Algorithm

Defn: It is combination of sequence of finite steps to solve a particular problem.

Ex: addition of 2 numbers

```
add ()
{
  ① Take 2-num (a, b)
  ② c = a+b;
  ③ printf (c);
}
```


Properties of Algorithm :-

1. It should terminate after finite time.
2. It should produce atleast 1 output
3. It is independent of the programming language.
4. Every statement in the algorithm should be unambiguous.
   (For random generator, the algorithm might be non-deterministic i.e. produce diff outputs on same i/p)

Steps required to design algorithm :-

1. Problem definition. (knowing problem clearly)
2. Design algo [select one of the existing algorithms]
   ex: Divide & Conquer ....
3. Draw flowchart.
4. Testing
5. Coding / Implementation.

⑥ Analysis (finding time and space complexity.
↓
CPU time main memory space.

CH-1 Analysis of Algorithms

If the problem contains more than 1 solution Best one will be ~~defi~~ decided by analysis based on the ~~time~~ 2 factors
- Time
- Space.

Time Complexity :

$$T(P) = C(P) + R(P)$$

- $T(P)$ → Time complexity
- Program
- $C(P)$ → Compile Time → Compiler (S/W) → depends on Lang & Compiler
- $R(P)$ → Run Time → processor → H(W) → Type of H/W.

Java compiler can be written in C Lang.
C program executes in less time as compared to Java.
Earlier, Lang (Java) compiler cannot be written in Java But now it can be.
Bootstraping :- using others (ex. Java C for Compliers) for your use if you are incapable of it & later

Types of Analysis

postpone

Silent

A Posteriary analysis

① It is dependent of Lang of compiler and type of Hardware

② Exact

③ Time complexity changes from computer to computer H/w (Ex: i3 – slow, i7 – fast)

A Priori Analys

① It is independent of Lang. of compiler & type of H/w.

② Approximate.

③ Time complexity is constant in every computer.

Apriori Analysis :-

It is a determination of order of magnitude of a statement.

order of mag. → No of times a stmt will execute by processor

Ex-1

```
main ()
{
  x = y+z;  — 1
}
```

Time complexity = O(1)

Big-O notation

Ex-2

```
main ()
{
① x = y+z;  —— 1
```

```
for (i=1; i<=n; i++)
{
② x=y+z; ———————— n
}
}
```

Time complexity = $n+1 = O(n)$

↑ Larger of $n, 1$

Ex3.

```
main()
{
① x= y+z; ———————— 1
  for (i=1 to n)
  {
② x= y+z; ———————— n
  }
  for (i=1 to n)
  {
    for (j=1 to n)
    {
③ x= y+z; ——— n²
    }
  }
}
```

if change 'j' for loop like this

```
for (j=1 to n/2)
{
}
```

Time complexity = $n \cdot \frac{n}{2}$

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

Time complexity = $n^2+n+1 = O(n^2)$

Time complexity means finding the largest loop. If no loop in the program, then constant time complexity (1).

↓ where CPU spending max. time

$n=8 \quad 8/2 = 4/2 = 2/2 = 1$

$n=5 \quad 5/2 = 2.5/2 = 1.25/2 = 0.625$

$n=6 \quad 6/2 = 3/2 = 1.5/2 = 0.75$

Ex 4:

Divide

```
main()
{
    while (n >= 1)
    {
        n = n/2;
    }
}
```

$\longrightarrow \log_2 n$

$64 \geq 1$ → n

$32 \geq 1$ → $n/2$

$16 \geq 1$ → $n/4 = n/2^2$

$8 \geq 1$ → $n/8 = n/2^3$

$4 \geq 1$ → $n/16 = n/2^4$

$2 \geq 1$ → $n/32 = n/2^5$

$1 \geq 1$ → $n/64 = n/2^6$

$\log_2 n$

$16 \Rightarrow 5 = (4+1)$

$32 \Rightarrow 6$

$64 \Rightarrow 7$

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

$n = 2^k$

$\boxed{k = \log_2 n}$

If condition changed to $(n > 2)$ then $\frac{n}{2^k} = 2$

$\Rightarrow 2^{k+1} = n$

$k = \log_2 n - 1$

If $n = n/2$ changed to $n = n/3$ then $\frac{n}{3^k} = 1$

$\Rightarrow k = \log_3 n$

If inside while loop, $\left.\begin{matrix} n = n/2; \\ n = n/3; \end{matrix}\right\} \Rightarrow n = n/6$

Ex-5

Multiply

```
main()
{
  i = 1;
  while (i <= n)
  {
    i = 2 * i;
  }
}
```

Termination condtn is $i > n$

$n = 16$
$i = 1$
2
4
⋮
$2^k = 16$

$\log_2 2^k = \log_2 2^4$

$\Rightarrow k = 4 = \log_2 n$

Time Complexity $= \log_2 n + 1$

$= O(\log_2 n)$

- If $i = 2 * i$ is replaced by $i = \underline{20} * i$, then time complexity $= O(\log_{\underline{20}} n)$
- If i value initialized to 10, no change in time complexity, $\log_2 n - 10 = O(\log_2 n)$
- If inside while, $\left.\begin{matrix} i = 2 * i; \\ i = 3 * i; \end{matrix}\right\} i = 6 * i;$
- If inside while, $\left.\begin{matrix} i = 2 * i; \\ i = 3 * i; \\ i = 5 * i; \\ i = i/10; \end{matrix}\right\} i = 3 * i; \Rightarrow \log_3 n$

Ex 6. Subtraction

```
main()
{
  while (n >= 1)
  {
    n = n - 1;
  }
}
```

$n \to n+1 = O(n)$

n, $n-1$, $\vdots$, $\frac{1}{0}$

```
while(n > 1)
{
  n = n - 10;
}
```

$n-10$, $n-20$, $\vdots$

$n - 10 \cdot k = 1$

$k = \frac{n-1}{10} = O(n)$

```
while (n > 15)
{
  n = n - 50;
}
```

$n - 50k = 15$

$k = \frac{n-15}{50} = O(n)$

```
while (n > 1)
{
  n = n - 10;
  n = n - 20;
}
```

$\Big\}\ n = n - 30;$

Ex 7:

Addition

```
main()
{
    i = 0;
    while (i < n)
    {
        i = i + 1;
    }
}
```

$n = 100$

0, 1, 2, 3, ..., 100 } $n + 1 = O(n)$

```
i = 1;
while (i < n)
{
    i = i + 10
}
```

1, 1+10, 1+2*10, ..., 1+k*10 = n

$k = \frac{n-1}{10} = O(n)$

```
while (i < n)
{
    i = i + 10
    i = i + 20
}
```

$i = i + 30 \Rightarrow n/30 = O(n)$

```
while (i < n)
{
    i = i + 10
    i = i + 20
    i = i - 5
}
```

$i = i + 25 \Rightarrow n/25 = O(n)$

Ex 8

```
main()
{
    while (n>2)
    {
        n = √n
    }
}
```

Time Complexity = $\log_2(\log_2 n)$

$n = 16$

$\sqrt{16} = 4$ $\quad n^{1/2}$

$\sqrt{4} = 2$ $\quad n^{1/4}$

$n^{1/2^k}$ $\quad n^{1/2^k}$

$n^{1/2^k} = 2$

$\Rightarrow \frac{1}{2^k} = \log_n 2$

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

$\Rightarrow k = \log_2\left(\frac{1}{\log_n 2}\right)$

$= \log_2(\log_2 n)$

-
```
while (n > 20)
{
    n = √n
}
```

$\Rightarrow n^{1/2^k} = 20$

$\frac{1}{2^k}\log_{20} n = 1$

$\Rightarrow 2^k = \log_{20} n$

$\log_2 2^k = \log_2 \log_{20} n$

$\boxed{k = \log_2 \log_{20} n}$

$\log_2$ due to $\sqrt{}$; $\log_{20}$ due to while condtn

-
```
while (n > 125)
{
    n = n^(1/23)
}
```

$\Rightarrow \log_{23} \log_{125} n$

-
```
while (n > 125)
{
    n = n^(1/5)
    n = n^(1/3)
}
```
} $n = n^{1/4}$

$\Rightarrow \log_{15} \log_{125} n$

Ex 9.

```
main ()
{  i=2
   while (i<n)
   {
      i = i^2
   }
}
```

$\Rightarrow O(\log_2 \log_2 n)$

2
2^2
2^4
:
2^{2^k}

$2^{2^k} = n$

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

$\Rightarrow 2^k = \log_2 n$

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

-
```
i=2
while (i<n)
{
   i = i^23
}
```

2
2^{23}
$(2^{23})^{23} = 2^{23^2}$
:
2^{23^k}

$2^{23^k} = n$

$\log_2 2^{23^k} = \log_2 n$

$23^k = \log_2 n \Rightarrow \boxed{k = \log_{23} \log_2 n}$

-
```
i=25
while (i<n)
{
   i = i^2
   i = i^3
}
```

$i = i^6 \Rightarrow \log_6 \log_{25} n$

```
while (n > 5)
{
    n = n - 2
    n = n / 2
    n = √n
}
```

If simplification not possible, remove the unnecessary & take that stmt which has max impact.
Here, $n = \sqrt{n}$

Ex 10:

```
main()
{
    for (i = 10; i ≤ n^3; i = i^15 * i)        → log_15 (n^3/10)
    {
        for (j = n^5; j > 10; j = j^1/16)      → log_16 (5 log_10 n)
        {
            for (k = 1; k ≤ n; k = k + 10)     → n/10
            {
                x = x + y
            }
        }
    }
```

10	10	n^5
~~10 * k~~	15 * 10	$n^{5/16}$
~~10 * k * k~~,	10 * 15 * 15	$n^{5/16^2}$
	⋮	⋮
	$10 * 15^k = n^3$	$n^{5/16^k} = 10$

1
1 + 10
1 + 2·10
⋮
1 + k·10 = n + 1
k = (n + 1 − 1)/10

$\Rightarrow 15^k = n^3/10$

$\Rightarrow k = \log_{15}\left(\frac{n^2}{10}\right)$

$\log_{n^{16^k}} n^{\frac{5}{16^k}} = \log_n 10$

$\Rightarrow \frac{5}{16^k} = \log_n 10$

$\Rightarrow 16^k = 5 \log_{10} n$

$k = \log_{16}(5 \log_{10} n)$

Time complexity $= \log_{15}\left(\frac{n^3}{10}\right) * \frac{\log_{16}(5\log_{10} n)}{} * \frac{n}{10}$

$\Downarrow$

$\log_{16} \log_{10} n^5$

Ex 11:

```
main ()
{
   for (i = 10; i ≤ n^2; i = i^4)  ⟶ log_4(log_10 n^2)
   {
      for (j = 1 ; j ≤ n ; j++)  ⟶ n(n+1)/2
      {
         for (k = 1 ; k ≤ j ; k = k++)
         {
            x = y + 2
         }
      }
   }
}
```

Dependency (So, individual cannot be taken)

$j=1 \quad j=2 \quad j=3 \qquad j=n$

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

10
10^4
10^{4^2}
:
$10^{4^k} = n^2$

$\log_{10} 10^{4^k} = \log_{10} n^2$

$\Rightarrow 4^k = \log_{10} n^2$

$k = \log_4(\log_{10} n^2)$

Time Complexity =

$\log_4(\log_{10} n^2) \cdot \frac{n(n+1)}{2}$

EX12

```
main()
{
  for(i=1; i<=n; i++)
  {
    for(j=1; j<=i^2; j++)
    {
      for(k=n^3; k>5; k=k^(1/6))
      {
        x=y+z;
      }
    }
  }
}
```

$\Rightarrow \frac{n(n+1)(2n+1)}{6}$

$\Rightarrow \log_6 \log_5 n^3$

$i=1$	$i=2$	$i=3$	$\dots$	$i=n$
$j=1$	$j=4$	$j=9$		$j=n^2$

$\Rightarrow 1+4+9+\dots = \frac{n(n+1)(2n+1)}{6}$

$\times 5$

$5^{1/6}$

$5^{1/6^2}$

$\vdots$

$5^{1/6^k} = n^3$

Time complexity $= \frac{n(n+1)(2n+1)}{6} \cdot \log_6 \log_5 n^3$

$= n^3 \cdot \log_6 \log_5 n^3$

$\Rightarrow \log_5 5^{1/6^k} = \log_5 n^3$

$\Rightarrow \frac{1}{6^k} = \log_5 n^3 \Rightarrow \log_6 6^{-k} = \log_6 \log_5 n^3$

$\Rightarrow k = -\log_6 \log_5 n^3$

Ex 12.

```
main()
{
    for (i=1; i<=n; i++)
    {
        for (i=1; i<=n^2; i++)
        {
            for (i=1; i<=n^3; i++)
            {
                x = y+z;
            }
        }
    }
```

n^3

$= O(n^3)$

$i = 1, 2, 3, \ldots, n^3, n^3+1, n^3+2, n^3+3$

```
for (i=1; i<=n; i++)  ⟶ 1
{
    for (j=1; i<=n^2; i++)  ⟶ n^2
    {
        for (k=1; k<=n^3; k++)  ⟶ n^3
        {
            x = y+z;
        }
    }
}
```

$i = 1, 2, \ldots, n^2, n^2+1, n^2+2$

$\Rightarrow n^3 \cdot n^2$

$= O(n^5)$

```
for (i=1; i≤n; i++)
{
    for (i=1; i≤n^4; i++)
    {
        for (i=1; i≤n^3; i++)
        {
            x = y + z
        }
    }
}
```

i = 1 1 1 ... n^3 n^3+1 n^3+2 ... $\leq n^4$

n^3 n^3+1 n^3+2 ... n^8 n^3+1 n^3+2 ...

Infinite Loop

So, it is not an algorithm

Ex 14

```
main()
{
    for(i=1; i≤n; i++)  ⟶ n
    {
        for(j=1; j≤n; j++)  ⟶ n
        {
            if(n%j==0)   n²   j=1 k=1 to n   j=n k=1 to n
            {
                for(k=1; k≤n; k++) ⟶ 2
                {
                    x = y + z;   2n² — Total.
                }
            }
        }
    }
}
```

n is prime

n^2 : j=1 (k=1 to n) (n), j=2 — j=n (k=1 to n) (n)

j=1 n ; j=n n

$\Rightarrow n \cdot (2n) \Rightarrow n^2 + 2n$ (out of n^2 times, $2n$)

$\Rightarrow O(n^2)$

Time complexity = $O(n^2)$

- for (i=1; i≤n; i++) ⟶ n

{

for (j=1; j≤n; j++) ⟶ n

{

if (n%2 == 0) ⟹ n^2

{

for (k=0; k≤n; k++)

{

x=y+2; ⟶ 0

}

}

}

Time complexity

$= O(n^2)$

n — prime

- for (i=1; i≤n; i++) ⟶ n

{

for (j=1; j≤n; j++) ⟶ n

{

if (j%2 == 0)

{

for (k=0; k≤n; k++) ⟶ n/2

{

x=y+2;

}

}

}

$n \cdot n \cdot \frac{n}{2} = \frac{n^3}{3} = O(n^3)$

Ex15

```
main()
{
    for(i=1; i^2 <= n; i++)          ---> sqrt(n)
    {
        for(j=1; j^5 <= n^10; j++)   ---> n^2
        {
            for(k=1; k^25 <= n^100; k++)  ---> n^4
            {
                x=y+z;
            }
        }
    }
}
```

1^{25}

2^{25}

3^{25}

$\vdots$

$k^{25} = n^{100}$

$k = n^{100/25}$

$= n^4$

$j^5 = n^{10}$

$j = n^{10/5}$

$= n^2$

$i^2 = n$

$i = \sqrt{n}$

Time Complexity $= \sqrt{n} \cdot n^2 \cdot n^4 = n^{13/2}$

$= O(n^{13/2})$

Ex 16.

```
A(n)
{
  if (n < 2)
    return;
  else
    return (A (√n))
}
```

n

$\sqrt{n} = n^{1/2}$

$n^{1/4}$

$n^{1/8}$

$n^{1/2^k} = 2$

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

$\Rightarrow 2^k = \log_2 n$

$$\boxed{k = \log_2 \log_2 n}$$

- For every recursive program, there exists an equivalent non-recursive program also.

Ex 17.

```
main()
{
  p=1;
  for (i=1; i<=n; i=2*i)
    p++;
}
```

```
main()
{
  p=1, q=1
  for (k=1; k<=n; k++)  ---> n
  {
    p=1;
    for (i=1; i<=n; i=2*i)  ---> log n (base 2)
      p++;   p = log n (base 2)
    for (j=1; j<=p; j=2*j)  ---> log (log n) (base 2)
      q++;
  }
}
```

i = 1, 2, 4, 8, ...

Time complexity = $n \cdot (\log_2 n + \log_2(\log_2 n))$ Take larger

$= n \cdot \log_2 n$

q-value = $n \cdot \log_2(\log_2 n)$, p-value = $n \log_2 n$

Ex18.

```
main()
{
  x=1;
  for (i=1; i<=n; i=i+10) ⟶ n/10.
  {
    for (j=1; j<=n; j=2*j) ⟶ log2 n
    {
      x=x+n;
    }
  }
}
```

Time complexity = $\frac{n}{10} \cdot \log_2 n$.

$= O(n \log_2 n)$

x-value = $1 + \left(\frac{n}{10} \cdot \log_2 n\right) \cdot n$

$= 1 + \frac{n^2}{10} \log_2 n$

$= 1 + n^2 \log_2 n = O(n^2 \log_2 n)$

Asymptotic Notation

1. Big - oh (O)
2. Omega (Ω)
3. Theta (Θ)

Let $f(n)$ && $g(n)$ be 2 positive functions

Big - oh notation : → order of

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

iff

$$f(n) \leq c \cdot g(n), \forall n, n \geq n_0$$

such that $\exists$ 2 - tive constants c & n_0

where $c > 0$ && $n_0 \geq 1$

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

$g(n) = n^2$

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

$$f(n) \leq c \cdot g(n)$$

$$\Rightarrow n^2 + n + 1 \leq c \cdot n^2$$

$$\Downarrow \quad \Downarrow \quad \Downarrow$$

$$n^2 \quad n^2 \quad n^2$$

$$3n^2$$

$$\Rightarrow n^2 + n + 1 \leq c \cdot n^2, \forall n, n \geq n_0$$

$$\Downarrow \qquad \Downarrow$$

$$3 \qquad 1$$

$\Rightarrow n^2+n+1 = O(n^2)$

If $c=2$, $n_0=2$.

$n^2+n+1 \leq n^2$ (X)

$n^2+n+1 = O(n^2)$ ✓

(C-hidden)

$a = O(b)$ means b is asymptotically greater than a.

↓ constant use

$a \leq b$ means b is mathematically greater than a

Ex 2: $f(n) = n+10$

$g(n) = n-10$ Prove $f(n) = O(g(n))$

$\Rightarrow n+10 \leq c(n-10)$

$c=2$

$n+10 \leq 2(n-10)$

$2n-n \geq 30$

$n \geq 30$

$n_0 = 30$ and $c=2$

$\Rightarrow (n+10) = O(n-10)$

$100 = O(2)$

$100 \leq c \cdot 2$

$\Downarrow$

51

$c=2$

$n+10 \leq 2n-20$

$30 \leq 2n-n$

$n \geq 30$

41

$2(31-10)$

2×21

42 ✓

50

$2(40-10)$

60 ✓

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

$g(n) = n$

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

$n^2 \leq c \cdot n$

$\Downarrow$

n

(but it is not a constant)

~~$n \geq 10$~~

~~$100 \leq 11 \cdot 10$~~ ~~$c = 11$~~

$n = 20$

~~$400 \leq 11 \cdot 20$~~

$\Rightarrow n^2 \neq O(n)$

Ex 4. $f(n) = n + 10$

$g(n) = n + 10$ ——

$\Rightarrow (n-10) \leq c \cdot (n+10)$

$\Downarrow$

1

$n - 10 \leq n + 10 \quad \forall n, n_0 \geq 11$

(as fncts should be +ive)

$\Rightarrow n - 10 = O(n+10)$

$n+10$ is mathematically as well as asymptotically greater.

Omega Notation:

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

iff

$$f(n) \geq c \cdot g(n) \quad \forall\ n,\ n \geq n_0$$

such that $\exists$ 2 +ive constants c & n_0

where $c > 0$ && $n_0 \geq 1$

Ex1 $f(n) = n^2$

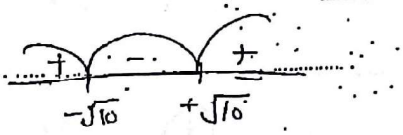
$g(n) = n^2 + 10$

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

$$n^2 \geq c \cdot (n^2 + 10) \qquad c = 1/2$$

$$n^2 \geq \frac{n^2}{2} + 5$$

$$\frac{n^2}{2} \geq 5 \Rightarrow$$



$$n^2 - 10 \geq 0 \Rightarrow (n+\sqrt{10})(n-\sqrt{10}) \geq 0$$

$$\Rightarrow n \geq \sqrt{10}$$

So, $c = 1/2$ and $n \geq 4$.

$$\Rightarrow n^2 = \Omega(n^2 + 10)$$

Ex2. $f(n) = n+10, \quad g(n) = n-10$

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

$$\Rightarrow n+10 \geq c(n-10) \qquad c = 1 \ \&\ n \geq 11.$$

already true

Ex 2: $f(n) = n$, $g(n) = n^2$ Prove $f(n) = \Omega(g(n))$

$\Rightarrow n \geq c \cdot n^2$

$\downarrow$

$1/n$ (which is not a constant)

so, not possible.

~~f(n)~~ $n \neq \Omega(n^2)$

Theta (Θ) - notation :-

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

iff

① $f(n) \leq c_1 g(n)$

&&

② $f(n) \geq c_2 \cdot g(n)$

$\forall n, n \geq n_0$.

Ex-1: $f(n) = n^2$, $g(n) = n^2 + 10$

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

$n^2 \leq c_1 \cdot (n^2 + 10)$

$c_1 = 1, n_0 = 1$

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

$n^2 \geq c_2 \cdot (n^2 + 10)$

$c_2 = \frac{1}{2}, n_0 = 4$

So, $c_1 = 1$, $c_2 = 1/2$ and $n_0 = 4$.

$\Rightarrow n^2 = \theta(n^2+10)$ ✓

Ex2:

$f(n) = n+10$

$g(n) = n-10$

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

$\Rightarrow n+10 \leq c_1(n-10)$

$c_1 = 2$; $n_0 = 30$

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

$n+10 \geq c_2(n-10)$

$c_2 = 1$, $n_0 = 11$

$\Rightarrow c_1 = 2$, $c_2 = 1$, $n_0 = 30$

$\Rightarrow (n+10) = \theta(n-10)$ ✓

Ex3:

$f(n) = n$

$g(n) = n$

$n \leq c_1 n$

$\Downarrow$

1

$n = O(n)$

$n \geq c_2 \cdot n$

$\Downarrow$

1

$0 = \Omega(n)$

$\Rightarrow n = \theta(n)$ ✓

Ex 4: $f(n) = n$

$g(n) = n^2$

$n \leq c_1 n^2$

$\Downarrow$

1

$n = O(n^2)$ ✓

$n \geq c_2 \cdot n^2$

$\Downarrow$

$1/n$

$n \neq \Omega(n^2)$

So, $n \neq \Theta(n^2)$

$100 = \Theta(1)$ ✓

$c_1 = 1$

(Omega ✓)

$c_1 = 100$

(Big-oh ✓)

Big-oh Notation ($\leq$) {There exists c}

$n^2 = O(n^2) \Rightarrow$ Tighest UB

$= O(n) \Rightarrow$ Upper Bound

$= O(n^3)$

$= O(n^{10})$ } NTUB

$A = O(B)$

$\Downarrow$

UB

T NT

Small-oh-notation ($<$) {For all c}

~~$n^2 = O(n^2)$~~ X

$O(n^3)$ ✓ $\Rightarrow$ UB

$O(n^{10})$ ✓ $\Downarrow$ NT

$A = o(B)$

$\Downarrow$

NTUB

Omega Notation ($\geq$) {There exists c}

$n^3 = \Omega(n)$

$\Omega(n^2)$ } — Lower Bound

$\Omega(n^3) \Rightarrow$ TLB

Small - omega notation (>)

$n^3 = \omega(n) \Longrightarrow$ NTLB

$= \omega(n^2)$

$= \omega(n^3)$ ×

$A = \{ \omega(B)$

⇓

NTLB

Theta Notation

$n^3 = O(n^3) \Rightarrow$ TUB

&

$n^3 = \Omega(n^3) \Rightarrow$ TLB

⇕

$n^3 = \Theta(n^3)$

⇕

$A = \Theta(B)$

⇓

TUB

TLB

If there is an algorithm A, time complexity of algorithm is T(A) then if $T(A) = O(n^3)$, it means that upper bound for time complexity is n^3.

Problem P

(algorithms)

A B

Time Complexity: T(A) T(B)

$T(A) = O(T(B))$

↓

smaller or equal to T(B)

$\Rightarrow$ A is better algorithm

$T(A) = T(B)$ (n^3 vs n^3) ✓

$T(A) = \theta(T(B))$ (n^3 vs n^2+10) ⟶ constant diff. may be there.

Complexity Classes :- (Imp)

(1) Constant :- $O(1)$

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

(2) Logarithmic :- $O(\log n)$

$O(\sqrt{n})$

(3) Linear :- $O(n)$

$O(n \log n)$

(4) Quadratic :- $O(n^2)$

(5) Cubic :- $O(n^3)$

(6) Polynomial :- $O(n^c)$ where c is constant, $c > 0$

(7) Exponential :- $O(c^n)$ where c is constant, $c > 1$

increasing order ↓

(8) $c^n < n^n$ ⟹ $c=2$ ⟹ $2^n < n^n$

$2^n = O(n^n)$ ✓

$o(A)$ satisfied $\Longrightarrow$ $O(A)$ satisfied.

$O(A)$ satisfied $\not\Longrightarrow$ $o(A)$ satisfied.

- $2^n > n^{\log n}$ Taking log $n \log 2 > \log n . \log n$.

(9)
- $n > \log n$
- $n > (\log n)^2$

To check:

① $\sqrt{n} . \sqrt{n} > (\log n) . (\log n)$ $\{\sqrt{n} > \log n\}$

Proved.

② $n > (\log n)^2$

Taking log, $\log n > 2 \log(\log n)$ $\{\log n > \log \log n\}$

(2 ↓ const.)

Proved.

⋮

- $n > (\log n)^{1000}$

Imp
- $n < (\log n)^{\log n}$

To check,

$\log n . 1 < \log n . \log(\log n)$

$\log(\log n) > 1$.

(10) • $n! < n^n$ $\quad \underbrace{n.(n-1)\ldots 1}_{n\text{-times}} \;\Big|\; \underbrace{n.n.n\ldots n}_{n\text{ times}}$

• $n! > 2^n$ $\quad \underbrace{n.(n-1)\ldots 1}_{n\text{-times}} \;\Big|\; \underbrace{2.2.2\ldots 2}_{n\text{ times}}$

$2^n < n! < n^n$

(11) $\log_2 n = \theta(\log_3 n)$

$\log_2 n > \log_3 n$

$\Downarrow$

$\frac{\log_2 n}{\log_2 3} = \frac{\log_2 n}{1.5}$

Difference is constant.

$\Rightarrow$ They are asymptotically equal.

(12) $2^n < 3^n$

$\Downarrow$

$(2 * 1.5)^n$

$\Downarrow$

$2^n \cdot 1.5^n$ → greater by function so not asymptotically equal.

$2^n = \theta(3^n)$

↓

small-o

~~$n^2 < n^3$~~ Take log

$2 \log n \;|\; 3 \log n$

If ignore constant, it seems

Simplify first, then apply log.

n^2	n^3
~~x^x~~	$n \cdot x^x$
1	n

Take Log

$\log 1 = 0 < \log n$ ✓

Ques 1: Check the foll. stmts are true/false?

(a) $250\, n \log n = O\left(\frac{n \log n}{250}\right)$ ✓ $c = 250 \times 250$

(b) $\sqrt{\log n} = O(\log \log n)$ X — Ω, ω ✓

(c) If $0 < x < y$ then $n^x = O(n^y)$ ✓ — θ, Ω X (Function Difference)

(d) $2^n \neq O(n^c)$ where c is const, $c > 1$ ✓
Exponential Polynomial.

$\sqrt{n} > \log n$

$\sqrt{\log n} > \log \log n$

$0 < x < y$ $\quad n^x \quad n^y$ $\quad x=2, y=3$

$n^2 < n^3$ ✓

$2^n \quad n^c \rightarrow n \log 2 > c \log n$

Ques 2: Check the foll. statements true / false ?

(a) $(n+a)^b = O(n^b)$ where a & b are +ive const. ✓
(b) $2^{n+5} = O(2^n)$ ✓
(c) $2^{2n} = O(2^n)$ ✗
(d) $f(n) = O((f(n))^2)$ ✗ < Depends on Function > Fails for Decreasing Function

$(n+a)^b$ $\equiv$ n^b

2^{n+5} $2^5 \cdot 2^n$

$2^n \cdot 2^5$

$(2^2)^n$ 2^n

$= 2^n \cdot 2^n$ 2^n

$2^n > 1$

$(n+1)^2$ n^2

n^2+n+1 $3n^2$

$n < O(n^2)$ ✓

n^2 $O(n^4)$ ✓

$\sqrt{n} > \sqrt{n^2-n}$

Function Diff

If Θ is possible $\Rightarrow$ O, ω not possible

$f(n) > 1 \longrightarrow$ Increasing Fnctn

$f(n) < 1 \longrightarrow$ Decreasing "

Ques 3: Check the foll. statements are true/false.

O, o ✓

(a) $n^5 \, 64^{\log_8 n} = \theta(n^{10})$ ✗

(b) $32^{\log_2 n} \; 64^{\log_8 n} = O(n^{10})$ ✓ (O or ✓)

(c) ~~log~~ $32^{\log_2 n} \; 64^{\log_2 n} = \Omega(n^3)$ ✓ (Ω ✓)

(d) $\frac{4^n}{2^n} = \theta(2^n)$ ✓ (O, Ω ✗)

$n^5 \; 64^{\log_8 n} \Rightarrow n^5 \cdot 8^{\log_8 n^2}$ — $n^5 \cdot n^5$ — n^5 — n^5

~~$\log_8 n^2$~~ ~~$\log_8 n^2$~~ ~~$\log_8 n^2$~~ ~~$\log x = \log n^2$~~ ~~$\log 8$~~

$32^{\log_2 n} \; 64^{\log_8 n}$ — n^{10}

$2^{\log_2 n^5} \cdot 8^{\log_8 n^2}$ — n^{10}

$n^5 \cdot n^2$ — n^{10}

n^7 — n^{10}

$2^{\log_2 n^5} \cdot 2^{\log_2 n^6}$ | n^3

$n^5 \cdot n^6$ | n^3

n^{11} | n^3

$\frac{4^n}{2^n}$ | 2^n

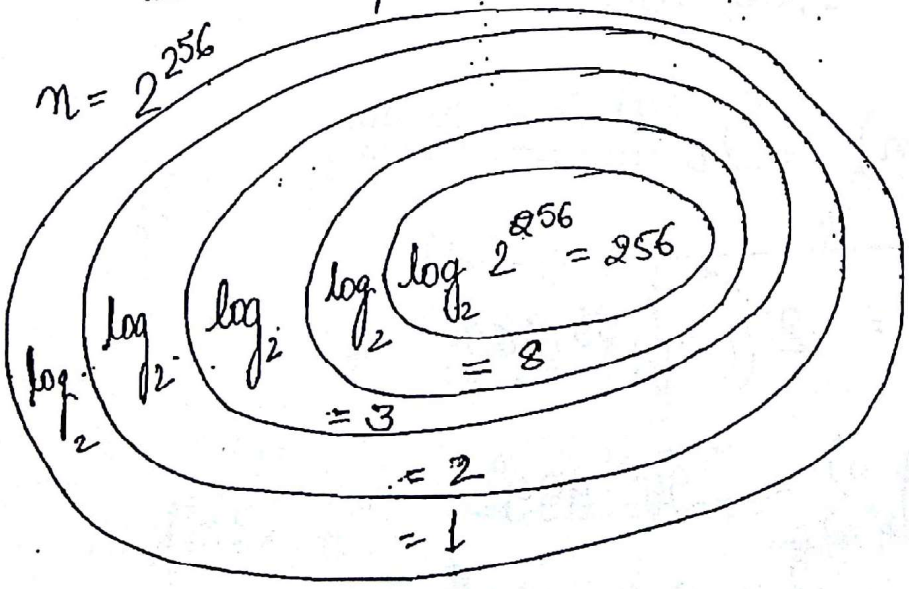
$\frac{2^{2n}}{2^n} = 2^n$ | 2^n

Note: $a^{\log_b n} = n^{\log_b a}$

Ques 4: $f(n) = \log^*(\log n)$

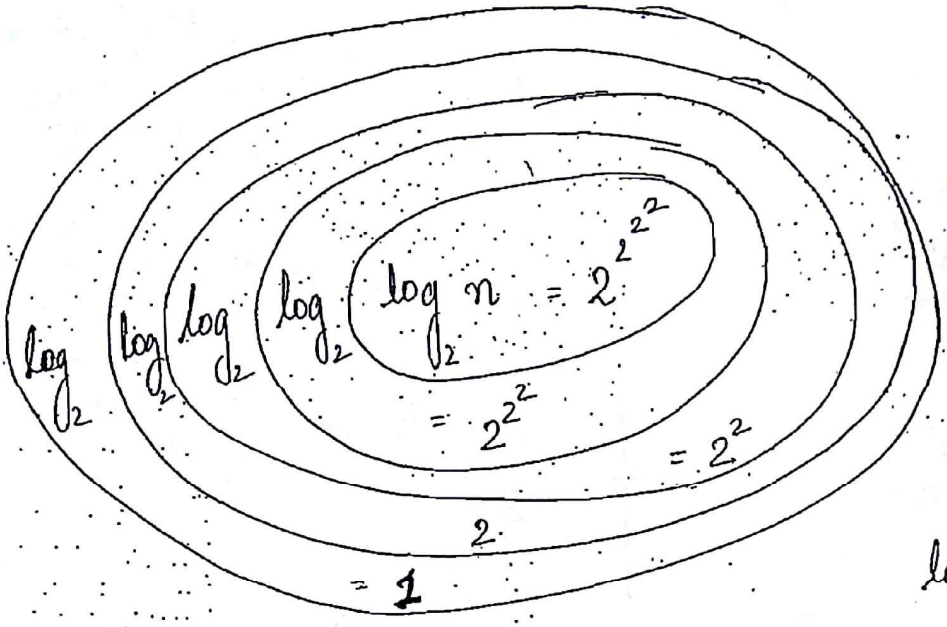
$g(n) = \log(\log^* n)$

Relation b/w these two?

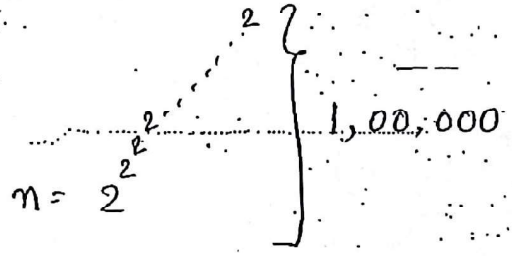


$\log^* n = 5$

$$n = 2^{2^{2^{2}}}$$



$\log^* n = 5$



$$\log_2^* n = 1,00,000$$

$$\log_2 (\log_2^* n) = 16$$

$$\log_2 n = 2^{2^{\cdots^{2}}} \Big\} 99,999$$

$$\log_2^* (\log_2 n) = 99,999$$

$$\log^* (\log^* n) = S2 \left(\log (\log^* n)\right)$$

$\Rightarrow \boxed{f(n) = \Omega(g(n))}$

Ques 5:

(a) $$f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^5 & n \geq 10000 \end{cases}$$

$$g(n) = \begin{cases} n^7, & 0 < n < 100 \\ n^4, & n \geq 100. \end{cases}$$

Relation b/w these two?

$0 < n < 100$

$f(n) = n^3$ $\qquad$ $g(n) = n^7$

$f(n) < g(n)$

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

$100 \leq n < 10000$

$f(n) = n^3$ $\qquad$ $g(n) = n^4$

$f(n) < g(n)$

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

$n \geq 10000$

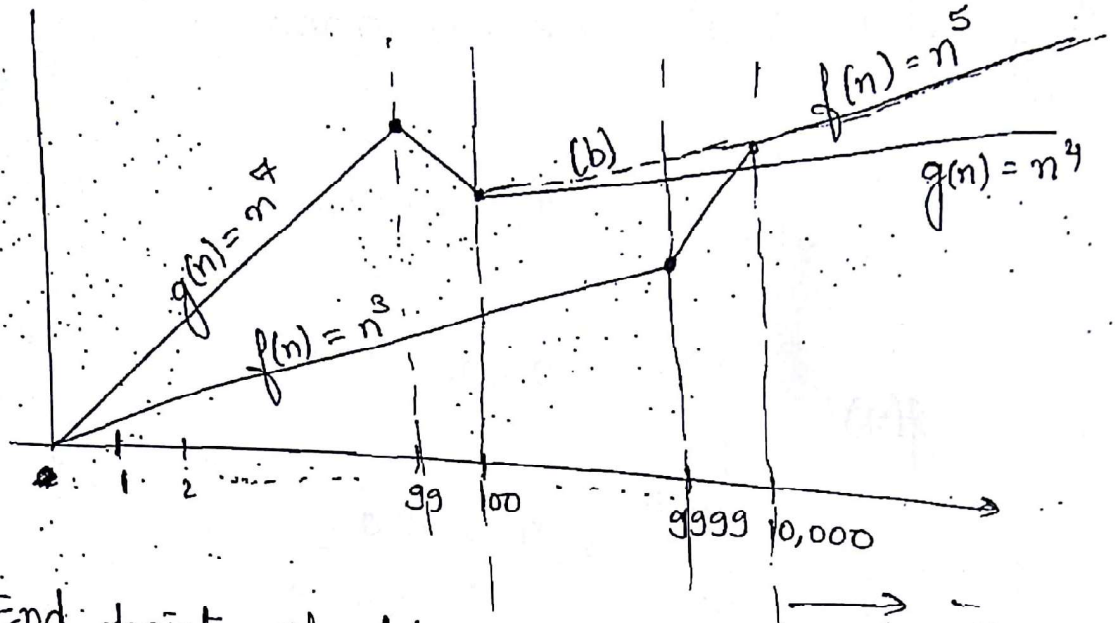
$f(n) = n^5$ $\qquad$ $g(n) = n^4$

$f(n) > g(n)$

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

Finally,

$\Rightarrow f(n) = \Omega(g(n))$, $n > 10{,}000$.



End point should be infinity.

So, $f(n) = \Omega(g(n))$, $n_0 = 10{,}000$

(b) $f(n)$ same

$$g(n) = \begin{cases} n^7 & , \quad 0 < n < 100 \\ n^5 & , \quad n \geq 100 \end{cases}.$$

$f(n) = \Theta(g(n))$ $\quad n_0 = 10000$

OR

$f(n) = O(g(n))$, $n_0 = 1$

$\downarrow$

$\leq$

Ques 6: $f(n) = n^{2+\sin n}$

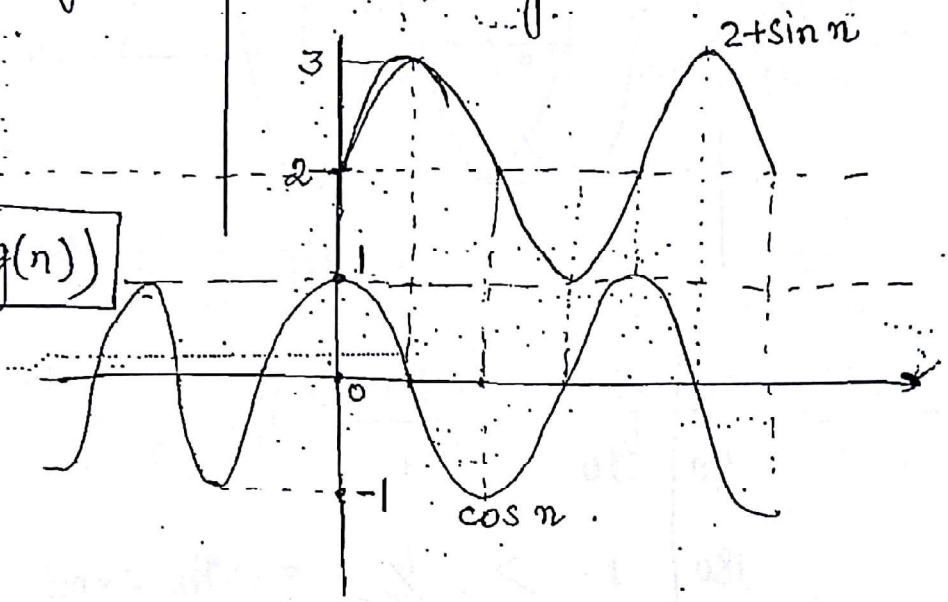
$g(n) = n^{\cos n}$.

Relation b/w them?

$n^{2+\sin n}$ $\quad$ $n^{\cos n}$

$(2+\sin n)\log n > \cos n \log n$

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

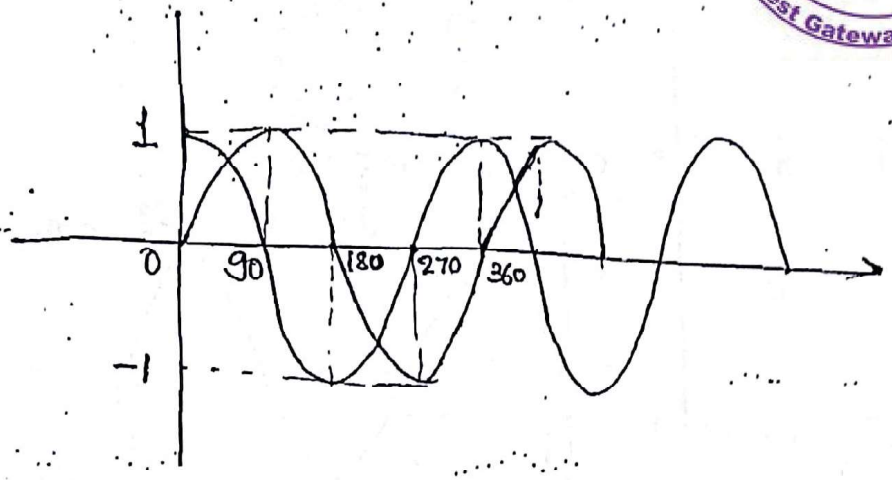


n	$f(n)$		$g(n)$
90	90^3	$>$	90^0
180	180^2	$>$	$\frac{1}{180}$
270	270	$>$	1
360	360^2	$>$	360

Ques 7: $f(n) = n^{\sin n}$

$g(n) = n^{\cos n}$

Relation b/w them?



n	$f(n)$		$g(n)$
90	90	>	1
180	1	>	1/180
270	1/270	<	1
360	1	<	360

The same pattern will repeat for next 360°.

So, there is no relation

Ques 8: Check the foll. statements are true/false?

(a) $\frac{1}{n} = O(1)$ ✓

(b) $\frac{1}{n} = \Theta(1)$ ✗

Good (c) If $f(n) = O(g(n))$

then

$2^{f(n)} = O(2^{g(n)})$ ✗ (Same is applicable for Ω)

(d) $1000 = \theta(1)$ ✓

$\frac{1}{n} \leq c_1 \cdot 1$

$\frac{1}{n}$ 1

Imp: n is no. of i/ps. So it cannot be fraction ✓

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

$f(n) \leq c_1 \cdot g(n)$

$2^{f(n)} \leq c_1 2^{g(n)}$ ✗

n+10 $\quad$ n-10

n $\quad$ n^2

2^n $\quad$ 2^{n^2} ✓

$f(n) = 2n$, $g(n) = n$

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

then

$2^{2n} \leq 2^n$ {which is false}

$2^n \cdot 2^n \leq c \cdot 2^n$ (✗)

$\downarrow$

2^n

Properties of Asymptotic Notation

① Reflexive property :-

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$
- $f(n) = \Theta(f(n))$

o, ω fails.

② Symmetric property :-

If $f(n) = O(g(n))$ then $g(n) = O(f(n))$ ✗

$n^2 = O(n^3) \implies n^3 = O(n^2)$ ✗

O, ω, Θ, Ω fails ✗

Θ pass ✓

③ Transitive property :-

If $f(n) = O(g(n))$ && $g(n) = O(h(n))$

then

$f(n) = O(h(n))$ ✓

O, Ω, Θ, o, ω pass.

④ If $f(n) = O(g(n))$

then

$h(n) \cdot f(n) = O(h(n) \cdot g(n))$

⑤ If $f(n) = O(g(n))$

&&

$d(n) = O(e(n))$

then

(i) $f(n) + d(n) = \max(g(n), e(n)) = O(g(n) + e(n))$

(ii) $f(n) \cdot d(n) = O(g(n) \cdot e(n))$

Ques 1: Let $f(n)$, $g(n)$ & $h(n)$ be 3 positive fnctns which are defined as follows:-

i) $f(n) = O(g(n))$ & $g(n) \neq O(f(n))$

ii) $g(n) = O(h(n))$ & $h(n) = O(g(n))$

then

T/F

a) $f(n) = \Omega(h(n))$ ✗

b) $f(n) \cdot h(n) = \Theta(g(n) \cdot h(n))$ ✗

c) $g(n) \cdot h(n) = O(g(n) \cdot g(n))$ ✓

d) $f(n) + h(n) = O(g(n))$ ✓

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

Ques 10 : If $T_1(n) = O(f(n))$ && $T_2(n) = O(f(n))$ then

If $T_1(n) = \Theta(f(n))$ && $T_2(n) = \Theta(f(n))$ all a, b, c, d (True)

T/F

(a) $T_1(n) + T_2(n) = O(f(n))$ ✓

(b) $T_1(n) = O(T_2(n))$ ... ×

(c) $T_1(n) = \Omega(T_2(n))$ ×

(d) $T_1(n) = \Theta(T_2(n))$ ×

} No relation b/w T_1 & T_2

Divide & Conquer

BASICS

1. Recursion
2. Recurrence relation
3. Recurrence relation solving.

Recursion ① function calling itself.

② Recursion is nothing but solving big problems in terms of smaller ones.

Ex: $f(6) = 6 * f(5)$ → 720

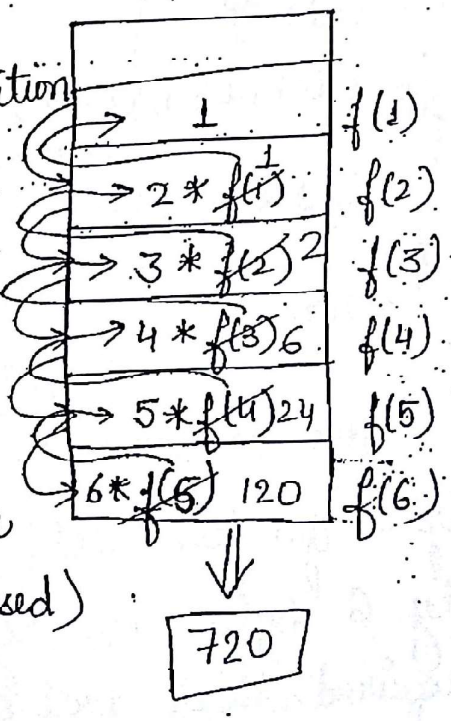
$5 * f(4)$ → 120

$4 * f(3)$ → 24

$3 * f(2)$ → 6

$2 * f(1)$ → 2

$f(1)$ → 1



(3) Every recursive program should have termination condition otherwise program will go to infinite loop * error message of Stack Overflow

(4) To execute the recursive program, stack data structure is used. (Queue cannot be used)
↓
FIFO

(5) In recursion, from 1 function call to another, parameter value will change but not no. of parameters and names of parameters.

(6) For every recursive program, equivalent non-recursive program exists / is possible.

```
main()
{
    int n=6;
    printf("%d", f(n));
}
```

n: 6 (1000)

6 → Non-recursive

6 → Recursive

Non-recursive

```
f(int n)
{
    int i,s;
    s=1;
    for(i=1;i<=n;i++)
    {
        s=s*i;
    }
    return s;
}
```

n: 6 (2000) | i: 1 | s: 1

int – 2 bytes

Only 1 function call, only 6 bytes memory required which will be updated on each iteration

Recursive

```
f(int n)
{
    if(n==1)
        return 1;
    else
    {
        b=f(n-1);
        c=n*b;
    }
    return c;
}
```

n: 6 | b: | c:

For every function call, 6 bytes of memory allocated for n, b, c ⇒ 36 bytes of memory

(4) Comparing recursion & non-recursion, recursion will take more stack space. (but path will take stack space) because of more fnctn calls.

Note:-

(8) Recursion & Non-recursive program does not affect time complexity. Logic affects time complexity.

↳ Recursive programs are beneficial to programmer and non-recursive programs are beneficial to computer.

6/12/16

Ex1

```
fact (n)
{
  if (n ≤ 1)
    return 1;
  else
    return n * fact(n-1);
}
```

$$fact(n) = \begin{cases} 1, & n \leq 1 \\ n * fact(n-1), & n > 1 \end{cases}$$

recurrence relation (for value)

For a recursive program, more than 1 recurrence relations exists (for time, value etc)

Ex2 Write a recursive program & recurrence relation to find power of (a,b), a, b > 0.
(Best & Worst & Average case same)

```
power (a, b)
{
  if (b == 1)
    return a;
  else if (a == 1)
    return 1;
  else
    return a * power (a, b-1);
}
```

pow(5,3) = 125
⇓ 25
5 * pow(5,2)
⇓
5 * pow(5,1)
⇓
5

Recurrence Relation :

$$power(a,b) = \begin{cases} a & , b = 1 \\ 1 & , a = 1 \\ a * power(a, b-1) & , b > 1 \end{cases}$$

Time complexity = $O(b) = O(n)$

Ex 3 Write a recursive program & recurrence relation to find gcd(m, n) where m, n > 0 & integers

```
gcd (m, n)
{
  if(m == 0)
    return n;
  if (n == 0)
    return m;
  if (m % n == 0)
    return n;
  if (n % m == 0)
    return m;
  else
  {
if-else not reqd, it will automatically reverse if after 1 fn call.
    return gcd (n%m, m);
  }
}
```

m	n
24	32

greatest common divisor

2, 4, 8, ... 12

12, 16 → 6, 8 → 3, 4

2*2*1 ... 2*1 ... 1

(m % j == 0 && n % j == 0)

Termination Condition

GCD(20, 5) = 5

GCD(13, 17) = 1

GCD(250, 50) = 50

GCD(10, 0) = 10

GCD(0, 10) = 10

GCD(0, 0) = ∞

Procedure

13) 17 (1
13
4) 13 (3
12
1) 4 (4
4
X

GCD

(24, 32) ⇒ 24) 32 (1
24
8

(8, 24) 8) 24 (3
24
0

(0, 8) ✓

Recursive Program

```
gcd(m,n)
{
  if (m==0 && n==0)
      return (inf); // printf("Error");
  if (m==0)
      return n;
  if (n==0)
      return m;
  else
      return gcd(n%m, m);
}
```

Recursence Relation :-

$$gcd(m,n) = \begin{cases} \infty, & m=0 \&\& n=0 \\ n, & m=0 \&\& n!=0 \\ m, & n=0 \&\& m!=0 \\ gcd(m\%n, n), & \text{otherwise} \end{cases}$$

Time complexity = ~~log n~~ $O(\log_2(\max(m,n)))$

due to cont. division

Base can be anything as only constant diff. (division is not by a const number)

GCD (23, 53)

$$23\overline{)53}\,(2$$
$$\underline{46}$$
$$7$$

GCD (7, 23)

$$7\overline{)23}\,(3$$
$$\underline{21}$$
$$2$$

GCD (2, 7)

$$2\overline{)7}\,(3$$
$$\underline{6}$$
$$1$$

GCD (1, 2)

$$1\overline{)2}\,(2$$
$$\underline{2}$$
$$\times 0$$

GCD (0, 1) = 1

Best Case for gcd :-

Nos. m & n are multiple of each other

$O(1)$ Ex: gcd (400, 20000)

Worst Case for gcd :-

Nos. m & n are prime.

$O(\log_2 n)$ if $m < n$

Average case for gcd :-

b/w best & worst case.

Best Case, worst & average case are determined by else condition of recursion.

Ex4 Write a recursive program & recurrence relation to find nth fibonacci no.

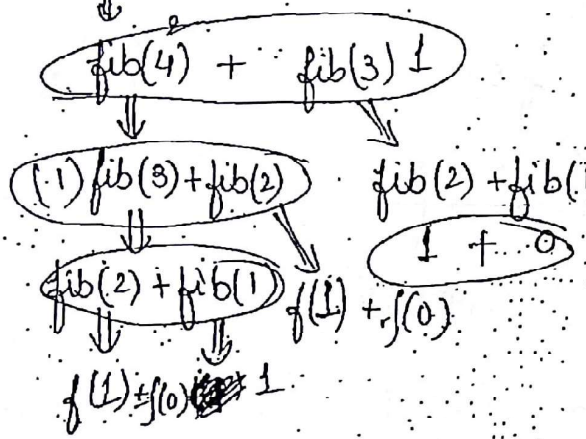
0 1 2 3 4 5 6 7

0, 1, 1, 2, 3, 5, 8, 13, - - - -

```
fibonacci (n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

if (n == 0 || n == 1)
    return n;

Ex: fib(5)



Recurrence Relation :

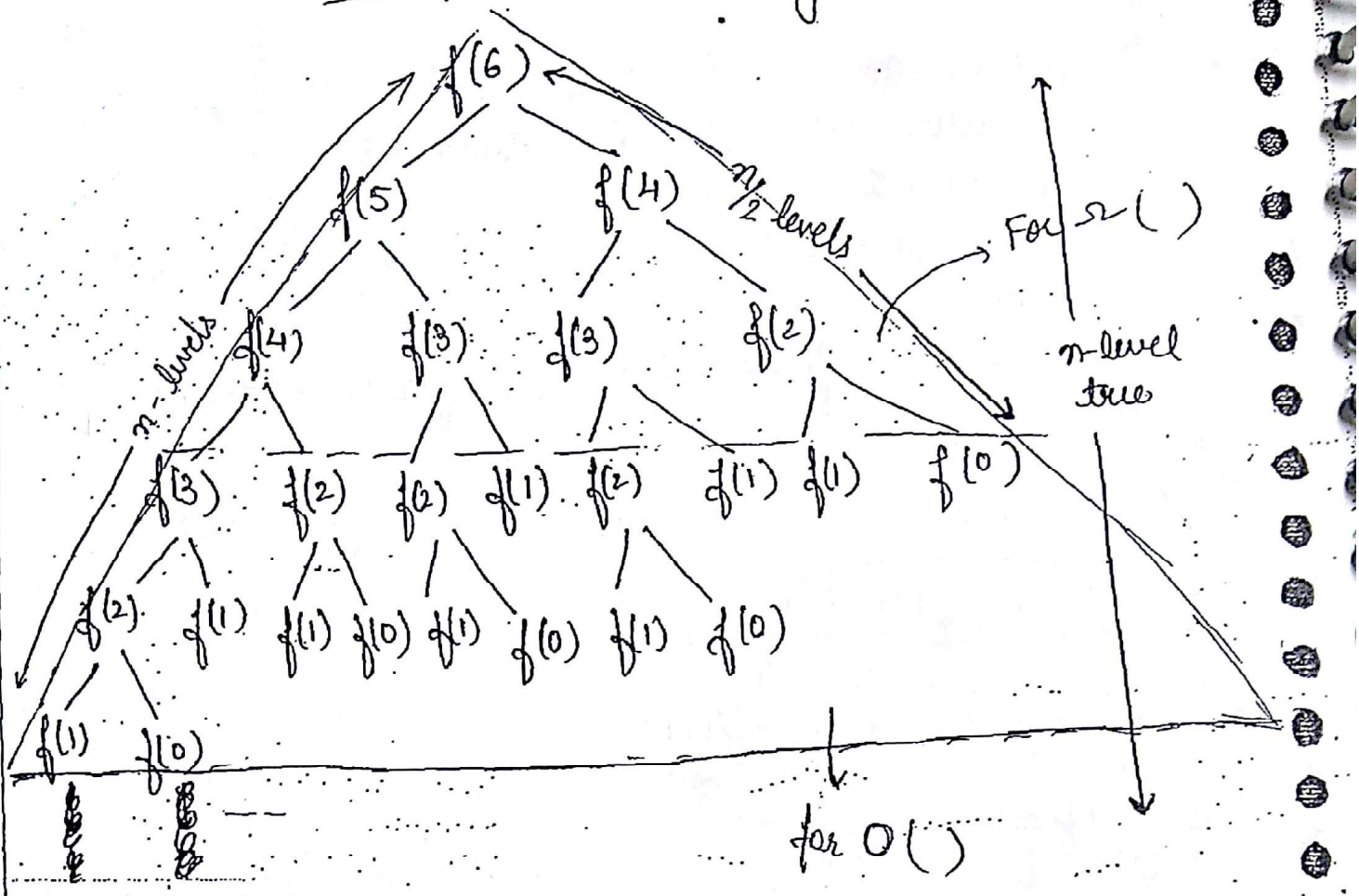
$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & \text{otherwise} \end{cases}$$

~~Time Complexity = O(1) Best Case~~

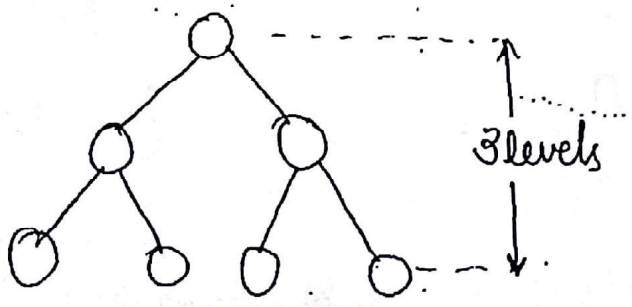
~~O(n)~~ ~~worst Case~~

~~Best~~

For fibonacci (Binary Tree)



Complete Binary Tree



No. of Nodes = $2^3 - 1$

$= 2^n - 1$ (max)

Min. nodes = n. (Unary Tree)

fib(n) : n-level Binary Tree but not complete

⇓

n-level Complete Binary Tree (Upper Bound)

⇓

$(2^n - 1)$ nodes.

⇓

2^n - nodes (ignore -1)

⇓

$O(2^n) \Leftarrow 2^n$ function calls

$T(fib(n)) = O(2^n)$ {Ulppu Bound} ⇒ Not worst case (more than actual ans)

$T(fib(n)) = \Omega(2^{n/2})$ {Lower Bound} ⇒ Not best case (less than actual ans)

$$2^{n/2} \leq T(fib(n)) \leq 2^n$$

If in else,

- return $\underbrace{fib(n-1)}_{max} + fib(n-2) + fib(n-3)$

will give n-level Ternary tree

$\Rightarrow T(f(n)) = O(3^n)$

$T(f(n)) = \Omega(3^{n/3})$

- return $\underbrace{fib(n-1)}_{n\,(max)} \overset{\text{can be any operator}}{(+)} \underbrace{fib(n/2)}_{\log n\,(min)}$ Binary Tree

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

$T(f(n)) = \Omega(2^{\log n})$

- return $\underbrace{fib(n/3)}_{\log_3 n\,(min)} + \underbrace{fib(n/2)}_{\log_2 n\,(max)}$ Binary Tree

$T(f(n)) = O(2^{\log_2 n}) = O(n)$

$T(f(n)) = \Omega(2^{\log_3 n})$

- return $\underbrace{fib(n-1)}_{n} + \underbrace{fib(n-1)}_{n}$

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

$T(f(n)) = \Omega(2^n)$

$T(f(n)) = \Theta(2^n)$

Exact

No Gap in Binary Tree.

In fibonacci function, there is no best case & worst case. (always left side will be n-level tree & right side will be $n/2$ - level tree)

Recurrence Relation Solving

1. Substitution method
2. recursive tree method
3. Master Theorem

SUBSTITUTION METHOD

Note: Substituting the given function repeatedly until given function is eliminated.

Ex 1.

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

→ if stopping condtn not given, take any constant

~~Find~~ Solve the recurrence relation?

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$$\vdots \quad (n-1) \text{ times (to get } T(1) \text{ which is termination condtn)}$$

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

$$T(1) + 2 + 3 + 4 + \ldots + (n-1) + n$$

$$\Downarrow$$

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

$$T(n) = \frac{n(n+1)}{2} = O(n^2)$$
$$= \Omega(n^2)$$
$$= \theta(n^2)$$

Ex2 $T(n) = \begin{cases} 1, & n = 1 \\ T(n-1) + \log(n), & \text{if } n > 1 \end{cases}$

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

$\Downarrow$

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

$\Downarrow$

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

$\vdots$

$(n-1)$ times

$\vdots$

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

$T(n) = 1 + \log 2 + \log 3 + \log 4 + \ldots + \log(n-1) + \log n$

$= 1 + \log(2.3.4\ldots(n-1).n)$

$= 1 + \log(n!)$

$2^n < n! < n^n$

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

$= \Omega(\log 2^n) = \underline{\Omega(n)}$

Ex3: $T(n) = \begin{cases} 1 & , n = 0 \\ T(n-2) + n^2 & , n > 0 \end{cases}$

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

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

$\Downarrow$

$T(n-4) + (n-2)^2 + n^2$

$\Downarrow$

$T(n-6) + (n-4)^2 + (n-2)^2 + n^2$

$\vdots$ (n times)

$\Downarrow$

$n-(n$

$T(n-n) + (n-(n-2))^2 + \dots + (n-2)^2 + n^2$

$T(0) + 2^2 + 4^2 + \dots + (n-2)^2 + n^2$

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

$= 1 + (2.1)^2 + (2.2)^2 + (2.3)^2 + \dots + \left(2.\frac{n-2}{2}\right)^2 + \left(2.\frac{n}{2}\right)^2$

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

$= 1 + 4 \cdot \frac{1}{6} \cdot \frac{n}{2}\left(\frac{n}{2}+1\right)(n+1)$

$= O(n^3) = \Omega(n^3) = \Theta(n^3)$

(Upper Bound) (Lower Bound)

No Best & worst Case

Ex 4: $T(n) = \begin{cases} 1 & , n = 1 \\ 2T(n/2) + n & , n > 1 \end{cases}$

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

$\Downarrow$

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

$\Downarrow$

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

$\vdots$ $\log n$ times $= 2^3 T\left(\frac{n}{8}\right) + 2^2 \cdot \frac{n}{2^2} + n + n$

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

$2^k = n$

$k = \log n$

~~$2T(1) + 2$~~

$= 2^{\log n} T(1) + \underbrace{n + n + n + \ldots + n}_{\log n \text{ times}}$

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

$= n + n \log n = O(n \log n)$ $\rightarrow c = 2$

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

$\rightarrow c = 1$

Ex5. $T(n) = \begin{cases} 1, & n = 1 \\ T(n/2) + c, & n > 1 \end{cases}$ ← same as

Variations :-

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

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

$T(n) = T\left(\frac{n}{2} - 10\right) + c$

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

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

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

$2^k = n$

$\log K = \log n$

$K = \log n$ times

$= \{ T(1) + \underbrace{c + c + c + \ldots + c}_{\log n \text{ times}} \}$

$= 1 + c \cdot \log n$

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

Ex6. $T(n) = \begin{cases} 1, & n = 1 \\ T(n/2) + n, & n > 1 \end{cases}$

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

$$\Downarrow$$

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

$$\Downarrow$$

$$T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$

$\log n$ times

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

$$k = \log n$$

$$T(1) + \frac{n}{2^{\log_2 n - 1}} + \cdots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + \frac{n}{2^0} \qquad \downarrow \frac{n}{n} \times (i)$$

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

~~$= 1 + n.(\frac{1}{n} - 1) = \Theta(n^2)$~~

~~$= \Omega(n^2)$~~

~~$= \Theta(n^2)$~~

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

$\frac{1}{n}$ can be ignored as $\frac{1}{n} \to 0$ when $n \to \infty$

$= O(n)$

$= \Omega(n)$

$= \Theta(n)$

Decreasing G.P. series : $|r| < 1$

$$\frac{1}{2^0} + \left(\frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{\log_2 n}}\right) \quad n \to \infty.$$

(1) + (1) = 2

Ex 7. $T(n) = \begin{cases} 2 & , n = 2 \\ 7T\left(\frac{n}{2}\right) + n^2 & , n > 2 \end{cases}$

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

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

$7\left(\frac{1}{2}\right)^2 = \frac{7}{4}$

$$= 7^2 \cdot \left[7T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + 7\cdot\left(\frac{n}{2}\right)^2 + n^2$$

$7\left(\frac{1}{2}\right)^4 = \frac{7}{16}$

$$= 7^3 \cdot T\left(\frac{n}{2^3}\right) + 7^2 \cdot \left(\frac{n}{4}\right)^2 + 7\left(\frac{n}{2}\right)^2 + n^2$$

$\frac{n}{2^k} = 2$; $\log n - 1$

$n = 2^{k+1}$

$k+1 = \log n$

$$= 7^{\log_2 n - 1} \cdot T(2) + 7^{\log_2 n - 2} \cdot \left(\frac{n}{2^{\log_2 n - 2}}\right)^2 + \cdots + 7\left(\frac{n}{2}\right)^2 + n^2$$

$\frac{2}{2^{\log_2 n - 1}}$

$$= 2\cdot 7^{\log n - 1} + n^2\left(1 + 7\left(\frac{1}{2}\right)^2 + 7^2\left(\frac{1}{2}\right)^4 + \cdots + 7^{\log_2 n - 2}\left(\frac{1}{2^{\log_2 n}}\right)\right)$$

$$= 2\cdot 7^{\log n - 1} + n^2\left[1\cdot\frac{\left(\frac{7}{4}\right)^{\log_2 n - 1} - 1}{\frac{7}{4} - 1}\right]$$

$$= 2.7^{\log_2 n - 1} + n^2 \cdot \frac{4}{3}\left[\left(\frac{7}{4}\right)^{\log_2 n - 1} - 1\right]$$

$$= 7^{\log_2 n} + n^2 \cdot \left[\left(\frac{7}{4}\right)^{\log n}\right] \quad \{\text{After eliminating constants}\}$$

$$= 7^{\log_2 n} + n^2 \frac{7^{\log n}}{2^{\log_2 n^2}} = 7^{\log_2 n} + \frac{n^2 \, 7^{\log_2 n}}{n^2}$$

$$= 2.7^{\log_2 n} = O(7^{\log_2 n})$$

$$= O(2^{\log n \cdot \log_2 7}) = O\left(n^{\log_2 7}\right)$$

$$= O(n^{2.81})$$

Imp ×8

$$T(n) = \begin{cases} 2 & ; \quad n = 2 \\ \sqrt{n}\,T(\sqrt{n}) + n & , \quad n > 2 \end{cases}$$

$(n)^{1/2} + (n^{1/2})$

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

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

$n^{\frac{1}{2}} \cdot n^{\frac{1}{2}}$

$n^{\frac{1+1}{2}}$

$\frac{1}{8} + \frac{1}{4}$

$\frac{1+2}{8}$

$$= n^{1/2^3} \cdot T(n^{1/4}) + n + n$$

$\frac{1}{2} \cdot \frac{1}{2}$

$\sqrt{n} \cdot \sqrt{n} = n$

$$= n^{1/2^3}\left(n^{1/8} T(n^{1/8}) + n^{1/4}\right) + n + n$$

$n^{\frac{1}{2}} \cdot n^{\frac{1}{4}}$

$\frac{1}{2} + \frac{1}{4}$

$\frac{2+1}{4} = 3/4$

$$= T(n^{1/8}) + n$$

$\frac{3}{4} \times \frac{1}{8}$ $\frac{6+1}{8}$ $7/8$

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

$\frac{1}{2}+\frac{1}{4}$

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

$$\Downarrow$$

$\frac{3}{2^2}+\frac{1}{2^2}$

$$n^{3/2^2}\left[n^{1/2^3} T\left(n^{1/2^3}\right) + n^{1/2^2}\right] + n + n$$

$\frac{6+1}{2^3}$

$$= n^{7/2^3} T\left(n^{1/2^3}\right) + n + n + n$$

$$\vdots$$

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

$$\vdots$$

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

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

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

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

$\Rightarrow \log_2 n = k$

$k = \log \log n$

$2^{\log_2 n}$

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

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

$$= 2n + \frac{n \log n}{2} = O(n \log n)$$

$$= \frac{n}{n^{2\log_2 \log n}} T(2) + n \cdot \log \log n \qquad 2^{\log_2 (\log n)}$$

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

$$= \frac{n}{2^{\log_2 \log n}} \cdot 2 + n \cdot \log_2 \log_2 n$$

$$= n + n \log \log n = O(n \log_2 \log_2 n)$$

Ex 9: used in $\rightarrow$ $\boxed{T(n) = 2T(n-1) + n}$

AGP

$$T(n) = 1 \cdot \left(\tfrac{1}{2}\right)^1 + 2\left(\tfrac{1}{2}\right)^2 + 3\left(\tfrac{1}{2}\right)^3 + \cdots + (n-1)\left(\tfrac{1}{2}\right)^{n-1} + n\left(\tfrac{1}{2}\right)^n$$

$$\Rightarrow T(n) = 1 \cdot \left(\tfrac{1}{2}\right)^1 + 2\left(\tfrac{1}{2}\right)^2 + 3\left(\tfrac{1}{2}\right)^3 + \cdots (n-1)\left(\tfrac{1}{2}\right)^{n-1} + n\left(\tfrac{1}{2}\right)^n$$

$$\tfrac{1}{2}T(n) = \quad 1 \cdot \left(\tfrac{1}{2}\right)^2 + 2\left(\tfrac{1}{2}\right)^3 + \cdots + (n-2)\left(\tfrac{1}{2}\right)^{n-1} + (n-1)\left(\tfrac{1}{2}\right)^n + n\left(\tfrac{1}{2}\right)^{n+1}$$

$$\left(1 - \tfrac{1}{2}\right)T(n) = \left[\left(\tfrac{1}{2}\right) + \left(\tfrac{1}{2}\right)^2 + \left(\tfrac{1}{2}\right)^3 + \cdots + \left(\tfrac{1}{2}\right)^{n-1} + \left(\tfrac{1}{2}\right)^n\right] - n\left(\tfrac{1}{2}\right)^{n+1}$$

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

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

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

$$= \frac{(n+2)}{2^{n+1}} = 2\left[1 - n \cdot \underbrace{\left(\frac{1}{2^{n+1}}\right)}_{\text{decreasing}}\right]$$

Divide & Conquer (DAC)

Divide :- the given problem into some sub-problems

Conquer the subproblems by calling recursively until we will get subproblem solution.

Combine the subproblem solution to get final problem solution.

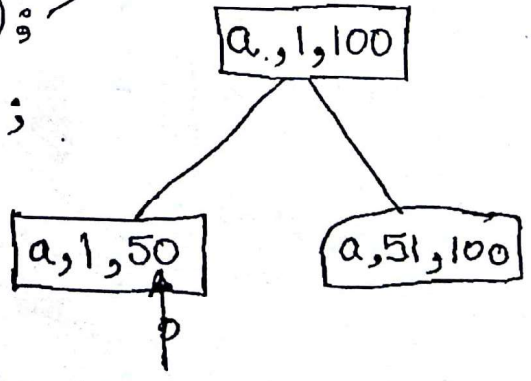
Quick Sort, Binary Search ⇒ Partially divide & Conquer
(No combine.)

Ex: Sorting of array :-

Divide and Conquer Abstract Algorithm

DAC (a, i, j) $\xrightarrow{\text{For } n}$ T(n)

```
{
    if (small (a,i,j))  ⟹ O(1)
        return (solution (a, i, j));
    else
        p = divide (a, i, j);   → O(f1(n))
        L = DAC (a, i, p);      → T(n/2)
        R = DAC (a, p, j);      → T(n/2)
        C = Combine (L, R);     ← O(f2(n))
        return C;
}
```



Code of small(), solution(), divide() & combine() is dependent on programmer and varies problem by problem.

- If some problem can be solved using Divide & Conquer algorithm, it should be possible to define the above 4 functions.

Time Complexity of any problem solved using DAC :-

If $T(n)$ is the time taken to solve the entire DAC algorithm for the problem for n elements.

$T(n/2) \longrightarrow$ for $n/2$ elements

Assume 2 subproblems with size $n/2$ each.

$$T(n) = \begin{cases} O(1) & \text{, for small problem} \\ O(f_1(n)) + 2T(n/2) + O(f_2(n)) & \text{, for big problem} \end{cases}$$

$$= 2T(n/2) + \underbrace{O(f_1(n)) + O(f_2(n))}$$

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

- 2 : no. of subproblems
- $T(n/2)$: Time taken to solve a subproblem; $n/2$: size of subproblem
- $O(f(n))$: Time taken in divide & combine

$$= aT(n/b) + O(f(n)) \quad \{\text{In general}\}$$

where $a \geq 1$, $b > 1$; $f(n)$: Time Taken

For Quick Sort $T(n) = 2T(n/2) + n = O(n \log n)$

no. of subproblems; size of subproblem

Binary Search $T(n) = T(n/2) + c$

1 subproblem; size of subproblem.

Matrix Multiplication $T(n) = 8T(n/2) + n^2$

Time for divide & combine

APPLICATIONS of Divide & Conquer

1) Finding max. & min. element
2) Power of an element
3) Binary Search
4) Merge Sort
5) Quick Sort
6) Selection procedure
7) Strassen's Matrix multiplication

1) Finding Max & Min

i/p: array of n elements.

o/p: max & min element

Ex: A [25 (1), 79 (2), 88 (3), 16 (4), 32 (5), 11 (6), 17 (7)].

~~max = 25 79~~

~~min = 25~~

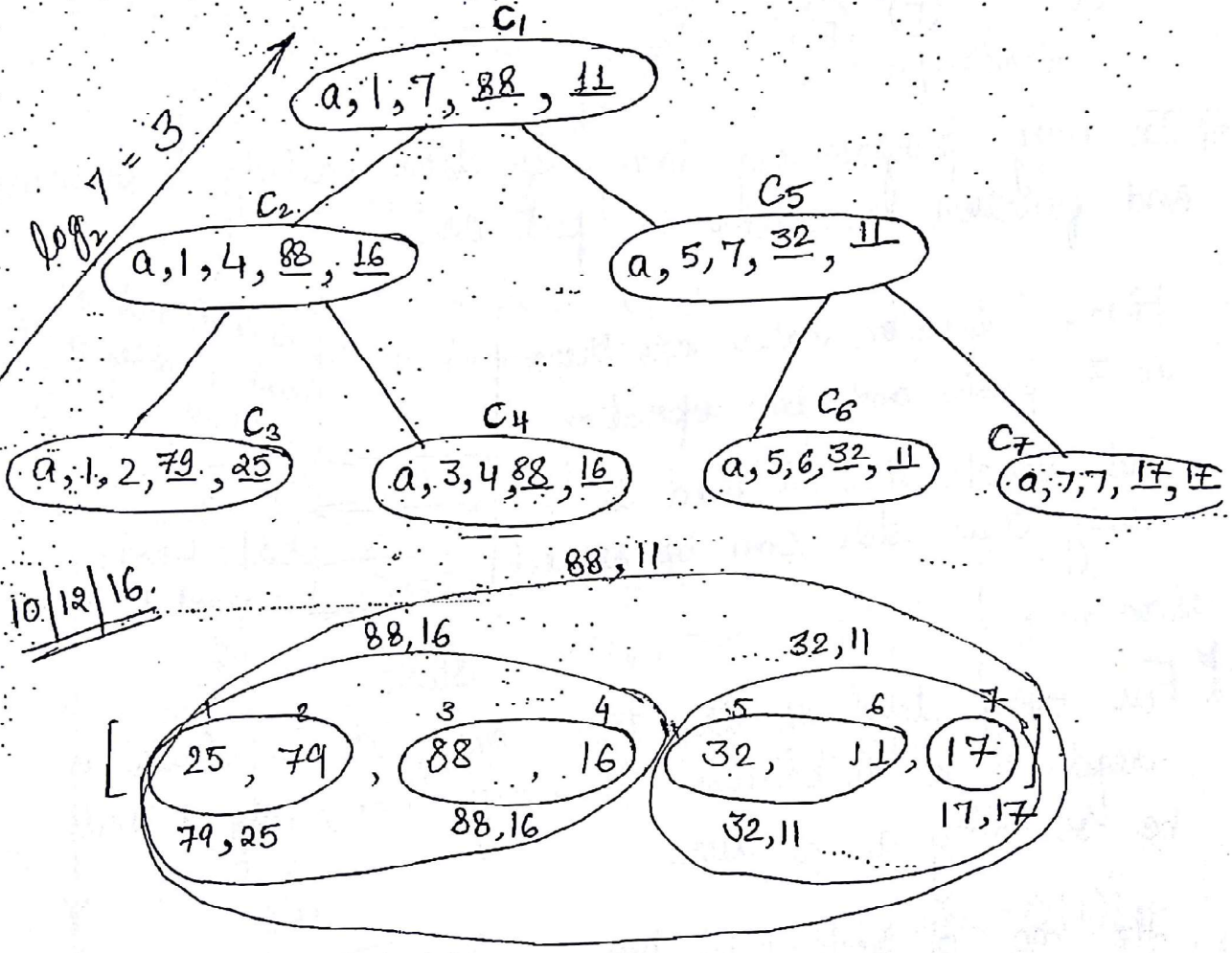
Define small problem,

small

⇓

1 element ⇒ 0 comparisons

2 elements ⇒ 1 comparison



10 | 12 | 16.

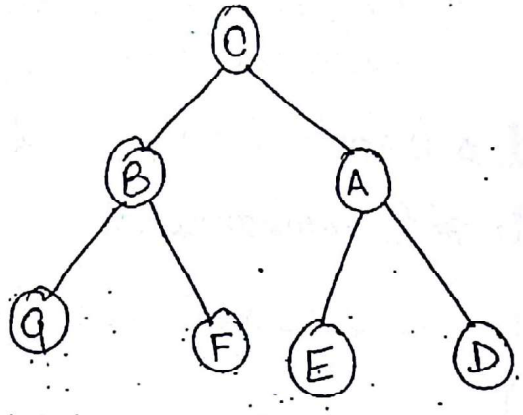
Every node in the tree is a function call.

Preorder :- Root - Left - Right

Inorder :- Left - Root - Right

Postorder :- Left - Right - Root.

} on the basis of position of root



Pre-order :

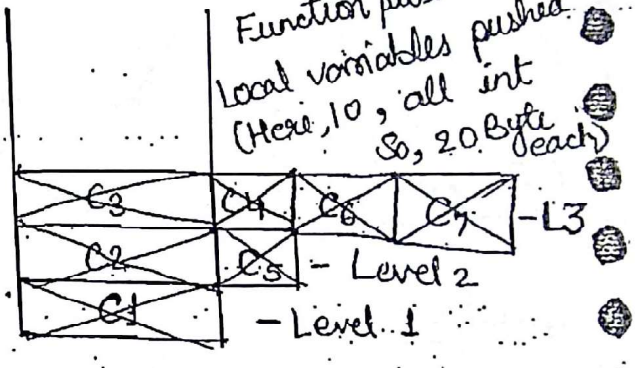
CBGFAED.

Post order :-

GFBEDAC

In every programming lang, function calling is preorder and function execution is post-order.

Here, 7 function calls are there so 7 push and pop operations. But stack size utilized is only 3 as slots can be reused.



Stack

* For every level of the tree, one slot in stack is reqd. It is so because at every level, only 1 will be running at a time.

The no. of levels in tree = $\log_2 n$ where n is the no. of records in the array = No. of slots in stack space.

Space complexity = total space utilized in running the program

$$= \underbrace{\log n}_{\text{Stack space}} + \underbrace{n}_{\text{input size}} = O(n)$$

Return , break & exit :-

Return → out of fnctn

break → out of Block

exit → out of entire program

Program Code :-

DAC minmax (a, i, j) → T(n) → T(n) comparisons

{

small

1-element

if (i == j)

{ max = min = a[i]; } 0-comparisons

return (max, min);

}

2-element

if (i == (j-1))

{

if (a[i] > a[j])

max = a[i]; min = a[j]; } 1 comparison

else

max = a[j]; min = a[i];

return (max, min);

}

else

{

Divide {O(1)}

mid = ⌊(i+j)/2⌋ → 0 comp.

Conquer

(max_1, min_1) = DAC minmax (a, i, mid); → $T(n/2)$

(max_2, min_2) = DAC minmax (a, mid+1, j); → $T(n/2)$

if (max1 > max2)

max = max1;

else

max = max2;

$T(n/2)$ comp.

```
if (min1 < min2)
    min = min1;
else
    min = min2;
return (max, min);
}
```

Combine {O(1)}.
2 comparisons.

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

Let $T(n)$ = time complexity of ~~this~~ above algorithm for n elements.

Recurrence Relation =

$$T(n) = \begin{cases} O(1) & , \ n = 1 \text{ or } n = 2 \\ 2T\left(\frac{n}{2}\right) + O(1) + O(1) & , \ n > 2 \end{cases}$$

$2T\left(\frac{n}{2}\right)$ ↓ Conquer, $O(1)$ ↓ Divide, $O(1)$ ↓ Combine

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

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

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

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

$$\left(\text{Keep stopping at } n=2\right) \frac{n}{2^k} = 1$$

$$\Rightarrow k = \log_2 n$$

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

$$= n \cdot T(1) + 2^{\log n - 1} \cdot c + \ldots + 2^2 c + 2^1 c + 2^0 c$$

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

$$= n + c(n-1) = O(n) = \Omega(n) = \Theta(n)$$

// Time Complexity

Let $T(n)$ = No. of comparisons for the above algorithm on 'n' elements.

Recurrence Relation =

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 0 + 2T(n/2) + 2, & n>2 \end{cases}$$

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

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

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

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

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

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

$$k = \log_2 n - 1$$

Stack Space

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

$$= \frac{n}{2} T(2) + 2^{\log_2 n - 1} + \ldots + 2^3 + 2^2 + 2$$

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

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

$$= \frac{3n}{2} - 2$$

{ Comparisons ⇓ Exact no. reqd. }

- Better algorithm than this not available for max & min.
- Best Case / Worst Case same here (as without comparison we cannot know if element is max/min).

2) Power of an element

i/p : 2 +ive integers a & n.
where $a \geq 2$, $n \geq 1$

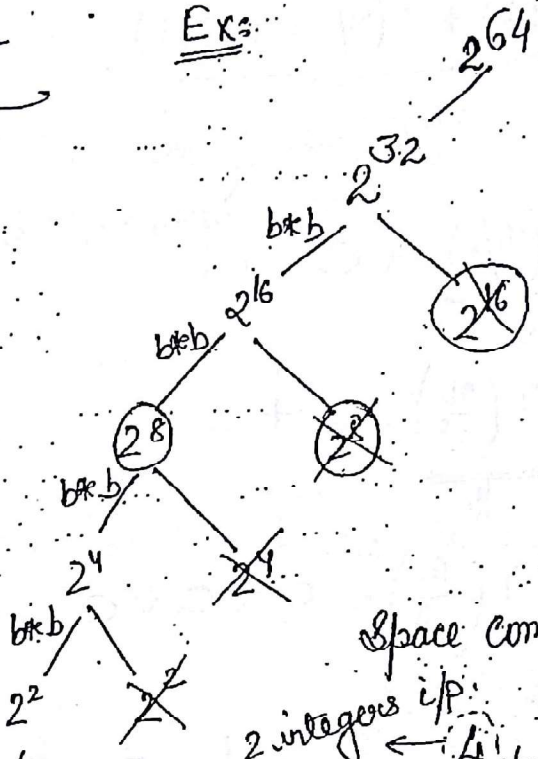
o/p : Find a^n

$a^n = \underbrace{a^{n/2} \cdot a^{n/2}}_{\text{Divide}}$ (b . b)

b

Small:
$n = 1$
return a

Ex:



Space complexity :=
2 integers i/p ← 4 + log n
= O(log n)

Algorithm :-

DACPower (a, n) ⟹ T(n)
{
if (n == 1) → 0 mult.
return a;
else
{
For n is odd → if (n%2 == 0)
return power;
else
return a * power;

mid = n/2 ; → 0 mult. Divide, O(1)
b = DACPower (a, mid) ; → T(n/2), Conquer
power = b * b ; → T(n/2) mult.
return power; → O(1), 1 mult. combine
}
}

Let $T(n)$ = time complexity of above algorithm to find a^n

$$T(n) = \begin{cases} O(1) & , n = 1 \\ \underbrace{O(1) + T(n/2) + O(1)}_{T(n/2) + c} & , n > 1 \end{cases}$$

$c \rightarrow$ Divide time + Combine Time

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

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

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

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n \text{ (Stack Space)}$$

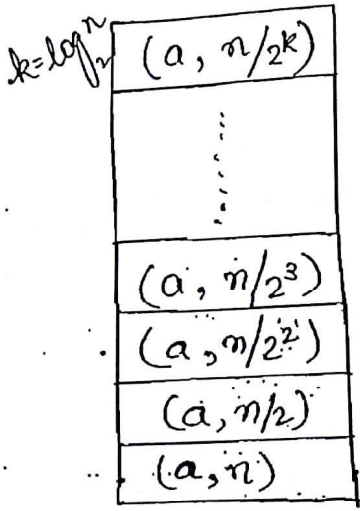
$$= T\left(\frac{n}{2^k}\right) + k \cdot c$$

$$= \underbrace{T(1)}_{= O(1)} + \underbrace{c \cdot \log_2 n}_{\Downarrow} = O(\log_2 n) \begin{cases} \text{Best/Worst/} \\ \text{Avg Case} \end{cases}$$

$= O(1)$ which is Small problem Cost

division & combine cost at all levels ($\log_2 n$)

$= \Theta(\log_2 n)$



Total space = i/p + extra

$= 2(2) + 10 \times \log_2 n$

$= 4 + 10 \log_2 n$

Let $T(n)$ = No. of multiplications

Recurrence Relation

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

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

$\Downarrow$

$T(n/4) + 1 + 1$

$\Downarrow$

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

$\vdots$

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

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

$= 0 + \log_2 n = \log_2 n$

3) Binary Search

Linear Search:

i/p: an array & an element x

o/p: position of x

Best Case	Worst Case	Avg Case
$O(1)$	$O(n)$	$\frac{(n+1)}{2} = O(n)$

For all elements:
$1+2+3+\dots n = \frac{n(n+1)}{2}$

$= \frac{n+1}{2}$

No stack reqd. so better in comparison to Binary Search in terms of space.

Binary Search: (There is no combine, so, Binary Search is partial appl^n of Divide & Conquer)

i/p: a **sorted** array of n elements & element x

o/p: position of x

i/p : (10 (1) 20 (2) 30 (3) 40 (4) 50 (5) 60 (6) ..... 70 (7))

Program:

```
BS(a, i, j, x)
{
    if (i == j)
    {
        if (a[i] == x)
            return i;
        else
            return -1;
    }
```

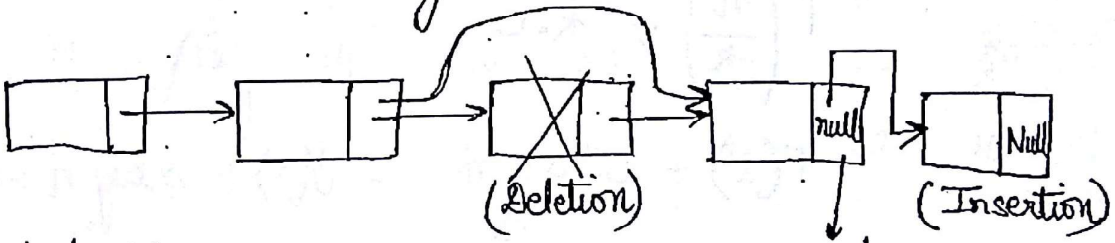
```
else
{
    mid = ⌊(i+j)/2⌋;          ⟹ O(1)
    if (x == a[mid])          ⟹ O(1)
        return mid;
    if (x < a[mid])           ⟹ O(1)
    {
        j = mid - 1;
        pos = BS(a, i, j, x); ⟹ T(n/2)
    }
    else { if "else" is removed, it will go on both sides
    {                                  ⇒ Linear Search }
        i = mid + 1;
        pos = BS(a, i, j, x); ⇒ T(n/2)
    }
    return pos;
}
```

O(1) = C

Best Case ← stops here

Linked List & Array :- Linked List → dynamic allocation

Size is not fixed; easily expandible, element can be deleted/inserted at any position.



In Linked List, random access is not possible (only drawback)

Array → size specified, random access possible, expansion not allowed.

Let $T(n)$ = Time complexity of above algorithm on n-elements, then

Recurrence Relation :=

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

$$T(n) = T\left(\frac{n}{2}\right) + c \quad \leftarrow \text{time to decide which side (L/R) to go}$$

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

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

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

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

$$= T(1) + c \cdot \log_2 n = O(1) + c \log_2 n = O(\log_2 n)$$

Best Case for Binary Search : $O(1)$ Avg./Worst Case

When middle element is search element.

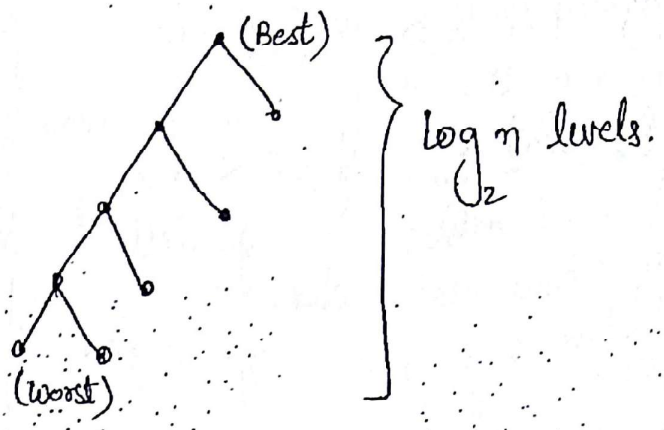
Avg case Proof :-

$1+2+3+\ldots+\log_2 n$

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

$Avg = \frac{\log n(\log n+1)}{2 \cdot \log n}$

$= \frac{\log n+1}{2} = O(\log_2 n)$



Main() —array passed→ BS(a, i, j, x)

(all values are not passed, its base address only passed ⇒ Call by Reference)

Binary Search is not possible in LinkedList.

Ques:

A	-150	-120	-110	-100	-80	-50	-20	-10	0	5
	1	2	3	4	5	6	7	8	9	10

8	10	12	14	20	25	30	50	80
11	12	13	14	15	16	17	18	19

100	130	150	180	200	250	value
20	21	22	23	24	25	pos

i/p: a sorted array of n- distinct elements

o/p: find any elements $a[i]$ such that $a[i] == i$

① LS $\Rightarrow$ $O(n)$

② BS : if ($p < V$) go left
else go right
} $O(\log_2 n)$

mid
<12 V: 12 >12

12 p: 13 14

<12 cannot be equal to 12.

>12 can be 14. Go Right.

- If sorted array not given,

(maybe) mid

Value: 12 .... 12-0 14 (may be)

pos: 12 13 14

< Not possible >

- If sorted array but not distinct elements :-

$\leftarrow$ mid $\rightarrow$

value : 12 15 15 15

pos : 12 13 14 15

Both sides possible $\Rightarrow$ Binary Search not possible.

Imp Ques: i/p : an array of n elements in which until some place all are integers & afterward all are ∞.

o/p : find position of 1st ∞. (array is not sorted in asc/desc order yet Binary Search possible)

⇓

Best algo, worst case

A	-150	+120	+110	-50	-10	6	8	90	∞	∞	∞
	1	2	3	4	5	6	7	8	9	10	11

←

∞	∞	∞	∞	∞	∞	∞	∞	∞
12	13	14	15	16	17	18	19	20

∞	∞	∞	∞	∞
21	22	23	24	25

① LS ⇒ O(n) ✓

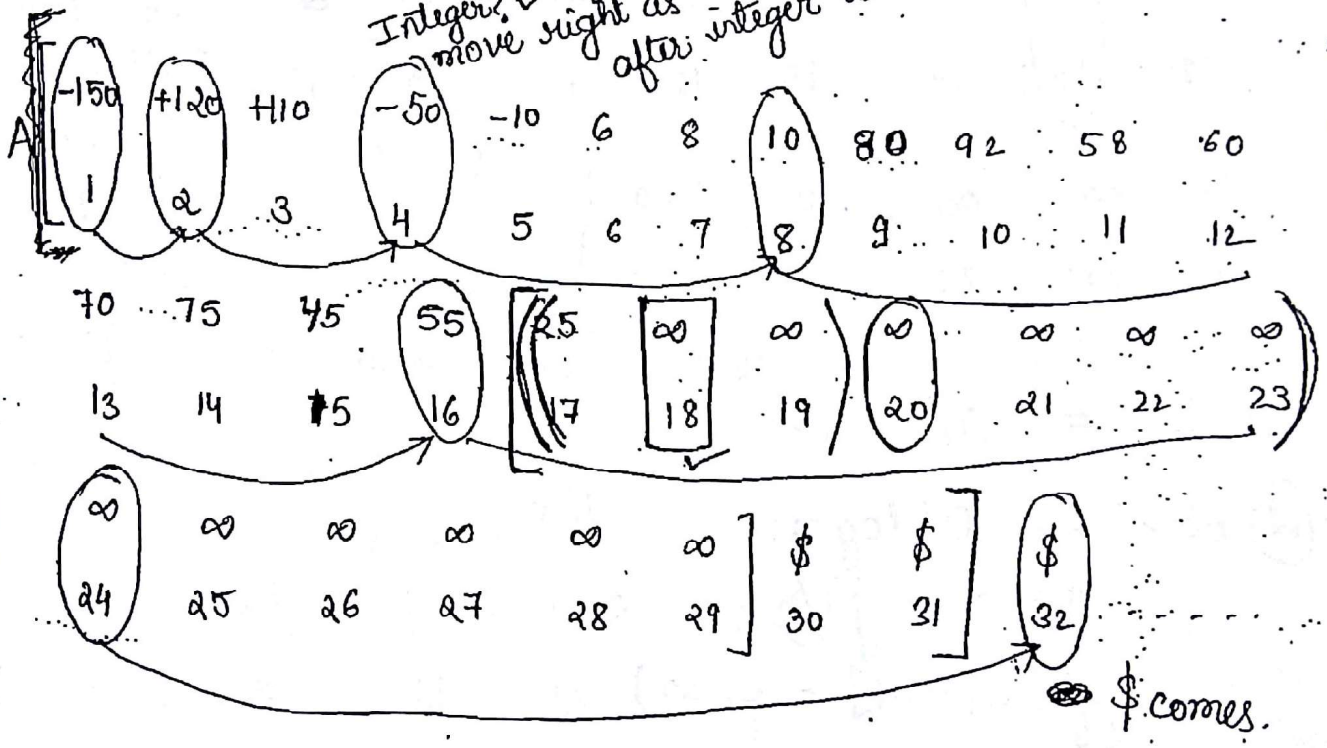
② BS ✓ ⇒ O(log n)

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

```
        i = mid + 1;
        return BS(a, i, j);
    }
```

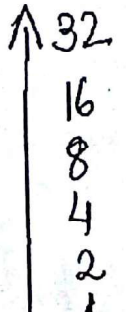
Ques: i/p: an array of n elements in which until some place all are integers & afterwards all are ∞.
[n is unknown & afterwards all are $]

o/p: find position of first ∞.



n is unkown means n cannot be used in algorithm.

Instead of dividing array into halves, multiply by 2.



Ques: i/p : a sorted array of n elements.

o/p : find any 2 elements a & b such that $a+b > 1000$.

100	200	300	400	500	600	700
1	2	3	4	5	6	7

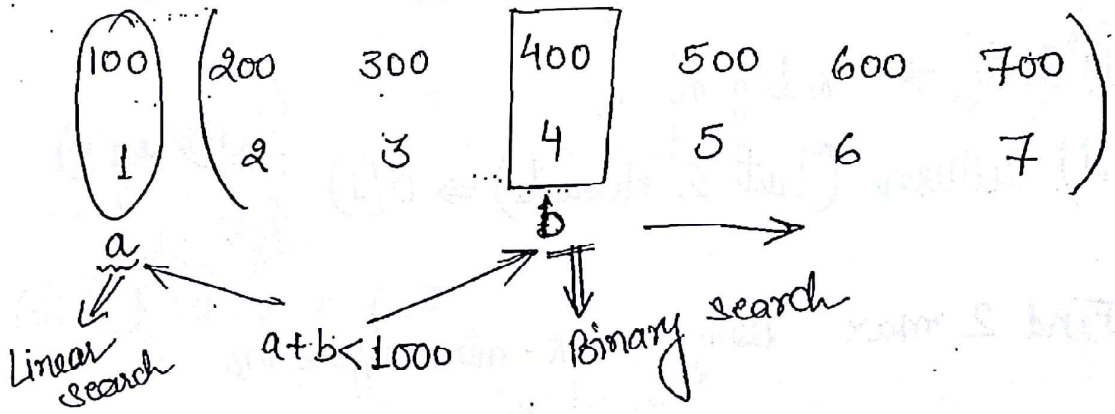
① Linear Search ✓ (n-times)

```
for (i = 1; i <= 7; i++)
{
    for (j = i; j <= 7; j++)
    {
        if (a[i] + a[j] > 1000)
            break;
    }
}
```

Sum of (n-1) nos :-

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

② Binary Search ✓ (n-times)



$O(n \log n)$

③ Check for last 2 elements :

if possible ⇒ they are a & b.

otherwise, no pair possible → O(1) ✓

Ques same as prev., only the i/p array is not sorted.

1) Linear Search ✓ $O(n^2)$

2) Binary Search ✗

✓ But if we sort the array and then apply Binary search

For sort $\Rightarrow n \log n$

For Binary search $\Rightarrow n \log n$

Total $= n \log n + n \log n$

$= O(n \log_2 n)$

(3) i) Sort $\Rightarrow n \log n$

ii) return (Last 2 elements) $\Rightarrow O(1)$ } $O(n \log n)$

(4) Find 2 max. using max-min algorithm

Best

i) 1st max $\Rightarrow O(n)$

ii) 2nd max $\Rightarrow O(n)$ } $O(n)$

iii) Then sum (1st max, 2nd max) > 1000 (Ans)

else no pair possible.

To calculate max & min, use 2 passes of Bubble

Sort

After 1st iteration, max element at last place $\Rightarrow n$

After 2nd iteration, 2nd max at 2nd last place $\Rightarrow n-1$

$$\text{Total} = 2n-1$$
$$= O(n)$$

Ques: i/p: sorted array of n-elements

o/p: find any 2 elements such that $a+b=1000$

① Linear search $O(n^2)$ ~

```
for(i=1 to n)
{
  for(j=i to n)
  {
    if(a[i]+a[j]==1000)
      break;
  }
}
```

② Binary search $O(n \log n)$ ✓

100	300	400	700	800	900
1	2	3	4	5	6

a →

③ Greedy Algo

Let $a+b==950$

$a+b=950$

100	200	300	450	500	600	700
1	2	3	4	5	6	7
			a	b		

a → a → a ... b

$a+b = 1000 > 950$ (dec b)

$a+b = 900 < 950$

Check $a+b = 800 < 950$ (inc. smaller i.e. a)

(i) a – 1st b → last

(ii) while (a+b != 1000)

```
{
    if (a+b < 1000)
        a = a+1;
    else
        b = b-1;
}
```

⇒ O(n) // Best

If the array is not sorted,

Sort it first ⇒ $n \log n$

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

If a+b+c = 1000. then fix a, apply greedy on b&c. and iterate on a.

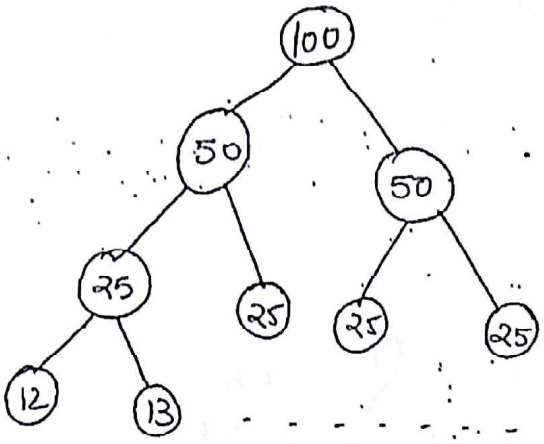
⇒ $O(n^2)$ ✓

Linear Search ⇒ $O(n^3)$

Binary Search ⇒ $O(n^2 \log n)$

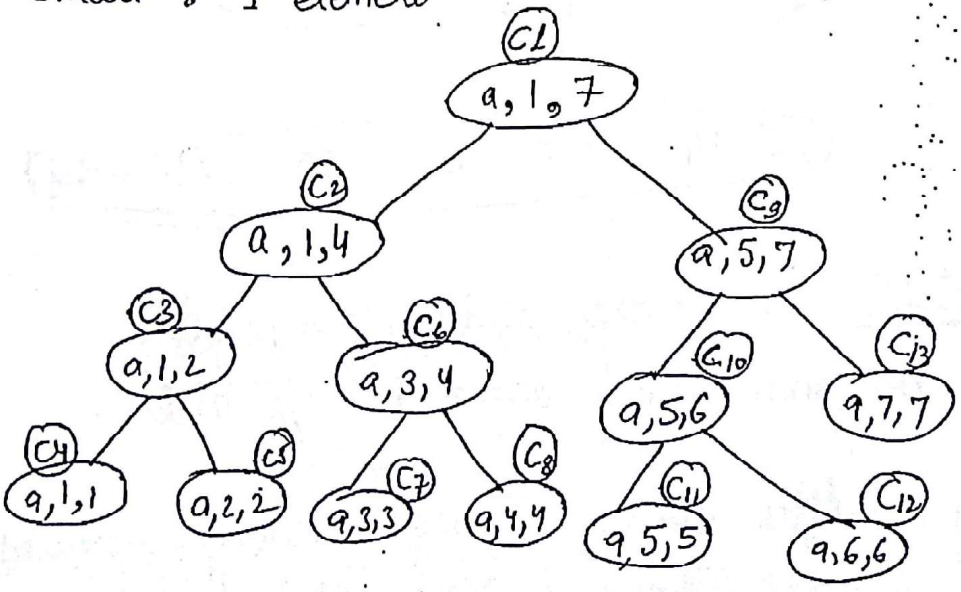
4) Merge Sort

Note: Merging 2 sorted subarrays.

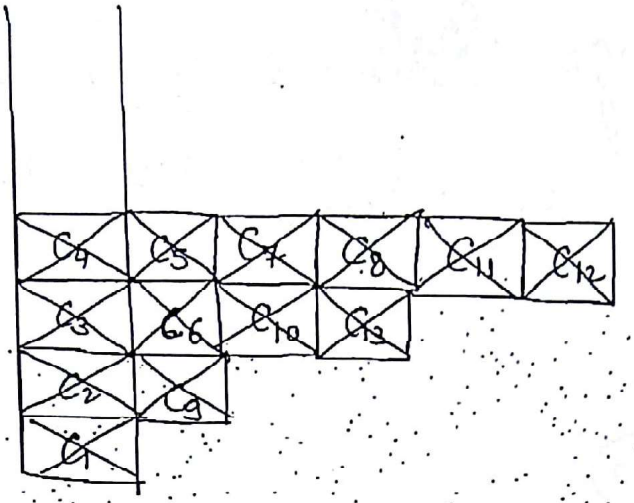


Ex: i/p A = [75 15 90 19 65 28 31]
1 2 3 4 5 6 7

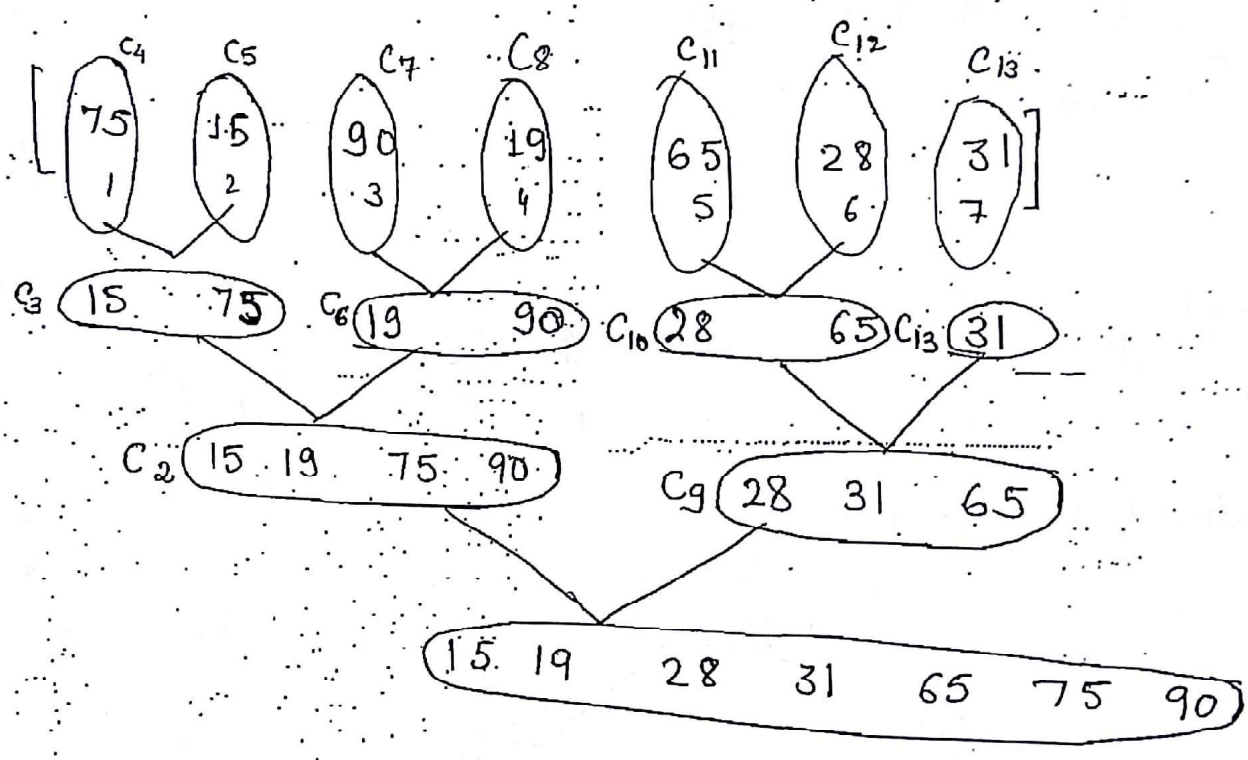
Small : 1 element



Stack :-



Stack Space = 4

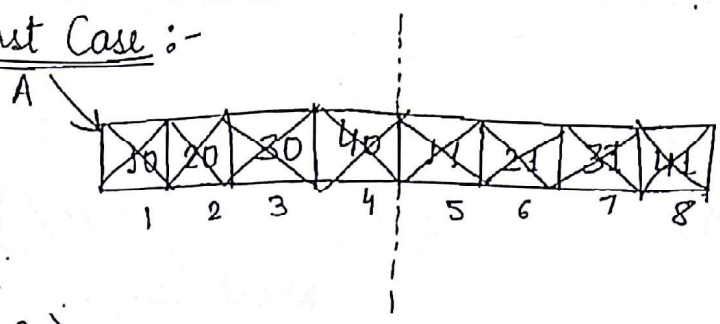


Merge sort is stable i.e. the order of repeated elements should be maintained same as i/p order.

- Stable Sorting :- After sorting & before sorting, repeated elements order is not changed. Merge sort is stable but quick sort is not stable.

Merge Algorithm :-

Worst Case :-



10, 11 ⇒ 10
20, 11 ⇒ 11
20, 21 ⇒ 20
30, 21 ⇒ 21
30, 31 ⇒ 30
40, 31 ⇒ 31
40, 41 ⇒ 40
41 (No comparison)

Comparisons

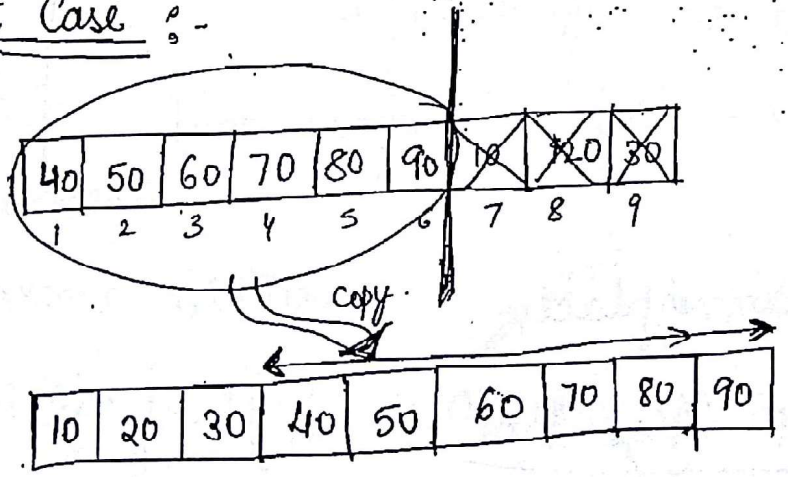
4, 4 ⇒ 4+4−1
m, n ⇒ m+n−1
n/2, n/2 ⇒ n/2 + n/2 − 1

Moves

4, 4 ⇒ 8
m, n ⇒ m+n
n/2, n/2 ⇒ n/2 + n/2 = n

Moves > Comparisons
⇒ Time complexity depends on moves

Best Case :-



40, 10 ⇒ 10
40, 20 ⇒ 20
40, 30, ⇒ 30
40
50
60
70

moves

4, 4 ⇒ 4 + 4
m, n ⇒ m + n
n/2, n/2 ⇒ n/2 + n/2 = n

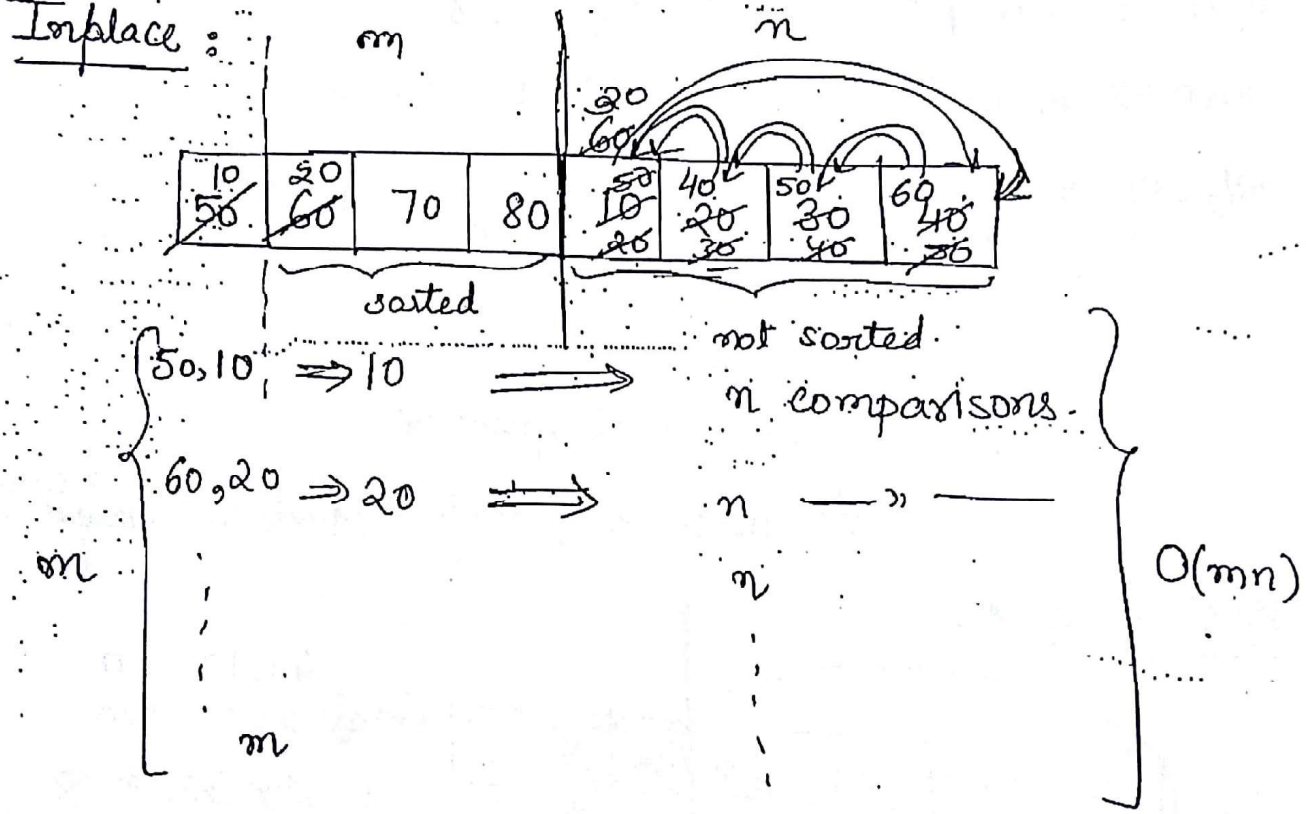
Comparisons

4, 4 ⇒ 4
m, n ⇒ min (m, n)
n/2, n/2 ⇒ n/2

Note :-

- Merging 2 sorted subarrays each of size m & n will take $O(m+n)$ [Best Case / Avg case / Worst case].
- Merge algorithm is requires another extra array for o/p and is called outplace merge algo.
- If space is saved by inplace merge algo,
  Time complexity = $m * n$. and not $m+n$

Inplace :



Quick sort is inplace.

Merge Sort Algorithm :-

$\Rightarrow T(n)$

```
Merge Sort (a, i, j)
{
    if (i == j)
        return a[i];
    else
    {
        mid = ⌊(i+j)/2⌋        → O(1)
        Merge Sort (a, i, mid);      → T(n/2)
        Merge Sort (a, mid+1, j);    → T(n/2)
        Merge (a, i, mid, mid+1, j)
        return a;
    }
}
```

If it is inplace

$\Rightarrow m * n = \frac{n}{2} \cdot \frac{n}{2}$

$= n^2/4 = O(n^2)$

It is outplace

So, $m+n = \frac{n}{2} + \frac{n}{2}$

$= n$

Since new 'b' array used, data copied back from b to a which is again $n/2 + n/2 = n$

So, no change $\Rightarrow O(n)$.

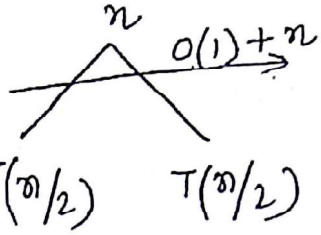
Let $T(n)$ = Time complexity of above algorithm on n-elements.

Recurrence Relation

Combine ↗ Divide ↗

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

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

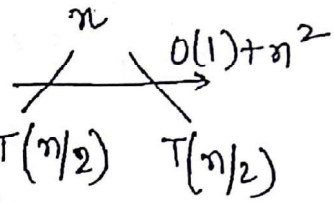


Outplace

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

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

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



Inplace

$$k = \log n$$

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

$$= nO(1) + n.k$$

$$= n + n\log n = O(n\log_2 n)$$ {Best / Avg / Worst case}

n: For every leaf node a cost. Total leaf = n

$n\log n$: All levels cost. Total levels = $\log n$. Cost/level = n.

Space complexity :-

i/p + extra

i/p: n

extra: Stack ($\log n$), B-array (outplace) (n).

$n + \log n + n$

$= O(n)$

Important Points :-

① Merge sort is outplace because in merge sort we take 1 extra array B.

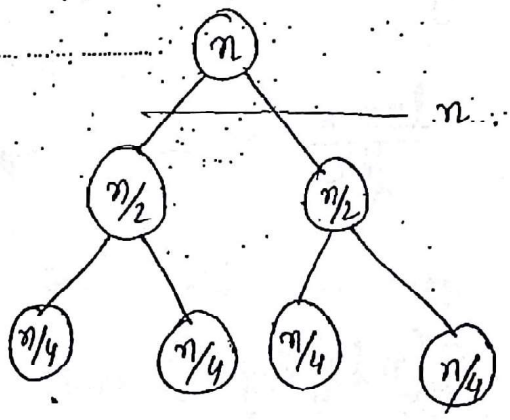
② Merge sort is not advisable for smaller size arrays. { Insertion sort is suitable for smaller arrays }

Ques 1 i/p : $\log n$ sorted subarrays each of size $\frac{n}{\log n}$

o/p : find single sorted array of n element.

~~$= \log n \cdot O(1) + \log n \cdot n$~~

~~$= \log n + n \log n$~~

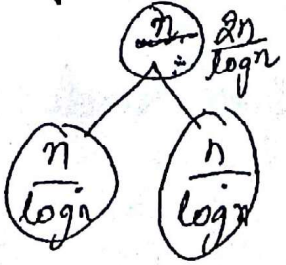


Try

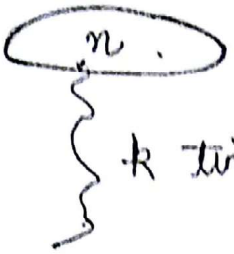
$\frac{2^k \cdot n}{\log n} = 2n$

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

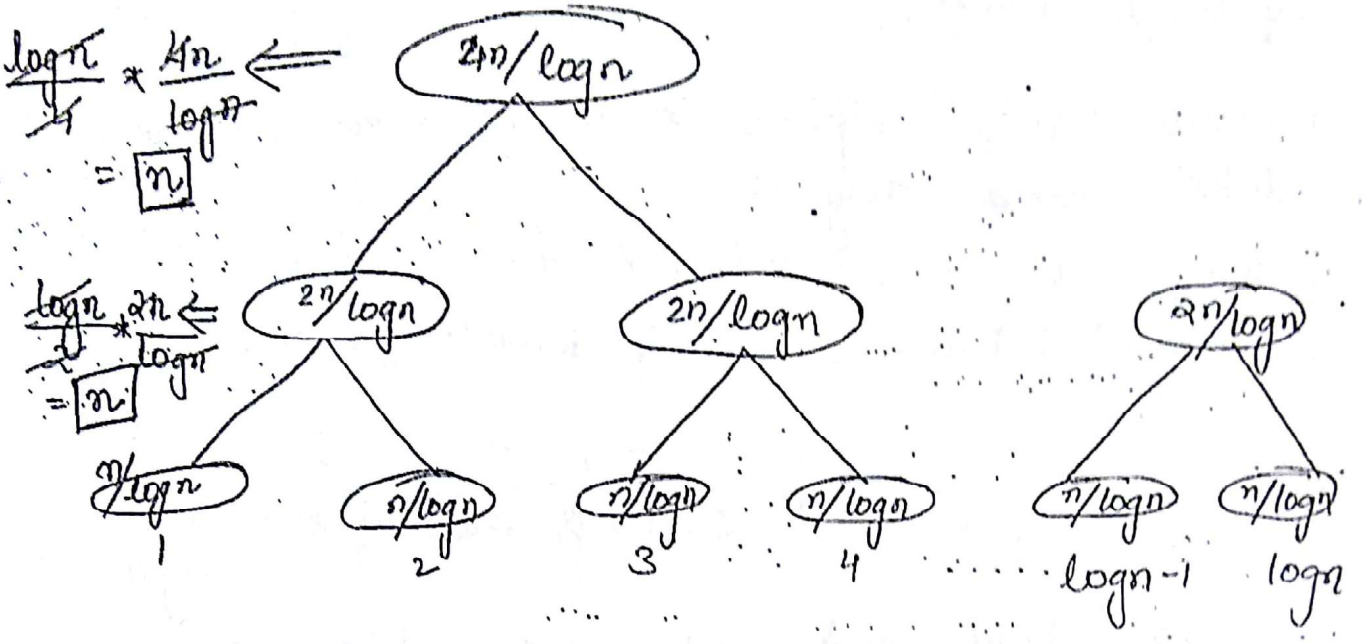
$\Rightarrow k = \log\left(\frac{\log n}{\log n}\right)$



Correct $\frac{\log n}{2^k} * \frac{2^k n}{\log n} = \boxed{n}$ ⟸



$\frac{\log n}{1} * \frac{4n}{\log n} = \boxed{n}$ ⟸



$\frac{\log n}{2} * \frac{2n}{\log n} = \boxed{n}$ ⟸

Cost of each level = n

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

$$k = \log \log n$$

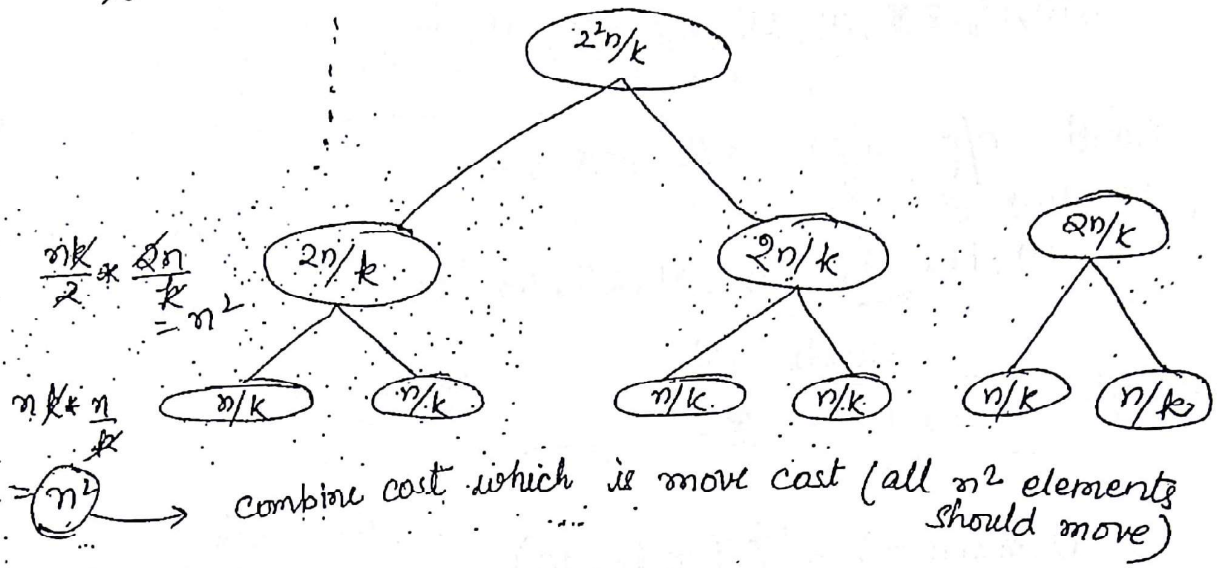
$\Rightarrow O(n \log \log n)$

If in Ques, 1 sorted array given ⇒ Binary Search
more than 1 sorted array ⇒ Merge Sort.

Ques 2 i/p : nk sorted sub-arrays each of size $\frac{n}{k}$

o/p : find single sorted array.

$\frac{nk}{2^p} * \frac{2^p n}{k} = n^2$



→ combine cost which is move cost (all n^2 elements should move)

$$\frac{nk}{2^p} = 1 \Rightarrow nk = 2^p$$

$$\Rightarrow p = \log nk.$$

$O(n^2 \log nk)$

Ques3: Consider the foll. array.

25 10 15 7 99 88 64 21 15 11

what will be o/p after 2nd pass of the straight 2-way merge sort algorithm {without dividing start} combining.

after 8th pass: 7,10,15,21,25,64,88,99

11,15

after 2nd pass: 7,10,15,25 | 21,64,88,99 | 11,15 | 11,15

1st pass o/p: 10,25 | 7,15 | 88,99 | 21,64 | 11,15

25 10 15 7 99 88 64 21 15 11

o/p after 2nd pass :

7, 10, 15_a, 25, 21, 64, 88, 99, 11, 15_b

Final o/p after 4th pass :-

7, 10, 11, 15_a, 15_b, 21, 25, 64, 88, 99

stable sort

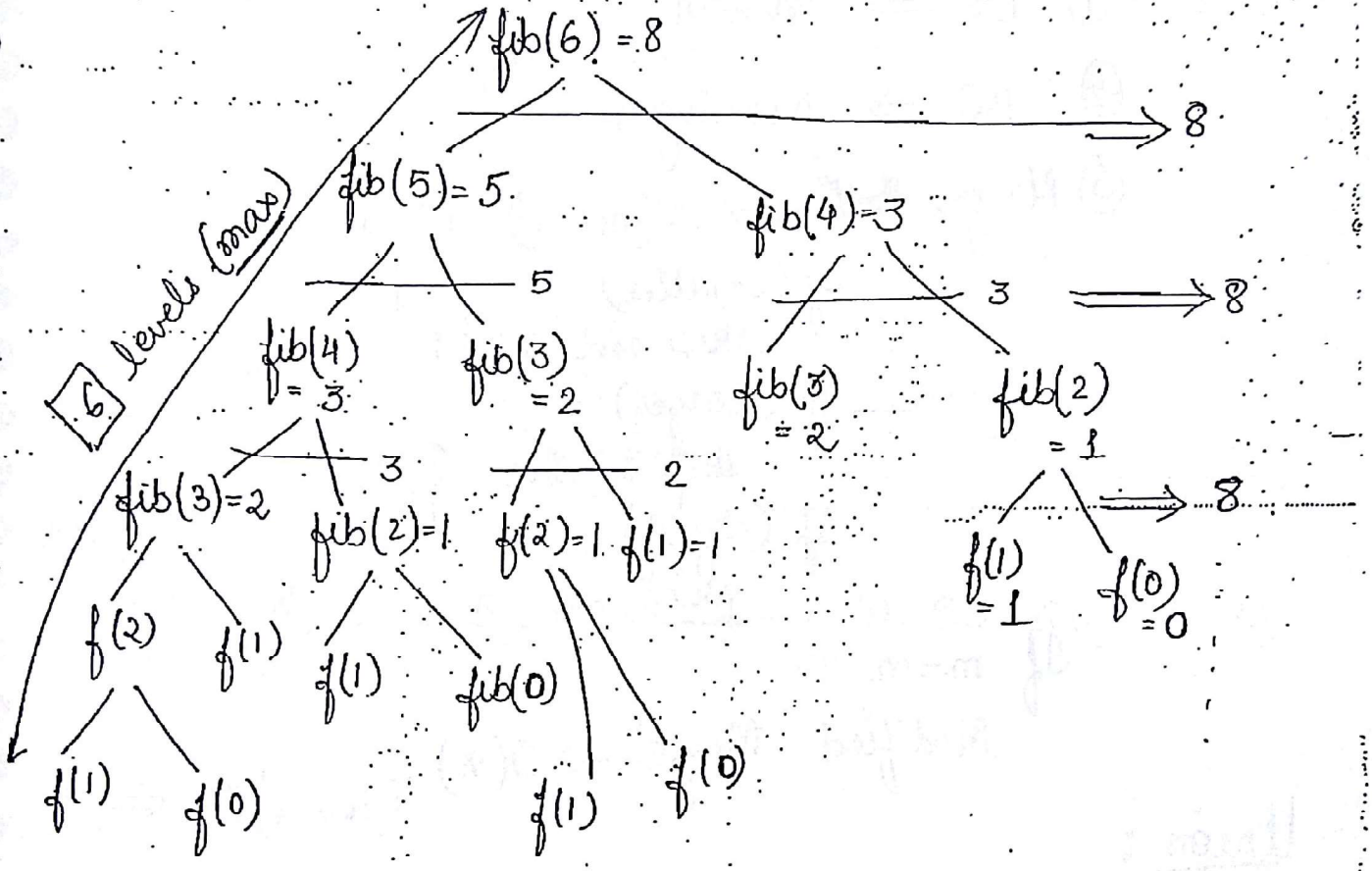
#Levels $= \lceil \log_2 10 \rceil = 4$

$O(n \log n) = O(10 \log_2 10)$

Ques4-

n	0	1	2	3	4	5	6	7	8
$fib(n)$	0	1	1	2	3	5	8	13	21

Using Fibonacci - merge sort ; Time taken to sort $f(n)$ elements.



Total cost $= O(6 \cdot fib(6)) = O(k \cdot fib(k))$

Ex5: i/p : 2 sorted arrays A – m
B – n

o/p : find intersection & union of A & B.

Intersection :-

A : 10 20 30 40 50 60 70.

B : 5 8 10 15 20 25 50 65 70 80

A ∩ B : ① LS ⇒ $m * n$.

② BS ⇒ $m * \log n$

③ Merge ⇒ $O(m+n)$

if (smaller)
skip and inc i
if (larger)
skip & inc. j
if (equal)
print

If $m = n$.
Modified Merge ⇒ $O(n)$.

Union :

A ∪ B : ① LS ⇒ (i) Add A.
(ii) Add B also if not in A
to check this L.S.

⇒ Time = $m + n * m$
$= O(m * n)$

② BS ⇒ Time = $m + n * \log m$
$= O(n \log m)$

③ Modified merge ⇒

```
if(A[i] > B[j])
{
    print (B[j]);
    j = j+1;
}
if(A[i] < B[j])
{
    print (A[i]);
    i = i+1;
}
if (A[i] == B[j])
{
    print (A[i]);
    i = i+1;
    j = j+1;
}
```

A ∪ B =

⇒ {5, 8, 10, 15, 20, 25, 30, 40, 50, 60, 65, 70, 80}

⇒ $O(m+n)$

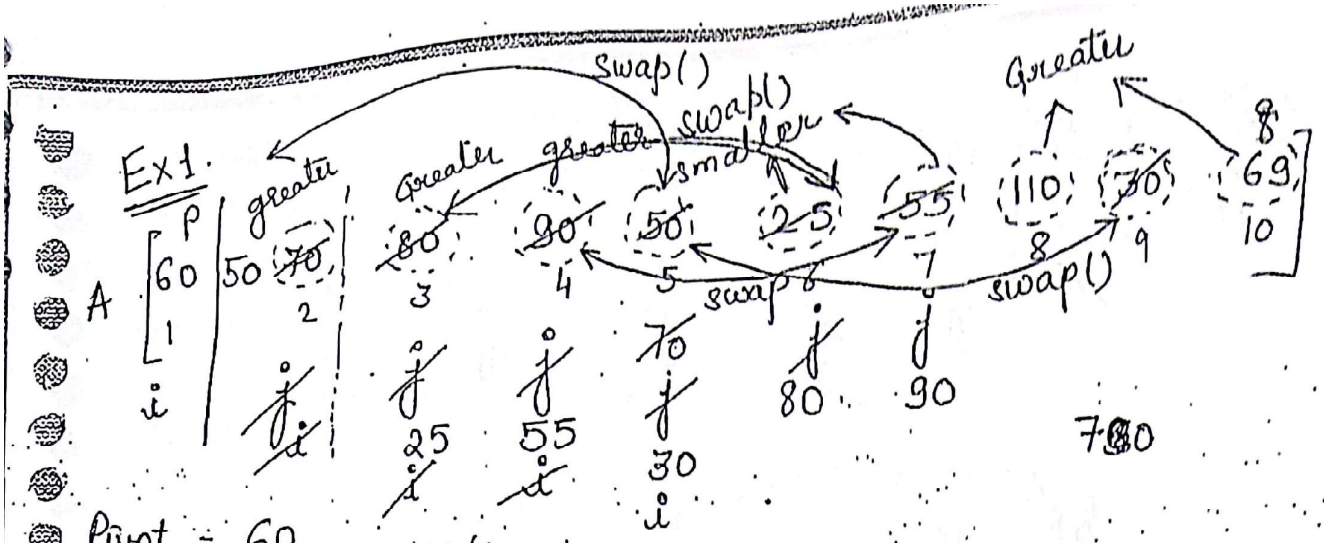
QUICK SORT (Tony Hoare)

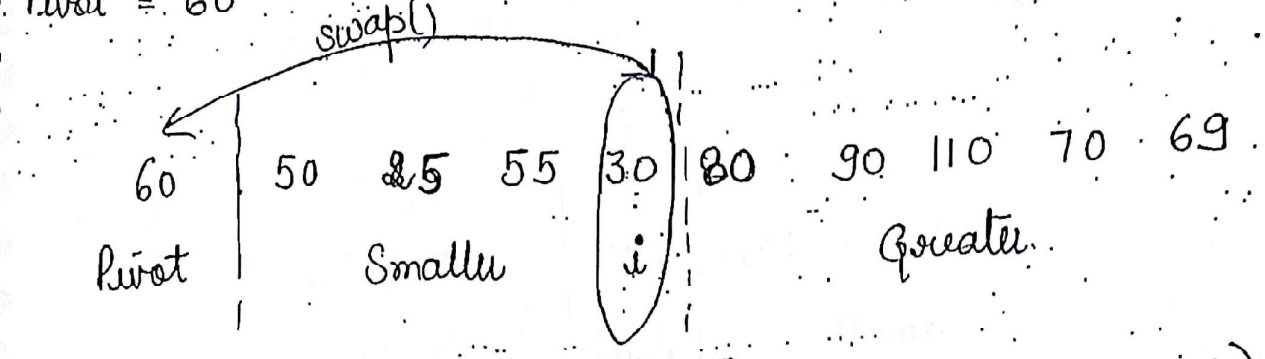
(Fastest Sorting Algorithm)

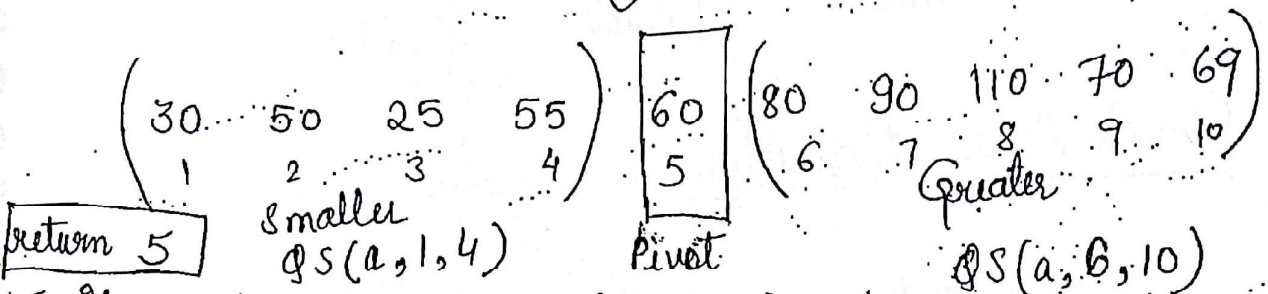
1. DAC application
2. Inplace (no extra space reqd.)
3. not stable
4. Most practically used sorting algo.

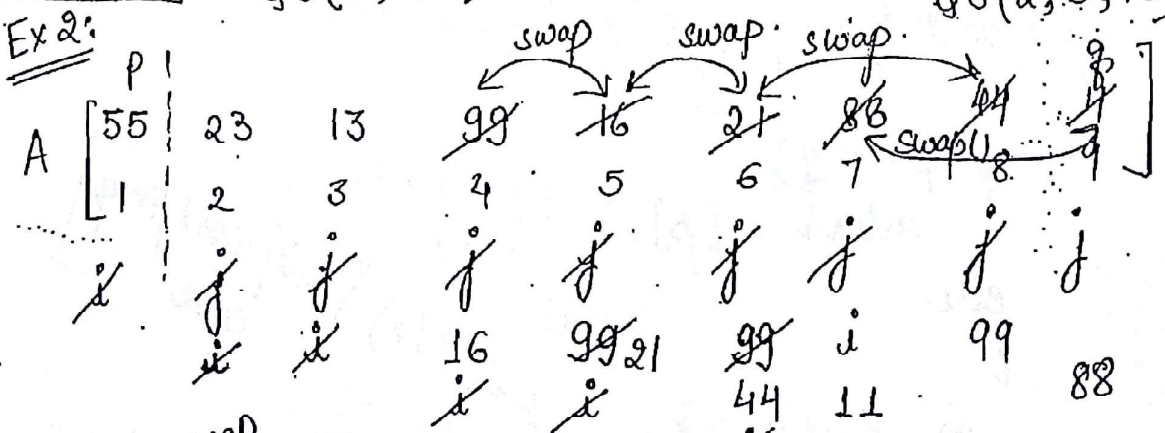
Partition :-

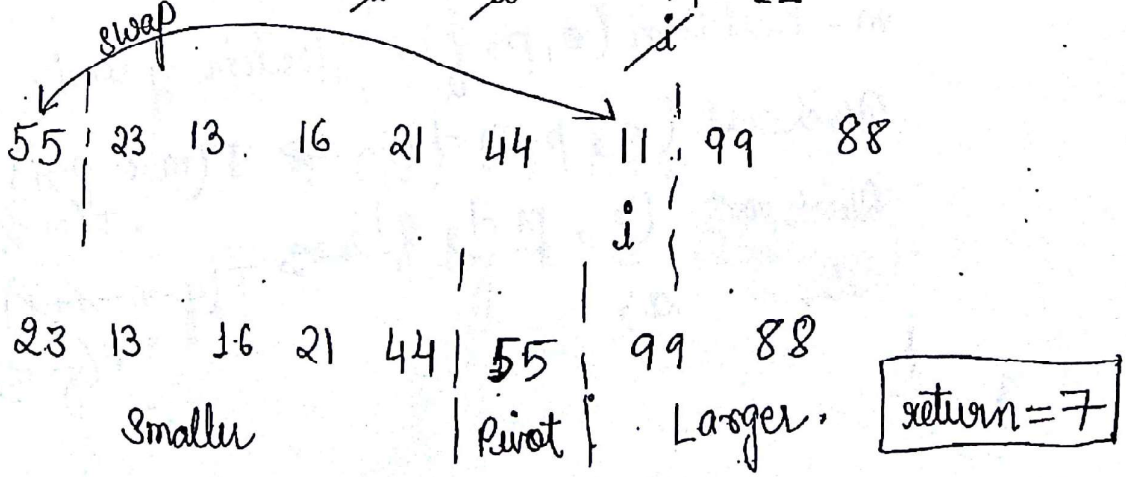

```
Partition (a, p, q)
{
    x = a[p];
    i = p;
    for (j = i+1; j <= q; j++)
    {
        if (a[j] <= x)
        {
            i = i+1;
            swap (a[i], a[j]);
        }
    }
    swap (a[i], a[p]);
    return i;
}
```


Ex 1:
swap()
swap()
swap()
swap()
Greater
Greater
greater
greater
smaller
A
P
i
j
Pivot = 60


swap()
60 50 25 55 30 80 90 110 70 69
Pivot
Smaller
i
Greater


30 50 25 55
60
80 90 110 70 69
Smaller
QS(a,1,4)
Pivot
Greater
QS(a,6,10)
return 5


Ex 2:
swap
swap
swap
swap()
A
P
i
j


swap
55 23 13 16 21 44 11 99 88
i
11 23 13 16 21 44 55 99 88
Smaller
Pivot
Larger
return = 7

Ex 3.

	p					q
A =	88	19	120	15	140	7
	1	2	3	4	5	6

i, j, j, j, j, j

i, i, i

88 19 15 7 140 120

i

7 19 15 | 88 | 140 120

Small | Pivot | Greater

Quick Sort Algorithm :-

```
Quicksort ( a , p , q)  ⇒ T(n)
{
    if (p == q)
        return a[p];
    else
    {
      m = Partition (a, p, q);  // Position of Pivot.   ⇒ O(n) [best/worst/Avg case]
      Quicksort ( a , p , m-1);  ⇒ T(m-1-p+1) = T(m-p)
      Quicksort ( a , m+1 , q);  ⇒ T(q-m-1+1) = T(q-m)
      return a;
    }
}
```

Let $T(n)$ be the time complexity of above program on an array of n elements

Recurrence Relation

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

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

$\Rightarrow$

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

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

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

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

$$k = \log n$$

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

$$= n.O(1) + n.\log n$$

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

☆ When 1st element chosen as pivot, always, then it is Quick Sort. When element chosen randomly, it is randomized Quick Sort.

$$T(n) = \begin{cases} O(1) & , n = 1 \\ O(n) + T(m-p) + T(q-m) & , n > 1 \end{cases}$$

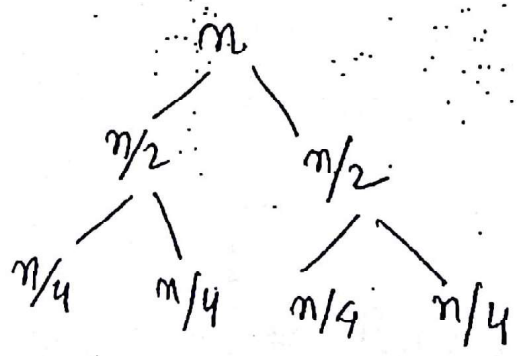
Best Case or Avg Case — Worst Case

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

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

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

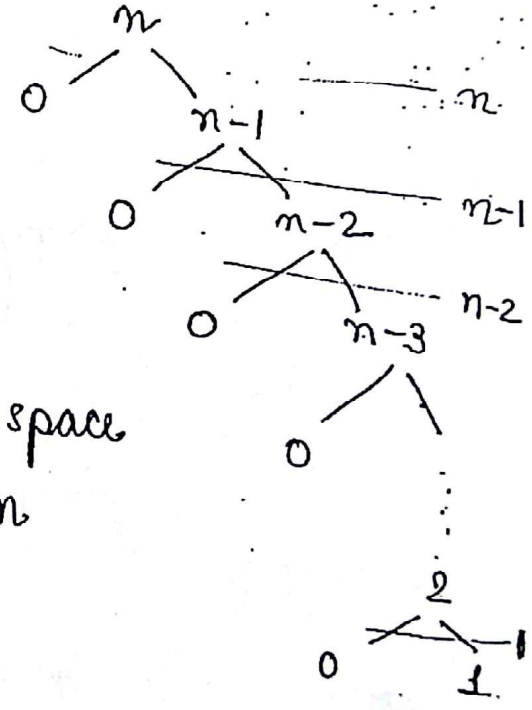
It will give Balanced tree



Stack Space = $\log n$.

$$T(n) = n + \underbrace{T(0)}_{\text{1 Pivot}} + T(n-1)$$

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



Stack space = n

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

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

Example:

① 40 10 50 60 30 20 70
1 2 3 4 5 6 7

o/p (10 30 20) 40 (50 60 70)

② 70 10 50 60 30 20 40
1 2 3 4 5 6 7

o/p: (10 50 60 30 20 40) 70 ( )

③ 20 70 50 60 30 10 40.

o/p: (10) 20 (70 50 60 30 40)

Stack Space (worst Case) = n $\xrightarrow[\text{Programming (Equivalent non-recursive program)}]{\text{Using Better}}$ $\log n$.

Average Case :-

Lucky (best)
unlucky (worst)

Avg case (LULU ..... LU
or
ULUL ..... UL)

Avg (LULU ... LU) $\Rightarrow T(n) = 2T(n/2) + n$

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

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

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

Avg (ULULU...UL) $\Rightarrow T(n) = T(n-1) + n$

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

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

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

$$= O(n \log n) \quad \{\text{Same as Best Case}\}$$

Ques: Quick Sort algorithm is applied on 2 i/ps :-

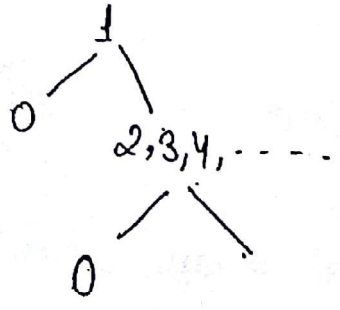
i) 1, 2, 3, 4, ... $n-1, n$

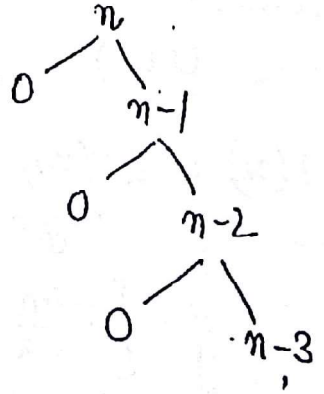
ii) $n, n-1, n-2, \ldots\ldots 3, 2, 1$

Let C_1 & C_2 be the no. of comparisons made for the i/p (i) & (ii) respectively.

What is the relation b/w them

a) $C_1 = C_2$, b) $C_1 < C_2$, c) $C_1 > C_2$, d) not comparable.

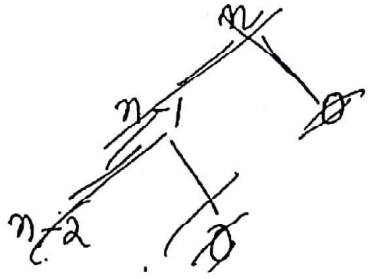




$C_1 =$ [illegible] $n-1 + \ldots\ldots + 1$

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

$= O(n^2)$



$$C_2 = \frac{n(n-1)}{2} = O(n^2)$$

i) Taking example of some array :-

	10	20	30	40	50	60	70
	1	2	3	4	5	6	7
i	i	j	j	j	j	j	j

$\Rightarrow$ 6 comparisons, 1 swap.

( )(10)(20 30 40 50 60 70)

$\Downarrow$ $\Rightarrow$ 5 comparisons,

( )(20)(30 40 50 60 70) 1 swap

$\Downarrow$ $\Rightarrow$ 4, 1

( )(30)(40 50 60 70)

$\Downarrow$ $\Rightarrow$ 3, 1

( )(40)(50 60 70)

$\Downarrow$ $\Rightarrow$ 2, 1

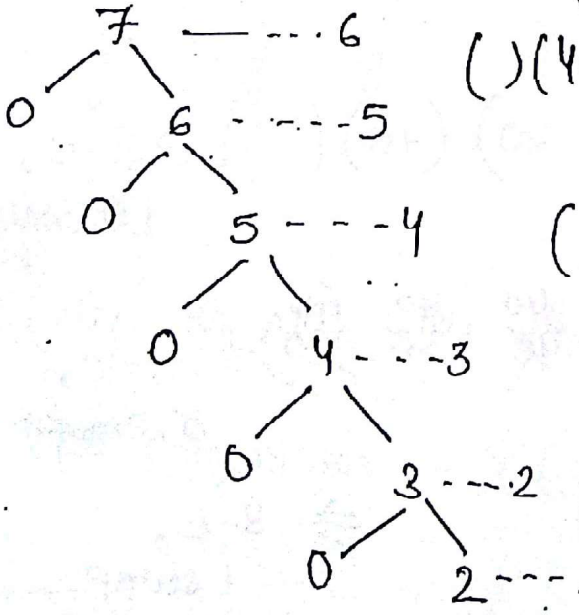
( )(50)(60 70)

$\Downarrow$ $\Rightarrow$ 1, 1

( )(60)(70)

$\Downarrow$

small problem.



Recurrence Relation :-

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

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

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

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

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

Swaps = $1+1+1+\dots$ $(n-1)$ times

$= n-1$

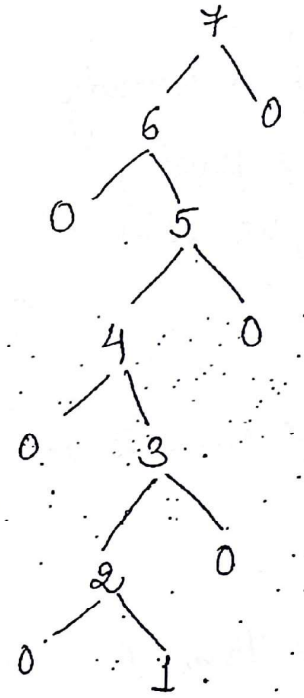
ii) 70, 60, 50, 40, 30, 20, 10 ⇒ 6-comparisons, 7 swaps

(10 | 60 50 40 30 20)(70)( ) ⇒ 5-c, 1 swaps

( )(10)(60 50 40 30 20) ⇒ 4-c, 5 swaps

(20 50 40 30)(60)( ) ⇒ 3-c, 1 swap

( )(20)(50 40 30) ⇒ 2-c, 3 swaps



No. of comparisons $(C_2) = \sum(n-1)$

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

$= O(n^2)$

A) $(C_1 = = C_2)$

Total swaps $\Rightarrow \approx n+1+n+1+\cdots\cdots$

$= \frac{n}{2} \cdot n + \frac{n}{2} \cdot 1$

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

For swaps, (B) $C_1 < C_2$

In partition algorithm, swaps might change but no. of comparisons remain same (always).

iii) swap

10_f ~~10_a~~ ~~10_b~~ ~~10_c~~ ~~10_d~~ ~~10_e~~ ~~10_f~~ 10_a

1 2 3 4 5 6.

~~i~~ ~~j~~ ~~j~~ ~~j~~ ~~j~~ j

i i ~~i~~ ~~i~~ i

Unbalanced Tree $(10_f \; 10_b \; 10_c \; 10_d \; 10_e)(10_a)( )$ // Repeated elements order changed.

Total comparisons $= \sum(n-1) = \frac{n(n-1)}{2} = O(n^2)$

Total swaps $= \frac{n(n-1)}{2} = O(n^2)$

Quick sort worst case is when array is already sorted. (prev. ques.) and 1st element is pivot element. If middle element is pivot then it'll be best case.

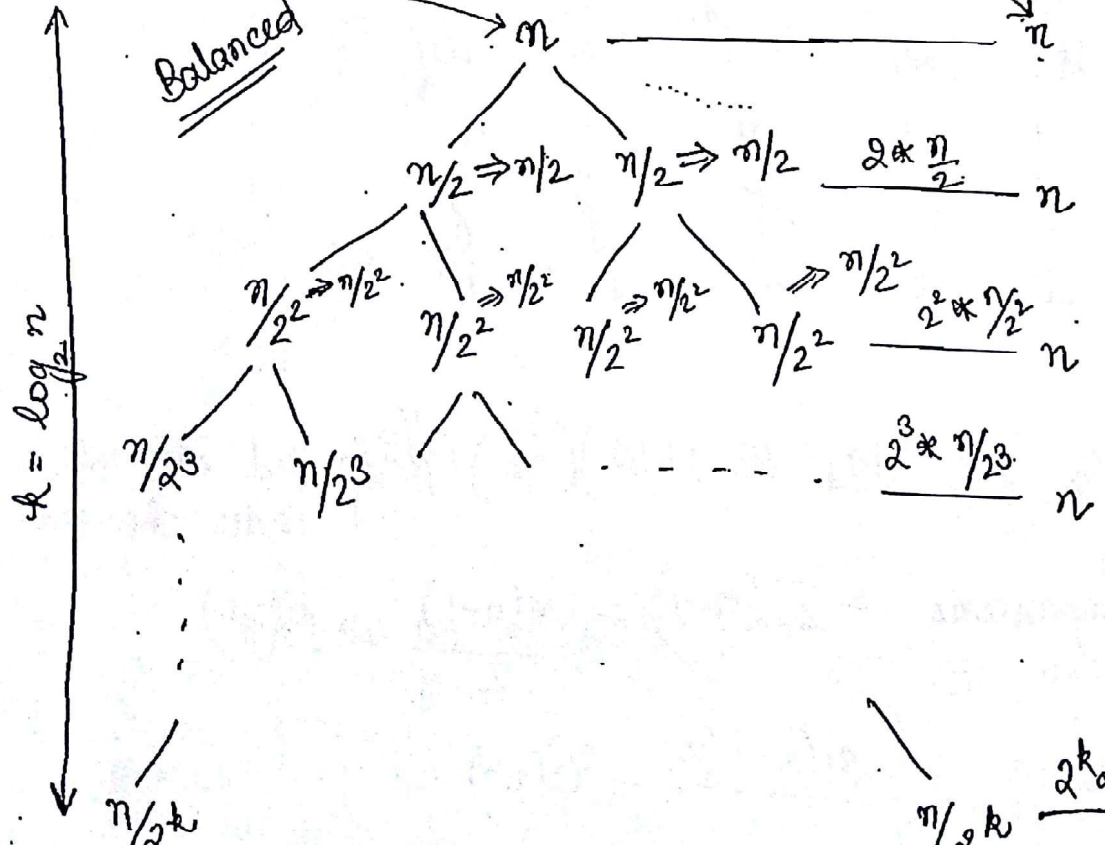
Insertion sort is prefered when max. elements are already sorted.

When recurrence relation has more than 1 function call, Recursive tree method is used.

Ex: $T(n) = n + T(n/5) + T(4n/5)$

Recursive Tree Method

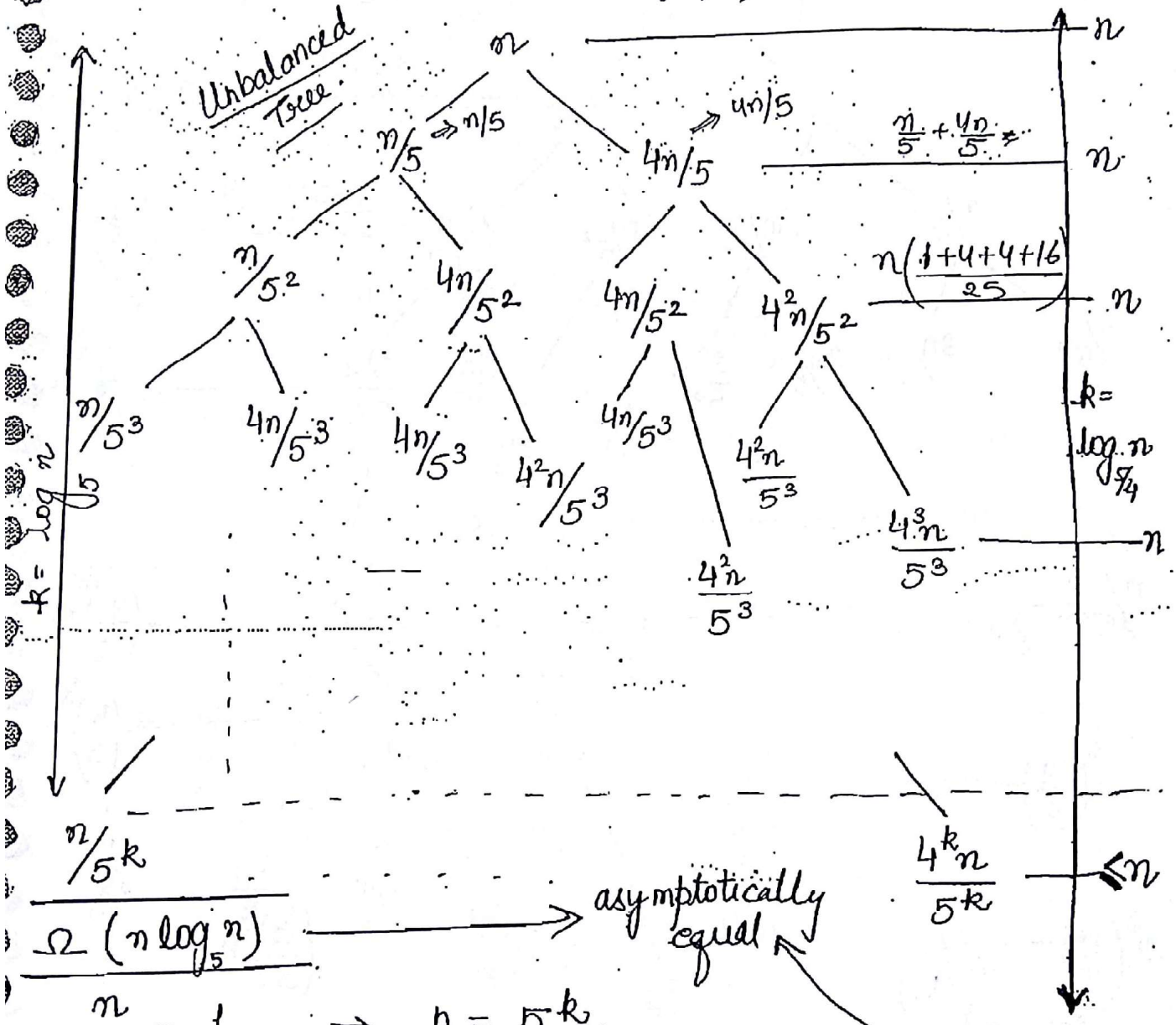
(1) $T(n) = n + T(n/2) + T(n/2)$



$\frac{n}{25} + \frac{4n}{25} + \frac{4n}{25} + \frac{16n}{25}$

$T(n) = \theta(n \log_2 n) = O(n \log_2 n) = \Omega(n \log_2 n)$

② $T(n) = n + T(n/5) + T(4n/5)$



$\frac{n}{5^k} = 1 \Rightarrow n = 5^k$

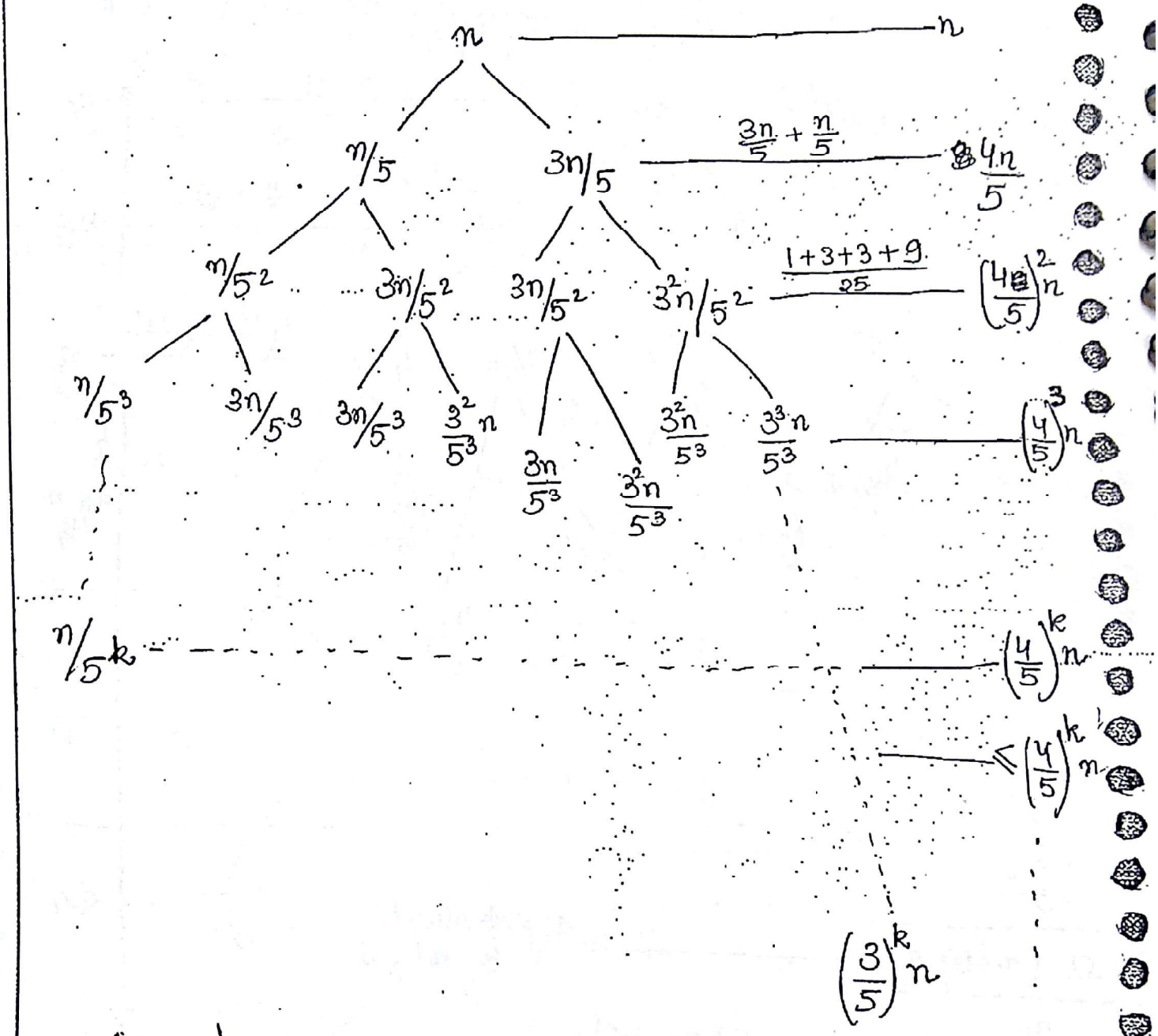
$k = \log_5 n$

$\frac{n}{(5/4)^k} = 1 \Rightarrow n = \left(\frac{5}{4}\right)^k$

$\Rightarrow k = \log_{5/4} n$

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

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



$\frac{n}{5^k} = 1$

$\Rightarrow k = \log_5 n$

$\Rightarrow n + \frac{4n}{5} + \left(\frac{4}{5}\right)^2 n + \cdots \left(\frac{4}{5}\right)^{\log_5 n} n$

$\Rightarrow n\left(1 + \frac{4}{5} + \left(\frac{4}{5}\right)^2 + \cdots \left(\frac{4}{5}\right)^{\log n}\right)$

$= 2n \rightarrow \Omega(n)$

Ques: In quick sort, the sorting of n nos., we n/7 th largest element is selected as pivot using $O(\log n)$ time. Then what will be worst case time complexity.

$\frac{n}{7}$ th largest $\Longrightarrow$ $\left(n - \frac{n}{7}\right)$ th smallest

$= \left(\frac{6n}{7}\right)$ th smallest

$\Rightarrow$ $\underbrace{\qquad\qquad}_{\frac{6n}{7}}$ $\left(\frac{6n}{7}+1\right)$ $\underbrace{\qquad\qquad}_{n/7-1}$

$\Rightarrow T(n) = O(\log n) + O(n) + T\left(\frac{6n}{7}\right) + T\left(\frac{n}{7}-1\right)$

$= n + \log n + T\left(\frac{6n}{7}\right) + T\left(\frac{n}{7}\right)$

$= n + T\left(\frac{6n}{7}\right) + T\left(\frac{n}{7}\right)$

$= O(n \log n)$

$\left(\frac{n}{2}\right)$ th smallest

Ques: In quick sort, the sorting of n nos., median is selected as pivot using $O(n^2)$ time complexity algo. Then what will be worst case time complexity?

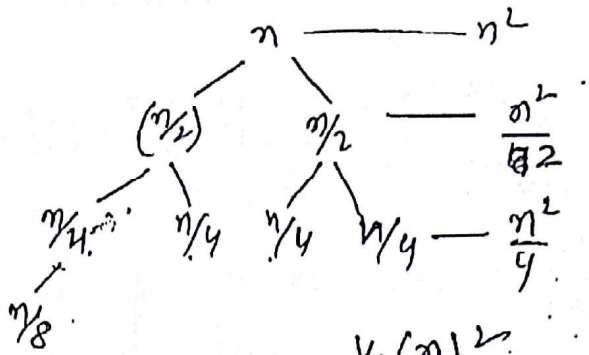
$\underbrace{\qquad\qquad}_{n/2}$ O $\underbrace{\qquad\qquad}_{n/2}$

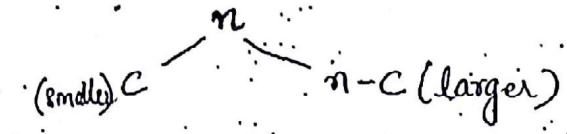
$T(n) = O(n^2) + O(n) + 2T\left(\frac{n}{2}\right)$

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

$\Rightarrow n^2 \left\{1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots \right\}$.

$\Rightarrow O(n^2)$.





Apply recursion at smaller side and while loop (no recursion) at larger side → using this algo the stack space can be reduced to $\log n$ in worst case.

Inplace → any algo taking extra space less than or equal to $\log n$.

Outplace → $> \log n$ ... $\log n$.

So, Quick sort is inplace and merge sort is outplace $(n + \log n)$

Randomized Quick Sort :- Here, the pivot element is chosen at random. The worst case probability gets reduced. Best/Avg Case ⇒ $O(n \log n)$, Worst Case ⇒ $O(n^2)$

Ex: 40 10 20 30 40 10 50 60 70
1 2 3 4 5 6 7

Extra 2 lines: (let 4) = r = RG(1,7) → Random generator
swap(a[p], a[r]) → After this, becomes Same as Quick Sort

(20 30 10) 40 (50 60 70)

(No combine ⇒ Partial DAC appl")

SELECTION Procedure

i/p : an array of n- elements & integer - k

o/p : return kth smallest element

Ex:

A[	60	20	50	45	30	80	70	25	85	90	55	15	
	1	2	3	4	5	6	7	8	9	10	11	12	
	28	99	18	]	$k=10$								
	13	14	15										

⇓ Partition algorithm

10

m = (10 15 20 25 28 30 45 50 55) (60) (80 70 85 90 99)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

if (m == k) stop & return a[m]

⇓ m ≠ k (Partition)

Let $k=13$.

Right side $k > m$

(70) (80) (85 90 99)

11 12 13 14 15.

⇓ m ≠ k (Partition on right)

( ) (85) (90 99)

13 14 15

$m=k$ ✓

Return A[m]

Algorithm :-

```
Selection_Procedure (a, p, q, k)          ⇒ T(n)
{
    if (p == q)  // small problem
        return (a[p]);                    ⇒ O(1)
    else
    {
        m = Partition (a, p, q);          ⇒ O(n)
        if (m == k)
            return a[m];                  ⇒ O(1)
        else
        {
            if (k < m)                    ⇒ T(m-p)
                Selection_Procedure (a, p, m-1, k);
            else
                Selection_Procedure (a, m+1, q, k);   ⇒ T(q-m)
        }
    }
}
```

m-1-p+1
q-m-1+1

$$T(n) = \begin{cases} O(n) + T(m-p) \text{ or } T(q-m), & n > 1 \\ O(1), & n = 1 \end{cases}$$

(Time Complexity)

Best Case :- $T(n) = n + T(n/2) \xrightarrow[\text{levels}]{\log n} O(n)$

Worst Case :- $T(n) = n + T(n-1) \xrightarrow[\text{levels}]{n} O(n^2)$

↓

when array already sorted But when array already sorted, kth smallest element can be obtained by a[k].

Strassen's Matrix Multiplication

Without DAC,

matrix addition $\Rightarrow O(n^2)$

matrix multiplication $\Rightarrow O(n^3)$.

With DAC,

Matrix multiplication

① 2×2 or less than 2×2 (Small)

② Square matrices

③ Rows & columns are powers of 2

Ex:

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

a_{11} a_{12} a_{21} a_{22} $\frac{4\times4}{2} = 2\times2$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

b_{11} b_{12} b_{21} b_{22} $\frac{4\times4}{2} = 2\times2$

$C = A \times B$.

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$

$$a_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

1 - Matrix multiplication - 4×4

⇓

8 - MM — 2×2

+

4 - Add^n - 2×2

1 - MM — 8×8

⇓

8 - MM — 4×4

+

4 - Add - 4×4

1 - MM — n×n

⇓

8 MM — n/2 × n/2

+

4 - Add - n/2 × n/2

Let T(n) = Time complexity required to multiply n×n matrices.

$T(4) = 8 * T(4/2) + 4.(4/2)^2$

$T(8) = 8 * T(8/2) + 4 * (8/2)^2$

$T(64) = 8 * T(64/2) + 4 * (64/2)^2$

⇓

$$T(n) = \begin{cases} 8T(n/2) + 4(n/2)^2 = 8T(n/2) + n^2 & , \ n > 2 \\ O(1) & , \ n \leq 2 \end{cases}$$

$T(n) = 8T(n/2) + n^2 = O(n^3)$

Space complexity = log n

⇒ With DAC, there is no change/improvement in the time complexity. Instead since ~~stack is~~ recursion is used, stack space of "log n" required which is not reqd. in without DAC matrix multiplication. That is why this method is not preferred.

Strassen reduced the no. of multiplications by 1 and increased the no. of additions.

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

$$= 7T\left(\frac{n}{2}\right) + \frac{9}{2}n^2 \quad \text{--}$$

$$\Downarrow$$

$$O\left(n^{\log_2 7}\right) = O\left(n^{2.81}\right)$$

At present, algorithm available for matrix multiplication has time complexity $O(n^{2.32})$

Homework

① i/p: an array of n-elements in which until some place x all are increasing & afterwards all are decreasing.

o/p: find x.

Ans LS: $\xi O(n)$
BS: $O(\log n)$

② i/p: an array of n-elements

o/p: find no. of inversions

30	20	10	5	40	3
1	2	3	4	5	6

inversion is $1 < 2$

$a[1] > a[2]$

30 ⇒ 20, 10, 5, 3
20 ⇒ 10, 5, 3
10 ⇒ 5, 3
5 ⇒ 3
40 ⇒ 3
3 ⇒ —

Total = 11

Ans:
LS ⇒ n^2
DAC ⇒ $n \log n$

(3) i/p: an array of n-points in the (x,y) plane
o/p: find the closest pair.

Ans: DAC: $n \log n$
LS: n^2

Master Theorem

Any recurrence relation of the form DAC, then only master theorem is valid.

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

- a: No. of sub-problems
- n/b: size of sub-problem
- $f(n)$: cost of divide & combine

where $a \geq 1$ and $b > 1$, $f(n)$ should be +ive fnctn.

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ where $\epsilon > 0$ & constant

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

$\frac{n^{\log_b a}}{n^{\epsilon}} = f(n)$

(polynomial) n^{ϵ} times greater than $f(n)$

Case 2: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ where $\epsilon > 0$ & const.

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

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

↓ smaller by n^{ϵ}

Case 3: If $f(n) = \Theta(n^{\log_b a})$

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

Ex1:

$T(n) = 8T(n/2) + n^2$

$a = 8$

$b = 2$

$f(n) = n^2$

$n^{\log_b a} = n^{\log_2 2^3} = n^3$

Case 1 → n^3 (greater of the two)

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

Ex2:

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

$a = 2$

$b = 2$

$f(n) = n^2$

$n^{\log_2 2} = n$

Case 2 → n^2

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

Ex3:

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

$a = 2$

$b = 2$

$f(n) = n$

$n^{\log_2 2} = n$

Case 3 → $n \log n$

Case 1: $n^{\log_b a}$ is polynomial time greater than $f(n)$

Case 2: $n^{\log_b a}$ is ——"—— smaller ——"——

Case 3: $n^{\log_b a}$ is asymptotically equal to $f(n)$.

Ex 4: $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$. Ans. $n \cdot (\log n)^2$ Substitution method.

$a = 2$

$b = 2$

$f(n) = n\log n$

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

n is logarithmic time and not polynomial time smaller than $f(n)$

So, Master Theorem is not applicable.

$n^{\log_b a}$ is logarithmic time smaller than $f(n)$ but not polynomial time. So, Master Theorem is not valid.

$f(n)$	$n^{\log_b a}$
~~n^3~~	$n^4 \log n$
	$= n^3 \cdot n\log n$

It is $n\log n$ times greater. Due to polynomial "n" in "$n\log n$". So, Master theorem is valid.

$f(n)$	$n^{\log_b a}$
~~n^3~~	~~$n^3 \log n$~~ <Not Valid>

NOTE 1:- If $n^{\log_b a}$ is logarithmic time smaller greater than $f(n)$, then

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

$n^{\log_b a} \Rightarrow n(\log n)^{p+1}$ $\quad f(n) \Rightarrow n\log n$

$\Downarrow$

$n.\underline{(\log n)^1} = n\log n$

$T(n) = n(\log n)^{1+1} \ldots = n(\log n)^2$

$n^{\log_b a}$	$f(n)$
n	$n\cdot(\log n)^3$
$\Downarrow$	
$n\cdot(\log n)^3$	

$T(n) = n(\log n)^4$

NOTE 2: If recurrence relation contains root operator

Ex: $T(n) = T(\sqrt{n}) + c$

① assume $n = 2^k$

$T(2^k) = T(2^{k/2}) + c$

② assume $T(2^k) = S(k)$

$S(k) = S(k/2) + c$

Master Theorem valid.

$k^{\log_b a} = k^{\log_2 1} = k^0 = 1$

$f(k) = c = 1$

$\Rightarrow S(k) = O(\log k)$

③ Reverse the substitution 2.

$$T(2^k) = O(\log k)$$

④ Reverse the substitution 1.

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

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

① $n = 2^k$

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

② assume $T(2^k) = S(k)$

$$S(k) = 2S(k/2) + \log_2 2^k$$

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

$k^{\log_b a}$ $\quad a = 2 \quad b = 2 \quad f(k) = k$

$= k^{\log_2 2}$

$= k^1$

$k \cdot \log k$

$\Rightarrow S(k) = O(k \log k)$

③ $T(2^k) = O(k \log k)$

④ $T(n) = O(\log n \cdot \log(\log n))$

Ques: Consider foll. program,

Good Ques:

```
A(n) ⇒ T(n)
{
  if (n <= 1)  ⇒ O(1)
    return 1;
  else
    return ((A(n/2) + A(n/2) + (n)));
}
```

O(1) for addition; no function call/loop.

$T(n/2)$ $T(n/2)$

Time complexity = ?

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

A(n)
↓
A(n/2) + A(n/2) + n

~~a = 2, b = 2~~

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

~~$O(n \log n)$~~

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

$n^{\log_b a} = n$ — $O(n)$

Imp # To find factorial of a number, time complexity is $O(n)$. But the value of factorial of n cannot exceed n^n i.e. $n! = O(n^n)$.

Imp # Similarly,

Sum of 'n' natural nos. value $= \frac{n(n+1)}{2}$

$= O(n^2)$

i.e. it cannot exceed n^2.

But to find sum of 'n' natural nos., time complexity is $O(n)$.

Ques: Find time complexity of foll. C Program:

```
A(n) ⇒ T(n)
{
  if(n <= 1)  ⇒ O(1)
    return n;
  else
    return ( 5A(n/2) + 3A(n/2) + MA(n) );
}
```

(5A(n/2): T(n/2); 3A(n/2): T(n/2); MA(n): $O(n^2)$ — matrix addition)

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

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

$a = 2, \; b = 2$

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

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

Ques: Consider the foll. C Program,

```
A(n) ⟹ T(n)
{
  if(n≤1) ⟹ O(1)
    return 1;
  else
    return (n * A(n-1));   A(n-1) ⟹ T(n-1)
}
```

n * → O(1)

A(n-1) → else → func. call

```
else
{
  a = A(n-1);
  b = n*a; → O(1)
  return b;
}
```

Time complexity ?

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

$T(n-2) + c + c$

$T(n-3) + c + c + c$

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

$T(n) = T(n-k) + kc$

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

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

Recurrence relation for multiplications :-

```
else
{
  a = A(n-1); ⟹ T(n-1)
  b = n*a;    ⟹ 1
  return b;
}
```

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

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

$$\Downarrow$$

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

$$\Downarrow$$

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

$$\vdots$$

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

$$= 0 + n - 1 = n - 1.$$

Recurrence relation for $n!$ value :-

$$T(n) = \begin{cases} 1 , & n \leq 1 \\ n * T(n-1) , & n > 1 \end{cases}$$

$$T(n) = n * (n-1) * (n-2) * \dots 1$$

$$= n! = O(n^n)$$

Recurrence relation for additions :-

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

$$T(n) = 0 + 0 + \dots + 0 = 0$$

Bubble Sort (easy to implement)

i/p : 80 20 50 30 15 11 13 17
1 2 3 4 5 6 7 8

Pass 1:

20 ~~80~~ ~~80~~ ~~80~~ ~~80~~ ~~80~~ ~~80~~ 80
50 30 15 11 13 17

(20 50 30 15 11 13 17) 80

Total comparisons = 7 , Best worst
0–7 swaps

Pass 2

6 comp: 20 ~~50~~ ~~30~~ ~~15~~ ~~11~~ ~~13~~ ~~17~~ 80
30 ~~50~~ ~~50~~ ~~50~~ ~~50~~ 50
15 11 13 17

⇒ (20 ~~30~~ ~~15~~ ~~11~~ ~~13~~ ~~17~~) 50 80
15 ~~30~~ ~~30~~ ~~30~~ 30
11 13 17

Pass 3 (~~20~~ ~~15~~ ~~11~~ ~~13~~ ~~17~~) 30 50 80
5 comp: 15 ~~20~~ ~~20~~ ~~20~~ 20
11 13 17

Pass 4: (~~15~~ ~~11~~ ~~13~~ 17) 20 30 50 80
4 comp: 11 ~~15~~ ~~15~~
13

Pass 5: (11 13 15) 17 20 30 50 80.
3 comp

Pass 6: (11 13) 15 17 20 30 50 80
2 comp

Total comparisons =

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

$= \frac{n(n-1)}{2}$ [Every Case]

Total swaps :-

Best Case = 0.

Worst Case : $(n-1)+(n-2)+\ldots+1$

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

Time = comparisons + swaps $= n^2 + \frac{0-n^2}{}$

$= O(n^2)$ [Every Case]

becoz comparisons are always n^2

In bubble sort, if in any ~~consecutive~~ passes, there are 0 swaps, then stop the algorithm as array has already been sorted.

Ex: 60 10 20 30 40 50

⇓

Pass 1: (10 20 30 40 50) 60 ↕ 0 swaps.

Pass 2: 10 20 30 40 50 60.

The time complexity after including above condition in Bubble sort = $O(n)$ [Best Case], Avg/Worst Case

Selection Sort (min. no. of swaps in worst case)

i/p :

80	20	50	30	15	11	13	17
1	2	3	4	5	6	7	8

Pass 1

~~80 50 30 20 15 13 17~~

swap(a[i], a[min])

i = 1

min = ~~1~~ ~~2~~ ~~5~~ 6

11 (20 50 30 15 80 13 17)

7 comparisons, 1 swap (Best/Worst)

Pass 2

i = 2

min = ~~2~~ ~~5~~ 7

11 13 (50 30 15 80 20 17)

6C, 1-S

Pass 3

i = 3

min = ~~3~~ ~~4~~ 5

11 13 15 (30 50 80 20 17)

5C, 1S

Pass 4

i = 4

min = 4 ~~7~~ 8

11 13 15 17 (50 80 20 30)

Pass 5

i = 5

min = ~~5~~ 7

11 13 15 17 20 (80 50 30)

Pass 6

i = 6

11 13 15 17 20 30 (50 80)

Pass 7 11 13 15 17 20 30 ~~50~~ 50 80

$i = 7$

min = ~~7~~ 7

Total Comparisons := $(n-1) + (n-2) + \ldots + 1$

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

Total swaps = $1 + 1 + \ldots + 1$ ($n-1$ times)

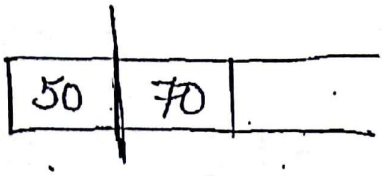
$= (n-1) . = O(n)$.

	Swaps.	
	Best Case	Worst Case
Bubble sort	0	n^2
Selection Sort	$n-1$	$n-1$

Insertion Sort (Best Case Best performance) for swaps & comparisons.

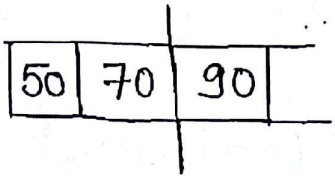
I/p : 50 70 90 75 55 40 100 59

Pass 1 :



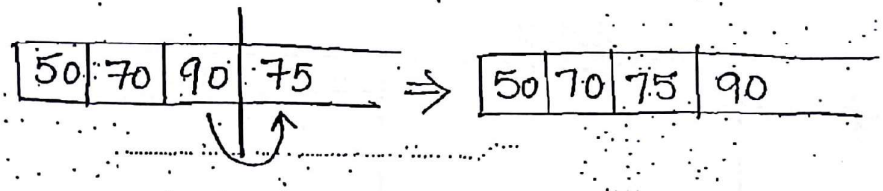
Comparison = 1
Swap = 0

Pass 2 :



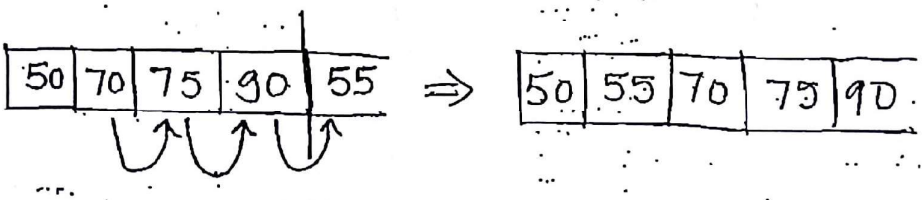
Comparison = 1, swaps = 0

Pass 3 :



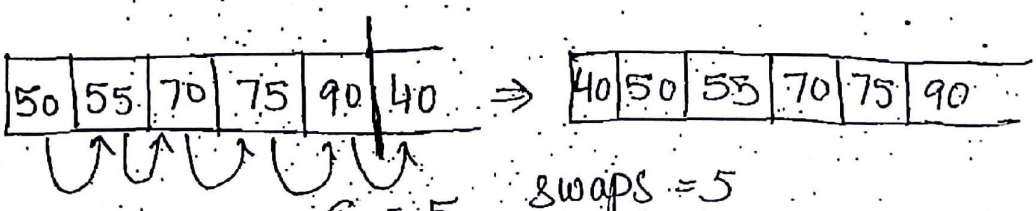
Comparison = 2, swaps = 1

Pass 4



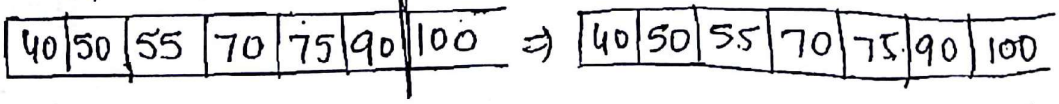
Comparisons = 4, swaps = 3

Pass 5

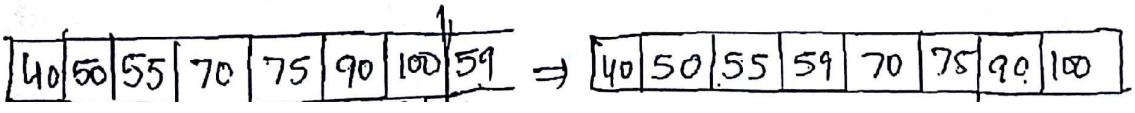


C = 5 swaps = 5

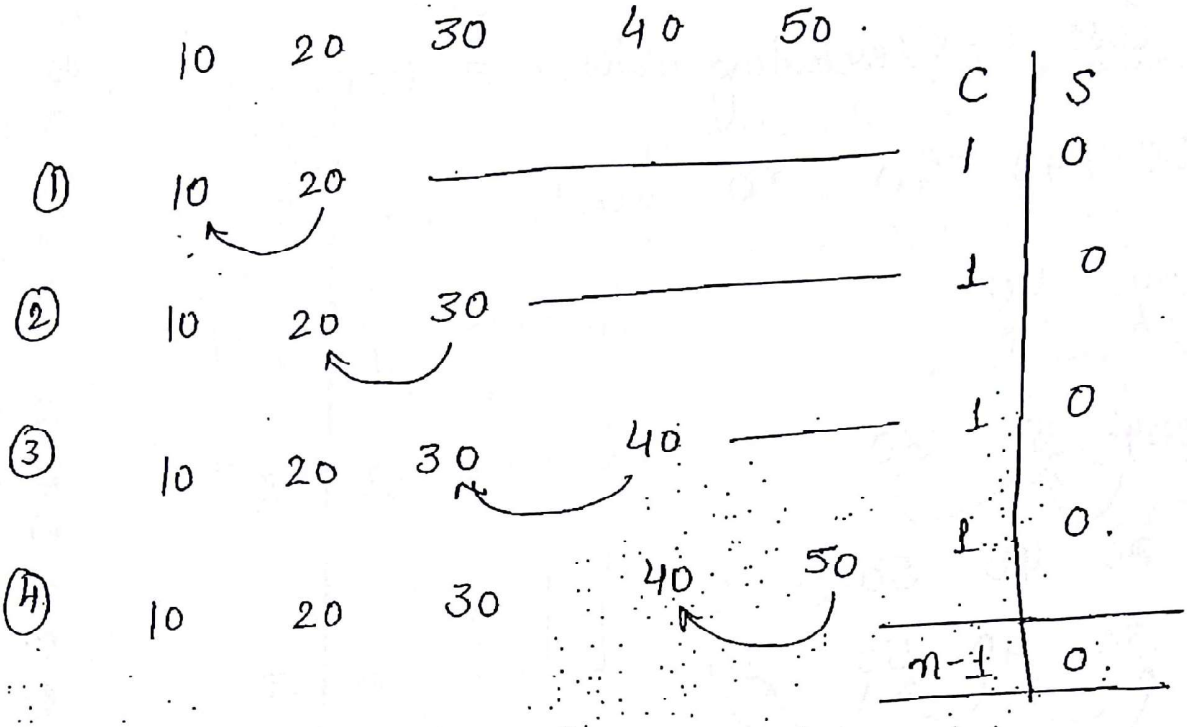
Pass 6



Pass 7



Best Case :-



Best Case time complexity = $O(n)$

- Best Case Best Algorithm ⇒ Insertion Sort

Important Points :-

Insertion Sort Algo will take Best Case $(n-1)$ comparisons to sort the array in Best Case.

In the given array, most of the elements are sorted in ascending order, then Insertion sort algo will give best case performance $O(n)$.

If in given array, max 'n' inversions are there, then insertion sort algo will take $O(n)$ time because most of elements are sorted.

Ex: 10 20 30 40 50 3.

No. of inversions = 5 = $(n-1)$ = No. of swaps

No. of inversions in given array = No. of swap operations in insertion sort algo.

Worst Case :- <Descending order> $= O(n^2)$

50 40 30 20 10

	C	S
① 50 40	1	1
② 40 50 30 → 30 40 50	2	2
③ 30 40 50 20 → 20 30 40 50	3	3
④ 20 30 40 50 10 → 10 20 30 40 50	4	4
	n^2	n^2

Total time = swaps + comp

$= n^2 + n^2$

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

Avg Case :-

10 20 30 40 50 | 9 8 7 6 5

$\frac{n}{2} \cdot 1$ | $\frac{n}{2} \cdot n$

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

H.W.

(1) 1 2 3 4 5 [6] 3 2 (n=8, x=6)

```
BS(a, p, q){
  mid = (p+q)/2
  if (a[mid+1] < a[mid] && a[mid-1] < a[mid])
      return mid;
  else if (a[mid+1] > a[mid])
  return B.S. (a, mid+1, q);
  else
  return BS(a, p, mid-1);
}
```

mid = 4

1 2 3 4 5 6 7 8 → (5 6 3 2)

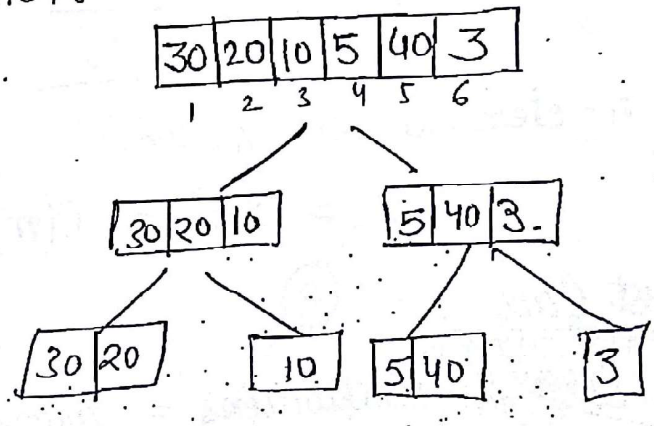
mid = 13/2 = 6.5

1 7 5 [3] 2 1 0

1 [7] 5

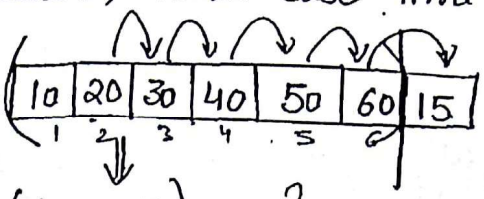
(2) i/p: array of n - elements
o/p: no. of invasions.

DAC



In insertion sort, if we replace Linear search by Binary Search, worst case Time complexity?

Binary Search



Worst Case

Finding place : $\log n$ comparisons

Swapping : $n-1$

For 1 element : $n + \log n$

For n-elements : $n.(n+\log n)$

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

Linear Search

1- element : n - comparison

n - swaps

$2n$

n - elements $\Rightarrow n \cdot 2n$

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

Best Case :-

For 1-element # comparisons = $\log n$

Swaps = 0

$\log n$

For n-elements $\Rightarrow n\log n = O(n\log n)$

If Linked List,

LS: For 1-element : n comparisons.
0 swaps.
n

For n-elements, $O(n^2)$

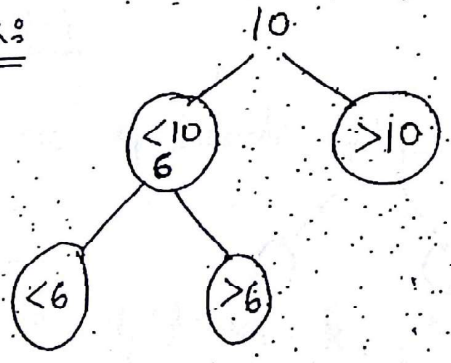
BS: mid cannot be calculated (random access not possible) and if calculate, more than log n time reqd. ⇒ O(n)
n comparison to calculate Length & mid → n/2

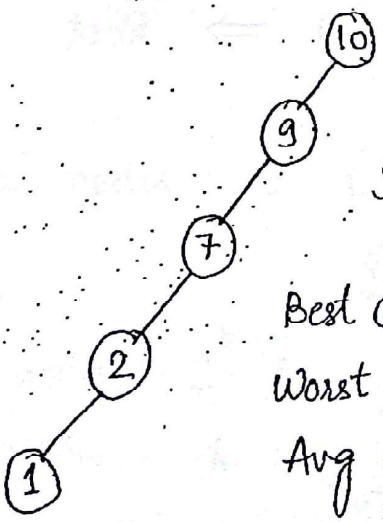
Imp: Heap Sort

Almost complete Binary Tree → ① without completing Left, don't go to Right
max 2 children ② After completion of current level, go to next level

Binary Search tree: should be Binary Tree & left side data smaller than right side data

Ex:





Searching an element;
Best Case : O(1)
Worst Case : O(n)
Avg Case : O(log n)

So, n nodes available, max levels in Binary Search tree = n & avg/min = log n

AVL tree : is a binary search tree (BST) as well as balanced. ⇒ max levels : $\log n$

For searching,

Best Case = $O(1)$

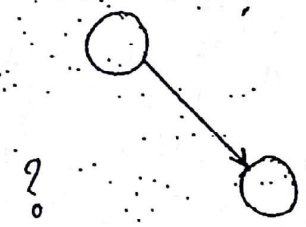
Worst Case = $O(\log n)$

B & B+ tree are balanced trees. Balancing is done using node splitting & node coalesce.

In B+ tree, random & sequential access possible.

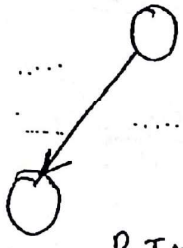
In B tree, only random access possible.

Almost Complete Binary Tree :



B.T. ✓

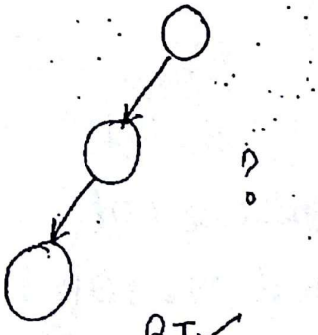
ACBT ✗



BT ✓

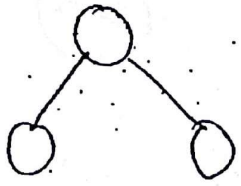
ACBT ✓

In-degree = 0 ⇒ Root



BT ✓

ACBT ✗



BT ✓

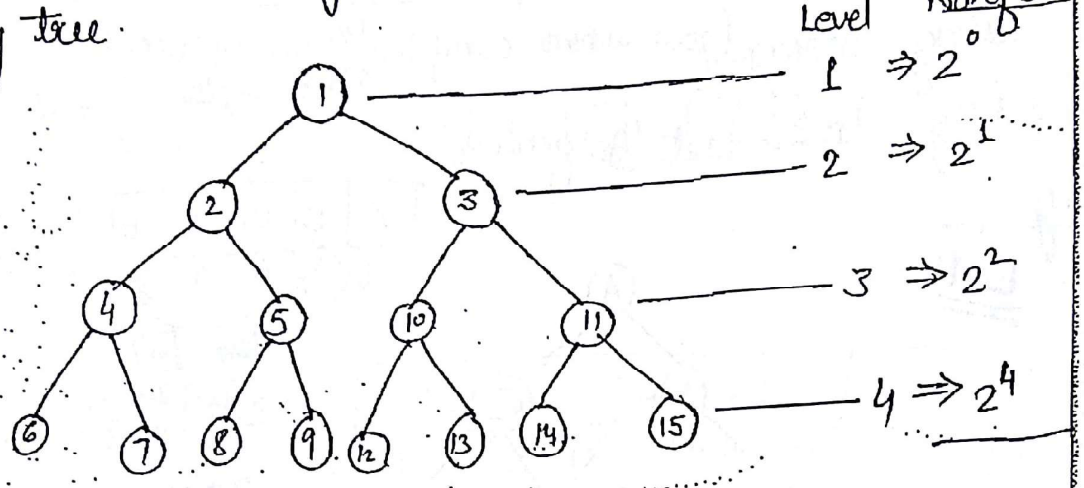
ACBT ✓

Defn :-

In the given Binary Tree, at every node :-

1) After completion of left only go to right.
2) After completion of current level, go to next.

- Every complete Binary tree is an almost complete Binary tree.



$2^0 + 2^1 + 2^2 + 2^4 = 1\left(\frac{2^4-1}{2-1}\right) = 2^4 - 1$

ACBT :

If no. of levels = k, Total nodes (max) = $2^k - 1$

min. nodes = 2^{k-1}

CBT

If no. of levels = k, Total nodes = $2^k - 1$

BT

If no. of levels = k, Max nodes = $2^k - 1$

Min nodes = k.

If CBT contains n-nodes,

Leaf Nodes = $\lceil \frac{n}{2} \rceil$, Non-leaf Node = $\lfloor \frac{n}{2} \rfloor$

If CBT contains n-nodes & k-levels

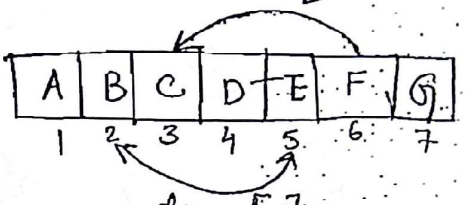
$$n = 2^k - 1.$$

$$\boxed{k = \log_2(n+1)}$$

Representations of Binary Tree

1. using array (good when complete Binary Tree).
2. using Linked List (in general)

almost complete $\frac{6}{2} = 3$



$\lceil n \rceil$

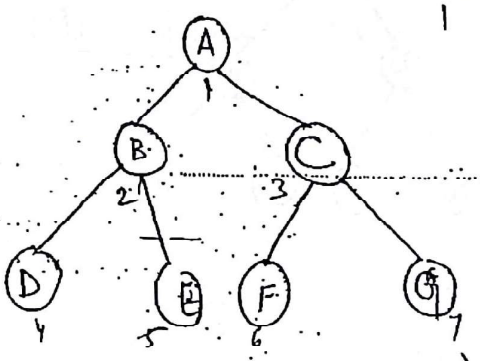
$= \lceil \frac{5}{2} \rceil = 2$ (Child)

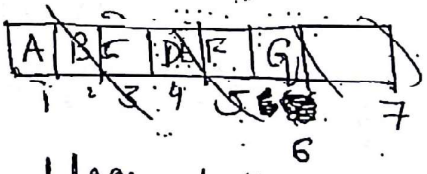
$2 * n + 1 = 2 * 2 + 1$

$= 5$ (Parent)

Array :-

Ex 1:-





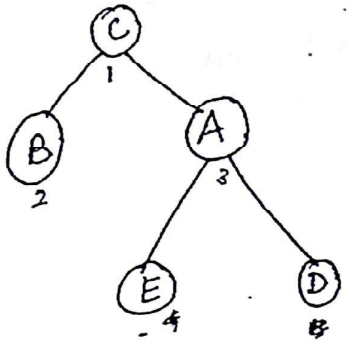
If a node is stored in i^{th} place of the array :-

(i) Parent $(i) = \lfloor i/2 \rfloor$

(ii) Left-Child $(i) = 2 * i$

Right-Child $(i) = 2 * i + 1$

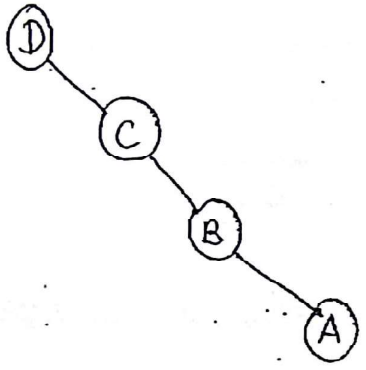
Ex 2:



C	B	A	E	D		
1	2	3	4	5		

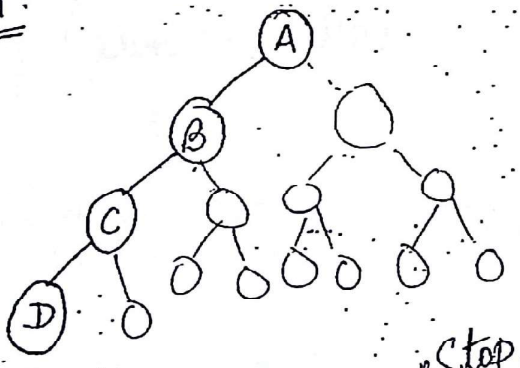
C	B	A			E	D
1	2	3	4	5	6	7

Ex 3:



D	-	C	-	-	-	B								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Ex 4:

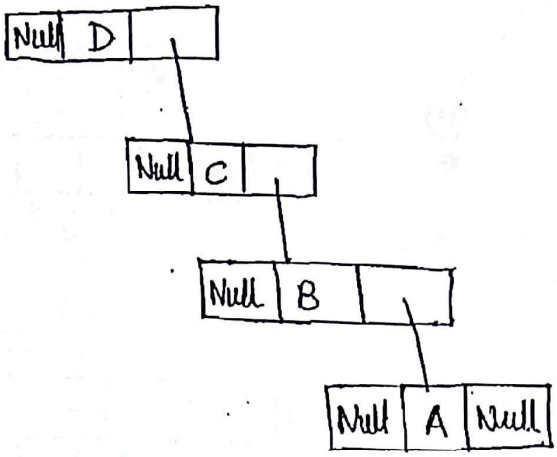


Stop here:

A	B	-	C	-	-	-	D							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A	B	-	C	-	-	-	D
1	2	3	4	5	6	7	9

Note: ① If the given binary tree is complete/almost complete binary tree, array is preferred.



② If Binary tree contains n-nodes, it is represented using array

min
n
(complete BT)

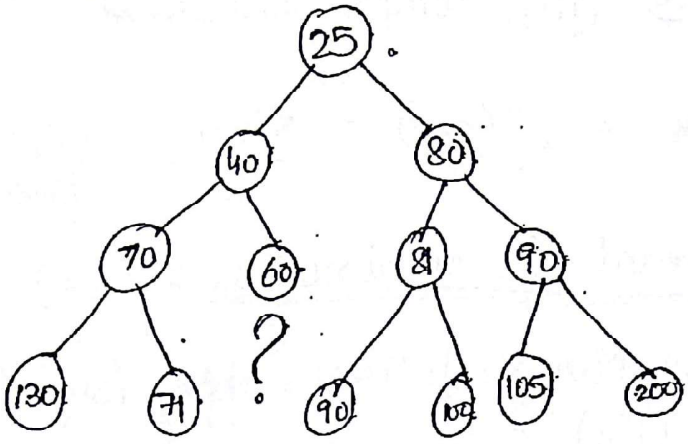
max
$2^n - 1$
(all right child)

Heap Tree

- Min Heap Tree
- Maxi Heap Tree

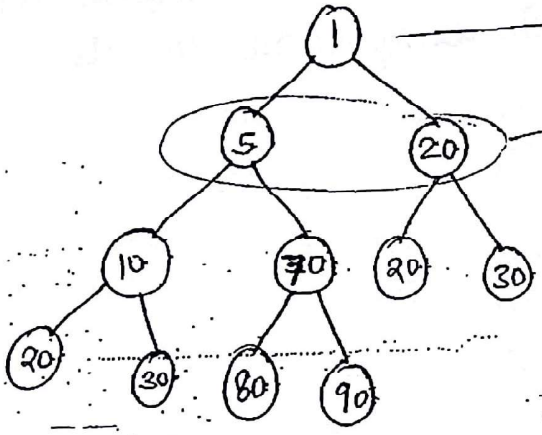
Min Heap Tree :- In the given almost complete Binary tree, root is minimum or equal comparing to its children, then it is called Min-Heap Tree.

EX:



< Not almost complete Binary Tree > ⇒ not Min Heap Tree

Ex 2:



ACBT ✓

Min Heap Tree ✓

If there are n nodes in a min Heap Tree, max/min levels = $2^n - 1$

1	5	20	10	70	20	30	20	30	80	90
1	2	3	4	5	6	7	8	9	10	11

- In a min Heap tree, the root contains the min element

So, to find min element ⇒ O(1) a[1]

1st min — 1 —— 0 comp. — O(1)

2nd min — 2 —— 1 comp — O(1)

3rd min — 3 —— 2 comp — O(1)

10th min — 9 comp — O(1)

To find nth (max) min $\Rightarrow$ (n-1) comparisons ~~of Eat~~

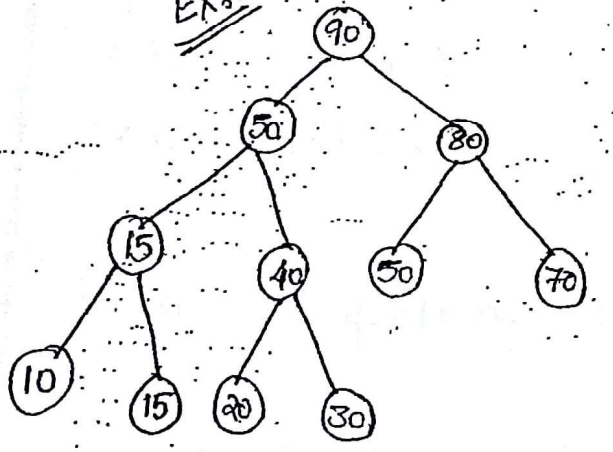
Also, total comparisons $= \sum(n-1) = \frac{n(n-1)}{2} = O(n^2)$ (Never Use)

Min Heap Tree is meant for minimums $\Rightarrow O(1)$

If max is reqd. from Min Heap Tree, apply Bubble Sort 1st pass $\Rightarrow O(n)$.

Max Heap Tree: In a given almost complete BT, root is greater/max or equal in comparison to its children.

Ex:



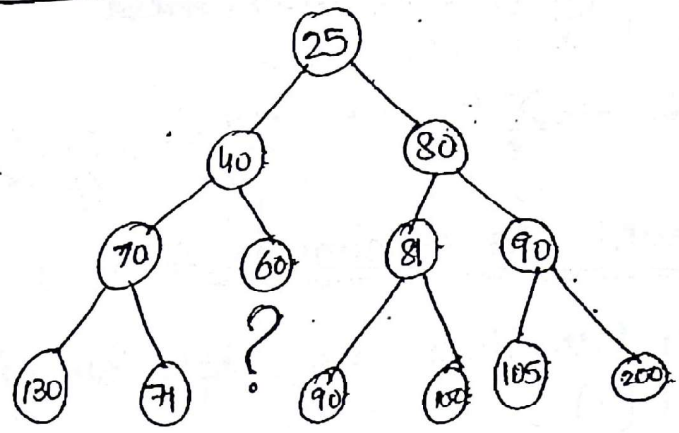
→ Finding max is $O(1)$. Finding min → no use, Selection Sort 1st pass $\Rightarrow O(n)$

Min Heapify Bottom Algo

Inserting an element in the min Heap tree.

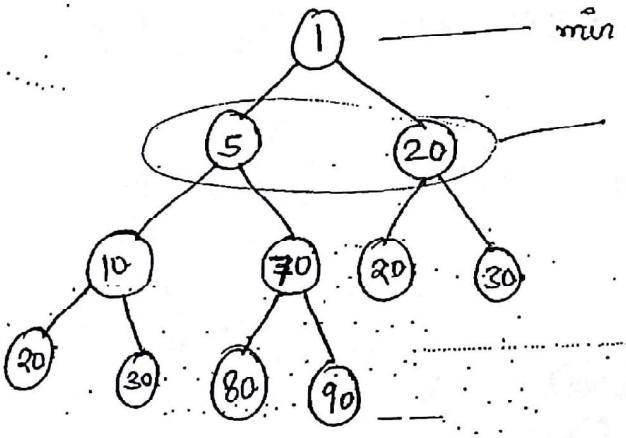
⇓

insert the element at the end of array and then make changes.



< Not almost complete Binary Tree > ⇒ not Min Heap Tree

Ex 2:



ACBT ✓

Min Heap Tree ✓

If there are n nodes in a min Heap Tree, max/min levels = $2^n - 1$

1	5	20	10	70	20	30	20	30	80	90
1	2	3	4	5	6	7	8	9	10	11

- In a min Heap tree, the root contains the min. element

So, to find min element ⇒ O(1) a[1]

1st min — 1 —— 0 comp. — O(1)

2nd min — 2 —— 1 comp — O(1)

3rd min — 3 —— 2 comp — O(1)

⋮

10th min — 9 comp — O(1)

To find nth min (max) $\Rightarrow$ (n-1) comparisons

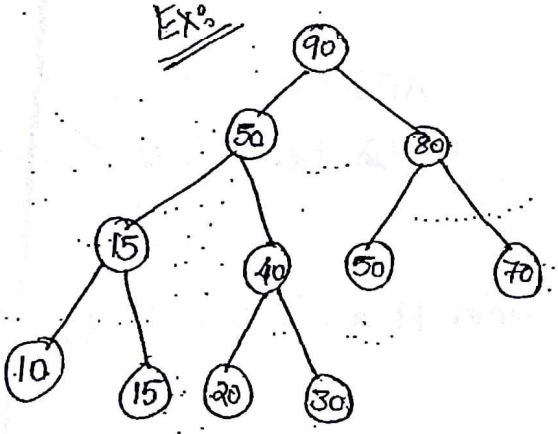
Also, total comparisons $= \sum_{i}(n-1) = \frac{n(n-1)}{2} = O(n^2)$ (Never Use)

Min Heap Tree is meant for minimums $\Rightarrow O(1)$

If max is reqd. from Min Heap Tree, apply Bubble Sort 1st pass $\Rightarrow O(n)$.

Max Heap Tree: In a given almost complete BT, root is greater/max or equal in comparison to its children.

Ex:



→ Finding max is O(1). Finding min → no use, Selection Sort 1st pass $\Rightarrow O(n)$.

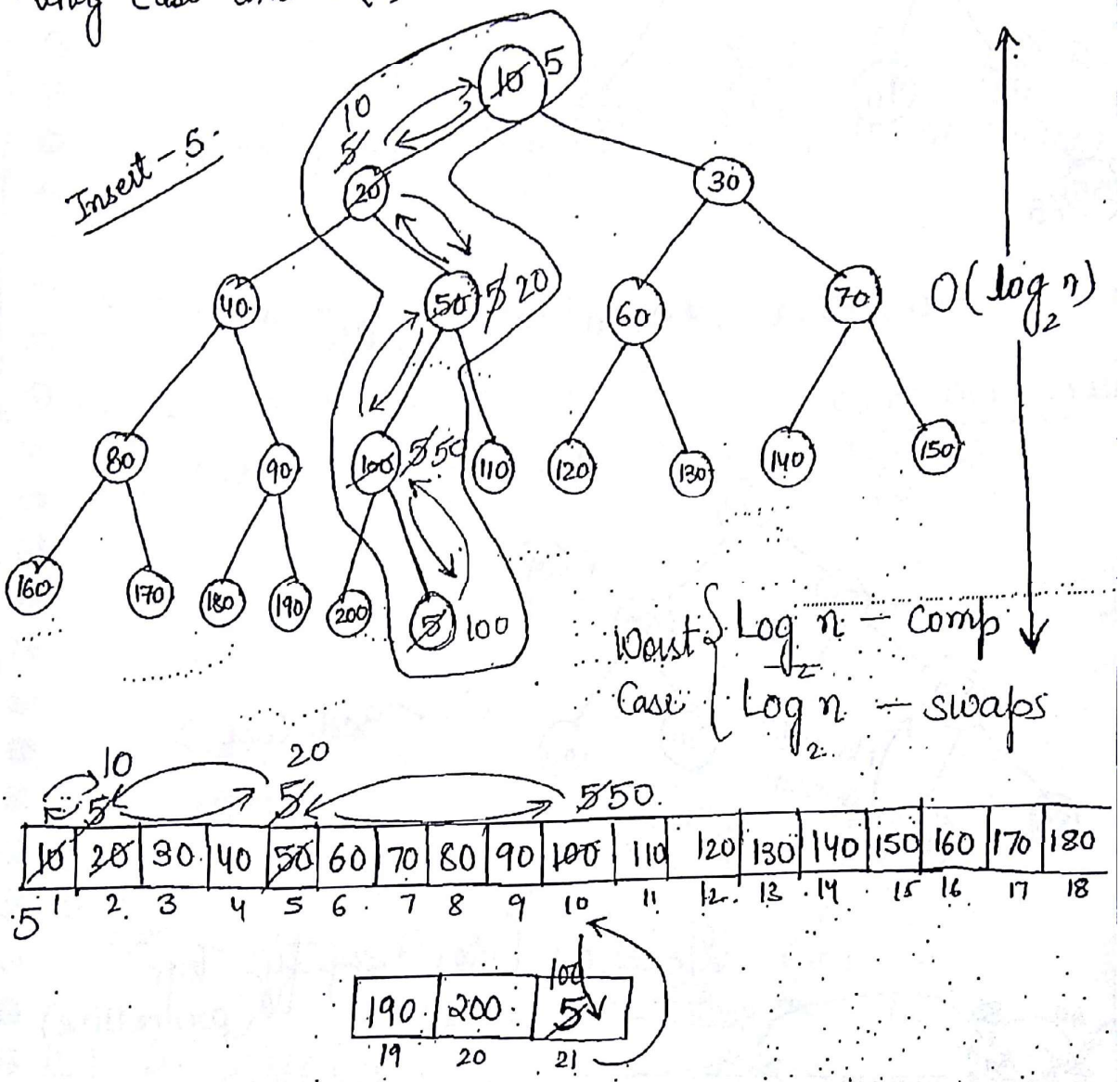
Min Heapify Bottom Algo

Inserting an element in the min Heap tree.

⇓

insert the element at the end of array and then make changes.

Inserting an element into min Heap / max Heap which already contains n-elements will take $O(\log n)$ – Worst/Avg Case and $O(1)$ – Best Case.

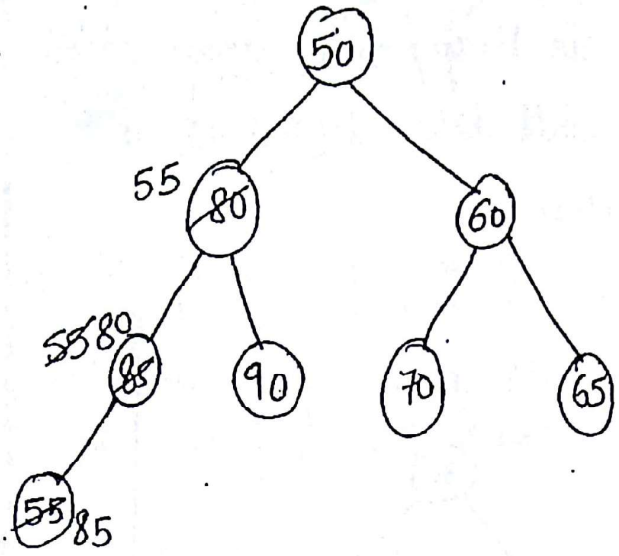


Inserting element :- $n = 20$

$n = 21$

$m = m + 1$

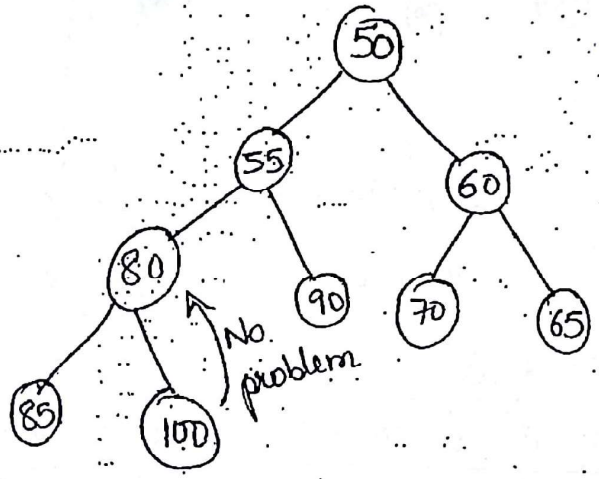
$a[m] = x$



Insert -55

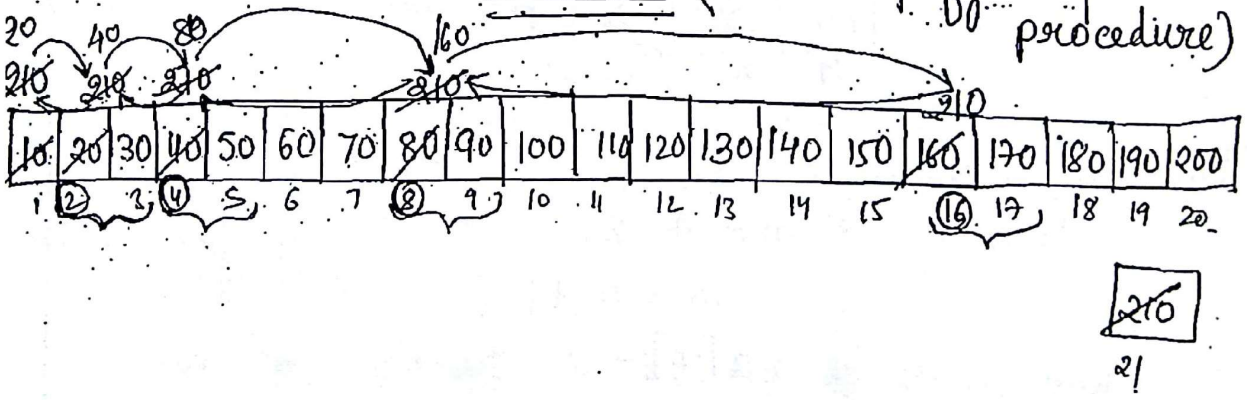
50, 55, 60, 80, 90, 70, 65, 85

Insert - 100 now



< Best Case >

Deletion (Min Heapify Top procedure)



Deletion means delete the minimum element ⇒ root and it will be replaced by last element

- Heap Sort is not stable as

Ex 3.

10_a	10_b	10_c	10_d	10_e	10_f	10_g

Build - Heap Method

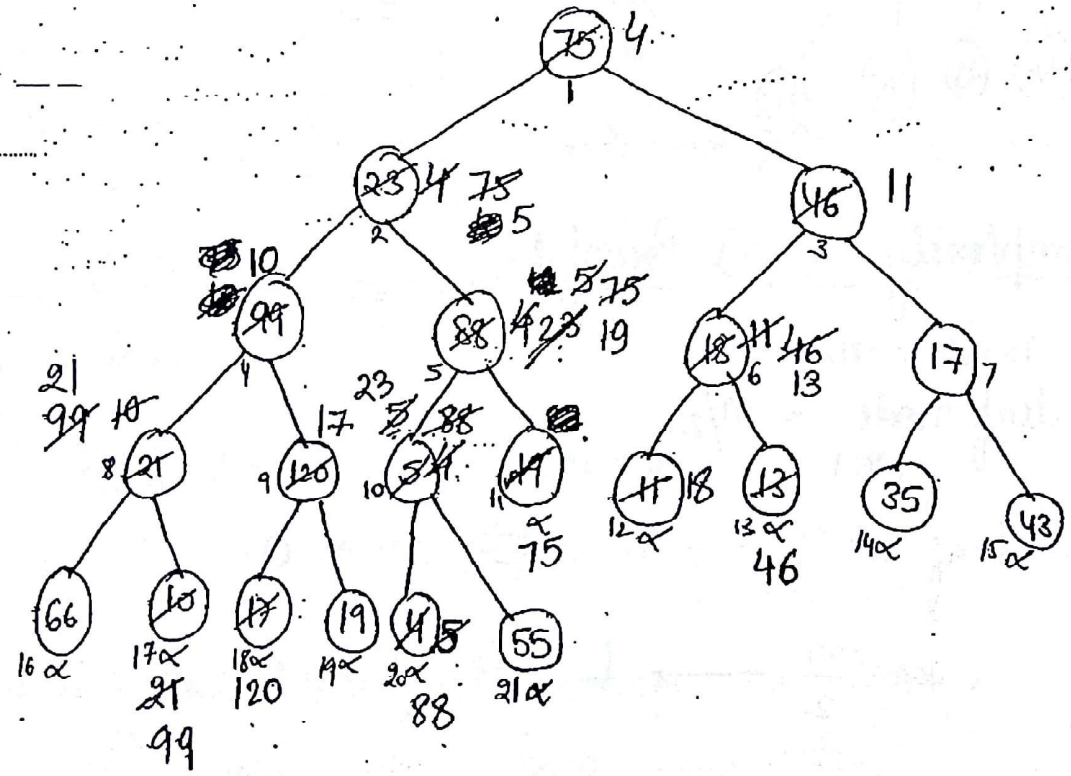
(constructing max-heap / min-heap with O(n) time)

Ex 1.

75 23 46 99 88 18 17 21 120 5 19 11 13 35

43 66 10 17 19 4 55

For loop in reverse order & minheapify Top.



4

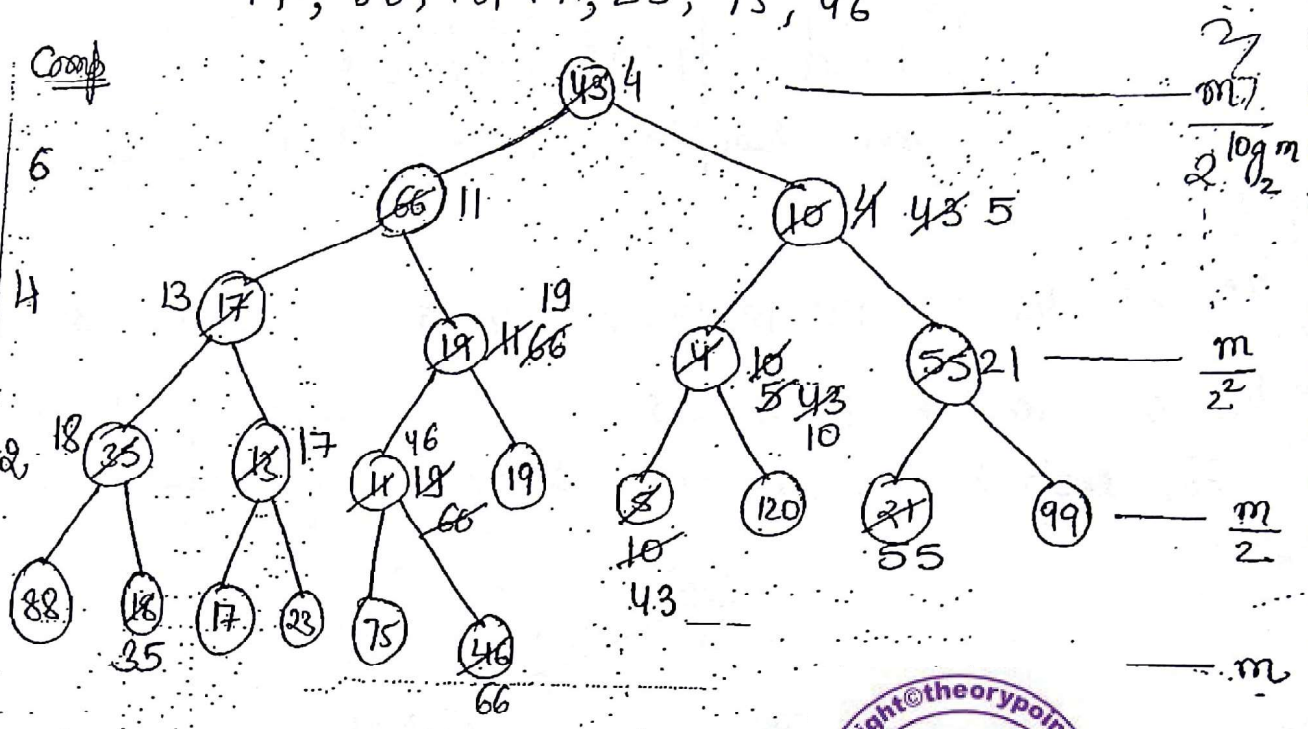
5 11

10 A 13 17

21 17 23 75 18 46

Ex2:

43, 66, 10, 17, 19, 4, 55, 35, 13, 11, 19, 5, 120, 21, 99, 88, 18, 17, 23, 75, 46



Time Complexity O(n) Proof:

Total nodes = n

leaf nodes (m) = n/2

Swaps: leaf nodes $_m$ — 0! $\Rightarrow m * 0$.

$\frac{m}{2}$ — 1 $\Rightarrow \frac{m}{2} * 1$.

$\frac{m}{2^{\log_2 m}}$ — $\log_2 m \Rightarrow \frac{m}{2^{\log_2 m}} * \log_2 m$

$$\text{Swaps} = m\Big[\underbrace{0 + 1\cdot\frac{1}{2} + 2\left(\frac{1}{2}\right)^2 + \cdots\cdots \log_2 m\left(\frac{1}{2}\right)^{\log_2 m}}_{AGP.}\Big]$$

$$= m\left[\left(\frac{1}{2}\right) + \left(\frac{1}{2}\right)^2 + \cdots\cdots + \left(\frac{1}{2}\right)^{\log_2 m}\right]$$

$$= m \cdot O(1).$$

Comparisons $\Rightarrow$ 2 * Swaps.

$= 2m$

Time complexity (Build Heap) $= m + 2m$

$= 3m$

$= \frac{3n}{2} = O(n)$

Heap Sort Algorithm :-

(1) Using build heap, create max-heap $\Longrightarrow O(n)$

(2) Delete one-by-one and store from R to L $\Longrightarrow n \log n$

Total time $= n + n\log n$

$= O(n \log n)$ [Every Case]

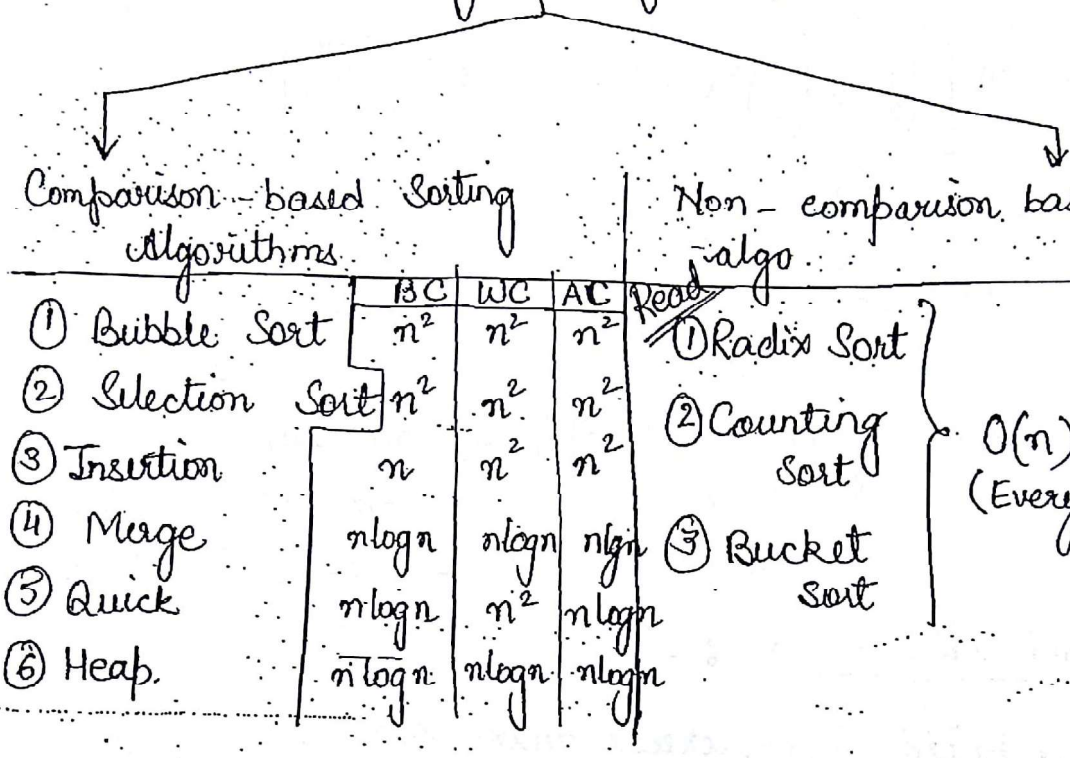
Special Case : When all elements are same,

(1) create max-heap $\longrightarrow O(n)$

(2) Deleting $\longrightarrow O(1)$.

Total time $= O(n)$

Sorting Algorithms



Comparison - based Sorting Algorithms	BC	WC	AC
① Bubble Sort	n^2	n^2	n^2
② Selection Sort	n^2	n^2	n^2
③ Insertion	n	n^2	n^2
④ Merge	$n\log n$	$n\log n$	$n\log n$
⑤ Quick	$n\log n$	n^2	$n\log n$
⑥ Heap	$n\log n$	$n\log n$	$n\log n$

Non - comparison based Sorting algo

Read

① Radix Sort
② Counting Sort
③ Bucket Sort

$O(n)$ (Every Case)

- In all comparison based sorting algo, upper bounds (worst cases), lower bound is $n\log n$.
- In all comparison based sorting algo lower bounds (best cases), upper bound is n^2.

Greedy Technique (Shortcut)

Note: Most of the problems in greedy contains 'n' i/ps. and our objective is finding a subset which will satisfy our conditions and optimizes our goal.

Basics:

1) Solution Space :- The set of all possible solutions over the given 'n' no. of i/ps is called Solution Space.

2) Feasible Solution :- The set of all solutions which will satisfy our conditions is called feasible solution.

3) Optimal Solution :- The set of all feasible solutions which will optimize our goal is called optimal soln. (need not be unique).

Ex: Finding Top 10 students ~~from~~ in a class of 150 students.

① Solution Space :-

$10_1, 10_2, 10_3, \ldots\ldots {}^{150}C_{10}$

② Feasible Soln :- (more than 85% marks → 50 students)

$10_1, 10_2, 10_3, \ldots\ldots {}^{50}C_{10}$

③ Optimal Soln :-

10_1.

Applications of Greedy Technique :-

1) Job Sequencing with deadline.
2) Knapsack problem
3) Huffman coding
4) Optimal merge pattern
5) Minimum Cost Spanning Tree (MCST)
   - Prims algo.
   - Kruskal algo.
6) Single source shortest path
   - Dijkstra's algo
   - Bellman Ford algo
   - Breadth First Traversal (BFT)

Job Sequencing with Deadline

(i) single CPU
(ii) no pre emption
(iii) one unit running time for each job
(iv) arrival time of every job is same.

Ques: $n = 4$

Jobs	J_1	J_2	J_3	J_4
Profits	150	200	250	175
Deadline	2	1	1	2

(J_1, J_2) ✗ (J_1, J_3) ✗ (J_1, J_4) ✓ = 325

✓ $(J_2, J_1) = 350$, (J_2, J_3) ✗ ✓ $(J_2, J_4) = 375$

✓ $(J_3, J_1) = 400$, (J_3, J_2) ✗ , ✓ $\boxed{(J_3, J_4) = 425}$ (Optimal)

✓ $(J_4, J_1) = 325$, (J_4, J_2) ✗ , (J_4, J_3) ✗

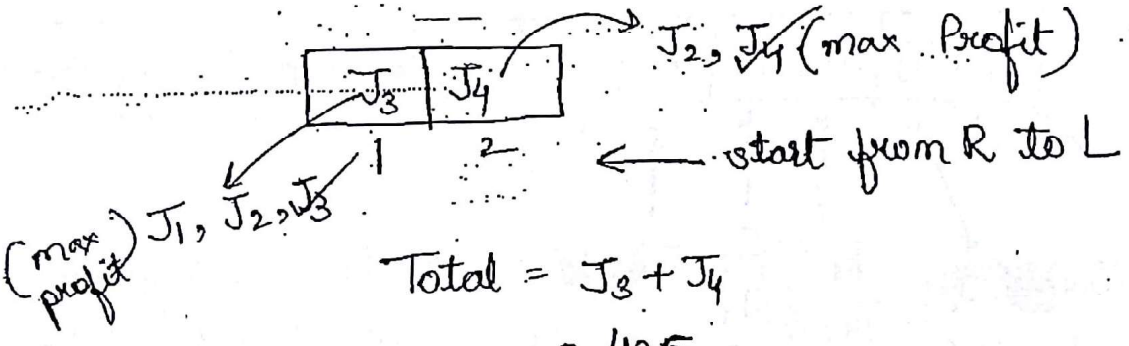
$(J_1) = 150$, $(J_2) = 200$, $(J_3) = 250$, $(J_4) = 175$.

$( ) = 0$ {Don't do anything}

Total feasible solns = 11.

Shortcut:

Take max deadline :- (Here, 2)

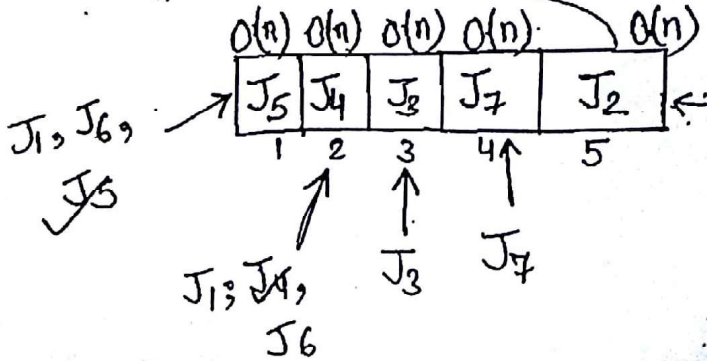


Total = $J_3 + J_4$

= 425.

Ex2 n = 7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	75	25	55	100	120	60	50
Deadlines	2	5	3	2	1	2	4



$\Rightarrow 120 + 100 + 55 + 50 + 25$

$\Rightarrow 270 + 80 = \underline{350}$

Jobs left out = J_1, J_6.

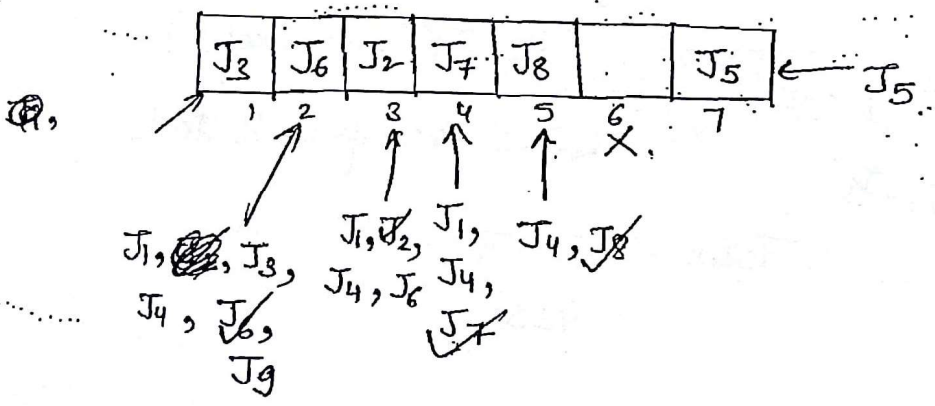
Penalty = $J_1 + J_6$

$= 60 + 75 = \underline{135}$.

∴ Every slot linear search applied ⇒ Time complexity $= O(n^2)$

Ex3: $n = 9$

Jobs	J_1	✓ J_2	✓ J_3	J_4	✓ J_5	✓ J_6	✓ J_7	✓ J_8	J_9
Profit	25	75	55	20	30	60	90	55	45
Deadline	4	3	2	5	7	3	4	5	2



⇒ 75 + 55 + 30 + 60 + 90 + 55

⇒ 200 + 90 + 75

= 365

110
90
200

Method 2:

① Arrange all jobs in descending order of profit

J_7	J_2	J_6	J_3	J_8	J_9	J_5	J_1	J_4
90	75	60	55	55	45	30	25	20
1	3	3	2	5	2	7	4	5

Sorting time complexity = $n \log n$ (Merge sort)

②

J_3	J_6	J_2	J_7	J_8	—	J_5
1	2	3	4	5	6	7

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

$\Rightarrow$ Time complexity = $n^2 + n \log n$

$= O(n^2)$ {Worst Case}.

Best Case :- $n + n \log n$

$= O(n \log n)$.

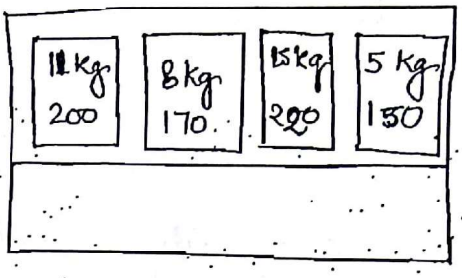
If answer does not match, check if any jobs can swap like (J_3 & J_6 can swap).

Binary Search not possible $\Rightarrow$ searching done using deadline & sorting done using profits.

18/12/16

Knapsack

M = 25.



Obj 1

11 — 200

1 — $\frac{200}{11}$ = 18.1

Obj 2

8 — 170

1 — $\frac{170}{8}$ = 21.25

Obj 3

15 — 200

1 — $\frac{200}{15}$ = 14.6

Obj 4

5 — 150

1 — $\frac{150}{5}$ = 30

(x_1 x_2 x_3 x_4)

x_1 1 * 200

x_2 1 * 170

x_3 $\frac{1}{15} * 200$

x_4 1 * 150

25

x_4 25 − 5 = 20.

x_2 20 − 8 = 12

x_1 12 − 11 = 1

x_3 1 − 1 = 0

Profit = 200 + 170 + 150 + 14.6 = 534.6

Greedy Knapsack will give always max profit by giving priority to both weight & profit.

Ex 2 $M = 27$ $n = 7$

Objects	1	2	3	4	5	6	7
Profit	25	85	50	30	20	55	45
Weight	2	10	5	4	3	5	6

Obj 1 ✓

$2 \text{ —— } 25$

$1 \text{ —— } \frac{25}{2} = 12.5$

3 ✓

$5 \text{ —— } 50$

$1 \text{ —— } \frac{50}{5} = 10$

2 ✓

$10 \text{ —— } 85$

$1 \text{ —— } \frac{85}{10} = 8.5$

4 ✓

$4 \text{ —— } 30$

$1 \text{ —— } \frac{30}{4} = 7.5$

5

$3 \text{ —— } 20$

$1 \text{ —— } \frac{20}{3} = 6.66$

6 ✓

$5 \text{ —— } 55$

$1 \text{ —— } 11$

7

$6 \text{ —— } 48$

$1 \text{ —— } \frac{45}{6} = 7.5$

$27 - 2 = 25$

$25 - 5 = 20$

$20 - 5 = 15$

$15 - 10 = 5$

$5 - 4 = 1$

$1 - 1 = 0$

(— — — — — — —)

$1 * 25 = 25$

$1 * 85 = 85$

$1 * 50 = 50$

$1 * 30$

$1 * 55 = 55$

$\frac{1}{6} * 45$

7.5
55
80
25
85
252.5

$= \boxed{252.5}$

Step 1 Calculate Profit/weight $\longrightarrow O(n)$

Step 2 Sort the ratio in decreasing order $\longrightarrow O(n \log n)$

Step 3: Evaluate profit & updated weight for every object until limit finishes $\Longrightarrow O(n)$

$n + n\log n + n$

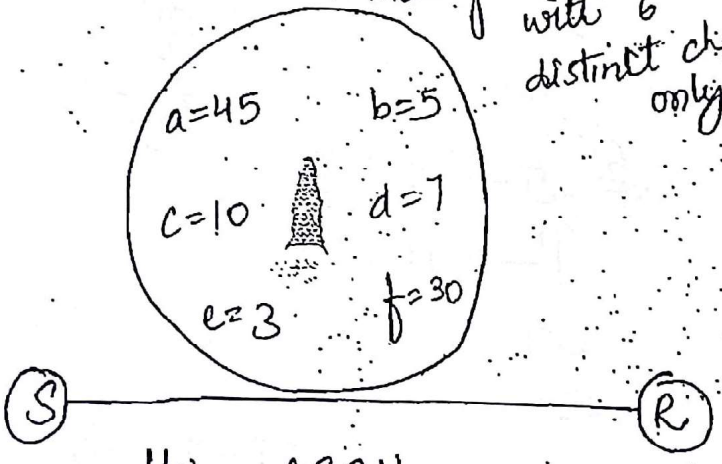
$= O(n \log n)$

#Infinite feasible solutions exist

Huffman Coding (one of the data compression/ encoding technique)

Ex1:

message = 100 char with 6 distinct characters only



Using ASCII.

(97) a — 45 * 7

b — 5 * 7

c — 10 * 7

d — 7 * 7

e — 3 * 7

f — 30 * 7

100 char 100 * 7 = 700 bits

1 char — 7 bits/char

a — 000

b — 001

c — 010

d — 011

e — 100

f — 101

Using 3-bits

a — 45 * 3

b — 5 * 3

c — 10 * 3

d — 7 * 3

e — 3 * 3

f — 30 * 3

100 char — 300 bits

1 char — 3 bits/char

Huffman Coding

⇓

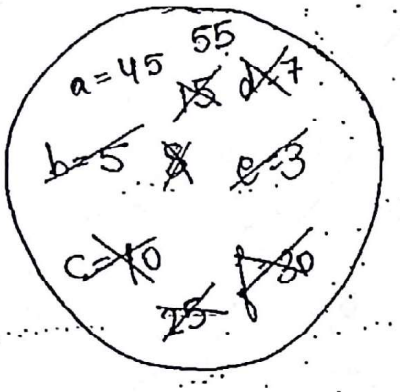
Non-uniform coding

⇓

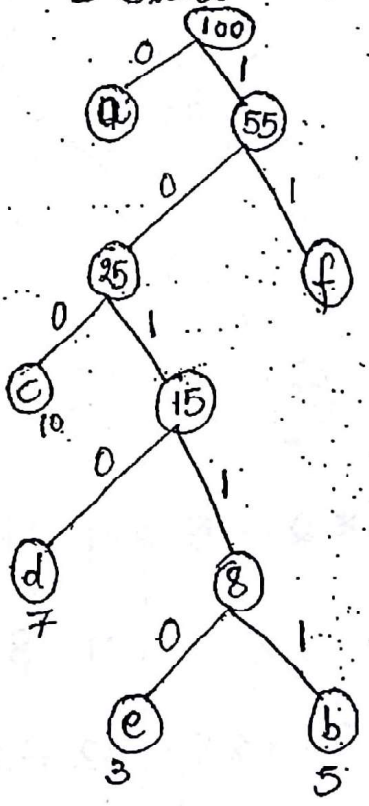
more freq ⇒ Less bits

⇓

Less freq ⇒ more bits



- start with taking 2 min.



(1) Huffman Code :-

a - 0

b - 10 111

c - 100

d - 1010

e - 10110

f - 11

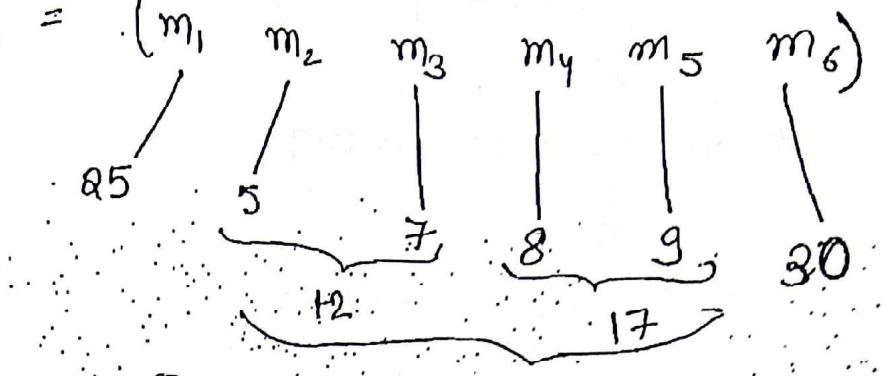
(2) Total Bits = $1*45+5*5+3*10+4*7+5*3+2*30$

$= 45+25+30+28+15+60$

$= 203$ Bits $= 2.03$ bits/char

Ex 2:

Message M = $(m_1\ m_2\ m_3\ m_4\ m_5\ m_6)$



Assume left-0 right-1

(1) m_1 — 10

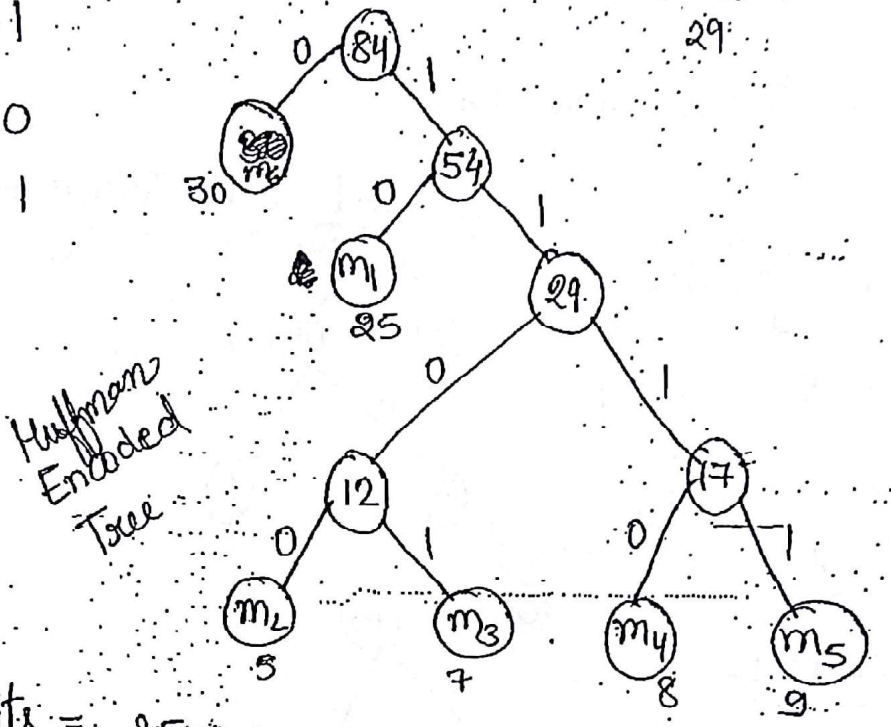
m_2 — 1100

m_3 — 1101

m_4 — 1110

m_5 — 1111

m_6 — 0



Huffman Embedded Tree

25
29
54

(2) Total bits = $25*2 + 5*4 + 7*4 + 8*4 + 9*4 + 30*1$

$= 50 + 20 + 28 + 32 + 36 + 30$

$= 196$ bits

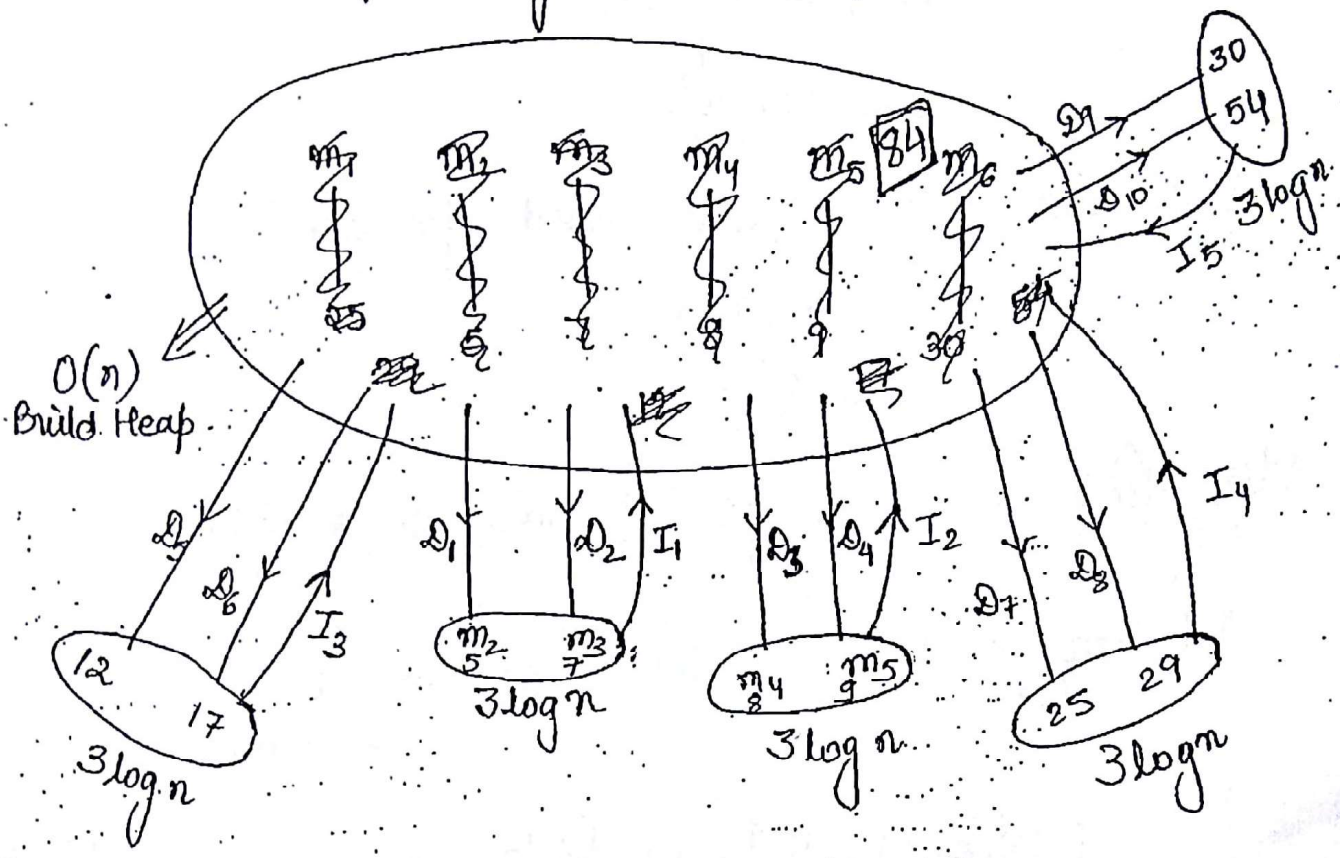
$\Rightarrow$ 84 char ——— 196 Bits

1 char ——— $\frac{196}{84}$ 2.2 Bits/char

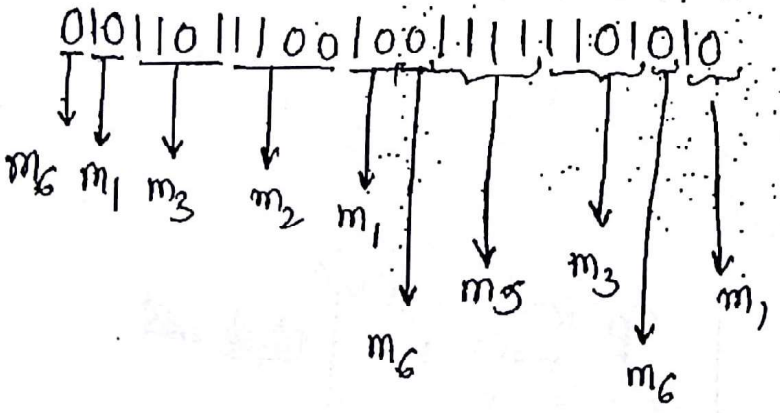
84
84
2
168

196
168
28

Since at every stage, 2 minimums are reqd. So, create min Heap using Build Heap method $O(n)$.



(3) Encoded message:



Decoded message: $m_6\ m_1\ m_3\ m_2\ m_1\ m_6\ m_5\ m_3\ m_6\ m_1$

Time Complexity (Huffman Coding) $= n + \underbrace{(n-1).3\log n}_{\text{insert 1 \& delete 2}} \Rightarrow \underline{O(n \log n)}$

n: min Heap

If merge sort used instead of min heap =

$$n + n\log n \cdot n$$

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

If selection sort 2 passes used for 2 min

$$= n + 2n \cdot n$$

$$= O(n^2)$$

Min Heap is worst for 1st min $\Rightarrow n + \log n$

$$= O(n)$$

Selection sort for 1st min $\Rightarrow n$

$$= O(n)$$

Optimal Merge Pattern

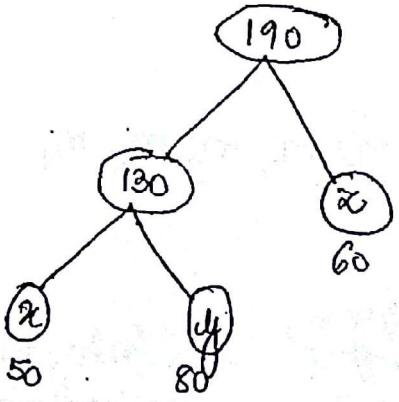
Ex 1 $n = 3$ (files)

$x = 50$ records

$y = 80$ "

$z = 60$ "

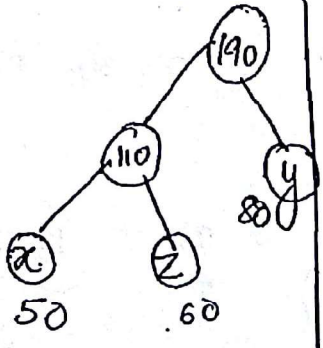
<2-way merge pattern>



Total record movements $= 130 + 190$

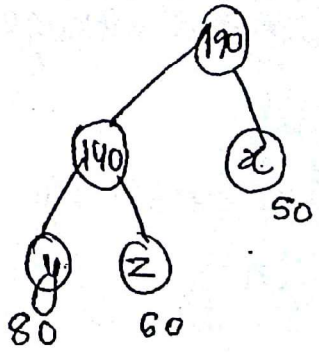
$= 320$

Optimal merge pattern



$= 110 + 190$

$= 300$ ✓



$= 140 + 190$

Ex 2. $n = 7$

~~A — 13~~

~~B — 10~~

~~C — 7~~

~~D — 14~~

~~E — 8~~

~~F — 20~~

~~G — 11~~

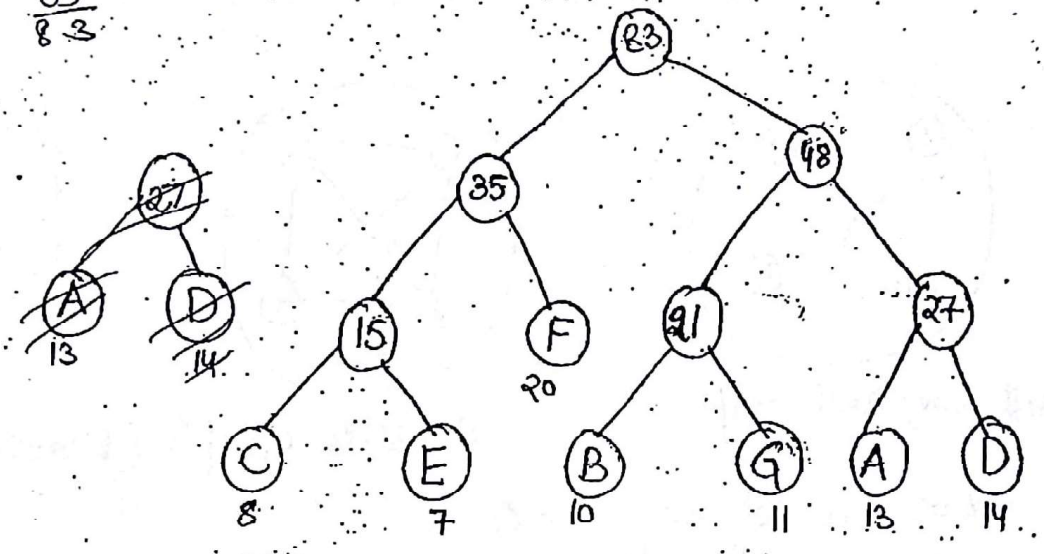
~~15~~

21

27

35

1
48
35
83



Total merging = $15 + 21 + 27 + 35 + 48 + 83$ (2)

= 229 records.

Time complexity (OMP) = $\underbrace{n}_{\text{Build Heap}} + \underbrace{(n-1) \cdot 3 \log n}_{\text{2 Deletion \& 1 Insertion (same as prev. algo)}}$

$= n + n \log n$

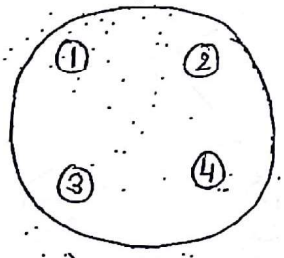
$= O(n \log n)$

Minimum Cost Spanning Tree

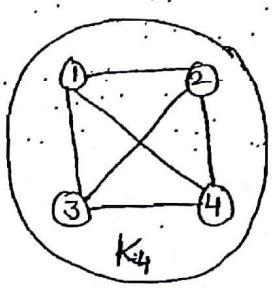
Note: Simple graph with 'n' vertices,

max degree = n-1

min degree = 0



(Best case) null graph



Complete Graph (Worst Case)

min no. of edges = 0

max no. of edges = $\frac{n(n-1)}{2}$

Let G(V, E) be a simple graph,

$$E \leq \frac{V(V-1)}{2}$$

$$E \leq c \cdot V^2$$

$$E = O(V^2)$$

$$\log E = O(\log V)$$

Connected (undirected graph): there is a path b/w every 2 vertices.

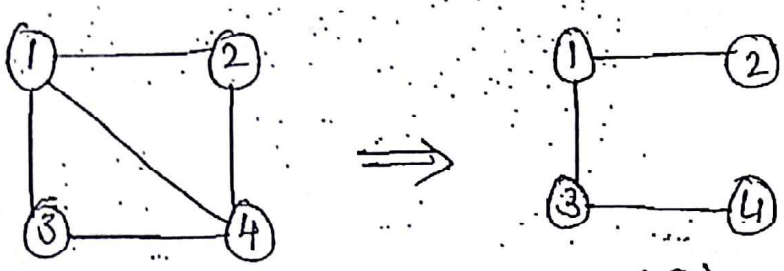
Strongly connected (directed graph): there is a directed path b/w every 2 vertices.

If strongly connected fails then remove directions & check if it is connected. If connected → weakly connected.

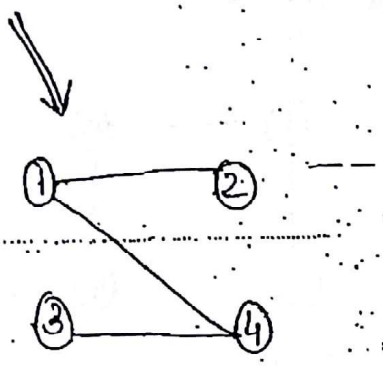
Spanning Tree : A subgraph S of given graph G is said to be spanning tree iff

(i) S should contain all vertices of G.

(ii) S should contain (V-1) edges without any cycle where V is no. of vertices



(G) (S)

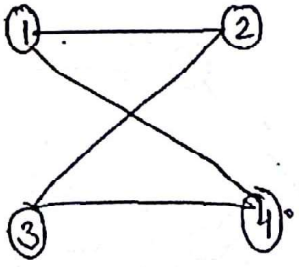


Tree :- not necessary to cover all vertices.

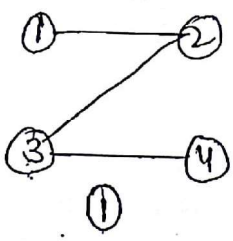
- No Cycle
- Connected

Every spanning tree is a tree but every tree need not be a spanning tree.

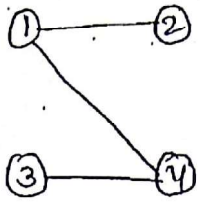
Ex1: Consider the foll. graph :-



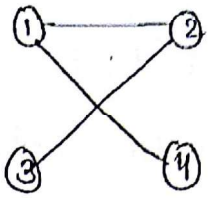
Find total no. of spanning trees possible?



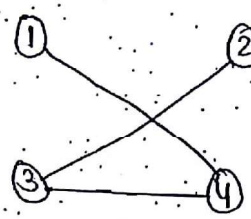
(1)



(2)



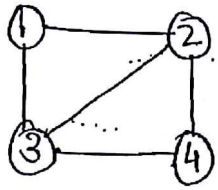
(3)

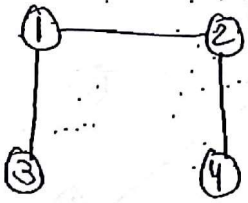


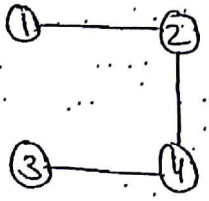
(4)

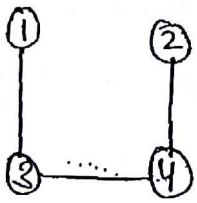
Ex2: Consider fall graph :-

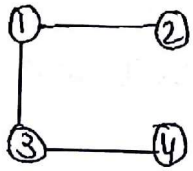
Q6

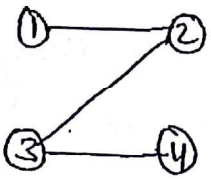


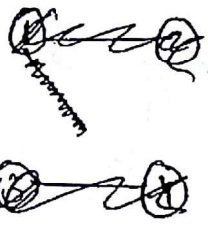


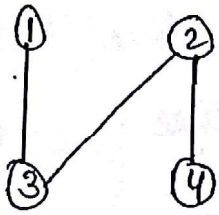


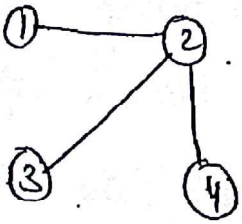


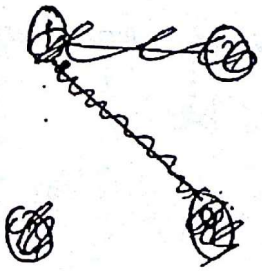


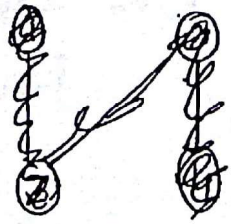


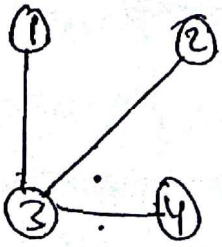










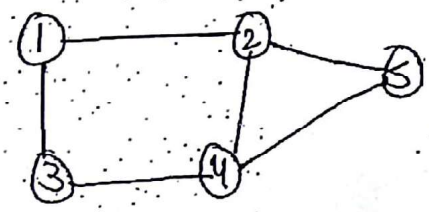


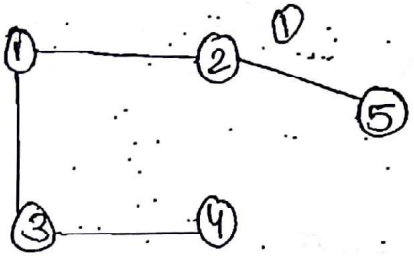
The number of spanning trees in K_n = n^{n-2}

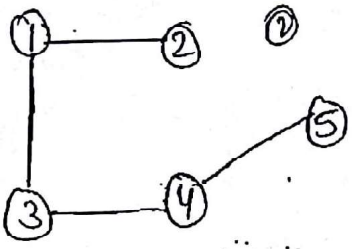
(K_n : Complete Graph)

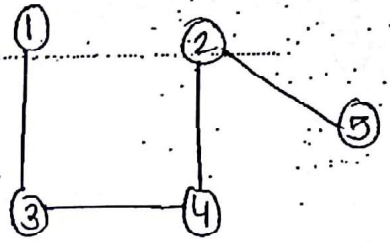
Ex: Consider foll. graph

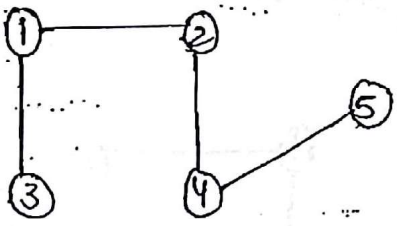
Find Spanning trees?

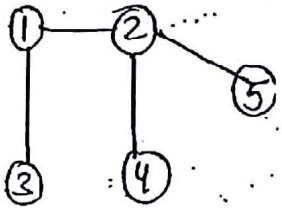


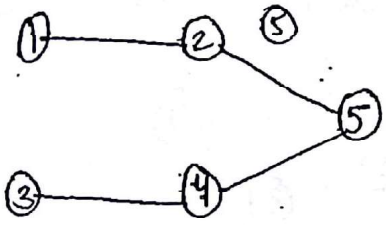


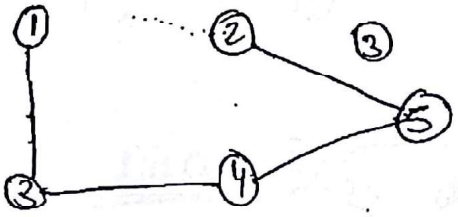


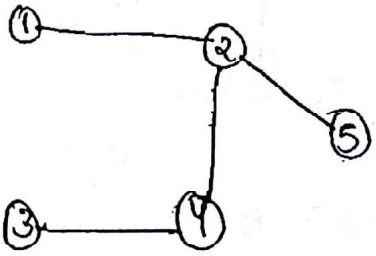


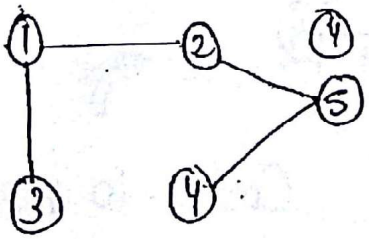


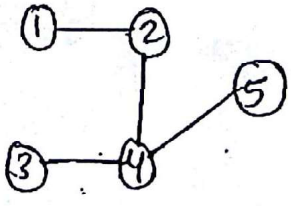




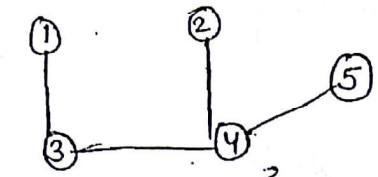








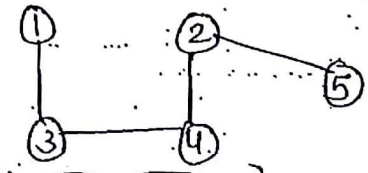
Take cycle :-



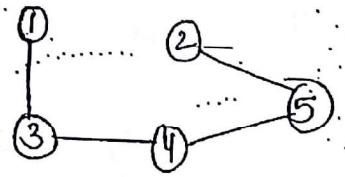
rotate = 4

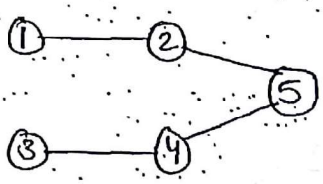
Total = 4+4+3

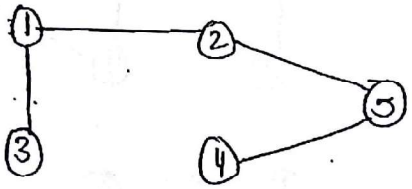
= 11



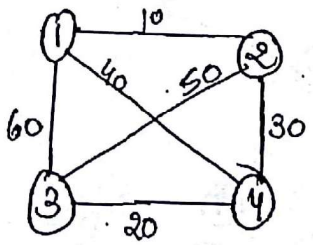
Rotate = 4

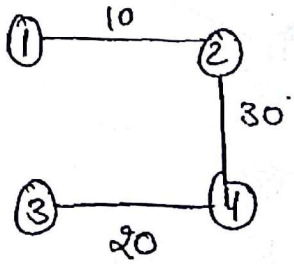






Ex:





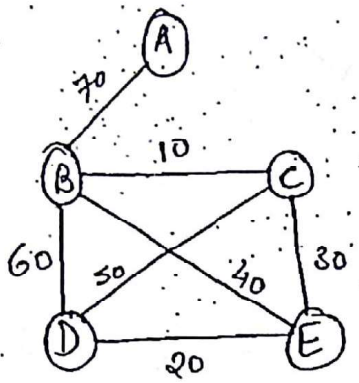
(MST)

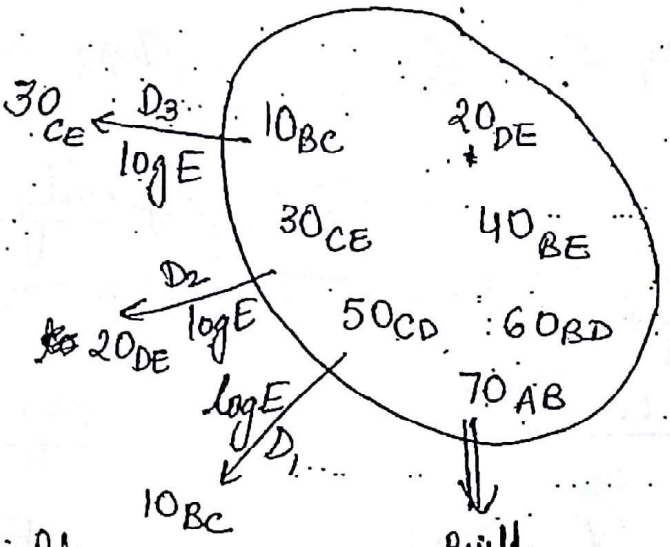
Cost = 10 + 30 + 20 = 60.

Using Prims and Kruskal Algo, we can find out min. cost spanning tree directly.

Kruskal Algo :-

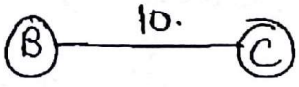
Ex:



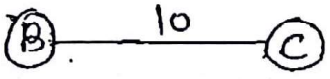


(1) Take min edge weight & add to MST

$10_{BC} \rightarrow \log E$

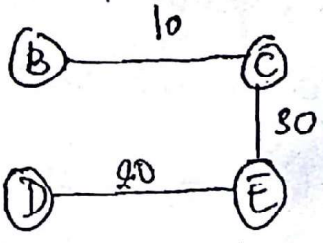


(2) Take next min edge weight & add to MST

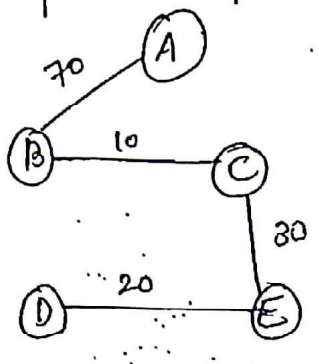


Disconnected

(3) Take next min edge weight & add to MST if no cycle.



④ Repeat step 3 -



No Cycle

40_{BE} ✗ $\log E$

50_{CD} ✗ "

60_{BD} ✗ "

70_{AB} ✓ "

Time complexity

Best Case

$\Omega(E)$ + $(V-1)\log E$

$\Omega(E)$ — Build Heap

$\Rightarrow E + V\log E$

$\Rightarrow E + V\log V$

$= O(E + V\log V)$

If complete graph $E = V^2$ → $O(E)$

If null graph $E = 0$ → $O(V\log V)$

Avg / Worst Case

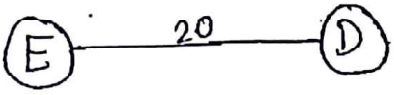
$E + E\log E$

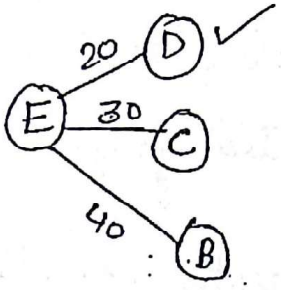
$= E + E\log V$

$= O(E\log V)$

Prims Algo :- in all steps, graph remains connected as adjacent vertices and their edges are considered.

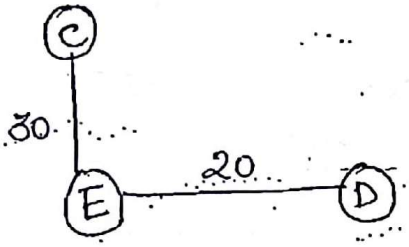
① Take any vertex and find adj of that vertex and take min from this.

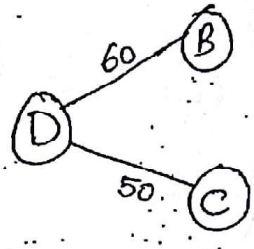


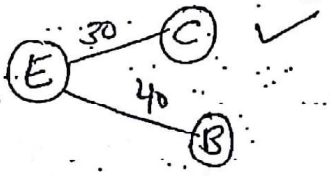


To check adj vertex : V
To check min. : V
2V = O(V)

② Find adj of new vertex and take min of these and prev.

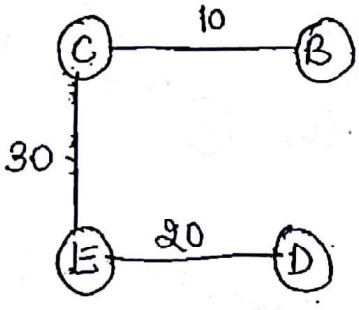


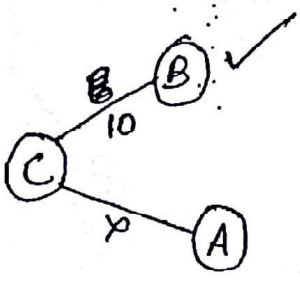




⇒ O(V).

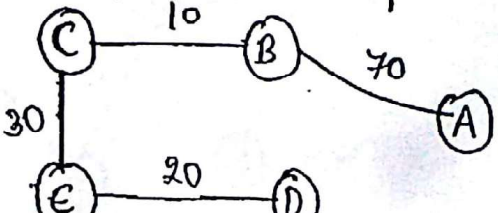
③ Find adj of new vertex and take min of these and prev and if no cycle.

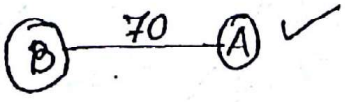




⇒ O(V)

④ Repeat previous step

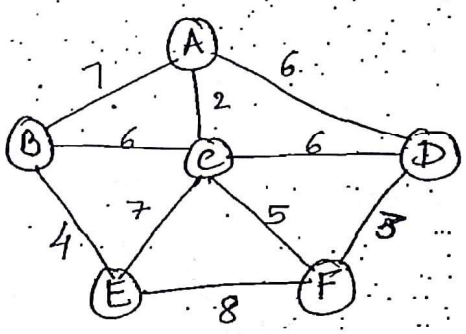




Cost = 130

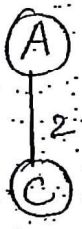
NOTE: Prims algorithm will always generate connected graph but Kruskal algo in the middle may generate disconnected graph.

Ex2: Consider foll. graph:-

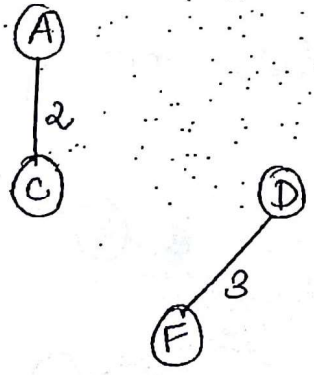


Kruskal Algo:-

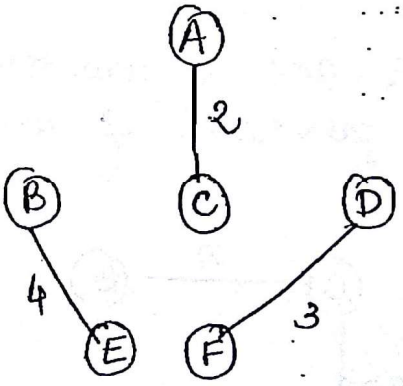
(1)



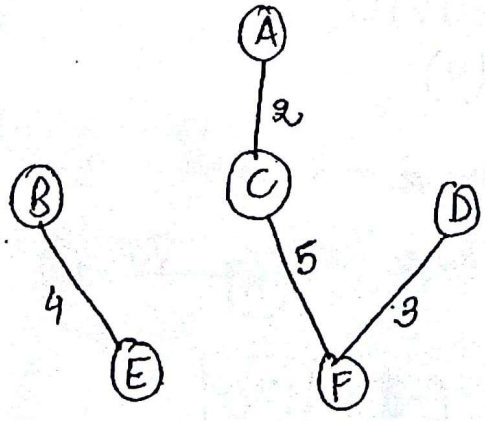
(2)



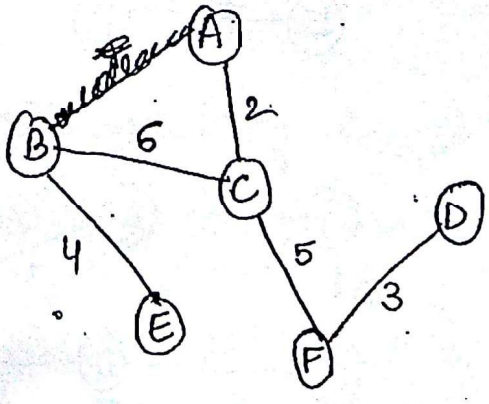
(3)



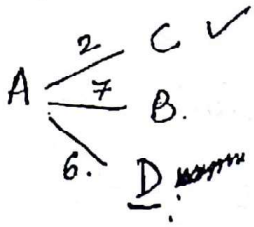
(4)



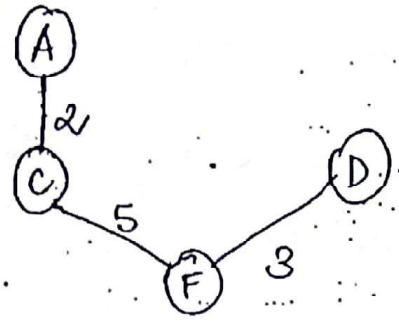
(5)

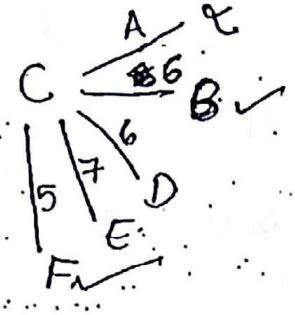


Prims Algo :- (1)

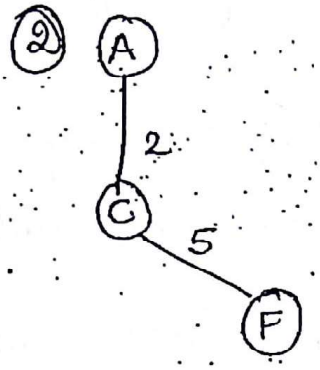


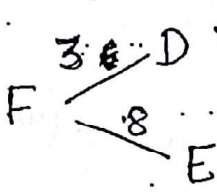
(3)



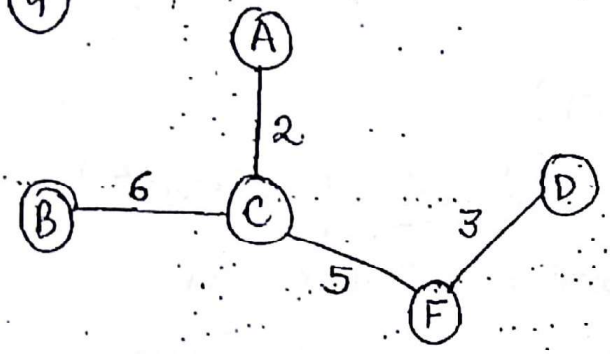


(2)

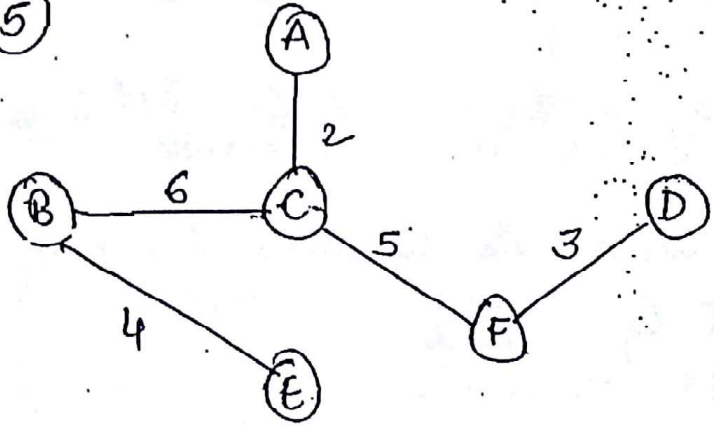


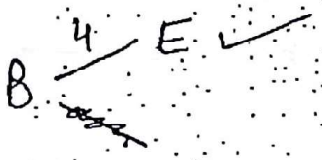


(4)



(5)





Minimum Cost = 2+3+5+6+4

= 20

In prims algo, check for adjacency first in all options, then check min property in options which satisfy adjacency.

Ex 3:

Let $G(V, E)$ be a UWCG with distinct edge weights

e_{max} :- edge with max edge weight

e_{min} :- edge with min " "

T/F ?

a) G contains UMST (unique MST) — True {distinct edge weights}

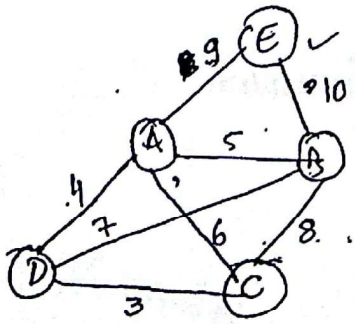
b) Every MST of G must contain e_{max} — False

c) ——" —— may —— " —— True

d) ——" —— must ——" — e_{min} True

e) ——" —— may contain e_{min} False

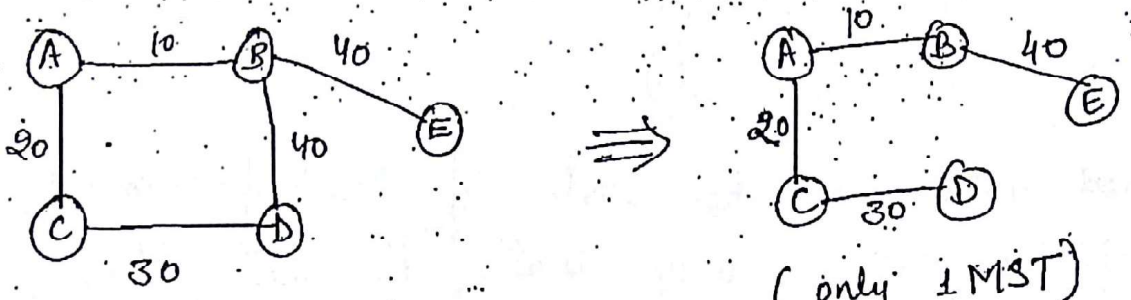
f) If e_{max} is in MST then its removal from G must disconnect G. True



For a given graph, if more than 1 MST exists, then repetition of weights must be there.

For a given graph, if repetition of weights exists then graph may or may not have more than 1 MST.

Ex:



(only 1 MST)

Ex4.

Let G be a UWCG with n-vertices & w be the minimum edge weight among all edge weights & e be a specific edge with weight w.

T/F

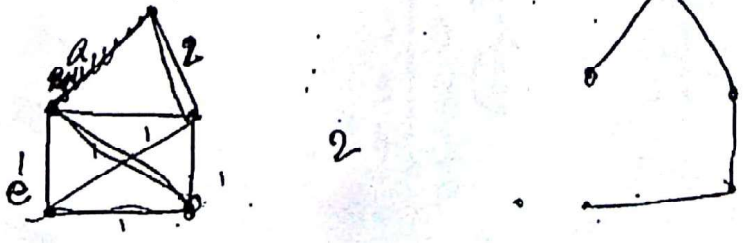
a) G contain U-MST False

b) Every ~~edge~~ MST of G must contain edge e. False.

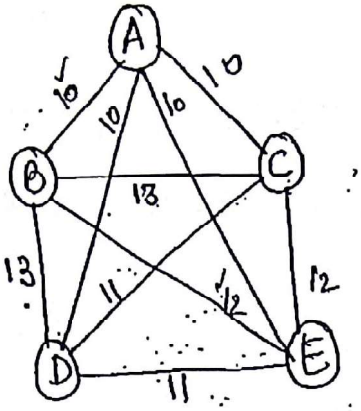
c) " " " " " " atleast 1 edge with weight w True

d) e may be there in MST of G True

e) If e is not in MST then in that cycle all edges contain same weight w. True.

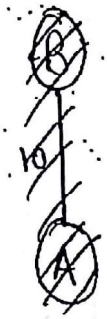


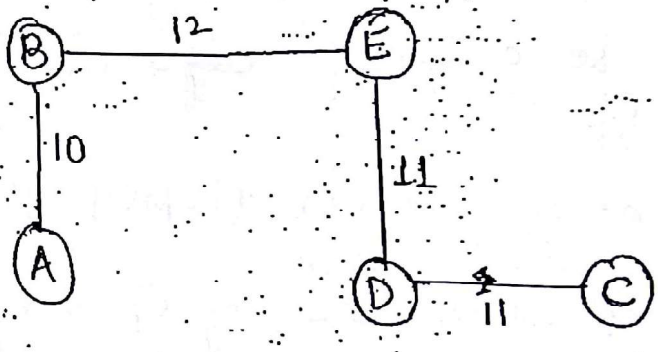
Ex5: Consider the foll. graph :-

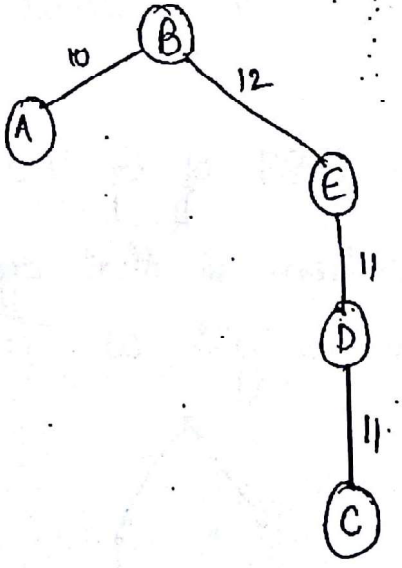


What will be the cost of MST for above graph in such a way that in that MST A will be the leaf node.

degree = 1



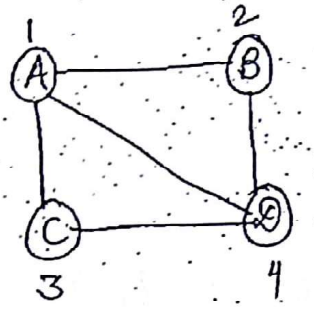




$\equiv$ 44

Representations of Graphs :-

(1) Adjacency Matrix :- (good when more edges ⇒ dense graph)



⟹

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Deg of vertex = Row-sum = $O(V)$

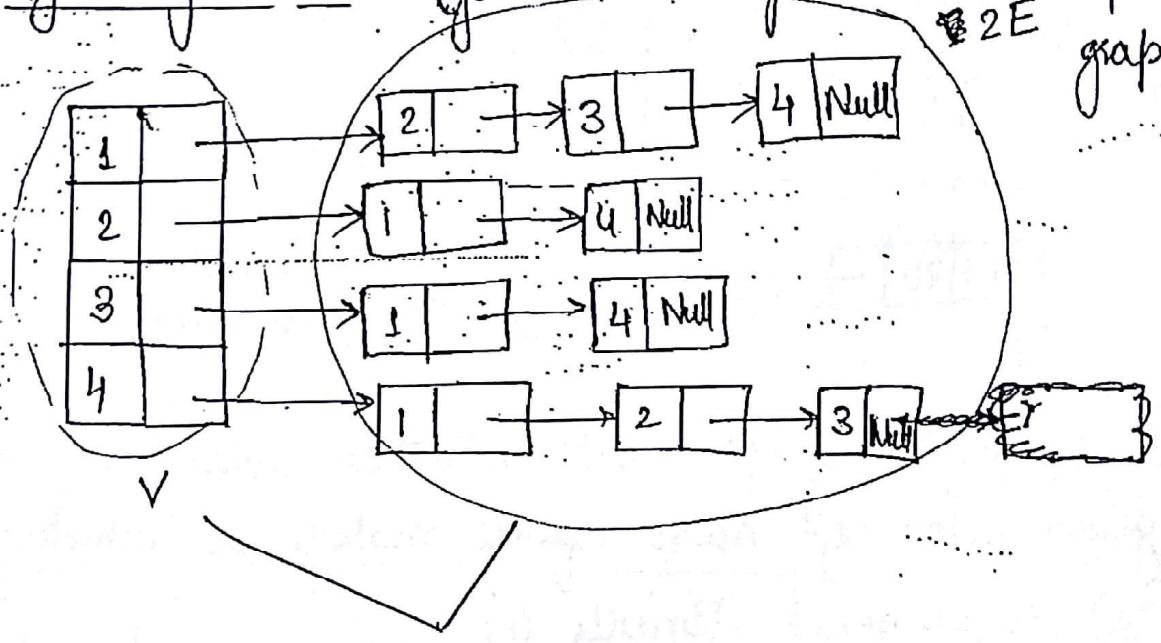
No. of edges = entire matrix = $O(V^2)$

$O(V^2)$ space (For every case)

Even if $e = 0$, Adj matrix takes V^2 space :-

(2) Adjacency List :- (good when edges are less ⇒ sparse graph)

$\leq 2E$

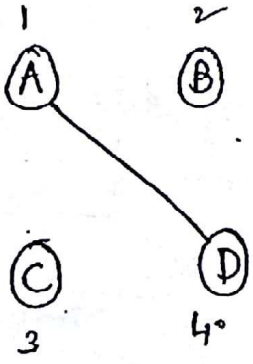


V+2E

⇓

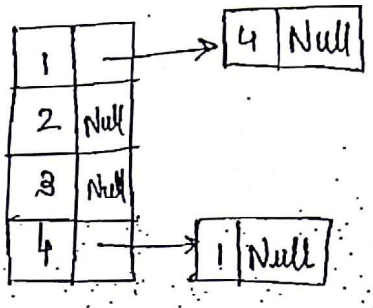
$O(V+E)$ {Depends on vertex & edges}

Ex1:



	1	2	3	4
1	0	0	0	1
2	0	0	0	0
3	0	0	0	0
4	1	0	0	0

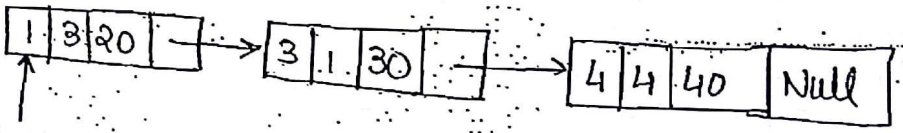
Adjacency list



*For storing sparse matrix, linked list is best representation.

Ex:

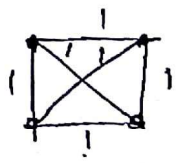
	1	2	3	4
1	0	0	20	0
2	0	0	0	0
3	30	0	0	0
4	0	0	0	40



Let G be a graph with n-vertices whose adj matrix given below by n x n square matrix in which.

(i) all diagonal elements 0's

(ii) all non-diagonal elements 1's



T/F

(a) G contain unique MST X.

(b) G contain multiple MST's each of diff. cost X.

(c) G contain multiple MST ——,,—— same cost

(d) G doesn't have MST. X

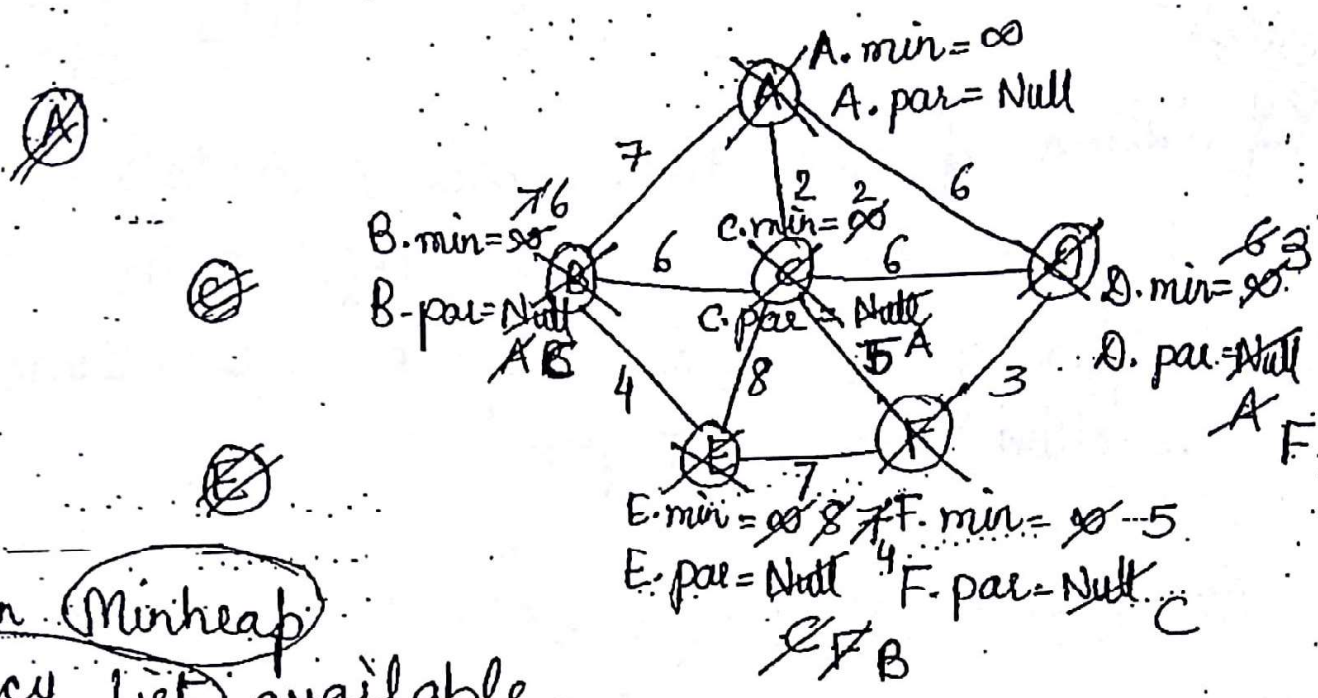
multiple MST → n^{n-2}

same cost → $n-1$

mplexity of Prims Algorithm

crease key / Decrease key operations in Min Heap
leap will take $O(\log n)$ in worst/avg case &
, Best Case

↓
decrease the value at
the key position (less than
current)



in (Minheap)
ency List available.

B	C	D	E	F	build Heap
∞	∞	∞	∞	∞	⇒ O(V)
N	N	N	N	N	
7	2	6	∞	∞	Decrease Key
A	A	A	N	N	⇒ 3 + 3 log V
6		6	8	5	Find adj elements
C		A	C	C	⇒ 5 + 4 log V
6		3	7		

Time Complexity (Prim's Algo) $= V + V\log V + 2E + E\log V$

$= V\log V + E\log V$

$= O[(V+E)\log V]$ Using Binary Minheap & Adjacency List

Note:-

* If instead of min-heap, array is taken, decrease key will take $O(1)$ time but finding min. will require $O(V)$ time using selection sort 1st pass.

$\Rightarrow TC = V^2 + V + 2E + E$ } Array & Adjacency List

$= O(V^2)$

For storing ∞, null in array for 1st time

* Array & Adjacency matrix

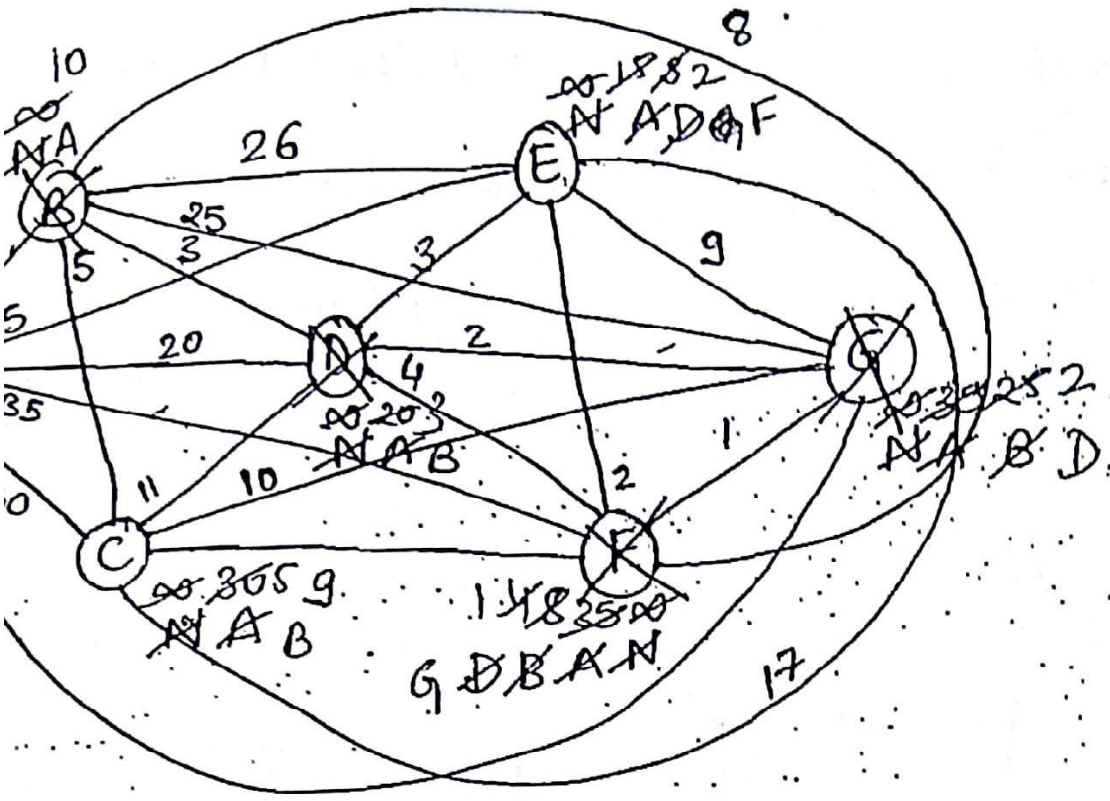
$TC = V^2 + V + V^2 + E$

$= O(V^2)$

* If sorted array used & adjacency

$TC = V + V + 2E + E.V$ (sort after decrease key)

$= O(E.V)$

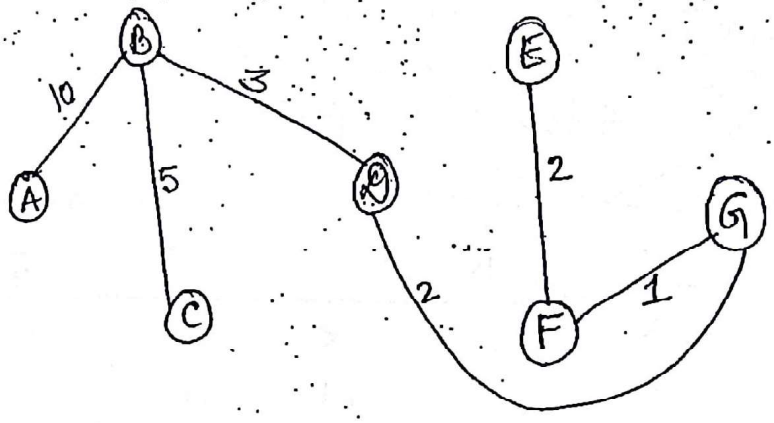


	B	C	D	E	F	G	
	∞	∞	∞	∞	∞	∞	⇒O(V)
	N	N	N	N	N	N	
	10	30	20	15	35	35	⇒6+6
V	A	A	A	A	A	A	logV
	5	3	15	8	25		
V	B	B	A	B	B		⇒6+5 logV
	5		3	4	2		
	B		D	D	D		⇒6+4 logV
	5		3	1			
	B		D	G			⇒6+3 logV
	5		2				
	B		F/G				⇒6+2logV
	5						
	B						⇒6+logV

Time complexity = $V + V\log V + 2E + E\log V$.

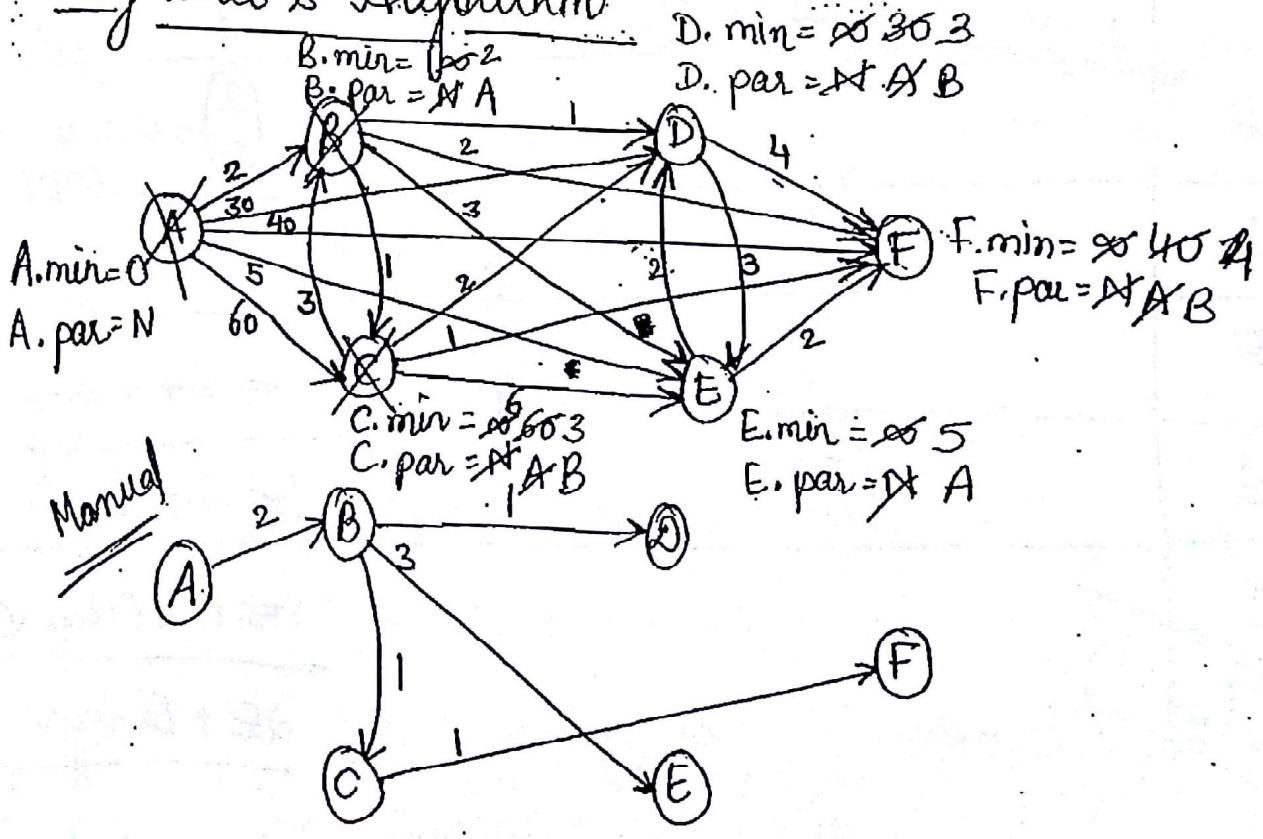
(Prims Algo) = $V\log V + E\log V$

= $O[(V+E)\log V]$ Using Binary MinHeap & Adjacency List



Single source Shortest Path

Dijkstra's Algorithm

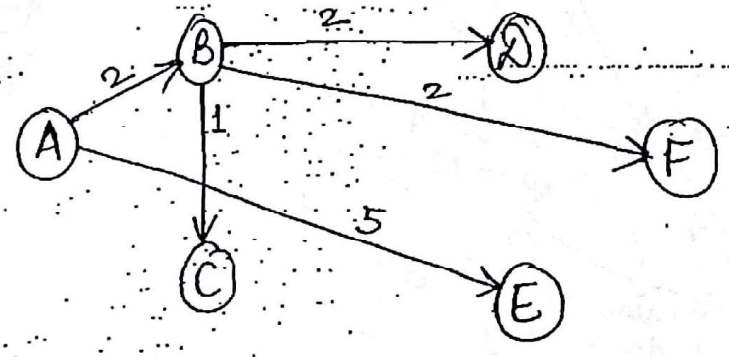


Manual

Present in Min Heap.

	A	B	C	D	E	F	
log V	(0, N)	∞, N	∞, N	∞, N	∞, N	∞, N	⇒ O(V)
A		(2, A)	60, A	30, A	5, A	40, A	⇒ 5 + 5 log V
B			(3, B)	3, B	5, A	4, B	⇒ 4 + 4 log V
C				(3, B)	5, A	4, B	⇒ 4 + 3 log V
D					5, A	(4, B)	⇒ 2 + 2 log V
F					(5, A)		⇒ 0 + 0 log V
E							⇒ 2 + 0 log V
√ log V							

⇒ E + E log V



① Sequence of vertices in order of shortest Distance if source is A

A, B, C, D, F, E.

② A – F (4) : Cost of shortest path from A to F

③ A – F (A – B – F)

Path traversed.

$$TC(Dijkstra) = V + V\log V + E + E\log V$$

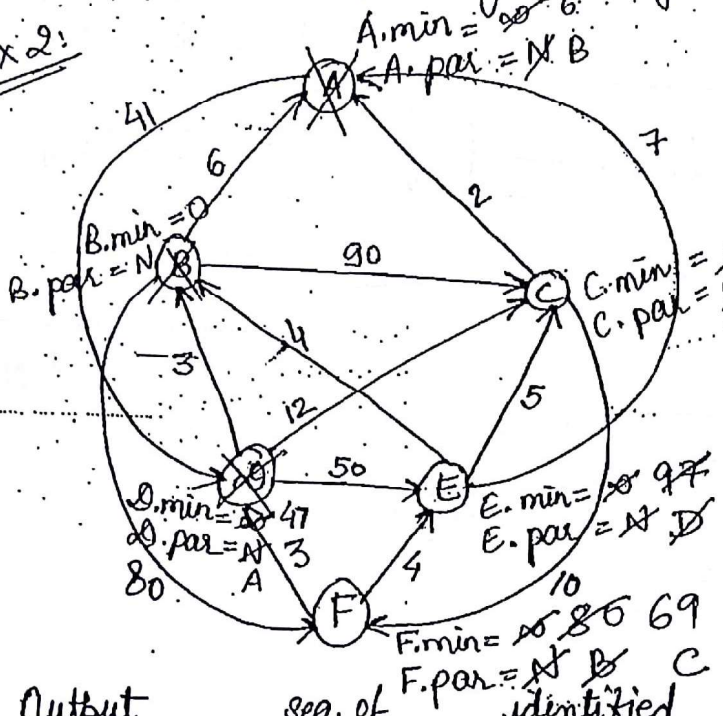
$$= V\log V + E\log V$$

$$= O[(V+E)\log V]$$

Using min-Heap & Adjacency List

$TC = O(V^2)$ // Array & Adjacency List

Ex 2:



① Output seq. of ~~Identify~~ the vertices ~~selected~~ identified using Dijkstra's Algo when algorithm started from B.

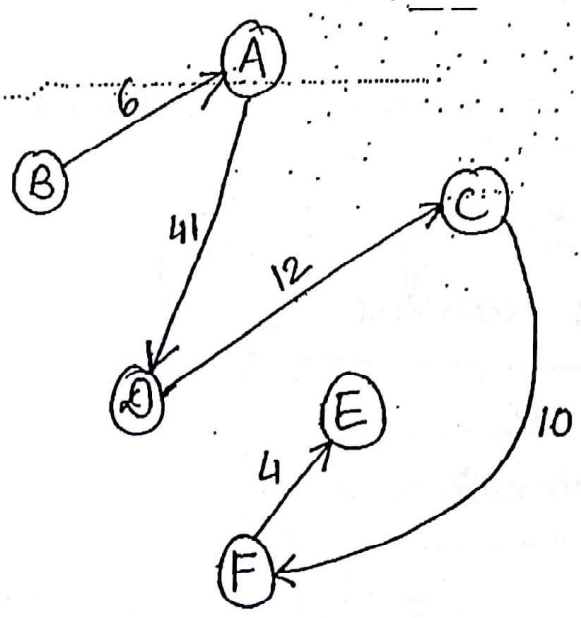
B A D C F E

② What will be the cost of shortest path from B to E? 73

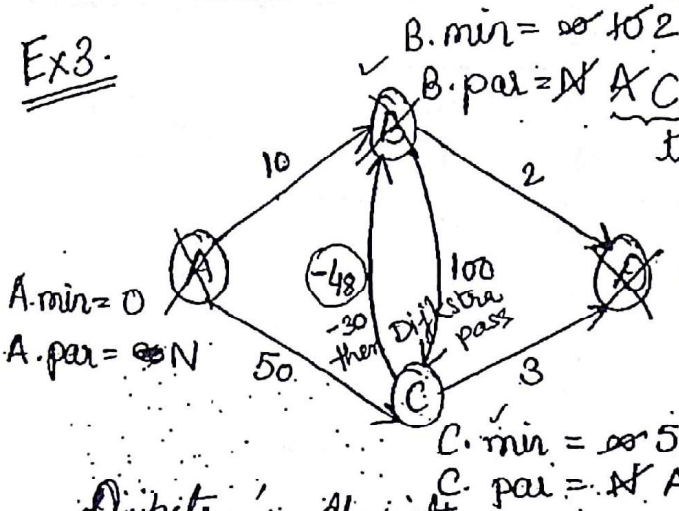
③ What will be the shortest path from B to E?

B - A - D - C - F - E

	A	B	C	D	E	F
	∞ N	0 N	∞ N	∞ N	∞ N	∞ N
B	6 B		90 B	∞ N	∞ N	80 B
A			90 B	47 A	∞ N	80 B
D			59 D		91 D	80 B
C					97 D	69 C
F					73 F	
E						



Ex3.



Dijkstra's Algorithm may fail if edge weights are negative. ⇒ Solution is to use Bellman Ford Algo.

	not always	
A → A : 0		A → A : 0
A → B : 2		A → B : 10
A → C : ~~∞~~ 50		A → C : 50
A → D : ~~∞~~ 4		A → D : 12

In Dijkstra's algo, a variable is relaxed only once.

at A-vertex-adj edges, perform decrease key

⇓

Relaxation at A

⇓

all vertices relaxed once

⇓

all edges decrease key over

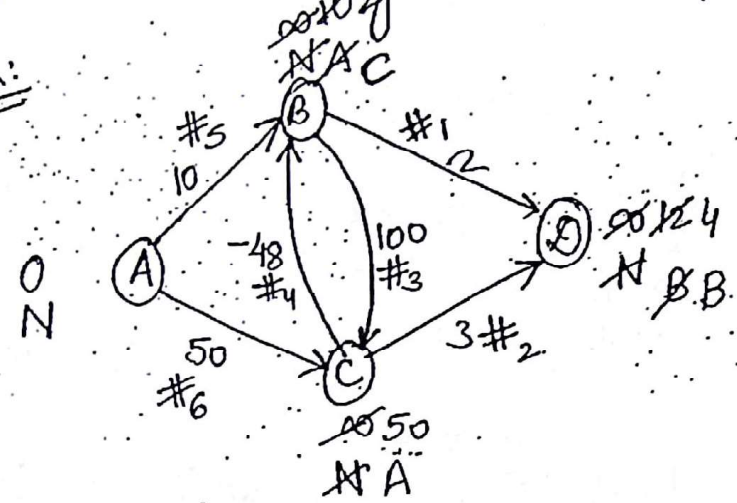
⇓

$E \log V$

Bellman Ford Algo (works with negative edge weights also)

present in array (Min-Heap not reqd.)

Ex:



No. of vertices = V.

Rounds = V-1

	A	B	C	D	
	0	∞	∞	∞	
	N	N	N	N	
i=1	0	10	50	∞	⇒ E * O(1) 1-length path
	N	A	A	N	
i=2	0	2	50	12	⇒ E * O(1) 2-length path (best)
	N	C	A	B	
i=3	0	2	50	4	⇒ E * O(1) 3-length.. (best)
	N	C	A	B.	

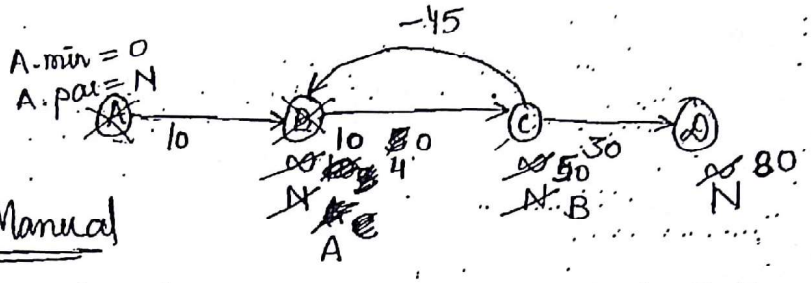
E (V-1)

Time Complexity = O(EV)
(Bellman Ford)

If the graph contains all postive edge weights, Dijkstra's Algo will always give correct ans. If graph contains -ive edge weights, Dijkstra's Algo may fail.

If the graph contains -ive/+ive edge weights, Bellman Ford algo will always give correct answer.

Ex 4:



Manual

A — A : 0

A — B : $-\infty$

A — C : $-\infty$

A — D : $-\infty$

10 - 5 - 5

10 + 40 - 45 + 40

10 - 5 + 40

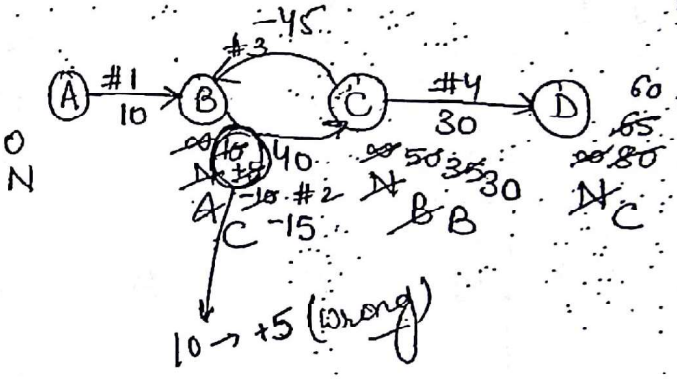
Dijkstra

A — A : 0

A — B : 10

A — C : 50

A — D : 80

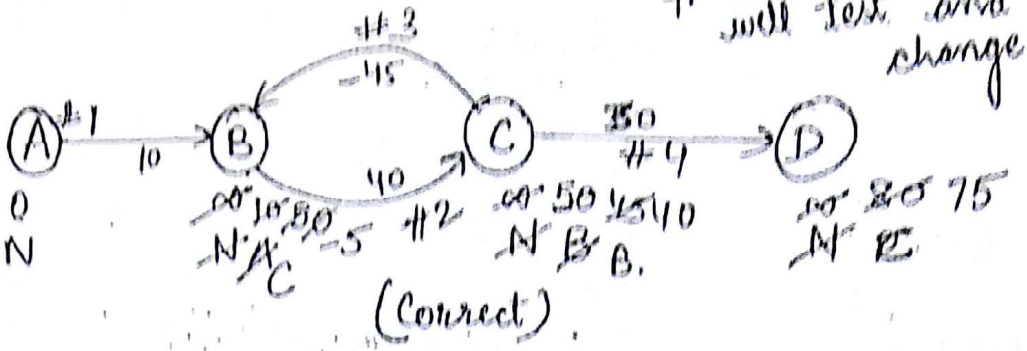


Bellman Ford

Rounds = 3.

	A	B	C	D
	0	∞	∞	∞
	N	N	N	N
i=1	0	+5	50	80
	N	C	B	C
i=2	0	0	45	75
	N	C	B	C
i=3	0	-5	40	70
	N	C	B	C

// The next iteration (i=4) will test and show change in result.



(Correct)

If the graph has some part disconnected; then the cost of reaching the vertices of disconnected part will always be ∞.

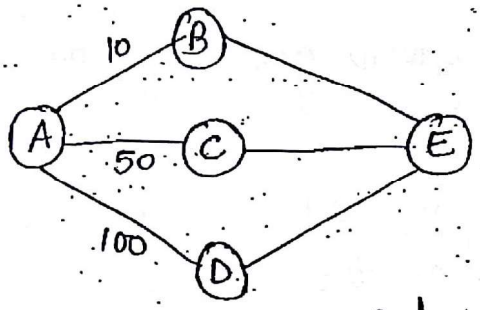
// Bellman Ford Algo will find out all -ive edge weight cycles present in the given graph if they are reachable from the source.

// If any vertex is dependent upon/ part of negative edge weight cycle, it will return the answer as $-\infty$ (undefined). For other vertices it will return cost of shortest path which is well defined.

Dynamic Programming

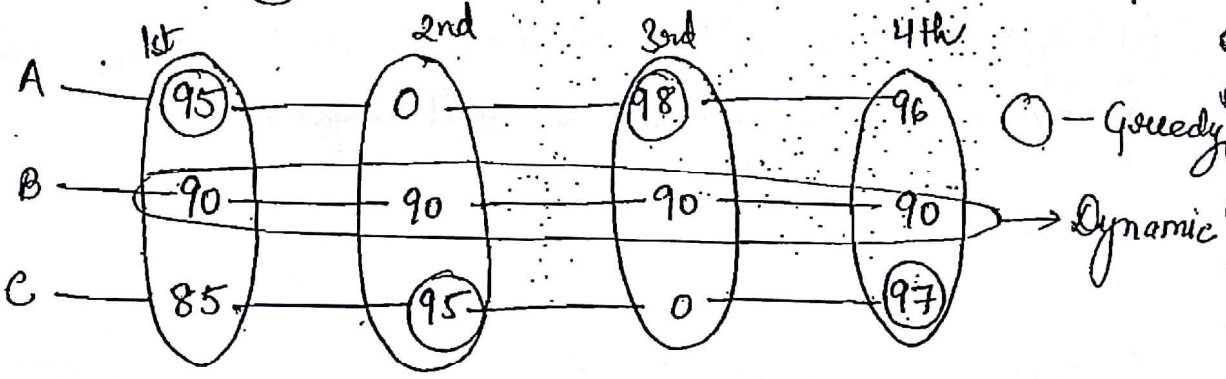
Greedy Technique	Dynamic Programming
① Less possibilities	① Covers more possibilities
② Sometimes wrong answer	② Every time correct answer
③ Less time	③ More time

Ex:



Greedy → will give path A → B

Dynamic → give entire data



Greedy is local optimization and DP is global optimization.

Applications of Dynamic Programming :

1. Fibonacci Series
2. Longest Common Subsequence
3. Matrix chain multiplication
4. 0/1 Knapsack
5. Sum of Subsets
6. All pairs Shortest Path
7. Optimal Cost BST.

(1) Fibonacci Series

n	0	1	2	3	4	5	6	7	8	9
fib(n)	0	1	1	2	3	5	8	13	21	34

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

Recurrence Relation (Value) :-

$$f(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ f(n-1)+f(n-2) & \text{if } n>1 \end{cases}$$

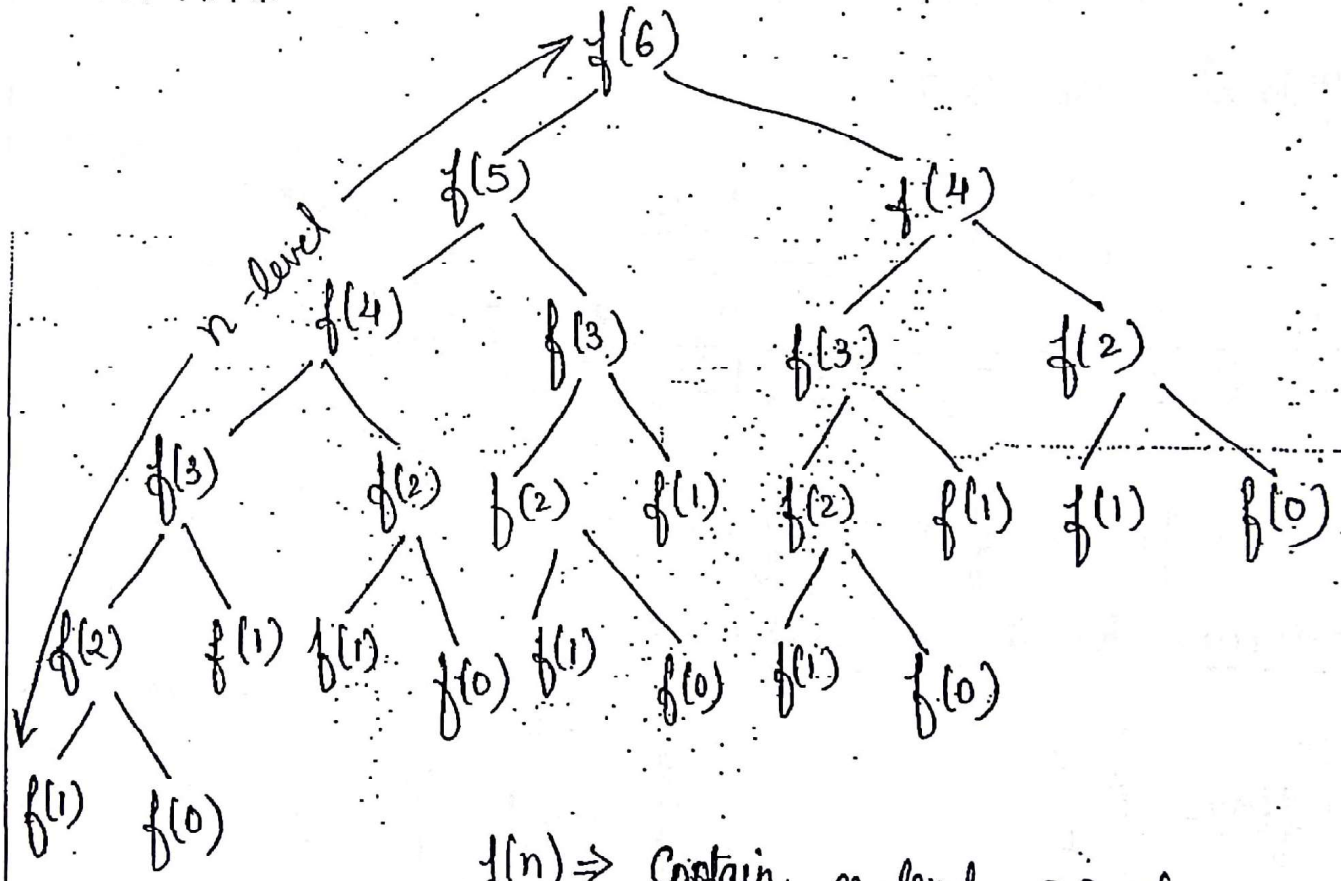
Recursive Program :

$f(n) \Rightarrow T(n)$

```
{
    if(n==0 || n==1) return n;        => O(1)
    else
    {
        a = f(n-1);                   => T(n-1)
```

```
    b = f(n-2);  ⇒ T(n-2)
    c = a+b;  ⇒ O(1)
    return c;
  }
}
```

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



f(n) ⇒ Contain n-level CBT(UB)

⇓

2^{n-1} nodes

⇓

2^n function call

⇓

$2^n * O(1)$

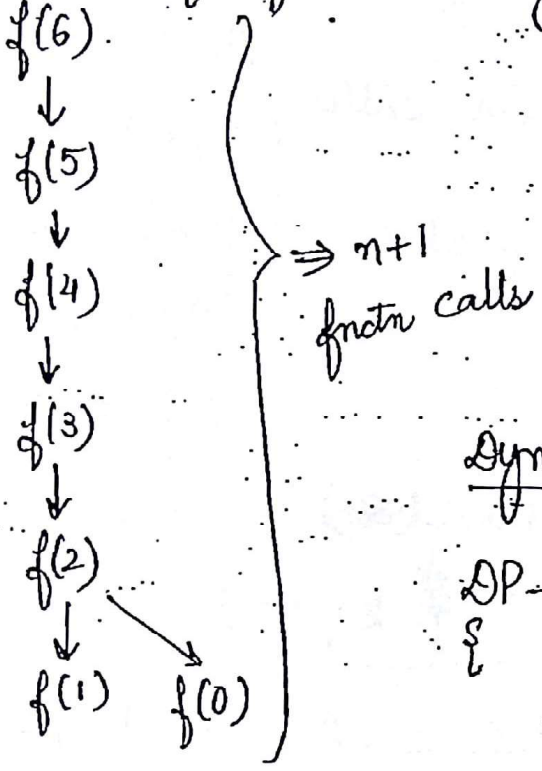
⇓

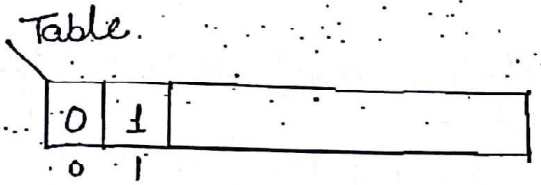
Space = i/p + extra

nB nB.

O(n)

In case of Dynamic Programming function calls:-





Dynamic Programming Program

```
DP-fib (n)
{
    if (n == 0 || n == 1)
        return n;
    else
    {
        if (table [n-1] == null)
            table [n-1] = DP-fib (n-1);
        if (table [n-2] == null)
            table [n-2] = DP-fib (n-2);
        table [n] = table [n-1] + table [n-2];
        return table [n];
    }
}
```

In the recursive tree, many function calls are repeating (overlapping subproblems). In dynamic programming, we'll compute only distinct fnctn calls because as we compute, we store its value in a table so that we can reuse it if needed afterwards.

$f(n)$ – $(n+1)$ distinct function calls.

$= (n+1) * O(1)$

$= O(n)$

Space – i/p + extra

2B Stack ⇓ nB Table ⇓ nB

2n B

$O(n)$ //Mathematically, space reqd. is more but asymptotically remains space.

(2) Longest Common Subsequence

Subsequence :- of a given sequence is just the given sequence only in which 0 or more ~~sequence~~ symbol left out.

Ex: $S = (A, B, B, A, B, B)$ (1 2 3 4 5 6)

$S_1 = (A, A)$ (1 4) $S_3 = (A, B, B, A, B, B)$ (1 2 3 4 5 6)

$S_2 = (B, B, B, B)$ (2 3 5 6) $S_4 = ( )$ $S_5 = (B, A, B, A)$ (2 4 5 ?)

Common Subsequence : Z is a common subsequence of 2 sequences x & y iff Z is subsequence to x and subsequence to y also.

Ex 1 $x = (A_1, B_2, B_3, A_4, B_5, B_6)$

$y = (B_1, A_2, A_3, B_4, A_5, A_6)$

$Z_1 = (A, A)$

$Z_2 = (A, B, A) \Rightarrow$ Longest Common Subsequence.

$Z_3 = (B)$

$Z_4 = ()$

Ex 2: $x = (A, B, A, B, A, B)$

$y = (B, A, B, A, B, A)$

$Z_1 = (B, A, B, A, B)$ or (A, B, A, B, A)

$Z_2 = (A, B, A, B)$

$Z_3 = (A, B, A)$

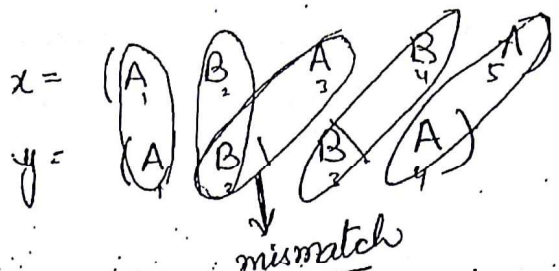
$Z_4 = (A, B)$

$Z_5 = (A)$

$Z_6 = ()$

LCS(m, n) = Length of the longest common subsequence possible with 2 sequences x & y where x contains 'm' symbols & y contains 'n' symbols.

Ex:



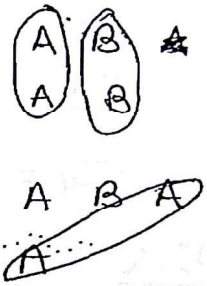
$LCS(5,4) = 1 + LCS(4,3)$ (4, 3)

$\Downarrow$

$1 + LCS(3,2)$ (3, 2)

$\Downarrow$

$\max \begin{cases} LCS(2,2) \rightarrow 2 & A\ B \\ & A\ B \\ LCS(3,1) \rightarrow 1 & A \end{cases}$



Recurrence Relation :- (for value)

$$LCS(m,n) = \begin{cases} 0, & \text{if } m=0 \text{ or } n=0 \\ 1 + LCS(m-1,n-1) & \text{if } x_m == y_n \\ \max\{LCS(m,n-1),\ LCS(m-1,n)\} & \text{if } x_m != y_n \end{cases}$$

Recursive Program

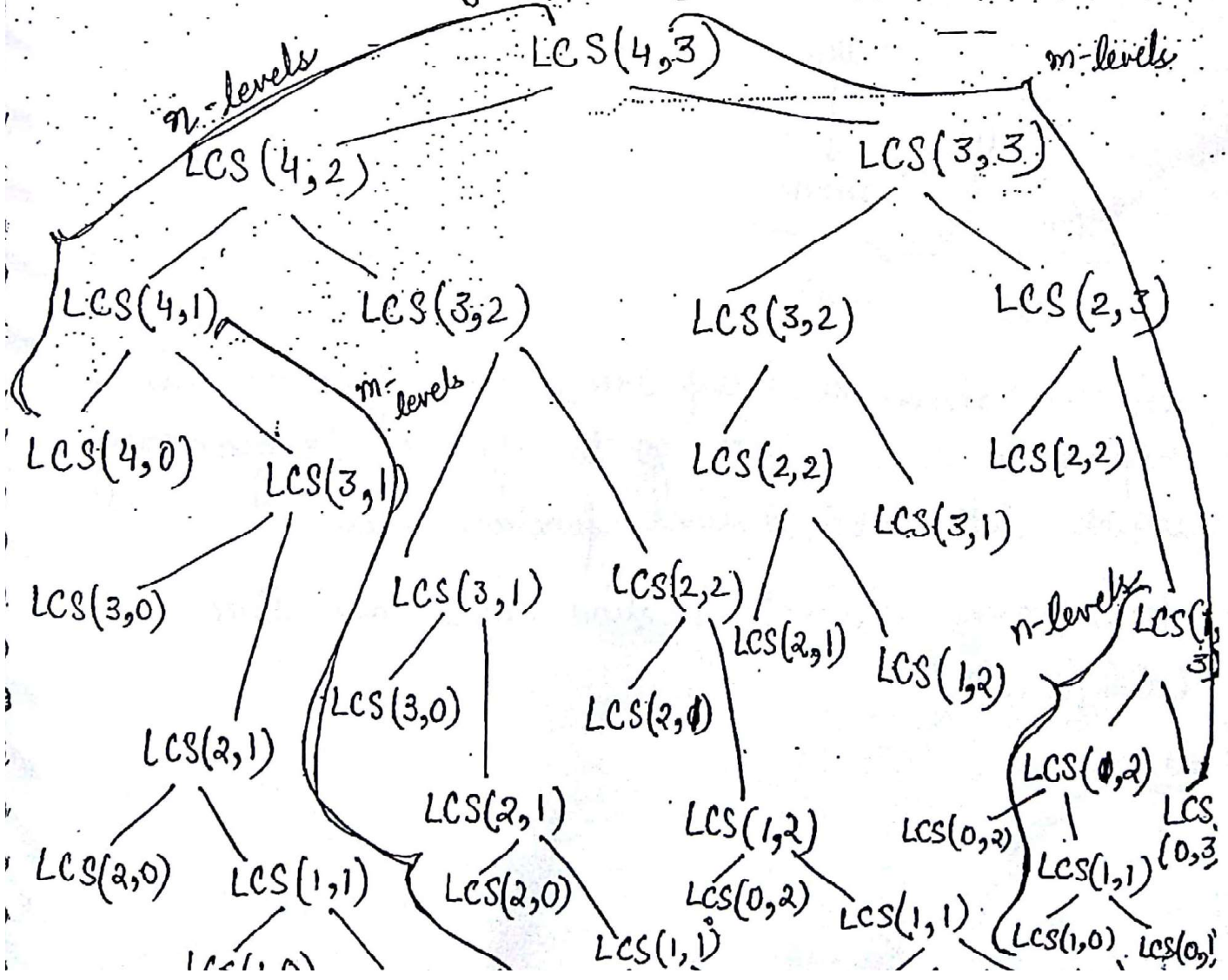
```
LCS(x, m, y, n)
{
  if (m == 0 || n == 0)
    return 0;
  else if (x[m] == y[n])
    return 1 + LCS(x, m-1, y, n-1);
  if (x[m] != y[n]
```

$O(\min(m,n))$ {Best case}

```
else
{
    a = LCS(x, m, y, n-1);
    b = LCS(x, m-1, y, n);
    if (a > b)
        return a;
    else
        return b;
}
}
}
```

Recursive Tree : $x = (A_1, A_2, A_3, A_4)$

$y = (B_1, B_2, B_3)$ Total levels = $m+n$



LCS (m, n)

↓

m+n -level CBT (UB)

↓

2^{m+n} nodes

↓

2^{m+n} function calls

↓

$O(2^{m+n}) = O(2^m \cdot 2^n)$

Stack space = No. of levels = $O(m+n)$.

⇓

i/p + extra

(x, y)

(m + n) m+n

$O(m+n)$

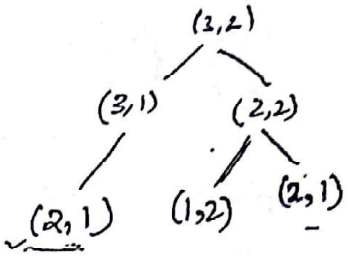
In the above recursive tree, many function calls are repeating so we will go to dynamic programming which will solve distinct function calls.

How many distinct function calls are there in LCS (m, n)?

Ans:

LCS(5, 10)

4	9
3	8
2	7
1	6
0	5
	4
	0
m+1	n+1



Distinct function calls = $(m+1).(n+1)$

$= O(m.n)$

Space (with DP) = i/p + extra

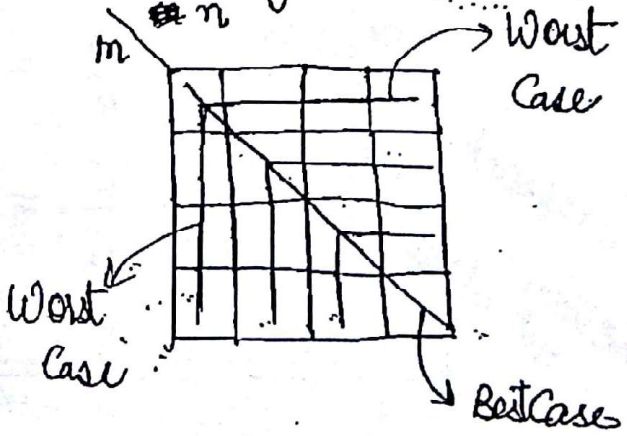
i/p : $m+n$

extra : Stack $(m+n)$, Table $(m.n)$

$= m+n + m+n + mn$

$= O(m.n)$

Dynamic Programming is worst idea if repetition are very less. As time is decreased very less and space is increased substantially.



③ 0/1 Knapsack

Ex: $n = 3$ $m = 10$

Objects:	obj 1	obj 2	obj 3
Profit:	93	50	55
Weight:	7	4	6

Manually,

000 → 0
001 → 55
010 → 50
011 → 105
100 → 93
101 → ✗
110 → ✗
111 → ✗

Greedy, obj 1 ⇒ $\frac{93}{7} = 13.2$

obj 2 ⇒ $\frac{50}{4} = 12.5$

obj 3 ⇒ $\frac{55}{6} = 9.1$

Ans 0/1 knapsack fraction not allowed

(1 0 0) ⇒ 93, correct ans ⇒ 105

↑ 93

For 0/1 knapsack problem, Greedy will fail. So, we have to go to Dynamic Programming.

Using Dynamic Programming :-

Ex: $n = 5$, $m = 15$

objects :	obj 1	obj 2	obj 3	obj 4	obj 5
profits :	25	75	15	45	35
weight :	2	5	3	4	7

01KS(m, n) = max profit you will get in 0/1 Knapsack problem where capacity of knapsack is m and no. of objects are n.

① 01KS$\begin{pmatrix} 5,0 \\ 0,5 \\ m,n \\ 0,0 \end{pmatrix}$ $= 0$

② 01KS(m, n) $=$ 01KS$(m, n-1)$ if $w_n > m$ $(70 > 15)$

③ 01KS$(m, n) =$ $\max \begin{cases} \text{01KS}(m - w_m, n-1) + P_n \\ \text{01KS}(m, n-1) \end{cases}$ if $w_m \leq m$

every time n decreases, So, Binary tree with n-levels formed.

01KS(m, n)

⇓

n-level CBT (VB)

⇓

$2^n - 1$ nodes

⇓

$O(2^n)$ —

Space : i/p + extra

(obj info) 1 array n-level Stack

$\underbrace{\qquad\qquad\qquad\qquad}_{n}$

$O(n)$

How many distinct function calls are there?

01KS(m, n)

15 5
14 4
13 3
. .
. .
0 0

(approx) $m+1$ $n+1$

$m \cdot n$ distinct fnctn call

$= O(m \cdot n)$

Space : i/p + extra

n Stack: n Table: m.n

$= O(m \cdot n)$

Note : Because of very less repetitions, time complexity of 0/1 Knapsack $O(m.n)$ is approx equal to $O(2^n)$. So it is one of the NP-complete problem.

↓

Solved in $O(2^n)$ time i.e. no soln apart from Brute Force.

(4) Matrix Chain Multiplication

Ex 1 $A_{2\times4}$ $B_{4\times3}$

$C = A.B.$

↓

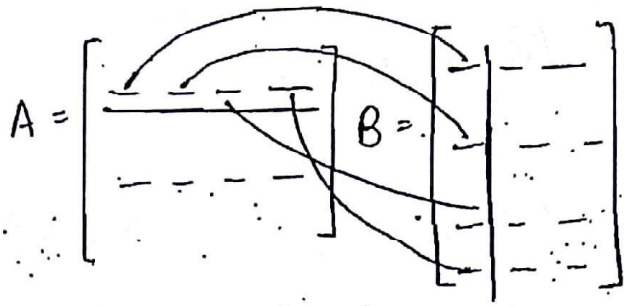
2×3 elements

⇓

$2*3*4$ multiplications

⇓

24 mul.



4 mul. reqd / element

Ex 2: $A_{10\times5}$ $B_{5\times7}$

$C = A * B.$

⇓

10×7 elements

⇓

$10\times7\times5$ mult.

$= 350$.

Ex 3: $A_{2\times3}$ $B_{3\times4}$ $C_{4\times2}$

$D = ABC$

AB_{2*4} C_{4*2}

$\Rightarrow 2*4*3 + 2*2*4$

$24 + 16 = 40$

A_{2*3} BC_{3*2}

$D = ABC$

$\Rightarrow 2*2*3 + 3*2*4$

$\Rightarrow 12 + 24 = \boxed{36}$

Ex 4: $A_{5\times2}$ $B_{2\times3}$ $C_{3\times7}$

$D = ABC$

$AB = 5\times3\times2$

$= 30$

$AB_{5\times3}\ C_{3\times7} = 5\times7\times3 + 30$

$= 105 + 30 = 135$

$BC = 2\times7\times3 = 42$

$A_{5\times2}\ BC_{2\times7} = 5\times7\times2 + 42$

$= 70 + 42 = \boxed{112}$

Ex 5: $A_{2\times3}$ $B_{3\times1}$ $C_{1\times4}$ $D_{4\times3}$

$E = ABCD$

$(AB)_{2\times1} \cdot (CD)_{1\times3}$

$2\times1\times3$ $\quad$ $1\times3\times4$

$= 6 + 12 + {}_{2\times3\times4}$

$= \boxed{24}$

$B_{3\times1}.CD_{1\times3} = 3\times3 + 12$

$= 9 + 12$

$= 21$

$A_{2\times3}.(BCD)_{3\times3} = 2\times3\times3$

$= 18 + 21$

$= 39$

$AB_{2\times1}.C_{1\times4} = 2\times4\times1$

$+ 6$

$= 8 + 6 = 14$

$ABC.D_{2\times4\ 4\times3} = 2\times3\times4 + 14$

$= 24 + 14$

$= 38$

$(BC)_{3\times4} = 3\times4\times1$

$= 12$

$A.BC_{2\times3\ 3\times4} = 2\times4\times3 + 12$

$= 24 + 12$

$= 36$

$A(BC)D_{2\times4\ 4\times3} = 2\times3\times4 + 36$

$= 24 + 36$

$= 60$

$(BC) = 12$

$BCD_{3\times4\ 4\times3} = 3\times3\times4 + 12$

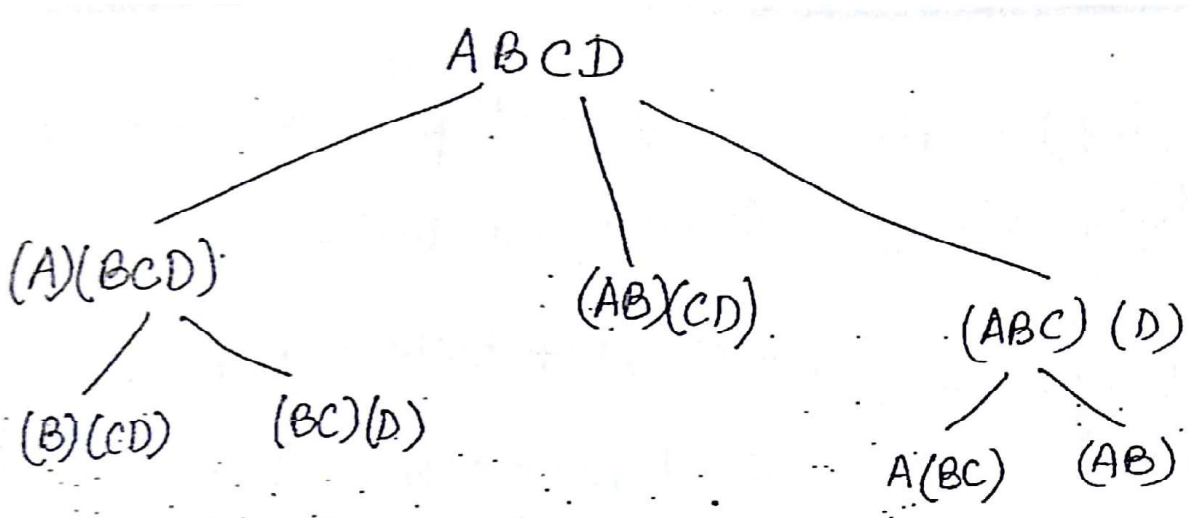
$= 36 + 12$

$= 48$

$A(BCD)_{2\times3\ 3\times3} = 2\times3\times3 + 48$

$= 18 + 48$

$= 66$



This is also known as optimal way of parathesization

Ex 6.

$A_{1\times2}$ $B_{2\times1}$ $C_{1\times1}$ $D_{1\times2}$

E = ABCD

10 | 6 | 5

$1\times2\times2$ = (A)(BCD) — 1×2 2×2

$1\times2\times1$ = (AB)(CD) — 1×1 1×2

1×1 1×2
2 = (ABC)(D)

6 | 6

(B)(CD)

$CD = 1\times2 = 2$ (1×1 1×2)

$B.CD = 2\times2+2 = 4+2 = 6$ (2×1 1×2)

(BC)(D)

$BC = 2\times1 = 2$ (2×1 1×1)

$BC.D = 2\times2 + 2 = 6$ (2×1 1×2)

$AB = 1\times1\times2 = 2$ (1×2 2×1)

$CD = 2$

4 | 3

(A)(BC)

$BC = 2$

$A.BC = 1\times1\times2 = 2$ (1×2 2×1)

(AB)(C)

$AB = 2$

$AB.C = 1\times1$ (1×1 1×1)

$= 1$

MCM(i, j) = min. no. of multiplications reqd. to multiply i to j matrices.

$$MCM(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ MCM(i,k) + MCM(k+1,j) + d_{i-1} * d_j * d_k \} & \text{if } i \neq j \end{cases}$$

$$MCM(1,4) = \min \begin{cases} MCM(1,1) + MCM(2,4) + 4 \\ MCM(1,2) + MCM(3,4) + 2 \\ MCM(1,3) + MCM(4,4) + 2 \end{cases}$$

Assume,

$M_8 \Rightarrow M_{25} \Rightarrow \dots M_i$

Size: $d_8 \times d_7$; $d_{24} \times d_{25}$; $d_{i-1} \times d_i$

$d_{i-1} \times d_k$; $d_{i-1} \times d_i$; $d_{k-1} \times d_k$

$d_k \times d_{k+1}$; $d_{j-1} \times d_j$

$d_{i-1} \times d_k$; $d_k \times d_j$

$d_{i-1} \times d_j$

$\forall k$, $i \le k < j$

$\Rightarrow$ Combined multiplication Cost $= d_{i-1} * d_j * d_k$

Note :-

MCM(1, n) will generate n-level n-ary tree.

almost (UB)

Without DP

MCM (1, n)

↓

n-level complete (n-1)-ary tree (UB)

↓

$(n-1)^n - 1$ nodes

↓

n^n nodes

↓

n^n function calls

↓

$n^n * O(n) \Rightarrow O(n^{n+1})$

Every func^n call will take O(n) For computing min of n elements

In the above recursive tree, many function calls are repeating So we will go to Dynamic Programming which will solve distinct function calls.

How many distinct function calls are there in MCM (1, n)?

MCM (1, n)

1	4
2	3
3	2
4	1
n	n

↓

n^2 distinct function calls

$n^2 * O(n) = \boxed{O(n^3)}$

Space complexity (without DP):-

i/p + extra

i/p → only size of matrices read. row, column → 4B. → 4n

extra → Stack → n

$= 5n = O(n)$

With DP :- i/p + extra

i/p → 4nB

extra → Stack (n) ---- Table (n^2 (distinct fnctn calls data o/p))

$= O(n^2)$

(5) Sum of subsets //Decision Problem

i/p : Set of n-elements and integer m.

o/p : Is there any subset whose sum is m?

Ex: S = (50, 25, 100, 20, 30, 75, 40)
1 2 3 4 5 6 7

m = 150 → (50, 100) / (100, 20, 30)

SOS(m, n) = Given the given set of n - elements, is there any subset whose sum = m. $A = \phi$

① $SOS(m, n) = A$ if $m = 0$

② $SOS(m, n) = error$ if $n = 0$

③ $SOS(m, n) = SOS(m, n-1)$ if $S_n > m$

④ $$SOS(m, n) = \begin{cases} SOS(m - w_m, n-1), & A = A \cup \{S_n\} \\ SOS(m, n-1), & \end{cases}$$

if $S_n \leq m$

It seems equal to 0/1 knapsack problem. It is NP complete problem as 2^n time reqd.

SOS (m, n)

↓

n-level CBT (UB)

↓

$2^n - 1$ nodes

↓

2^n function calls

↓

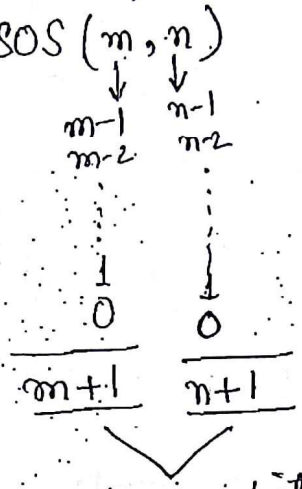
$2^n * O(1) = O(2^n)$

Space complexity : i/p + extra

↓

n Stack (n), n-level tree

How many distinct function calls are there?



m.n distinct fnctn calls

$= O(m.n)$

Space Complexity = i/p + extra

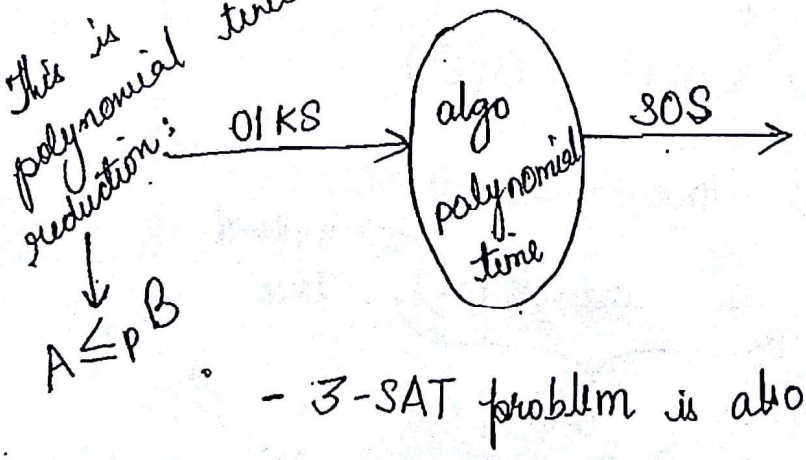
i/p → n

Stack → n

Table → mn

$= O(m.n)$

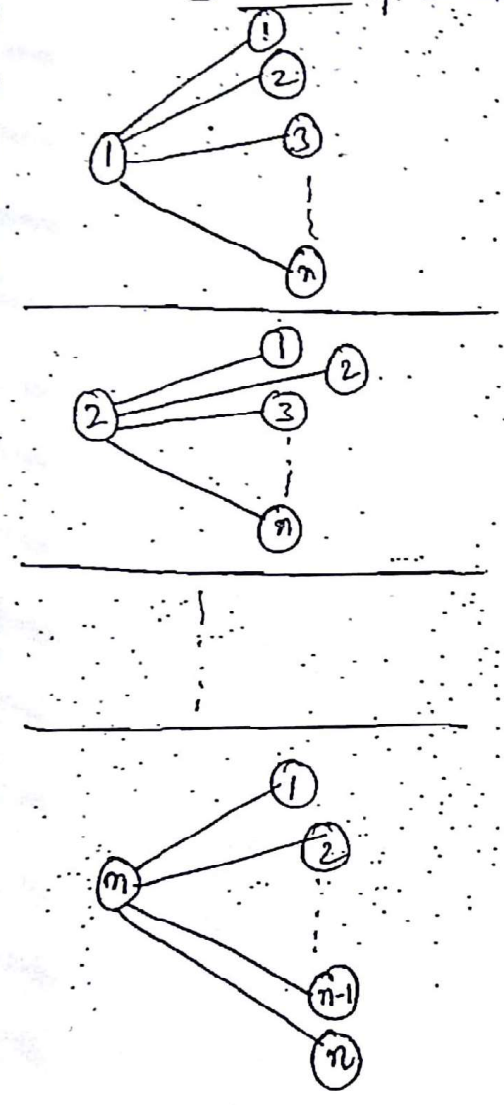
Note: Because of very less repetitions, time complexity of sum of subsets problem $O(mn)$ using DP. is approx equal to $O(2^{n/2})$. So, it is one of the NP complete problem.



- 3-SAT problem is also NP-complete

Note: 0/1 Knapsack problem is polynomially converted into Sum of Subsets problem. Because 0/1 Knapsack problem is NP-Complete, sum of Subsets also NP-complete.

(6) All pairs Shortest Path



n^2 pairs

+ive edge weights:

To find all pairs Shortest Path, apply Dijkstra's algo V times ~~at~~ keeping all vertices as soure :-

$V * Dijkstra = V * (V+E) \log V$

-ive edge weights:

Bellman Ford algo.

$V * Bellman\ Ford \Rightarrow$

$V * EV = O(EV^2)$

Floyd-Warshall Algorithm

Using DP, $TC = O(V^3)$

It works for +ive/-ive edge weights but not for -ive edge weight cycle.