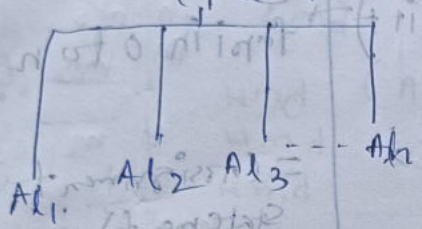
27/7/23

CW

Design & Analysis of Algorithms

Algorithm = Informal description of a set of ~~Stated~~ Statements, in orders to solve any computation program e/ or the soln of a computational Problem.

CP (Computational Problem)



for a CP we have many algos (Al1, Al2 .. Aln)

Characteristics of Algorithms

1) Input (there can be 0 or more input for a given algorithm)
   - I/P requirement
   - Pre algo

2) output (output requirement of an algo says that an algo must produce atleast one o/p during solving a particular computational Problem)

3) Definite (Each and every statements of an algo must have to be unambiguous)
   - ness

4) Finiteness (An algo must get completed (finished after a finite amount of time)

5) Effectiveness (All the statements in an algo should be non redundant & in such a way a statement is written that they

can be executed simply using pen & paper) / simple or basic such that all the statements can be executed simply using pen & paper).

Algorithm Add(A,n) ← you can pass the parameters required also ← C lang Syntax

Begin { S=0;

{ to denote start

for(i=0; i<n; i++)
{
S = S+A[i];
}
return S;
} / End

or

for i in 0 to n

= (assignment statement) is somewhere used by := (colon) or we can use ← (in Ruby lang)

i=0) you can write by this way
i=3 [for i=0 to n in step 3]

TC

Frequency Count method (time complexity count method) [this method is 1st simple]

① Algorithm Add(A,n)

Begin S=0; ——— 1
for(i=0; i<n; i++) ——— n+1 (n+1 times)
{ S = S+A[i]; ——— n
}
return S; ——— 1
End

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

[f(n) = Time Complexity function]

c=a+b ; c=a*5+d-3; → i.e, assignment statement ~~usely~~ basically takes 1 unit of time (assumption). Return statement will execute only 1 times. For loop (single) will execute for n times.

TC of this particular algo $\boxed{O(n) = 2n+3}$
(the highest order of polynomial)
i.e

Space Complexity $n=n$; $S \approx O(n)$

(remaining is reused)
A ——— n word
n ——— 1 word (As they all are scalar quantity)
i ——— 1 word
S ——— 1 word
n+3

$S(n) = n+3$ words

$\boxed{O(n) = n+3}$

Algorithm (subtraction (matrix) (A, B, n)) square matrix (n×n)

the thing inside outer for loop will execute n times so, that statement will also execute by n times. the things enter inner for loop will execute for n times (i.e. n+n) of the inner for loop have the value n+1 (i.n(n+1)) times

for (i=0; i<n; i++) ——— n+1
{
for (j=0; j<n; j++) ——— (n+1)*n
{
C[i][j] = A[i][j] - B[i][j]; ——— n*n
}
}
return C; ——— 1

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

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

A — n+n
B — n+n
C — n+n
n — 1
i — 1
j — 1

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

Space Complexity (SC) = $\boxed{O(n^2) = 3n^2+3}$ (SC)

(3) Algorithm mul (A, B, m, n) (let m, n)

n = res_col (col_)
m = res_row (row)

n = m (Assume that is a square matrix)

```
{
  for(i=0; i<n; i++)          —— n+1
  { for(j=0; j<n; j++)        —— n*(n+1)
    {
      c[i][j] = 0;            —— n*n
      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
      }
    }
  }
  return c;
}
```

TC,

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

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

$\boxed{O(n^3) = TC}$

SC,

A — n+n
B — n+n
C — n+n
i — 1
j — 1
k — 1
n — 1

$f(n) = 3n^2+4$ (space complexity func)

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

[we will consider the constant while making tc and SC]

4) for(i=0; i<n; i++) —— n+1
{ stmt 1; —— n
stmt 100; —— 100n
}

$f(n) = 101n+1$

$TC = \boxed{O(n) = \cancel{3n+101n+1} \; 101n+1}$

order of tc | TC

{ return it's 10 or 1000 Statement it doesn't matters it will have the orders n

5) for (i=n; i>0; i--) —— n+1
{ stmt; —— n
}

$f(n) = 2n+1$

$\boxed{TC = O(n) = 2n+1}$

order of tc

TC

the constant while making tc and sc]

if it's not a square matrix then,

```
for (i=0; i<n; i++)        → n+1
  {
    for(j=0; j<m; j++)     → n*(m+1)
      {
        C[i][j] = A[i][j] - B[i][j]   → n*m
      }
    return C;   → 1   // this can be neglected
  }
```

→ now, → cal, not n<m → cal

To maintain simplicity we avoid this case

$\therefore \boxed{TC = O(nm)}$

SC,

A → (~~m+n~~ n*m)
B → (n*m)
C → (n*m)
i → 1
j → 1
n → 1
m → 1

$\therefore \boxed{SC = O(nm)}$

m+n m*n
p+q p*q
m+1 m*1
O(mn+p+q+m)

6) for (i = 0 ; i < n ; i = i + 3) — $\frac{n}{3}+1$

$\frac{n}{3}$ ↓ we can ignore it doesn't matter

{ Stmt ; — $\frac{n}{3}$ }

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

$\boxed{TC = O(n) = \frac{2n}{3}}$ → order of TC

TC

0, 3, 6, 9, 12 — n/3
3 gap = n/3 [cal. a, r, n]
get increasing, n/3 times increasing

if i+3 will be 100, 1000 .. 10000 it doesn't affect the order of n.

7) for (i = 1 ; i <= n ; i = i * 2) ← particularly if loop increasing this way

{ S-tmt ; — [The loop will run till the condition doesn't reach i <= n, let the condition will reach i <= n at kth iteration, i.e at kth it, i = 2^k & if i > n then the loop will terminate, i.e loop will run till k (the val. of k) th it ∴ the val of k] [Actually 2^{k-1} but, −1 isn't consider here]

}

(with satisfying condition as it's final, the condition satisfies)

1 − 1
2 − 2 } double
3 − 2^2 } double
4 − 2^3
⋮
k − 2^k at kth iteration

$i \le n$
$2^k > n$ → at this time the loop will terminate
⇒ $2^k = n$
⇒ $k = \log_2 n$ → the no. of times the loop will executed ~~after that~~

i.e $\le \log_2 n$ [as only i is dependent we are changing only i]

∴ the loop will run till $\log_2 n$ ∴ the order of $n = \log_2 n$

$\boxed{TC = O(\log_2 n)}$ → this will be the order of TC

TC

i = 0 taking ① the approach
i = 1 taking ② the approach
as 2^0 = 1

TC – The valid algorithm takes a finite amount of time for execution. The time returned by the algorithm to solve given problem is called time complexity of the algo. This ~~yet~~ is very useful measure in algo analysis.

SC – Problem solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algo. to solve given problem is called space complexity of the algo.

Frequency count method – [It specifies the no. of times a statement is to be executed, so how many times the statement within our algo. is executed.] It is a method for calculating TC & SC of an given algo.

floor, celling value (add)

3/8/23

Que

- for (i=1; i<n; i=i*2) [if i = i*4]

then TC = $O\log_4 n$

for (i=1; i<n; i=i*2)

let, n = 16 → [this is power of 2]

i = 1
= 2
= 4
= 8
= 16 ×

4 times executed

$\log_2 16 = 4$

~~TC = [$\log_2 n$]~~ [is the value of exactly]

let, n = 20

i = 1
= 2
= 4
= 8
= 16
= 32 ×

5 times executed

$\log_2 20 = 4.2$

$TC = O(\lceil \log_2 n \rceil)$

[⌈ ⌉ = take the celling value, i.e the highest round off
eg 4.2 → 5
1.3 → 2 etc]

8) for (i = n; i >= 1; i = i/2)
{ stmt; }

$n, n/2, n/2^2 \ldots n/2^k$
(1st) (2nd) .. (kth)

the loop is going to be executed when,

i < 1

⇒ i = 1 (for simplification)

⇒ $n/2^k = 1$

→ $2^k = n$

⇒ $k = \log_2 n$

$TC = O(\lceil \log_2 n \rceil)$

[if i >> 1 then TC = $O(\log_2 n)$]

let, n = 16

i = 16
i = 8
i = 4
i = 2
i = 1
i = 1/2 ×

5

[Similarly if i = i/10 then $\log_{10} n$]

[$\log_2 16 = 4.0$
; we are taking celling val. here so, the loop will execute 5 times]

9) for (i = 1; i * i <= n; i++) → $\sqrt{n}$
{ stmt; } → $\sqrt{n}$

$i^2 = n$
$i = \sqrt{n}$ ← the no. of times the loop is going to be executed

$TC = O(\sqrt{n})$

10) for (i = 1; i <= n; i++) → n
{
for (j = 1; j <= n; j = j + i) → n
{ stmt; } → n * (n+1)
}
}

[$\to \sqrt{n}$; $\sqrt{n}$; $\sqrt{n}$; $O(\sqrt{n})$]

[intuition
1 loop iteration transition this thing]

i = 1 (i considered)
j = 1, 2, 3, 4 ... n times (the total execution time of loop)

i = 2 (i considered)
j = 1, 3, 5, 7 ... n/2 , [~~for odd n/2~~] As increasing by +2
i = 3
j = 1, 4, 7, 10 ... n/3 , [As increasing by +3]

i = 1; j = 1, 2, 3 ... n
i = 2; j = 1, 3, 5 ... n
i = 3; j = 1, 4 ... n
(n/3) · n

where even there (2) will be like that i+1 = n then at any O(√n) if, i*i = n O(√n) like that

$TC = \boxed{O(\log_2 n)}$ [if, $i > 1$ then $TC = \boxed{O(\log_2 n)}$]

For counting TC,

1) Check by trace out method

2) take an eg. for giving a conclusion that is the val is '=' or it is taking '[ ]' or '[ ]' val.

3) If you can't give a conclusion by taking 2's powers as eg, tha take not 2's powers (20, 30 ... etc) ..

4) If value is giving conclu. '=' then 2's powers are not 2's pow's then try not 2's pow or 2's pow's value respectively.

5) If you are getting conclusion about the value is '[ ]' or '[ ]', from the very 1st eg whether it is of 2's powers or not 2's power, then no need to try another 2's powers or not 2's power eg.

[ ] = take the ceiling value, i.e the highest round off eg 4.2 → 5, 1.3 → 2 etc]

[As the value of exactly]

$n/2^k$ (kth)

$i = 1$ (i considered)

$j = 1, 2, 3, 4 \ldots$ (j considered)

$i = 2$

n times (the total time of ... increasing ...)

will be like that $i+i = n$ then at worst $O(\sqrt{n})$ if, $i \times i = n$ $O(\sqrt{n})$

$n = 16$

it	i
at 1st →	$\frac{16}{2} = 8$
1 st →	9
2nd →	10
3rd →	11
4th →	12
5th →	13
~~16th~~	
6th →	14
7th →	15
8th →	16

8 times $\frac{16}{2}$ times, $\frac{5}{2}$ times

$n = \frac{16}{2}$

$8 <= 16$

$\frac{8}{2}$

$\frac{5}{2}$ times

of times the loop is going to be executed

$\rightarrow \sqrt{n} = 2\sqrt{n}$

$\sqrt{n}$ $O(\sqrt{n})$

$j = i, j = i+1, \ldots n$

$i = 2, j = 3, 4 \ldots n$

$i = 3, j = 4 \ldots n$

$n^2 + n$

Where ever there 2 will be like that $i+i = n$ then always $O(\sqrt{n})$ if $i \times i = n$ $O(\sqrt{n})$ like that

1 (i considered)

1, 2, 3, 4 ... considered)

$n/3$, [As increasing by +3]

$\boxed{\log_{10} n}$

[Similarly, if $i = i/10$

for (i = n; i >= 1; i = i/2)

$1, 2 \ldots n$

$n, n/2 \ldots 2, 1$

True

$i = n, n/2, n/2^2, n/2^3$

at kth iteration, (loop will run the kth it)

$i = 1$ [$i > 1$ → this condition is satisfied by before values of $n/2^k$ $(n, n/2, n/2^2 \ldots)$]

$n/2^k$ (at kth iteration)

$\Rightarrow n/2^k = 1$

$\Rightarrow n = 2^k$

$\Rightarrow k = \log_2 n$

$O(\log_2 n)$

$n/2$

$n/2^2$

$n/2^3$

$i = 2; j = 3, 4 \ldots n$

$i = 3, \ldots n$

$(n^2) + n$

Where every there (2) ... be

$\{$ stmt;

$\cancel{n*(n+1)}$

execution

instruction level parallelism

$TC = \boxed{O(\log_2 n)}$

Let, $n = 16$

[Similarly, if $i = i/10$ then $\boxed{\log_{10} n}$]

Time

$n = 16$

$i = 1$
$= 2$
$= 4$
$= 8$
$= 16$ ✗

4

if i is 2^n then exact value i.e. $\log_2 n$

$\log_2 16 = \log_2 2^4$
$= 4$ (exact 4)

Now if $n = 20$

$i = 1$
$= 2$
$= 4$
$= 8$
$= 16$
$= 32$ ✗

5

$\log_2 20$
$= 4.2$
(not exact 4 but the value is (no. of its) $= 5$
∴ we are taking ceiling value.
∴ $\boxed{O(\log_2 n)}$

$n/2^2$, $n/2^3$

$n/2^k$

(at the iteration)

$\log_2 4$

Wherever there (i) will be like that $i+i = n$ then always $O(\sqrt{n})$
if, $i \times i = n$
$O(\sqrt{n})$

let $n = 25$ ← test (tracing)

$i = 1$

$j = 1, 2, 3, 4 \ldots 25 \rightarrow n$

(current+1) (current+2)

$i = 2$

$j = 1, 3, 5, 7, 9, 11, 13 \ldots 25 \rightarrow n/2$

(current+1) (current+2)

$i = 3$

$j = 1, 4, 7, 10, 13, 16, 19, 22, 25 \rightarrow n/3$

when $i = n$

$j = 1$ time

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

for inner loop

for outer loop

$$= n^2\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}\right) \quad \left[\because \log n = 1 + \frac{1}{2} + \frac{1}{3} \ldots + \frac{1}{n}\right]$$

$$= n^2 \log n$$

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

[By taking eg. it is $O([2 \log 2n])$ more effective.]

↳ no need through (mine)

11) $for(i = n/2; i <= n; i++)$ $\longrightarrow n/2$

{

$for(j = 1; j <= n; j = 2 * j)$ — $\log_2 n$

{

$for(k = 1; k <= n; k = k * 2)$ — $\log_2 n$

{ $stmnt;$ } $\rightarrow n/2 * \log_2 n * \log_2 n$

$= \frac{n}{2} * (\log_2 n)^2$

}

}

$$\boxed{TC = O(n(\log_2 n)^2)}$$

As, stmnts of j for m/2

n/2

$\frac{1}{2}$ = const term

12) $for(i = 1; i <= n; i++)$ — n

{ $for(j = 1; j <= i^2; j++)$

{ $for(k = 1; k <= n/2; k++)$ — $n/2$

{ $stmt;$ }

}

}

$i = 1$

$j = 1$ times (as $j <= i^2$, $i^2 = 1$) [j considering i]

$k = n/2$ times (considering)

[As, i has effect of j so, while adding we are not considering i as it is already considered in j, but k is independent of i, j for all cases. So, we have to consider it while adding & have to do consider it]

$i = 2$

$j = 1, 2, 3, 4 = 2^2$ times (as $i^2 = 2^2$) $\rightarrow 5^x$

$k = \frac{n}{2}$ times

$i = 3$

$j = 1 \ldots 9 = 3^2$ (as $i^2 = 3^2$)

$k = \frac{n}{2}$ times

$i = n$

$j = n^2$ times

$k = \frac{n}{2}$ times

$$n + \left[(1^2 + n/2) + (2^2 + n/2) + (3^2 + n/2) \cdots (n^2 + n/2)\right]$$

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

$$= \frac{n^2}{3} + \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{1}{18}\left[n^3 (n+1)(2n+1)\right]$$

$$\boxed{TC = O(n^5)}$$

$$\left[(1^2 + n/2) + (2^2 + n/2) + (3^2 + n/2) + \cdots + (n^2 + n/2)\right]$$

$$= \frac{n}{2}\left[1^2 + 2^2 + 3^2 + n^2\right]$$

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

$$= \frac{1}{12}\left[n^2 (n+1)(2n+1)\right]$$

$$\boxed{TC = O(n^4)}$$

Cw

4/8/23

(13)

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

If the for loops are basic then can apply 1st methods (basic)

But if they are linked with other loop's variable then use tracking method

i	j		#stmt (no. of times the statement is going to be executed)
0	0		0 times
1	0	✓	1 time
1	1	×	
2	0	✓	2 time
2	1	✓	
2	2	×	
⋮			
n-1			n-1 times

1 will not effect TC so, n-1 ≈ n can consider

$\therefore 0 + 1 + 2 + \cdots + (n-1)$

$= \frac{n(n+1)}{2}$ [∵ the sum of the 1st n natural no. $[1+2+\cdots+n]$ as there is 0, so $0+1+\cdots+n-1$]

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

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

JK

(14) m=0;

for (i=1; m<=n; i++)

{ m=m+i;

}

[for the case in which m is dependent on the i, we have to trace out i, m linked with each other, so trace out]

[m<=n will not affect the tracing]

traceout

i	m
1	(initial) 0 + 1 = 1
2	1 + 2 = 3
3	1+2 + 3 = 6
4	1+2+3 + 4 = 10
⋮	⋮
K	1+2+3+ --- +K = $\frac{K(K+1)}{2}$

[n, i<=n x, m<=n so, consider i-stmt K, not n]

at K the loop will exit as at K m>n. i.e upto Kth iteration the loop is going to be run.

[m<n, con. will be satisfied by i=1, 2, --- K-1 m=n, con. will be satisfied by, i=K th iteration]

$m = n$

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

$\Rightarrow K^2 + K = 2n$ [2 & K are negligible]

$\Rightarrow K \cong \sqrt{n}$

$\therefore$ $\boxed{TC = O(\sqrt{n})}$

(15) main()

{ — — —

3 $1+2+3$

$1+2+3+\cdots+p = \frac{p(p+1)}{2}$

at the pth iteration the value of s is exactly equal to the n

$[s = n \Rightarrow \frac{p(p+1)}{2} = n]$

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

$\Rightarrow p \cong \sqrt{n}$

$\boxed{TC = O(\sqrt{n})}$

(18)
```
func()
{ i=0;
  if (a>20)
  {
    stmt;
  }
  else
  {
    while (i<=n)
    {
      stmt;
      i++;
    }
  }
}
```

min — 1 — $O(1)$ — if the condition will satisfied then stmt will be executed for 1

max — $O(n)$

Under func, 2 seperate loop, considers seperate tc for 2 seperate loop & while what is max & min

If loop will execute for only when the con. is satisfied if the condition is not a logic of loop then tc = 0

Hence in this func the

$\boxed{min\ TC = O(1)}$ & $\boxed{max\ TC = O(n)}$

(19)
```
k=1;
for (i=1; i<=n/2; i++)
{
  for (j=1; j<=i; j++)
  {
    while (k<=500)
    {
      stmt;
      k++;
    }
  }
}
```

$i = 1$

$j = 1$ time

$k = 500+1$ times

(i & j both are considered)

$i = 2$

$j = 2$ times

$k = 500+2$ times

$i = 3$

$j = 3$ times

$k = 500+3$ times

$\therefore i = \frac{n}{2}$

$j = n/2$

$k = 500 + \frac{n}{2}$ times

$500+1, 500+2, \ldots$ as the while loop's just immidiate above loop = j loop & j is determined by i i.e in j loop i & j are com. So. $500+1, 500+2, \ldots$

$\therefore$ the total no. of times the k loop is going to be executed.

$500+1 + 500+2 + 500+3 + \cdots + 500+n/2$

$= 500(1+2+\cdots+n/2)$

$= 500 + \frac{n(n+1)}{4}$

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

```
{
    stmt;
    k++;
}
}
}
```

500+1, 500+2, ---
as the while loop's just immidiate above loop = j loop
& j is determined by i
i.e in j loop i & j same conn.
So. 500+1, 500+2 - - -

$i = 1$

We are considering max, min like this if the con. inside if () will satisfiy then the stmt will execute otherwise not, i.e isn't sure, but if it will execute it will execute only for once, so it is min.

But while loop will be executing until the loop condition fails, so it is max.

$i = \frac{n}{2}$

$j = n/2$

$k = 500 + \frac{n}{2}$ times

(20) $n/2$

while ($n>1$)

{ $n = n/2$; [$n/2$, $n/4$ -- $n/10$ doesn't matter]

}

while loops are interconnected, then no need of thinking about [T, L]

while there is only 1 with some advance condition not with basic conditions, then directly think about [T, L]

Let us consider,

$n = 2^{(1)}$ [whenever the value of $n=2$, the loop is going to be executed one]

→ 1 times

condition is satisfied

$n = 4^{(2)}$ → value of n is powers of 2

→ 2 times

2

1

$n = 2^k$

then the loop is going to be executed k times

$n = 2^k$

$\therefore k = \log_2 n$

$n = 30$ → value of n is not the powers of 2

$\log_2 30 \simeq 4.907$

$T = O(\lfloor \log_2 n \rfloor)$

4 times

15

7

3

1

$\therefore \boxed{T = O(\lfloor \log_2 n \rfloor)}$

while dividing if not isn't then throw remaining part

while taking not 2^k format consider n = big value

(21) $n = 2^{2^k}$

for (i = 1; i<=n; i++) — n

{ j = 2;

while (j <= n) ← [loop this is depending upon how j is progressing i.e, on j, not for any prev case n wasn't dependent on any, it was independent but here n is dependent on k, so we have to trace, j & k]

{ j = j^2;

stmt ;

}

}

~~for each value of i, 2*n times~~

k = 1

n = 4

j = 2, 4 (3 times)

satisfied satisfied

$\therefore$ 2*n times (for each value of i, this is going to be happen)

k = 2

n = 16

j = 2, 4, 16

for k = 3, 4*n times

3*n times

doing like this as part of loop tracing, so this technique

$\therefore$ for kth iteration (for particular value of k) the statement is going to be executed

$k \longrightarrow (k+1)*n$

for (i = 1, k = n, i ++) → n

{ j = 2;

i = n (for every i's)

~~i = 2 it i (tracing i)~~
~~at 1st 2~~

k = 1, n = 4

j = 2, 4 → 2 times * n (for)
1 time 2nd time

k = 2 → n = 16

j = 2, 4, 16 → 3 times * n
1st time 2nd time 3rd time

k = 3, n = 256

j = 2, 4, 16, 256 → 4 times * n

⋮

k = k, $n = 2^{2^k}$

$j = 2, 4, \ldots, 2^{2^k}$ → k+1 times * n

at kth iteration (as just 1 loop is affecting) ⇒ $j = n$, $n = 2^{2^k}$ [as both j & n are depending so we are choosing both]

~~∴ T(n) = 2*n + 3*n + 4*n + ... + (k+1) n~~

∴ (k+1) * n = T(n)

now, $n = 2^{2^k}$ ⇒ $k = \log(\log n)$

∴ T(n) = (log(log n) + 1) * n

∴ $T = O(n \log \log n)$ (log 2 n)

as, partially this terminate

∴ for kth iteration (for particular i value) the statement is going to be executed

$\because n = 2^k$

$\Rightarrow k = \log_2(\log_2 n)$

$n * (\log_2 \log_2 n + 1)$ $[\because k = \log_2(\log_2 n)]$

$\boxed{TC = O(n \log_2 \log_2 n)}$ [as $n \log_2 \log_2 n$ is a leading term]

Types of Time Function (Class of the algo.)

- $O(1) \longrightarrow$ constant
- $O(n) \longrightarrow$ linear
- $O(n^2) \longrightarrow$ quadratic
- $O(n^3)$ — cubic
- $O(2^n)$ or $O(3^n)$ or $O(n^n)$ — exponential
- $O(\log n)$ — Logarithmic

Try to do more examples from net.

- The another name of TC is time function. The diff. is time function is defined by only an equation & denoted by $f(n)$ & the TC is defined by the highest order of the $f(n)$ & denoted by big-O [like $O(n)$] (n/m or something that is the variable of that algo)

Constant time function — A constant function is a function whose (output) value is the same for every input value. E.g. ~~$f(n) = 0$~~ $TC = O(1)$ if you give any i/p value like $n/m = 4, 5, 10$ ... etc the TC will be the same.

Linear time function — we will get linear time complexity (function) when the running time of an algo. increases linearly with the size of input. This means that when a func has an iteration that iterates over i/p size n, it is said to have a time complexity of linear time function — $O(n)$ e.g.

Quadratic time function — Quadratic time represents a function whose performance is proportional to the square of the input given. E.g. Quadratic time — $O(n^2)$

Cubic time ~~algorithm~~ function — ~~An~~ we will get cubic time complexity (function) for an algo. if ~~an~~ the algo. is said to run in cubic time if the running time of the three loops is proportional to the cube of N. When N triples the running time increases by N*N*N. E.g. Cubic time function — $O(n^3)$

Exponential time function — we will get an exponential time complexity (function) for an algo, if that case happen. An exponential time algo is one whose running time grows as an exponential function of the size of its input. E.g. exponential time function: 2^n or 3^n etc.

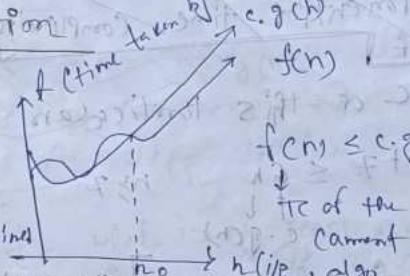
Logarithmic time function — Logarithmic t.c (function) $\log(n)$: Represented in Big O notation as $O(\log n)$, when an algo has $O(\log n)$ running time, it means that as the input size grows the no. of operations grows very slowly. E.g. binary search.

8/8/23

Asymtotic notation

Big on O —

O indicates the worst time that will be required to solve the particular problem. Our objective is to find out worst tc for the ip value above n_0.



$f(n) \le c.g(n)$

↓

tc of the current algo.

$c, n_0 \rightarrow$ constant real values

$c > 0$, $n_0 \ge 1$

$n \ge n_0$

[tc / rate of growth of time to solve the an algo.]

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

(after $n = n_0$, the $c.g(n) > f(n)$ always)

∴ the tc for that algo.

$\boxed{f(n) = O(g(n))}$

Suppose after analysing a particular algo we get, $f(n) = 5n+7$, can we say that tc of that particular algo $O(n)$?

means have to see $O(n)$, $g(n)$ or not / $n = g(n)$ or not

$f(n) = 5n+7$

$5n+7 \le c.g(n)$

$5n+7 \le 5n+7n$ $\quad n \ge 1$ $\quad$ [$5n+7n$ → we have considered 'n']

$\le 12n$ $\quad n \ge 1$

$(\le c.g(n))$

∴ $f(n) \le 12n$ $(c.g(n))$

~~Exponential time function — we will get an exponential time complexity (function) if~~

∴ tc of this particular algo is $O(n)$.

[• $5n+7 \le 6n$ $\quad n \ge 7$
(another consideration to prove that) $\quad \uparrow\downarrow$ $c \cdot g(n)$
$f(n) = O(n)$] // not too needed

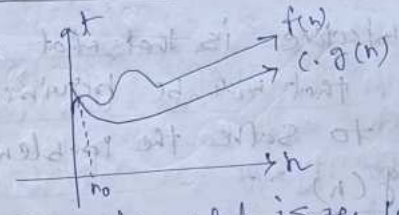
[$f(n) = 5n+7$ $\quad n \ge 1$
$5n+7 \le 12n^2$ $\quad n_0 = 1$ } It is also correct for $f(n)$ [that particular algo]
$f(n) = O(n^2)$

n^3, n^4 — so on also correct but we will always go for the tightest lower bound

[but / but try to find out the tightest / more closest upperbound of that algo]

[But ~~$f(n) = O(\sqrt{n})$~~ → this is now correct.
↳ as this isn't not tightest lowerbound the lowerbound is less than the actual bound]
$(1 > \frac{1}{2})$

2) Big Omega Ω



Our objective is what is the least amount of time required for our algo / to solve the particular problem.

if, $f(n) \ge c \cdot g(n)$ ← [this cond. is always satisfied for all n where $n \ge n_0$]
$c, n_0 \quad n \ge n_0$

then, $\boxed{f(n) = \Omega(g(n))}$ ← the t.c. of this algo

Suppose, $f(n) = 4n+5$, what is the order of this algo?

$4n+5 \ge 1 \cdot n$, $n \ge 1$, $n_0 = 1$ $(\because n \ge n_0)$
$\uparrow\uparrow$
$c \quad g(n)$

$f(n) = \Omega(n)$, $c = 1$, $n_0 = 1$
where

∴ tc of this algo $\Omega(n)$.

3. Big theta Θ

Here the objective is that what is the avg time that will be required by the algo. to solve the problem.

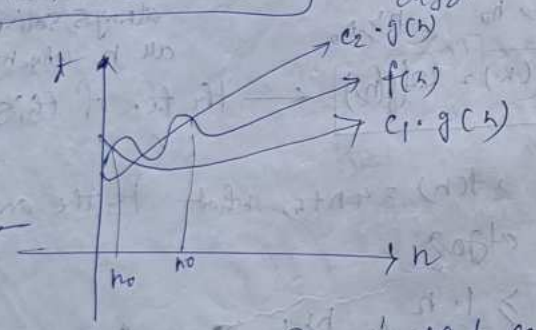
$g(n)$

$n \geq n_0$

$c_1, c_2 \rightarrow$ constant

if, $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$\boxed{f(n) = \Theta(g(n))}$ ← tc. of this algo.



• In some algo $O = \Omega$, in that case we will go with that Θ notation.

→ v $f(n) = 3n+5$, what will be the tc ?

Suppose

$\because 3n+5 \leq 8n \quad n \geq 1$

and $3n+5 \geq 1 \cdot n \quad n \geq 1$

$\underset{c_1}{1 \cdot n} \leq 3n+3 \leq \underset{c_2}{8 \cdot n}$ $\quad [g(n) = n]$

$\therefore$ ~~the~~ $f(n) = 3n+5 = \Theta(n) \quad \forall\, n \geq 1$

where $c_1 = 1$, $c_2 = 8$

$\therefore$ $\boxed{\text{tc of this algo} = \Theta(n)}$

Asymptotic notation – This is used to describe the running time of an algorithm [how much time an algorithm takes with a given input, n]. There are 3 different notations: big O, big Theta (Θ), big omega (Ω)

Big O notation – It is a tool used to describe the time complexity of algorithms. [It calculates the time taken to run an algo as the input grows.] In other words, it calculate the worst-case time complexity of an algorithm. [Big O notation in Data structure describes the upper bound of an algorithm's runtime.]

Big omega notation – It is defined as lower bound & lower bound on an algorithm is the least amount of time required (the most efficient way possible [in other words best case]). Just like O notation provide an asymptotic upper bound, Ω notation provides asymptotic lower bound.]

Big O notation – (In simple language)

Big O notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm. ①

10/8/23

CW

Recursion – It's a function is a func that calls itself repeatedly.

```
f(n)
{
  if(.....)
  return f(n-1);
}
```

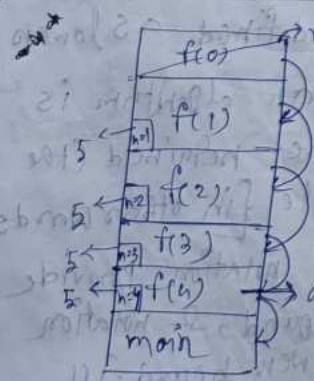
← E.g

```
f(n)
{
1. if (n<=0)
2. return ;
3. else
4. ~~return~~ f(n-1);
5. Print (n);
}
```

base condition / stopping / anchor

↳ recurrence relation = T(n) = T(n-1) + c

1) Recursion stack method / Stack method



removed from the stack

control return to that cal.

o/p = 1, 2, 3, 4

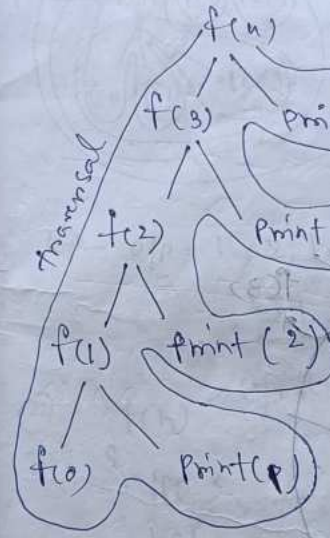
activation record

It is time consuming method

2) Recursion tree –

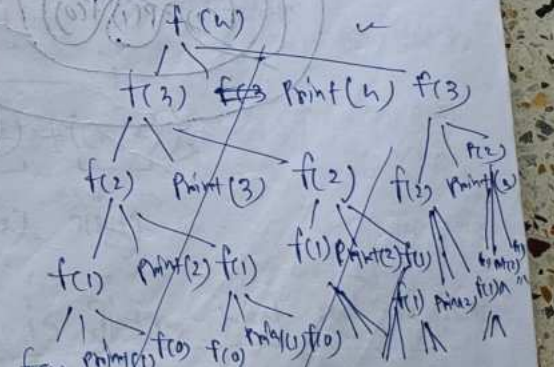
(Here we will ignore the base/anchor condition, we will only consider the statement that is ~~done~~ going to execute repeatedly in that recursive programme) recursive statement

n=4



o/p = 1, 2, 3, 4

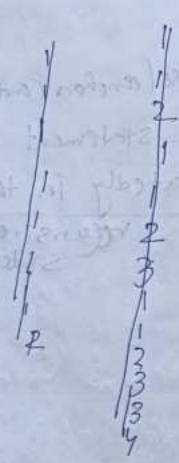
n=4

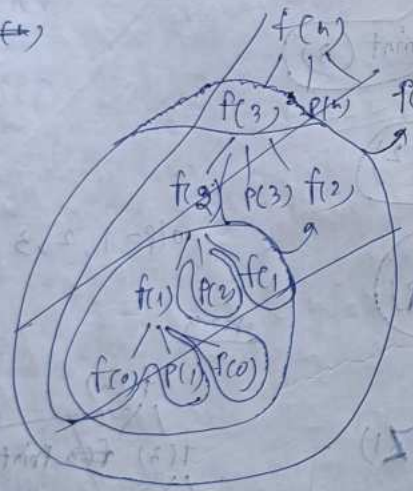


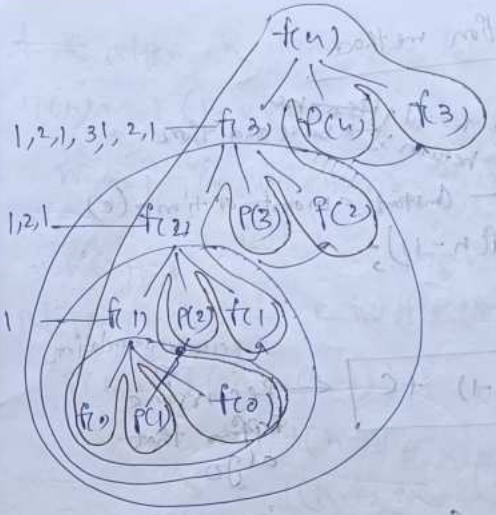
① f(n)

```
if (n<=1)
return ;
f(n-1);
print(n)
f(n-1);
}
```

↳ recurrence relation = T(n) = 2T(n-1) + c







search from root & try to soln

O/P = 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, ~~1, 2, 1, 3,~~
2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1

(22) $f(n) \rightarrow T(n)$

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

```
f(n)
{
  if( )
  return ; — 1
  else ———— 1
  return ( f(n/2) + f(n/2) );
}
```

how to construct recurrence relation from algo

$f(n/2) \rightarrow T(n/2)$, $f(n/2) \rightarrow T(n/2)$

If recursive exp is given, what will be t.c of that algo. This has 3 methods →

1) Back substitution method

```
f(n)
{
    if(n>1)  — (instant amount of time (c))
    return (f(n-1));
}
```

if n==1 then return → base condition

$$\boxed{T(n) = T(n-1) + c}$$ ← Recursive relation or Recursive approach that algo

$T(n) = T(n-1) + c$ —— ①

$T(n-1) = T(n-2) + c$ —— ②

$T(n-2) = T(n-3) + c$ —— ③

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

Put ② in ①

$T(n) = [T(n-2)] + c = T(n-2) + 2c$ —— ④

Put ③ in ④

$T(n) = (T(n-3) + c) + 2c$

$T(n) = T(n-3) + 3c$ —— ⑤

$$T(n) = \begin{cases} T(n-1) + c & n>1 \\ c & \text{otherwise} \end{cases}$$ } exact recursive approach this algo

After k iterations we will get, (after k th iteration if this loop will exit then n=1)

$T(n) = T(n-k) + k \cdot c$ (exit cond.)

$n-k=1$ (as when $n==1$ then the loop will exit)

$\Rightarrow k = n-1$

Put the value of k in the eqn,

$T(n) = T(n-(n-1)) + (n-1) \cdot c$

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

$= c + (n-1) \cdot c$ [∵ $T(1) = c$, c time will be returned to perform $T(1)$]

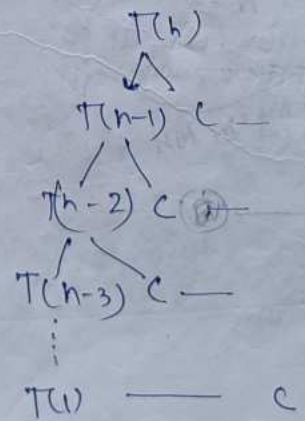
$T(n) = n \cdot c$

∴ the tc of this algo, $O(n)$ [∵ $T(n) = n \cdot c$ → $g(n)$] (at worst case)

Disadvantage – Its very slow, its take time.

2) Recursive tree method

(Same in above)



$n * c$

total no. of levels — constant time taken by algo

∴ $T(n) = c * n$ [∵ $T(n) = \Theta(n)$]

∴ TC of this Problem $O(n)$

③ $T(n) = \cancel{2(n)} \cdot 2T(n/2) + c$

1) Back substitution method –

[whenever we are with T(n) we must be able to break that T into two parts]

$T(n) = 2T(n/2) + c$ —— ①

$T(n/2) = 2T(n/4) + c$ —— ②

$T(n/4) = 2T(n/2^3) + c$ —— ③

$\vdots$

$T(n/2^k) = 2T(n/2^{k+1}) + c$ —— (k)

k+1 no. of levels

when $n = 2^k$ then $T(n/2^k) = 1$, that case we don't have to split the problem again & we have to repeat only constant amount of time & the problem will be solved, so,

$n = 2^k$

$\Rightarrow k = \log_2 k$

put ② in ①

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

$= 2^2 T(n/4) + 3c$ —— ④

at kth iteration,

put ③ into ④

~~$T(n) = 2^k T(n/2^k) + (k+1) \cdot c$~~

$T(n) = 2^3 [2T(n/8) + c] + 3c$

$= 8T(n/8) + 7c$

— at kth level,

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

$= n \cdot T(1) + (n-1) \cdot c$ $[\because n = 2^k]$

$= n \cdot c + (n-1) \cdot c$

$= nc$

$\therefore TC = O(n)$

[master's theorem, $f(n) = c$ ~~$\log_b a$~~

$n^{\log_b a}$

$= n^{\log_2 2}$

$= n^1$

$= n$]

Not needed (though it is correct)

11/8/23

(do prev no need of this)

Back substraction method,

① $T(n) = 2T(n/2) + c$ —— ①

$T(n/2) = 2T(n/4) + c$ —— ②

$T(n/4) = 2T(n/8) + c$ —— ③

⋮

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

put ② in ①

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

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

$= 2^2 T(n/4) + 3c$ —— ④

put ③ in ④

2^2, $2^1 c$
2^3, $4c + 2c + c$

$T(n) = 2^2 (2T(n/8) + c) + 3c$

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

$= 2^3 T(n/8) + 4c + 2c + c$

$n = 2^k$ (at k th iteration (at this condition the loop will exit))

$= 2^k T(n/2^k) + 2^{k-1} \cdot c + \dots + 4c + 2c + c$

$= 2^k T(1) + 2^{k-1} \cdot c + \dots + 4c + 2c + c$

the recurrence relation,

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

$\therefore 2^k \cdot c + 2^{k-1} \cdot c + \dots + 4c + 2c + c$

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

$= c(1 + 2 + 2^2 + 2^3 + \dots + 2^{k-1} + 2^k)$

$= c \cdot 1 \cdot \frac{2^{k+1} - 1}{2 - 1}$

$\left[\because S_n = a\left(\frac{r^n - 1}{r - 1}\right)\right.$

$t_n = a r^{n-1}$ (of GP)]

t_n — nth term

a = 1st term

r = common diff

$= c(2 \cdot 2^k - 1)$

$= c(2n - 1)$ $[\because n = 2^k]$

$\uparrow$ $\uparrow$

$\therefore$ $\Theta(n)$

$n - 1 = k$

k no. of terms

$\therefore n - 1 = k$ here

$\therefore T(c) = \Theta(n)$

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

Recurrence relation

Back substitution method,

$T(n) = T(n-1) + n$ —— ①

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

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

⋮

$T(n-k) = T(n-k) + (n-k+1)$ —— (k)

[Put ② in ①] not correct ✗

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

$= 4T(n-2) + 2(n-1) + n$

$= 2^2 T(n-2) + 2(n-1) + n$ — ④

put ③ in ④

$\Rightarrow T(n) = 2^2(2T(n-3) + T(n-2)) + 2(n-1) + n$

$= 2^3 T(n-3) + 2^2(n-2) + 2^1(n-1) + 2^0(n-0)$

at k^{th} iteration, $n = k$

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

the loop will exit when $n-k=1$

$\therefore = 2^{n-1} T(1) + 2^{n-2}(n-(n-2)) + 2^{n-3}(n-(n-3)) + n$

$= 2^{n-1} T(1) + 2^{n-2}(2) + 2^{n-3}(3) + \dots$

$= 2^{n-1} \cdot c + 2^{n-2} \cdot c + 2^{n-3} \cdot c + \dots + n$

Put ② in ①

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

$= T(n-2) + T(n-1) + n$ — ④

put ③ in ④,

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

at k^{th} iteration

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

the loop will exit at $n-k=1$ [Let us consider $n-k=1$, $\therefore k = n-1$]

[Here all k will be replaced by $n-1$ in reference]

$\therefore = T(1) + \cancel{n-k}\ 2 + \dots (n-2) + (n-1) + n$

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

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

[$n-(k-1)$, $\therefore n-k+1$, $\therefore n-k+1+1$]

[c means order of 1, so instead of $T(1)$ we can write 1]

$\therefore T(n) = O(n^2)$

[$\because \frac{n(n+1)}{2} = 1+2+\dots(n-2)+(n-1)+n$ (st natural no.s)]

[Not all problems can be solved by back substraction or recursive tree methods neither master's theorem. If it's solved by master's theorem stop. But if ~~it's~~ it can't be solved by master's theorem, then try to solve it by recursion tree method, if

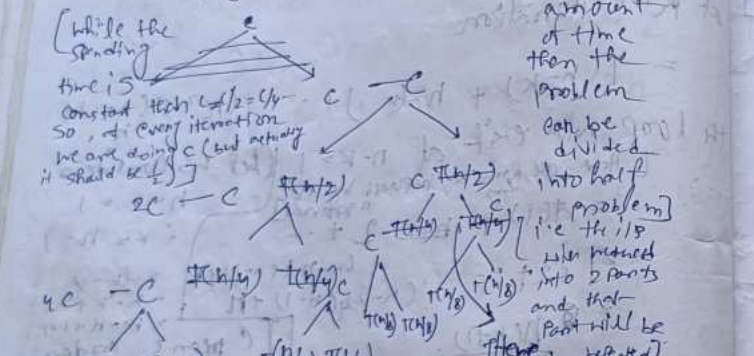
it also fails then think of back subs. traction method. 1st you have to see the problem deeply & then choose one method to solve the problem]

Recursive tree method

(3) $T(n) = 2T(n/2) + c \; ; \; n>1$
$= c \; ; \; n=1$

[Significance] If we spend constant amount of time then the problem can be divided into half problem] i.e the i/p size reduced into 2 parts and that part will be repeated

[while the spending time is constant then $c/2 = c/4$ so, at every iteration we are doing c (but actually it should be $\frac{c}{2}$)]



[If we consider $n=2^k$, ultimately we can reduce that problem size into T(1)]

[here, i/p size will be n/2 but it will have 2 parts of same i/p]

$\therefore T(1) \; \cdots \; T(1)$ — $n \cdot c$ [$\because 2^k = n$]

$\therefore T(n) = c + 2c + 4c + 8c + \cdots n \cdot c$ [at $T(n/n)$]

$= c(1 + 2 + 4 + 8 + \cdots + 2^k)$ [$\because n = 2^k$]

$= c \cdot 1 \cdot \left(\frac{2^{k+1} - 1}{2 - 1}\right)$ (g.p with k+1 terms) [$1 + \cdots 2^k$ → k+1 no. of terms]

$= c \cdot (2^k \cdot 2 - 1)$ [$\because$ ratio $= 2$]

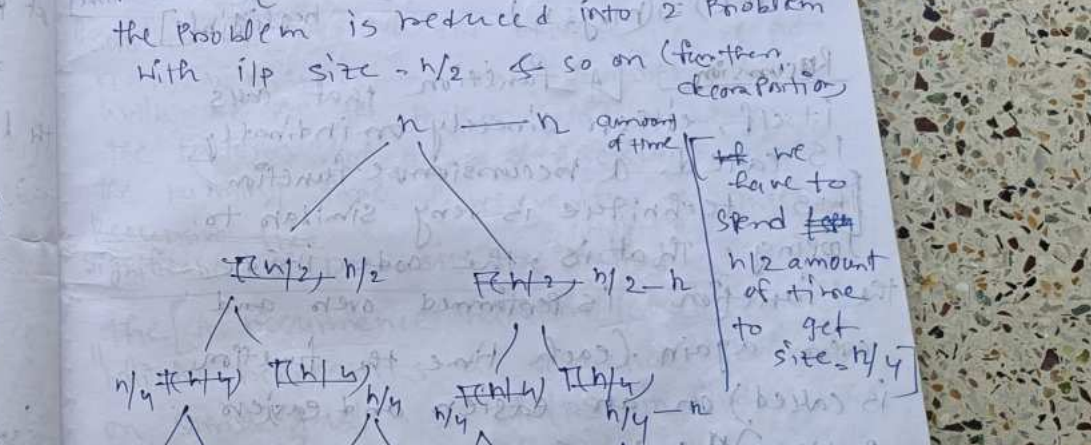
$= c \cdot (2n - 1)$ → gen (order of the polynomial)

$= O(n)$

$\therefore T(c) = O(n)$

(4) $T(n) = 2T(n/2) + n \; ; \; n>1$
$= 1 \; ; \; n=1$

If we spend n amount of time then the problem is reduced into 2 problem with i/p size = n/2 & so on (further decomposition)



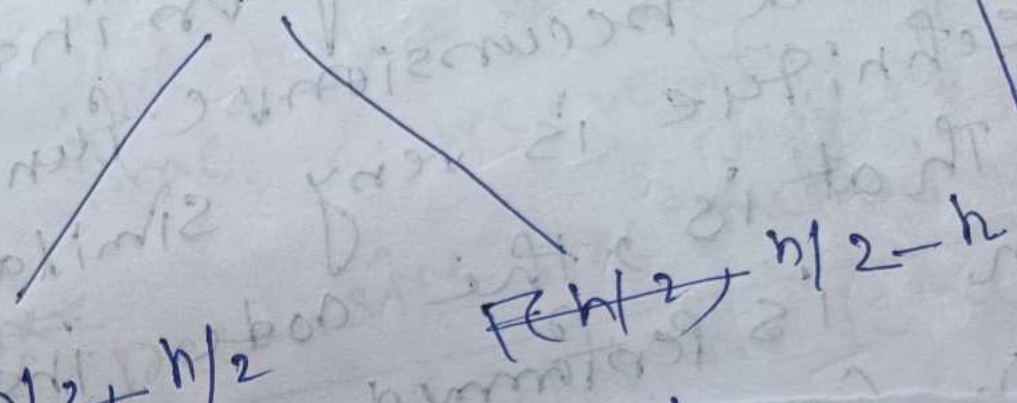
[~~If~~ we have to spend ~~[illegible]~~ n/2 amount of time to get size = n/4]

if we consider $n = 2^k$ the size of the problem will be T(1)

$T(n/n) \; \cdots \; T(1) \; \cdots \; T(1)$ — n

[the no. of such levels is (k+1), no. of levels $\underbrace{n/2^0, n/2^1, n/2^2 \; \cdots \; n/2^k}_{k}$ +1]

~~total no of~~ height / depth of the tree = k+1



spend n/2 of to

$= c(1 + 2 + 4 + 8 + \cdots + 2^k)$

$= c(\underbrace{2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^k}_{k+1 \text{ no. of terms}})$

$= c(1$

$T(n/n) \cdots T(1) \cdots T(1)$

$[\because 2^k = n]$

no. of such levels is (k

$n/2^2$

[for simplicity we are considering $h = l$ not $h = l+1$, but in reality $h = l+1$]

$T(n) = n(k+1)$ [as there are $(k+1)$ no. of levels & each level is taking n amount of time]

$= n(\log_2 n + 1)$ [$\because n = 2^k$]

$= n\log_2 n + n$ [n will be negligible]

$\therefore T(n) = O(n\log_2 n)$

Recursion - [A function that calls itself, either directly or indirectly is called a recursive function. This technique is very similar to looping. That is, the code inside the function is performed over and over again (each time the function is called) on an easier and easier problem]

Recursion stack method/ stack method -

The stack algo (SA) ~~(Tarjan, 1969)~~ is a metric first search performed iteratively. An ordered stack S containing previously examined paths (of different lengths in bits and in symbols) is maintained.

Recursion tree method - It is a way of solving recurrence relations. In this method, a recurrence relation is converted into recursive trees. [Each node represents the cost incurred at various levels of recursion. To find the total cost, cost of all levels are summed up.]

Back substitution method - In backward substitution, we do the opposite i.e we put $n = n, n-1, n-2$ — or $n = n, n/2, n/4$ — until we see the pattern. After we see the pattern, we make a guesswork for the running time & we verify the guesswork.

~~Recursive tree~~

Significance of recursive tree method -

~~The~~ [A recurrence relation is an equation which represents a sequence based on some rule. It helps in finding the subsequent term (next term) dependent upon the preceding term (previous term). If we know the previous term in a given series, then we can easily determine the next term.] It is imp method to solve recurrence relations. Generally, these recurrence relations follow the divide & conquer approach to solve a problem, e.g $T(n) = \cancel{T(n/2)} + T(n-1) + T(n-2) + k$, is a recurrence rela

-tion as Problem size 'n' is dividing into Problems of size n-1 & n-2.

(2) 17/8/23

Ch

③ Master's theorem

Comparing two functions

① $f(n)$ $g(n)$ (we have to see wheter $f(n) \geq g(n)$ or $f(n) = g(n)$ or $f(n) \leq g(n)$)

$\downarrow$ $\downarrow$

2^n n^2

apply log on both sides.

$2^n \longrightarrow n\log_2 2$

$n^2 \longrightarrow 2\log n$

let, $n = 1024$

$\therefore n\log_2 2 = 1024 \times 1$

$2\log n = 2\log_2(1024)$

$= 2\log_2 2^{10}$

$= 2 \times 10$

$= 20$

$\therefore 2^n$ is assymptotically bigger ($>$) than n^2.

when tc = constant, then $O(1)$

the tc relations (which are $\leq$ or $\geq$ than which one)

$1 \leq \log n \leq \sqrt{n} \leq n \leq n^2 \leq n^3$ — $\leq n^n \leq 2^n \leq$

3^n — $\leq n\log n \leq$

$\leq n^n$

② $f(n) \leq g(n)$

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

(log n base 2)

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

let $n = 2^{128}$ $= \frac{1}{2}\log_2 2^{128}$

$\therefore = \log_2(\log_2 2^{128})$ $= \frac{1}{2} \times 128$

by

$= \log_2(128)$

$= \log_2 2^7$

$= 7$ [You can double check here also by taking n a more greater no.]

$f(n) < g(n)$

③ $n^2 \geq n\log n$

applying log on both sides

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

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

let $n = 2^{2048}$

[If we have very small value we can't decide that which one is bigger]

$= 2\log_2 2^{2048}$

$= 2 \times 2048$

$= 4096$

$= \log_2 2^{2048} + \log_2(\log_2 2^{2048})$

$= 2048 + \log_2(2048)$

$= 2048 + \log_2 2^{2^{10}}$

$= 2048 + 11$

$= \cancel{2068}\ 2059$

$\begin{array}{r} 2048 \\ 2 \\ \hline 4096 \end{array}$

④ $f(n) \geq g(n)$

$\sqrt{\log n} \qquad \log(\log n)$

$= \frac{1}{2}\log_2(\log_2 n) \qquad = \log_2(\log_2(\log_2 n))$

let, $n = 2^{2^{128}}$

$= \frac{1}{2}\log(\log_2 2^{2^{128}})$

$= \frac{1}{2}\log_2(2^{128})$

$= \frac{1}{2} \times 128$

$= 64$

[$\log_2$ is taken for simplicity]

$= \log_2(\log_2 \log_2 2^{2^{128}})$

$= \log_2(\log_2 2^{128})$

$= \log(2^7)$

$= 7$

$\geq$ = at some fixed values we will get exactly the same value for 2 times.

[Eg. $\begin{array}{ccc} n & 2^n & n^2 \\ 1 & =2 & =1 \\ 2 & =4 & =4 \end{array}$]

⑤ $f(n) \geq g(n)$

$n\log n \qquad \sqrt[8]{n}$

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

Let, $n = 2^{2^{128}}$

$= \frac{1}{2}\log_2 2^{2^{128}}$

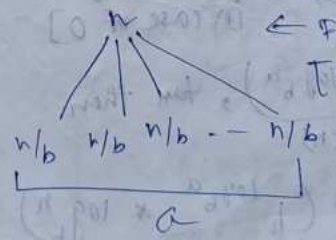
$= \frac{1}{2} \times 2^{128}$

$= \log_2 2^{2^{128}} + \log(\log_2 2^{2^{128}})$

$= 2^{128} + \log_2(2^{128})$

<u>master's theorem</u> (Not for all recurrance relation the master's theorem is applicable). The recurrence relation is needed to ~~be~~ convert into that format -

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



← Intuition behind that [If the i/p size n is should be divided in n/b no. of n/b & this n/b no. of n/b should be equal to a]

Here always, if, $a \geq 1$, $b > 1$ then only the master's

theorem is applicable.

[As, if $b = 0.5$

then $n/0.5 = n*2$, we are complicating the i/p size for 2 times.

So, the problem will never going to be solved]

when,

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

$f(n)$ ← '+' sign that can't be solved ~~in~~ by master's theorem

3-P (numerical)

Master's theorem

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

1. If $f(n) = O(\underbrace{n^{\log_b a - \epsilon}}_{g(n)})$, then

(asymptotically ~~lesser than~~) $<$

$T(n) = O(n^{\log_b a})$ [we can use θ in case of O]

2. If $f(n) = \theta(\underbrace{n^{\log_b a}}_{g(n)})$, ~~then~~ then

(asymptotically equal) $=$

$T(n) = \cancel{n^{\log_b a}}\ \theta(n^{\log_b a} * \log_b n)$

3. If $f(n) = \Omega(\underbrace{n^{\log_b a + \epsilon}}_{g(n)})$, then

(asymptotically ~~lesser than~~) $>$

$T(n) = \theta(f(n))$ [$\epsilon > 0$, constant]

① $T(n) = \underset{a}{4}T(n/\underset{b}{2}) + \underset{f(n)}{n^2}$ $a \geq 1$, $b > 1$

$f(n)$: n^2

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

n^2 & $n^{\log_b a}$ are exactly the same.

$\therefore$ TC of this algo $\theta(g(n) * \log_b n)$

$= \theta(n^2 * \log_2 n)$

18/8/23

Class

$T(n) = aT(n/b) + f(n)$ $a \geq 1$, $b > 1$

Ex-① $T(n) = \underset{a}{4}T(n/\underset{b}{2}) + \underset{f(n)}{n^2}$ (without remembering that particular method, trick)

$f(n) = n^2$

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

$\therefore T(n) = O(n^2 \log_b n)$ [the orders of f(n) or g(n)]

[If they are equal then take anyone]

② $T(n) = 3T(n/2) + n^2$

a, b

$f(n) = n^2$ $\quad > \quad n^{\log_b a} = n^{\log_2 3} = n^{1.5}$

$[\log_2 3 = 1.\text{Something}]$ $\left[\because \frac{\log(3)}{\log(2)}\right]$

$\therefore T(n) = O(n^2)$

③ $T(n) = 16T(n/4) + n!$

a, b

$f(n) = n!$ $\quad > \quad n^{\log_b a} = n^{\log_4 16} = n^{\log_4 4^2} = n^2$

$[\because n! = n(n-1)(n-2)(n-3)\ldots 2 \cdot 3 \cdot 1]$

$\therefore T(n) = O(n!)$

④ $T(n) = 2T(n/4) + n^{0.67}$

a, b

$f(n) = n^{0.67}$ $\quad > \quad n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

$\left[\log_4 2 = \frac{1}{2}\right]$ $\left[\frac{\log(2)}{\log 4} = 0.5\right]$

$\therefore T(n) = O(n^{0.67})$

⑤ $T(n) = 2T(n/2) + n\log_2 n$

a, b

$f(n) = n\log_2 n$ $\quad > \quad n^{\log_b a} = n^{\log_2 2} = n$

$\therefore T(n) = O(n\log_2 n)$

⑥ $T(n) = \sqrt{2}\, T(n/2) + \log_2 n$

a, b

$f(n) = \log_2 n$ $\quad < \quad n^{\log_b a} = n^{\log_2 \sqrt{2}} = n^{\log_2 2^{\frac{1}{2}}} = n^{\frac{1}{2}} = \sqrt{n}$

$\therefore T(n) = O(\sqrt{n})$

checking, let $n = 2^{2^{1024}}$

$\therefore \log(\log_2 n) = \log(\log_2 2^{2^{1024}}) = \log(2^{1024}) = 1024$

$\quad < \quad \frac{1}{2}\log_2 n = \frac{1}{2}\log_2 2^{2^{1024}} = \frac{1}{2} \times 2^{1024}$

$\therefore \sqrt{n}$ is larger.

(*) $T(n) = 3T(n/2) + \underbrace{n\log_2 n}_{f(n)}$

~~mat~~ master's theorem can't be applied here. To solve this we have to use the back substitution method or recursion tree method.

Suppose,

$$T(n) = 0.7T(n/2) + n\log_2 n$$

$a \geq 1 \rightarrow \times$

Master's theorem ~~isn't~~ can't be applied here.

$$T(n) = 3T(n/1) + n\log_2 n$$

in this case we can't also solve the problem [$\because b > 1$ ×]

<u>Space complexity</u> – whenever we are executing a problem what is the amount of ~~s~~ RAM (memory) covered by that problem.

① A ( ) → func not an array

```
{
n, i, j;
for(i=1; i<=n; i++)
}
```

→ 1 word of memory will be required for n

$\longrightarrow 1+1+1 = 3 = S(n) = O(1)$

$\therefore S(C) = O(1)$

② ~~A(n)~~ A()

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

n (array)

$S(n) = 1+1+1+n = 3+n$

$\therefore S(C) = O(n)$

③ ~~A(A)~~ A(A) — A array is passed through an argument but it isn't used in this algo / whole process

```
{
i, j, n;
for(i=1; i<=n; i++)
}
```

$\longrightarrow 1+1+1 = 3$

$\therefore S(C) = O(1)$ [As the array is passed as an argument]

④ A1(n)

```
{
i, j, n, A[200][200];
for(i=1; i<=n; i++)
}
```

A → matrix

n×n → for 2 matrix

m×n, n×q → for 2 matrix

$m \simeq n$

$n \times n$

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

(5)

```
A()
{
i, j, n, A[200][200], B[200][300],
        C[200][300];
for(i=1; i<=n; i++)
{ ___
  ___
}
}
```

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

$\therefore S(C) = O(n^2)$

~~20/8/23~~

SC for recursive algorithms

(1)

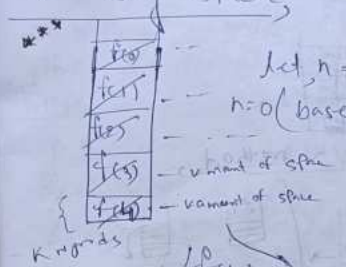
```
f(n)
{
if (n≥1)
{
f(n-1); // head recursion
Print(n);
}
}
```

(The recursive call is in the begining of the recursive algo)

(The - is made at end — tail recursion)

1) Head recursion
2) tail "
3) Body. (the recursion is made at the body)

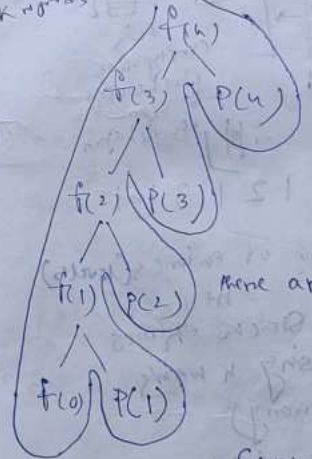
SC using stack



let, n = 4

n = 0 (base condition)

[while finding the tc of any recursive algo then we only consider the recursive statements & we ignore other statements]



→ traverse from root to bottom & left to write.

o/p → 1 2 3 4

there are n+1 no. of entries levels ~~(no. of elements)~~

(for n=4, there are 5 (n+1) entries)

$S(n) = (n+1) * K$ [∵ (n+1) entries & each entries are taking no. of amount of space]

$= nK + K$ [K = constant]

$S(n) = O(K \cdot n)$ [All recursive call are equal amount of variables]

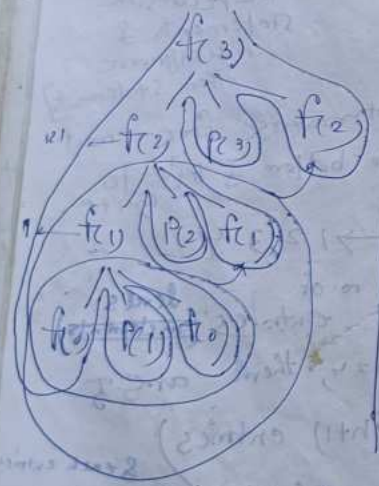
$= O(n)$

(memory = K) [∵ each recursive call K amount of mem is req]

② ***

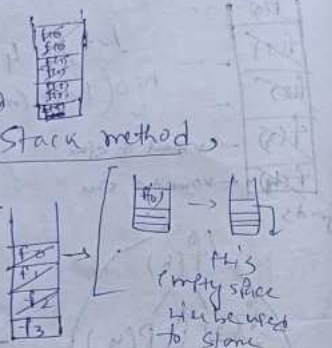
```
f(n)
{
  if(n≥1)
  {
    f(n-1);
    Print (n);
    f(n-1);
  }
}
```

let, n=3



if you can't construct the stack, construct the tree & get the no. of entries

Stack method,



o/p = | 2 | 3 | 2 |

the no. of entries (levels) = ht,

each stack entries are using k words (k memory)

$S(n) = (ht+1)*k$
$= hk+k$
$= O(hk)$
$= O(h)$

Divide & conquer

It is a programming paradigm where the original prob is divided into sub probs that is identical to the original, that sub problem will be divided into other sub problems & this will go on until the subproblem becomes very small.

Imp Steps –

1) Divide
2) Conquer
3) combine

***
(1) quick sort
(2) merge
(3) Binary search
(4) Strassen's matrix multiplication

this algo use divide & conquer method

Binary search

Best case tc = ~~to~~ O(1)
Worst case tc = O(n)

0	1	2	3	4	5	6	7	8
2	5	8	11	12	15	17	19	23

low index i — upper index k

Given a list of element, sort it in ascending order)

Search element = 8 (x)

mid = $\lfloor (i+k)/2 \rfloor$

mid = 4

(the search space is reduced by 2 times now)

whether a[mid] == key

[If we get key at that 1st search, then tc() = O(1)]

now, mid $= (0+3)/2$
$= \lfloor 1.5 \rfloor$
$= 1$

a[mid] == key ? or not

0	1	2	3
2	5	8	11

mid

$A[2] = 8$ (key)

Binary search Algorithm

[Every iterative prog can be written in recursive prog & vice versa (i.e no one is better than other)]

```
Algo Binary search (a, i, k, x)
{
  if (k == i) then // base condition
  {
    if (x == a[i]) then
      return i; // i is the index of the searched element
    else
      return -1; // is not successful (key not found)
  }
  else
  {
    // P is divided into sub problems
    mid := ⌊(i+k)/2⌋;
    if (x == a[mid]) then
      return mid; // a[mid] = key value
    else if (x < a[mid]) then
      return BinRSearch (a, i, mid-1, x);
    else
      return BinRSearch (a, mid+1, k, x);
  }
}
```

(they are exactly capable the fn & sc may differ)

(small sub problem in terms of divide & conquer)

Time Complexity

$T(n) = aT(n/b) + f(n)$ [We have to divide the problem into a no. of n/b]

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

recursive function for the binary search.

$a = 1$
$b = 2$
$f(n) = 1$

[$T(n/2)$ as we are finding the mid]

applying back substitution method,

$T(n) = T(n/2) + 1$
$= (T(n/4) + 1) + 1$
$= T(n/4) + 2$
$= (T(n/8) + 1) + 2$
$= T(n/8) + 3$
$= T(n/2^k) + k$

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

31/8/23

Now

Applying master's theorem,

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

$f(n) = 1, \quad a = 1, \quad b = 2$

$f(n)$	$n^{\log_b a}$
1	$n^{\log_2 1}$
	$= n^0 = 1$

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

$= n^{\log_2 1} * \log_2 n$

$= 1 * \log_2 n$

$= \log_2 n$ $\quad [\because \log_2 1 = 0]$

$\therefore T(n) = O(\log_2 n)$

Insertion Sort

The objective of this sort is to insert the element in it's proper position.

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

$j-1$, $j=2$, i, 0, key = 2

$i = j-1 = 2-1$

$A[1] = 7 \rightarrow A[2] = 7$

$i = 1-1 = 0$

$\therefore A[1] = key = 2 \rightarrow$

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

$\rightarrow$

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

```
Insertion_sort(A)
{
    for (j = 2 ; j<=n ; j++)
    {
        key = A[j];      //insert the jth element at its proper position inside the Array A[1, ... j-1]
        i = j-1
        while (i>0 &&
                A[i] > key){
            A[i+1] = A[i];
            i = i-1;
        }
        A[i+1] = key;
    }
}
```

TC, $j = 2$ (1st iteration)

7 2 3 5 9 1 4

(1 + 1) (2 + 2) (3 + 3)

comparison, movement (almost 2 movement) (almost 3 movements)

at $n-1$ iteration,

$(n-1) + (n-1)$

∴ the no. of comparisons (at most this no of execution is returned)

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

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

$= 2(1+2+3+\ldots+n-1)$ → sum of 1st n-1 natural numbers $\left[\because \frac{n(n+1)}{2}\right]$

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

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

$T(C) = O(n^2)$ (worst case) [Pick 1st n natural no.s]

↳ this is the max amount of time to perform insertion sort (the worst case tc, in the worst case is the array ~~will be~~ is reversed)

Best case, the array is sorted.

1 2 3 4 5 7 9

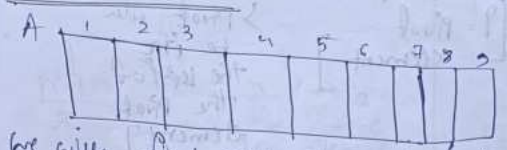
$(1+1+1 \ldots)$ ∴ $T(n) = 1+1+1+\ldots = n-1 = O(n)$

↳ only 1 comparison with 2 returned, no movement as it is already sorted (Best case tc)

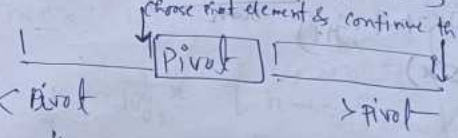
1/09/23

Chr

Quick Sort ***



we will find any element / choose any element from that array A, & it is called Pivote element. Our objective is that to place this Pivot element at the proper position of it. Quick Sort follow divide & conquer method.



this elements may not be in their proper position

this process will continue until we get just a single element.

This algo will contain 2 sub algos. & 2 function as well.

```
Partition (A, p, r)
{                          // Partition algo (that is a func)
}
```

[the purpose of the Partition algo – to divide the original list & return a list where all element < pivot will be in left of

Rough explanation –

```
Quick - Sort (A, p, r)
{
    if (p<r){
        q = Partition (A, p, r);
        Quick_sort (A, p, q-1);
```

Quick sort (A, q+1, r);
}
}

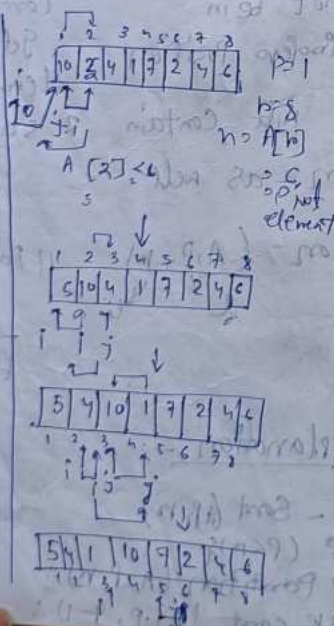
[q = pos of pivot element]

Pivot element & all elements > pivot will be in the left of the pivot element]

Algo

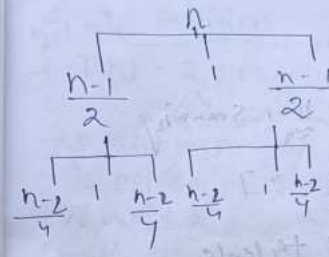
Partition (A, p, r)
{
x = A[r]; // x is the pivot element
i = p-1;
for (j=p; j<=r-1; j++)
{
if (A[j] ≤ x)
{
i = i+1;
swap A[i] with A[j]
}
}
swap A[i+1] with A[r]
}

(It works better when the no. of elements are low)



Space complexity

Best ~~Worst~~ case,



→ [as one element (pivot) is already in it's pos & it will divide the array in 2 parts so $\frac{n-1}{2}$]

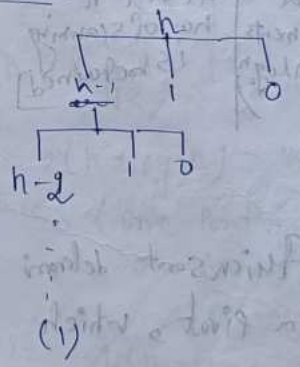
hight of the tree = $\log_2 n$

$SC = k * \log_2 n$ [$n-1 \approx n$]

Constant [k = the memory req for storing n elements]

∴ at ~~best~~ ~~worst~~ best case $SC = O(\log_2 n)$

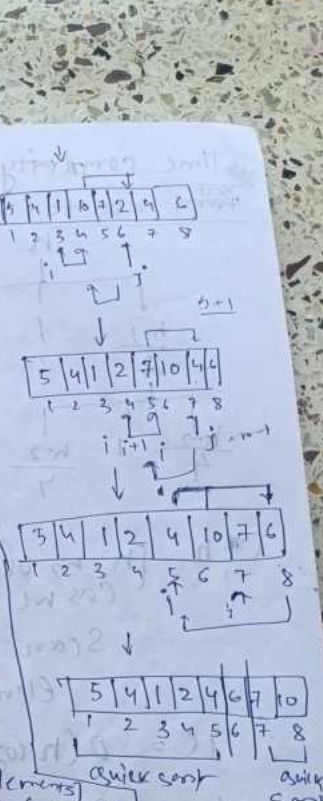
~~Best~~ worst Case,



no. of levels will be approximately = n

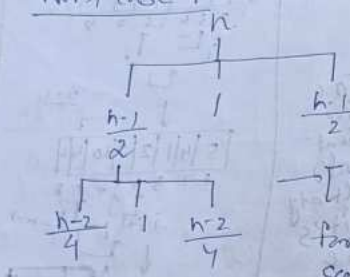
total space required = k * n [k = the total memory required for this sort]

∴ Worst case $SC = O(n)$ [∵ k = const]



Time complexity

Best ~~Worst~~ case tc:



→ [n-1 scanning ≈n]

→ [n-2 " " ≈n]

for all the levels scanning ≈n & there are n (actual n/2) of such levels.

(* n [n no. of scanning is returned as we know the quick sort scan from 1st to last-1 element)]

∴ TC = $O(n \log_2 n)$

[As there are n no. of levels & in each level n no. of scanning is returned]

- Worst case happen, when Ascending / descending / All equal elements → ($n \log_2 n$)
- otherwise in all cases, 1:4, 1:10, 1:99 (elements splited (already ratio))

Pivot element – [First, Quicksort determines something called a pivot, which is] // Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element.

5/9/23

CW recurrance relation

Imp q = Partition → pivot element

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

a, b

as partition is applied at the left & right of pivot element

as we are dividing the array in 2

applying master's theorem (best & avg case) → TC

$f(n) = O(n)$

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

$\therefore O(n^{\log_b a} \log_b n)$

$= O(n \log_2 n)$

∴ TC = $O(n \log_2 n)$ ← Best case tc

Imp Worst case → (One partition got no element & another partition will contain n-1 elements) for

$T(n) = T(n-1) + T(0) + O(n)$ ← Partitioning

$= T(n-1) + cn$ → time $O(n)$ (time not fixed for partition)

$= T(n-2) + c(n-1) + c \cdot n$

$\vdots$

$T(n-k) + c(n-k+1) + \cdots + c(n-i) + c \cdot n$

$\therefore T(1) + c \cdot 2 + c \cdot 3 \cdots + c(n-1) + c \cdot n$

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

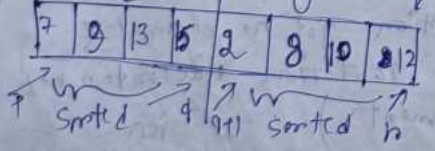
$= \frac{c \cdot n(n-1)}{2}$ $[\because 1+2+3+\cdots n-1+n = \frac{n(n-1)}{2}]$

$\cancel{T(n)} = 0$ $T(n) = O(n^2)$

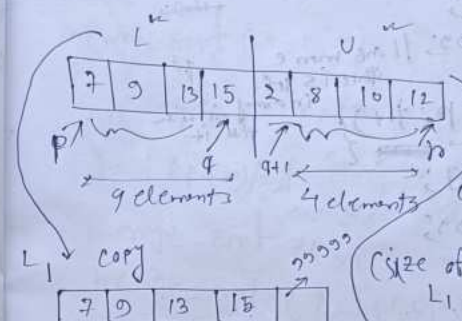
Merge Sort 7/9/23

It works better when the array size is high (much elements). so it works better. using the merging procedure we will apply the merge sort.

Merging Procedure



but the whole list is not sorted order (lower & upper list is in sorted order). After merging process the merged list will be in sorted order.



(this comparing by pointer i, j is called pointer method)

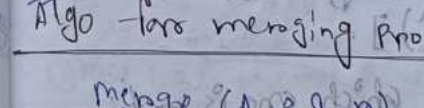
[In lab instead of creating 2 array we can use a single array where the elements will be like this format]

(size of L_1 = no of elements in $L+1$)

(same for another list)

(have keep very large no. at end of this 2 arrays)

the whole array (again)



[i, j, k will work for scanning the list elements]

$n \to$ even
$n/2$ $n/2$
$n \to$ odd
$\lfloor (l+u)/2 \rfloor$
$\frac{n-1}{2}$ $\frac{n+1}{2}$
$[n \to \frac{n(n+1)}{2}]$

Algo for merging procedure

```
merge (A, p, q, r)
{
    n1 = q - p + 1
    n2 = r - q
    create L1[n1+1] & L2[n2+1]
```

```
for(i=1; i<=q; i++){
    L1[i] = A[i];
}L1[i] = 99999; //one more space is left in array if we add it
for(j=q+1; j<=p; j++){
    L2[j] = A[j];
}L2[j] = 99999;
i=1, j=1;
for(k=p, k<=p; k++)
  if(L1[i] <= L2[j])
  {A[k] = L1[i];
   i=i+1;
  }
  else
  {
   //(L1[i] >= L2[j])
   A[k] = L2[j];
   j=j+1;
  }
}
```

$L1[0] = A[0] = \infty$

$L2[0] = \infty$

[<= as if $L1[i] = L2[j]$ then copy any one]

[so we are merging this = condition with < cond]

(*) In general merge sort is better as in across all cases, viz. best, avg, worst case [$T.C - O(n \log_2 n)$]

Among all sorting algos which one is best, merge sort is ~~most~~ ~~more~~ efficient & works faster than all other sorts in case of larger array size or data sets, whereas quick sort is efficient & works faster than merge sort in case of smaller array or data sets and all other sorts (*)

Merge sort (follows divide & conquer method)

Algo

```
merge_sort(A, p, r)
{
  if (p<r)   //if p=r then we are not going to call the recursive merge sort algo, so.
  q = ⌊(p+r)/2⌋
  merge_sort(A, p, q);
  merge_sort(A, q+1, r);
  merge(A, p, q, r);
}
```

$\left\lfloor \frac{0+7}{2} \right\rfloor = \lfloor 3.5 \rfloor = 3$

E.g.

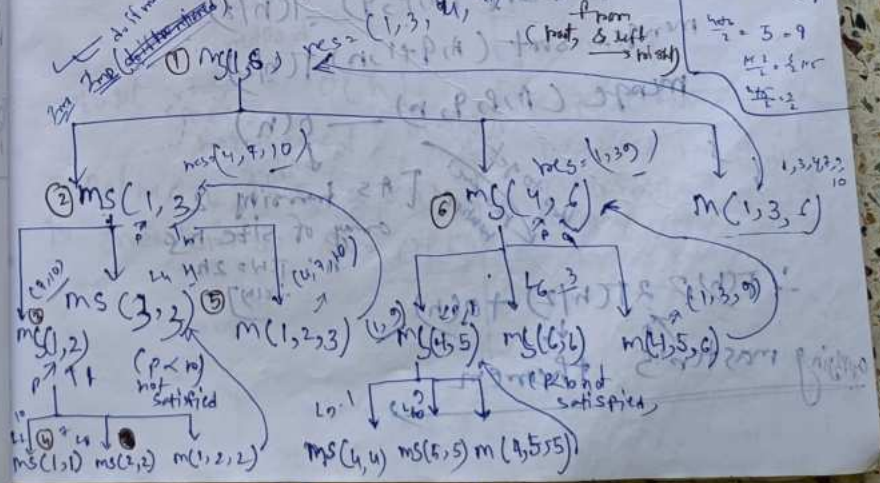
10	7	4	1	5	3
p=1	2	3	4	5	r=6

~~ms(1,6)~~

$q = \lfloor (1+6)/2 \rfloor = \lfloor 3.5 \rfloor = 3$

$\left\lfloor \frac{1+3}{2} \right\rfloor = \frac{4}{2} = 2$

$\left\lfloor \frac{4+6}{2} \right\rfloor = 5$



$L_1(10, 7, 4)$, $L_2(1, 3, 3)$ (from prev tree diagram) (indices are same as here too)

$L_3(10,7)$ $L_4(4)$ $L_5(1,3)$ $L_6(3)$

$L_7(10)$ $L_8(7)$ $L_9(1)$ $L_{10}(3)$

~~7, 10, 4~~

~~merge(1,1,2) n(3)~~

TC,

merge_sort (A, p, n) — T(n)

{

if (p < n)

{

q = [(p+n)/2]

merge_sort (A, p, q) — T(n/2)

merge_sort (A, q+1, n) — T(n/2)

merge (A, p, q, n) — O(n)

}

}

[As working 2 array of size n/2 n/2, T(n) = 2n/2 + O(n)]

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

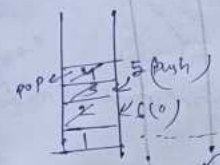
applying master's theorem

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

$\therefore TC = O(n \log_2 n)$ [$\because T(n) = O(n^{\log_b a} \cdot \log_b n)$]

$\therefore TC = O(n \log_2 n)$ [for all cases]

SC,



soon (by doing this we get)

no. of levels = n, ~~no. of elements~~ (n unit memory is occupied)

So ~~O(1)~~ $\log_2 n$ (∵ height of the tree = $\log_2 n$)

$\therefore$ $k \log_2 n$ → for stack procedure method

n → for merge procedure SC

$\leq$ [∵ n is greater]

[k = Const]

total space requirement = $O(n) + O(k \log_2 n)$ [the total memory required for storing n elements] (in stack procedure method) (merge sort)

Between this, $2O(n)$ is greater so $SC = O(n)$

8/09/23

[O(n) → searching the whole array]

Heap

	insert an elem	delete an elem	find min
unsorted A	O(1) (considering insert at end)	O(n)	O(n)
sorted A	O(n)	O(1)	O(1)
Linked List	O(1)	O(1)	O(n)
Heap	$O(\log_2 n)$	$O(\log_2 n)$	O(1)

8 10 11 13 17

O(n) – ls search for shifting the from array to = O(n)

O(log n) B.S.

as we have to just point to another node

Heap → It is a binary tree almost complete

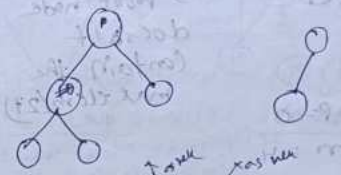
max heap → Root node (parent node) contains the max value

min heap → Here this (1) will be the Root min value of the heap

this above property is true for all the subtrees of the heap.

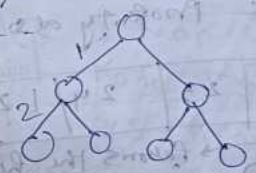
this almost complete

At binary tree the leaf node can be present at the last level & the above level of the last level's node will fill up from l → r. So we can't have a right child of a node that has no left child. tree.

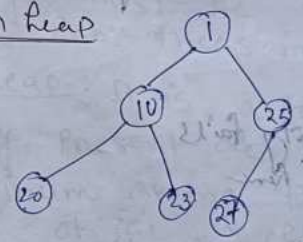


[Heap will take more memory than an array representation for pointers]

Complete binary tree



min heap



Here, no consideration like CBT or BT since R will be max from the LC

Array representation

1	2	3	4	5	6
1	10	25	20	23	27

Root L R

(2*1) (2*1+1)

LC = 2*1

R = 2*1+1

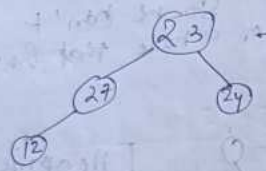
Consider this eg

i → root node

2*i → left child

2*i+1 → Right child

Q) Given, | 23 | 27 | 24 | 12 | → it is max heap or not?



→ this isn't a max heap as root node doesn't contain the max elem (27)

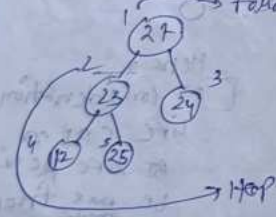
the size of heap = 0

∴ am = 4

the size property also fails here.

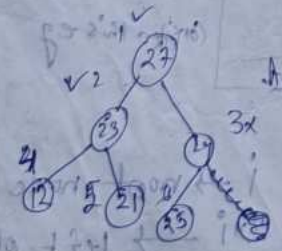
| 27 | 23 | 24 | 12 | 25 |

→ follows the heap property



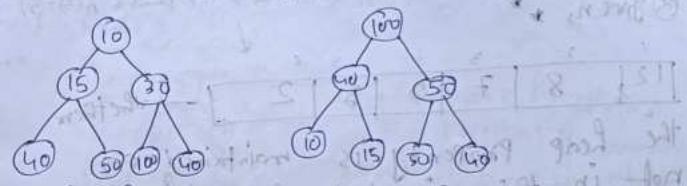
→ Heap property fails here

A. heapsize = 1



A. heapsize = 2 (the no. of nodes of subtrees including the root of the tree in which heap property is satisfying)

Heap — A heap is a special tree-based data structure in which the tree is almost a complete binary tree.



min heap max heap

Max heap: It is a type of heap where the key present at root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that almost Binary Tree.

Min heap: It is a type of heap where the key present at root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub trees in that almost Binary tree.

14/9/23

the parent node of the childrens.

$P(i) = \lfloor i/2 \rfloor$

eg, $i = 5$, $\lfloor 5/2 \rfloor = 2$

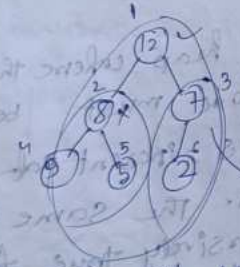
A(2) = 13 is parent of 5

A(2) = 8 is the parent of A(5) = 5

② given,

1	2	3	4	5	6
12	8	7	9	5	2

→ whether the heap property is maintained or not in this inserted sequence of array? (either max heap or min heap property)



length(A) = 6

len_Heapsize = 1

(as there is no ~~it~~)

After 3, all the nodes are leafs.

Leafs starts from index → $\lfloor n/2 \rfloor + 1$ to n

Here, $\lfloor 6/2 \rfloor + 1$ to 6 [n = no. of nodes]

n = 4 to 6

from 4 to 6 all the nodes are leafs.

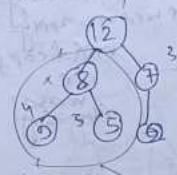
Non leafs = 1, 2, 3

Non-leaf nodes 1 (root) to $\lfloor n/2 \rfloor$

Here, 1 to 3 are non leafs [n = no. of nodes]

largest index among the non leafs = 3 (last non-leaf)

we will start a function max_Heapify() from the largest non-leaf index to the starting root node. (last non-leaf)

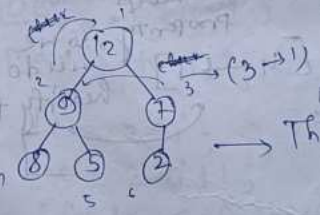


← this follows heap property

9, 5, 2 → no child so following heap property.

7 → has one child that is less than it so follows the heap property.

replace its largest child with it's root (subtrees)



(3 → 1)

max_Heapify() process.

→ This is a heap now

③

1	2	3	4	5	6	7	8
3	7	12	15	17	19	21	23

whether it is following max_Heapify property or not?

Descending array → max_Heapify (Apply) ← to create a max or get the max

Ascending array → min_Heapify (Apply)

replace its largest child with it's root (subtrees)

Property.

7 → has one child that's less than it so follows the heap Property.

for getting min element

if array,

desending order = apply min-h

ascending order = get min & apply min-h

for getting max elem

If array,

desending order = ~~get~~ max-h

ascending order = max-h

Create max-h array,

desending order = already max-h

ascending order = apply max-h

Create min-h array,

desending - min-h

ascending already min-h

Random array

depends upon

(2) near ascending → min-h

(3) near descending → max-h

Property or not?

Desending array → max-Heapify

Apply

Apply

to ~~create a~~ get the max or min ele

7 → has one child that is less than it so follows the heap property.

replace its largest child with it's root (subtrees)

where,

array descending order → already max-h → so apply min-heapify

— ascending — → already min-h → so apply max-heapify

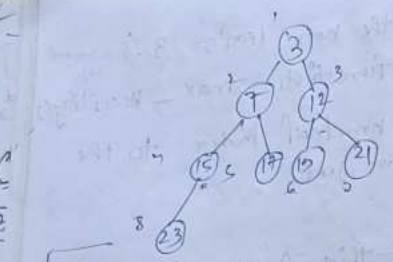
PAGE NO.
DATE: / /20

whether it is following max-Heapify property or not?

Apply → max-Heapify (to ~~create a~~ get the max)

Descending array → max-Heapify

Apply



[If it will in ascending order then it will be already sorted, no need of to apply min heap so, in ascending we need to apply max heap & vice versa (descending)]

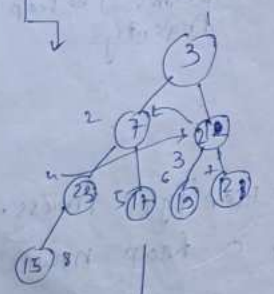
Leafs → $\lfloor n/2 \rfloor + 1$ to n

$= \lfloor 8/2 \rfloor + 1$ to 8

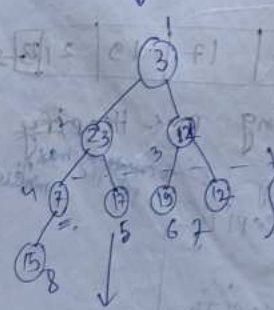
$=$ 5 to 8

Non leafs → 1 to $\lfloor n/2 \rfloor$

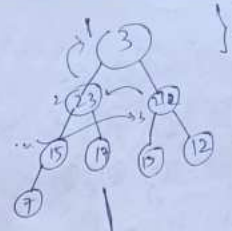
1 to 4.



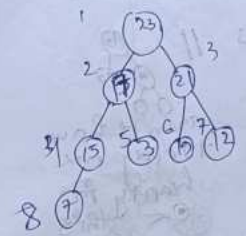
[All the leaf nodes follows the heap property, so we will do heapify from [4 → 1]]



[Check if after substituting the lower level follows heap property or not, if yes the process, if not then substitute again]



→ can merge this this



→ this is following heap property now

23	17	21	15	3	19	12	7

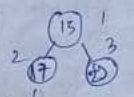
max Heapify Algo (create min heap also) (convert heap)

```
max_Heapify (A, i)
{
  lc = 2*i;
  rc = 2*i+1;
  if(lc ≤ A.Heapsize
     && A[lc] > A[i])
  {largest = lc;
  else,
   largest = i;
  if {rc ≤ A.Heapsize && A[rc]
        > A[largest])
  }
```

→ will check whether this particular node is having children or not (how long we will go, that will determine the limit)

$\boxed{A.Heapsize = A.length}$

(to ensure that we have applied the max heapify func for all nodes)



→ then max(A,i)

largest = RC;

~~else~~ }

~~largest = i~~

if (largest ≠ i)

{SWAP A[i] & ans with A[largest]

}

max_Heapify (A, largest); //

}

Create_max_Heap(A)

{

A.Heapsize = A.length;

for (i = ⌊A.length/2⌋ to 1) // ⌊A.length/2⌋ = last non leaf node as well as largest non leaf ~~ele~~ id* → (i)

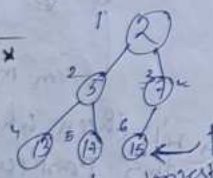
{max_Heapify (A, i);

}

}

again we have to apply max heapify for this

e.g.



A.length = 6 = A.Heapsize

⌊6/2⌋ to 1

3 to 1

this is largest

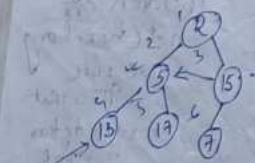
MH(A, 3)

LC = 6

LC = 6 = A.HS &&

A[LC] > A[i]

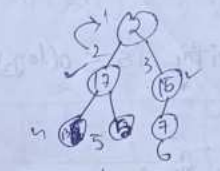
A[6] > A[3]



largest

MH(A, 6) → no condition will be satisfied so it will do nothing

Here LC = 4, RC = 5, 4 < 6, 5 < 6, i.e. i = 2 has 2 child at id 4 & 5.



MH(A, 2)

LC = 4, RC = 5

LC = 4 < ~~A.flag~~ A.HS = 6

A[4] > A[i]

A[4] > A[2]

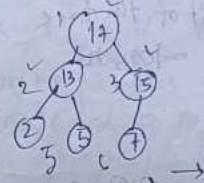
13 > 5

2nd loop,

17 > 13

A[RC] > A[largest]

A[6] > A[4]

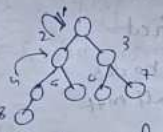


will do the same above & this tree will follow the heap property now. max-Heapify has done.

17	13	15	2	5	7

(array rep)

max_heap



TC

1	5	7	9	12	15

15/9/23

CW

If the max_heapify is applied at root node, then the max no of comparison is going to be happen. (max no. of comparisons)

→ 2 comparisons (depending upon which node the tree has max element)

∴ T(n) = 2 * $\log_2 n$ ← max TC is that (this same comparison will be done for rest of the nodes (full tree) [n ~~to~~ will be given in this case])

∴ T(n) = O($\log_2 n$)

max_heapify

$\log_2 n$ = height of the tree]

If we apply max_heapify at node 8 then the no of comparison < 2. So, the TC in this

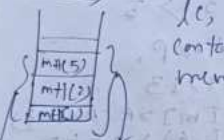
(1)

case will be $\leq O(\log_2 n)$.

So, the tc for max_heapify is $\leq O(\log_2 n)$

$\therefore \boxed{TC \leq O(\log_2 n)}$

SC → for the max heapify func



le, pc, longest this will contain constant amount of memory. (K)

the max no. of those call cells = height of the tree $= \log_2 n$

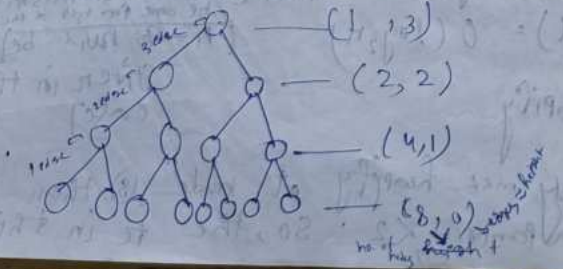
$\therefore S(n) = K * \log_2 n$

$\boxed{SC = O(\log_2 n)}$

(the max among 3 nodes ①, ②, ③ will call m+1 recursively, in that max element & it's 2 child nodes. Which one is greater will call m+1 recursively again, so it will go on for the whole tree).

TC of Creat max_heapify,

Height of a particular node = longest no. of edges from that particular node to the leaf node.



height(BT) = height of (root node)

$= 3$

1	5	7	9	12	15

a CBT of n (given) no of nodes

$\left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$ (h.0), $\left\lceil \frac{15}{2^{1+1}} \right\rceil = 4$ h.1

at a particular no. of nodes n the max no. of nodes at ~~level~~ height h is $= \boxed{\left\lceil \frac{n}{2^{h+1}} \right\rceil}$

(total how many nodes are available at a particular node)

If we apply m+1 at leaf node, the tc $O(0)$ as no comparison,

- - - -

$$= \frac{c \cdot n}{2} \sum_{h=0}^{\log_2 n} \frac{h}{2^h}$$

[∵ as c, n, 2 are const]

[∵ $2^{n+1} = 2^n \cdot 2^1$]

$$\frac{c \cdot n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$$\therefore \leq \left(\frac{c \cdot n}{2} \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^h}}_{2} \right)$$

→ 2 (this will be a constant)

$$\leq \left(\frac{c \cdot n}{\cancel{2}} \cdot \cancel{2} \right)$$

$$\leq c \cdot n$$

$$\boxed{\text{T.C. of Create_max_Heapify} \leq O(n)}$$

SC of create_max_Heapify

Here we only called mh, & mh will call the largest index to the root node. In this case we are going to use stack. the max no. of entries in that stack, height of tree, in that case, it will contain const ~~no.~~ amount of memory.

$\therefore S(n) = k + \log_2 n$

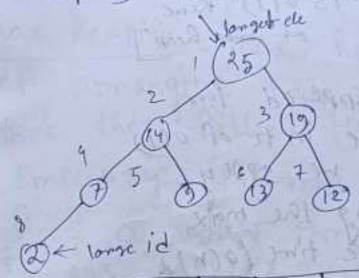
$$\boxed{SC \quad S(n) = O(\log_2 n)}$$

[as create_max_Heap only calls max_Heapify, so the SC will be exactly equal to MH's SC]

① Exact_max_Elem( )

↓

the purpose of that func is to extract the max element from the max heap & going to delete that max ele from that heap & rebuild the heap.



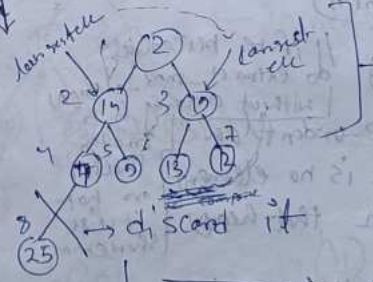
largest id = 8

25	14	19	7	9	13	12	2

→ [to make this, logical approach,
25 → r,
25/2 → rem = 0 → LC
3/2 → rem = 1 → RC
quotient = 1 (from value)
So on ...]

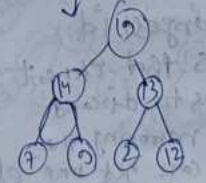
max = A[1]

exchange A[largest id] with A[~~max~~]

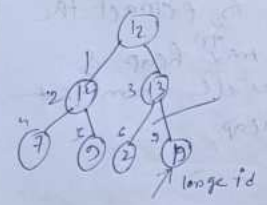


→ this way we can extract the max ele & rebuild the heap

↓ → apply max heapify func



max = A[1] = 10



[tc to find out the max ele is o(1) here at max heap]

min heap can be applied for finding out the min ele in tc of o(1). But ~~min heap can't~~ if we apply min heap for searching the max element that will take time f o(n) & same for max heap, if we try to find min ele in max-heap ~~the~~ it will take time. we will find out max ele from max heap & min ele from min heap.

Algo (array rep.)

```
Extract_Max_Elem(A)
{
    if (A.Heapsize < 1)        // if we repeatedly do extract_max_elem(A) without checking it is greater than 1 or not it will show error
        error "Heap underflow"
        (there is no element in that heap)
    max = A[1];
    A[1] = A[A.Heapsize];      // A.Heapsize = largest id
    A.Heapsize = A.Heapsize - 1;   // as that element has to discard
    max_Heapify(A,1);          // after removing the ele we can get max ele ~~heap~~ again from the remaining heap
    return max;
}
```

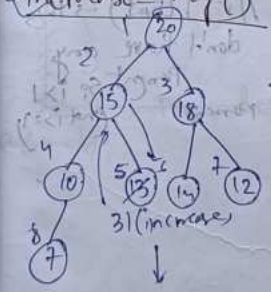
$\boxed{tc = O(\log_2 n)}$

$\boxed{sc = O(\log_2 n)}$ → its max-heapify is here, rest of the statements has to have const time complexity

Max heapify - max heapify is the process of arranging nodes in a b.t so that they follow the max-heap property. [In max heap every node has a value greater than or equal to its children. the root node contains the largest value element]

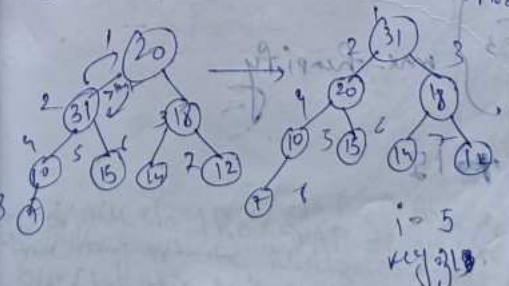
~~extract~~ max_heap() [operations]

~~increase_key()~~ 20/9/23



increase_key() ← (operation)

If we update the value, if the parent node is greater then it then exchange it. This process will go on ~~the~~ till the tree is following max heap property (till root node)



(traverse)

We will perform this increase key operation from the key to root, as the height of the tree is $\log_2 n$, so the tc of this operation,

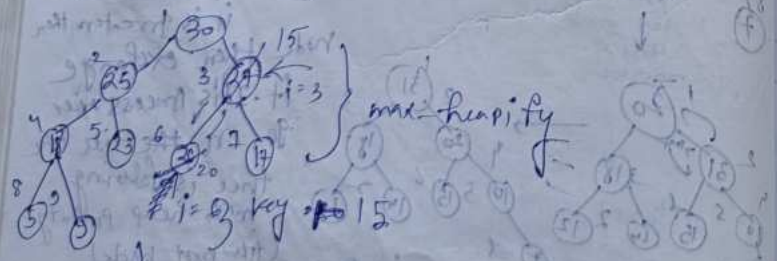
$$\boxed{TC = O(\log_2 n)}$$

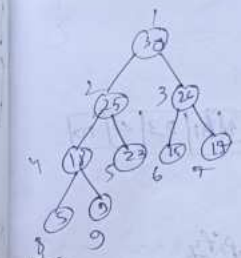
Algo **

increase-key(A, i, key) //i = ~~Parent of~~ ~~node~~ index of ~~key node~~ element which we want to increase

{ if (A[i] > key) {
  error "node value already greater than key";
  }
A[i] = key;
while (i > 1 && A[i/2] < A[i]) //i = 1 (root) → this don't have any parent, so i > 1 (not i = 1)
  {swap A[i/2] with A[i]
  i = i/2; //shifting to parent's parent
  }
}

decrease-key() **





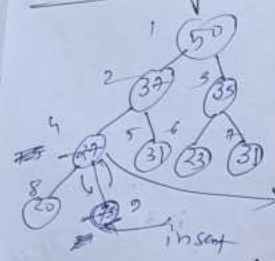
Algo

decrease-key(A, i, key)
{ if (A[i] < key) {
  error "node value already lesser than key";
  }
A[i] = key;
~~while (~~ max-heapify(A, i);
~~}~~
~~→ this root, here we can also do,~~

// we are heapifying ~~the~~ the subtree ~~that~~ contains the root/parent node. i.e. we aren't heapifying the full ~~tree~~ heap tree as it is not needed (ex - ~~max~~ (A, 3), there elems → 26, 17 can never be > than 30 as it was at 1st a max heap so it was following max heap property. we are doing all here, ~~as~~ if increase-key, this ~~tree may not be heapified correctly~~ will also give correct o/p (but this isn't needed)

we ~~will~~ will do the swap ~~if~~ o/p like

insert-key



50	37	35	24	31	23	21	20

apply max heapify here (generalized approach)

insert

Actual approach after knowing ~~insert~~

increase-key operation

if an element is greater than root it definitely greater than its

(4, 9, 75)

A[size] = -99999,
increase-key(A, size, key)
size++;

Algo

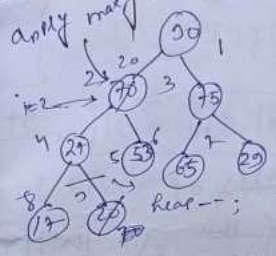
insert-key(A, size, key)
{ size = size + 1;
A[size] = -99999; // assigning a very small value to last id
increase-key(A, i, key);

size = n

}

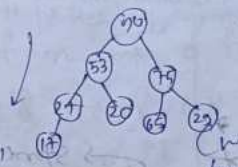
delete-key heapify

apply max heapify



Here i (index pos) must required from where we want to delete the node

heap--;



delete-key(A, i) // i = 2
max(A, 2) → 2(i)

$TC = O(\log_2 n)$

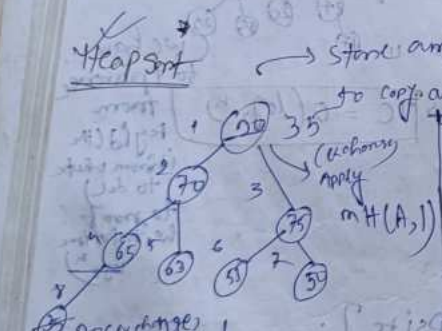
we have to traverse from key[i] (the id from where to del) to root heapify of tree $\log_2 n$

Algo

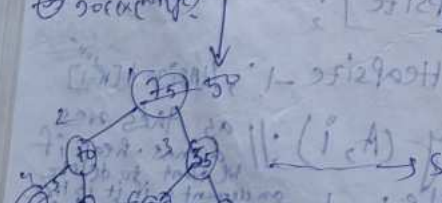
delete-key(A, i)
{ d_ele = A[i];
A[i] = A[A.Heapsize];
A.Heapsize = A.Heapsize - 1; // A[i] = A[n-1]
max_Heapify(A, i); // as this was a max-heap if we want to delete an element & if it is there at root after exchanging it with last node element it can never be greater than root as the tree was following max heap prop. so we apply max
return d_ele;
// Print d_ele has been deleted
}

Heapsort

→ store array ~~size~~ at another array (random) to copy array size



30	55	63	65	70	75	90



→ same thing will be repeated till the array contains a single element

Algo

```
HeapSort (A)
{
  Create_max_heap(A);
  for (A.size to 2){ (*)
      swap[A[size] with A[1]]
      A.size = A.size - 1;
  }   max.Heapify(A, 1);
}
```

// as 1st ~~el~~ extracted element (root node) will be the max, so for it no ~~for~~ heapify ~~is returned after swapping~~ is returned. It will be extracted only by swapping & discarding but after that to make the ~~full~~ heap max-heap we need mH till id=2 as // → not correct

tc, $\boxed{TC = O(n \log_2 n)}$

we have to atleast perform mH ~~a~~ for atleast n/2 ~~full p~~ times & the tc of mH is $\log_2 n$. so the $tc = O(n * \log_2 n)$

(*) we are taking for loop for A.size to 2 as after continuous swapping & heapifying when it will reach at its end then there will be 2 elements at id=1 & 2. when id=2; the for loop will run, it only ~~dis~~ swap & discard, mH will not work here as only 2 elts. ~~to~~ After that id=1 ele=p, so we just extract it or print it, no need to use for loop here.

~~So~~

matrix multiplication 20/2/23

$$A\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

2×2 $\quad$ 2×2

$$C\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

2×2

$C_{11} = a_{11} * b_{11} + a_{12} * b_{21}$

$C_{12} = a_{11} * b_{12} + a_{12} * b_{22}$

$C_{21} = a_{21} * b_{11} + a_{22} * b_{21}$

$C_{22} = a_{21} * b_{12} + a_{22} * b_{22}$

$A \times B$

$m \times n \quad n \times p$

$C = m \times p$

Algo matrix mul

```
for(i=0; i<m; i++)
{
    for(j=0; j<p; j++)
    {
        C[i][j]=0;
        for(k=0; k<n; k++)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

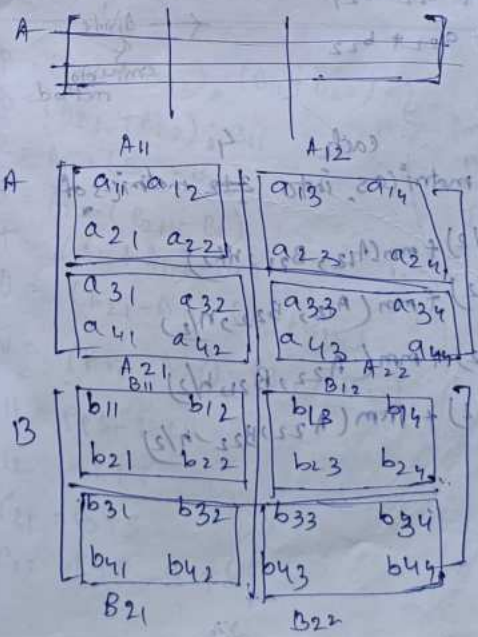
[AS B: $n \times p$]

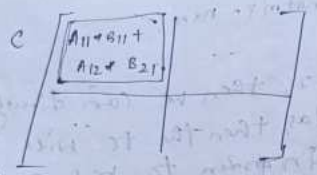
If it is a square matrix of $n \times n$; then TC = $O(n^3)$

If the matrix is 2×2, then we can directly use that 4 formulas, then the TC will be const = $O(1)$. In order to reduce the TC we will split a large matrix into 2×2 matrix & consider 2×2 a single matrix & will perform multiplication.

If, $A_{4\times 4}$ $\quad$ $B_{4\times 4}$

If, $A = 5\times 5$, then fill the other rows with 0, to make it power of 2×2 (dimention).



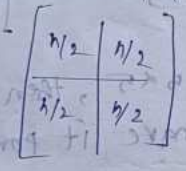


③ matrix mul for larger dimention algo

mm(A, B, n) → this is square matrix & power of 2 also

[If $A = [a_{ij}]$, $B = [b_{ij}]$ then $C = [a_{ij} + b_{ij}]$]

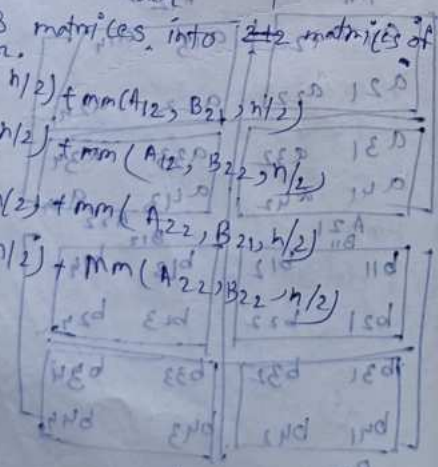
Dimension 4×4



{
if (n ≤ 2)
{
$c_{11} = a_{11} + b_{11} + a_{12} + b_{21}$
$c_{12} = a_{11} + b_{12} + a_{12} + b_{22}$
$c_{21} = a_{21} + b_{11} + a_{22} + b_{21}$
$c_{22} = a_{21} + b_{12} + a_{22} + b_{22}$
}

← Divide & conquer method

else
{
split A & B matrices into ~~2×2~~ 4 matrices of each n/2 dimention.
mm(A_{11}, B_{11}, n/2) + mm(A_{12}, B_{21}, n/2)
mm(A_{11}, B_{12}, n/2) + mm(A_{12}, B_{22}, n/2)
mm(A_{21}, B_{11}, n/2) + mm(A_{22}, B_{21}, n/2)
mm(A_{21}, B_{12}, n/2) + mm(A_{22}, B_{22}, n/2)
}
}



whenever the mul or nested mul will be lessen & addition (it takes lessen time) will be more then, tc will be reduced.

$$T(n) = \begin{cases} O(1) = 1; & \text{if } n \le 2 \\ 8T(n/2) + n^2, & \text{if } n > 2 \end{cases}$$

matrix add $= O(n^2)$ to perform mul of dim n/2 the time taken (n/2) & there are such 8 mul. so 8T(n/2)

$f(n) = n^2$

$n^{\log_b a} = n^{\log_2 8} = n^3$

$\therefore O(n^3)$

$\therefore tc = O(n^3)$ [no improvement after applying this divide & conquer method]

Strassen's matrix multiplication (tried to reduce this tc)

$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
$Q = (A_{21} + A_{22}) \cdot B_{11}$
$R = A_{11} \cdot (B_{12} - B_{22})$
$S = A_{22} \cdot (B_{21} - B_{11})$
$T = (A_{11} + A_{12}) \cdot B_{22}$
$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

formulas are not so imp for exam

Algo → this is square matrix & power of 2 also [if not same case not will be any 11 formulas]

mm(A,B,n)

{ if (n ≤ 2)

{

$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$

$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$

$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$

$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$

}

else

{

split A & B matrices each into 4 matrices of n/2 dim

$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) \cdot B_{11}$

$R = A_{11} \cdot (B_{12} - B_{22})$

$S = A_{22} \cdot (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) \cdot B_{22}$

$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

$C_{11} = P + S - T + U$

$C_{12} = R + T$

$C_{21} = Q + S$

} $C_{22} = P + R - Q + U$

$$T(n) = \begin{cases} 1 & \text{if } n \le 2 \\ 7T(n/2) + n^2, & \text{if } n > 2 \end{cases}$$

$f(n) = n^2$

$n^{\log_b a} = n^{\log_2 7} \cong n^{2.81}$

$\therefore T(C) = O(n^{2.81})$

TC of Strassen's matrix mul.

Ch — Greedy method — 5/10/23

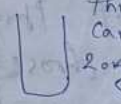
It is used to solve Optimization Problem

Optimization
- → maximization
- → minimization

Knapsack Problem

↓ meaning

Bag / container



this knapsack can carry maximum = 20 kg

Prob

I# (item)	I_1	I_2	I_3	I_4	I_5	I_6	m = 20 (max)
wt (weight)	5	3	4	6	6	4	n = 6 (no. of items)
Profit	15	12	14	16	17	15	

Approaches — decreasing → increasing order

① I_2, I_6, I_3, I_1, I_5

wt	3	4	4	5	4 (5)	
Profit	12	15	14	15	16 (17)	15 = 10.66

(by choosing this we are going to gain more Profit)

total Profit = 12 + 15 + 14 + 15 + 11 + 33 = 66.66

(2) (get max Profit) (In descending order of Profit) — max Profit

	I_5	I_7	I_4	I_6
Wt	6	7	5	2/4
Profit	17	16	15	7.5

= 55.5

(3)

Profit/Wt 3 4 3.5 2.83 3.75

(In descending order of max Profit/Wt) (max amount of Profit per unit)

	I_2	I_5	I_3	I_1	I_5
Wt	3	4	4	5	4/6
Profit	12	15	14	15	4/6 × 17

= 64.33

this above way is giving maximum profit. In greedy method we have to chose such a way that will give us the maximum profit.

[we which we think about the diff ways to transport products (divisable product) & to chose the way between them that give us the max profit in knapsack problem.]

[Knapsack problem – The knapsack is a optimization challange where items with different weights & values must be selected to maximize the total value, while staying within a given weight constraint.]

Huffman Coding

ASCII
↓ 8 bits (char)
15×8 = 120 bits
↳ original ASCII binary bits

A B B C A D C B A E D B A B C

we are trying to reduce the amount of bits per char. so that during transferring it the no. of bits get reduced.

	frequency count
A	4
B	5
C	3
D	2
E	1
	15

Huffman coding is a compression algo that assigns variable based on their length codes to for it. ill chars based on their frequencies with shorter codes for more frequent chars.

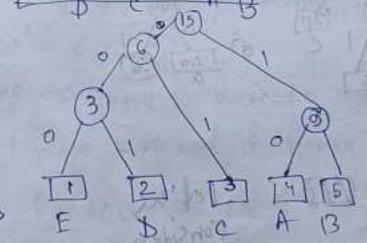
Optimal merging Procedure / algo

we are going to merge frequencies of 2 lowest frequencies & sum them up.

A B C D E

(descending order according to their fc)

E D C A B



Huffman trees

[0 to the left edge
1 to the right edge]

mapping binary table

E – 000
D – 001
C – 01
A – 10
B – 11

[the path from root]

$\sum_{i=1}^{5} d_i + f_i$ (total binary bits of this path)

= 2×4 + 2+5 + 2+3 + 3+2 + 3+1

= 8 + 10 + 6 + 6 + 3

= 33 bits (this is much much lesser than the original no. of bits that we will send it we start to this algo)

8×5 + 12 (total no. of binary bits in this mapping table)

∴ 52 bits is need to send

[d_i: no. of total binary mapping bits of each ith char in binary table]

we will not transfer the original ~~AS~~ ASCII binary bits, we will transfer this -

10 11 — see this

A BB CA D C BA F B B A B C

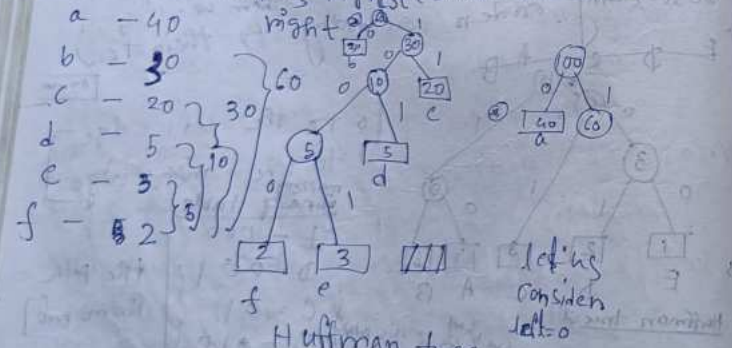
the mapping table also needs to be send to distinguish which binary value is for which character

- The compression is going to be much much higher if the total no. of chars increase as 52 will remain same.

- **Huffman coding** 6/10/23

For data compression technique we use this. [Always put left lowest freq at left & highest (between that 2) at right]

a – 40
b – 30
c – 20
d – 5
e – 3
f – 2



Huffman tree

a → 0
b → 10
c → 111
d → 1101
e → 11001
f → 11000

Prefix Code notation – Rep. of one particular character will not be the prefix of any other character's representation.

Eg – 0, 10, 111, 1101, 11001, 11000

Huffman coding always follow the prefix code notation. (this is used (huffman coding's prefix code notation following technique) is used to remove ambiguities, but they aren't fully removed.)

$1 \times 40 + 2 \times 30 + 3 \times 20 + \cancel{3 \times 20}$ $4 \times 5 + 5 \times 3 + 5 \times 2$
$= 205$

As ASCII char

So, we have to pass the table as well,

$68 + 20$ (no. of total chars) → alongwith i-1's mapping

original = 68 bits

total we have to send = 68 + 205 = 273 bits

Actually without huffman coding we have to pass

$= 8 \times 100$
$= 800$ bits

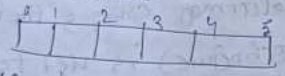
- this is call in greedy method as we are using a greedy approach & finding the lowest frequency of occuring term. (that can be possible)

Job sequencing with deadlines

→ It is a scheduling problem where tasks with associated deadlines & profits must be arranged to maximize total profit. The goal is to complete the jobs with the deadlines while optimizing the overall gain.

	J_1	J_2	J_3	J_4	J_5	J_6
Deadline D:	5	3	3	2	4	2
Profit P:	200	180	190	300	120	100

Every job ~~needs~~ require 1 unit of time for completion.



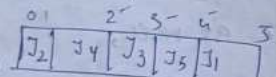
as the deadline is 5 (only 5 task can be performed so in 1 hour, 1 job will be performed)

Sort the job according to their profit's descending order

$J_4, J_1, J_3, J_2, J_5, J_6$

300 200 190 180 120 x (Reject this job)

We will place the jobs as later as possible following the deadline.



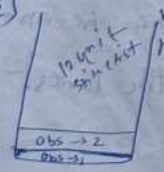
optimal job sequencing for (greedy approach)

∴ total profit = 990

Fractional Knapsack algo

10/10/23

[eg-1 & one]



m=15
n=7

Obj	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
Wts	2	3	5	7	1	4	1
P/W	5	5/3	3	1	6	4.5	3

```
Greedy knapsack()
{
  for i ← 1 to n
      Compute Pi/Wi
  Sort object by Pi/Wi in descending order
  for i ← 1 to n from the sorted obj list
      if(m > 0 && Wi ≤ m)
          m = m - Wi
          P = P + Pi
      else
          break;
  if (m > 0)
      P = P + Pi * (m/Wi)
}
```

m>0 → if the container is not yet fully filled

& wt of the current obj is greater than the currently available obj in container

space will reduce by the wt of the current obj

[Pi = Profit of that element]

Graph

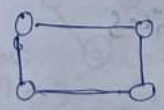
Spanning tree is a sub graph of main graph including all the V & of course making a tree.

Spanning tree

12/10/23

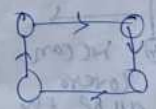
that is

A graph is a ds, consisting of 2 types of eles, set of vertices (V) & set of edges (E)



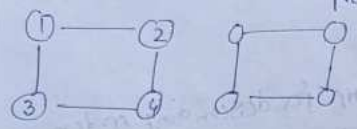
G = (V, E)

Directed graph (digraph) – edges have direction



Undirected graph – no direction

Another classification (we assign values over edges) → weighted, unweighted (there are no values over edges)

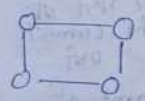


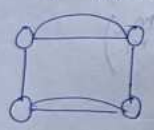
no weights

Another way of classify the graphs

simple → only one edge between a pair of vertices

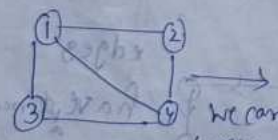
multigraph → more than one edges in between a pair of vertices





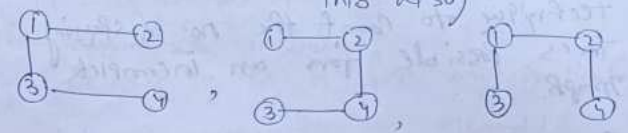
Spanning tree –

- Spanning tree is a subgraph of ~~graph where there is one where connection between the Vs and~~ we have the min no. of edges to connect all of the ~~edges~~ vertices



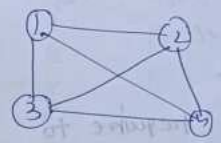
we can covers all of the vertices by only 3 edges.

similarly, (we can covers all vertices by this also)



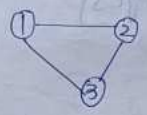
max no. of spanning trees possible for a complete graph

Complete graph →

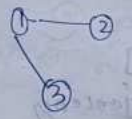


for this max no. of spanning trees $= 4^{4-2} = 4^2$

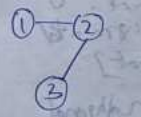
$\boxed{n^{n-2}}$ [n = no. of nodes]

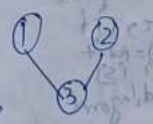


~~Spanning tree~~ → subgraph



→ Spanning tree





Here, $n = 3$

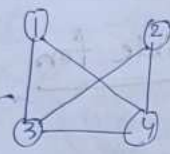
$\therefore n^{n-2} = 3^{3-2} = 3$

max/total no. of spanning tree possible

- Spanning tree is only applicable for complete graph.

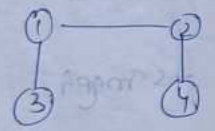
~~Kirsh~~ Kirchoff law → for Incomplete graph

technique to count the no. of spanning trees possible for an incomplete graph.



graph → n

no. of edges require to draw a spanning tree of a given graph of no. of nodes n → $\boxed{e=n-1}$ [e = edges]



Adjacency matrix

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

[0 or ∞]

[A = adjacency mat.
A' = Augmented mat]

[non diagonal ele = (i,j) = put it as it is]

[for diagonal ele = put the degree of that node]

[non diagonal ele = (i,j) = put (-1) decrease]

$$A' = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 2_{deg} & 0 & -1 & -1 \\ 2 & 0 & 2_{deg} & -1 & -1 \\ 3 & -1 & -1 & 3_{deg} & -1 \\ 4 & -1 & -1 & -1 & 3_{deg} \end{array}$$

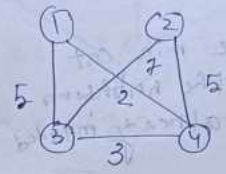
no of ST possible,

[Here in A' (Augmented mat) if sum of any row/any col = 0, then our calculation is wrong]

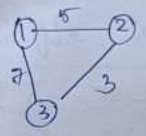
|ST| = co-factor of any element of the augmented matrix (we have to do it for any particular element)

total no. of spanning tree possible

minimum cost spanning tree

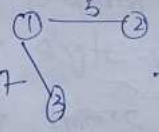


(we will have the minimum cost if we sum up this)



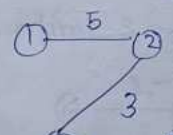
→ graph

from spanning trees if we sum up the weights (costs), then the total costs are minimum



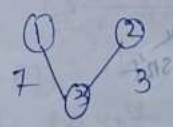
total cost = 3+7 = 12

→ ST ①



total cost = 5+3 = 8

→ ST ②



total cost = 7+3 = 10

→ ST ③

this is minimum cost spanning tree

element)

minimum cost spanning tree

Now delete any 1 row & any 1 col from A'

$$A' = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 0 & -1 & -1 \\ 2 & 0 & 2 & -1 & -1 \\ 3 & -1 & -1 & 3 & -1 \\ 4 & -1 & -1 & -1 & 3 \end{array}$$

we are removing / deleting 1st row & 1st col.

Now, $A' = \begin{array}{c|ccc} & 2 & 3 & 4 \\ \hline 2 & 2 & -1 & -1 \\ 3 & -1 & 3 & -1 \\ 4 & -1 & -1 & 3 \end{array}$

$\Rightarrow |A'|$ [determinent of A'] =

$$\begin{vmatrix} 2 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{vmatrix}$$

$= 2(9-1) + 1(-3-1) - 1(1+3)$

$= 2 \times 8 - 4 - 4$

$= 16 - 8$

$= 8$

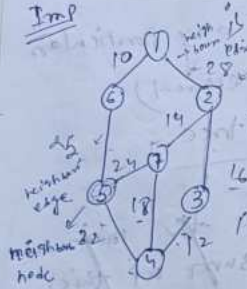
$\therefore$ max / total no. of spanning tree possible = 8

① Prim's Algo ② Krouskal Algo

It is used to find the minimum spanning tree of a connected undirected graph, minimizing the total edge weights while spanning all vertices.

① Prim's Algo. Procedure –

Imp

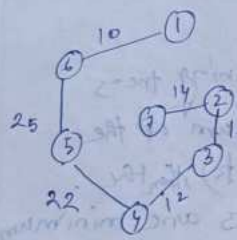


i) we have to find the min cost edge from the entire graph

ii) Include that in your path

iii) Find out the min cost edge from the neighbours edges of already included edge.

--- so on



28
24
18
~~X~~
14

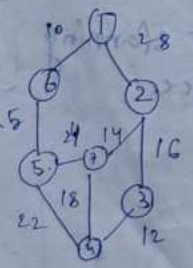
→ All the nodes have been included

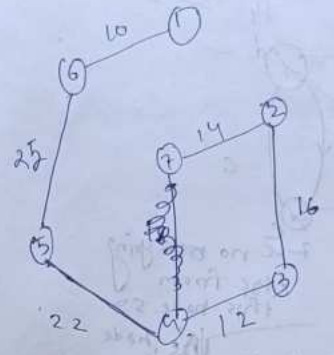
n = 7

e.n-1 = 6 (expected no. of edges in MST that)

∴ MST = 99

② Krouskal Algo → It finds the minimum spanning tree in a connected undirected graph by iteratively adding the shortest edge that doesn't form a cycle.





MST = 99

[18 will not be added as it is forming cycle]

[24 ----]

[e = n-1 = 7-1 = 6]

Imp

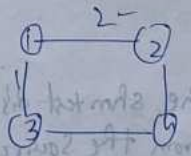
13/10/23

Single Source shortest distance Algorithm

(Dijkastra Algo) or minimization algo

to find out the shortest path from a single node.

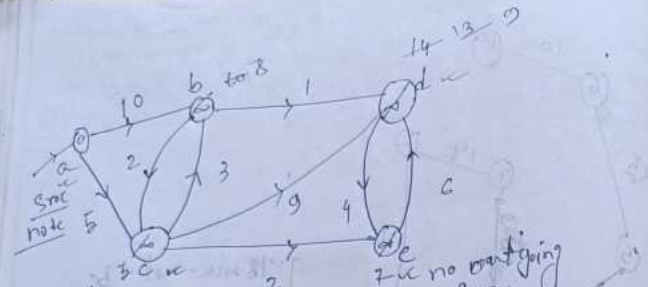
The Prim's & Krouskal's algo doesn't work sometimes – cause below eg



(D.A)

- It doesn't work for negative weights.

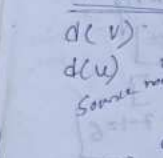
So shortest path = 3 unit but 2 is also shortest path, ∴ Prime's or Krouskal's algo doesn't work efficiently here. So, Dijkastra Algo is needed



(destination)

no outgoing edge from this node so this node is also remarked as visited.

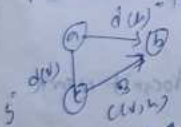
relaxing an edge



if $d(u) > d(v) + c(v,u)$

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

↳ relaxing an edge technique



[after a node (node is the shortest (inclusive node)]

$d(u) > d(v) + c(v,u)$

$\therefore d(u) = 8$

$d(e) = \infty > d(u) + c(u,e)$

$\to 7$

$d(d) > d(e) + d(e,d)$

$d(d) = 13$

$d(d) = 13 > 9$

$d(b) + c(b,d)$

$8 + 1 = 9$

$\therefore d(d) = 9$

the shortest distance from the source node (a) to (b) is = 8

$d(a,b) = 8$

$d(a,c) = 5$

$d(a,d) = 9$

$d(a,e) = 7$

Dijkstra Algo

Dijkstra (G, w, s) → source vertex

{ || w → weight ~~weight~~ matrix

Initialize the singleSource (G, s) || — O(1) → O(V) [V]

d(s,s) = 0,

S = φ || S with some all nodes we have been visited d(s,v) = ∞

Q = G.Nodes || Build heap min O(n)

while (Q ≠ φ) → O(V) [n = no of nodes]

u = Extract_min(Q) → log V (log₂ n)

S = S ∪ {u} — O(1) || S = φ ∪ {a} = {a}

for each ~~return~~ vertex v ∈ G.Adjacent(u) [if node has been already visited]

{ Relax(v, u, w); → log n* [E = no of edges]

}

Relax (v, u, w) → (nothing but decrease key operation) E*

{

if d(u) > d(v) + c(v, u) [the tc for building a min heap O(n)]

d(u) = d(v) + c(v, u)

}

T.C — $\boxed{O(E * \log n)}$ (this has to be n*(n-1)* log₂ n) but it will be (E* log₂ n) no. of nodes

[O(1) + O(n) + (n* log₂ n) + O(E * log₂ n)] (max strength out term)

[Relaxing an edge is nothing but updating a particular node like decrease key]

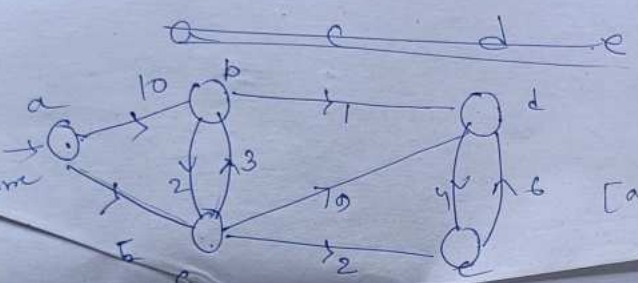
[for E no of edges it will be executed]

Shortest distance from Source (a) to b

		shortest path distance	s. path
~~d(a~~ Ans	a → b	8	a c b
	a → c	5	a c
	a → d	~~13~~ 9	a c b d
	a → e	7	a c e

Initial point (Source)

final point (destination)



[a, d not consecutive ∞]

c → b = 3

	a	c	d	e	b
Initial	$\boxed{0}$*	∞	∞	∞	∞
selected a = 0	0	min(~~5~~, ∞) 5+0 = $\boxed{5}$*	∞	∞	min(∞, 10+0) = 10
selected c = 5	0	5	min(∞, 9+5) = 13	min(∞, 2+5) = $\boxed{7}$*	min(10, 8+5) = 8
selected e = 7	0	5	min(13, 6+7) = 13	7	min (8, ∞) = $\boxed{8}$*
b					

[You can put the destination node/vertex at last if it is the last vertex already (like here d is last not b). But if the destination v. is not last vertex then put it in it's alphabetical order otherwise the back track will be in loop]

→ as it is as no edge (b selected from have so been back track starts from here)

[Executed]

b to 8

an edge

d(u) → destination node

c(v, u)

) > d(v) + c(v, u)

) = d(v) + c(v, u)

an edge technique

[after a node c node is the shortest consecutive node]

c(e, e)

the shortest distance from the source node (a) to (b is = 8

d(a, b) = 8

d(a, c) = 5

d(a, d) = 9

d(a, e) = 7

d(a) = 9

no outgoing edge from this node so this node also remains unvisited.

PAGE NO.
DATE : / /20

the current of selected v

	a	b	c	d	e
	0	∞	∞	∞	∞
a=0	0	10	$\boxed{5}$ +	∞	∞
c=5	0	8	5	13	$\boxed{7}$ +
e=7	0	$\boxed{8}$ +	5	13	7
b=8	0	8	5	$\boxed{9}$ +	7
d=9	0	8	5	9	7

a-a = 0
a-b = 8
a-c = 5
a-d = 9
a-e = 7

path

a-b
a←c←b
a-c
a←c
a-d
a←c←b←d
a-e
a←c←e

31/10/23

Prim's algo

cost → n×n matrix

Prim (E, cost, n, t)
// mst is computed & stored in array t [1:n-1, 1:2]
{
Let (k, l) be an edge of min cost in E. // E = set of edges
min cost = cost[k, l];
t[1,1] = k, t[1,2] = l;
for i := 1 to n do
{ if (cost[i, l] < cost[i, k])
{ then near[i] = l;
else near[i] = k;
}
near[k] = near[l] = 0
for i = 2 to n-1 do — O(n)
{ Let j be an index such that near[j] ≠ 0 (→ n×n) and cost[j, near[j]] is minimum
t[i, 1] = j, t[i, 2] = near[j]
mincost = mincost + cost[j, near[j]]
near[j] = 0; // to denote that j is already visited
}
}

t[i,2], t[i,1] → this is used to reflect the fact that 1, 6 included are in the soln.

// min cost = 10

1 —10— 6

t[1,1] = 1
t[1,2] = 6

cost[1,1]
cost[1,6]
cost[2,1]
cost[2,6]
cost[7,1]
cost[7,6]
neighbour nodes

// 1 ~~node~~ edge is already selected
So n-2 more nodes are left.
n - 2 + 1 = n-1 (no. of nodes)
~~near~~ near[5] = 0 = 6

t[i,1], near[i]
cost[5, 6]

	1	2	3	4	5	6	7
	1	17	1	6	8	1	

k = 1
cost[1,6] < cost[1,1]
near[1] = 1
i = 2
cost[2,6] < cost[2,1]
near[1] = 1
cost[3,6] < cost[3,1]
near[1] = 1

for k = 1 to n do — n×n → there might be any better path / distance
if (near[k] ≠ 0) & (cost[k, near[k]] > cost[k, j]))
near[k] = j;
}
return mincost;
}

c[1,6] = 10 ✗
c[1,1] = 0
near[1] = 1
c[2,1] = 28 < c[2,6] = ∞

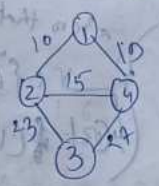
BOOK → Command (this algo)

Cla 2/11/23

Named // Prim's algo using minheap (another one)

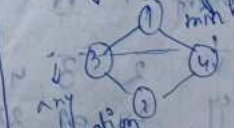
Prim (G, cost, root) key
{
for each node u ∈ G nodes
{ u.key = ∞
u.Parent = NIL } n times (const) O(n) is also const
}
~~1.key = ∞~~
~~1.Parent = NIL~~
root.key = 0 — O(1)
Q = G.nodes // Build min heap
while (Q ≠ φ) — n
{
u = Extract-min (Q) — O(log n)
for each node 'v' adjacent to 'u'
{ — n (for this case consider it will execute from n. of times)

the best tc of Prim's algo is by using fibonacci heap
tc = O(E + V log n)



1.key = ∞
1.Parent = NIL

root
1 2 3 4
key 0 ∞ ∞ ∞
Parent NIL NIL NIL
(Building the min heap)



any combination as 1, 2, 4 = ∞

2 will be different if this is a complete graph ←

if $v \in Q$ and $cost(u, v) < v.key$

{ v.Parent = u

v.key = w(u, v) → [nothing but, decrease key operation]

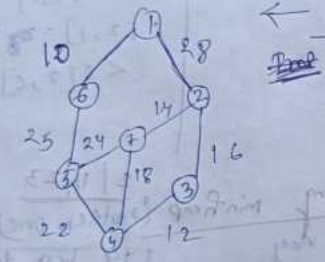
→ $O(\log n)$

}

}

}

← Explanation



	1	2	3	4	5	6	7
key	0	28	∞	∞	∞	10	∞
Parent	N	1	N	N	N	1	N

after that ① will be added

$V = \{6, 2\}$

cost (1, 6) < v.key

THIS is how the execution will go on.

	2	3	4	5	6	7
key	28	∞	∞	25	10	∞
Parent	1	N	N	6	1	N

	2	3	4	5	7
key	28	∞	22	25	∞ 24
Parent	1	N	5	6	N

	2	3	4	7
key	28	∞	22	24
Parent	1	N	5	N

4 = 4

$v \in u.adj$

$u \in \{5, 7\}$

	2	3	4	7
key	28	12	22	18
Parent	1	4	5	4

	2	3	7
key	16 28	12	18
Parent	3	4	4

	2	7
key	16	14
Parent	3	2

	7
key	14
Parent	2

// No need

the sequence of consideration – 1, 6, 5, 4, 3, 2, 7

min cost. $10 + 25 + 22 + 12 + 16 + 14 = 99$

TC → Say, there are n no. of nodes (vertices)

$O(n) + O(1) + O(n \log n) + O(E \log n)$ [for decrease key op → $O(n^2 \log n)$ but the TC will be here $O(E \log n)$, E = no. of edges]

$\therefore \boxed{TC = O(E \log n)}$ [$\because n \le E$]

Prim's algo (without min heap)

[If there is any node that is near to both i, k nodes then we have to choose one on the basis of min edge cost

Like eg, 1 —10— 2 —25— 3
i=1, k=3 — this chosen to both i & k but 1-5 and 5-6 ... ∴ choose 6 as near]

	1	2	3	4	5	6	7
near	X	1	1	1	6	6	1
cost	0	28	∞	22	25	0	∞

	t	
	1	2
1	(1, 6)	
2	(5, 6)	
3	(4, 5)	
4	(3, 4)	
5	(2, 3)	

(next step)

i = 2

~~t[2, 1] =~~

mincost = 10 + 25 + ... = 99

	1	2	3	4	5	6	7
near	X	1	1	1	6	6	1
				0			

next step (...)

K = 2

cost[2, 1] ≯ cost[2, 5] → no change

K = 3

cost[3, 1] ≯ cost[3, 5] → no change

K = 4

cost[4, 1] > cost[4, 5]

	1	2	3	4	5	6	7
	X	1	1	5	6	6	1
	0	28	∞	22	25	0	∞

K = 7

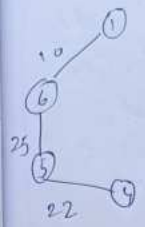
cost[7, 1] ≯ cost[7, 5]

	1	2	3	4	5	6	7
	X	1	1	5	6	6	5
	0	28	∞	22	25	0	21

i = 3

t[3, 1] = 4 t[3, 2] = 5

∴ i = 4

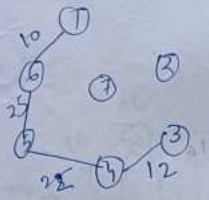


	1	2	3	4	5	6	7
	X	1	1	5	6	6	5

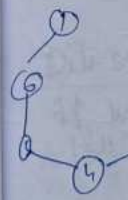
	1	2	3	4	5	6	7
near	X	1	4	5	6	6	54
	0	28	12	22	25	0	18

i = 4 t[4, 1] = 3 t[4, 2] = 4

	1	2	3	4	5	6	7
	X	1	X	5	6	6	4
	0	0	0	0	0	0	



cost[2, 1], cost[2, 3]



	1	2	3	4	5	6	7
	X	X	X	5	6	6	4
	0	0	0	0	0	0	

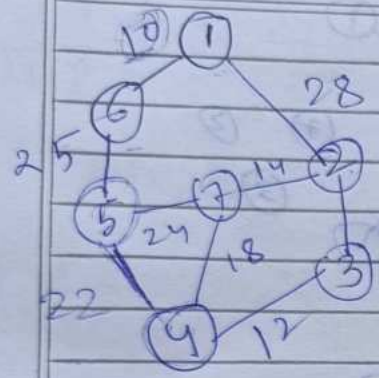
i = 2

cost[7, 4], cost[7, 2]

	1	2	3	4	5	6	7
	X	X	X	5	6	6	X2

Whenever there is no ~~note~~ edge b/w 2 nodes the cost will be ∞.

PAGE NO.
DATE: / /20



t

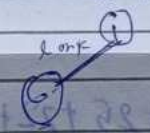
1.	(1, 6)
2.	(5, 6)
3.	(4, 5)
4.	(3, 4)
5.	(2, 3)
6.	(7, 2)

x not ~~c = choosed~~ C = Choosed

1)

	1	2	3	4	5	6	7
near	0	1	6	6	6	0	6
cost	C	28	∞	∞	25	C	∞

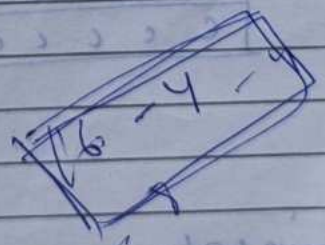
no edge b/w



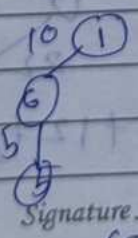
min cost = 10

2)

1	2	3	4	5	6	7
0	1	6	5	0	0	5
C	28	∞	29	C	C	24



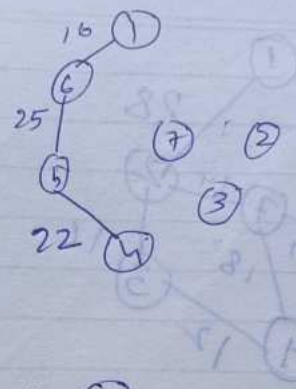
min cost = 10 + 25



Signature

3)

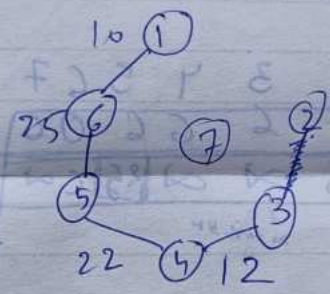
1	2	3	4	5	6	7
0	1	4	0	0	0	4
C	28	12	C	C	C	18



min cost = 10+25+22

4)

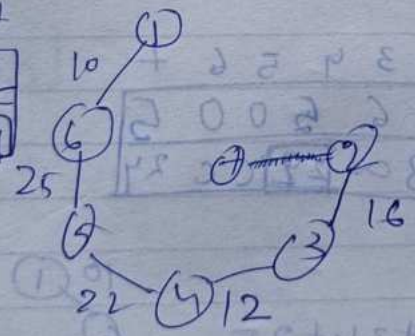
1	2	3	4	5	6	7
0	3	0	0	0	0	4
C	16	C	C	C	C	18



min cost = 10+25+22+12

5.

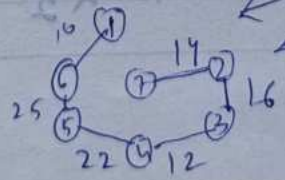
1	2	3	4	5	6	7
0	0	0	0	0	0	2
C	C	C	C	C	C	14



min cost = 10+25+22+12+16

6.

1	2	3	4	5	6	7
0	0	0	0	0	0	0
C	C	C	C	C	C	C



← MST
← min cost = 99

min cost = 10+25+22+12+16+14

tc

$$\boxed{\begin{aligned} tc &= n + n + n + n \times n \\ &= O(n^2) \end{aligned}}$$

7/11/23

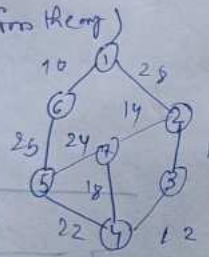
CW
no need

Prim's algo using array

selectedEdges = ϕ
visited = {1}
while (visited ≠ V)
  let (u, v) be the lowest cost edge such time
  u ∈ visited & v ∈ V − visited
  selectedEdges = selectedEdges ∪ {u, v}
  visited = visited ∪ {v}

Python (code) — (no need for the array)

N = 7
INFINITY = 99999
G = [[0, 28, 0, 0, 0, 10, 0],
  [28, 0, 16, 0, 0, 0, 14],
  [0, 16, 0, 12, 0, 0, 0],
  [0, 0, 12, 0, 22, 0, 18],
  [0, 0, 0, 22, 0, 25, 24],
  [10, 0, 0, 0, 25, 0, 0],
  [0, 14, 0, 18, 24, 0, 0]]

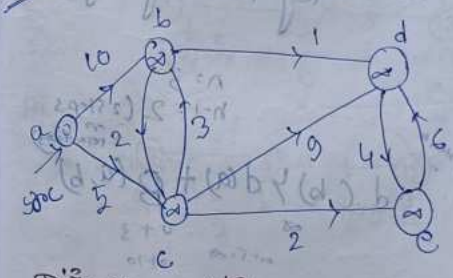


[edges = n − 1 if v = n]

visited = [0, 0, 0, 0, 0, 0, 0] // no node has been visited yet
no_edges = 0
visited[0] = True
mincost = 0
while (no_edges < N−1):
  min = INFINITY
  u = 0
  v = 0
  for i in range(N):
    if visited[i]:
      for j in range(N):
        if (not visited[j] and G[i][j]):
          if min > G[i][j]:
            min = G[i][j]
            u = i
            v = j
print(str(u+1) + " → " + str(v+1) + ": " + str(G[u][v]))
visited[v] = True
mincost = mincost + G[u][v]
no_edges = no_edges + 1
print("mst cost = " + mincost)

9/11/23

CW



Dijkstra Algo

If we use min heap concept here too, the tc will be decreased.

Bellman ford algo (for negative weight edge)

It will decleare that we have a neg weight edge, if there is no neg edge then it will give us the correct ans. But the dijkstra algo doesn't decleare that & give us ans. that may be wrong.

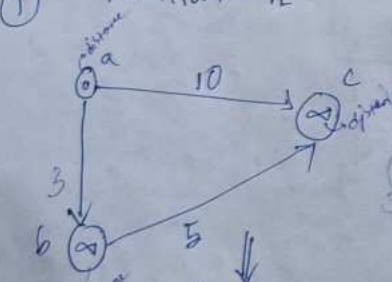
(n-1) + (1) → To check whether there is any neg edge or not

(n-1) → to find out the shortest path by performing relaxing edge operations

(1) → we are performing relaxing edge operations again.

So total times of performing relaxing edge operation = n

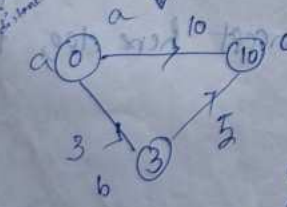
(1)



n = 3
n-1 = 2 (2 steps or 2 relaxations)

d(b) > d(a) + β(a, b)
∞ > 0 + 3
∞ > 3

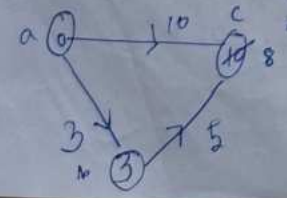
II -



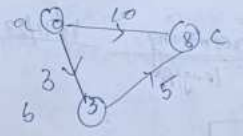
[now look at this graph (poor graph) & calculate the next step]

d(c) = 10
d(b) = 3
w(b, c) = 5 (forming edge)
8 < 10

II -



III -



(that extra relaxation)

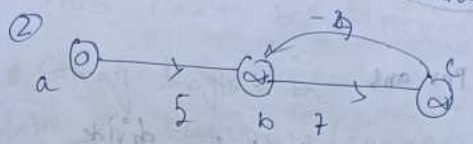
there is no negative weight in that graph, if had then we will get the result.

reduced

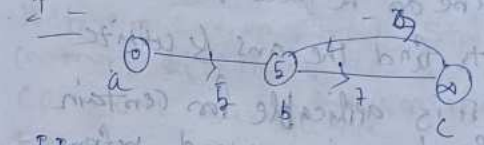
c(u, v) → dijkstra only takes give here
→ bellman-ford takes both give & give there

n = 3
n-1 = 2 (relaxation)
1 = extra ~~relaxation~~ relaxation

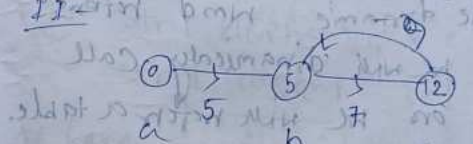
(2)



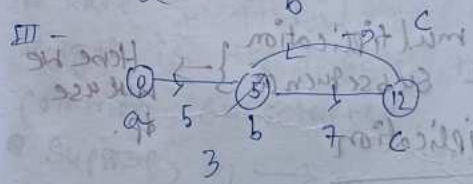
I -



II -



III -



[Following the arrow]

[But if you don't follow arrow then also it is or as if we get better result then only the node value will be changed]

$TC = n * E * O(1) = O(n * E)$

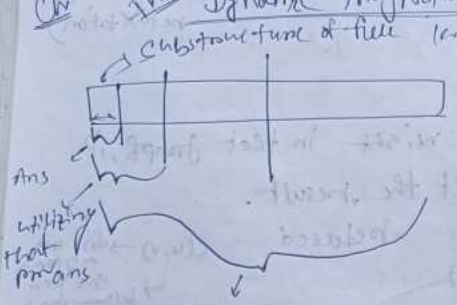
n → performing relaxation on every edges

E → no. of nodes, no of edges

$\boxed{TC = O(n * E)}$

dijkstra TC < Bellman ford TC

CW Time Dynamic Programming (DP) 10/11/23



optimal substructure or we need to divide
we need
it into equal parts, find the ans & utilize
that ans in dp. It is applicable for certain
set of problems. the dynamic word refers
to that whether we will dinamicaly call
the function once or we will refer a table.

1. matrix chain multiplication
2. longest common subsequence } → Here we will use dp.

matrix chain multiplication

$B = A_1 * A_2 * A_3 * \ldots A_j$

longest common subsequence

substring → 1 2 3 4 5 6 7 8
elephant
{e} {el} {ele}
1 1 2

chars should be present at the consecutive sequence
position of the original string.

subsequence → Some of chars of original string
subset of

But the chars may not be present at the consecutive location of the original string.

eg- {e e h n} {e l p a}
1 3 5 7 1 2 4 6

But the order of the chars must be in ascending orders. i.e, e.g. {l e l e}
2 1 7 3
this can't be happen.

- String length = m

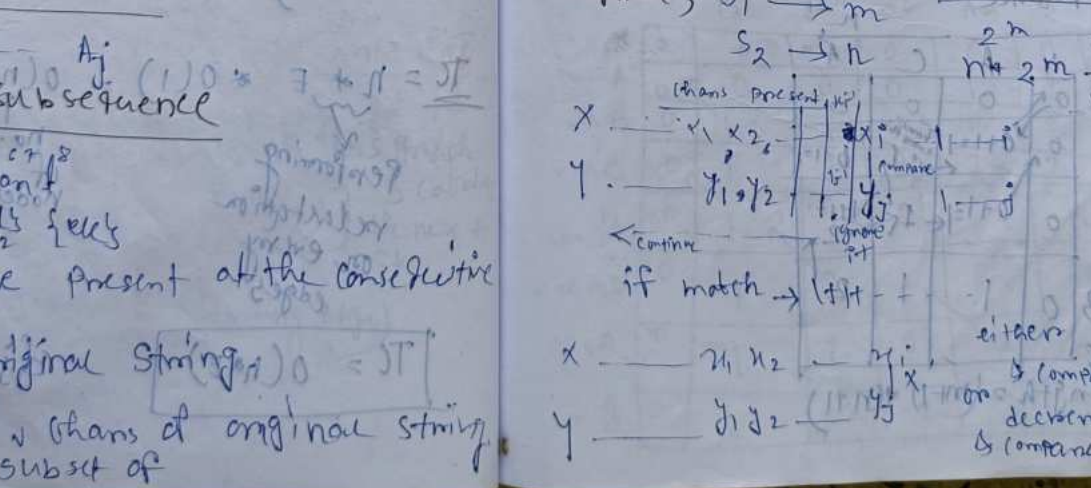
total No. of substring / subsequence = $\boxed{2^m}$

Whenever 2 or more strings, if we have common set of chars then it will be called common subsequence in both or more strin's subsequence

Eg- str1 elephant = {e l e a}
str2 elegant = {e l e a} } → common subsequence.

we have to findout the subsequence which length is the longest, this is called longest common subsequence.

- Suppose, $S_1 \rightarrow m$ needed subsequence
$S_2 \rightarrow n$ 2^m
$n + 2m$ → finding the subsequence that is present on other string.



& take max between 2 cases.

If two subsequence (taken from 2 string) don't have atleast 1 common char then, the length of ~~com~~ longest common subsequence = 0.

Eg. ~~AB C~~ X = A B C A i=3 j=4 i=j=4
Y = A D B A

(one subsequence length can be greater than another subsequence)

length of L.C.S = 1+1+1 = 3

{ABC, AD} → {ABC, A} {AB, AD}

{AB, AD} → {A, AD} {A, A}

$$C[i,j] = \begin{cases} 0 ; & i=0 \text{ or } j=0 \\ 1+C[i-1,j-1] ; & i=j \\ \max(C[i-1,j], C[i,j-1]) \end{cases}$$

[if one string is empty then with nothing we can compare so LCS=0]

How to find out longest common subsequence

X = A A B C
Y = A C A B

		Y	A	C	A
X	0	0	0	0	0
A	0	0+1=1	1	1	1
A	0	0+1=1	1	1+1=2	2
B	0	1	1	2	2

length = (m+1) * (n+1)

max value in that matrix = 2
length of the lcs = 2
LCS string sequence / string = |AA| (one way) or |AA| (another way)

Ch
Imp Longest common subsequence 21/11/23

Str1 = ABCAB (1 2 3 4 5)
Substring, Substr1 = {A, B, AB, ABC}
Subsequence, Subseq1 = ACA (1 3 4)
must be in ascending order

X = ACBA (m)
Y = BCAD (n)

DP $\xrightarrow{TC}$ O(m*n)

LCS → X = {A B C B D A B} l(x) = 7
Y = {B D C A B A} l(y) = 6
(7+1) * (6+1) = 8*7

		Y	B	D	C	A	B	A
X	0	0	0	0	0	0	0	
A	0	0	0	0	1	1	1	
B	0	1	1	1	1	2	2	
C	0	1	1	2	2	2	2	
B	0	1	1	2	2	3	3	
D	0	1	2	2	2	3	3	
A	0	1	2	2	3	3	4	
B	0	1	2	2	3	4	4	

← 1st len of 1st str = 0 = length of lcm = 0

length of the lcm = 4

'BCBA' → one ~~lcs~~ lcs (the path)
'BCAB' → Another "
'BDAB' → " "
total no. of lcs = |LCS| = 3
length of lcs = l(LCS) = 4
LCS = {BCBA, BCAB, BDAB}

from X, 2 3 4 6
B C B A
Y, 1 2 3 5
B C B A (verify)

Algo

```
LCS (x, y)
{
  m = length(x)
  n = length(y)
  Let c[0..m, 0..n] be a new table
  for i = 0 to m
      c[i,0] = 0
  for j = 0 to n
      c[0,j] = 0
  for i = 1 to m
      for j = 1 to n
          if x_i = y_j
              c[i,j] = c[i-1, j-1] + 1
          else
              c[i,j] = max(c[i, j-1], c[i-1, j])
```

Diagonal ele


```
  return c[m,n]
}
```

Algo for Printing the sub sequences of lcs

```
int id = c[m][n];
char lcs[id+1]
lcs[id+1] = '\0'
i = m, j = n;
while(i>0 && j>0)
    if (x_{i-1} == y_{j-1})
        lcs[id-1] = x_{i-1}
        i--, j--, id--
    else if (c[i-1, j] > c[i, j-1])
        i--;
    else
        j--;
Print(lcs)
```

for 1 subsequence only

for multiple all condition c[i-1,j], c[i,j-1] & c[i-1,j-1] are identical or not

~~for multiple~~ ~~else if~~

X = BCBA m = len(x) = 4
Y = BCBA n = len(y) = 4 ← no need

	Y	B	C	B	A
X					
B					
C					
B					
B					

// no need

23/11/23

Ch

Matrix chain multiplication

Imp

2×3 $(m \times n)$

$$A \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

3×2 $(n \times p)$

$$B \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

$A \times B$ 2×2

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

No of mul. required $= 3 + 3 + 3 + 3 = 12$

$2 + 3 + 2$

$\boxed{= m + n + p}$

① $B = (A_1 \times A_2) \times (A_3 \times A_4)$ $[A_1, A_2, A_3, A_4 = \text{matrix}]$ $\xrightarrow{W_1}$ 1st way of multiplication

$= A_1 \times ((A_2 \times A_3) \times A_4)$ $\xrightarrow{W_2}$ 2nd way

$= A_1 \times (A_2 \times (A_3 \times A_4))$ $\xrightarrow{W_3}$ 3rd

$= ((A_1 \times A_2) \times A_3) \times A_4$ $\xrightarrow{W_4}$ 4th

$= (A_1 \times (A_2 \times A_3)) \times A_4$ $\xrightarrow{W_5}$ 5th

A_1 A_2 A_3 A_4

20×30 30×40 40×50 50×100

the no. of scalar multiplication

$W_1 \Rightarrow 20 \times 30 \times 40 + 50 \times 40 \times 100 + 20 \times 40 \times 100$

$= 30000 + 200000$

$= 2300000$

$W_2 = 30 \times 50 \times 40 = 60000$ (for $A_2 \times A_3$)

$(30 \times 50) \times (40 \times 100) \rightarrow A_4$

dim of $A_2 \times A_3$

$30 \times 50 \times 100 = 120000$

$A_2 \leftarrow (20 \times 30) \times (30 \times 100) \times \cancel{(20 \times 30)}$

dim of $(A_1 \times A_2) \times A_4$

$20 \times 30 \times 100 = 60000$

$60000 + 120000 + 60000 = 2,40,000$

$W_3 \rightarrow 50 \times 40 \times 100 = 200000$ (for $A_3 \times A_4$)

$(50 \times 50) \times (50 \times 100)$

$30 \times 50 \times 100 = 150000$

$(20 \times 30) \times (30 \times 100)$

$20 \times 30 \times 100 = 60000$

$200000 + 150000 + 60000$

$= 4,10,000$

$W_4 \rightarrow 20 \times 30 \times 50 = 36000$

$(20 \times 50) \times (50 \times 40)$

$20 \times 50 \times 40 = 40000$

$(20 \times 40) \times (40 \times 100)$

$20 \times 40 \times 100 = 80000$

$W_5 \rightarrow 1,50,000$

$36000 + 40000 + 80000 = 1,56,000$

In order to reduce the no. of scalar mul required we need dp.

$A_1 A_2 \rightarrow 1$

$A_1 (A_2 A_3) \mid (A_1 A_2) A_3 \rightarrow 2$

$A_1 A_2 A_3 A_4 \rightarrow 5$

$A_1 A_2 A_3 A_4 A_5 \rightarrow 14$

when, no. of matrix ∴ no. of ways by which we can multiply them $= \boxed{n-1}$

(only for n = 2, 3)

$A_1 A_2 A_3 A_4$

m[1, 4]

m(1,1) m(2,4) m(1,2) m(3,4)

m(1,3) m(4,4)

m(2,2) m(3,4)

m(1,1) m(2,2) m(3,3) m(4,4)

m(1,1) m(2,3)

m(1,2) m(3,3)

m(2,3) m(4,4)

m(1,1) m(2,2)

m(2,2) m(3,3)

$$\frac{2n!}{(n!)(n+1)!} = \frac{8!}{3!\,4!} = 5$$

hence, the no. of ways given to (n+1) matrices,

∴ $n+1 = 4$

$\Rightarrow n = 3$

$$\frac{8!}{4! \times 5!}$$

$= 2 \times 7 \times 6 \times 5 / (3 \times 2)$

$= 14$

unique entries –

m(1,1)	m(2,2)	m(3,3)	m(4,4)

dim = 1

m(1,2)	m(2,3)	m(3,4)

dim = 2

m(1,1)	m(2,2)	m(3,3)	m(4,4)	– 4
m(1,2)	m(2,3)	m(3,4)		– 3
m(1,3)	m(2,4)			– 2
m(1,4)				– 1

∴ 4+3+2+1 = 10 unique entries

$A_1 \quad A_2 \quad A_3 \quad A_4$

$1\times2 \quad 2\times4 \quad 4\times3 \quad 3\times4$

(0)

0	0	0	0
8	24	48	
20	48		
32			

← affinity = 0 | the matrix not multiplying with itself = 0

$(A_1 \cdot A_2)(A_3 A_4)$

$1+2+4 = 8$ $\quad 4+3+4 = 48$

$(1+4) + (4+3) = 1+4 = 4$

$8 + 48 + 4 = 60$

56 ← $A_1 * ((A_2 * A_3) * A_4)$

88 ← $A_1 * (A_2 * (A_3 * A_4))$

min ← 32 ← $((A_1 * A_2) * A_3) * A_4$

42 ← $(A_1 * (A_2 * A_3)) * A_4$

$A_1 \cdot A_2$: $1 \times 2 \times 4$

$A_2 A_3$: $2+4+1$

$A_3 A_4$: $4+1+4 = 16$

$(A_1 A_2) A_3$: $1+2+4 = 8$; $1+4+3 = 12$; min ← $8+12 = 20$

$A_1 (A_2 A_3)$: $2+4+3 = 24$; $1+2+3 = 6$; $24+6 = 30$

$(A_2 A_3) A_4$: $2+4+3 = 24$; $2+3+4 = 24$; min ← $24+24 = 48$

$A_2 (A_3 A_4)$: $4+3+4 = 48$; $2+4+4 = 32$; $48+32 = 80$

$2+4+3 = 24$; $2+3+4 = 24$; $1+2+4 = 8$; $1+2+4 = 8$; $1+4+3 = 12$; $8+12 = 20$

$2+4+3 = 24$

$1+2+3 = 6$

$1+3+4 = 12$

$4+3+4 = 48$

$2+4+4 = 32$

$1+2+4 = 8$

$1+2+4 = 8$

$1+4+3 = 12$

$2+3+4 = 24$

$1+2+4 = 8$

$1+4+3 = 12$

$2+1+4 = 8$

$1+2+4 = 8$

$1+4+3 = 12$

28

24/4/23

matrix chain multiplication

(5) $A_1 \cdot A_2 A_3 \ldots A_n$

$((A_1 \cdot A_2) \cdot A_3) \cdot A_4$

$1\times2 \quad 2\times1 \quad 1\times4 \quad 4\times3$

$((A_1 \cdot (A_2 A_3)) A_4)$

$1\times2\times1 = 2$

$1\times1\times4 = 4$

$1\times4\times3 = 12$

1	m[1,1] 0	m[2,2] 0	m[3,3] 0	m[4,4] 0
2	m[1,2] 2 1	m[2,3] 8	m[3,4] 12 3	
3	m[1,3] 6 2	m[2,4] 18 2		
4	m[1,4] 17 2			

$m[2,4)$

$m[2,3] + m[3,4] + 2 \times 4 \times 3 = 32$

$m[2,2] + m[3,4] + 2 \times 1 \times 3$

$= 8$

$m[1,4]$

$= m[1,1] + m[2,4] + 1 \times 2 \times 3$

$= 0 + 18 + 6$

$= 24$

$m[1,2] + m[3,4] + 1 \times 1 \times 3$

$= 2 + 12 + 3$

$= 17 \rightarrow min$

$m[1,3] + m[4,4] + 1 \times 4 \times 3$

$= 6 + 0 + 12$

$= 18$

$m[1,3]$
$= m[1,2] + m[3,3] + 1 \times 1 \times 4$
$= 2 + 0 + 4$
$= 6$

total no. of scalar mul in this case =

$= 8 (A_1 A_2)(A_3 A_4)$

$= 2 + 12 + 17$

$= 31$

$|A \times B| = 2$
$2 \times 1 \times 4 = 8$
$1 \times 4 \times 3 = 12$

$|A \times B| = 2$
$1 \times 1 \times 4 = 4$
$2 \times 1 \times 4 = 8$
$1 \times 2 \times 4 = 8$
$2 \times 1 \times 3 = 6$
$2 \times 4 \times 3 = 12$
$1 \times 4 \times 3 = 12$
$2 \times 1 \times 3 = 6$

3) matrix chain mul - It is an optimization problem that involves finding the most significient way to multiply a sequence of matrix.

Longest common subsequence - The longest common subsequence is the length of the longest sequence of chars that appears in the same orders within two given strings.

28/11/23

Q) $A_1 \quad A_2 \quad A_3 \quad A_4$

$10 \times 100 \quad 100 \times 20 \quad 20 \times 5 \quad 5 \times 80$

$P_0 \; P_1 \quad P_1 \; P_2 \quad P_2 \; P_3 \quad P_3 \; P_4$

m(1,1) 0	m(2,2) 0	m(3,3) 0	m(4,4) 0
m(1,2) 20,000 1	m(2,3) 10000 2	m(3,4) 8000 3	
m(1,3) 15000 1	m(2,4) 50000 3		
m(1,4) 15000 3			

$(A_1(A_2 A_3)) \quad ((A_1 A_2) A_3)$

$(1,3) = min \begin{cases} m(1,1) + m(2,3) + P_0 \times P_1 \times P_3 = 0 + 10000 + 5000 = 15,000 \\ m(1,2) + m(3,3) + P_0 \times P_2 \times P_3 = 20000 + 0 + 1000 = 21,000 \end{cases}$

$(A_2 (A_3 A_4))$

$(2,4) = min \begin{cases} m(2,2) + m(3,4) + P_1 \times P_2 \times P_4 = 0 + 8000 + 20 \times 80 = 16,000 \\ m(2,3) + m(4,4) + P_1 \times P_3 \times P_4 = 10,000 + 0 + 100 \times 5 \times 80 = 50,000 \end{cases}$

50×100

$10 \times 20 \times 5$

$2000 \; 10$

$40,000$

$100 \times 20 \times 10$

$16,0000$

8000

$A_1 A_2 A_3 A_4$

$$(1,4) = \min \begin{cases} m(1,1) + m(2,4) + 10 * 100 * 80 \\ \quad = 0 + 50{,}000 + 80000 = 1{,}30{,}000 \\ m(1,2) + m(3,4) + 10 * 20 * 80 \\ \quad = 20{,}000 + 8000 + 16000 = 44{,}000 \\ m(1,3) + m(4,4) + 10 * 5 * 80 \\ \quad = 15{,}000 + 0 + 40000 \\ \quad = 19{,}000 \end{cases}$$

$((A_1 (A_2 A_3)) A_4)$ → parenthesise should be like that to get lesser tc (lesser no. of scalar mul)

∴ the total no. of scalar mul ~~here~~ required in order to mult the chain of this 4 matrixes =

$19{,}000 + 15{,}000 + 10{,}000$

$= 44{,}000$

(DP approach with tabular method)

tc,

tc of table $= O(n^2)$

tc of split $= O(n)$

∴ $tc = O(n^2) * O(n)$

$\boxed{tc = O(n^3)}$

[basically in split tc = 2^n if we use dp approach then tc will be $= O(n^3)$]

SC,

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

Algo for matrix chain mul (iterative method) (DP approach)

```
matrix-chain(p)        // p = array with dims of n mats
{
  n = len(p) - 1
  let m[1..n, 1..n] & s[1..n-1, 2..n] be new tables
  for i = 1 to n
      m[i,i] = 0 → recurrence relation of m
  for l = 2 to n
  {
    for i = 1 to n - l + 1
    {
      j = i + l - 1
      m[i,j] = ∞
      for k = i to j-1
      {
        q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
        if (q < m[i,j])
        {
          m[i,j] = q
          s[i,j] = k
        }
      }
    }
  }
  return m, s
}
```

[At recursive method the SC will be proportional]

	1	2	3	4
1	m(1,1) 0	m(1,2)	m(1,3) 0	m(1,4) 0
2	m(2,2)	m(2,3)	m(2,4)	
3	m(3,3)	m(3,4)		
4	m(4,4)			

$1 + 2 - 1 = 2$

$2 + 2 - 1 = 3$

↳ recurrence relation of m

0/1 knapsack

1/12/23

Fractional Knapsack,

$n = 3$

	obj1	obj2	obj3
w	1	2	4
p	10	12	28

capacity = 6 → can be solved using greedy method

Pr. ~~p/w~~ 10 6 7

6	obj2/2 = 1
28	obj3
10	obj1

total Profit = 44

0/1 Knapsack,

→ can be solved using dp approach

28	4
10	1

total Profit = 38 (not the best Profit)

12	2
28	4

total Profit = 40 (best Profit)

Suppose there n obj's with weight w. For Every obj have 2 choice (either put it fully or not put it). so, $2 \times 2 \times 2 \times \dots n$

$= 2^n$

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

to reduce the tc we have to apply dp. approach to the 0/1 knapsack.

0/1 knapsack prob

no. of obj = i

weight = w

$$KS(i, w) = \begin{cases} \max(p_i + KS(i-1, w-w_i), KS(i-1, w)) & ; w_i \le w \\ 0 ; i = 0 \text{ or } w = 0 \text{ (base point)} \\ KS(i-1, w) ; w_i > w \end{cases}$$

no of obj after putting ith obj & the remaining weight = $w - w_i$ (eg. $c = 7$)

if we'll include the ith obj

Recursive eqn using DP approach.

$(n+1)$ row $\times$ $(c+1)$ col

↳ we have to make tables of this dimention

eg. if $c = 6$ & obj 4 then we have no choice than ~~that~~ to exact exclude it & consider remaining objs

$n = 3$

$C = 6$

	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	12	22	22	22	22
3	0	10	12	22	28	38	40

$KS(i, w)$: obj, max capacity

$w_1 = 1$

$p_1 = 10$

$KS(1,1) = \max\{10 + KS(0,0), KS(0,1)\}$

~~KS(2,1)~~ mat(n, c) = 40 = max Profit

$KS(2,1) = KS(1,1)$

$KS(2,2)$

$\max(12 + KS(1,0), KS(1,2))$

$KS(3,4)$

$\max(28 + KS(2,0), KS(2,4))$

obj1	obj2	obj3
1	2	4
10	12	28

How much weight & how much profit for ith obj

tc → $\boxed{tc = O(n \times w)}$ [without considering dp approach $= O(n \times 2^n)$]

If the tc is below 2^n then we have to use ~~the~~ fractional knapsack]

So, $\boxed{tc = O(n \times w)}$

[In DP, tc = no. of ~~sales~~ cells]

Fractional knapsack problem – It is about efficiently selecting items with varying weights & values to maximize the total value with a limited capacity knapsack, allowing for partial inclusion.

0/1 knapsack problem – The 0/1 knapsack problem involves selecting items with weights & values to maximize total value without exceeding a set capacity where items cannot be divided.

0/1 knapsack

	1	2	~~w~~ 3
w	1	2	4
P	10	12	28

$$KS(i,w) = \begin{cases} \max(P_i + KS(i-1, w-w_i),\ KS(i-1, w)) & ; w_i \le w \\ 0 & ; i=0 \text{ or } w=0 \\ KS(i-1, w) & ; w_i > w \end{cases}$$

Algo

Input: $\{w_1, w_2, \ldots, w_n\}$, $\{P_1, P_2, \ldots P_n\}$, C (capacity of knapsack)

Output: $KS[n+1, C+1]$

```
{for i in 0 to c
      KS[0, i] = 0
 for i in 0 to n
      KS[i, 0] = 0
 for i = 1 to n
 {  for j = 1 to c
    {  if (w_i ≤ j) and P_i + KS[i-1], j-w_i] > KS[i-1, j]
       {  KS[i, j] = P_i + KS[i-1, j-w_i]
       }
       else {
          KS[i, j] = KS[i-1, j]
       }
    }
 }
 Print(KS)
 //return KS[n][c] → max profit
}
```

2

PYQ(daa) - 2022

Gp A

1) Dijkstra's Algorithm is used to solve single source shortest path problem.

2) Kruskal's algo is a greedy algorithm

3) What is the average case time complexity of merge sort O(nlogn).

4) What is the median of three techniques in quick sort? choosing median of first last & middle element as pivot.
~~Quick sort with random choice of pivot~~

5) Which of the following methods can be used to solve the knapsack problem? brute force, recursion & dynamic programming

6) Consider the ~~m~~ 2 matrices P & Q which are 10x20 & 20x30 matrices respectively. What is the number of multiplications required to multiply the 2 matrices 10*20*30.

G-B

1. ~~Describe matrix chain multiplicatio~~
Define big-O, big-theta notation. Write the graphical interpretation of it.
Done, Done

2) Suppose you have two sorted array of size n. How you get a single sorted array from this two using merging technique?
Done (with eg ~~#~~)

3) Describe how to solve knapsack problem using greedy algorithm. Done(with eg)

4) Describe Breadth first search & Depth First search of graph with example. (Not in syllabus)

5) Define spanning tree of graph? How to find an independent set of tree using greedy technique?
Done. Not in syllabus

G-C

1) Describe matrix chain multiplication problem. Write & explain the reccurence relation for solving matrix chain multiplication problem using dynamic programming technique. Solve following matrix chain multiplication problem using dynamic programming:

A1: 20x5 A2: 5x50 A3: 50x10 A4: 10x30

MCM is an optimization problem that involves finding the most significant way to multiply a sequence of matrix. It's goal is to minimize the numbers of scalar multiplications required. The problem is not about performing the multiplications, but rather deciding the sequence of the matrix multiplications. This is because matrix multiplication is associative,

i.e, the result will be the same regardless of how the product is parenthesized.

The optimal solution can be achieved using dp. ~~This~~ This involves constructing a table to store intermediate results. The table stores the optimal costs of multiplying various subchains of matrices.

The recurrence relation for solving mcm using dp —

$$M[i,j] = \begin{cases} 0 & ; i=j \\ \min\limits_{i \le k < j} \{ M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \} \end{cases}$$

In matrix chain multiplication problem the objective is to minimize the total no. of multiplications made in computing the product of n matrices $A_1, A_2 \ldots A_n$ whose dimentions are d_0 by d_1, d_1 by d_2 ... d_{n-1} by d_n, respectively. All intermediate products of 2 matrices are done by brute force algorithm. M is matrix here. 2-dimentions.

$A1: 20\times5$, $A2: 5\times50$, $A3: 50\times10$, $A4: 10\times30$

$p_0\ p_1$; $p_1\ p_2$; $p_2\ p_3$; $p_3\ p_4$

m(1,1) 0	m(2,2) 0	m(3,3) 0	m(4,4) 0
m(1,2) 5000 1	m(2,3) 2500 2	m(3,4) 15,000 3	
m(1,3) 3500 1	m(2,4) 4000 2		
m(1,4) 7000 1			

$20\times5\times50$
100×50
5000
$5\times50\times10$
5×500
2500
$(A_1A_2)A_3$
(A_1A_2)
$50\times10\times30$
500×30
$15,000$

$(m(1,2)\ p_3)$
$((A_1\times A_2)A_3)$

$$m(1,3) = \min \begin{cases} m(1,1) + m(2,3) + p_0 \times p_1 \times p_3 \\ = 0 + 2500 + 20\times5\times10 \\ = 3500 \\ m(1,2) + m(3,3) + p_0 \times p_2 \times p_3 \\ = 5000 + 0 + 20\times50\times10 \\ = 15000 \end{cases}$$

$(A_2\ A_3)\ A_4$
$(A_2(A_3A_4))$

$$m(2,4) = \min \begin{cases} m(2,3) + m(4,4) + p_1 \times p_3 \times p_4 \\ = 2500 + 0 + 5\times10\times30 \\ = 4000 \\ m(2,2) + m(3,4) + p_1 \times p_2 \times p_4 \\ = 0 + 15,000 + 5\times50\times30 \\ = 22500 \end{cases}$$

$A_1(A_2A_3A_4)$
$(A_1A_2)(A_3A_4)$
$(A_1A_2A_3)A_4$

$$\min(1,4) = \min \begin{cases} m(1,1) + m(2,4) + p_0 \times p_1 \times p_4 \\ = 0 + 4000 + 20\times5\times30 = 7000 \\ m(1,2) + m(3,4) + p_0 \times p_2 \times p_4 \\ = 5000 + 15000 + 20\times50\times30 = 50,000 \\ m(1,3) + m(4,4) + p_0 \times p_3 \times p_4 \\ = 3500 + 0 + 20\times10\times30 = 9500 \end{cases}$$

~~the~~ : parenthesise should be like that to get lessers no. of scalers mul =

$(A_1((A_2 \ A_3) \ A_4))$

∴ the total no. of scalers mul. required ~~to get lessers no. of scalers mul~~ (lessers no of scalers mul) in ordered to mul the chain of 4 mat. = ~~2000 +~~

2500 + 4000 + 7000 = 13,500

⇒ In which complexity class the following function belong? $f(n) = 5n^3 + 6n$. Prove that for any polynomial $P(n)$ of degree K, $P(n) = O(n^K)$. Solve the following recurrence relation by using master's theorem:

$T(n) = 7T(n/2) + 18n^2$

$f(n) = 5n^3 + 6n$, let's say $c = 20$, & $n_0 = 1$

~~$n_0 = 1$~~ ∴ $n \geq n_0$, ∴ $n \geq 1$, let's $g(n) = n^4$

∴ $f(n) \leq c \cdot g(n)$

⇒ $5n^3 + 6n \leq 20n^4$

Plugging $n = 1$, we get

putting

⇒ $5(1)^3 + 6(1) \leq 20 \cdot (1)^4$

⇒ $11 \leq 20$ → that is true

∴ we can conclude that $f(n) = 5n^3 + 6n$ belongs to $O(n^4)$ or big-O complexity class.

To prove that $P(n) = O(n^K)$, we must show that there exists a constant c such that $|P(n)| \leq cn^K$ for all n sufficiently large.

~~$f(n) = 5n^3 + 6n$~~

let $c = c'$ & $g(n) = n^K$ [K = degree]

$n_0 = 1$, ∴ $n \geq n_0$, ∴ $n \geq 1$

now,

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

⇒ $P(n) \leq c' \cdot n^K$

now we know that degree of $P(n)$ is K.

let assume P as $= c'' n^K$ [where $c'' < c'$]

∴ ⇒ $c'' n^K \leq c' n^K$

∴ we can say tc of this polynomial

$P(n) = O(n^K)$ [Proved]

(a) we know that tc of any algo or polynomial is $O(n^{\text{its highest degree}})$. Now the polynomial is of degree K, i.e $P(n) = c_1 n^K + c_2 n^{K-1} + c_3 n^{K-2} + \cdots + 1$

i.e. the heighest degree of the polynomial is K. According to this tc should be $= O(n^K)$

∴ $T(n) = O(n^2)$ (Ans)

$T(n) = 7T(n/2) + 18n^2$ where, $a = 7$, $b = 2$

applying master's theorem we get,

$f(n) = 18n^2$

$f(x)$ $\quad 18 \quad n^{\log_b a}$

$= n^{\log_2 7}$

$= n^{2.81}$

$\therefore$ time $= O(n^{2.81})$

3) State the 0/1 knapsack problem. Consider instance of the 0/1 knapsack problem as below with p depicting the value & w depicting the weight of each item whereas m denotes the total ~~w~~ weight carrying capacity of the knapsack. Find optimal answer using dynamic programming technique.

$p: [10\ 40\ 30\ 50]$ $w = [5\ 4\ 6\ 3]$, $m = 10$

If fractional part of items are allowed to take then what will be the optimal solution.

Done

$p_i [10\ 40\ 30\ 50]$ $w = [5\ 4\ 6\ 3]$, $m = 10$ → max capacity

P	~~10~~	~~40~~	~~30~~	~~50~~
W				

	obj1	obj2	obj3	obj4
W	5	4	6	3
P	10	40	30	50

~~no. of objs(n) = 4~~ no. of obj $(n) = 4$, $m = 10$

$$KS(i,W) = \begin{cases} \max\left(p_i + KS(i-1, W-w_i),\ KS(i-1,W)\right) & ; w_i \le W \\ 0 ; i = 0, \text{ or } W = 0 & \\ KS(i-1, W) ; w_i > W & \end{cases}$$

i denotes ith no. of Obj.

$p_0 + KS(3,0)$, $KS(3,0)$
$= 50 + 40$
$KS(3,1)$ ...

$p_3 + KS(2,3)$, $KS(2,3)$
$= 30 + 40$
$KS(2,3)$ ...

$30 + KS(2,0)$, $KS(2,3)$
$= 40$

$p_2 + KS(1,1)$, $KS(1,3)$
$= 30 + 10$
$KS(1,1)$, $KS(1,3)$
$= 30 + 10$
$= 40$

$p_2 + KS(1,0)$, $KS(1,3)$
$= 30 + 0$
$KS(1,0)$, $KS(1,3)$

$p_1 + KS(0,1)$, $KS(0,1)$
$= 10 + 0$
$= 10$

$10 + KS(0,0)$, $KS(0,1)$
$= 10$

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

∴ max Profit = 90

If fractional part of items are allowed then we have to use fractional knapsack in this case, the optimal soln will be –

Item	I_1	I_2	I_3	I_4	
~~weight~~ W	5	4	6	3	m = 10 (max capacity)
P	10	40	30	50	
P/W	2	10	5	16.67	

By solving the knapsack Problem in descending order of max ~~Profit~~ P/W we get –

	I_4	I_2	I_3
W	3	4	$\frac{3}{6}$
P	50	40	$\frac{3}{6} \times 30 = 15$

∴ total Profit (max Profit) = 50 + 40 + 15
= 105

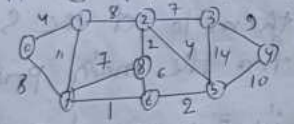
4) Analyse time complexity of Quick sort algo (write the recurrence relation & solve it) when it achieves quadratic complexity? what is the difference b/w divide and conquer & greedy techniques write the generalized eqn of divide & conquer technique. Done. Done

divide & Conquer	greedy technique
i) many decisions & sequences are guaranteed & all the overlapping sub-instances are considered.	i) divide the given problem into many sub problems. Find the individual solutions & combine them to get the solution for the main problem.
ii) Follows bottom-up technique.	ii) follows top down technique.
iii) Sub Problems are dependent on the main Problem.	iii) sub problems are independent of the main Problem.
iv) splits the input at every points rather than at a particular point.	iv) split the input, only at specific points, each problem is independent.

generalized eqn of divide & conquer technique

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

5) Describe single source shortest path problem. For the following graph find the shortest paths from the source (vertex 0) to all vertices using Dijkstra algorithm. What is optimal substructure property of dynamic programming & how it is preserved in this algorithm?



← undirected graph (Dijkstra works for both directed & undirected graph)

The single-source shortest path (SSSP) problem is a classic algorithmic graph ~~thro~~ theory problem. It involves finding the shortest paths from a single vertex to all other vertices in a graph. The sum of the weights of the edges in the path is minimized. This problem can be solved using algos such as BFS (Breadth-first search) for unweighted graphs or Dijkstra. The steps – i) mark the ending vertex with a distance of 0. ii) Find all vertices leading to the current vertex. iii) Calculate their distances to the end. iv) mark the current vertex as visited. v) mark the vertex with the smallest distance as current, and repeat from step 2.

source = vertex 0

	0	1	2	3	4	5	6	7	8
Initial	$\boxed{0}$*	∞	∞	∞	∞	∞	∞	∞	∞
selected 0 = 0	0	min(∞, 4+0) = $\boxed{4}$*	∞	∞	∞	∞	∞	min(∞, 8+0) = 8	∞
selected 1 = 4	0	4	min(∞, 8+4) = 12	∞	∞	∞	∞	min(8, 11+4) = $\boxed{8}$*	∞
selected 7 = 8	0	4	min(12, 12)	∞	∞	∞	min(∞, 1+8) = $\boxed{9}$*	8	min(∞, 7+8) = 15
selected 6 = 9	0	4	12	∞	∞	min(∞, 2+9) = $\boxed{11}$*	9	8	min(15, 6+9) = 15
selected 5 = 11	0	4	min(12, 4+11) = $\boxed{12}$*	min(∞, 14+11) = 25	min(∞, 10+11) = 21	11	9	8	15
selected 2 = 12	0	4	12	min(25, 7+12) = 19	21	11	9	8	min(15, 2+12) = $\boxed{14}$*
selected 8 = 14	0	4	12	$\boxed{19}$*	21	11	9	8	14
selected 3 = 19	0	4	12	19	min(21, 7+19) = $\boxed{21}$*	11	9	8	14
selected 4 = 21	0	4	12	19	21	11	9	8	14

6=9									
selected 4=11	0	4	min (12, 4+11) = 12 *	min (∞, 14+11) = 25	min (∞, 10+11) = 21	11	9	8	min (15 2 =

shortest distance

0 — 0 = 0
0 — 1 = 4
0 — 2 = 12
0 — 3 = 19
0 — 4 = 21
0 — 5 = 11
0 — 6 = 9
0 — 7 = 8
0 — 8 = 14

shortest path

- (no need)
0 ← 1
0 — 1 — 2
0 — 1 — 2 — 3
0 — 7 — 6 — 5 — 4
0 — 7 — 6 — 5
0 — 7 — 6
0 — 7
0 — 1 — 2 — 8

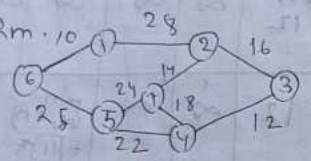
selected	0	4	12						

In dP, the optimal substructure property states that the optimal solution to a problem can be found by combining the best solutions to its subproblems, i.e, when building a soln for a problem, we can split the problem into smaller problems & only consider the optimal solution to the smaller problems.

Optimal S.P
[function(n)]
depends on
[function(n-1)]

Here it is preserved by following - if we are splitting this problem into smaller problems & taking the optimal soln to the smaller problems.

6) Construct the minimum spanning tree (MST) for the given graph using prim's algorithm.



Is it a greedy technique? If yes then why? Describe the greedy move of the Prim's algorithm. Describe the difference b/w dynamic programming & divide & conquer technique.

Done

yes, it is a greedy technique.

It is a greedy technique because it makes the locally optimal choice at each step. It is an adaptive greedy technique as it reorders it's input to choose the cheapest edge. At every iteration, the algo tries to readjust the i/p to it's own convenience.

(ii) ——————— here we are trying to get the most sufficient way to get the minimum cost spanning tree.

Greedy move of Prim's algo - It is to add edge with the minimum weight to the spanning tree, as long as the addition of the edge doesn't create a cycle. This move is greedy because it only considers the current state of the spanning tree & doesn't look ahead to see if there is a better soln that could be found by adding a different edge later.

However, the greedy move of Prim's algo is guaranteed to find minimum spanning tree of the graph.

DP	Divide & Conquer technique
i) Solve Problems by dividing Problems into subproblems.	i) Solve Problems by dividing Problems into subproblems.
ii) Subproblems are dependent ~~are~~ dependent of each other.	ii) Subproblems aren't dependent of each other.
iii) Bottom-up algo.	(iii) top-down algo.
iv) Non-recursive	iv) Recursive
v) more efficient	v) Less efficient
vi) ~~more~~ Less time consuming	vi) ~~Less~~ more time consuming.

Eg – mcm, 0/1 knap-sack.

Eg – merge, quick sort, binary search.

2021

⑤ A = [3 6 19 30 55 60]

B = [15 18 20 35 40 44]

applying merging technique

A =

3	6	19	30	55	60
i 0	i 1	2	3	4	5

B =

15	18	20	35	40	44
j 0	1	2	3	4	5

merged array

3	6	15	18	19	20	30	35	40	44	55	60
low	k	k	k	k	k	k	k	k	k	k	k

no. of Comparisons = 10

[In general no. of comparisons = total no. of elems in two-tele no. of elems in merged array, but here 55 & 60 is placing by while loop so no. comparisons ~~here~~ in their case]

⑥

item	obj1	obj2	obj3	obj4	obj5
w	80	10	40	20	90
P	40	10	50	30	60

m (max capacity weight carrying) = 110

total no. of objects (n) = 5

max capacity (W) = 110

	0	10	20	30	40	50	60	70	80	90	100	110
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	40	40	40	40
2	0	10	10	10	10	10	10	10	40	50	50	50
3	0	10	10	10	50	60	60	60	60	60	60	60
4	0	10	30	40	50	60	80	90	90	90	90	90
5	0	10	30	40	50	60	80	90	90	90	90	90

n+1

recurrence relation for 0/1 knapsack

$$KS(i,W) = \begin{cases} \max(P_i + KS[i-1, W-W_i],\ KS[i-1,W]) & ; W_i \le W \\ 0 & ; i=0 \text{ or } W=0 \\ KS[i-1,W] & ; W_i > W \end{cases}$$

i = ith no. obj

W = weight

[max profit = 90]

Path

i = 1, W_i = 80

KS(1,10) = KS(0,10)

KS(1,80)

40 + KS(0,0) KS(0,80)

KS(1,10) KS(0,70)

i = 2, W_i = 10

KS(2,10)

10 + KS(1,0)

10 + KS(1,10) KS(1,10)

KS(1,80)

i = 3 W_i = 40

KS(3,10)

KS(2,10)

KS(2,70)

50 + KS(2,0) KS(2,70)

i = 4

W_i = 20

KS(4,70)

KS(3,10)

KS(4,50)

50 + KS(3,0), KS(3,30)

i = 5 W_i = 90

KS(5,10)

KS(4,10)

KS(5,70)

60 + KS(4,0)

60 KS(5,70)

If fractional part is allowed then we will use fractional knapsack

Item:	I_1	I_2	I_3	I_4	I_5
W:	80	10	40	20	90
P:	40	10	50	30	60
P/W:	0.5	1	1.25	1.5	0.67

given m:110

~~we will~~ applying Greedy knapsack

we will select the items according to ~~their~~ the descending orders of Profit/weight.

∴ then we get –

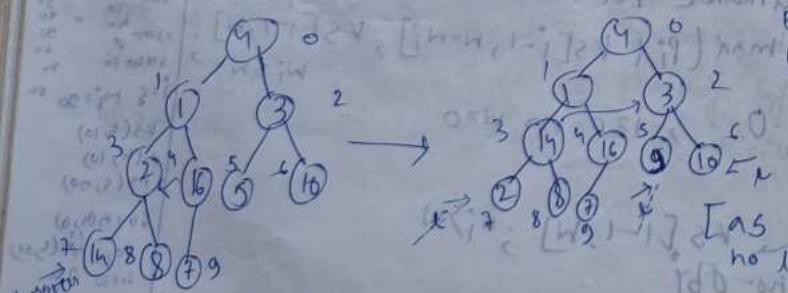
~~m:110~~

Item:	I_4	I_3	I_2	I_5
W:	20	40	10	40/90
P:	30	50	10	$\frac{40}{90} \times 60 = 26.67$

∴ Max Profit = 116.67

(8)

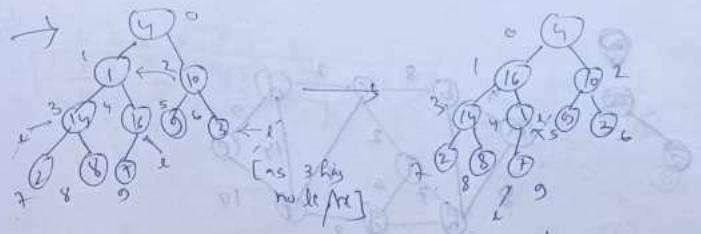
4	1	3	2	16	9	10	14	8	7

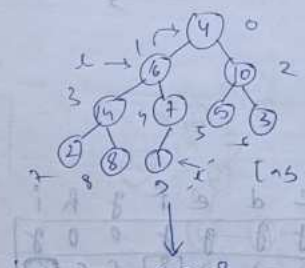


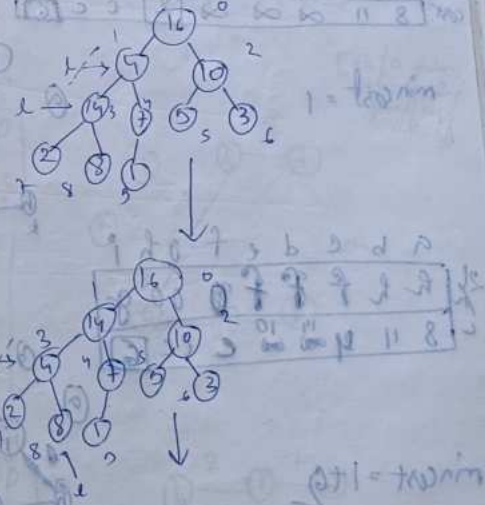
l = longest Pointer [for pointing]

last non leaf = 4 $\left[\left\lfloor \frac{n-1}{2} \right\rfloor = 4\right]$

we will apply mh from = 0 to 4

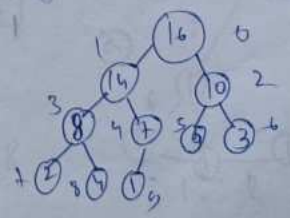




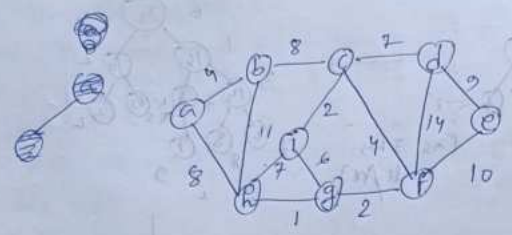


∴ after mh the array is

16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9



7



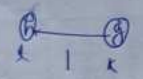
apply Prim's algo

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

1)

	a	b	c	d	e	f	g	h	i
near	h	h	g	g	g	g	0	0	g
cost	8	11	∞	∞	∞	2	c	c	6

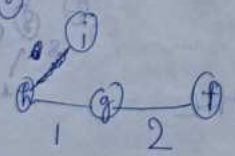
mincost = 1



2)

	a	b	c	d	e	f	g	h	i
h	h	h	f	f	f	0	0	0	g
c	8	11	4	14	10	c	cc	c	6

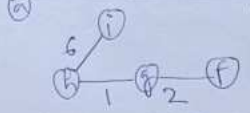
mincost = 1+2



3)

	a	b	c	d	e	f	g	h	i
h	h	h	i	f	f	0	0	0	0
c	8	11	2	14	10	c	c	c	c

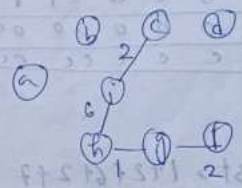
mincost = 1+2+6



4)

	a	b	c	d	e	f	g	h	i
h	h	c	0	c	f	0	0	0	0
c	8	8	c	7	10	c	c	c	c

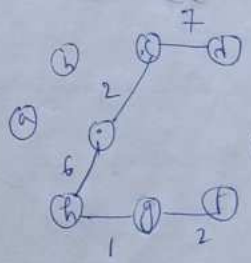
mincost = 1+2+6+2



5)

	a	b	c	d	e	f	g	h	i
h	h	c	0	0	d	0	0	0	0
c	8	8	c	c	9	c	c	c	c

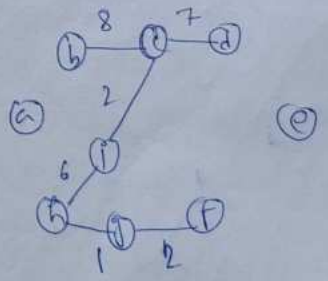
mincost = 1+2+6+2+7



6)

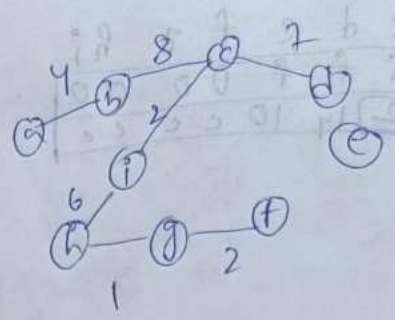
	a	b	c	d	e	f	g	h	i
h	b	0	0	0	d	0	0	0	0
c	4	c	c	c	9	c	c	c	c

mincost = 1+2+6+2+7+8



7)

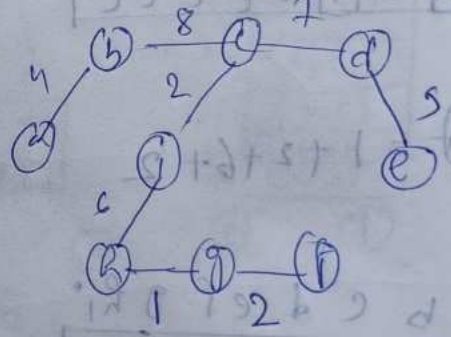
	a	b	c	d	e	f	g	h	i
h	0	0	0	0	d	0	0	0	0
c	c	c	c	c	$\boxed{g}$	c	c	c	c

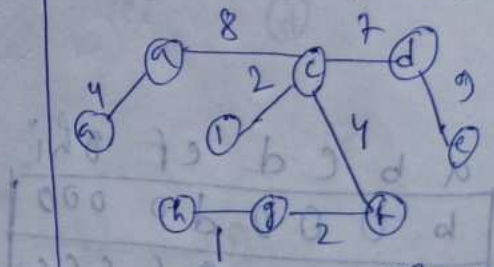


mincost = 1+2+6+2+7+8+4

8)

	a	b	c	d	e	f	g	h	i
h	0	0	0	0	0	0	0	0	0
c	c	c	c	c	c	c	c	c	c



mincost = 1+2+6+2+7
+8+4+9
= 39

MST = 4+8+7+9+2
+4+2+1
= 37