2. for each vertex (V) ∈ V[G]

3. do make_set(V)

4. sort the edges of E in non-decreasing order.

5. for each edge (u,v) ∈ E.

6. do if Find_set(u) ≠ Find_set(V)

7. then A ← A ∪ {(u,v)}

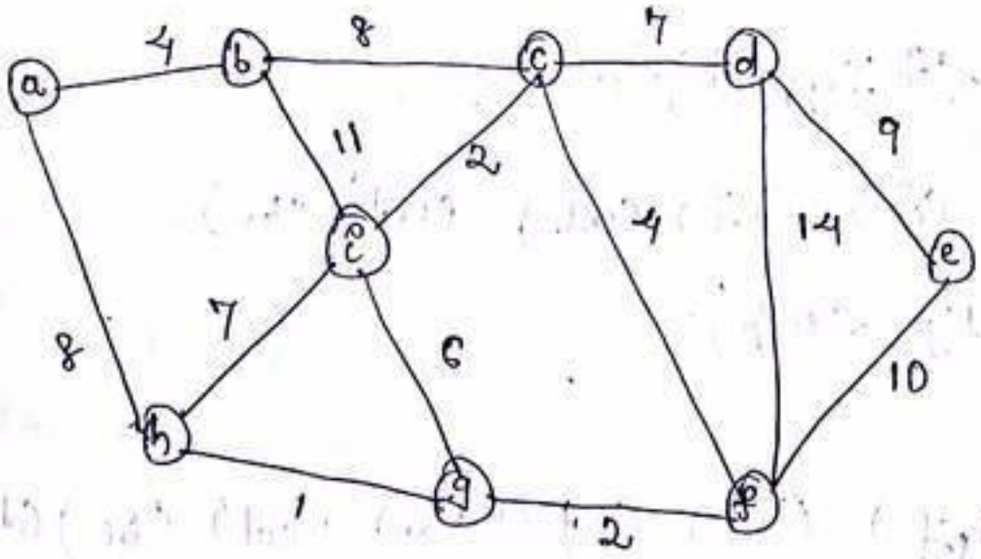
8. Union (u,v)

9. return A

10. end.

Set A will be containing the edges included in minimum spanning tree.

Ex:



Def:)

An algorithm is a sequence of computational steps that transforms the input into output. (Cormen).

Def:)

An algorithm is a finite sequence of computational steps that accomplishes some task or that accomplishes a particular task.

*- An algorithm must contain finite number of statements and the statements (computational steps) must be well-defined.

Properties of an algorithm.

i) Input.

ii) Output.

iii) Definiteness : All the computational steps must be well-defined. (clear and unambiguous).

iv) Finiteness : It must contain finite number of computational steps.

v) Effectiveness : The statement (computational step) must be carried out by pencil and paper i.e., the operation defined in

the computational step should be feasible operation.

Ex: Sorting Problem.

Input: A sequence of numbers $\langle a_1, a_2, \cdots, a_n \rangle$.

Output: A permutation of the input sequence $\langle a_1', a_2', \cdots a_n' \rangle$ such that $a_1' \leq a_2' \cdots a_n'$

Ex: Input: $\langle 91, 82, 63, 9, 56 \rangle$.
Output: $\langle 9, 56, 63, 82, 91 \rangle$.
} Instance of the input sequence. or Instance of the problem.

Correct Algorithm.

An algorithm is said to be correct if it halts with correct output for all instances of a problem.

*- A correct algorithm solves a computational problem.

Correctness Proof.

*- Mathematically, the algorithms have to be proved as correct.

Design.

* Well-defined design techniques are used for each algorithm or to design algorithms.

Analysis.

To design fast algorithms on the basis of efficiency.

Problem.

Given two numbers m and n.
Find their greatest common divisor (GCD)

Ans:

Input: Two numbers m and n.

Output: GCD(m,n).

Algorithm1: Factoring method.

Algorithm2: Euclid's method.

Algorithm 1:

15/7/2015.

1. Factorise m: Find primes $m_1, m_2, \ldots, m_x$ such that $m = m_1 * m_2 * \ldots m_x$.

2. Factorise n: Find primes $n_1, n_2, \ldots, n_y$ such that $n = n_1 * n_2 * \ldots * n_y$.

3. Find common factors.
Multiply them and find the GCD.

simple statement
loop
conditional statement.

Algorithm 2 :

1. Check if m divides n .
2. If m divides n, returns m as GCD.
3. else .
4. while (m doesn't divide n)
5. {
6. remainder (r) = n mod m .
7. n = m
8. m = r .
9. end of while .
10. return m as GCD .

Pseudocode .

Instance.

m = 36 , n = 48 .

1:

Algorithm 1 :

m = 2 x 2 x 3 x 3 .

n = 2 x 2 x 2 x 2 x 3

Common factors : 2, 2, 3 .

GCD = 2 x 2 x 3 = 12 .

" " 12

2 | 36
2 | 18
3 | 9
3 | 3
0

2 | 48
2 | 24
2 | 12
2 | 6
3 | 3
0

No. of operations by Algorithm 1 = 11 .

Algorithm 2 :

12

36 | 48 | 1
36
12.

Iteration | m | n | Step-1.
1 | 36 | 48 | n % m

Step-1.
r = 48 % 36
= 12
n = 36, m = 12.

36 % 12 (divides).

12 | 36

GCD = 12.

No. of operations by Algorithm 2 = 3.

* We compare algorithms on the basis of the running time of an algorithm.

→ Number of primitive operations (conditions, assignment, logical, relational, mathematical operations) or step executed to solve the problem.

Instance:

m = 434, n = 966.

Algorithm 1:

m = 2 × 7 × 31.

n = 2 × 7 × 139.

Common factors: 2, 7.

GCD = 14. No. of operations = 7. (manual)

966 = 2 × 234 + 98

checking for prime

Algorithm 2:

Iteration | m | n | Step 1.
1 | 434 | 966 | n % m (not divides)

Step-1.
r = 966 % 434 = 98

98 434

$n = m$
$m = r$

$434 = 98 \times 4 + 42$

$r = 434 \% 98$
$= 42$

$98 = 2 \times 42 + 14$

$r = 98 \% 42$
$= 14$

GCD = 14

$r = 42 \% 14$
$= 0$ (divides)

No. of operations = 5

* The running time and the time complexity also depends on the instances or the inputs.

16/7/2015

Analysis : Running time of an algorithm.

Running time : No. of primitive steps executed to complete the task.

Given an algorithm how to compute the running time.

Eg: Insertion sort

31, 9, 10, 27, 85.

9 31 10 27 85

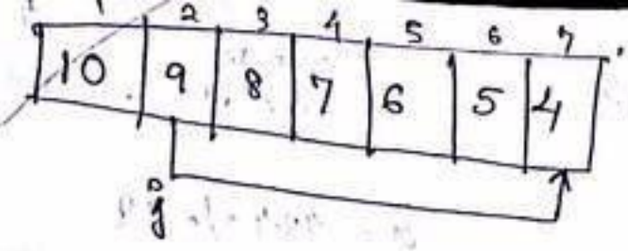
9 10 31 27 85

compare

9 10 31 27 85

9 10 27 31 85

9 10 27 31 85



$j = 2 \ldots n$ (as 1 is sorted)

Key = A[j] element to be sorted.

if A[j] is the key, then the algorithm assumes the array elements A[1...(j-1)] is already sorted.

j=2:

A[1] is sorted.

Key = A[2]

i start for A[j-1].

if A[i] > key

A[i+1] = A[i]

i = i-1.

Insertion Sort (A).

1.	For j = 2 to length (A).	n times	(termination of loop)
2.	key = A[j]	(n-1)	
3.	i = j-1.	(n-1)	
4.	while (i > 0 && A[i] > key),	$\sum_{j=2}^{n} t_j$	
5.	A[i+1] = A[i]	$\sum_{j=2}^{n} (t_j - 1)$	
6.	i = i-1.	$\sum_{j=2}^{n} (t_j - 1)$	
7.	end of while.		
8.	A[i+1] = key.	(n-1)	
9.	end.		

Let the 4th statement will execute t_j times for a value of j.

Total time taken by 4th statement = $\sum_{j=2}^{n} t_j$.

Algorithm (Insertion Sort):

		Cost	Times
	Insertion sort (A)		
1.	For j=2 to length(A)	c_1	n
2.	key = A[j]	c_2	n-1
3.	i = j-1 .	c_3	n-1
4.	while (i>0 && A[i] > key)	c_4	$\sum_{j=2}^{n} t_j$
5.	A[i+1] = A[i] .	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6.	i = i-1 .	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7.	end of while		
8.	A[i+1] = key .	c_8	n-1

Total time taken by insertion sort:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

There are 3 diff. states for time complexity:

(i) Best case : sorted input

(ii) Worst case : input in reverse order

(iii) Average case : not completely sorted not reverse.

The states depends on the type of input.

Best Case Time Complexity.

Input is in sorted order.

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \times 0 + c_6 \times 0 + c_8(n-1)$

$= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8)$.

Let $c_1 + c_2 + c_3 + c_4 + c_8 = a$,

$-c_2 + c_3 + c_4 + c_8 = b$

$\Rightarrow T_n = an + b$.

$\Rightarrow T_n = O(n)$.

Worst Case Time Complexity.

Input is in reverse order

$T(n) = c_1 n + c_2(n-1) + c_3 n - c_3 + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_8(n-1) + c_8 n - c_8$.

$\sum_{j=2}^{n} (t_j)$

$= 2 + 3 + 4 + \cdots + n$

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

$\sum_{j=2}^{n} (t_j - 1)$

$= 1 + 2 + 3 + 4 + \cdots + (n-1)$

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

$\Rightarrow T(n) = \left(c_1 + c_2 + c_3 + \frac{1}{2}c_4 - \frac{1}{2}c_5 - \frac{1}{2}c_6\right)n + \left(\frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6\right)n^2 - (c_2 + c_3 + c_4 + c_8)$.

Let

Let $a = \frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6$, $b = c_1 + c_2 + c_3 + \frac{1}{4}c_4 - \frac{1}{2}c_5 - \frac{1}{2}c_6$,

$c = -(c_2 + c_3 + c_4 + c_8)$.

$\Rightarrow T(n) = an^2 + bn + c$

$\Rightarrow T(n) = O(n^2)$.

$n \rightarrow$ linear

$n^2 \rightarrow$ quadratic.

21/7/2015.

Growth of functions.

Running time / execution time.

$T(n)$ → function of input size.

n → input size.

* When $n \to \infty$ becomes very large, the behaviour of the running time is characterised by asymptotic notations.

* Asymptotic behaviour of execution time means with large n how the function will grow.

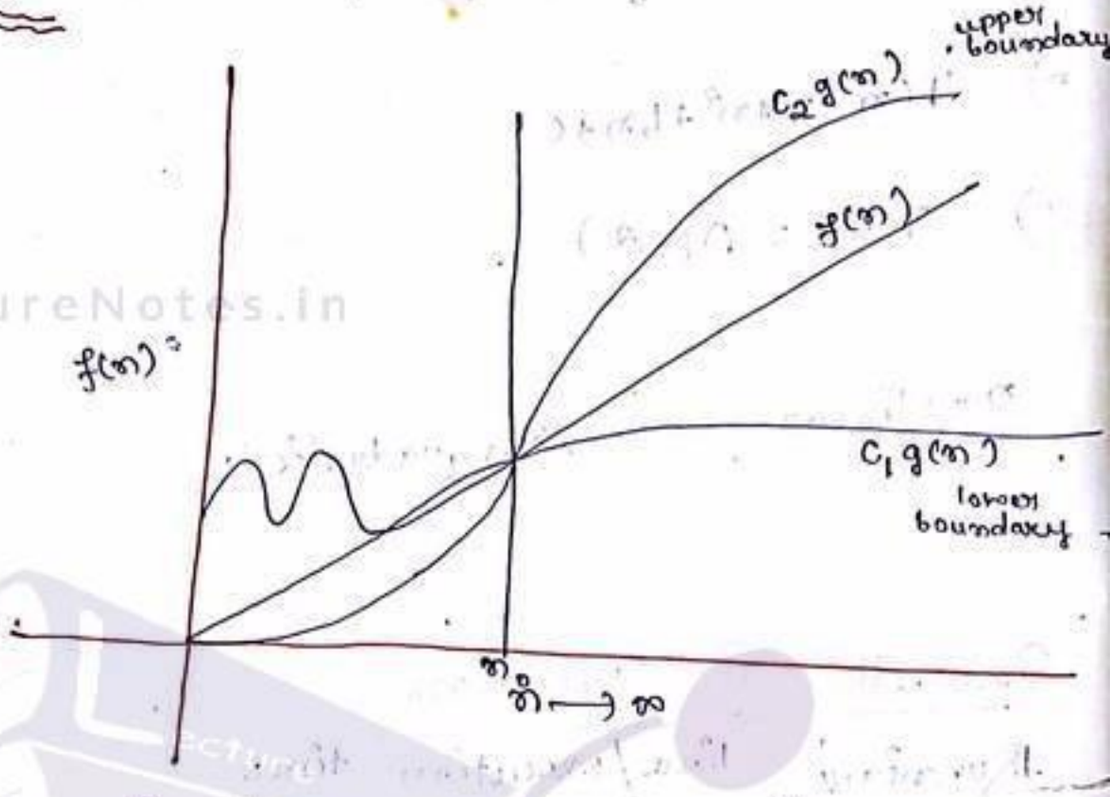
* Asymptotic notations are used to characterize the asymptotic growth of the functions.

* There are 5 asymptotic notations:
  i) Θ (Theta) notation.
  ii) O (Big-oh) notation.
  iii) Ω (ohmega) notation.

iv) O (little-oh) notation.

v) ω (little omega) notation.

i) Θ-Notation.



For a given function $g(n)$, $\Theta(g(n))$ gives a set of function which we can write.

$\Theta(g(n)) = \{ f(n)$. If there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$.

*- Θ-notation says that for some positive constants c_1 and c_2, $f(n)$ is sandwiched. bet^n $c_1 g(n)$ & $c_2 g(n)$ for $n \geq n_0$.
Θ-notation gives the lower and upper boundary of the fun^n $g(n)$ when $n \to \infty$

It gives asymptotic tight boundary.

Q. Prove that

$\frac{1}{2}n^2 - 3n = \Theta(n^2)$

$c_1 g(n) \le f(n) - (1)$ $f(n) \le c_2 g(n) - (2)$

$c_1 g(n) \le f(n) \Rightarrow c_1 n^2 \le \frac{n^2}{2} - 3n$

$\Rightarrow c_1 \le \frac{1}{2} - \frac{3}{n}$.

Put n

Proof :

$f(n) = \frac{1}{2}n^2 - 3n$

$g(n) = n^2$.

$f(n) \le c_2 g(n)$

$\frac{n^2}{2} - 3n \le c_2 n^2$

$\frac{1}{2} - \frac{3}{n} \le c_2$

Find c_1, c_2, n_0 so that

$0 \le c_1 n^2 \le \frac{n^2}{2} - 3n \le c_2 n^2 \quad \forall n \ge n_0$.

Divide the eq^n by n^2, ——— (1)

$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$.

consider this

$c_1 \le \frac{1}{2} - \frac{3}{n}$ ——— (2)

$\frac{1}{2} - \frac{3}{n} \le c_2$ ——— (3)

For c_2.

For $n=1$, $\frac{1}{2} - \frac{3}{n} = -5/2$.

$n=2$, $\frac{1}{2} - \frac{3}{2} = -1$.

$\frac{1}{2} - 3 = \frac{1-6}{2} = -3/2$.

$n=3$, $\frac{1}{2} - 1 = -1/2$

$n=4$, $\frac{1}{2} - \frac{3}{4} = -1/4$.

$n=5$, $\frac{1}{2} - \frac{3}{5} = \frac{5-6}{10} = -1/10$.

$n=6$, $\frac{1}{2} - \frac{1}{2} = 0$

$n=7$, $\frac{1}{2} - \frac{3}{7} = \frac{7-6}{14} = 1/14$.

$n=8$, $\frac{1}{2} - \frac{3}{8} = \frac{4-3}{8} = 1/8$

$c_1 = 1/14$ (as c_1 is positive), $n_0 = 7$.

$c_2 = 1/2$

So, $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Q: $f(n) = 2n^2+3n$
$g(n) = n^2$

Prove Q.1 @ $f(n) = \Theta(g(n))$

Proof:

$0 \le c_1 n^2 \le 2n^2+3n \le c_2 n^2 \quad \forall\ n \ge n_0$

$\Rightarrow c_1 \le 2 + \frac{3}{n} \le c_2$

$c_1 \le 2+\frac{3}{n}$, $2+\frac{3}{n} \le c_2$

$n=1$	$2+\frac{3}{1} = 5$
$n=2$	$2+\frac{3}{2} = \frac{4+3}{2} = \frac{10}{4} = \frac{5}{2}$
$n=3$	$2+\frac{3}{3} = \frac{18+3}{9} = \frac{20}{9}$
$n=4$	$2+\frac{3}{16} = \frac{32+3}{16} = \frac{34}{16} = \frac{17}{8}$

~~$n=1 \Rightarrow 2+\frac{3}{1} = 5$~~

$3n \le n^2 \quad \forall\ n \ge 3$

$\Rightarrow 3n+2n^2 \le n^2+2n^2$

$\Rightarrow 3n+2n^2 \le 3n^2 \quad \forall\ n \ge 3.$

$\Rightarrow 3n+2n^2 \le c_2 n^2$ where $c_2 = 3$,

$n^2 \le 2n^2 \quad \forall\ n \ge 1$

$\Rightarrow 3n^2 \le 2n^2+3n$

$\Rightarrow c_1 n^2 \le 2n^2+3n$ where $c_1 = 1$.

$2n^2+3n = \Theta(n^2)$.

$c_1 = 1$, $c_2 = 3$, $n_0 = 3$ ∴

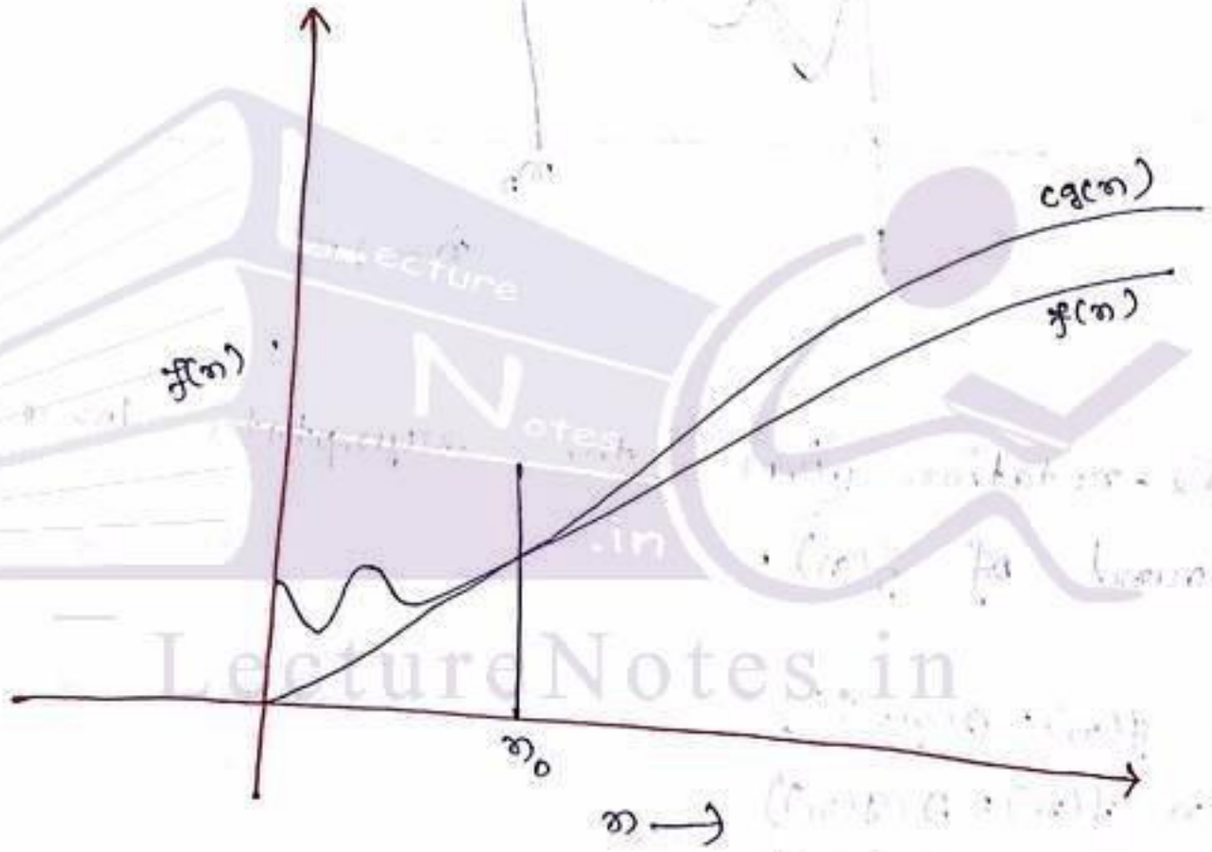
(ii) O (Big-oh) Notation.

For a given function $g(n)$, $O(g(n))$ gives a set of functions

$O(g(n)) = \{ f(n) :$ If there exist positive constants C and n_0 such that

$0 \leq f(n) \leq C\, g(n)$

for all $n \geq n_0 \}$.



* O-notation gives the upper boundary of $f(n)$. It gives the asymptotic upper boundary.

(iii) Ω (Ohmega) Notation.

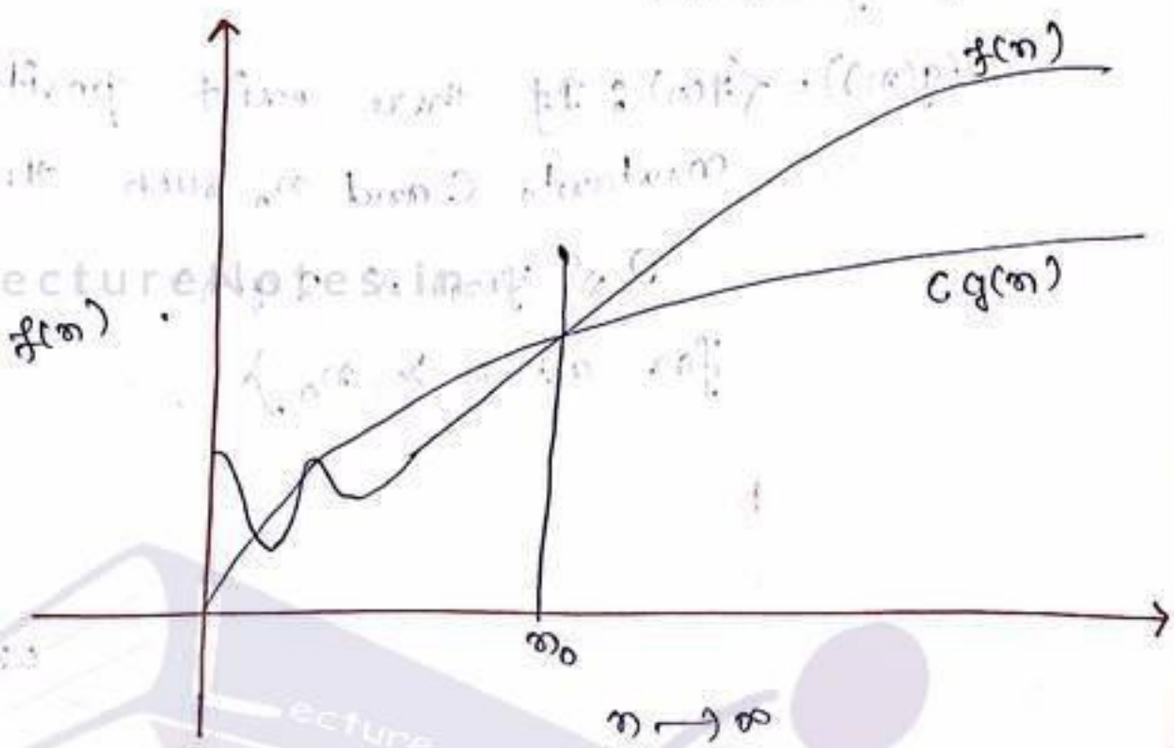
For a given $g(n)$, $\Omega(g(n))$ gives a set of functions

$\Omega(g(n)) = \{ f(n) :$ If there exist positive

constants c_1 and n_0 such that

$0 \leq c\,g(n) \leq f(n)$

for all $n \geq n_0$.



* Ω-notation gives the asymptotic lower bound of $f(n)$.

If $f(n) = \Theta(g(n))$,
then $f(n) = O(g(n))$
$f(n) = \Omega(g(n))$.

22/7/2015

$\Theta \rightarrow f(n) = \Theta(g(n))$
the $g(n)$ is asymptotic tight bound of $f(n)$.

$O \rightarrow$ asymptotic upper bound.

$\Omega \rightarrow$ asymptotic lower bound. $n^2/2 = \Omega(n)$ not

(iv) o (little-oh) Notation

* o (little-oh) notation is used to denote an upper bound that is not asymptotically tight.

o(g(n)) denotes a set of functions

$o(g(n)) = \{ f(n) :$ for any +ve constant c, there exist a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0 \}$.

Ex: $2n = o(n^2)$.

$f(n) = 2n$

$g(n) = n^2$.

Ex: $2n^2 \neq o(n^2)$.

If $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$.

but the reverse is not true.

To know $f(n)$ is o or O,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

a) $f(n) = o(g(n))$.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant}$$

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

ω (little omega) · Notation ·

ω (little-omega) defines the lower bound which is not asymptotically tight.

$\omega(g(n))$ denotes a set of functions.

$\omega(g(n)) = \{ f(n) :$ for any +ve constant c_3 there exist a constant $n_0 > 0$ such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0 \}$ ..

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

$f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

*Note:

1. $f(n) \in \omega(g(n))$

2. $f(n) = \omega(g(n))$

   $\Rightarrow g(n) = o(f(n))$ (little-oh)..

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

23/7/2015

Transitivity

1. $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$.
2. $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$.
3. $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$.
4. $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$.
5. $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

Reflexivity

1. $f(n) = \Theta(f(n))$.
2. $f(n) = O(f(n))$.
3. $f(n) = \Omega(f(n))$.

Symmetric

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

Transpose Symmetry

1. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
2. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Asymptotic Function comparison

Real Numbers - (a, b) ; (Interpretation of the notation).

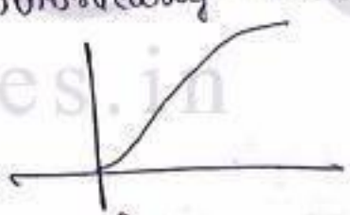
1. $f(n) = O(g(n)) \approx a \leq b$.
2. $f(n) = \Theta(g(n)) \approx a = b$.
3. $f(n) = \Omega(g(n)) \approx a \gg b$.
4. $f(n) = o(g(n)) \approx a < b$.
5. $f(n) = \omega(g(n)) \approx a > b$.

Monotonically Increasing Function

Given m and n if $m \gg n$, it implies

$$f(m) \gg f(n),$$

then $f(\cdot)$ is called a monotonically increasing function.



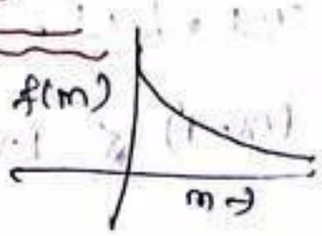
Strictly monotonically increasing function

Given m and n if $m > n$, it implies

$$f(m) > f(n),$$

then $f(\cdot)$ is called a strictly monotonically increasing function.

Monotonically Decreasing Function.



Given m and n, if $m \geq n$ implies

$$f(m) \leq f(n),$$

then $f(\cdot)$ is called a monotonically decreasing function.

Strictly Monotonically Decreasing Function.

Given m and n if $m > n$ implies

$$f(m) < f(n),$$

then $f(\cdot)$ is called a strictly monotonically decreasing function.

$\lceil x \rceil \rightarrow$ ceiling.

The least integer that is greater than or equal to x.

$\lfloor x \rfloor \rightarrow$ floor.

The greatest integer that is less than or equal to x.

Eg: $\lceil 5 \rceil = 5$ $\quad \lceil 5.97 \rceil = 6$

$\lfloor 5 \rfloor = 5$ $\quad \lfloor 5.9 \rfloor = 5$

Q: $x-1$, $\lfloor x \rfloor$, x , $\lceil x \rceil$, $x+1$.

$(x-1) \leq \lfloor x \rfloor \leq x \leq \lceil x \rceil \leq x+1$.

Polynomial.

Constant $* x^d$, $d \geq 0$.

→ Polynomial of degree (d)

$$P(x) = \sum_{i=0}^{d} a_i x^i$$ where a_d is not zero.

27/7/2015.

Standard Notation and Common Functions.

1: Floor and Ceiling

$(x-1) < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$.

$f(x) = \lceil x \rceil$

$f(x) = \lfloor x \rfloor$

} monotonically increasing functions.

2: Modulaar Arithmetic.

Gives remainder of the division.

$5 \% 2 = 1$. $5 \mid . \mid 2 = 1$

3. Polynomial.

$P(n) = \sum_{i=0}^{d} a_i n^i$, where $a_0, a_1, \ldots, a_i$ → co-efficient
n^i → exponent.
$i \geq 0$.

Polynomial $P(n)$ is of degree (d) if n^d, $d \geq 0$ and $a_d \neq 0$.

$P(n)$ is called asymptotically positive polynomial if and only if a_d is positive. $(a_d > 0)$.

$P(n)$ is called polynomially bounded if it is asymptotically +ve and

$$\boxed{P(n) = O(n^k)}$$

for some constant k.

Ex. $P(x) = 1 + x + x^2$.

$\Rightarrow P(x) = O(x^2)$.

It is a polynomially bounded function.

$P(x) = 2x + 3x^2 + 4x^3$.

$\Rightarrow P(x) = O(x^3)$.

* - All asymptotically positive polynomial are polynomially bounded.

n^a → monotonically increasing if $a \geq 0$. n → variable, a → constant

n^a → monotonically decreasing if $a \leq 0$.

Exponentials :

For all real

$a > 0$

and m, n

$a^0 = 1$

$a^1 = a$

$(a^m)^n = a^{mn}$

$(a^m)^n = a^{nm}$

$a^m \cdot a^n = a^{m+n}$.

all these terms are exponentials .

* The rates of growth of polynomials and exponentials can be related by, for all real constants a and b, $a > 1$

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

$n^b \rightarrow$ polynomial

$a^n \rightarrow$ exponential .

$\Rightarrow \quad n^b = o(a^n)$.

needs to

* We design algorithms which are polynomially bound not exponential .

2. $e^x \rightarrow$ exponential form .

Logarithmic

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x ,$$

for all x.

Logarithmic.

$\lg n = \log_2 n$

$\ln n = \log_e n$

$\lg^k n = (\lg n)^k$.

$\lg \lg n = \lg(\lg n)$

logarithm is monotonically increasing fun^n.

For $b > 1$, for $n > 0$, $\log_b n$ is strictly increasing function.

$$b^{\log_b a} = a$$

Lim:

$$2^{\log_2 n} = n$$

A function $f(n)$ is called polylogarithmically bounded

$$f(n) = O(\lg^k n)$$

$$\lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0 \Rightarrow \lg^b n = o(n^a).$$

28/7/2019

$$\lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0$$

for any constant $a > 0$,

so, $\lg^b n = o(n^a)$.

A polynomial function grows faster than polylogarithmic function.

Factorials

$$n! = \begin{cases} 1 & , n = 0 \\ n(n-1)! & , \text{if } n > 0 \end{cases}$$

Stirling's Approximation

$$n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + \Theta(1/n))$$

n^n is a weak upperbound of $n!$.

$n! = o(n^n)$

$n! = \omega(2^n)$

$\lg(n!) = \Theta(n \lg n)$

For $n \geq 1$,

$$n! = \sqrt{2\pi n}\,(n/e)^n\,e^{\alpha_n},$$

where $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$

Functional Iteration.

$f^{(i)}(n)$ denote the function $f(n)$ iteratively, applied i times to an initial value of n.

Let $f(n)$ be a function on real numbers. For non-negative value of i

$$f^i(n) = \begin{cases} n & , \text{if } i = 0 \\ f(f^{i-1}(n)) & , \text{if } i > 0 \end{cases}$$

Base condition $f^0(n) = n$

Ex: $f^1(n) = 2n = 2f^0(n)$

$f^3(n) = 2^3 n$.

$f^2(n) = 2f(f^1(n)) = 2f$

$f^i(n)$ → applied i times on n.

$f^1(n) = f(f^0(n)) = f(n)$

$f(n) = 2^n + 1 \quad (f^0(n) = n)$.

$f^2(n) = f(f(n)) = f(n+1) = (n+1)+1 = n+2$.

$f^3(n) = f(f^2(n))$
$= f(n+2) = n+2+1$
$= n+3$.

$f^i(n) = f(f^{i-1}(n))$
$= f(n+(i-1)) = (n+(i-1))+1$
$= n+i$.

Iterated Logarithm Function.

$\log^{*} n$

$\log^{*} 2 = 1$

$\log^{*} 4 = 2$

$\log^{*} 16 = 3$.

$\log^{*} 2^{65536} = 5$.

* - Growth rate of iterative logarithmic function is very slow.

Maximum is 5.

$A = \log^{*} n$, $B = n^6$.

$A = O(B)$ and $A = o(B)$.

as $\lim_{n \to \infty} \frac{\lg^b n}{n^a} = 0$,

$A = n^k$, $B = c^n$

$\lim_{n \to \infty} A/B = C$

$\Rightarrow A = \Theta(B)$

$\lim_{n \to \infty} A/B = 0$

$\Rightarrow A = O(B)$

$\lim_{n \to \infty} A/B = \infty$

$\Rightarrow A = \Omega(B)$

* $A = n^{1/2}$, $B = n^{\sin n}$

$A = \Theta(B)$, $A = O(B)$, $A = \Omega(B)$

* $A = 2^n$, $B = 2^{n/2}$

$$\lim_{n \to \infty} \frac{2^n}{2^{n/2}}$$

$$= \lim_{n \to \infty} 2^n \cdot \frac{1}{2^{n/2}} \quad \text{(Divide } 2^n\text{)}$$

$$= \lim_{n \to \infty} \frac{1}{1/2^{n/2}} = \infty$$

$\Rightarrow A = \Omega(B)$

$\Rightarrow A = \Omega(B)$

$$\frac{1/1}{1/2^{n/2}} = 1/1 \times 2^{n/2}/1$$

* $A = n^{\log c}$ $B = c^{\log n}$

* $A = \log(n!)$, $B = \log n^n$

$= n \log n$ $= n \log n$

$$\lim_{n \to \infty} \frac{\log(n!)}{n \log n} = 1$$

$\Rightarrow A = O(B)$, $A = \Omega(B)$

$A = n^{\log C}$, $B = C^{\log n}$ $C > 0$ (n can never be 0 or 1)

$\lim_{n \to \infty} \frac{n^{\log C}}{C^{\log n}}$

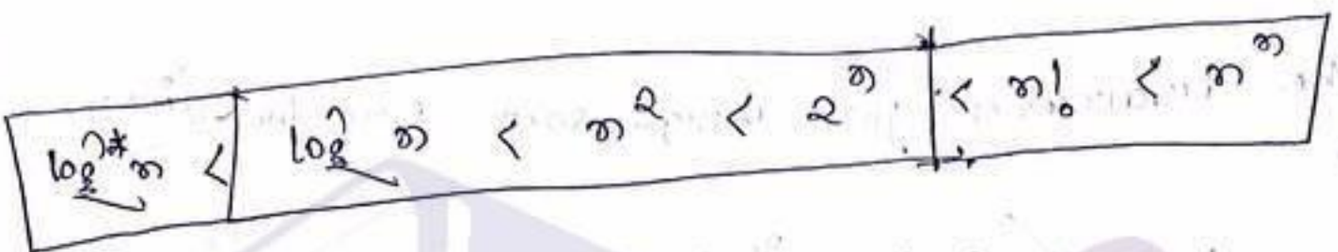
$n^{\log C}$ → polylogarithm

$C^{\log n}$ → exponential

$\Rightarrow \lim_{n \to \infty} \frac{n^{\log C}}{C^{\log n}} = 0$

$\Rightarrow A = O(C^{\log n}) = O(B)$

$A = O(B)$.



Ex: $\log(\log^* n)$, $2^{\log^* n}$, $(\sqrt{2})^{\log n}$, n^2, $n!$, $(\log n)!$

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

54.

$f(n) = \frac{1}{2}n^2 - 3n$ $\quad g(n) = n^2$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\frac{1}{2}n^2 - 3n}{n^2} = \lim_{n \to \infty} \left(\frac{1}{2} - \frac{3}{n}\right)$

$= \frac{1}{2}$.

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

Recurrences.

When an algorithm makes a recursive call to itself, its running time can be expressed by recurrences.

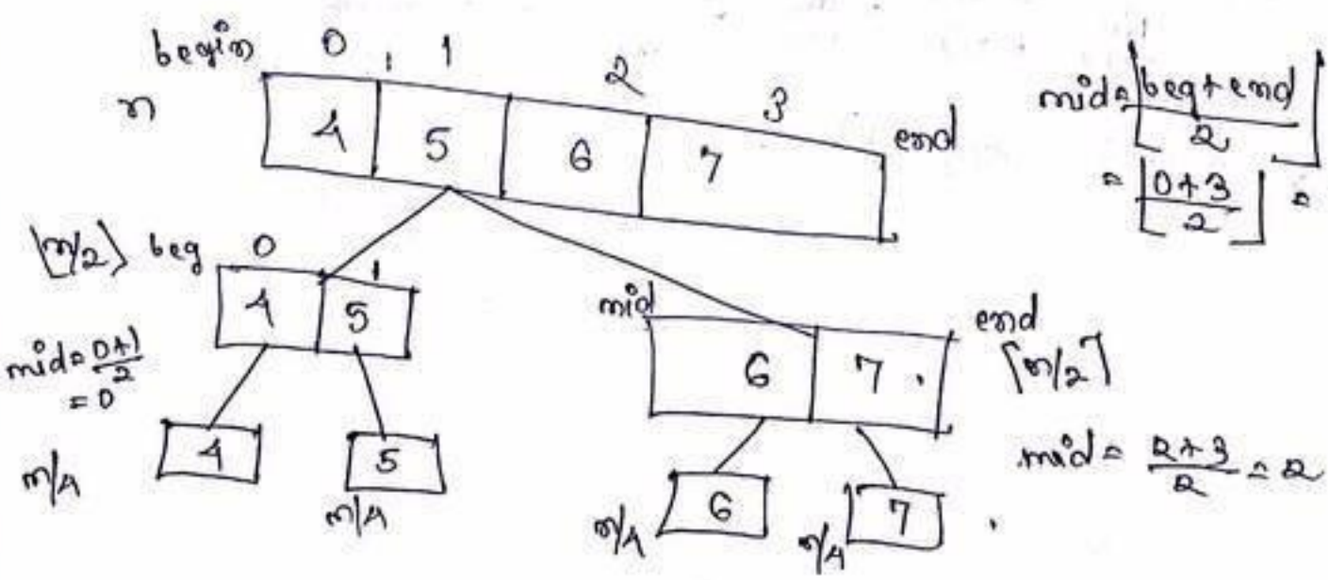
* A recurrence is an equation or inequality that describes a function in terms of its value for smaller inputs.

Eg: The recurrence for merge sort can be given as

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ 2T(n/2) + O(n) & \text{if } n>1 \end{cases}$$

Divide and Conquer.

1. Divide the problem into sub-problem.
2. Conquer: solve the sub-problems.
3. Combine: Combine the result of sub-problem to whole solution.



* The problem of size (n) is divided into "a" parts each part is of size $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ and $f(n)$ is asymptotically dividing and combining time.

Theorem 4.1

Let $a \geq 1$, $b > 1$ be constants. Let $f(n)$ be a function and let $T(n)$ be defined on the non-negative integers by the recurrence recurrence

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

then $T(n)$ can be bounded asymptotically as follows:

(i) if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(ii) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

(iii) if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $\boxed{a f(n/b) \leq c f(n)}$ → Regularity condition for some constant $c < 1$ and all sufficiently large n, then

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

Q. Solve the given recurrence

$$T(n) = 9T(n/3) + n$$

Sol:—

$$T(n) = 9T(n/3) + n$$

Here, $a = 9$, $b = 3$, $f(n) = n$.

Step-1:

Now, $n^{\log_b a} = n^{\log_3 9} = n^{\log_3 3^2} = n^2$

Step-II:

$f(n) = n$

$= O(n)$

$= O(n^{2-1})$

$= O(n^{\log_b a - 1})$

$= O(n^{\log_b a - \epsilon})$, where $\epsilon = 1$.

$f(n) = O(n)$

$\Rightarrow f(n) \leq cg(n)$

$n \leq cn$

$\Rightarrow f(n) \leq (c)n$

$\therefore f(n) = O(n^{\log_b a - \epsilon})$

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

$= \Theta(n^2)$

Q. $T(2n/3) + 1$

$a = 1$, $b = 3/2$, $f(n) = 1$.

$\log 1 \rightarrow$ any base is 0.

Step-I:

$n^{\log_b a} = n^{\log_{3/2} 1}$

$= n^0 = 1$.

Step-II:

$f(n) = 1$

$= \Theta(1)$

$= \Theta(n^{\log_b a})$

$1 = \Theta(n)$

$= O(n)$

$= \Omega(n)$.

$\therefore T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n) = \Theta(\lg n)$

Q: $T(n) = 3T(n/4) + n \log n$.

Sol:

$a = 3,\ b = 4,\ f(n) = n \log n$.

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$.

$f(n) = n$

$= n^{0.793 + 0.207}$

$= n^{0.793} \cdot n^{0.207}$

$= n^{\log_b a + \epsilon}$ if $\epsilon = 0.207$.

$= \Omega(n^{\log_b a + \epsilon})$.

4/7/2015.

The rule-3 will fit for the solution.

Check for regularity condition.

$a f(n/b) = 3 f(n/4) = 3\left(\frac{n}{4}\right)$. $(\because f(n) = n)$.

$\Rightarrow 3(n/4) \leq 0.9(n)$, $c = 0.9$.

$\therefore$, $f(n)$ satisfies regularity condition.

$\therefore$, $T(n) = \Theta(f(n))$

$= \Theta(n)$.

Q: $T(n) = 4\ T(n/2) + n$.

$a = 4,\ b = 2,\ f(n) = n$.

$n^{\log_b a} = n^{\log_2 4} = n^{0.79} \quad n^{\log_2 2^2} = n^2$

$f(n) = n$

$= n^{2-1}$

$= \left(n^{\log_b a - \epsilon}\right)$. $= \Omega(n^{\log_b a - \epsilon})$

∴ $T(n) = \Theta(n^{\log_b a})$.

$= \Theta(n^2)$ (By rule-1).

Q. $T(n) = 4T(n/2) + n^2$.

$a = 4, b = 2, f(n) = n^2$.

$n^{\log_b a} = n^{\log_2 4} = n^2 = \Theta(n^2) = \Theta(n^{\log_b a})$.

$n^2 = n \cdot n$.

$f(n) = n^2$

$= n^{\log_b a}$.

∴ $T(n) = \Theta(n^2 \log n)$.

Q. $T(n) = 4T(n/2) + n^3$.

$a = 4, b = 2, f(n) = n^3$.

$n^{\log_b a} = n^{\log_2 4} = n^2$.

$f(n) = n^3 = n^{2+1} = n^{\log_b a + \epsilon} = \Omega(n^{\log_b a + \epsilon})$

∴ $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

$af(n/b) = 4f(n/2) = 4 \cdot \frac{n^3}{8} = n^3/2 \leq 0.2(n^3)$.

∴ $T(n) = \Theta(n^3)$.

Observations of Master's Rule :

In rule-1, $f(n)$ is asymptotically less than $n^{\log_b a}$ by the amount or value n^{ϵ} for $\epsilon > 0$.

i.e., $f(n)$ is polynomially smaller than $n^{\log_b a}$,

$f(n) = O(n^{\log_b a})$.

$\Rightarrow$ $f(n)$ is equal to $n^{\log_b a}$. (Rule-2)

$f(n) = \Omega(n^{\log_b a + \epsilon})$ (Rule-3)

$f(n)$ is greater than $n^{\log_b a}$ by amount n^{ϵ}.

i.e., $f(n)$ is polynomially greater than $n^{\log_b a}$.

Q: $T(n) = 2T(n/2) + n \log n$.

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$.

$f(n) = n \log n$

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

$\Rightarrow \frac{f(n)}{n^{\log_b a}} = \log n$. (Logarithmically greater)

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

$a = 2,\ b = 2,\ f(n) = n \log^2 n$

$n^{\log_b a} = n^1 = n.$

$f(n) = n \log n = \Omega(n^{\log_b a})$ i.e. $= \Omega(n)$.

$f(n)$ is asymptotically larger than $n^{\log_b a}$. But, not polynomially larger.

Regularity cond^n:

$2f(n/2) \le cf(n)$

$2(n/2)\log^2(n/2) \le 0.2\, n \log^2 n$

$n \log^2 n - n \log n \le 0.2\, n \log^2 n$ (satisfies).

Thm: 4.4.1

If $f(n) = \Theta(n^{\log_b a} \log^k n)$,

then the master recurrence

$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

$f(n) = n \log n$

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

$= \Theta(n^{\log_b a} \log^1 n)$

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

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

iii) The Recursion Tree Method.

Recurrence Tree.

Given recurrence for merge sort

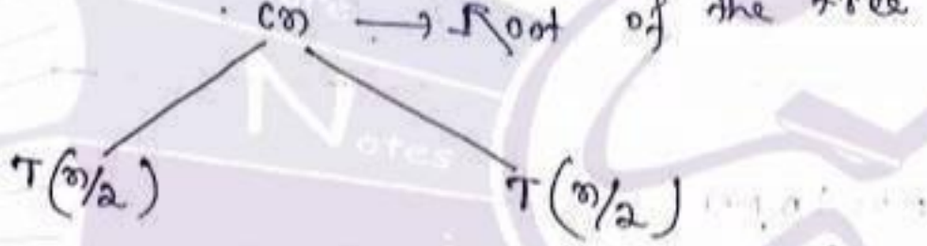
$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Construction of recursion tree.

(a) $T(n)$

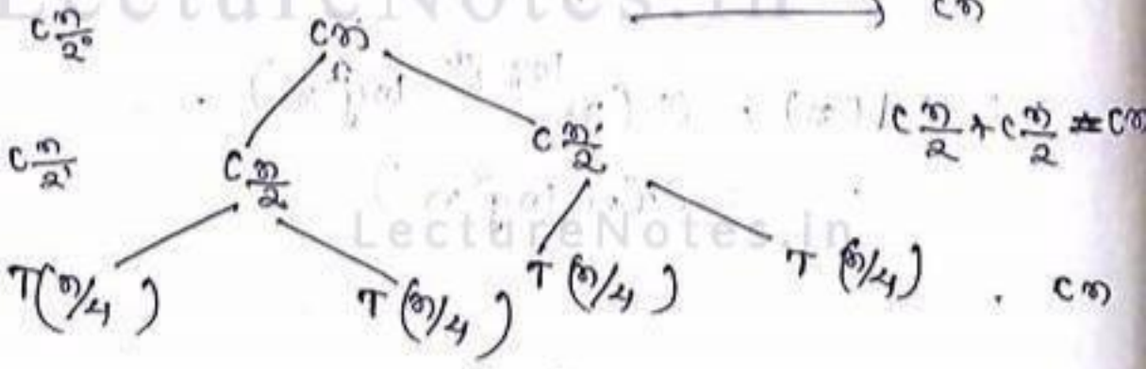
(b) $T(n) = T(n/2) + T(n/2) + \Theta(n)$

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

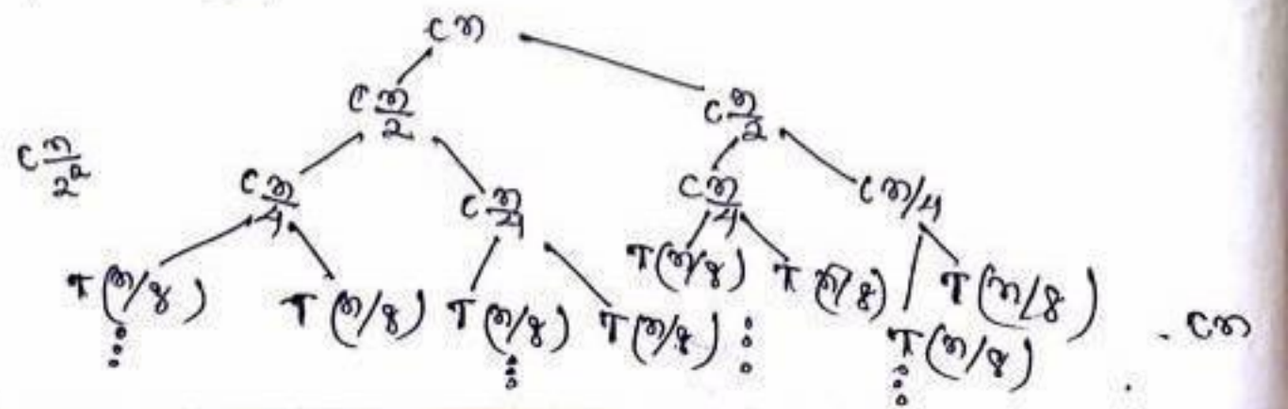


$T(n/2) = T(n/4) + T(n/4) + c\frac{n}{2}$

Complete binary tree



$T(n/4) = T(n/8) + T(n/8) + c\frac{n}{4}$



$\frac{c_n}{2^k}$ $T(1) = c$ c c c c c c , $\longrightarrow cn$

No. of c is equal to no. of n.

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

$\Rightarrow n/2^k = 1$

$\Rightarrow \log(n) = \log(2^k)$

$\Rightarrow k = \log n$.

$\Rightarrow T(n) = cn(\log n + 1)$ (for height $(k+1)$).

$= cn\log n + cn$

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

Q. $T(n) = \begin{cases} T(n/3) + T\left(\frac{2n}{3}\right) + cn & \text{if } n > 1 \\ 1 & n = 1 \end{cases}$

6/8/2015.

Arranging the following functions in ascending order:

$n \log n, (\log n)^3, 2^n, 4^{\log n}, 2^{2^n}, n^3, n^{100}$.

$\log^3 n < n \log n < 4^{\log n} < n^3 < n^{100} < 2^{2^n}$.

$$a^{\log_b c} = c^{\log_b c}$$

$4^{\log n} = n^{\log 4} = n^{\log 2^2} = n^{2 \log 2} = n^{2 \cdot 1} = n^2$.

Q: $n^3 + 2n^2 + 2n = \Theta(n^3)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \Rightarrow f(n) = \Theta(g(n))$$

A:

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

$g(n) = n^3$.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^3 + 2n^2 + 2n}{n^3}$$

$$= \lim_{n \to \infty} \left(1 + \frac{2}{n} + \frac{2}{n^2}\right)$$

$$= 1 + 2/\infty + 2/\infty^2$$

$$= 1 + 0 + 0$$

$= 1$.

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

ii) $n^3 + 2n^2 + 2n = \Theta(n^3)$.

Q: P.T $\log(n!) = \Theta(n \log n)$.

Sol:

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

$g(n) = n \log n$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$

$\Rightarrow \lim_{n \to \infty} \frac{\log(n!)}{n \log n}$

$= \lim_{n \to \infty}$

By Stirling's approximation

$n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + \Theta(1/n))$

$\Rightarrow \log(n!) = \log\left(\sqrt{2\pi n}\,(n/e)^n\,(1 + \Theta(1/n))\right)$

$= \log(\sqrt{2\pi n}) + \log(n/e)^n + \log(1 + \Theta(1/n))$

$= \sqrt{2\pi}\,\log n^{1/2} + n \log(n/e) + \log(\Theta(1/n))$

(∵ neglecting one)

$= \sqrt{2\pi} \cdot 1/2 \log n + n \log n - n \log e + \log(c\,\frac{1}{n})$

$= \sqrt{\pi/2}\,\log n + n \log n - n \log e + \log c - \log n$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \sqrt{\pi/2}\,\frac{\log n}{n \log n} + \frac{n \log n}{n \log n} - n\frac{\log e}{n \log n} + \frac{\log c}{n \log n} - \frac{\log n}{n \log n}$$

$$= \sqrt{\pi/2}\,\frac{1}{n}^{0} + 1 - \frac{\log e}{\log n}^{0} + \frac{\log c}{n \log n}^{0} - \frac{1}{n}^{0}$$

$= 0 + 1 - 0 + 0 - 0$

$= 1$

$\Rightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \Rightarrow f(n) = \Theta(g(n)) \Rightarrow$ **$\log(n!) = \Theta(n \log n)$.**

10/8/2015.

Q. Solve the recurrence

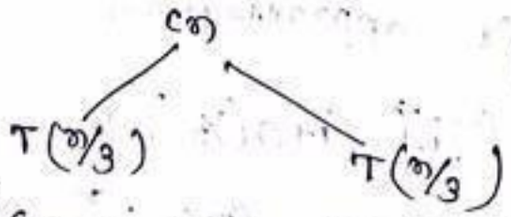
$$T(n) = T(n/3) + T\left(\frac{2n}{3}\right) + O(n)$$

$$= T(n/3) + T\left(\frac{2n}{3}\right) + cn \quad \text{if } n > 1$$

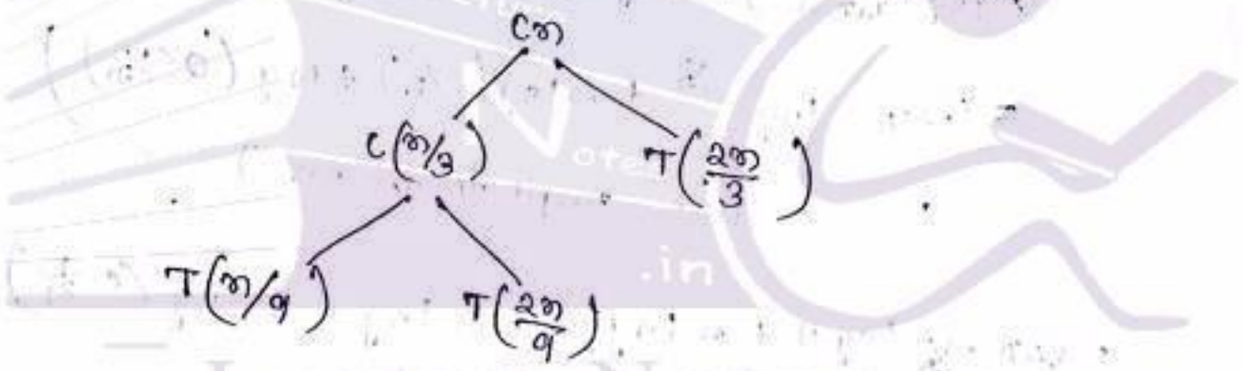
$$T(n) = 1 \quad \text{if } n = 1.$$

(a) $T(n)$

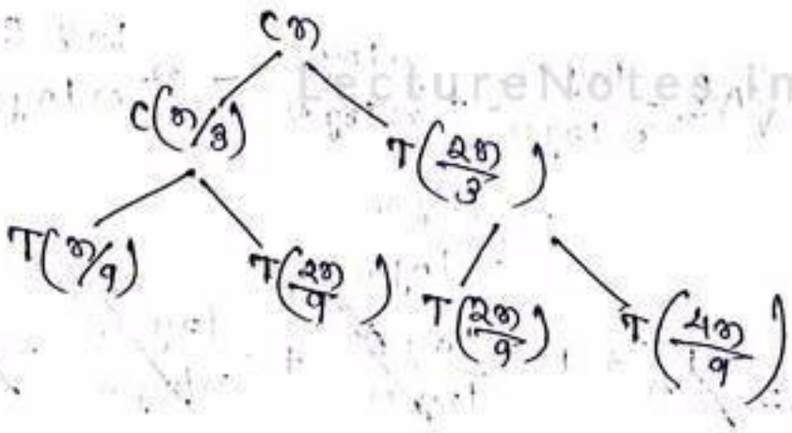
(b)



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



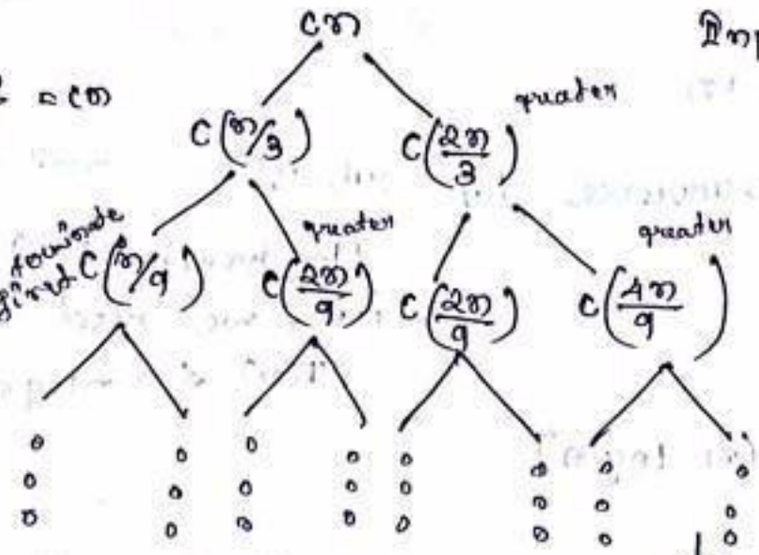
$$T\left(\frac{2n}{3}\right) = T\left(\frac{2n}{9}\right) + T\left(\frac{4n}{9}\right) + c\left(\frac{2n}{3}\right)$$



Cost : cn

Input size → n

$c(\frac{n}{3}) + c\frac{2n}{3} = cn$



length of ↓ Path

longest branch = height

$n + \frac{2n}{3} + \frac{4n}{9} + \cdots (\frac{2}{3})^k n$

Height = $(2/3)^k n = 1$

$\Rightarrow n = 1/(2/3)^k = (3/2)^k$

$\Rightarrow \log_{3/2} n = k$

$\Rightarrow k = \log_{3/2} n \simeq \log n$

$T(n) = cn \log \cdot cn \cdot k$

$= cn \cdot \log n$

$\Rightarrow$ **$T(n) = O(n \log n)$.**

Substitution Method

Step-I:

Make a guess.

Step-II:

Prove the guess by induction method.

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

Solve the recurrence by substitution method.

Ans:

To prove $T(n) = O(n \log n)$
i.e., we need to prove
$T(n) \leq c\, n \log n$ for $c > 0$.

Step-1.

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

Step-2.

Assume that the solution holds for $n/2$.

i.e., $T(n/2) = O(n/2 \log(n/2))$

i.e., $T(n/2) \leq c\, n/2 \log(n/2)$ for $c > 0$.

With this assumption, prove that

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

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

$= 2O(n/2 \log(n/2)) + n$

$\leq 2c(n/2 \log(n/2)) + n$

$\leq c\, n \log(n/2) + n$

$\leq c\, n (\log n - \log 2) + n$

$\leq cn \log n - cn + n$

$\leq cn \log n$ for $c \geq 1$

$\Rightarrow T(n) \leq cn \log n$

$\Rightarrow \boxed{T(n) = O(n \log n)}$

Prove the base condition

$T(1) = 1$

$T(n) \leq n \log n$

For $n = 1$

$T(1) \leq c\,1 \log 1 \quad (c = 1)$

$\Rightarrow 1 \leq 0$ (not satisfies) ∴

$n = 2$

$T(2) = 2\,T(2/2) + 2$

$= 2 \cdot 1 + 2 = 4$.

$T(2) \leq cn \log n$

$\leq 2c\,2 \log 2 \quad (\because c = 2)$

$\leq 4 \log 2$

≤ 4 for $n = 2$, $c \geq 2$.

$T(2) = O(n \log n)$ for $n = 2$, $c \geq 2$.

Hence, it is proved.

Matrix - Matrix Multiplication.

Divide and Conquer Method.

Matrix - Matrix Multiplication.

Divide and Conquer Method.

8/2015.

Merge Sort.

Input : A sequence of numbers

$\langle a_1, a_2, \cdots, a_n \rangle$

Output : A permutation of the input sequence

$\langle a_1', a_2', \cdots, a_n' \rangle$ such that

$a_1' \leq a_2' \leq \cdots \leq a_n'$

Ex: Input sequence : $\langle 5\ 4\ 3\ 2\ 1 \rangle$

Output : $\langle 1\ 2\ 3\ 4\ 5 \rangle$.

*- It follows divide and conquer approach.

*- Any algorithm which is recursive in nature follows divide and conquer approach.

Ex:

```
fact(n)
{
    result = n * fact(n-1);
}
```

Divide and Conquer.

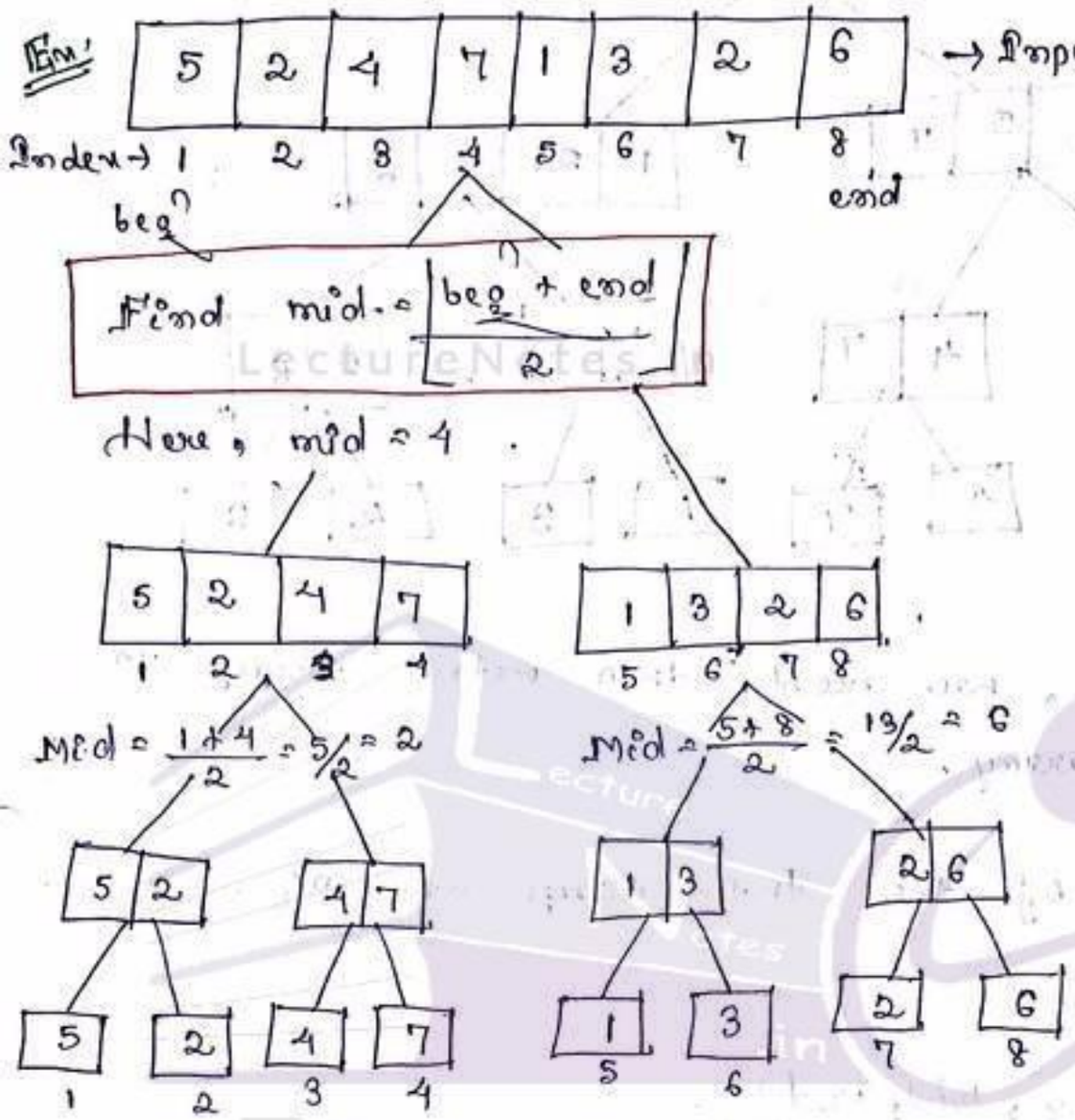
3 steps :

(i) Divide : the problem into sub-problems of same kind.

(ii) Conquer : the sub-problems/solve the sub-problem.

iii) Combine : the solution of the sub-problem to find the whole solution.

Eg:



Algorithm of Merge Sort. MergeSort (A, beg, end)

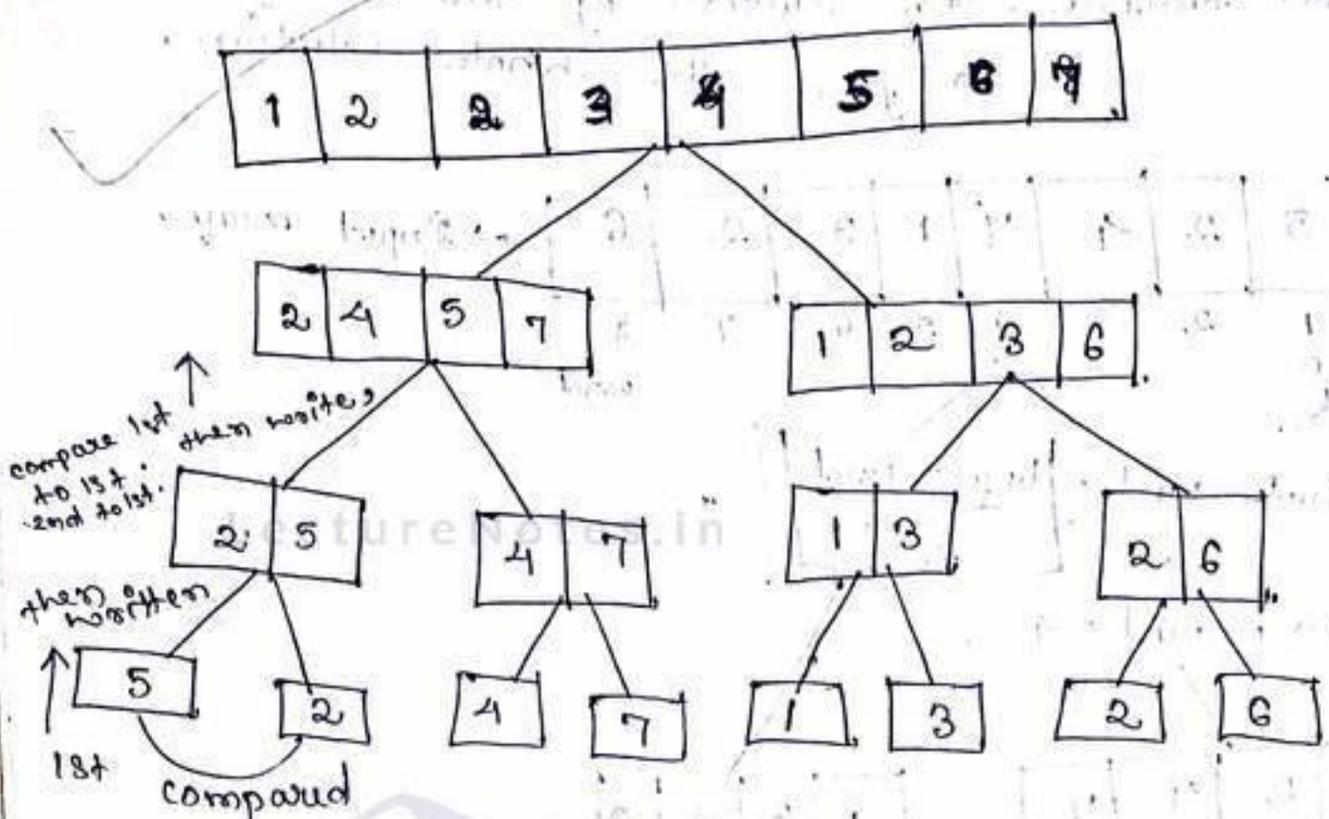
1: if beg < end

2: mid = $\frac{beg + end}{2}$

3: mergesort (A, beg, mid)

4: mergesort (A, beg, mid+1, end)

5: merge (A, beg, mid, end)



*- At a time, we need two extra arrays to sort the array.

*- The size of the two arrays are $n/2$.

Merge (A, beg, mid, end)

1: $n_1 \leftarrow$ mid - beg + 1

2: $n_2 \leftarrow$ end - (mid+1) + 1 = end - mid - 1 + 1 = end - mid.

3: Create two arrays $L_1 = [n_1 + 1]$
and $L_2 = [n_2 + 1]$.

4: for $i \leftarrow 1$ to n_1

5: do $L[i] \leftarrow A[beg + i - 1]$.

(Line 4 and 5 of the algorithm copies the 1st 1 to n_1 elements of A to L_1 and it assumes

that those elements are already sorted).

12/8/2015.

6: for $j = 1$ to n_2

7: do $L_2[j] \leftarrow A[mid + j]$;

// The algorithm assumes that elements of L_1 and L_2
// are sorted.

8: $L_1[n_1+1] \leftarrow \infty$

9: $L_2[n_2+1] \leftarrow \infty$

10: $i \leftarrow 1$

10: for $k = beg$ to end.

11: $j \leftarrow 1$

12: for $k = beg$ to end.

13: if $L_1[i] \leq L_2[j]$.

14: $A[k] \leftarrow L_1[i]$

15: $i \leftarrow i+1$

16: else

17: $A[k] \leftarrow L_2[j]$

18: $j \leftarrow j+1$.

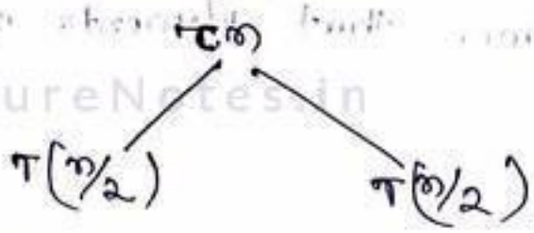
19: [end of for]

20: [end merge].

Analysis of Merge Sort

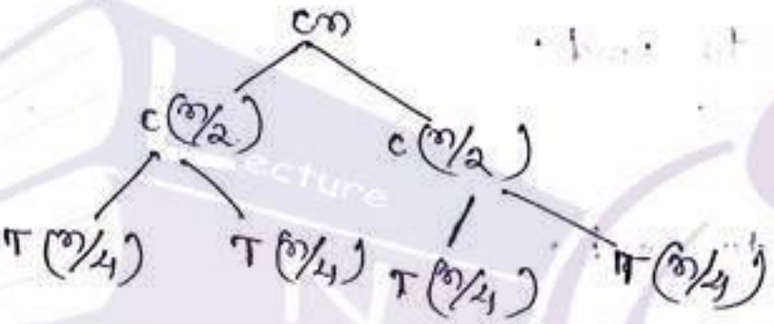
Recurrence

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise} \\ = 2T(n/2) + cn \end{cases}$$

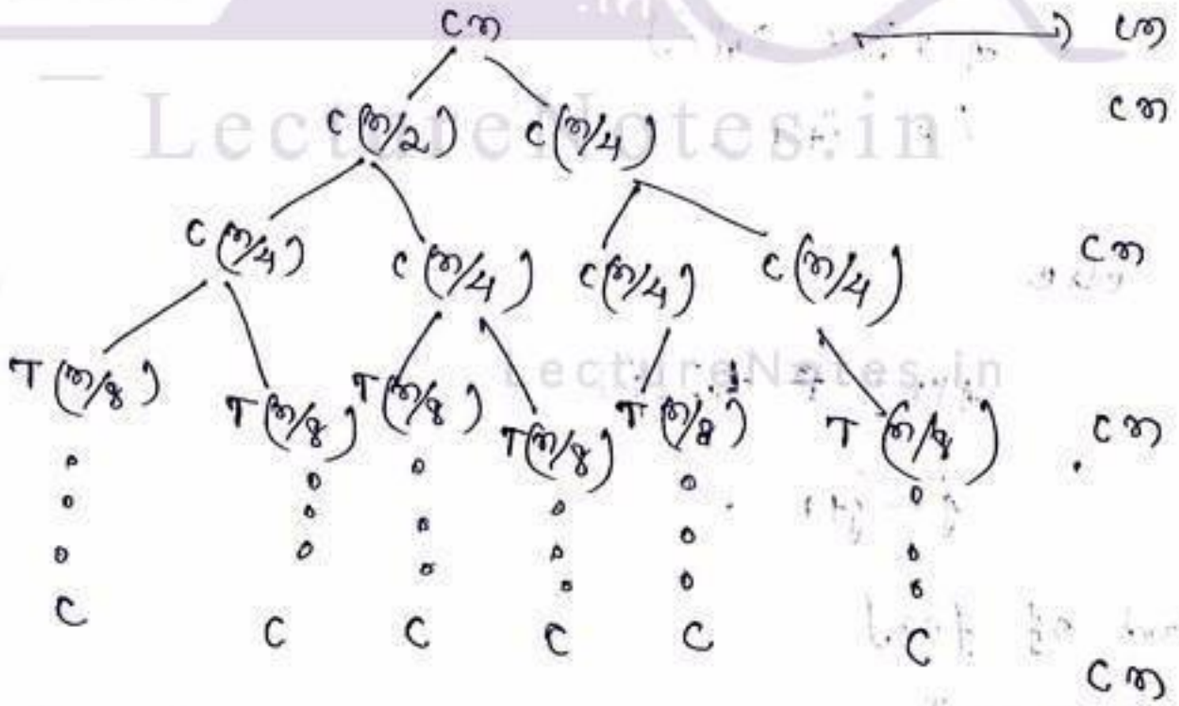


Height of complete binary tree is $\log n$.

$T(n/2) = 2T(n/4) + c(n/2)$



$T(n/4) = 2T(n/8) + c(n/4)$



Total cost.

$T(n) = cn + c(n) + \cdots + c(n)$

$= cn \log n$

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

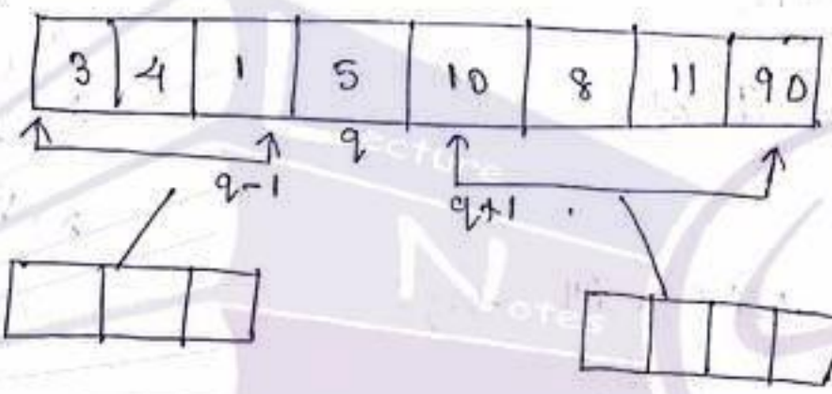
13/8/2015.

Quick Sort.

- Follows divide and conquer approach.

Divide: Partitions the input array $A[p \ldots r]$ into two sub-arrays $A[p \ldots (q-1)]$ and $A[q+1 \ldots r]$ such that all the elements of $A[p \ldots (q-1)]$ are less than or equal to $A[q]$ and all the elements of $A[(q+1) \ldots r]$ are greater than or equal to $A[q]$.

A:



Conquer: Sort the sub-arrays $A[p \ldots (q-1)]$ and $A[(q+1) \ldots r]$ by calling quick sort recursively.

Combine: Since the sub-arrays are sorted in place no need to combine and $A[p \ldots r]$ is now sorted.

Algorithm.

Quick sort (A, p, r)

if $p < r$

then $q \leftarrow$ Partition (A, p, r).

p → beginning of index
r → end of index of array.

3: Quicksort (A, P, (q-1))

4: Quicksort (A, (q+1), r)

5: End.

Algorithm of Partition.

Partition (A, P, r)

1: Pivot ← A[r]

2: i ← (P-1)

3: for (j = P to (r-1))

4: if A[j] <= Pivot

5: i ← i+1

6: swap A[i] with A[j]

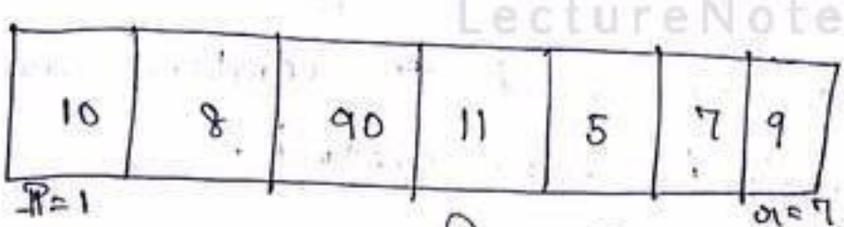
7: End of for.

8: Exchange A[i+1] with A[r]

9: Return (i+1)

10: End.

3 parts
A[P] A[q-1]
A[q] → pivot
A[q+1] A[r]



Find q by calling partition().

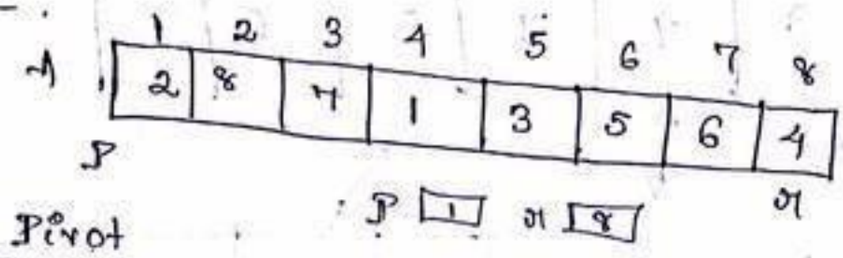
Pivot element = A[r] = 9.

i = P-1 = 0.

17/8/2015.

When p = r, the array contains single elements.

Input:

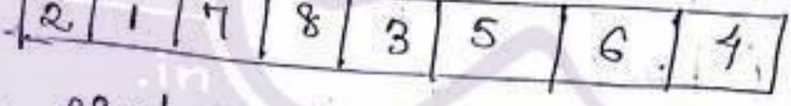


A[p] = 2
A[p+1] = 8.

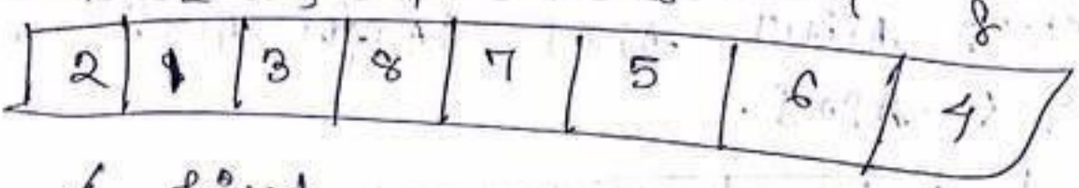
p 1 r 8

Pivot
4

i 0
j 1 A[j] = 2 < pivot.
i 1 swap.
j 2 A[j] = 8 ≮ pivot.
j 3 A[j] = 7 ≮ pivot.
j 4 A[j] = 1 < pivot.
i 2
swap A[i], A[j]

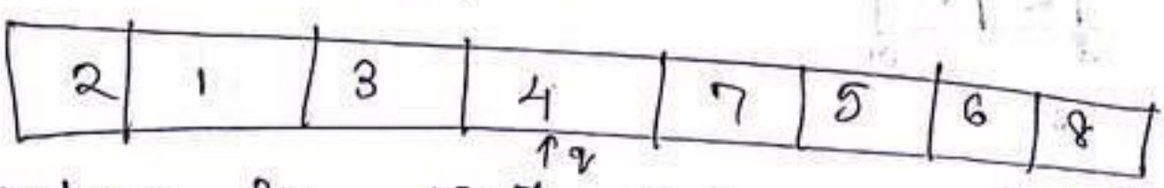


j 5 A[j] = 3 < pivot
i 3
swap



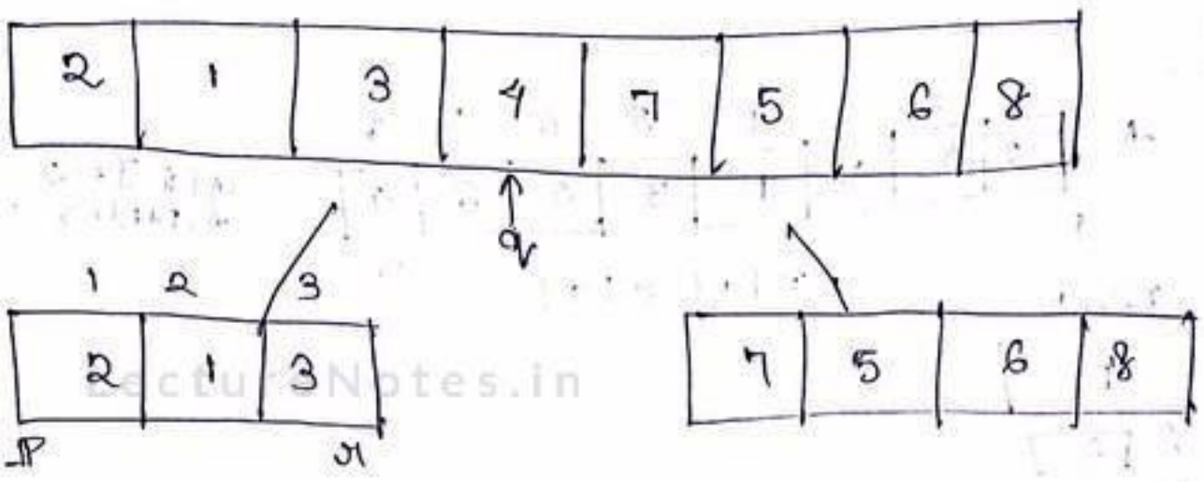
j 6 ≮ pivot
j 7 ≮ pivot (end of loop)

swap i+1 [4], A[r].



return i+1 A[4] → q.

After the 1st partition, q takes its perfect position.

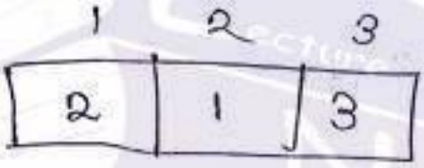


Pivot A[r] = 3

i = 0

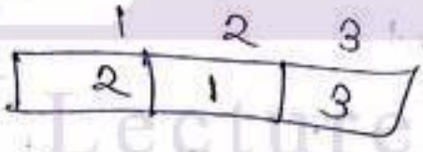
j = 1 ≤ pivot

i = 1



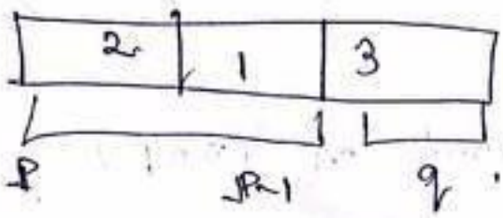
j = 2 ≤ pivot

i = 2

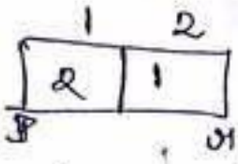


swap A[i+1] = A[3], A[j] = A[3]

q ← A[r].



(Imbalance partitioning).



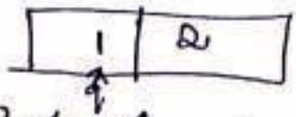
p < r.

Pivot 1

i = 0

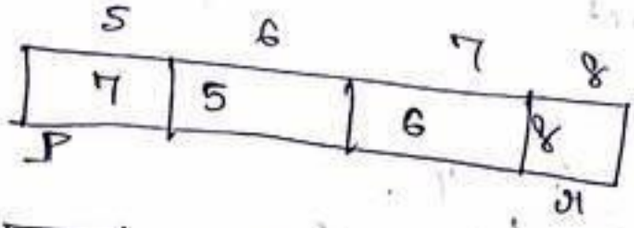
j = 1 ≤ pivot

swap A[i+1] = A[1], A[r] = A[2]



q ← 1

Sort



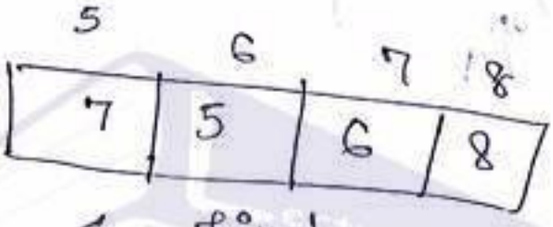
Pivot 8 p 5

i 4

j 5 ≤ pivot

i 5

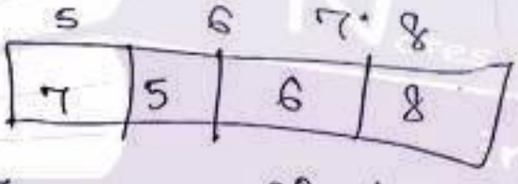
swap



j 6 ≤ pivot

i 6

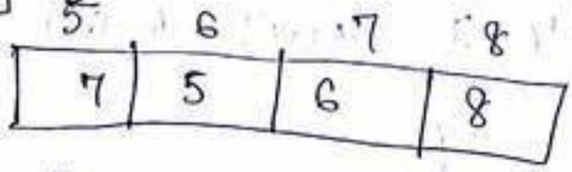
swap



j 7 ≤ pivot

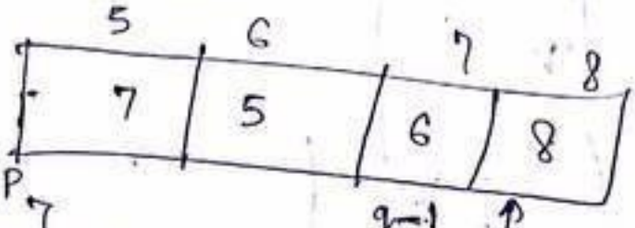
i 7

swap

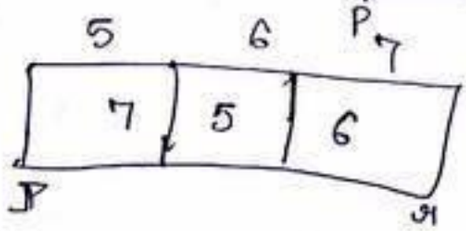


swap A[i+1] = A[8], A[r] = 8

q ← i+1 = 8



Sort



p 5

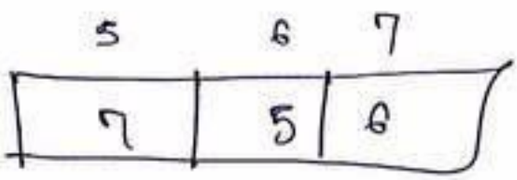
Pivot 6

i 4

j 5 ≤ pivot

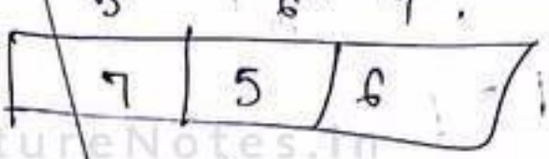
i 5

swap

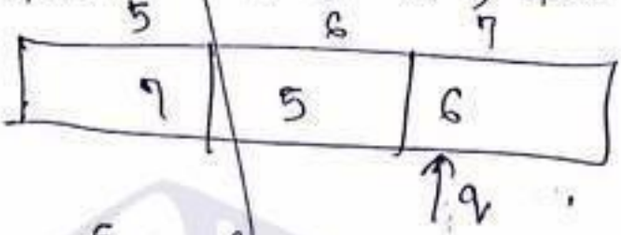


j 6 ≤ pivot

i 6



swap A[i+1] = A[7] = 6 , A[8] = A[7] = 6



Sort

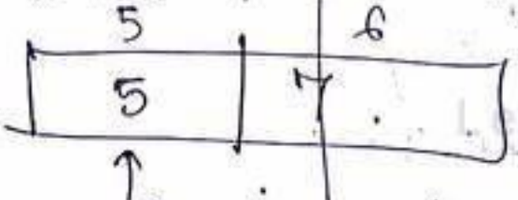
Pivot 5 P = 5

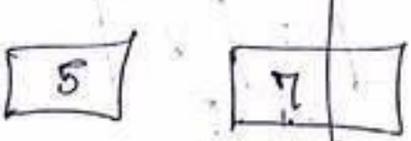
i 4

j 5 ≤ pivot

~~i 6~~

swap A[i+1] = A[5] = 7 & A[8] = A[6]

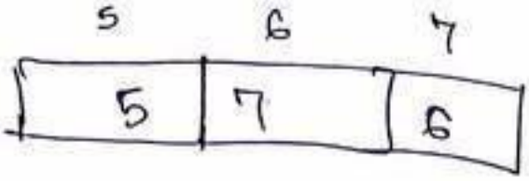




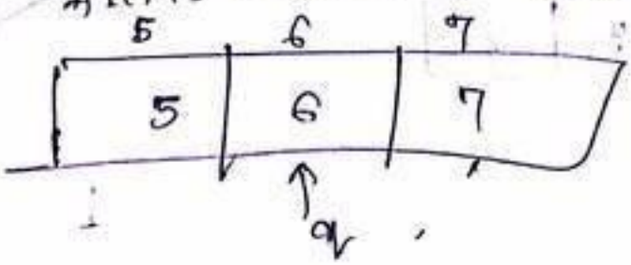
j 6 ≤ pivot

i 5

Swap



swap A[i+1] = A[6] ; A[x] = A[7]



2 ← A[6].

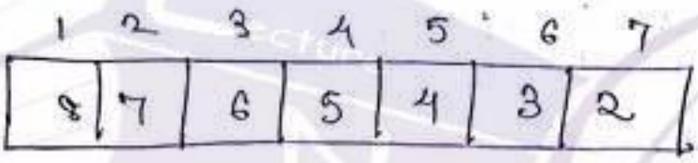
5 | 7.

18/8/2015.

Q: A | 8 | 7 | 6 | 5 | 4 | 3 | 2 |

Given an input array A. S.T state of the array after each partitioning?

A[1:7]:



p [1] r [7]

Pivot [2]

i [0]

j [1] ≠ pivot

j [2] ≠ pivot

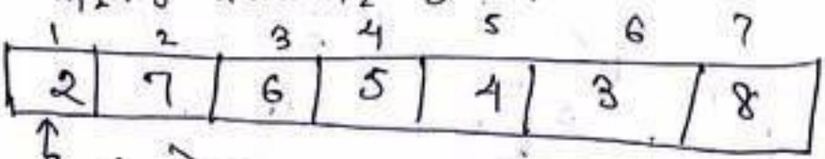
j [3] ≠ pivot

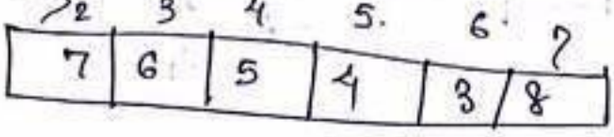
j [4] ≠ pivot

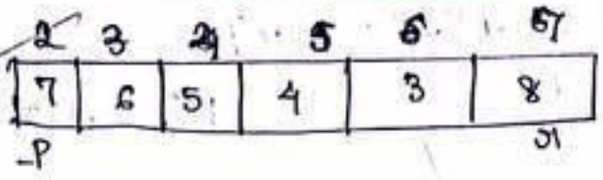
j [5] ≠ pivot

j [6] ≠ pivot

swap A[1] and A[7].







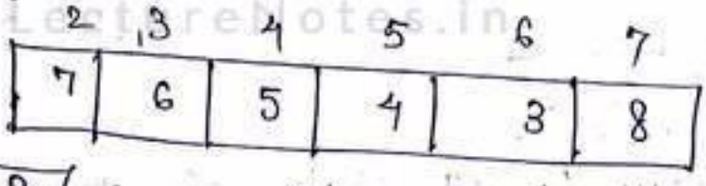
or 7

Pivot 8

i 1

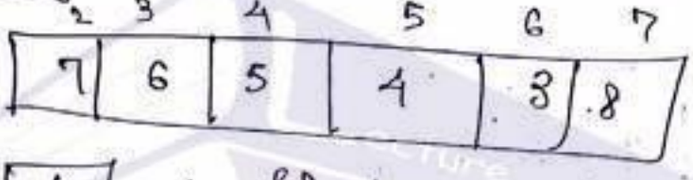
j 2 < pivot.

i 2



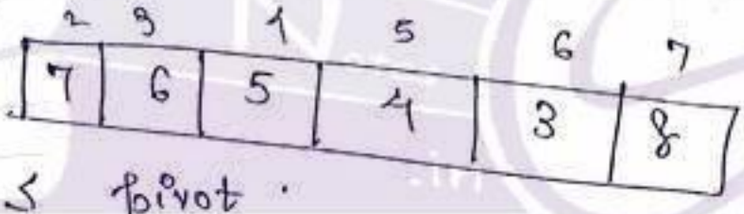
j 3 < pivot

i 3



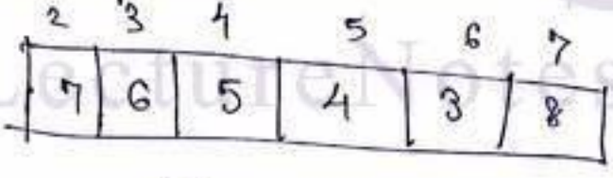
j 4 < pivot

i 4



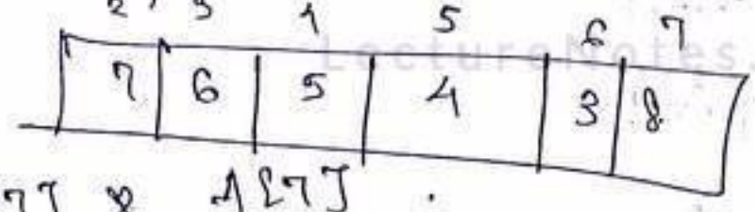
j 5 < pivot.

i 5

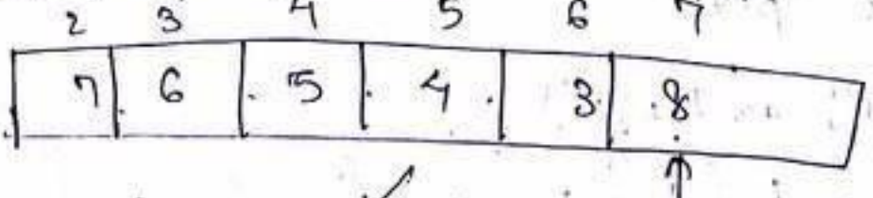


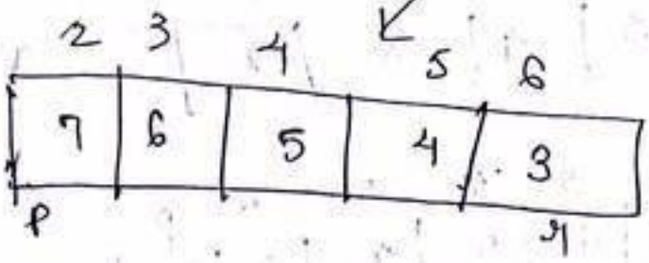
j 6 < pivot

i 6



swap A[7] & A[7].





p [2] r [6]

Pivot [3]

i [1]

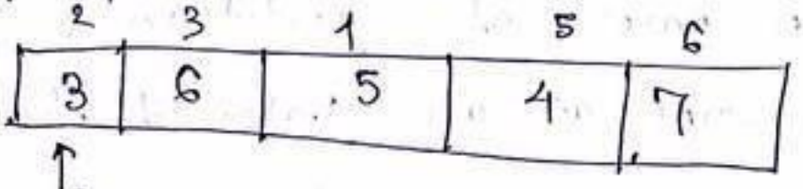
j [2] ≰ pivot

j [3] ≰ pivot

j [4] ≰ pivot

j [5] ≰ pivot

swap A[2] & A[6]



p [3] r [6]

Pivot [7]

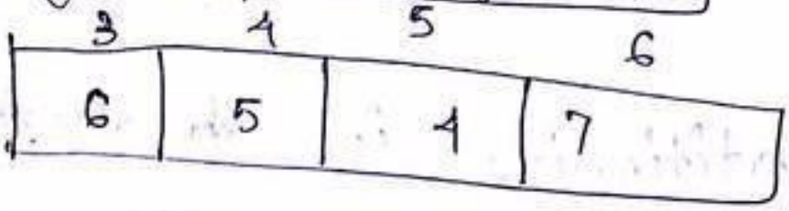
i [2] j [3] ≤ pivot

j [4] ≤ pivot

j [5] ≤ pivot

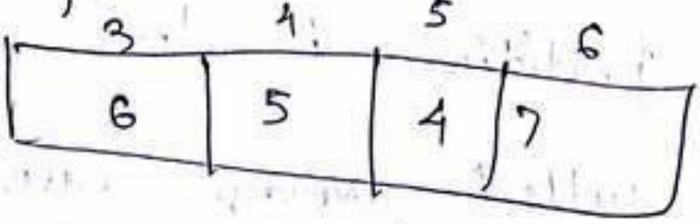
swap A[3] with A[6]

i [3]



j [4] ≤ pivot

i [4]



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

$T(n) = T(n-1) + \Theta(n)$.

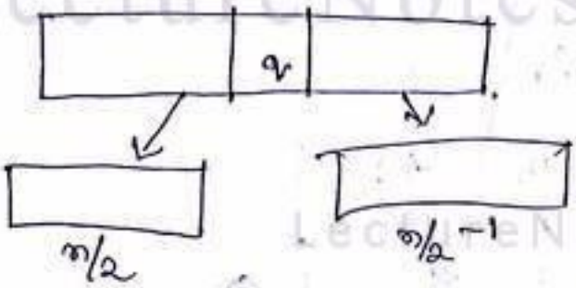
Analysis of Quick Sort.

* The time taken by the quick sort depends on the partition point of the array.

* There are three cases of partition :

(i) partition is not at all balanced i.e., rest at least of the elements remain either after q or before q where q is the partitioning point. This case is called worst case partitioning.

(ii) The partition is completely balanced.

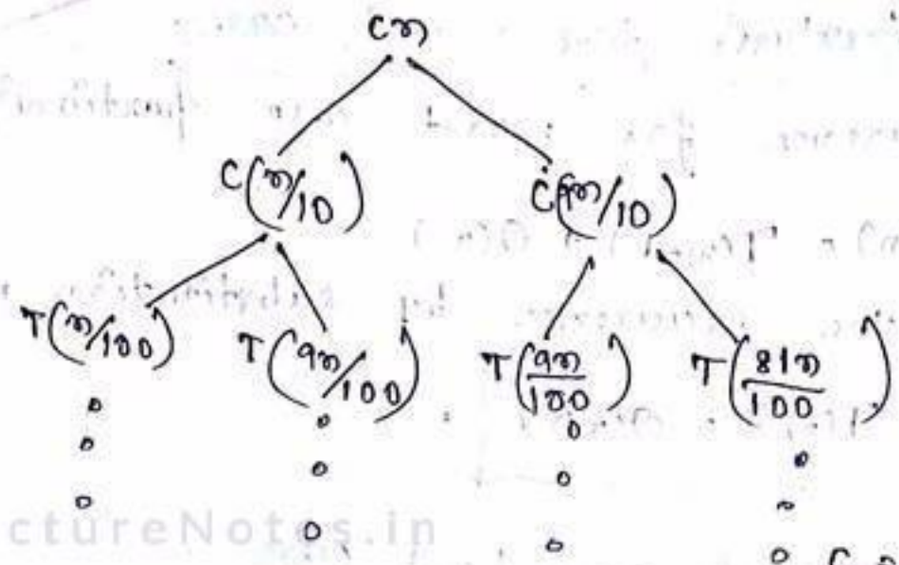


This is called best case partitioning.

(iii) The partitioning is somehow balanced.

Let the partition is done in 9:1 ratio.

This is called average case partitioning.



Let after k number of steps the value of the leaf node in the longest path becomes 1.

$$n\left(\frac{9}{10}\right)^k = 1$$

$$\Rightarrow n\left(\frac{9}{10}\right)^k = 1$$

$$\Rightarrow n = \left(\frac{10}{9}\right)^k$$

$$\Rightarrow n = k \Rightarrow k = \log_{\frac{10}{9}} n \approx \log n.$$

Total cost

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

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

Quick sort is the best sorting algorithm.

19/8/2015

Randomized Quick Sort.

* Worst case running time of quick sort = $\Theta(n^2)$ due to imbalanced partitioning.

  Best case and average case = $\Theta(n \log n)$.

* Imbalanced partitioning comes as the last element is chosen as pivot element always.

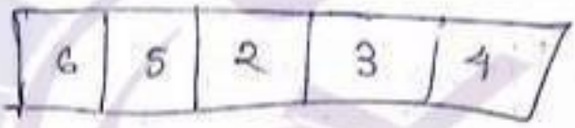
Random.

It creates a random number between p and r.

Eg:

p = 1, r = 5 i = 3

swap A[3] & A[r]



Algorithm.

Randomized Quick Sort (A, P, r)

1. If P < r
2. q ← Randomized - partition (A, P, r)
3. Randomized - Quicksort (A, P, (q-1))
4. Randomized - Quicksort (A, (q+1), r)
5. End.

Algorithm of Randomized-Partition (A, P, r)

Randomized-Partition (A, P, r)

1: i ← Random (P, r)

2: Swap (A[i], A[r])

3: q ← Partition (A, P, r)

4: return q.

Time complexity is $\Theta(n \log n)$.

Heap Sort

Heap:

A heap is a data structure represented by an array and can be viewed as a nearly complete binary tree.

* In the tree, each node corresponds to the value of the array. It is associated with two elements:
  (i) Array length given as length [A]
  (ii) heap size given as heap-size [A].

* A[1] is the root of the tree.

* For an element (i), its parent resides at $\lfloor i/2 \rfloor$ except the root.

```
Parent(i)
    returns ⌊i/2⌋ ;
```

* For node (i), its left child resides at $2i$ location.

```
left(i)
    returns 2i ;
```

* For node (i), its right child resides at $(2i+1)$ location.

```
right(i)
    returns (2i+1) ;
```

* There are two types of heaps :

  i) Max heap
  ii) Min heap.

i) **Max Heap**.

$A[\text{Parent}(i)] \geq A[i]$;

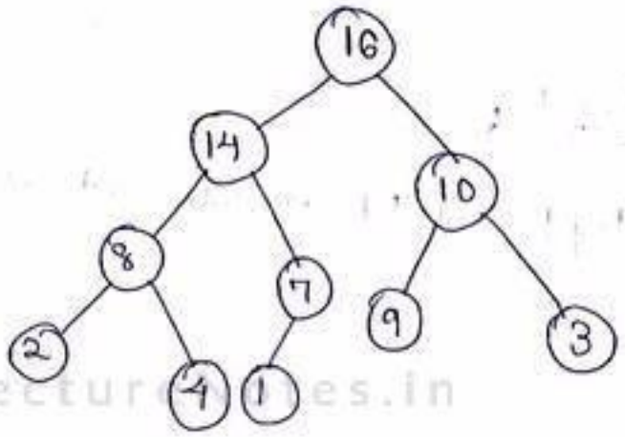
ii) **Min Heap**.

$A[\text{Parent}(i)] \leq A[i]$;

Leaf starts at $\lfloor n/2 \rfloor + 1$.

Ex:

	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1



In almost complete binary tree, if level given no. of node $= 2^l$.

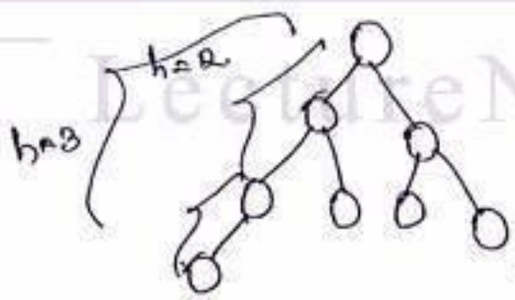
20/8/2015.

24/8/2015.

(i) Min_heap (A[Parent(i)] $\leq$ A[i]).

(ii) Max_heap (A[Parent(i)] $\geq$ A[i]).

An almost complete binary tree of height (h) is complete upto height (h-1) and the tree is filled from left to right



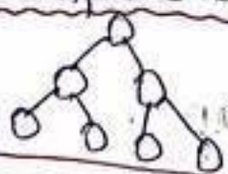
h = 3

level 0 2^h

level 1

level 2

Complete binary tree



* Minimum number of nodes $= 2^h$

(filled upto h-1 : nodes $= 2^{h-1+1} - 1$
$= 2^h - 1$

for min^m almost complete binary tree $= 2^h$
$= 2^{h-1+1}$

Total nodes $= 2^{h+1} - 1$

* Maximum number of nodes $= 2^{h+1} - 1 - 1$
$= 2^{h+1} - 2$

*- Heap sort uses three procedures for sorting an input array :

(i) Max_heapify. ③

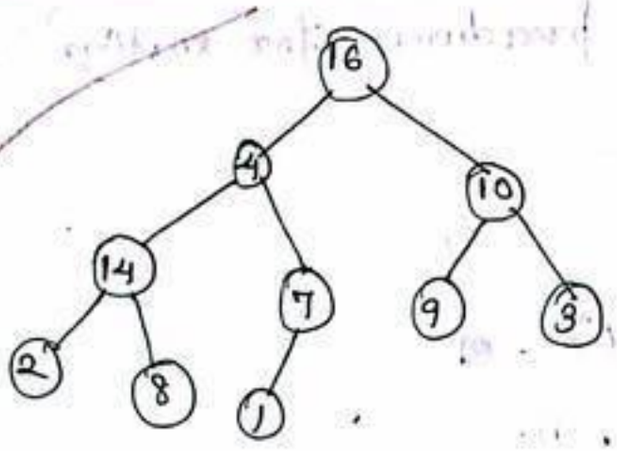
(ii) Build_Max_heap() . ②

(iii) Heap_Sort() . ①

(i) Max_heapify (A, i) : i is the index on which the heapify procedure will be applied.

1. l ← left of (i) . returns 2i
2. r ← right(i) returns (2i+1).
3. If l ≤ heapsize(A) && A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heapsize(A) && A[r] > A[largest]
7. largest ← r .
8. if largest ≠ i
9. swap (A[i], A[largest])
10. max_heapify (A, largest) .
11. End .

Ans:



16	4	10	14	7	9	3	2	8	1
1	2	3	4	5	6	7	8	9	10

Ans:-

i° [4] l [14] r [7]

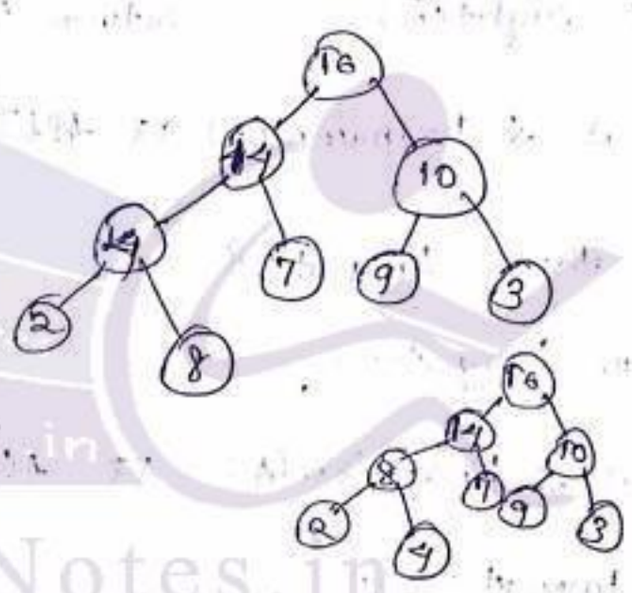
A[l] ≥ A[i]

largest = A[l] = 14

largest ≥ A[r]

largest ≠ i

swap A[i], A[largest]



i [4] l [2] r [8]

largest [4]

A[l] ≥ largest

swap A[i], A[largest]

l > heapsize

r > heapsize

* 26/8/2015.

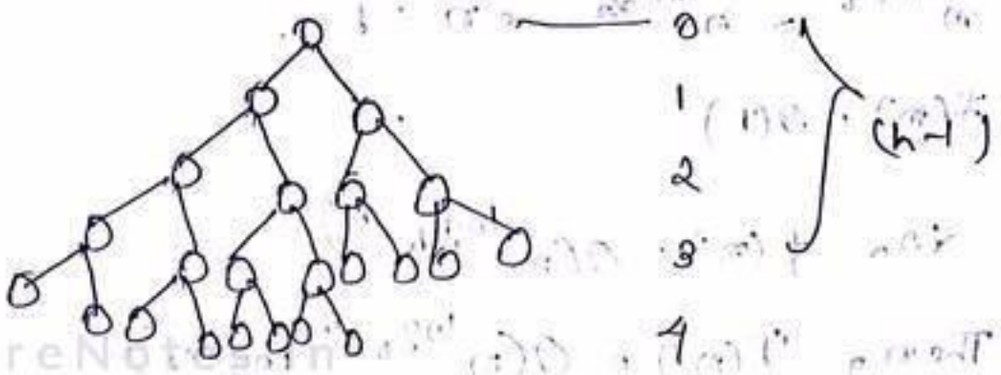
When i is leaf node, the left and right child exceeds the array size.

Time Complexity of Max_heap.

* Finding largest $\Theta(1)$ and exchange time is some constant c_2.

In worst case, the recursion will go upto log of height

Worst case: $O(h) = \Theta(\log n)$ height, $h = \log n$.



Total no. of nodes upto height (h-1) is:

$2^{h-1+1} - 1 = 2^h$ (ignoring one)

No. of nodes at level $2^l = 2^{h-1}$

To find nodes at level (h) $(\because l = h-1)$

$\Rightarrow 2^l/2 = 2^l \cdot 2^{-1} = 2^{l-1} = 2^{h-2}$

Total nodes of last level when tree is half full at last bcoz level

$= 2^{h-2} \times 2 = 2^{h-1}$

Total nodes of the tree is

$2^h + 2^{h-1}$

$\Rightarrow 2^h + 2^{h-1} = n$ $(\because$ total no. of nodes $= n)$

$\Rightarrow 2^{h-1}(2+1) = n$

$\Rightarrow 2^{h-1} = n/3$

$\Rightarrow \boxed{2^h = 2n/3}$

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

Here, $a = 1$, $b = 3/2$.

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

$f(n) = \Theta(1)$

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

Then, $T(n) = \Theta(n^{\log_b a} \log n)$

$\Rightarrow T(n) = \Theta(1 \cdot \log n)$

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

27/8/2015

ii) Build Max_heap (A).

Given an array (A). Construct a max_heap from the input array (A)

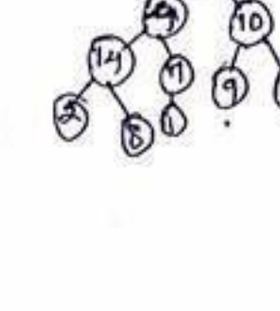
(It constructs or converts the input array (A) to a max_heap (A)).

$(n/2 + 1)$ 1st leaf node in area.
non-leaf → $n/2$.

Let A

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

$i = 10 (n/2) \rightarrow 16$.



$i = 4$

$i = 3$

$i = 2$

$i = 1$

16 14 10 8 7 9 3 2 4 1

1. Heapsize(A) ← length(A)
2. For i = n/2 down to 1
3. Max_heapify(A, i).

Property of heap:

An n element heap has height $\lfloor \log n \rfloor$ and atmost $\lceil n/2^{h+1} \rceil$ nodes at any height.

Time Complexity of build max_heap:

$$T(n) = \sum_{h=0}^{\log n} n/2^{h+1} \; O(\log n)$$

$$= \sum_{h=0}^{\log n} n/2^{h+1} \times h$$

$$= 1/2\, n \left(\sum_{h=0}^{\log n} h/2^h \right)$$

$$= n/2 \times 2 \qquad \left(\because \sum_{h=0}^{\infty} h/2^h = 2 \right)$$

$$= n$$

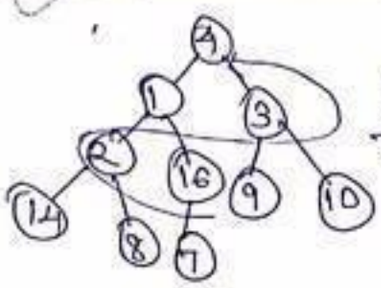
$$\Rightarrow T(n) = O(n)$$

31/8/2015

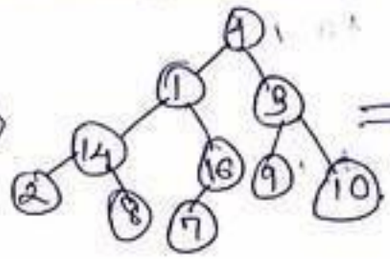
iii) Heapsort(A):

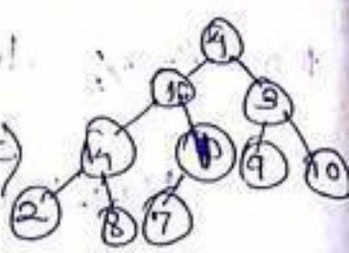
1. Build_max_heap(A).
2. for (i = length[A] down to 2)
3. exchanges A[1] ↔ A[i].
4. heapsize[A] ← heapsize[A] − 1
5. Max_heapify(A, 1).

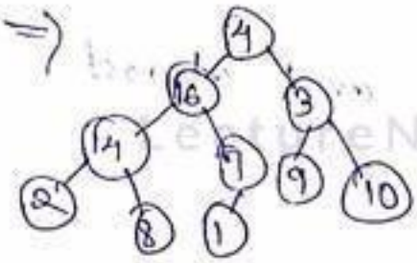
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

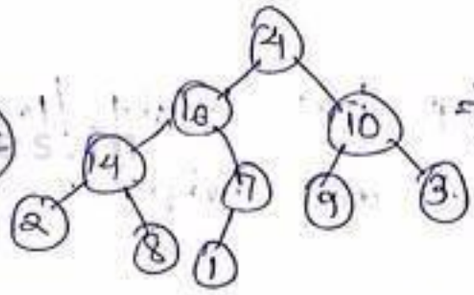


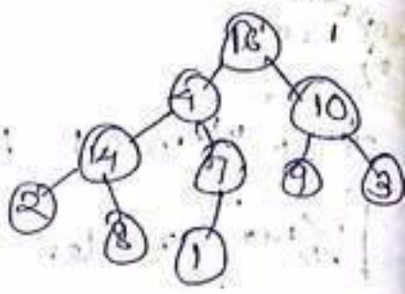
Build max-heap

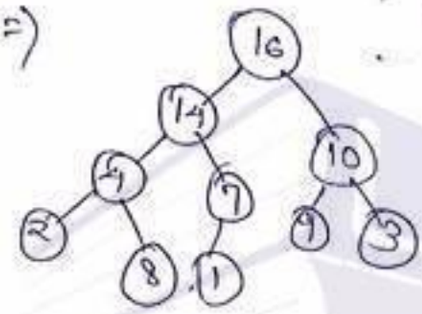


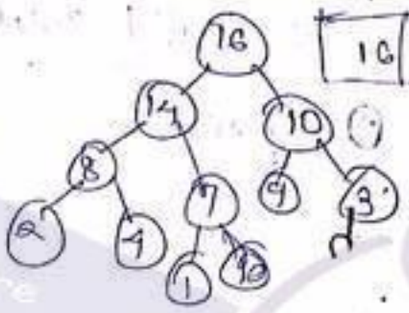












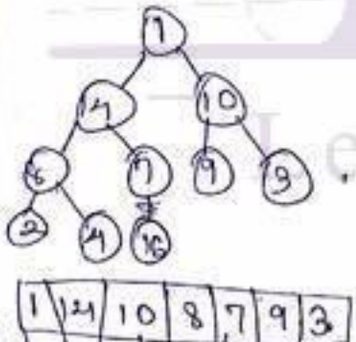
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

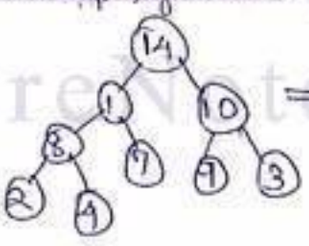
for (i = length[A] to 2)

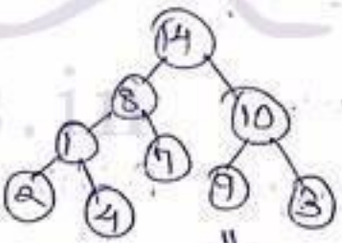
exchange A[1], A[i] i = 10

length[A] = 10.

Max-heapify (A, 1)

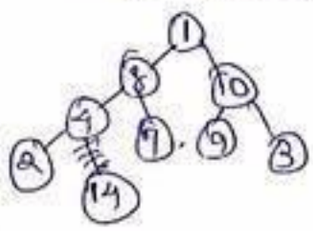




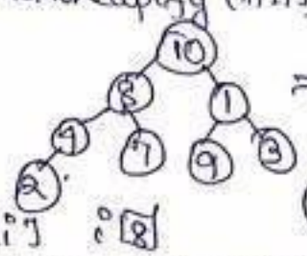


1	14	10	8	7	9	3	2	4	16

exchange A[1], A[i] i = 9



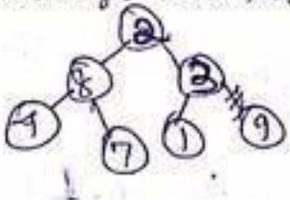
Max-heapify (A, 1)



exchange A[1], A[i] i = 8

Max-heapify (A, 1)

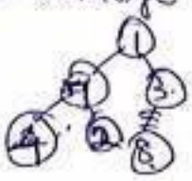
exchange A[1], A[i]



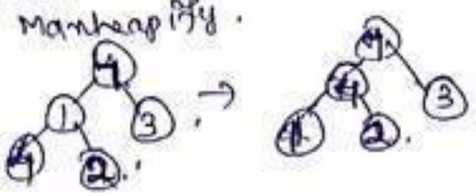
9 8 2 4 7 1 3

max-heapify (A, 2)

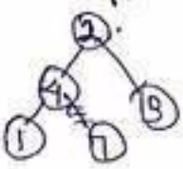
exchange A[1], A[3]



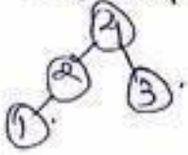
max-heapify.



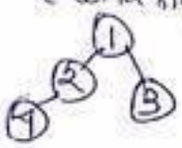
exchange A[1], A[2]



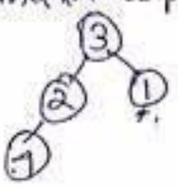
max-heapify (A, 1)



exchange



max-heap.



Time Complexity of heap-sort.

line 2 : $O(n)$.

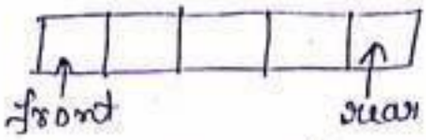
line 3-5 : $(n-1)\log n = n\log n - \log n = O(n\log n)$.

Total time = $O(n) + O(n\log n) = O(n\log n)$ / $O(n^2 + n) = O(n^2)$.

11/9/2015.

Priority Queue → used in job scheduler.

Queue is a data structure,

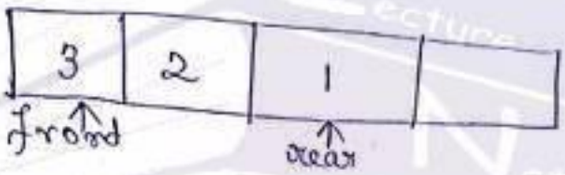


Insertion takes place at the rear end.

Deletion takes place from the front end.

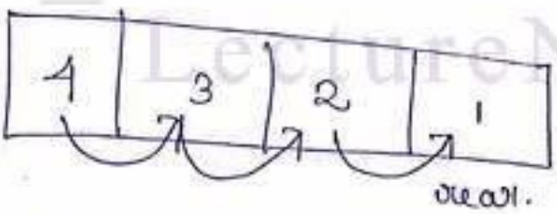
*- In priority queue, each element is associated with a priority or key which decides the priority of that element.

*- More prior element is deleted first.

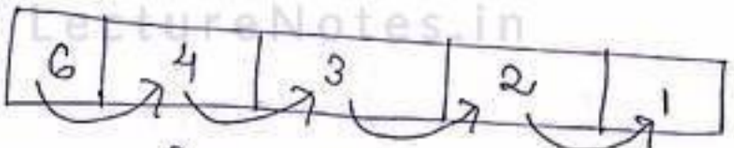


Job 1 — 3
Job 2 — 2,
Job 3 — 1

Job 4 — 4. (higher priority) (as 4 should be at front, increment rear).



Job 5 — 6.

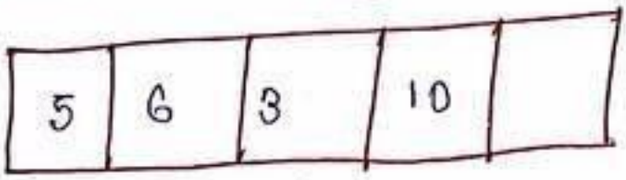


Time taken to insert into the queue is $O(n)$ → for worst case as $(n-1)$ comparison and shifting are required.

Time taken for deletion : $O(1) = O(n)$.

Best case : $\theta(1) = O(n)$

Implementing Priority Queue Using Array.



Job1 — 5
Job2 — 6
Job3 — 3
Job4 — 10

Insertion takes O(1) constant time.

Deletion.

*- First search the element with highest priority. Then, take out that. After that rearrange.

Time taken is O(n)

*- If we are taking linear search, then the time taken for insertion and deletion is O(n).

* An efficient algorithm to implement priority queue are :

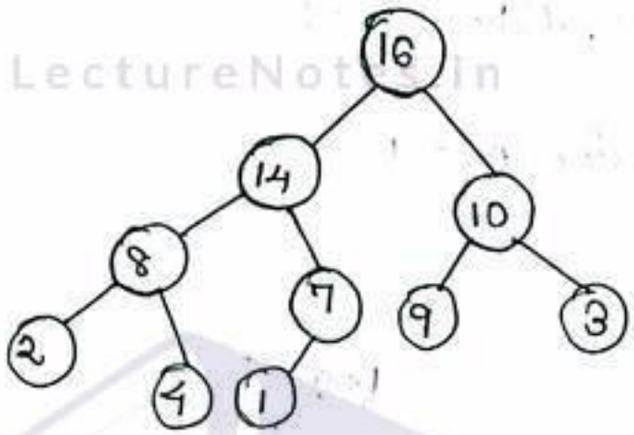
ii) Heap (Min / Max-heap).

The insertion or deletion time

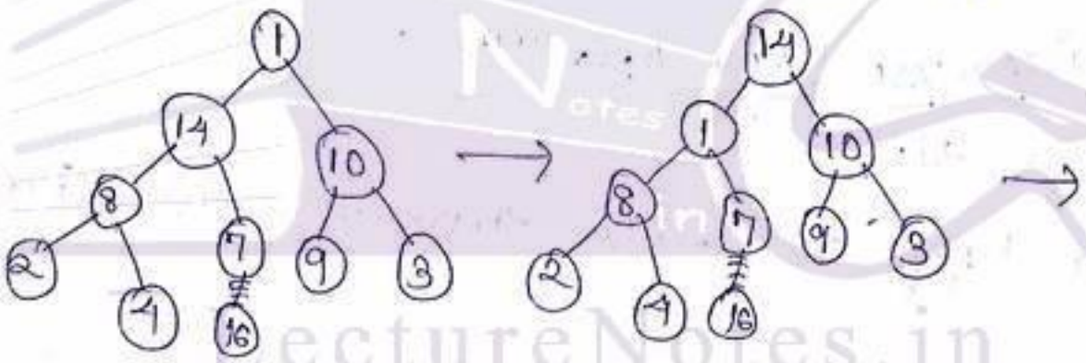
$T(n) = O(\log n)$.

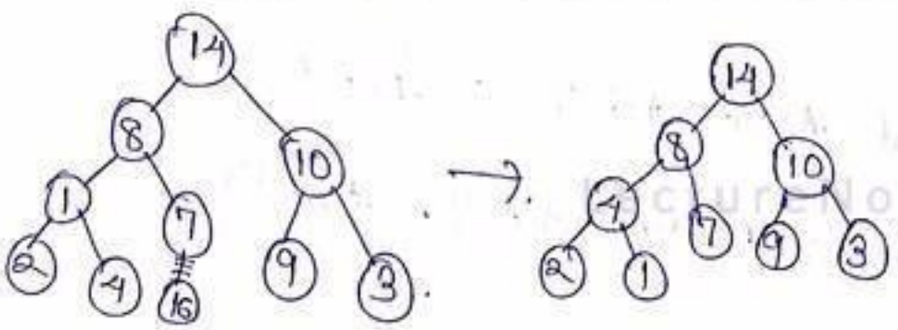
Implementing Priority Queue using Heap (max-heap)

* It uses three functions :
  (i) heap_extract_max()
  (ii) heap_increase_key()
  (iii) max_heap_insert()



(i) To find maximum element, apply heapsort.





Time complexity for heap_extract_max() is O(log n) as swaping only takes constant time, then rearranging will start from root to the leaf which is the height of the tree = O(log n).

3/9/2015

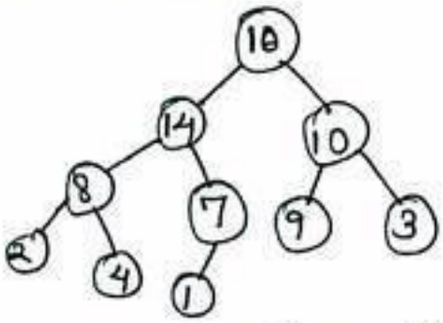
Heap_Extract_Max(A) → A is a max-heap

1. if heapsize(A) < 1
2. then Error "Heap underflow".
3. Max ← A[1].
4. Heapsize(A) swap(A[1], A[heapsize]) → last index of heap
5. Heapsize(A) ← Heapsize(A) − 1
6. Max-heapify(A, 1).
7. Return max.

log n

Heap-increase_key(A, i, key).

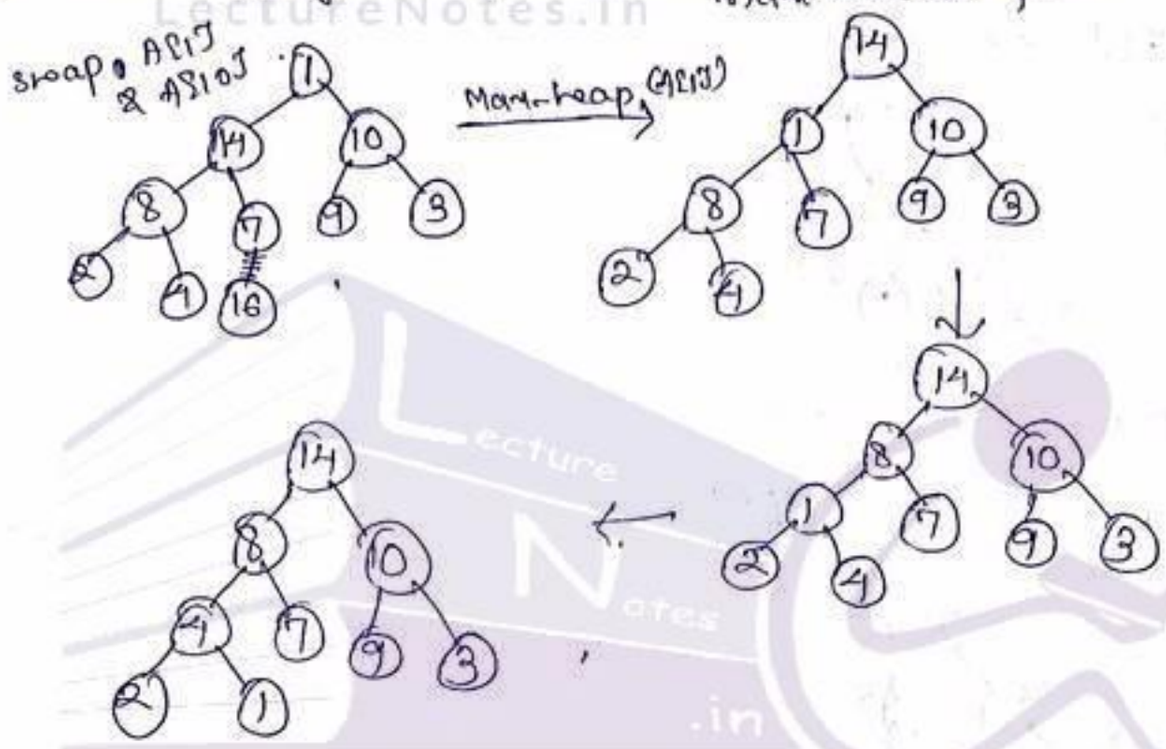
1. if key < A[i].
2. error "key is smaller than the current key".
3. A[i] ← key
4. While i > 1 and A[Parent(i)] < A[i].
5. do exchange (A[Parent(i)], A[i]).
6. i ← Parent(i).



(Max-heap).

The values of the nodes represents the key value of a job.

* 1st apply heap-extract-max. (extracts the max value from the heap).



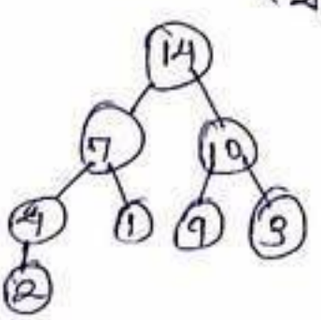
* Apply heap-increase-key. (It increase the key value of a node).

The input to procedure-2 is

(i) A, which is a max-heap.

(ii) i, index of the node A whose key-value we want to increase.

(iii) key, new key value which is of i, ie

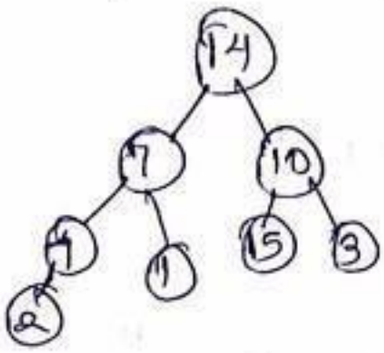


i = 6

key = 15

A[6] < key .

A[6] ← key (15) .



i [6] > 1

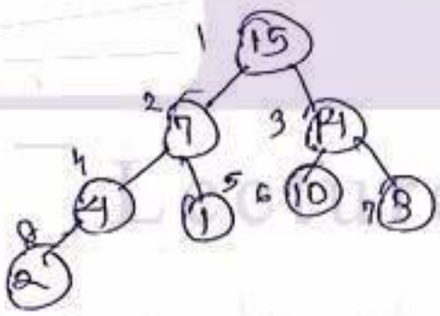
A[parent[i]] < A[6]

exchange

i ← parent(i) i [3] .

i [3] > 1 parent < i [1]

swap



i ← parent(i) i [1] .

3. Max_heap_insert(A, key) .

increase heap size

assign $-\infty$ to new node, with a priority.

Then, apply heap_increase_key.

1. Heapsize(A) ← Heapsize(A)+1

2. A[Heapsize] ← −∞

3. Heap-Increase-Key (A, heapsize, key).

7/9/2015.

Time Complexity of Priority Queue.

Insertion = O(log n)
Deletion = O(log n)
Increase Key = O(log n)

} using array or queue. O(n).

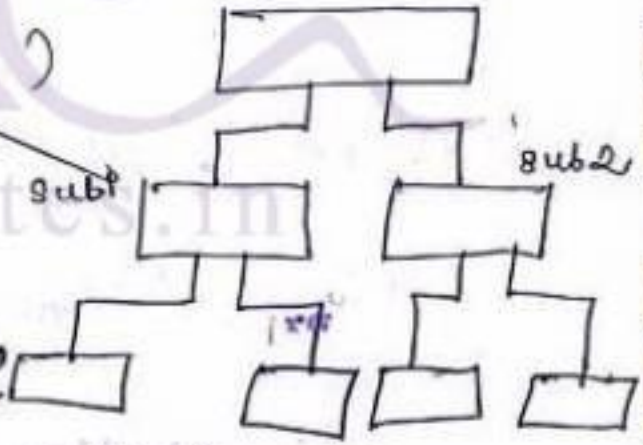
using Heap → O(log n)

Dynamming Programming

*- Dynamming programming approach solves the problems by combining the solution of sub-problems.

*- In divide and conquer, approach, the sub-problems are independent/ disjoint and they solved recurrively.

*- In dynamic programming approach, the sub-problems are over-lapping sub-problems, i.e., sub-problems share sub-sub problem.



sub1, sub2 are disjoint.

* - The dynamic programming is mainly used for optimization problems.

Optimization problems have many solutions.

* - Each solution has some value.

* - Objective is to choose a solution with maximum or minimum value.

* - Four major steps of Dynamic programming approach :

(i) characterize the structure of an optimal solution.

(ii) recursively define the value of an optimal solution. (Recursive / Recurrence).

(iii) Compute the value of an optimal solution (typically by bottom-up approach) (using table computation is done).

(iv) construct an optimal solution from computed information. (back-track method as we are using bottom-up approach).

Matrix - Chain Multiplication.

$[C]_{p \times p} \quad [A]_{n \times m} \quad [B]_{m \times p}$.

Let $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & & a_{nm} \end{bmatrix}$

$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & & & \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix}$

$C = \begin{bmatrix} c_{11} & & c_{1p} \\ \vdots & & \vdots \\ c_{n1} & & c_{np} \end{bmatrix}$

$c_{11} = a_{11} b_{11} + a_{12} b_{21} + \cdots + a_{1m} b_{m1}$

$c_{np} = a_{n1} b_{1p} + a_{n2} b_{2p} + \cdots + a_{nm} b_{mp}$

Form

Matrix_Multiply (A, B).

1. Define C.
2. for (i = 1 to n). $(n \times p \times m)$.
3. for j = 1 to p. (p no. of m multiplications $(p \times m)$.
4. C[i][j] = 0.

5: for k=1 to n. (n multiplications & additions).

6: C[i][j] = C[i][j] + a[i][k] * b[k][j].

7: end of for

8: end of for,

9: end of for.

10: end. $n^3(n-1)$.

Time Complexity for matrix multiplication.

Time complexity of matrix multiplication.

$= O(n \times p \times m)$, $n, p, m \rightarrow$ scalars.

$= O(n^3)$ (if all are square matrix $n \times n$).

For chain matrix multiplication

$= O(n^3(n-1))$

$= O(n^4)$.

8/9/2015

Matrix-Chain Multiplication.

Given a chain of n matrices $\langle A_1, A_2, \cdots, A_n \rangle$.

where for $i = 1, 2, \cdots, n$ matrix (A_i) has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1, A_2, \cdots, A_n$ in a way that minimizes the number of scalar multiplication.

$(AB)_{p \times r} = (A)_{p \times q} \; (B)_{q \times r}$.

$T(n) = O(pqr)$.

*- Given a sequence or chain of matrices $A_1, A_2, \cdots, A_n$.

Objective is to compute their product $A_1 A_2 \cdots A_n$.

To compute the chain of products we need to parenthesize the matrices.

Fully Parenthesized.

A product of matrices is fully parenthesized if (i) it is a single matrix.

(ii) the product of two fully parenthesized matrix products surrounded by parentheses.

Let $((A_1 A_2)(A_3 A_4))$ ⟶ fully parenthesized.

or, $(A_1 (A_2 A_3))$.

or, $((A_1 A_2) A_3)$.

The fully parenthesized matrix product gives the order in which the matrices should be multiplied.

Ex: $A_1 A_2 A_3$.

Fully parenthesized product: $(A_1 (A_2 A_3))$

or, $((A_1 A_2) A_3)$.

*- Since the matrix multiplication is associative. So, all parenthesis gives the same result.

Ex: $(A_1)_{p \times q}$ $(A_2)_{q \times r}$ $(A_3)_{r \times s}$.

$A_1 A_2 A_3 = (A_1 A_2 A_3)_{p \times s}$.

$= ((A_1 A_2)_{p \times r} A_3)_{p \times s}$ $= (A_1 (A_2 A_3))_{p \times s}$.

Ex: Given a sequence $\langle A_1 A_2 A_3 A_4 \rangle$ of matrices. The product $A_1 A_2 A_3 A_4$ can be parenthesized in how many ways?

A:-

$(((A_1 A_2) A_3) A_4)$, $((A_1 (A_2 A_3)) A_4)$,

$(A_1((A_2 A_3) A_4))$, $(A_1(A_2(A_3 A_4)))$,

$((A_1 A_2)(A_3 A_4))$.

Need of Parenthesizing :

$(A_1)_{10\times100}$ $(A_2)_{100\times5}$ $(A_3)_{5\times50}$ $((A_1 A_2) A_3)$

$(A_1(A_2 A_3))$.

Result : $(A_1 A_2 A_3)_{10\times50}$.

$((A_1 A_2) A_3)$

How many scalar multiplications we need?

← Two times the matrix multiplication routine will be called to perform the multiplication.

A:-

To compute $A_1 A_2$,

number of scalar multiplications needs $= 10\times100\times5$

$= 5000$.

To compute $((A_1 A_2) A_3)$

$= 10\times5\times50$ $(\because (A_1 A_2)_{10\times5})$.

$= 2500$.

Total no. of scalar multiplications

$= 5000 + 2500$

$= 7500$.

To compute $(A_1 (A_2 A_3))$

Total computations needed to compute $(A_2 A_3)$

$= 100 \times 5 \times 50 = 25,000$.

$(A_1 (A_2 A_3)) = 10 \times 100 \times 50 = 50,000$.

Total no. of scalar multiplications $= 75,000$.

Here, the 1st one is 10 times faster than 2nd one.

9/9/2015 -

*- The parenthesization of the matrices which gives lowest number of scalar multiplication is called optimal parenthesization.

Objective of Matrix Chain Mutiplication Problem -

To find out the optimal parenthesization.

Matrix Chain Multiplication Problem.

Given a chain of n matrices

$\langle A_1 A_2 \cdots A_n \rangle$

where for $i = 1, 2, \cdots, n$ matrix (A_i) has dimension $P_{i-1} \times P_i$.

Fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplication.

$P_0 \times P_1$
$P_1 \times P_2$
$\vdots$
$P_{n-1} \times P_n$ $\longrightarrow$ Resulting matrix $((A_1 A_2) A_3)_{P_0 \times P_n}$.

*- The matrix chain multiplication can be solved by dynamic programming approach.

(i) Characterize the structure of an optimal sol^n.

$A_i A_{i+1} \cdots A_j$ $i < j$. $\longrightarrow$ $A_{i \cdots j}$.

$((A_i \cdots A_{i+1})(A_{i+2} \cdots A_j))$

Splitting occurs at A_k & A_{k+1}.

$((A_i \cdots A_k)(A_{k+1} \cdots A_j))$

$= (((A_{i \cdots k})_{P_{i-1} \times P_k} (A_{k+1 \cdots j})_{P_k \times P_j}))$ $\longrightarrow$ $O(P_{i-1} \times P_k \times P_j)$.

10/9/2015.

Note:

Exhaustive searching of optimal parenthesization from all possible parenthesization doesn't yield an efficient algorithm.

So, a dynamic programming approach

will be used to find out the optimal parenthesization

Let $A_{i...j}$ where $i \leq j$ is used to denote the matrix results from evaluating $A_i, A_{i+1} \cdots A_j$.

* When $i < j$, then any parenthesization of the product $A_i, A_{i+1} \cdots, A_j$ must split the product between A_k and A_{k+1} for some integer (k) in the range $i \leq k < j$.

* The cost of computing $A_{i...j}$ = cost of computing $A_{i...k}$ + cost of computing $A_{k+1...j}$ + cost of computing their product.

(e) Characteristics of Optimal Solution.

* Let an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}.

Then, the parenthesization of the prefix subchain $A_i \cdots A_k$ within this optimal parenthesization of $A_i, A_{i+1} \cdots A_j$ must be

$m[i,i] = 0$ (single matrix so cost = 0)
(no multiplication).

for $i = 1 \ldots n$.

* If $i = j$, the problem is trivial. i.e., the chain consists of just one matrix. So, no scalar multiplication needed to compute the product.

* Assuming the optimal parenthesization of the chain $A_i, A_{i+1} \ldots A_j$ for $i < j$ splits between the parenthesis A_k and A_{k+1}

Then, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j$.

$A_{i \ldots k}$: $A_{p_{i-1} \times p_i}$ $A_{p_i \times p_{i+1}} \ldots A_{p_{k-1} \times p_k}$

$(A_{k+1} \ldots A_j)_{p_k \times p_{k+1}} \ldots A_{p_{j-1} \times p_j}$

$(A_{i \ldots k}) \cdot (A_{k+1 \ldots j})_{p_k \times p_j}$.

The recurrence to compute optimal parenthesization for matrix chain multiplication is

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

Value of k can be $i, i+1, \ldots, j-1$.

Depending on k, the no. of optimal sol^n depends.

Objective of matrix chain multiplication is to find optimal sol^n with parenthesization. 14/9/2015.

Find Optimal Sol^n.

Ex: Let $n=6$.

The chain of the matrices is

$A_1, A_2, A_3, A_4, A_5, A_6$.

Our objective is to find the parenthesization of the matrices so, that the number of scalar multiplication will be minimized

$A_1 : 30 \times 35$ ($P_0 \times P_1$)
$A_2 : 35 \times 15$
$A_3 : 15 \times 5$
$A_4 : 5 \times 10$
$A_5 : 10 \times 20$
$A_6 : 20 \times 25$

	P_0	P_1	P_2	P_3	P_4	P_5	P_6
P_n	30	35	15	5	10	20	25

The optimal value of the chain

$A_1 A_2 \cdots A_6 = A_{1 \cdots 6}$

which gives the minimum scalar multiplication to compute the product $A_{1 \cdots 6}$ is $m[1,6]$.

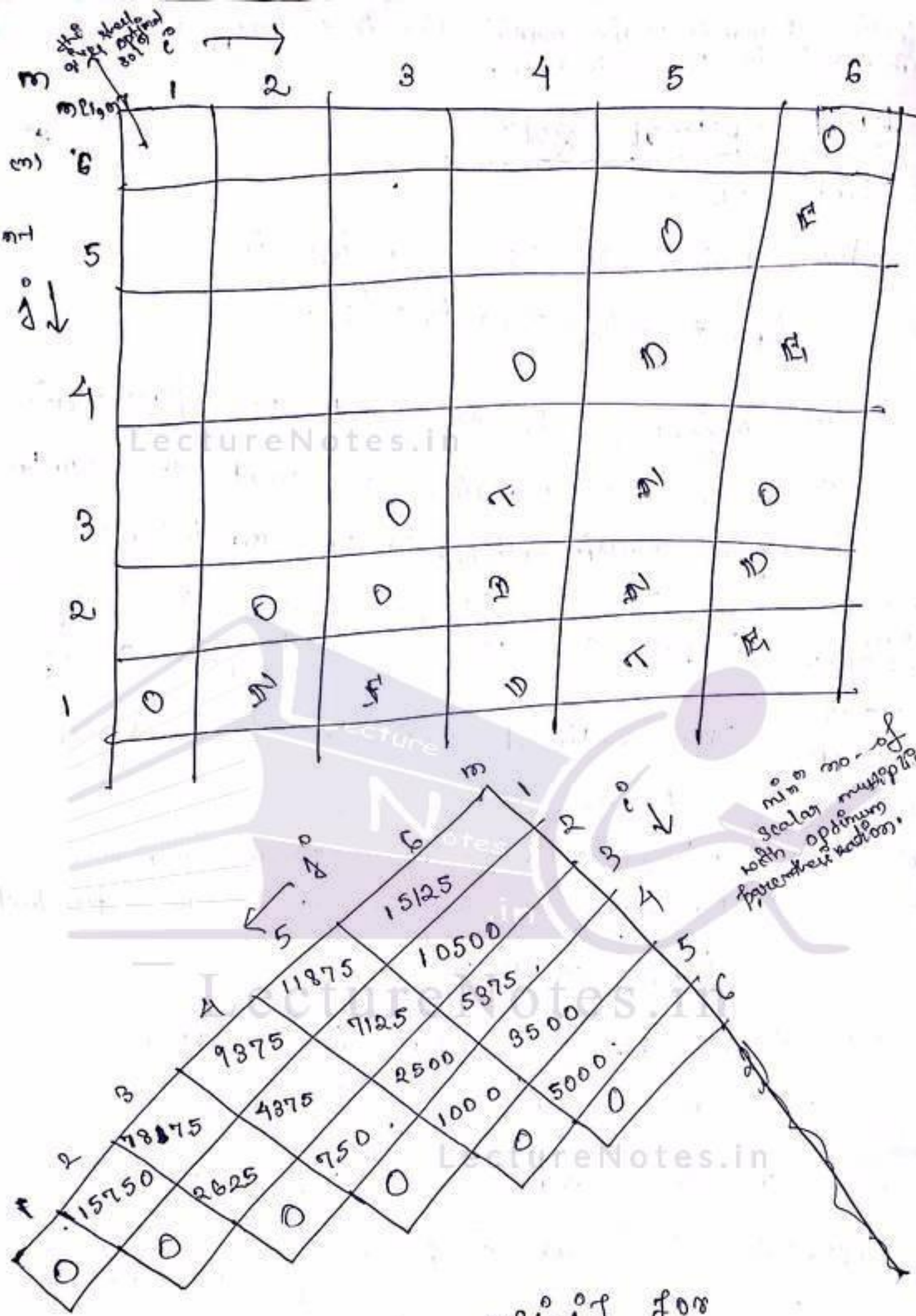
If there are n number of matrices

$m[1,n]$.

$m[5,2]$ is invalid as $i > j$.

$m[5,5] = 0$, as $i = j$, (as no scalar multiplications).

* A dynamic programming table of size $(n \times n)$ is computed to find out $m[1,n]$.



min no. of scalar multiplications with optimum parenthesization.

* Compute the value $m[i,j]$ for

$i = 1 \text{ to } 6$

$j = 1 \text{ to } 6$.

with bottom-up approach.

$m[1,1], m[2,2], m[3,3], m[4,4], m[5,5], m[6,6] = 0$

* - Compute $m[1,2]$ means $A_1 \times A_2$.

$i = 1, j = 2$.

Using recurrence

$k = 1$.

The parenthesis can be split at point $k = 1$.

$m[1,2] = m[1,1] + m[2,2] + P_0 P_1 P_2$

$= 0 + 0 + (30)(35)(15)$

$= 15750$

$35 \times 15 \times 2 = 1050$

$35 \times 15 \times 2 = 1050$

$m[2,3]$

$i = 2, j = 3$

$k = 2$

$m[2,3] = m[2,2] + m[3,3] + P_1 P_2 P_3$

$= 0 + 0 + (35)(15)(5)$

$= 2625$

$35 \times 15 \times 2 = 1050$

$35 \times 5 \times 2 = 350$

$m[3,4]$

$i = 3, j = 4$

$k = 3$

$m[3,4] = m[3,3] + m[4,4] + P_2 P_3 P_4$

$= 750$

$m[4,5]$

$i = 4, j = 5$

$k = 4$

$m[4,5] = m[4,4] + m[5,5] + P_3 P_4 P_5$

$= 5 \times 10 \times 20 = 1000$.

$m[5,6]$

$i = 5, \; j = 6$

$\boxed{k = 5}$.

$m[5,6] = m[5,5] + m[6,6] + p_4 p_5 p_6$

$= 10 \times 20 \times 25$

$= 5000$.

15/9/2015

$m[1,3]$

$i = 1, \; j = 3$.

$\boxed{k = 1, 2}$

$m[1,3] = \min(m[1,3]_{k=1}, m[1,3]_{k=2})$

$m[1,3]$ when $k = 1$

$= m[1,1] + m[2,3] + p_0 p_1 p_3$

$= 0 + 2625 + (30)(35)(5)$

$= 2625 + 7875$

$m[1,3]$ when $k = 2$

$= m[1,2] + m[3,3] + p_0 p_2 p_3$

$= 15750 + 30 \times 15 \times 5$

$= 18000$

As $m[1,3]$ when $\boxed{k = 1}$ gives optimal no. of scalar multiplication, so we will consider this one

$m[2,4]$

$i = 2 \quad j = 4$

$k = 2, 3$.

$m[2,4] = \min_{k=2,3} (m[2,4], m[2,4])$.

$m[2,4]$ when $k=2$.

$= m[2,2] + m[3,4] + p_1 p_2 p_4$

$= 0 + 750 + 35 \times 15 \times 10$

$= 6000$.

$m[2,4]$ when $\boxed{k=3}$

$= m[2,3] + m[4,4] + p_1 p_3 p_4$

$= 2625 + 0 + (35)(5)(10)$.

$= 4375$.

$m[3,5]$ =

$i = 3, \quad j = 5 \qquad m[3,5] = \min_{k=3,4} (m[3,5], m[3,5])$

$k = 3, 4$.

$m[3,5]$ when $\boxed{k=3}$

$= m[3,3] + m[4,5] + p_2 p_3 p_5$

$= 0 + 1000 + (15)(5)(20)$

$= 2500$.

$m[3,5]$ when $k=4$.

$= m[3,4] + m[5,5] + p_2 p_4 p_5$

$= 750 + (15)(10)(20)$

$= 3750$.

$m[4,6]$

$i = 4 \quad j = 6$

$k = 4, 5$.

$$m[4,6] = \min_{k=4,5} (m[4,6], m[4,6])$$

$m[4,6]$ when $k = 4$

$= m[4,4] + m[5,6] + P_3 P_4 P_6$

$= 5000 + (5)(10)(25)$

$= 6250$.

$m[4,6]$ when $\boxed{k = 5}$.

$= m[4,5] + m[6,6] + P_3 P_5 P_6$

$= 1000 + (5)(20)(25)$

$= 3500$.

$m[1,4]$

$i = 1, \quad j = 4$

$k = 1, 2, 3$.

$$m[1,4] = \min_{k=1,2,3} (m[1,4], m[1,4], m[1,4])$$

$m[1,4]$ when $k = 1$

$= m[1,1] + m[2,4] + P_0 P_1 P_4$

$= m[1,1] + m[2,4] + P_0 P_1 P_4$

$= 0 + 4375 + (30)(35)(10)$

$= 14875$.

$m[1,4]$ when $\boxed{k=3}$

$= m[1,3] + m[4,4] + p_0 p_3 p_4$

$= 7875 + (30)(5)(10)$

$= 9375$

```
  1500
  7875
  9375
```

when $k=2$

$m[1,4] = m[1,2] + m[3,4] + p_0 p_2 p_4$

$= 15750 + 750 + (30)(15)(10)$

$= 21,000$

```
   151
   13
  1500
   750
   15
  15250
  15750
  21000
```

$m[2,5]$

$i=2 \quad j=5$

$k = 2, 3, 4$

$m[2,5] = \min(m[2,5], m[2,5], m[2,5])$

$k=2 \quad k=3 \quad k=4$

$m[2,5]$ when $k=2$

$= m[2,2] + m[3,5] + p_1 p_2 p_5$

$= 2500 + (35)(15)(20)$

$= 13,000$

```
   352
  1500
   175
   35
  5250
 10500
  2500
 13000
```

$m[2,5]$ when $\boxed{k=3}$.

$= m[2,3] + m[4,5] + p_1 p_3 p_5$

$= 2625 + 1000 + (35)(5)(20)$

$= 7125$

```
   352
   5
  1751
   21
  3500
  1000
  2625
  7125
```

$m[2,5]$ when $k=4$

$= m[2,4] + m[5,5] + p_1 p_4 p_5$

$= 4375 + (35)(10)(20)$

$= 11375$

```
   351
  7000
  4375
 11375
```

$m[3,6]$

$i = 3, \ldots, j = 6$

$k = 3, 4, 5$.

$m[3,6] = \min(m[3,6]_{k=3},\ m[3,6]_{k=4} + m[3,6]_{k=5})$.

$m[3,6]$ when $\boxed{k=3}$

$= m[3,3] + m[4,6] + P_2 P_3 P_6$

$= 3500 + (15)(5)(25)$

$= 5375$.

$m[3,6]$ when $k=4$

$= m[3,4] + m[5,6] + P_2 P_4 P_6$

$= 750 + 5000 + (15)(10)(25)$

$= 9500$.

$m[3,6]$ when $k=5$.

$= m[3,5] + m[6,6] + P_2 P_5 P_6$

$= 2500 + (15)(20)(25)$

$= 10000$.

$m[1,5]$

$i = 1, \ldots, j = 5$

$k = 1, 2, 3, 4$.

$m[1,5] = \min(m[1,5]_{k=1},\ m[1,5]_{k=2},\ m[1,5]_{k=3},\ m[1,5]_{k=4}$

$m[1,5]$ when $k=1$

$= m[1,1] + m[2,5] + P_0 P_1 P_5$

$= 7125 + (30)(35)(20)$

$= 28125$.

$m[1,5]$ when $k=2$

$= m[1,2] + m[3,5] + P_0 P_2 P_5$

$= 15750 + 2500 + (30)(15)(20)$

$= \cancel{25750}\ 27250$.

$m[1,5]$ when $k=4$

$= m[1,4] + m[5,5] + P_0 P_4 P_5$

$= 9375 + (30)(10)(20)$

$= 15375$.

$m[1,5]$ when $\boxed{k=3}$

$= m[1,3] + m[4,5] + P_0 P_3 P_5$

$= 7875 + 1000 + (30)(5)(20)$

$= \cancel{13375}\ 11875$

$m[2,6]$

$i=2 \quad j=6$

$k=2,3,4,5$.

$m[2,6] = \min(m[2,6], m[2,6], m[2,6], m[2,6])$
$k=2 \quad k=3 \quad k=4 \quad k=5$

$m[2,6]$ when $k=2$.

$= m[2,2] + m[3,6] + P_1 P_2 P_6$.

$= 5375 + (35)(15)(25)$

$= 18500$

$m[2,6]$ when $k=3$.

$= m[2,3] + m[4,6] + P_1 P_3 P_6$

$= 2625 + 3500 + (35)(5)(25)$

$= 10500$.

$m[2,6]$ when $k=4$.

$= m[2,4] + m[5,6] + P_1 P_4 P_6$

$= 4375 + 5000 + (35)(10)(25)$

$= 11125$.

$m[2,6]$ when $k=5$

$= m[2,5] + m[6,6] + P_1 P_5 P_6$

$= 7125 + (35)(20)(25)$

$= 10625$

$m[1,6]$

$i=1 \quad j=6$

$k = 1, 2, 3, 4, 5$.

$m[1,6] = \min_{k=1,2,3,4,5} (m[1,6], m[1,6], m[1,6], m[1,6], m[1,6])$

$m[1,6]$ when $k=1$

$= m[1,1] + m[2,6] + P_0 P_1 P_6$

$= 10500 + (30)(35)(25)$

$= 15750$.

$m[1,6]$ when $k=2$

$= m[1,2] + m[3,6] + P_0 P_2 P_6$.

$= 15750 + 5875 + (30)(15)(25)$

$= 32375$.

$m[1,6]$ when $\boxed{k=3}$

$= m[1,3] + m[4,6] + p_0 p_3 p_6$

$= 7875 + 3500 + (30)(5)(25)$

$= 15125$.

$m[1,6]$ when $k=4$

$= m[1,4] + m[5,6] + p_0 p_4 p_6$

$= 9375 + 5000 + (30)(10)(25)$

$= 21875$.

$m[1,6]$ when $k=5$

$= m[1,5] + m[6,6] + p_0 p_5 p_6$

$= 11875 + (30)(20)(25)$.

$= 26875$.

16/9/2015.

(iv) Find optimal soln from optimal value.

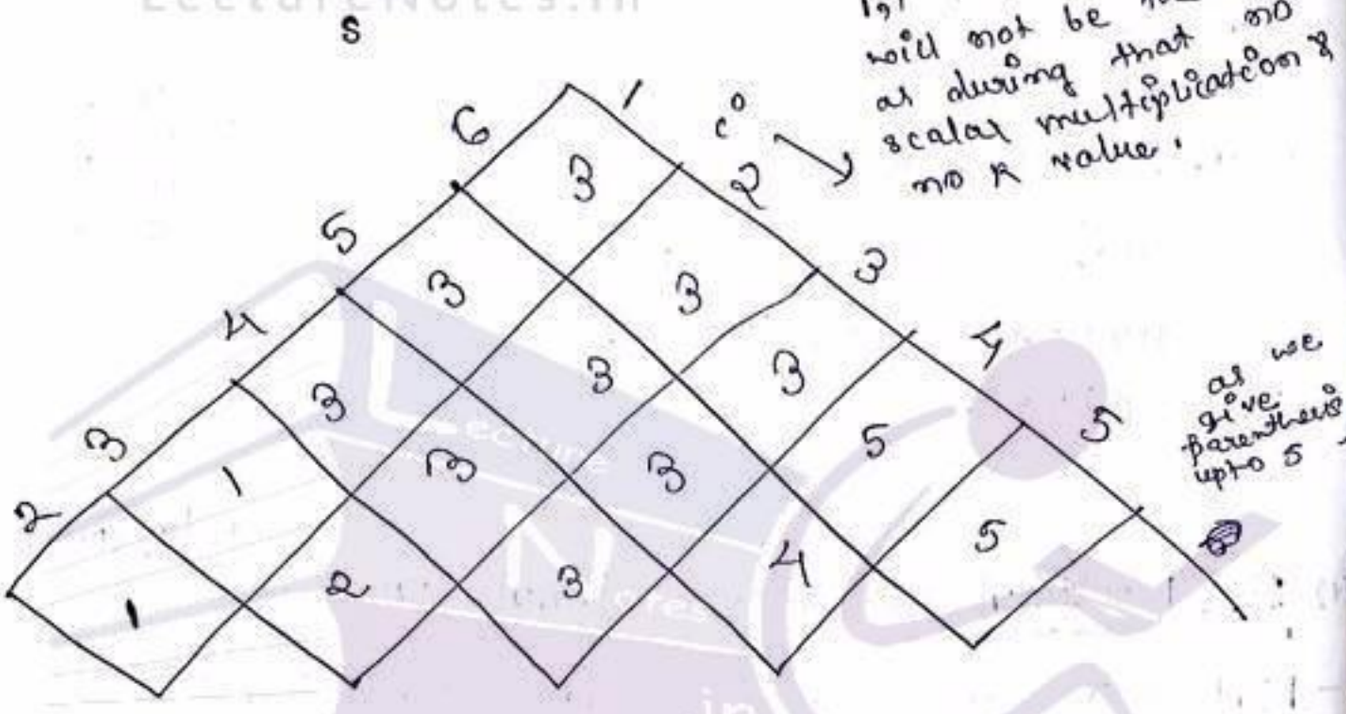
* Matrix (m) of matrix chain multiplication problem gives the minimum number of scalar multiplication need to compute the product of chain of matrices A_i to A_j, i.e., $A_{i \cdots j}$.

$m[1,n]$ i.e., $m[1,6]$ in the example gives the minimum number of scalar multiplication needed to compute the product of of chain $A_1, A_2, \cdots, A_6$, i.e., $A_{1 \cdots 6}$.

So, $\boxed{m[1,6] = 15125}$ → optimal value.

- To get the optimal solution, another table $s[1 \ldots n, 1 \ldots n]$ is constructed.

Every $s[i, j]$ will contain the value of k for which $m[i, j]$ gives minimum scalar multiplication.



Algorithm to compute optimal parenthesis.

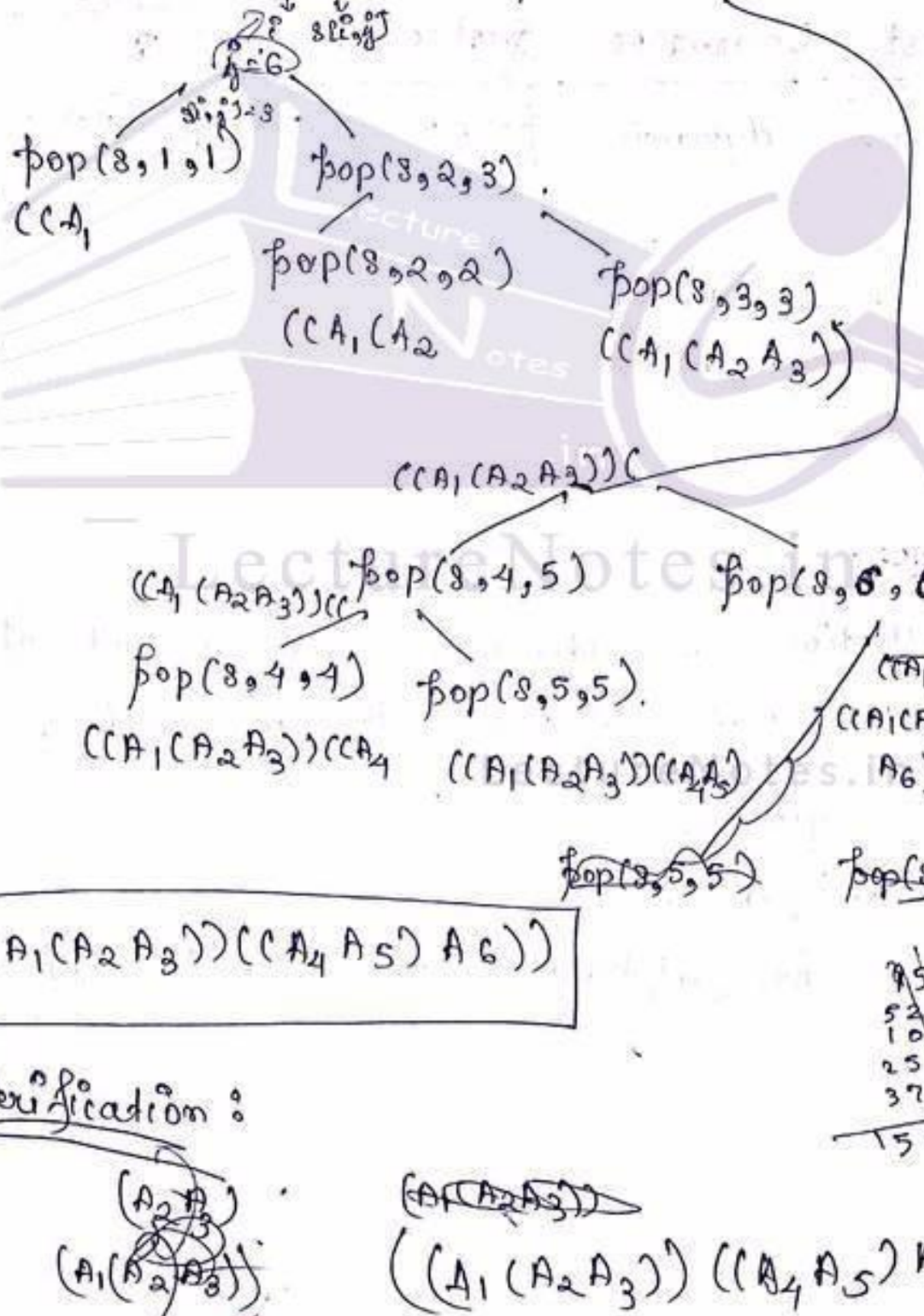
1. Print optimal parenthesis (s, i, j)
2. if i == j
3. then print A_i
4. else print '('
5. Print_optimal_parenthesis (s, i, s[i, j])
6. Print_optimal_parenthesis (s, s[i, j]+1, j)
7. Print ')'.

7778888232

* To find the optimal solution i.e., optimal parenthesization of $A_1 \ldots A_n$ i.e., the given matrix chain algorithm print_opt(s, 1, n) (pop : print_optimal_parenthesis) will be invoked first.

Eg, pop(s, 1, 6) ⟶ input.

1st: pop(s, 1, 3) pop(s, 4, 6)



$((A_1(A_2A_3))((A_4A_5)A_6))$

Verification:

$((A_1(A_2A_3))((A_4A_5)A_6))$

Optimum soln $m[1,6] = 15125$.

Exercise
15.2.1

Find the optimal parenthesization of the matrix chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

21/9/2015

Longest Common Subsequence (LCS)

* By using dynamic programming approach.

Sequence
Subsequence (Defn)
Substring
Common Subsequence
LCS

Sequence

A collection of characters over a set of alphabets and arranged in an order.

Ex: DNA sequence
aatcgaccggtcc,
defined on alphabet set = {a, t, c, g}.

Sub-sequence :

$S = A\ B\ C\ A\ C\ D\ B$

$S_1 = ABA$ (Subsequence of S).

$S_2 = ADB$ (Subsequence of S).

$S_3 = DBABB$ × (order of the characters should be maintained).

Given a sequence

$$X = \langle x_1\ x_2 \cdots x_m \rangle$$

another sequence $k \leq m$.

$$Z = \langle z_1\ z_2 \cdots z_k \rangle$$

is called subsequence of X if there exist a strictly increasing sequence

$$\langle i_1, i_2 \cdots i_k \rangle$$

of idies of X for $j = 1 \cdots k$.

*- In a subsequence of a sequence the characters may not be continous, but they have to be contigous.

Ex: $S = aatcgc$

$S_1 = aatc$ (it is a substring & subsequence)

$S_2 = atgc$ (it subsequence) but not substring.

Common Subsequence.

S_1 = ABBCDEA.

S_2 = ADBADECA.

S_1 = ABD (Subsequence of S_1 and S_2). so, common subsequence.

S_2 = ABC

S_3 = ABDEA. (longest common subsequence of S_1 and S_2).

Objective of LCS Problem.

To find the LCS between two given strings.

Longest Common Subsequence Problem.

Given

$X = \langle x_1, x_2 \cdots x_m \rangle$

and

$Y = \langle y_1, y_2 \cdots y_n \rangle$

The LCS problem tries to find max length common subsequence of X and Y.

1 brute force approach.

Find all subsequence of X.

Check if it is a subsequence of Y,

then find the largest among them.

* Given a string of length(m), it can have 2^m subsequence.

Finding LCS of two given strings take exponential time which is impractical for long strings.

Solution:

Solve it by dynamic programming approach.

Prefix:

Given a string

$X = \langle x_1\ x_2 \cdots x_m \rangle$,

the ith prefix of X is

$X_i = \langle x_1\ x_2 \cdots x_i \rangle$.

Ex. $X = \langle A\ B\ C\ B\ D\ A\ B \rangle$

$X_4 = \langle A\ B\ C\ B \rangle$. (4th prefix of X).

$X_5 = \langle A\ B\ C\ B\ D \rangle$

X_0 = empty prefix of X.

Theorem 15.1 :

Let $X = \langle x_1\ x_2 \cdots x_m \rangle$

and

$$Y = \langle y_1\ y_2 \cdots y_n \rangle$$

be sequences and let

$$Z = \langle z_1\ z_2 \cdots z_k \rangle$$

be the LCS of X and Y

(i) if $x_m = y_n$,
then $z_k = x_m = y_n$ and z_{k-1} is the LCS of x_{m-1} and y_{n-1}.

(ii) if $x_m \neq y_n$,
then $z_k \neq x_m$ implies Z is LCS of x_{m-1} and y_n.

(iii) if $x_m \neq y_n$,
then $z_k \neq y_n$ implies Z is LCS of X and y_{n-1}.

Proof :

Ex. $X = ABCBDA$.

$Y = BDCABA$

$Z = BCBA$.

$x_m = A$

$y_n = A$

$Z_K = A$

$Z_{K-1} = BCB$

$X_{m-1} = ABCBD$

$Y_{n-1} = BDCABB$

LCS = BCB

$\therefore Z_{K-1} = LCS$.

(iii) Let us proof it by binomial theorem;

Let $Z_K \neq (X_m = Y_n)$.

Then, we could append $(X_m = Y_n)$ to Z to obtain a common subsequence of X and Y of length (K+1) which contradicts the supposition that Z is the longest common subsequence of X and Y.

Thus, $X_m = Y_n = Z_K$.

To prove Z_{K-1} is the LCS of X_{m-1} and Y_{n-1} when $X_m = Y_n$ & $Z_K = X_m = Y_n$.

Let Z_{K-1} is not the LCS of X_{m-1} and Y_{m-1} and there exist another subsequence W which is LCS of X_{m-1} and Y_{n-1} and of length greater than (K-1).

$X_m = Y_n$ can be appended to last of W

to obtain LCS of X and Y and of length greater than k which is a contradiction. So, z_{k-1} is the LCS of x_{m-1} and y_{m-1}.

Proof :-

(i) $X = ABCBDAC$

$Y = BDCABABD$

23/9/2015

if $x_m \neq y_m$

and $z_k \neq x_m$,

then Z is LCS of X_{m-1} and Y.

Ex: $X = ABCBDAC$

$Y = BDCABAD$

$Z = x_m$ or y_m or X_{m-1} or Y_{m-1}.

Step-1:

(Optimal characterization of the problem)

Let $X = \langle x_1, \cdots, x_m \rangle$

and $Y = \langle y_1, \cdots, y_n \rangle$

be two given sequences.

To find the LCS of X and Y,

if $X_m = Y_n$,

then find X_{m-1} and Y_{n-1}; then append

$(X_m = Y_n)$ to last to get LCS of X and Y.

Ex: $X = ABCBDA$

$Y = BDCABA$.

$x_m = y_n = A$.

To find LCS of X and Y.

(i) find LCS of $X_{m-1} = ABCBD$

$Y_{n-1} = BDCAB$.

$Z_{k-1} = BCB$.

To find LCS of X and Y_n append $X_m = Y_n$ i.e., A at end of Z_{k-1} i.e., BCB.

So, $Z = BCBA$.

If $x_m \neq y_n$,

then find LCS of X_{m-1} and Y.

find LCS of X and Y_{n-1}.

Which ever is greater becomes LCS of X and Y.

Ex: $X = ABCBDACE$

$Y = BDCABAED$.

$X = ABCBDACE$

$Y_{n-1} = BDCABAE$

LCS1 = BCBAE

$X_{m-1} = ABCBDAC$E

$Y = BDCABAED$.

LCS 2 = BCBA .

LCS 1 > LCS 2

So, LCS 1 is LCS of X and Y.

Step-2: (Define Recursive Sol^m).

Let C(i, j) is the length of LCS of X_i and Y_j,

then

$$C(i,j) = \begin{cases} 0 & , i=0 \text{ and } j=0 . \\ C(i-1, j-1)+1 & , \text{if } X_i = Y_i . \\ \max(C(C(i-1), j), C(i, (j-1))) & \text{if } X_i \neq Y_j . \end{cases}$$

Step-3 (Constructing optimal value).

* The optimal value C(m, n) is obtained by constructing a dynamic programming table of size (m+1) x (n+1).

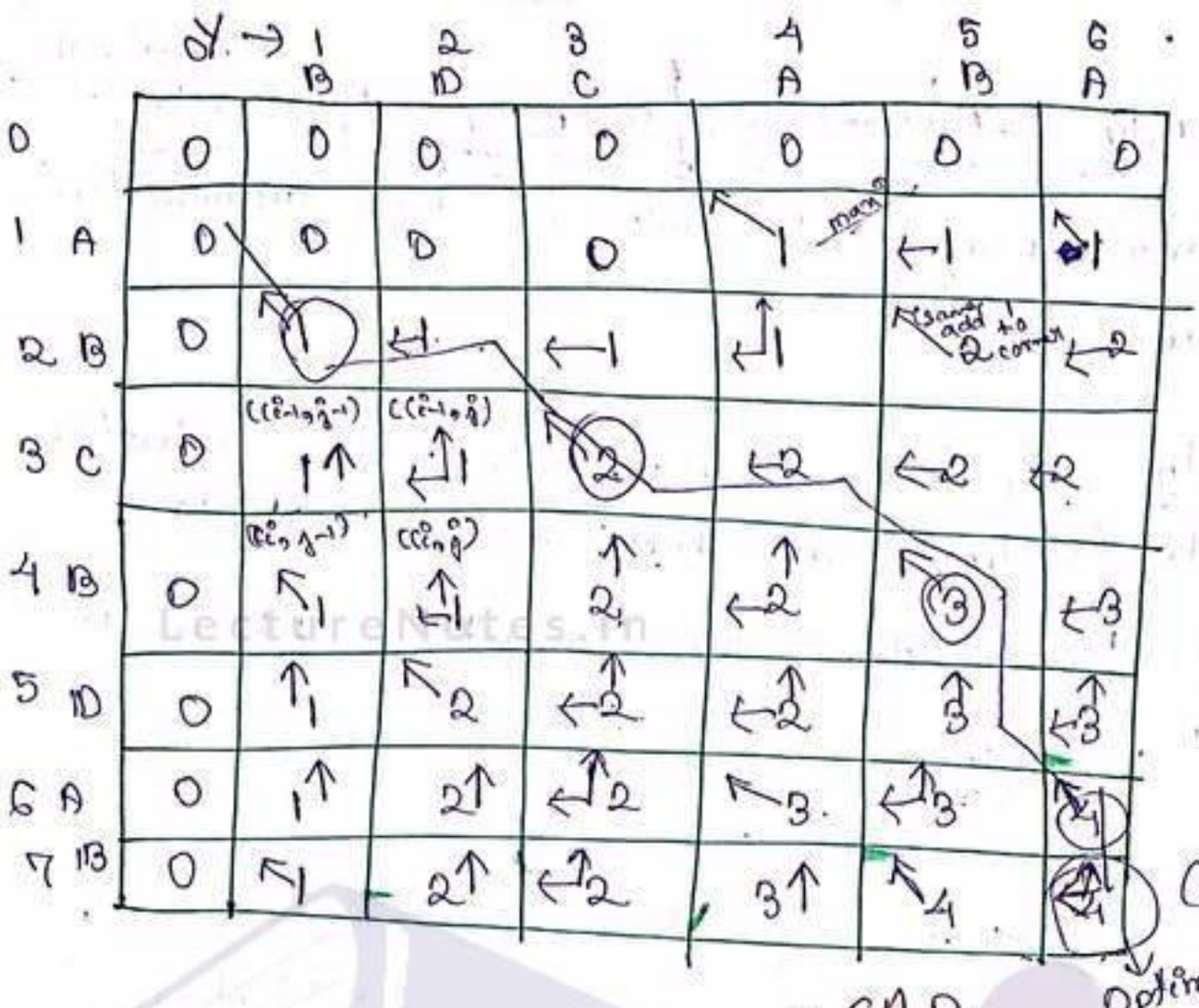
* Each value of the table is computed by using the recurrence.

Ex: X = ABCBDAB

Y = BDCABA .

|X| = 7

|Y| = 6 .



BCAB

C(m,n) gives the optimal length of LCS.

Step-4: (Find optimal Path).

* - To get the optimal solution starting from C(m,n) cell and back traper to the cell containing the zero value using the arrows.

Take the up arrow.
(left)
A
(diagonal arrow)

BCBA is the LCS of both the strings.
and BCAB also.

Time complexity.

24/9/2018.

Greedy Algorithm / Greedy Approach

*- Another way of solving the optimization problem.

*- Greedy algorithm make choice of solution which is best at that moment. It has, They look for the solution which is best locally.

Ex:

Greedy chooses the path P_4 initially.

*- Greedy algorithm never guarantees optimal solution. But, for many problems it has been proved that it gives optimal sol^n.

In dynamic programming, the sub-problems are dependent known as dependent or overlapping sub-problems.

Activity Selection Problem.

* Let a set of activities

$S = \langle a_1, \ldots, a_n \rangle$

i.e., n number of set activities available who wants to share a common resource.

* The activity selection problem aims to find maximum size subset of mutually compatible activities.

* Each activity (a_i) has a start time (s_i) and finish time f_i i.e., $[s_i, f_i)$ with $0 \leq s_i < f_i$ (s_i included, f_i may or may not included).

* Two activities a_i and a_j are called mutually compatible if $a_i = [s_i, f_i)$ and $a_j = [s_j, f_j)$.

if $s_i \geq f_j$ or $s_j \geq f_i$.

* Compatible means they don't overlap.

28/9/2015.

IGM.

Ex: $S = \{a_1, \ldots, a_{11}\}$.

$n = 11$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

$S_1 = \{a_1\ a_4\ a_8\ a_{11}\}$

$S_2 = \{a_2\ a_4\ a_8\ a_{11}\}$

$S_3 = \{a_2\ a_4\ a_9\ a_{11}\}$

$S_4 = \{a_2\ a_6\ a_{11}\}$

$S_5 = \{a_3\ a_7\ a_{11}\}$

$S_6 = \{a_3\ a_8\ a_{11}\}$.

Goal:

$S_1 = \{a_1\ a_4\ a_8\ a_{11}\}$

$S_2 = \{a_2\ a_4\ a_8\ a_{11}\}$

$S_3 = \{a_2\ a_4\ a_9\ a_{11}\}$

any one of the following.

*- First find the optimal substructure of the problem by using dynamic programming approach, then convert it to solve by greedy approach.

(Defn.):

Let S_{ij} is the subset of S which is defined as

$$S_{ij} = \{ a_k : f_i \leq s_k < f_k \leq s_j \}, \quad S_{ij} \subset S$$

(a_k: activity of S.)

S_{ij} contains all jobs which are mutually compatible with a_i and a_j and all the jobs which are finishing before a_i and all the jobs which are starting after a_j finishes.

s_k : start time of a_k

f_k : finish time of a_k.

$S_{0,8}$ can be written as $S_{0,n+1}$ for some fictious activities a_0 and a_{n+1} such that

$$f_0 \leq s_1 < f_1 \cdots f_n \leq s_{n+1}.$$

Let assume the activities are sorted in monotonically increasing order of their finish time.

So, $f_0 \leq f_1 \leq f_2 \cdots \leq f_{n+1}$.

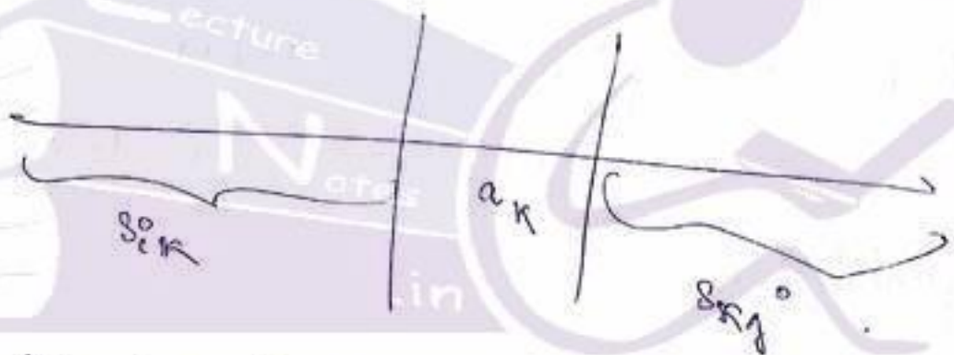
If $i \geq j$

$S_{ij} = \phi$.

S_{ij} → collection of activities which are mutually compatible with a_i and a_j.

* Find optimal sub-structure

$S_{0,n+1}$
(S_{ij})



⇒ $S_{ij} = \text{sol}^n(S_{ik}) + \text{sol}^n(S_{kj}) + a_k$.

29/9/2015.

Consider some non-empty sub-problem (S_{ij}).

Suppose a solution to S_{ij} includes some activity (a_k) such that

$f_i \leq s_k < f_k \leq s_j$.

So, activity (a_k) generates two sub-problems

(i) S_{ik} (activities that start after a_i finishes and finish a_k starts).

If S_{ij} is non-empty,

$c[i,j] = c[i,k] + c[k,j] + 1$

$c[i,k] \rightarrow$ no. of activities in A_{ik}.

$c[k,j] \rightarrow$ no. of activities in A_{kj}.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ c[i,k] + c[k,j] + 1 & \text{if } S_{ij} \text{ is non-empty} \end{cases}$$

* Converting the dynamic programming soln to Greedy soln.

Theorem 16.1

Consider any non-empty sub-problem (S_{ij}) and let a_m be the activity in S_{ij} with the earliest finish time.

If f_m be the finish time of a_m, then

$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

Then,

(i) activity (a_m) is used in some maximum sized subset of mutually compatible activities of S_{ij}. (a_m must be

included in the soln of S_{ij}).

(ii) the sub-problem (S_{im}) is empty ($S_{im} = \phi$) so, choosing a_m leaves the sub-problem S_{mj} is the only for one that may be non-empty.

($a_m \rightarrow$ 1st activity of A_{ij}).

Algorithm:

Recursive-Activity-Selection (s, f, i, j)

1. $m \leftarrow i+1$.
2. while $m < j$ and $s_m < f_i$.
3. do $m \leftarrow m+1$
4. if $m < j$
5. then return $\{a_m\} \cup$ Recursive-Activity-Selection (s, f, m, j).
6. else return ϕ.

$s \rightarrow$ array of start for time of all the activities,
$f \rightarrow$ Array of finish time of all the activities.

*- For the first call, the algorithm will be called for $i = 0$, $j = n+1$.

Recursive-Activity-Selection $(s, f, 0, (n+1))$.

i	1	2	3	4	5	6	7	8	9	10	11
s	1	3	0	5	3	5	6	8	8	2	12
f	4	5	6	7	8	9	10	11	12	13	14

* The greedy approach for solving the activity selection problem chooses the activity (a_m) for solving the sub-problem, the activity with earliest finish time.

* The activity picked is of a greedy choice in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled.

* The greedy choice is the one that maximizes the amount of unscheduled remaining time.

GREEDY-ACTIVITY-SELECTOR(s, f, 0, m+1) $m = 12$. $f_0 = 0$.

sol^n set = $\{a_1, a_4, a_8, a_{11}\}$

i j m

0 12 1

ϕ

RAS(s, f, 1, 12)

i	j	m
1	12	2 → 3 → 4

if (m < j)
    return a_4.

RAS(s, f, 4, 12)

i	j	m
4	12	5 → 6 → 7 → 8

RAS(s, f, 8, 12)

i	j	m
8	12	9 → 10 → 11

RAS(s, f, 11, 12)

i	j	m
11	12	12

return ϕ.

Time Complexity

The algorithm needs f in sorted order.
So, if the sorting takes $O(n \log n)$.
Then, time taken by activity selection problem will be

$$O(n \lg n) + O(n)$$

RAS

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

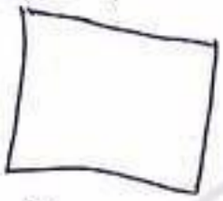
1|10|2015.

Huffmann Codes.

*- Huffman codes are efficiently and effectively used for data compression. It follows greedy approach for compression of data.

Data Compression → reducing the size of file.

Text file



It contains character

1,00,000 characters.

(1,00,000 × 8) bits = 100,000 bytes

= 100 KB

1 bit → store 2 chr codes (0, 1)

2 bits → 4 chr

3 bits → 8

4 → 16

n bits → store 2^n chrs.

chrs.	a	b	c	d	e	f
frequency (in)	45	13	12	16	9	5

*- If the characters are represented using binary code, then to represent 6 characters, only 3 bits are enough.

*- If 3 bit code is used to represent as each character code present in file.

Then, the file size is

Ans: (3,00,000) bits

37.5 KB.

So, for compressing a file, binary character code is used to represent the characters of a file.

* The binary character code can be of
  (i) fixed length code
  (ii) variable length code.

Fixed length code means the length of binary code to represent each character is same for all the characters. (takes more memory).

In variable length code, the frequent characters are represented by short length code and infrequent characters are represented by long codes or codewords.

Ex: (Variable length code)

a	b	c	d	e	f
45	13	12	16	9	5

Characters	Codeword
a	0
b	101
c	100

d	111
e	1101
f	1100

File size

= 45×1 + 13×3 + 12×3 + 9×4 + 5×4 + 16×3

= 45 + 39 + 36 + 36 + 20 + 48

= 224,000 bits (as all are taken in thousand 45000, 13000)

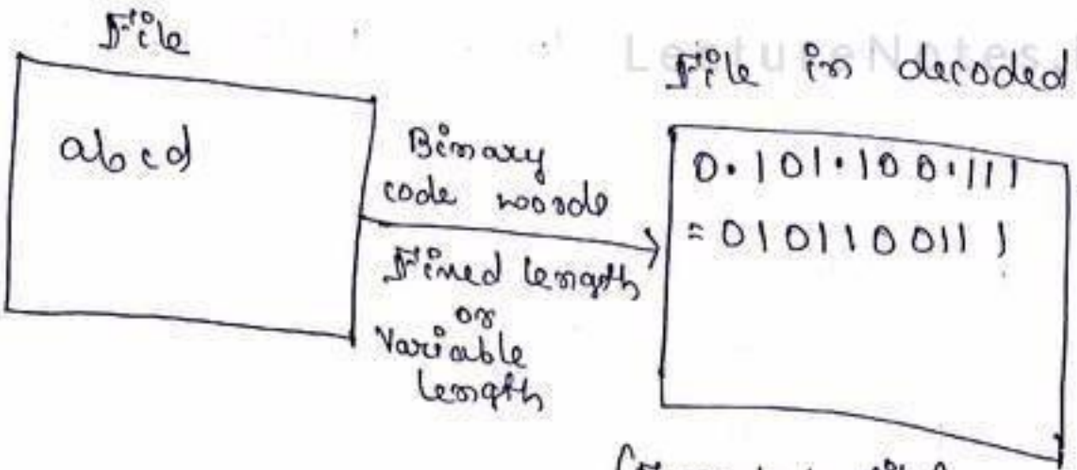
≈ 2 KB

45
13
12
9
5
16
30

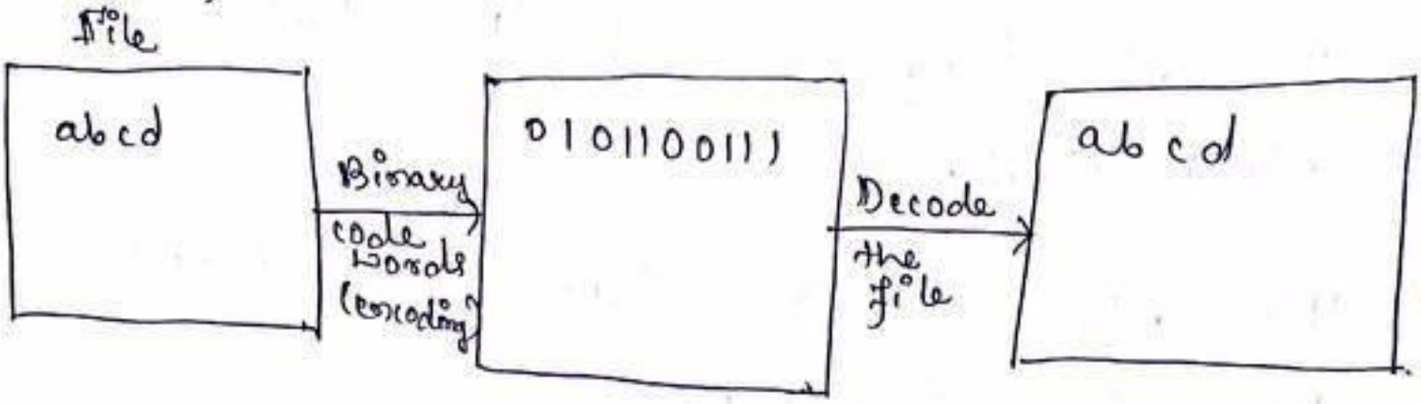
Prefix Code.

Encoding.

Encoding is the process of representing the characters of original file by using fixed or variable length binary codeword.



To read the file, the receiver has to decode the file.



* No codeword will be a prefix of another codeword.

a ~ 0
b ~ 101
c ~ 100
d ~ 111
e ~ 1101
f ~ 1100

→ Prefix code.

0 0 101 111
↓ ↓ ↓ ↓
a a b d

5/10/2015

* For efficient decoding process, the codewords has to be represented in an efficient manner. One of such representation is done by using binary tree.
* In binary tree, the leaves represents the characters along with their frequency and each internal node

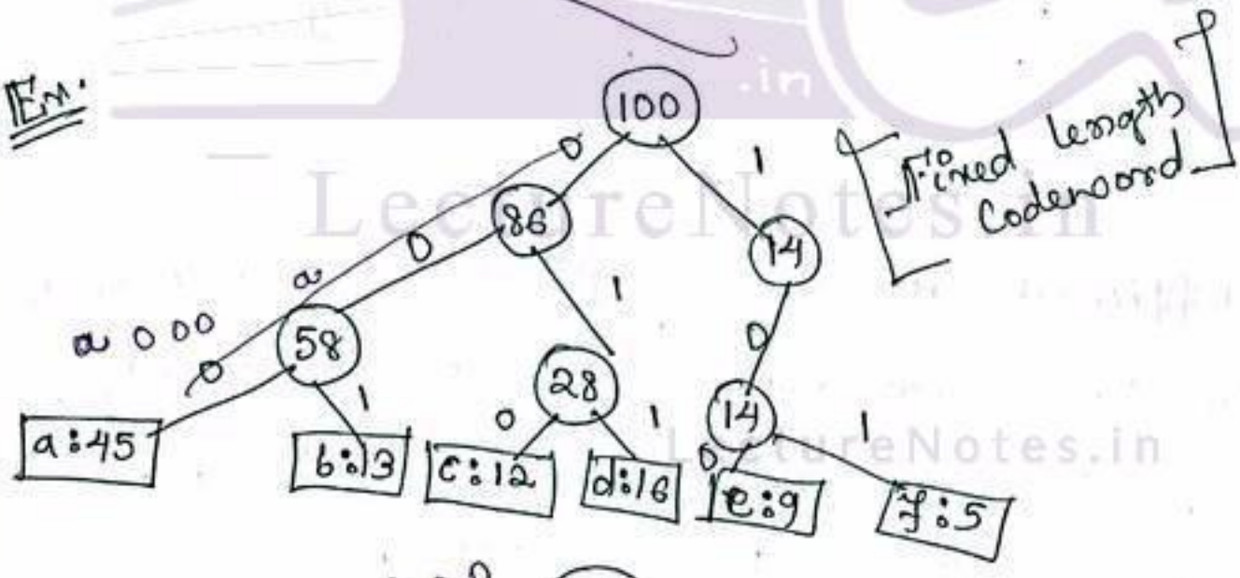
of the tree contains the cumulative or sum of frequency of its left and right child.

*- The edges of the tree are labelled by using 0 and 1.

0 means go to left child and 1 means go to the right child.

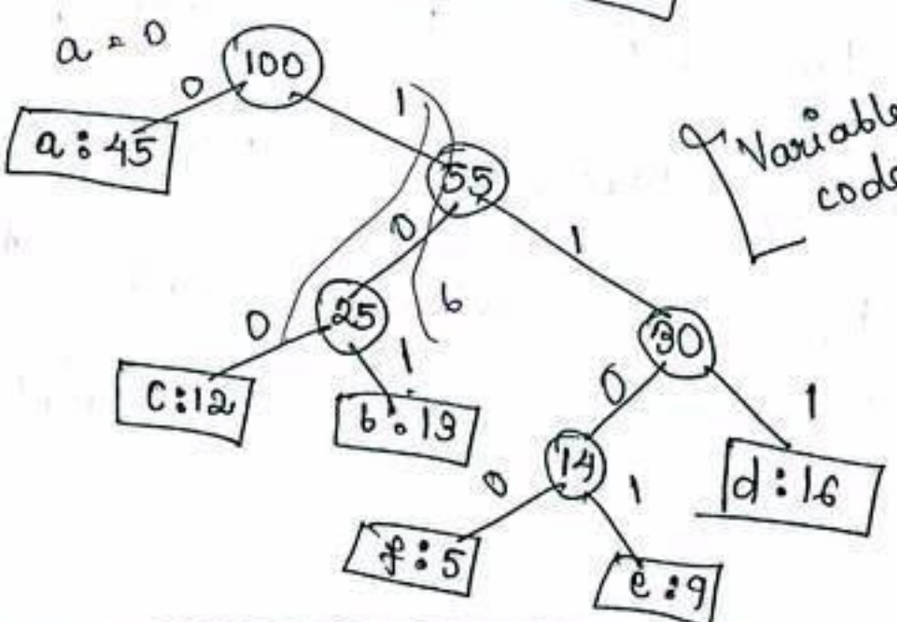
*- The codeword of a character is the concatenation of the edges of the levels starting from root to itself.

Ex.



Fixed length codeword

a = 000



Variable length codeword

Full binary tree / binary 2-Tree

↓ every internal node has no child or 2 child.

* An optimal code for a file is always represented by a full binary tree in which every non-leaf node has two children.

* If "C" is the alphabet from which the ch's characters are derived and the ch's have positive frequency, then the tree for an optimal prefix code has exactly |C| leaves one for each letter of the alphabet and has exactly |C|-1 number of internal nodes.

$C = \langle a, b, c, d, e, f \rangle$.

* Given a tree (T) corresponding to a prefix code, then the number of bits required to encode a file

$$B(T) = \sum_{c \in C} f(c)\, d_T(c)$$

$f(c) \rightarrow$ frequency of c.

$d_T(c) \rightarrow$ Depth of c from root.
↓ no. of bits required to encode the ch.

Ex: $B(T) = 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4$

$= 2,24,000$ bits.

HUFFMAN - Algorithm:

* maintains a priority queue (Min-Priority Queue).
* HUFFMAN invented a greedy algorithm that constructs an optimal prefix code called HUFFMAN code.
* Let "C" is a set of alphabets with n number of characters. Each character in "C" $c \in C$ has a positive frequency.
* The algorithm builds the tree (T) in a bottom-up approach. It begins with |C| number of leaves and performs a sequence of |C|-1 number of merging operation to create the final tree.
* A min priority queue (Q) keyed on frequency (f) is used to identify

the two least frequent objects to be merged together.

* The result of merging operation of two objects is a new object where the frequency is the sum of the frequencies of the two objects.

1. $n \leftarrow |C|$
2. $Q \leftarrow C$ (constructs a min heap out of Q).
3. for $i \leftarrow 1$ to $(n-1)$
4. do allocate a new node (z)
5. left(z) $\leftarrow x \leftarrow$ EXTRACT_MIN(Q).
6. right(z) $\leftarrow y \leftarrow$ EXTRACT_MIN(Q).
7. $f(z) \leftarrow f(x) + f(y)$.
8. INSERT(Q, z).
9. return EXTRACT_MIN(Q).
10. end.

2nd step constructs a min-heap which works in $O(n)$ time

From 3 to 8, $(n-1)$ times it applies extract min from the min-heap which takes $O(\log n)$.

So, the total time taken from 3 to 8 is $O(n \log n)$.

9th step will return take constant time.

Total time taken by HUFFMAN code is

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

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

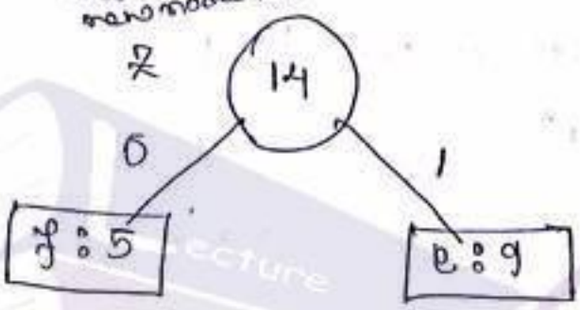
Ex: a min heap is to be formed.

Q: f:5 | e:9 | c:12 | b:13 | d:16 | a:45

(min-priority queue)

Step-I:

Create a new node (z)

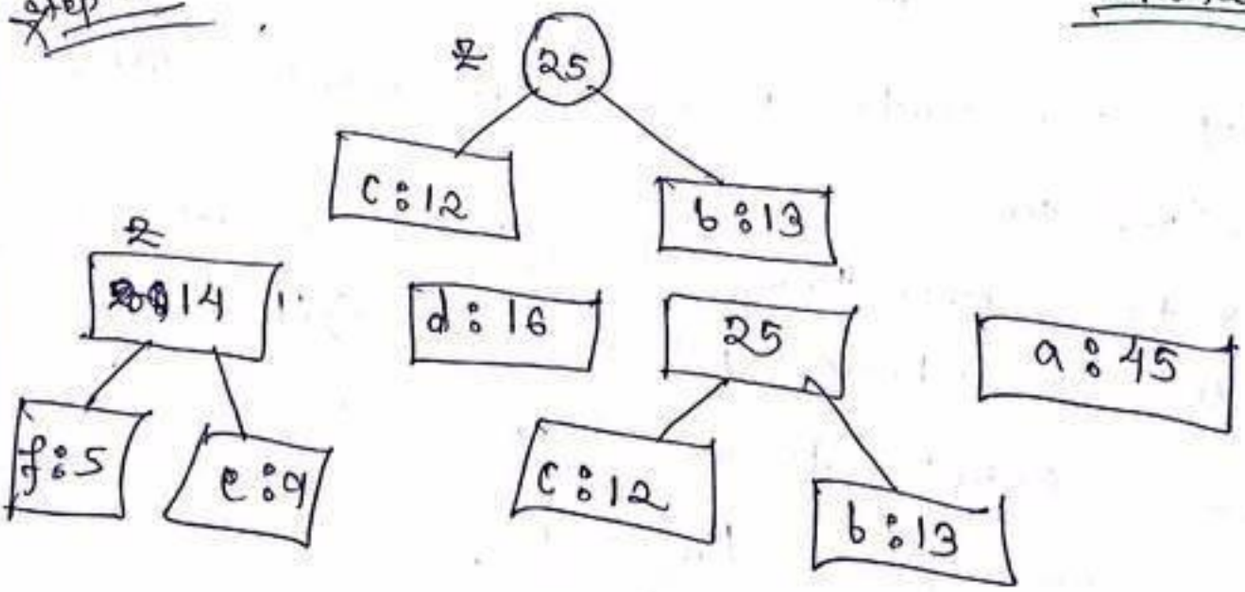


Insert z into priority queue.

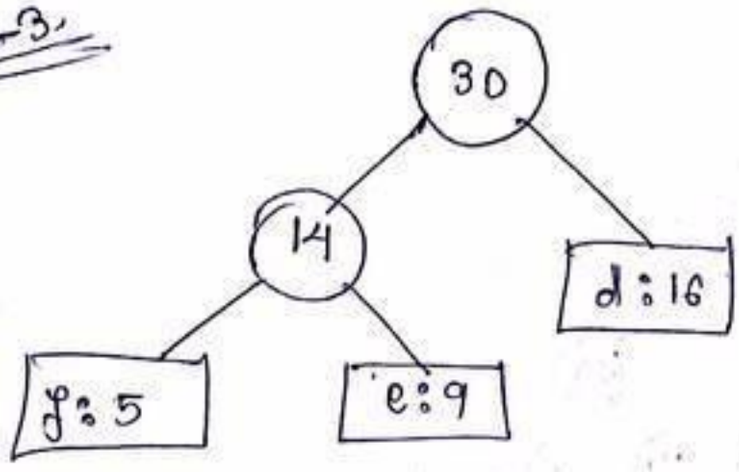
Q: c:12 | b:13 | z:14 | d:16 | a:45

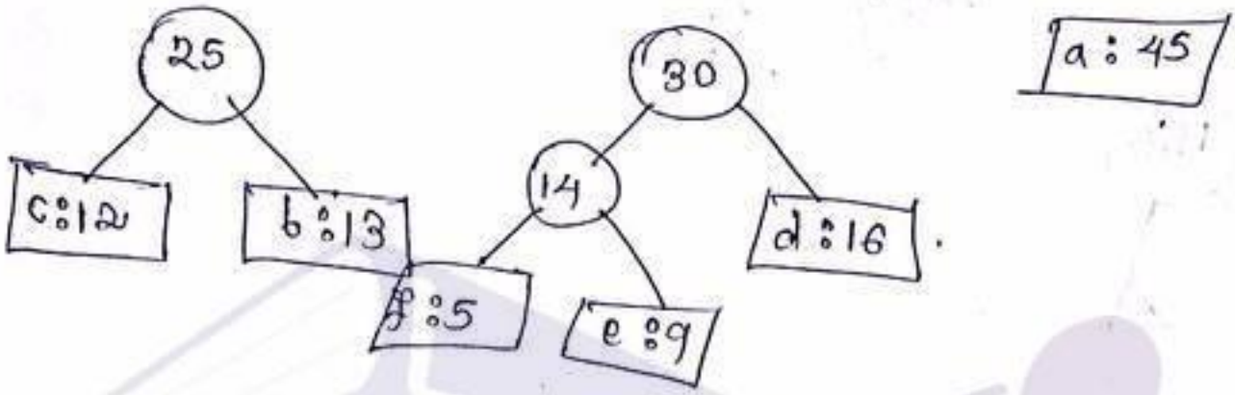
Step-II:

6/10/2015.

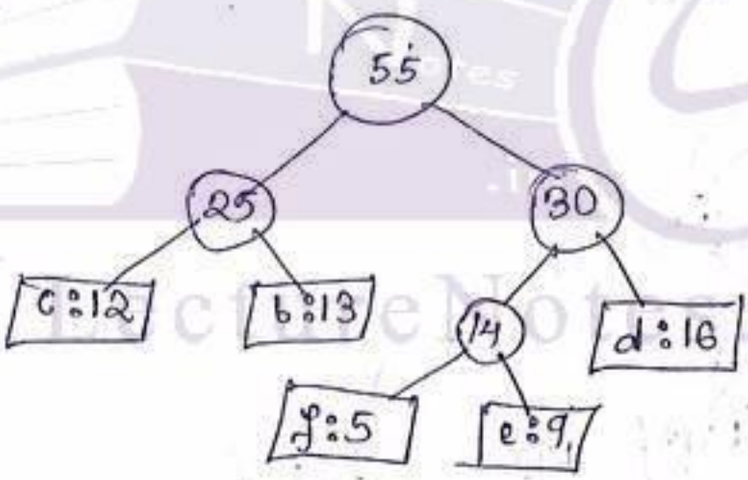


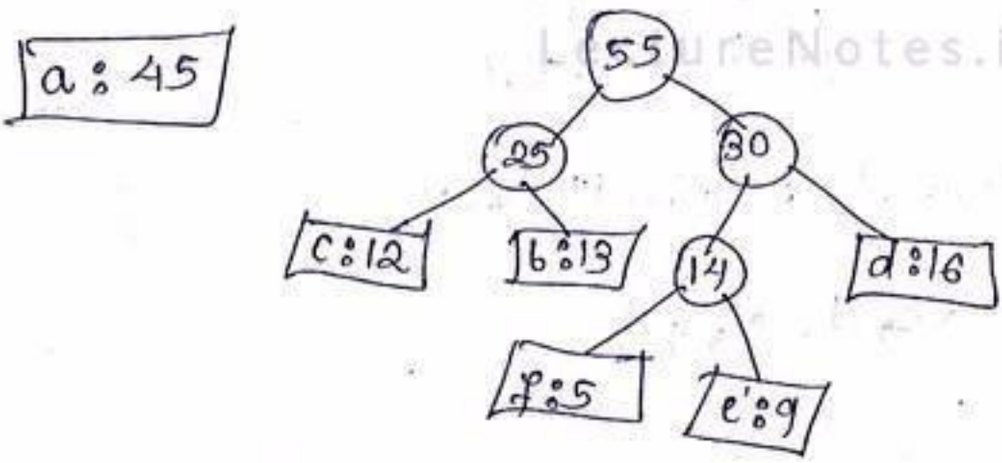
Step-3.


30
14
d:16
f:5
e:9

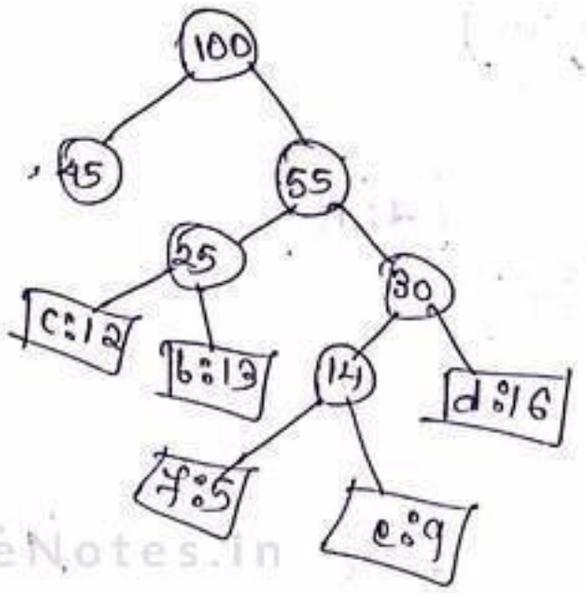

25
30
a:45
c:12
b:13
14
d:16
f:5
e:9

Step-4.

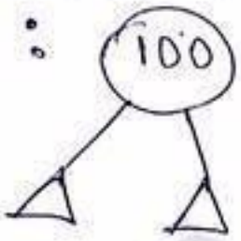

55
25
30
c:12
b:13
14
d:16
f:5
e:9


a:45
55
25
30
c:12
b:13
14
d:16
f:5
e:9

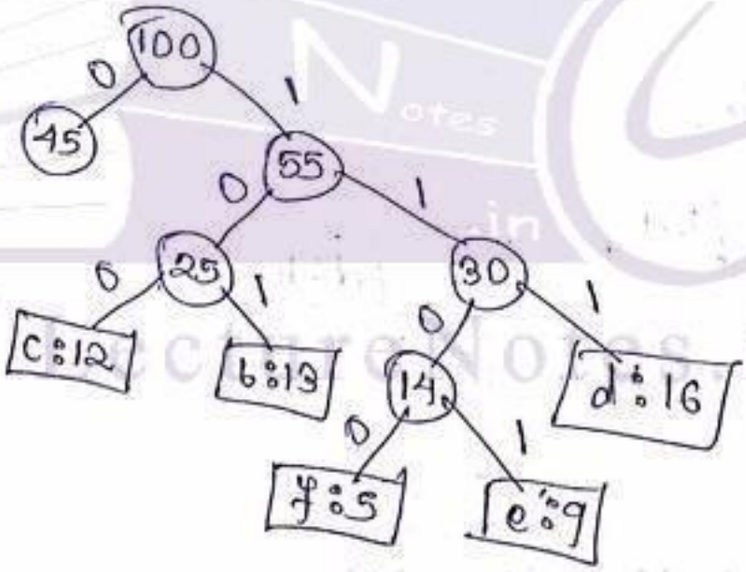
Steps:



Q:



Hoffman Code Tree:



No. of merging required = (n-1).

$C = \{a, b, c, d, e, f\}$

$|C| = 6$.

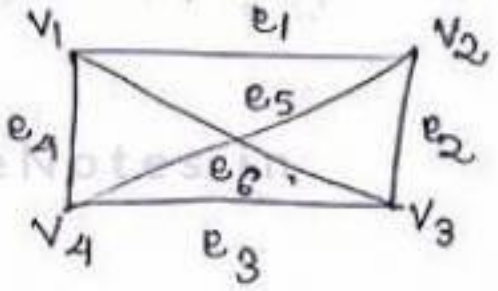
* Number of leaves in the prefix code representation tree is 6 i.e., n.

* No. of non-leaf nodes = 5 (n-1).

Graph Algorithms

Graph:

$G = (V, E)$



A graph

$G = (V, E)$

is a collection of non-zero set of vertices and edges.

$V = \{v_1, v_2, v_3, \cdots, v_n\}$ if $|V| = n$.

$E = \{e_1, e_2, e_3, \cdots, e_m\}$ if $|E| = m$.

$e_1 = (u, v)$.

In a graph (G) if

$(v_1, v_2) = (v_2, v_1)$

that graph is called undirected graph.

If the graph is directed, then

$e = (v_1, v_2) \neq (v_2, v_1)$.

Adjacent Vertices.

(V_i, V_j), the vertex V_i is adjacent to V_j if there is an directed edge from V_i to V_j.

V_1 is adjacent to V_2, but V_2 is not adjacent to V_1.

Complete Graph.

In a complete graph (G), any vertex is adjacent to any other vertex.

Connected Graph.

Every vertex is connected to any other vertex of the graph.

Connected.

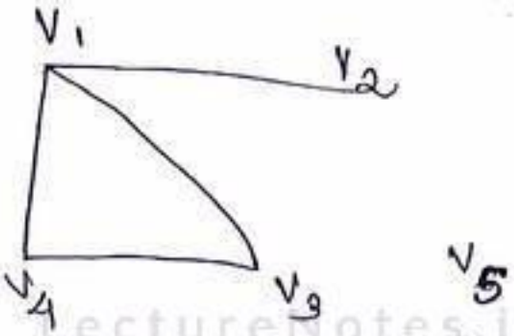
V_1 is connected to V_n if there exist a path from V_1 to V_n.

Path.

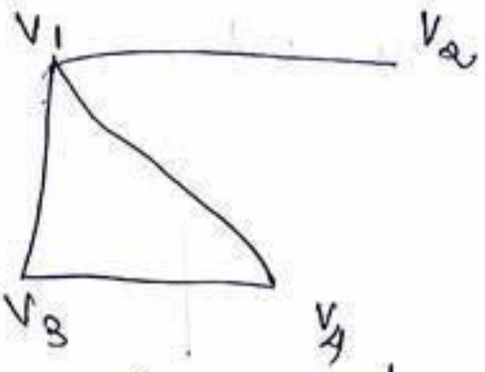
A path from V_1 to V_n is a collection of vertices $\{V_1, V_2, V_3, \ldots, V_{n-1}, V_n\}$ where V_1 is adjacent to V_2, V_2 is adjacent to V_3, ..., V_i is adjacent to V_j, ..., V_{n-1} is adjacent to V_n.

* If the path starts and ends at the same vertex, it is a cyclic graph.

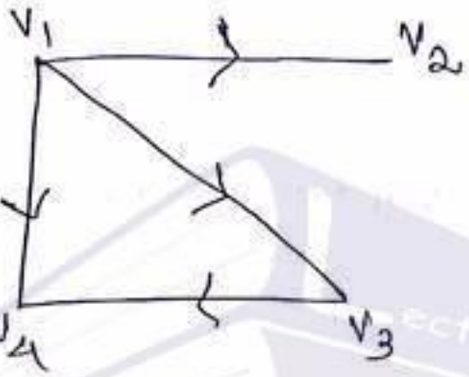
Ex:



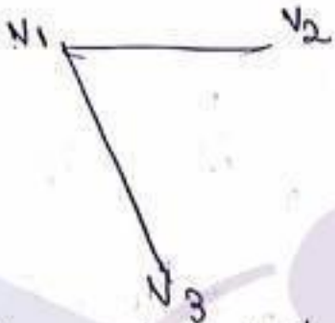
Not connected



Connected



Not connected.



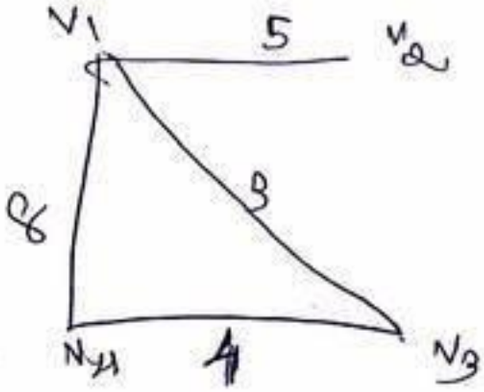
Connected but not complete.

Complete

If each edge is associated with a non-zero value, then it is a weighted graph.

Weight can be +ve or -ve.

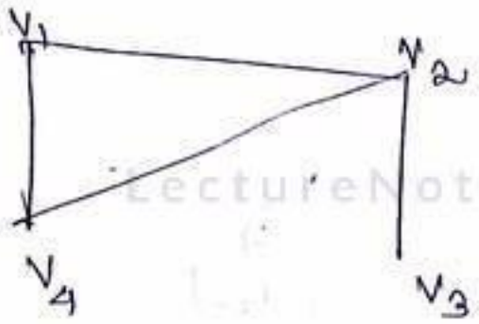
Ex:



Representation of a Graph.

Array Representation:

Represented by adjacency matrix and adjacency list.

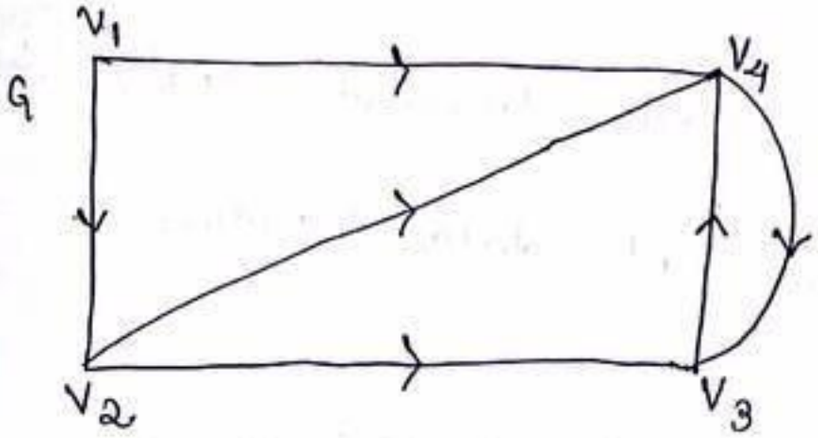


$A = (a_{ij})$

$$a_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise} \end{cases}$$

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	1
v_3	0	1	0	0
v_4	1	1	0	0

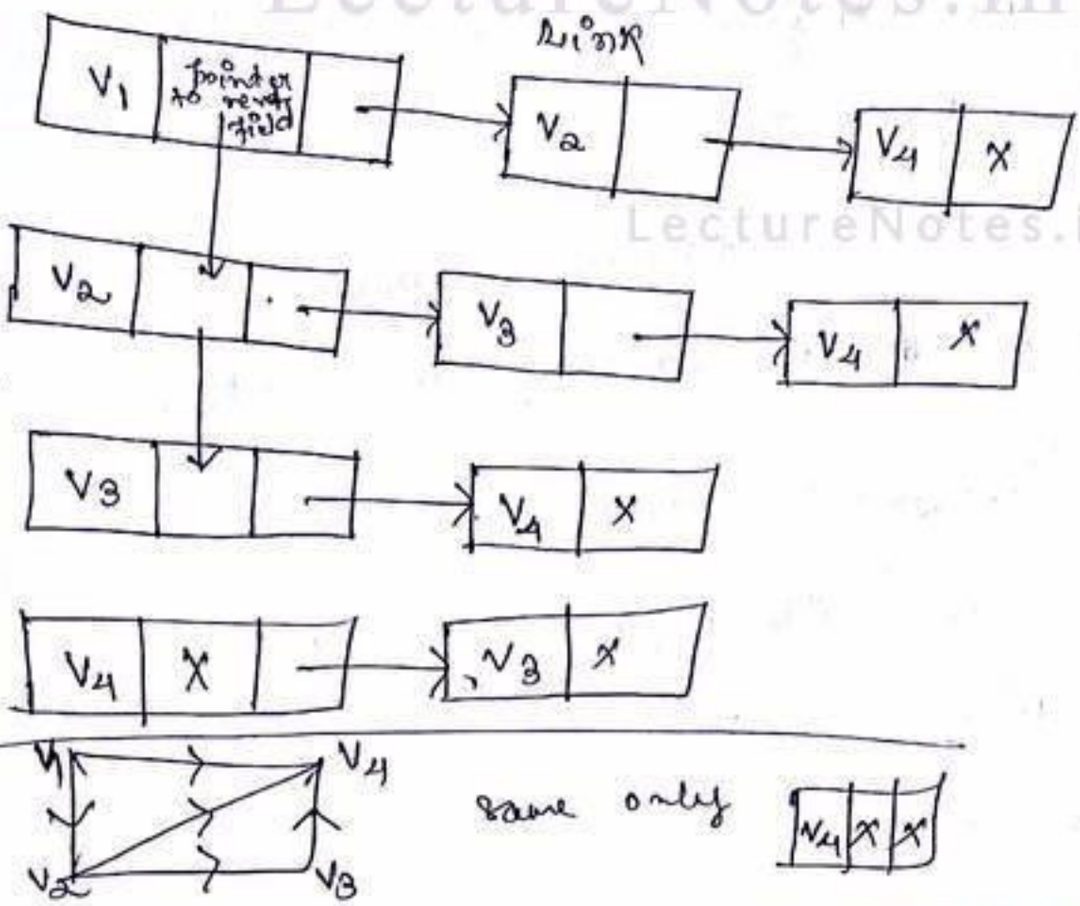
(iii) Linked Representation.



Nodes	Adjacency Matrix
V_1	V_2 V_4
V_2	V_3 V_4
V_3	V_4
V_4	V_3

Node

Link



same only

Data Structure for Disjoint Sets.

* Grouping n distinct elements into disjoint sets is addressed by disjoint set data structure.

* To group n distinct element into disjoint sets normally involves the operation of identifying which object belongs to which set and uniting the two sets.

n elements → Graph containing n vertices.

S_1, S_2, S_3 are disjoint : there will be no overlapping elements.

* A disjoint set data structure maintains a collection (set) of disjoint dynamic sets.

$$S = \{S_1, S_2, \ldots S_n\}$$

* Each set is identified by a representative which is a member of same set.

Disjoint set operations.

* In dynamic set implementation, each element of a set is represented by an object.

Let x denotes an object then, the following operations are used:

(i) Make-set(x): This operation creates a new set whose only member is x. (Since the sets are disjoint so, x should not be there in any other set).

(ii) Find-set(x): This operation returns the representative of the set which contains x.

(iii) UNION(x, y): Let S_x be a dynamic set containing x and S_y be a dynamic set containing y. Then, this operation unites S_x and S_y ($S_x \cup S_y$) only when their representatives are not same. That means they are two disjoint sets. Then, after uniting these two sets, S_x and S_y will be removed from S.

$S = \{\{x\}, \{y\}, \{x, y\}\}$.

* Since sets are disjoint, each union operation reduces the no. of sets by 1.
* Then, after (n-1) steps or (n-1) union operations, all the objects will belong to a single set.

Application of Disjoint Sets.

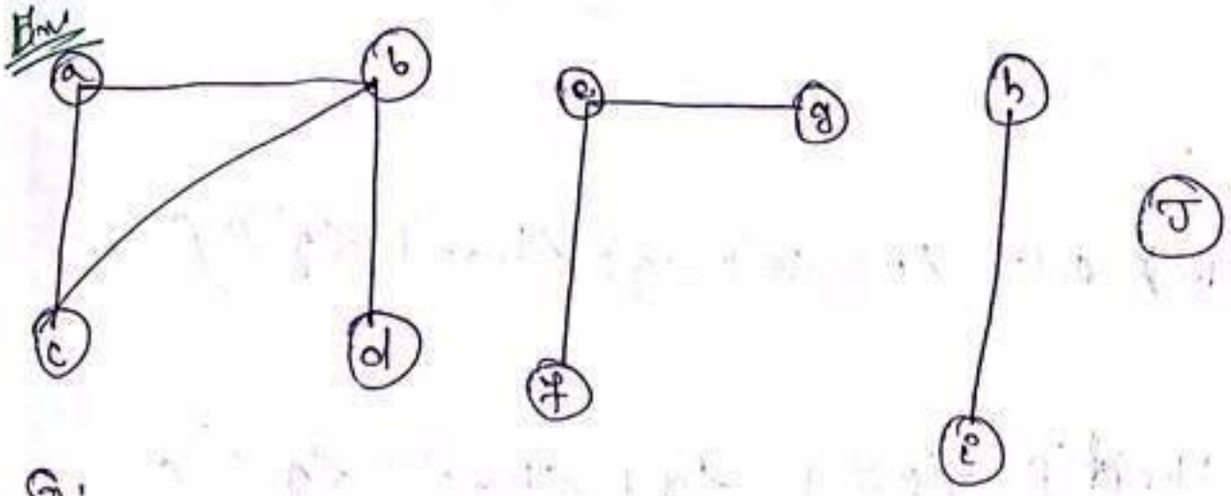
1. Find Connected_Component of an undirected graph.

Algorithm.

```
CONNECTED_COMPONENT(G).
1. for each vertex v ∈ V[G].
2.     do makeset(v).
3. for each edge (u,v) ∈ E[G].
4.     do if Findset(u) ≠ Findset(v)
5.         then UNION(u,v)
6. end.
```

E[G] → set of edges of G.
V[G] → set of vertices of G.

Ex:



Q:

V{a b c d e f g h i j}. E{(a,b) (a,c) (b,c) (b,d) (e,f) (e,g) (h,i) (h,j)}

Step 1:

S = {{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}}

(a,b).

rep(a) = a rep(b) = b.

union {a,b}

S = {{a,b} {c} {d} {e} {f} {g} {h} {i} {j}}.

(a,c).

rep{a,b} = a or b. rep{c} = c.

union {a,b,c}

S = {{a,b,c}, {d}, {e}, {f}, {g} {h} {i} {j}}.

(b,c) do nothing.

(b,d)

S = {{a b c d} {e} {f} {g} {h} {i} {j}}.

Objective of the algorithm:

to find {{a b c d} {e f g} {h i} {j}}.

the disjoint sets.

(h, i)

S = {{a b c d} {e} {f} {g} {h, i} {j}}

(e, f)

S = {{a b c d} {e f} {g} {h, i} {j}}

(e, g)

S = {{a b c d} {e f g} {h i} {j}}

(j) do nothing.

8/10/2015.

Minimum Spanning Tree (MST)

*- The design of electronic circuit having n equipments needs inter-connection of the pins of the equipments. This problem can be mapped to an undirected graph

$G = (V, E)$

where V is set of pins and E is the set of inter-connections among the pair of pins.

*- The objective of such kind of problem is to find an acyclic subset (T)

T ⊆ E (set of inter-connections)

that connects all of the vertices whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

is minimized.

LectureNotes.in

$w(u,v) \rightarrow$ cost of inter-connection of u with v.

(u,v) is an edge and $w(u,v)$ is the cost or weight.

* Since T is acyclic and connects all of the vertices, it is called a spanning tree and the problem of determining tree(T) is called minimum spanning tree problem.

Algorithm to find MST.

Generic_MST(G, w).

1: $A \leftarrow \phi$

2: While A doesnot form a spanning tree.

3: do find edge (u,v) that is safe for A.

4: $A \leftarrow A \cup (u,v)$.

5: return A.

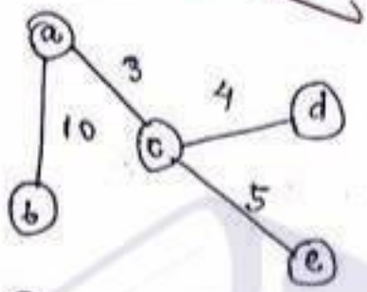
Cut:

A cut of a graph

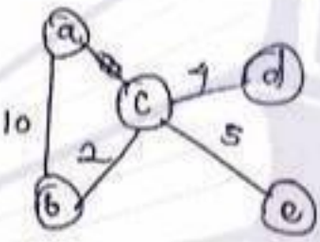
$G = (V, E)$, $(S, V-S)$

12/10/2015.

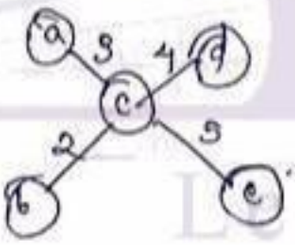
Spanning Tree. (min^{m} no. of edges which connects all the vertices of a graph).



$T(c) = 10+3+4+5$
$= 22$.



$T(c) = 10+1+5+2$
$= 21$



$T(c) = 3+4+3+2$.
$= 14$

* - Spanning tree with minimum cost will be known as minimum spanning tree.

Sum of the weight of edges of a tree is known as cost of the tree.

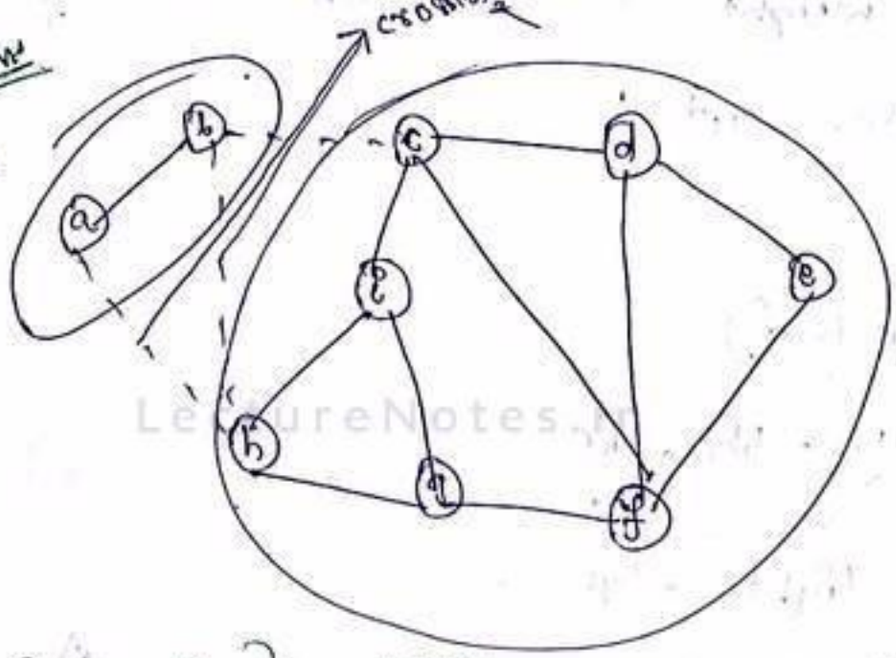
* Cut:

A cut $(S, V-S)$ of an undirected graph

$G = (V, E)$

is a partition of V.

Eg:



Crossing Edge. (edge which connects the two cuts).

$(u,v) \in E$ crosses the cut $(S, V-S)$ if one of its end point is in S and another is in $V-S$.

*- A cut respects a set (A) of edges if no the edges of in A crosses the cut.

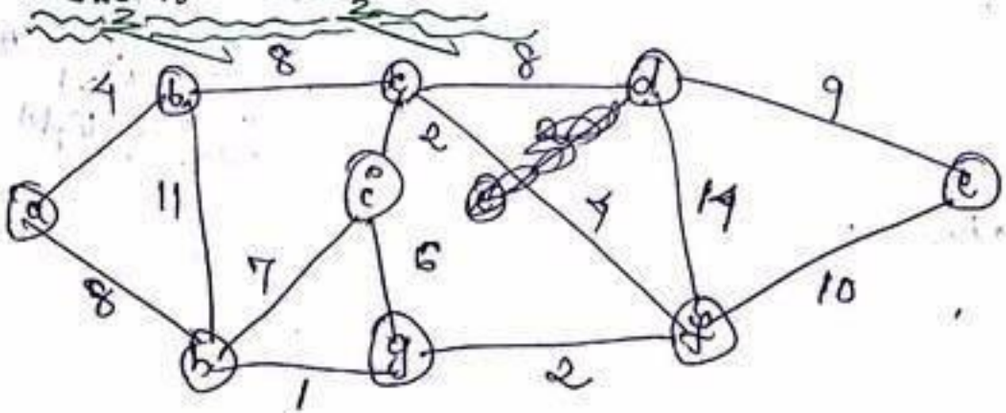
if one in S and other in V-S (crossing)

Eg: $A = \{(a,b)\}$ respects $S = \{a, b\}$.

$S = \{a, b\}$ respects $A = \{(h,g), (i,g)\}$.

$S = \{a, b\}$ not respects $A = \{(a,b)(a,h)(i,g)\}$.

Light Edge



A edge is a light edge if it crosses a cut and its weight is minimum of any edge crossing the cut.

$S = \langle a, b \rangle$

$V - S = \langle c, d, e, f, g, h \rangle$

crossing edge = $\cdot ah, bh, bc$

light edge.

Kruskal's Algorithm. (uses greedy approach).

Theorem 23.1.

Let $G = (V, E)$ be a connected, undirected graph with real valued weight functions (w) defined on E.

Let A be a subset of the tree that is included in some minimum spanning tree of G.

Let $(S, V-S)$, is the cut that represents respects A and let (u, v) be a light edge, crossing $(S, V-S)$, then edge (u, v) is the safe edge for A.

cut respects A and (u,v) is the light edge.

Algorithm.

MST-Kruskal (G, w).

$A \leftarrow \phi$

Step-1: $A = \{\}$

$S = \{\{a\}\{b\}\{c\}\{d\}\{e\}\{f\}\{g\}\{h\}\{i\}\}$

Step-2:

$E = \{(hg), (gf), (i,c), (c,f), (ab), (ig), (cd), (ih), (bc), (ah), (de), (ef), (bi), (df)\}$

$A = \{(h,g)$ (find_set(h) ≠ find_set(g)

$S = \{\{a\}\{b\}\{c\}\{d\}\{e\}\{f\}\{hg\}\{i\}\}$

$A = \{(hg), (g,f)$

$S = \{\{a\}\{b\}\{c\}\{d\}\{e\}\{fgh\}\{i\}\}$

$A = \{(hg), (gf), (ic)$

$S = \{\{a\}\{b\}\{d\}\{e\}\{fgh\}\{ic\}\}$

$A = \{(hg)\ (gf)\ (ic)\ (cf)$

$S = \{\{a\}\{b\}\{d\}\{e\}\{icfgh\}\}$

$A = \{(hg)\ (gf)\ (ic)\ (cf)\ (ab)$

$S = \{\{ab\}\{d\}\{e\}\{icfgh\}\}$

$c^{\circ}g$ (root)

$A = \{(hg)\ (gf)\ (ic)\ (cf)\ (ab)\ (cd)$

$S = \{\{ab\}\{e\}\{icfghd\}\}$

$c^{\circ}h$ (root)

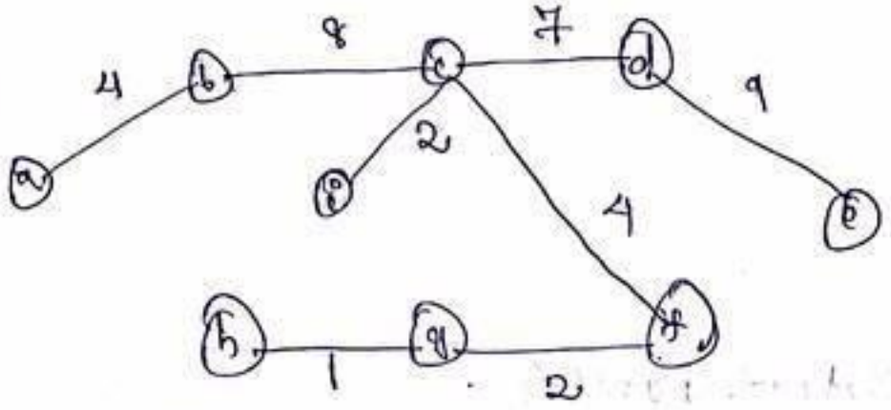
$A = \{(hg)\ (gf)\ (ic)\ (cf)\ (ab)\ (cd)\ (bc)$

$S = \{\{abicfghd\}\{e\}\}$

ah (root)

$A = \{(hg)\ (gf)\ (ic)\ (cf)\ (ab)\ (cd)\ (bc)\ (de)$

$E = \{(a,b), (b,c), (c,d), (d,e), (e,f), (f,g), (g,h), (h,a), (b,h), (c,f)\}$



$T(c) = 4+8+2+7+9+4+2+1$

$= 37$

14/10/2015

Time Complexity of Kruskal's Algorithm

Line-1	$O(1)$
Line(2-3)	$O(V)$
Line 4	$O(E \log E)$
Line (5-8)	$O(E)$

$T(n) = O(V) + O(E \lg E) + O(E)$

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

$|E| \leq |V| - 1$

Prim's Algorithm. (use greedy approach)

Algorithm:

MST_Prim's (G, w, r)

1. for each $u \in V[G]$

2. $key[u] \leftarrow \infty$

3.

$\pi[u] \leftarrow NIL$.

4. key[u] ← 0 .

5. Q ← V[G] .

6. while $Q \neq \phi$

7. do u ← Extract_MIN(Q) .

8. for each $v \in Adj[u]$.

9. do if $v \in Q$ and $w(u,v) < key[v]$.

10. then $\pi[v] \leftarrow u$.

11. $key[v] \leftarrow w(u,v)$.

12. end .

G → Graph

w → real valued weight.

r → root of the tree.

V[G] → set of vertices.

key[u] → the min^n weight associated with the vertex (u).

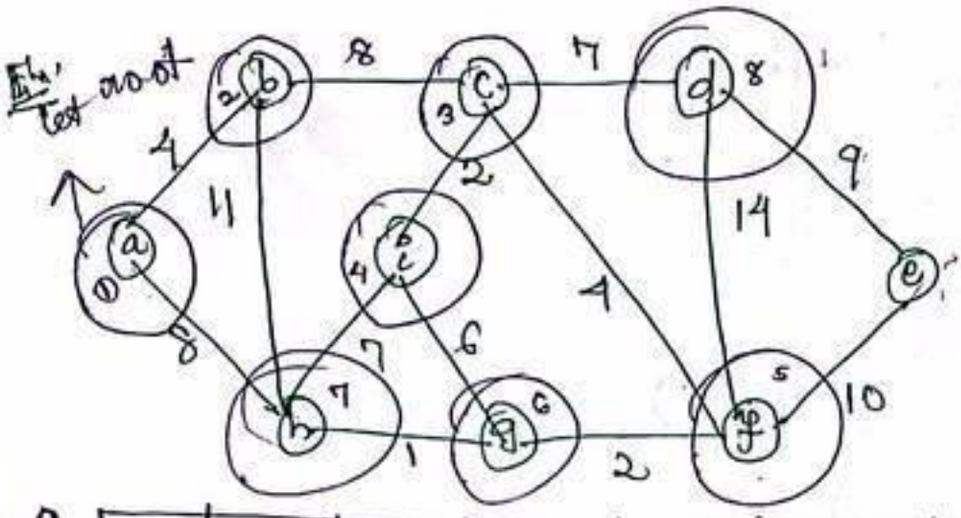
$\pi[u]$ → parent of the node (u).

Q → priority queue. (min-priority queue).

Adj[u] → All adjacent nodes of u.

w(u,v) → weight associated with u,v..

Let root



Q

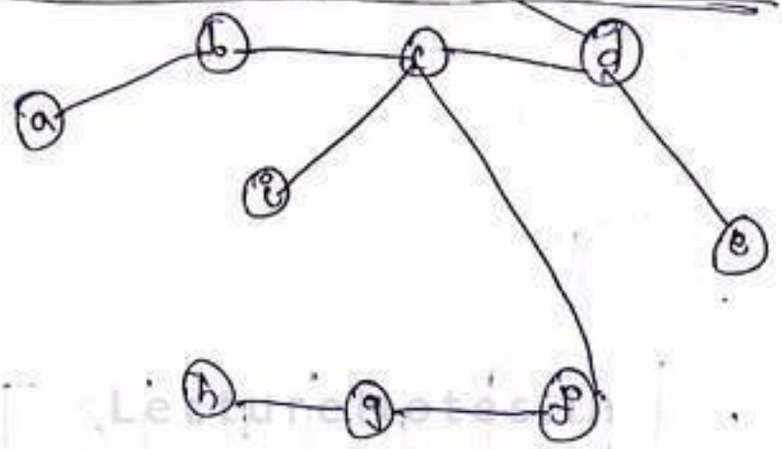
a	b	c	d	e	f	g	h	i

Nodes	Key	π	Adjacent nodes.
a	~~∞~~ 0 ~~x~~	Nil	b, h
b	~~∞~~ ~~4~~ x	~~Nil~~ a	a, c, h.
c	~~∞~~ ~~8~~ x	~~Nil~~ b	b, d, i, f.
d	~~∞~~ ~~7~~ x	~~Nil~~ c	c, f, e
e	~~∞~~ ~~9~~	~~Nil~~ d	d, f.
f	~~∞~~ ~~4~~ x	~~Nil~~ c	g, c, d, e
g	~~∞~~ ~~2~~ x	~~Nil~~ f	h, i, f.
h	~~∞~~ ~~8~~ ~~7~~ ~~1~~ x	~~Nil~~ ~~a~~ i g	a, b, i, g
i	~~∞~~ ~~8~~ x	~~Nil~~ c	c, g, h

1. Extract the node with minimum key value, so extract a.

~~a~~	~~b~~	~~c~~	~~d~~	~~e~~	~~f~~	~~g~~	~~h~~	~~i~~

Minimum Spanning tree :



Time Complexity.

Line (1-3) = $O(V)$.
Line - 4 = $O(1)$
Line - 5 = $O(V \log V)$
Line - 6 = $O(V)$
Line 7 = $O(V \log V)$
Line (8-12) = $O(V-1)$.
= $O(E)$.

Total time
= $O(V) + O(V \log V) + O(V) + O(V \log V) + O(E)$
= $O(V \log V)$.

20/10/2015
29/10/2015

Shortest Path Problem

to find the shortest path between two vertices.

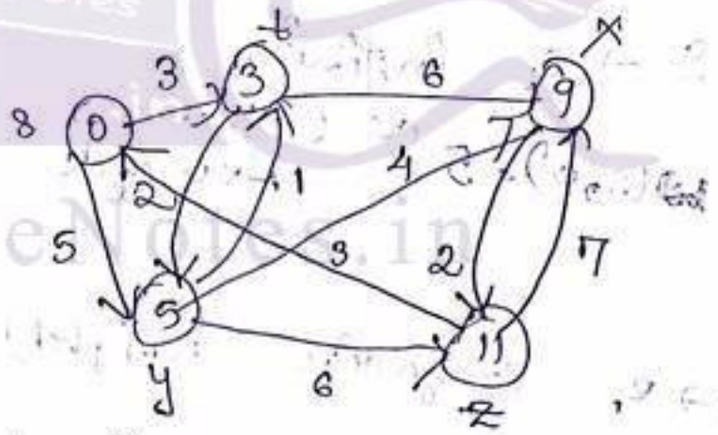
Input : directed graph $G = (V, E)$.
weight function $w : E \to \mathbb{R}$.

$p = \langle v_0, v_1, \ldots, v_k \rangle$ (path)

$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

shortest path weight $\delta(u, v) = \min \{ w(p) : u \overset{p}{\rightsquigarrow} v \text{ if there exists a path from } u \text{ to } v \}$, ∞ otherwise

there might be multiple shortest paths from u to v



single source shortest paths
$G = (V, E)$ => find a shortest path from a given source vertex (s) to each vertex $v \in V$.

Single dest
given vertex (t) from each vertex $v \in V$.

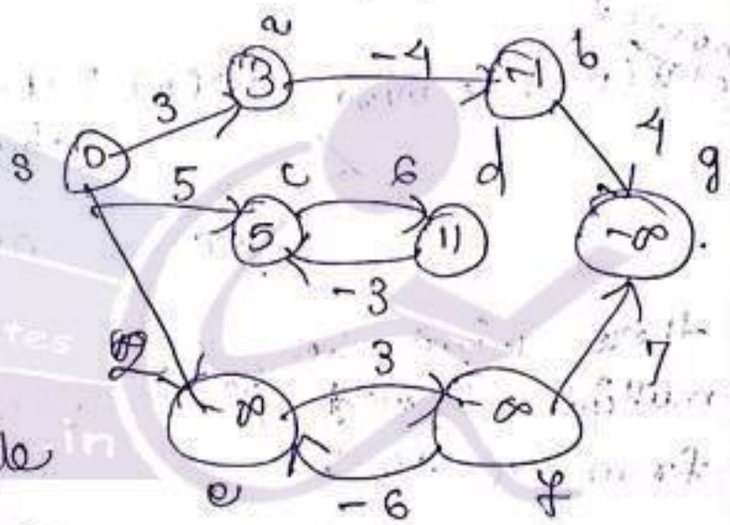
single - pair.

: shortest path from u to v for given vertices u & v.

a shortest path

negative weight edges may for negative weight cycles,

if such cycles are reachable from source, then $\delta(u, v)$ is not properly defined.



$s \to a = 3$

$s \to b = -1$

$s \to c$ infinite as c has

$\delta(s, c) = 5$ ∴ -ve cycle

$s \to e$ (negative weight) $3 + (-6) = -3$

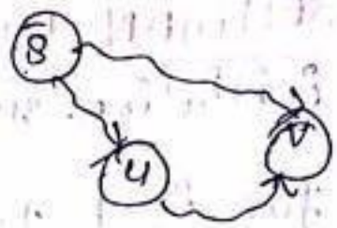
$\delta(s \to e) = -\infty$ (no shortest path)

can find paths from s to e with arbitarily large -ve weights.

Cycles:

Zero - weight:

Triangle Inequality.



For all $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$$

—x—

$d[v]$ = length of shortest path to the vertex (v).

$\pi[v]$ = parent of v.
(predecessor)

4/11/2015.

$\delta(u, v) \longrightarrow$ length of shortest path.

Initialize-single-source(V, s)

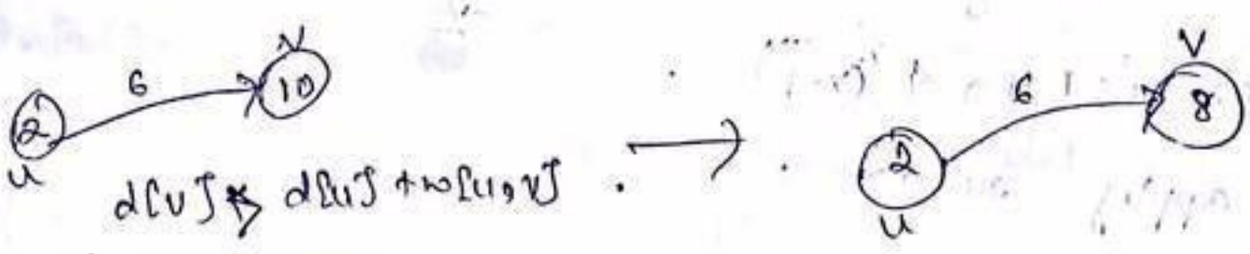
1. for each $v \in V$
2. do $d[v] \leftarrow \infty$
3. $\pi[v] \leftarrow$ NIL.
4. $d[s] \leftarrow 0$.

Relaxation(u, v, w) → to test whether the shortest path can be improved.

if $d[v] > d[u] + w(u, v)$.

⇒ $d[v] = d[u] + w(u, v)$

⇒ $\pi[v] \leftarrow u$.



$d[v] > d[u] + w[u, v]$.

⇒ $10 > 6 + 2$

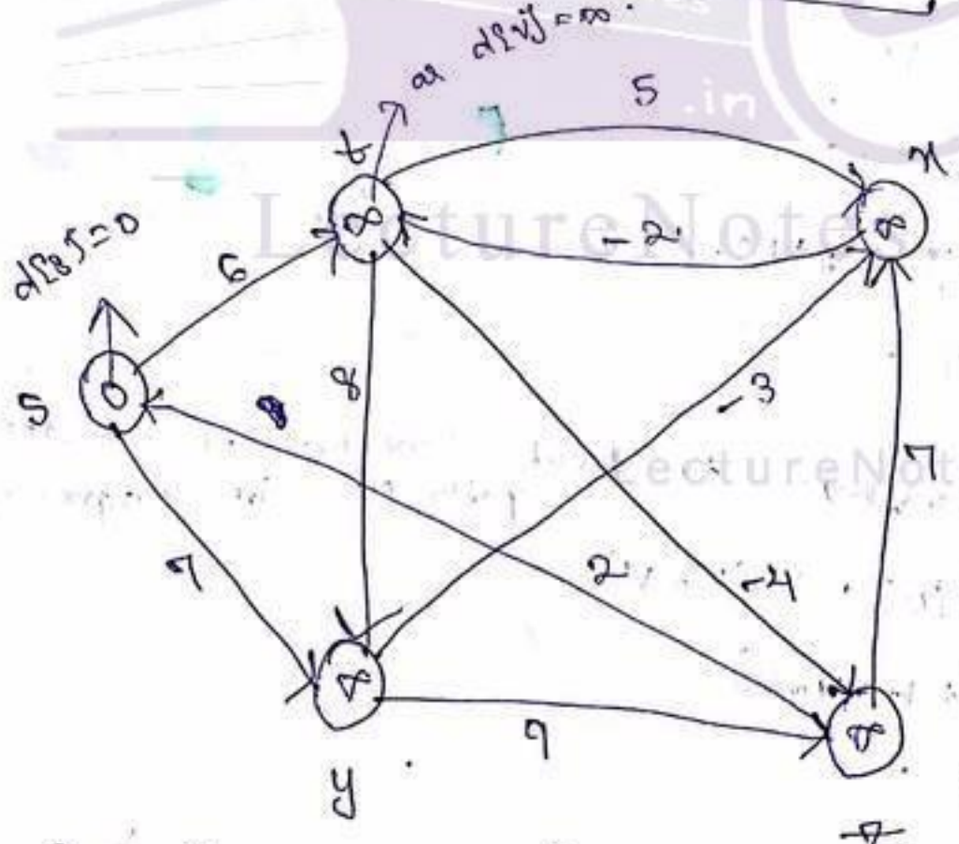
weights can be -ve must not contain -ve cycle.

BELLMAN-FORD (V, E, w, s)

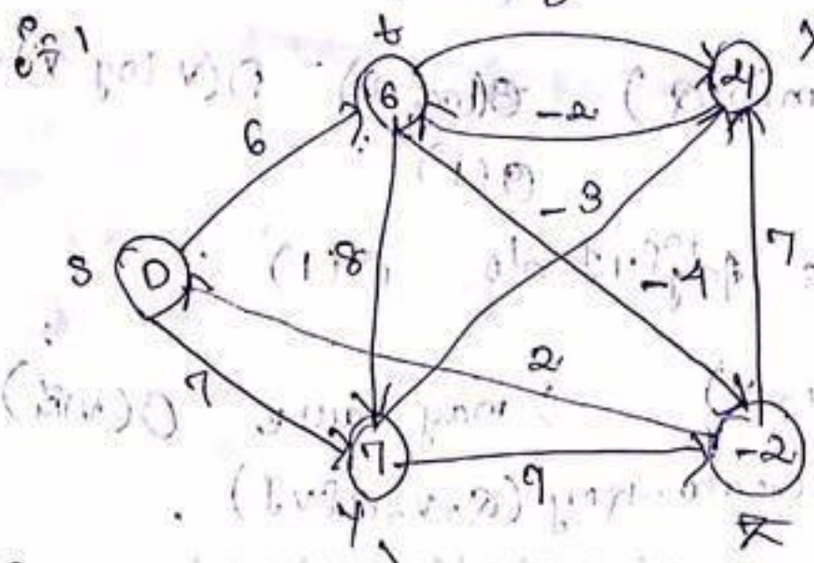
1. initialise-single-source (V, s) $\leftarrow O(V)$
2. for $i \leftarrow 1$ to $|V|-1$ $\leftarrow O(V)$
3. do for each $(u, v) \in E$ $\leftarrow O(E)$ } $O(VE)$
4. do relax (u, v, w)
5. for each edge $(u, v) \in E$ $\leftarrow O(E)$
6. do if $d[v] > d[u] + w(u, v)$
7. then return false
8. return true.

} checks -ve cycle.

$$T(n) = O(V + VE + E) = O(VE)$$



11. for $i = 1$ to $4(V-1)$.
apply relax.



$i=2$ (relax)

no change.

$i=3$ } no change.
$i=4$ }

$\therefore \delta(s,t) = 6$, $\delta(s,y) = 7$, $\delta(s,z) = -9$.

It gives the shortest path by maintaining predecessor matrix.

10/11/2015.

Dijkstra's Algorithm:

find single source shortest path.

non-negative edge weight.

if all edge-weights are equal, then we BFS, otherwise we this algorithm.

Use Q = priority queue keyed on d[v] values

(note: BFS we FIFO Q).

Dijkstra(G, s)

Init(G, s) — $O(V)$.

S ← ϕ — set of discovered nodes. $O(1)$

Q ← V[G] — $O(\log V)$.

4: while $Q \neq \phi$ do $O(V)$

5: $u \leftarrow$ extract-min(Q) $O(\log V)$ $O(V \log V)$.

6: $S \leftarrow S \cup \{u\}$ $O(1)$.

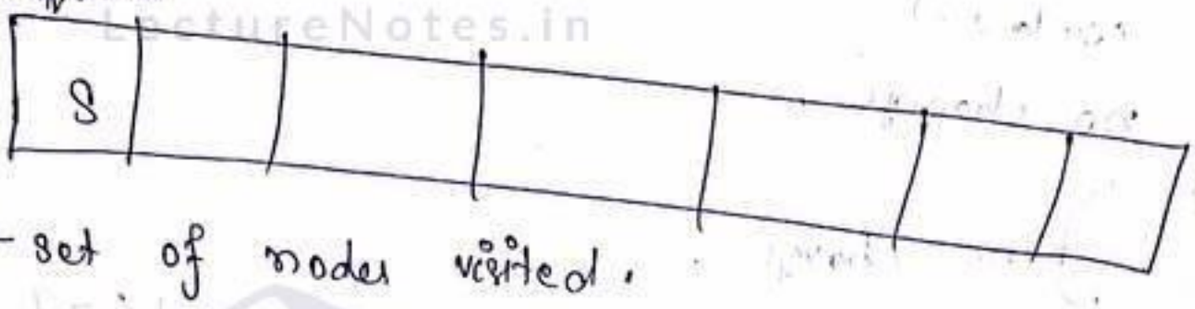
7: for each $v \in Adj[u]$ do $O(1)$

8: Relax(u, v) > may cause $O(VE)$.

> decrease-key(Q, v, d[v]).

Min priority queue

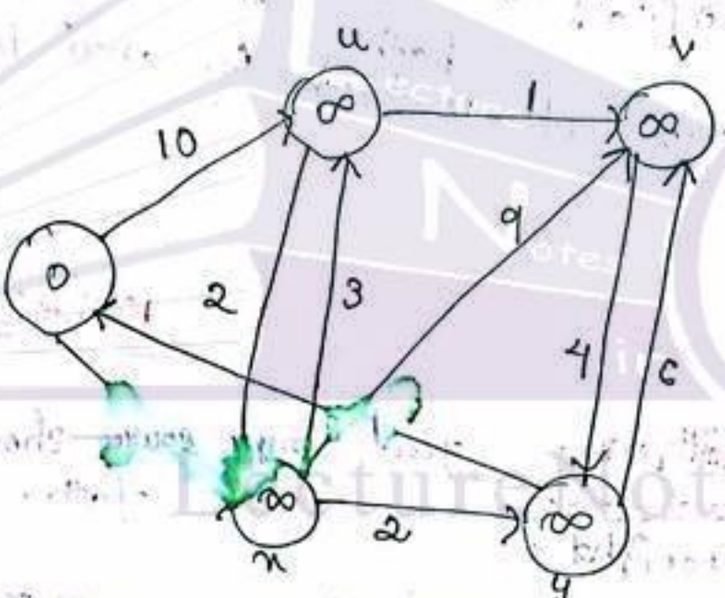
Q



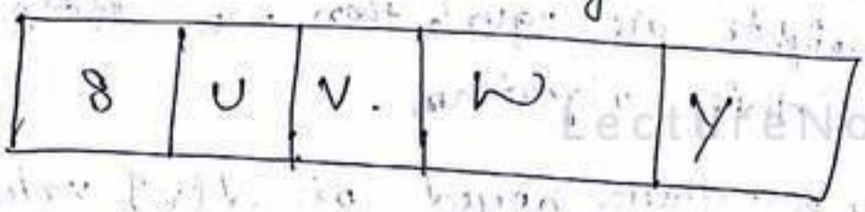
S ← set of nodes visited.

Ex:

1:



3: Q

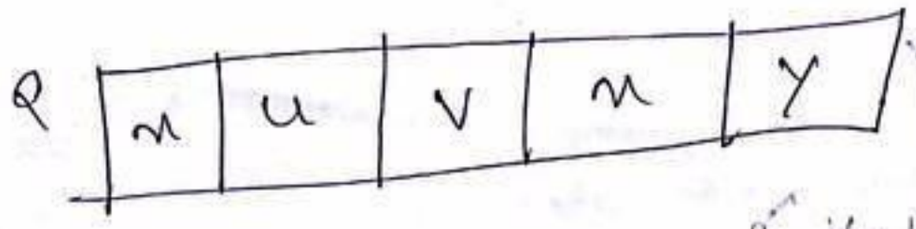


2: $S = \{\phi\}$.

4: $S = \{s, \quad\}$.

5: $Adj = u, x$.

8:



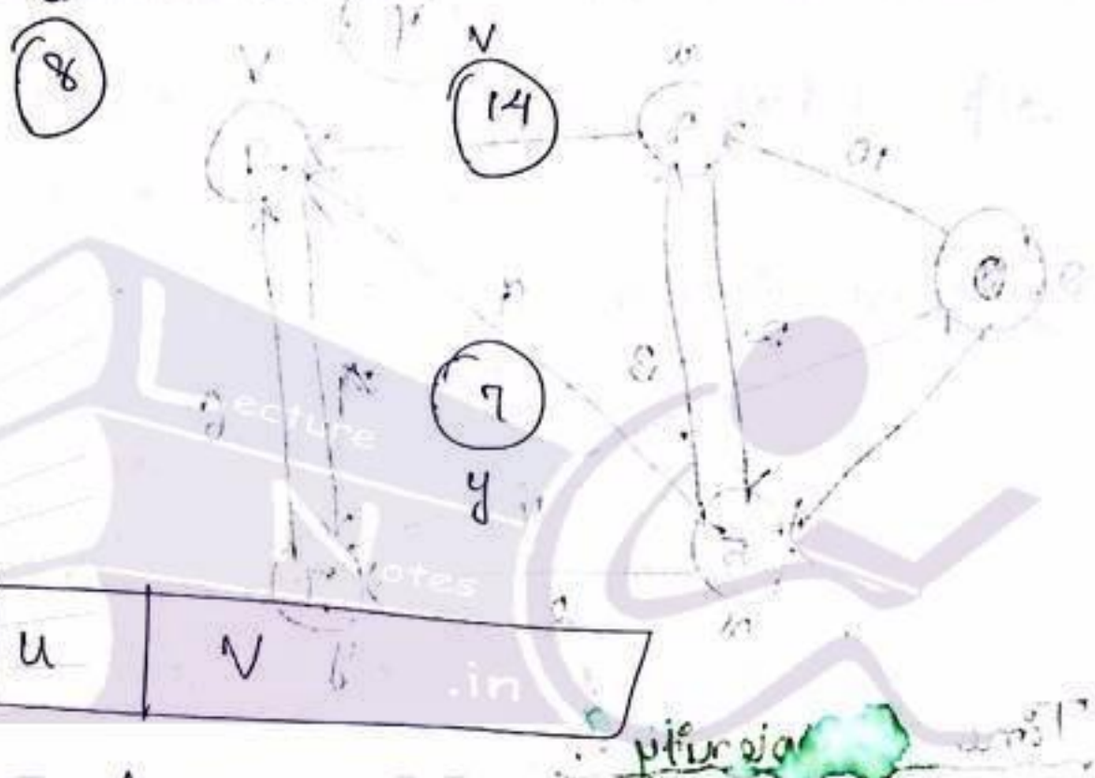
$S = \langle s, x$

$Adj[x] = u, v, y$

relax(u) = 8

v = 14

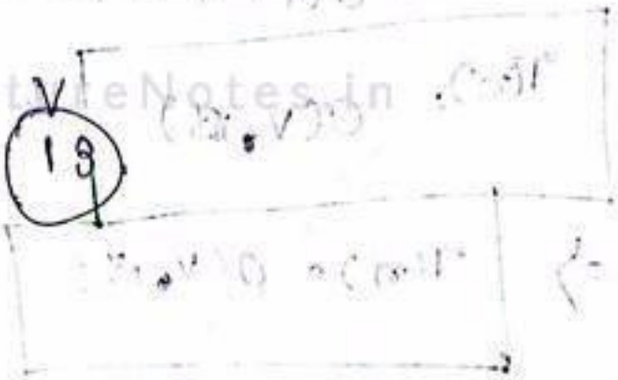
y = 7

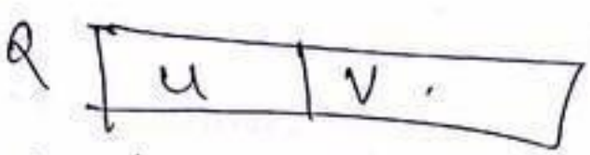


$S = \langle s, x, y \rangle$

$Adj[y] = s, v$

relax(v) = 13





$S = \langle s, x, y, u \rangle$

$Adj[u] = x, v$

relax(x) = 5

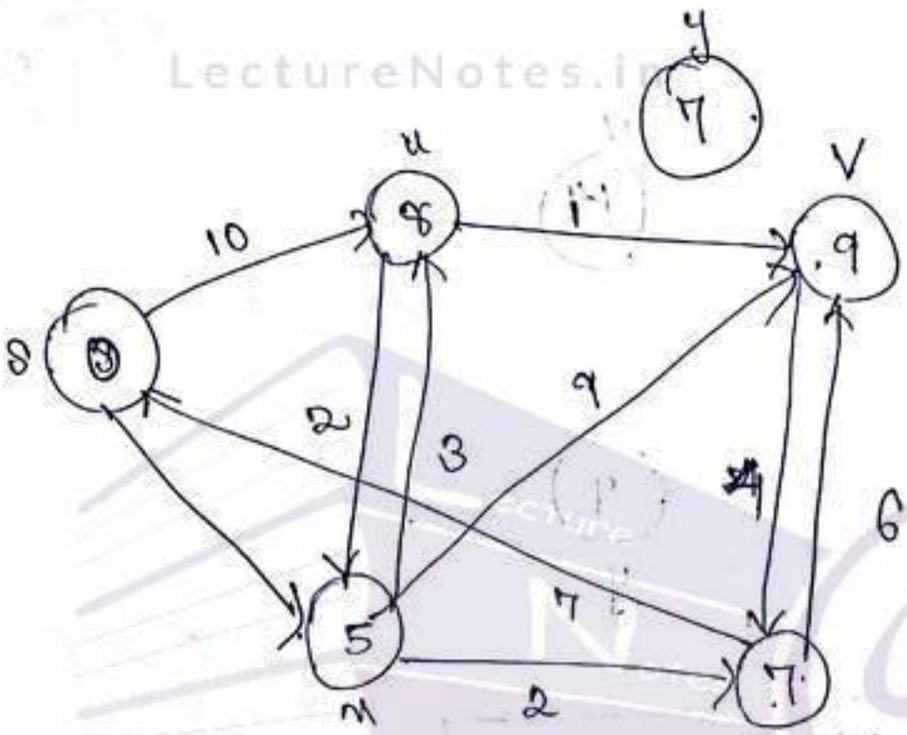
v = 9

Q V

$S = \{s, x, y, u, v\}$

$Adj[v] = y$

relax $(y) = 7$.



Time complexity:

$T(n) = \Theta(V) + \Theta(1) + O(\log V) + O(V) + O(V \log V)$

$+ O(1) + O(1) + O(V.E)$

$T(n) = O(V.E)$

$\Rightarrow$ $T(n) \approx O(V.E)$

Flow Graph.

Flow is the rate that material moves through the network.

source vertex (s) → from where material is produced.

sink vertex (t) → where material is consumed.

Goal : determine max^{m} rate of material flow from source to sink.

For all other vertices : what goes in must go out.

Formal Max Flow Problem.

Graph $G = (V, E)$ → a flow network

* directed, each edge has capacity $c(u, v) \geq 0$
two special vertices s & t.
for any other vertex (v), there is a path
$s \to \cdots \to v \to \cdots \to t$.

Flow → a fun^{n} $f : V \times V \to \mathbb{R}$.

* capacity constraint : for all $u, v \in V$: $f(u, v) \leq c(u, v)$
skew symmetry : for all $u, v \in V$: $f(u, v) = -f(u, v)$
flow conservation : for all $u \in V - \{s, t\}$:

$$\sum_{v \in V} f(u, v) = f(u, V) = 0$$

or,

$$\sum_{v \in V} f(v, u) = f(V, u) = 0.$$

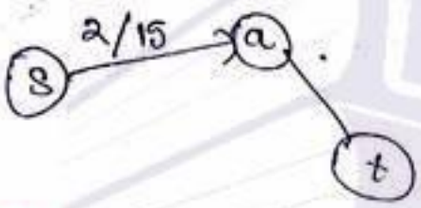
$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$. (flow conservation).

Cancellation of flows -

Avoid the flows in opposite dir^n bet^n the same pair of vertices.

Such flows cancel each other due to skew symmetry

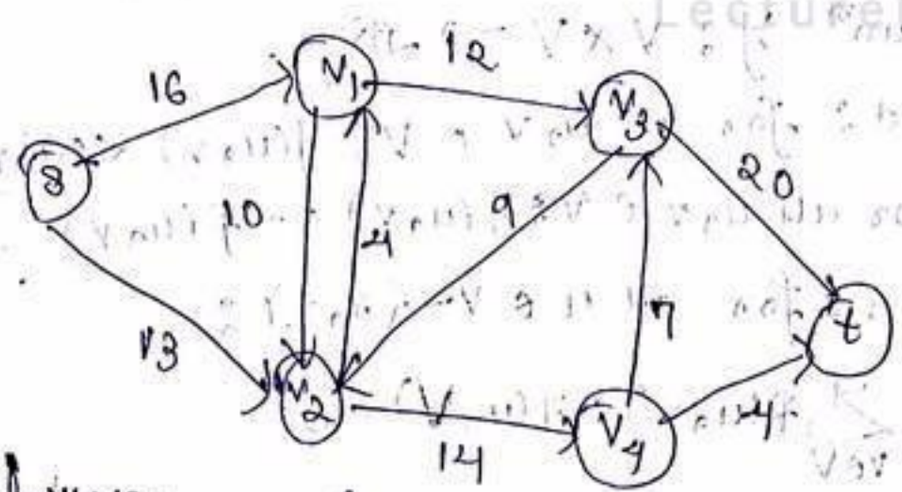
$f(s,a) = 2$ $\quad f(a,s) = -2$.



Max Flow -

Find a flow of max^n value from the source to the sink.

Denoted by $|f|$.



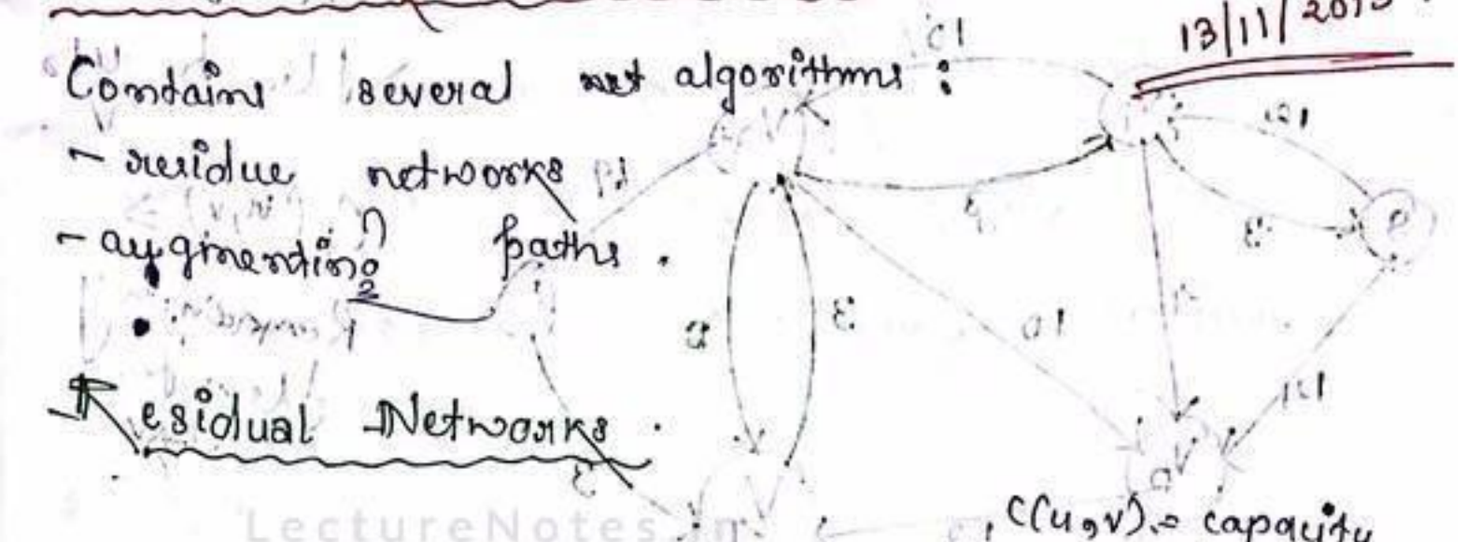
Lucky Duck distribution network.

Ford - Fulkerson Method

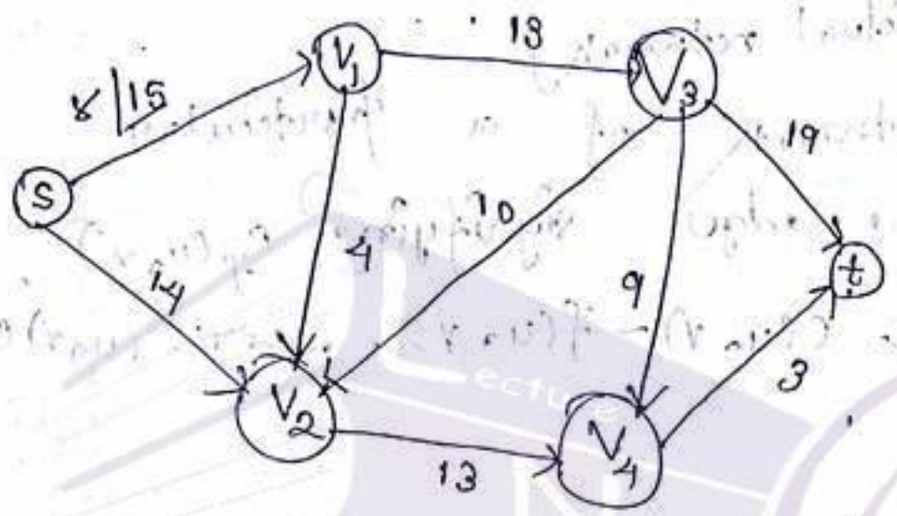
13/11/2015.

Contains several algorithms:
- residue networks
- augmenting paths.

Residual Networks.



C(u,v) = capacity from u to v



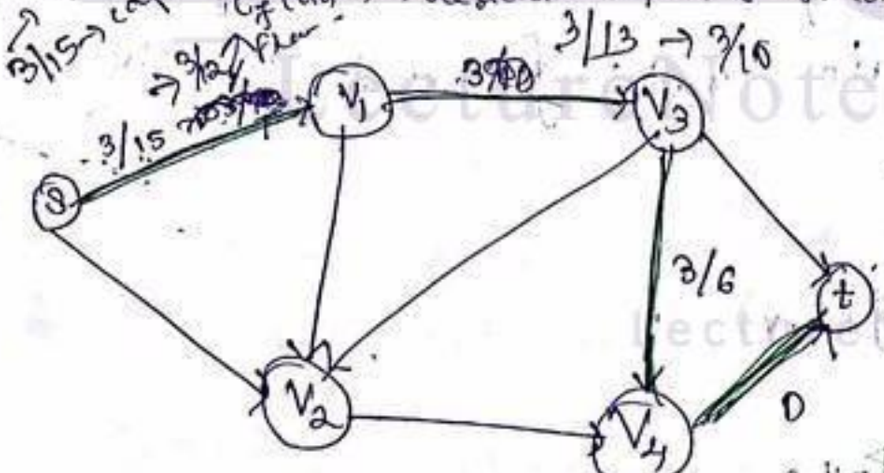
Graph (G). Network

flow → 3/15 → capacity

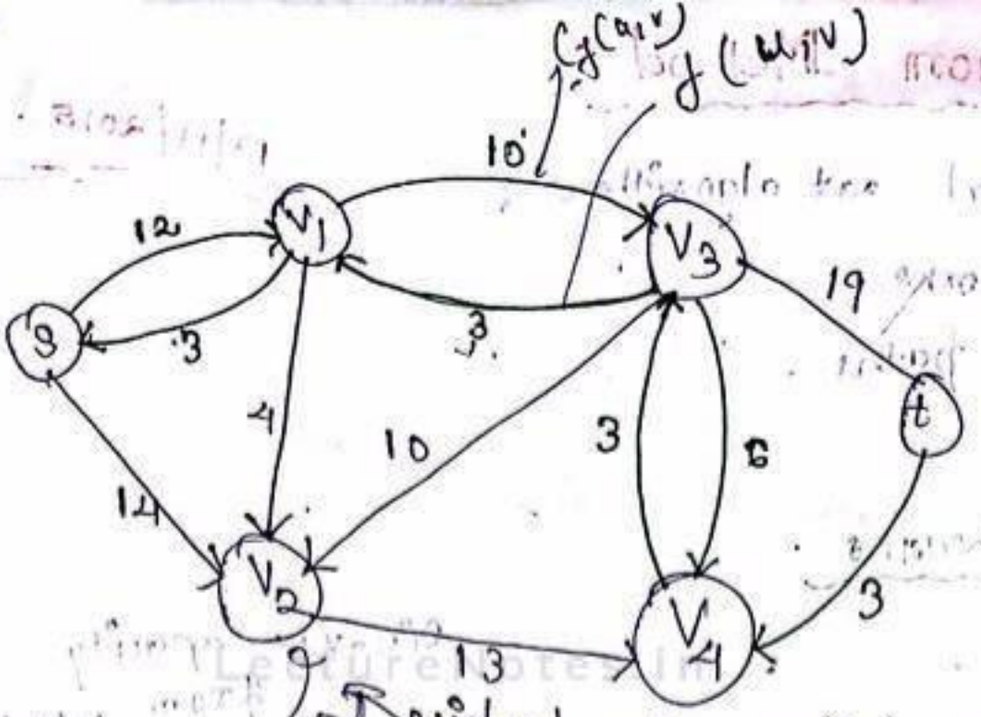
$C_f(u,v)$ → residual capacity of edge (u,v)

flow/capacity

flow/R.C



$$C_f(u,v) = C(u,v) - f(u,v)$$



⇒ flow by u to v.

$c_f(u,v)$ ⇒ Remaining flow from u to v.

Residual network

Residual network of a particular graph contains edges signifying $c_f(u,v)$ i.e., $c_f(u,v) = c(u,v) - f(u,v)$, where $(u,v) \in E$.

Residual graph does not contain an edge with $c_f(u,v) = 0$. It may contain edges which are not present in the original graph.

Here, $c_f(v,u) = f(u,v)$

Augmenting Paths.

The augmenting path of a residual graph decides the amount of flow that will be added to the previous flow to increase the net flow.

* The Ford Fulkerson algorithm terminates when the residual network carries no path from source to destination by returning maximum flow.

* find a path (p) from s to t, such that there is

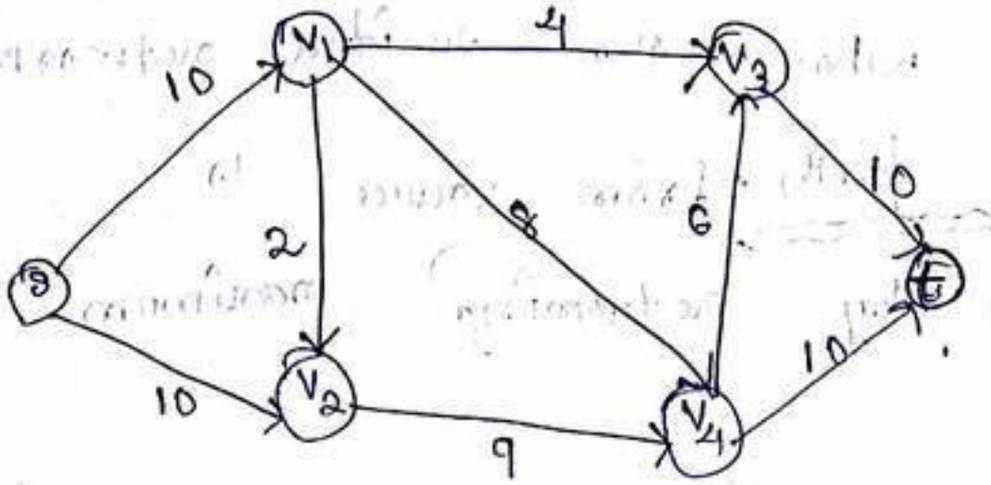
Residual Capacity of a path.

$$C_f(P) = \min \{ C_f(u,v) : (u,v) \in P \}$$

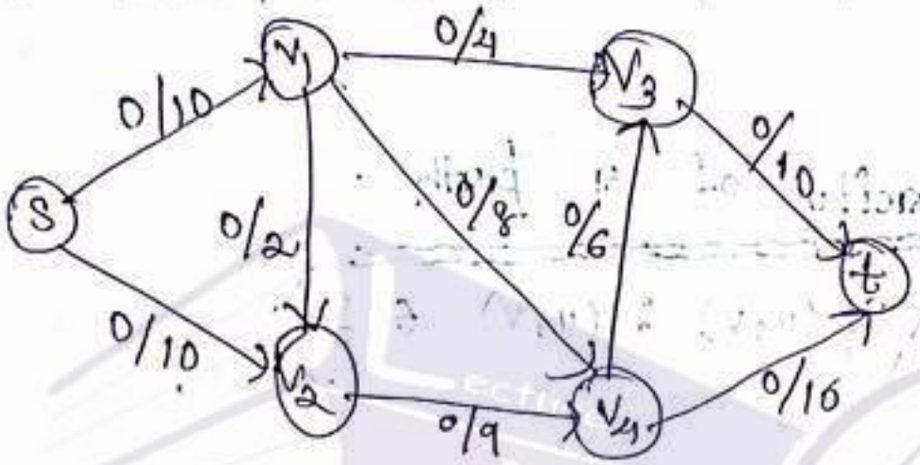
↓
residual capacity of path.

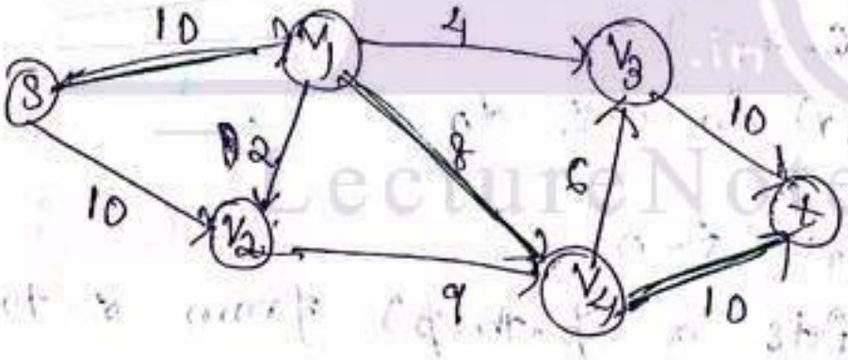
Ford-Fulkerson (G, s, t)

1: for each edge (u,v) in $G.E$ do
2: $f(u,v) \leftarrow f(v,u) \leftarrow 0$
3: while there exists a path (p) from s to t in residual network (G_f) do
4: $c_f = \min\{c_f(u,v) : (u,v)$ is in $p\}$
5: for each edge (u,v) in p do
6: $f(u,v) \leftarrow f(u,v) + c_f$
7: $f(v,u) \leftarrow -f(u,v)$
8: return f.



step 1:





minm no. amount of data from source to destination

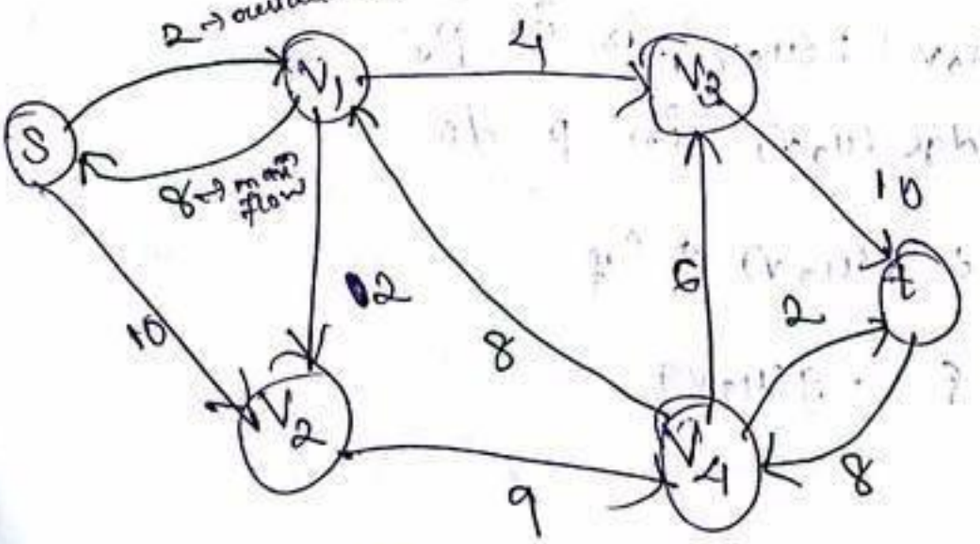
residual capacity of the path

= 8.

$C_f(p) = 8$

2 → residual capacity

8 → max flow



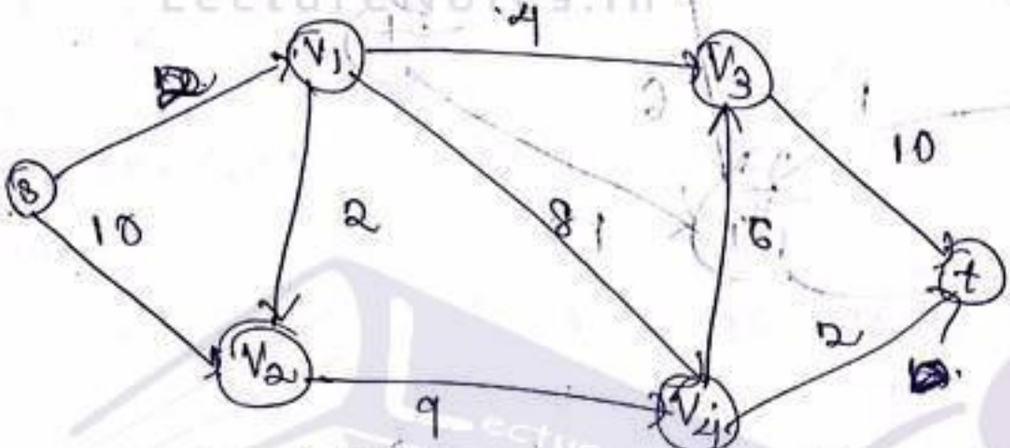
Consider path $s \to v_1 \to v_2 \to v_4 \to t$.

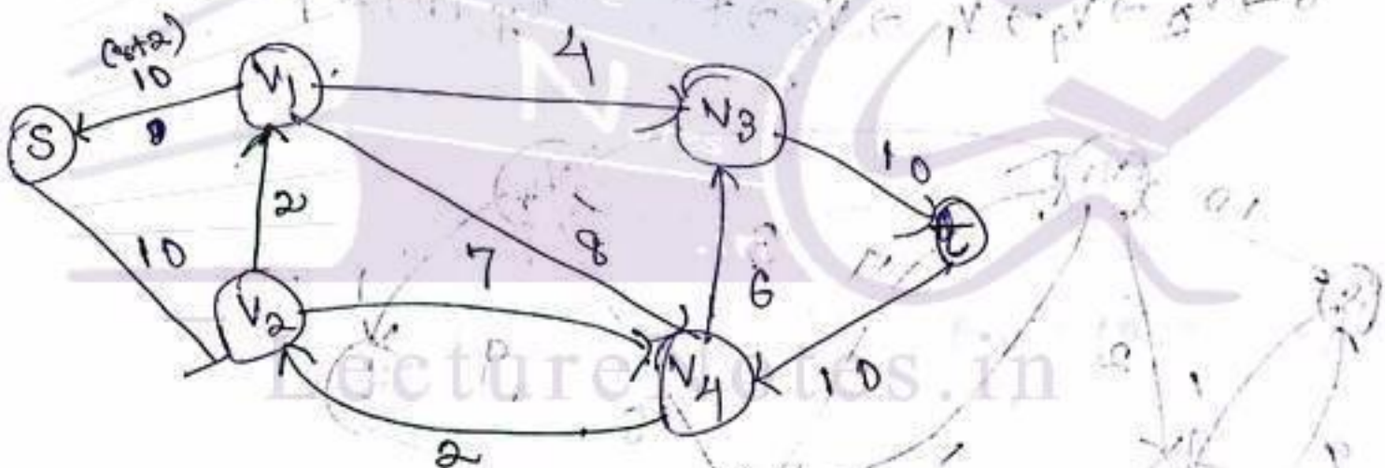
$c_f(p) = 2$.

Flow of the network

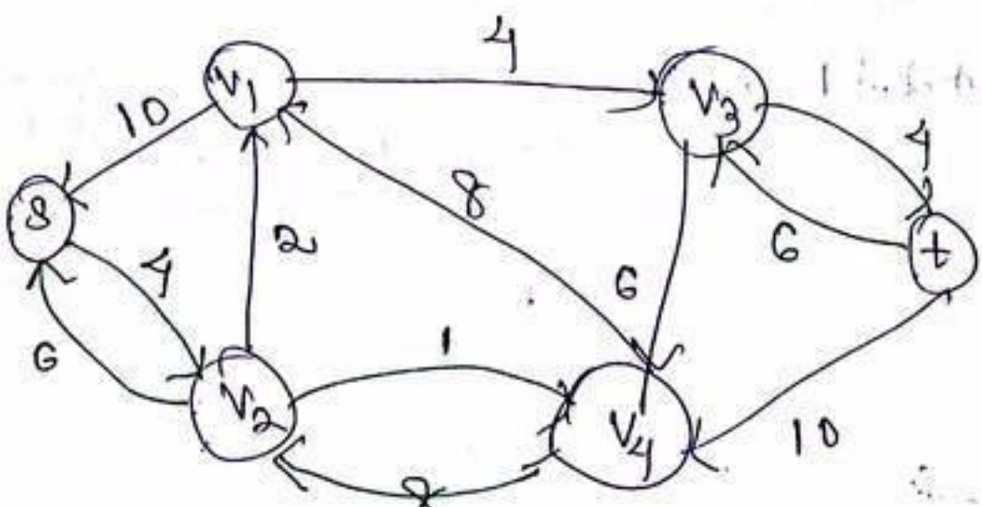
$|f| = \sum$ residual flow of each path.

$|f| = 0+8+2$.



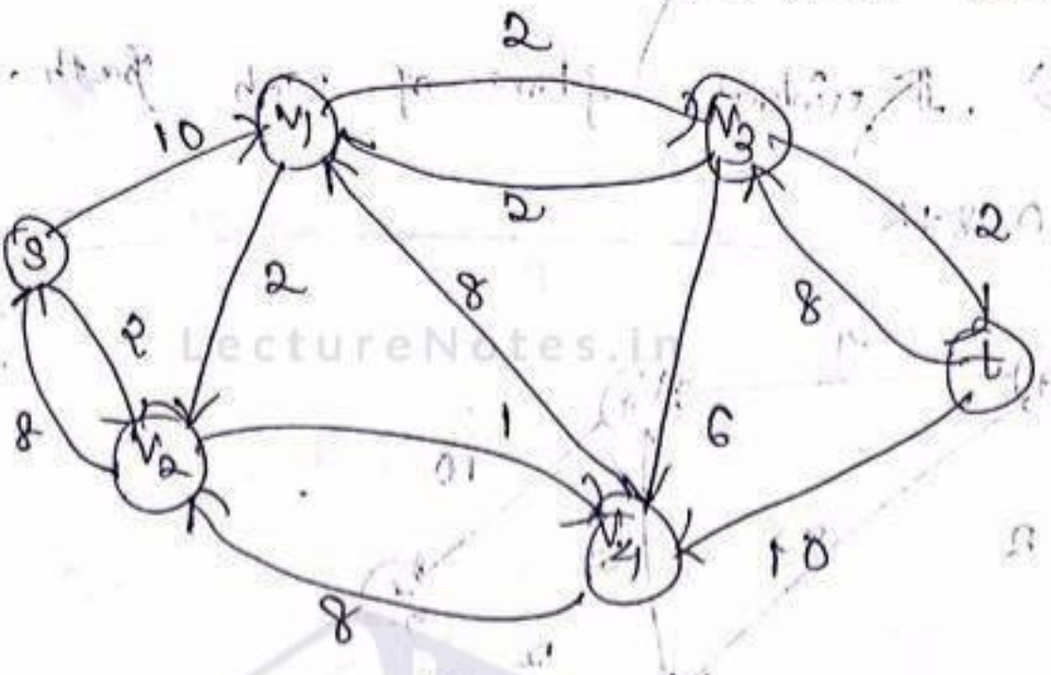


P_3 $s \to v_2 \to v_4 \to v_3 \to t$ $\quad c_f(p) = 6$.

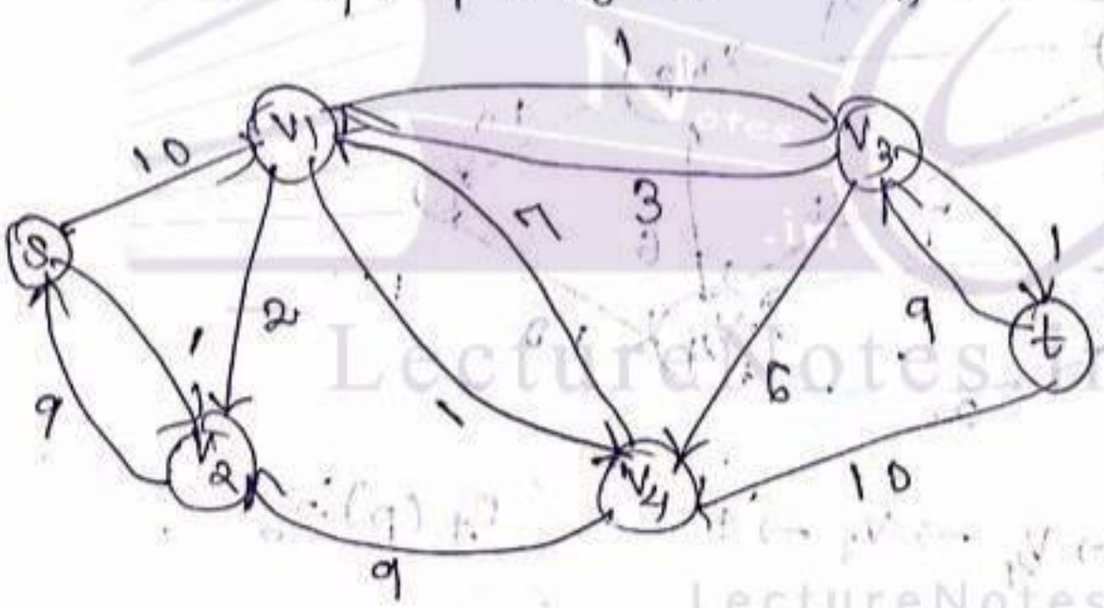


$s \to v_2 \to v_1 \to v_3 \to t$ $\quad c_f(P) = 2$

$|f| = 8 + 2 + 6 + 2$



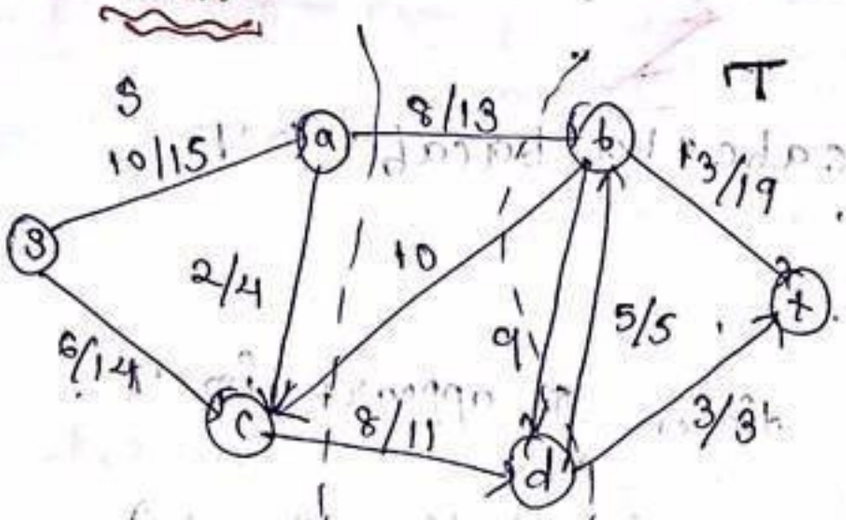
$s \to v_2 \to v_4 \to v_1 \to v_3 \to t$ $\quad c_f(P) = 1$



$|f| = 8 + 2 + 6 + 2 + 1$

$= 19$

Cut.



$S = \{s, a, c\}$

$T = \{b, d, t\}$

net flow of cut = sum of flows from a to b and c to d

$= 13 + 11 + (-10)$

$= (13 + 11) - 10 = 14$

↓ sum of flows from s to T ↓ sum of flows from T to s.

16/11/2015

Capacity of the cut

$= \sum_{u \in S, v \in T} c(u, v)$

(657)

Theorem: Max flow min cut

String Matching

Given string : cabccbacbacab = T[1...n]

Pattern (P) = aac

Find how many times P appears in T?

Eg: Nucleotides : {a t c g}. (DNA sequence)

P = {a a a c g g t}. (motif)

Let t_0 = 1st three ch^rs

= {c a b}

Match t_0 with P:

If not, then shift:

t_1 = {a b c}.

Match with P:

upto (n-m). |T| = n |P| = m.

Naive String matching (T, P)

1: n ← length[T]
2: m ← length[m]
3: for s ← 0 to (n-m)
4: if P[1...m] = = T[s+1...s+m].
5: print("s is a valid shift");

$T(n) = O((n-m) * m)$

$= O(n^2) = O(n * m)$.

Rabin - Karp Algorithm.

Σ = set of alphabets $d = |\Sigma|$.

Base

Ex:- For English alphabets

$d = |\Sigma| = 26$.

0 ~ 9 $\quad d = |\Sigma| = 10$

Assume $\Sigma = \{0, 1, 2, \ldots 9\}$ so that each chr is a decimal digit.

Ex: $P = 3141.5$ (pattern) $\quad$ 64

It's corresponding decimal formula

Horner's formula

$P = P[m] + 10(P[m-1] + 10(P[m-2] + 0.0.$
$+ 10(P[2] + 10(P[1]) \ldots))$.

~~$5 + 10 + 40 + 10 + 30 = 95$~~.

~~$= 5 + 10(1 + 10(10 * 4 + 10 + 30$~~

$= 5 + 10(1 + 10(4 + 10(1 + 10(3))))$

$= 5 + 10(1 + 10(4 + 10(31)))$

$= 5 + 10(1 + 10(314))$

$= 5 + 10(3141)$

$= 5 + 31410$

$= 3141.5$.

$T \rightarrow$ string.

$t_0 = T[m] + 10(T[m-1] + 10(T[m-2] + \ldots 10(T[1])))$.

$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1]$.

modulo q.

p to ... t_{n-m}

$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

q is a prime no. chosen by user.

$h = d^{m-1} \pmod q$

Rabin-Karp matcher(T, P, d, q)

```
1. n ← length[T].
2. m ← length[P].
3. h ← d^(m-1) (mod q)
4. p ← 0
5. t0 ← 0.   (1st m chars of T)
6. for i ← 1 to m   (Preprocessing)
7.     p ← (dp + P[i]) mod q.
8.     t0 ← (dt0 + T[i]) mod q.
9. for s ← 0 to n-m   (matching)
10.    if p = ts
11.        if P[1...m] = T[s+1...s+m]
12.            "Pattern occurs with shift(s)"
13.    if s < (n-m)
14.        ts+1 ← (d(ts - T[s+1]h) + T[s+m+1]) mod q.
```

$T(n) = O(n) + O(m(v + n/q))$

Ex:

$p = 31415$

$p_0 = 0$

$t_0 = 0$

$d = 10$

$q = 11$

i=1

$p \leftarrow 3 \% 11 = 3$

$t_0 \leftarrow 2 \% 11 = 2$

p 31415

t_0 2390

i=2

$p \leftarrow 3 * 1 + 1 = 31$

$t_0 \leftarrow 20 + 3 = 23$

i=3

$p \leftarrow 31 + 4 \rightarrow 314$

$q \leftarrow 23 + 5 \rightarrow 235$

i=4

$p \leftarrow 314 + 1 \rightarrow 3141$

$q \leftarrow 230 + 9 = 2359$

i=5

$p = 3141 \times 10 + 5 = 31415$

$q = 23590 + 0 = 23590$.

If $n \rightarrow$ polynomial

⇒ Degree bound = $(n+1)$.

Degree → Highest power of the polynomial.

Ex: $P(x) = 3x^3 + 2x^2 + 7$.

⇒ Degree = 3.

Degree bound = 4.

$O(n^2)$ → Evaluation of polynomial.

FFT $O(n \log n)$.

↓

to evaluate value of a polynomial.

T = 3 1 5 4 7 2 9 1 8 (1 2 3) P = 1 7 3

t_0 = 315 ; t_1 = 154. p = 173

m = 3 s = 2

3 . . . 5

m = 3 p = P[3] + 10 (P[2] + 10 (

= 3 + 10 (7 + 10 x 1)

0 to n-m = 173

n-m X m $\Theta(m)$