Algorithm	Program
- Design part of project	- Implementation part of project
- Should have domain Knowledge	- Programmers
- Any language	- Programming language
- H/w and OS independent	- H/w and OS dependent
- Analyse	- Testing

Priori Analysis	Posteriori Analysis
- Algorithm	- Program
- Independent of language	- Language dependent
- Hardware independent	- Hardware dependent
- Time & Space function	- Watch time & Bytes

Characteristics of algorithm

(i) i/p → 0 or more
(ii) o/p → atleast 1 o/p
(iii) Definiteness
(iv) Finiteness
(v) Effectiveness

How to analyse an algorithm?

(i) Time
(ii) Space
(iii) Network Consumption
(iv) Power
(v) CPU Registers

(Ex) Algorithm swap(a,b)

```
{
temp = a;   ——— (1)
a = b;      ——— (1)
b = temp;   ——— (1)
}
```

$f(n) = 3$

$O(1)$

Space

a - 1

b - 1

temp - 1

$f(n) = 3$

$O(1)$

Frequency Count Method

A	8	3	9	7	2
	0	1	2	3	4

space

$A \longrightarrow n$

$n \longrightarrow 1$

$s \longrightarrow 1$

$i \longrightarrow 1$

$S(n) = n+3$

$\longrightarrow O(n)$

Q (Sum of all elements in arr)

```
Algorithm Sum (A,n){
    s = 0                          —— 1
    for (i=0; i<n; i++){           —— 1, n+1, n      → 2n+2 = n+1
        s = s + A[i];              —— n
    }
    return s;                      —— 1
}
```

$f(n) = 2n+3$

$\longrightarrow O(n)$

Q. Sum of two matrices

```
Algorithm Add (A,B,n){
    for (i=0; i<n; i++){           —— 1, n+1, n      → 2n+2 → n+1
        for (j=0; j<n; j++){       —— n(n+1)
            c[i,j] = A[i,j]+B[i,j]; —— n(n)
        }
    }
```

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

$O(n^2)$

Space

$A \longrightarrow n \times n \longrightarrow n^2$

$B \longrightarrow n \times n \longrightarrow n^2$

$c \longrightarrow n \times n \longrightarrow n^2$

$n \longrightarrow 1$

$i \longrightarrow 1$

$j \longrightarrow 1$

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

$\Rightarrow O(n^2)$

Q Multiplication of two matrices

```
Algorithm multiply (A,B,n){
    for (i=0; i<n; i++){           —— 1, n+1, n      → 2n+2 → n+1
        for (j=0; j<n; j++){       —— n(n+1) → n²+n
            c[i,j]=0;              —— n(n) → n²
            for (k=0; k<n; k++){   → n×n×(n+1)
                c[i,j]=c[i,j]+A[i,k]*B[k,j]; → n×n×n
            }
        }
    }
```

Page No:
Date:

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

Time $\longrightarrow O(n^3)$

Space

A – n^2
B – n^2
C – n^2
n – 1
i – 1
j – 1
k – 1

$S(n) = 3n^2 + 4$
$\longrightarrow O(n^2)$

Q1
```
for (i=0; i<n; i++)
{
    start;
}
```
$\longrightarrow 2n+2 \longrightarrow n+1$

$\longrightarrow n$

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

Q2
```
for (i=n; i>0; i--)
{
    start;
}
```
$\longrightarrow n-1$

$\longrightarrow n$

$f(n) = 2n-1$
$O(n)$

Q3
```
for (i=1; i<n; i=i+2)
{
    start;
}
```
$\longrightarrow n/2$

$O(n)$

Q4
```
for (i=0; i<n; i++)
{
    for (j=0; j<n; j++)
    {
        start;
    }
}
```
$\longrightarrow n+1$

$\longrightarrow n(n+1)$

$\longrightarrow n(n)$

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

i	j	Stmt
0	0	0
1	0 1	1
2	0 1 2	2
3	0 1 2 3	3
		⋮
		n

$1+2+3+4+\dots n$
$= n(n+1)/2 = n^2$
$O(n^2)$

Q5

```
p=0
for (i=1; p<=n; i++){
    p=p+i;
}
```

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3 = 6
⋮	⋮
K	1+2+... K = K(K+1)/2

Assume $p > n$

$\because p = \frac{K(K+1)}{2}$

$K(K+1)/2 > n$

$K^2 > n$

$K > \sqrt{n} \longrightarrow O(\sqrt{n})$

Q6

```
for (i=1; i<n; i=i*2)
{
    start;
}
```

i

$1 \times 2 = 2$

$2 \times 2 = 2^2$

⋮

2^K

Assume $i >= n$

$\because i = 2^K$

$\therefore 2^K >= n$

$2^K = n$

$\boxed{K = \log_2 n} \longrightarrow O(\log_2 n)$

```
for (i=0; i<=n; i++)
{
    start;
}
```

$i = 1+1+1+\cdots 1 = n$

$K = n$

Q7

```
for (i=1; i<n; i=i*2)
{
    start;
}
```

(n=8) i		(n=10) i	
1		1	
2	3	2	4
4		4	
8 ✗		8	
		16 ✗	

Q8

```
for (i=n; i>=1; i=i/2)
{
  start;
}
```

i/n

$n/2$

$n/2^2$

$n/2^3 \ldots n/2^k$

Assume $i<1$

$n/2^k < 1$

$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$ → $O(\log_2 n)$

Q9

```
for (i=0; i*i<n; i++)
{
  stmt;
}
```

$i*i<n$

$i*i>=n$

$i^2 = n$

$i = \sqrt{n}$

$O(\sqrt{n})$

Q10

```
for (i=0; i<n; i++){
  stmt;
} for (j=0; j<n; j++){
  stmt;
}
```

n

n

$f(n) = 2n$

$O(n)$

Q11

```
P=0
for (i=1; i<n; i=i*2){
  P++;
}
for(j=1; j<P; j=j*2){
  stmt;
}
```

→ $p = \log n$

→ $\log p$

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

Q12

```
for (i=0; i<n; i++){
  for (j=1; j<n; j=j*2){
    stmt;
  }
}
```

→ n

→ $n*\log n$

→ $n*\log n$

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

→ $O(n\log n)$

Analysis of if and while

Q
```
i=0;          —— 1
while (i<n)   —— n+1
    stmt;     —— n
    i++;      —— n
}
```

$3n+2 \longrightarrow O(n)$

Q
```
a=1;
while (a<b){
    stmt;
    a=a*2;
}
```

a	Terminate
1	$a \geq b$
$1 \times 2 = 2$	$a = 2^k$
$2 \times 2 = 2^2$	$2^k \geq b$
$2 \times 2 \times 2 = 2^3$	$2^k = b$
$\vdots\ 2^k$	$k = \log_2 b$

$(\log n)$

Q
```
i=1;
k=1;
while (k<n){
    stmt;
    k=k+i;
    i++;
}
```

i	k
1	1
2	1+1=2
3	2+2=4
4	2+2+3
5	2+2+3+4
⋮	⋮
m	$2+2+3+4+\dots m = m(m+1)/2$

$k \geq n$

$\frac{m(m+1)}{2} \geq n$

$m^2 \geq n \longrightarrow m = \sqrt{n} \quad O(\sqrt{n})$

Q GCD of two numbers

```
while (m!=n)
    if (m>n)
        m=m-n;
    else
        n=n-m;
}
```

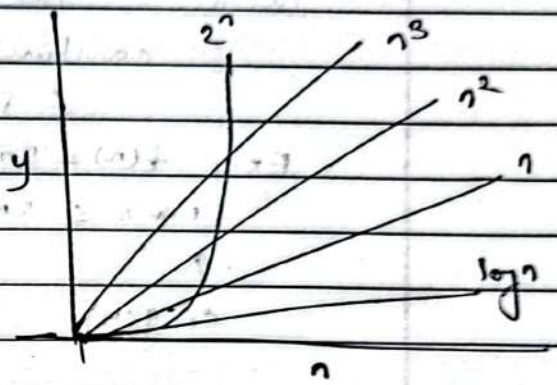
m=16	n=2	
16	2	
12	2	
10	2	
8	2	n/2
6	2	n/2
4	2	
2	2	O(n)

Types of time functions:

$O(1) \longrightarrow$ Constant
$O(n) \longrightarrow$ Linear
$O(n^3) \longrightarrow$ Cubic
$O(\log n) \longrightarrow$ Logarithmic
$O(n^2) \longrightarrow$ Quadratic
$O(2^n) \longrightarrow$ Exponential

Compare class of function: $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

$\log n$	n	n^2	2^n
0	1	1	2
1	2	4	4
2	4	16	16
3	8	64	256
3.1	9	81	512



Asymptotic Notation:

- O big-oh upperbound
- Ω big-o-mega lowerbound
- θ theta average bound.

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

lower bound — average bound — upper bound

- **Big Oh** — The function $f(n) = O(g(n))$ iff $\exists$ +ve constants c and n_0 such that $f(n) \le c * g(n)$ $\forall\ n \geq n_0$

Ex: $f(n) = 2n+3$ → let

$2n+3 \le 2n+3n$

$2n+3 \le 5n \longrightarrow n \geq 1$

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

$2n+3 \le 5n^2$

Nearest one most appropriate

$f(n) = O(n)$ ✓

$f(n) = O(n^2)$ ✓

$f(n) = O(2^n)$ ✓

$f(n) = O(\log n)$ ✗

• **Big Omega** – The function $f(n) = \Omega(g(n))$ iff $\exists$ +ve constants c and n_0 i.e. $f(n) \geq c * g(n)$ $\forall\ n \geq n_0$

Ex. $f(n) = 2n+3$ $\longrightarrow$ $f(n) = \Omega(n^2)$ ✓

$2n+3 \geq 1 * n$ $\forall\ n \geq 1$ $\qquad f(n) = \Omega(\log n)^2$ ✓

$\qquad\qquad f(n) = \Omega(n^2)$ ✗

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

• **Theta** – The function $f(n) = \Theta(g(n))$ iff $\exists$ +ve constants c_1, c_2 and n_0 so that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

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

$1 * n \leq 2n+3 \leq 5 * n$ $\qquad f(n) = \Theta(n)$ ✓

$\qquad\qquad g(n) = \Theta(n^2)$ ✗

$c_1\, g(n)$ $\quad f(n)$ $\quad c_2\, g(n)$

Q $f(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$ $\qquad f(n) = O(n^2)$

$2n^2 + 3n + 4 \leq 9n^2 \longrightarrow n \geq 1$

Ex: $f(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 \geq 1 * n^2$ $\qquad \Omega(n^2)$

$n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$ $\qquad \Theta(n^2)$

Ex: $f(n) = n^2 \log n + n$

$1 * n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$

$O(n^2 \log n)$ $\qquad \Omega(n^2 \log n)$ $\qquad \Theta(n^2 \log n)$

Ex: $f(n) = n! = n \times (n-1) \times (n-2) \times \cdots 3 \times 2 \times 1$

$1 \times 1 \times 1 \cdots 1 \times 1 \leq 1 \times 2 \times 3 \times \cdots \times n \leq n \times n \times n \times \cdots n$

$1 \leq n! \leq n^n$

$O(n^n)$

$\Omega(1)$

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

$\log(1\times1\times1\cdots1) \le \log(1\times2\times3\times\cdots n) \le \log(n\times n\times n\cdots n)$

$1 \le \log n! \le \log n^n$

$\Omega(1)$

$O(n\log n)$

Properties of asymptotic notations

① General properties

If $f(n)$ is $O(g(n))$ then $a * f(n)$ is $O(g(n))$

Ex: $f(n) = 2n^2+5$ is $O(n^2)$

then $7.f(n) = 7(2n^2+5)$

$= 14n^2+35 \longrightarrow O(n^2)$

(ii) Reflexive

If $f(n)$ is given then $f(n)$ is $O(f(n))$

Ex: $f(n) = n^2 \longrightarrow O(n^2)$

(iii) Transitive

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$

Ex: $f(n)=n$, $g(n)=n^2$ $h(n)=n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$

then n is $O(n^3)$

(iv) Symmetric

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$

Ex: $f(n) = n^2$ $g(n) = n^2$

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

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

(v) Transpose Symmetric

If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$

Ex: $f(n) = n$ $g(n) = n^2$

then n is $O(n^2)$ and n^2 is $\Omega(n)$

if $f(n) = O(g(n))$
and $f(n) = \Omega(g(n))$
$g(n) \leq f(n) \leq g(n)$
$\therefore f(n) = \Theta(g(n))$

if $f(n) = O(g(n))$
and $d(n) = O(e(n))$
then $f(n) + d(n)$

Ex

$f(n) = n = O(n)$
$d(n) = n^2 = O(n^2)$
$\therefore f(n) + d(n) = n + n^2$
$= O(n^2)$

Best, Worst and Average Case Analysis

* Linear Search

Best case time – searching key element present at first index

$B(n) = O(1)$

Worst case time – $O(n)$ → Searching a key at last index

$$\text{Avg case} = \frac{\text{All possible case time}}{\text{No. of cases}}$$

Avg. time $= (n+1)/2$

* Binary Search

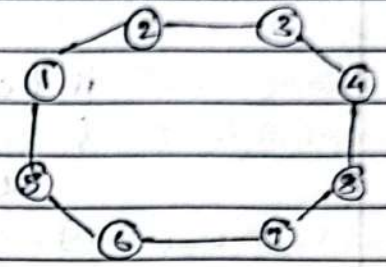
Best case time : $O(1)$
Worst case : $\log n$ $\quad O(\log n)$

Page No:
Date:

Disjoint Sets

1. Disjoint sets and Operations
2. Detecting a cycle
3. Graphical representation
4. Array representation
5. Weighted Union & Collapsing ~~first~~ Find.

Operations: 1. Find
2. Union



$S1 = \{1, 2, 3, 4\}$

$S2 = \{5, 6, 7, 8\}$

$S1 \cap S2 = \phi$

$S3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

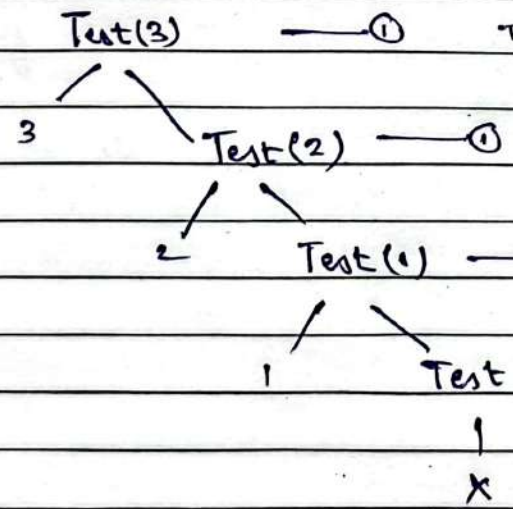
$S_1 \cup S_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Divide and Conquer

Algorithm that recursively breaks down a problem into two or more subproblems of the same or related type, until these become simple enough to be solved directly.

```
DAC(P)
{
    if (small(P))
    {
        S(P);
    }
    else
    {
        divide into P1, P2 .... Px
        Apply DAC(P1), DAC(P2) ....
        Combine (DAC(P1), DAC(P2) ....)
    }
}
```

(i) Binary Search
(ii) Finding maximum and minimum
(iii) Mergesort
(iv) Quicksort
(v) Strassen's Matrix Multiplication




```
T(n)= void Test (int n)
{
    if (n>0)
    {
        1 = printf("%d",n)
        T(n-1) = Test (n-1);
    }
}
```

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

$f(n) =$ n+1 calls

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

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

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

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

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

$\vdots$ K times

$\boxed{T(n) = T(n-K) + K}$

Assuming $n - K = 0$

$\therefore K = n$

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

$T(n) = \cancel{T(0)}^{1} + n$

$T(n) = 1 + n \rightarrow O(n)$

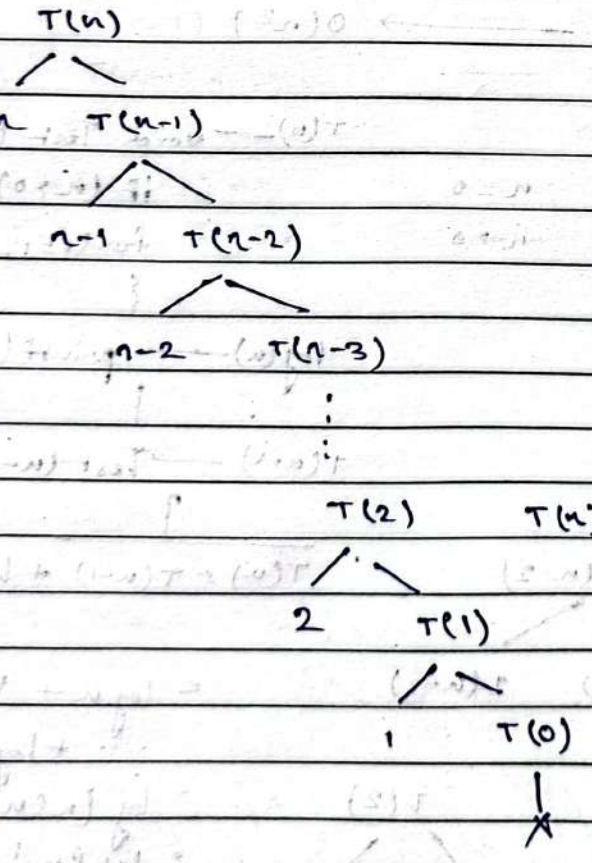
RR: $T(n) = T(n-1) + 1$

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

```
T(n) —  void Test(int n)
        {
1    —  if(n>0)
        {
n+1  —  for (i=0; i<n; i++)
        {
n    —  pf("%d", n);
        }
T(n-1)  Test(n-1);
        }
        }
```

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

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



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

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

$O(n^2)$

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

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

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

$T(n) = T(n-2) + (n-1) + n$ —— ②

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

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

$\vdots$

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

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

Assume $n-k=0 \Rightarrow n=k$ —— ④

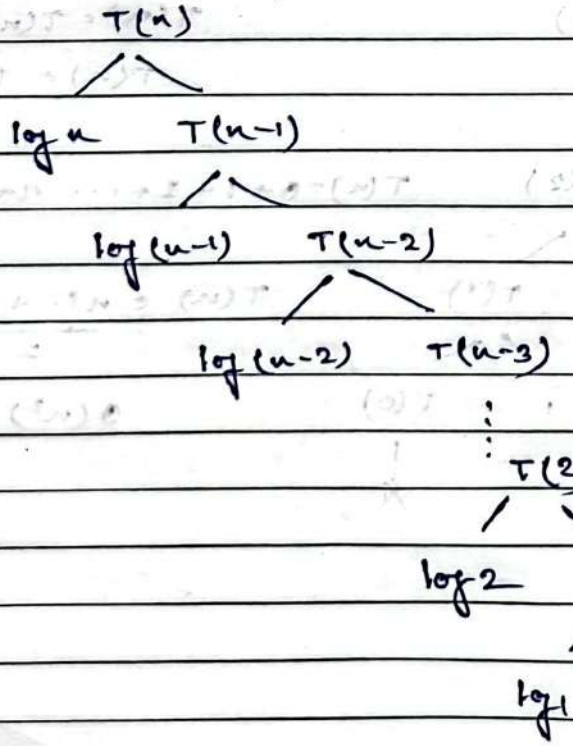
$T(n) = \cancel{T(0)}^{1} + 1 + 2 + 3 + \cdots (n-1) + (n)$

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

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

Q

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + \log n & n>0 \end{cases}$$




```
T(n) —— void Test (int n) {
        if (n>0) {
        for(i=1; i<n; i=i*2)
        {
log(n) —— printf(" %d", i);
        }
T(n-1) —— Test(n-1);
        }
```

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

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

$= \log[n(n-1) \times \cdots \times 2 \times 1]$

$= \log(n!)$

$\Rightarrow O(n \log n)$

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

$T(n) = T(n-1) + \log n$ —— ①

$T(n) = [T(n-2) + \log(n-1)] + \log n$

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

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

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

$\vdots$

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

$T(n-1) = T(n-k) + [\log 1 + \log 2 + \log 3 + \dots (\log(n-1)) + \log(n)]$

Now $n-k=0 \Rightarrow n=k$

$T(n) = T(0)$

$T(n-1) = \cancel{T(0)}^{1} + \log n!$

$T(n-1) = 1 + \log n! \longrightarrow O(n \log n)$

$T(n) = T(n-1) + 1$ —— $O(n)$

$T(n) = T(n-1) + n$ —— $O(n^2)$

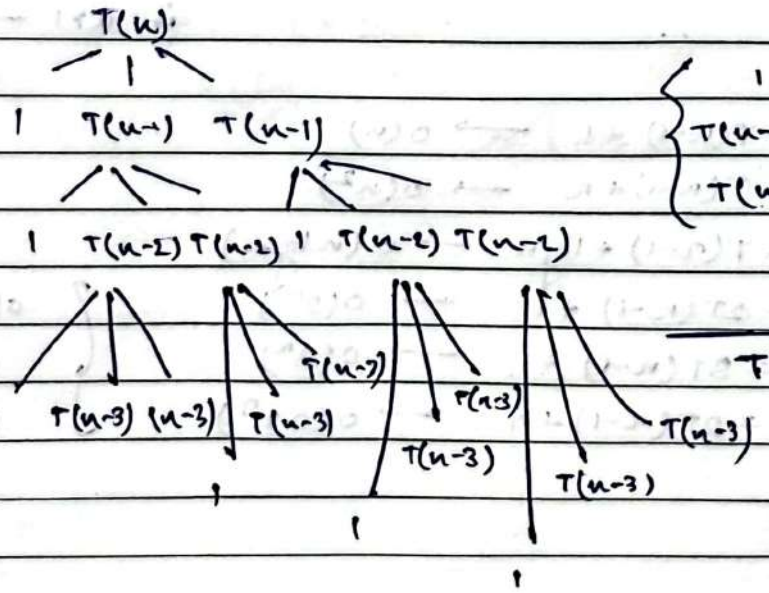
$T(n) = T(n-1) + \log n$ —— $O(n \log n)$

$T(n) = T(n-1) + n^2$ —— $O(n^3)$

$T(n) = T(n-2) + 1$ —— $n/2$ —— $O(n)$

$T(n) = T(n-100) + n$ —— $O(n^2)$

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




```
T(n) — Algorithm Test(int n)
{
  if (n > 0)
  {
    1 — printf("%d", n);
    T(n-1) — Test(n-1);
    T(n-1) — Test(n-1);
  }
}
```

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

$1+2^1+2^2+2^3+\dots+2^k$

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

Assume $n-k=0$

$n=k$

$2^{n+1}-1 \longrightarrow \theta(2^n)$

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

$T(n) = 2T(n-1)+1$ —— ①

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

$= 2^2T(n-2)+2+1$ —— ②

$= 2^2[2T(n-3)+1]+2+1$

$T(n) = 2^3T(n-3)+2^2+2+1$ —— ③

$\vdots$

$T(n) = 2^kT(n-k)+2^{k-1}$
$+2^{k-2}+\dots 2^2+2+1$
—— ④

Assume $n-k=0$

$n=k$

$2^2T(0)+1+2+2^2+\dots 2^{k-1}$

$= 2^2\times 1+2-1$

$= 2^2+2^n-1$

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

Master theorem for decreasing function

$T(n) = aT(n-b)+f(n)$ $\longrightarrow$ General form for RR

$a>0$, $b>0$ and $f(n) = O(n^k)$ where $k \geq 0$

if $a=1 \longrightarrow O(n^{k+1})$
$O(n*f(n))$

if $a>1 \longrightarrow O(n^ka^n)$

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

$T(n) = T(n-1)+n \longrightarrow O(n^2)$

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

$T(n) = 2T(n-1)+1 \longrightarrow O(2^n)$

$T(n) = 3T(n-1)+1 \longrightarrow O(3^n)$

$T(n) = 2T(n-1)+n \longrightarrow O(n2^n)$

$O(n^ka^{n/b})$
$O(f(n)a^{n/b})$

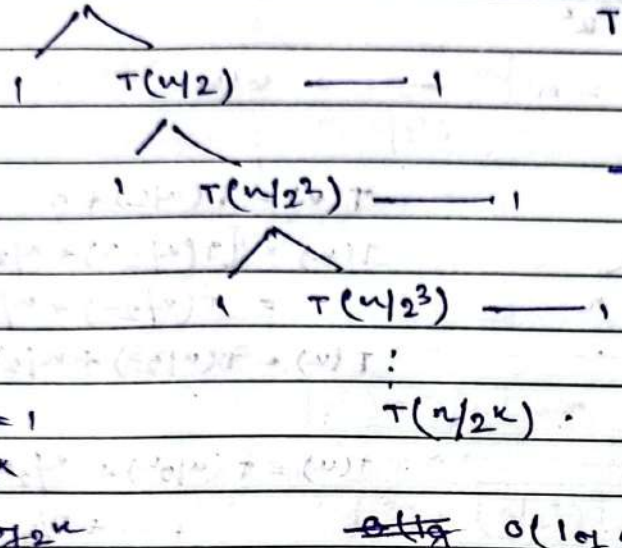
Recurrence Relation Dividing Function:

Tree

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

```
T(n) —— Algorithm Test (int n) {
        if (n>1) {
1 —— printf("%d", n);
T(n/2) —— Test (n/2);
        }
        }
```

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



$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$ ~~O(lg~~ $O(\log n)$

Substitution

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

$T(n) = [T(n/2^2)+1]+1$ —— (1)

$= T(n/2^2)+2$ —— (2)

$T(n) = T(n/2^3)+3$ —— (3)

⋮

$T(n) = T(n/2^k)+k$ —— (4)

Assume, $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$

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

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

$O(\log n)$

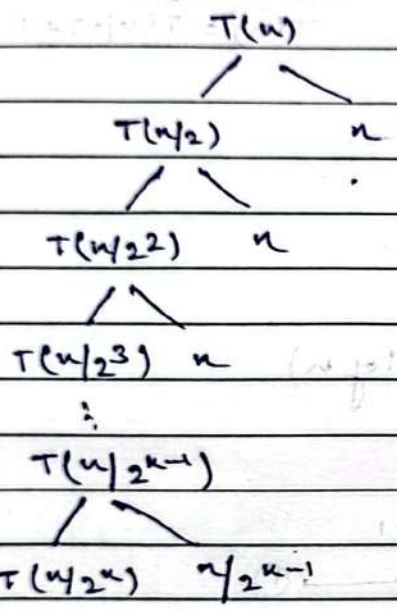
Ans.

Q $T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k}$

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

$$= n \sum_{i=0}^{k} \frac{1}{2^k} \to 1$$

$$= n \times 1 = n \longrightarrow O(n)$$



$T(n) = T(n/2) + n$ —— ①

$T(n) = [T(n/2^2) + n/2] + n$

$= T(n/2^2) + n/2 + n$ —— ②

$T(n) = T(n/2^3) + n/2^2 + n/2 + 2$ —— ③

$\vdots$

$T(n) = T(n/2^k) + n/2^{k-1} + n/2^{k-2}$

$+ \cdots + n/2 + n$ —— ④

$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

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

$T(n) = T(1) + n[1+1]$

$T(n) = 1 + n[1+1] = 1 + 2n$

$T(n) \longrightarrow O(n)$

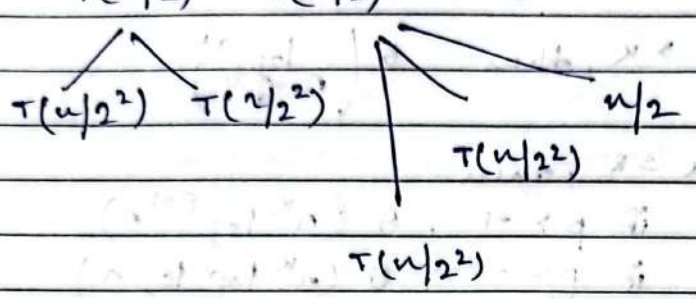
Q

```
void test(int n) {          → T(n)
  if (n>1) {
  for(i=0; i<n; i++){
      stmt;                 } — n
  }
  Test(n/2);                — T(n/2)
  Test(n/2);                — T(n/2)
  }
}
```

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

Page No:
Date:

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



Assume $n/2^k = 1$

$n = 2^k$

$k = \log n$

$\longrightarrow O(n \log n)$

Ans.

$$T(n) = 2T(n/2) + n \quad \text{——— (1)}$$
$$= 2[2T(n/2^2) + n/2] + n$$
$$= 2^2 T(n/2^2) + n + 2 \quad \text{——— (2)}$$
$$= 2^2[2T(n/2^3) + (n/2^2)] + 2n$$
$$= 2^3 T(n/2^3) + 3n \quad \text{——— (3)}$$
$$\vdots$$
$$2^k T(n/2^k) + kn$$

$$T(n/2^k) = T(1)$$
$$\therefore n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$$

$$T(n) = 2^k T(1) + kn$$
$$= n \times 1 + n \log n$$
$$O(n \log n)$$

Ans.

Master Theorem for Dividing functions :-

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

$a > 1$, $b > 1$ $\quad f(n) = \theta(n^k \log^p n)$

Case 1: If $\log_b a > k$, then $\theta(n^{\log_b a})$

Case 2: if $\log_b a = k$, then
- if $p > -1$, $\theta(n^k \log^{p+1} n)$
- if $p = -1$, $\theta(n^k \log \log n)$
- if $p < -1$, $\theta(n^k)$

Case 3: If $\log_b a < k$, if $p \geq 0$, $\theta(n^k \log^p n)$
- if $p < 0$ $\theta(n^k)$

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

$a = 2$, $b = 2$ $\quad \log_2 2 = 1 > k = 0$

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

$= \theta(n^0 \log^0 n)$

$k = 0$, $p = 0$

Q $T(n) = 4T(n/2) + n$ $\quad \log_2 4 = 2 > k = 1$

$a = 4$, $b = 2$

$\rightarrow \theta(n^2)$

Q $T(n) = 8T(n/2) + n$ $\quad \log_2 8 = 3 > k = 1$

$a = 8$, $b = 2$

$\rightarrow \theta(n^3)$

Q $T(n) = 9T(n/3) + 1$ $\quad \log_3 9 = 2 > k = 0$

$a = 9$, $b = 3$

$\rightarrow \theta(n^2)$

Page No:
Date:

Q $T(n) = 9T(n/3) + n^2$

$\log_3^9 = 2 \Rightarrow k = 2 \longrightarrow \Theta(n^2)$

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

$\log_2^2 = 1 \Rightarrow k = 1 \quad p = 0 \longrightarrow \Theta(n \log n)$

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

$\log_2^4 = 2 \quad k = 2 \longrightarrow \Theta(n^2 \log n)$

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

$\log_2^4 = 2 \quad k = 2 \longrightarrow \Theta(n^2 \log^2 n)$

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

$\log_2^2 = 1 \quad k = 1 \quad p = -1 \longrightarrow \Theta(n \log \log n)$

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

$\log_2^2 = 1 \quad k = 1 \quad p = -2 \longrightarrow \Theta(n)$

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

$\log_2^1 = 0 < k = 2 \longrightarrow \Theta(n^2)$

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

$\log_2^2 = 1 < k = 2 \longrightarrow \Theta(n^2 \log^2 n)$

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

$\log_2^4 = 2 < k = 3 \longrightarrow \Theta(n^3)$

Root function

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

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

$T(n) = T(n^{1/2}) + 1$ —— (1)

$T(n) = T(n^{1/2^2}) + 2$ —— (2)

$T(n) = T(n^{1/2^3}) + 3$ —— (3)

⋮

$T(n) = T(n^{1/2^n}) + n$ —— (4)

Assume, $n = 2^m$ ⟶ $m = \log_2 n$

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

Assume,

$T(m/2^k) = T(2^1)$

$\therefore m/2^k = 1$

$\Rightarrow m = 2^k$

$\boxed{k = \log m}$

$k = \log\log_2 n$

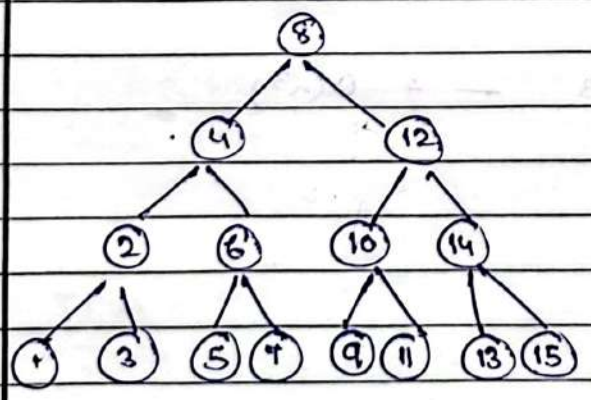
$\theta(\log\log_2 n)$

```
T(n) —— void Test (int n){
        if (n>2){
1 —— stmt;
T(√n) —— Test(√n);
        }
}
```

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

Binary Search ⟶ must be in sorted order

A	3	6	8	12	14	17	25	29	31	36	42	47	53	55	62
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



min ⟶ $O(1)$

max ⟶ $O(\log n)$

```
int BinSearch (A, n, key){
    l=1, h=n;
    while (l ≤ h){
        mid = (l+h)/2;
        if(key == A[mid])
            return mid;
        if(key < A[mid])
            h = mid-1;
        else
            l = mid+1;
    } return 0;
}
```

- Recursive algorithm for binary search

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

```
T(n) —— Algorithm RBinSearch (l,h,key)
{
    if (l==h){
        if (A[l] == key) {
            return l;
        else
            return 0;
    } else {
1 —— mid = (l+h)/2;
1 —— if (key == A[mid])
            return mid;
1 —— if (key < A[mid])
            return RBinSearch
                (l, mid-1, key);
T(n/2) { else
            return RBinSearch
                (mid+1, h, key);
    }
}
```

- Merge Sort

* 2 way merge sort (iterative)

	1	2	3	4	5	6	7	8
A	9	3	7	5	6	4	8	2

(1st) 3,9 5,7 4,6 2,8 → (n)

(2nd) 3579 2468 → (n)

(3rd) 23456789 → (n)

$8 = 2^3 \rightarrow \log_2 8 = 3$

No. of passes = $\log n$

$\Theta(n \log n)$

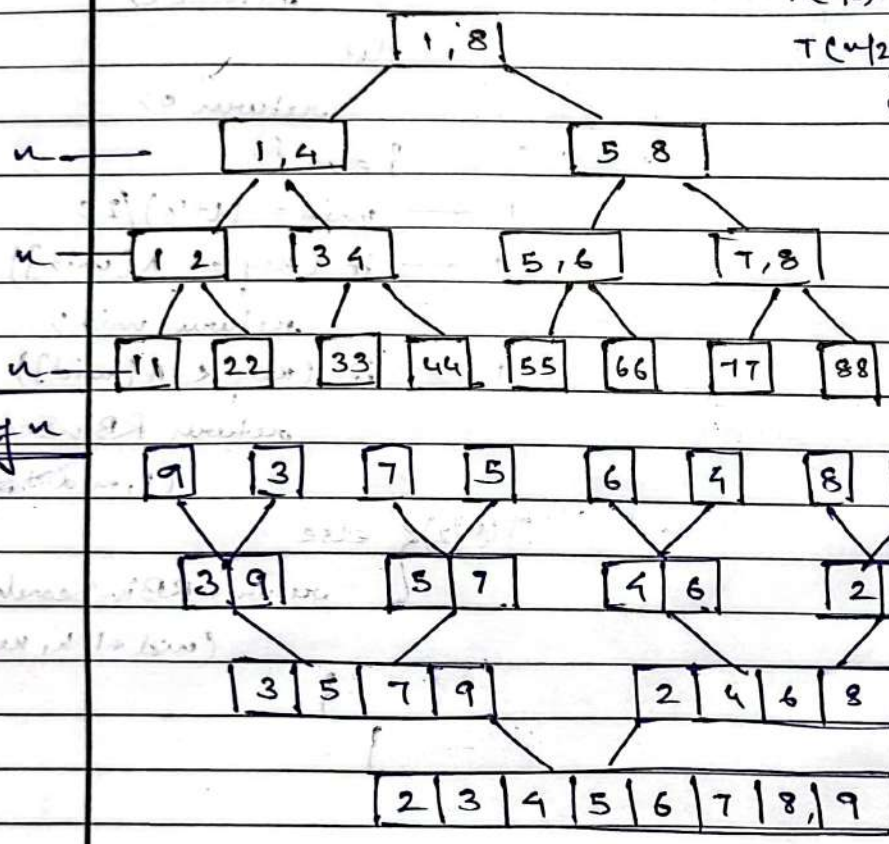
```
Algorithm Merge (A,B,m,n)
{
    i=1; j=1; k=1;
    while (i<=m && j<=n){
        if (A[i] < B[i])
            c[k++] = A[i++];
        else
            c[k++] = B[j++];
    }
    for( ; i<=m; i++)
        c[k++] = A[i];
    for( ; i<=n; j++)
        c[k++] = B[j]
}
```

Merge Sort (Recursive)

	1	2	3	4	5	6	7	8
A	9	3	7	5	6	4	8	2
	l			mid				h

```
T(n) Algorithm MergeSort(l,h){
    if(l<h){
        mid = (l+h)/2;          ①
        MergeSort(l,mid);       T(n/2)
        MergeSort(mid+1,h);     T(n/2)
        Merge(l,mid,h);         n
    }
}
```

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



$n\log n$

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

$a=2, b=2 \quad f(n)=n$

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

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

$\log_b^a = \log_2^2 = 1 \qquad n^k = n^1$

$\log_2^2 = k$

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

- **Pros:**
  - (i) Large size list
  - (ii) linked list
  - (iii) External Sorting Support
  - (iv) Stable
- **Cons**
  - (i) Extra Space (not inplace Sort)
  - (ii) No small problem
  - (iii) Recursive

Quick Sort

A

10	80	30	90	40

Pivot

- Compare 10 with the pivot, and since its less than pivot arrange it accordingly.

10	80	30	90	40

- Compare 80 with the pivot, it is greater than the pivot

10	80	30	90	40

80 > Pivot → Hence No Swap, Move to 30

- Compare 30 with pivot, it is less than pivot so arrange it accordingly.

10	80	30	90	40

30 < pivot → Swap 80 with 30

10	80	30	90	40

- Compare 90 with the pivot, it is greater than pivot

10	30	80	90	40

90 > Pivot → Hence No Swap, End traversal

- Arrange the pivot in its correct position

10	30	80	90	40

Swap 80 & 40 →

PIVOT

10	30	40	90	80

↓

10	30	40	80	90

Page No:
Date:

~~Partition (l,h){~~
~~pivot = arr[h];~~
~~i = l-1;~~

```
Partition (l, h) {
    pivot = A[l];
    i=l; j=h;
    while (i<j){
    do {
        i++;
    } while (A[i] ≤ pivot) {
      do {
          j--;
    } while (A[i] > pivot) {
      if (i<j){
          swap (A[i], A[j]);
      }
    }

QuickSort(l,h) {
  if (l<h) {
      j = Partition (l,h);
    QuickSort (l,j);
    QuickSort (j+1, h);
  }
}
```

Best case → $O(n \log n)$
Worst case → $O(n^2)$

Page No:
Date:

GREEDY METHOD

- Mainly used for optimization problems
  - Greedy method
  - Dynamic Programming
  - Branch and Bound

$n = 5$

a	a_1	a_2	a_3	a_4	a_5
	1	2	3	4	5

```
Algorithm Greedy (a, n)
{
  for i=1 to n do
  {
    x = Select (a);
    if feasible (x) Then
    {
      Solution = solution + x;
    }
  }
```

Knapsack Problem

$n = 7$

$m = 15$

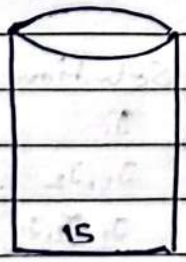
Objects:	1	2	3	4	5	6	7
Profits:	10	5	15	7	6	18	3
Weights:	2	3	5	7	1	4	1
P/w:	5	1.3	3	1	6	4.5	3
	1	2/3	1	0	1	1	1
	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Constraint

$\sum x_i w_i \le m$

Objective

max $\sum x_i P_i$



$15 - 1 = 14$
$14 - 2 = 12$
$12 - 4 = 8$
$8 - 5 = 3$
$3 - 1 = 2$
$2 - 2 = 0$

$= 1\times2 + 2/3\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1$

$= 15$

$\sum x_i w_i = 15$

Profit $= 1\times10 + 2/3\times5 + 1\times15 + 1\times6 + 1\times18 + 1\times3$

$= 54.6$

$\sum x_i P_i = 54.6$

लेकिन optimize करना है cost maximize करना है

Job Sequencing with Deadlines

$n=5$

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1
Deadlines	2	2	1	3	3

$$\underbrace{20 + 15 + 5}_{J_1 \quad J_2 \quad J_4} = 40$$

$$0 \xrightarrow{J_2} 1 \xrightarrow{J_1} 2 \xrightarrow{J_4} 3$$

$\{ J_2 \rightarrow J_1 \rightarrow J_4 \}$

$\{ J_1 \rightarrow J_2 \rightarrow J_4 \}$

$0 \text{ —— } 1 \text{ —— } 2 \text{ —— } 3$

Jobs Consider	Slot Assign	Solution	Profit
—	—	ϕ	0
J_1	[1, 2]	J_1	20
J_2	[0,1] [1,2]	J_1, J_2	20+15
J_3	[0,1] [1,2]	J_1, J_2	20+15
J_4	[0,1] [1,2] [2,3]	J_1, J_2, J_4	20+15+5
J_5	[0,1] [1,2] [2,3]	J_1, J_2, J_4	40

$n=7$

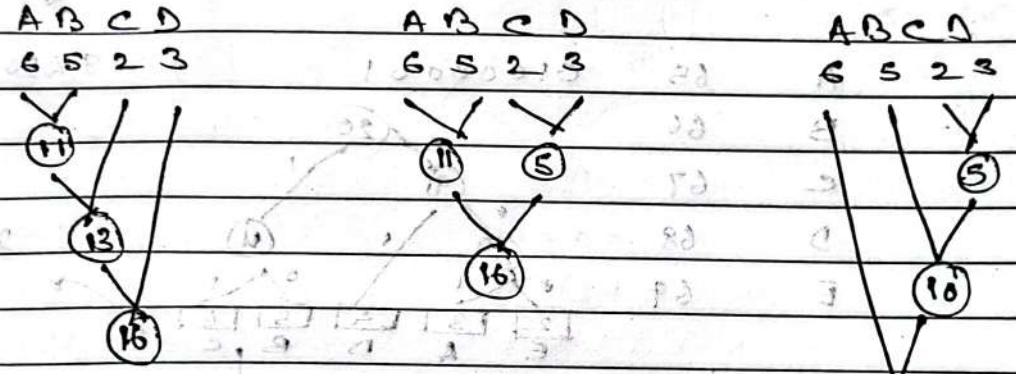
Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profits	35	30	25	20	15	12	5
Deadlines	3	4	4	2	3	1	2

$$0 \underset{20}{\overset{J_4}{—}} 1 \underset{25}{\overset{J_3}{—}} 2 \underset{35}{\overset{J_1}{—}} 3 \underset{30}{\overset{J_2}{—}} 4 \rightarrow 110$$

Jobs Consider	Slot Assign	Solution	Profit
J_1	[2,3]	J_1	35
J_2	[2,3][3,4]	J_1, J_2	35+30
J_3	[2,3][3,4][1,2]	J_1, J_2, J_3	35+30+25
J_4	[2,3][3,4][1,2][0,1]	$J_1 J_2 J_3 J_4$	35+30+25+20
J_5	"	"	110
J_6	"	"	110
J_7	"	"	110

Optimal Merge Pattern

List → A B C D
Size → 6 5 2 3

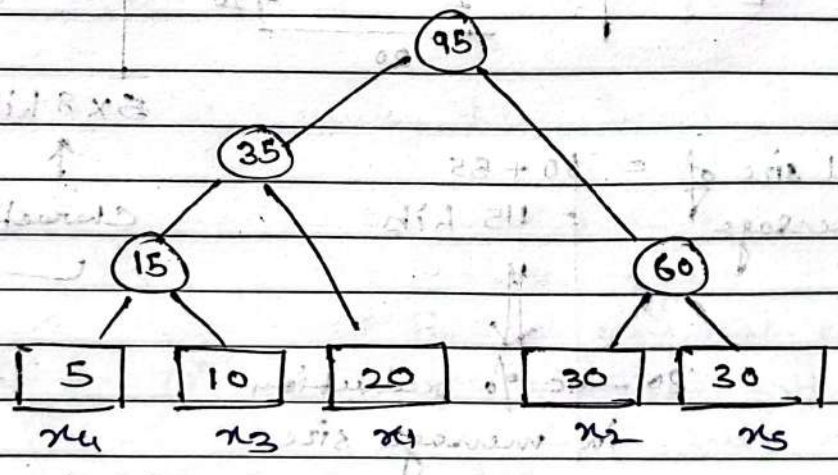


$11+13+16 = 40$

$11+5+16 = 32$

$5+10+16 = 31$

Lists →	x_1	x_2	x_3	x_4	x_5
sizes →	20	30	10	5	30



$15+60+35+95 = 205$

Total cost of merging $= \sum d_i * x_i$

Huffmann Coding

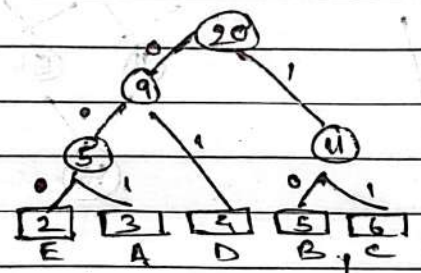
Message → BCCABBDDAECCBBAEDDCC

length → 20

ASCII → 8 bits

A	65	01000001
B	66	
C	67	
D	68	
E	69	

$8 \times 20 = 160$ bits



$20 \times 3 = 60$ bits

Character	Count/Frequency		Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		

5×8 bits ↑ Characters

5×3 ↑ Codes

$40 + 15 = 55$ bits

Total size of message $= 60 + 55 = 115$ bits

⇓

30 – 40 % reduction in message size.

Char	Count	Code	
A	3	001	$3 \times 3 = 9$
B	5	10	$5 \times 2 = 10$
C	6	11	$6 \times 2 = 12$
D	4	01	$4 \times 2 = 8$
E	2	000	$2 \times 3 = 6$
$8 \times 5 = 40$ bits	20	12 bits	45 bits

msg → 45 bits

tree table → 52 bits

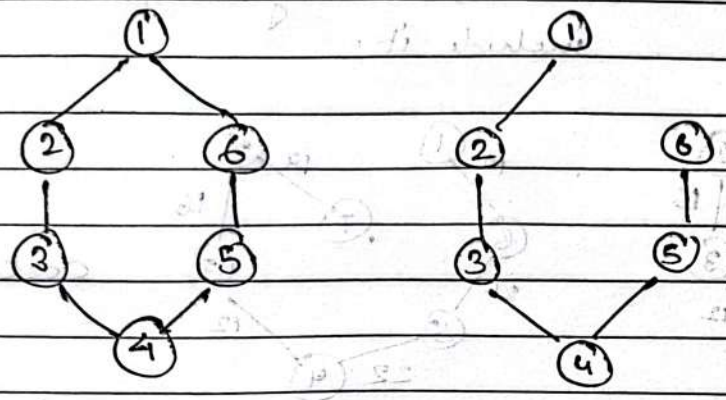
97 bits

$\sum d_i * f_i$

$= 3 \times 2 + 3 \times 3 + 2 \times 4 + 2 \times 5 + 2 \times 6$

$= 45$ bits

Minimum Cost Spanning Tree



$S \subseteq G$

$S = (V' E')$

$V' = V$

$|E'| = |V| - E$

$G = (V, E)$

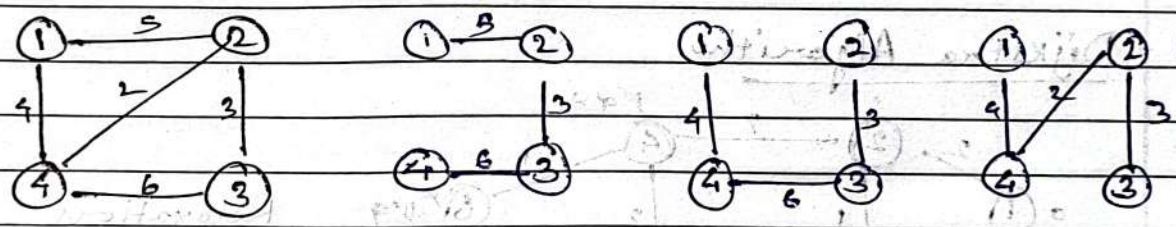
$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1,2)(3,4) \ldots\}$

$|V| = n = 6$

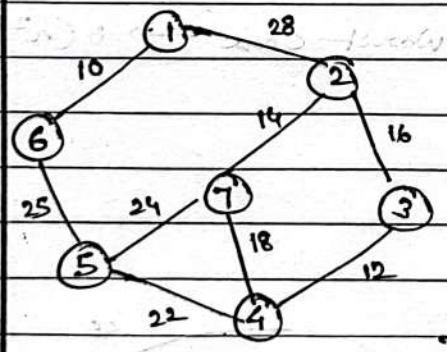
$|V| - 1 = 5$

Q

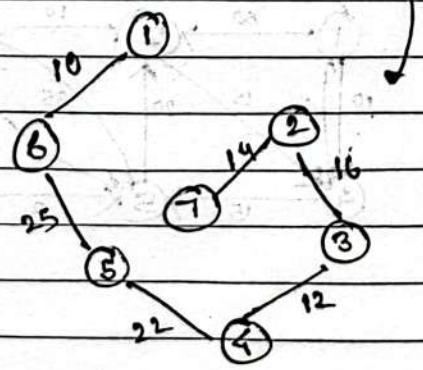


cost = 14 cost = 13 cost = 9

Minimum Cost of Spanning tree through Prims Algorithm



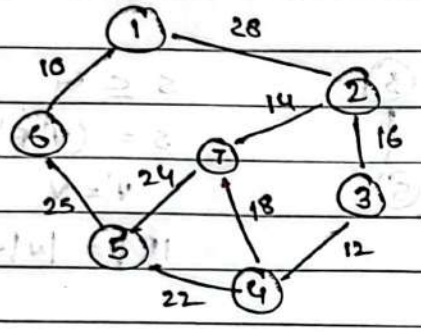
* Initialize an MST with randomly chosen vertex
* Find all edges that connects the tree with new vertices. From the edges found, select min. edge and add to the tree

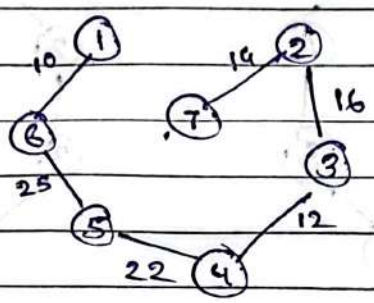


Cost = 99

Kruskal's Method

— Always select the minimum cost but if its forming a cycle don't include it.



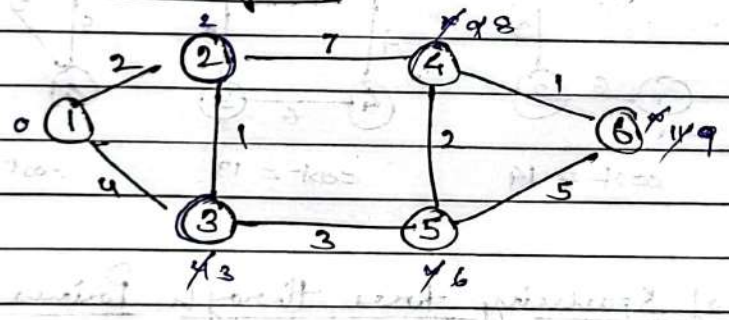


Cost = 99

$\theta(|V||E|)$

$\theta(n \cdot e) = \theta(n^2)$

Dijkstra Algorithm



Relaxation

if $(d[u] + c(u,v) < d[v])$

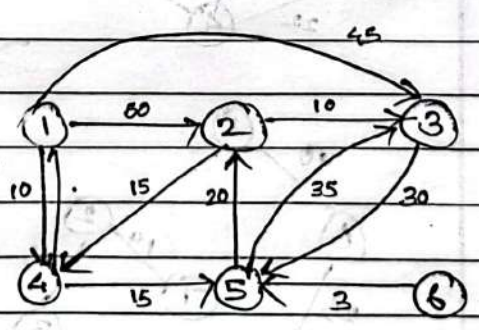
$d[v] = d[u] + c(u,v)$

v	d[v]
2	2
3	3
4	8
5	6
6	8

Worst Case $\longrightarrow \theta(n^2)$

Starting vertex = 1

Selected vertex	2	3	4	5	6
1	50	45	10	∞	∞
5	50	45	10	25	∞
2	45	45	10	25	∞
3	45	45	10	25	∞
6	45	45	10	25	∞



Dynamic Programming

It is an algorithm design method used for solving problems whose solutions are viewed as a result of making a set / sequence of decisions

One way of making these decisions is to make them one at a time in a step wise manner and never make an erroneous decision.

i) Enumerate all decision sequences (brute force) and then pick the best solution (optimal)
Drawback - Excessive time/space requirements

ii) DP based on enumeration often tries to reduce the amount of enumeration by curtailing those decision sequence from where there is no sense of possibility of getting optimal solution.

In DP these set of optimal decisions are made by applying principles of optimality.

⇓

States that whether the initial state and decision are the remaining sequence of decision must constitute an optimal decision sequence with regard to the state resulting from the first decision.

* Diff. b/w Greedy and DP is that Greedy always generates only one decision sequence whereas in DP enumeration many sequence decisions can be generated.

iii) One important feature of DP is that optimal solution of the sub problems are retained (cached / stored in the table) to avoid recomputing their values.

* DP Implementation → (i) Memoization (top/down) → Recursive
→ (ii) Tabulation (Bottom-up) → Iterative

Properties of DP

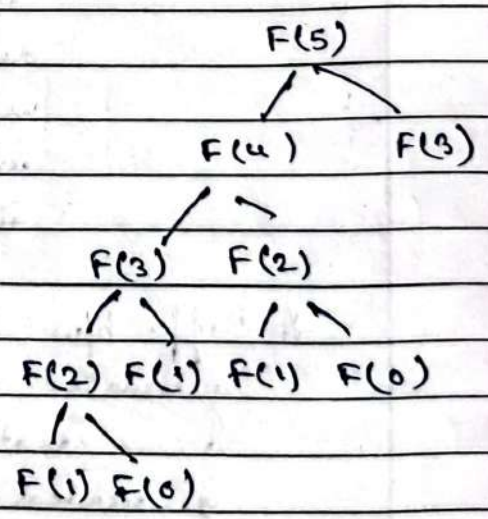
(i) Splitting of original problems into subproblem
(ii) Subproblems optimality
(iii) Overlapping subproblems

(Ex) Computing n^{th} fibonacci numbers
Fibonacci Number
Fib series : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$Fib(n) = Fib(n-1) + Fib(n-2) \quad , n > 1$$
$$Fib(n) = 1 \quad , n = 1$$
$$Fib(n) = 0 \quad , n = 0$$

Normal recursive implementation of Fib(n) → menFib

```
Algo Fib(int n)
{
  if (n ≤ 1) return (n);
  else
  {
    return(Fib(n-1) + Fib(n-2));
  }
}
```



$$T(n) = c \quad , n \le 1$$
$$= T(n-1) + T(n-2) + a \quad , n > 1$$

→ Decrease / Conquer

$$T(n) = T(n-1) + T(n-2) + a \quad \text{——— (I)}$$
$$T(n-2) < T(n-1)$$

$$\therefore T(n) < T(n-1) + T(n-2) + a$$
$$T(n) < 2T(n-1) + a \quad \text{——— (II)}$$

$$\boxed{T(n) < 2^n}$$

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

* Top down - memoized implementation of fib(n)

m[0 . . . . n]

```
Algo memFib(n)
{
  if (m[n] is undefined)
  {
    if (n<=1) result = n;
    else
    result = memFib(n-1)
             + memFib(n-2);
  m[n] = result; // memoizing
  }
  return (m[n]);
}
```

	0	1	2	3	4	5
m	0	1	1	2	3	5

memFib(5) res = 3+2 = 5

- → memFib(4) res = 3
  - → memFib(3) res = 2+1
    - → memFib(2) res = 1
      - → memFib(1) = 1
      - → memFib(0) = 0
    - → memFib(1) = 1
    - → memFib(2) = 1
- → memFib(3) = 2

time : O(n)
Recursion Stack : O(n)

most commonly used because recursion, stacks not used

Page No:
Date:

Bottom up approach of DP for fib(n)

```
Algo Memfib(n)
{
  M[0] = 0
  M[1] = 1
  for i = 2 to n
  {
    M[i] = M[i-1] + M[i-2];
  }
  return (M[n]);
}
```

	0	1	2	3	4	5
M	0	1	1	2	3	5

time: O(n)

- When to use tabulation and when memoization:
  - (i) If the original problem requires all subproblems to be solved, tabulation outperforms memoization.
  - (ii) Tabulation has no overhead of Recursion and can use preallocated array.
  - (iii) If only some of the subproblems need to be solved in the original problem, then memoization technique is preferable because the subproblems are solved lazily.

Greedy: Building up of the solution to the problem is done in a stepwise manner (incrementally) by applying local options only (local optimality)

Divide & Conquer: Breaking up a problem into separate problems (independent) then solve each subproblem separately (independently) and combine the solution of the subproblem to get the solution of original problem.

Page No:
Date:

Dynamic Programming: Breaking up of a problem into a series of overlapping subproblems and building up solution of larger and larger subproblems.

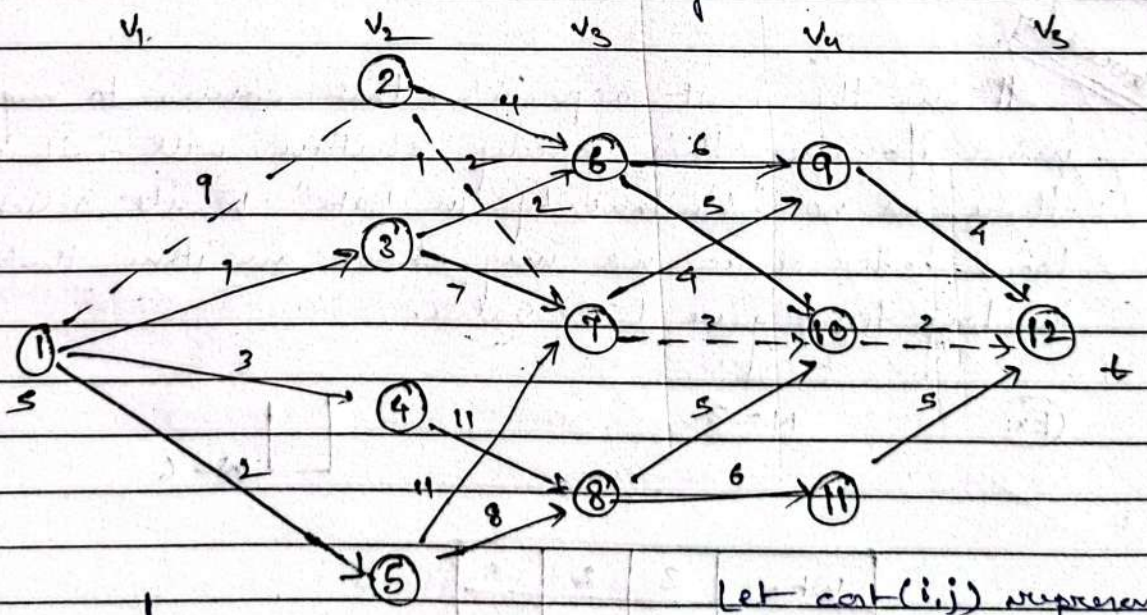
Unlike divide and conquer, DP typically involves solving all subproblems rather than a small portion of a subproblem.

DP tends to solve each subproblem only once since the results of subproblems are stored which are used later again when required. This is going to reduce the computation drastically

Ex - Fib
- Brute Force : $O(2^n)$
- DP Implementation : $O(n)$

Multistage Graph $\Longrightarrow G = (V, E)$ $|V| = n$, $|E| = e$

$\| L$ stage $L > 2 \|$



c	1	2	3	.....	12
1	-	9	7		
2					
3					
⋮					
12					

$c(i,j)$ = edge cost

Let cost(i,j) represent cost of the path from vertex j present in stage i to reach destination vertex t

cost(1,1)

0/1 Knapsack Problem

A thief wants to rob a store, he is carrying a bag of capacity w. The store has 'n' items. Its weight is given by the 'wt' array and its value by the 'val' array. He can either include an item in its knapsack or exclude it but partially have it as a fraction. We need to find the max. value of items that the thief can steal.

N = 4 W = 5

Wt.	1	2	4	5
Val.	5	4	8	6

Answer = 13

The thief can select items valued 8 and 5 with weight 4 and 1 respectively

Greedy.

It is the first approach that comes to ~~my~~ our mind. However this greedy solution will fail since there is no "Uniformity in Data". While selecting a local better choice we may choose an item that will in long term give less value.

(Ex) N = 3 W = 6

Wt.	3	2	5
Val.	30	40	60

* A greedy solution will take the most valuable item first so we will take an item on index 2, with a value of 60 and put in the knapsack. Now the remaining capacity will be 1. Therefore we cannot add any more item. So the greedy item gives 60.

∴ A non greedy approach → taking the first two items will give us the value of 70 (30 + 40) in the given capacity of the Knapsack.

Now greedy

① Express the problem in terms of indexes

We are given n items. Their weight is represented by the 'wt' array and the value by the 'val' array. So clearly one parameter will be 'ind' i.e. index upto which the array items are being considered.

There is one more parameter 'w'. We need the capacity of the Knapsack to decide whether we can pick an array item or not in the Knapsack.

So initially we need to find $f(n-1, w)$ where
w = Overall capacity (given)

$f(n-1, w)$ → means we are finding the maximum value of items that the thief can steal from items with index 0 to n-1 capacity w of the Knapsack.

$f(ind, w)$ → Max value of items from index 0 to ind with capacity of the Knapsack w

Base Case: if ind==0

It means we are at the first item, so in this case we'll check whether the items capacity is less than or equal to the current capacity w. If it is, we simply return its value (val[0]) else we return 0.

```
f(ind, w) {
    if (ind == 0) {
        if (wt[0] <= w)

        return 0
    }
}
```

② Try out all choices possible at a given index

```
f(ind, w) {
    if (ind == 0) {
        if (wt[0] <= w)
            return val[0]
        return 0
    }
    notTake = 0 + f(ind-1, w)
    take = INT_MIN
    if (wt[ind] <= w)
        take = val[ind] + f(ind-1, w - wt[ind])
}
```

Final Pseudocode

```
f(ind, w) {
    if (ind == 0) {
        if (wt[0] <= w)
            return val[0]
        return 0
    }
    notTake = 0 + f(ind-1, w)
    take = INT_MIN
    if (wt[IND] <= w)
        take = val[ind] + f(ind-1, w - wt[ind])
    return max(take, notTake)
}
```

Tabulation Approach

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it to 0.

(i) At ind == 0, we are considering the first element, if the capacity of the Knapsack is greater than the weight of the first item, we return val[0] as answer. This can be achieved using for loop.

(ii) Next, we are done with the first row, so our "ind" variable will move from 1 to n-1 whereas our 'cap' variable will move from 0 to 'w'. We will set the nested loops to traverse the dp array.

(iii) Inside the nested loop we apply recursive logic to find the answer of the call.

(iv) When nested loop execution has ended, we ~~apply recursive logic~~ return dp[n-1][w] as answer

```
int Knapsack( int wt, vector <int> val, int n, int w)
        dp(n, vector<int> (w+1), 0));
        for ( int i = wt[0] ; i <= w ; i++){
                dp [0][i] = val[0];
        }

        for ( int ind = 1 ; ind < n; ind++){
                for ( int cap = 0; cap <= w; cap++){
                        int notTaken = dp [ind-1] [cap];
                        int taken = INT_MIN;

                        if (wt[ind] <= cap){
                                taken = val[ind] + dp [ind-1]
                                        [cap - wt[ind]];
                        }
```

```
            dp[ind][cap] = max(notTaken, taken);
        }
    }
    return dp[n-1][w];
}
```

Ex weight[] = {1,2,3} profit[] = {10,15,40} Capacity = 6

- If no element is filled, then the possible profit is 0

weight → item ↓	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

- For filling the first item in the bag; if we follow the above mentioned procedure, the table will look something like the following

weight → item ↓	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2							
3							

- For filling the second item in the bag,

  When jth weight = 2, max possible profit
  = max(10, DP[1][2-2] + 15) = max(10,15) = 15

  When jth weight = 3, max possible profit
  = max(2 not put, 2 is put into bag) =
  = max(DP[1][3], 15 + DP[1][3-2])
  = max(10,25) = 25

weight → item ↓	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- for filling the third item

When jth weight = 3, max possible profit
$= \max(DP[2][3], 40 + DP[2][3-3])$
$= \max(25, 40) = 40$

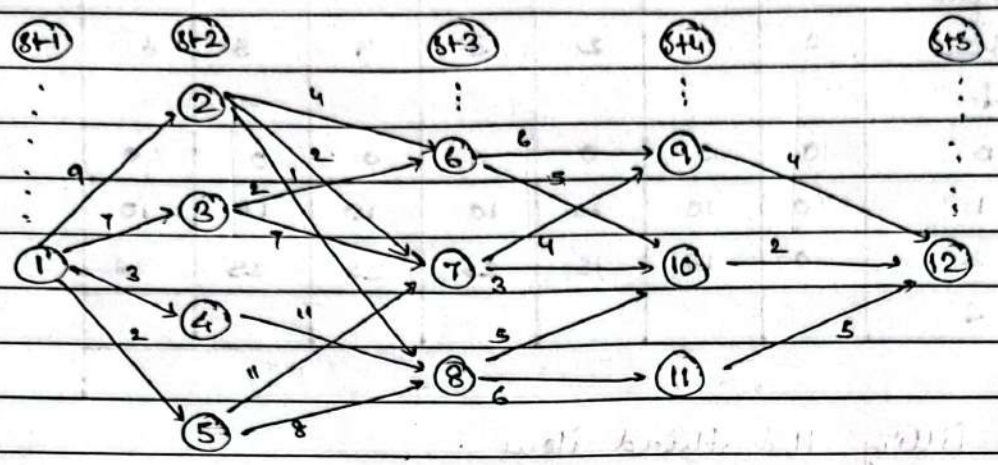
When jth weight = 4, max possible profit
$= \max(DP[2][4], 40 + DP[2][4-3])$
$= \max(25, 50) = 50$

When jth weight = 5, max possible profit
$= \max(DP[2][5], 40 + DP[2][5-3])$
$= \max(25, 55) = 55$

When jth weight = 6, max possible profit
$= \max(DP[2][6], 40 + DP[2][6-3])$
$= \max(25, 65) = 65$

weight → item ↓	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Multistage Graph using DP



Vertex	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
Distance	2/3	7	6	8	8	10	10	10	12	12	12	12

We start from destination and go towards source

cost (stage, vertex)

cost (5,12) = 0

cost (4,9) = 4

cost (4,10) = 2

cost (4,11) = 5

cost (3,6) = min[c(6,9) + cost(4,9), c(6,10) + cost(4,10)]

= min(6+4, 5+2) = min(10,7) = 7

cost (3,7) = min[c(7,9) + cost(4,9), c(7,10) + cost(4,10)]

= min(4+4, 4+2) = min(8,6) = 6

cost (3,8) = min[c(8,10) + cost(4,10), c(8,11) + cost(4,11)]

= min(5+2, 6+5) = min(7,11) = 7

cost (2,2) = min[c(2,6) + cost(3,6), c(2,7) + cost(3,7), c(2,8) + cost(3,8)]

= min(4+7, 2+5, 1+7) = 7

cost (2,3) = min[c(3,6) + cost(3,6), c(3,7) + cost(3,7)]

= min[2+7, 7+5]

= 9

Page No: .....
Date: .....

Framing the formula,

$$cost(\underset{\text{stage}}{i}, \underset{\text{vertex}}{j}) = \min_{\substack{\langle j, l \rangle \in E \\ l \in stage(i+1)}} \{ c(j, l) + cost(i+1, l) \}$$

We start from source to destination

$\longrightarrow$ Forward method

$1 \longrightarrow 2 \longrightarrow 7 \longrightarrow 10 \longrightarrow 12$

$1 \longrightarrow 3 \longrightarrow 6 \longrightarrow 10 \longrightarrow 12$

Multiple paths are possible

cost (minimum) will be same.

Matrix Chain Multiplication

(I) MATRIX MULTIPLICATION

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}_{\substack{2\times 3 \\ m\times n}} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}_{3\times 2}$$

$$C = A \times B$$

Condition:

$$n = o$$

Dimension of resultant matrix $(C) = m \times p$
$= 2 \times 2$

- Total no. of multiplications performed to get values in C?

$$C = \begin{bmatrix} a_{11}\times b_{11} + a_{12}\times b_{21} + a_{13}\times b_{31} & a_{11}\times b_{12} + a_{12} b_{22} + a_{13}\times b_{32} \\ a_{21}\times b_{11} + a_{22}\times b_{21} + a_{23}\times b_{31} & a_{21}\times b_{12} + a_{22} b_{22} + a_{23} b_{32} \end{bmatrix}$$

total multiplications $= 12 = 2 \times 3 \times 2$
$= m \times n \times p$
$= d_0 \times d_1 \times d_2$
(total 3 dimensions)
– for multiplying 2 matrices

(II) MATRIX CHAIN MULTIPLICATION

$$A_1 * A_2 * A_3$$
$$(2\times 3) \quad (3\times 4) \quad (4\times 2)$$
$$(d_0 \times d_1) \quad (d_1 \times d_2) \quad (d_2 \times d_3)$$

$(A_1 \times A_2) \times A_3$ $\qquad$ $A_1 \times (A_2 \times A_3)$

(Result will be same but the no. of multiplications will be different)

Page No: .....
Date: .....

$((2\times3)(3\times4))\times(4\times2)$

Total = $2\times3\times4 = 24+0$

$(2\times4)\times(4\times2)$

Total = $2\times4\times2 = 16$

$\boxed{\text{Total} = 24+16 = 40}$

$(2\times3)\times((3\times4)\times(4\times2))$

Total = $0 + 3\times4\times2$

$= 24$

$(2\times3)\times(3\times2)$

Total = $2\times3\times2 = 12$

$\boxed{\text{Total} = 24+12 = 36}$

** Cost of multiplication is less for : $A_1 \times (A_2 \times A_3)$

Principle ⟹ First find for what parenthesis cost will be less, then do the actual multiplication

(III) DYNAMIC PROGRAMMING APPROACH

→ Explore all posible ways of parenthesization
→ Pick the minimum

$A_1 \times A_2 \times A_3$

(2×3) (3×4) (4×2)

$(d_0\times d_1)$ $(d_1\times d_2)$ $(d_2\times d_3)$

$(A_1\times A_2)\times A_3$

$c[1,2] = 24$ $c[3,3]=0$

(2×4) (4×2)

i k $k+1$ j d_{i-1} k d_j

$c[1,3] = c[1,2] + c[3,3] + d_0\times d_2\times d_3$

$c[1,3] = 24 + 0 + 2\times4\times2$

$c[1,3] = 40$

$A_1\times(A_2\times A_3)$

$c[1,1]$ $c[2,3] = 24$

(2×3) (3×2)

$c[1,3] = c[1,1] + c[2,3] + d_0\times d_1\times d_3$

$c[1,3] = 0 + 24 + 12$

$c[1,3] = 36$

$$c[i,j] = \min\{c(i,k) + c(k+1,j) + d_{i-1}\times d_k\times d_j\}$$

where $i \le k < j$

$$c[1,4] \underset{i\ j}{\min} \begin{cases} c[1,1] + c[2,4] + d_0 d_1 d_4 & K=1 \\ c[1,2] + c[3,4] + d_0 d_2 d_4 & K=2 \\ c[1,3] + c[4,4] + d_0 d_3 d_4 & K=3 \end{cases} \begin{matrix} \rightarrow A(BCD) \\ \rightarrow (AB)(CD) \\ \rightarrow (ABC)D \end{matrix} \quad 1 \le K < 4$$

$$c[2,4] = \min_{i\ j} \begin{cases} c[2,2] + c[3,4] + d_1 d_2 d_4 \\ c[2,3] + c[4,4] + d_1 d_3 d_4 \end{cases} \begin{matrix} \rightarrow B(CD) \\ \rightarrow (BC)D \end{matrix} \quad 2 \le K < 4$$

Since we are solving same sub problems repeatedly, we should arrange to solve each problem just once.

(method)

* Make 2 matrices
  - i) For storing minimum cost
  - ii) for storing respective values of (S)

(m)

	1	2	3	4
1	0			
2		0		
3			0	
4				0

(S)

	1	2	3	4
1				
2				
3				
4				

$$\underbrace{A_1}_{d_0} \times \underbrace{A_2}_{d_1} \times \underbrace{A_3}_{d_2} \times \underbrace{4}_{d_3}$$

$$\underbrace{3}_{d_0} \cdot \underbrace{2 \quad 2}_{d_1} \quad \underbrace{4 \quad 4}_{d_2} \quad \underbrace{2 \quad 2}_{d_3} \quad \underbrace{5}_{d_4}$$

$$c[1,2] = \min \left\{ c[1,1] + c[2,2] + d_0 d_1 d_2 \right. = 0 + 0 + 3 \times 2 \times 4 = 24 \quad K=1$$

$$c[2,3] = \min \left\{ c[2,2] + c[3,3] + d_1 d_2 d_3 \right. = 0 + 0 + 2 \times 4 \times 2 = 16$$

(m)

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

(S)

	1	1	3
		2	3
			3

Page No:
Date:

(iv) No. OF POSSIBLE WAYS OF PARENTHESIZATION

(i) If 3 matrices – 2 ways (A(BC)) & ((AB)C)

(ii) If 4 matrices – A B C D

- A((BC)D)
- A(B(CD))
- (AB)(CD)
- (A(BC))D
- ((AB)C)D

} 5 ways

(iii) If n matrices

$$\text{No. of ways} = \left(\frac{2n!}{(n+1)!\,n!}\right)$$

where n = No. of matrices − 1

* Using formula,

(i) 5 matrices, $n = 5-1 = 4$

$$\text{No. of ways} = \frac{2\times 4!}{(4+1)!\,4!} = \frac{8!}{5!\,4!}$$

$$= \frac{8\times 7\times 6\times 5!}{5!\times 4\times 3\times 2\times 1}$$

$$= 14$$

(ii) 4 matrices, $n = 4-1 = 3$

$$\text{No. of ways} = \frac{2\times 3!}{(3+1)!\,3!} = 5$$

(v) FINDING MINIMUM USING FORMULA

$$c[i,j] = \min_{i\le k<j}\left[c(i,k) + c(k+1,j)\right] + d_{i-1}\,d_k\,d_j$$

$$\underset{(d_0\times d_1)}{A_1} \times \underset{(d_1\times d_2)}{A_2} \times \underset{(d_2\times d_3)}{A_3} \times \underset{(d_3\times d_4)}{A_4}$$

$$c[3,4] = \min_{k=3} \{ c[3,3] + c[4,4] + d_2 d_3 d_4 = 0 + 0 + 4\times2\times5 = 40$$

$$c[1,3] = \min \begin{cases} k=1 & c[1,1] + c[2,3] + d_0 d_1 d_3 = 0 + 16 + 3\times2\times2 = 28 \\ k=2 & c[1,2] + c[3,3] + d_0 d_2 d_3 = 24 + 0 + 3\times4\times3 = 48 \end{cases}$$

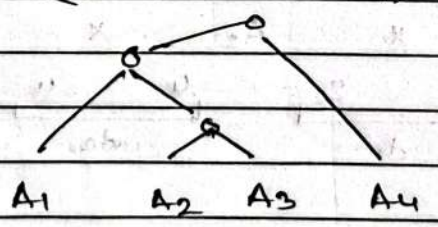
$$c[2,4] = \min \begin{cases} k=2 & c[2,2] + c[3,4] + d_1 d_2 d_4 = 0 + 40 + 2\times4\times5 = 80 \\ k=3 & c[2,3] + c[4,4] + d_1 d_3 d_4 = 16 + 0 + 2\times2\times5 = 36 \end{cases}$$

$$c[1,4] = \min \begin{cases} k=1 & c[1,1] + c[2,4] + d_0 d_1 d_4 = 66 \\ k=2 & c[1,2] + c[3,4] + d_0 d_2 d_4 = 124 \\ k=3 & c[1,3] + c[4,4] + d_0 d_3 d_4 = 58 \end{cases}$$

For parenthesizing, refers (8) table

For (1,4) → value is (3) → parenthesize at (3)
$(A_1\ A_2\ A_3)\ A_4$

For (1,3) → value is (1) → parenthesize at (1)
$((A_1)(A_2\ A_3)(A_4))$



Time complexity

Time to fill the table = $O(n^2)$
Time to fill each cell = finding all possible (k)
$= O(n)$

Total $= O(n^3)$

LONGEST COMMON SUBSEQUENCE

(Ex) String 1 : a b c d e f g h i
String 2 : c d g i

* Characters of string 2 appear in string 1 in increasing sequential order, gaps maybe present

$\boxed{LCS = cdgi}$

(Ex) String 1: a b c d e f g h i j
String 2: c c d g i
Subsequence 1 : cgi
Subsequence 2: cdgi

$\boxed{LCS = cdgi}$

(Ex) String 1: abdace
String 2: babce
Subsequence 1: bace
Subsequence 2: abce

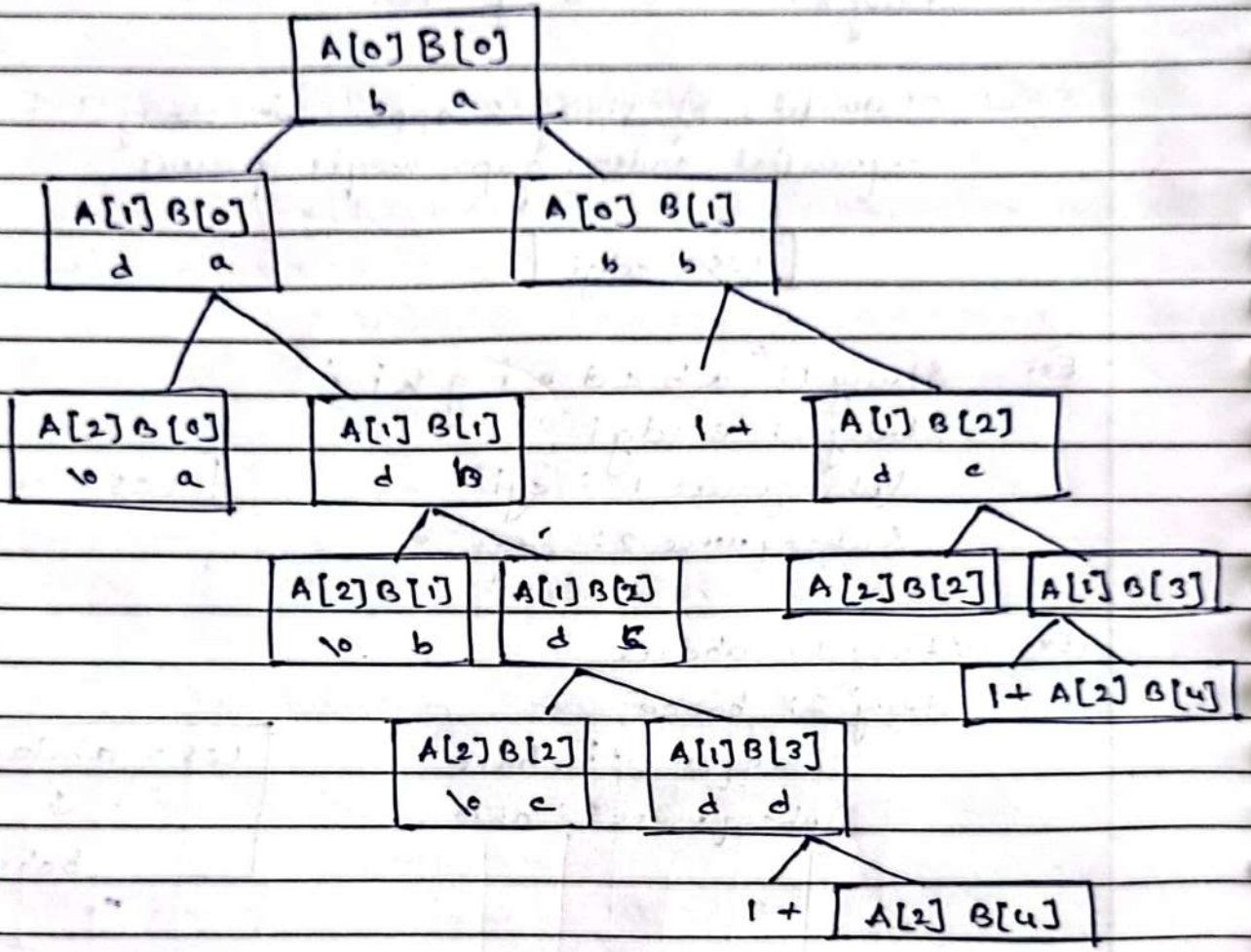
$\boxed{\begin{array}{c} LCS = abdace \\ \downarrow \\ babce \end{array}}$

LCS Using Recursion

```
int LCS (i, j)
{
    if (A[i] == '\0' || B[j] == '\0')
        return 0;
    else if (A[i] == B[i])
        return (1 + LCS (i+1, j+1));
    else
        return max (LCS(i+1, j), LCS(i, j+1));
}
```

	0	1	2
A	b	d	\0

	0	1	2	3	4
B	a	b	c	d	\0



We can do a memoization to reduce recursive calls

LCS using dynamic programming

if (A[i] = B[j])

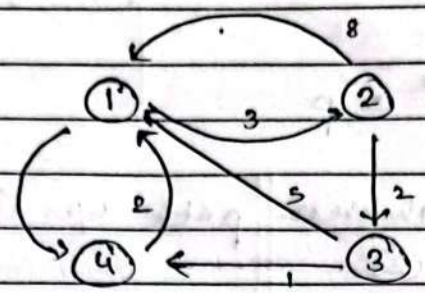
LCS [i,j] = 1+ LCS [i-1, j-1]

else

LCS [i,j] = max [LCS(i-1, j), LCS(i, j-1)]

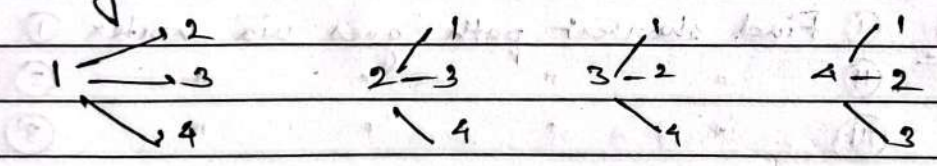
		0	1 (a)	2 (b)	3 (c)	4 (d)
	0	0	0	0	0	0
b	1	0	0	1	1	1
d	2	0	0	1	1	2

All pair shortest Path



$$A^0 \text{ Direct path} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

Starting vertex : 1 SV : 2 SV : 3 SV : 4



Dijkstra Algo – Single Pair Shortest Path

If we apply Dijkstra to all vertices

↳ We can get all pair shortest path

Time complexity of Dijkstra → $O(E \log V)$

Applying to all vertices → $O(V \cdot E \log V)$

for dense graphs : $E = V^2$

$\Rightarrow O(V \cdot V^2 \log V)$

$= O(V^3 \log V)$

Let a graph has vertices $(V) = \{1, 2, \dots n\}$

Let i, j be 2 vertices $\in V$

Let shortest path b/w i, j be $(d_{ij}) =$ "p"

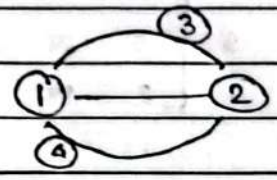
let shortest path b/w (i,j) be via 'k' vertices
$\rightarrow \{1, 2, 3, \ldots k\}$

$$(d_{ij})^k = p$$

<u>Concept:</u> Find the shortest path b/w ① and ②

① —— ②

A direct path may exist or a shorter path via ③ may also exist or via ④



<u>Steps</u> (i) Find shortest path goes via vertex ①
(ii) " " " " ②
(iii) " " " " ③

(vertex ① is intermediate vertex)

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

$A^0[2,3]$ $\quad A^0[2,1] + A^0[1,\infty]$
$2 < 8 + \infty$

$A^0[2,4]$ $\quad A^0[2,1] + A^0[1,4]$
$\infty > 8 + 7 = 15$

$A^0[3,2]$ $\quad A^0[3,1] + A^0[1,2]$
$\infty \quad 5 \quad 3$
$\infty > 8$

$A^0[3,4] = A^0[3,1] + A^0[1,4]$
$1 < 5 + 7$

$A^0[4,2]$ $\quad A^0[4,1] + A^0[1,2]$
$\infty \quad 2 \quad 3$
$\infty > 5$

$A^0[4,3]$ $\quad A^0[4,1] + A^0[1,2]$
$\infty \quad 2 \quad \infty$

Page No:
Date:

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

A^2 should be generated from A^1

2nd column & 2nd row remains same

$A^1[1,3] > A^1[1,2] + A^1[2,3]$

$\infty > 3 \quad 2$

$A^1[1,4] = A^1[1,2] + A^1[2,4]$

$7 < 3 \quad 15$

$A^1[3,4] = A^1[3,2] + A^1[2,4]$

$1 < 8 + 15$

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

$5 < 8 + 8$

$A^1[4,1] \quad A^1[4,2] + A^1[2,1]$

$2 < 5 + 8$

$A^1[4,3] \quad A^1[4,2] + A^1[2,3]$

$\infty > 5 + 2$

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

3rd row & 3rd column remains as generated A^3 from A^2

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

$3 < 5 + 8$

$A^2[1,4] = A^2[1,3] + A^2[3,4]$

$7 > 5 + 1$

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

$8 < 2 + 5$

$A^2[2,4] = A^2[2,3] + A^2[3,4]$

$15 > 2 \quad 1$

$A^2[4,1] = A^2[4,3] + A^2[3,1]$

$2 < 7 + 5$

$A^2[4,2] = A^2[4,3] + A^2[3,2]$

$5 < 7 \quad 8$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 3 & 6 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$A^3[1,2] = A^3[1,4] + A^3[4,2]$ || $A^3[2,1] = A^3[2,4] + A^3[4,1]$

$3 < 6+5$ || $7 > 3+2$

(A^4) is shortest path b/w all pair of vertices

■ Generation of formulae

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

• Generation of pseudocode

```
for (k=1; k<=n; k++)
{
    for (i=1; i<=n; i++)
    {
        for (j=1; j<=n; j++)
            A[i,j] = min(A[i,j], A[i,k] + A[k,j]);
    }
}
```

generate matrix

Single instance of A[i,j] is updated

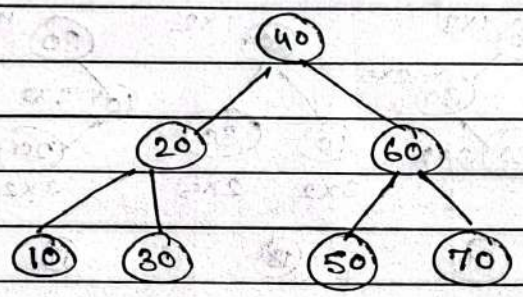
Time complexity = $O(n^3)$

Optimal Binary Search Tree

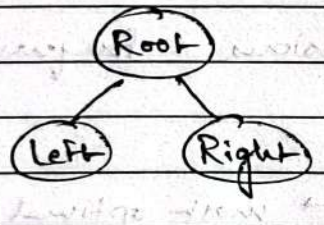
Recall search algo - linear search, binary search.
Can we utilise binary trees for searching ?

Keys = { 10, 20, 30, 40, 50, 60, 70 }

Construct a binary tree



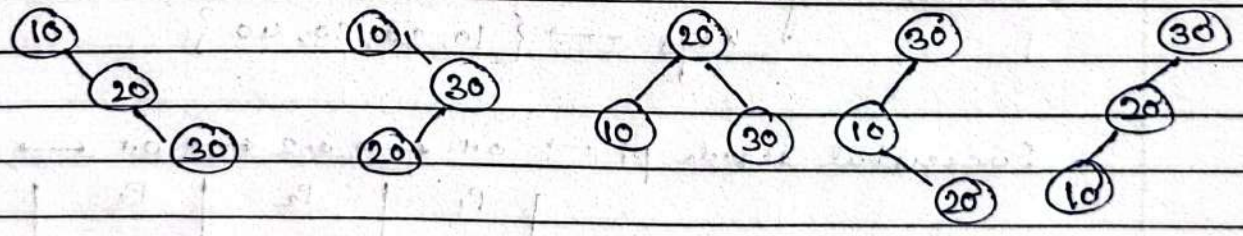
$O(\log n)$



Left < Root
Right > Root

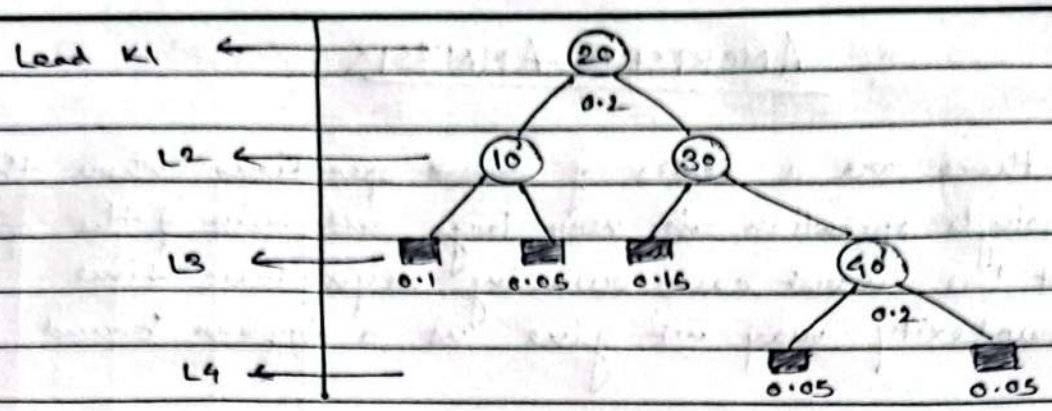
For a given set of elements, how many binary trees possible?

Keys = {10, 20, 30}



Total no. of possibilities $= \frac{{}^{2n}C_n}{n+1} = \frac{{}^{6}C_3}{4} = \frac{6!/3! \times 3!}{4} = 5$

Find the no. of comparisions in each binary tree.
Balanced tree - has less no. of comparisions



Total no. of unsuccessful searches possible = $n+1$ = 5

Total cost = [Successful Search] + [Unsuccessful Search]

= No. of search × prob. + No. of search × prob

$= \left[1\times0.2 + 2\times0.1 + 2\times0.1 + 3\times0.2\right] + \left[2\times0.1 + 2\times0.05 + 2\times0.15 + 3\times0.05 + 3\times0.05\right]$

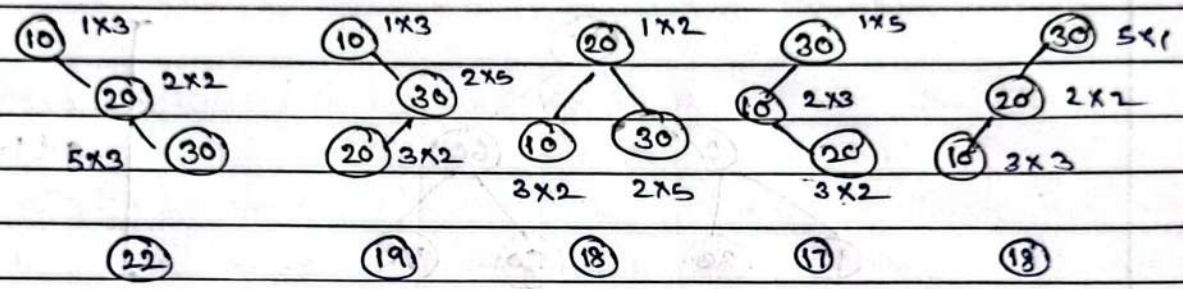
$= 2.1$

Optimal BST

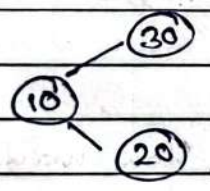
Suppose keys are searched at different frequencies

Keys = { 10 , 20 , 30 }
frequencies 3 2 5

(30) is searched 5 times



No. of searches = No. of comparisions × frequency



17 // ⟶ most optimal tree structure

- Consider negative searches also

Keys ⟶ { 10, 20, 30, 40 }

Successful search p_i ⟶ 0.1 ⟷ 0.2 ⟷ 0.1 ⟷ 0.2 ⟶
P_1 P_2 P_3 P_4

Unsuccessful search q_i 0.1 0.05 0.15 0.05 0.05
q_0 q_1 q_2 q_3 q_4

$$\text{No. of trees} = \frac{{}^{2n}C_n}{n+1} = \frac{{}^{8}C_4}{5} = 14 \text{ trees}$$

AMORTSED ANALYSIS

If there are a series of ~~event~~ operations where the cost of a single operation is very large and rest of the operations cost less, worst case running asymptotic time complexity may not give us a tighter bound.

In amortised analysis, your ~~average~~ average or worst case measure is not input dependent.

Amortised analysis is used to analyse time complexity of hash table, splay tree, disjoint sets etc.

Hence we avg. the time ~~complexity~~ required to perform a sequence of data structures operations all over the operations performed

* Three common techniques:
  (i) Aggregate analysis / aggregate method
  (ii) Accounting method
  (iii) Potential method.

Aggregate Analysis / Aggregate Method

In aggregate analysis, we show that for all n, a sequence of n operations takes worst case time $T(n)$ in total.

In the worst case, the average cost or amortized cost, per operation is therefore $T(n)/n$.

This amortized cost applies to each operation, even when there are several types of operations in the sequence.

(Ex1) Stack Operations

In our first example of aggregate analysis, we analyse stacks that have been augmented with a new operation.

Page No:
Date:

(i) Push (s, x) : Pushes object x onto stack s
(ii) Pop (s): Pop the top of the stack s and returns the popped object.
(iii) Multipop (s, k): Remove the k top objects of stack S

Pseudocode for multipop operation

```
while not STACK_EMPTY(S) and K > 0
        POP(S)
        K = K - 1
```

1 push operation → $O(1)$
n push operation → $O(n)$

1 pop operation → $O(1)$
n pop operation → $O(n)$

1 multipop operation → $O(n)$
n multipop operation → $O(n^2)$
∴ Worst case $T_c = O(n^2)$

Asymptotic Analysis

lets analyse a sequence of n push, pop and multipop operations on an initially empty stack.

The worst case cost of a multipop operation in the sequence is $O(n)$, since the stack size is atmost n.

The worst case time of any stack operation is therefore $O(n)$ and hence a sequence of n operations costs $O(n^2)$ since we may have $O(n)$ multipop operations costing $O(n)$ each.

Although this analysis is correct, the $O(n^2)$ result which we obtained by considering the worst case cost of each operation individually, is not ~~tight~~ right

Pseudocode

```
INCREMENT (A)
i = 0
while i < A.length and A[i] == 1
        A[i] = 0
        i = i+1
if i < A.length
        A[i] = 1
```

CV = counter value

4 bit binary

CV	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1

CV	A[3]	A[2]	A[1]	A[0]	cost	total cost
0	0	0	0	0	0	0
1	0	0	0	1	1	0+1=1
2	0	0	1	0	2	1+2=3
3	0	0	1	1	1	3+1=4
4	0	1	0	0	3	4+3=7
5	0	1	0	1	1	7+1=8
6	0	1	1	0	2	8+2=10
7	0	1	1	1	1	10+1=11
8	1	0	0	0	4	11+4=15

4 bit binary counter as its value goes from 0 to 8 by a sequence of 8 increment operation.

Total cost is always less than twice the total no. of increment operations

$\therefore$ $15 \times 2 \times 8$

15×16

Asymptotic Analysis

We observe some operations only flip one bit and some operations flip more than one bit. A single execution of Increment takes time $O(k)$ in the worst case.

Thus a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case. But this is not the right bound.

Aggregate analysis as amortised analysis

We can observe that not all the bits flip each time INCREMENT.

A[0] flip each time INCREMENT

A[1] flip only every other time

i.e. A[1] to flip $n/2$ time.

Similarly,

A[2] flips every fourth time or $n/4$ time in a sequence of n INCREMENT operations.

In general, for $i = 0, 1, \ldots k-1$

A[1] flips $n/2$ times

A[2] flips $n/2^2$ times

So bit A[i] flips $n/2^i$ times

For $i \geq k$, bit A[i] does not exist and so its cannot flip.

The total no. of flip in the sequence is thus

$$\sum_{i=0}^{k-1} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i} \qquad \left[S_\infty = a/1-r\right]$$

$$< \frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \ldots + \infty$$

$$< \frac{n}{1 - 1/n}$$

$$< 2n$$

The worst case complexity for a sequence if n increment operations on an initially zero counter is therefore $O(n)$

The average cost of each operation or amortized cost per ~~sec~~ operation is $O(n)/n = O(1)$

Pseudocode

```
TABLE - INSERT (T, x)
  If T.size == 0
      allocate T.table with 1 slot
      T.size == 1
  If T.num == T.size
      allocate T.table with 2.T.size slots
      Insert all items in T.table into new table
      free T.table
      T.table == new-Table
      T.size == 2.T.size
  Insert x into T.table
  T.num = T.num + 1
```

In this pseudocode, we assume that T is an object representing the table

T.table contains a pointer to the block of storage representing the table

T.num contains the no. of items in the table.
T.size gives total no. of slot in the table

Initially

→ [1] Insert 2. But size is full. Make a copy of this array of size x 2

→ [1 | 2] Insert 3. (Make a copy of this array x2)

→ [1 | 2 | 3 | 4] Insert 4 (Space is there, no need to create array)

→ [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]

→ [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15]

Page No: .....
Date: .....

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
size	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16
cost	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1

Worst case TC for single insert $\longrightarrow O(n)$

(As we need to copy all previous element and then insert single element)

Worst case TC for n insert $\longrightarrow O(n \times n)$

$= O(n^2)$

Amortized Analysis

We observe that i^{th} operation causes an expansion only when $i-1$ is an exact ~~pow~~ power of 2.

The amortised cost of an operation is in fact $O(1)$ as we can show using aggregate analysis

The cost of the ith operation:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

The total cost of n table-insert operations is therefore

$$\begin{aligned}
\sum_{i=1}^{n} c_i &\le 1+2+3+1+5+1+1+1+1 \ldots n \text{ terms} \\
&\le (1+1+\ldots+n \text{ times}) + (1+2+4+\ldots+\log n) \\
&\le n + (2^0+2^1+2^2+\ldots 2^{\log n - 1}) \\
&\le n + \frac{2^0(2^{\log(n-1)+1}-1)}{2-1} \\
&\le n + 2^{\log_2(n-1)+\log_2 2} - 1 \\
&\le n + 2^{\log_2 2(n-1)} - 1 \\
&\le n + 2(n-1) - 1 \\
&\le n + 2n - 2 - 1 \\
&\le 3n - 3 \\
&\quad O(n)
\end{aligned}$$

Cost of n insertion $= \sum_{i=1}^{n} c_i = O(n)$

Cost of single insertion $= \frac{O(n)}{n} = O(1)$

The average cost of each insertion or amortized cost per insertion is $O(1)$.

Accounting Method

In accounting method of amortized analysis, we assign differing charges to different operations with some operations charged more or less than they actually cost. We call the amount we charge an operation its amortized cost.

When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit.

Credit can help pay for later operations whose amortized cost is less than their actual cost.

If we denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\hat{c_i}$, we require

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i \text{ ; for all sequences of n operations}$$

The total credit stored is the difference b/w the total amortized cost and total actual cost

$$\sum_{i=1}^{n} \hat{c_i} - \sum_{i=1}^{n} c_i$$

(Ex1) Stack Operations

We know actual cost of stack operations were

PUSH	1
POP	1
MULTIPOP	min(s, K)

Lets assign the amortized cost (according to our choice

~~Pop~~	PUSH	2
	POP	0
	MULTIPOP	0

Lets analyse a sequence of push and pop operation on an initially empty stack.

Operations	Amortized Cost	Actual Cost	Credit
Push (A, S)	2	1	1
Push (B, S)	2	1	1+(2-1) = 1+1 = 2
Pop (S)	0	1	2+(0-1) = 2-1 = 1
Push (C, S)	2	1	1+(2-1) = 2
Pop (S)	0	1	2+(0-1) = 1
pop (S)	0	1	1+(0-1) = 0
push (D, S)	2	1	0+(2-1) = 1
pop (S)	0	1	1+(0-1) = 0

Analysis

Let analyse a sequence for n push, pop and multipop operation on an empty stack.

Since stack is empty, so pushes must be done first and this builds a credit.

And all pops are charged against this credit.

Note: there can be never be more pops (of either type) than pushes

$\therefore$ Total amortized cost $= 2+2+0+2+0+0+2+0$

$= 8$ (for 4 push operations)

$= 2 \times 4$

for n operation $= 2n$

$\therefore$ Amortized cost is $O(n)$

Ex 2 Incrementing a binary module

2 for setting a bit to 1

0 for setting a bit to 0

Counter value	A[3]	A[2]	A[1]	A[0]	Amortized cost	Actual cost	Credit
0	0	0	0	0	0	0	0
1	0	0	0	1	2	1	1
2	0	0	1	0	2	2	$1+(2-2)=1$
3	0	0	1	1	2	1	$1+(2-1)=2$
4	0	1	0	0	2	3	$2+(2-3)=1$
5	0	1	0	1	2	1	$1+(2-1)=2$
6	0	1	1	0	2	2	$2+(2-2)=2$
7	0	1	1	1	2	1	$2+(2-1)=3$
8	1	0	0	0	2	4	$3+(2-4)=1$

Analysis for increment process

We know,

worst case $T_c = O(nk) \simeq O(n^2)$

(Asymptotic Analysis)

$\therefore$ total amortized cost $= \sum_{i=1}^{n} \hat{c}_i$ (for n increment)

$= 0+2+2+2+2+2+2+2+2$

$= 2 \times 8$ (for 8 increments)

$= 2 \times n$ (for n increments)

$= O(n)$

Potential Method : Instead of representing prepaid work as credit stored with specific objects in the data structure, the potential method of amortized analysis represents the prepaid work as "Potential Energy" or just "Potential", which can be released to pay for future operations.

We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, \ldots n$ we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}.

A potential function ϕ maps each data structure D_i to a real number $\phi(D_i)$ which is a potential associated with data structure D_i.

The amortized cost c_i of the ith operation with respect to potential function ϕ is defined by

$$\boxed{\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})}$$

Total amortized cost, 1/11/24

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

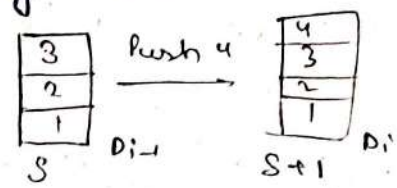
e.g.: <u>Stack operation</u>

- consider some elements are already there is the stack

<u>Push</u> Potential change $= \phi(D_i) - \phi(D_{i-1})$

$= s+1-s$

$= 1$



∴ Amortized cost,

$\Rightarrow \hat{c}_i = c_i + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{1}$ [cost of 1 push operation = 1]

$= 1 + 1$

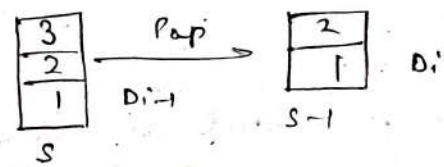
$= 2$

$\simeq O(1)$

<u>Pop</u> Potential change, $\phi(D_i) - \phi(D_{i-1})$

$= s-1-s$

$= -1$



Amortized cost

$\Rightarrow \hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$ [cost of 1 pop operation = 1]

$= 1 + (-1)$

$= 1 - 1 = 0$

∴ $\simeq O(1)$

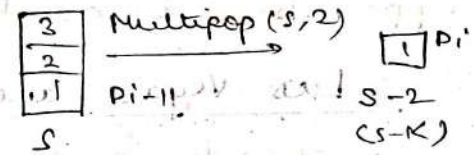
<u>Multipop</u>

Potential change $\phi(D_i) - \phi(D_{i-1})$

$= s - K - s = -K$

$= \cancel{s - K - (s-2)}$

$= \cancel{s - 2 - s - K}$



Amortized cost,

$\Rightarrow \hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$ [cost of K ~~po~~ multipop = K]

$= K + (-K)$

$= K - K = 0 \simeq O(1)$

. The amortized cost of each of the three operations is $O(1)$. And thus the total amortized cost of a sequence of n operations is $O(n)$.

As, $\phi(D_i) \geq \phi(D_0)$

it implies, total amortized cost of n operations is an upper bound on the total actual cost.

eg: Incrementing a binary number

We again look at incrementing a binary counter. Cost 2 if set bit to 1, cost 0 to set bit 0.

Counter value	A[3]	A[2]	A[1]	A[0]	Actual cost	$\phi(D_i)$	$\phi(D_{i-1})$	Potential difference $\phi(D_i) - \phi(D_{i-1})$	Amortized cost Actual cost + Potential diff
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1	2
2	0	0	1	0	2	1	1	0	2
3	0	0	1	1	1	2	1	1	2
4	0	1	0	0	3	1	2	-1	2
5	0	1	0	1	1	2	1	1	2
6	0	1	1	0	2	2	2	0	2
7	0	1	1	1	1	3	2	1	2
8	1	0	0	0	4	1	3	-2	2

check, how many bits are 1.

Total amortized cost of 8 operations is

$= (0+2+2+2+2+2+2+2+2)$

$= 2 \times 8$

$= 2 \times n$ (for n no. of increment operations)

$\simeq O(n)$

As $\phi(D_i) \geq \phi(D_0)$

it implies, total amortized cost of n operations is an upper bound on the total actual cost.

Las Vegas and Monte Carlo algorithm

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random values as an input to guide its behaviour, in the hope of achieving good performance in the average case over all possible choices of random values.

An algorithm that uses random number to decide what to do next anywhere in its logic is called ~~rando~~ randomized algorithm eg. randomized quick sort, we use random number to pick the next pivot.

Types: → Las Vegas

→ Monte Carlo.

Las Vegas: Algorithm that use the random input so that they always terminate with the correct answer, but where the expected running time is finite.

Monte Carlo: Algorithm which have a chance a producing an incorrect result, but the runtime is fixed.

eg: Finding an element in an array

```
Las Vegas SearchA(A, n)
{ while (true) {
      randomly select an element out of n elements
      if (a is found) {
          return true;
      }
}
```

The no. of iterations varies and can be arbitrarily large

```
Monte Carlo search_A(p, n, x) {
  i=0, tag = false
  while (i ≤ x) {
      randomly select an element out of n elements
      i+=1;
      if (a is found) {
          tag = true;
      }
  return tag;
}
```

In the above example, the L.V. algorithm always outputs the correct answer, but its running time is a random variable. The Monte Carlo algorithm is guaranteed to complete in an amount of time that can be bounded by a function the input size and its parameters x, but allows a small probability of error.

Karger's Min cut Algorithm

It is used to select the edge with mini-weight & according this min. weight edge next move is done in a network graph.

For its better working followed points must be taken into consideration.

A cut of connected graph is obtained by dividing vertex set V of graph G into 2 sets V_1 & V_2

i) There are no common vertices in V_1 & V_2 that is two sets are disjoint.

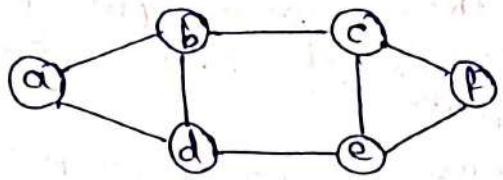
ii) $V_1 \cup V_2 = V$

Steps

1) Repeat step 2 to 4 until only two vertices are left
2) Pick an edge e(u, v) at random
3) Merge u & v
4) Remove self loops from E
5) Return |E|

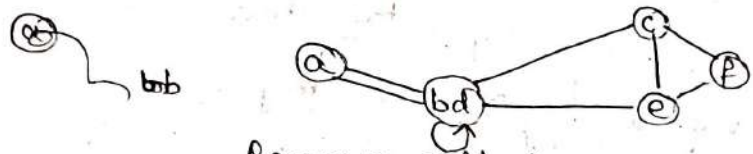
Q) Next Page

Q)

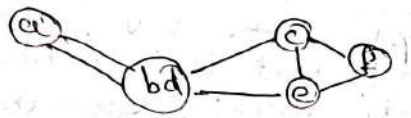


Sol The graph is $G(V,E)$ where $V = \{a, b, c, d, e, f\}$
$E = \{(a,b), (a,d), (b,c), (b,d), (c,e), (e,f), (d,e), (e,f)\}$

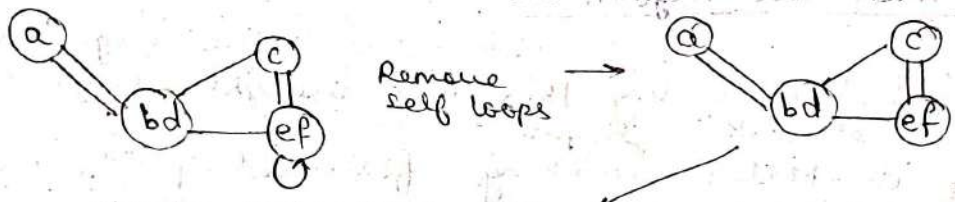
Step 1 check $|V| = 6 > 2$, yes
Step 2 Pick an edge at random say (b,d)
Step 3 Merge b & d. Now we have a new vertex (b,d)
all edge incident on b are incident on this vertex
$V = \{a, bd, c, d, e\}$



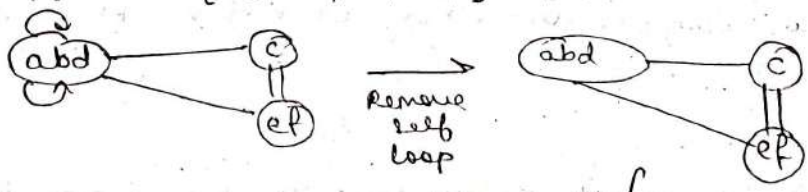
Remove self loops



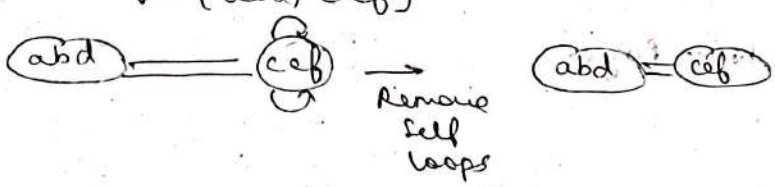
$|V| = 5 > 2$ yes
Again pick an edge at random say (e,f)
$V = \{a, bd, c, ef\}$



Check $|V| = 4 > 2$ yes
Pick an edge (a, bd)
$\therefore V = \{abd, c, ef\}$



Check $|V| = 3 > 2$ yes
pick an edge (c, ef)
~~V = {abbd a~~
$V = \{abd, cef\}$



$|V| = 2 > 2$ NO.
$\therefore$ Return $|E|$

Randomized QuickSort

$(\;) \; (1) \; (2 \;\; 3 \;\; 4 \;\; 5)$

p 0 1 2 3 4 q

x

Any element can come for 1/n.

Let $r = 3$

1	2	3	4	5
3	2	1	4	5

$2 \le 3$ ✓
$1 \le 3$ ✓
$4 \le 3$ ✗
$5 \le 3$ ✗

Now apply partition here

```
RQS(A, p, q) {
  if (p < q) {
    r = RGF(p, q)
    swap(A[p], A[r])
    m = Partition(A, p, q)
    RQS(A, p, m-1)
    RQS(A, m+1, q)
  }
}
```

★ When array is sorted in ascending or descending order, $T(n) = O(n \log n)$ [Proof Not Required]

- When all elements of an array are same, then both Quick Sort and Randomised Quick Sort takes TC $= O(n^2)$

Module 5 (Research topics)

1/11/14

NP-Hard and NP-Complete

Polynomial Time

Linear Search - n
Binary Search - $\log n$
Insertion Sort - n^2
Mergesort - $n \log n$
Matrix Multiplication - n^3

Exponential Time

0/1 Knapsack - 2^n
Travelling SP - 2^n
Sum of Subset - 2^n
Graph Coloring - 2^n
Hamiltonian Cycle - 2^n

Time consuming

- We basically want faster time execution for these for processes.

There are frameworks (guidelines) for doing research on these topics. These frameworks are called as np-hard and np-complete problems.

- We will show association between these problems
- write non-deterministic algorithms.

Algorithms which we write are deterministic i.e. we know how they are knowing.
Algorithms which we write and we don't know how they are working, Basically used to preserve our research work are called as non-deterministic algorithms. Each non-deterministic takes $O(1)$ time.

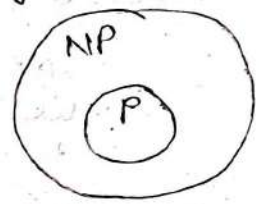
eg: ____ lines are non-deterministic

```
Algorithm NSearch(A, n, Key){
    j = choice();          ---> O(1)
    if (Key == A[j]) {
        write(j);          ---> O(1)
        success();         ---> O(1)
    }
    write(0);              ---> O(1)
    failure();             ---> O(1)
}
```

P: Deterministic algorithms taking polynomial time.
eg: linear search, binary search, etc

NP: Non-Deterministic algorithms ~~po~~ taking polynomial time.

- all deterministic algorithms were once non-deterministic.



Tractable Problems	Intractable Problems
• Solved in polynomial time complexity.	• Solved in exponential time complexity.
• Shorter runtime	• Longer runtime
• Feasible for larger inputs	• Not suitable for larger inputs.
• Solved using efficient algorithms.	• Uses approximations
• Belongs to class P.	• Belongs to NP or even worst classes
eg: Quick Sort, Merge Sort, etc	• eg: TSP, 0/1 Knapsack.

Decision Making	Optimization Problem
• Determine whether a specific condition is met or not	• More optimal solution of an algorithm is found.
• Simple answer is in Yes/No.	• Maximizes or minimizes our requirement.
• Consumes less time	• Time consuming
• Less complexity	• More complexity.
• eg: checks whether a path exists between two nodes in a graph.	• eg: Finding shortest path in a graph.

—— xxx ——

~~Satisfact~~

3-CNF-Satisfiability : To find out for what value of x_i this formula works.

$x_i = \{x_1, x_2, x_3\}$

eg Formula $CNF = \underbrace{(x_1 \vee \bar{x}_2 \vee x_3)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}_{C_2}$

$C_1, C_2 \rightarrow$ clauses

In 3-CNF, each clause has 3 literals.

Possible values of

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$8 \Downarrow 2^3 \Downarrow 2^n \rightarrow$ Exponential

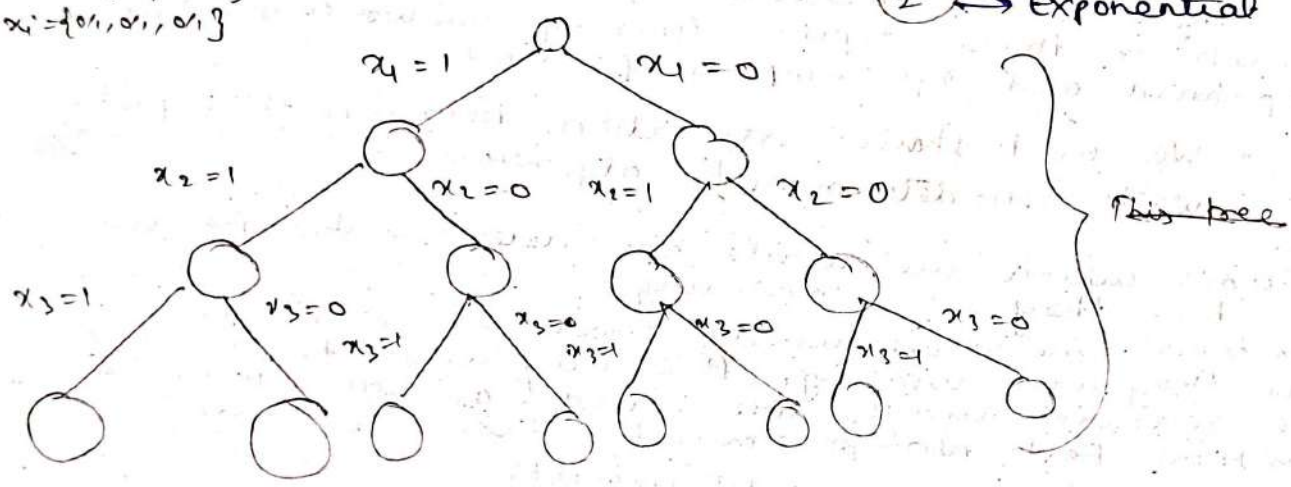
For solution, we need to create a state space tree.

Lets Relate with 0/1 Knapsack problem

$P = \{10, 8, 12\}$ $n = 3$

$w = \{5, 4, 3\}$ $M = 8$

$x_i = \{0/1, 0/1, 0/1\}$



Satisfiability $= O(2^n)$

3-CNF SAT, we are given a boolean formula ϕ in 3-CNF and we have to figure out if it satisfiable

$\phi = (x_1 \vee \bar{x}_1 \vee \bar{x}_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4)$

Let $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$

$\phi = (1 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 0) \wedge (0 \vee 1 \vee 0)$

$= 1 \wedge 1 \wedge 1 = 1$

$\therefore \phi = 1$ (NP-Complete).

- Let take satisfiable problem as a base problem from which we will relate other problems.

$\therefore$ Satisfiable is NP-hard

Reduction : SAT (I_1) $\propto$ 0/1 Knapsack (I_2)

- SAT reduces to 0/1 Knapsack
- We take one instance of SAT and convert it to an instance of 0/1 Knapsack.
- If 0/1 Knapsack can then be solved in polynomial time, then that same algorithm can be used to find the ~~0/1 Knapsack~~ SAT in ~~a~~ polynomial time
- For conversion of I_1 to I_2, polynomial time

- IF SAT is reducing to '0/1' Knapsack problem, Then 0/1 Knapsack is also be NP-hard. 1/11/24

∴ SAT is NP Hard
0/1 Knapsack is also NP Hard.

* $SAT \propto L_1$ $L_1 \propto L_2$
All SAT, L_1, L_2 are NP-Hard.

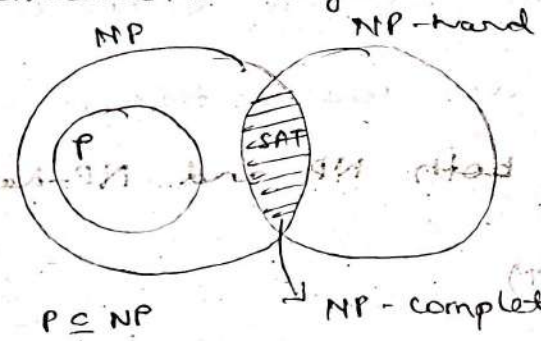
- whenever SAT reduces to any problem, both are called as NP-hard.

(SAT) $\propto$ L

- SAT is NP-hard.
- If any problem has non-deterministic polynomial time algorithm, then it is np-complete. SAT has that, and thus, it is NP-complete.

- when an algorithm is reduced from another algorithm, it is NP hard.
- when we write a non-deterministic polynomial time algorithm for it, it is then called as NP complete

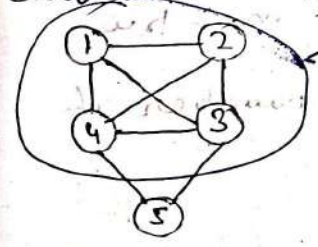
* A problem which is current non-deterministic can become deterministic in the future.
But it does not means currently P = NP.
If currently P = NP, then it means we need to discover the deterministic algorithm from the non-deterministic algorithm



NP complete = NP + NP-hard

Clique Decision Problem (CDP)

- Clique - sub graph of a graph which is complete



clique of size 4.

Complexity classes for following selected problems

1) Vertex Cover

Vertex cover is in NP :

- Given a graph $G = (V, E)$ and a number K, a certificate for this problem is a subset of vertices $C \subseteq V$ with $|C| \leq K$.
- We can verify in polynomial time that C is a vertex cover by checking each edge in E to see if atleast one endpoint is in C.
- This takes $O(|E|)$, so vertex cover is in NP.

Vertex cover is NP-hard .

- We have to reduce vertex cover from 3-SAT.
- Given a 3-SAT formula with clauses $C_1, C_2, \ldots, C_m$ and variables $x_1, x_2, \ldots, x_n$, construct a graph such that there is a vertex cover of size K if and only if the formula is ~~satif~~ satisfiable
- Construct a graph where:
  - → Each variable x_i has two vertices x_i and $-x_i$ connected by an edge, representing the two possible truth assignments.
  - → For each clause C_j, create a triangle connecting the literals in that clause.
  - → Add edges from each literal to its negation to enforce that both cannot be selected in the cover
  - → Set $K = n + m$
  - → If there exists a vertex cover, then formula is satisfiable and vice-versa.

∴ Vertex cover is both NP and NP-hard, thus it is NP-complete.

2) 3-CNF Satisfiability (3-SAT)

3-SAT is in NP :

- Given a truth assignment for the variables, we can verify in polynomial time whether each clause in the 3-CNF formula has atleast one true literal.
- This takes $O(m)$ time, where m is the number of clauses, so 3-SAT is in NP.

3-SAT is NP-hard :

- 3-SAT is actually one of the original NP-complete problems proved by Stephen Cook in his theorem.
- It remains NP-complete because we can convert any instance of SAT into an instance of 3-SAT in polynomial time by restructuring clauses to contain

3) Maximal clique Problem

Maximal clique is in NP:

- Given a graph $G = (V, E)$ and an integer K, a certificate would be a subset of vertices that forms a clique of size K.
- we can verify in polynomial time by checking that all pairs of vertices in the subset are connected by an edge, which takes $O(k^2)$ time, so Maximal clique is in NP.

Maximal clique is NP-hard:

- we can prove this by reducing from 3-SAT.
- Given a 3-SAT formula, construct a ~~formula~~ graph where each clause is represented by a triangle of vertices (one for each literal in the clause). connect literals that cannot be true simultaneously.
- A clique of size K = (no. of clauses) corresponds to a selection of one literal per clause, forming a satisfying assignment.

∴ Maximal clique is NP & NP-hard and thus it is NP-complete.

4) Hamiltonian Cycle Problem

Hamiltonian cycle is in NP:

- Given a sequence of vertices that form a cycle, we can verify in polynomial time that each vertex appears exactly once in the cycle, and all edges in the ~~cycle~~ cycle exist in the graph.
- This takes $O(|V|)$ time, so Hamiltonian cycle is in NP.

Hamiltonian Cycle is in NP hard:

- Given an instance of a known NP-complete problem, construct a graph where finding a Hamiltonian cycle corresponds to a solution for the original problem, which implies that Hamiltonian cycle is atleast as hard as any NP-complete problem.

∴ Hamiltonian cycle is both NP and NP-hard, it is NP-complete.

5) Travelling Salesman Problem (But this is NP-hard)

TSP is NP

- The decision version of TSP asks whether there exists a tour of length $\leq K$.
- Given a proposed tour, we can verify that it visits each city exactly once, and that its length is within K in polynomial time.
- This takes $O(n)$ time, so the decision version of TSP is in NP.

TSP is NP-hard

- TSP is ~~hard~~ NP-hard because it generalizes the Hamiltonian cycle problem
- Given a Hamiltonian cycle instance, we can construct a

graph would solve the TSP. 24/1/24

∴ TSP is NP and NP-hard. (TSP is only NP-hard in optimized version)

∴ TSP is NP-complete (~~only~~ only the optimization version) in its decision form.

Approximation Algorithms

An approximation algorithm is way of dealing with NP-completeness for optimization problem. The goal of approximation algo is come close as ~~op~~ possible to optimal solution in polynomial time.

$c \rightarrow$ Cost of solution

$c^* \rightarrow$ Cost of optimal solution

$P(n) \rightarrow$ Approximation ratio

$n \rightarrow$ input size

1) Maximisation
2) Minimisation

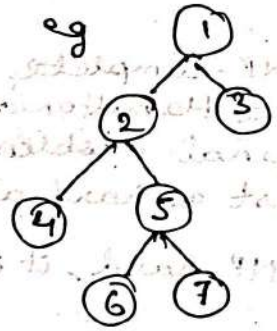
- Maximisation : $\frac{c^*}{c} \leq P(n)$

$\boxed{P(n) \neq 1}$

- Minimisation : $\frac{c}{c^*} \leq P(n)$

Vertex Cover Problem : A vertex cover of a graph is a subset of vertices which ~~co~~ cover every edges. Vertex cover problem is the minimum size vertex ~~co~~ cover.

eg



Subset {1, 2, ~~8~~ 5, 3} cover all edges

Subset {5, 6, 2, 1} also cover all edges.

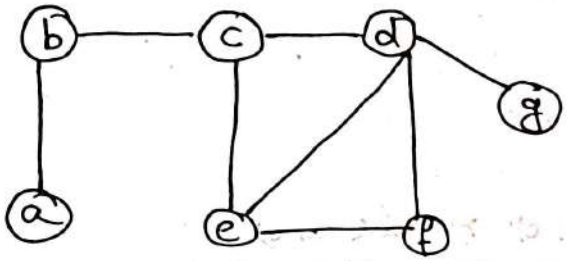
Like this many subsets exists.

- we have to find minimum no. of vertices, which cover all the edges.

{2, 3, 5} → covers all the edges.

How vertex cover problem is solved by approximation problem?

```
C ← ∅
E' ← E[G]
while E' ≠ ∅
    do let (u, v) an arbitrary edge in E'
        C ← C ∪ {u, v}
        Remove from E' every edge incident either u or v.
Return C.
```



Initially,

$C = \phi$

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

↳ contains every edge of graph G

while $E' \neq \phi$, run the loop

Lets select (b,c).
$\downarrow \quad \downarrow$
$u \quad v$

$\Rightarrow C = \phi \cup \{u,v\}$
$= \{b,c\}$

Now in E', ~~{b,c} is not present~~ removing all incident edges of b,c like (a,b), (c,d) and (c,e).

Now $E' = \{(e,d), (e,f), (d,f), (d,g)\}$

Lets select (e,f)

$E' \neq \phi \longrightarrow \therefore C = \phi \{b,c\} \cup \{e,f\} = \cancel{\{(b,c),(e,f)\}}\ \{b,c,e,f\}$

$\therefore$ remove all incident edges of (e,f) from E'
i.e. $\{e,d\}$ and $\{d,f\}$

$\boxed{\therefore E' = \{(d,g)\}}$

$E' \neq \phi \quad \therefore$ select (d,g)

$\cancel{C = \{(b,c), (e,f)\} \cup \{(d,g)\}}$

$C = \{(b,c), (e,f), (d,g)\}$

$C = \{b,c,e,f\} \cup \{d,g\}$

$C = \{b,c,e,f,d,g\} = 6$

But this is not optimal solution

Optimal solution is $\{b,d,e\}$. But this is the main concept of approximation problem which reaches closes to optimal solution.

(C^*) Cost of optimal solution $= 3$

(C) cost of ~~approx soluti~~ approximation solution $= 6$

This is a minimisation problem

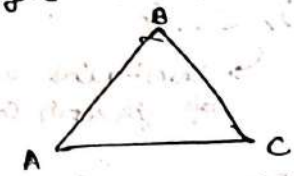
$$\therefore \frac{C}{C^*} \leq \rho(n)$$

$$\Rightarrow \frac{6}{3} \leq \rho(n) \quad \Rightarrow \boxed{\rho(n) \geq 2}$$

Travelling Salesperson ~~algorithm~~ using approximation algorithm

Special case

- Triangle Inequality Problem



- $w(a,b) + w(b,c) >= w(a,c)$
- $2+3 > = 5$ or less than 5

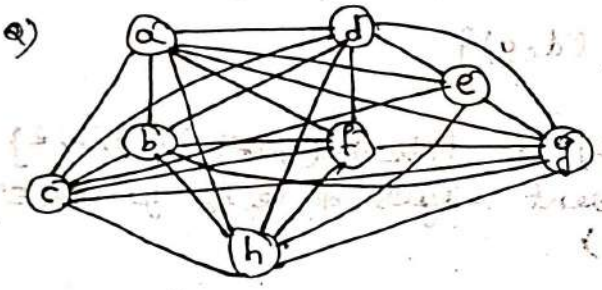
Step 1: Construct a MST

Step 2: Perform Pre order walk i.e. travel around each node following the edges and finally return back to the first vertex.

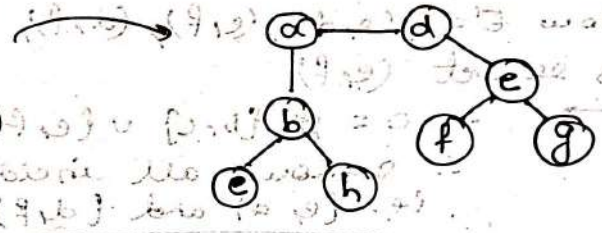
Step 3: Then edges are joined in this order to obtain approx tour walk

ex

Q)



MST



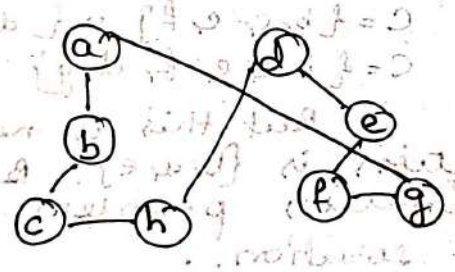
- Pre order Path

a b e b h b a d e f e g ~~e g~~ e d a

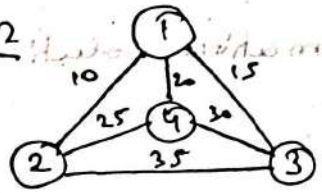
- Hamilton Path

a b c h d e f g a

∴

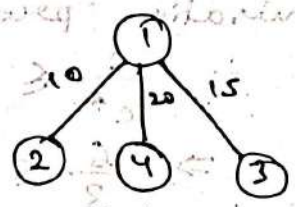


eg 2

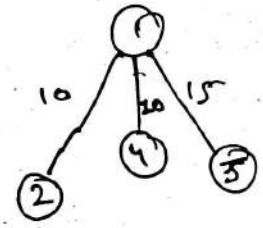


MST

Order : 10, ~~20~~ 15, 20, 25, 30, 35



∴ MST is



- Pre order walk: 1 2 1 4 1 5 1
- Hamilton path: 1 2 4 5 1