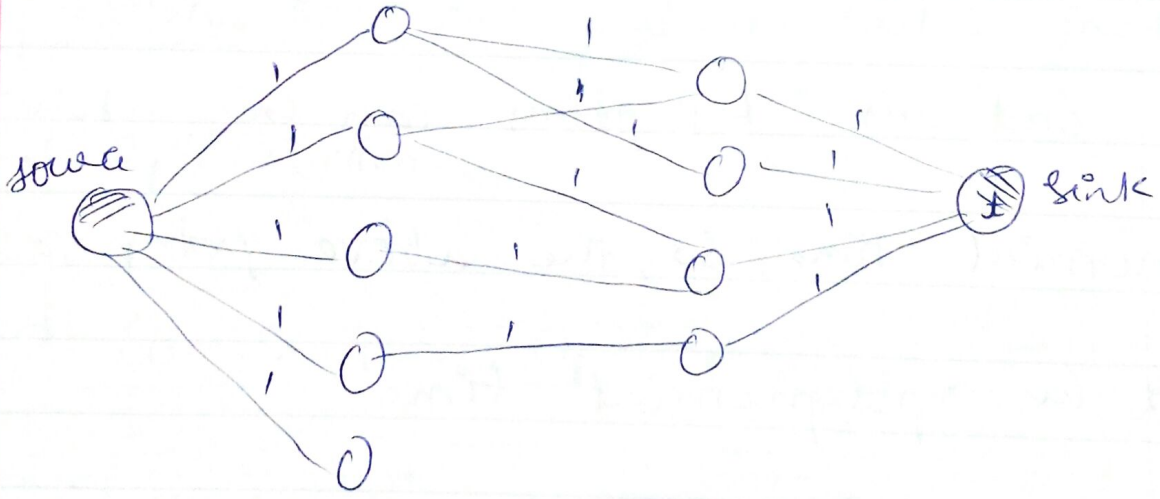
- Let us maximise the no of tasks we are able to get done

- How do you find such an Assignment?

- We can convert above pblm into a flow pblm.

Steps:

- Make a artificial source & sink & give a capacity of '1'



- Send the max flow through the network.

- Maximum flow = Optimal no. of tasks you will be able to solve.

- we can't send "fractions" as flow, the max-flow has a property.

→ **Integrality Property**

If every number, (every capacity) in a graph is a integer (whole number) then the optimal flow is also a Integer (whole numbers)

- As "max-flow" is a linear programming pblm and as L-P pblms can be solved in polynomial time, so, the above pblm can be solved in "polynomial time"

- So, because of such a dichotomy, Algorithms get divided into

Easy Problems

* Linear Programming
(~~least square~~
$x \in R$)

Hard Problems

* Linear Programming
(Integer Values)
Variables

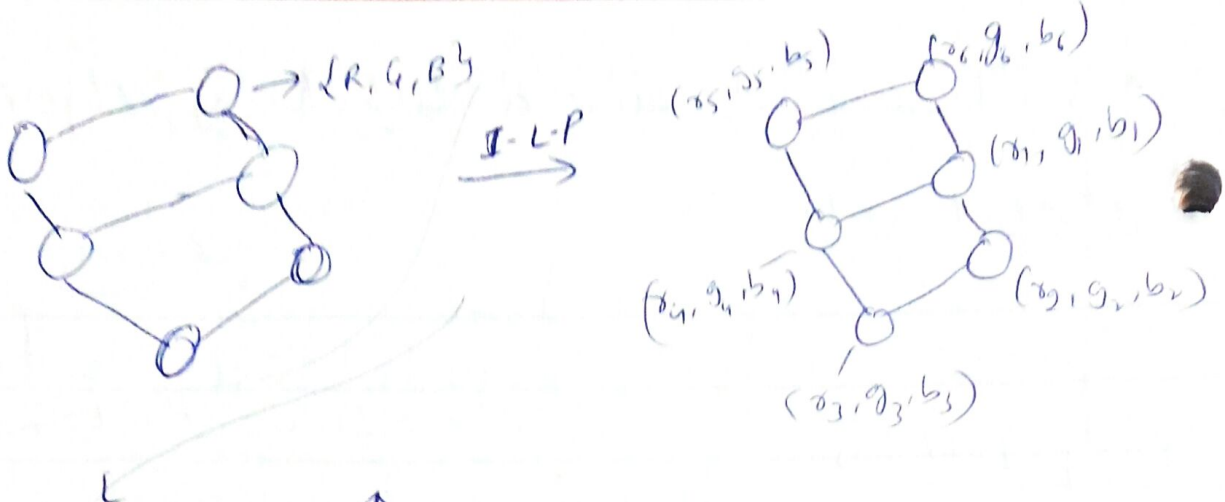
It all started from the field of "Operations Research" where the business pblms need optimisation.

- # Combinatorial Optimisation Pblms:-

Integer L.P : max linear f^n
s.t $Ax \leq b$ } constraints
$\bar{x} \in \mathbb{Z}^n$

- If it is 0-1 LP then $x \in \{0,1\}^n$

Ex:- Graph colouring Pblm
{ 3 colours }



$r_i, g_i, b_i \in \{0, 1\}$

$r_i + g_i + b_i = 1$ → exactly only one color

i — j

$r_i + r_j \leq 1$ } only one of the neighbouring nodes be red.

$g_i + g_j \leq 1$

$b_i + b_j \leq 1$

∴ The above such pblms are called

NP-Complete Problems: Hard Problems:

- After 50 years of smart ppl trying, no one seems to know if an "efficient" solution is "possible/impossible"

NP- Non-deterministic polynomial time

- All pblms discussed in class, in this context will be saying "Yes" (or) "No" (Decision Pblms)

Ex:

3-color: Is there an assignment of 3-colors s.t no two vertices connected by edge have same colors?

TSP: Cost $\leq K$?, whether the cost of the tour $\leq$ 'K'

Decision Pblm:

- "Yes" (or) "No" (solution)
- If yes, give certificate. so that we can verify the solution.
- We can check the solution in polynomial time.

Ex:- 3-colorable pblm.

graph, g : 3 colorable?

if yes, certificate check done in $\Theta(n)$

Formally, problem $\in$ NP iff

- If the answer to a problem instance is yes,
  - $\exists$ a certificate, length of certificate is polynomial in problem size
  - the certificate can be checked in polynomial.

If answer is "No", there is no succint reason, why it is not 3-colorable

Ex: TSP

graph, G, cost, k; Is there a T.S.P tour of graph whose total weight $\leq k$?

Ex: 0-1 ; ILP:

maximise $c_1x_1 + c_2x_2 + \cdots c_nx_n$

s.t $a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1$

$\vdots$

$x_1, x_2 \cdots x_n \in \{0, 1\}$

Q Is there an answer $\geq k$.

- ✓ super simple check
- ✓ Polynomial time check of certificate

SAT Problem:

- Boolean satisfiability pblm:

- $x_1, x_2 \ldots x_n$ boolean variables

T F

$((x_1 \wedge x_2) \vee x_3) \wedge !x_4$: made with $\wedge, \vee, !$

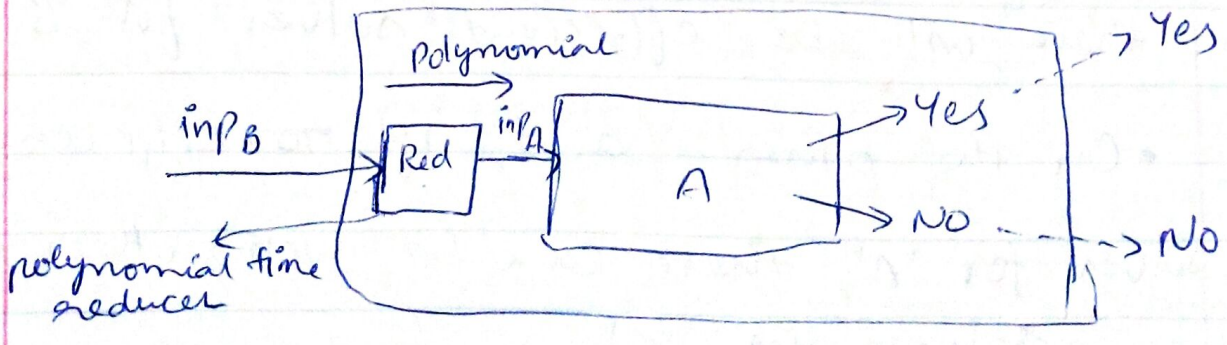
⇒ Is there a truth assignment that makes the formula true.

(circuit SAT)

NP problems can be reduced from one to another

NP ⟶ NP reduction

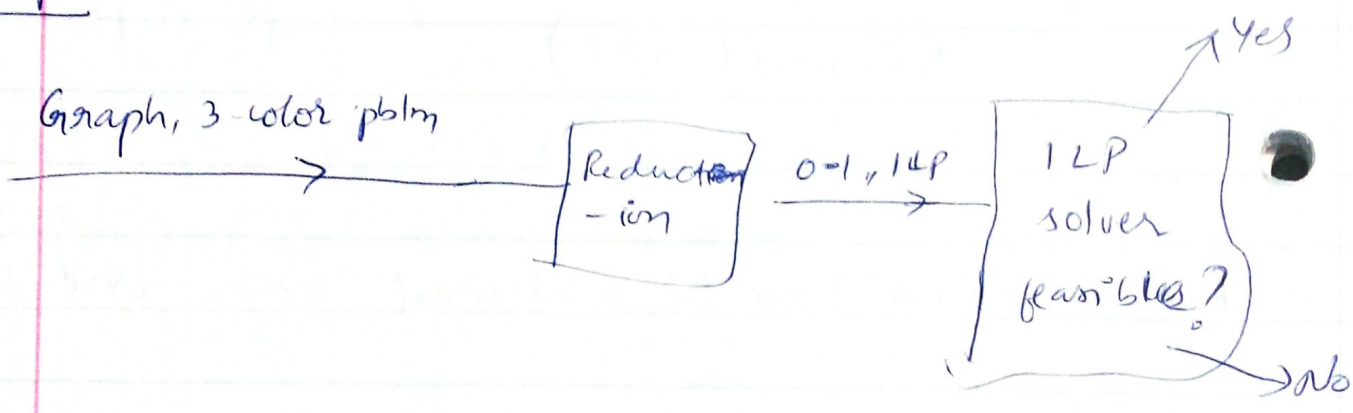
- Suppose, I have pblm 'A', which I know how to solve, then I can make it to solve pblm "B", by reducing it to 'A'



- The reductions, which are reduced in polynomial time are called "polynomial time reductions"

- "log space reductions" ⇒ "polynomial time reductions + (we ~~used~~ should at max use space in logarithmic terms),

Example:



⇒ ILP efficient solver ⇒ 3-color efficient solver.

- If there is efficient solver for 'A', then there will be efficient solver for 'B'.
- On the other hand, if no efficient solver for 'A', there can be some other efficient solver for 'B'.

- If 'B' cannot be solved efficiently, then 'A' cannot be solved efficiently.

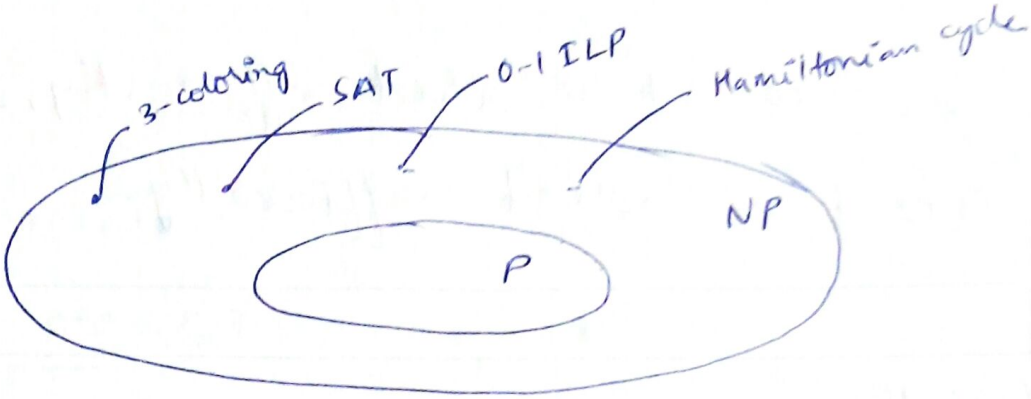
Lecture 15:

NP- Completeness

1. A problem is in NP, iff
   - Certificate for Yes answers
   - Certificate must be "succint"
     $|c| = O(|x|^c)$
   - Cert must be checkable in Polynomial time.

* NP: Non-deterministic Polynomial time
* P: All-deterministic polynomial time problems
  (These are problems which can be solved in polynomial time. No need of certificate)

Ex: Given graph G, Is it acyclic?
$= \Theta(|V|+|E|)$



Why 'NP' problems aren't in 'P'??

famous P=NP → question

- whether all pblms in "NP" have efficient solution?
- Is there is a way to generate the solution efficiently?

⟶

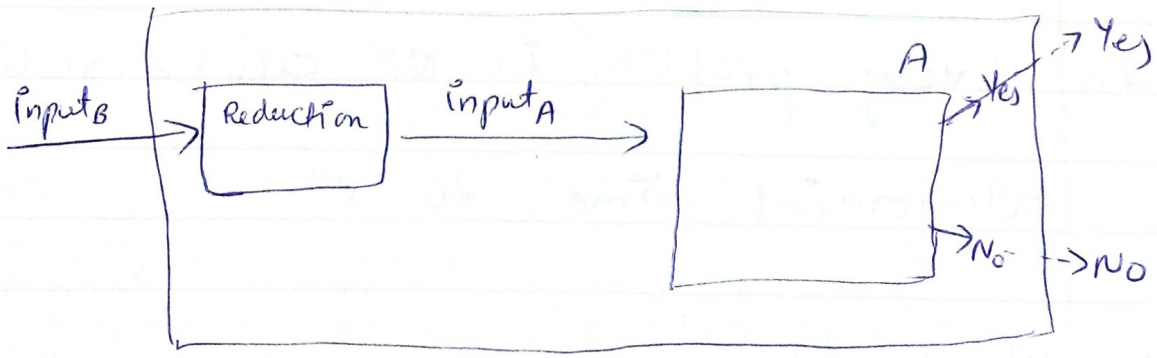
But, there are certain class of pblms, if you are able to solve them in polynomial time, you'll be able to solve all other "NP" pblms in "Polynomial time".

Those are "NP-COMPLETE" pblms

* If single "NP-Complete" pblms, shown ~~not~~ never to have a polynomial time algorithm then every "NP" problem will not have a polynomial time algorithm

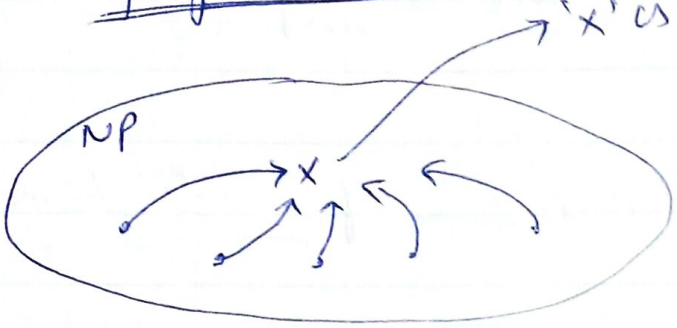
- It works because of reductions b/w the problems

REDUCTION :-



Define an NP-Complete:-

. We can say problem, 'x' is np-complete, if every problem in "NP" can be reduced to 'x' in "polynomial time"



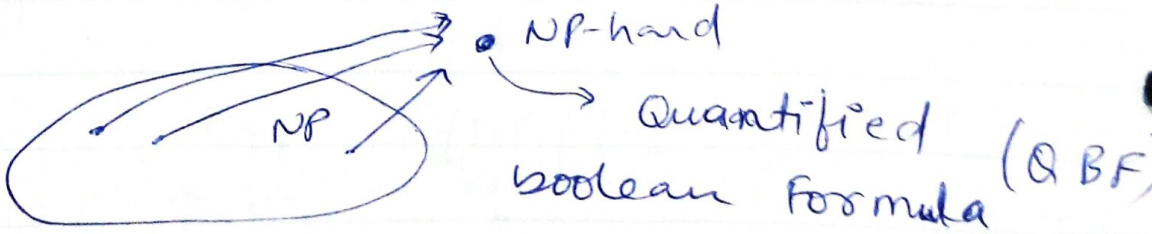
NP-Complete conditions:-

1. 'x' $\in$ NP
2. Every problem in NP can be reduced in polynomial time to 'x'

⟹ NP-Hard:-

. If only condition "2", holds in above statements, then it is NP-Hard

Ex:-

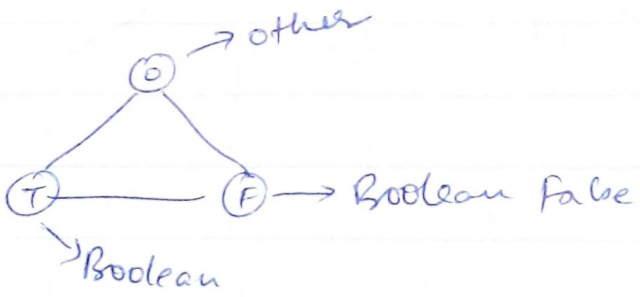


Cook & Levin Theorem :-

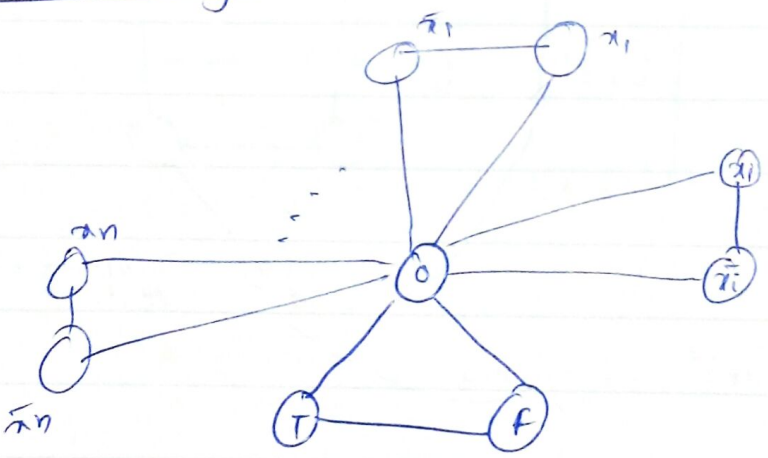
CIRCUIT-SAT is np-complete

3-Color Problem NP-Completeness :-

Gadget :

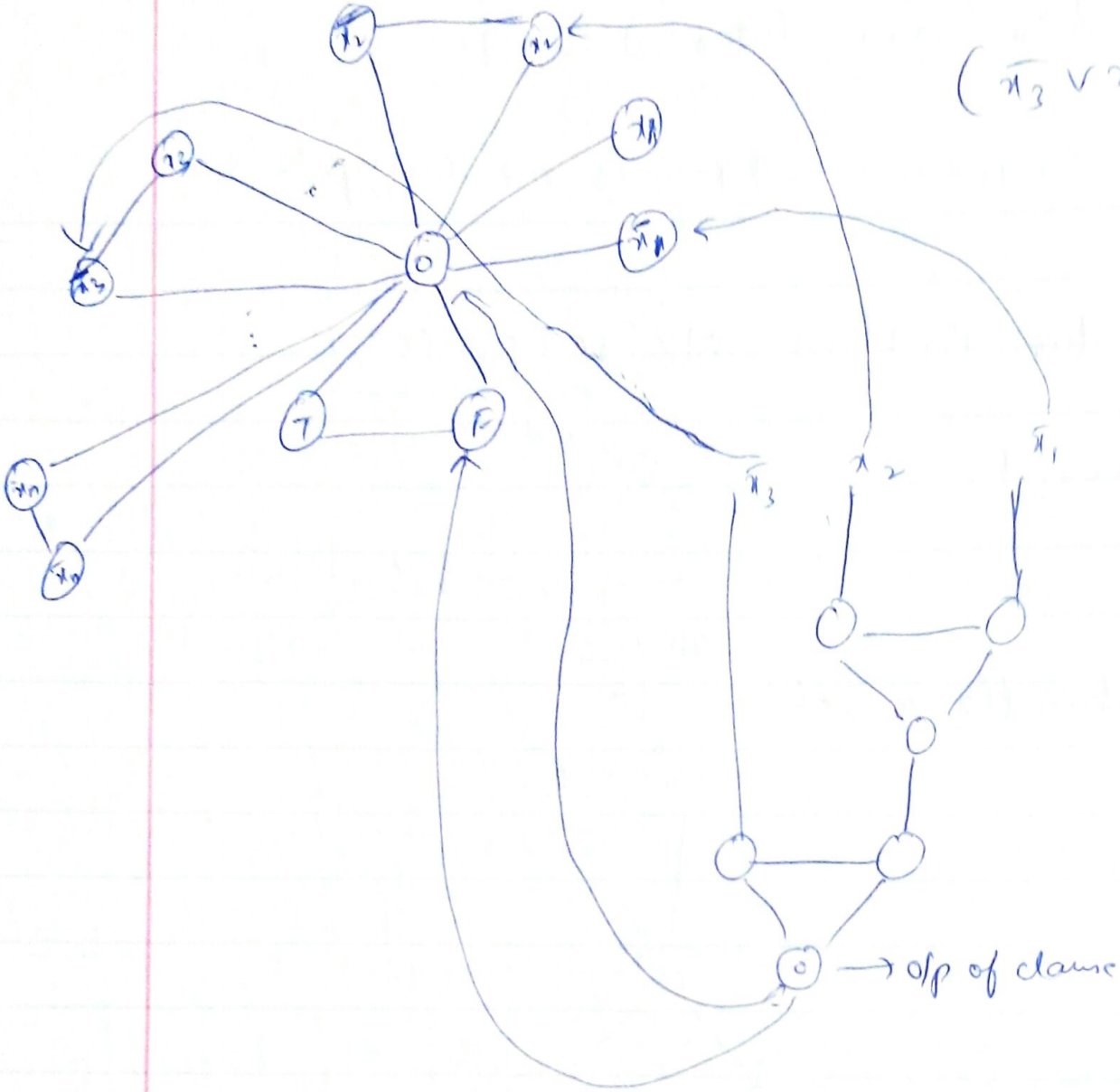


Variable Gadget :



Clause Gadget :

- Connect input of clause gadget to variables in variable gadget & output to "T" & "F".



- Now, the only thing the o/p can receive is true, if it 3-colourable as it is connected to both 'False' & 'Other'.

- We have to show that, if all three inputs are 'F' then, the graph is not 3-colorable if any one is 'true', the graph is 3-colorable