1/01/19

Unit - I

Introduction

- Why do we need to study algorithms?
  → Theoretical stand point - Algorithms → corner stone of CS.
  → Practical stand point.

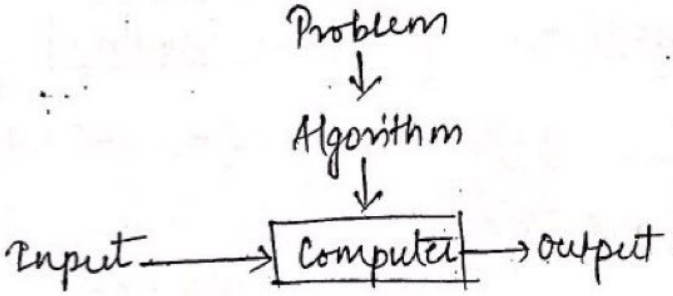
Study of algorithms is called algorithmics.

→ Standard set of algorithms
new algorithm and analyse
* time complexity.
* space complexity.

ALGORITHM: It is a sequence of unambiguous instruction for solving a problem i.e. for obtaining a required o/p for any legitimate i/p in finite amount of time.

Properties of algorithms:

1) Finiteness - Terminates after finite no. of steps.
2) Definiteness - Each step must be unambiguously specified.
3) Input - Valid i/p must be clearly specified.
4) Output - The data that results upon completion must be specified.
5) Effectiveness - Steps must be sufficiently simple & basic.



Scanned with CamScanner

- Finding Gcd of two numbers using Euclid Algorithm:

Step 1: If $n=0$, return m as answer and stop, otherwise proceed to step 2.

Step 2: Divide m by n and assign the value of remainder to r.

Step 3: Assign the value of ~~n~~ to m and r to n, goto Step 1.

Eg: gcd(48, 18)

Here, $m=48$ and $n=18$.

$$\begin{array}{r} 18)48(2 \\ \underline{36} \\ 12 \end{array}$$

i.e, $r=12$, $m=18$ and $n=12$.

Again check $n=0$, not zero

$$\begin{array}{r} 12)18(1 \\ \underline{-12} \\ 6 \end{array}$$

$\therefore r=6$, $m=12$ and $n=6$.

Again, $$\begin{array}{r} 6)12(2 \\ \underline{-12} \\ r=0. \end{array}$$

Now $m=6$ and $n=0$.

$\therefore$ gcd(48, 18) = 6.

$\therefore$ gcd(48, 18) = gcd(18, 12)
= gcd(12, 6)
= gcd(6, 0).
↓
gcd.
return 6.

$$\boxed{gcd(m,n) = gcd(n, m \bmod n)}$$

- ALGORITHM {pseudo code}
(Formal algorithm.)

[· Initialization :
while (n ≠ 0
r ← m mod n
m ← n
n ← r
return m.]

Scanned with CamScanner

Algorithm Euclid(m, n)
// Compute gcd(m, n) by Euclid method.
// Input: two non-negative, not both zero integers m and n.
// Output: greatest common divisor of m and n.

```
while(n≠0) do
    r ← m mod n
    m ← n
    n ← r
return m.
```

Eg: gcd(78, 57)

Here, m=78 and n=57.

```
57)78(1
  -57
  r=21
```

, m=57 and n=21.

Now again,

```
21)57(2
  42
  r=15
```

, m=21 and n=15

Again,

```
15)21(1
  15
  r=6
```

, m=15 and n=6.

Again,

```
6)15(2
 -12
 r=3
```

, m=6 and n=3.

Now.

```
3)6(2
 -6
 r=0
```

, m=3 and n=0.

∴ gcd(78, 57) = 3.

or,

gcd(78, 57) = gcd(57, 21)
= gcd(21, 15)
= gcd(15, 6)
= gcd(6, 3)
= gcd(3, 0).
↑ gcd.
return 3.

Scanned with CamScanner

- Consecutive integer checking to compute gcd(m,n).

Step 1: Assign the value of minimum {m,n} to t.

Step 2: Divide m by t. If the remainder of this division is zero.
Goto Step 3, otherwise goto Step 4.

Step 3: Divide n by t. If the remainder of this division is zero, return the value of t as the answer and Stop; otherwise proceed to step 4.

Step 4: Decrease the value of t by 1. Goto Step 2.

- Middle school procedure for computing gcd(m,n).

Eg: gcd(56,21)

Find prime factors of both m, n.

$56 = 2 \times 2 \times 2 \times 7$.

$21 = 3 \times 7$.

$\therefore gcd(56,21) = 7$.

2 | 56
2 | 28
2 | 14
| 7

Eg: gcd(60,24)

$60 = 2 \times 2 \times 3 \times 5$

$24 = 2 \times 2 \times 2 \times 3$.

$\therefore gcd(60,24) = 2 \times 2 \times 3 = 12$

2 | 60
2 | 30
3 | 15
| 5

2 | 24
2 | 12
2 | 6
| 3

Step 1: Find the prime factors of m.

Step 2: Find the prime factors of n.

Step 3: Identity all the common factors in the two prime expansions, found in Step 1 and Step 2.

[If p is a common factor occuring p_m and p_n times in m and n respectively it should be repeated min $\{p_m, p_n\}$ times.]

Step 4: Compute the product of all the common factors and return it as the gcd of given numbers.

Scanned with CamScanner

1> WAP to check whether the given number is prime or not.
2> WAP to generate prime number upto n.

①
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int n, i, flag=0;
        printf("Enter a positive no: \n");
        scanf("%d", &n);
        for(i=2; i<=n/2; i++)
        {
            if(n%i==0)
            {
                flag=1;
                break;
            }
        }
        if(flag==0)
            printf("The number is prime\n");
        else
            printf("The number is not prime\n");
        return 0;
}
```

②
```
int main()
{
    int i, j, m, n, c;
    printf("Enter value of n:\n");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        for(j=2; j<=sqrt(i); j++)
        {
            if(i%j==0)
                c++;
        }
        if(c>=1)
        {
            printf("Not a prime no.");
        }
```


Scanned with CamScanner

```
        else
            printf("Prime numbers:");
        }
    }
}
```

- Sieve of Eratosthenes:

n = 18.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
[2]	3		5		7		9		11		13		15		17	
2	[3]		5		7				11		13				17	

∴ prime numbers are 2 3 5 7 11 13 17, upto 18.

Eg: To check for 50.

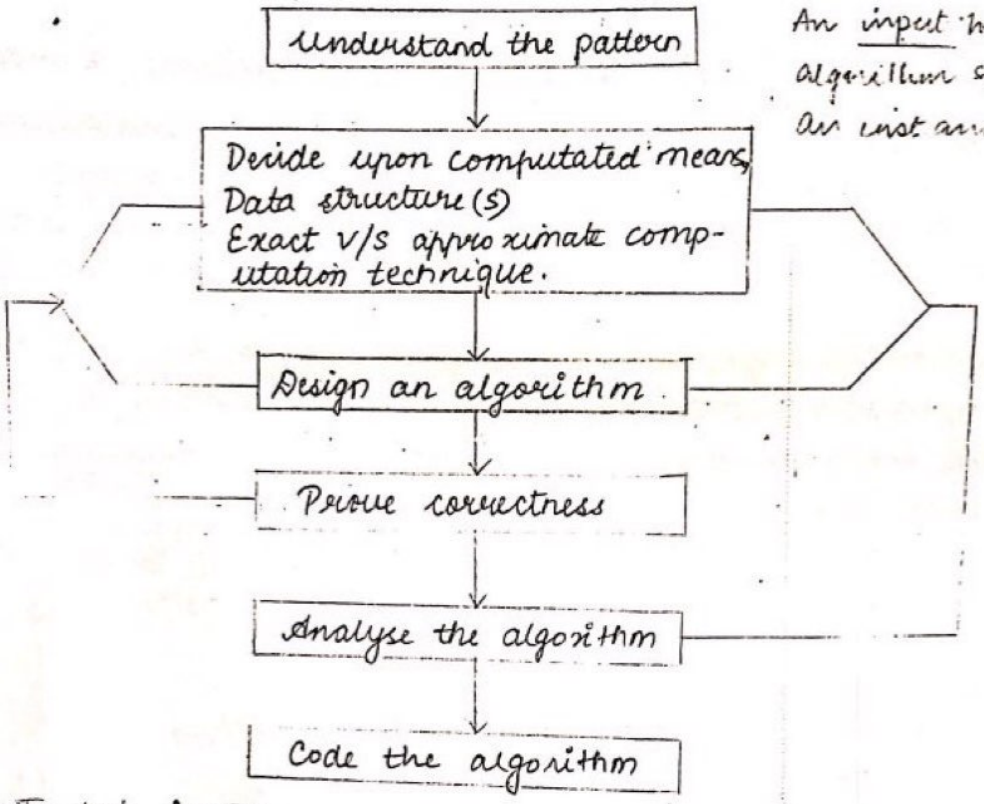
- ALGORITHM-

```
Algorithm Seive(n)
// Input: an integer n>=2
// Output: array L of all prime numbers less than or
            equal to n.
for p<-2 to n do
    A[p]<-p
for p<-2 to [√n] do
    if A[p]≠0
        j<-p*p
    while (j<=n) do
        A[j]<-0
        j<-j+p
// Copy non-zero elements of A to array L.
i<-0
for p<-2 to n do
    if A[p]≠0
        L[i]<-A[p]
        i<-i+1
return L
```


Scanned with CamScanner

3/01/19

- Fundamental of algorithmic problem solving



An input to an algorithm specifies an instance.

Eg: To check for 50.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50.

[2] 3 5 7 9 11 13 15 17 19 21 23 25 27
29 31 33 35 37 39 41 43 45 47 49
2 [3] 5 7 11 13 17 19 23 25 29 31 35
37 41 43 47 49
2 3 [5] 7 11 13 17 19 23 29 31 37 41
43 47 49.
2 3 5 [7] 11 13 17 19 23 29 31 37 41 43 47.

∴ Prime numbers are 2 3 5 7 11 13 17 19 23 29 31
37 41 43 47.

Scanned with CamScanner

04/01/19

Understand problems.

* Solve problem by hand.
* Legitimate input range.
* Ascertaining the capability of computational device.

Von neumann machine: Random access machine.
Sequential algorithm.
[instruction stored sequentially & executed sequentially.]

In recent days, parallel algorithms is used depending upon computational device, also the nature of m/c.

* we need to chose b/w exact v/s approximate algorithms:

[In approximate result, error shouldn't be ≥ 0.001 or > 0.0001]

* Deciding on the data structures.

Algorithm + Data Structure = Program.
It also depends upon - Analysing its performance.

* Algorithm Design Technique: We should chose proper algorithm for the program.

* Method for specifying algorithm

natural language / pseudo code.

PSEUDO CODE - It's a mixture of natural language and programming language constraints / constructs.

* Prove algorithmic correctness: It should yeild correctness in a finite amount of time with legitimate input.

[we use mathematical induction to prove the correctness.]

Scanned with CamScanner

* Analyse the algorithm: Two types of algorithm efficiency
  - Time complexcity.
  - Space complexity.

- Time efficiency/complexity indicates how fast the algorithm runs.
- Space efficiency indicates how much extra space (memory) the algorithm needs.
- Algorithm should be simple enough for everyone to understand - Simplicity.
- Generality i.e it should be general not for some specific input [for only particular value].

* Code the algorithm.

Scanned with CamScanner

05/01/19

Analysing efficiency of an algorithm -

* Measuring input size:

To investigate an algorithm's efficiency as a function of some parameter is quite of the value 'n' indicating the algorithm's input size.

$P(x) = a_n x^n + \ldots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

Time of execution depends upon the number / size of the input.

polynomial - degree.
matrix - order.

* How to measure time:

Running time -

Let Cop be execution time of algorithms, basic operation on a particular computer.

Let C(n) be the number of times this operation needs to be executed for this algorithm, then

T(n) be the running time can be estimated by,

$$\boxed{T(n) \approx C_{op} C(n)}$$

'Cop' is an approximate & C(n) doesn't contain any information about operations, that are not basic.

∴ This formula can be give a reasonable estimate of the algorithms running time.

1) Assuming $C(n) = \frac{1}{2} n(n-1)$. How much longer will be the algorithm run if it doubles it input size.

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

Scanned with CamScanner

If 'n' is too large as compared to n^2, n can be neglected

$\therefore C(n) \approx \frac{n^2}{2}$.

$\therefore T(n) = Cop\, C(n) = Cop \frac{n^2}{2}$.

& $T(2n) = Cop\, C(2n) = Cop \frac{(2n)^2}{2}$

$= Cop \frac{4n^2}{2}$.

Now, $\frac{T(2n)}{T(n)} = \frac{Cop \frac{4n^2}{2}}{Cop \cdot \frac{n^2}{2}}$

$\therefore \frac{T(2n)}{T(n)} = 4 \Rightarrow \boxed{T(2n) = 4T(n)}$

Hence, it requires 4 times when input is doubled.

* Order of growth:

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	n!
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		

. Values (some approximate) of several functions important for analysis of algorithm.

Scanned with CamScanner

07/01/19

Worst case, best case and average case efficiency:

* Algorithm sequential search (A[0...n-1]).

```
Algorithm sequential search(A[0...n-1], k)
// Searches for a given value in a given array by sequential
   search.
// Input : An array A[0...n-1] and a search key k.
// Output : An index of the first element of A that
            matches k or -1 if there are no matching
            elements.
i ← 0
while i<n and A[i] ≠ k do
      i ← i+1
if i<n return i
else return -1
```

Best case: $C_{best}(n) = 1$

Worst case: $C_{worst}(n) = n$.

* Assumption for analysis of algorithm-

- The probability of a successful search is equal to ($0 \le p < 1$).
- The probability of 1st match occuring at ith position is same for every i. ($0 \le i < n$).
- Now we find average number of key comparison as follows

  In case of successful search, the probability of first match occuring at i^{th} position of the list is p/n, for every i the number of comparison is i.

  In case of unsuccessful search, the number of comparison is 'n' with the probability of such a search being $(1-p)$.

Scanned with CamScanner

$$C_{avg}(n) = \left[1\cdot\frac{p}{n} + 2\cdot\frac{p}{n} + 3\cdot\frac{p}{n} + \ldots\ldots + n\cdot\frac{p}{n}\right] + n\cdot(1-p)$$

$$= \frac{p}{n}\left[1+2+3+\ldots\ldots+n\right] + n\cdot(1-p)$$

$$= \frac{p}{n}\left[\frac{n(n+1)}{2}\right] + n(1-p).$$

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

$$\therefore C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$ for sequential search.

→ For successful search, $p=1$

$\therefore C_{avg}(n) = \frac{n+1}{2} \approx \frac{n}{2}$.

→ For unsuccessful search, $p=0$

$\therefore C_{avg}(n) = n$.

Asymptotic notations and basic efficiency classes –

There are 3 asymptotic notations:

- big-oh (O)
- big-theta (Θ)
- big-omega (Ω)

Let $t(n)$ and $g(n)$ be the non-negative functions, defined on the set of natural numbers.

O-notation –

$O(g(n))$ is a set of all function with a smaller or same order of growth as $g(n)$.

Eg: $n \in O(n^2)$, $t(n) = n$ and $g(n) = n^2$

n	n^2
0	0
1	1
2	4
3	9
4	16

Scanned with CamScanner

Eg:- $100n+5 \in O(n^2)$

[Order of linear function will be smaller than order of quadratic function]

Substitute $n=101$.

$100 \times 101 + 5 = 10105$.

$n^2 = 101 \times 101$.

$= 10201$

n	$100n+5$	$g(n)$ n^2
0	5	0
1	105	1
2	205	4
3	305	9
⋮	⋮	⋮
101	10105	10201 (larger)
102	10205	104.04

$g(n) < 5$

$\therefore 100n+5 \in O(n^2)$ for $n_0 \geq 101$ ✓

Scanned with CamScanner

11/01/19

- **O notation:** $t(n) \in O(g(n))$, small order or same.
- **Ω notation:** $t(n) \in \Omega(g(n))$, larger or same order.
- **Θ notation:** $t(n) \in \Theta(g(n))$, same order.

Eg: $t(n) = n^2 + n \in O(n^3)$.

$n^3 \in \Omega(n^2)$.

$an^2 + bn + c \in \Theta(n^2 + n)$.

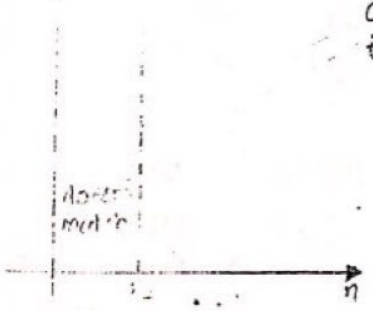
$n^3 + n^2 \in \Theta(n^3 + n)$.

$200n + 5 \notin \Theta(n^2)$.
linear quadratic.

1) **O notation:** A function $t(n)$ is said to be in big-oh(n) denoted as,

$t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n i.e. if there exists some positive constant 'c' and some non-negative integer n_0 such that:

$$\boxed{t(n) \leq c\,g(n) \quad \forall \quad n \geq n_0}$$



Eg: $100n + 5 \in O(n^2)$.

* $100n + 5 \in 101n$

n	LHS		RHS
n=0	5		0
n=1	105	>	101
n=2	205	>	202
n=3	305	>	303
n=4	405	>	404
n=5	505	=	505
n=6	605	<	606

$\therefore 100n + 5 \in 101n \quad \forall \quad n \geq 5$ and $c = 101$.

* $101n \in 101n^2 \quad \forall \quad n \geq 0$ and $c = 101$.

Scanned with CamScanner

* $\frac{1}{2}n(n-1) \in O(n^2)$.

n	LHS		RHS
$n=0$	0	=	0
$n=1$	0	<	1
$n=2$	1	<	4.

$\therefore \frac{1}{2}n(n-1) \in O(n^2)$ $\forall$ $n \geq 0$ and, $C=1$.

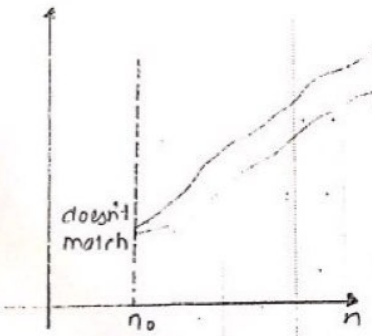
2) Ω notation: A function $t(n)$ is said to be in $\Omega(g(n))$ denoted,

$t(n) \in \Omega g(n)$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e. if there exist some positive constant C and some non-negative integer n_0 such that:

$$t(n) \geq C g(n) \quad \forall \quad n \geq n_0$$

Eg: $n^3 \in \Omega(n^2)$ $\forall$ $n \geq 0$ and $C=1$

* $0.001n^3 \in \Omega(n^2)$. $\forall$ $n \geq 10$ and $C=1$.



n	LHS	RHS
$n=0$	0	0
$n=1$	0.001	1
$n=2$	0.008	4.
⋮	⋮	⋮
$n=1000$	$0.001(1000)^3$	$=(1000)^2$
$n=999$	.997 002	< 999001
$n=1001$	1.003003	> 1002001

[We need to consider C as the coefficient of higher degree]

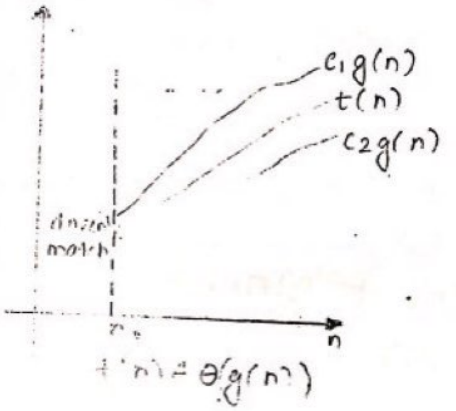
3) Θ notation: A function $t(n)$ is said to be in $\Theta(g(n))$ denoted,

$t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both upper/above and below by some

Scanned with CamScanner

positive constants multiple of $g(n)$. For all larger n, i.e. if there exists some positive constant c_1 and c_2 and some non-negative integer n_0 such that:

$$\boxed{c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \forall \; n \geq n_0}$$



Eg: $\frac{1}{2} n(n-1) \in \Theta(n^2)$.

$\frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{n}{2} < \frac{1}{2} n^2 \; \forall \; n \geq 0$

Consider, $t(n) = \frac{1}{2} n^2 - \frac{1}{2} n$ and

$g(n) = \frac{1}{2} n^2$. $c_1 = \frac{1}{2}$.

n	$t(n) = \frac{n^2}{2} - \frac{n}{2}$		$g(n) = \frac{1}{2} n^2 \; \forall \; n \geq 0$.
$n=0$	0	=	0
$n=1$	0	<	$\frac{1}{2}$
$n=2$	1	<	2
$n=3$	3	<	$\frac{9}{2}$
$n=4$	6	<	8

$\frac{1}{2} n^2 - \frac{1}{2} n \geq c_2 n^2$

n	$t(n)$		$g(n)$.	
$n=0$	0	=	0	
$n=1$	0	<	$\frac{1}{4} = c_2 \times$.	For what value of c_2 $t(n) \geq g(n)$.
$n=2$	1	>	$4c_2$ ✓	If $c_2 = 1/4$.
$n=3$	3	>	$9c_2$	
$n=4$	6	>	$16 c_2$.	

i.e. $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{4} n^2 \; \forall \; n \geq 2$.

$\therefore \frac{1}{2} n^2 - \frac{1}{2} n \in \Theta(n^2) \; \forall \; n \geq 2$ and $c_1 = \frac{1}{2}$ and $c_2 = \frac{1}{4}$.

Scanned with CamScanner

14/01/19

* If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,

$$t_1(n) + t_2(n) \in O \max\{g_1(n), g_2(n)\}.$$

Proof: Let a_1, b_1, a_2, b_2 be four arbitary real numbers.
If $a_1 \leq b_1$ and $a_2 \leq b_2$ then,

$$a_1 + a_2 \leq 2 \max\{b_1, b_2\}.$$

Since $t_1(n) \in O(g_1(n))$
there exists some (+ve) constant c_1 and some non-negative integer n_1 such that,

$$t_1(n) \leq c_1 g_1(n) \quad \forall\ n \geq n_1.$$

Similarly; $t_2(n) \in O(g_2(n))$
such that,

$$t_2(n) \leq c_2 g_2(n) \quad \forall\ n \geq n_2.$$

Let us consider $c = \max\{c_1, c_2\}$ and $n \geq \max\{n_1, n_2\}$, so that we can use both inequalities.

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 [g_1(n) + g_2(n)] \\ &\leq c_3\ 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O \max\{g_1(n), g_2(n)\}$ with c and n_0, where $c = 2c_3 = 2 \max\{c_1, c_2\}$ and: $n_0 = \max\{n_1, n_2\}$.

* Using limits for comparing order of growth:

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0, & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n). \\ c > 0, & \Rightarrow t(n) \text{ has the same order of growth as } g(n). \\ \infty, & \Rightarrow t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Scanned with CamScanner

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}.$$

↑ 'L'Hospital rule.

$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ [Sterling's Formula].
for large value of n.

- Compare order of growth of $\frac{1}{2}n(n-1)$ and n^2.

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \lim_{n\to\infty} \frac{\frac{1}{2}[n^2-n]}{n^2}$$

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

$$= \frac{1}{2}\times\left(1-\frac{1}{\infty}\right) = \frac{1}{2}(1-0).$$

$$= \frac{1}{2}. \quad \left(\because \frac{1}{\infty} > 0\right)$$

$\therefore \frac{1}{2}n(n-1) \in \Theta(n^2)$.

- Compare the order of growths of $n!$ and 2^n.

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}$$

$$= \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n.$$

$$= \infty.$$

$\therefore n! \in \Omega(2^n)$
though 2^n grows fast but $n!$ grows still faster.

t	constant
$\log n$	logarithms
n	linear
n^2	quadratic
n^3	cube
2^n	exponential
$n!$	Factorial.

Scanned with CamScanner

- **Mathematical Analysis of non-recursive algorithms.**

General Plan for analysing time efficiency in non-recursive algorithm -

Step 1 : Decide on a parameter indicating input size.

Step 2 : Identify the algorithm basic operations. (located in its innermost loop).

Step 3 : Check whether the number of times the basic operation is executed depends only on the size of an input.
If it also depends on some additional property the worst case, avg. case, and if necessary best case efficiency have to be investigated seperately.

Step 4 : Setup a sum expressing the number of times the algorithms basic operation is executed.

Step 5 : Using the standard formula & rules of some manipulation either find a closed formula for the count or at the very least establish its order of growth.

Scanned with CamScanner

Table 2.2 Basic Asymptotic Efficiency Classes.

Class	Name	Comments
1	constant	short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	logarithmic	Typically, a result of cutting a problem size by a constant factor on each iteration of the algorithm. Note that a logarithm algorithm cannot take into account all its input (or even a fixed fraction of it) any algorithm that does so will have atleast linear running time.
n	linear	Algorithms that scan a list of size n belong to this class.
$n \log n$	n-log-n	Many divide-and-conquer algorithms including mergesort & quicksort in the average case, fall into this category.
n^2	quadratic	Typically, characterizes efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	cubic	Typically, characterizes efficiency of algorithms with three embedded loops. Several non-trivial algorithms from linear algebra fall into this class.
2^n	exponential	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and faster orders

Scanned with CamScanner

Class	Name	Comments
		of growth as well.
$n!$	factorial	Typical for algorithms that generate all permutations of an n-element set.

Algorithm MaxElement (A[0 ... n-1])

- Find maximum element in an array and analyse its complexity.

//Determines the value of the largest element in a given array.
//Input : Array of 0 to n-1 of real numbers.
//Output: The value of largest element in A.

```
Maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > Maxval
        Maxval ← A[i]
return Maxval.
```

→ Analysis

input size ← n

basic operation ← comparison

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \quad \text{or} \quad n-1+1-1 = n-1 \text{ [By formula].}$$

$$\therefore C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n) \qquad \{u-l+1\}.$$

Scanned with CamScanner

18/01/19

- Find whether all the elements in a given array is distinct or not.

Algorithm ElementUniqueness(A[0..... n-1])

// Determines all the element in a given array is distinct.

// Input : An array A[0] to A[n-1].

// Output : Returns true, if all the elements are distinct
false otherwise.

```
ElementUnique ← A[0]
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] = A[j]
            return false
return true
```

Tracing - n: 6

A[0] 42
[1] 5
[2] 36
[3] 12
[4] 10
[5] 2

→ Analysis

input size ← n

basic operation ← comparison

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1)$$

$$= (n-1)+(n-2)+(n-3)+\dots\dots+2+1$$

$$= 1+2+3+\dots\dots+(n-1)$$

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

$\therefore \frac{n^2}{2} \in \Theta(n^2)$.

Scanned with CamScanner

20/01/19

- Algorithm MinDistance(A[0.....n-1])

```
dmin ← 9999
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        if i ≠ j & |A[i] - A[j]| < dmin
            dmin ← |A[i] - A[j]|
return dmin
```

a) What does this algorithm compute?
b) What is the basic operation?
c) What is the complexity of this algorithm?
d) How do you improve this algorithm?
e) How many times the basic operation executed?
f) What is the efficiency class of the new algorithm?

a) Finds the distance btw the 2 closest elements in an array numbers (min distance btw 2 elements of array).

b) Comparison.

c) $$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2$$

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

$$= 2 \underbrace{(n + n + \dots + n)}_{n \text{ times}} \; n \cdot n = n^2$$

$$= 2n^2 \in \theta(n^2)$$

d) Algorithm MinDistance(A[0...n-1])

```
dmin ← 9999
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if |A[i] - A[j]| < dmin
            dmin ← |A[i] - A[j]|
return dmin
```


Scanned with CamScanner

c) $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

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

$= (n-1) + (n-2) + (n-3) \ldots\ldots 3+2+1$

$= 1+2+3+\ldots\ldots (n-3)+(n-2)+(n-1)$

$= \frac{(n-1)n}{2} \approx \frac{n^2}{2} \in \theta(n^2)$

b) $\theta(n^2)$

- Algorithm Enigma ($A[0 \ldots\ldots n-1]$)

```
for i ← 0 to n-2 do                    // palindrome..
    for j ← i+1 to n-1 do
        if A[i,j] ≠ A[j,i]
            return false.
return true.
```

Answer the following questions:

a) What does this algorithm do?

b) What is the basic operation?

c) How many times the basic operation executed?

d) What is the efficiency class of this algorithm?

a) It finds whether the matrix is symmetric or not over principal diagonal.

b) Comparison

c) $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

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

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

$= (n-1) + (n-2) + \ldots\ldots +3+2+1$

$= 1+2+3+\ldots\ldots +(n-1)$

$= \frac{(n-1)n}{2} \approx \frac{n^2}{2} \in \theta(n^2)$.

d) $\theta(n^2)$.

Scanned with CamScanner

- Algorithm secret(A[0.... n-1])

```
minval ← A[0]; maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval - minval
```

a) To find maximum and minimum elements in an array.

b) Comparison.

c) $C(n) = \sum_{i=1}^{n-1} 2$

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

$= 2(n-1) \approx 2n \in \Theta(n)$

d) Improved algorithm

Algorithm secret(A[0.... n-1])

```
        else if A[i] > maxval
```

1) Estimate the running time of multiplication of 2 matrices of order $n \times n$.

Algorithm Matrix Multiplication(A[0....n-1, 0---- n-1], B[0..n-1, 0..n-1]

```
// Multiplies 2 n x n matrices
// Input : 2 n x n matrices A and B
// Output : matrix C = A.B
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        c[i,j] ← 0
        for k ← 0 to n-1 do        A[i,k] * B[k,j]
            c[i,j] ← c[i,j] + A[i,j] * B[i,j]
return C
```


Scanned with CamScanner

Sol: Basic operations are addition and multiplication it is sufficient if we analyse for multiplication operation as it is more time consuming than addition.

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n-1+1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

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

$$= \underbrace{(n^2+n^2+\dots+n^2)}_{n\text{-times}} = n^2 \times n = n^3 \in \theta(n^3).$$

If C_m is the time required to perform one multiplication operation then running time $= T(n) \simeq C_m M(n)$

$$T(n) = C_m \cdot n^3$$

To get more accurate estimate we need to take into account, time spent on addition also,

$\therefore T(n) \approx C_m M(n) + C_a A(n)$ where C_a is the time for one addition.

$\approx C_m n^3 + C_a n^3$

$$\boxed{\therefore T(n) \approx n^3 (C_m + C_a)}$$

Scanned with CamScanner

Q1)

- Algorithm to find the binary digits in a binary representation of positive decimal integer.

Algorithm binaryCount(n)

// Finds number of binary digits
// Input: Positive decimal integer.
// Output: Number of binary digits in n's binary representation.

```
count ← 1                          or count ← 0
while n > 1 do                        while n > 0 do.
    count ← count + 1
    n ← [n/2]
return count                       Count ← 1
```

→ Analysis

input size ← n

basic ← Comparison in while loop condition [because one extra time it will get executed]. Entry control loop.

Complexity ← $n/2^i = 1$

$\because n/2, n/4, n/8 \cdots\cdots n/2^i$

WKT, $n/2^i = 1$

$\therefore i = \log_2 n$

$\therefore (\log_2 n + 1) \approx \log_2 n \in \Theta(\log_2 n)$ ← extra comparison

Scanned with CamScanner

Recursive Algorithm:

$F(n) = n!$

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

$\boxed{n! = n * (n-1)!}$ Recursively

$n! = n * (n-1)! \quad \forall\ n \geq 1$

$0! = 1$

Algorithm F(n)

// Compute n! recursively

// Input: A non negative integer n

// Output: The value of n!

if $n = 0$ return 1

else return $F(n-1) * n$

→ Analysis

input size ← n

basic operation ← Multiplication.

Recursive relation.

$M(n) = M(n-1) + 1$ for $\forall\ n > 0$

$M(1) = 0$

→ to multiply $F(n-1)$ by n.

no. of multiplication required to compute $F(n-1)$

- To solve recurrence relation, Backward substitution is required.

$M(n) = M(n-1) + 1 \quad \forall\ n > 0$

$M(n) = 0$

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

$= (M(n-2) + 1) + 1$ Substitute $M(n-1) = M(n-2) + 1$

$= M(n-2) + 2$

$= M(n-3) + 1 + 2$ Substitute $M(n-2) = M(n-3) + 1$

Scanned with CamScanner

$= M(n-3)+3$.

$=$ ⋮

$= M(n-i)+i$

$= M(n-n)+n$

$= M(0)+n$

$= 0+n$.

$\therefore M(n) = n$

- General plan for analysing time efficiency of recursive algorithm -

Step 1: Decide on a parameter indication / indicating input size.

Step 2: Identify the algorithm basic operation.

Step 3: Check whether the number of times the basic operation is executed can vary and on different input of same size. If it can in the worst case, best case and avg case efficiency must be investigated seperately.

Step 4: Setup a recurrence relation with an appropriate initial condition for the number of times the basic operation is executed.

Step 5: Solve the recurrence relation.

* Write recursive algorithm to compute sum of first n cubes and analyse its complexity.
Compare the performance of this algorithm with straightforward non-recursive algorithm.

$$S(n) = 1^3+2^3+3^3+\ldots\ldots+n^3$$

Scanned with CamScanner

Algorithm $S(n)$
// Compute the sum of frist n^3
// Input: A positive integer n
// Output: returns sum of frist n^3.
if $n=1$ return 1
else return $S(n-1) + n*n*n$

→ Analysis

input size ← n
basic operation ← Multiplication
Recurrence relation
$M(n) = M(n-1) + 2 \quad \forall\ n \geq 1$
$M(1) = 0$
$M(n) = M(n-1)+2 = (M(n-2)+2)+2 = M(n-2)+4$
$= M(n-3)+2+4 = M(n-3)+6$
$= \ldots\ldots\ldots = M(n-i)+2i$
$= M(n-(n-1))+2(n-1) \quad \therefore \boxed{M(n) = 2n-2 \approx 2n \in \theta(n)}$

Tower of Hanoi

```
void towerofHanoi(int disk, char from, char to, char aux)
{
    if (disk==1)
    {
        printf("Move disk 1 from %c peg to %c\n", from, to);
        return;
    }
    towerofHanoi(disk-1, from, aux, to);
    printf("Move disk %d from %c peg to %c peg", disk, from,
                                                  to);
    towerofHanoi(disk-1, aux, to, from);
}
```


Scanned with CamScanner

23/01/19

- **Non-recursive Algorithm**

// Compute the sum of first n^3:

// Input: A positive integer n.

// Output: A positive integer that is sum of first n^3.

Algorithm S(n)

```
Sum ← 0
for i ← 1 to n do
    sum ← sum + i * i * i
return sum
```

→ **Analysis**

i) Input size ← n

ii) Basic operation ← multiplication

Complexity:

$$M(n) = \sum_{i=1}^{n} 2$$

$$= 2 \sum_{i=1}^{n} 1$$

$$= 2 \times (n - 1 + 1)$$

$$= 2n \in \Theta(n)$$

$$\therefore \boxed{M(n) \in \Theta(n)}$$

[Complexity of non-recursive algo is greater as compa to recursive algo].

This is same as same as recursive version but non recursive version does not carry the time and space overhead associated with recursive stack.

- **Tower of Hanoi.**

input size ← number of disks n

basic operation ← no. of moves of disk.

Scanned with CamScanner

For every invocation of function it recursively calls itself twice.

Complexity: $M(n) = M(n-1) + 1 + M(n-1)$

$= 2M(n-1) + 1$ for all $n > 1$

$M(1) = 1$.

$\therefore M(n) = 2M(n-1) + 1$ Substitute $M(n-1) = 2M(n-2)+1$

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

$= 2^2 M(n-2) + (2+1)$. Sub. $M(n-2) = 2M(n-3)+1$

$= 2 \cdot M(n-3) + 2^2 + 2 + 1$

i.e. $2^2(2M(n-3)+1) + (2+1)$.

Subs. $M(n-3) = 2M(n-4)+1$.

i.e.

$2^3 M(n-3) + 2^2 + 2 + 1$

$= 2^3(2M(n-4)+1) + 2^2 + 2 + 1$

$= 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$.

$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2^2 + 2^1 + 2^0$

Substitute $n - i = 1$

$n - 1 = i$.

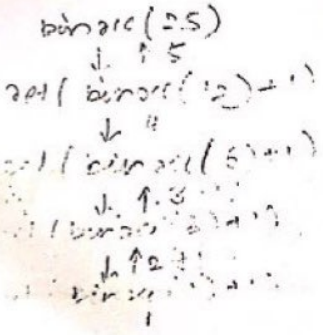
$\therefore M(n) = 2^{n-1}\left(M(n-(n-1))\right) + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$

$= 2^{n-1}(M(1)) + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$.

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2^1 + 2^0$.

$= 2^{n-1+1} - 1 = 2^n - 1$.

$\therefore M(n) = 2^n - 1$



Scanned with CamScanner

- Write a recursive algorithm to find number of digits in binary representation of positive decimal integer.

```
//Compute the binary digits in binary representation.
// Input: Positive decimal integer.
// Output: Number of binary digits in n's binary repre
          entation.
if (n=1) return 1
else return binrec([n/2])+1
```

→ $A(n) = A(n/2)+1 \quad \forall\ n>1$.

$A(1) = 0$.

$A(n) = A(n/2)+1$ // Put $n = 2^k$

$= A\left(\frac{n-2}{2}\right)+1$ ✗

WKT, $M(n) = M(n-1)+1$ ✗

$\therefore A(2^k) = A\left(\frac{2^k}{2}\right)+1$

$= A(2^{k-1})+1$

Smoothness Rule:

Let, $n = 2^k$ then recurrence relation represented a

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

$A(2^k) = A(2^{k-1})+1, \forall\ n>1$ // if we

then,

$A(2^{k-1}) = A($

$= A($

$\therefore A(2^0) = 0.$

$A(2^k) = A(2^{k-1})+1 \quad \forall\ k>0$

$= A(2^{k-2})+1+1$ Substitution,

$A(2^{k-1}) = A(2^{k-2})+$

$A(2^{k-2}) = A(2^{k-3})+$

$= A(2^{k-3})+1+1+1$

$= A(2^{k-3})+3$

$\vdots$

$A(2^k) = A(2^{k-i})+i$ $\quad k-i=0$

$= A(2^{k-k})+k$ $\quad \therefore i=k$

$\therefore A(2^k) = k.$

$2^k = n$

$k = \log_2 n.$

$\therefore \boxed{A(n) = \log_2 n} \in \Theta(\log_2 n)$

Scanned with CamScanner

Algorithm Tower of Hanoi:

// Algorithm for tower of hanoi.
// Input: No. of disk n, name of three pegs
// Output: Move of disk from one peg to another.

```
Algorithm TowerofHanoi(n, src, aux, dest)
if disk == 1
    move disk from src to dest
    return
TowerofHanoi(n-1, src, dest, aux)
move disk from src to dest
TowerofHanoi(n-1, aux, src, dest)
```


Scanned with CamScanner

25|01|19

→ Solve the following recurrence relation:

i) $x(n) = x(n-1) + 5$ for all $n \geq 1$ and $x(1) = 0$.

Sol°: Given: $x(n) = x(n-1) + 5 \quad \forall\ n \geq 1$

and,

$x(1) = 0$.

$x(n) = x(n-1) + 5$

$= (x(n-2) + 5) + 5$

$= x(n-2) + 5 + 5$

$= x(n-3) + 5 + 5 + 5$

$= x(n-3) + 3 \cdot 5$

$\vdots$

$= x(n-i) + 5i$

Substitute:

$x(n-2) = x(n-2) + 5$

Now, $n - i = 1 \Rightarrow i = n-1$ bcoz we have if $n-i$

$= x(n-(n-1)) + 5(n-1)$

$= x(n-n+1) + 5(n-1)$

$= x(1) + 5(n-1)$.

$\therefore x(n) = x(1) + 5(n-1)$

$= 0 + 5(n-1)$.

$\therefore \boxed{x(n) = 5(n-1)} \in \Theta(n)$

ii) $x(n) = 3x(n-1)$ for all $n \geq 1$ and $x(1) = 4$.

Sol°: Given, $x(n) = 3 \cdot x(n-1) \quad \forall\ n \geq 1$

$\Rightarrow x(n) = 3 \cdot 3\, x(n-2)$

$= 3 \cdot 3 \cdot 3\, x(n-3)$

$= 3^3 x(n-3)$.

$\vdots$

$= 3^i x(n-i)$.

$\vdots$

$= 3^{n-1} x(n-n+1)$

$= 3^{n-1} x(1)$

$n - i = 1$

$i = n-1$

Scanned with CamScanner

$\Rightarrow x(n) = 3^{n-1} \times 4.$

$= 4 \cdot 3^{n-1}.$

$\therefore \boxed{x(n) = 4 \cdot 3^{n-1}} \in \theta(3^n).$

iii) $x(n) = x(n-1) + n \quad \forall\ n > 0$ and $x(0) = 0.$

Sol°: Given: $x(n) = x(n-1) + n$

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

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

$= x(n-4) + (n-1) + (n-2) +$ ✗

Given: $x(n) = x(n-1) + n$

$= x(n-2) + (n-1) + n$ Subs. $x(n-1) = x(n-2) + (n-1)$

$= x(n-3) + (n-2) + (n-1) + n.$

$\vdots$

$= x(n-i) + [(n-(i-1)) + (n-(i-2)) + \ldots + (n-1) + n].$

put, $n - i = 0$, $n = i$. i.e $i = n$

$x(n) = x(0) + [1 + 2 + 3 + \ldots + n]$

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

$\therefore \boxed{x(n) = \frac{n(n+1)}{2}} \in \theta(n^2).$

iv) $x(n) = x(n/3) + 1 \quad \forall\ n \geq 1,\ x(1) = 1.$

Sol°, Given: $x(n) = x(n/3) + 1.$

By Smoothness Rule,

$n = 3^k.$

$\Rightarrow x(n) = x(n/3) + 1$

$\Rightarrow x(3^k) = x(3^{k-1}) + 1 \quad \forall\ k \geq 0,\ x(3^0) = 1.$

$= x(3^{k-2}) + 1 + 1$

$= x(3^{k-3}) + 1 + 1 + 1$

$= x(3^{k-3}) + 3$

Scanned with CamScanner

$\therefore x(3^k) = x(3^{k-3}) + 3$

$\vdots$

$x(3^k) = x(3^{k-i}) + i$; $k-i=0$

$= x(3^{k-k+i}) + i$ $k=i$ $i=k$.

$= x(3^0) + k$

$= k+1$.

$\therefore x(3^k) = k+1$.

$\therefore \boxed{x(n) = 1+\log_3 n \in \theta(\log_3 n).}$

v) $x(n) = x(n/2) + n$ $\forall$ $n>1$, $x(1)=1$.

Sol°: Given, $x(n) = x(n/2) + n$ $\forall$ $k>1$

Put $n=2^k$.

$x(2^k) = x(2^{k-1}) + 2^k$, Substitute $x(2^{k-1}) = x(2^{k-2}) + 2^{k-1}$

$= x(2^{k-2}) + 2^{k-1}$

$= x(2^{k-3}) + 2^{k-2} + 2^{k-1}$

$\vdots$

$= x(2^{k-i}) + 2^{k-(i-1)} + 2^{k-(i-2)} + \ldots + 2^k$

Put $k-i=0$ $\boxed{i=k.}$ $= x(2^{k-k}) + 2^{k-(k-1)} + 2^{k-(k-2)} + \ldots + 2^k$.

$\therefore x(2^k) = x(2^0) + 2^1 + 2^2 + 2^3 + \ldots + 2^k$

$= x(1) + [2 + 2^2 + 2^3 + \ldots + 2^k]$

$= 1 + 2 + 2^2 + 2^3 + \ldots + 2^k$

$= 2^0 + 2^1 + 2^2 + \ldots + 2^k$

$= 2^{n+1} - 1$ here, $n=k$.

$\therefore \boxed{x(2^k) = 2^{k+1} - 1} \in \theta(n)$

$\therefore x(2^k) = 2 \cdot 2^k - 1$

$\boxed{x(n) = 2n-1} \in \theta(n)$

Scanned with CamScanner

28/01/19

* Bruteforce Method:

→ **Selection sort** – (No other method can be used to solve sort i.e selection and bubble sort except this method)

Algorithm Selection Sort (A[0.... n-1])

// Sorts a given array by selection sort.
// Input: An array (A[0].... [n-1]) of orderable elements.
// Output: Array (A[0].... [n-1]) sorted in ascending order.

```
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]
```

→ **Analysis**

Input size ← n
Basic operation ← comparison

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

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

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

$$= (n-1)+(n-2)+(n-3)+ \dots +2+1$$

$$= \frac{n(n-1)}{2} \approx \frac{n^2}{2} \in \Theta(n^2).$$

$$E(n) = \sum_{i=0}^{n-2} 1 \quad \text{// Swapping}$$

$$= n-2-0+1$$

$$= n-1.$$

$$\therefore \boxed{E(n) = n-1 \in \Theta(n)}$$

Scanned with CamScanner

Tracing -

Pass 0	1	2	3	4	5	6
	45	60	-12	42	10	-115
i=0	-45 |	45	60	-12	42	10	20
i=1	-45	-12 |	60	45	42	10	20
i=2	-45	-12	10 |	45	42	60	20
i=3	-45	-12	10	15 |	42	60	20
i=4	-45	-12	10	15	20 |	60	42
i=5	-45	-12	10	15	20	45 |	42
	-45	-12	10	15	20	45	60

→ Bubble Sorting:

Algorithm Bubblesort(A[0.... n-1])

// Sorts a given array by bubblesort

// Input: An array A[0.. n-1] of orderable elements

// Output: Array (A[0.. n-1]) sorted in ascending order.

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j] > A[j+1]
            swap A[j] and A[j+1]
```

→ Analysis

Input size ← n

Basic operation ← comparison

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-2-i} 1$$

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

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

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

Scanned with CamScanner

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

$$= \frac{n(n-1)}{2} \approx \frac{n^2}{2} \in \theta(n^2).$$

$$E(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} 1$$

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

$$\therefore \boxed{E(n) = \frac{n^2}{2} \in \theta(n^2)}$$

[Selection sort is better than bubble sort because its iteration is linear in nature i.e $E(n) \in \theta(n)$ but in bubble sort $E(n) \in \theta(n^2)$.]

Tracing :

	0	1	2	3	4	5	
	20	45	60	-12	72	10	
i=0	20	45	-12	60	10	-45	
i=1	20	-12	45	10	-45	15	60
i=2	-12	20	10	-45	15	45	
i=3	-12	10	-45	15	20	45	
i=4	-12	-45	10	15	20	45	60
i=5	-45	-12	10	15	20	45	60

	0	1	2	3	4	5	6	
	20	45	60	-12	72	10	-45	15
i=0	20	45	-12	60	10	-45	15	
			45	10	-45	15	60	
		20	10	-45	15	45	60	
		10	-45	15	20	45	60	
i=4		45	10	15	20	45	60	
		-12	10	15				

Scanned with CamScanner

Work –

1) Consider 10 elements and sort by selection and bubble sort.

SELECTION SORT TRACE

	0	1	2	3	4	5	6	7	8	9
	20	9	45	60	52	68	-12	72	10	-45
i=0	-45	9	45	60	52	68	-12	72	10	20
i=1	-45	-12	45	60	52	68	9	72	10	20
i=2	-45	-12	9	60	52	68	45	72	10	20
i=3	-45	-12	9	10	52	68	45	72	60	20
i=4	-45	-12	9	10	20	68	45	72	60	52
i=5	-45	-12	9	10	20	45	68	72	60	52
i=6	-45	-12	9	10	20	45	52	72	60	68
i=7	-45	-12	9	10	20	45	52	60	72	68
i=8	-45	-12	9	10	20	45	52	60	68	72

∴ The traced sorting is done.

Bubble Sort:

Let the elements be – 20, 9, 45, 60, 52, 68, -12, 72, 10, -45.

BUBBLE SORT TRACE

	20	9	45	60	52	68	-12	72	10	-45
i=0	9	20	45	52	60	-12	68	10	-45	72
i=1	9	20	45	52	-12	60	10	-45	68	72
i=2	9	20	45	-12	52	10	-45	60	68	72
i=3	9	20	-12	45	10	-45	52	60	68	72
i=4	9	-12	20	10	-45	45	52	60	68	72
i=5	-12	9	10	-45	20	45	52	60	68	72
i=6	-12	9	-45	10	20	45	52	60	68	72
i=7	-12	-45	9	10	20	45	52	60	68	72
i=8	-45	-12	9	10	20	45	52	60	68	72

Scanned with CamScanner

29/1/19

- **Sequential Search:**

Algorithm SequentialSearch(A[0--- n-1], k)

// · Implements sequential search with a given search key element.

// Input: An array A[0... n-1] and a search key element

// Output: returns the positions of the key element if it is found otherwise returns -1.

```
A[n] ← k
i ← 0
while A[i] ≠ k do
        i ← i+1
if i<n return i
else return -1
```

- **Bruteforce String matching:**

Algorithm BruteforceStringMatch(T[0..... n-1], P[0.... m-1])

// Implements bruteforce string matching.

// Input: An array T[0..... n-1] of n character representing a text and an array P[0....m-1] of m characters representing a pattern.

// Output: The index of first character in the text that starts a matching substring or -1 if the search is unsuccessful.

```
for i ← 0 to n-m do
    j ← 0
    while j<m and P[j]=T[i+j] do
            j ← j+1
    if j=m return i
return -1
```

7-3
=5

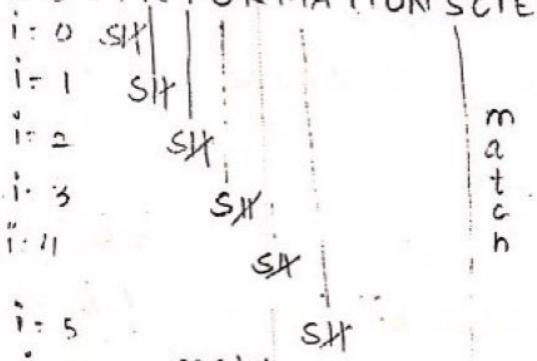
Scanned with CamScanner

Tracing:

n = 25, m = 7.

Text: INFORMATION SCIENCE AND ENGG (25)

Pattern: SCIENCE (0 1 2 3 4 5 6)



then j = m returns 11

i: 1 P[1] : T[12] ✓

j: 2 P[2] : T[13] ✓

i: 5 P[5] : T[16] ✓ match.

→ j < m i.e 4 < 7 not equal goto if i.e j = m ✓ i.e 6 = 6

j = m True returns i i.e 11.

i = 0.

P[0] = T[0].

S = I not equal ✗

comeout.

j = 0 i.e j ≠ m ✗

i = 1

j = 0.

P[0] = T[1].

S = N ✗.

j ≠ m ✗ again for loop

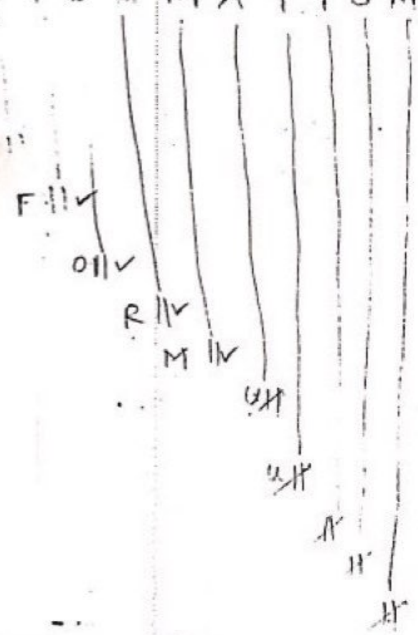
i = 2.

Total comparison made = 18.

Pattern: FORMULA (0 1 2 3 4 5 6)

A with U. ✗

Text: INFORMATION SCIENCE AND (0 1 2 3 4 5 6 7 8 9 10)



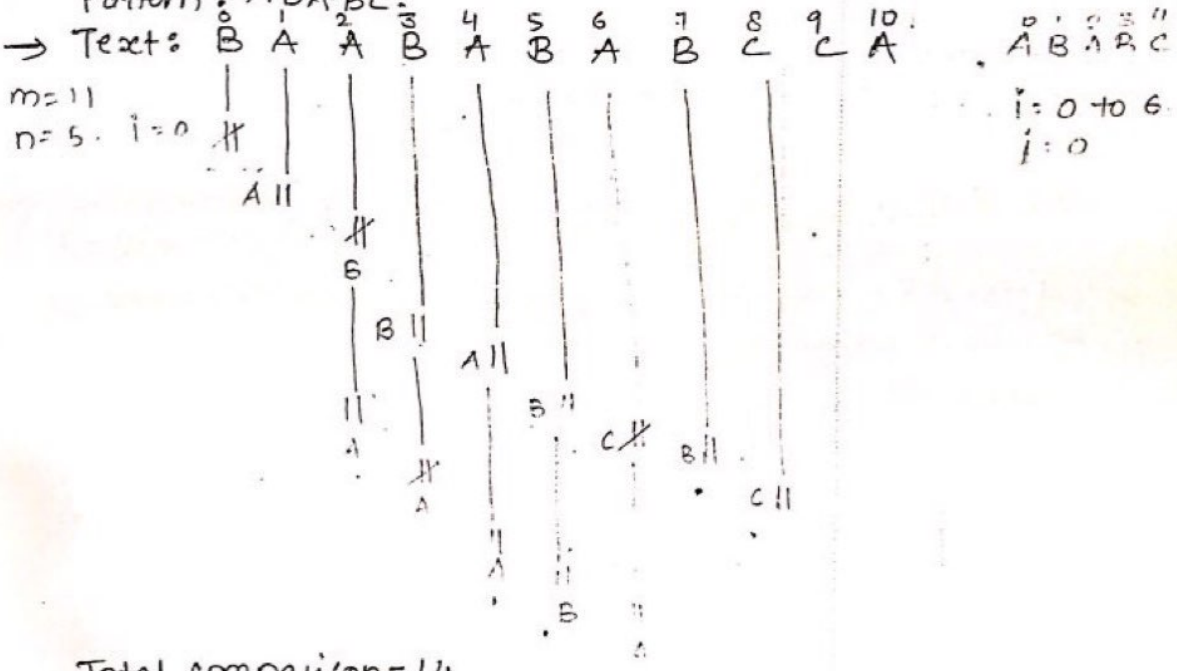
Total 23 comparison.

Scanned with CamScanner

1) Find the number of character comparison that can be made by straight forward string matching for the pattern ABABC in the following text, BAABABABCCA.

Pattern: ABABC.

→ Text: B A A B A B A B C C A

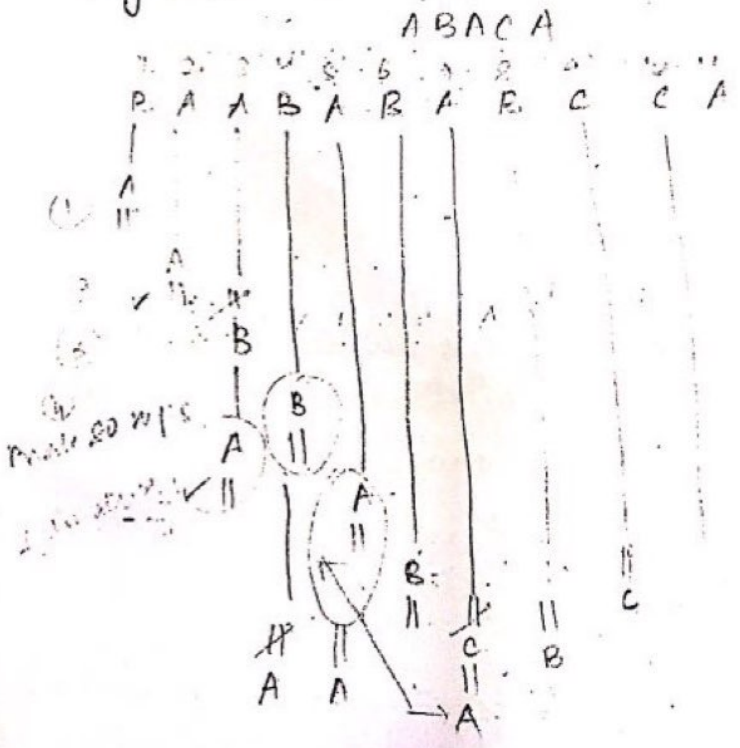


Total comparison = 14

$C_{best}(n) = m \in \theta(m)$

$C_{worst}(n) = (n-m+1)\,m \in \theta(nm)$

$C_{avg}(n) = (n+m) \in \theta(m+n) \in \theta(n)$, linear.



Scanned with CamScanner

30/01/19 Unit-II

• DIVIDE AND CONQUER

→ Finding sum of 'n' numbers

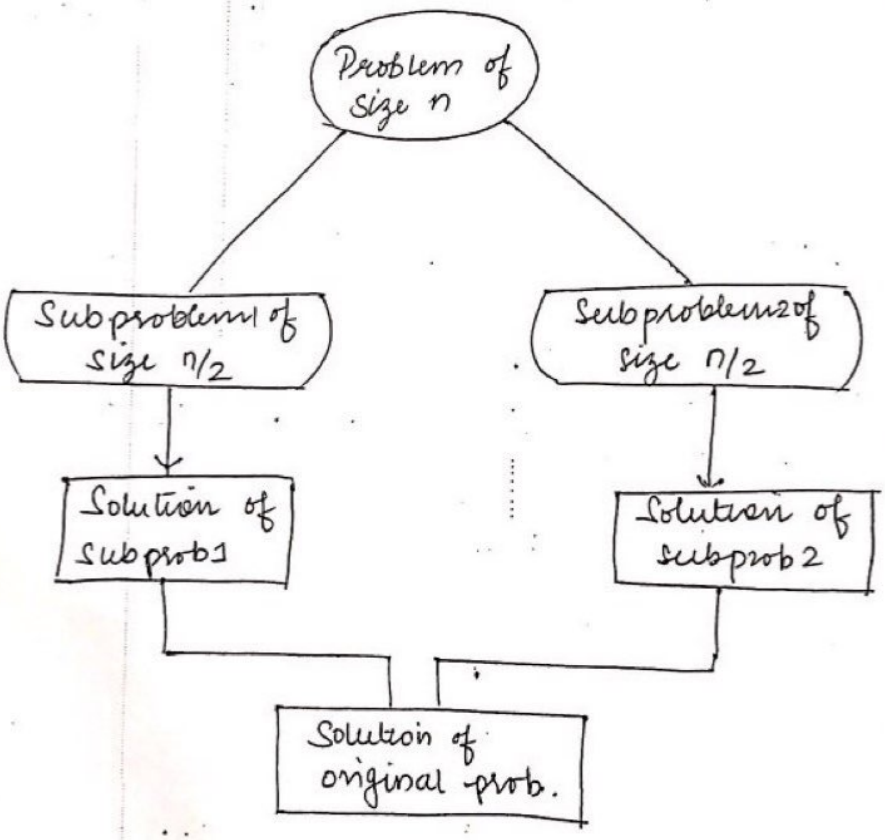
$a_0 + a_1 + a_2 + \dots\dots\dots + a_{n-1}$

$= (a_0 + a_1 + \dots\dots + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + \dots\dots\dots + a_{n-1})$

$= (a_0 + a_1 + \dots\dots\dots + a_{11}) + (a_{12} + a_{13} + \dots\dots\dots + a_{24})$

$= (a_0 + a_1 + \dots\dots + a_5) + (a_6 + \dots\dots + a_{11}) + (a_{12} + a_{13} + \dots\dots + a_{17}) + (a_{18} + a_{19} + \dots\dots\dots + a_{24})$

→ If $n = 25$ i.e. $\frac{n}{2} = 12$.



A problem instance of size 'n' is divided into two instances of size $n/2$. generally an instance of size 'n' can be divided into 'b' instance of size n/b, with 'a' of them need to be solved. ('a' and 'b' are constants with $a \geq 1$ and $b > 1$).

Scanned with CamScanner

Assuming that size 'n' is a power of 'b', to simplify the analysis we get following recurrence relation for the running time $T(n)$.

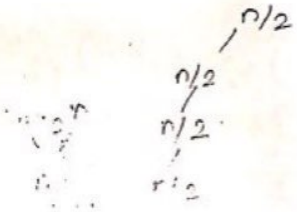
$$\boxed{T(n) = a \cdot T(n/b) + f(n)} \quad \text{— (1)}$$

general divide and conquer recurrence.

where $f(n)$ refers to the time spent on dividing the problem into smaller ones and combining their solutions.

* Master Theorem: If $f(n) \in \theta(n^d)$ with $d \geq 0$ in recurrence relation (1), then

$$T(n) \in \begin{cases} \theta(n^d), & \text{if } a < b^d. \\ \theta(n^d \log_2 n), & \text{if } a = b^d. \\ \theta(n^{\log_b a}), & \text{if } a > b^d. \end{cases}$$



Eg: Computing sum of 'n' numbers.

Sol: Let us consider $n = 2^k$.

$A(n) = 2 \cdot A(n/2) + 1 \quad \forall \; n \geq 1$ and $A(1) = 0$

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

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

$b^d = 2^0 = 1$

$a > b^d$

i.e $2 > 1$

$A(n) \in \theta(n^{\log_b a})$

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

$\therefore A(n) \in \theta(n)$

$f(n) \in \theta(n^d)$

$n^d = 1 \Rightarrow d = 0.$

$n^0 = 1.$

$\Rightarrow A(2^k) = 2$

$\boxed{\therefore A(n) \in \theta(n)}$

Scanned with CamScanner

Eg: $C(n) = 2C(n/2) + (n-1)$ $\forall n \geq 1$

$C(1) = 0$.

Sol°, $a = 2, b = 2 \sim f(n) = n-1$

$b^d = 2^1 = 2$ $\quad n^d = n-1$

$n^d \approx n \Rightarrow d = 1$

i.e,

$a = b^d$.

$\therefore C(n) \in \theta(n^d \log n)$

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

$\therefore \boxed{C(n) \in \theta(n \log n)}$

Backward Substitution:

Given: $C(n) = 2C(n/2) + (n-1)$

Put $n = 2^k$.

$C(2^k) = 2C(2^{k-1}) + (2^k - 1)$, Substitute,

$C(2^{k-1}) = 2C(2^{k-2}) + (2^{k-1} - 1)$

$C(2^{k-2}) \Rightarrow 2C(2^{k-3}) + (2^{k-2} - 1)$

$= 2C(2^{k-2}) + (2^{k-1} - 1) + (2^k - 1)$

$= 2C(2^{k-3}) + (2^{k-2} - 1) + (2^{k-1} - 1) + (2^k - 1)$.

$\vdots$

$= 2C(2^{k-i}) + (2^{k-(i-1)} - 1) + (2^{k-(i-2)} - 1) + \ldots$

put $k - i = 0$

$i = k$

$= 2C(1) + (2^1 - 1) + (2^2 - 1) + (2^3 - 1) + \ldots + (2^k - 1)$

$= 2 \times 0 + [2^1 + 2^2 + 2^3 + \ldots + 2^k] - [1 + 1 + 1 + \ldots k]$.

$= [2^1 + 2^2 + 2^3 + \ldots + 2^k] - k$.

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

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

$C(n) = 2^k \cdot 2 - 2 - k = 2n - 2 - \log_2 n$.

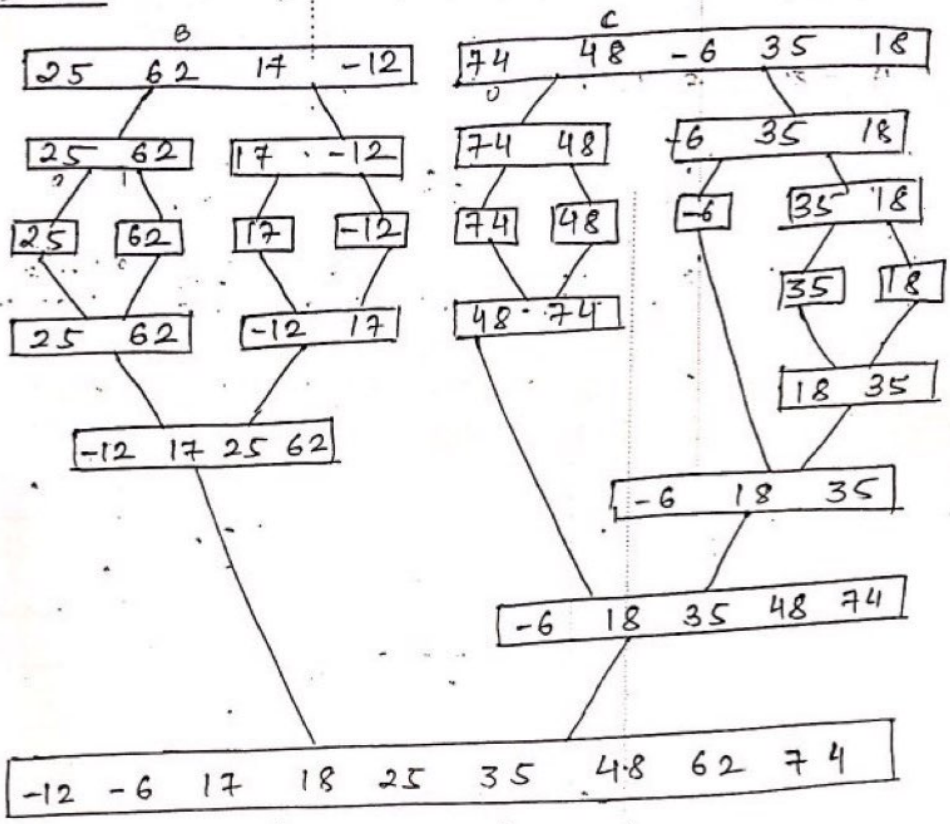
$= 2(n-1) - \log_2 n$

Scanned with CamScanner

• Merge sort- A = 25, 62, 12, -12, 74, 48, -6, 35, 18

Eg:

n=9



Algorithm Mergesort (A[0.... n-1])

// Sorts array of A[0..... n-1] by recursive merge sort.

// Input: An array of A[0..... n-1] of orderable elements.

// Output: Array of A[0.... n-1] sorted in non-decreasing order.

if n > 1

copy A[0.... (n/2)-1] to B[0..... [n/2]-1]

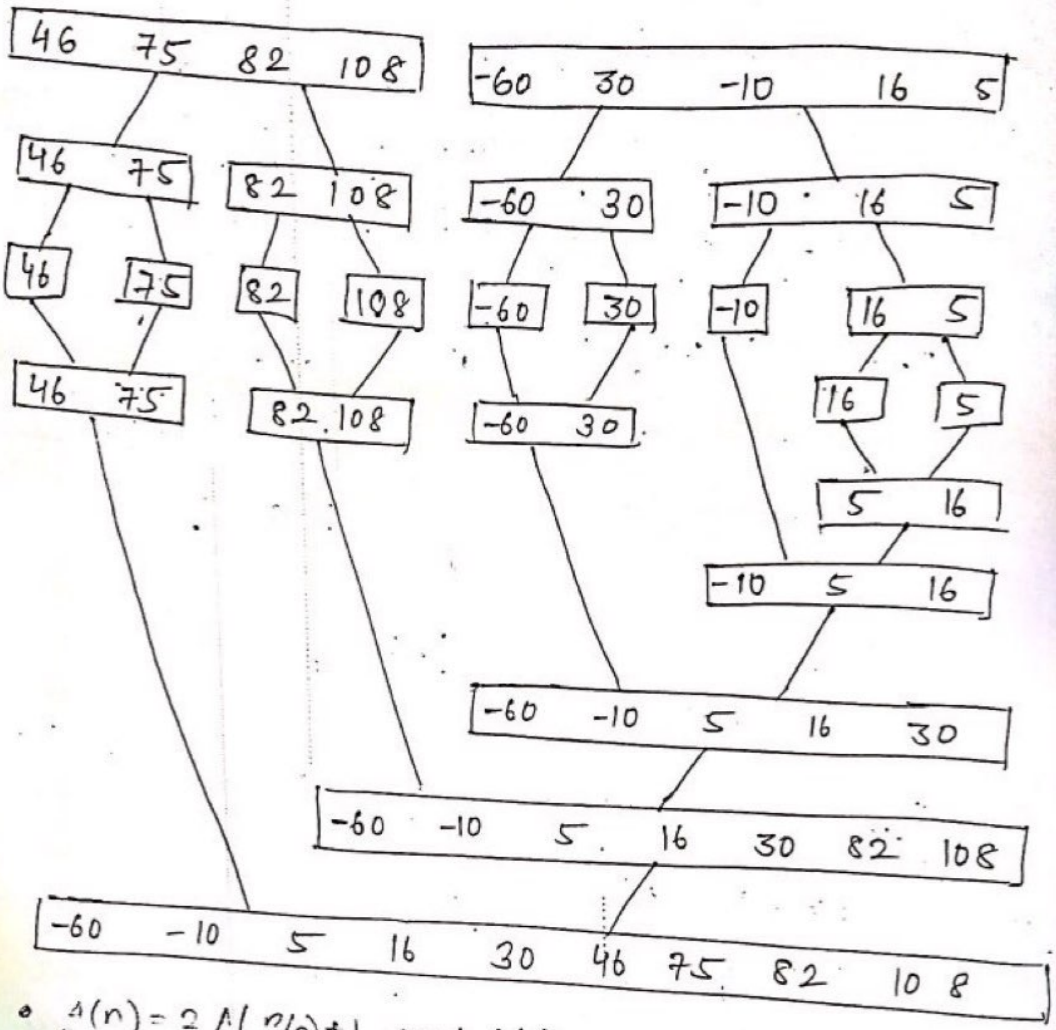
copy A[(n/2) .... (n-1)] to C[0..... [n/2]-1]

Mergesort (B[0..... [n/2]-1])

Mergesort (C[0..... [n/2]-1])

Merge (B, C, A)

Scanned with CamScanner



- $A(n) = 2A(n/2) + 1$ and $A(1) = 0$.

Put $n = 2^k$.

$A(2^k) = 2A(2^{k-1}) + 1$, Substitute , $A(2^{k-1}) = 2A(2^{k-2}) + 1$

$= 2[2A(2^{k-2}) + 1] + 1$ $\quad\quad$ $A(2^{k-2}) = 2A(2^{k-3}) + 1$

$= 4A(2^{k-2}) + 2 + 1 = 2.2[2A(2^{k-3}) + 1] + 2 + 1$

$= 2.2.2A(2^{k-3}) + 4 + 2 + 1$

$= 2^3 A(2^{k-3}) + 2^2 + 2^1 + 2^0$

$= 2^3[2A(2^{k-4}) + 1] + 2^2 + 2^1 + 2^0$

$= 2^4 A(2^{k-4}) + 2^3 + 2^2 + 2^1 + 2^0 + 2^4$

$\vdots$

$= 2^i A(2^{k-i}) + [2^1 + 2^0 + 2^3 + 2^4 + \ldots\ldots + 2^k]$

put $k - i = 0$.

$k = i$

Scanned with CamScanner

$= 2^k A(2^0) + [2^{k+1} - 1]$

$= 2^k \cdot 0 + (2^{k+1} - 1)$

$\because 2^k \cdot 2 - 1 = 2n - 1 \approx 2n$

$\therefore \boxed{A(n) \approx 2n \in \Theta(n)}$

- $C(n) = 2C(n/2) + (n-1)$, $C(1) = 0$ $[n-1 = n]$.

$C(2^k) = 2C(2^{k-1}) + (2^k - 1)$, Substitute, $C(2^{k-1}) = 2C(2^{k-2}) + (2^{k-1} - 1)$

$= 2[2C(2^{k-2}) + (2^{k-1} - 1)] + (2^k - 1)$. $C(2^{k-2}) = 2C(2^{k-3}) + (2^{k-2} - 1)$

$= 2^2 C(2^{k-2}) + 2(2^{k-1} - 1) + (2^k - 1)$

$= 2^2 \cdot [2C(2^{k-3}) + (2^{k-2} - 1)] + 2(2^{k-1} - 1) + (2^k - 1)$.

$= 2^3 C(2^{k-3}) + 2^2(2^{k-2} - 1) + 2(2^{k-1} - 1) + (2^k - 1)$

$= 2^3 C(2^{k-3}) + 2^2(2^{k-2} - 1) + 2^1(2^{k-1} - 1) + 2^0(2^k - 1)$

$\vdots$

$= 2^i C(2^{k-i}) + 2^{i-1}(2^{k-(i-1)} - 1) + 2^{i-2}(2^{k-(i-2)} - 1) \ldots$

Let $k - i = 0$

$i = k$.

$= 2^k C(2^0) + 2^{k-1}(2 - 1) + 2^{k-2}(2^2 - 1) + \ldots$

$= 2^k C(1) + (2^k - 2^{k-1}) + (2^k - 2^{k-2}) + \ldots$

$[k \cdot 2^k + 2^k + 2^k + \ldots \ldots n \cdot 2^k] - [2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 2^0]$

$\rightarrow C(2^k) = 2C(2^{k-1}) + 2^k$

$= 2[2C(2^{k-2}) + 2^{k-1}] + 2^k$

$= 2^2 C(2^{k-2}) + 2^k + 2^k$

$\vdots$

$= 2^i C(2^{k-i}) + i \cdot 2^k$.

$2^k C(1) + k \cdot 2^k$.

$2^k \cdot k$.

$n \log n$. $\boxed{\therefore C(n) = n \log_2 n}$

Scanned with CamScanner

1|02|19

- **Algorithm Merge(B[0,.....p-1], C[0....q-1], A[0....p+q-1])**

// Merges two sorted array into one sorted array.
// Input: An array B[0....p-1] and C[0....q-1] both sorted.
// Output: Sorted array A[0.... p+q-1] of the elements of B and C.

```
i ← 0; j ← 0; k ← 0
while(i < p and j < q) do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i+1
    else
        A[k] ← C[j]; j ← j+1
    k ← k+1
if i=p
    copy C[j...q-1] to A[k.... p+q-1]
else
    copy B[i.....p-1] to A[k.... p+q-1]
```

→ **Analysis**

Input ← n
Basic operation ← comparison

$C_{worst}(n) = 2 \cdot C(n/2) + (n-1)$ for $n \geq 1$ and $C(1) = 0$.

$a = 2, b = 2, f(n) = n-1$ Complexity class for n^d

$b^d = 2^1 = 2$ $n^d = n^1$

i.e. $n^d \approx n \Rightarrow d = 1$.

$a = b^d$

i.e.

$\therefore C(n) \in \Theta(n^d \log n)$

$\in (n \log n)$

$\therefore C(n) \in \Theta(n \log n)$

Scanned with CamScanner

3/02/19

Quick Sort:

Eg: n=8

25 56 78 -6 18 8 46 68

pivot i j j

pivot > A[j]

pivot < A[i] swap

i < j swap.

i = j pivot → A[j]

25 8 18 -6 78 56

j

-6 8 18 | 25 | 78 56 46 68

j i j j pivot i j j partitions

pivot

-6 | 8 18 | 78 | 68 56 46

pivot i j pivot i j

8 | 18 | 68 | 46 56

pivot i j

46 | 56

Sorted array: -6 8 18 25 46 56 68 78.

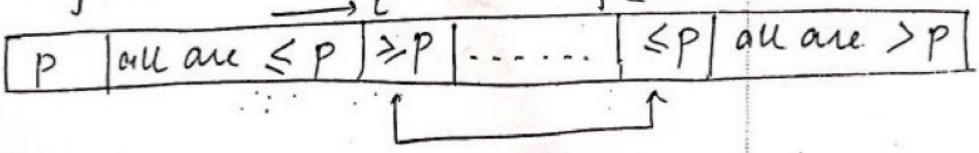
General form of Quick Sort:

A[0] A[1] ..... A[n-2] A[n-1].

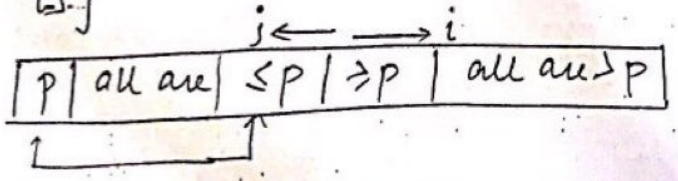
A[0] ..... A[s-1] | A[s] | A[s+1] ..... A[n-1]

< A[s] > A[s]

i) i < j



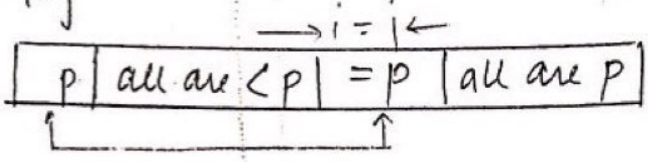
ii) i = j



time for partitioning

Scanned with CamScanner

iii) $i = j$



time for partitioning

- **Algorithm QuickSort(A[l....r])**

// Sorts a subarray by Quick sod.

// Input: A subarray A[l...r] of 0 to n-1 defined by left & right indices l and r.

// Output: Sub array A[l...r] sorted in nondecreasing order.

```
if l < r
    s ← partition(A[l...r)] // s is split position
    QuickSort(A[l...s-1])
    QuickSort(A[s+1....r])
```

- **Algorithm Partition(A[l...r])**

// Partitions a subarray by considering it's first element as pivot.

// Input: A subarray A[l...r] of 0 to n-1 define by its left and right indices l and r.

// Output: A partition A[l..r] with the split position n returned as this function's value.

```
p ← A[l]
i ← l; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ p && i ≤ r
    repeat j ← j-1 until A[j] ≤ p && j ≥ L
    swap A[i] and A[j]
until i ≥ j
swap A[i] and A[j]  //undo last swap when i ≥ j
swap A[l] and A[j]
return j
```

[repeat until loop → when cond° becomes true it comes out of loop. whereas in do while when cond° is false then it comes out].

Scanned with CamScanner

04/02/19

n = 8

0	1	2	3	4	5	6	7
45	72	56	7	19	-2	65	35

pivot = 45 (i)

45 35 56 7 19 -2 65 72

A[i] ≥ P, stops, 56 > 45

45 35 -2 7 19 56 65 72

A[i] > P, j = 5

swap 19 and 56

pivot = 45

19	35	-2	7	45	56	65	72

i = 0, pivot

A[i] < P, 7 < 19, swap

19 7 -2 35 45 56 65 72

-2	7	19	35	45	56	65	72

pivot, A[i] ≥ P

-2 7 19 35 45 56 65 72

Sorted array: -2 7 19 35 45 56 65 72

Scanned with CamScanner

→ Analysis

Input size ← n

Basic operation ← Comparison

$C_{best}(n) = 2\,C_{best}(n/2) + n$ for all $n \geq 1$

$C_{best}(1) = 0$.

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

Here, $a = 2$, $b = 2$ $f(n) = n \in \theta(n^d)$

$d = 1$.

$b^d = 2^1 = 2$

$C_{best}(n) \in \theta(n^d \log n)$

$C_{best}(n) \in \theta(n \log n)$

$C_{worst}(n) = (n+1) + n + (n-1) + \ldots + 3$.

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

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

$\approx \frac{n^2}{2} \in \theta(n^2)$.

$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} \left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)\right]$.

(1st partition; pivot)

$C_{avg}(1) = 0$ $\forall\ n \geq 1$

$\therefore C_{avg}(n) = 2n \log_e n$

$\approx 1.38 n \log_2 n$

Scanned with CamScanner

05/02/19

- Binary Search

Algorithm BinarySearch(A[0...n-1], k) [Iterative]

// Searches for a given key element k for an array A, non-recursively.

// Input: An array A[0...n-1] sorted in ascending order and a search key k.

// Output: An index of the element that is equal to k or -1.

```
l ← 0 ; h ← n-1
while (l ≤ h) do
    mid ← [(l+h)/2]
    if A[mid] = k  return mid
    if A[mid] > k  h ← mid-1
    else  l ← mid+1
return -1.
```

10 15 22 35 41 52 64

l = 5

mid = 7.

A[7] = 64 = k

l = mid+1 = 8.

h = 9.

Algorithm BinarySearch(k, A, l, h) {Recursive}

```
if l > h  return -1
mid ← [(l+h)/2]
if A[mid] = k  return mid
if A[mid] > k
    return BinarySearch(k, A, l, mid-1)
else
    return BinarySearch(k, A, mid+1, h)
```

C(n): C(n/2)+1.

C(1) = 1

Scanned with CamScanner

→ Analysis

Input size ← n

Basic operation ← comparison

Complexity : $C(n/2)$ because we check half part not $2C(n/2)$.

$C(n) = C(n/2) + 1$ ← one comparison $\forall n > 1$

& $C(1) = 1$ ← bcoz we do 1 verification with min element.

$\Rightarrow C(n) = C(n/2) + 1$ and $C(1) = 1$, if $n > 1$.

Put $n = 2^k$. [By backward subs°].

Sol°: Here, $a = 1$, $b = 2$, $f(n) = 1 \in \theta(n^d)$

$\therefore d = 0$.

$b^d = 2^0 = 1$

$a = 1 = b^d$

$C(n) = \theta(n^d \log_2 n)$

$\in \theta(\log_2 n)$

$\boxed{\therefore C(n) \in \theta(\log_2 n)}$

By Backward :

Put $n = 2^k$.

$C(2^k) = C(2^{k-1}) + 1$, Substitute $C(2^{k-1}) = C(2^{k-2}) + 1$

$= C(2^{k-2}) + 1 + 1$, Substitute $C(2^{k-2}) = C(2^{k-3}) + 1$

$= C(2^{k-3}) + 1 + 1 + 1$

$\vdots$

$= C(2^{k-i}) + i$

Put $k - i = 0$

$k = i$

$= C(2^{k-k}) + k.$

$= C(2^0) + k. = C(1) + k$

$C(2^k) = 1 + k$

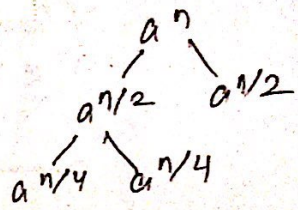
$C(n) = 1 + \log_2 n$

$\in \theta(\log_2 n)$

Scanned with CamScanner

DECREASE & CONQUER

1. Decrease by a constant value — Decrease by one
2. " ~~do~~ " constant factor — Divide and conquer
3. variable size decreases



a^n

Brute force: $\underbrace{a \times a \times a \times \cdots \times a}_{n \text{ times}}$

→ Divide and conquer

$$\begin{cases} a^{n/2} \times a^{n/2} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

→ Decrease by one

$$\begin{cases} a^{n-1} * a & \text{if } n > 1 \\ a & \text{if } n < 1 \end{cases}$$

→ Decrease by a constant factor

$$\begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even} \\ (a^{n-1/2})^2 \times a & \text{if } n \text{ is odd} \\ a & \text{if } n = 1 \end{cases}$$

Eg: $a^8 = (a^4)^2$

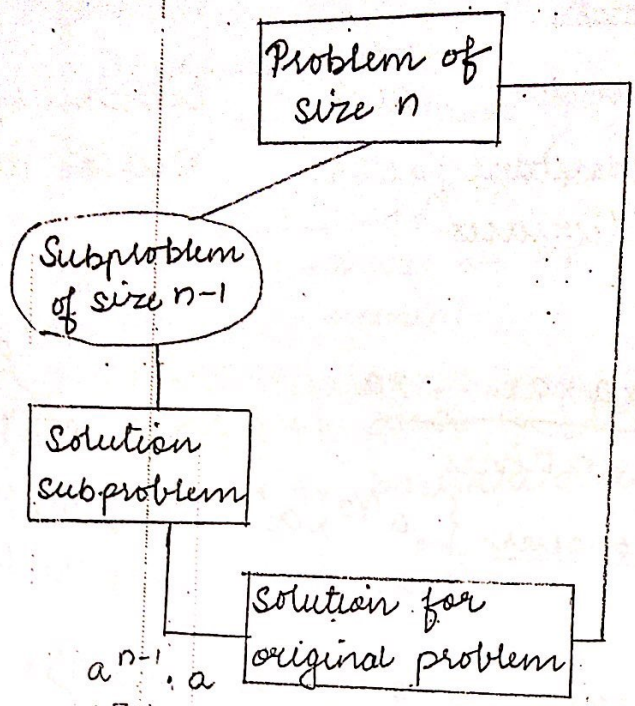
$a^9 = (a^4)^2 \cdot a$

→ variable size decrease:

Eg: gcd using Euclid's algorithm

$gcd(m, n) = gcd(n, m \bmod n)$

06/02/19



n=1
a=2

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

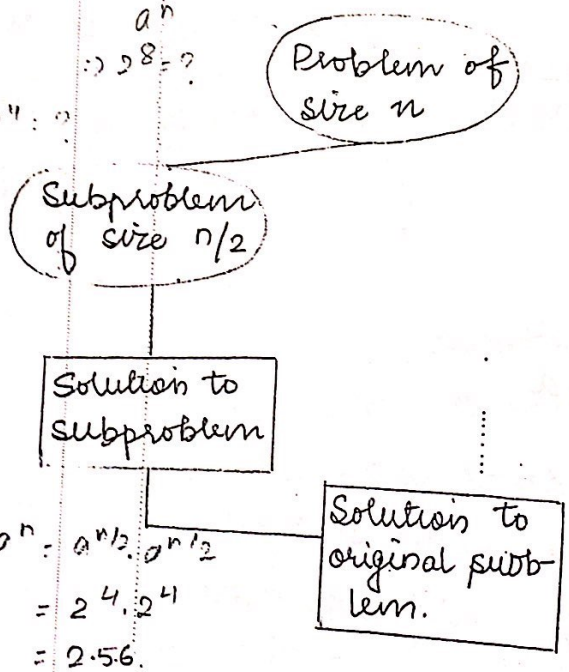
$2^3 \cdot 2$

$= 2^4 = 16$

a^n

$\Rightarrow 2^8 = ?$

$2^{n/2} : 2^4 : ?$



$2^4 \cdot 16$

$a^n = a^{n/2} \cdot a^{n/2}$

$= 2^4 \cdot 2^4$

$= 256.$

Decrease by one.

Insertion Sort

n = 8 Passes = n-1

	56	45	-10	62	35	43	18	5
v: 45	45	56	-10	62	35	43	18	5
	-10	45	56	62	35	43	18	5
	-10	45	56	62	35	43	18	5
	-10	35	45	56	62	43	18	5
	-10	35	43	45	56	62	18	5
	-10	18	35	43	45	56	62	5
	-10	5	18	35	43	45	56	62

- Algorithm InsertionSort(A[0...n-1])

// Sorts a given array by Insertion Sort.
// Input: An array of A[0.....n-1] of n elements.
// Output: An array of A[0.....n-1] sorted in non decreasing order.

```
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

Tracing:

0	1	2	3	4	5	6	7
78	62	-5	23	16	35	28	8
62	78	-5	23	16	35	28	8
-5	62	78	23	16	35	28	8
-5	23	62	78	16	35	28	8
-5	16	23	62	78	35	28	8
-5	16	23	35	62	78	28	8
-5	16	23	28	35	62	78	8
-5	8	16	23	28	35	62	78

→ Analysis:

Input size ← n

Basic operation ← comparison.

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1-1+1 = n-1 \in \Theta(n)$$

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

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

$$= 1+2+\ldots n-1$$

$$= \frac{(n-1)(n)}{2} \approx \frac{n^2}{2} \in \Theta(n^2).$$

Average case Analysis:

$$C_{avg}(n) = \sum_{i=1}^{n-1} \left(\frac{i+1}{2}\right)$$

$$= \frac{1}{2} \sum_{i=1}^{n-1} (i+1) = \frac{1}{2}(2+3+4+\ldots\ldots+n).$$

$$\therefore C_{avg}(n) = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) = \frac{1}{2}\left(\frac{n^2}{2}\right) = \frac{n^2}{4}.$$

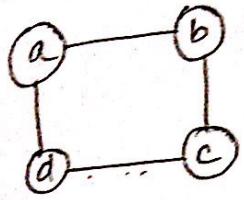
use this when the order is almost sorted.

08/02/19

GRAPH

$G = (V, E)$

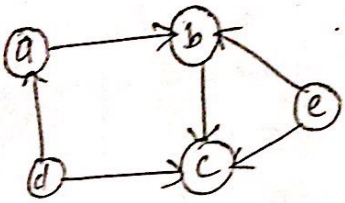
G_1:



$V = \{a, b, c, d\}$

$E = \{(a,b), (a,d), (b,c), (c,d)\}$

G_2:



$V = \{a, b, c, d, e\}$

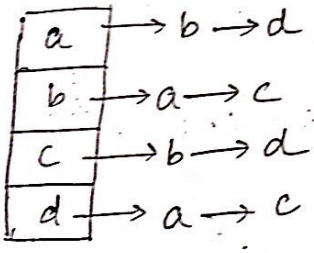
$E = \{(a,b), (b,c), (c,a), (d,a), (d,c), (c,b), (e,c)\}$.

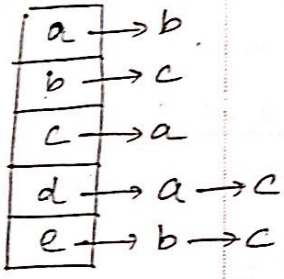
Adjacency matrix of G_1 & G_2:

	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

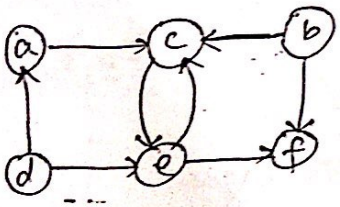
	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	0	0
c	1	0	0	0	0
d	1	0	1	0	0
e	0	1	1	0	0

Linked list representation of G_1 & G_2 (Adjacency list):





[G_1 matrix representation is symmetric on its principal diagonal & is undirected].



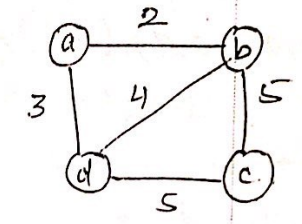
$V = \{a, b, c, d, e, f\}$

$E = \{(a,c), (b,c), (b,f), (c,e), (e,c), (e,f), (d,e), (d,a)\}$.

Adjacency matrix:

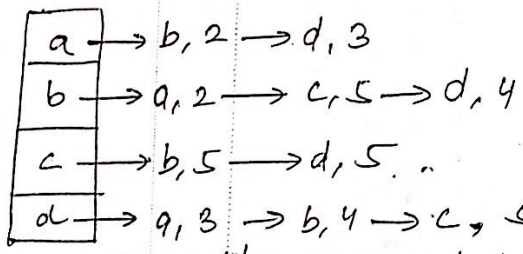
	a	b	c	d	e	f
a	0	0	1	0	0	0
b	0	0	1	0	0	1
c	0	0	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	0	0	1
f	0	0	0	0	0	0

⇒



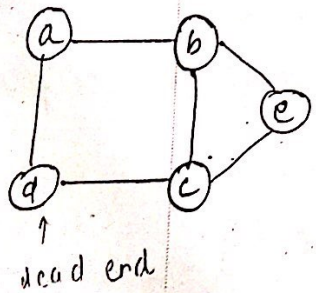
	a	b	c	d
a	0	2	∞	3
b	2	0	5	4
c	∞	5	0	5
d	3	4	5	0

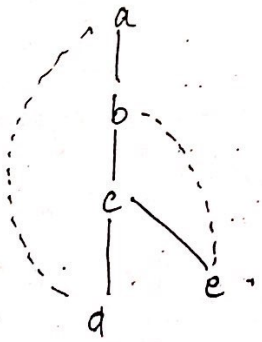
linked list representation

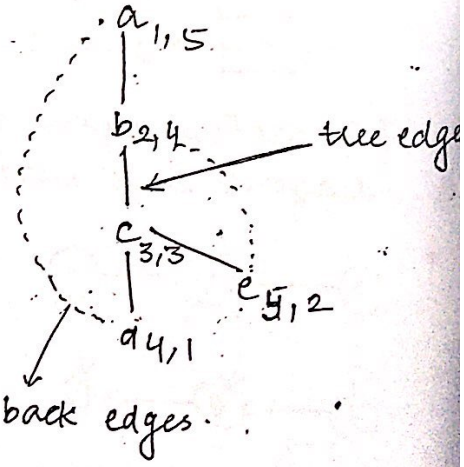


[To traverse vertices or edges of a graph in systematic fashion.

DFS [Depth First Search] & BFS [Breadth First Search]

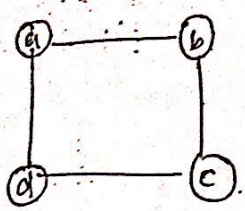




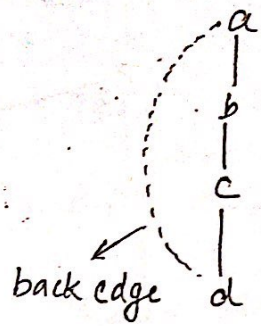


a 1,5
b 2,4 ...
c 3,3 stack.
d 4,1
e 5,2

[To solve DF's, stack data structure is used]



DFS' Tree



Push, pop

a 1,4

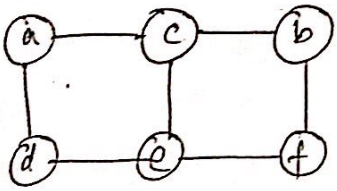
b 2,1

c 3,2

d 4,1

[Back edge signifies cycle]

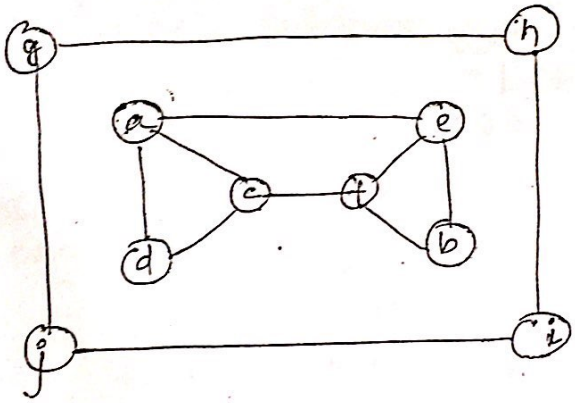
1).



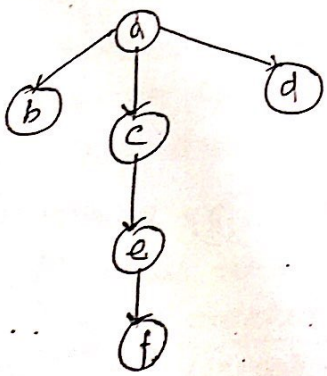
① Starting vertex 'a'

② Starting vertex 'e'.

2).

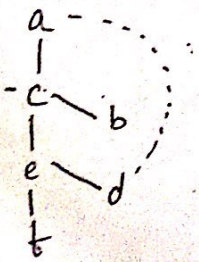


3).

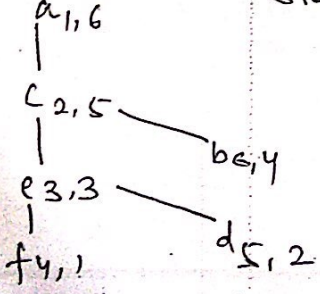


Answers:

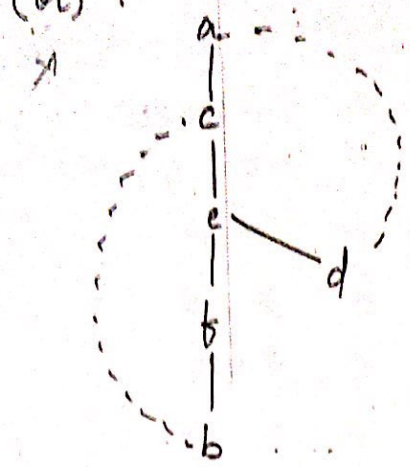
1. i).

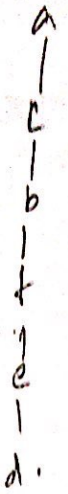


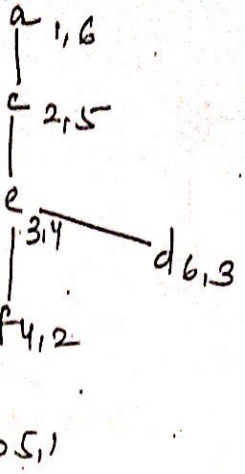
Starting vertex 'e'.



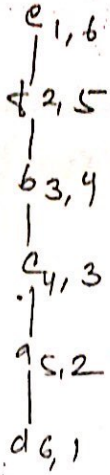
(or)



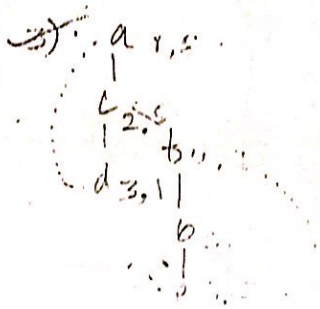




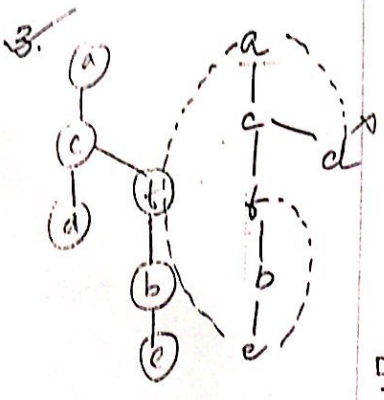
(or)



Starting vertex 'e':



3.

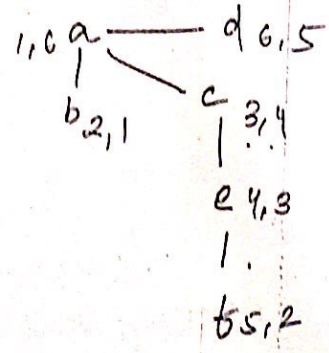


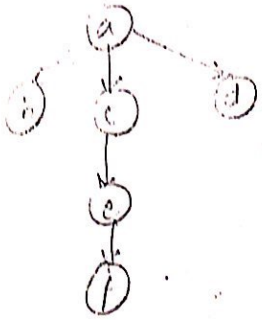
a 1,6
c 2,5
d 3,1
f 4,4
b 5,3
c 6,2

g 7,10
h 8,9
i 9,8
j 10,7

DF's forest

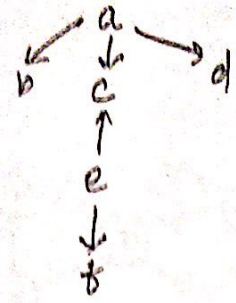
4.

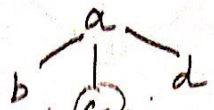




alphabetical order

11/02/19





c is not ancestor for e.
∴ back edge should not be drawn.
DF's forest.

a 1, 4
b 2, 1
c 3, 2
d 4, 3
e 5, 6
f 6, 5

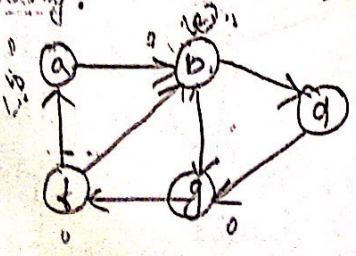
* Algorithm DFS(G)

// Implements DF's traversal of a given graph.
// Input : Graph G is equal to (V, E).
// Output: Graph G with its vertices marked with consecutive integers in the order they have been encountered by DF's traversal.

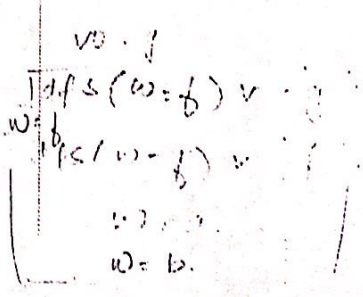
mark each vertex v in V with '0' as a mark of being "unvisited"
count ← 0
for each vertex 'v' in V do.
    if v is marked with 0
    dfs(v)

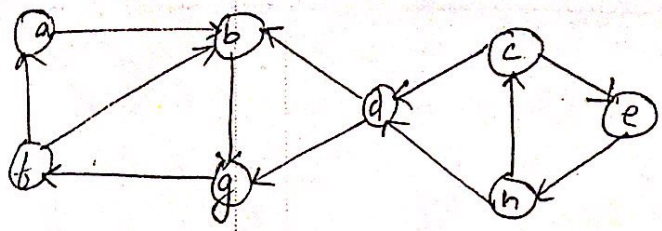
• Algorithm dfs(v)
// visits recursively all the unvisited vertices connected to vertex v by a path and numbers them in order they are encounter, via global variable count.
count ← count + 1 ; mark v with count
for each vertex w in V adjacent to v do
    if w marked with 0
        dfs(w).

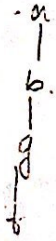
Tracing:



V = {a, b, d, f, g}
count = 0, 1, 2, 3, 4, 5
dfs(v = a)
v = 'a'
w = b
dfs(w = b) v = 'b'
w = g
w = d
"dfs(w = d) v = 'd'"



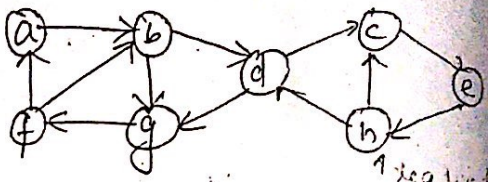




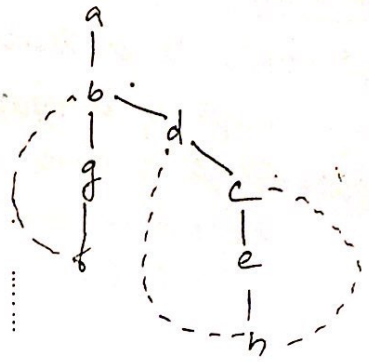
$V = \{a, b, c, d, e, f, g, h\}$.

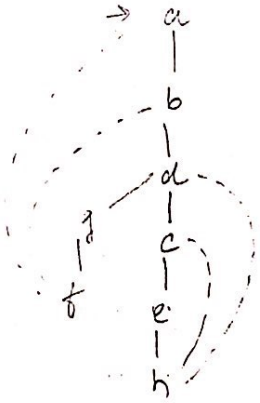
```
dfs(v: a)
v: 'a'
count = 1
dfs(w: 'b') v: b
w: g
dfs(w: 'g') v: g
w: b
dfs(w: 'b') v: f
w: b
dfs(w: 'f') v: f
w: a
w: b
dfs(w: 'd') v: d
w: a
w: b
dfs(w: 'g') v: g
(w: b)
w: g
```

2)



DFS' forest.



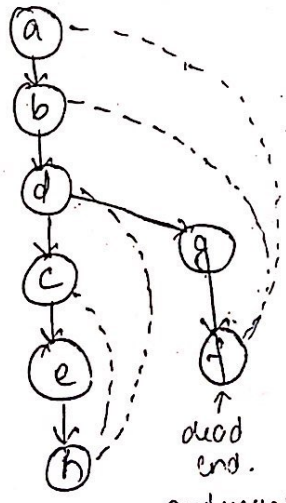


c 1, 8
b 2, 7
d 3, 6
c 4, 5
e 5, 2
h 6, 1
g 7, 5
b 8, 4.

Whether they are predecessor not a parent

a 1, 4
b 2, 3
g 3, 2
f 4, 1
c 5, 8
d 6, 5
e 7, 7
h 8, 6

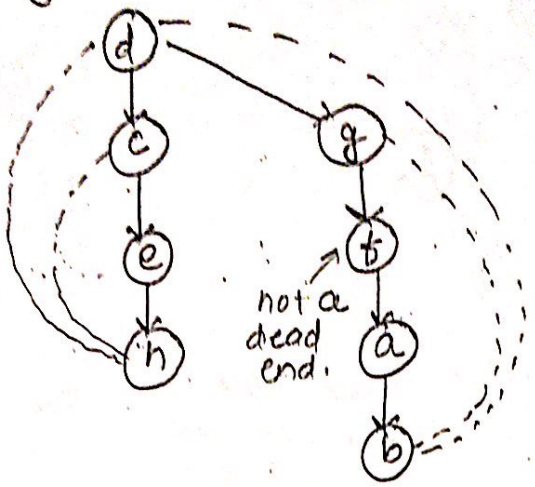
Starting vertex 'a'.



a 1, 8.
b 2, 7
d 3, 6
c 4, 3
e 5, 2
h 6, 1
g 7, 5
f 8, 4

12/02/19

2) starting vertex 'd'.



d 1, 8
c 2, 3
e 3, 2
h 4, 1
g 5, 7
f 6, 6
a 7, 5
b 8, 4

DFS' forest.

[Connected graph as we can reach all the vertices from 'd'].

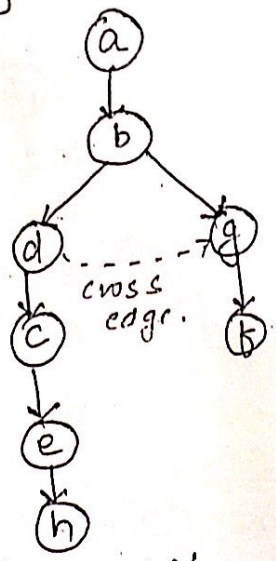
Here $|V| = 8$.

- Adjacency matrix: $\Theta(|V|^2)$
- Adjacency list: $\Theta(|E|+|V|)$

} both BFS' & DFS

BFS: [Question 2].

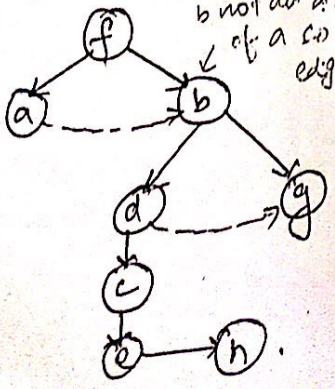
2) starting vertex 'a'.



a 1, ① ← same order.
b 2
d 3
g 4
c 5
~~f 6~~ e a
e 7 h 7
~~h 8~~ f 8

a	b	d	g	c	e

Starting vertex 'f'.



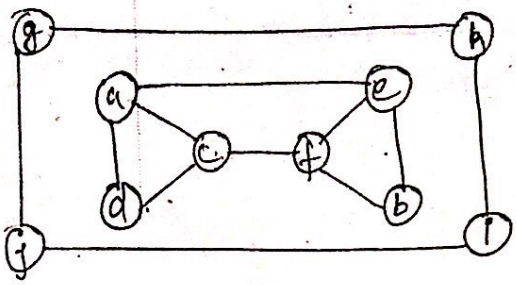
BFS trees.

f 1
a 2
b 3
d 4
g 5
c 6
e 7
h 8

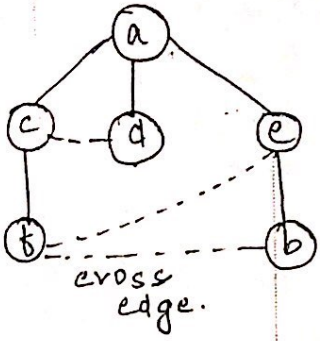
If some nodes not visited then forest

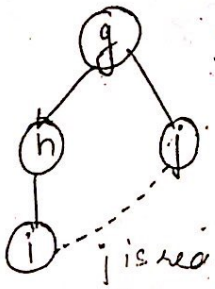
Scanned with CamScanner

Q.



[Queue data structure is used for BFS]



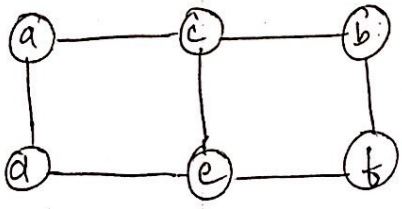


a 1
c 2
d 3
e 4
f 5
b 6

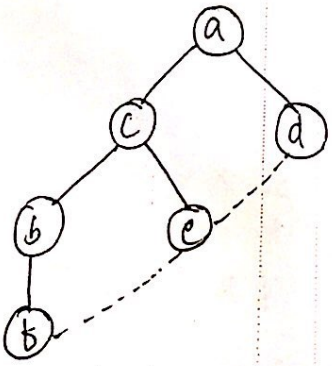
g 7
h 8
j 9
i 10

BFS forest.

Q1.

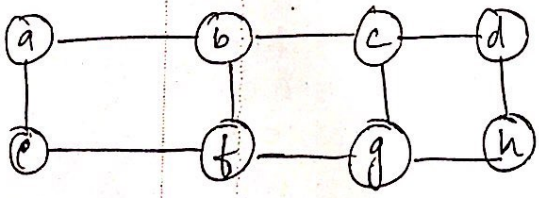


Starting vertex 'a'.

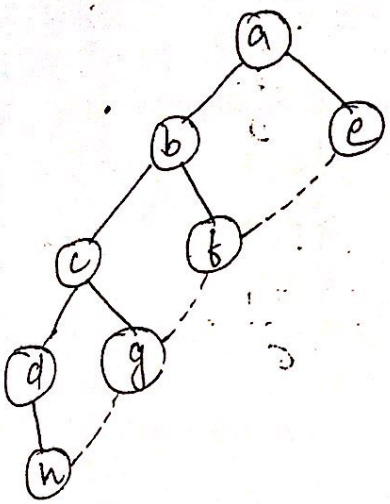


a 1
c 2
d 3
b 4
e 5
f 6

Q2.

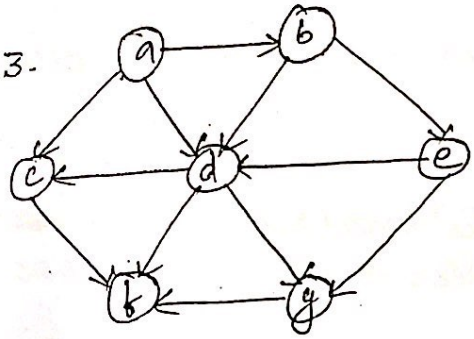


Starting vertex 'a'.

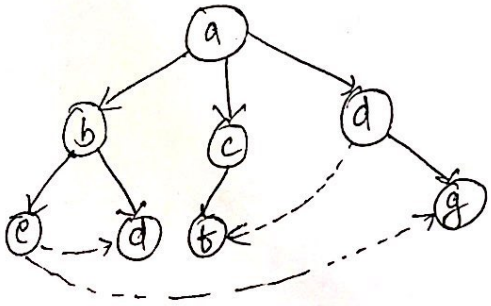


a 1
b 2
e 3
c 4
f 5
d 6
g 7

Q3.

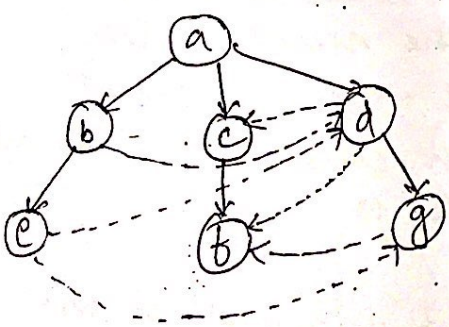


Starting vertex 'a'
Starting vertex 'd'.



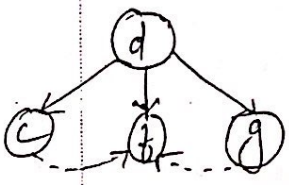
a 1
b 2
c 3
d 4
e 5
d 6
f 7
g 8

d 1
c 2
f 3
g 4
a 5
b 6
e

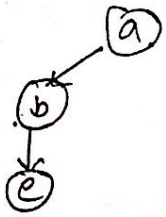


BFS Tree.

a 1
b 2
c 3
d 4
e 5
f 6
g 7

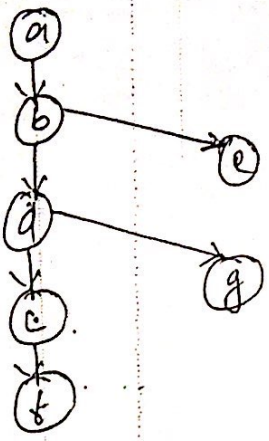


BFS forest.



DFS

Q. 3.



a 1, 7
b 2, 6
d 3, 4
c 4, 2
f 5, 1
g 6, 3
e 7, 5

- Algorithm DFS(G)

// Implements breadth first search traversal of a given graph.

// Input: Graph G = (V, E)

// Output: Graph G with its vertices marked with consecutive order, they have been visited by the DFS traversal.

mark each vertex v or V with 0 as a mark of being "unvisited".

count ← 0

for each vertex v in V do

if v is marked with 0

bfs(v) ← comes here after # then for bfs

- Algorithm bfs(v)

// visits all the unvisited vertex connected to vertex v by a path and assigns them the number in order they are visited by global variable count.

count ← count+1, mark v with count and initialize a queue with v.

while queue is not empty do.

for each vertex w in V adjacent to the front vertex do

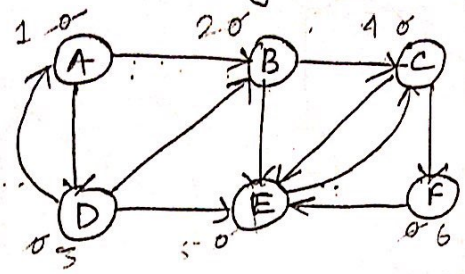
if w is marked with 0

count ← count+1, mask w with count.

below count
add w to the queue.
below if
Remove front vertex from the queue.

Eg: Consider this graph: (Tracing)

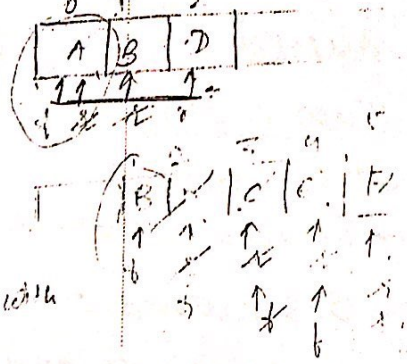
G:



V = {A, B, C, D, E, F}

V = A.
bfs(A)
W = B, D — adjacent nodes of A
W = C, E — adjacent to B.
W = A, B, G — adjacent to D
W = E, F — adjacent to C.
W = C — adjacent to E not marked with 0, so remove.
W = E

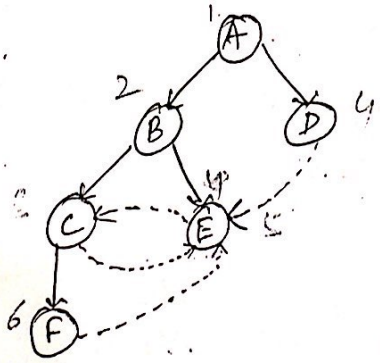
count ← 0, 1, 2, 3, 4, 5, 6



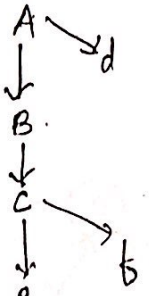
f = 1 = r
queue becomes empty

BFS tree

A 1
B 2
D 3
C 4
E 5
F 6



DFS tree



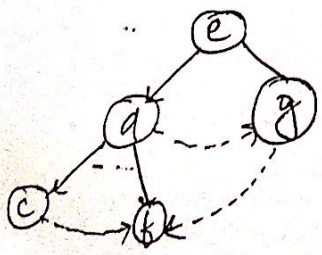
a 1, 6
b 2, 4
c 3, 3
e 4, 1
f 5, 2
d 6, 5

B not a dead end

no back edges

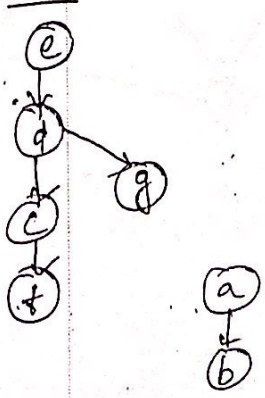
Q3:
Starting vertex 'e'.

BFS tree



e 1
d 2
g 3
c 4
b 5
a 6
b 7

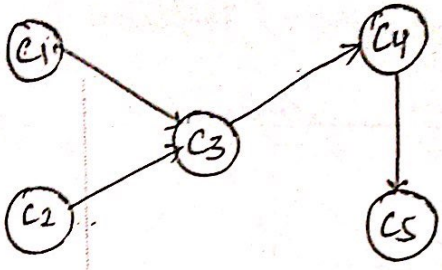
DFS:



e 1, 5
d 2, 4
c 3, 2
f 4, 1
g 5, 3
a 6, 7
b 7, 6

• Topological Sorting (Ordering)

C_1, C_2, C_3, C_4



directed graph (digraph)

directed acyclic graph (DAG) dag.

Two methods:

→ DFS method

→ Source removal method.

DPS method:

popped off order: C_5, C_4, C_3, C_1, C_2

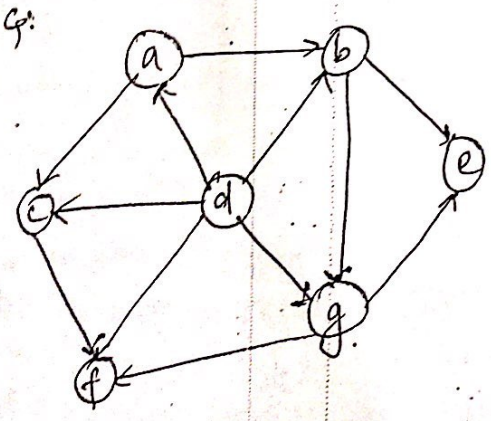
Source C_2: $C_1 \longrightarrow C_3 \longrightarrow C_4 \longrightarrow C_5$

$C_2 \to C_3 \to C_4 \to C_5$

$C_1 \to C_3 \to C_4 \to C_5$

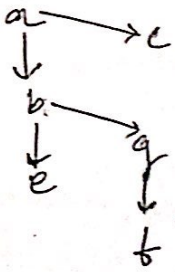
C_1,
$C_2, 5$
C_3,
$C_4, 2$
C_5 1

Q:



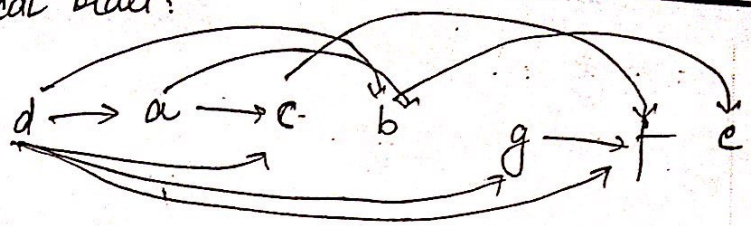
DFS

a, 1, 6.
b 2, 4
e 3, 1
g 4, 3
f 5, 2
c 6, 5
d 7, 7

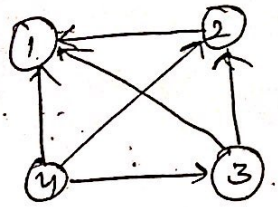


Popped off order: e, f, g, b, c, a, d

Topological order:



Q:

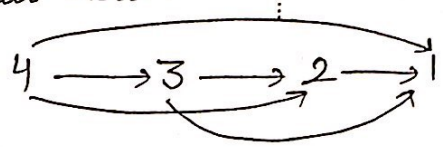


DFS.

1 1,1
2 2,2
3 3,3
4 4,4

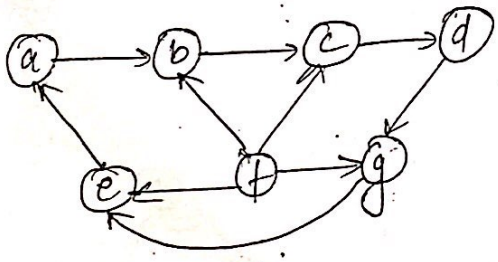
popped off order: 1 2 3 4.

Topological order:



4→3→2→1
4→2→1
4→3→1
4→1

Q:



[Since cycle appears
No topological order].

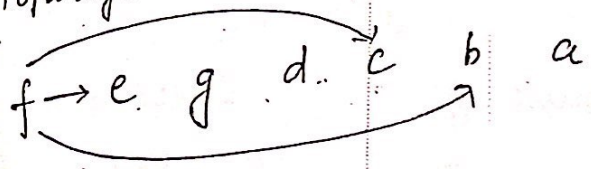
DFS

a 1,6
b 2,5
c 3,4
d 4,3
g 5,2
e 6,1
f 7,7.

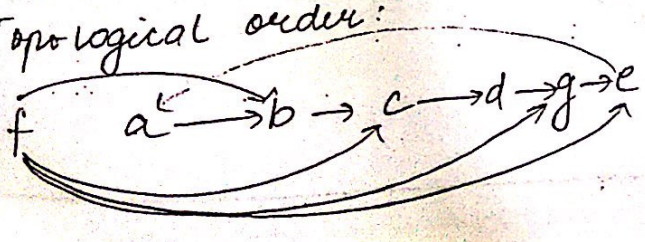
popped off order:

e g d c b a f

Topological order:

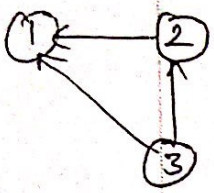


Topological order:



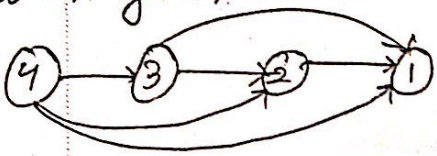
- **Source Removal method:**

Remove vertex 4 | Remove 3 | Remove 2 | Remove 1



4, 3, 2, 1.

{ 4 has no indegree, hence remove 4 first. }



Q: a → 1, 2 Indegree. Outdegree.

b → 1, 2

c → 2, 1

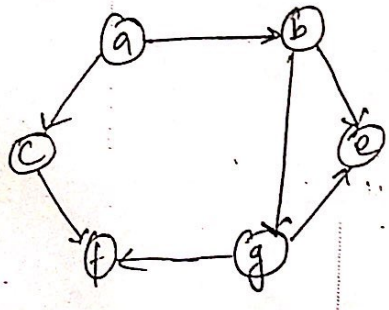
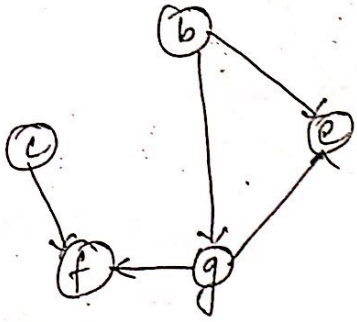
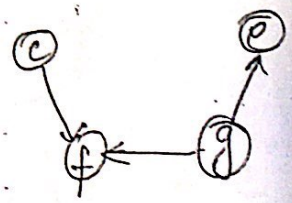
d → 0, 5

e → 2, 0.

f → 2, 0.

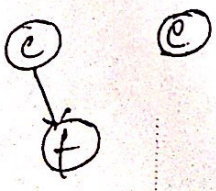
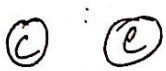
g → 2, 2.

Remove d. | Remove a. | Remove b.

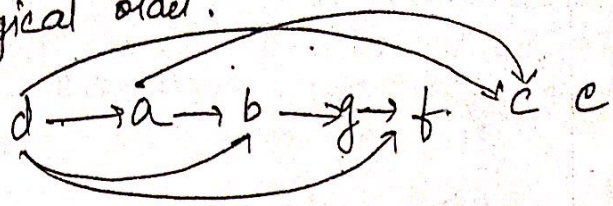




d, a, b, g, f, c, e.

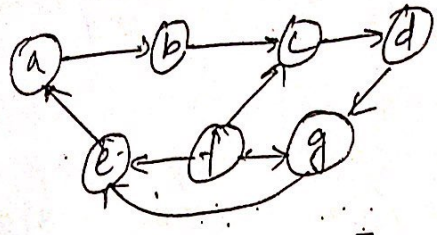
Source. g | Source. f | Source c | Source e


Topological order.



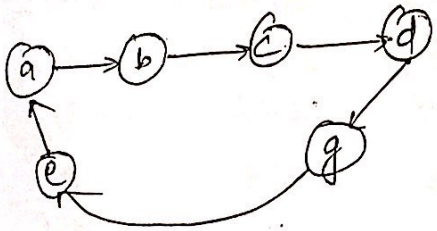
4:



f → 0, 3.
e → 2, 1.

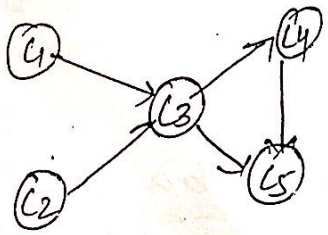
a → 1, 1
b → 1, 1
c → 1, 1
d → 1, 1
g → 1, 1

Remove f.

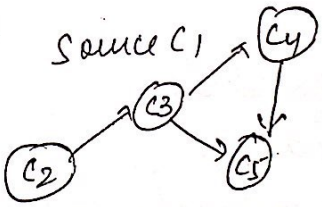


after this there is no vertex with indegree 0. ∴ no topological order exists.

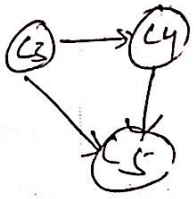
4:



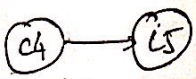
Source c_1



Source c_2



Source c_3

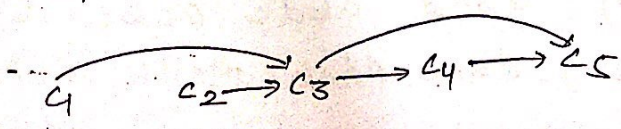


Source c_4

Source c_5.

proped order.

c_1, c_2, c_3, c_4, c_5.



15/02/19

Generating combinatorial objects -

Minimal change algorithm: n=1,2,3.

Initial : 1

Insert 2 to previous solution : 12 (RL) 21 (LR)
from right to left

Insert 3 to the previous sol' :
from right to left (12) : 123 (RL) 132 (LR) 312 (RL)
from left to right (21) : 321 (LR) 231 (RL) 213 (LL)

Insert 4 to the previous sol° : 1234 1243 1423
from right to left (123) 4123.
from left to right (132) : 4132 1432 1342 132
from right to left (312) : 3124 3142 3412 431
from left to right (321) : 4321 3421 3241 321
from right to left (231) : 2314 2341 2431 423
from left to right (213) : 4213 2413 2143 2134

Elements: a b c d

Initial: a :

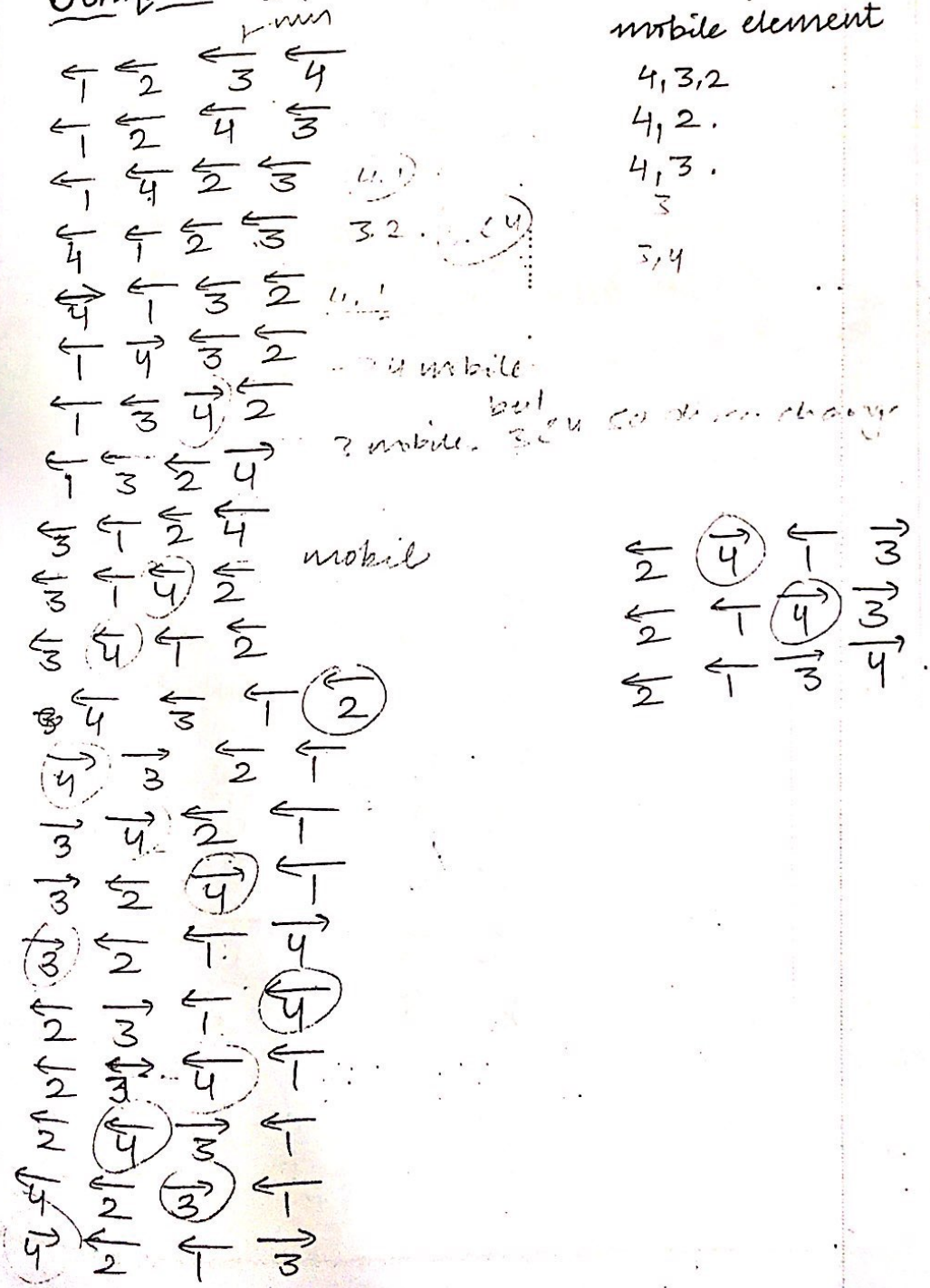
Insert b to previous sol° : ab ba
from right to left

Insert c to previous sol'
from right to left (ab) : abc acb cab
from left to right (ba) : cba bca bac

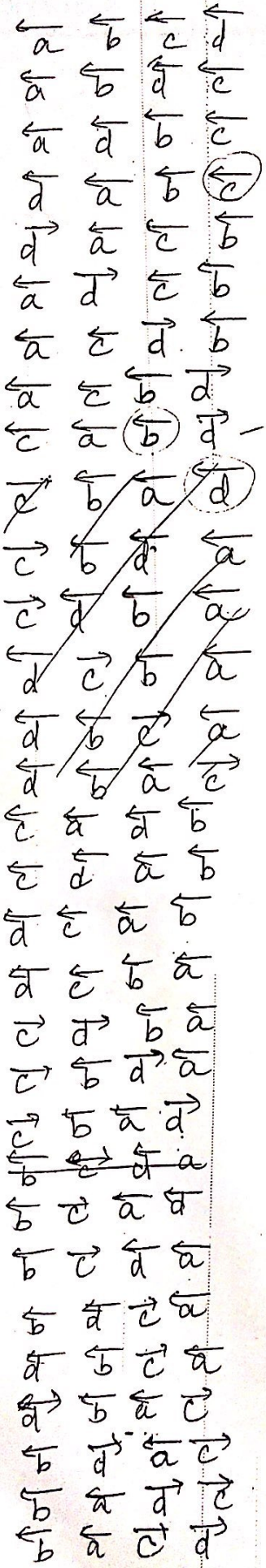
Insert d in previous
from R to L (abc) : abcd abdc adbc dabc
from L to R (in acb) : dacb adcb acdb acbd
from R to L (cab) : cabd cadb cdab dcab

from L to R (cba) : dcba cdba cbda cbad
from R to L (bca) : bcad bcda bdca dbca.
from L to R (bac) : dbac bdac badc bacd.

Johnson - Trotter Algorithm :

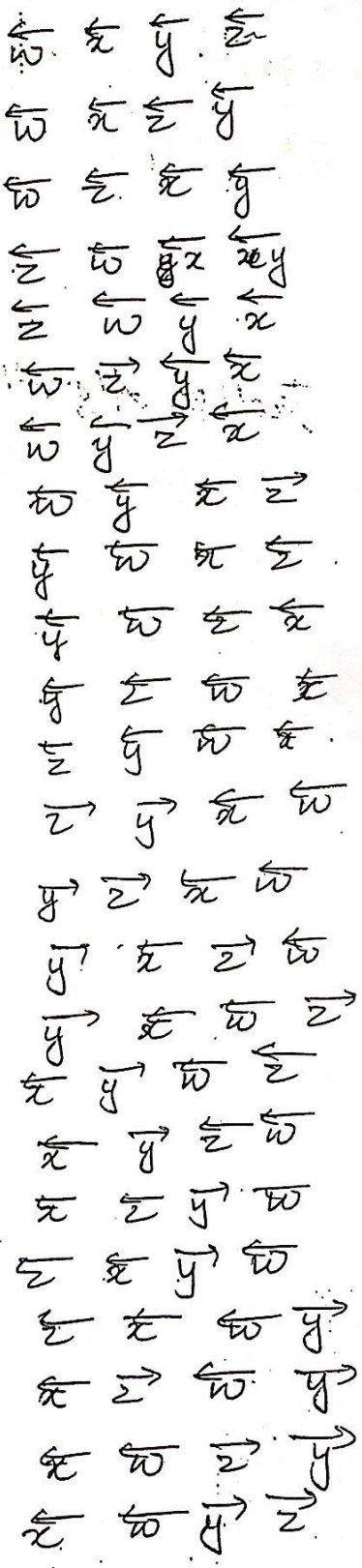


$a < b < d$



$w < x < y < z$

mobile
$z > y > x$



17/02/19

Algorithm Johnson Trotter(n).

// Implements Johnson-Trotter algorithm for generating permutation.

// Input: A positive integer n

// Output: A list of all permutation of $\{1, \ldots, n\}$

Initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$

while there is a mobile integer k do.

find the largest mobile integer k

swap k and the adjacent integer it's arrow points to

reverse the direction of all integers that all larger than k.

- Lexicographic Ordering:

If $a_{n-1} < a_n$ swap a_{n-1} and a_n, otherwise,

i) Scan current permutation from right to left looking for 1st pair of consecutive element, a_i and a_{i+1} such that $a_i < a_{i+1}$ $\forall$ $i < n$.

ii) Find the smallest element in the pair larger than a_i and put it in position i.

iii) The positions from i+1, through n are filled with the elements $a_i, a_{i+1} \ldots a_n$ from which the element written in ith position has been eliminated, in the increasing order.

Example: 123.

123 2<3

132 ✓

312

321

213 ✓ 1<3.

231 ✓

312 ✓ 1<2

321 ✓

1 2 3 4. 4>3. 3<4.

1 2 4 3.

1 3 2 4. 2<4.

1 3 4 2

1 4 2 3. 2<3.

1 4 3 2 1<4 we need to find just largest number i.e. 2.

2 1 3 4 3<4.

2 1 4 3

2 3 1 4 1<4

2 3 4 1

2 4 1 3 3<1

2 4 3 1 ——→ 4>3

3 1 2 4 ✓ 2<3.4

3 1 4 2 ✓

~~3 4 1 2 ✓ 1<4.~~

~~3 2 4 1~~ ~~3 4 2 1 1<2~~

~~3 4 2 1 3~~

3 2 1 4 just larger

3 2 4 1 1 3 4 2 4>3

3 4 1 2 1 4 2 3. just smaller.

3 4 2 1

4 1 2 3

4 1 3 2 1<3 ✓ swap by just larger.

4 2 1 3

4 2 3 1 2<3 ✓ swap

4 3 1 2

4 3 2 1

Scanned with CamScanner

• 4 6 2 5 3 1.

4 6 2 5 3 1

~~4 6 5 1 2 3.~~

~~4 6 5 1 3 2~~

4 6 3 1 2 5

4 6 3 1 5 2.

3 ≠ 1.

5 Pc → 120.

• a < b < c < d.

a b c d.

a b d c.

a c b d.

a c d b.

~~a c b d.~~

a d c b

a d b c.

b a c d

b a d c

b c a d

b c d a

b d a c

b d c a.

c a b d

c a d c

c b a d

c b d a

c d a b

c d b a

d a b c

d a c b

d b a c

d b c a

d c a b

d c b a.

w < x < y < z.

w x y z

w x z y

w y x z

w y z x.

w z x y

w z y x

x w y z

x w z y

x y x z

x y z x

x z w y

x z y w

y w x z

y w z x

y x w z

y x z w

y z w x

y z x w

z w x y

z w y x

z x w y

z x y w

z y w x

z y x w.

• Generating Subset:

$P = \{a_1, a_2, a_3\}$

r	
0	ϕ
1	ϕ $\{a_1\}$
2	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$
4	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$ $\{a_4\}$ $\{a_1, a_4\}$ $\{a_2, a_4\}$ $\{a_3, a_4\}$ $\{a_1, a_2, a_4\}$ $\{a_1, a_3, a_4\}$ $\{a_2, a_3, a_4\}$ $\{a_1 a_2 a_3 a_4\}$

Binary Reflected grey code-

minimal change algorithm

000 001 011 010 110 100 101 111

18/02/19

TRANSFER AND CONQUER

Unit-III

Transformation stage
- instance simplification
- representation change
- problem reduction.

Instance simplification:
The given problem is converted into another form.
Eg: presorting, algorithm_AVL tree.

Representation changes:
Eg: Heap sort

Problem reduction:
Eg: Computing LCM of two numbers using GCD.

Presorting Algorithm:

```
// Presorts element uniqueness in an array.
// Input: An array A[0....n-1].
    //Solves elements uniqueness problem by sorting
      the array first.
// Input: An array A[0....n-1] of orderable elements.
// Output: Returns true if A has no equal elements,
           false otherwise.

sort the array A
for i ← 0 to n-2 do
        if A[i] = A[i+1]
                return false.
return true
```

Tracing:

8
12
13
31
36

A[0] = A[1]

→ Analysis

Input size ← n

Basic operation ← comparison

Complexity :

$T(n) = T_{sort}(n) + T_{scan}(n)$

$\in \theta(n \log n) + \theta(n)$

$\in \max\{\theta(n \log n), \theta(n)\}$

$\in \theta(n \log n)$

- Searching Problem:

Binary search

Insert the statement in the previous algorithm, "Sort the array".

→ Andlysis

complexity :

$T(n) = T_{Sort}(n) + T_{search}(n)$

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

without sorting complexity.

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

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

- Compute a mode:

Algorithm

- Searching Problem:

Binary Search

Insert the statement in the previous algorithm "sort the array".

→ Analysis

Complexity :

$T(n) = T_{sort}(n) + T_{search}(n)$

$\in \theta(n \log n) + \theta(n)$

$\in \max\{\theta(n \log n), \theta(n)\}$

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

• Algorithm Q(n).
if n=1 return 1.
else return Q(n-1)+3.

→ A(n) = A(n-1) (+1) ∀ n>1.
A(1) = 0.

~~Element~~ uniqness we should do by Bruteforce method.

because addition is done only once.. one add.
+3=1
+3+3 = 2 - two add.

• Algorithm presentMode(A[0... n-1])
Sort the array A
j ← 0.

• Computing a mode-
The number which has more frequency of occurance.

Eg: 1 2 5 1 3 6 5 2 1 5 2 5
mod: 5

sort
1 1 1 2 2 2 3 5 5 5 6
without sorting.

→ Analysis
Complexity:

$$\sum_{i=1}^{n} (i-1) = 0+1+2+\cdots+(n-1) = \frac{n(n-1)}{2} \in \theta(n^2) + \theta(n)$$

with sorting
$T(n) \in \theta(n \log n)$

$\therefore T_{sort}(n) + T_{scan}(n)$

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

↑
max.

Computing a mode -.

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

{A mode-value that occurs most often in a given list of numbers.}

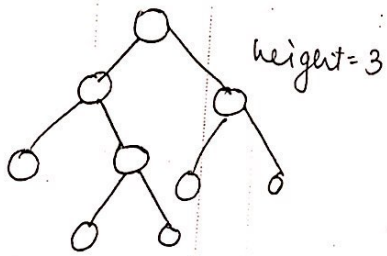
19/02/14

* Algorithm Presort-Mode(A[0...n-1])

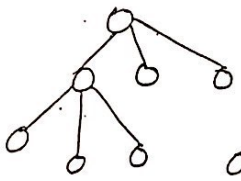
// Compute the mode of an array by sorting it first
// Input: Array A[0...n-1]
// Output: mode of array.

Sort the array A
i ← 0
modefrequency ← 0
while i ≤ n-1 do
    runlength ← 1, runvalue ← A[i]
    while i+runlength ≤ n-1 and A[i+runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength
        modevalue ← runvalue
    i ← i + runlength.
return mode_value

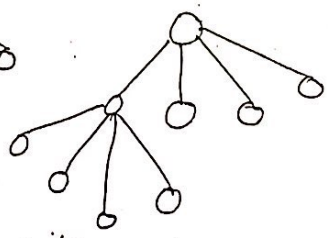
* Balanced search Tree:



Binary tree



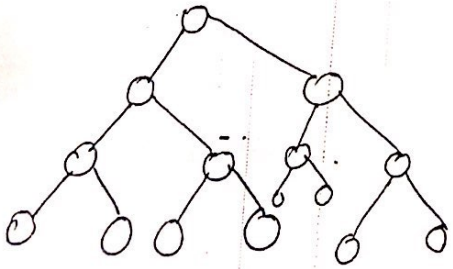
Ternary tree



4-ary tree

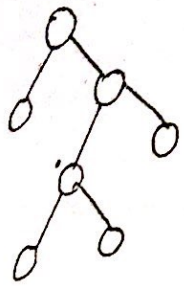
m-ary.

Trees arranged in hierachical manner.

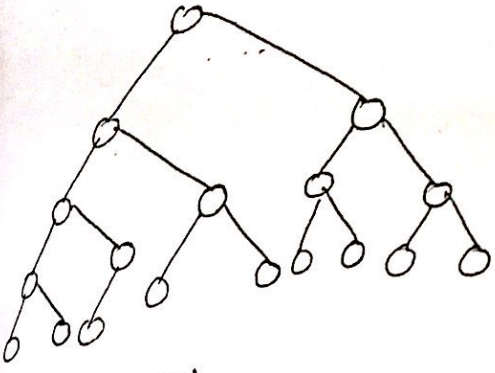


Complete Binary tree
no. of nodes = $2^{n+1} - 1$

$n = [\log_2 n] + 1.$



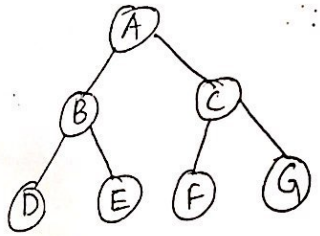
strictly binary tree
No. of ~~non~~ children must be 0 or 2.



almost complete b.t.
upto h-1,
It is complete and nodes will present form left order.

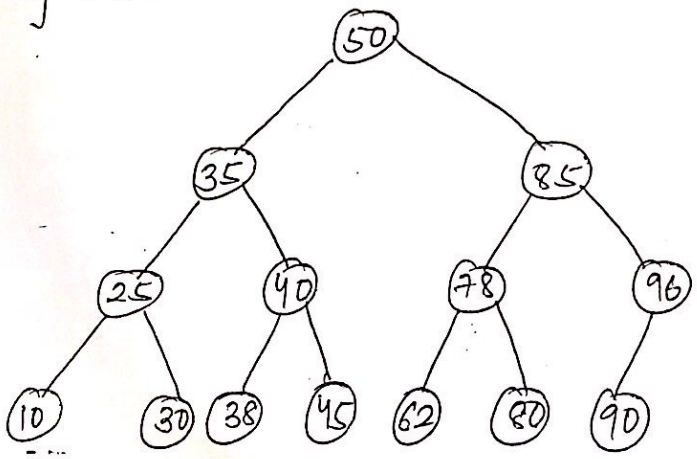
Traversal

→ Preorder : root → left → right
→ Inorder : left → root → right
→ post order : left → right → root



Preorder: A B D E C F G
Inorder: D B E A F C G
Postorder: D E B F G C A

• Binary search trees -



Preorder : 50, 35, 25, 10, 30, 40, 38, 45, 85, 78, 62, 80, 96, 90.

Inorder : 10, 25, 30, 35, 38, 40, 45, 50, 62, 78, 80, 85, 90, 96.

Postorder : 10, 30, 25, 38, 45, 40, 35, 62, 80, 78, 90, 96, 85, 50.

Balanced Search Tree -

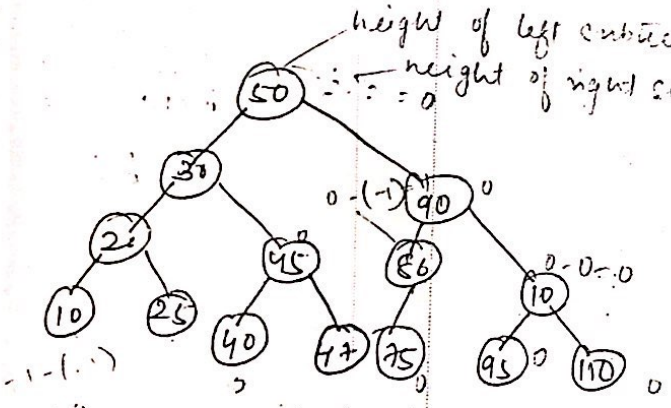
There are balanced binary search tree.

AVL Tree :

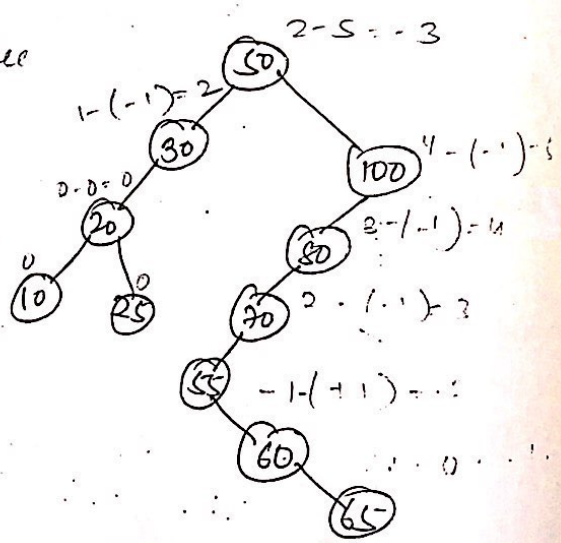
It is a balanced search tree.

AVL → Adelson, Velskey & Landis

AVL tree is binary search tree in which balance factor of every node, which is defined as the difference between the height of node's left and right subtrees is either 0, +1 or -1. (The height of empty tree is defined as -1)

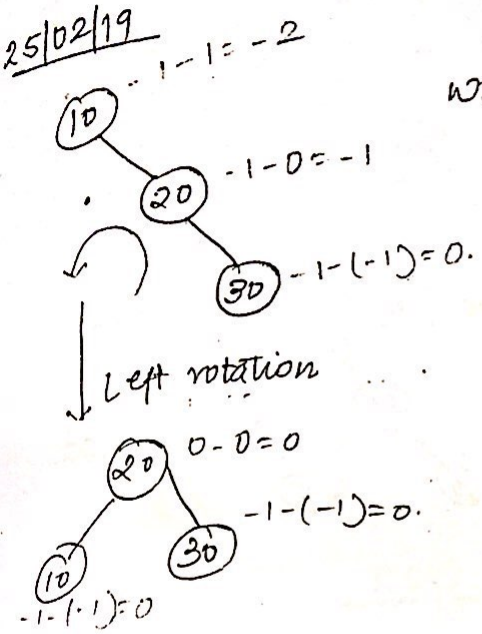


AVL tree.

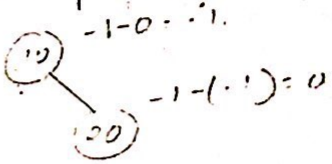


Not an AVL tree.

25/02/19



What is the balance factor?



Singe rotation

Single L- rotation

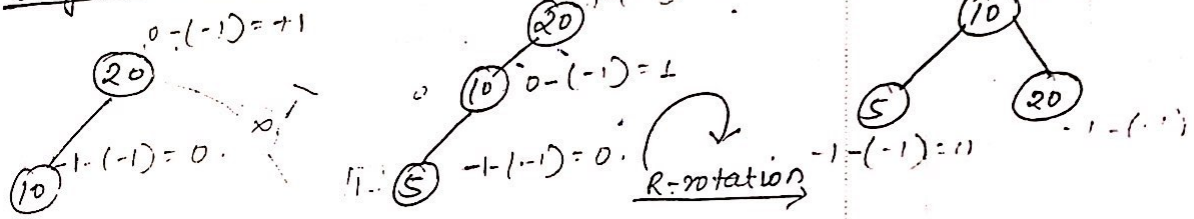
Single R-rotation

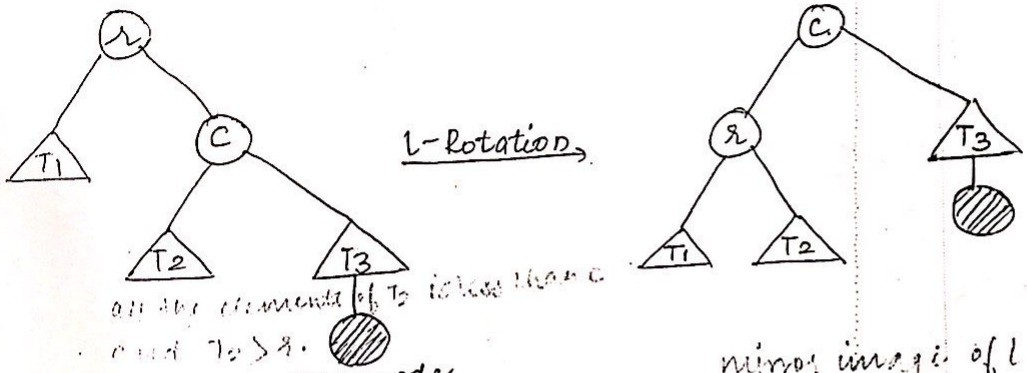
Double rotation

LR - rotation

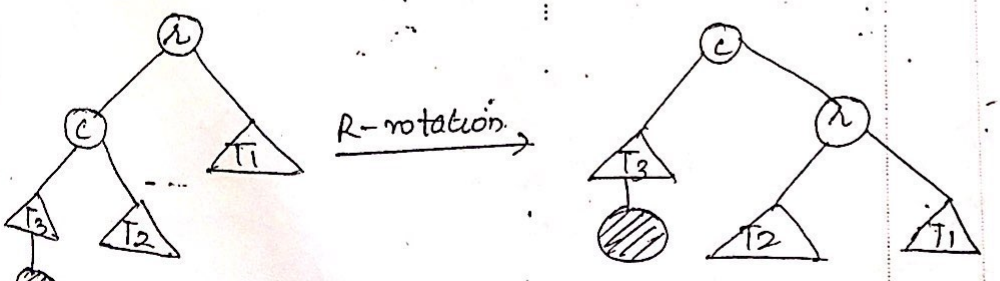
RL- rotation

• Single R-rotation

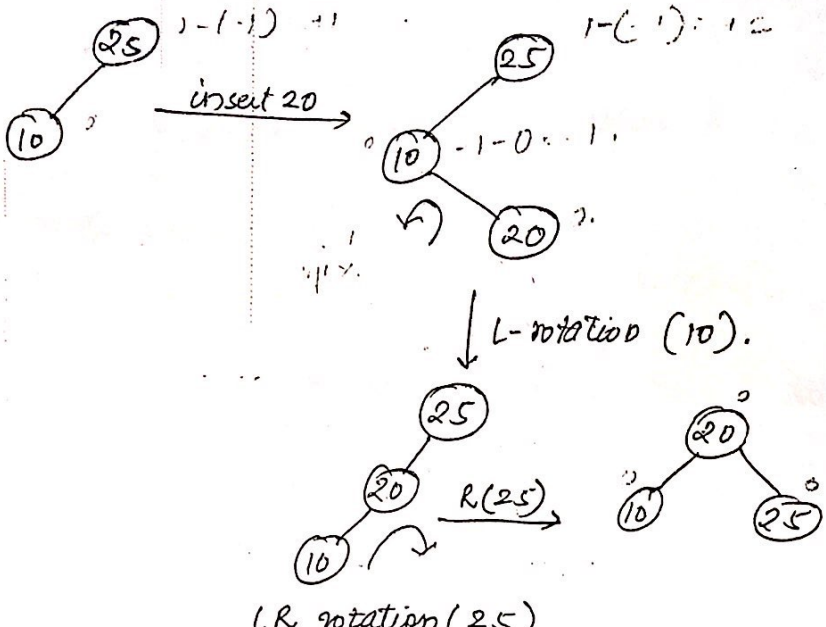




mirror image of L-rotation is R-rotation.

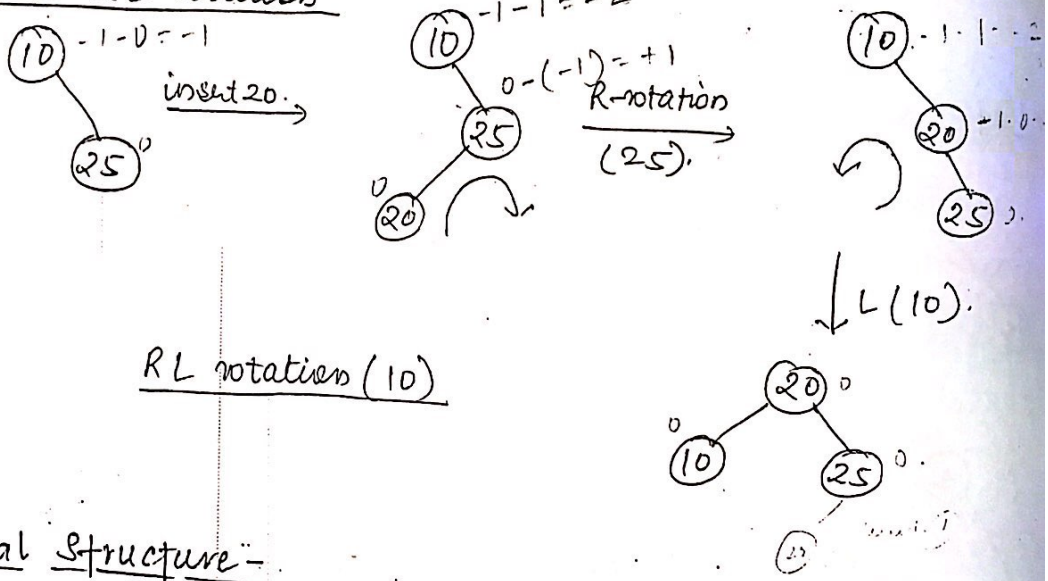


Double LR rotation -



LR rotation (25)

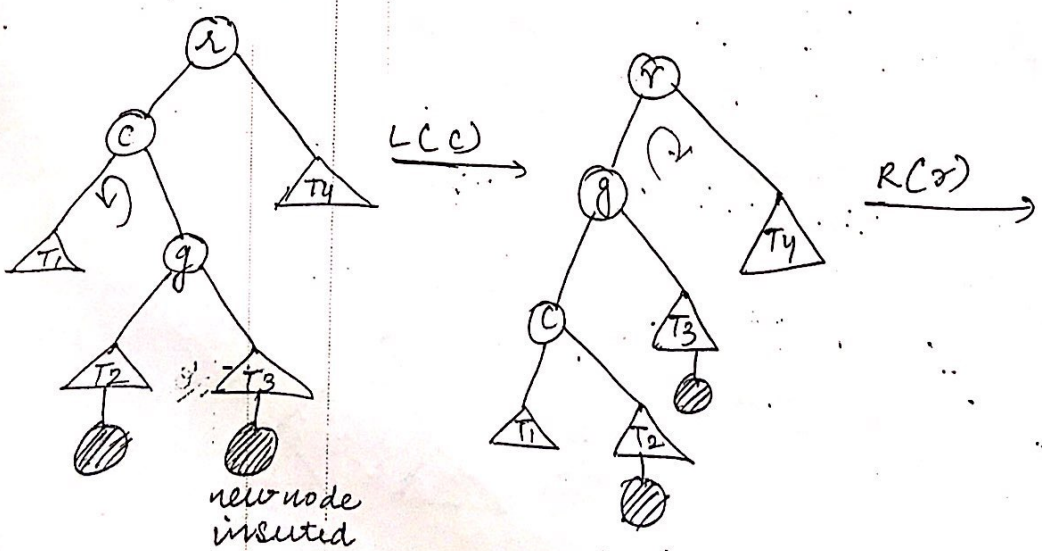
Double RL rotation -

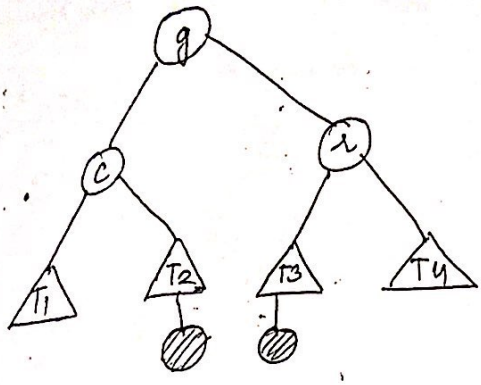


RL rotation (10)

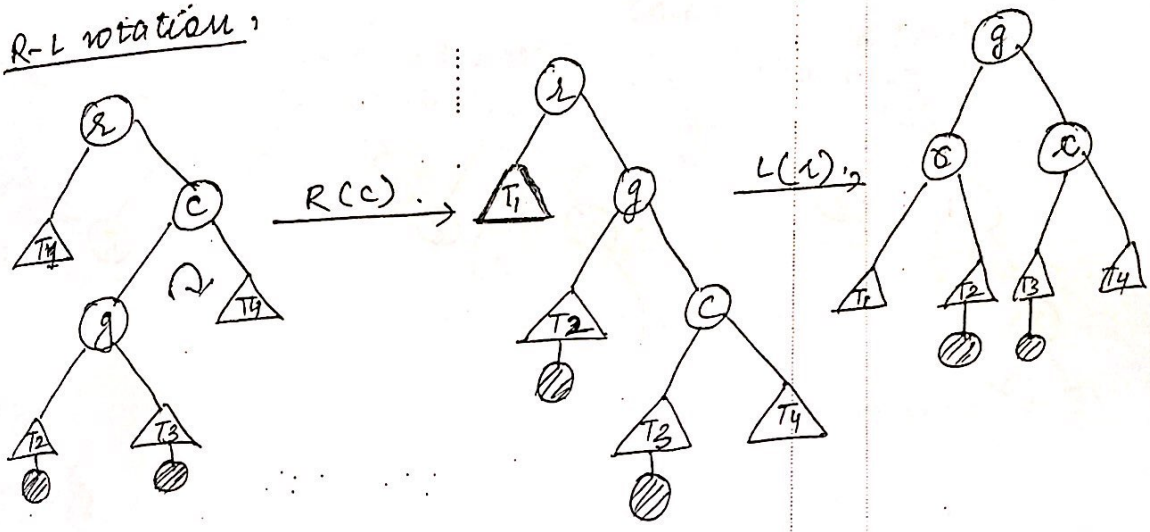
General Structure -

L-R rotation:





R-L rotation:



// Non-recursive algorithm to calculate sum of first [illegible] natural numbers.

```
sum ← 0
for i ← 1 to n do
    sum = sum + i * i * i
return sum;
// Input : N natural numbers
// Output : Sum of 1st cube of 'n' natural numbers.
```

// Recursive algorithm Sum.

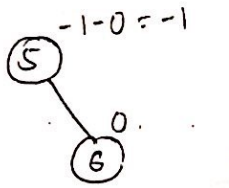
```
Sum ← 0.
// if n
if ( n == 0
    return sum
else    sum ← i * i * i.
return ( n*n*n + sum(n-1)).
```

- **AVL Tree:**

1) Construct the AVL tree for the list 5, 6, 8, 3, 2, 4, 7.

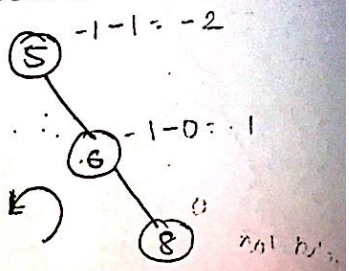
Insert 5.

Insert 6.



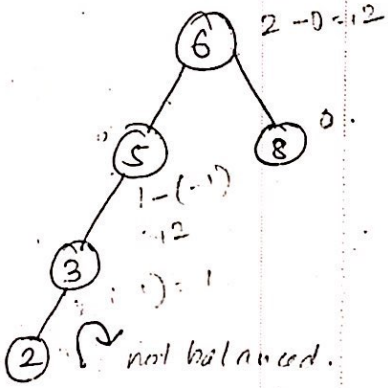
balance -1/+1.

Insert 8.



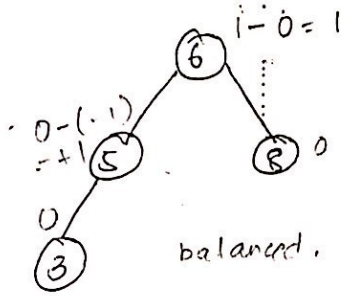
not bal...

Insert 2.

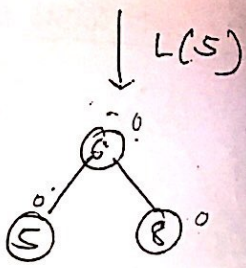


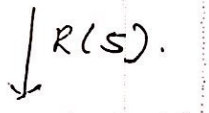
not balanced.

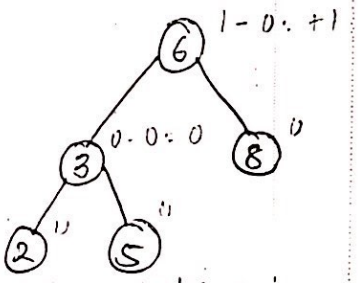
Insert 3.



balanced.

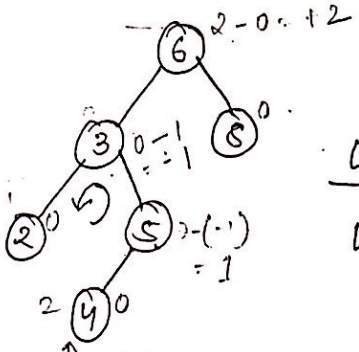




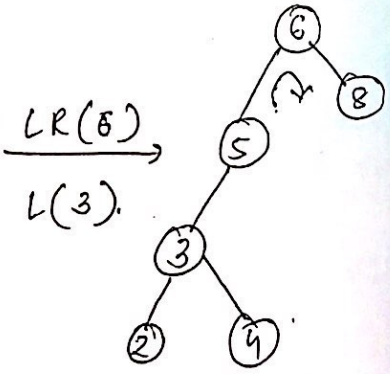


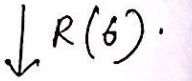
balanced.

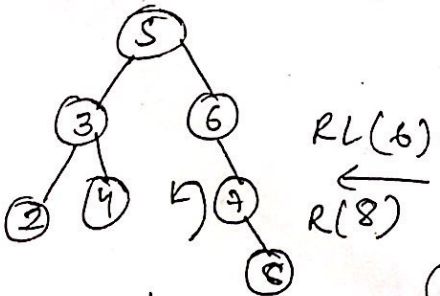
Insert 4.



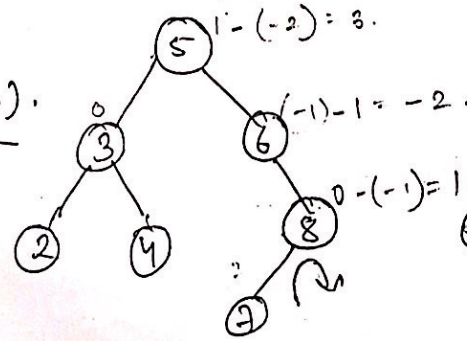
↑ not balanced, new node.

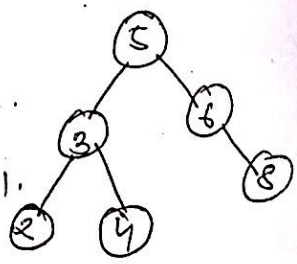


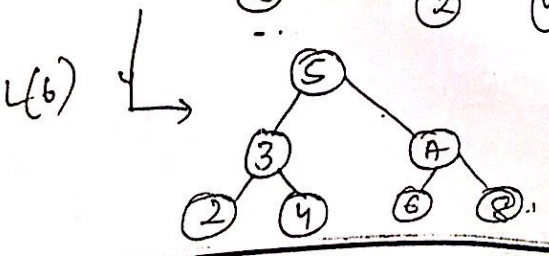




Insert 7.





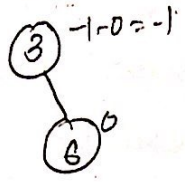


required AVL tree.

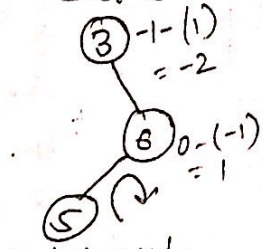
Construct the AVL tree for the list 3, 6, 5, 1, 2, 4.

Insert 3.

Insert 6.

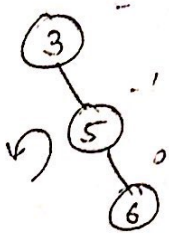


Insert 5.

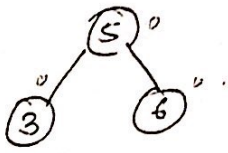


unbalanced.

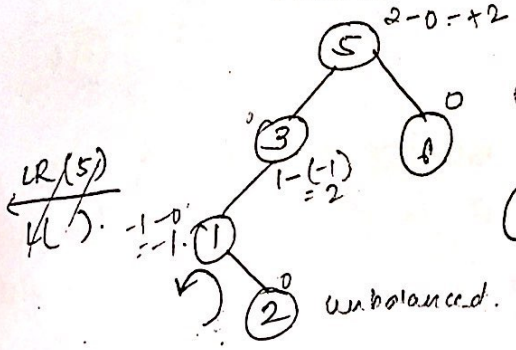
RL(3).
R(6).



L(3).



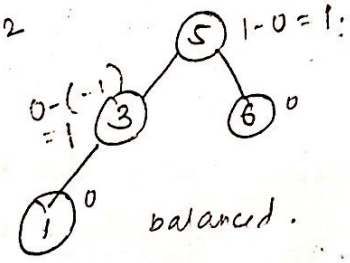
Insert 2.



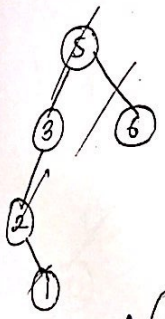
LR(5)
RL(5).

unbalanced.

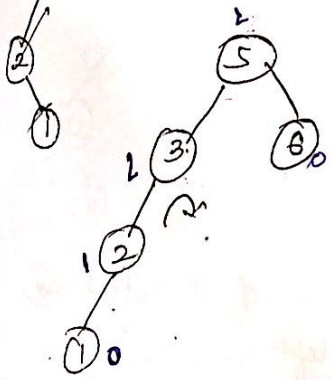
Insert 1.



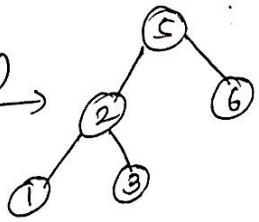
balanced.



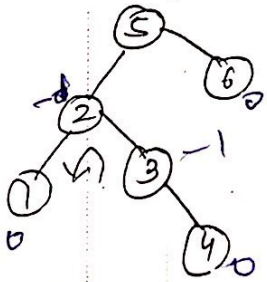
L&R(3)
L(1)+R(3).



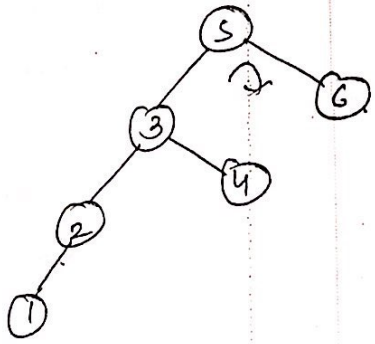
R(3)



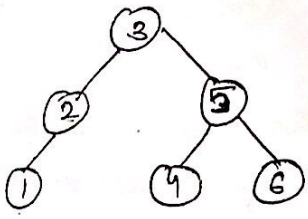
insert 4.



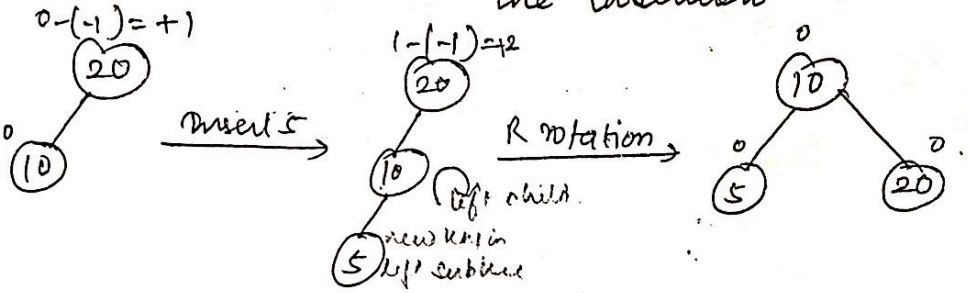
LR(5)
L(2).



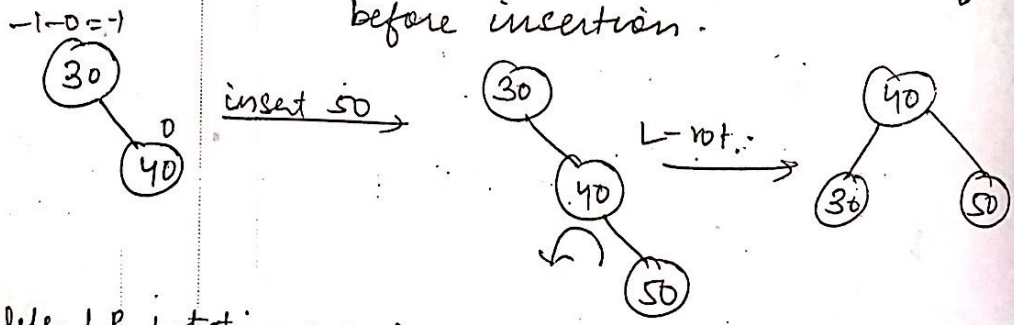
R(5).



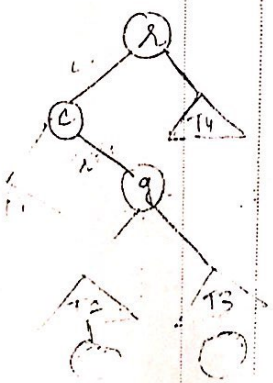
- **Single Right rotation:** It is performed after a new key is inserted into the left subtree of the left child of a tree whose root had balance of +1 before the insertion.



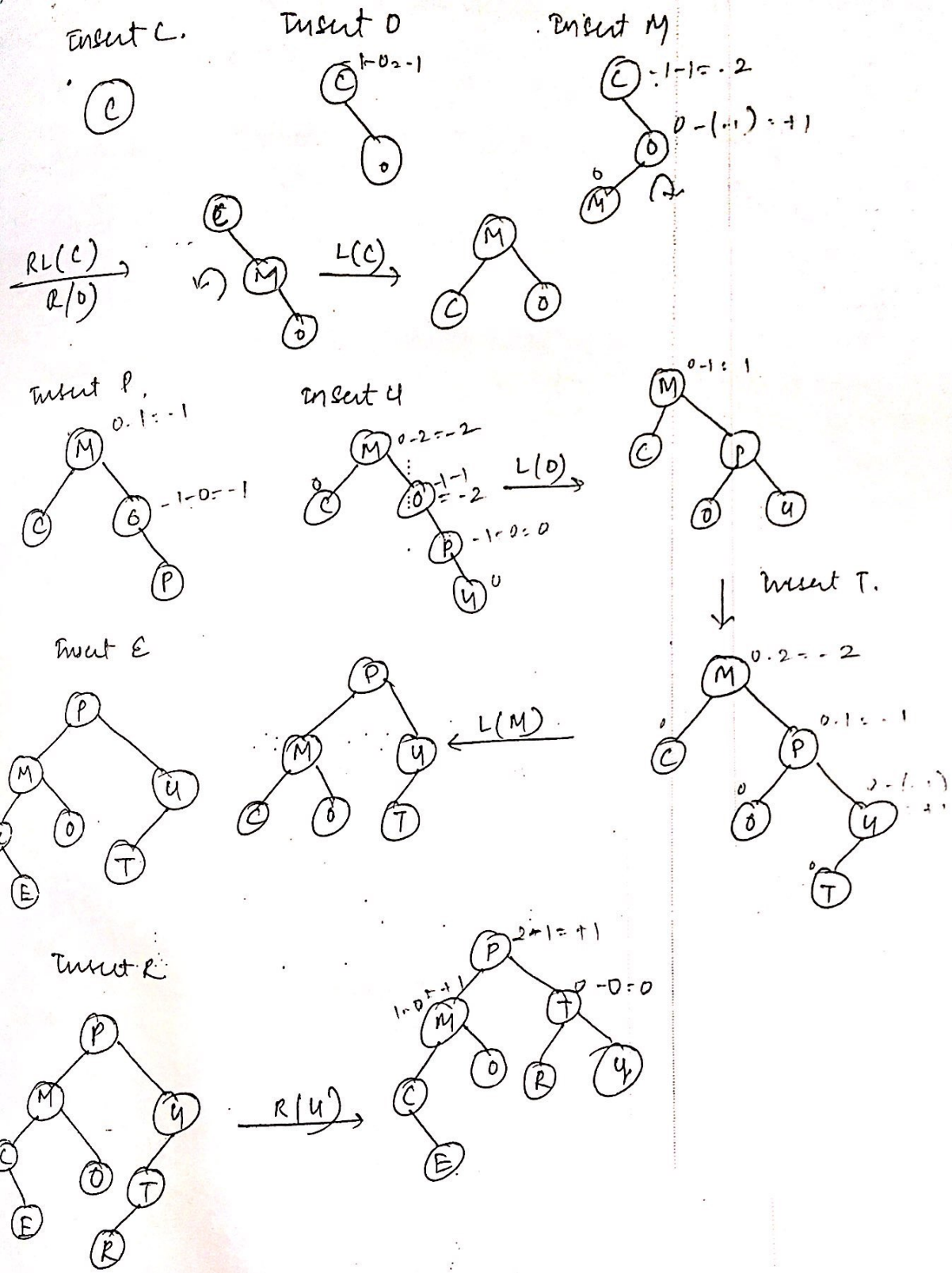
- **Single L rotation:** It is the mirror image of single R rotation it is performed after a new key is inserted at the right subtree of root right child of a tree whose root had balance of -1 before insertion.



- **Double LR rotation:** It is a combination of 2 rotations, L rotation of the left subtree of root r followed by the R rotation of the new tree rooted at r. It is performed after a new keys is inserted into the right subtree of the left child of a tree whose root had balance of +1 before insertion operation.



3. C, O, M, P, U, T, E, R.

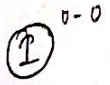


05/09/19

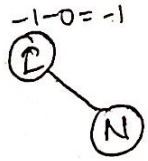
Construct the AVL tree for following data.

I, N, F, O, R, M, A, T, L, O, N.

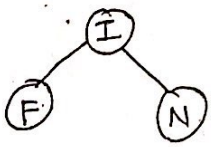
Insert I



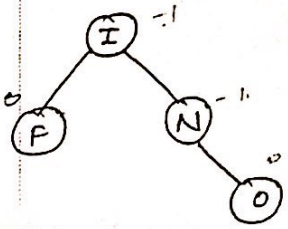
Insert N.



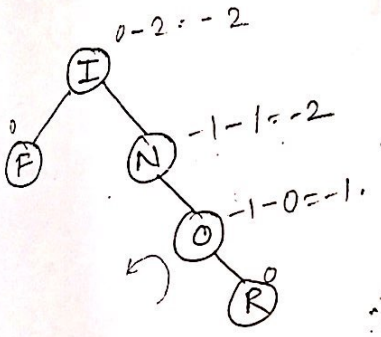
Insert F



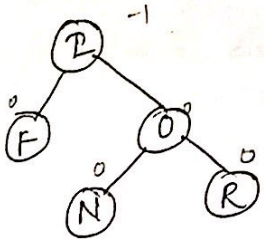
Insert O.



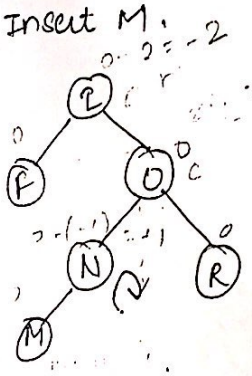
Insert R.



L(N)

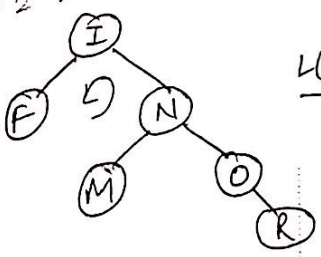


Insert M.

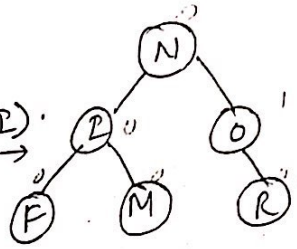


RL(I)

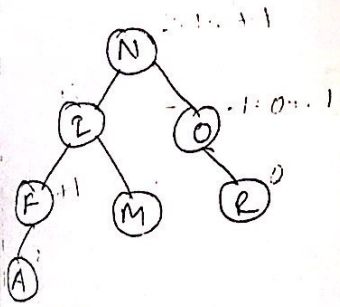
R(O)
+L(I)



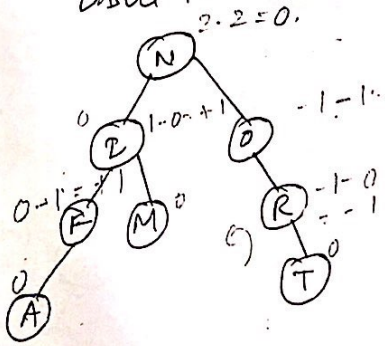
L(I)



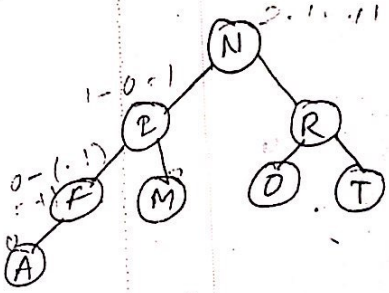
Insert A.



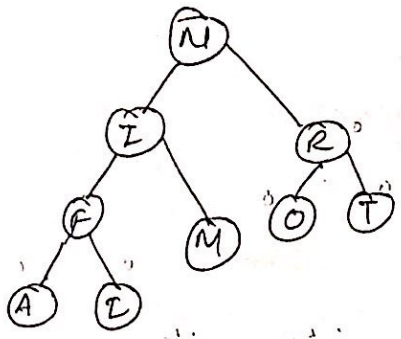
Insert T.



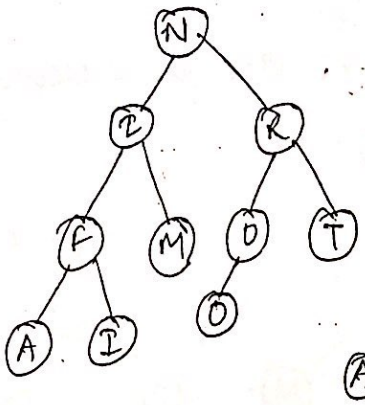
L(O)



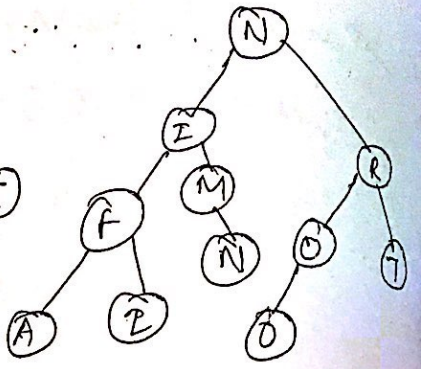
Insert 2.



Insert O



Insert N.

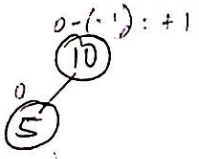


Construct AVL tree.

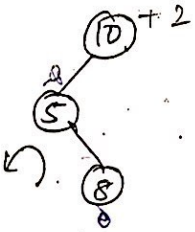
10, 5, 8, 15, 7, 25, 35, 50, 40, 20, 18

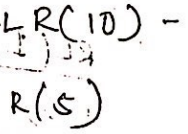
Insert 10.

Insert 5

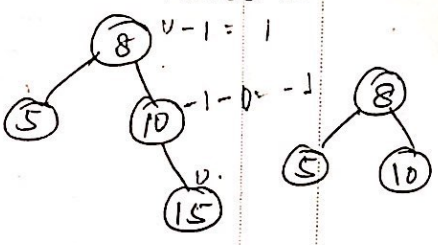


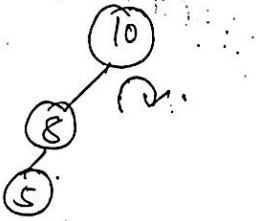
Insert 8



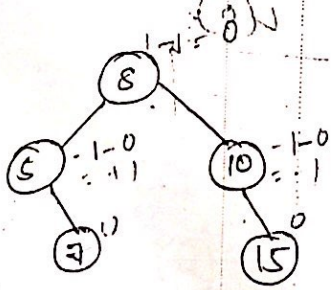


Insert 15.

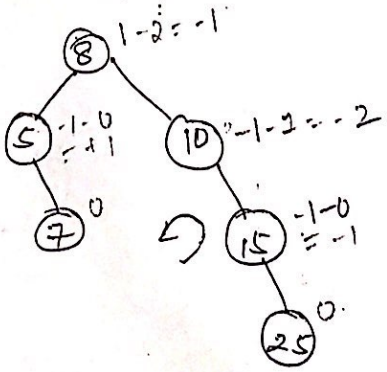


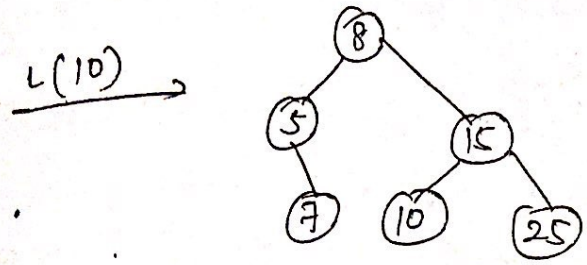


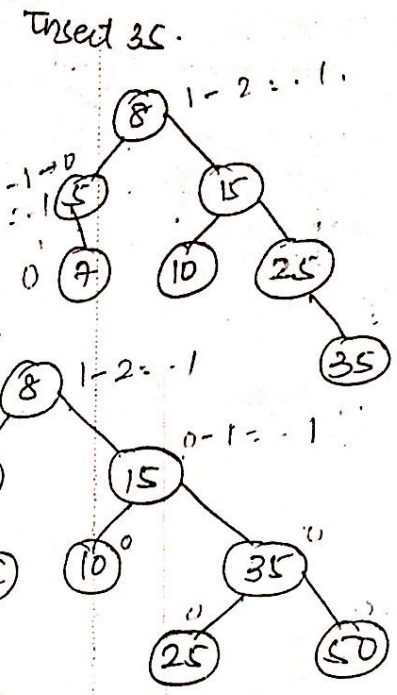
Insert 7.

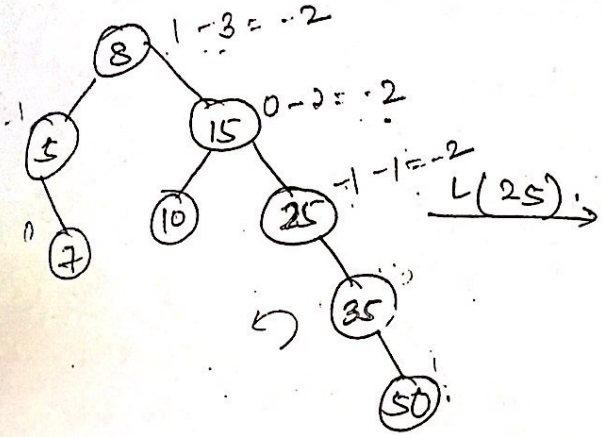


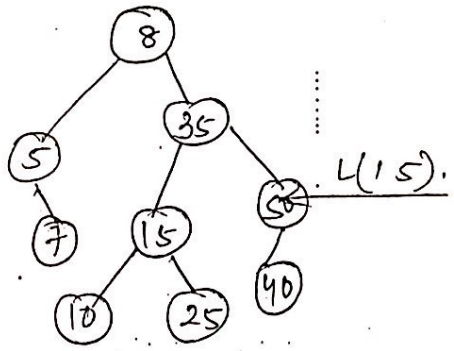
Insert 25.

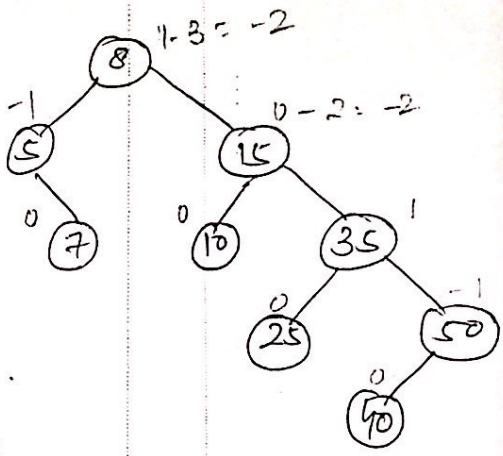


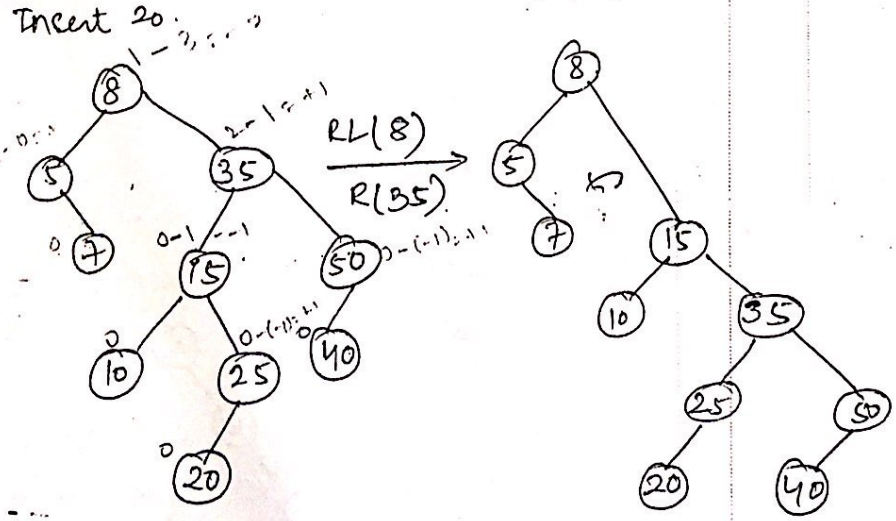


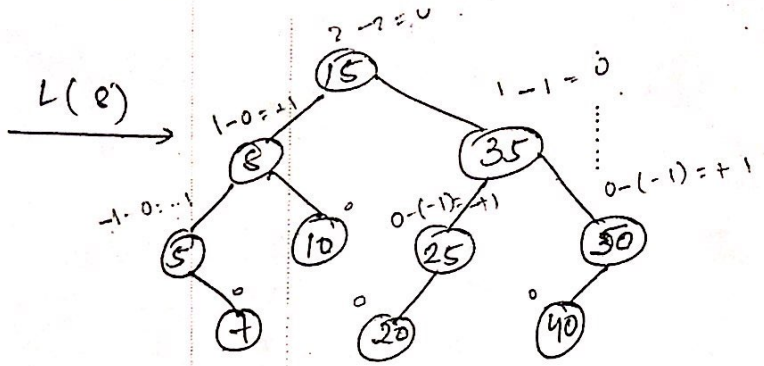




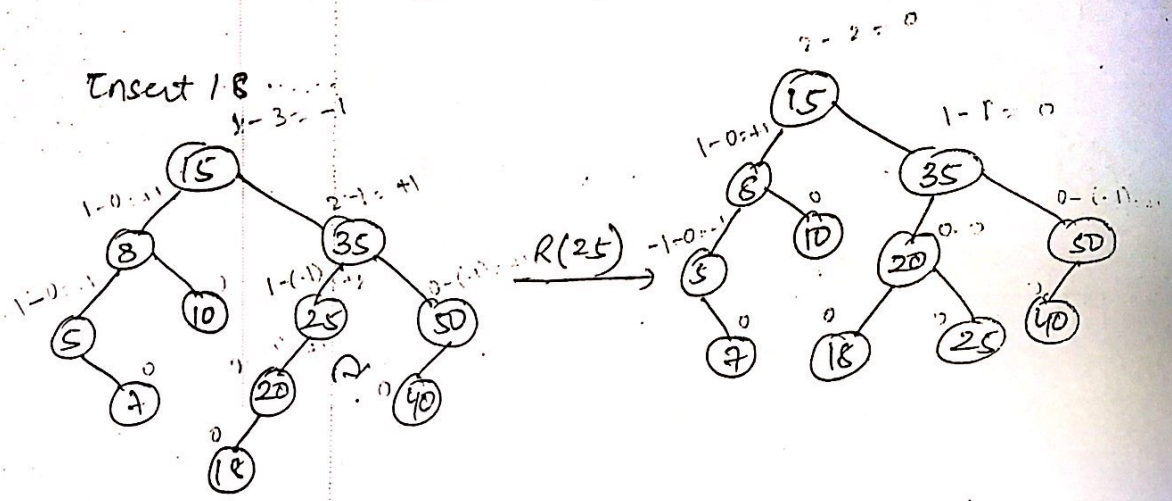








Insert 18 ....



[... AVL ... not vice versa]

[No of comparisons are logarithmic in nature].

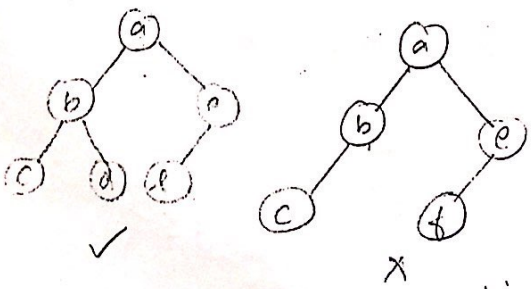
in complete binary tree:

$h = \log_2 n$

or $h = \log_2 n + 1$ [Balanced tree].

In unbalanced → comparison is n.

Almost complete BT:



d is missing

6/03/19

Heaps and heapsort -

A heap can be defined as a binary tree and is assigned to its nodes (one keyper node) provided that the following cond° are meet.

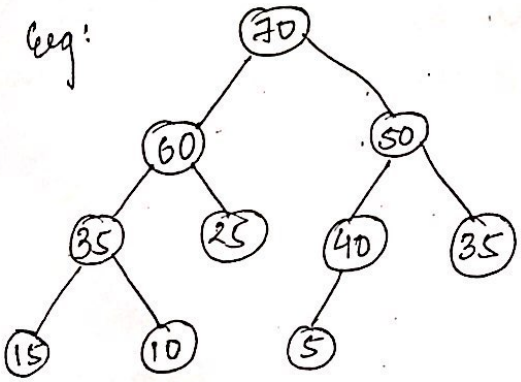
i) The tree shape requirement.

The Binary Tree is essentially complete i.e. all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

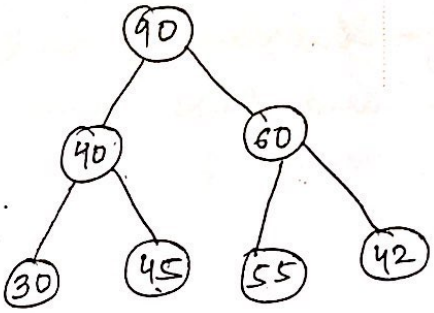
ii) The parental dominance requirement:

The key at each node is greater than or equal to the key at its' children.

Eg:



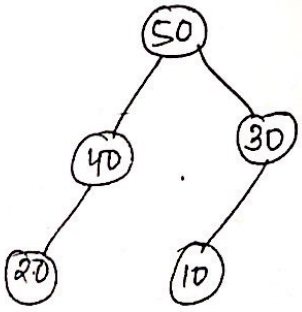
Heap.



Not a heap.

[45 is there].

bcoz parent dominance requirement not there.



Not a heap because tree shap req.

Property of Heap -

i) There exists exactly one binary tree with 'n' nodes with height being $[\log_2 n]$.

ii) Root of a heap contains largest element.

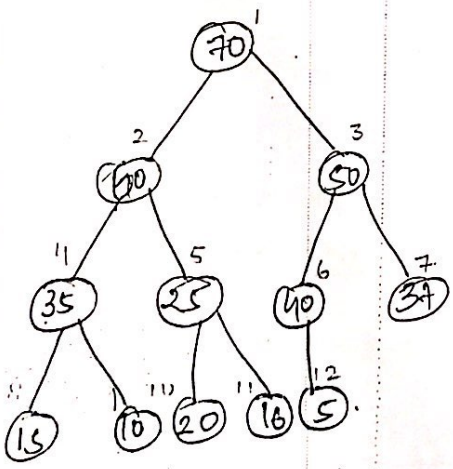
iii) A node of a heap considered with all its decendent is also a heap.

iv) A heap can be implemented as an array by recording its elements in the top-down left to right fashion.

a- Parental node keys will be in the first $n/2$ positions of the array, while the leaf keys will occupy last $n/2$ positions.

b- Children of a key are at position to $2i$ and $2i+1$ where 'i' is the parental position, parent of a child 'i' is a position $[i/2]$.

- Represent of given heap in an array.



1	2	3	4	5	6	7	8	9	10	11	12
70	60	50	35	25	40	37	15	10	20	16	5

$i = 9$.

then $[i/2] = 4$.

$\therefore$ at 4 i.e 35 is the parent of 10 (i.e the element in i).

$\therefore i[9] = 10$

& $i[4] = 35$.

Heap is an array $H[1 \ldots n]$ in which every elements in position i, in first half of the array is greater than or equal to the elements in position $2i$ and $2i+1$.

$$H[i] \geq \max\{H[2i], H[2i+1]\} \text{ for all } i = 1 \ldots \lfloor n/2 \rfloor$$

If array elements are indexed from zero then parent at position i and its left child is at position $2i+1$ and right child at $2i+2$.

Construction of Heap -

i) Bottom up approach.

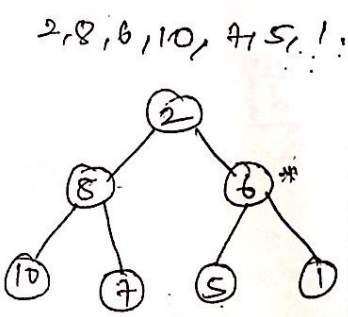
ii) Top down approach.

(Heapify algo starts from bottom up approach.)

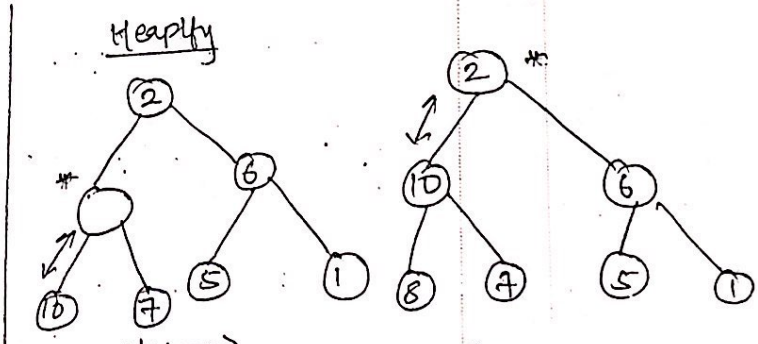
- Bottom up approach -

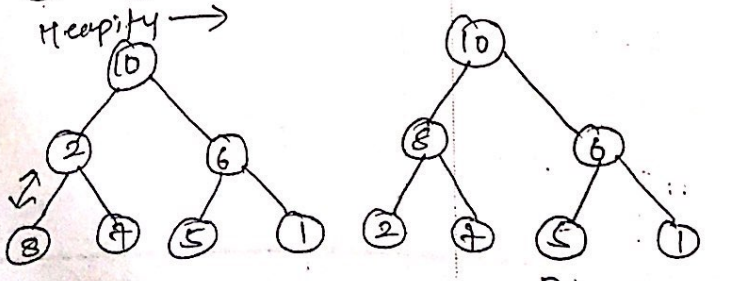
i) Initialize complete binary tree with 'n' nodes by placing keys in the order given.

ii) Heapify - Start with last parental node. check whether parental dominance holds for the key at this node. If it doesn't then exchange the node's key with larger key of its children and again check for dominance. This will continue for every level of tree.



const. complete BT.





Final heap

8|03|19

- **Algorithm BottomupHeap(H[1....n])**

// Constructs a heap from the elements of a given array.

// Input: An array H[1.... n] of orderable elements.

// Output: A heap H[1.....n]

```
for i ← [n/2] down to 1 do
    k ← i ; v ← H[i]
    heap ← false
    while not heap and 2*k ≤ n
        j ← 2*k
        if j < n  (/* If both the children are present.)
            if H[j] < H[j+1]
                j ← j+1
        if v ≥ H[j]
            heap ← true
        else
            H[k] ⟷ H[j]
            k ← j
    H[k] ← v.
```

- 50, 12, 16, 5, 75, 82, 18.

n = 7.

$\left[\frac{n}{2}\right] = 3$.

1	2	3	4	5	6	7
50	12	16	5	75	82	18

i ←

k ← 3, v ← 16

heap ← false

j ← 6.

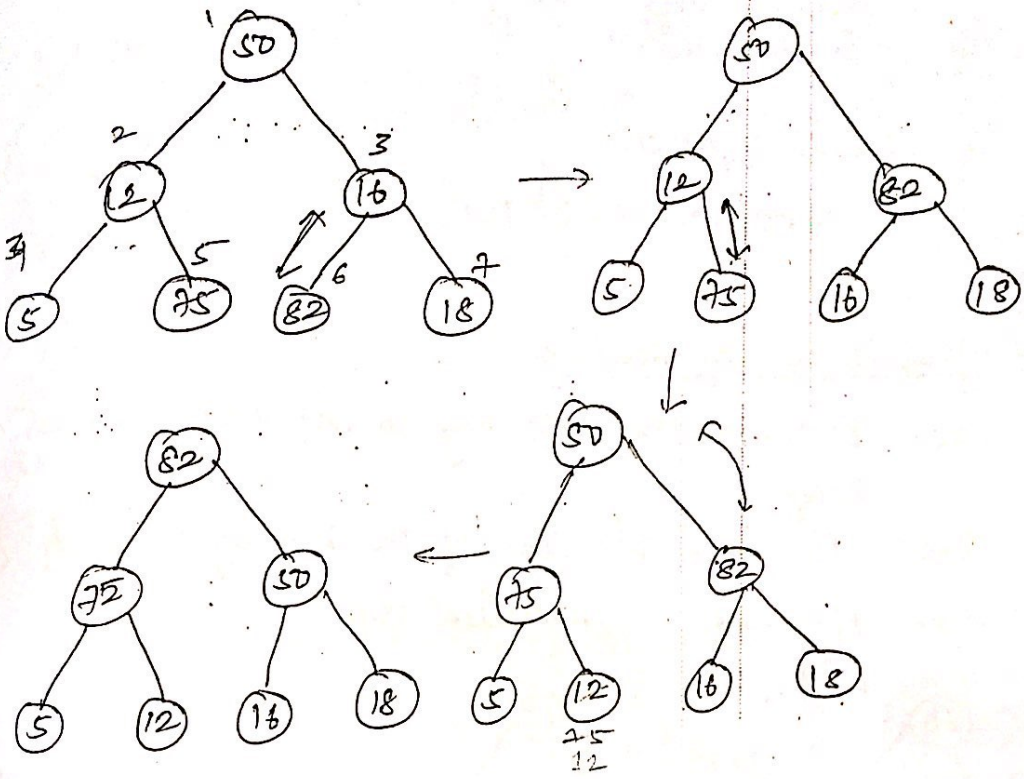
H[6] < H[7]

12 < 18

v > H[6].

16 ≯ 82

1	2	3	4	5	6	7
82	75	50	5	12	16	18



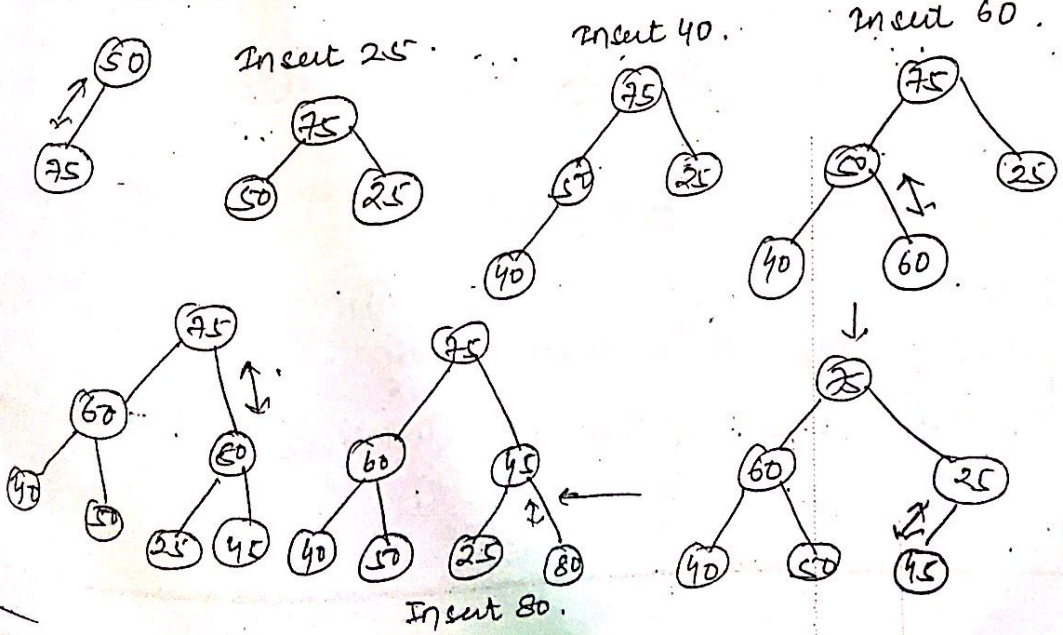
Comparison:

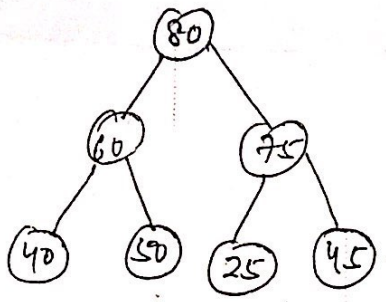
$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{node } i} 2(h-i)\cdot 2^i$$

$$= 2(n - \log_2(n+1))$$

fewer than 2^n.

• Top down approach: 50, 75, 25, 40, 60, 80, 70, 80.





Top down approach.

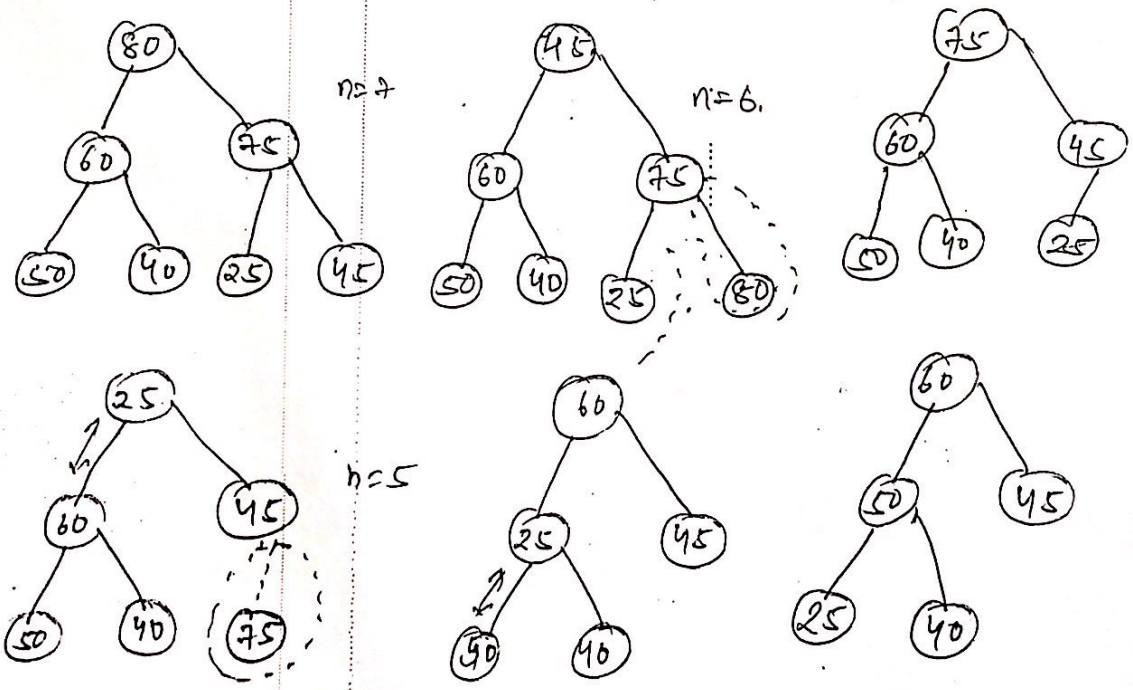
$C_{worst}(n) = O(\log_2 n)$.

- Maximum key deletion:

Step 1: Exchange the roots key to the last key 'k' of the heap.

Step 2: Decrease the size of the key by 1.

Step 3: Heapify smaller tree.



[Discovered by JWJ Williams].

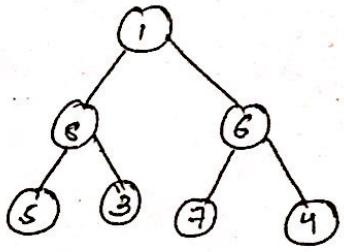
Heap construction -

→ Construct a heap for a given array.

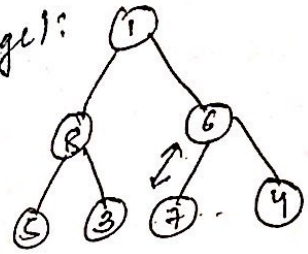
→ Apply root deletion operation n-1 times to the remaining key.

15/03/19

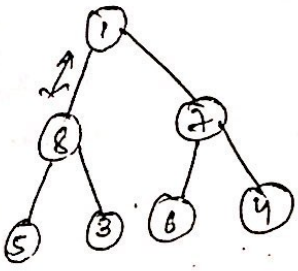
Eg: 1, 8, 6, 5, 3, 7, 4



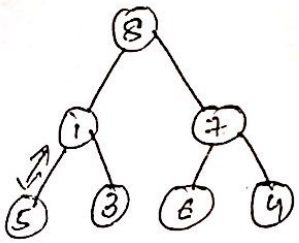
Stage 1:

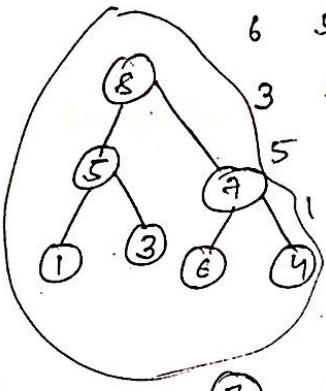


1	8	6	5	3	7	4
1	8	7	5	3	6	4
8	5	7	1	3	6	4. Heapify

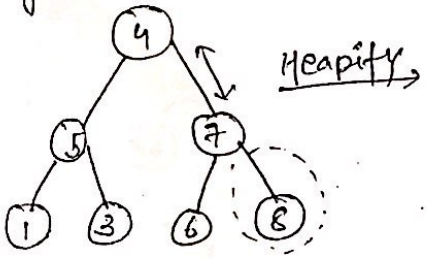


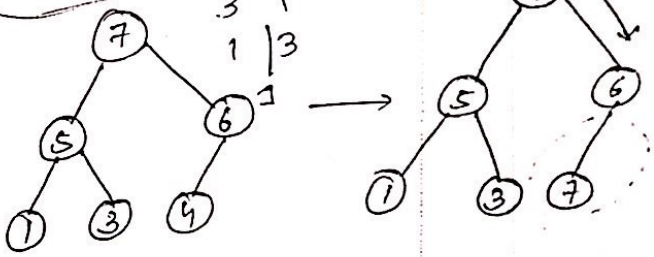
4	5	7	1	3	6	8
7	5	6	1	3	4.	
4	5	6	1	3	7	
6	5	4	1	3.		
3	5	4	1	6.		
5	3	4	1			
1	3	4	5			
4	3	1				
1	3	4.				
3	1					
1	3					

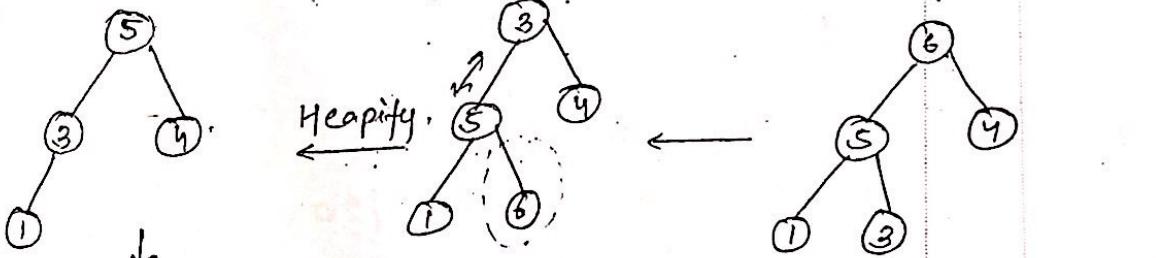


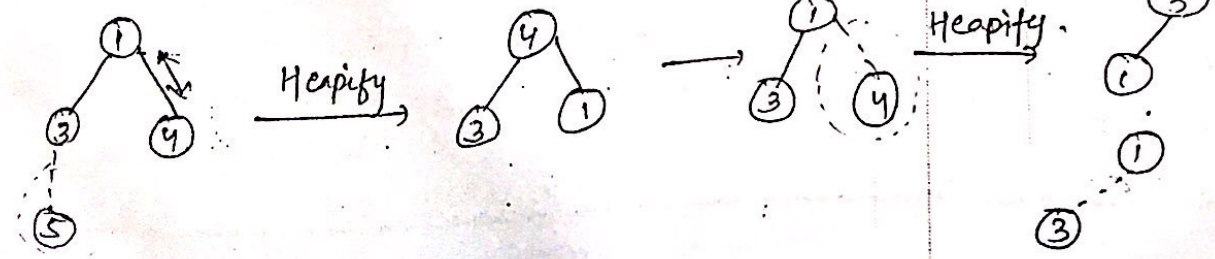


Stage 2:





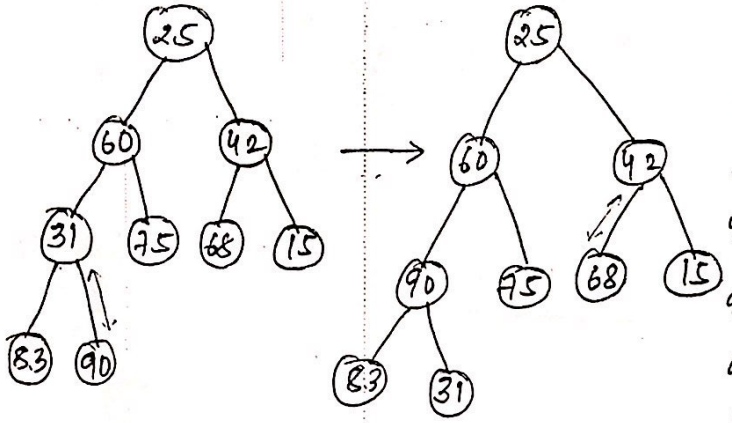




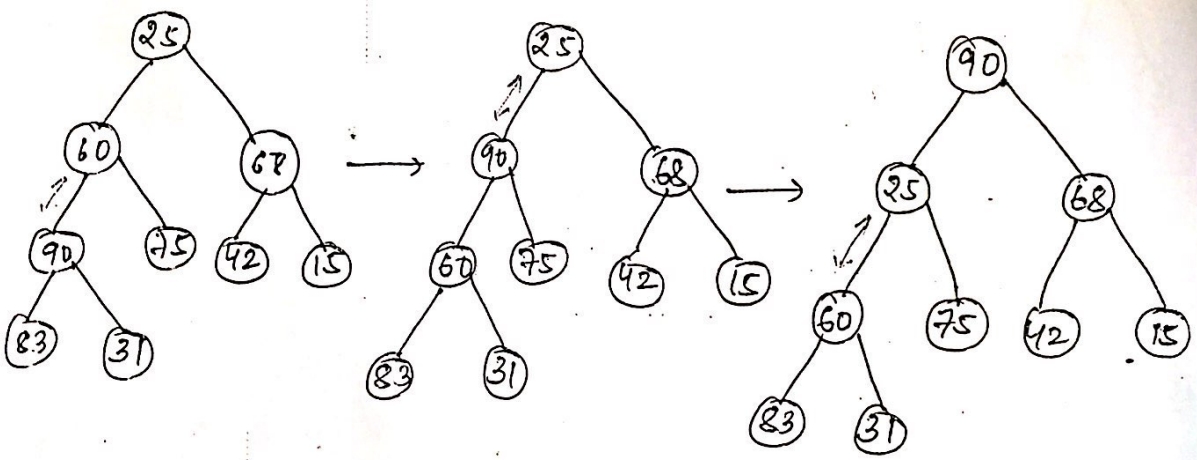
1	2	3	4	5	6	7
1	3	4	5	6	7	8

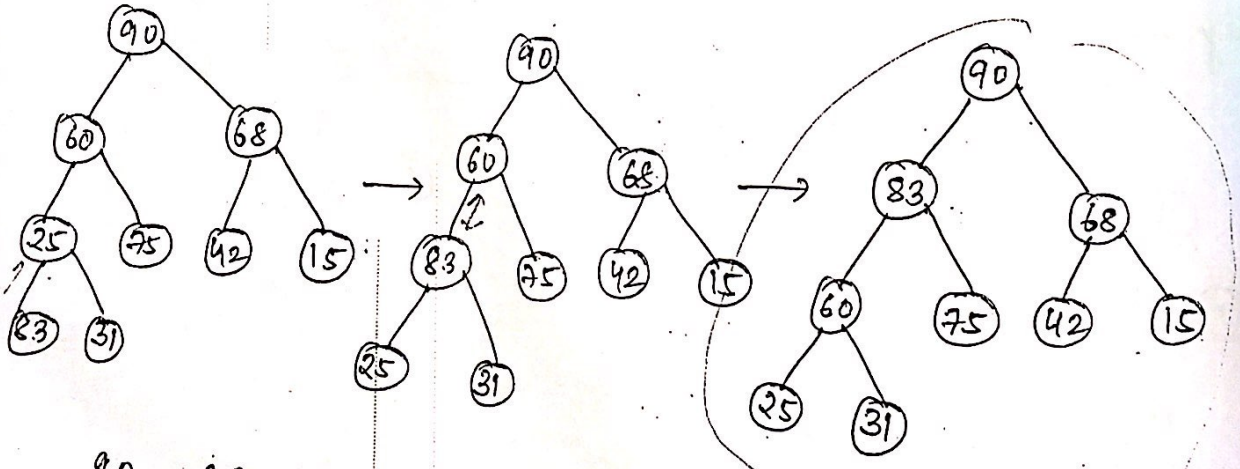
n = 7.

Eg: 25, 60, 42, 31, 75, 68, 15, 83, 90.

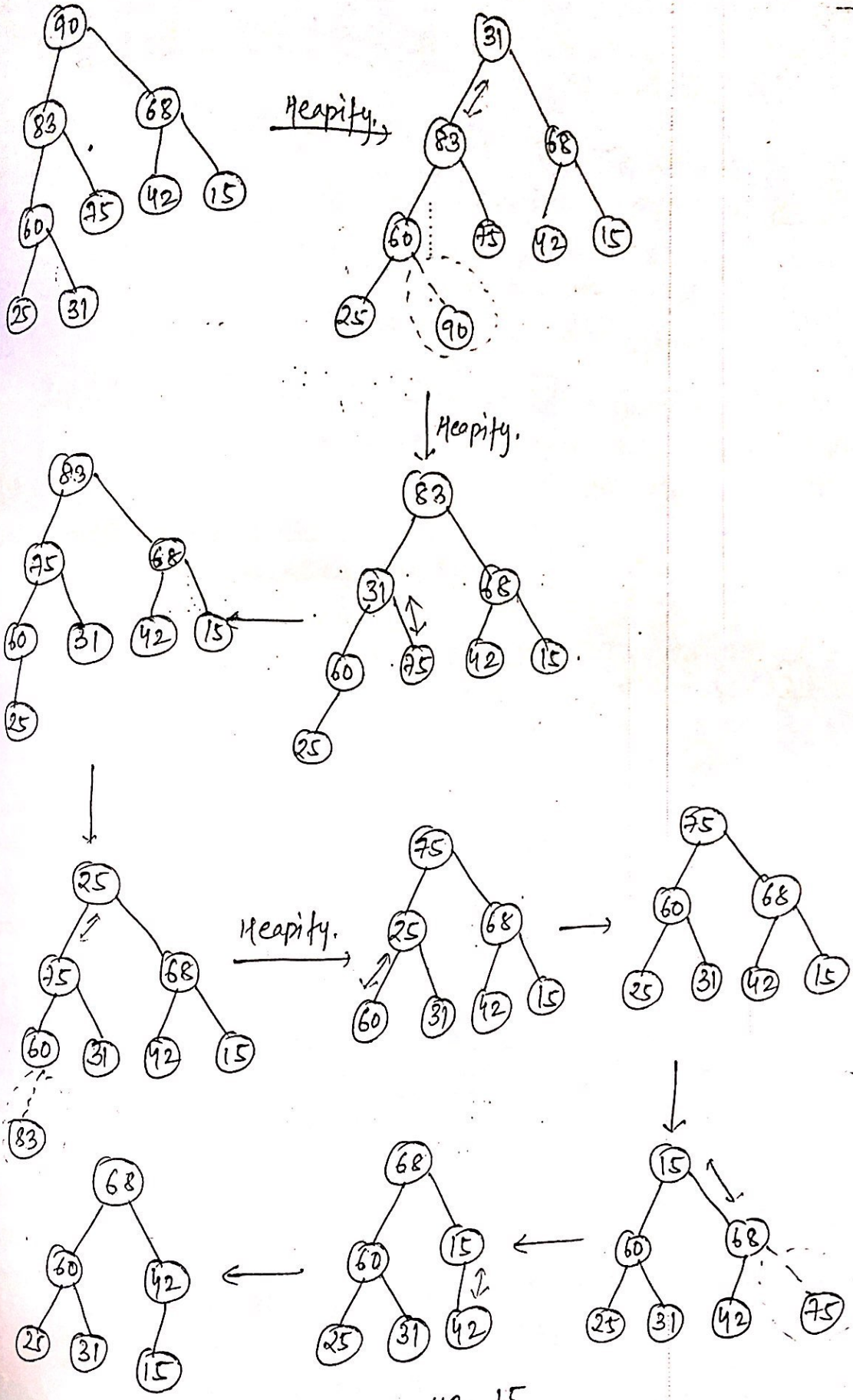


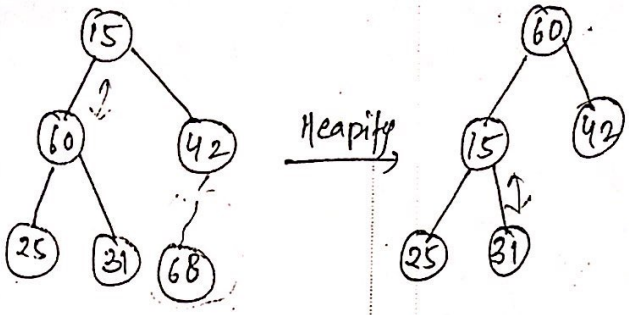
25 60 42 31 75 68 15 83 90
25 60 42 90 75 68 15 83 31
25 60 68 90 75 42 15 83 31
25 90 68 60 75 42 15 83 31
90 25 68 60 75 42 15 83 31
90 60 68 25 75 42 15 83 31
90 60 68 83 75 42 15 25 31
90 83 68 60 75 42 15 25 31

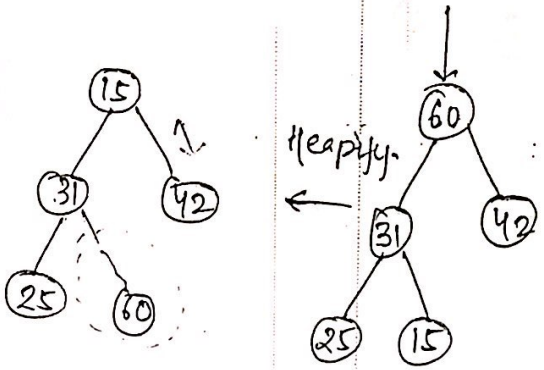


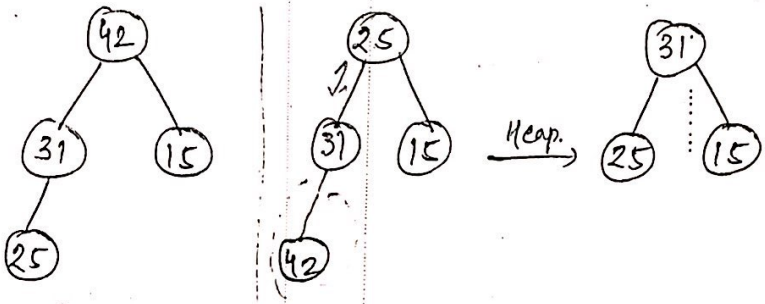


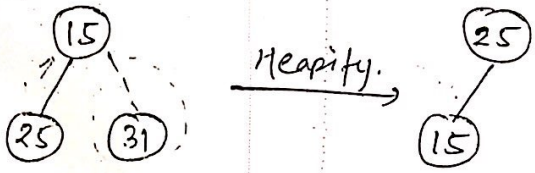
90 83 68 60 75 42 15 25 31
31 83 68 60 75 42 15 25 | 90
83 31 68 60 75 42 15 25
83 75 68 60 31 42 15 25
25 75 68 60 31 42 15 | 83

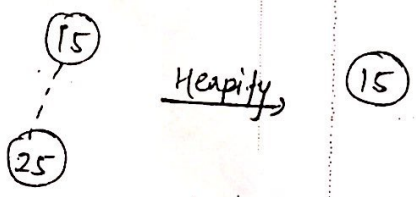












1	2	3	4	5	6	7	8	9
15	25	31	42	60	68	75	83	90

$n = 9$.

$C_{worst}(n) = \theta(n \log_2 n)$.

68	60	42	25	31	15
15	60	42	25	31	68
60	15	42	25	31	
60	31	42	25	15	
15	31	42	25	60	
42	31	15	25		

25	31	15	42
31	25	15	
15	25	31	
25	15		
15	25		
15			

Time and Space trade Off:

Input enhancement: It refers to the preprocessing the problem input in whole or part and storing the additional inform' obtained to accelerate solving the problem afterward.

Prestructuring: It uses extra space to facilitate faster and/or, more flexible access to the data.

- **Input Enhancement-**

→ Comparison counting sort.
→ Distribution counting sort.

* Comparison Counting sort:

n=8 Array A[0.....7].

	25	46	72	16	5	20	36	54.
initial count	0	0	0	0	0	0	0.	0
i=1 count[ ]	3	1	1	0	0	0	1	1
i=2 count[ ]		5	2	0	0	0	1	2
i=3 count[ ]			7	0	0	0	1	2
i=4 count[ ]				1	0	1	2	3
i=5 count [ ]					0	2	3	4
i=6						2	4	5
i=7							4	6
	3	5	7	1	0	2	4	6.
S[0..7].	5	16	20	25	36	46	54	72

After pass i= 0.

18/03/19

Eg: 85, 92, 18, 55, 68, 43, 36, 10.

A[0...7]

initially count=0

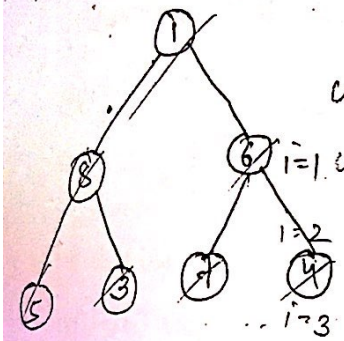
	85	92	18	55	68	43	36	10
	0	0	0	0	0	0	0	0
pass i=0 count[ ]	6	1	0	0	0	0	0	0
i=1 count[ ]		7	0	0	0	0	0	0
i=2 count[ ]			1	1	1	1	1	0
i=3 count[ ]				4	2	1	1	0
i=4 count[ ]					5	1	1	0
i=5 count[ ]						3	1	0
i=6 count[ ]							2	0
	6	7	1	4	5	3	2	0
S[0...7] =	10	18	36	43	55	68	85	92

- Algorithm ComparisonCountingSort(A[0...n-1]).

// Sorts an array A[0...n-1] by comparison counting
// Input: An array A[0...n-1] of orderable elements.
// Output: An array S[0...n-1] of A's element sorted in non decreasing order.

```
for i ← 0 to n-1 do
        count[i] ← 0
for i ← 0 to n-2 do
        for j ← i+1 to n-1 do
                if A[i] < A[j]
                        count[j] ← count[j]+1
                else
                        count[i] ← count[i]+1.
for i ← 0 to n-1 do
        S[count[i]] ← A[i]
return S
```

1, 8, 6, 5, 3, 7, 4. Tracing:



n=6 A. 35 72 6 29 38 45

count [ ] 0 0 0 0 0 0

i=1 count [ ] 2 1 0 0 1 1

i=2 count [ ] 1 0 0 1 1

i=3 count [ ] 5 0 0 1 1

i=4 count [ ] 0 1 2 2

i=5 count [ ] 1 3 3

S[0 ... 5] 3 4

6 29 35 38 45 72

S[count[0]] = S[2] ← A[0]

S[5] ← A[1]

S[0] ← A[2] S[4] ← A[5]

S[1] ← A[3]

S[3] ← A[4]

Analysis

Complexity $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

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

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

$= 1+2+3+\dots+(n-2)+(n-1)$

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

$\approx \frac{n^2}{2} \in \Theta(n^2)$.

• Distribution Counting Sorting:

13 12 11 12 11 12 13 11

	11	12	13
Frequency	3	3	2
Distribution value	3	6	8

distribu value	11	12	13
A[7]=11	(3)	6	8
A[6]=13	2	(6)	(8)
	2	(5)	7
A[5]=12	(2)	5	7
A[4]=11	1	(5)	7
A[3]=12	(1)	4	7
A[2]=11	0	(4)	7
A[1]=12	0	3	(7)
A[0]=13	0	3	6

	0	1	2	3	4	5	6	7
S			11					13
		11				12		
					12			
	11							
				12				
								13

Assuming that a set of possible values {22, 23, 24, 25} Sort the following list in ascending order by using distribution counting algorithm.

	0	1	2	3	4	5	6	7	8	9
	24	23	24	22	25	23	25	25	22	23
	22	23	24	25						
Frq.	2	3	2	3						
Distribution value	2	5	7	10						

	22	23	24	25
A[9]=23	2	(5)	7	10
A[8]=22	(2)	4	7	10
A[7]=25	1	4	7	(10)
A[6]=25	1	4	7	9
A[5]=23	1	4	7	8
A[4]=25	1	3	7	8
A[3]=22	1	3	7	7
A[2]=24	0	3	7	7
A[1]=23	0	3	6	7
A[0]=24	0	2	6	7
	0	2	5	7

	0	1	2	3	4	5	6	7	8	9
					23					25
		22							25	
				23						
								24		
	22									
							24			
			23							
						24				
	22	22	23	23	23	24	24	24	25	25

Assuming that set of possible values in {a, b, c, d} sort the following list in alphabetical order by distribution counting algorithm.

b c d c b a a b

	a	b	c	d
frequency	2	3	2	1
Distrib' values.	2	5	7	8.

distri value.	a	b	c	d
A[7]=b	2	(5)	7	8
A[6]=a	(2)	4	7	8
A[5]=a	(1)	4	7	8
A[4]=b	0	(4)	7	8
A[3]=c	0	3	(7)	8
A[2]=d	0	3	6	(8)
A[1]=c	0	3	(6)	7
A[0]=b	0	(3)	5	7
	0	2	5	7

0	1	2	3	4	5	6	7
				b			
	a						
a							
			b				
						c	
							d
					c		
		b					
a	a	b	b	b	c	c	d

• Algorithm Distribution Counting (A[0...n-1])

// Sorts an array of integers from a limited range by distribution counting.

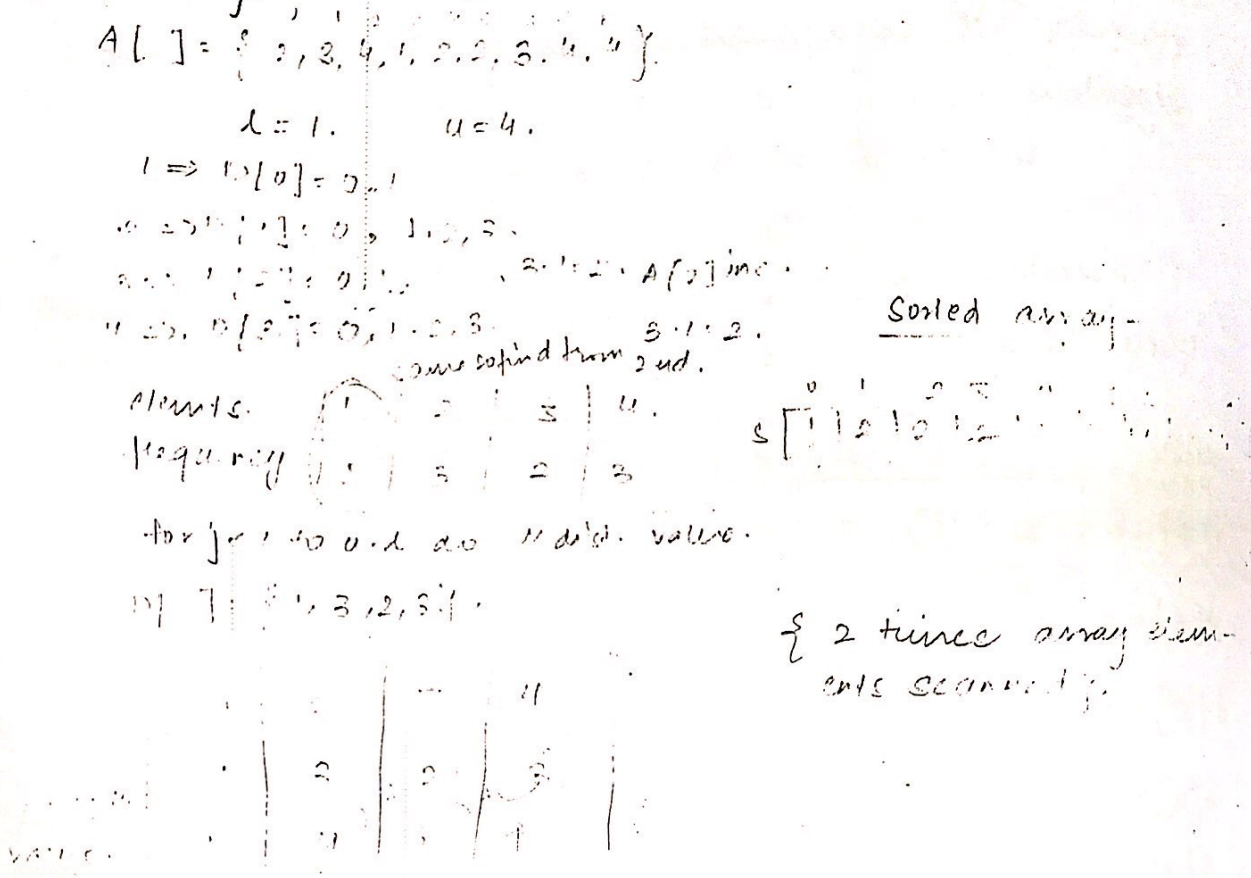
// Input: Array A[0...n-1] of integers btw l and u (l ≤ u)

// Output: Array S[0...n-1] of A's elements sorted in non-decreasing order.

```
for j ← 0 to u-l do D[j] ← 0 //Initialize frequency.
for i ← 0 to n-1 do D[A[i]-l] ← D[A[i]-l] + 1 //Compute f...
for j ← 1 to u-l do D[j] ← D[j-1] + D[j] //Reuse for di...
for i ← n-1 down to 0 do
    j ← A[i] - l
    S[D[j]-1] ← A[i]
    D[j] ← D[j]-1
return S.
```

22/3/19

Tracing:



- Input enhancement String matching

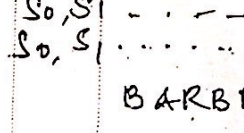
Horspool's String matching

Consider eg: of searching for the pattern 'BARBER' in text of length 'n' character.

Horspool's algorithm determines the size of the shift by looking at character 'c' of the text.

In General there are 4 possible cases:

Case i: If there are no c's in the pattern, then shift the pattern by its entire length.



Case ii: If there are occurences character 'c' in the pattern but not the last one, then shift should align with rightmost occurence

of c in pattern with the 'c' in the text.

$S_0, S_1 \ldots\ldots B \ldots\ldots S_{n-1}$ text

BARBER. pattern:

BARBER

case iii: If 'c' happens to be the last character in the pattern, there are no 'c's among other n-1 characters, the shift should be similar to case i., i.e shift by the entire pattern by length (m).

$S_0, S_1 \ldots\ldots MER \ldots S_{n-1}$

LEADER

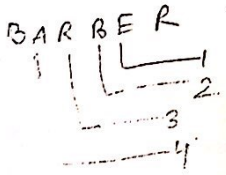
LEADER.

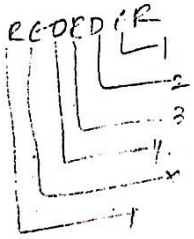
case iv: If 'c' is the Last character in the pattern and there are other c's among its 1st (n-1) characters, shift should be similar to that of case ii, i.e. the rightmost occurrence of c, first n-1 character in the pattern should be aligned with char c of the text.

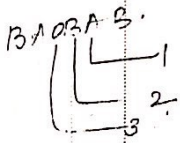
$S_0, S_1 \ldots MER \ldots\ldots S_{n-1}$

REORDER

REORDER.







25/03/19

- Algorithm ShiftTable(P[0...m-1]).

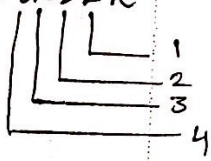
// Fills the shift table used by Horspool's algorithm.
// Input: Pattern P[0,...m-1] and alphabet of possible characters.
// Output: Table[0...size-1] indexed by the alphabet's characters and filled with shift sizes computed by formula:

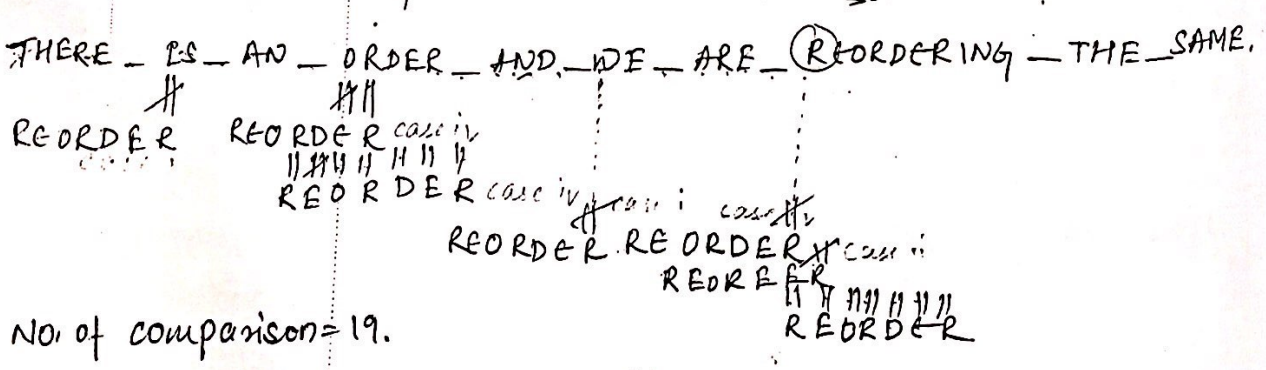
Initialize all the elements of the table with m-1
for j ← 0 to m-2 do
    Table[P[j]] ← m-1-j.
return Table.

Eg: COMPUTER_SCIENCE Pattern_SCIE

Eg: THERE_IS_AN_ORDER_AND_WE_ARE_REORDERING_THE_SAME.

Pattern- REORDER





No. of comparison = 19.

- Algorithm HorspoolMatching(P[0...m-1], T[0...n-1]).

// Implements Horspool's algorithm for string matching.
// Input: Pattern P[0...m-1] and Text[0...n-1]
// Output: The index of the left end of the first matching
// substring or -1 if there are no matches.

```
shift Table[P(0...m-1)]          //generates table for shift.
i ← m-1                          //position of the pattern right end.
while i ≤ n-1 do
    k ← 0                        // no. of matched characters
    while k ≤ m-1 and P[m-1-k] = T[i-k]
        k ← k+1.
    if k = m
        return i-m+1.
    else
        i ← i + Table[T[i]].
return -1.
```

Eg: SIDDAGA_INSTITUTE_OF_TECHNOLOGY.

Pattern: TECHNO

m=6, n=31.

i ← 5

k ← 0.

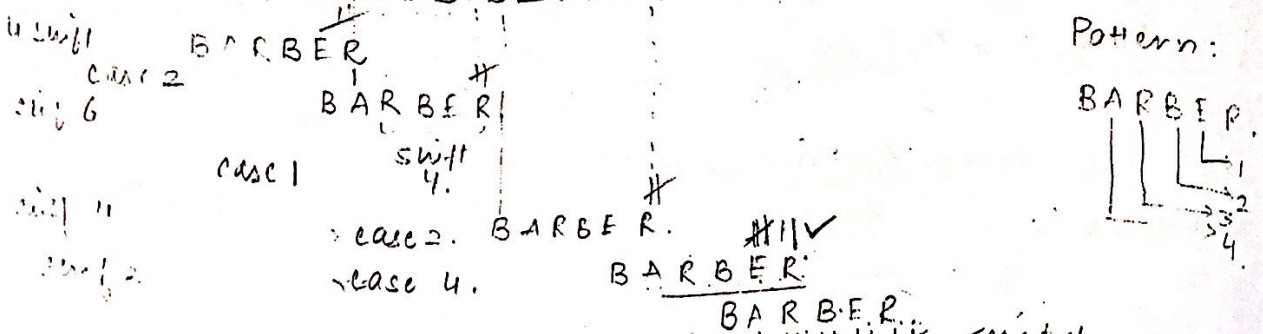
P[6-1-0] P[5] T[5]

Number of comparison = 4+6 = 10.

26/03/19

Pattern: BARBER. m = 6. n = 28.

1) Text: JIM_SAW_BOB_2N_A_BARBER_SHOP.

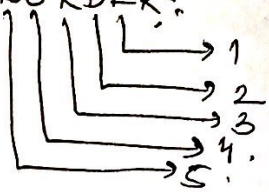


No. of comparison = 11.

2) ABCBRADBORDERCARER

3) ABCBORCBABERBBORDERABC.

Pattern: BORDER.



2) ABCBRADBORDERCARER.

BORDER.

case 1: BORDER

case 2: BORDER

No. of comparison = 8.

3) ABCBORCBABERBBORDERABC.

case 4: BORDER

case 1: BORDER

BORDER

case 2: BORDER

No. of comparison = 10.

• Bayer–Moore's Algorithm:-

1) Bad symbol shift : d_1

2) Good suffix shift : d_2

1). So $S_1 \ldots\ldots\ldots C \ldots\ldots\ldots S_{1-k+1} \ldots\ldots S_1 \ldots\ldots S_{n-1}$

$P_0 P_1$ $\quad$ P_{m-k-1} $\quad$ P_{m-k}.

$d_1 = t(c) - k$ defined as $d_1 = \max\{t(c) - k, 1\}$.

2).

k	Pattern.	d_2
1	BAOBABAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB.	5
5	BAOBAB	5

Good suffix table.

k	Pattern	d_2
1.	AT_THAT	3.
2.	AT_THAT	4
3.	AT_THAT	5
4.	AT_THAT	5
5.	AT_THAT	5
6.	AT_THAT	5

k	Pattern	d_2
1.	REORDER	3
2.	REORDER	6.
3.	REORDER	6.
4.	REORDER	6.
5.	REORDER	6.
6.	REORDER	6.

k	GANGA	d_2
1.	GANGA	3
2.	GANGA	4
3.	GANGA	4
4.	GANGA	4

• Bayer-moore's algorithm

Step 1: For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described.

Step 2: Using the pattern, construct the good-shift table as described.

Step 3: Align the pattern against the beginning of the text.

Step 4: Repeat the following step until either a matching substring is found the pattern reaches beyond the last character of the text pattern, compare the corresponding characters in the pattern and the text until either all m characters pairs are matched or a mismatching pair is encountered. After $k \geq 0$ character pairs are matched successfully. In the last case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character.

If $k > 0$ also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by formula.

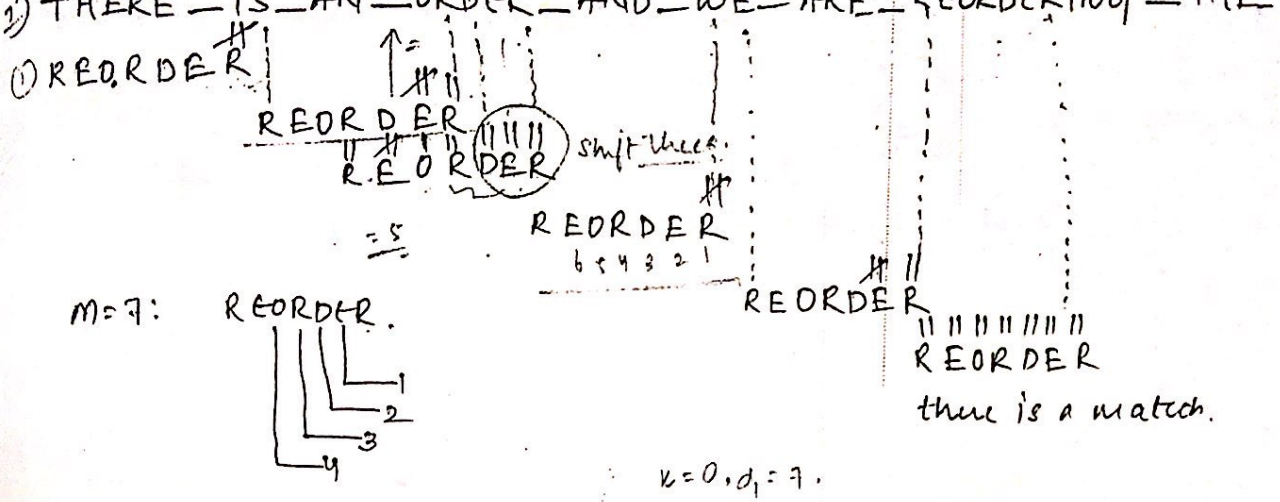
$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max\{d_1, d_2\} & \text{if } k > 0 \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

match character.

27/03/19

2) THERE_IS_AN_ORDER_AND_WE_ARE_REORDERING_THE



M=7:

k	Pattern	d_2
1.	REORDER	3
2.	REORDER	6
3.	REORDER	6
4.	REORDER	6
5.	REORDER	6
6.	REORDER	6

$k=0, d_1=7.$

① $t_1(c)-0 \{7-0,1\}$

② $k=1.$
$d_1 = \{t_1(0)-1,1\} : 3.$
$d_2 = 3.$
$\max\{d_1, d_2\}$ shift by 2 characters.

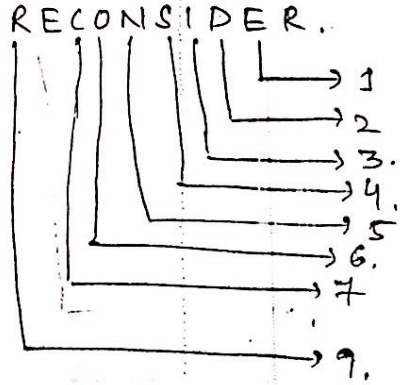
③ $k=5$ [illegible] match
$d_i = \{t_1(5)-k,1\} = \{t_1(5)-5,1\}$
$= 2.$
$d_2 = 6.$
$\max\{2,6\} = 6.$

④ $k=0, d_1 = \max\{t_1(w)-\cdots$

⑤ $k=1, d_1 = \max\{t_1(-)-\cdots$
$d_2 = 3.$
$\max\{6,3\} = 6.$

2) · PROBLEM_MUST_BE_RECONSIDERED_FOR_OPTIMALITY

k	RECONSIDER	d_2
1.	RECONSIDER	9.
2.	RECONSIDER	9.
	⋮	
8.		9.
9.		9.



m = 10.

PROBLEM — MUST — BE — RECONSIDERED — FOR — OPTIMAL.

RECONSIDER #.

RECONSIDER #

" " " " " " " "

RECONSIDER

There is a match!

∴ No. of comparison = 12.

(1) k = 0. compute d_1

$d_1 = \max\{t_1(h) - 0.1\}$

$= 10.$

(2) k = 0, $d_1 = 6.$

$d_1 = \max\{(1|0) - 1, 1)$

$= 7 - 1 = 6.$

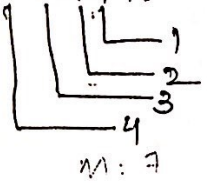
3) WHICH — FINALLY — HALTS — AT — THAT.

AT — THAT #

AT — THAT #

AT — THAT #.

AT — THAT.



(1) k = 0,

$d_1 = \max\{t_1(i) - 0.1\}$

$= 7.$

(2) k = 0.

$d_1 = \max\{t_1(h) - 0.1\}$.

[Dynamic programming is a technique for solving problem with overlapping subproblems.]

• DYNAMIC PROGRAMMING

Computing Binomial coefficient

$$C(n,k) = \begin{cases} C(n-1,k-1) + C(n-1,k) & \text{if } n>k>0. \\ C(n,n) = C(0,0) = 1 \end{cases}$$

$$\binom{n}{k} = \frac{n!}{(n-r)!\,k!} \quad \text{i.e } {}^{n}C_{r}.$$

$n+1$ rows.
$k+1$ columns.

	0	1	2	3	4	5	6	7	8	9
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126	84	36	9	1
10	1	10	45	120	210	252	210	120	45	10

$C(8,4) = 70.$
$C(5,3) = 10.$
$C(6,3) = 20.$
$C(5,4) = 5.$
$C(7,3) = 35.$
$C(10,6) = 210.$

$O(n) - S$
(n)
n^3

• Algorithm Binomial(n, k).

// Computes C(n,k) by dynamic programming algorithm
// Input: Positive integer n, k, n ≥ k ≥ 0.
// Output: value of C(n,k).

```
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = k
            C[i,j] ← 1
        else
            C[i,j] ← C[i-1, j-1] + C[i-1, j].
```

→ Analysis: (return C[n,k])

$$A(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1.$$

(triangle) (rectangle)

$$= \sum_{i=1}^{k} i-1-1+1 + \sum_{i=k+1}^{n} k-1+1.$$

$$= \sum_{i=1}^{k} i-1 + \sum_{i=k+1}^{n} k.$$

$$= 0+1+2+\ldots+k-1 + \left(\sum_{i=k+1}^{n} 1\right).$$

• $$A(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1.$$

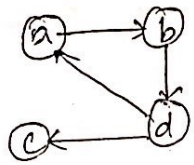
$$= \sum_{i=1}^{k} (i-1) + \sum_{i=k+1}^{n} k.$$

$$= \frac{k(k-2)}{2} + k(n-k) \approx \Theta(nk).$$

29/03/19

• **Transitive Closure**: Transitive closure of a directed graph with 'n' vertices can be defined as, n-by-n boolean matrix,

T = {t_ij} in which element in i^th row and j^th column is 1, if there exists a non-trivial directed path i.e. path of positive length 'n' (The path of length from vertex i to j) otherwise P_ij is zero.



$R^{(0)}$ A =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Adjacency matrix

T = $R^{(4)}$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

Transitive closure.

• Rules for generating elements of matrix: $R^{(k)}$ from elements of matrix $R^{(k-1)}$.

i) An element R_{ij} is 1 in $R^{(k-1)}$, it remains 1 in R^k.

ii) For element R_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in R^k iff the element in its row i and column j, and the elements in its row k and column j are both 1's in $R^{(k-1)}$.

A =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	0	1	1

R' =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	0	1	1

R2 =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

A(n).

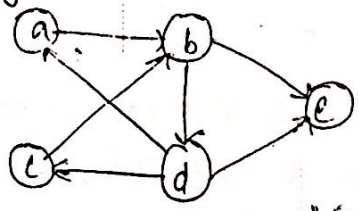
$R^{(3)}$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

R^4 =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

Eg:-



$e^0 = A =$

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	1	1
c	0	1	0	0	0
d	1	0	1	0	1
e	0	0	0	0	0

$R^1 =$

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	1	1
c	0	1	0	0	0
d	1	0	1	0	1
e	0	0	0	0	0

$R^2 =$

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	1	1
c	0	1	0	0	0
d	1	1	1	0	1
e	0	0	0	0	0

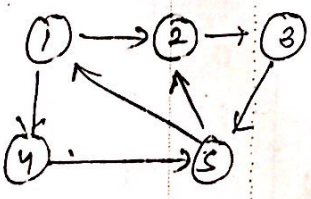
$R^3 =$

	a	b	c	d	e
a	0	1	0	1	1
b	0	0	0	1	1
c	0	1	0	1	1
d	1	1	1	1	1
e	0	0	0	0	0

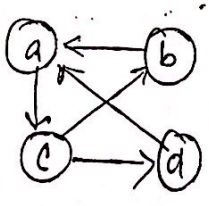
$R^4 =$

	a	b	c	d	e
a	1	1	1	1	1
b	1	1	1	1	1
c	1	1	1	1	1
d	1	1	1	1	1
e	0	0	0	0	0

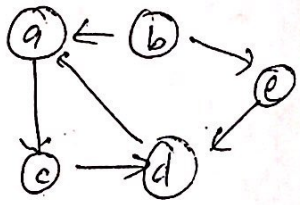
1).

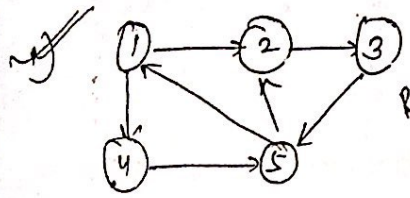


2)



3).





$R^0 = A =$

	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	1	1	0	0	0

$R^1 =$

	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	1	1	1	0	1

$R^2 =$

	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	1	1	1	1	0

$R^3 =$

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	1	1	1	1	1

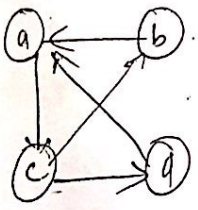
$R^4 =$

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	1	1	1	1	1

$R^5 =$

	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
5	1	1	1	1	1

2)



$R^0 =$

	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	1	0	0	0

$R^1 =$

	a	b	c	d
a	0	0	1	0
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

$R^2 =$

	a	b	c	d
a	0	0	1	0
b	1	0	1	0
c	1	1	1	1
d	1	0	1	0

4

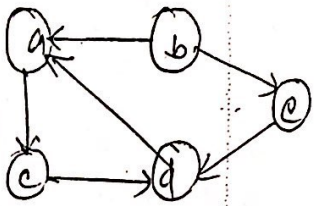
R^3 =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	1	1	1	1

R^4 =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	1	1	1	1

3)



$R^0 = A =$

	a	b	c	d	e
a	0	0	1	0	0
b	1	0	0	0	1
c	0	0	0	1	0
d	1	0	0	0	0
e	0	0	0	1	0

R^1 =

	a	b	c	d	e
a	0	0	1	0	0
b	1	0	1	0	1
c	0	0	0	1	0
d	1	0	1	0	0
e	0	0	0	1	0

R^2 =

	a	b	c	d	e
a	0	0	1	0	0
b	1	0	1	0	1
c	0	0	0	1	0
d	1	0	1	0	0
e	0	0	0	1	0

R^3 =

	a	b	c	d	e
a	0	0	1	1	0
b	1	0	1	1	1
c	0	0	0	1	0
d	1	0	1	1	0
e	0	0	0	1	0

R^4 =

	a	b	c	d	e
a	1	0	1	1	0
b	1	0	1	1	1
c	1	0	1	1	0
d	1	0	1	1	0
e	1	0	1	1	0

R^5 =

	a	b	c	d	e
a	1	0	1	1	0
b	1	0	1	1	1
c	1	0	1	1	0
d	1	0	1	1	0
e	1	0	1	1	0

03/04/19

- Algorithm Warshall (A[1....n, 1....n]).

// Implements Warshall's algorithm for computing the transitive closure.

// Input: The adjacency matrix A of a digraph with 'n' vertex.

// Output: The transitive closure of the digraph.

$R^{(0)} \leftarrow A$

for k ← 1 to n do

  for i ← 1. to. n do

    for j ← 1 to n do

$R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or ($R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$.)

return $R^{(n)}$.

→ Analysis [Basic operation : logic operation].

$$A(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$= \sum_{k=1}^{n} \sum_{i=1}^{n} n. = \sum_{k=1}^{n} (\underbrace{n+n+\dots+n}_{n^2}).$$

$$= n^2 + n^2 + \dots + n^2$$

$$= n^3.$$

$$= \theta(n^3).$$

$$A(n) \in \theta(n^3)$$

- Difference –

In bfs/dfs independently each node is taken as parent node.

In warshall, directly reachability is found.

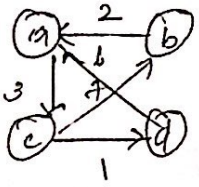
- Floyd's Algorithm:

→ It computes all pairs shortest path.

→ It considers cost matrix or weighed matrix cost matrix indicates the cost to travel from one to another

→ weight should be +ve otherwise not applicable.

Eg:



$$D^0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{bmatrix} a & b & c & d \\ 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}$$

$\infty + \infty = \infty$.

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{bmatrix} a & b & c & d \\ 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$2+3+\infty = \infty$

$5+\infty \rightarrow 5$ is min. so 5.

$2+7 = 9$ min.

$$D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{bmatrix} a & b & c & d \\ 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{bmatrix} a & b & c & d \\ 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

$$D^4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{bmatrix} a & b & c & d \\ 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

$i < 4$ so ?

$i = 6+1 = 7$.

- **Algorithm Floyd** ($W[1\ldots n, 1\ldots n]$). ← Weighted matrix

// Implements floyd's algorithm for all-pairs shortest paths prob.

// Input: The weight matrix W of a graph

// Output: The distance matrix of the shortest path's length

```
D ← W
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D.
```

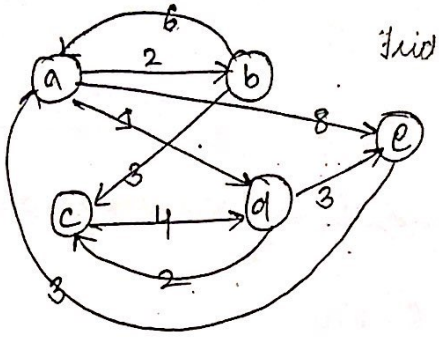
→ Analysis

→ Basic operation ← comparison

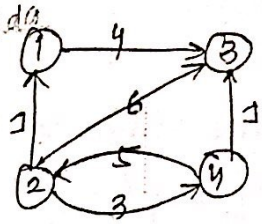
→ $A(n) \in \Theta(n^3)$.

Friday quiz ads

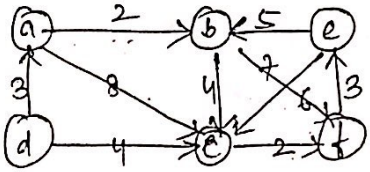
Problem: 1).



2)



3).



• The Knapsack problem:

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0. \\ V[i-1,j] & \text{if } j - w_i \leq 0. \end{cases}$$

$V[0,j] = 0$ and $V[i,0] = 0$ for $i \geq 0$ for $j \geq 0$.

Eg: Consider an instance given with following data.
Find the optimal solution.

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$

capacity j

	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	12	12	12	12	
2	0	10	12	22	22	22	$w_2 = 1, v_2 = 10$
3	0	10	12	22	30	32	$w_3 = 3, v_3 = 20$
4	0	10	15	25	30	37	$w_4 = 2, v_4 = 15$
5							

V[4,5] = 37

The maximal value is V[4,5] = 37.

V[4,5] ≠ V[3,5] is included in Knapsack and is a part of optimal sol.

Remaining capacity = 5 − w_4
= 5 − 2 = 3.

V[3,3] = V[2,3] not a part of minimal sol.

V[2,3] ≠ V[1,3] is a part of optimal sol.

Remcap = 3 − w_2
= 3 − 1 = 2

V[1,2] ≠ V[0,2] a part.

op. = {1, 1, 0, 1} ✓

12/04/19

• The Knapsack Problem :

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25.

Capacity $W = 10$.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42	42	42
2	0	0	0	12	12	12	12	42	42	42	54
3	0	0	0	12	40	40	40	52	52	52	54
4	0	0	0	12	40	40	40	52	52	65	**65**

The maximum value is $V[4,10] = \$65$.

$V[4,10] \neq V[3,10]$, item 4 is included into Knapsack.

Remaining capacity of Knapsack $= 10 - w_4 = 10 - 5 = 5$.

$V[3,5] \neq V[2,5]$, item 3 is a part of optimal solution.

Remaining capacity of Knapsack $= 5 - 4 = 1$.

$V[2,1] = V[1,1]$, item 2 is not included.

$V[1,1] = V[0,1]$, item 1 is not included.

optimal Solution = {item 3, item 4}.

$= \{0, 0, 1, 1\}$

1) Using dynamic programming solve following Knapsack

item	weight	value
1	1	$18
2	2	$16
3	2	$6

} we constructed this.

Capacity $w = 4$.

$n = 3$, $[w_1\ w_2\ w_3] = [1, 2, 2]$ and.

$[P_1, P_2, P_3] = [18, 6, 6]$ and $w = 4$

} Given.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	18	18	18	18	18
2	0	18	34	34	34	34
3	0	18	18	34	34	40.

column 5 not required

Maximal value is $34.

Optimal solution:

$V[3,4] = V[2,4]$ items not included

$V[2,4] \neq V[1,4]$, item is included

Remaining capacity $= 4-2 = 2$.

$V[1,2] \neq V[0,2]$ item is included

Rem. capacity $= 2-1 = 2$.

2)

item	weight	value
1	2	$20
2	1	$10
3	3	$25
4	2	$30
5	4	$15

$W = 10$.

3) $n = 4$, capacity $m = 8$.

$[v_1, v_2, v_3, v_4] = \{15, 10, 9, 5\}$

$[w_1, w_2, w_3, w_4] = \{1\ 5\ 3\ 4\}$.

2)

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											

- How many bytes contain
  a) exactly two 1's?
  $^8C_2 \times {}^6C_6$ ways
  b) exactly 4 1's?
  $^8C_4 \times {}^4C_4$ ways.
  c) atleast six 1's?
  $(^8C_6 \times {}^2C_2) + (^8C_7 \times {}^1C_1) + (^8C_8)$ ways.

GREEDY TECHNIQUE

Change Making Problem:

Rs 10, Rs 5, Re 1

RS 28,

2	coins of	10		20
1	coin of	5		5
3	coin of	1		1
				28.

We need to select [illegible] coin i.e. [illegible] denomination.

Should [illegible]

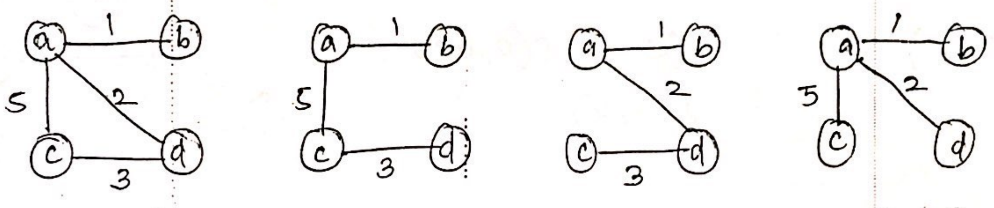
feasible

locally optimal

irrevocable [illegible]

Spanning tree:

Minimum spanning tree (MST).

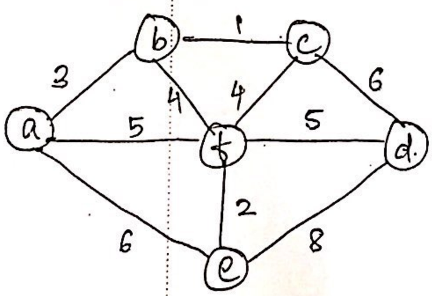


cost = 9   cost = 6   cost = 8.

↓

{Minimum spanning tree because cost is minimum among others.}

1)

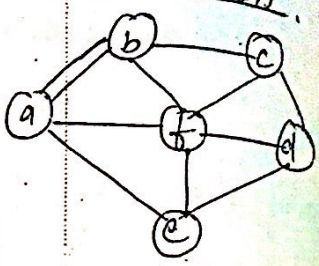
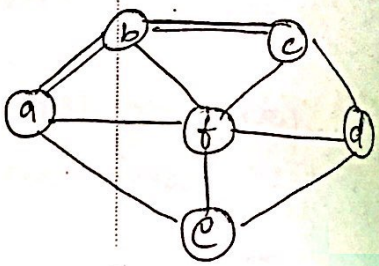
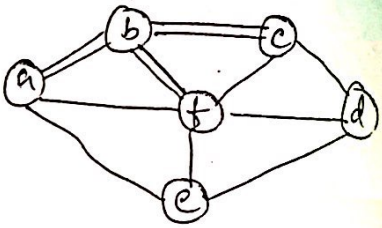
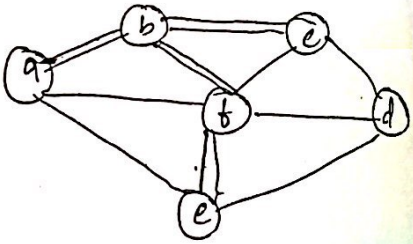
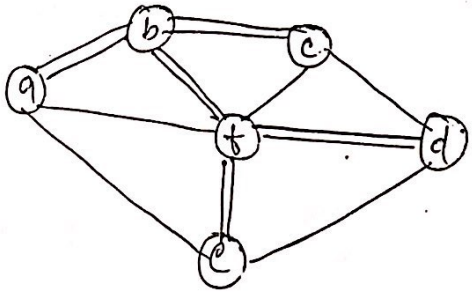


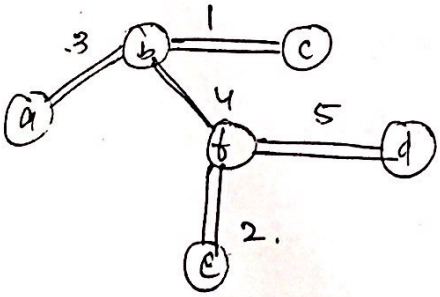
Two types of algorithm:

1) PRIM's

2) Kruskal's

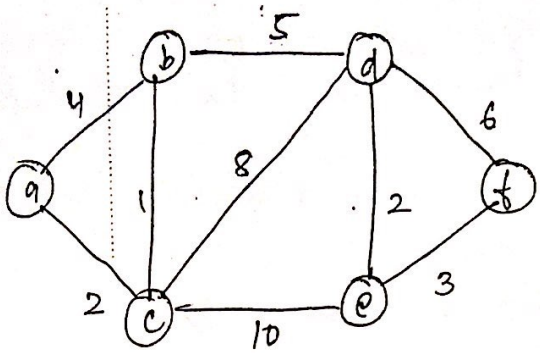
back.

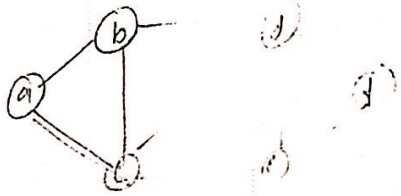
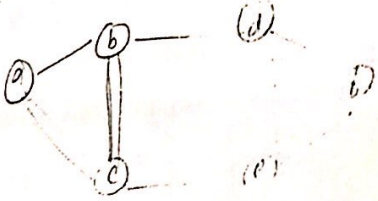
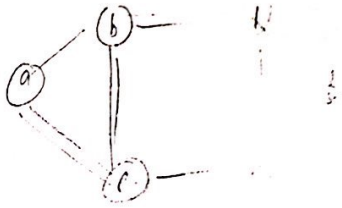
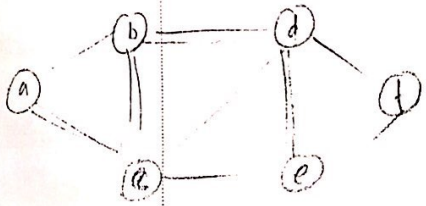
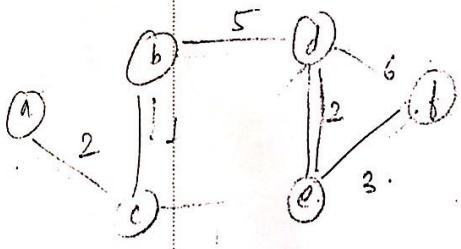
Tree vertex	Remaining Vertex	Illustration
a(–,–)	b(a,3), c(–,∞), d(–,∞) e(a,6), f(a,5)	
	among these minimum is b(a,3)	
b(a,3)	write rem. c,d,e,f. c(b,1) d(–,∞) e(a,6), f(b,4). comp → f(a,5) minimum. 4. So write 4. connection	
c(b,1)	d(c,6), e(a,6), f(b,4).	
f(b,4)	d(f,5), e(f,2)	
e(f,2)	d(f,5)	
d(f,5)		



weight of MST = 15 {3+1+4+5+2}.

2)



Tree vertex.	Remaining vertex	Illustration
a(-,-)	b(a,4), c(a,2), d(-,∞), e(-,∞), f(-,∞)	
c(a,2)	b(c,1), d(c,8), e(c,10), f(-,∞)	
b(c,1)	d(b,5), e(c,10), f(-,∞)	
d(b,5)	e(d,2), f(d,6)	
e(d,2)	f(e,3)	

MST weight = 2+1+5+2+3

= 13.

	0	1	2	3	4	5
	12	11	12	13	12	12
Array		11	12	13		
Freq.		1	3	2		
cumulative value		1	4	6		

distinct value	12	13
A[0] = 12	1	6
A[1] = 12	3	6
A[2] = 13	3	6
A[3] = 12	3	5
A[4] = 12	2	5
A[5] = 11	1	5

	0	1	2	3	4	5
				12		
			12			
						13
		12				
	11					
					13	
	11	12	12			

13/04/19

Algorithm PRIM(G)-

// Constructs the minimal spanning tree.

// Input: A weighted connected graph $G(V, E)$

// Output: Set of edges composes a minimum spanning tree of G.

$V_T \leftarrow \{v_0\}$ /* any arbitary vertex */

$E_T \leftarrow \phi$

for $i \leftarrow 1$ to $|V|-1$ do

find a mini-weighted edge $e^* = (v^*, u^*)$ among all edges.

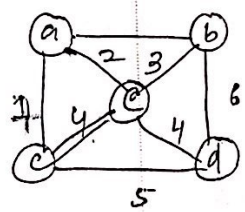
(u, v) such that v is in V_T and u in $V - V_T$.

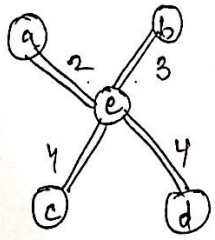
$V_T \leftarrow V_T \cup \{u^*\}$

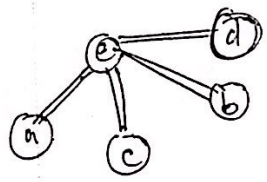
$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Tree vectors	Remaining vector
c(-,-)	a(c,7), b(-,∞) d(c,5) e(c,4)
e(c,4)	a(e,2), b(e,3), d(e,4)
a(e,2)	b(e,3), d(e,4)
b(e,3)	d(e,4)
a(e,4)	
$e^* = (e^*, d^*)$	





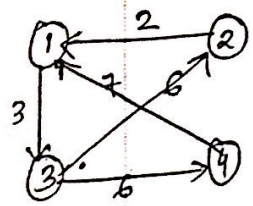


weight of MST

$V_T = \{c, e, a, b, d\}$

$E_T = \{(c,e), (e,a), (e,b), (e,d)\}$

1)

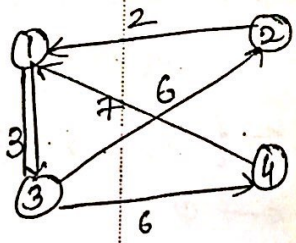


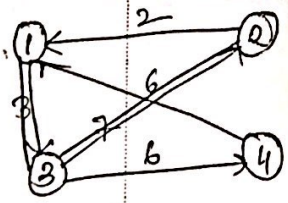
$V = \{1, 2, 3, 4\}$

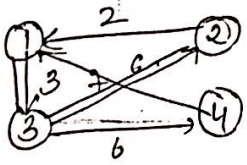
$E = \{(2,1), (1,3), (3,4), (3,2), (4,1)\}$.

Illustration

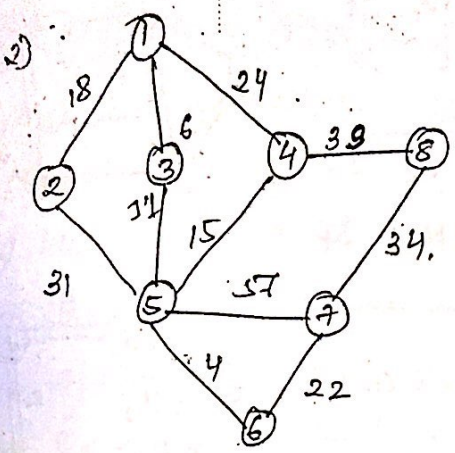
Tree vertex	Remaining vertex
1(-,-)	2(-,∞), 3(1,3), 4(-,∞)
3(1,3)	2(3,6), 4(3,6)
2(3,6)	







2)

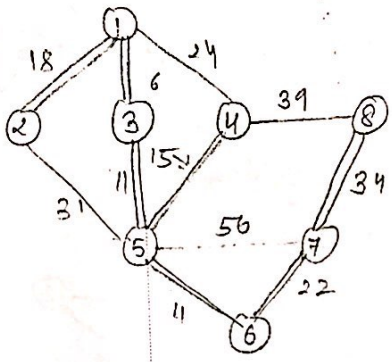


$E = \{(1,3), (3,5), (5,4), (5,6), (1,2), (6,7), (7,8)\}$.

$= 6+11+4+15+18+22+34.$

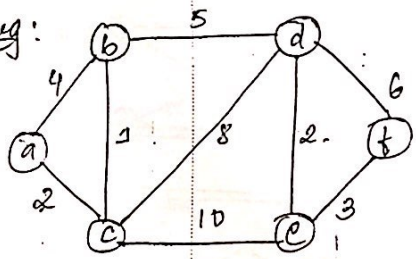
$= 110.$

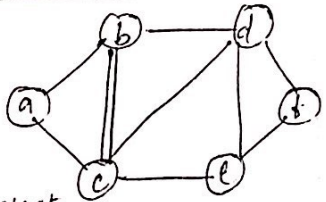
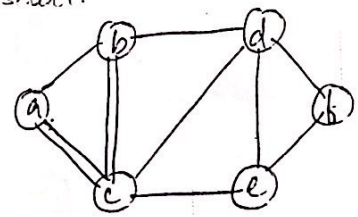
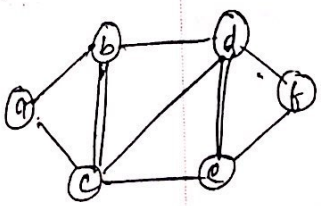
→



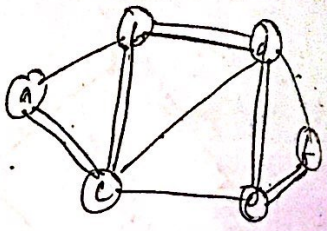
- **Kruskal's algorithm:** [Joseph Kruskal]

Eg:



Tree edges	Sorted list of edges	Illustration
bc 1	bc ac de ef ab bd 1 2 2 3 4 5 df cd ce 6 8 10 → doesn't form any cycle select.	
ac 2.	ac de ef ab bd 2 2 3 4 5 df cd ce 6 8 10	
de 2	de ef ab bd df 2 3 4 2 6 cd ce 8 10	

ef	ef ✓	ab	bd	df	cd	ce
3	3	4	5	6	8	10

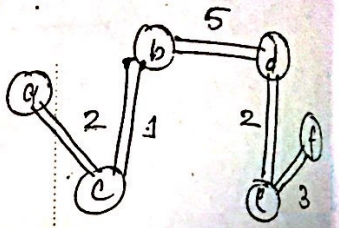


~~x (forms cycle don't select).~~

bd	ab	bd	df	cd	ce
5	4	5	6	8	10

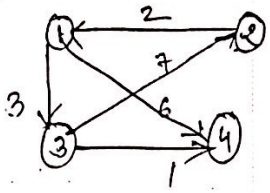
df	cd	ce
6	8	10.

} don't select.

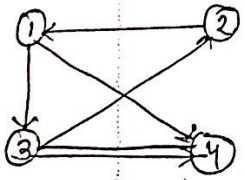


weight = 13.

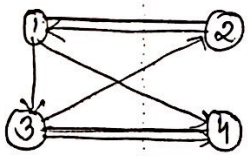
Eg:



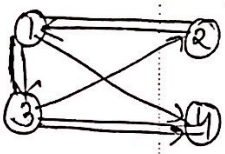
34	34 ✓	21	13	14	32
1	1	2	3	6	7



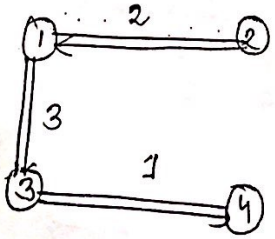
21	21 ✓	13	14	32
2	2	3	6	7



13	13 ✓	14	32
3	3	6	7



14 ✗	32 ✗
6	7



weight of MST = 3 + 2 + 1
= 6.

• Algorithm Kruskal (G):

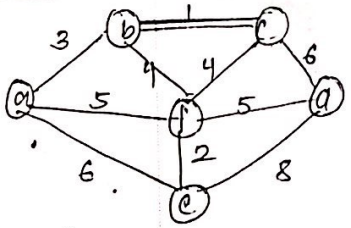
// Kruskal algorithm constructs the minimum spanning tree.
// Input: A weighted connected graph $G(V, E)$.
// Output: E_T a set of edges composing a MST of G.

Sort E in non-decreasing order of the edge weights
$w(e_{i_1}) \leq w(e_{i_2}) \leq \ldots \leq w(e_{i_{|E|}})$.
$E_T \leftarrow \phi$; ecounter $\leftarrow 0$.
(null set)
$k \leftarrow 0$
while ecounter $< |V| - 1$ do
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$
        ecounter $\leftarrow$ ecounter $+ 1$.
return E_T.

Eg:

Tree edges	Sorted list of edges.					
bc	bc	ef	ab	bf	cf	af
1	1	2	3	4	4	5
	df	ae	cd	ce		
	5	6	6	8		

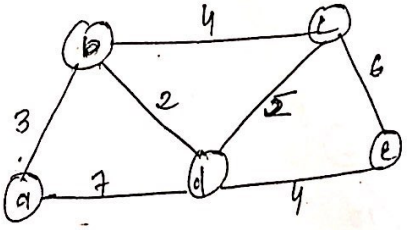
Illustration



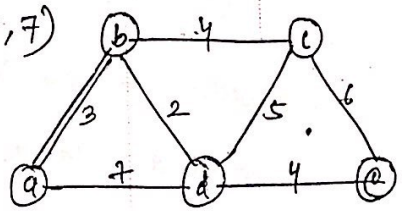
$E_T = \{(b,c), (e,f), (a,b), (b,f), (d,f)\}$.

• Dijkstra's algorithm

Eg:

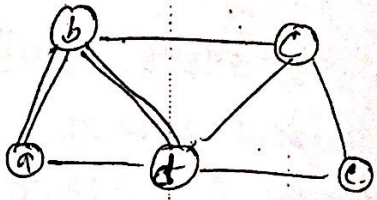


Tree vertex	Remaining vertex	Illustration
a(-, 0)	b(a, 3), c(-, ∞), d(a, 7)	
	e(-, ∞)	



b(a,3) c(b, 3+4) d(b, 3+2).

e(-,∞)

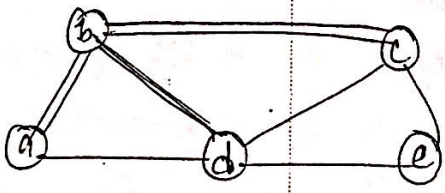


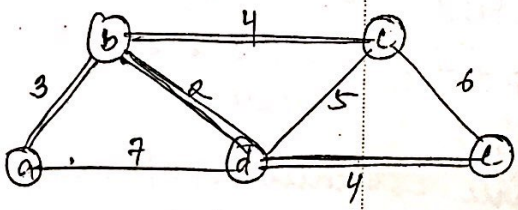
min

d(b,5) e(b,7), e(d,5+4)

c(b,7) e(d,9)

e(d,9)





a → b : 3 from a to b

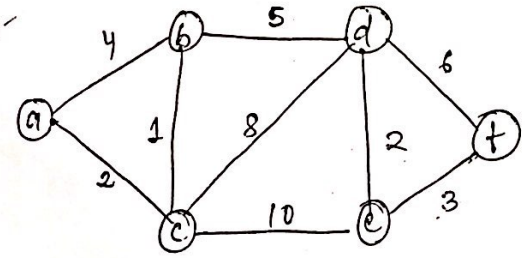
a → b → c : 7 from a to c

a → b → d : 5 from a to d

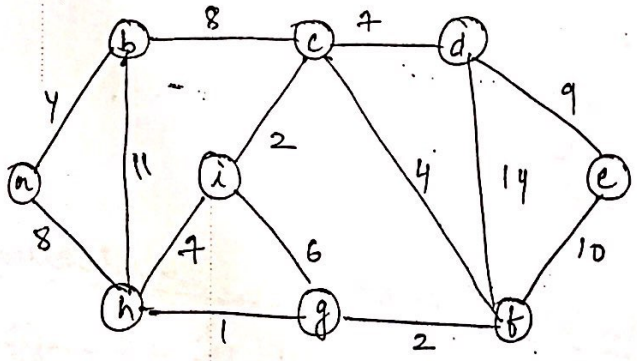
(a → e) i.e a → b → d → e : 9 from a to e.

Problem: Solve the following instance of single source shortest path problem with vertex a as first.

1)



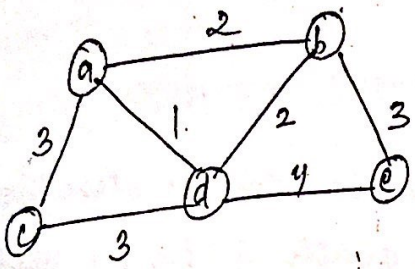
2)



i) Consider 'h' as source vertex

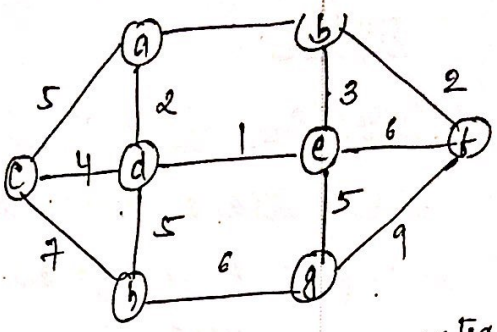
ii) Consider 'd' as source vertex.

3)



with 'c' as source vertex.

4)



i) with 'c' as source vertex

ii) with 'a' as source vertex.

15/02/19

- **Algorithm Dijkstra(G,S)**

// Dijkstra's algorithm for singee-source shortest paths.
// Input: A weighted connected graph $G=(V, E)$ and its vertex s.
// Output: The length d_v of a shortest path from s to v. and its penultimate vertex p_v for every vertex v in V.

```
Initialize (Q)   //Initialize vertex priority queue to empty
for every vertex v in V do
        dv ← ∞; pv ← null
ds ← 0; Decrease(Q,S,ds)
VT ← ∅
for i ← 0 to |V|-1 do
        u* ← DeleteMin(Q)
        VT ← VT ∪ {u*}
        for every vertex u in V-VT that is adjacent to u* do
                if du* + w(u*, u) < du
                        du ← du* + w(u*, u); pu ← u*
                        Decrease(Q, u, du).
```

- State space tree, promising & not promising node, track-table & non-tracable, decision tree, backtracking.
- Backtracking: n-Queen's Problem:

16/09/19

- **Subset sum problem-**

$S = \{1, 3, 4, 5, 6, 8\}$ $d = 13$ (i.e sum).

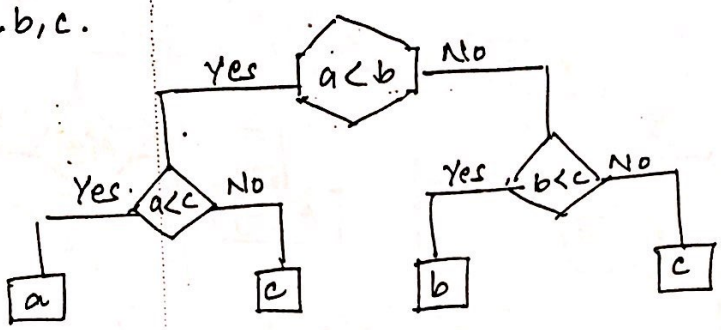
$S_1 = \{3, 4, 6\}$ $S_2 = \{5, 8\}$

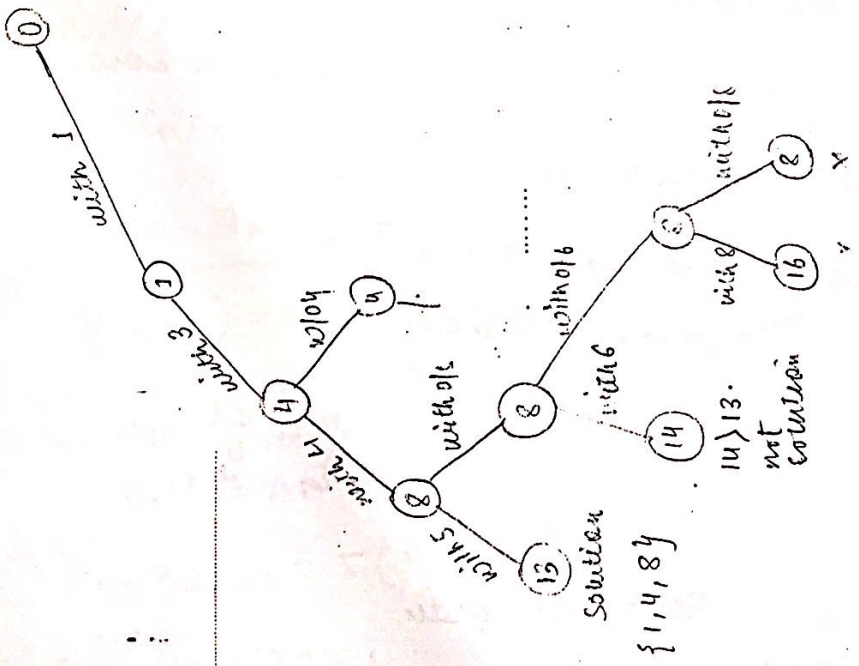
$S_3 = \{1, 3, 4, 5\}$ $S_4 = \{1, 4, 8\}$

- **Decision tree :**

1) finding minimum of three nos.

a, b, c.





2) Decision for binary search in a 4 element array.

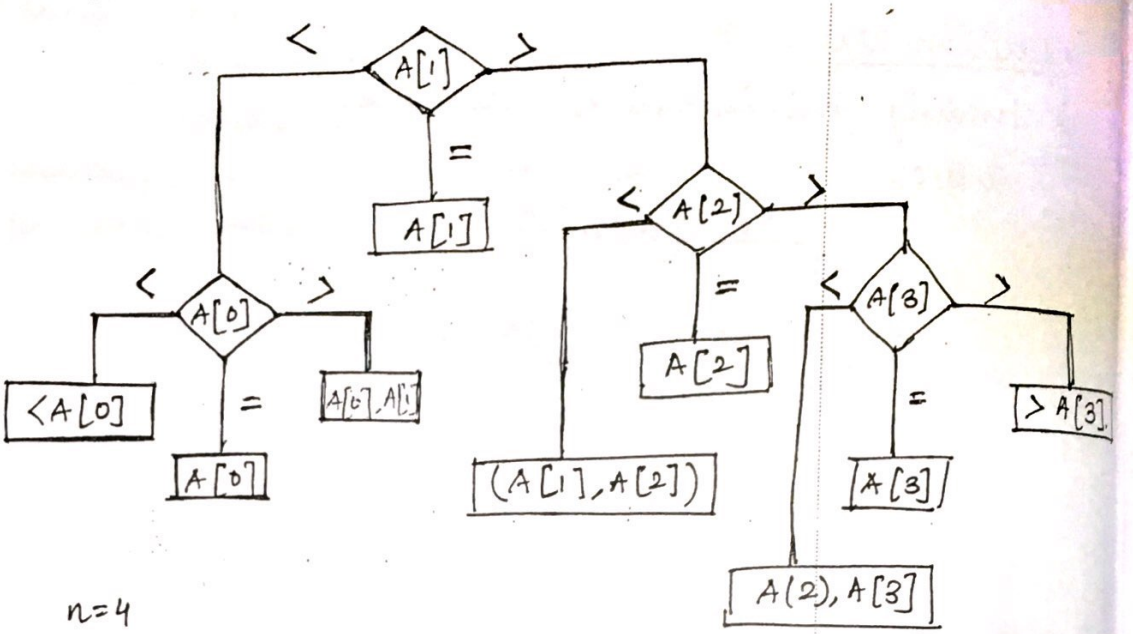
A[0]
A[1]
A[2]
A[3]

mid, low = 0, high = 3.

$mid = \left[(0+3)/2 \right] = 1$

high = mid - 1 = 0
mid = 0

low = 1 + 1 = 2
$mid = \frac{2+1}{2} = 2$



n = 4

9 heap node.

∴ 2n+1 heap node,
out of which n leaf nodes successful search.
n+1 leaf nodes unsuccessful search.

3) Decision tree for three element selection sort.

in' elements then n-1 pass.

a
b
c

a < b true then a < c T a.
→ false, then b < c — b
F c

b c
a c
c b
a

a
b
c

if b < c False acb order.
True
abc order.

• Non-Homogeneous equation –

Consider $a_n + c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} = f(n)$.

1st order

where $f(n) = (b_t n^t + b_{t-1} n^{t-1} + \dots + b_1 n + b_0) s^n$.

where $b_1, b_2, b_3 \dots b_t$ are constants.

$a_n = a_n^{(h)} + a_n^{(p)}$ → solution associated with particular part.

↑ Sol: related with homogeneous sol°

To find particular sol° –

$$F(n) = (b_t n^t + b_{t-1} n^{t-1} + \dots + b_0) s^n.$$

Case i: If s is not a root associated with characteristic.

$$a_n^{(p)} = (p_t n^t + p_{t-1} n^{t-1} + \dots + p_1 n + p_0) s^n.$$

Case ii: If s is a root then exists multiplicity "m" of the characteristic then $a_n^{(p)} =$

$$a_n^{(p)} = n^m (p_t n^t + p_{t-1} n^t + \dots + p_0) s^n.$$

Case iii: If $F(n) = c \sin \alpha n + C \cos \alpha n$.

$a_n^{(p)} = p_1 \cos \alpha n + p_2 \sin \alpha n$.

Case iv: $F(n) = c s^n \cos \alpha n$ (or) $c s^n \sin \alpha n$.

$a_n^{(p)} = p_1 s^n \cos \alpha n + p_2 s^n \sin \alpha n$.

2nd order $c_n a_n + c_{n-1} a_{n-1} + c_{n-2} a_{n-2} \dots c_{n-k} a_{n-k} = f(n)$.

G.S. $a_n = a_n^{(h)} + a_n^{(p)}$.

If $f(n) = r^n$, $a_n^{(p)} = A r^n$.

$a_n^{(p)} = A n r^n$

$a_n^{(p)} = A n^2 r^n$.

① $a_n - 3a_{n-1} = 5(7)^n$ → 1st order.

$a_n = 3a_{n-1}$.

$a_n^n = c_1 3^n$ $\quad f(n) = (7)^n$.

Eg: a=10 b=15 c=12 — 10 — a<b True → a<c True → a. (smaller).

15
12

b<c False → acb. if true abc order.

pass 1.

10
12
13

pass 2.

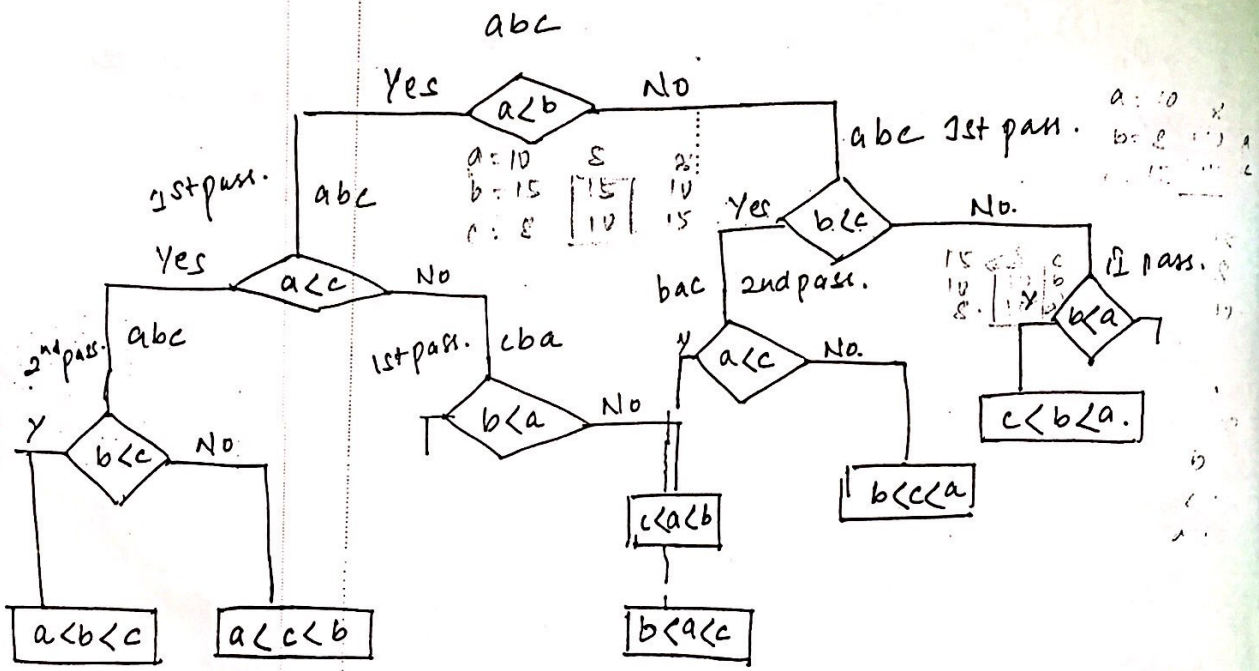
Eg: a=15 b=12 c=10 — 10 12 15 — pass 1.

a<b False b<c → c.

b<a True → cba i.e. 10 12 15 pass 2.

↓ False.

cab.



Eg: a=8 b=10 c=15 — a b c — 8 10 15 — COMPARED.

a=8 b=15 c=10 — 8 15 10 — 8 10 15