24/7/20

DAA

- Algorithm Design & Analysis (ADA)
(Create algos for prob., then analyse the algos. created – which algo is better)

Algo: Stepwise rep. of the solu. ~~to for~~ a ~~particular~~ prob. having the characteristics: (IDEF)

1) Input & output – Every algo should have a non-empty set of i/o
I/p – keyboard / disk file / i/p port
O/p – display / printer / disk file

2) Definiteness – Every inst. that we write in algo should be definite, have a unique meaning, not ambiguous.

Eg: $a = b + c$ ✓
$a = b + (c \text{ or } d) \rightarrow a = b + c$ OR $a = b + d$ ✗

3) Effectiveness – Each st. The st. we write, ~~the~~ those st. must be computed by a human in a finite ~~no~~ amt. of time with a pen & paper
Eg: $a = b^c$
Hence, the comp. can also do it in a much lesser time. ~~If~~ (finite)
If human → ∞ time,
comp. → ∞ time

4) Finiteness – Taking all st. together Every algo must be terminated in a finite ~~no~~ amt. of time period. It should have finite no. of steps. Every algo should have a terminating cond. which satisfies in a finite amt. of time.

Eg. – no ∞ loop

Scanned with CamScanner

One algo that has no terminating cond., ~~but~~ No finiteness prop. But still it is an algo. That algo. is the OS algo. (once the sys. is started, it doesn't know when to end), forcibly terminated by user.)

28/7/20.

The area on which we study about an algo:

i) Design - how to devise an algo. for a particular prob.
No formal procedure to devise any algo. (an art)
(Not like drawing a comb. ckt in DEC)
Still some design criterias/procedures are there for designing (no fixed criteria/procedure - dep. from prob. to prob.)

ii) (roughest area of studying algo.) Validity/Probability: When we design an algo, there is a theory ~~that~~ ~~says~~ to prove that it works for any i/p. on the computer. That theory is called validity/prob. theory.

iii) Analysis: Analyse the algo provided it is correct.

Performance analysis: The end user of an algo is the comp. We can't do anything with an algo unless we run/execute it. We see, how does the algo perform on a comp.
- ~~Execution Time~~: We check whether the algo takes a lot of res. / less res. of the comp.
- ~~M/M Requirement~~: If the algo that takes less res. compared to others, is better than the one that ~~makes~~ takes more. Comp. has a lot of res., so which res. should we consider while doing the performance analysis.

Page No.
Date
Prime Merit

Scanned with CamScanner

* So ~~but~~ less exe. time → better performance → better algo
However, if need arises, we may consider mem. as a para.

Two main res. that ~~a~~ the comp. takes always for running an ~~an~~ algo.:

i) CPU ii) M/M ~~req~~.

a) Execution/Run Time (CPU time)
b) M/M req. ~~[illegible]~~

] Less exe. time & less mem. → better algo. (better performance)

Less CPU time & less M/M is a better one than ~~two~~ the one that takes more CPU time & more M/M.

Q) Is CPU time & m/m req. is directly/inversely prop. or are they not related?

Ans) Loop – ~~more~~ less mem., more time
St. – ~~less~~ more mem., ~~even~~ less ~~more~~ time

] → inversely prop.

This happens in some cases. In some cases → directly prop.
So, CPU & m/m req. are neither directly nor inversely prop.

* Normally when we go for analysis, looking a the availability of res., we consider the exe. time (more priority), not m/m req for performance analysis, because we have plenty of m/m, but there is a ~~scarcity~~ scarcity in exe. time. *

ii) Testing: Testing is reqd. to check if the algo works for every kind of i/p. data (test cases)
Compared to design, validity, analysis, it is the simplest ~~op~~ of them.

We need to test whether the algo. is working for every test cases and every real life cases (When eg.: 1000 users access the same prog.), for any dataset, whether the algo is accessed by multiple users in the internet, whether availability is there, whether it works ~~in~~ for single user / multiple user. Eg: CS prob.

Prime Merit
Page No.
Date

Scanned with CamScanner

D , A , V , T of Algo.

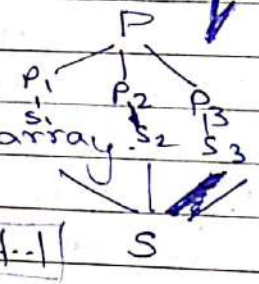
D A (V) (T) A (algo.) ⟶ D A A (syllabus)

Design Methodology:

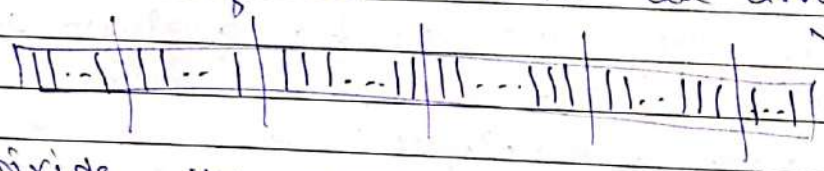
Numerous design meth. are there to apply. The fou. are most imp.t and used in most algo.

i) Divide and Conquer:

Given a large prob., it is divided into no. of subprob. and we find the soln. to each of these subprob. These solns. are combined (conquered) to find the final soln.



Eg. Sum of 1000 nos. in an array.

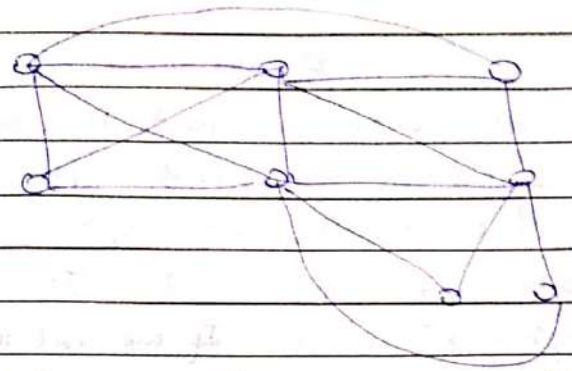


Divide the array ~~when each~~ into 10 parts where each part consists of 10 ele. Find the sum of ele. of each part, and then find the sum of those results.

ii) Greedy Approach:

It is used to find an optimal soln.

Eg.: Given a graph where each nodes rep. diff. cities of a country.



We need to find the shortest dist. b/w 2 cities.

specified by Greedy approach

Basic approach: There is a criteria using which the vars. are arranged in increasing/decreasing order (sorted order).

sorted criteria

i) Arrange in a particular order for some para.

Feasibility Test

ii) Go on choosing item and check if it is feasible. If feasible → optimal soln. (always) 'S' (Feasibility Test). There is a soln. vector

Go on adding that item to S, only if the the soln. is feasible. If not feasible → discard.

Feasibility test is also guided by Greedy approach.

Prime Merit
Page No.
Date

Scanned with CamScanner

iii) Dynamic Programming. Also used to find the optimal soln. But the obj. items are not arranged in sorted order. Every item will have two options for choosing either to take it or not. If we take it → optimal soln. If we not take it → optimal soln.

Imp!

The probs. which are solved by Greedy approach ~~algo.~~ can also be solved by dynamic prog. But the reverse is not always true. There are some probs. which the dynamic prog. solves, but Greedy ~~algo.~~ approach does not provide the optimal soln.

But we study Greedy approach because it is very easy to solve. Hence, we first check if it can be solved by Greedy. If not, we go for dynamic approach.

Looking on the exe. time of algo., we classify ~~them~~ them as:

P, NP, NP complete, NP Hard

↓

Less exe. time algo.

Prime Merit
Page No.
Date

Scanned with CamScanner

20/7/20

Commu. delay analysis is only considered when the algo is executed in a multi comp. sys.

How do we find exe. time of algo.:

Given an algo.:

Least ~~considered~~ concerned with what the algo is doing. Only care on the finding of the exe. time.

Find: i) Exe. time of each statements (t_i) available in algo.

ii) No. of times (freq. count) particular st. gets executed (n_i)

$\therefore$ Total Exe. time: $\boxed{T_e = \sum_{i=1}^{k} t_i n_i}$

k = no. of diff. st.

Imp:
While considering the st., only consider the executable st. Non executable st. are not considered. (Eg.: all declaration st.) made for compiler, no declaration exists in exe. code

Algo. 1:

	Freq. Count
int i, j, a; ✗	0
a = 0 ✓ → not declaration, initialization	1
i = 0 ✓	1
for (j = 10; i < j; i++) ✓	11
a = a + i; ✓	10
return ( ); ✓	1
	$f(n)$ = 24

Prime Mark
Page No.
Date

Scanned with CamScanner

Algo. 2:

	Freq Count
void algo2 (int n)	
{	
int a[100], i;	0
i = 0	1
while (i < n)	n+1
a[i] += a[i] + 2	n
~~a[i]++~~	
return ();	1
}	$f(n) = 2n+3$

We will execute $(2n+3)$ diff. st. when n is an integer (i/p).

algo 3:

	Freq. Count
void algo3 (int flag)	
{	
if (flag) → executable statement	1
{	
for i = 1 to n do	n+1
S1;	n
}	
else → not executable st. ? compiler directive	
{	
for i = 1 to n^2 do	n^2+1
{	
S2;	n^2
S3;	n^2
}	
}	
}	

Prime Mart
Page No.
Date

Scanned with CamScanner

$$\therefore f(n) = \begin{cases} 2n+2, & flag = 1 \\ 3n^2+2 & flag = 0 \end{cases}$$

↓

if st. also have to be ~~exe~~ executed

algo4:

	freq. Count
int algo4 (int a, int b)	
{	
if (a == b)	1
return (a)	1 / 0
else if (a < b)	1 / 0
{	
for i = 1 to n do	n + 1
S1	n
return (1);	1
}	
else ×	
{	
for i = 1 to n^2 do	n^2+1
{	
S2	n^2
S3;	n^2
}	
return (2)	1
}	
}	

$$\therefore f(n) = \begin{cases} 2, & a = b \quad \checkmark \text{ best case} \\ 2n+4, & a < b \\ 3n^2+4, & a > b \quad \checkmark \text{ worst case} \end{cases}$$

Same algo. is having diff. f(n). So algo may not execute fixed no. of st. So exe. time is not fixed.

Prime Merit
Page No.
Date

Scanned with CamScanner

Constraints for determining exe. time of algo.

i) Algo. does not execute fixed no. of times. It varies within a prog. depending on i/p data.

Every for loop is rep. by if and goto.

⋮

return i;

i++ → though executable, but will not be considered since it is not executed.

↓

$f_n = 0$

3/8/20

simple statements → executed once → comp. $f(n) = 1$

func. call → treated sep.

Algo 5:

```
void algo5()
{
    for i = 1 to n do
        for j = 1 to n do
            S1;
    return ();
}
```

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

Algo 6:

```
void algo6()
{
    for i = 1 to n do
        for j = i to n do
            S1;
    return ();
}
```

Freq. Count
$n+1$
$\frac{n(n+1)}{2} + n$
$\frac{n(n+1)}{2}$
1

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

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

Prime Merit
Page No.
Date

Scanned with CamScanner

~~i = 1 : j = 1 to n+1~~ ~~i0 = S1~~

~~i = 2 : j = 2 to~~

i	j	S1
1	n+1	n
2	n	n-1
3	n-1	n-2
⋮	⋮	⋮
n	2	1

→ no. of executions.

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

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

When we call a func inside a prog., its freq. ~~coun~~ count will not necessarily be 1. It will dep. on the no. of times the func. will be called.

Eg)

```
void fun(int n)
{
    for i = 1 to n do
        S1;
    return ();
}
```

Freq. count

n+1
n
1

$g(n) = 2n+2$

```
void main()
{
    for j = 1 to m do
        fun(5);
    return ();
}
```

m+1
m(g(5))
1

$f(m) = m+1 + m(g(5)) + 1$

$= m+1 + m.12 + 1$

$= 13m + 2$

for j = 1 → g(5)

j = 2 → g(5)

⋮

j = m → g(5)

$g(5) = 12$

Prime Mark
Page No.
Date

Scanned with CamScanner

Freq. Count

Eg:
```
void fun(int a)
{
    for i = 1 to n do        n+1
        S1;                  n
    return();                1
}
```
$g(n) = 2n+2$

```
void main()
{
    for j=1 to m do          m+1
        fun(j)               m(g(j))   j will vary from 1 to m
    return();                1
}
```
$f(m) = m+1 + m(g(j)) + 1$

will not be multiplied by m

$= m+2 + (g(1) + g(2) + \dots + g(m))$

$= m+2 + (4+6+8+\dots+2m+2)$

$= m + (2(1+2+3+\dots+(m+1)))$

$= m + (m+1)(m+2)$

$= m^2 + 4m + 2$

for $j = 1 \rightarrow fun(1)$

$j = 2 \rightarrow fun(2)$

$j = 3 \rightarrow fun(3)$

$\vdots$

$j = m \rightarrow fun(m)$

$f = g(1) + g(2) + \dots + g(m)$

$g(1) = 4$

$g(2) = 6$

$\vdots$

~~Recurs~~ Recursive Functions:

Freq. Count

```
int r(int n)
{
    if n==1                  1
        return 1;            1
    else
        return (r(n-1) + n);  f(n-1)
}
```
$f(n) = f(n-1) + 2$

Prime Merit
Page No.
Date

Scanned with CamScanner

$f(1) = 2$

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

$\downarrow$

$n > 1$

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

freq. count

Eg:

```
int r(int n)
{
    if n == 1                                   1
        return 1;                               1
    else
        return (r(n-1) + r(n-2));               f(n-1) +
                                                f(n-2)
```

$\Sigma f(n) = f(n-1) + f(n-2) + 2$

$f(1) = 2$

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

$\downarrow$

$n > 1$

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

Prime Merit
Page No.
Date

Scanned with CamScanner

4/8/20

Factors that affect/limit to obtain exact exe. time of algo.:

There are the factors due to which we will not be able to calc. the exact exe. time of an algo.
~~Even if we know~~ In a prog. the ~~freq. counts~~ every st. in high level lang. will be translated to ~~an~~ assembly level lang. st.
~~In a prog.~~
The freq. count we calculated, was restricted to the no. of st. of high level lang.

Assuming that we know the freq. count of assembly level lang. st., every st. is simple and contains only one operation.

Eg: $A = B + C$ $\quad$ ~~$A = (B+C) * D$~~
$X = Y * Z$ $\quad$ $A = (B+C) * (D - F)$
$\rightarrow X$

Factors:

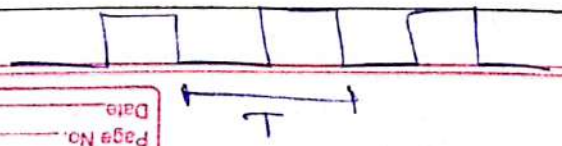
1) No. of clock cycles (T-states) for each op.:

The no. of clock cycles reqd. for each. op. is not known to us.

Eg.: $A = B + C$

By referring to some books, eg: Intel proc. book we may say that this op. takes 10 ~~clock cycles~~ T-states.

T-states: a single clock cycle
Every mach. is born. with a particular freq.



Prime Merit
Page No.
Date

Scanned with CamScanner

But when we write an algo, we do not know the type (arch.) of mach. it is to be executed. The same '+' ~~app~~ op. will take diff. T-states in diff. arch. comp. (proc.)

So the no. of clock cycles taken to execute an algo. is not known beforehand. So the exe. time of the algo. cannot be calculated.

Suppose, we fix the no. of T-states for all mach.

2) Freq. of machine: ~~any mach.~~ (proc.) ~~(not restricted to one)~~

→ It shouldn't be done because the algo. should be able to run on any proc. Suppose we fix the no. of T-states for all mach. (Eg. only Intel mach.) But we specifically do not know which mach. it is to be executed. The freq. of a mach. varies from one proc. to other.

If the freq. is not ~~known the~~ known, exact exe. time of each st. cannot be calc.

Exe. time of each st. cannot be determined unless and until we know the no. of T-states & time period for each T-state. These two ~~facto~~ para. varies from proc. to proc.

3) Programming Language:

Even if we fix the above two para., ~~we~~ when we write an algo., we do not know in which lang. we will write.

~~Also~~ The no. of assembly level lang. st. generated by C lang. may not be same ~~for~~ for a Pascal/Java lang. The same expn. in algorithmic lang., will have diff. assembly level lang. code which varies from prog. lang to lang.

Page No.
Date
Prime Mark

Scanned with CamScanner

4) Compiler:

If we assume that the algo. will be executed on a particular mach. in C lang. The same C prog. may be run using diff. C compiler. Eg. TurboC, Unix.

If no. of the compiler varies and diff. type of optimisations are implemented in a compiler, the same C lang code will produce diff. assembly level lang. code when compiled by diff. compilers.

Eg: a = a * 2
a = a << 1 } diff. code for '*' op.

faster than '*'

So when the compiler varies, the code generated will vary, and the exe. time will vary.

Hence, exact exe. time of an algo. is not known apriori, because of lack of prior knowledge of environment (arch., OS, compilers) where it would be executed.

Execution Time Analysis:

i) Apriori Analysis:

Much before exe., we estimate the exe time. Exact exe. time can never be calc.

Prime Merit
Page No.
Date

Scanned with CamScanner

→ Post exe.

ii) Posteriori Analysis:

During exe, we will measure the exe. time of an algo. We start measuring at the beg. of exe. and at the end of exe., we see the time and get the whole exe. time of the algo.

We get to know the exe. time after the algo. has completed its exe

We first go for posteriori analysis because we want to know the exact time. If it is not possible, then we estimate (apriori)

Usually the algos. take less than a second to execute unless there is a 'scanf()' st. — human delay.

OS has helps us in finding the exe. time — service of OS.
Linux gives has many ways, one of which is the func. call is called sys. call — gettime().

gettime() — returns the curr. sys. time

Eg.:
```
void toe()
{
    ts = gettime();
    st;
    te = gettime();
    tt = te - ts;
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

The time taken by an algo to execute is usually in μs, ns, s. So if gettime() returns the time in hrs, mins, s, the time for reqd. for exe. will show zero ($t_e = 0$). ~~So to get th~~

So to get the actual exe. time, we run the algo. for multiple times.
Eg.: pendulum expt.

Modified algo.:

```
void toe()
{
    ts = gettime();
    for i = 1 to 10 do    // count may be increased
        st.;
    te = (te - ts)/10;
}
```

Here, the 'for' st. will be a little overhead to the sys. But since prev. we got $t_e = 0$, and now we get $t_e \neq 0$, we ignore the overhead.

Still if we get $t_e = 0$, we increase the no. of count in 'for' loop.

However, this method will not work under certain circumstances.

Context Switches: The algo. may not necessarily be run on a uni-user sys.

Posteriori analysis works very well in a uni-user sys. where the CPU will only execute 1 prog. at a time. Eg.: Batch OS. (unless one prog. finishes its exe., the next prog. will not start its exe.)

This will not work in a multiprogramming environment due to concurrent exe. of multiple task. Eg.: Time sharing OS (after a part of exe., the CPU switches to another algo. - context switch) Eg: RR scheduling policy, priority scheduling.

So the context switching times and the time reqd to execute the next algo. will also be included in calc. the exe. time of an algo.

Hence, posteriori analysis fails to measure the exe. time of a prog. in a multi user environment due to concurrent exe. of multiple tasks.

Hence, we go for apriori analysis.

i) Apriori Analysis:

Here, we estimate the exe. time for large no. of i/ps. We perform a comparative analysis instead of calc. the exact exe. time.

Eg.: If we want to find out the fastest among an aeroplane, bike & human, in the first 1 min, human will be the fastest. In the first 10 mins, bike will be the fastest. But if we consider for a long period of time, aeroplane would be the fastest → correct ans.

Hence, we always do a comparative analysis for a large no. of i/ps.

Eg.: sorting → 10,000 ele.

matrix mult. → 100 by 100 matrices

We estimate the exe. time for a large no. of i/ps. using some asymptotic notations to make a comparitive analysis with other algos.

Asymptotic Notations:

Three main types: (mostly used) (though many more are there)

a) Big Oh (O) : ~~Were~~ Used as worst case analysis. Upper bound (maximum) notation on running time. (upper estimation)

b) Omega (Ω) : ~~Bes~~ Used as best case analysis. Lower bound (minimum) notation on running time. (lower estimation)

c) Theta (Θ) : ~~Avg~~ Used as avg. case analysis. Tightly bound notation on running time. (exact value)

avg. case → real life i/ps. or equally likely (random) i/ps.

Big Oh (O):

The func. $f(n)$ is $O(g(n))$, iff there exist 2 ~~pos~~ +ve constants c, n_0, such that

$f(n) \leq c\,g(n)$ for all $n \geq n_0$ (a higher value of n)

$f(n)$ → freq. count

larger value of i/p

~~shorter value~~ certain lower of i/p

While calc. upper bound, we do not bother about the contents of algo, how it is written, who has written. Only focus on freq. count of algo.

Prime Merit
Page No.
Date

Scanned with CamScanner

$f(n) \rightarrow$ given

$g(n) \rightarrow$ should not have any const. terms

c

$\rightarrow$ choose to prove

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

for higher i/ps. (n)

Eg: $f(n) = 2n + 3$, n = no. of i/ps.

Choose $g(n) = n$, $c = 3$

$\hookrightarrow$ value can be anything

n	$f(n)$ $(2n+3)$	$\leq$	$c\,g(n)$ $(3n)$	T/F
1	5		3	F
2	7		6	F
3 $\rightarrow n_0$	9		9	T
4	11		12	T
5	13		15	T

True for all bigger values of 'n'. (may not be 3, could have been 100)

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

Eg: $f(n) = 3n^2 + 4n + 6$

Choose $g(n) = n^2$, $c = 5$

$\hookrightarrow$ value can be anything

n	$f(n)$ $(3n^2+4n+6)$	$\leq$	$c\,g(n)$ $(5n^2)$	T/F
1	13		5	F
2	26		20	F
3 $\rightarrow n_0$	45		45	T
4	70		80	T
5	101		125	T

$\Rightarrow f(n) = O(n^2)$

Prime Mark
Page No.
Date

Scanned with CamScanner

On taking a bigger value of 'c', we would easily / quickly be satisfied with $f(n) \leq c\,g(n)$.

Eg. Choosing of $g(n)$:

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

i) Choose $c = 3$, $g(n) = n$

$\therefore 2n+3 \leq 3n \rightarrow$ True

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

ii) Choose $c = 3$, $g(n) = n^2$

$\therefore 2n+3 = 3n^2 \rightarrow$ True

$\Rightarrow f(n) = O(n^2)$

iii) Choose $c = 3$, $g(n) = n^3$

$\therefore 2n+3 = 3n^3 \rightarrow$ True

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

2) $f(n) = 3n^2 + 4n + 6$

i) Choose $c = 4$, $g(n) = n^2$

$\therefore 3n^2 + 4n + 6 \leq 4n^2$

$\Rightarrow f(n) = O(n^2)$

ii) Choose $c = 4$, $g(n) = n^3$

$\therefore 3n^2 + 4n + 6 \leq 4n^3$

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

iii) Choose $c = 80$, $g(n) = n$

$3n^2 + 4n + 6 \leq 80n \rightarrow$ False

↳ will not be true for all higher values of 'n'.

$\Rightarrow f(n) \neq O(n)$

Prime Merit
Page No.
Date

Scanned with CamScanner

Upper bound estimation means that the real value are not cross that value.

Upper bound estimation value should be the lower value for which the criteria ~~satisfies~~ satisfies. It must be the lower order $g(n)$ for which the given inequality satisfies.

So for $f(n) = 2n+3$,
$g(n) = n$

for $f(n) = 3n^2 + 4n + 6$,
$g(n) = n^2$

So $g(n)$ should be the lower order func. to satisfy $f(n) \leq c\,g(n)$ for all higher values of n.

for higher inputs,
$c \rightarrow$ any bigger const. → ~~choose~~ choose such a value that ~~satisfi~~ proves the inequality (not disproves it)
$g(n) \rightarrow$ lower order func

5/8/20

Eg: $f(n) = 10n^5 + 7n^4 + 3n^3 + n^2 + 100$
↳ order : n^5

given any polynomial, if we want to rep. it in Big Oh notation, then we should choose the higher order ~~value~~ polynomial present in the expn.

$\therefore f(n) \leq 11n^5$ for all higher values of n
$\Rightarrow f(n) = O(n^5)$

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg.: $f(n) = 10n^{50} + 7n^{40} + 3n^{30} + n^{20} + 100$

higher order term = n^{50} $\leq 11 n^{50}$ for all higher values of n

can be any value (100, 1000) → any bigger +ve constant

$\Rightarrow f(n) = O(n^{50})$

Eg.: $f(n) = 10n^{55} + 2^n + 3n^3 + n^2 + 100$

Exp. term will always be larger than any poly. terms for ~~value~~ of 'n'. higher values of 'n'. For all higher values ($n = 1000, 5000$), $2^n > n^{55}$.

$f(n) \leq 3 \cdot 2^n$ for all higher value of n

$\Rightarrow f(n) = O(2^n)$

When we find a comb. of a polynomial & an exp. term, we ~~will consider~~ find out the exp. term to be the larger one, for higher values of n.

Choose the higher order term from the freq. count expn. (for higher values of n) for g(n) in the expn. of Big-O notation.

Order of Growth:

Increasing order of growth:

$1, \log n, n, n \log n, n^2, n^3, \ldots, 2^n, 3^n, n^n$

If an expn. contains n and log n, we will choose $g(n) = n$. We don't bother about the const. (coeff.) attached with the higher order poly. in the expn. while calc. g(n).

Prime Merit
Page No.
Date

Scanned with CamScanner

Q) $f(n) = 10 \log n + \sqrt{n} + 3$
Find $g(n)$.

Soln) $\log n < n^{0.0000}$
$\Rightarrow \log n < n^{0.1}$

Eg. $n = 256$

$\sqrt{n} = 16 \qquad \log n = 8$

Eg.: $n = 1024$
$\log n = 10 \qquad \sqrt{n} >> \log n$

Eg.: $n = 2048$
$\log n = 11 \qquad \sqrt{n} >> \log n$

$\therefore$ $\log n$ is always smaller than any power of n, ~~Except~~ for all higher values of n.

$\therefore g(n) = \sqrt{n}$
$\Rightarrow f(n) = O(\sqrt{n})$ (Ans)

Q) $f(n) = 5 \log n + n^{0.1} + 3$

$\Rightarrow f(n) = O(n^{0.1})$

Q) Arrange in increasing order of growth:
n, $\log n$, $\log(\log n)$, $\sqrt{n} \log n$, $\sqrt{n} \log^2 n$, $(\log n)^{5 \log n}$, $2^{\log n}$

Prime Merit
Page No.
Date

Scanned with CamScanner

Q) Let $f(n) = 2^{2n}$. Is $f(n) = O(2^n)$?

Soln: (i) If $f(n) = O(2^n)$, it should be

$$f(n) \leq c \cdot 2^n$$

$$f(n) = 2^{2n} \leq c \cdot 2^n$$
$$\Rightarrow 2^n \cdot 2^n \leq c \cdot 2^n$$
$$\Rightarrow 2^n \leq c$$ → False for higher values of n

(ii) If $f(n) = 2^{O(n)}$

$$f(n) = 2^{2n} \leq 2^{3n}$$ $c \geq 2$ (any higher value)
$$\Rightarrow 2^{2n} \leq 2^{c \cdot n}$$
$$\Rightarrow 2^{2n} \leq 2^{O(n)}$$ (proved)

Omega (Ω) Notation:

The func. $f(n) = \Omega(g(n))$ if there exist 2 +ve constants c and n_0, such that $f(n) \geq c\,g(n)$ for all $n \geq n_0$.

Given $f(n)$,
choose $g(n)$ and c to prove the inequality for all higher values of n.
$\Rightarrow f(n) = \Omega(g(n))$

c → any smaller value → choose such a value that proves the inequality, not disprove it

can be 0.0001

Eg: $f(n) = 2n + 3$

choose $g(n) = n$, $c = 2$

n	$f(n) \geq cg(n)$		T/F
	$2n + 3$	$2n$	
1	5	2	T
2	7	4	T
3	9	6	T
4	11	8	T
5	13	10	T

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

Eg:) $f(n) = 3n^2 + 4n + 6$

Choose $g(n) = n^2$, $c = 3$

n	$f(n) \geq c\,g(n)$		T/F
	$3n^2+4n+6$	n^2	
1	13	3	T
2	26	12	T
3	45	27	T
4	70	48	T
5	101	75	T

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

Eg: $f(n) = 2n+3$

i) choose $g(n) = n$, $c = 2$

$2n + 3 \geq 2n$

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

Prime Merit
Page No.
Date

Scanned with CamScanner

ii) Choose $g(n) = n^2$, $c = 0.1$

$\therefore 2n + 3 \geq 0.1n^2 \longrightarrow$ False

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

iii) Choose $g(n) = n^3$, $c = 0.2$

$\therefore 2n+3 \geq 0.2n^3 \longrightarrow$ False

$\Rightarrow f(n) \neq \Omega(n^3)$

Eg: $f(n) = 3n^2 + 4n + 6$

i) Choose $g(n) = n^2$, $c = 3$

$\therefore 3n^2 + 4n + 6 \geq 3n^2$

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

ii) Choose $g(n) = 1$, $c = 4$

$\therefore 3n^2 + 4n + 6 \geq 4 \cdot 1 \longrightarrow$ False

$\Rightarrow f(n) = \Omega(1)$

iii) Choose $g(n) = n$, $c = 3$

$\therefore 3n^2 + 4n + 6 \geq 3n$

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

'Ω' is used for lower bound estimation. So when we go for lower bound estimations, we should go for a higher value which is close to the real value.

(lower value $\leq$ real value $\leq$ upper value)

Prime Marit
Page No.
Date

Scanned with CamScanner

For Ω notations, for higher i/ps.,

$c \rightarrow$ any smaller value that satisfies the inequality

$g(n) \rightarrow$ higher order func. to satisfy the inequality

Q1) $f(n) = 10n^5 + 7n^4 + 3n^3 + n^2 + 100$

For any given freq. count poly., to rep. it in omega notation, the higher order poly. term will be chosen for $g(n)$.

$\therefore 10n^5 + 7n^4 + 3n^3 + n^2 + 100 \geq 9n^5$ for all higher values of n

$\Rightarrow f(n) = \Omega(n^5)$

Q2) $f(n) = 10n^{50} + 7n^{40} + 3n^{30} + n^{20} + 100$
$\geq 10n^{50}$ for all higher values of n

$\Rightarrow f(n) = \Omega(n^{50})$

Q3) $f(n) = 10n^{55} + 2^n + 3n^3 + n^2 + 100$
$\geq 1.2^n$ for all higher values of 'n'.

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

Choose the higher order term (for higher values) of 'n' for $g(n)$ for both omega and Big-Oh notation

Prime Merit
Page No.
Date

Scanned with CamScanner

10/8/20

Theta Notation:

The func., $f(n) = \Theta(g(n))$ iff there exist a +ve constants c_1, c_2 and n_0, such that
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

$\Rightarrow f(n) = \Theta(n)$ (diff. consts. $\rightarrow c_1, c_2$ but same $g(n)$)

Eg. $f(n) = 2n+3$. Choose $g(n) = n$, $c_1 = 2$, $c_2 = 3$

$c_1 \rightarrow$ as small as possible
$c_2 \rightarrow$ as large as possible

$c_1 g(n)$		$f(n)$		$c_2 g(n)$
$2n$	$\leq$	$2n+3$	$\leq$	$3n$

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

Eg.: $f(n) = 3n^2 + 4n + 5$. Choose $g(n) = n^2$, $c_1 = 3$, $c_2 = 4$

$c_1 g(n)$		$f(n)$		$c_2 g(n)$
$3n^2$	$\leq$	$3n^2+4n+5$	$\leq$	$4n^2$

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

If not satisfied, replace with $40n^2$/ $400n^2$.

If $g(n) = n^3 \rightarrow$ first cond. unsatisfied
If $g(n) = n \rightarrow$ second " "

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Eg: $f(n) = 10n^5 + 7n^4 + 3n^3 + n^2 + 100$~~

$f(n) \leq c_2 g(n) \rightarrow f(n) = O(g(n))$

$f(n) \geq c_1 g(n) \rightarrow f(n) = \Omega(g(n))$

$\Rightarrow \boxed{\begin{array}{l} f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ \&} \\ f(n) = \Omega(g(n)) \end{array}}$

Eg: $f(n) = 10n^5 + 7n^4 + 3n^3 + n^2 + 100$

Let $g(n) = n^5$, $c_1 = 9$, $c_2 = 11$

$\therefore 9n^5 \leq 10n^5 + 7n^4 + 3n^3 + n^2 + 100 \leq 11n^5$, for all higher values of n.

$\Rightarrow f(n) = \Theta(n^5)$

Eg: $f(n) = 10n^{55} + 2^n + 3n^3 + n^2 + 100$

Let $g(n) = 2^n$, $c_1 = 1$, $c_2 = 3$

$\therefore 1.2^n \leq 10n^{55} + 2^n + 3n^3 + n^2 + 100 \leq 3.2^n$, for all higher values of n

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

Choose the higher order term (for higher values of n) present in the expn. as $g(n)$ for theta notations.

Given any $f(n)$, always choose the higher order term for $g(n)$ for Ω, O, Θ notations.

Prime Albert
Page No.
Date

Scanned with CamScanner

Diff. b/w Ω, O, Θ notations:

For algo. A1,

$$\left.\begin{aligned} f_1(n) &= 8n+1 \\ f_2(n) &= 6n+5 \\ f_3(n) &= 2n+9 \end{aligned}\right] \Rightarrow O(n), \Theta(n)$$

~~Rep. in big on notation.~~

Despite having diff. $f(n)$, $O(n)$ is same for all.

Suppose there is a var. x in A1,

$f_1(n) = 8n+1 \rightarrow x = 0$
$f_2(n) = 6n+5 \rightarrow x = +ve$
$f_3(n) = 2n+9 \rightarrow x = -ve$

So dep. on the value of x, the $f(n)$ value varies. Even the algo has diff. $f(n)$, but same asymptotic value, we will not rep. the algo in O or Ω. The algo. is more tightly bound and hence, it is rep. using Θ notation.

For algo. A2,

$$\left.\begin{aligned} f_1(n) &= 6n+5 \\ f_2(n) &= 2n+9 \\ f_3(n) &= n^2+1 \end{aligned}\right] \rightarrow$$ Order of $f(n)$ differs in each $f(n)$. (n, n^2)

Whenever the algo. will have diff. asymptotic cases, the algo. exe. is loosely bound & hence, rep. in lower & upper bound notation.

$f(n) = \Omega(n), O(n^2)$
(Since exe. of algo. varies from n to n^2).

Prime Merit
Page No.
Date

Scanned with CamScanner

for algo. A3,

$f_1(n) = 2n+1$ → $\Omega(n)$,
$f_2(n) = 3n^2+2$ → $O(n^3)$
$f_3(n) = n^3+1$ (n^2 → no notation)

When an algo. has diff. asymptotic values for diff. cases:

tightly bound → Θ
loosely bound → Ω, O

Observations:

1) Avoid counting simple st., if they are executed for constant no. of times. (If st. are " for const. no. of times ~~for any~~ → $O(1)$)

If st. are executed for 'n' no. of times → $O(n)$ because 'n' is not a constant.

$1+1+1+\dots+1$ (100 times) $= 1$
$1+1+1+\dots+1$ (n times) $= n$
$n+n = n$
$n+n+\dots+n$ (100 times) $= n$
$n+n+\dots+n$ (n times) $= n^2$

2) Do not count the inner st. of a loop/if st., if they are simple and no. of st. are const.

if (cond.) → 1
<st> → 0/1
↓
simple (no loops)

→ 'if' st. always has higher f(n) than ~~the~~ the sts. within it. Same with 'for'.

So only consider if st, not the st inside it.

Prime Merit
Page No.
Date

Scanned with CamScanner

3) Eg. S1; → n^2 times
S2; → $< n^2$ times → do not know exact value.

In such a case, we do not need to calc. the ~~We do no~~ $f(n)$ for ~~stm~~ S2.

Stop counting the st., if you feel their freq would be asymptotically less than some prev. st.

Time Complexities of Popular algos.:

1) Linear Search:

No. of comparison will vary from ~~to~~ 1 to n.
Min. = 1 → Best Case = $\Omega(1)$
Max. = n → Worst Case = $O(n)$

If an algo. is loosely bound, we can never rep. it using Θ notation. It is used only when algo. is tightly bound which has only 1 case which has only 1 asymptotic value.

Average Case: Item can be present in any loc. (has equal prob.)

Whenever I run the algo., I assume that the ele is present in the next pos., if the curr. pos does not contain the reqd. ele.

If I run it n times, ~~no. of comp. ab~~
no. of comparisons = $1+2+3+\dots+n$
$= \frac{n(n+1)}{2}$

Prime Merit
Page No.
Date

Scanned with CamScanner

If algo. is run 'n' times = n^2
If algo. is run once = n

Here, avg case analysis is very imp because mostly we consider the avg case exe. time to be the exe. time of an algo.

∴ Average case = $O(n)$
(If avg. case = 1, → $\Omega(1)$, not $O(1)$ because best case is 1 and worst is 'n').

2) Finding Largest: (Unsorted Array)

No. of comparisons = exactly 'n'
→ tightly bound

∴ Best case = $\Theta(n)$
Worst case = $\Theta(n)$
Avg. case = $\Theta(n)$
→ If best & worst cases are same, avg has to be same.

3) Linear Search: (unavailable item)

No. of comparisons = exactly 'n'
→ tightly bound

∴ Best case = $\Theta(n)$
Worst case = $\Theta(n)$
Avg. case = $\Theta(n)$

Q) There is a task for 3 groups, G1, G2, G3.
G1 → $\Omega(n)$ → best case
G2 → $O(n)$ → worst case
G3 → $\Theta(n)$ → avg. case
Which algo is a better one?

Ans) G2 (best case can be $\Omega(1)$, & worst case can not exceed $O(n)$)
[G1 → worst case can be $O(n^2)$ / $O(n^3)$
G3 → best case can never be less than $\Theta(n)$]

Page No.
Date

Scanned with CamScanner

11/8/20

Solving Recurrences:

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

n = no. of i/ps.

$T(n)$ → is a recurrence.

RHS will contain a diminishing term of n than compared to LHS ($n/2$, can be $n-1$)

c = const no. of st. or 1.

A recurrence is an eqn. that describes a func. in terms of its value on lower & smaller func.

Eg.: LHS: $T(n)$

RHS: $T\left(\frac{n}{2}\right)$ / $T\left(\frac{n}{3}\right)$ / $T(n-1)$

Eg.: $s(n) = \begin{cases} 0, & n = 0 \\ c + s(n-1), & n > 0 \end{cases}$

RHS can never be $s(2n)$ or $s(3n)$ if LHS is $s(n)$. RHS will always be a diminishing term.

Eg: $s(n) = \begin{cases} 0, & n = 0 \\ n + s(n-1), & n > 0 \end{cases}$

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

Eg.: $T(n) = \begin{cases} c, & n = 1 \\ aT\left(\frac{n}{b}\right) + cn, & n > 1 \end{cases}$

a, b, c → constants.

Prime Merit
Page No.
Date

Scanned with CamScanner

Recursive funcs. $f(n) \rightarrow$ recurrence func.

Methods to Solve Recurrences:

1) Subs. Method
2) Iteration Method
3) Master Method

2) Iteration Method:

i) Expand the recurrence.
ii) Use some algebra to express as a summation
iii) Evaluate the summation.

Eg: $s(n) = \begin{cases} 0, & n = 0 \\ c + s(n-1), & n > 0 \end{cases}$

Soln: $s(n) = c + s(n-1)$

Given $s(0) = 0$.
Replace $s(n-1)$ by its value.

$\therefore s(n-1) = c + s((n-1)-1)$
$= c + s(n-2)$

$$\begin{aligned} \therefore s(n) &= c + s(n-1) \\ &= c + c + s(n-2) \\ &= 2c + s(n-2) \\ &= 2c + c + s(n-3) \\ &= 3c + c + s(n-4) \\ &\vdots \\ &= kc + s(n-k), \quad k = +ve \text{ const.} \\ &\qquad = 1, 2, \ldots \end{aligned}$$

Prime Merit
Page No.
Date

Scanned with CamScanner

$\therefore$ For $k = n$,

$$s(n) = cn + s(n-n)$$
$$= cn + s(0)$$
$$= cn + 0$$
$$= cn$$

$\Rightarrow s(n)$ will be of order 'n'.

Eg: $s(n) = \begin{cases} 0 & , n = 0 \\ n + s(n-1) & , n > 0 \end{cases}$

~~Soln.: $s(n) = n + s(n-1)$~~
~~$= n + n-1 + s(n-2)$~~
~~$=$~~

Soln.: $s(n) = n + s(n-1)$.

Given $s(0) = 0$
Replace $s(n-1)$ by its value.

$$\therefore s(n-1) = (n-1) + s((n-1)-1)$$
$$= n-1 + s(n-2)$$

$$\therefore s(n) = n + s(n-1)$$
$$= n + n-1 + s(n-2)$$
$$= n + n-1 + n-2 + s(n-3)$$
$$= n + n-1 + n-2 + n-3 + s(n-4)$$
$$\vdots$$

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

$k =$ +ve const.

For $k = n$,

$$s(n) = n + n-1 + n-2 + \dots + n-(n-1) + s(n-n)$$
$$= n + n-1 + n-2 + \dots + 1 + s(0)$$
$$= n + n-1 + n-2 + \dots + 1 + 0$$
$$= 1 + 2 + 3 + \dots + n-2 + n-1 + n$$
$$= \frac{n(n+1)}{2}$$

$\Rightarrow s(n)$ is of order 'n^2'.

Prime Merit
Page No.
Date

Scanned with CamScanner

eg.:

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

~~Soln.: Given T(1)~~

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

Replace $T\left(\frac{n}{2}\right)$ by its value.

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

$$= 4T\left(\frac{n}{4}\right) + c(1+2)$$

$$= 4\left(2T\left(\frac{n}{8}\right) + c\right) + c(1+2)$$

$$= 8T\left(\frac{n}{8}\right) + c(1+2+4)$$

$$\vdots$$

$$\Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1})$$

(assuming $n = 2^k$, i.e., $k = \log n$
where ~~(k a~~ is a +ve integer)

We are calc. the time for $n = 2, 4, 8, 16, 32, \dots$
For $n = 20$, time ~~complexity~~ ~~time~~ will be ~~same as~~ less than as for 32.

$$\Rightarrow T(n) = 2^k T\left(\frac{2^k}{2^k}\right) + c(2^k - 1) \quad \left[2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1\right]$$

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

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

$$= (2n-1)c$$

$\Rightarrow T(n)$ is of order 'n'.

Prime Merit
Page No.
Date

Scanned with CamScanner

12/8/20

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

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

Replace $T\left(\frac{n}{2}\right)$.

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

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

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

$= 8T\left(\frac{n}{8}\right) + 3n$

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

$\vdots$

$\Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$, k = +ve integ

(When $n = 2^k$, $\Rightarrow k = \log_2 n$

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

$\cancel{= 2^k T(1) + k \cdot 2^k}$

$\cancel{= 2^k \cdot 1 + k \cdot 2^k}$

$\cancel{= 2^k}$

$= nT(1) + nk$

$= n \cdot 1 + n\log_2 n$

$= n + n\log_2 n$

$\Rightarrow T(n)$ is of order $(n \log n)$.

Prime Merit
Page No.
Date

Scanned with CamScanner

Unless we know whether the value of $T(n)$ is loosely or tightly bound, we cannot use the notations O, Ω or Θ.

Q1) $T(n) = \begin{cases} 3T\left(\frac{n}{3}\right) + n & , n > 3 \\ 1 & , n = 3 \end{cases}$

Soln.: $T(n) = 3T\left(\frac{n}{3}\right) + n$

$$\Rightarrow T(n) = 3\left(3T\left(\frac{n}{9}\right) + \frac{n}{3}\right) + n$$

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

$$= 9\left(3T\left(\frac{n}{27}\right) + \frac{n}{9}\right) + 2n$$

$$= 27T\left(\frac{n}{27}\right) + 3n$$

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

$$\vdots$$

$$\Rightarrow T(n) = 3^k T\left(\frac{n}{3^k}\right) + kn \quad , k = +ve \text{ integer}$$

Assume $n = 3^k$, $\Rightarrow k = \log_3 n$

$$\Rightarrow T(n) = nT\left(\frac{3^k}{3^k}\right) + kn$$

$$= nT(1) + n\log_3 n$$

$$\cancel{= n + n\log_3 n}$$

~~Assuming~~ $\cancel{n = 3^{k-1}}$ $\cancel{, \Rightarrow k-1 = \log_3 n}$

$$\cancel{\Rightarrow T(n) = 3^{k-1} T\left(\frac{n}{3^k}\right) + (k-1)n}$$

Prime Mark
Page No.
Date

Scanned with CamScanner

~~Assuming $n = 3^{k-1}$, $\Rightarrow k-1 = \log_3 n$~~

$$\cancel{\therefore T(n) = 3^k T\left(\frac{3^{k-1}}{3^k}\right) +}$$

If $T(n)$ satisfies for k, then it must satisfy for $k-1$.

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

Since, $n = 3^k$, $\therefore 3^{k-1} = \frac{n}{3}$

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

$$= \frac{n}{3} T(3) + n(k-1)$$

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

$$= \frac{n}{3} + n(\log_3 n - 1)$$

$\therefore T(n)$ is of order $n \log_3 n$.

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

If $n = 3^k$,

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

RHS contains a diminishing term that goes on ~~decrea~~ diminishing (3^{k-1}, 3^{k-2}, 3^{k-3}, ...)

Always assume 'n' such that the RHS contain a diminishing term (like $k-1$).

Prime Merit
Page No.
Date.

Scanned with CamScanner

The value of 'n' can be ~~anything~~ any +ve integer (1, 2, 3, ...) but when the RHS contains terms like '$\frac{n}{2}$', we assume $n = 2^k$, so as to get a diminishing term (2^{k-1}).

Q2) $T(n) = \begin{cases} T\left(\frac{3n}{4}\right) + 1 & , n > 1 \\ 1 & , n = 1 \end{cases}$

<u>Soln.:</u> $T(n) = T\left(\frac{3n}{4}\right) + 1$

$= T\left(\frac{9n}{16}\right) + 1 + 1$

$= T\left(\frac{27n}{64}\right) + 1 + 1 + 1$

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

$\vdots$

$\Rightarrow T(n) = T\left(\left(\frac{3}{4}\right)^k n\right) + k$

~~Assuming $n = \left(\frac{4}{3}\right)^k$.~~

~~$\therefore T(n) = T\left(\left(\frac{3}{4}\right)^k \cdot \left(\frac{4}{3}\right)^k\right) + k$~~

~~$= T($~~

$\Rightarrow T(n) = T\left(\frac{n}{(4/3)^k}\right) + k$

Prime Merit
Page No.
Date

Scanned with CamScanner

Assuming $n = \left(\frac{4}{3}\right)^k$,

$\therefore \cancel{T(n) = T(1}$

$\therefore T(n) = T(1) + \log_{4/3} n$

$\therefore T(n)$ is of order $\log_{4/3} n$

Q3) $T(n) = \begin{cases} T(\sqrt{n}) + \log n & , n > 4 \\ 1 & , n = 4 \end{cases}$

Soln.: $T(n) = T(\sqrt{n}) + \log n$

$\Rightarrow T(n) = T(n^{1/4}) + \log n^{1/2} + \log n$

$= T(n^{1/4}) + \frac{1}{2}\log n + \log n$

$= T(n^{1/8}) + \log n^{1/4} + \frac{1}{2}\log n + \log n$

~~{E~~

~~$\Rightarrow T(n) = T(n^{1/2^k}) +$~~

$\Rightarrow T(n) = T(n^{1/8}) + \frac{1}{4}\log n + \frac{1}{2}\log n + \log n$

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

~~$\therefore T(n) =$~~

$\vdots$

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

Scanned with CamScanner

Assuming $n = 2^{2^k}$, $k > 0$

$\therefore \sqrt{n} = \left(2^{2^k}\right)^{1/2}$
$= 2^{2^k/2}$
$= 2^{2^{k-1}}$

If $T(\sqrt[3]{n})$,
$n = 3^{3^k}$

If $T(n^{1/5})$,
$n = 5^{5^k}$

~~If $n = 4$, $\Rightarrow 2^2 = n$~~

If $n = 4$,
$\Rightarrow n = 2^{2^1}$

and $\sqrt{n} = 2$
$\Rightarrow \sqrt{n} = 2^{2^0}$

If $n = 16$, and $\sqrt{n} = 4$,
$\Rightarrow n = 2^{2^2}$ $\Rightarrow \sqrt{n} = 2^{2^1}$

If $n = 256$ and $\sqrt{n} = 16$
$\Rightarrow n = 2^{2^3}$ $\Rightarrow \sqrt{n} = 2^{2^2}$

~~when $n = 2^{2^k}$~~

when $n = 2^{2^k}$, $\sqrt{n} = 2^{2^{k-1}}$

$\therefore T(n) = T\left(\left(2^{2^k}\right)^{1/2^k}\right)$

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

Try getting a diminishing term like $(k-1)$, $(k-2)$ in RHS not $(k/2)$, $(k/3)$, $(\sqrt{k})$. Result must be in terms of n (not k) if ques. is given in 'n'.

Prime Merit
Page No.
Date

Scanned with CamScanner

17/8/20

3) Master Theorem:

Gives soln. very quickly w/o any mathematical soln. Based on divide and conquer algo.

~~The prob. of size 'n' must be divided into a no. of subproblems and those where each of size 'a'.~~

The prob. of size 'n' is divided into 'a' no. of subprob., where each subprob. is of size '$\frac{n}{b}$'.

We solve each of the subprob. and conquer the subsoln. to get the final soln. In conquering the final soln, we may require another time '$f(n)$'.

$f(n)$: work to ~~dive~~ divide the prob. + combine the solved subprobs.

Eg: $T(n)$

Master Th. may not be ~~applie~~ applicable for all type of recurrences. If the ~~recu~~ recurrence is in the fou. form, we can appy th theorem.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$T(n)$ → soln. to final prob.

a → no. of subprob.

$\frac{n}{b}$ → size of each subprob.

Prime Merit
Page No.
Date

Scanned with CamScanner

Solving for master theorem:

Step 1) ~~Compare~~ from the given $f(n)$, compute $n^{\log_b a}$.

$a, b \rightarrow$ +ve constants

i) If $n^{\log_b a} > f(n)$:

Eg: $n^{\log_b a} = n^3$

$f(n) = n^2$

then, $T(n)$ can be rep. as $\Theta\left(n^{\log_b a}\right)$.

Whichever value is larger, we will go for that value.

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \quad \varepsilon > 0$$

Eg: $f(n) = n^{2.5}$

$= O(n^3)$

$= O(n^{3 - 0.5})$

ii) If $n^{\log_b a} = f(n)$:

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

$$= \Theta(f(n) \cdot \log n)$$

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

iii) If $n^{\log_b a} < f(n)$:

$f(n)$ is larger polynomially than $n^{\log_b a}$ (Not assymptotically). Both ~~terms~~ $f(n)$ & $n^{\log_b a}$ should contain polynomial terms.

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg: $\overset{\nearrow n^{\log_b a}}{n^2}$, $\overset{\nearrow f(n)}{n^2 \log n}$

larger : $n^2 \log n$

They are not polynomially related because 'n^2' cancels from both the terms. If the polynomial term cancels, then we cannot relate them polynomially.

~~Hence~~ Here, $f(n)$ is larger asymptotically, not polynomially.

If the poly. terms cancel, we cannot apply Master theorem to solve.

Eg: $f(n) = n^3$

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

Here, $f(n)$ is polynomially larger than $n^{\log_b a}$. $\rightarrow$ Master theorem can be applied.

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

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ and

$a f\left(\frac{n}{b}\right) < c f(n)$ for large n

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

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

$$\left.\begin{aligned} T(n) &= \Theta\left(n^{\log_b a}\right) \\ f(n) &= O\left(n^{\log_b a - \varepsilon}\right) \end{aligned}\right\} \begin{aligned} &\text{for } n^{\log_b a} > f(n) \\ &\varepsilon > 0 \end{aligned}$$

$$\left.\begin{aligned} T(n) &= \Theta\left(n^{\log_b a} \cdot \log n\right) \\ f(n) &= \Theta\left(n^{\log_b a}\right) \end{aligned}\right\} \text{for } n^{\log_b a} = f(n)$$

$$\left.\begin{aligned} T(n) &= \Theta\left(f(n)\right) \\ f(n) &= \Omega\left(n^{\log_b a + \varepsilon}\right) \\ &\text{and} \\ a f\left(\frac{n}{b}\right) &< c f(n) \\ &\text{for large } n \end{aligned}\right\} \begin{aligned} &\text{for } n^{\log_b a} < f(n) \\ &\varepsilon > 0, \\ &c < 1 \end{aligned}$$

Eg: $T(n) = 9T\left(\frac{n}{3}\right) + n$

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

$\therefore n^{\log_3 9} = n^2$ | $f(n) = O\left(n^{\log_3 9 - \varepsilon}\right)$

$\downarrow$ | $\Rightarrow n = O\left(n^{2-1}\right)$

$n^{\log_b a}$ | $\Rightarrow \varepsilon = 1$

$\Rightarrow n^{\log_b a} > f(n)$

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

Given this recurrence, we may also use O/Ω instead of Θ. Asymptotically we write Θ, but when we solve recurrences, depending upon the exe. time of the algo., we may also rep. it using O or Ω.

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg: Q1) $T(n) = 4T\left(\frac{n}{2}\right) + n$

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

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

$$\Rightarrow n^{\log_b a} > f(n)$$

$$\therefore f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$
$$\Rightarrow n = O\left(n^{2-1}\right)$$
$$\Rightarrow \varepsilon = 1$$

$$\therefore T(n) = \Theta(n^2) \quad \text{(Ans)}$$

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

Soln.: $a = 4, \quad b = 2, \quad f(n) = n^2$

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

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

$$\therefore T(n) = (n^2 \log n) \quad \text{(Ans)}$$

Here, if $f(n) = n^2 \log n$,

Master Th. couldn't be applied

Prime Merit
Page No.
Date

Scanned with CamScanner

Q3) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

Soln.: $a = 4, \quad b = 2, \quad f(n) = n^3$

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

$\therefore f(n) > n^{\log_b a}$

$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$

$\Rightarrow n^3 = \Omega\left(n^{2+1}\right), \quad \varepsilon = 1$

and $4\left(\frac{n}{2}\right)^3 < cn^3$

$\Rightarrow \frac{4n^3}{8} < cn^3$

$\Rightarrow \frac{n^3}{2} < cn^3, \quad c = \frac{1}{2}$

$\therefore T(n) = \Theta(n^3)$ (Ans)

Q4) $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$

Soln. $a = 4, \quad b = 2, \quad f(n) = \frac{n^2}{\log n}$

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

Here, master th. is not applicable because n^2 and $\frac{n^2}{\log n}$ are not poly. related.

Prime Merit
Page No.
Date

Scanned with CamScanner

Q5) $T(n) = 4T\left(\frac{n}{5}\right) + 5n$

Soln. $a = 4$, $b = 5$, $f(n) = 5n$

$n^{\log_b a} = n^{\log_5 4}$

$\cancel{\Rightarrow n^{\log_5 a}}$

$\Rightarrow \log_5 4 < 1$

$\cancel{\Rightarrow n^{\log_5 4}}$

$\therefore f(n) > n^{\log_b a}$ $(n > n^{0.\ldots})$

$\Rightarrow T(n) = \Theta(n)$ (Ans)

Q6) $T(n) = 5T\left(\frac{n}{4}\right) + 4n$

Soln. $a = 5$, $b = 4$, $f(n) = n$

$n^{\log_4 5} > 1$

$\therefore n^{\log_b a} > f(n)$

$\cancel{\Rightarrow T(n) = \Theta(\log_4 5)}$

$\Rightarrow T(n) = \Theta(n^{\log_4 5})$ (Ans)

~~Q7) T(n)~~ If $f(n) = n^2$,

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

Prime Merit
Page No.
Date

Scanned with CamScanner

Q7) $T(n) = 25T\left(\frac{n}{5}\right) + n^2$

Soln. $a = 25$, $b = 5$, $f(n) = n^2$

$n^{\log_5 25} = n^2$

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

$\Rightarrow T(n) = \Theta(n^2 \log n)$ (Ans)

If $T(n) = 24T\left(\frac{n}{5}\right)$

$\therefore n^{\log_5 24} > 1$ but < 2

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

If $T(n) = 26T\left(\frac{n}{5}\right)$

$\therefore n^{\log_5 26} > 2$

$\therefore T(n) = \Theta(n^{\log_5 26})$

Q8) $T(n) = 4T(\sqrt{n}) + \log n^5$

Soln. $a = 4$, $b = 1$, $f(n) = \log n^5$

Here, master th. cannot be applied because it is not in the form $aT\left(\frac{n}{b}\right)$. ($n$ is not present, $\sqrt{n}$ is present).

By applying other methods,

$T(n) = \Theta(\log n^5)$

Prime Merit
Page No.
Date

Scanned with CamScanner

Q9) $T(n) = 16T\left(\frac{n}{4}\right) + n^2 \log n$

Soln. $a = 16$, $b = 4$, $f(n) = n^2 \log n$

$n^{\log_4 16} = n^2$

Here, master th. cannot be applied because n^2 and $n^2 \log n$ are not poly. related.

i) If $f(n) = n^3 \log n$

$\Rightarrow T(n) = \Theta(n^3 \log n)$

ii) If $T(n) = 14T\left(\frac{n}{4}\right)$

$n^{\log_4 14} > 1$ but < 2

$\therefore T(n) = \Theta(n^2 \log n)$

iii) If $T(n) = 16T\left(\frac{n}{2}\right) + n^2 \log n$

$n^{\log_2 16} = n^4$

$\therefore T(n) = \Theta(n^4)$

Q10) $T(n) = T(n-1) + \log n$

Soln. $a = 1$, $b = 1$, $f(n) = \log n$

Here, master th. is not applicable because $T(n)$ is not in the form $aT\left(\frac{n}{b}\right)$.
(n is not present, $(n-1)$ is present).

Prime Merit
Page No.
Date

Scanned with CamScanner

Recursion Tree:

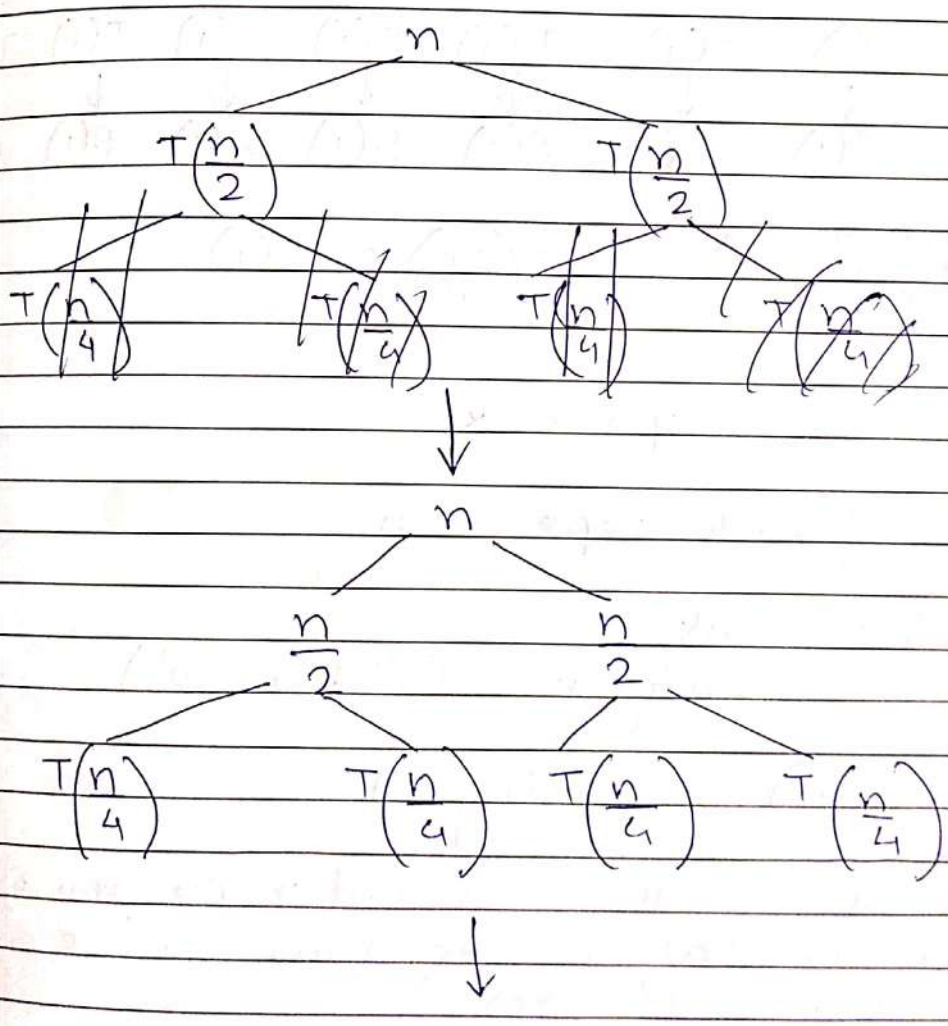
This is another method of solving recurrences.

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

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

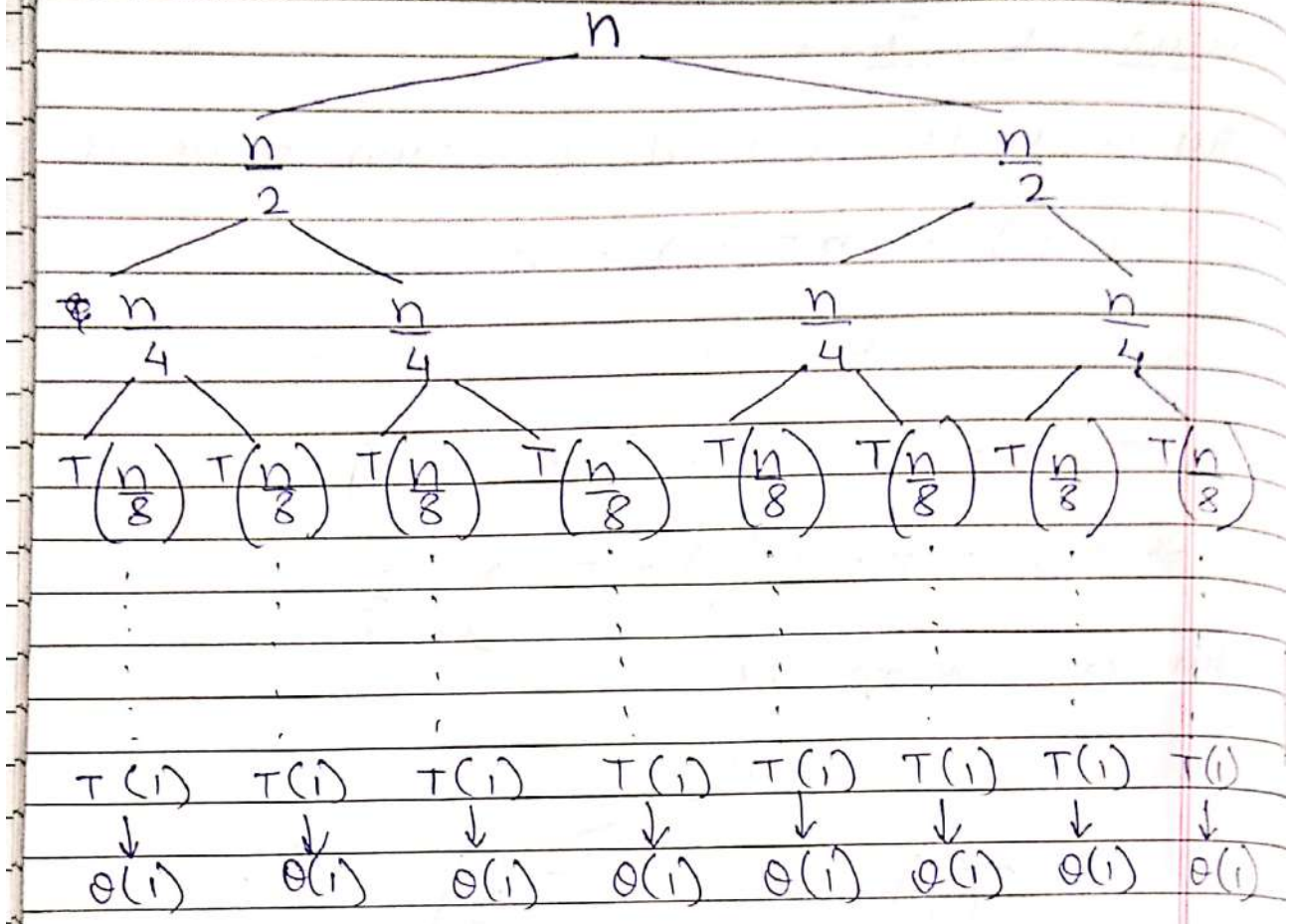
$\Rightarrow T(n) = n + T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + \frac{n}{2}$

This can be rep. as:



Prime Merit
Page No.
Date

Scanned with CamScanner



~~When~~ Assuming $n = 2^k$, $T\left(\frac{n}{n}\right) = T(1)$

$\therefore$ No. of steps $= k$

Sum of each step $= n$

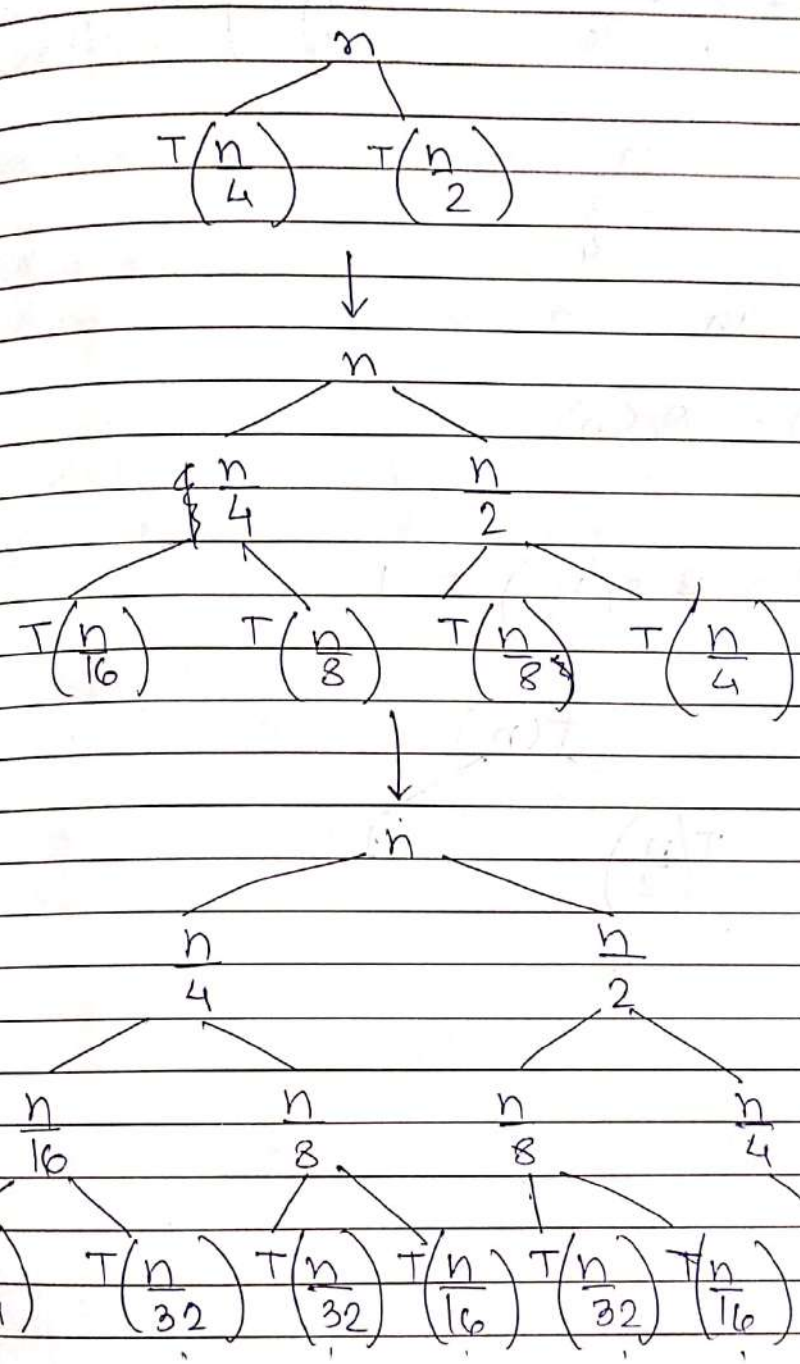
$\therefore$ Sum $= nk$

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

$\Rightarrow T(n) = \theta(n \log n)$

(when $n = 90$, it is not in the form of 2^k. So in that case, exe. time will be less than 2^7)

Q1) $T(n) = \begin{cases} T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2 & , n > 0 \\ 1 & , n = 1 \end{cases}$



Let no. of steps = k

Sum of each step = $\left(\frac{3}{4}\right)^k n$

$\frac{n}{16} + \frac{n}{16} + \frac{n}{32} + \frac{n}{8} + \frac{n}{8} + \frac{n}{64}$

$\frac{n}{8} + \frac{n}{4} + \frac{n}{32} + \frac{n}{64}$

$\frac{8n + 16n + 2n + n}{64}$ or $\frac{27}{64}n$

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

$\frac{n^2 + n}{16}$ $\frac{9n}{16}$ $\frac{n}{4}$ $\frac{3n}{4}$

Prime Merit
Page No.
Date

Scanned with CamScanner

$$\therefore \text{Sum} = n + \frac{3n}{4} + \frac{9n}{16} + \frac{27n}{64} + \dots + \left(\frac{3}{4}\right)^k n$$

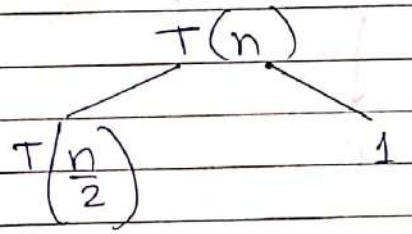
$$= \left(1 + \frac{3}{4} + \frac{9}{16} + \dots + \left(\frac{3}{4}\right)^k\right) n$$

$$= \frac{n}{1 - \frac{3}{4}}$$

$$= 4n$$

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

Q2) $T(n) = \cancel{2}\, T\left(\frac{n}{2}\right) + 1$



Prime Merit
Page No.
Date

Scanned with CamScanner

For recurrences, if the initial condition is not given, assume it to be $T(1) = 1$

18/8/20

Insertion Procedure:

It can be done in a sorted/unsorted array.

If there are 'n' no. of items in an array, after insertion, size will be $(n+1)$.

If unsorted array → add at the end
If sorted array → insert in a particular pos. so that after insertion, the ele. are in sorted array.

Eg:

0	1	2	3	4
11	22	44	55	66
1	2	3	4	5

→ sorted array

Insert 33. so that the new array is in sorted order.

$i = 6$
$key = 33$

```
j = i - 1;
while (j > 0 and a[j] > key)
{
    a[j+1] = a[j];
    j--;
}
a[j+1] = key;
```

For implementing this algo., the array index should start from 1 always, not zero.
~~This is because, for inserting in the first pos.~~

Prime Merit
Page No.
Date

Scanned with CamScanner

This algo. works considering that the ele. ~~Start~~ from 0 to i-1 are in sorted order.

Insertion Sort:

```
InsertionSort (a, n)
{
    for i = 2 to n
    {
        key = a[i];
        j = i - 1;
        while (j > 0) and (a[j] > key)
        {
            a[j+1] = a[j];
            j = j - 1;
        }
        a[j+1] = key;
    }
}
```

Analysis:

~~Step~~ To find the exe. time:

Step 1) Find the freq. count of every statement

	Freq. Count
for i = 2 to n{	$c_1 n$
key = a[i];	$c_2 (n-1)$
j = i-1;	$c_3 (n-1)$
while (j > 0) and (a[j] > key){	$c_4 T$
a[j+1] = a[j];	$c_5 (T-(n-1))$
j = j-1; }	$c_6 (T-(n-1))$
a[j+1] = key; }	$c_7 (n-1)$

Prime Merit
Page No.
Date

Scanned with CamScanner

$c_1 \rightarrow$ time taken for exe. of each 'for' st.

When $i = 2$, while ( ) $\rightarrow t_2$ timed exe
When $i = 3$, " $\rightarrow t_3$ " "
When $i = 4$, " $\rightarrow t_4$ " "
When $i = n$, " $\rightarrow t_n$ " "

$\therefore$ Total no. of exe. for 'while ( )'
$= t_2 + t_3 + t_4 + \cdots + t_n$

$\Rightarrow T = t_2 + t_3 + \cdots + t_n$

$\Rightarrow T = \sum_{i=2}^{n} t_i$, t_i is the freq. count of while st. for the ith iteration of for loop

When $i = 2$, inner st. of while() $\rightarrow (t_2 - 1)$
When $i = 3$, " " " " $\rightarrow (t_3 - 1)$
When $i = n$, " " " " $\rightarrow (t_n - 1)$

$\therefore$ total no. of exe. for inner st. of 'while ( )'
$= (t_2 - 1) + (t_3 - 1) + \cdots + (t_n - 1)$
$= t_2 + t_3 + \cdots + t_n - 1 - 1 - 1 - \cdots - 1$
$= T - \underbrace{(n-1)}_{2 \text{ to } n}$

If the exe. time of an algo. ~~dep~~ varies depending on the dataset, then the algo. is loosely bound.

If the exe. time of the algo. is fixed for any kind of dataset, then the algo. is tightly bound.

Prime Merit
Page No.
Date

Scanned with CamScanner

Total exe. time of algo.:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T-(n-1)) + c_6(T-(n-1)) + c_7(n-1)$$

$c_1, c_2, c_3, c_4, c_5, c_6, c_7 \rightarrow$ constants involving that are involved with diff. type of st. computations

$$\Rightarrow T(n) = c_8 T + c_9 n + c_{10}$$

where $T = t_2 + t_3 + \dots + t_n$

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

<u>Case 1:</u> Elements are already in increasing order

10, 20, 30, 40, 50, 60, 70, 80, 90

For 20, $t_2 = 1$ For 40, $t_4 = 1$

For 30, $t_3 = 1$ For 90, $t_n = 1$

$$\therefore T = t_2 + t_3 + \dots + t_n$$
$$= 1 + 1 + \dots + 1$$
$$= n-1$$

$$\therefore T(n) = c_8 T + c_9 n + c_{10}$$
$$= c_8(n-1) + c_9 n + c_{10}$$
$$= (c_8 + c_9) n + (c_{10} - c_8)$$

$$\Rightarrow T(n) \approx an + b$$

$$\therefore T(n) = \Omega(n)$$

Here, $T(n) \rightarrow$ best case exe. time.

Prime Merit
Page No.
Date

Scanned with CamScanner

Case 2: Elements are in descending order

90, 80, 70, 60, 50, 40, 30, 20, 10

For 80, $t_2 = 2$, (first – true, second – first cond. false)
For 70, $t_3 = 3$, (first, two – true, third – first cond. false)
For 10, $t_n = n$

$$\therefore T = t_2 + t_3 + \dots + t_n$$
$$\Rightarrow T = 2 + 3 + \dots + n$$
$$\Rightarrow T = \frac{n(n+1)}{2} - 1$$

$$\therefore T(n) = c_8\left(\frac{n(n+1)}{2} - 1\right) + c_9 n + c_{10}$$

$$\approx an^2 + bn + c$$

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

Here, $T(n)$ is the worst case exe. time.

Since insertion sort algo. has 2 time complexities dep. on the dataset, hence, it is loosely bound.

Case 3: Ele. are in random order

i) 50% of ele. prev. to the key are smaller to the key.

50, 20, 40, 10, 80, 30, 90, 70, 60

Eg: for inserting 100th item, 99 items are sorted out of which 50 items are smaller than the key and 49 items are larger than the key.

Prime Merit
Page No.
Date

Scanned with CamScanner

If all elements ~~are~~ before the key are smaller to the key $\rightarrow$ $t_2 = 1$ $(t_n = 1)$

If all ele. before the key are larger to the key $\rightarrow$ $t_2 = 2$ $(t_n = n)$

If 50% of ele. before the key are smaller than the key $\rightarrow$ $t_i = \frac{i}{2}$

$\therefore t_2 = \frac{2}{2}, \; t_3 = \frac{3}{2}, \; \ldots, \; t_n = \frac{n}{2}$

t_8

$\downarrow$

either 1 or 2

not 1.5

If 30% ele. ~~are smaller~~ before the key are smaller than key and 70% larger

$\rightarrow t_n = \frac{70\,n}{100}$

$$\therefore T = t_2 + t_3 + \ldots + t_n$$

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

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

$\Rightarrow \cancel{T \approx an^2 + bn + c}$

$$\therefore T(n) = \frac{\left[\frac{n(n+1)}{2} - 1\right]}{2} c_8 + c_9 n + c_{10}$$

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

Prime Merit
Page No.
Date

Scanned with CamScanner

Here, ~~this~~ in this case 'a' will be smaller to the 'a' in case 2.

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

Here, $T(n)$ is the avg. case exe. time which is same as the worst case exe. time.

So broadly the exe. time of insertion sort is n^2, except for 1 case when the ele. are already arranged in ascending order

If all ele. are equal, $T(n) = n$

If the while cond. is changed to $a[j] \geq key$, then $T(n) = n^2$

19/8/20

Recursion:

Recursion takes a lot of time to execute but we still use it because the code is easier to write.

Step 1:
Write a soln. for a simplest prob. of same type. ~~by i/ps~~ ~~simplest~~ Simplest prob. means for smaller values i/ps whose soln. is known to us. No need to go for mathematical calc. to find the soln. for a smaller prob./i/p.

Step 2: Assume that the soln. will ~~defin~~ definitely ~~certainly~~ work for any no. less than n (size of the prob.), i.e., $(n-1)$, $(n-2)$, ..., $\frac{n}{2}$, $\frac{n}{3}$, ..., 1.

It may not work for i/ps like $(n+1)$.

Prime Merit
Page No.
Date

Scanned with CamScanner

n → no. of i/ps. Eg.: no. of ele. in an array or no. of vertices in a graph.

A recursive func. is a func. that calls itself directly or indirectly.

Step 3: Conquer the partial soln. to find out the final soln.

~~Eg.)~~

Eg.: Factorial of a no. n:

```
int factorial (n) {
{
    if (n = = 1)
        return 1;
    else
        return (n * factorial (n-1));
}
```

→ step 1 (no math. calc. are reqd.) (small no. – value known)

step 2

step 3

Eg: Calc. the sum of digits of a k-digit no. (n):

```
int sod (n)
{
    if (n/10 = = 0)
        return n;
    else
        return ((n%10) + sod (n/10));
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

k^{th} digit → rightmost digit
1st digit → leftmost digit

rightmost digit → n % 10
(k-1) digits → n / 10

sod = rightmost digit + sod. ((k-1) digits)

Eg.: <u>Tower of Hanoi</u>:

In the tower of Hanoi, ~~we~~ there are a no. of discs and the discs are ~~ra~~ arranged from larger to smaller, one above the other. They are available in one of the tower A and we have to build the same tower by moving the same discs from A to B, making C as an intermediate tower, and everytime we can shift only 1 disc at a time from one tower to another, keeping in mind that no larger disc is placed on a smaller disc.

```
void TOH (n, A, B, C)
{
    if (n = = 1) {
        move disc from A to B;
        return;
    }
    else
    {
        TOH (n-1, A, C, B);
        move nth disc from A to B;
        TOH (n-1, C, B, A);
    }
}
```

A → source
B → destination
C → intermediate

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. Time:

if () $\rightarrow T(1) = 1 \quad , n = 1$

else () $\rightarrow T(n) = T(n-1) + 1 + T(n-1)$
$= 2T(n-1) + 1 \quad , n > 1$

~~$\therefore T(n) =$~~

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

$$\begin{aligned}
\therefore T(n) &= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 4T(n-2) + 3 \\
&= 4(2T(n-3) + 1) + 3 \\
&= 8T(n-3) + 7 \\
&\vdots \\
&= 2^k T(n-k) + (2^k - 1) \quad , k > 1
\end{aligned}$$

~~For $n = 2^k$, $\Rightarrow k = \log n$~~

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

For $k = n-1$,

$$\begin{aligned}
\therefore T(n) &= 2^{n-1} T(n-n+1) + (2^{n-1} - 1) \\
&= 2^{n-1} T(1) + 2^{n-1} - 1 \\
&= 2^{n-1} + 2^{n-1} - 1 \\
&= 2 \cdot 2^{n-1} - 1 \\
&= 2^n - 1
\end{aligned}$$

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

Prime Marks
Page No.
Date

Scanned with CamScanner

Eg: Find out the height of a binary tree:

If only 1 node is present in the tree, i.e., the root node, then height = 0

```
int hbt (t)                    t -> pointer
{
    if (!t -> left && !t -> right)   //no left or right
        return 0;                     subtree

    else if (t -> left && !t -> right)   // left subtree
        return (1 + hbt (t -> left));     exist

    else if (!t -> left && t -> right)   //right subtree
        return (1 + hbt (t -> right));    exist

    else                                 // both subtree
    {                                     exist
        l = hbt (t -> left);
        r = hbt (t -> right);
        if (l > r)
            return (1 + l);
        else
            return (1 + r);
    }
}
```

Eg: Count the no. of nodes in a binary tree:

```
int count (t)
{
    if (!t -> left && !t -> right)
        return 1;
```

Prime Merit
Page No.
Date

Scanned with CamScanner

```
else if ( t -> left && !t -> right)
        return (1 + count(t -> left));

else if (!t -> left && t -> right)
        return (1 + count (t -> right));

else
{
        l = count(t -> left);
        r = count (t -> right);
        return (1 + l + r);
}
}
```

Eg: Count No. of Leaf Nodes in a binary tree:

```
int leaf (t)
{
        if (!t -> left && !t -> right)
                return 1;

        else if (t -> left && !t -> right)
                return leaf (t -> left);

        else if (!t -> left && t -> right)
                return leaf (t -> right)

        else
        {
                l = leaf (t -> left);
                r = leaf (t -> right);
                return (l + r);
        }
}
```

Prime Merit
Page No.
Date.

Scanned with CamScanner

24/8/20

Divide and Conquer Approach:

Recursive func. are very much used in this approach.

~~In this approach,~~ When the prob. becomes a larger one, (size of prob. is large — no. of i/ps for which the soln. is to be ~~find~~ found is large), finding the soln. to such large no. of i/ps becomes difficult. Hence, we divide the prob. into no. of smaller probs. (all smaller probs. are more ~~of~~ or less of same type) and is being solved for less no. of i/ps). Once we get the soln. for each smaller prob., we conquer those soln. to find out the ~~larger~~ final soln.

1) Binary Search:

Assumption: Ele. are present in ~~ascending~~ sorted order.

Return: $-1 \rightarrow$ not present

+ve integer $\rightarrow$ index in which it is present

```
int bin-search(a, l, h, key)
{
    if (l == h)          //one data present
    {
        if (key == a[l])
            return l;
        else
            return -1;
    }
```

Prime Merit
Page No.
Date

Scanned with CamScanner

```
else                                    //more than 1 data present
{
        m = (l+h)/2;
        if (key = = a[m])
            return m;

        else if (key < a[m])
            bin-search (a, l, m-1, key);

        else
            bin-search (a, m+1, h, key);
    }
}
```

No. of ele. = n (from l to h)

For $n = 1$, $T(1) = (1)$ ($T(n) = 1$ → simple st.)

For $n > 1$,

<u>Case 1:</u> Assuming that ele. is present in the mid pos. of array

$$T(n) = (1)$$

<u>Case 2:</u> Item is not present in mid pos.

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

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

<u>Case 3:</u> Item not present in array

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

Prime Mark
Page No.
Date

Scanned with CamScanner

Case 2:

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

Using Master theorem,

~~Assuming $n = 2^k$.~~

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

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

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

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

$$= T\left(\frac{n}{8}\right) + 3$$

$$\vdots$$

$$= T\left(\frac{n}{2^k}\right) + k \quad , k >= 1$$

When $n = 2^k$,

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

$$= T(1) + k$$

$$= 1 + \log n$$

$\therefore T(n) = \cancel{\Theta(\log n)}$ order of $\log n$

Case 1 : $T(n) = 1$ (order)

Case 2 : $T(n) = \log n$ (order)

Exe. time of bin. search varies from $\Omega(1)$ to $O(\log n)$ $\longrightarrow$ Hence, the algo. is loosely bound.

$\therefore T(n) = \Omega(1)$, $= O(\log n)$ $\Big\} \rightarrow$ Avg case $= O(\log n)$

Date
Page No.
Prime Mark

Scanned with CamScanner

Item present in array:

$\Omega(1) \leq$ Exe. time $\leq O(\log n)$

Avg. case = $O(\log n)$

Item not present in array:

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

#) In Tower of Hanoi, $T(n) = 2^n$ (always for n)
when $n = 1$, $T(1) = 1$

In Binary search, $T(n) = 1$ / $\log n$ } both values possible for n

item found in mid pos.

item not found ~~or not present~~ in mid pos.

For $n = 1$, $T(1) = 1$
Item present in array
Since in bin. search, for $n > 1$, $T(n)$ has 2 values → loosely bound.

Since in TOH, for $n > 1$, $T(n)$ has only 1 value → tightly bound.

Q1) In KBC, Mr. X can ask only true/false question. A contestant, Prof. Y will think a secret no. (n) from 1 to 1000. How many true/false questions Mr. X ~~can at~~ must ask to Prof. Y to guess the secret no.?

Prime Merit
Page No.
Date

Scanned with CamScanner

Soln.: Eg.: $n = 234$

Q) Is the no. less than 500?
Ans) Yes

Q) Is the no. less than 250?
Ans) Yes

⋮

The prob. is of the form of bin. search.

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

~~Rano~~ Here, $n = 1000$

$\therefore$ No. of guesses $= \log_2 1000$

≈ 10 (Ans)

Q2) What is the running time of your algo. to find out the pos. of the leftmost 1 present in a binary sorted array?

Soln.: Eg.: 001111

Step 1: check the middle pos.
Step 2: If it is 1, check the left half, else right half.

~~Hence~~ $\therefore T(n) = \Theta(\log n)$ (not $O(\log n)$)

Here, the exe. time will be exactly $\Theta(\log n)$. (Ans)

Prime Merit
Page No.
Date

Scanned with CamScanner

Q3) In an unsorted array of size n, an item is a ~~to~~ leader if all items to its right are smaller. ~~Dec~~ Declare all the leaders in $\Theta(n)$ time.

Soln. Eg:

10	90	30	40	75	45	65	25	50	35
1	2	3	4	5	6	7	8	9	10

start

i) leader = 35
pos = 10

Traverse to the left. If we find a value < leader, do no op. If we find a value > leader, then leader ← new value, pos ← new pos.

ii) leader = 50
pos = 9

iii) pos = 8 → no leader

iv) leader = 65 (65 > 50) ← prev. leader
pos = 7

v) pos = 6 → no leader

vi) leader = 75 (75 > 65)
pos = 5

vii) pos = 4 → no leader

viii) pos = 3 → no leader

ix) leader = 90 (90 > 75)
pos = 2

x) pos = 1 → no leader

~~∴ no. of~~

∴ Pos. of leaders = 10, 9, 7, 5, 2

Prime Merit
Page No.
Date

Scanned with CamScanner

∴ We need to traverse the ~~whole~~ whole array linearly from right to left.

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

Q4) Each row of an $(n \times n)$ 2D ~~max~~ binary array is sorted in descending order. Write an efficient algo. to ~~row~~ determine the row no. containing maximum. 1.?

Soln. Eg.:

1 1 0 0 0	1 1 0 0
1 0 0 0 0	1 0 0 0
1 1 1 1 0 0	1 1 1 0
	1 0 0 0

Method 1: Search each row linearly.

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

Method 2: Using bin. search, find the no. of 1s in each row.

$T(n) = n \log n$

Use linear search to determine the max.

$T(n) = n$

∴ Total: $T(n) = n + n \log n$

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

Method 3: Do a linear search on the first row. Suppose the 5th ele. is the first 0.

$\Rightarrow$ No. of 1s = 4

Go to the next row, if the 5th ele. = 0, → discard the row.

Go to the next row, if the 5th ele. = 1, move right to find the first zero.

Prime Merit
Page No.
Date

Scanned with CamScanner

Suppose the ~~no~~ 20th ele. = 0.
$\Rightarrow$ No. of 1s = 19

Go to the next row, check the 20th pos.

$\therefore T(n) = 2n$
$\Rightarrow T(n) =$ order of n

In this case, linear search made the prob. faster than binary search.

25/3/20

2) Merge Sort:

Merge Procedure:

Given 2 sorted arrays. We have to merge the 2 sorted arrays to form a single sorted array. If the 2 arrays are not sorted, merge procedure cannot produce a single sorted ~~array~~ array.

Instead of taking 2 sorted arrays, we can take a single array which has been logically divided into 2 parts ~~where~~ where each of the parts are in sorted order.

Eg.: | 15 | 25 | 67 | 75 | 85 | 95 | 10 | 20 | 30 | 40 | 50 |

l . . . m $m+1$ . . . h

sorted array sorted array

Prime Merit
Page No.
Date

Scanned with CamScanner

```
void merge (a[], l, m, h)
{
    i = k = l;
    j = m+1;

    while ( i <= m && j <= h)
    {
        if (a[i] <= a[j])
            b[k++] = a[i++];

        else
            b[k++] = a[j++];
    }

    while (i <= m)
        b[k++] = a[i++];

    while (j <= h)
        b[k++] = a[j++];

    while (l <= h)
        a[l++] = b[l];
}
```

→ copying from b[] to a[] (reqd. because our aim is to make a[] sorted)

Exe. Time:

Let no. of ele. = n

No. of Ele. from l to m = n_1

No. of ele from $(m+1)$ to h = n_2

$\therefore n_1 + n_2 = n$

Prime Merit
Page No.
Date

Scanned with CamScanner

In a 100 ele. array,

$n_1 = 10$ or $n_1 = 70$

$n_2 = 90$ $\quad$ $n_2 = 30$

Initialization $\rightarrow f(n) = 1$

The st. inside while() have f(n) less than that of while() because they are simple st. So no need of calculating their f(n).

1st while() $\rightarrow f(n) =$ min. of n_1 and n_2

max. no of comparisons

$= n_1 + n_2$

~~So no. of min (n_1, n_2)~~ $\leq$ no. of comparisons $\leq$ ~~max~~ $(n_1 + n_2)$

~~min $(n_1, n_2$~~

Out of 2nd and 3rd while() st., only one of them will get executed. On combining 1st and 2nd while or 1st and 3rd while st, no. of ~~times~~ times it will execute $= n$

4th while() $\rightarrow f(n) = n$ (exactly n)

$\therefore$ Total exe. time :

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

$\Rightarrow T(n)$ is of order n.

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

Prime Merit
Page No.
Date

Scanned with CamScanner

Merge Sort:

```
void mergeSort (a[], l, r)
{
    if (l < r)
    {
        m = (l+r)/2
        mergeSort (a[], l, m);
        mergeSort (a[], m+1, r);
        merge (a[], l, m, r);
    }
}
```

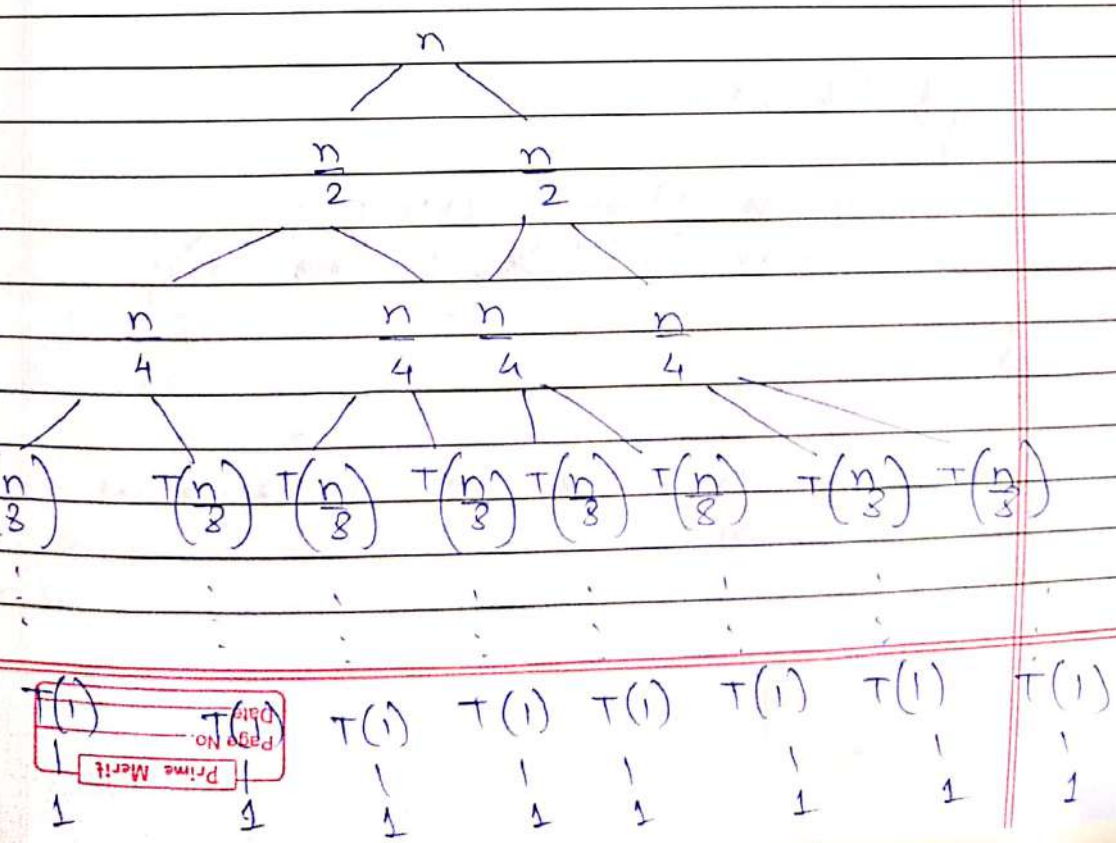
Exe. time:

For $n = 1$, $T(1) = 1$

For $n > 1$, $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn$

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

$(\Theta(n) = cn = n)$



Scanned with CamScanner

Assuming $n = 2^k$,

$$T\left(\frac{2^k}{2^k}\right) = 1$$

Sum of each step $= n$
No. of steps $= k$

$$\therefore T(n) = nk$$
$$= n \log n$$

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

This algo. is tightly bound. Irrespective of the type of ele. in the array, the array is always divided into 2 halves, so for all cases its exe. time is $n \log n$. Hence, it is tightly bound.

Modified mergeSort:

```
void mergeSort(a[], l, r)
{
    if (l < r)
    {
        mergeSort(a[], l+1, r);
        merge(a[], l, r);
    }
}
```

This algo. is more or less same as insertion sort.

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. time:

for $n = 1$, $T(1) = 1$

for $n > 1$, $T(n) = T(n-1) + \Theta(n)$
$= T(n-1) + n$

$\therefore T(n) = T(n-1) + n$
$= T(n-2) + n-1 + n$
$= T(n-2) + 2n - 1$
$= T(n-3) + 2n - 1 + n - 2$
$= T(n-3) + 3n - 3$
$\Rightarrow T(n) = T(n-4) + n - 3 + 3n - 3$
$= T(n-4) + 4n - 6$

$\vdots$

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

When $k = n$,

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

$\Rightarrow T(n) =$ order of n^2

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

This algo. is hence, rarely used because the order of $T(n)$ is n^2.

Prime Merit
Page No.
Date

Scanned with CamScanner

Instead of dividing the array into 2 equal halves, we can divide it ~~to~~ into 2 unequal ~~has~~ halves → $\frac{n}{3}, \frac{2n}{3}$.

In that case, the order of $T(n)$ would be $n \log_{3/2} n$ or $n \log_{1.5} n$.

When the base of log decreases, ~~the~~ exe. time increases. Hence, the best exe. time is obtained when the array is divided into 2 equal halves.

Q1) Two sets are given (S_1 and S_2) where the ele of the sets are in sorted array. Can merge ~~sort~~ procedure be applied to find the union of the 2 sets?

Soln.: While merging the arrays, if both the arrays contain same ele., then bring one of them to the new array and increase the index of both the initial arrays by 1.

Q2) Can merge procedure be applied to find the intersection of two sets?

Soln. If ele. of both ~~a~~ arrays are equal, bring the ele. to the new array and increase both of its indices. If one ele. is smaller and the other is greater, do not bring the ele. to the new array, only increase their indices by 1.

Prime Merit
Page No.
Date

Scanned with CamScanner

Q3) If the 2 arrays given are sorted, one in ascending and the other in descending order, can merge sort be applied?

Soln. choose the indices accordingly.
Ascending → index = initial pos.
~~Descending~~ → increment by 1

Descending → index = last pos.
→ decrement by 1

26/8/20

3) Quick Sort:

Like merge sort takes help of a merging procedure, quick sort takes help of a partition procedure.

Partion Procedure:

Eg.:

50	66	33	44	77	55	22	88	39	69	59
1	2	3	4	5	6	7	8	9	10	11

Divide the array ~~into~~ logically in 2 halves, not necessarily 2 equal halves. Ele. of the first half should be less than or equal to ele. of the ~~rio~~ second half. It returns the index pos. which rep. that ele. from starting to index pos. all are less than equal to ele. present from (index pos. + 1) to last pos.

ele. at index pos. → pivot ele.

Assume the first ele. of the array to be pivot initially. Initialize $i = l - 1$, $j = h + 1$ (l = st. index, h = highest index)

Prime Merit
Page No.
Date

Scanned with CamScanner

Step 1: pivot = 50

```
repeat
    j--;
until (pivot ≤ a[j]);
```

Step 2: j = 9, 50 > 39

```
repeat
    i++;
until (pivot ≥ a[i]);
```

swap a[i] and a[j]

Repeat steps 1, 2 till i < j.
When i has crossed j, return the partitioning index as j.

```
int partition (a[], l, h)
{
    pivot = a[l];
    i = l - 1;
    j = h + 1;

    while (1)
    {
        repeat
            j--;
        until (pivot ≤ a[j]); → x = 1, 2, 3, ..., n
        repeat
            i++;
        until (pivot ≥ a[i]); → y = (n-1), (n-2), ..., 1
```

Prime Merit
Page No.
Date

Scanned with CamScanner

```
    if (i<j)
        swap (a, i, j);   //exchange ith ele. with jth ele. in array a
    else
        return j;
    }
}
```

Exe. Time:

Assuming no. of ele. from 1 to n = n

Initialization → $f(n) = 1$

~~1st repeat until →~~

Let n = 100 and returns j = 40.
If j = 40, then i = 41.
~~1st repeat until~~
1st repeat until has been decremented from 101 to 40, not in one iteration, but in multiple iterations. → $f(n) = 101 - 40$
$= 61$

2nd repeat until → $f(n) = 41 - 1$
$= 40$

~~If j = 80~~ If return pos = 80.

Then 2nd repeat until would have executed for more no. of times.

If return pos = 10, the 1st repeat until loop is executed for more no. of times

Prime Merit
Page No.
Date

Scanned with CamScanner

So on adding the no. of exe. of both the repeat until loops, asymptotically it would be = n.

~~If~~ The if (i < j) - else st. would execute less than the sum of the repeat until st. → $\Theta(n)$ not considered

f(n) of swap → const.
T(n) = 1

∴ Exe. time $\Theta(n)$ or

∴ $x + y = n + 1$

∴ $\boxed{T(n) = \Theta(n)}$

It is tightly bound because sum of the repeat until st. is always exactly n.

Quick Sort:

```
void quickSort (a[], p, r)      // p → left index
{                                  r → right index
    if (p < r)
    {
        q = partition (a, p, r);
        quickSort (a, p, q);
        quickSort (a, q+1, r);
    }
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. Time:

Ele. from p to r = n.

For $n = 1$, $T(1) = 1$.

Case 1: Balanced Partitioning:

Partition will divide the array into 2 equal halves.

Eg: Suppose pivot = 50 and 50% of the items are smaller than 50 and 50% are larger. So the array will be partitioned into 2 equal halves.

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

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

On solving the above recurrence,

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

Case 2: Imbalanced Partitioning:

~~$T(n) = \Theta(1)$~~ The pivot ele. is picked up such that it is either the smallest or the largest. So one half will have only 1 ele. (pivot) and the other half will have $(n-1)$ ele.

Scanned with CamScanner

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

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

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

On solving the above recurrence, we get
$$T(n) = \Theta(n^2)$$

Hence, the exe. time varies w.r.t. the dataset and so it is loosely bound

$\therefore$ Best case $= \Omega(n \log n)$
Worst case $= O(n^2)$

Q1) If the array is in sorted order, what would be the exe. time of quick sort?

Soln. For a sorted array (ascending), if we consider the first ele. as pivot, after partitioning it will produce an imbalanced partitioning.

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

Prime Merit
Page No.
Date

Scanned with CamScanner

Q2) If the ele. are in descending order, what will be the exe. time?

Soln. First item = pivot = largest item.

So after partitioning → imbalanced partitioning

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

Hence, for sorted array, merge sort ~~produ~~ gives better results (lesser exe. time) than quick sort.

Q3) What will be the exe. time if all ele. of the array are equal?

Soln. Every ele. will unnecessarily be exchanged with each other and will result in balanced partitioning.

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

Scanned with CamScanner

31/8/20

Average Case Analysis:

For Linear Search:

~~Aase~~ Item can be present in any pos. from 1 to n^{th} pos. in the array. Considering the item is present in diff. pos. ~~we~~, find out the exe. time. Compute it for n times where in each time the ele. is present in a diff. pos. Divide the result by n to get the avg. case exe. time.

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

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

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

$\Rightarrow T(n)$ is of order n.

For Quick Sort:

Exe. time varies b/w balanced & imbalanced partitioning. The partitioning algo. ~~too~~ is tightly bound, but depending on the pos. of the partitioning, the exe. time varies from $n \log n$ to n^2.

Compute the exe. time for partitioning at pos. $1, 2, \dots, (n-1)$ and then divide the result by n to get the avg. case exe. time.

Prime Merit
Page No.
Date

Scanned with CamScanner

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

For $n = 1$, $T(1) = 1$

$C_a(n)$ is the avg. case exe. time for 'n' no. of i/ps.

$$C_a(n) = \underbrace{(n+1)}_{\frac{(n+1)n}{n}} + \frac{1}{n} \sum_{1 \le q < n} \{C_a(q) + C_a(n-q)\} \rightarrow (1)$$

Quick sort is divided into 3 parts.

i) Partitioning procedure $\rightarrow \Theta(n) = n+1$

ii) smaller quickSort $\rightarrow C_a(q)$

iii) " " $\rightarrow C_a(n-q)$

Since we are finding the avg. case, we divide it by n. , n = no. of ele. / i/ps.

Multiply n to both sides to (1).

$$\Rightarrow n C_a(n) = n(n+1) + \sum_{1 \le q < n} \{C_a(q) + C_a(n-q)\}$$

$$\Rightarrow n C_a(n) = n(n+1) + \{C_a(1) + C_a(n-1) + C_a(2) + C_a(n-2) + \dots + C_a(n-1) + C_a(1)\}$$

$$\Rightarrow n C_a(n) = n(n+1) + 2\{C_a(1) + C_a(2) + \dots + C_a(n-1)\} \rightarrow (2)$$

Prime Merit
Page No.
Date

Scanned with CamScanner

Replacing n by $(n-1)$ on both sides of ②

$$\Rightarrow (n-1)C_a(n-1) = (n-1)n + 2\{C_a(1) + C_a(2) + \cdots + C_a(n-2)\} \rightarrow ③$$

Subtracting ③ from ②:

$$\Rightarrow nC_a(n) - (n-1)C_a(n-1) = n(n+1) - n(n-1) + 2C_a(n-1)$$

$$\Rightarrow nC_a(n) = C_a(n-1)(2 + n - 1) + n(n+1-n+1)$$

$$\Rightarrow nC_a(n) = C_a(n-1)(n+1) + 2n$$

Dividing $n(n+1)$ in both sides.

$$\Rightarrow \frac{C_a(n)}{n+1} = \frac{C_a(n-1)}{n} + \frac{2}{n+1} \rightarrow \text{recurrence}$$

Solving the recurrence:

$$\therefore \frac{C_a(n)}{n+1} = \frac{C_a(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$ (Subs. the value of $n-1$)

$$= \frac{C_a(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$ (Subs. the value of $n-2$)

$$\vdots$$

$$\Rightarrow \frac{C_a(n)}{n+1} = \frac{C_a(1)}{2} + 2\sum_{3 \le k \le n+1} \frac{1}{k}$$

For quick sort, $C_a(1) = 1$

Prime Merit
Page No.
Date

Scanned with CamScanner

$$\Rightarrow \frac{C_a(n)}{n+1} = \frac{1}{2} + 2 \sum_{3 \le k \le n+1} \frac{1}{k}$$

$$\Rightarrow C_a(n) = \frac{n+1}{2} + 2(n+1) \sum_{3 \le k \le n+1} \frac{1}{k}$$

$$\Rightarrow C_a(n) = (n+1) + 2(n+1) \sum_{3 \le k \le n+1} \frac{1}{k}$$ (asymptotically $(n+1) = \frac{(n+1)}{2}$)

We know, $\sum_{k=i}^{n} \frac{1}{k} \le \int_{i-1}^{n} \frac{1}{x} dx$

When we increase the range, ~~$\int \frac{1}{x} dx$ is always~~ $\sum_{k=i}^{n} \frac{1}{k}$ will always be less than equal to $\int \frac{1}{x} dx$. (sum $\le$ integration)

$$\therefore C_a(n) \le (n+1) + 2(n+1) \int_{2}^{n+1} \frac{1}{x} dx$$

$$\Rightarrow C_a(n) \le (n+1) + 2(n+1) \left[\log x\right]_{2}^{n+1}$$

$$\Rightarrow C_a(n) \le (n+1) + 2(n+1) \left[\log(n+1) - \log 2\right]$$

$$\Rightarrow C_a(n) \le (n+1) + 2(n+1) \log(n+1)$$ (asymptotically)

$$\Rightarrow C_a(n) \le c.n \log n$$ (asymptotically)

$\Rightarrow C_a(n)$ is of order $n \log n$.

$\therefore$ $\boxed{\text{Avg. case} = \Omega(n \log n)}$, we put Ω because the order of avg. case is ~~not same~~ closer to $\Omega(n \log n)$ and not $O(n^2)$.

Prime Merit
Page No.
Date

Scanned with CamScanner

Generally, if it is asked to write the exe time of quick sort $= \Omega(n \log n)$ because its avg. case is $\Omega(n \log n)$

If the avg. case would have been of the order n^2, then we would have written it as $O(n^2)$.

Randomized Quick Sort:

Quick sort gives the best exe. time only ~~when the are~~ in the case of balanced partitioning. ~~So we~~ So to forcefully make the partitioning balanced, we ~~can~~ have to choose ~~such~~ a pivot ele. such that 50% of the ele. are smaller to it and 50% larger.

i = select (a[ ])

This selection procedure selects an item present in the ith pos. It returns such a pos. such that 50% of the items are smaller to the pivot ele. This selection procedure has an exe. time = 1.

Randomized Partition:

```
int ran_partition (a[], l, h)
{
    i = select (a[ ]);
    ~~a[l] → a[i]~~
    a[l] is exchanged with a[i]
    pivot = a[l];
    i = l - 1;
    j = h + 1;
    ⋮
}
```

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

Prime Merit
Page No.
Date

Scanned with CamScanner

This algo. will return the middle pos. of the array.

```
void ran-quickSort(a[], p, r)
{
    if (p < r)
    {
        q = ran-partition (a, p, r);
        ran-quickSort (a, p, q);
        ran-quickSort (a, q+1, r);
    }
}
```

$$T(n) = \Theta(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) \rightarrow \text{balanced partitioning}$$

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

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

Here, $T(n)$ will be rep. using Θ, because whatever the i/p may be, due to randomized partitioning, it will always produce balanced partitioning. Hence, best case = worst case → tighter bound.

Randomized Quick Sort works provided the randomized partitioning works where the selection procedure will have an exe. time of 1. If exe. time of selection procedure $\neq 1$, then $T(n) \neq \Theta(n \log n)$

Physical Interpretation of Big Oh Notation:

When we measure exe. time of an algo, its unit should be s, mins, hrs.

Eg: $f(n) = 2n + 3$

$\Rightarrow f(n) = O(n) \rightarrow$ the real exe. time would be less than or equal to a const. times n.

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

If we know the exe. time for $n = 10$, we can find out the value of c to calculate the exe. time for any i/p more than 10.

Suppose $c = 3$, then for $n = 1000$, the exe. time will be less than or equal to (3×1000).

$f(n) = 2n + 3$

- $(n+1)$ 'for' st. where each st. takes 10 units of time
- 'n' no. of '+' op. where each st. takes 6 units of time
- 1 'assignment' st. which takes 4 units of time
- 1 'return' st. which takes 15 units of time

This is fixed for one machine. This varies from machine to machine.

For $n = 10$,

Meth.1) Exe. time $= (10+1).10 + 10.6 + 4.1 + 15.1$

$= 110 + 60 + 4 + 15$

$= 189$ units of time

Prime Merit
Page No.
Date

Scanned with CamScanner

Meth. 2)
By making the sys. a uni-user sys., we run the code having $f(n) = 2n+3$, for $n = 10$.

$\therefore$ ~~Real~~ exe. time $= 189$.

$\therefore 189 \leq c \cdot 10$
$\Rightarrow c \approx 19$

$\therefore$ Real exe. time $(189) \leq 19n$

The value $c = 19$ is computed for $n = 10$. ~~It~~ This inequality may not work for $n < 10$. But for $n \geq 10$, the exe. time will always be less than equal to $19n$. units of time.

$\therefore$ Max. exe. time of 100 i/ps $= (19 \times 100)$
$= 1900$ units of time

So the exe. time will never be more than $19n$. This is the phy. interpretation of O notation.

Eg: $f(n) = n^2 + 2n + 1$
$\Rightarrow f(n) = O(n^2)$
$\Rightarrow f(n) \leq c \cdot n^2$

For $n = 100$, we can find the value of c by taking any assumptions. Whatever value we get, the exe. time will be less than or equal to $c \cdot n^2$, provided $n \geq 100$.

If we know that the no. of i/p will never be less than 100, then we take $n = 100$ to find the value of c.

Prime Merit
Page No.
Date

Scanned with CamScanner

Q) Between n^2 and $n\log n$, which one ~~is greater~~? will take more time to execute?

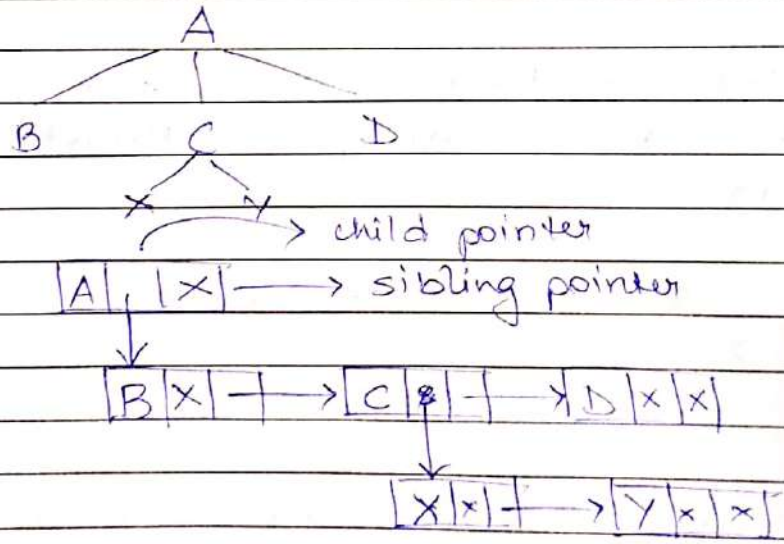
Soln. n^2 will take more exe. time than $n\log n$, only for higher values of n. For lower values of n, it may not always be true.

1/9/20

Heap Data Structure:

A tree is rep. by a LL. Each node will have 3 fields:

i) Data field
ii) Pointer pointing to one of its children
iii) Pointer pointing to one of its sibling



Size of a node:

i) 2 pointers each of size 4 bytes
ii) Data field is of type integer having 2 bytes

∴ Total size of a node $= ((4\times 2) + 2)$ bytes
$= 10$ bytes

Prime Merit
Page No.
Date

Scanned with CamScanner

If a tree has 10 nodes, ~~size of tree~~ space occupied in m/m due to storing it as a LL = (10 × 10) bytes
= 100 bytes

If the tree is Binary Tree, then ~~the~~ instead of the child and sibling pointer, there are left and right pointer.

A BT can be left skewed or right skewed.
Left skewed → extended towards the left, no right subtree
Right skewed → extended towards the right, no left subtree

~~If a BT is rep. using an array, the root node ~~cite~~ have an index = 1 and not~~

If a BT is rep. using an array:

i) root node → index = 1 (never 0)
ii) left subtree → 2i
iii) Right subtree → 2i + 1

} if i is the index no. of the parent

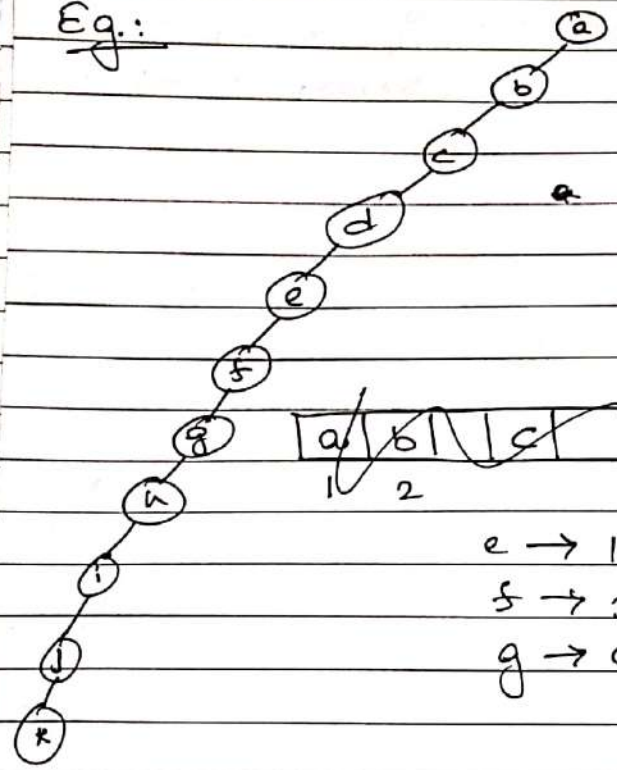
Eg: parent index = 5
left subtree index = 10
right subtree index = 11
root index = 1

~~For left skewed binary tree,~~

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg.:



left skewed bin. tree with 11 nodes.

a → 1
b → 2
c → 4
d → 8

a	b	c								
1	2									

e → 16
f → 32
g → 64
h → 128
i → 256
j → 512
k → 1024

Height of this tree = 10 (no. of paths)
No. of nodes = 11
∴ Size of array = 1024 ele.

If each data is of 2 bytes,
size of array = (1024 × 2) bytes
= 2048 bytes

If we rep. this BT using LL,

Size of LL = (10 × 11) bytes
size of node — no. of nodes

= 110 bytes.

So from mem. requirement point of view, LL rep. is better than array rep.
~~However,~~

Prime Merit
Page No.
Date

Scanned with CamScanner

If the BT is neither left skewed nor right skewed, → a complete BT.

Complete BT: If there are n no. of nodes all nodes must be present from 1 to n.

for BT having 20 nodes

Size of LL = (20 × 10) bytes

↓ no. of nodes ↓ size of a node

= 200 bytes

Size of array = (20 × 2) bytes
= 40 bytes

So in this case array rep. is better than LL.

Searching time in LL = n (sequential access)
Searching time in array = 1 (random access)

Array is faster ~~in~~ during ~~searching~~ accessing w.r.t. mem., sometimes array rep. takes less mem, sometimes LL rep. takes less mem.

Heap:

~~Ty~~ Types: i) Max heap
ii) Min heap.

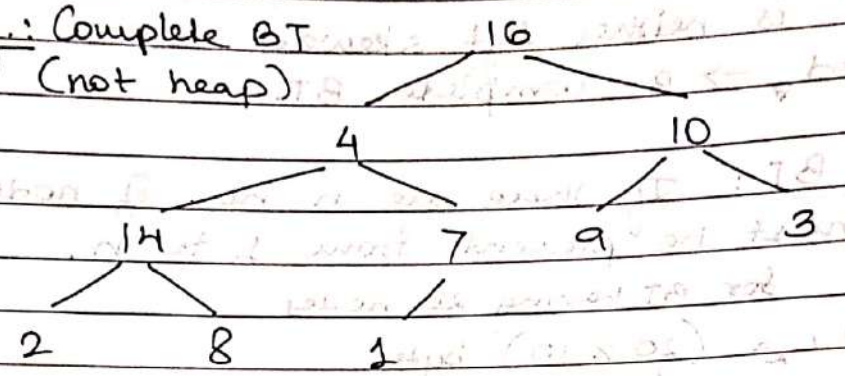
Heap is a complete BT ~~w~~ where the data value present in a node must be greater than its children → ~~Ma~~ Max Heap.

Heap is usually rep. as an array and not LL because it is a complete BT and ~~hence~~ thus array takes less space.

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg.: Complete BT (not heap)



~~a = 16 | 14 | 10~~

a = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

Parent of 6th node = 3rd index

If i is the node, its parent is present in $\left\lfloor \frac{i}{2} \right\rfloor$ (floor).

Properties of Heap:

i) Root node starts with index 1. All nodes must be present from 1, ..., n, if we have 'n' no. of nodes in the complete BT.

ii) a[parent(i)] $\geq$ a[i] (parent node $\geq$ ~~leaf~~ node) in max heap for i > 1. Value of a node is atmost the value of its parent in case of max heap.

* a[parent(i)] $\leq$ a[i] (parent node $\leq$ node) in min heap for i > 1.

iii) Since heap is a complete BT, the height of any node is atmost $\Theta(\log n)$.

Prime Merit
Page No.
Date

Scanned with CamScanner

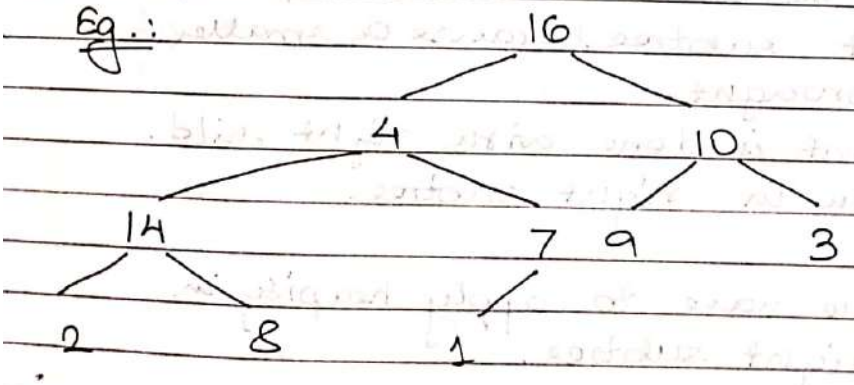
To rep. heap as an array:

```
int parent (i)                    // i = index
{
    return ⌊i/2⌋;
}

int left (i)
{
    return (2*i);
}

int right (i)
{
    return ((2*i) + 1);
}
```

Heapify Procedure:

- Heapify procedure is applicable to a particular node of a complete BT. When this procedure is applied to the ith node, its left and right subtree must maintain the heap property.

Eg.:



~~Heapify~~ a = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

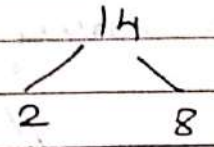
Heapify (a, 2)

index = 2
value = 4

Prime Merit
Page No.
Date

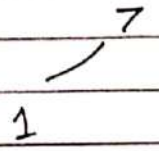
Scanned with CamScanner

Left subtree :



→ Follows heap prop.

Right Subtree :



→ Follows heap prop.

We can apply heapify to node at ~~index~~ index = 2, because its left and right subtree follow the heap prop.

Heapify → Give the heap prop. to a node without ~~violations~~ violating the heap prop. in its children. The data values can be rearranged, but the heap prop. should be maintained in both the children and the node itself.

<u>Step 1</u> : Compare 4 with 14 & 7. Replace 4 with the largest value.

If the replacement is done with the left child, there might be a violation of heap prop. in the left subtree because a smaller value is brought.
If replacement is done with right child.
→ violation in right subtree.

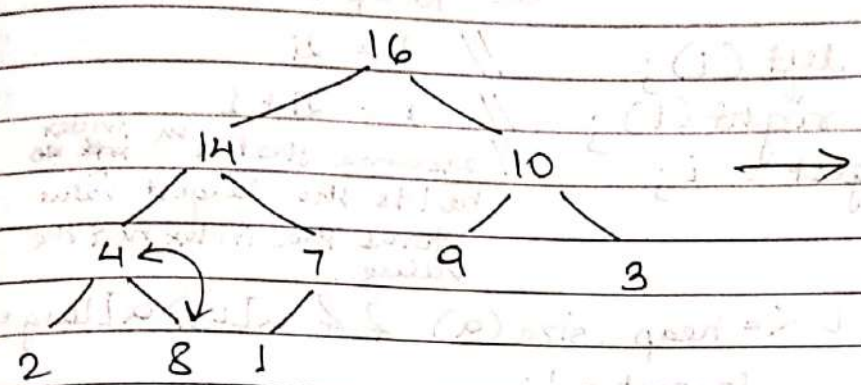
So again we have to apply heapify in the left/right subtree.

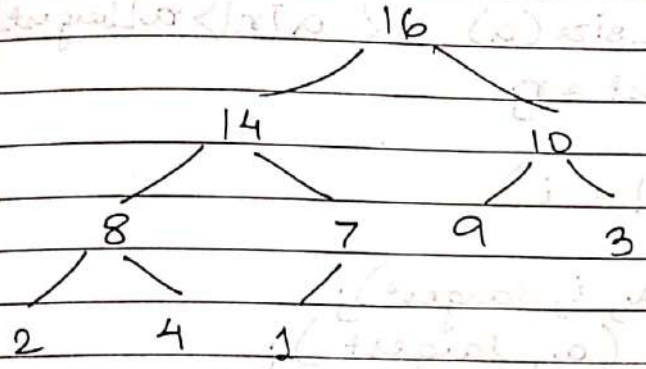
Continue doing this until we find out that the parent is largest or the node does not have any children (leaf node).
The leaf nodes hence, always follow the heap prop.

Prime Merit
Page No.
Date

Scanned with CamScanner

After replacing 4 ~~from~~ with 14.





~~Heapify (a, i)~~ ~~//i → node on which heapify is to be applied (i → index)~~
~~{~~

Algo. :

In this algo. we convert a BT to a max heap. We apply heapify to a node at a given index, assuming that its left and right subtree follow the heap property.

Prime Merit
Page No.
Date.

Scanned with CamScanner

```
Heapify (a, i)        //i = index of node that is to
{                        be heapified.
    l = left (i);     //  l = 2i
    r = right (i);    //  r = 2i + 1
    largest = i;      //assume that ith index
                      holds the largest value
                      Stores the index, not the
                      value.

    if (l <= heap_size (a) && a[l] > a[largest])
            largest = l;

    if (r <= heap_size (a) && a[r] > a[largest])
            largest = r;

    if (largest != i)
    {
        swap (a, i, largest);
        Heapify (a, largest);
    }
    return;
}
```

$l <=$ heap_size (a) → if left subtree exists
$r <=$ heap_size (a) → if right subtree exists

size of heap ≤ size of array.

If the index of left subtree (l) ≤ heap size
→ left subtree exists.

Scanned with CamScanner

Exe time:

Initialization → $T(n) = 1$

First if st. → 1
Second if st. → 1
Third if st. → 1
Swap func. → 1 (3 st.)

~~Heapify ( ) :~~

Eg: Heapify the root node:
When we call the heapify ( ), there are 2 possibilities.

i) Root node is largest among the left, right subtree and itself.

In this case, $T(n) = 1$ → no heapify is reqd (already heapified)

∴ Best case Exe. time = $\Omega(1)$.

ii) Assume the heap has 'n' no. of nodes.
Height of a complete BT = $\log n$.

∴ Max. no. of nodes in left subtree possible $= \frac{n}{2}$

∴ Max. no. of nodes possible in right subtree $= \frac{n}{2}$

→ This is because heap is a complete BT.

$n = 2^L - 1$, L = no. of levels.
$\Rightarrow L = \log_2(n+1)$
$\Rightarrow$ & Height $= T(\log_2 n)$

For a complete BT, containing $2^L - 1$ nodes, no. of nodes in left/right subtree $= \frac{2^L - 1}{2} = \frac{n}{2}$

Prime Merit
Page No.
Date

Scanned with CamScanner

After swapping 'i' and the 'largest', the ~~he~~ heapify () func. will be called for either the left subtree or the right subtree.

$\therefore$ Heapify () $\rightarrow T(n) = T\left(\frac{n}{2}\right)$

$\therefore$ Total Exe. time:

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

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

Applying Master th. :

$a = 1, b = 2, f(n) = 1 = n^0$

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

$\therefore \cancel{n^{\log_b a} < f(n)}$ $\therefore n^{\log_2 1} = f(n)$

$\therefore \cancel{n^{\log_b a} < f(n)}$

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

This is the worst case exe. time.

Prime Merit
Page No
Date

Scanned with CamScanner

2/9/20

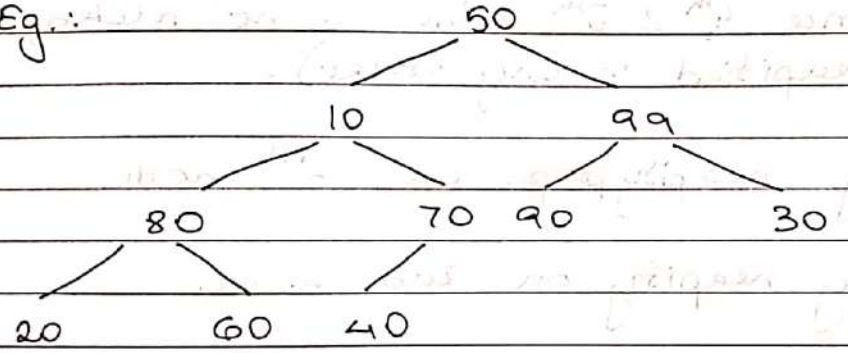
Given a complete binary tree, we will convert it to a max heap str. Every complete BT may not be a heap. Complete BT is one of the prop. of a heap.

i) Complete BT's array rep. always start from 1. It may go upto the end of the array or it may ~~go up~~ end in the mid of the array. So the size of the heap can be from 1 to size of the array.

ii) Size of the array >= size of the heap.

<u>Building a Heap:</u>

Eg.:



No. of nodes = 10

a =

50	10	99	80	70	90	30	20	60	40
1	2	3	4	5	6	7	8	9	10

Whatever the str. of the heap maybe, the leaf nodes always maintain the heap prop.

If there are n nodes, indices of the leaf nodes = $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$

Prime Merit
Page No.
Date

Scanned with CamScanner

Eg. If there are 20 nodes, indices of leaf nodes = 11, 12, ..., 20

Indices of ~~leaf~~ nodes that may not maintain heap prop. = 1 to $\lfloor \frac{n}{2} \rfloor$

Eg. If there are 71 nodes, indices of nodes that do not maintain heap prop.
= 1, 2, ..., 35

Step 1: We cannot call heapify on the root node because its left subtree does not follow the heap prop.

So we will first apply on 5th node, then 4th node (since 4th & 5th node are not related, they can be heapified in any order).

Step 2: Apply heapify ~~prop~~ on 3rd node.

Step 3: Apply heapify on 2nd node.

Step 4: Apply heapify on root (1st) node.

So while applying heapify, we will ~~orde~~ apply it in the order = $\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, \ldots, 1$

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Build~~ BuildHeap (a)

{

for $i = \left\lfloor \frac{heap_size}{2} \right\rfloor$ down to 1

Heapify (a, i);

}

~~$T(n) = \frac{n}{2} \times$~~

Exe. Time:

'for' st. $\rightarrow \frac{n}{2} + 1 \rightarrow T(n) = \Theta(n)$

Heapify () $\rightarrow O(\log n)$ [when ~~the~~ heapify () is called for ~~a~~ root node, there is a possibility of $T(n) = O(\log n)$]

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

$\therefore$ Exe. time $= (n \times \log n)$

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

For Best case:

$\boxed{T(n) = \Omega(n)}$ [Best case for heapify is $T(n) = \Omega(1)$]

a = array must contain ele. from 1 to size of the heap.

i/p. to BuildHeap = complete BT

o/p = array rep. of complete BT as max Heap

Prime Merit Data

Scanned with CamScanner

Heap Sort:

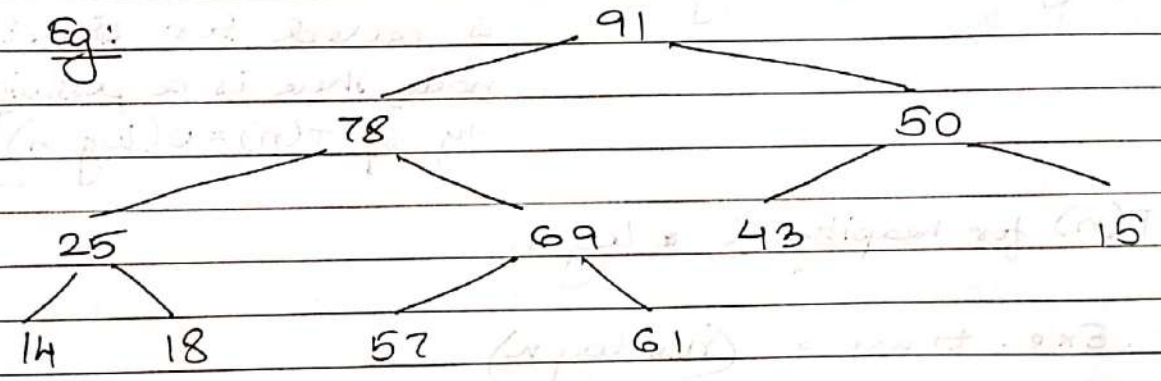
Given a heap str., heapsort can be applied to 2 i/ps:

i) Complete BT ~~[illegible]~~
ii) Max Heap

For complete BT, first call the BuildHeap() to convert into a heap str.

Given a heap str. (max heap / min heap) ~~we~~ we would have to arrange the ele. in ascending / ~~order~~ descending order.

Eg:



a =	91	78	50	25	69	43	15	14	18	57	61
	1	2	3	4	5	6	7	8	9	10	11

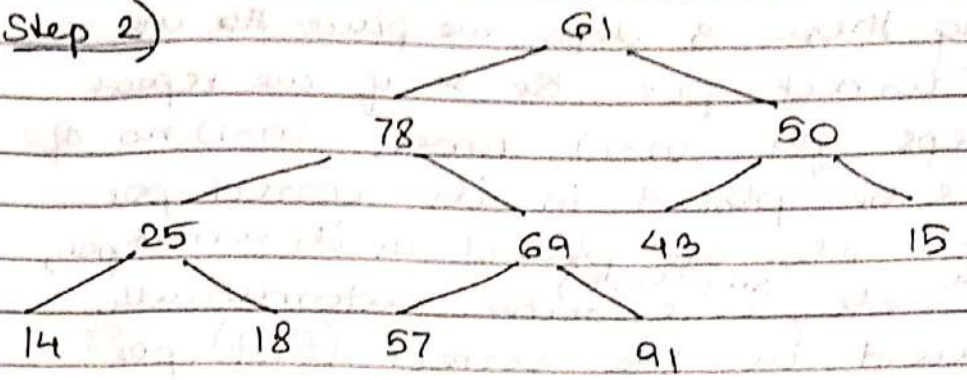
This array satisfies the max heap prop.

~~The fir~~ for sorting in ascending order:

Step1) First ele. is the largest ele. which must appear in the last pos.

~~swap (91,~~ swap (a, 1, 11)

Step 2)



a =	61	78	50	25	69	43	15	14	18	57	91
	1	2	3	4	5	6	7	8	9	10	11

Decrement the size of the heap by 1.
∴ Size of heap = 10

Step 3) ~~Again pass~~ Reheapify this heap for node 1.

heapify (a, 1)

We can heapify node 1 because both of its subtrees ~~are also~~ already follow the heap prop. If 91 is considered, we ~~would~~ could not have ~~made the~~ heapified node 1.

These 3 steps are done in one iteration. So after one iteration, the largest ele. is placed in the last pos. and we will have a heap from 1 to (n-1).

Steps for heapsort:

1) Largest ele. is at a[1]. — assumption
2) Swap a[1] with ele. at a[n].
3) Decrement n, i.e., heapsize (a).
4) Restore heap prop. at a[1] by calling Heapify (a[1])
5) Repeat ~~steps~~ above steps for (n-1) times.

Prime Merit
Page No.
Date

Scanned with CamScanner

By doing these 3 steps, we place ~~the~~ one ele. in its correct pos. ~~So~~ So if we repeat these steps for (n-1) times, (n-1) no. of ele. will be placed in its correct pos. If (n-1) ele. are placed in its correct pos., the nth ele. (smallest) will ~~auto~~ automatically be placed in the correct (first) pos.

<u>Algo. :</u>

```
Heapsort (a)
{
    BuildHeap (a);   // if a → complete BT, call this, else no need to call

    for i = heap_size (a) down to 2
    {
        swap (a[1], a[i]);
        ~~i--;~~ heap_size (a) --;
        Heapify (a, 1);
    }
}
```

<u>Exe. time :</u>

Build Heap() → $O(n \log n)$

'for' st. → $T(n) = n$ (asymptotically)

swap → ~~T(n)~~ 1

heap_size (a) -- → 1

Heapify → $O(\log n)$

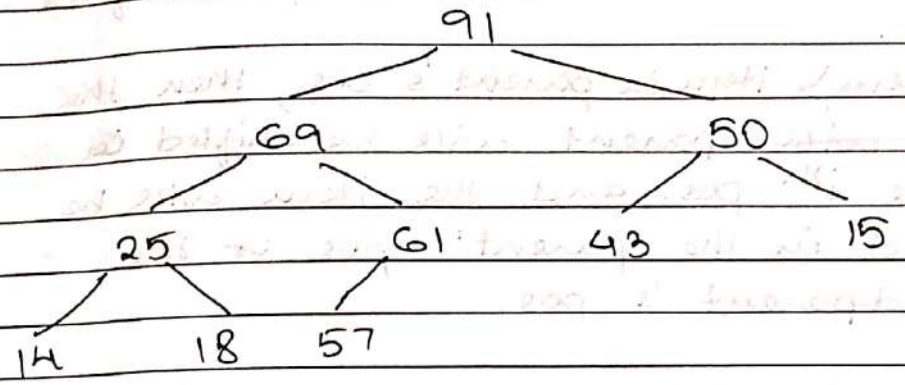
$\therefore T(n) = n \log n + (n \times \log n)$

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

Scanned with CamScanner

Heap Insertion:

Eg: Given a max heap.



a =	91	69	50	25	61	43	15	14	18	57
	1	2	3	4	5	6	7	8	9	10

~~Max h~~

Insert 78 to the existing heap.

Do not insert an ele. into a complete BT which does not satisfy the heap prop.

We are supposed to insert the item in the 11th pos.

~~i = 11 → size, heap_size += 1~~
~~item = 78~~

heap_size++; → 11
i = heap_size; → 11
item = 78;

If the item would have been 17, it could have easily be placed in the 11th pos.

Prime Mark
Page No.
Date

Scanned with CamScanner

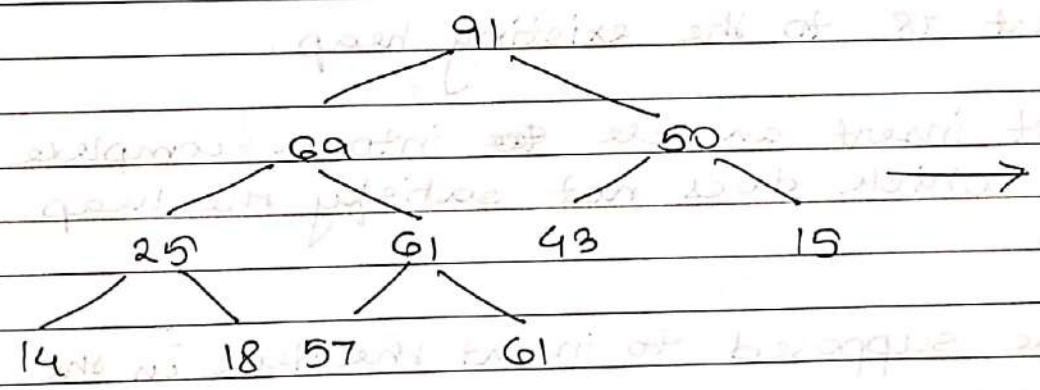
But in this case:

i) Insert in i^{th} pos.

ii) Compare the item with the parent of the ith pos.

iii) If item > item in parent's pos., then the ~~item will~~ parent will be shifted ~~to~~ to the ith pos. and the item will be placed in the parent's pos. or the grandparent's pos.

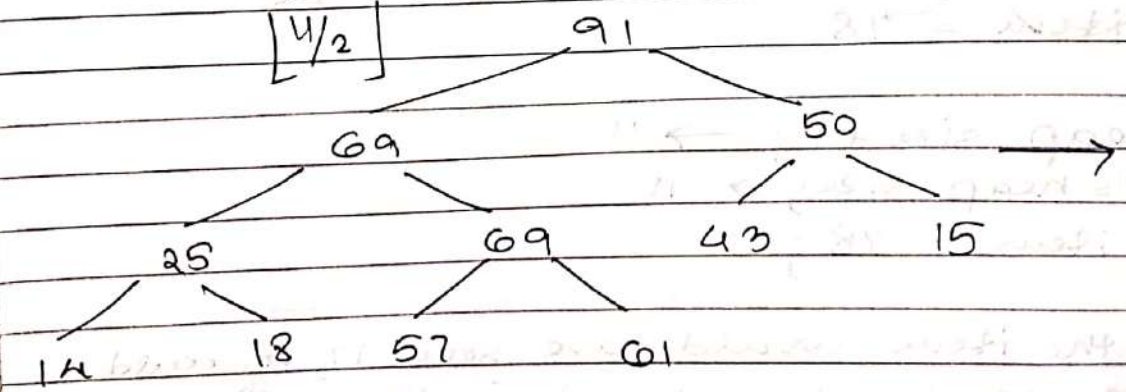
New $i = \left\lfloor \frac{i}{2} \right\rfloor$ → because we tried to insert i^{th} ele. in its parent's pos.



$i = 5$, item = 78

$\downarrow$

$\lfloor 4/2 \rfloor$



item = 78, $i = \left\lfloor \frac{5}{2} \right\rfloor = 2$

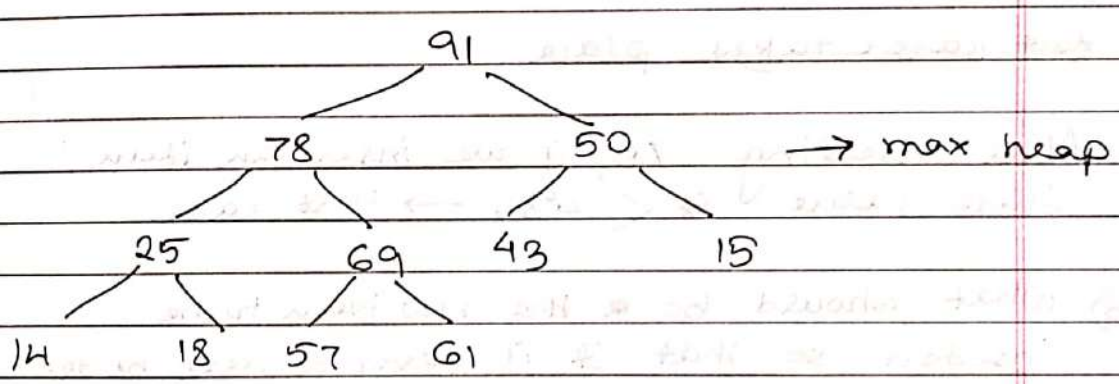
Parent of i = ~~90~~. 91.

item is smaller than parent of i. Hence, item is placed in the ith pos.

This process ~~con~~ ~~99~~ will continue till there exists a parent.

A node has a parent if $\left\lfloor \frac{i}{2} \right\rfloor \geq 1$.

If the item would have been 98, then 91 will be placed in 2nd pos. and 98 in 1st pos.



→ max heap

~~This heap comp~~ Algo.:

```
HeapInsert(a, key)   // a → array, key → item to be inserted
{
    heap_size[a]++;
    i = heap_size[a];
    while (i > 1 AND a[parent(i)] < key)
    {
        a[i] = a[parent(i)];
        i = parent(i);
    }
    a[i] = key;
}
```

$i > 1$ → parent exists

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. Time:

Initialization → 1

The while st. will be executed for max. height of the tree.
Height of a complete BT = log n → order

while = T(n) = log n

∴ $\boxed{T(n) = O(\log n)}$ → ~~Best~~ Worst Case

∴ $\boxed{T(n) = \Omega(1)}$ → Best case

~~Best case takes place~~

After inserting 78, if we insert an item whose value < 43, → best case.

Q) What should be the new item to be inserted so that it is never a leaf node?

Ans) item > 43 (Always draw the tree from the given array).

Q) 2 ele. will be given. What will be the ~~str.~~ contents of the array after inserting the 2 ele. ?

Q) Insert ~~an~~ 2 items and then delete an item. What will be the ~~str.~~ contents of the array?

Q) Find out the sum of the ele. of present in the leaf nodes after inserting 2 items.

Prime Merit
Page No.
Date

Scanned with CamScanner

For any tree, the ~~nodes~~ ele. present in any one of the path from root to leaf are in sorted order.

When we tried to insert an item, we found out ~~the~~ its correct pos. to be inserted by ~~binary~~ linear search. (moved linearly along the path).

For a big tree, linear search → time consuming. Hence, we go for binary search.

No. of nodes present in a path = h
(→ Height of tree)
$\therefore T(n) = O(h)$ → linear search.

For bin. search, $T(n) = O(\log h)$

~~For n no. of do~~
When No. of nodes = n
$\therefore$ Height of tree = $\log n$

~~$\therefore T(n)$ fo~~

$\therefore$ for bin. search, ~~insertion~~ searching time:
~~$T(n) = \log \log$~~

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

But for bin. search, insertion time = $\log n$ because insertion is done linearly. Hence, we do not go for binary ~~for~~ search in ~~bin~~ insertion. ~~Hence, for inserting~~ But for searching it can be applied.

Prime Merit
Page No.
Date

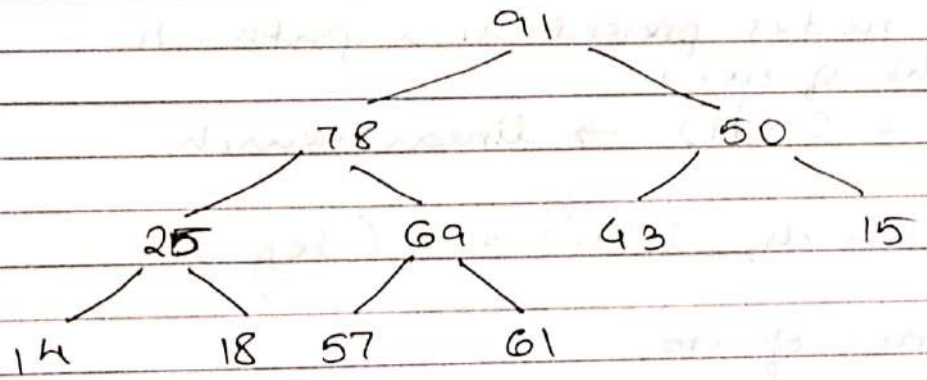
Scanned with CamScanner

8) Find whether the item 78 is present in the tree or not.

Ans: Binary search in tree
$= O(\log(\log n))$

Heap Deletion:

Eg: Delete the largest item from max heap. After deletion, the complete BT must also satisfy the heap prop.



a = | 91 | 78 | 50 | 25 | 69 | 43 | 15 | 14 | 18 | 57 | 61 |

Step 1) Take the largest item (root node) and store it in a variable 'max'.

max = a[1].

Step 2) Bring/copy the last item to the first pos.

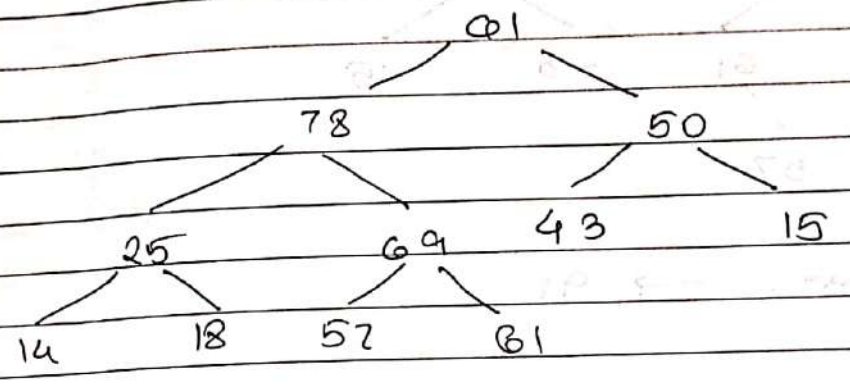
Step 3) Decrement the heap by 1 unit.

Step 4) ~~Re~~ Reheapify a[1] and return the largest value.

Prime Merit
Page No.
Date

Scanned with CamScanner

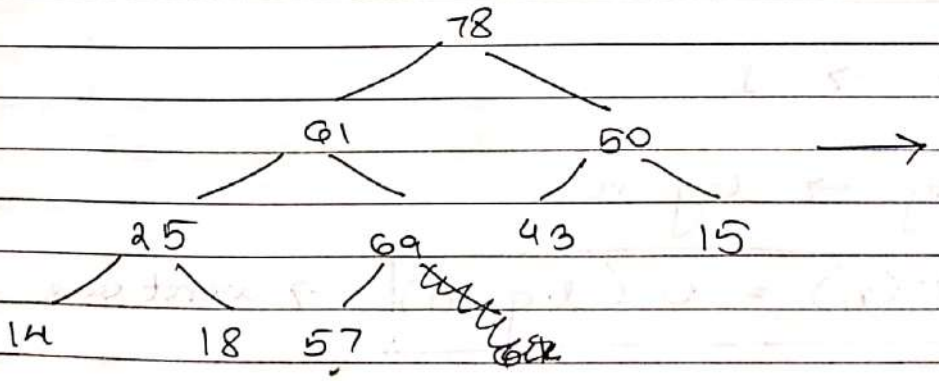
max = a[1] = 91
a[1] = a[heap_size(a)] = 61



heap_size(a) - - → 11 - 1 = 10

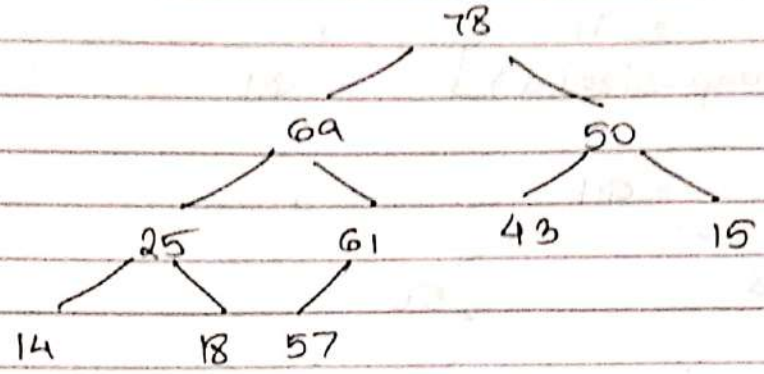
heapify (a, 1) → heapify will be called recursively to prepare a heap str. because we are ~~calling~~ calling heapify for root node.

If 91 and 61 would have been exchanged instead of 61 getting copied → heapify (a, 1) cannot be called because left subtree did not follow heap prop.



Prime Merit
Page No.
Date

Scanned with CamScanner



Return max. → 91

<u>Algo.:</u>

```
HeapExtractMax(a)
{
    if (heap_size[a] < 1)
        error;
    max = a[1];
    a[1] = a[heapsize[a]];
    heap_size[a] --;
    Heapify(a, 1);
    return max;
}
```

<u>Exe. Time:</u>

All st. → 1

Heapify → log n

$\therefore \boxed{T(n) = O(\log n)}$ → worst case

~~$\therefore T(n) = \Omega(1)$~~ <u>Best case</u>: $T(n) = \Omega(1)$.

But if we delete the root node (bring the leaf node to root) → heapify() can never be des $\Omega(1)$.

Scanned with CamScanner

It can only be ~~&~~ $\Omega(1)$ if there is only 1 ele.

If the height of ~~heap~~ tree = ~~n~~ h

$\therefore$ ~~Worst case = $O(\log h)$~~
~~Best case = $\Omega(h-1)$~~

$\therefore$ Worst case = $O(\log n) = O(h)$
Best case = $\Omega(h-1) = \Omega(h)$

~~$T(n) = \Theta(h)$~~

Best case → new ~~no~~ root node becomes inter. node.
Worst case → new node becomes ~~a~~ leaf node

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

Q) After deleting the root node, the ~~dee~~ ~~tree~~ smallest ele. (leaf node) is brought to the root. When we heapify this BT, will this new root node ever appear in a pos. where it is the parent of 2 children? or will it always be a leaf node?

Ans It ~~can~~ may be an intermediate node, but that inter. node will never have 2 children. It can have 1 child.
The height ~~can be~~ of that node can be either 'h' or 'h-1'.

Q) Given 4 arrays. Identify which of them is not a heap str.

Ans Draw tree for all the arrays and then check. Check for both max heap & min heap.

Prime Merit
Page No.
Date

Scanned with CamScanner

7/9/20

Greedy Approach:

There is a choice property. In the choice prop., the ele. of the array are arranged in a particular order.

Greedy & dynamic prog. approaches are used to find optimal soln.

Eg.: Suppose the cities of India are diff. ~~nodes~~ vertices of a graph and the edges rep. the dist. b/w the 2 cities. Find the shortest dist. b/w any 2 places. — optimal soln. prob.

In Greedy Approach:

i) Arrange the items in a sorted order based on some para. which dep. on the prob.

ii) Every time pick up the next item based on the sorted order. ~~Whic~~ while picking up, check whether by including the item, we will get a feasible soln. or not. (Feasibility test)

Eg.: There is a bag which can hold 5kg. If we choose an item = 10kg → not feasible

If the ~~item~~ soln. is feasible, include the item in the soln. vector, which will always give the optimal soln.

If the soln. is not ~~fo~~ feasible → discard the item and choose the next item.

Prime Merit
Page No.
Date

Scanned with CamScanner

General Procedure:

Step 1) Arrange the items in sorted order.

Step 2) Start with a ~~an initial~~ soln. vector which initially is an empty set. Assuming there are 'n' no. of items, start choosing each item from 1 to n. Every time we include an item (i^{th} item) to the existing soln. vector, if

existing soln. vector $\cup$ i $\rightarrow$ feasible.

$S = S \cup i$

If this is not feasible $\rightarrow$ do not include and check next item.

Go on checking ~~all~~ items. Sometimes we do not check all the items if we find that by including any further items we would not get ~~be~~ a feasible soln.

Sorting order criteria
Feasibility Test condition
$\rightarrow$ dep. ~~fro~~ on prob.

```
Greedy_Algorithm ()
{
    Arrange items in sorted order
    S = ϕ        //initialize soln. vector to NULL
    for i = 1 to n do
        if (S ∪ i feasible)
            S = S ∪ i;
    return S;
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

Prob. 1) Fractional Knapsack Prob.

Knapsack → bag / container which can hold ~~some~~ upto some weight.

Prob. St.: Given a set of items, each with a weight and a value, Determine a subset of items to include in a collection, so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Eg.: A farmer has a land in which a no. of vegetables are being produced of diff. variety. He wants to go to the market and he has a bag/knapsack which can carry max. 100kg of items. ~~Every~~ The quantity of every veg. that he sells gives him some profit. (Eg.: 50kg tomato → ₹200 profit, 20 kg potato → ₹90 profit)

Items	Weight	Profit
a	4	28
b	15	75
c	3	27
d	8	32
e	20	120
f	7	70
g	11	44
h	2	6
i	17	136
j	13	26
⋮		

$\boxed{m = 40}$

↳ max. weight that knapsack can hold

Page No.
Date
Prime Mark

Scanned with CamScanner

~~The farmer is interested~~ How can the farmer select the items so that he can get max. profit?

Soln. Way 1: $b \to 15$, $e \to 20$, $d \to 5$

way 2: $e \to 20$, $g \to 10$, $i \to 10$

~~He has~~ He has ∞ options, but he has to choose in such a way, that gives him max. profit.

Applications:

In many cases of resource allocations along with some constraint:

i) Finding the least wasteful way to cut raw materials.

ii) Portfolio optimization

iii) Cutting stock Probs.

Soln.:

$x_i \to$ soln. vector (either fraction of the item or ~~what~~ all items)

$\downarrow$

between 0 and n

$p_i \to$ profit of the ith item

Objective is to maximize $(x_i \cdot p_i)$.

$w_i \to$ weight of the ith item

Constraint $= \Sigma (x_i \cdot w_i) \leq W$

$\downarrow$

max. weight

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Arr~~ Step 1) Arrange the items in decreasing order of their profit : weight ratio.

$$\frac{P_{i+1}}{w_{i+1}} \leq \frac{P_i}{w_i} \quad , P_i \rightarrow \text{profit per unit}$$

Whichever item has the highest profit per unit, that item will be considered first.

Items	Weight	Profit	Select x[]	Sort
a	4	28	1	4
b	15	75	0	6
c	3	27	1	2
d	8	32	0	8
e	20	120	9/20 = 0.45	5
f	7	70	1	1
g	11	44	0	7
h	2	6	0	9
i	17	136	1	3
j	13	26	0	10

(1 → whole, fraction — part of item chosen)

m = 40, weight = 0

→ Total wt. of item that is already selected.

1) First we take item 'f'. We take the entire wt. of the item because 7 < 40.

~~m = 40~~ wt = 7

~~wt = 7~~

2) We take whole of 'c'.

~~m = 40~~ wt = 7 + 3 = 10

Prime Merit
Page No.
Date

Scanned with CamScanner

3) Select whole of 'i'.

$w = 10 + 17$

$= 27$

4) Select whole of 'a'

$w = 27 + 4$

$= 31$

5) Select 9 units of 'e'. → ~~fract~~

$w = 31 + 9$

$= 40$

This is called the feasibility test. → to determine whether or not to take the item and by how much.

Soln. Vector = $\underbrace{a, c, f, i}_{\text{whole}}, \underbrace{e}_{\text{partially}}$

<u>Algo. :</u>

<u>i/p</u> : i) no. of items
ii) weight
iii) profit
iv) Max. weight → m

<u>o/p</u> : Soln. Vector

Prime Merit
Page No.
Date

Scanned with CamScanner

```
Fractional_Knapsack (w[], p[], m, n)
{

    //arrange the items in decreasing order of
      profit : weight , i.e., pi/wi >= pi+1/wi+1

    ~~x[i] = 0 for i =~~

    for i = 1 to n do
        x[i] = 0;          //initialize soln. vector
                             to zero
    weight = 0;            //curr. wt. of knapsack

    for i = 1 to n do
    {
        if weight + w[i] <= m
        {
            x[i] = 1;
            weight += w[i];
        }
        else
        {
            x[i] = (m - weight) / w[i];
            weight = m ;
            break;
        }
    }
    return x;
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. Time:

This algo. will work if the items are in sorted order.

~~For sorting, best $T(n) = \Theta(n \log n)$~~

~~Best case~~ exe. time for best sorting algo.
$= \Theta(n \log n)$

~~Exe. time for this algo. : $T(n) = \Theta(n)$~~

Exe. time of this algo.:

first 'for' $= \Theta(n)$

Second 'for' $= O(n)$

$\therefore T(n) = \Theta(n) + O(n)$ ($\Theta(n) = n$, $O(n)$ = max. value n)
$= \Theta(n)$

$\therefore$ Total exe. time:
~~$T(n$~~
$T(n) = \Theta(n) + \Theta(n \log n)$
$= \Theta(n \log n)$

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

(0-1 knapsack)
There is a digital knapsack where either the item will be taken fully or the item will not be taken at all. For this, we use dynamic prog. Greedy approach does not give soln. for ~~dynamic~~ digital knapsack.

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Probs. on Heap Data Str.:~~

Heap Data Structure:

Q1) Data will be given. Create a heap by inserting one ele. after another.

Q2) Data/BT will be given. Prepare a min heap.

Q3) Delete 2 items and show the existing heap. ~~(Sho~~ (Delete one ele. after another and show the heap).

~~Q(4)~~ Necessities for Heap DS:

Insertion and deletion (of largest/smallest) op. are used in many app. like short term scheduling in OS or Disk scheduling.

For unsorted array:

Insertion = $\theta(1)$
Delete the largest item = $\theta(n)$

Insertion occurs for any data. But deletion is always for largest / smallest.
Eg.: SJF or priority scheduling

For sorted array:

Insertion = $O(n)$ → insertion sort
Deletion = $\theta(1)$
↓
largest

Prime Merit
Page No.
Date

Scanned with CamScanner

For Heap :

Insertion = $O(\log n)$
Deletion of largest = $\Theta(\log n)$

If n no. of ele. are inserted, we have to delete 'n' no. ele. also. (app. uses)

Eg. : n = 1024

For unsorted : insert = 1
delete = 1024

For sorted : insert = 1024
delete = 1

For heap : insert = $\log 1024 = 10$
delete = $\log 1024 = 10$

As 'n' becomes larger, heap will give a better result in case of computation./exe. time. (Only when deletion is of largest/smallest)

Q) Why is heap called as a priority queue? Why is heap used in priority queue app.?

Soln.: It is because based on priority either the largest/smallest ele. is deleted. Heap finds its app. in priority queue, shortest seek time first, SJF, priority scheduling, etc.

Prime Merit
Page No.
Date

Scanned with CamScanner

Probs. on Heap DS:

Q1) How many exact comparisons are reqd. to extract the smallest item from max-heap? What is its running time?

Soln. For max heap:

smallest item → leaf node (max heap)

No. of leaf nodes = $\lfloor n/2 \rfloor$

leaf nodes start from = $\left\lfloor \frac{n}{2} \right\rfloor + 1$

The leaf nodes are not necessarily in sorted order.

So we need to linearly search only the last $\frac{n}{2}$ ele. in the array.

~~∴ T(n)~~

~~∴ No. of comparisons~~

∴ Exact no. of comparisons = $\frac{n}{2}$

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

For deleting the smallest item = $\Theta(\log n)$

For deleting n items = $(n \log n)$

↳ each time deleting the smallest

Prime Merit
Page No.
Date

Scanned with CamScanner

Q2) How many comparisons are reqd to insert an item onto a heap, if binary search is reqd. instead of linear search?

Soln. Height of tree = $h = \log n$

$\therefore$ No. of comparisons reqd. to search for correct pos. $= O(\log h)$
$= O(\log(\log n))$

But total insertion time $= O(\log n)$

So searching for the pos. where the node will be inserted is $O(\log(\log n))$ but the ele. is pushed linearly to insert in the correct pos. ~~hence~~ taking time $O(\log n)$

Q3) Merge 2 max heaps into a single max heap. What is its running time, if sizes of the heaps are p and q?

~~Soln: Combine the 2 heaps.~~
~~$\therefore$ Size of ~~new~~ single heap~~

Soln: ~~Do~~ ~~copy~~ Add the ele. of 2nd array (heap) to the first array at the end.
$\hookrightarrow T(n) = \Theta(p)$ or $\Theta(q)$
$\therefore$ Size of new array $= p+q$

Call the Heapify() func.

$\therefore T(n) = (p+q)\log(p+q)$ (Ans)

Prime Merit
Page No.
Date

Scanned with CamScanner

Q4) Merge n no. of sorted arrays of size log n each to a single sorted array. Find its computation time.

Soln. All the sorted arrays are max heap.

Let n = 7 and each array is of size 100. and all the sorted arrays are in decreasing order.

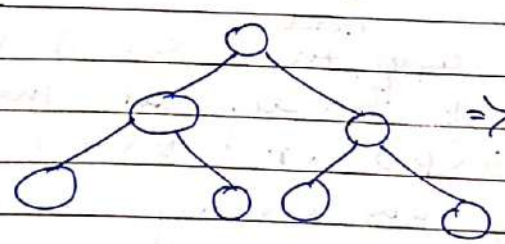
∴ Resultant array size after merging
= (7 × 100)
= 700

While searching for the largest ele. in the resultant array we have to check the first ele. of every array.

For preparing a heap of 7 ele.,
height of complete BT = $\log_2 7$

≈ 3

If height of root = 0, height of leaf nodes = 2



⇒ h = 2

Prepare a max heap str. by taking the first ele. of each of the arrays.

The root node will have a pointer not a data ele. pointing to the first pos. of the array which is holding the largest item.

Root node — structure.

Prime Merit
Page No.
Date

Scanned with CamScanner

Prepare a heap of 7 data ele., taking the first ele. of each array. So every node has a pointer, pointing to ~~an array~~ the first ele. of an array.

So the root node will have the array whose first ele. is largest among all items.

~~If~~ If we apply deletion, to this heap,
no. of nodes = 7
After deletion, no. of nodes = 6

For deletion: $T(n) = \Theta(\log 7)$

So after preparing a heap:

Step 1) Delete a node (root node (largest ele.)) $\rightarrow T(n) = \Theta(\log 7)$

Step 2) After deletion, the first item (largest) present in the array will be stored in the resultant array.

Step 3) Increase the pointer by 1 unit
↓
pointing to the ~~largest~~ first ele. of array which is the largest.

Taking the second ele. of this array, insert it into the heap from which the root node was deleted. It may be placed in the root / leaf / inter. node.

Exe. time for insertion: $T(n) = \Theta(\log_2 7)$

Prime Merit
Page No.
Date

Scanned with CamScanner

∴ Total exe. time for 3 steps :

$$T(n) = \log 7 + 1 + \log 7$$

$\log 7$ → deleting root node

1 → Insert in resultant array

$\log 7$ → Insert next higher item from the array having largest ele.

After doing 3 steps, no. of ele. brought into resultant array = 1

∴ No. of times these 3 steps need to be performed to bring all the ele. into the resultant array = 700

∴ Total exe. time = $700 (\log 7)$

<u>For the given prob.:</u>

Total no. of ele. in the resultant array = $n \log n$

∴ Insertion = $\log n$
Deletion = $\log n$

∴ Total computation time:

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

Q5) Find the computation time to merge $\log n$ no. of sorted arrays each of size n to a single sorted array.

∴ ~~T(n) =~~

Soln. Total no. of ele. = $n \log n$

$\therefore$ Insertion = $\log(\log n)$
Deletion = $\log(\log n)$

$\therefore T(n) = n \log n \cdot \log(\log n)$

Q6) Find the computation time to merge $\log n$ no. of sorted arrays, each of size $\log n$ to form a single sorted array.

Soln. Total no. of ele. = $\log n \cdot \log n$

$\therefore$ Insertion = $\log(\log n)$
Deletion = $\log(\log n)$

$\therefore T(n) = \log n \cdot \log n \cdot \log(\log n)$

Q7) In which order should we merge 3 sorted arrays A, B and C of sizes 20, 50 and 100 ele., resp., so that the no. of comparisons would be the least?

Prime Merit
Page No.
Date

Scanned with CamScanner

Greedy Approach Probs.:

Prob. 2) Activity Selection Problem:

Suppose in a college annual func., there is a ~~single stage~~, single pandal and lots of individuals/groups are interested to perform some activity on the pandal ~~stage~~. Every activity has a start time and also has an end time. In 1 day, the whole 24hrs are given to the performers.

Eg: CS1 student performs an activity ~~and the~~ which has a start time of 10am and an end time of ~~12am~~. 12pm.

CS2 student wants to perform an activity from 2pm to 6pm.

CS3 student wants to perform an activity from 14:00 to 15:00.

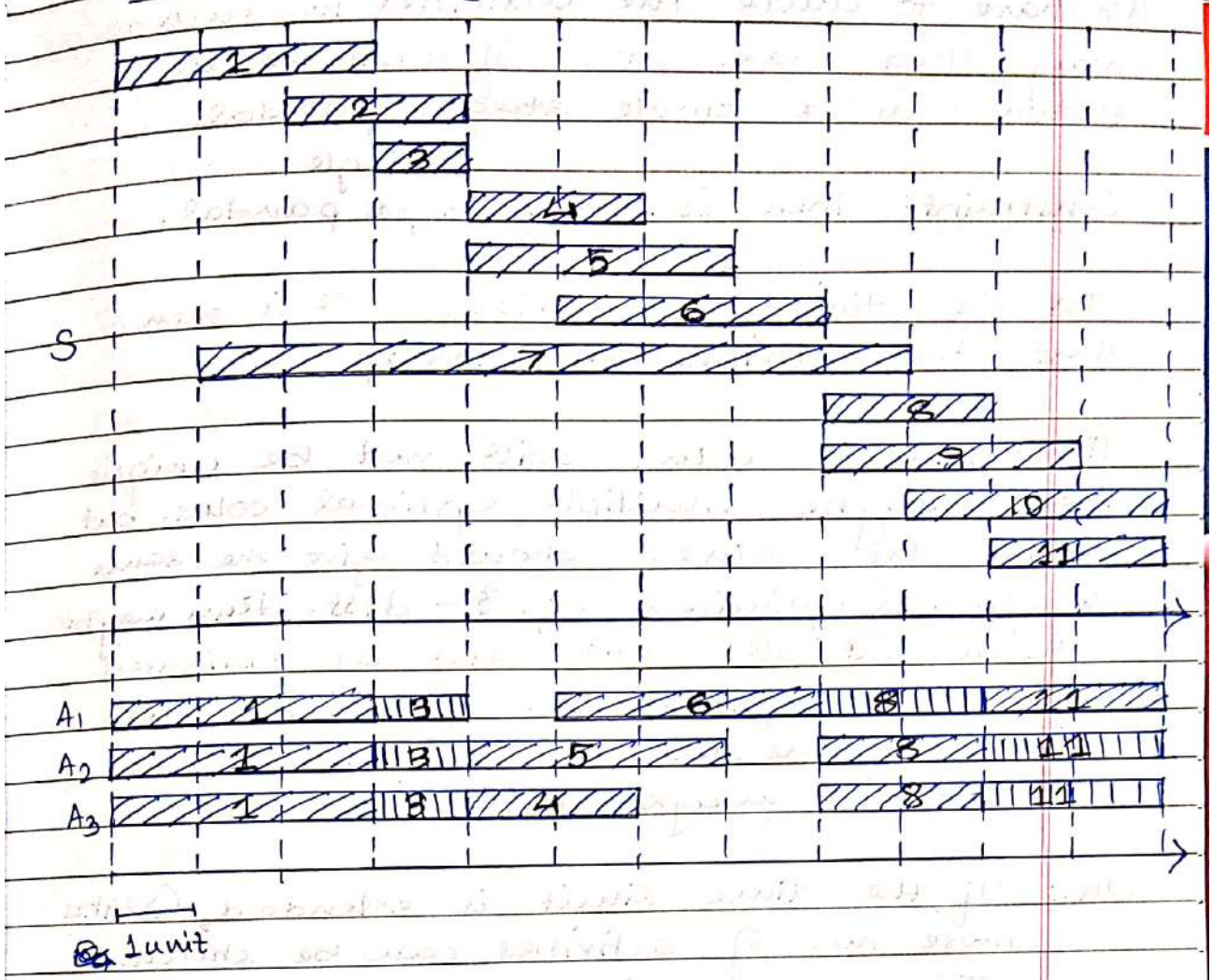
In this way, diff. group of students have diff. start time and end time.

Activity	Start Time	End Time	~~Select~~
a	4	9	
b	15	18	
c	3	6	
d	8	10	
e	20	24	
f	7	9	
g	11	14	
h	2	7	
i	17	19	
j	3	16	

Prime Merit
Page No.
Date

Scanned with CamScanner

Time frame diag.:



Each activity has to be ~~starde~~ started and ended ~~at~~ in the time specified. ~~no ots~~

Eg: If a activity is scheduled from 4pm to 6pm, it has to ~~do~~ ~~be started and end~~ follow this time. ~~we cannot~~ Since the activity takes 2 hrs, we cannot schedule it from 10pm to 12am.

Objective: To satisfy more group of students. ~~no of~~ No 2 activities should coincide with ~~at~~ ~~no~~ other. → (2 activities cannot take place simultaneously). If one activity ends, only then the second activity can be started.

Prime Merit
Page No.
Date

Scanned with CamScanner

We have to choose the activities in such a way that max. no. of activities are shown in a single ~~stage~~. pandal.

<u>Constraint</u>: There is only a single pandal.

In the time frame diag., it is seen that the activities ~~co~~ overlap.

The optimal soln. will not be unique. There may be multiple optimal solns. but all the solns. should give the same o/p. (optimised o/p. — diff. items may be chosen, but all should give an optimised o/p.) The optimal no. of activity must remain same but the type/name of the ~~the~~ activity may be diff.

<u>Soln.</u>: If the time limit is extended (>24hrs) more no. of activities can be shown. This is our principle

Based on this principle, we should choose items that ~~end~~ can be ended quickly — ending time is least. If an activity ends earlier, we ~~can~~ will get more time to accomodate more activities.

<u>Step 1</u>: Sort the activities in increasing order of their finish time.

~~Step 2:~~ When a new activity is selected, check whether its start time ~~is~~ ~~st~~ is greater than or equal to prev. activity's finish time.

Prime Merit
Page No.
Date

Scanned with CamScanner

Step2) new activity start time $\geq$ prev. activity's end time (already selected)

only then the new activity is included in the soln. vector.

For feasibility test: Everytime only one activity is selected. If the prev. cond. satisfies, we will include it in the soln. vector. otherwise discard it.

The activities in the time frame diag. are already ~~included~~ sorted in asc. order.

The feasibility test ~~have~~ do not have to be performed for selecting the first activity (first item of the array) because there is no prev. activity in the soln. vector.

Soln. Vectors: A_1, A_2, A_3 → all of them contain 5 activities

Activity	Start Time	End Time	Select
a	4	9	2nd → ×
b	15	18	4th
c	3	6	1st
d	8	10	3rd → ×
e	20	24	5th
f	7	9	2nd
g	11	14	3rd
h	2	7	2nd → ×
i	17	19	5th → ×
j	13	16	4th → ×

× – feasibility test failed

∴ Soln. Vector = c, f, g, b, e

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Q) There 10 or 11 total no. of activities.~~

Q) i) Suppose there are 400 groups of students who want to perform on a single pandal and each one of them ~~have~~ has been given a start time and an end time. If all the ~~studs~~ groups perform on a single pandal, only 5-6 groups will be satisfied. So what to do about the large no. of unsatisfied groups?

Soln1) The no. of pandals are increased.

Q ii) How many min. pandals are reqd. to show all activities?

Soln.) The activities which are selected for ~~the~~ pandal 1 are discarded from the list. From the rem. activities, find out the activities that can be shown in ~~a~~ pandal 2. Repeat this step till all the activities have been covered.

Let the no. of activities in pandal 1 = 5

$\therefore$ No. of activities in pandal 2 ≤ 5

The no. of act. in the optimal soln. for the subsequent pandals will be less than or equal to that of prev. pandals.

Drawback: A student who wants to see activities of 2 pandals will not be able to ~~see if there~~ unless there is a single pandal.

Prime Mark
Page No.
Date

Scanned with CamScanner

Soln. 2) Increase the no. of days. ~~The~~ Activities will be performed on a single ~~po~~ pandal but ~~on multiple days~~ the no. of days will be increased, i.e., the time limit is increased.

If the ~~tim~~ no. of days is increased to more than 7days, no one ~~o~~ will be interested to see the ~~per~~ activities.

So in that case, we have to put 2 parameters:

i) No. of days is limited, i.e., the time limit cannot exceed a certain value.

ii) ~~Aft~~ ~~Then~~ Applying the 1st para., find the no. of pandals.

Algo. :

i/p : i) n → call by value
ii) start time
iii) end time } → call by ref.

o/p : Soln. vector → binary array.
If selected → 1
If not selected → 0

Initialization : ~~soln. vector [~~
soln. vector [] = 0
↓
size = n (no. of activities)

Assume that the items are arranged in increasing order of their end time. Choose the first item w/o performing any feasibility test.

Prime Merit
Page No.
Date

Scanned with CamScanner

~~ActivitySelection (x, s, f, n)~~ (soln. vector, start time, finish time, no. of activities)

~~{~~

~~x[1] = 1;~~ // 1 → include, 0 → discard

i) ActivitySelection (x, s, f, n) (soln. vector, start time, finish time, no. of activities)

```
{
    for i = 1 to n do
        x[i] = 0;          //initialization
    x[1] = 1;   // 1 → include, 0 → discard
    k = 1;   // k is the last activity which has
                been included in the soln.
                vector.

    for i = 2 to n do
    {
        if s[i] >= f[k]
        {
            x[i] = 1;
            k = i;
        }
    }
}
```

Exe. Time:

$T(n) = \Theta(n \log n)$ → reqd. for sorting

The above algo. is for finding out the optimal soln. for 1 pandal 1.

Prime Merit
Page No.
Date

Scanned with CamScanner

ii) Algo. for finding out the no. of pandals req'd. for accomodating all the activities:

Call the prev. algo. once. ~~from the second calling onwards, check~~ After calling it once, the activities which have been selected, have been made '1' in the soln. vector. So everytime ~~when the~~ in the for loop, add another condition, that if the activity has not been selected till now, only then compare the start time with the prev. selected activity's end time.

```
ActivitySelection(x, s, f, n)
{
    for j = 1 to n do
    {
    ~~x[1] = 1;~~  if(x[j] == 0) {
    ~~k = 1;~~        x[j] = 1;
                      k = j; break;
    }              }
    for i = j+1 to n do
    {
        if x[i] == 0 && s[i] >= f[k]
        {
            x[i] = 1;
            k = i;
        }
    }
}

main()
{
    for i = 1 to n do
        x[i] = 0;
    while (true)
    {
```

```
main ()
{
    for i = 1 to n do
        x [i] = 0;      //initialization of soln. vector
    count = 0;
    while (true)
    {
        ~~Activity~~
        ActivitySelection(x, s, f, n);
        count ++ ;

        ~~for(i =~~ flag = 0;
        for i = 1 to n do
        {
            if (x[i] = = 0)
            {
                flag = 1;
                break;
            }
        }
        if (flag = = 0)
        {
            ~~not print~~
            print (count);
            return;
        }
    }
}
```

Steps for algo.:

1) Sort the activities by increasing order of finish time.
2) Schedule the first activity.
3) Schedule the next activity which starts after prev. activity finishes.
4) Repeat until no more activities are there.

Prime Marie

Scanned with CamScanner

Prob. Statement:

The activity selection prob. is to select the largest subset of mutually compatible activities

- Let S be $\{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use the same res. Only 1 activity can use the res. at a time.

- Activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i \leq f_i < \infty$.

- Activities a_i and a_j are compatible if the intervals (s_i, f_i) and (s_j, f_j) do not overlap.

Q) ~~The acti~~ Can there be an algo. for activity selection prob. such that the exe. time is of order 'n'? Assume that the activities are arranged in increasing order of their start time.

Soln.: Principle: To select more activities, there should be more time. i) Activity that is finished earlier is selected first.
ii) Activity that starts late is selected first — more time is available earlier.

So here, start selecting the ~~activies~~ ~~activities~~ activities from n down to 1.

For A_1: 11, 8, 6, 3, 1

Scanned with CamScanner

Since the ~~array is~~ order of activities is already given in asc. order of their start time, hence, no more sorting is req.d.

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

~~The nth item~~ Selection of items will be done from n to 1. The nth activity will certainly be selected w/o any feasibility test.

<u>Cond.</u> : $f[i] \leq s[k]$

~~where k =~~

<u>Algo.</u> :

```
ActivitySelection (x, s, f, n)
{
    x[n] = 1;  for i = 1 to n do
    k = n;         x[i] = 0;

    x[n] = 1;
    k = n;
                  down
    for i = n - 1 to 1 do
    {
        if f[i] <= s[k]
        {
            x[i] = 1;
            k = i;
        }
    }
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Suppose an event management team~~

Applications:

1) Event Management Team
2) Multiple Seminars taking place in college – find the no. of rooms reqd.

6/9/20

Huffman Coding:

Coding / Decoding is normally used in case of comp. / comm. Suppose there is a character set (a, b, ..., z), 1, 2, 3, ...). ~~an~~ We do a coding sys. in order to rep. these characters in binary bits. Given a binary code, we ~~could~~ should also be able to extract the original characters. ~~This is called coding-decoding / encoding - decoding.~~

Given a set of characters, we encode it in binary strings → coding / encoding

Given a binary string, we must be able to decode it to get the original strings → decoding.

It is reqd. in comm. because ~~in the d.~~ when we store info. in a disc, we store it in binary bits of 0, 1. When we pass this info. through a comm. channel, we pass these binary bits.

Prime Merit
Page No.
Date

Scanned with CamScanner

Objective of coding: Rep. the code using min. no. of bits.

Eg: ~~No. of~~ characters = a, b, c, d
∴ No. of min. binary bits reqd. to rep. these codes = 2 (00, 01, 10, 11)

Eg: No. of characters = 16
∴ No. of bits reqd = $\log_2 16$
= 4

This type of coding is called equal length coding. This means that every symbol / char. is coded with equal no. of bits, where each code is unique.

~~Eg:~~ Q) If there are 100 char. ~~file~~ in a file and the no. of diff. char. are 16, then what is the size of file in binary bits?

Soln) No. of diff. char. = 16
∴ No. of bits = 4

~~If the~~ No. of chars. in file = 100
∴ Total no. of bits reqd. = 4×100
= 400 (Ans)

Q) Can ~~these~~ this 100 char file be rep. in less no. of bits?

Soln:) Instead of going for equal length coding, we opt for an unequal length coding. This is called data compression.

Prime Merit
Page No.
Date

Scanned with CamScanner

Fixed length code: Each char. in a file is rep. by a diff. fixed length code.

Variable length code: Each char. in a file is rep. ~~w~~ by a diff. length code.

~~If there are 10 no. of diff. symbols,~~ one sym

Adv.:

1) Some bits maybe rep. in 1 or 2 bits, which in ~~fixed~~ fixed length coding would have taken 4 or 5 bits. So the ~~to~~ total no. of bits reqd. decreases.

In var. length coding, chars. that are freq. used → less no. of bits
Less freq. used char. → more no. of bits.

By doing this, possibly the length of the file would be less.

Decoding Var. length codes?

Eg.:

	A	B	C	D	E	F	
Freq.	5	2	3	4	10	1	
Fixed length codeword	000	001	010	011	100	101	→ 3 bits
Var. length codeword	111	1001	101	110	0	1000	

No. of diff. char. = 6
Total no. of char. = (5 + 2 + 3 + 4 + 10 + 1)
= ~~30~~ 25

Prime Mark
Page No.
Date

Scanned with CamScanner

For 25 char. file, for fixed length coding,
No. of bits = (25×3)
$= 75$

Decoding in fixed length coding → easy

BAD = $\underbrace{001}_{B}\underbrace{000}_{A}\underbrace{011}_{D}$ → 9 bits

In Var. length encoding, the code ~~should~~ must satisfy ~~be a~~ prefix code property.

Prefix Code: None of the codes should be a prefix to any other code.

~~BAD = 1001 111~~
BAD = 1001111110 → 10 bits

~~Ho~~ Decoding procedure:

i) After every sym. (seq. of 0, 1), put a dot.
BAD = 1001.111.110.
But these dots will also have to be rep. in binary. → more no. of bits reqd.

ii) Go on reading from the 1st bit, whenever a bit is found, check whether it rep. a code or not. If it does not rep. a code, go the next bit and combining the bits, check whether it rep. a code. Go on combining the ~~codes~~ bits till a code is ~~data~~ found. Then from the next bit, follow the same steps.

Prime Merit
Page No.
Date

Scanned with CamScanner

Let B = 0001. After reading the first 0, it is not understood that whether it is E or it could be B.

Hence, prefix coding is ~~applie~~ used to make the decoding possible (not easier). This decoding is a bit complicated and takes more time than that of fixed length coding.

We still use it because the comp. can decode ~~it in a~~ it at a very fast rate. Our objective is to have less storage space and to decrease the BW of the channel. Hence, we go for variable length coding.

So in var. length coding,

No. of char. in file = 25

No. of bits = $(5\times3) + (2\times4) + (3\times3) + (4\times3) + (10\times1) + (1\times4)$

$= 58$

$\therefore$ Percentage decrease in no. of bits used

$= \frac{75-58}{75} \approx 20\%$.

In case of a file size = 5 GB,

No. of bits reqd = (20% of 5) GB

= 4 GB

$\Rightarrow$ A lot of mem. is saved.

There is no guarantee that var. length coding would always take less no. of bits. It is only possible ~~on~~ when ~~the~~ few char. are used more freq. and ~~while~~ others are used less freq. If the char. are ~~used in the same freq.~~ used in more or less same freq. $\rightarrow$ no adv. moreover we would get some disadv.

Prime Merit
Page No.
Date

Scanned with CamScanner

There are diff. types of var. length coding ~~techniques~~ schemes, one of which is Huffman Coding.

Huffman coding:

It is a var length coding technique which uses prefix coding properties.

Prefix Coding:

i) Each code has a unique prefix.
Eg: 0, 101, 100

ii) Prefix constraint: The prefixes of an encoding of one char. cannot be equal to a complete encoding of another char.

iii) Decoding is never ambiguous.
- Identify the first char.
- Remove it from the file & repeat

Eg:

Character	Frequency	Code
a	5	
b	9	
c	12	
d	13	
e	16	
f	45	

Σ Freq. = 100

∴ In a 1000 ~~file~~ char. ~~the~~ file, a = 50, b = 90, c = 120, ...

Prime Merit
Page No.
Date

Scanned with CamScanner

Significance of freq. = ~~The freq. Acc. to~~ The freq. would vary proportionately vary with file size.

For equal length code,
no. of bits = $(\log_2 6 \times 100)$

$= 3 \times 100 = 300$

Generatio of Code Using Huffman Technique:

Step 1) Arrange the freq. in asc. order.

OR

Prepare a Min Heap for the chars. taking their freq. into consideration.
— Initialization

Step 2) Delete 2 nodes one after another from the heap. First node deleted — x, Second node deleted — y.

Step 3) Create a new node 'z' whose freq. will be the sum of the freq. of x and y.

Step 4) 'x' will become the left subtree of the newly created node 'z', 'y' will be the right subtree of the newly created node 'z'. ~~This~~ Then this node 'z' will be inserted onto the heap.

~~Continue~~ Repeat steps 2 to 4.

$x \rightarrow$ left subtree
$y \rightarrow$ right subtree
] $\rightarrow$ can be reversed

Prime Mart
Page No.
Date

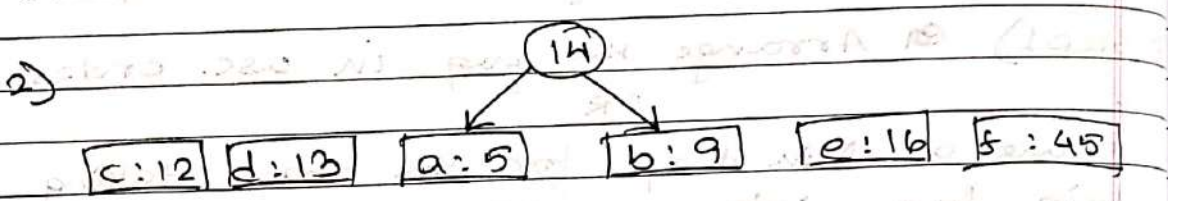
Scanned with CamScanner

Original no. of nodes in heap = 6
After deleting and inserting z, no. of nodes = (6 − 2 + 1)
= 5

For code generation, there will be a tree generated from min heap. ~~generated~~

1) [a : 5] [b : 9] [c : 12] [d : 13] [e : 16] [f : 45]

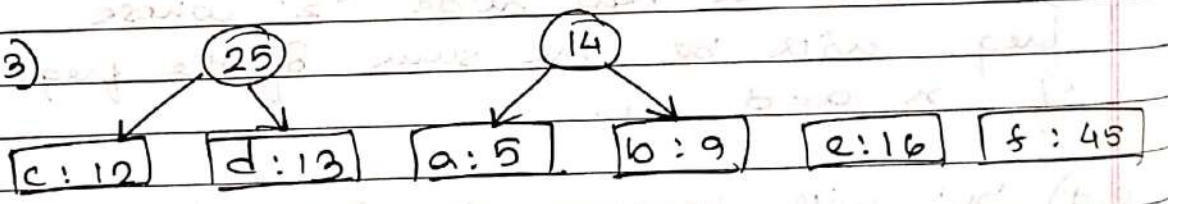
2)



~~Heap nodes = c, d c:12, d:13, 14,~~

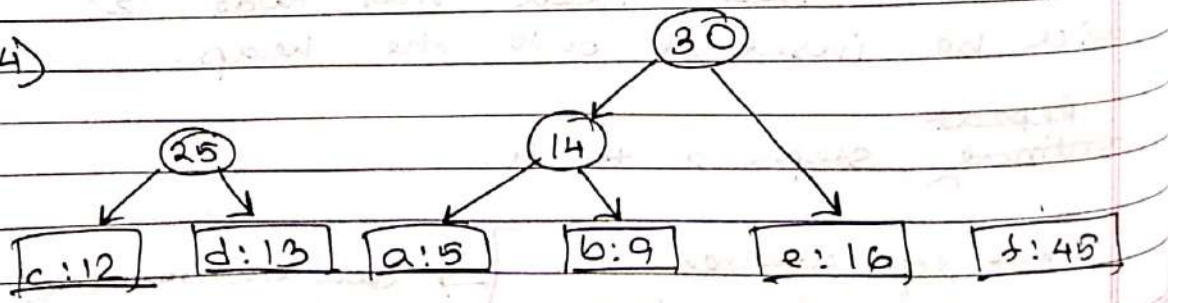
Heap nodes freq. = 12, 13, 14, 16, 45
Tree nodes freq. = 5, 9

3)



Heap nodes freq. = 25, 14, 16, 45
Root node in Heap = 14

4)

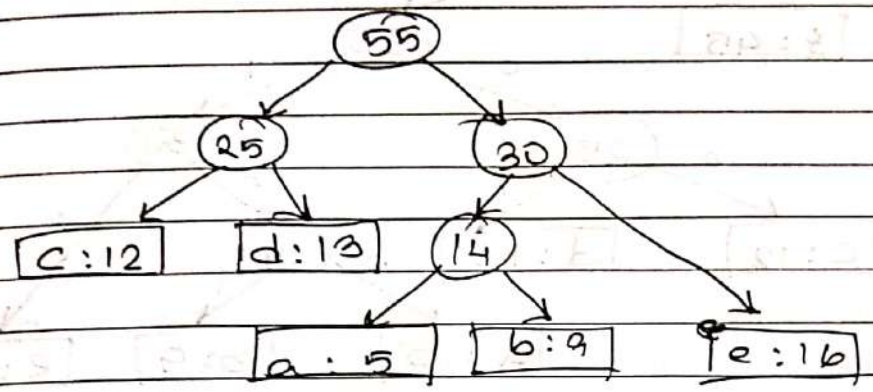


Heap nodes freq.
= 25, 30, 16, 45

Prime Merit
Page No.
Date

Scanned with CamScanner

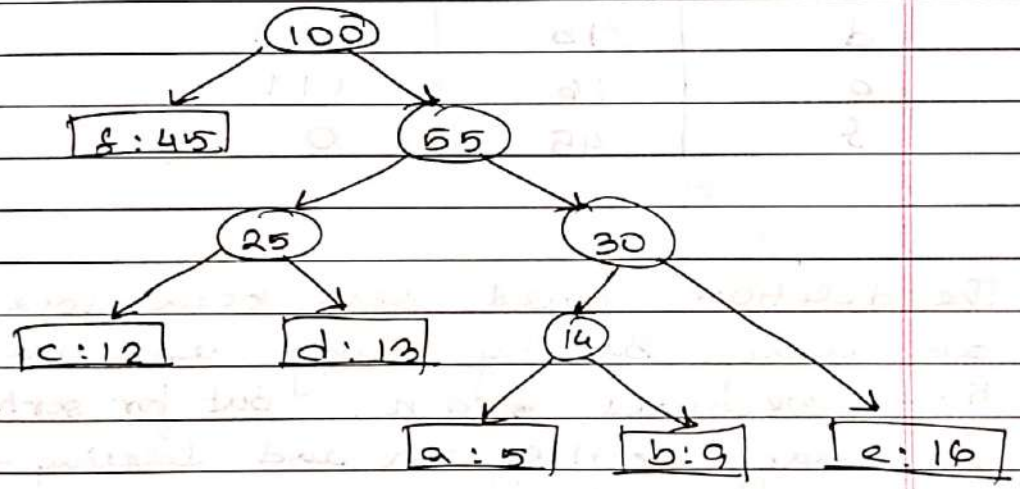
5) f : 45



Heap nodes = freq. = 55, 45

~~In the heap, nodes = freq.~~
~~Leaf nodes =~~

6)



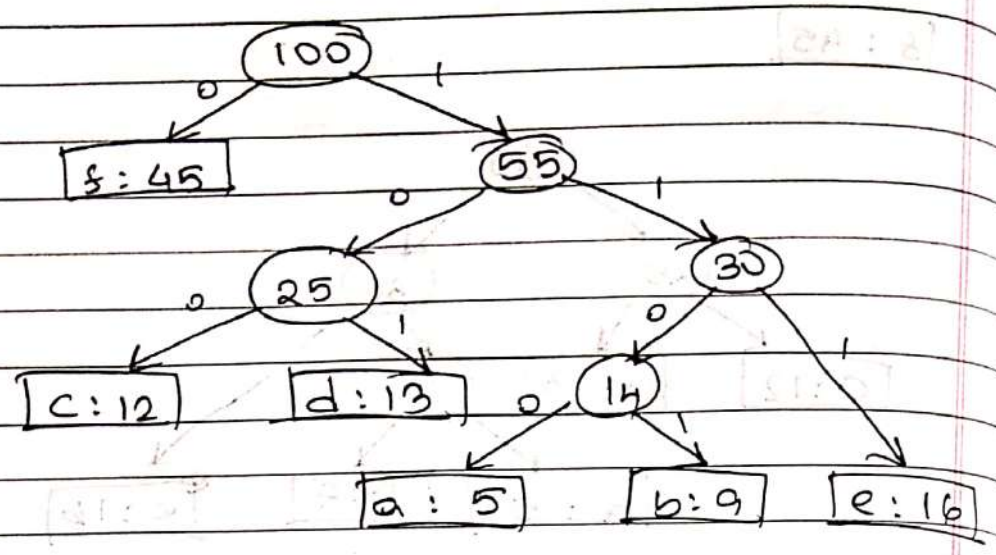
Intermediate nodes → freq.
Leaf nodes → symbols & freq.

This ~~to~~ binary tree is called a ~~Ho~~ Huffman Tree.

In this tree,
While moving left, assign → 0
While moving right, assign → 1
] can be reversed

Prime Merit
Page No.
Date

Scanned with CamScanner



Character	Freq.	Code
a	5	1100
b	9	1101
c	12	100
d	13	101
e	16	111
f	45	0

The deletion could have been done in an array. But by using heap, the exe. time becomes log n. But for sorting the array → n log n and deleting → 1

No. of times the nodes needs to be deleted (2 nodes deleted everytime)
$= (n-1)$, n = no. of nodes in heap.

Eg 2 nodes → deletion 1 time
3 nodes → deletion 2 times
⋮
n nodes → deletion $(n-1)$ times

Prime Merit
Page No.
Date

Scanned with CamScanner

Algo. for Creating Huffman Tree:

~~Huffman (c)~~

i/p : i) Freq.
ii) ~~ch~~ characters

Implementation: array containing the freq. of chars. ~~an~~ where each index rep. each char. Allocate every char. to an index.

By doing this, the ~~an~~ o/p will also be w.r.t. index.

So ~~there~~ another array is created

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

f =	5	9	12	13	16	45
	1	2	3	4	5	6

→ this array is passed to the func

~~Huffman (c)~~ // c → freq. array
~~{~~
~~n = |c|;~~ // length of array
~~[S = c ;]~~ // Prepare a min heap str. for all symbols/chars. taking freq. as a parameter.

func. ~~to prepare~~

The min heap is rep. as an array. The ele of the array are not integers. It is a structure having 2 data members:
i) character
ii) Frequency

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Huffman (c)~~

~~{~~

initialization

~~n = |c|;~~ // length of array

~~Q = c;~~ // Prepare a min heap str. for all symbols or chars. taking freq. as a parameter

~~for i = 1 to n-1 do~~

~~{~~

~~z = ALLOCATE_NODE();~~ // create a node

z → intermediate node of complete binary tree (Huffman tree). It will have 3 fields:

i) Data field (integer) → freq. - (not float)

ii) 2 pointers - pointing to left & right subtree, resp.

The intermediate node will not keep any symbol. (Only the freq.)

HUFFMAN (c)

{

initialization

n = |c| // length of array

Q = c; // Prepare a min heap str. for all symbols or chars. taking freq. as a parameter.

func. (not str.)

for i = 1 to n-1 do

{

z = ALLOCATE_NODE(); // create a node

left[z] = x = EXTRACT_MIN(Q);

right[z] = y = EXTRACT_MIN(Q);

```
        f[z] = f[x] + f[y];
        INSERT (Q, z);
    }
    return;
}
```

Exe. Time:

BuildHeap () → O(n log n)
for loop → n
Extract_min () → log n

∴ T(n) = O(n log n + n log n)

⇒ T(n) = O(n log n)

#) The node 'z' (intermediate node) will have 4 fields (not 3). All nodes should have same str. to maintain uniformity.
i) Data field (integer) — freq.
ii) Pointer pointing to left subtree
iii) Pointer pointing to right subtree
iv) Character field.

For ~~w~~ maintaining uniformity, we should go for a common minimum prog. to create the node str.
Common Min. prog. — union of all the diff. types.

For leaf nodes: i) Data field — freq.
ii) char. field — char.
iii) left subtree pointer — NULL
iv) Right subtree pointer — NULL

Prime Merit
Page No.
Date

Scanned with CamScanner

For intermediate nodes:

i) Data field — sum of freq.
ii) ~~Bai~~ Left subtree pointer — NOT NULL
iii) Right subtree pointer — NOT NULL
iv) char. field — NULL

Str. fields: i) Integer field
ii) char field
iii) 2 pointer fields

While creating node 'z' (Z = ALLOCATE_NODE()) initialize the char. field to NULL.

Hence, while creating the min heap, the array will contain ele. of structure type having 4 fields.

implementation - not in syllabus

Algo. For Creating Huffman Code:

~~Level order~~ postorder

We have to go for an ~~inorder~~ traversal of the Huffman Tree.

When moving left, assign 0.
When moving right, assign 1.

bit field add ~~values~~ to str.
For left → 0
For right → 1

~~Take of str.~~

Q1) If the ~~Q~~ length of the file is of 1000 symbols, find the length of the binary bits used to rep. it.

~~Soln.~~

~~Soln.~~ Q2) BACCDDAFFEC. How many no. of bits are reqd. to rep. it?

Prime Merit
Page No.
Date

Scanned with CamScanner

Prob. St.: Given a text (a sequence of characters), find an encoding scheme for the characters that satisfies the prefix constraint and that minimizes the no. of bits ~~need to~~ needed to encode the text.

Project Topics for Huffman Code:

1) Minimization of BW of comm. channel

2) Minimization of disk space in a disk server.

Algo. – Huffman code

~~da~~ ~~due~~ Due, to data compression, less no. of bits will be reqd to rep. → justification for using Huffman code algo.

Scanned with CamScanner

Disjoint Set Union Problem:

This is used in graph app. There are many data str. available to rep. a set, but for our ~~Data Str. used for rep. sets:~~ app., we refer to a simple DS to rep. a set. This rep. may not work for other app. ~~If~~ Even if it works, the operations would be complicated.

Assumption: $S_i \cap S_j = \phi$ → Not normally ϕ, but for our app. we assume this.

Prime Merit
Page No.
Date

Scanned with CamScanner

Operations

1. MakeSet (x) : Used for creating the set.

Assume, that the set ele. are all integers. If they are non-integers, map them to integers.

$$S = S \cup \{\{x\}\}$$

~~Operations~~

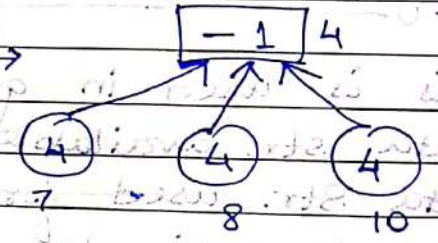
~~i) Union (S_i, S_j) : $S = S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$~~

~~Set opera~~

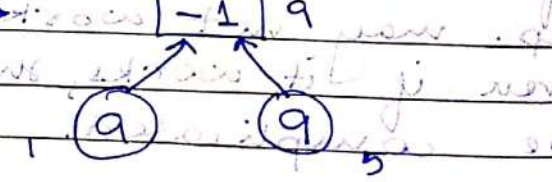
Set Representation:

Ele. of set are ~~used~~ rep. in an array.

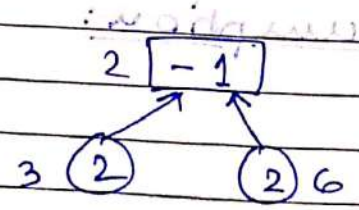
$S_1 = \{4, 7, 8, 10\}$ rep. →



$S_2 = \{1, 5, 9\}$ →



$S_3 = \{2, 3, 6\}$ →



Prime Merit
Page No.
Date

Scanned with CamScanner

Pictorially, a set is rep. in the form of an inverted tree.

Here, $S_1 \cap S_2 \cap S_3 = \phi$

<u>Inverted tree</u> — The children point to the parent.

Any one of the ele. of the set can become the root node of the set in our rep. and the others give to its children. Normally, we consider the first ele. as the root.

Since, the root node does not have a parent, hence, it is rep. by -1. The other nodes give have values of its parent.

In comp. rep:

indices

	1	2	3	4	5	6	7	8	9	10
S[] =	9	-1	2	-1	9	2	4	4	-1	4

content

Indices are 1 to 10 because there are total 10 ele. in ~~all~~ the sets.

Content of sets are basically the parent value of the indices (nodes).

The root rep. the sets. This means the ele. 5 belongs to the set 9 (not set 2 (S_2)). Ele. 4 belongs to the set 4.

Prime Merit
Page No.
Date

Scanned with CamScanner

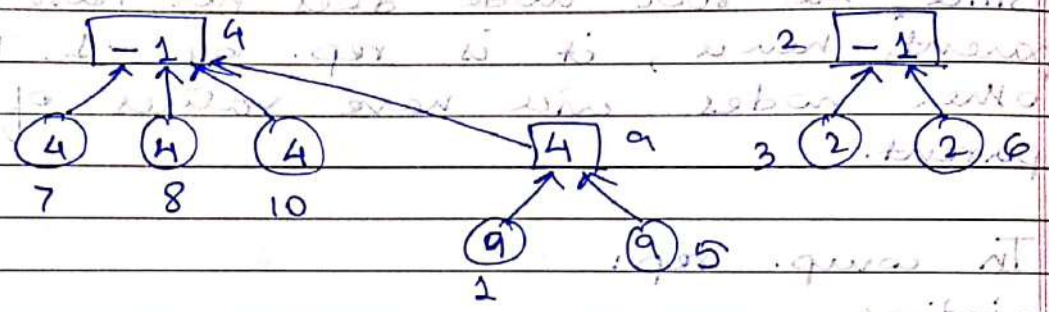
Operations:

1) Union Operation:

Assume that the intersection of the sets is ϕ.

Eg: Union (4, 9) → union b/w 2 sets, not 2 ele.

When we do union b/w 2 sets, any one of the node b/w 4 and 9 will remain as the root, and the other node will become its child.



Now, the ele. 5 belongs to set 4 (As ele always belongs to its root, not its parent).

	1	2	3	4	5	6	7	8	9	10
s[] =	9	-1	2	-1	9	2	4	4	4	4

2) Find Operation:

int findSet(5): returns 4
↳ root node of the set it belongs to
↳ returns the set it belongs to

Here, check the value of 5 in the array → parent of 5.
If it is -1 → end and return the index.
If it is +ve → check the parent of 9 in the array. Continue this as long as the parent is +ve.

<u>code:</u>

```
int FindSet (int i)
{
    while (parent(i) > 0)
        i = parent(i);
    return i;
}
```

<u>Algo. 1:</u>

1) void Union (i, j) → i, j are root nodes (Union op cannot be done b/w any 2 nodes. It can only be done b/w 2 sets)
{
    s[i] = j; or s[j] = i
}

2) int FindSet(i)
{
    while s[i] ≠ -1 do
        i = s[i];
    return (i);
}

Q) 1) If we want to perform a union op. b/w set 4 and set 2, ~~should~~ either parent of 4 will become 2 or parent of 2 will become 4. Which one should we prefer?

Prime Merit
Page No.
Date

Scanned with CamScanner

Soln.) Set 4 is made the parent of set 2.

In this case,

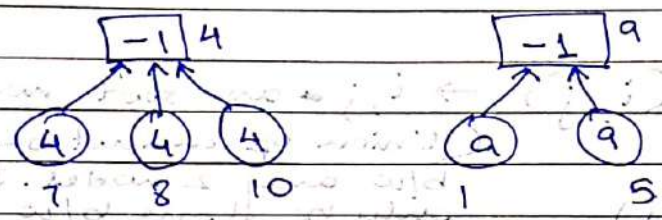
Max. no. of comparisons for any node = 3 (height of tree of set 4)

to find the ~~p~~ root node of 5.

If set 2 is made the parent of 4, max. no. of comparisons for any node = 4 (height of tree ~~of~~ after adding set 4 to set 2)

Hence, we always go for min. height of the tree.

Q2)



Which one should we make as the parent for performing union op.?

Soln.) Set 4 should be made as the parent of set 9, because ~~for~~ set 4 has more linkages.

For larger no. of nodes, comparisons = 2
For less no. of nodes, comparisons = 3

If we make ~~set~~ 9 as parent of 4,
For larger no. of nodes, comparisons = 3
For lesser no. of nodes, comparisons = 2

Hence, for more no. of nodes, the comparisons should be less.
The inverted tree having more no. of nodes is preferred to be made the root node.

Prime Merit
Page No.
Date

Scanned with CamScanner

Hence, for more no. of nodes, the comparisons should be less. The inversed tree having more no. of nodes is preferred to be made the root node. This is known as weighted union op.

~~For~~ Q3) How to find ~~the no~~ the root which has max. no. of nodes ~~in an~~ from the array?

Soln.) Method 1: Traverse the whole array and find out the ele. which has highest ~~freq~~ freq. But this method takes a lot of computation time.

Method 2: Instead of keeping '-1' as the value of the root nodes, keep a ~~neg~~ -ve value which indicates the no. of ~~nodes that root possesses~~ ele. that set has.

So for set 4, root value = -4
For set 9, root value = -3
For set 2, root value = -3

This value indicates:
i) The node is a root node because it has a -ve value.
ii) The no. of nodes the ~~root~~ set has.

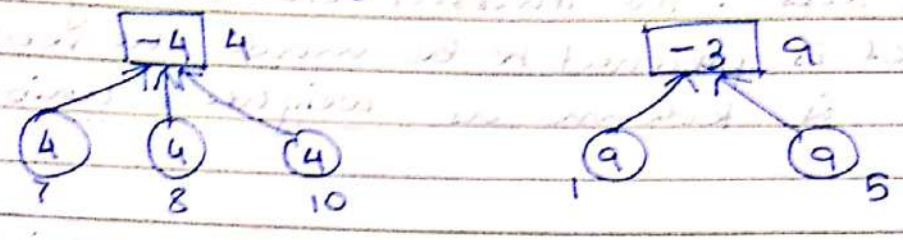
Algo.: ~~Union (i, j)~~

~~Step 1: Add the values of both the root nodes and store in a temp. var. 't'.~~

Step 2:

Chubbu

Steps for finding the union b/w set 4 and set 9:



Step 1) Add the parent value of both the root nodes and store it in a temporary var. 't'.

Step 2) Compare the parent values of both the root nodes.

If parent value of 9 > parent value of 4 → ~~ep to keep make 9 the child of 4~~

→ 4 will remain at the root node and parent of 9 will be 4, and parent value of 4 will be -7.

Graph Algorithms:

Graph, $G = (V, E)$

V ↓ set of vertices, E ↓ set of edges

Dense graph: Eg. Complete graph

$$|E| \approx |V|^2$$

Sparse graph: : $|E| \approx |V|$ (no. of edges = no. of vertices)

Scanned with CamScanner

Undirected graph : Two vertices are conn. by an edge, but there is no dir.

Edge (u, v) = edge (v, u) , $v, u \rightarrow$ vertices

No self loops exist.

Directed Graph : Edge (u, v) goes from $u \rightarrow v$ but not from $v \rightarrow u$.

Weighted graph : Edges / vertices are associated with weights.

Rep. of graph: ~~in a cov~~

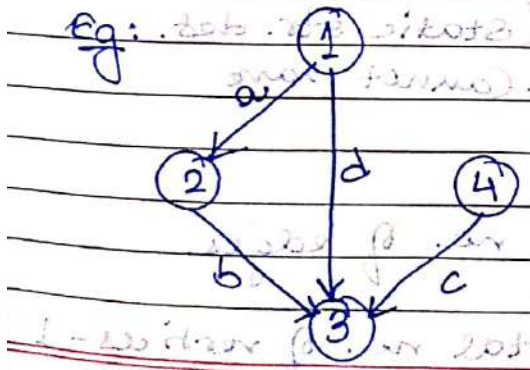
i) **Adjacency Matrix:**

$$A[i,j] = \begin{cases} 1, & \text{if there is an edge from } i \rightarrow j \\ 0, & \text{if there is no edge from } i \rightarrow j \end{cases}$$

Size of matrix = no. of vertices × no. of vertices
= $n \times n$, n = no. of vertices

Storage req. = $O(v^2)$
= $O(n^2)$

Eg:



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

Prime Merit
Page No.
Date

Scanned with CamScanner

Given an adjacency matrix, we can find out the no. of outgoing edges from a vertex or the ~~no. of~~ degree of a vertex.

Degree of a vertex = No. of edges.

~~Inde~~ Indegree : No. of edges that are coming towards the vertex.

Outdegree: No. of edges that are going out of a vertex.

For an undirected graph, there is no indegree and outdegree.

From the adj. matrix, we can find the outdegree of a vertex by ~~performing a linear search on~~ traversing that row and finding the sum of ele^s in that row (linear search)

If sum = 3 ~~→ no. of~~ outdegree = 3

ii) Adjacency List :

No. of data members in a node should be fixed, not dynamic. (Static str. def. not dynamic str. def.). Cannot have flexible no. of ptrs.

Way 1): No. of ~~data~~ ptrs. = max. no. of edges.

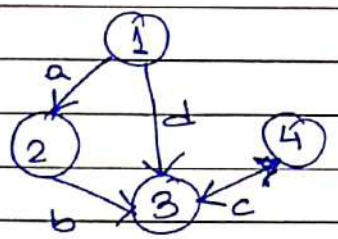
Max. no. of edges = ~~no~~ total no. of vertices - 1

Prime Merit
Page No.
Date

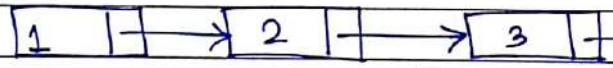
Scanned with CamScanner

Eg: If there a 100 vertices, max. no. of edges = 99. But in reality, ~~each~~ the graph can have vertices with max. 2 edges. So ~~see~~ if the LL will have 99 ptrs., there will be a huge wastage of space.

Way 2) No. of data members:
i) Data value
ii) Ptr.



The node 1 will not pt. directly to all other nodes that are being conn. So node 1 will pt. to any one of the nodes, either node 2 or node 3.



→ LL for node 1.

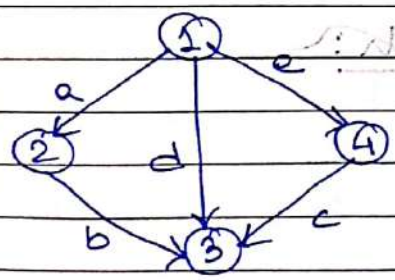
This means node 1 has a conn. with 2 nodes 2 and 3.

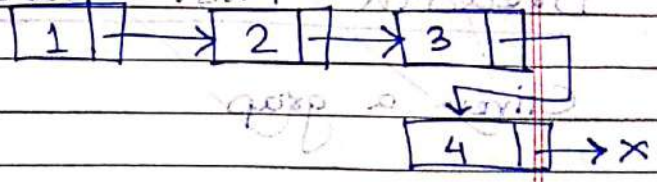
So for every vertice, there will be a LL.

∴ No. of LL = no. of vertices.

Each LL is made for each node only.

Eg:





This means, node 1 is conn. to nodes 2, 3, 4. Ptr. of 3 conn. to ptr. of 4 does not mean that node 3 is conn. to node 4.

Prime Merit
Page No.
Date

Scanned with CamScanner

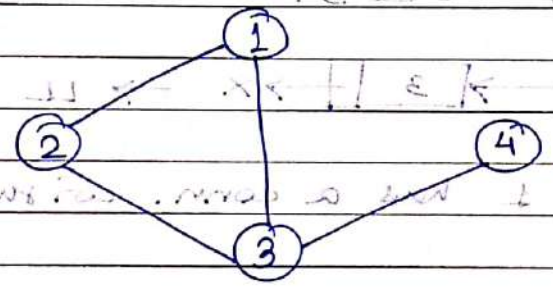
[4 |] → [3 |] → X → LL for node 4

[2 |] → [3 |] → X → LL for node 2

For undirected graphs, no. of edges will be twice. Size of LL will be twice the size for directed graphs.

Adj[1] = {2, 3}
Adj[2] = {3}
Adj[3] = { }
Adj[4] = {3}

Eg.:



Adj[1] = {2, 3}
Adj[2] = {1, 3}
Adj[3] = {1, 2, 4}
Adj[4] = {3}

Breadth First Search:

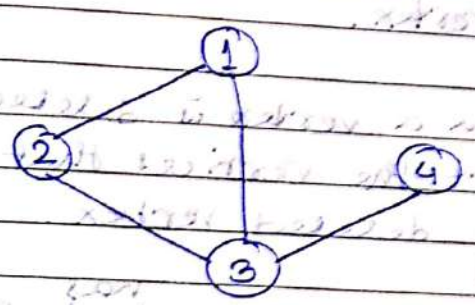
Given a graph

Graph Search:

Given a graph (directed/indirected), we need to search all vertices of the graph. Start searching/visiting from any one of the vertex (assuming undir. graph), considered as source vertex. From there, use a particular method/order to visit/access every other vertex of the graph.

Eg: In case of BT → preorder, inorder, postorder traversals

1) Breadth First Search:



Step 1) Source vertex = 1 (assume)

Step 2) visit all the adjacent vertices of 1. Then the vertex 1 will be explored/dead.

Step 3) Whichever adj. vertices were visited first, go to that vertex.

Color rep.: (In comp. → rep. by digits 1, 2, 3)

1) white vertices — rep. vertices that are yet to be discovered. Till now the vertice has not been touched.

2) grey vertices: — rep. vertices that have been visited but yet to be explored. A vertex will be explored if all its children are visited.

3) black vertice — rep. vertices that have been explored fully. Explored fully means all its adjacency vertices have already been visited.

~~BFS is implemented using a queue.~~

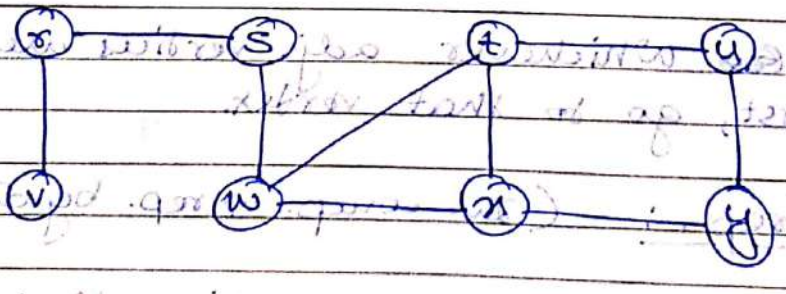
Implementation: (Queue)

Source → any vertex.

Everytime ~~a~~ when a vertex is deleted from the queue, ~~ad~~ insert the vertices that are adj. to the currently deleted vertex.

Restriction: If ~~a~~ the vertex ~~had~~ has already been inserted in the queue (explored), do not insert it again. ∴ A vertex will only be inserted in the queue once.

Eg:



Q: | s | w | r | t | x | v | u | y

↓
queue

BFS: s, w, r, t, x, v, u, y

'for' loop will run for 'E' times for directed graph.
For undir. graph → '2E' times (only for adj. list).

Algorithm:

```
BFS (G, s)
{
    initialize vertices;
    Q = {s};        // Q → simple queue (not pr. queue)
                       s → source vertex
    while (Q not empty)
    {
        u = removeTop(Q);   → V times
        for each v ∈ u→adj   → no. of edges (E)
        {
            if (v→color == WHITE)
            {
                v→color = GREY;

                v→d = u→d + 1
                v→p = u;
                Enqueue (Q, v);
            }
        }
        u→color = BLACK;
    }
}
```

d → distance/depth
p → predecessor

Total running time
$= O(V+E)$ → dir. gr.
$= O(V+2E)$ → undir. gr.
$= O(V+E)$

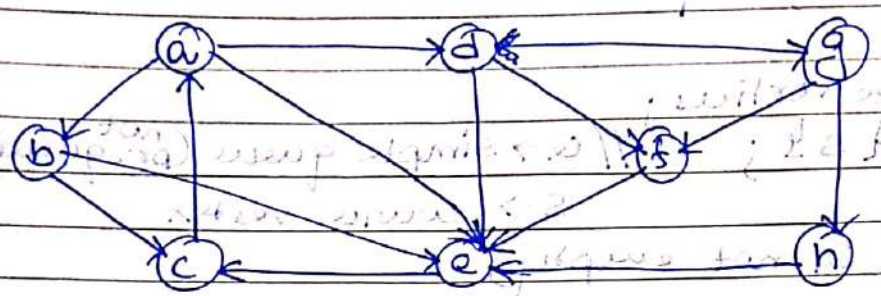
2) Depth First Search:

This is another strategy where we go deeper and deeper in the graph. A vertex will only be explored if all its children have been explored.

Prime Merit
Page No.
Date

Scanned with CamScanner

Implementation : (Stack)



source vertex → a

Insert the source vertex in the stack, find the adj. vertices and insert one of them to the stack after pop deleting the source vertex.

S : ~~a~~ → ~~b~~ → ~~c~~ → ~~e~~ → ~~d~~

↓

Stack : e, d | e, d | d

~~h~~ ← ~~g~~ ← ~~f~~

g

DFS : a, b, c, e, d, f, g, h

Color rep : white, grey, black → same as BFS

Algorithm:

```
DFS_Visit (u)
{
    u → color = GREY;
    time = time + 1;
    u → d = time;
```

Prime Merit
Page No.
Date

Scanned with CamScanner

```
for each v ∈ u→Adj[]
{
    if (v→color == WHITE)
        DFS_Visit(v);
}
u→color = BLACK;
time = time + 1;
u→f = time;
}

DFS (G)
{
for each vertex u ∈ G→V
{
    u→color = WHITE;
}
time = 0;
for each vertex u ∈ G→V
{
    if (u→color == WHITE)
        DFS_Visit(u);
}
}
```

Time Complexity:

~~DFS(G) : O(~~

DFS_Visit(u) : $O(n)$ → loop over Adj[] can run |V| no. of times

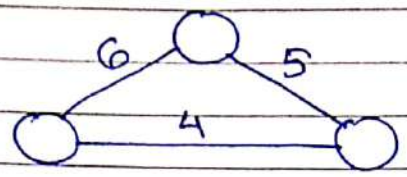
DFS(G) : $O(n^2)$ → call DFS_Visit(u) on each vertex.

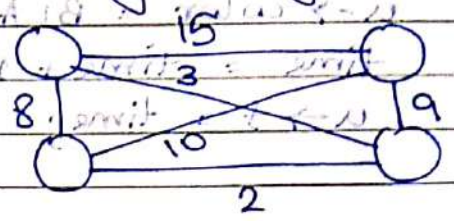
Scanned with CamScanner

Problems on Graph:

1) Minimum Cost Spanning Tree of a Graph:

Given a conn., undir., weighted graph:





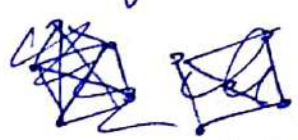
Prob. st.: A motherboard of a comp. is a printed circuitry board (PCB). It has lots of small wires which are conn. b/w any 2 pts.
~~The PCB can be rep. as a graph.~~

The PCB can be rep. as a graph. The vertices are ~~di~~ the diff. pins of the IC and the edges are the ~~diff.~~ conductors placed b/w 2 pins of the IC. (IC can be NAND / NOR gate etc) ~~eg.~~

If ~~each~~ any one of the vertices (pins) of the graph (IC) is given 5V, then all of the other vertices will also contain 5V; because ~~the~~ all of the vertices are interconnected. (in the above diag.)

In the first graph, to connect 3 vertices, 3 conductors are used. But 3 vertices can also be connected using 2 conductors.
↳ less reliable

~~The mothe~~ Similarly, as the motherboard has thousands of wires. (very clumsy). So it is possible that maybe the designer of the PCB



Scanned with CamScanner

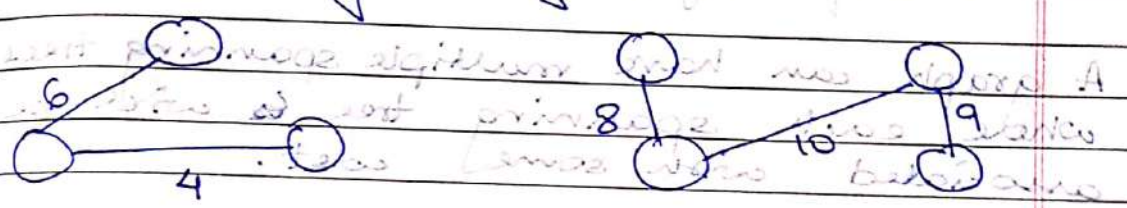
was put unnecessarily more conductors to connect the pins.

Adv:

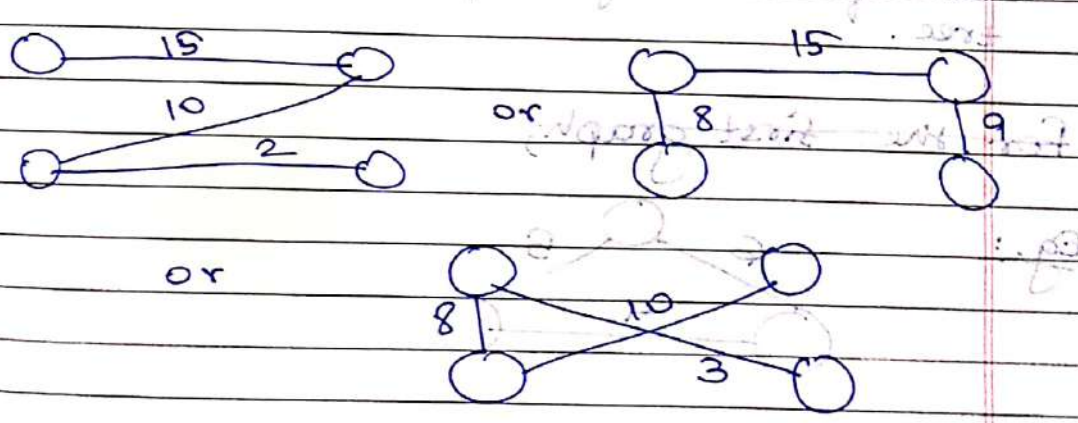
i) So we can connect 3 pins using 2 ~~conducto~~ conductors → conductors are saved.

ii) The no. of wires that are interconn. in a PCB are minimized. So possibly more no. of IC can be put in a mother board.

In the second graph, 4 vertices can be conn. using only 3 conductors.



There can be multiple ways to connect the vertices.



So when our task is performed in 3 wires, we choose 6 wires just to increase the reliability.

Avoiding the reliability factor, to connect 100 pins, we would require 99 wires. & → no. of conductors minimized.

Prime Mart
Page No.
Date

Scanned with CamScanner

So to design a ~~mothe~~ large PCB like a motherboard, the no. of wires can be minimized using a spanning tree.

Spanning Tree:

Given a graph with 'v' no. of nodes, they can be connected using 'v − 1' edges. This is called a spanning tree ~~(no more a graph)~~ (no cycle present)

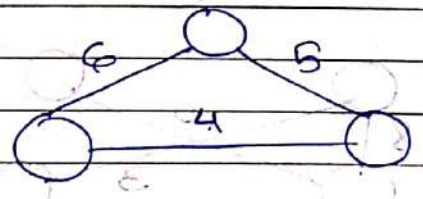
~~In a complete graph containing 'v' vertices, total no. of edges present~~

A graph can have multiple spanning trees, where each spanning tree will be associated with some cost.

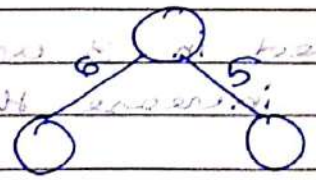
Cost of a spanning tree = sum of the cost of all edges present in the spanning tree.

~~For the first graph,~~

Eg.:

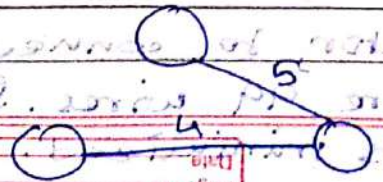


i)



Cost = 6 + 5 = 11

ii)



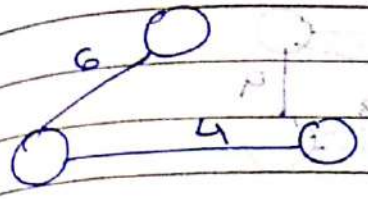
Cost = 5 + 4 = 9

↓

optimal

Prime Merit
Page No.
Date

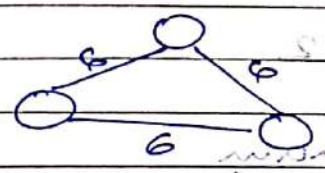
Scanned with CamScanner

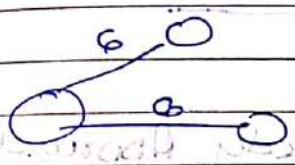
iii) 

Cost = 6 + 4
= 10

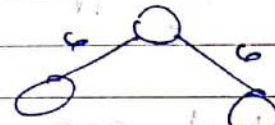
So for multiple spanning trees, we go for the one which has minimum cost → optimal

So a A graph can have multiple min. cost spanning trees. (cost will be same, only the edges will be different)

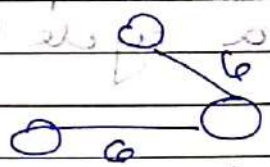
Eg: 

i) 

→ cost = 12

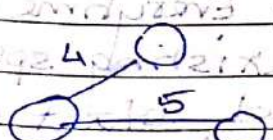
ii) 

→ cost = 12

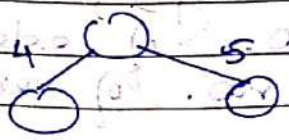
iii) 

→ cost = 12

Eg:

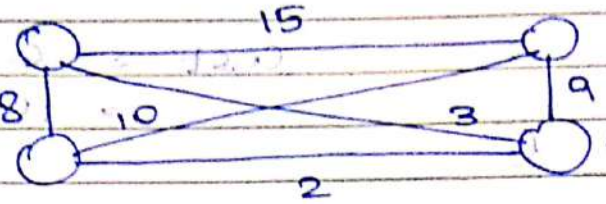
i) 

cost = 9

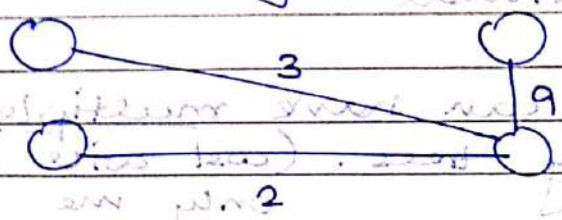
ii) 

cost = 9

Eg.:



Min. cost spanning tree:



Cost = 3 + 9 + 2
= 14

Algorithm for Minimum Spanning Tree:

1) Kruskal's MST: (Greedy Approach)

Feasibility Test: Constructed spanning tree should never have a cycle (trees have no cycle)

In a spanning tree, there should be all the vertices present but not all the edges.

Selection Criteria: Everytime add a new edge to the existing spanning tree (spanning tree should start by having no edges initially, only vertices) till the no. of edges become (n − 1), if no. of vertices is 'n'.

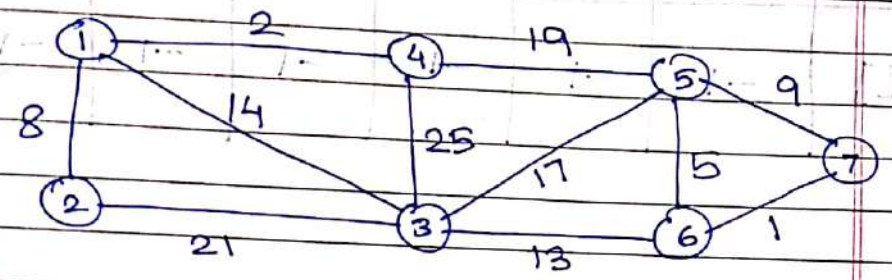
Prime Merit
Page No.
Date

Scanned with CamScanner

So everytime, add ~~the~~ a new edge ~~with~~ to the spanning tree having ~~min. cost~~ the next min. cost.

While ~~doing this~~ adding the edges in this manner, if it gives a cycle, then that edge is discarded. But we cannot discard a prev. edge instead of the curr. edge because the prev. edge had a lower value.

Eg.:



Initially, no edge present, only vertices present.

i) Add edge 1, conn. 6 & 7

ii) Add edge 2, conn. 1 and 4
(no cycle prod. → added)

iii) Add edge 5, conn. 5 & 6.
(no cycle)

iv) Add edge 8, conn. 1 & 2
(no cycle)

v.) ~~Add~~ Choose edge 9, conn. 5 & 7.
(cycle prod. conn. 5, 6, 7 → discarded)

vi). Add edge 13, conn. 3 and 6
(no cycle)

Prime Merit
Page No.
Date

Scanned with CamScanner

vii) Add edge 14, conn. 1 & 3 (no cycle)

Here, we stop adding any more edges because the graph has 7 vertices and 6 edges had already been included. By adding any more edges → cycle.

Node	1	2	3	4	5	6	7
S[]	-1	-1	-1	-1	-1	-1	-1

For the comp. to understand if there is a cycle in the spanning tree, ~~for~~ ~~for~~ for every vertex ~~a~~ prepare a set. set containing

S[] → parent vertices

i) Parent of every node is -1 → initialization
→ every vertex is independent
∴ No. of sets = no. of vertices

ii) Arrange the ~~vertices~~ edges in increasing order of their cost by using any sorting algo. or ~~heap~~ min heap str. → initialization

~~Every time~~ An edge is rep. by (u, v)
u, v → 2 vertices.

Prime Merit
Page No.
Date

Scanned with CamScanner

Iteration:

Step 1) Everytime delete an edge having the next min. cost.

Step 2) After deleting edge (u,v), find out to which set u and v belongs to and store it in ~~u~~ j ~~and~~ and k, resp.

Step 3) If j and k belong to the same set (j = k), then it gives a cycle and hence, the edge is discarded.

Step 4) If j and k do not belong to the same set (j ≠ k), then include the edge. While including the edge, do a union operation b/w j and k.
(so any one node, either j or k, will become the root).

Step 5) Include u and v in the soln. vector s1 and s2, resp.

Step 6) Increment i by 1 → one edge is added.

Repeat the above steps till (n-1) no. of edges are added.

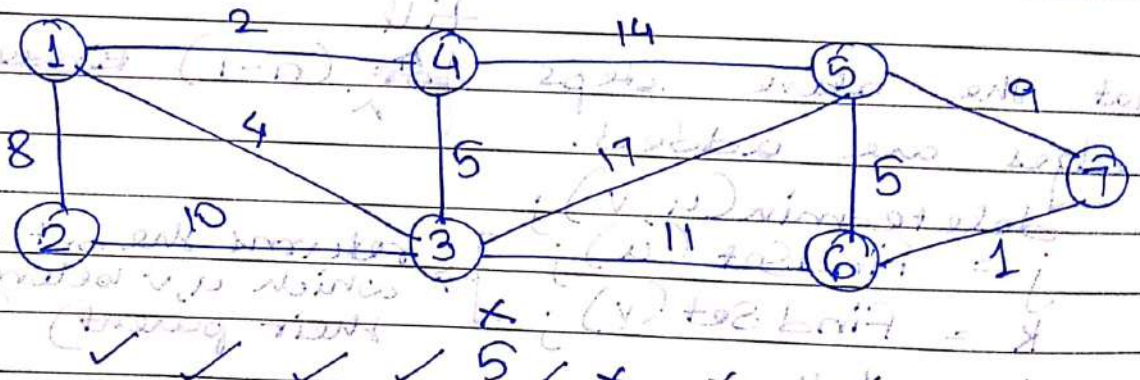
```
delete-min(u, v);
j = FindSet(u);
k = FindSet(v);
if j ≠ k
    Union(j, k);
    s1[i] = u;
    s2[i] = v;
    i++;
```

→ returns the set to which u,v belong (not their parent)

s1[], s2[] → arrays that contain the edges which are being included.

Prime Merit
Page No.
Date

Scanned with CamScanner



✓ ✓ ✓ ✓ ✗ ✓ ✗ ✗ ✓

Edges: 1, 2, 4, 5, 8, 9, 10, 11, 14, 17

Scanned with CamScanner

	1	2	4	5	5	8	9	10	11	14	17
u	6	1	1	5	4	1	5	2	3		
j	6	1	4	5	3	3	5	2	2		

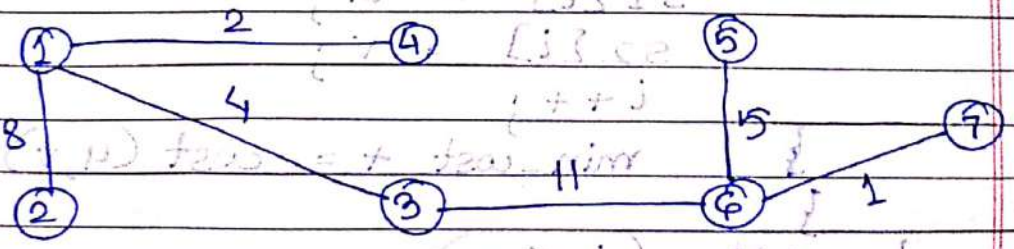
	1	2	4	5	5	8	9	10	11	14	17
v	7	4	3	6	3	2	7	3	6		
k	7	4	3	6	3	2	5	2	5		

arent:	6	4	3	5	×	2	×	×	2

Node	1	2	4	5	5	8	9	10

Node	1	2	3	4	5	6	7
s[]	~~-1~~ 4	-1	~~-1~~ 2	~~-1~~ 3	~~-1~~ 2	~~-1~~ 5	~~-1~~ 6
s1[]	6	1	1	5	1	3	
s2[]	7	4	3	6	2	6	

Iteration is stopped because 6 edges have been included.



Algorithm:

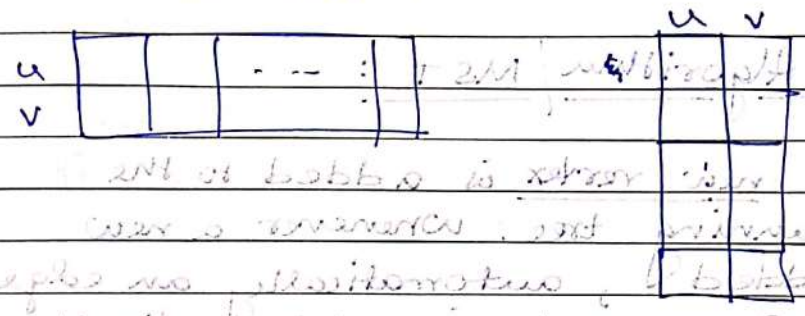
```
                                  cost matrix of graph
                                  ↓
Kruskal ( G, n )          // G → graph (adj. list / adj. matrix)
                                n → no. of vertices
{

    for i = 1 to n do     // initialization
        s[i] = -1;

    Q = E(G);             // prepare a min heap for all
    ↓                        edges of the graph taking
priority                     cost of the edges into
queue                        consideration

    min_cost = 0;

    i = 1;                // include one edge definitely

    do
    {
        (u,v) = ExtractMin (Q);
        j = FindSet (u);
        k = FindSet (v);

        if (j ≠ k)
        {
            Union(j,k);
            s1[i] = u;
            s2[i] = v;
            i++;
            min_cost += cost (u,v);
        }
    } while (i < n);
```

We have to finally display the edges that are included in the soln. vector. So either we can have a name for ~~each~~ an edge or we can use the name of 2 vertices to refer to an edge.

Since here, we are using 2 vertices to refer to an edge, hence, we need 2 soln. vectors s1[] and s2[].

OR

We can also take a 2D array of size ~~2×number~~ (2 × vertices) or (vertices × 2), where in first row, one ~~vertex~~ vertex is stored and in another row, another vertex is stored.



Exe. Time:

FindSet() → v (no. of vertices) (if no. of vertices is v, the height of bin. tree can never be > v.)

ExtractMin() → log E , E → edges

do-while loop ~~max exec. var min~~ (v−1) times
→ min exe : (no. of vertices − 1) times (best case)
max-exe. : (no. of edges) times (worst case)
E times

$$\boxed{T(E) = O(E \log E)}$$

Prime Merit
Page No.
Date

Scanned with CamScanner

for loop initialization

MakeSet () → O (V)
FindSet () → O (V)
Union () → O (V)

Heap : BuildHeap() → O (E Log E)
ExtractMin() → O (log E)] → O(E log

Sort : Sorting → O (E log E)
ExtractMin () → O(1) → O (E log E)

∴ Exe. Time = O (E. log. E)

2) Prim's Algorithm / MST :

Everytime a new vertex is added to the existing spanning tree. Whenever a new vertex is added, automatically an edge is added. So a vertex is added directly while an edge is added indirectly.

Eg: (f) —— 9 —— (g)

Assuming 'g' is already there in the spanning tree. When 'f' is added, automatically the edge (f, g) is added.

distance / key → key to source vertex = 0
initialization → key to rem. vertices = ∞

There will always be a source vertex in Prim's algo. Any one of the vertices can become the source vertex in constructing a spanning tree.

Prime Merit
Page No.
Date

Scanned with CamScanner

'key' is the smallest cost edge which conn. to a vertex could be added to the existing spanning tree

Everytime the vertex having the ~~low~~ least key value is added to the existing spanning tree.

After adding, find the vertices that are adj. to the newly inserted vertex.

If at all in future, we want to include ~~&~~ any of the adj. vertices ~~to~~ in the spanning tree, we need to connect the newly inserted vertex to the adj. ~~vertex~~ vertex. So earlier distance of the adj. vertex was ∞, but now it will be updated ~~to~~ as the ori. weight the edge had. This is because the adj. vertices have a conn. to the existing spanning tree vertex.

Everytime a ~~ne~~ new vertex is included, first find the adj. vertices to the existing vertex. Among these, first update their dist. and ~~e~~ if the new dist. / key is less, update the key value.

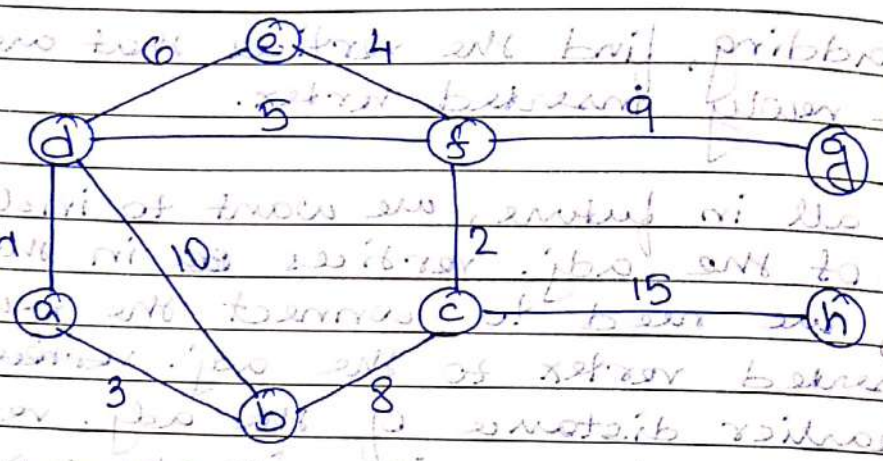
```
for each v ∈ Adj[u]
{
    if (v ∈ Q and w(u,v) < key[v])
    {
        pred[v] = u;
        key[v] = w(u,v);
    }
}
```

Prime Mark
Page No.
Date

Scanned with CamScanner

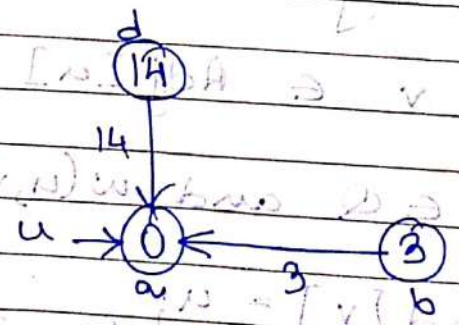
v → vertex not yet added
u → vertex already existing in spanning tree
Q → priority queue (not there in spanning tree)
w(u,v) → weight of edge (u,v)
pred[v] → predecessor/parent of vertex v

Eg.:



Index	a (source vertex)	b	c	d	e	f	g	h
key[]	0	~~∞~~ 3	~~∞~~ 8	~~∞~~ ~~14~~ ~~10~~ 5	~~∞~~ 4	~~∞~~ 2	~~∞~~ 9	~~∞~~ 15
pred[]		a	b	e	f	c	f	c

Iteration 1:

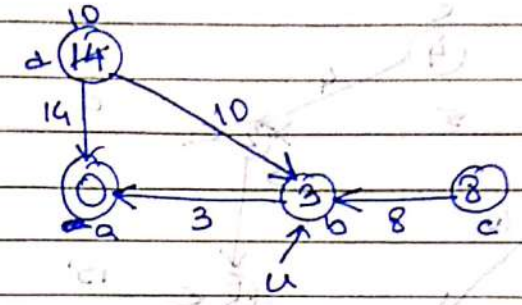


key value updated and pred. marked as 'a'. But spanning tree consists only 'a'. 'b' and 'd' not yet included.

Prime Merit
Page No.
Date

Scanned with CamScanner

It. 2)

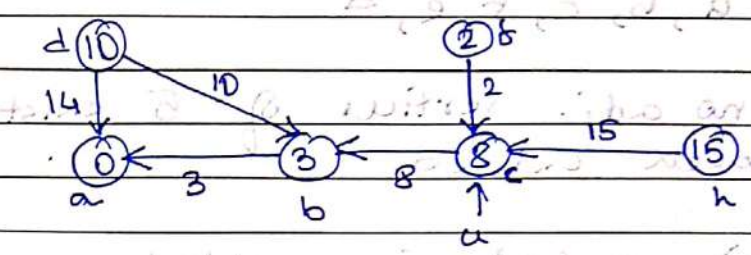


ST → a, b

pred. of d → b

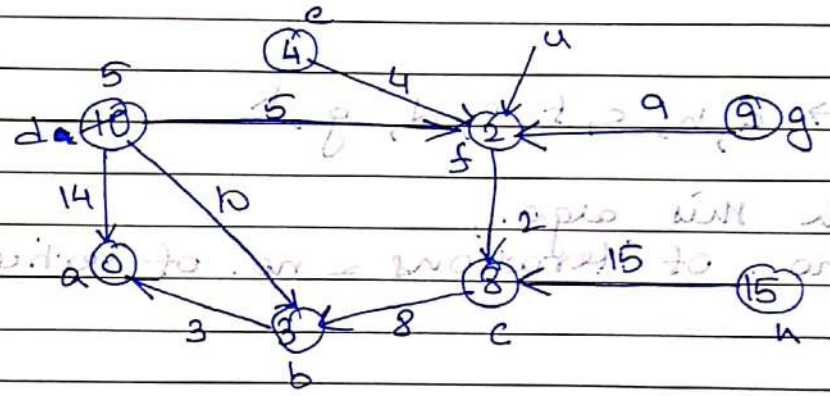
pred. of c → b

It. 3)



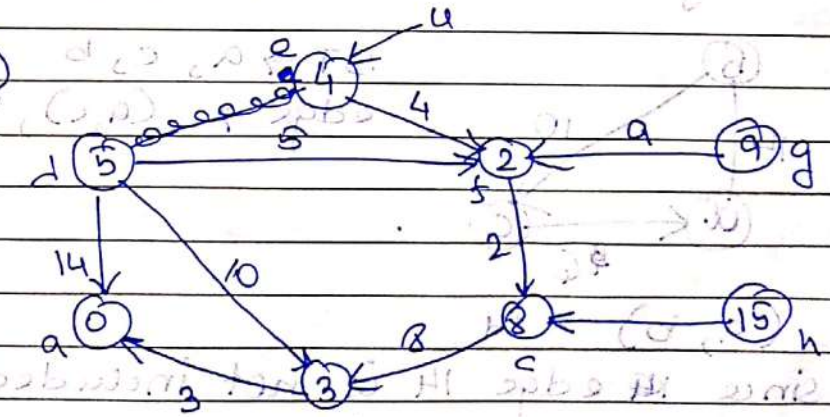
ST → a, b, c

It. 4)



ST → a, b, c, f

It. 5)

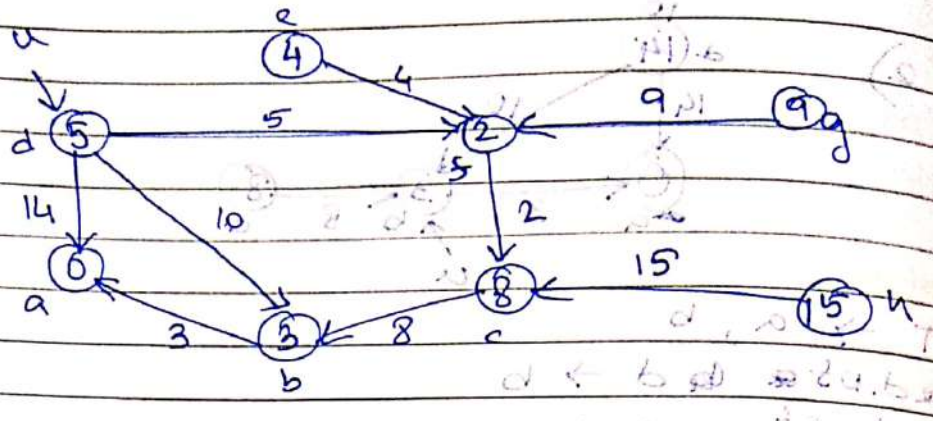


ST → a, b, c, f, e

edge (4, 5) is not updated to 6 because 6 > 5. (5 → ori. wt.)

Scanned with CamScanner

It. 6)



ST → a, b, c, f, e, d

Here, no adj. vertices of 5 exist that has not been added.

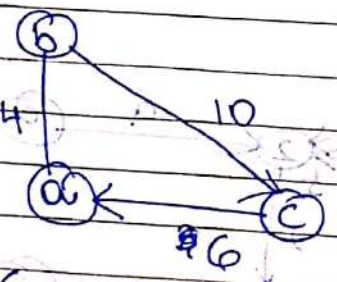
It. 7) 'g' is added

It. 8) 'h' is added.

ST → a, b, c, f, e, d, g, h

In this algo.,
no. of iterations = no. of vertices.

#) Spanning tree is not a shortest path prob.



ST → a, c, b
edge → (a, c), (c, b)

edge (a, b) = 14
But since edge 14 is not included,
total cost from a to b = 6 + 10
= 16

So the ~~to~~ cost from a to b increases. But this is not taken into consideration because it is not a shortest path prob.
Here, my aim is ~~to~~ to get the same voltage on ~~each~~ every node when a particular voltage is applied on any one of the node.

Here, all vertices are connected, but while moving from one ~~vertex~~ vertex to another ~~it~~ it is not necessary that we always choose the shortest path.

Using a spanning tree, we minimize the total cost and all vertices should be connected and this is possible with $(n-1)$ no. of edges.

<u>Algo.</u> :

```
                    graph   cost   source vertex
MST-Prim (G, w, s)
{
        [ for each u ∈ G do
              key[u] = ∞;
initialization — key[s] = 0;
        [ pred[s] = NULL;
        [ Q = V[G]; // prep. min heap for all vertices of the graph taking the key value into consideration
  while (Q ≠ empty)
  {
      u = ExtractMin(Q);
      for each v ∈ Adj[u]
      {
          if (v ∈ Q and w(u,v) < key[v])
          {
              pred[v] = u;
              key[v] = w(u,v);
              DecreaseKey(v, w(u,v));
          }
      }
  }
}
```

Prime Merit
Page No.
Date

Scanned with CamScanner

~~Algorithm~~ There is a heap str. for all the key values. If the key value is updated the updated key value will change its pos. in the heap.

DecreaseKey() → hidden cost & in the code (extra func. ~~or~~ that needs to be executed)

~~So the up~~

So the updated key will move up in the heap str. → same as insertion in a particular level of a heap.

DecreaseKey(v, w(u,v)) → Insert the value w(u,v) in the level which contained v in the heap.

↓

exe. time = log V (V = no. of vertices)

Exe. Time:

while loop → ~~O(exe)~~ O(V)

ExtractMin() → O(log V)

for loop (adj. vertices) → O(2E) → undirected graph

↓

total (sum of exe. of all while loop)

DecreaseKey() → log V

∴ Exe. Time = O(2E log V + V log V)

$$\boxed{\text{Exe. Time} = O(E \log V)}$$

(V < E)

Prime Merit
Page No.
Date

Scanned with CamScanner

Shortest Path Algos: (of graphs)

Types of Shortest Path:

1) Single source, single destination
2) Single source, multiple destinations
3) Multiple sources, multiple destinations (all pairs shortest path)

shortest path - cost of the edges present in that path should be min.

might be more than 1 optimal soln. (diff path, but same cost - Greedy)

Type 1: graph having 100 nodes

source : 1 , source = 48 , etc.
dest. : 50 dest. = 70

Type 2: graph having 100 nodes

source : 1 & source : 1 & source : 1 & ... & source : 1
dest : 2 dest : 3 dest. : 4 dest. : 100

Type 3: graph having 100 nodes

source : 1 & source : 1 & ... & source : 1 & source : 2 &
dest. : 2 dest. : 3 dest : 100 dest. : 1

source : 2 & ... & source : 2 & ... & source : 100 ....
source : 3 source : 100 dest : 1

If the soln. of any one of the types prob. is known, soln. of other probs. are automatically known.

Page No.
Date
Prime Merit

Scanned with CamScanner

Type 2 Prob. Using Dijkstra's Algo. : (like Prim's Algo.)

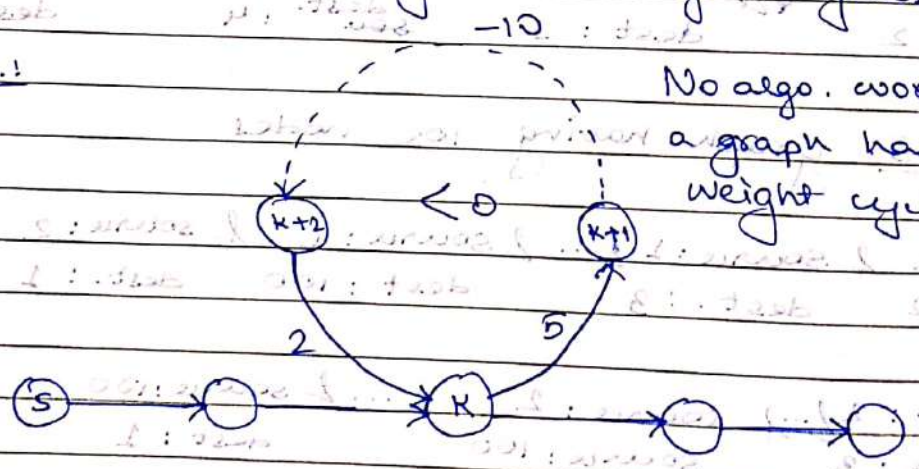
Given a dir. graph, find the min. wt. path from a given source vertex S to another vertex V. Weight of path is the sum of the edges. (like Type 1)

If source : 1 and an intermediate vertex = 7,
dest : 10

then the path from 1 to 7 used in the path from 1 to 10 must also be a shortest path. So all subpaths in the shortest path prob. must always be the shortest path./optimal path.

For finding the soln., the graph should not have negative weight cycles.

Eg.:



No algo. works if a graph has -ve weight cycles.

dist. covered from S to K = 100 (assume)
cost of $K \to K+1 \to K+2 \to K$ $= 5 - 10 + 2$
$= -3$
Again if we go $K \to K$ $= -3 + 5 - 10 + 2$
$= -6$ to $-\infty$

Like this, the -ve value keeps on decreasing, and will never be the shortest path b/w 2 vertices.

Prime Merit
Page No.
Date

Scanned with CamScanner

~~The~~ If instead of -10, it was -5, then the cycle cost $= 5 - 5 + 2$
$= 2$

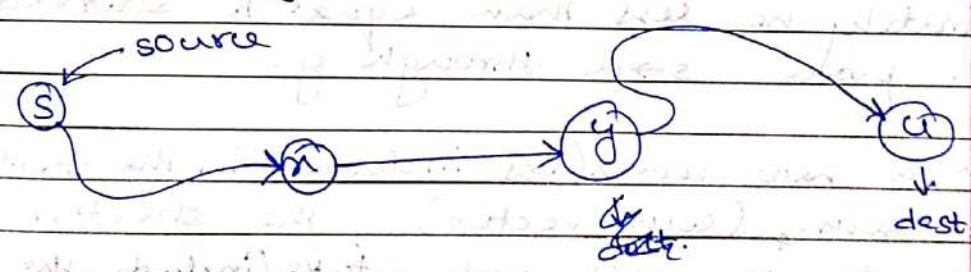
So there might be a $-ve$ edge, but not a $-ve$ cycle.

If the cycle cost is $-ve$, we go on rotating in the cycle to reduce the cost, which results in an $-\infty$ cost.

If the cycle cost is $+ve$, no need to rotate in the cycle again & again, because that is the min.

So the ~~shor~~ path from source to dest. should not contain $-ve$ weight cycles.

Dijkstra's algo. will not accept $-ve$ edges present from source to dest. There might be $-ve$ edges somewhere else.

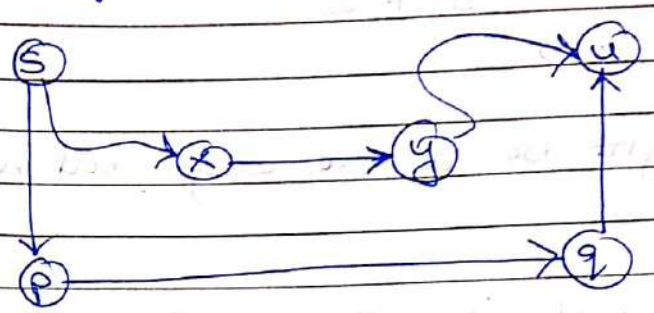


The shortest dist. from $s \to y$ have already been found with some intermediate vertices x $(x_1, x_2, x_3, \ldots)$

The shortest dist. from $s \to u$ will only cover the inter. vertices whose shortest dist. had already been determined. It means, if we are going $s \to u$ through x & y, we must have determined the

shortest dist. from $s \to x$ and $s \to y$.

Eg.:



If the shortest path from $s \to u$ was found to be through inter. vertices p & q and the shortest dist. from $s \to p$ and $s \to q$ are yet to be determined, then $s \to p$ & $s \to q$ has to be included in the soln. vector first.

Everytime a new vertex is added to the shortest path soln., it should be the vertex whose dist. is min.

Suppose
If $s \to x$ shortest path is determined and $s \to y$ and $s \to u$ are yet to be determined, If the shortest dist. exists from $s \to u$ through y, then the shortest dist. from $s \to y$ will definitely be less than equal to shortest dist. from $s \to u$ through y.

When a new vertex(u) is included in the shortest path soln. (soln. vector), the shortest path to u will only take/include the vertices available in the shortest path soln.

Relaxation:

Given source = s and soln. vector already contains $s \to u$.
$\therefore$ shortest dist. $s \to s = 0$

After adding u in the soln. vector it is

found that adj(u) is v. So there is an edge from $u \to v$, hence $s \to v$ is possible.

Let there ~~was~~ be path from $s \to v = 100$ (not ~~shortest~~ yet established to be shortest) and shortest dist. (established/proved) from $s \to u = 50$.

~~∴ shortest d $s \to v = 100$~~ from the prev. knowledge, we had, among them $s \to v$ is shortest.

Let Cost $(u, v) = 20$.

$\therefore$ $s \to v$ (through u) $= 50 + 20$

$= 70$ (not 100)

↳ probable shortest

So pred. of v = u

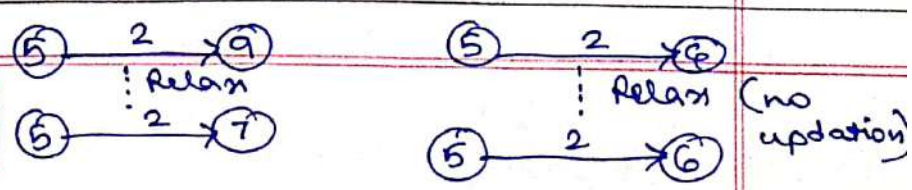
Let cost $(u, v) = 80$

$\therefore$ $s \to v = 50 + 80$

$= 130$ → this will not be updated because $130 > 100$

Relax (u, v, w)
{
if $(d[v] > (d[u] + w(u, v)))$ then
$d[v] = d[u] + w(u, v)$
}

This is known as relaxation. (whether we will relax the prev. with a new edge or not).

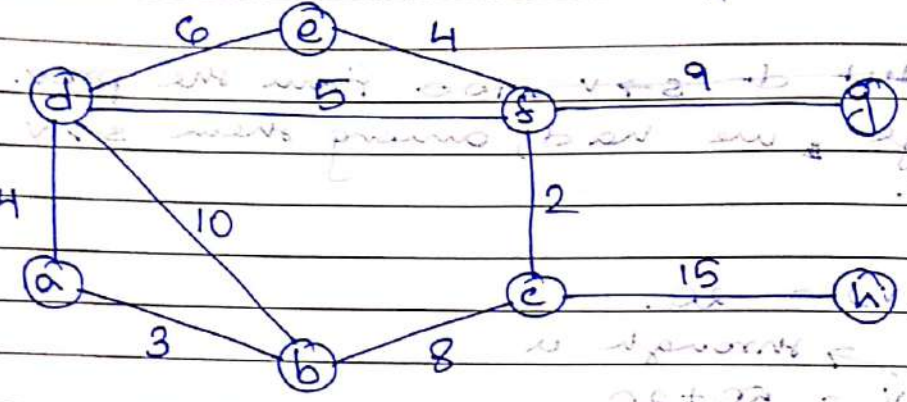


Prime Merit
Page No.
Date

Scanned with CamScanner

Alg Dijkstra's Algo. :

Everytime we delete a vertex whose dist. is least and add to the soln. vector. ~~The~~ check the dist. of the adj. vertices of that deleted vertex using the relaxation formula.



Index	a	b	c	d	e	f	g	h
d[]	0	~~∞~~ 3	~~∞~~ 11	~~∞~~ ~~14~~ 13	~~∞~~ 17	~~∞~~ 13	~~∞~~ 22	~~∞~~ 26
p[]		a	b	~~a~~ ~~b~~ 5	f	c	f	c

```
for each v ∈ u->Adj[]
{
    if (d[v] > d[u] + w(u,v))
    {
        d[v] = d[u] + w(u,v);
        p[v] = u;
    }
}
```

It. 1) SP → a

Pred : a → b , a → d
(3) (14)

It. 2) SP → a, b

Pred : b → d , b → c
(3+10 = 13) (3+8 = 11)

It. 3) SP → a, b, c

Pred. : c → h , c → f
~~(3+8~~ (11+15 = 26) (11+2 = 13)

It. 4) SP → a, b, c, f

Pred. : f → g , f → e ,
(13+9 = 22) (13+4 = 17)

f → d → not updated
(13+5 = 18) because d was prev. 13.

It. 5) SP → a, b, c, f, d

Pred : d → e
(13+6 = 19) → not updated because e was prev. 17

(Here, d is included in SP → always choose the vertex with min. dist. from the rem. vertices)

It. 6) SP → a, b, c, f, d, g

~~(Here, g is inclu~~

It. 7) SP → a, b, c, f, d, g, h

Prime Mart
Page No.
Date

Scanned with CamScanner

Q) Find shortest path $a \to h$.

Soln.) Always start searching from dest to source. (because in this way, we can find the predecessors)

$h \to c \to b \to a$

Q) $a \to g$: $g \to f \to c \to b \to a$

Algo.:

cost matrix is passed

Dijkstra (G, s) // or Dijkstra (G, n, s)

{ no. of vertices

for each $v \in V$ // initialization of dist. array

$d[v] = \infty$

$d[s] = 0$;

$Q = V[G]$; // prepare a min heap for all vertices taking min. dist. into consideration (not min cost)

front dist. array

while ($Q \neq$ empty)

{

u = ExtractMin (Q);

for each $v \in Adj[u]$ && $v \in Q$

{

if ($d[v] > d[u] + w[u,v]$)

{

relaxation step $\left[\begin{array}{l} d[v] = d[u] + w(u,v); \\ p[v] = u; \end{array} \right]$

$Q \to$ Decrease key

}

}

}

}

Prime Merit

Page No.

Date

Scanned with CamScanner

Q → ~~dec~~ DecreaseKey (') : Reverse heapify
or insert at a particular level.
↳ check the parent nodes

Heapify → compare the child nodes and insert

Reverse heapify → compare the parent nodes and insert

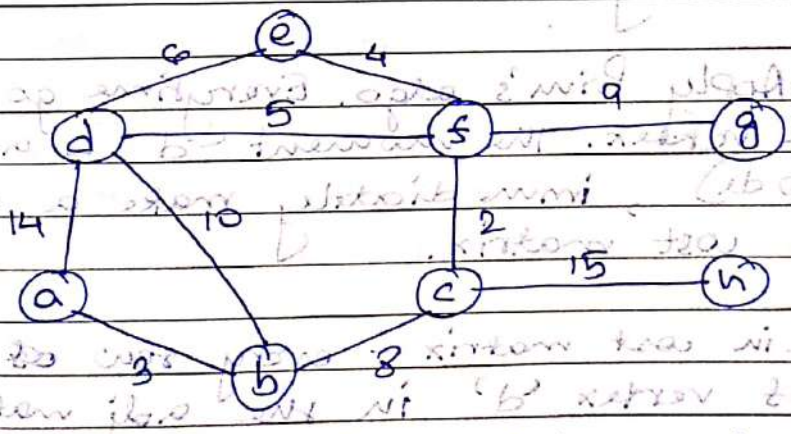
Time Complexity:

Same as Prim's algo.

$\boxed{\text{Exe. Time} = O(E \log V)}$
→ using binary heap

$\boxed{\text{Exe - Time} = O(V \log V + E)}$ → not in syllabus
→ using fibonacci heap

Q1)



Find out the min. cost spanning tree where the vertex 'd' will become the leaf node.

d → Nodalite point

Soln.) A tree has: root node, inter. node and leaf node.

leaf node → deg. 1
inter. node → ~~deg~~ > deg. 1

The moment 'd' is included in the ST, all its adj. nodes will not be considered. So no other ~~adj.~~ nodes ~~of 'd'~~ will be included which has any conn. with 'd'.

Eg.: a → b → d
So from here, the adj. nodes of 'd' i.e., 'e' and 'f' will not be considered as adj vertices of 'd'. So there is no possibility of d → e or d → f in future.

So dist. of d → e = ∞ and d → f = ∞
This has to be ~~up~~ changed in the cost matrix. So d → e and d → f is not possible anymore.

Algo.: Apply Prim's algo. Everytime go on adding a vertex. The moment 'd' is added (leaf node), immediately make a change in the cost matrix.

Change in cost matrix – every row ~~of~~ and col. of vertex 'd' in the adj matrix, will be made ∞. So no vertices henceforth will remain as adj. to 'd'.

Hence, 'd' will never be coming as an inter. vertex. → deg = 1. → leaf node

Prime Merit
Page No.
Date

Scanned with CamScanner

But in this way, the MST obtained will never be same as the ori. MST. It will always be greater than or equal to the ori. MST. But it is the MST with a restriction that 'd' will remain as the leaf node.

After making a change in the cost matrix (row, col. of d = ∞), ~~chose~~ choose the next vertex of the ST from the ~~min~~ min heap as the vertex with min. ~~cost~~ dist.

Eg.: If 'c' had to be made the leaf node in ST, then ~~the~~ vertex 'h' would become unreachable, because 'c' has to drop all of its adj. vertices.

Industry req.:

1) In a comp. network, we want to put diff. ~~workstat~~ workstations/hops/servers. So how to put them so that cost will be less?

2) If any such Naxalite point exists, we should avoid them and take the longer path because the wires may get caught many times.

Prime Merit
Page No.
Date

Scanned with CamScanner

Dynamic Programming:

Games:

1) There are 2 tables, each ~~con~~ table contains 10 ~~one~~ ₹1 coins. Two players play the game where a player can alternatively opt to remove either one coin from any table or 2 coins from both the tables. (If the player removes 2 coins, then it has to be one from each table.). The player who removes the last coin(s) is declared as the winner. Do you want to be the player 1 or player 2?

2) In an 8x8 chess board, a rook is placed at the leftmost top pos. (source). It would be brought to the rightmost bottom pos. (dest.) by 2 players ~~at~~ by moving alternatively. (The ~~roo~~ rook is not allowed to move back ~~either move~~ i The player bringing the rook to the dest. is declared as the winner. Do you want to be player 1 or player 2? (rook movements allowed – right, bottom)

3) Same as Q2, but instead of a rook, there is a queen and the dest. ~~ino~~ is specified (queen ~~&~~ no by the user. Eg: (8,7), not necessarily (8,8). source = (1,1)

Prime Mart
Page No.
Date

Scanned with CamScanner

Soln.:

i) W → win , L → lose

The entry will be made w.r.t. player 1.
Player 1 wins → W
" loses → L

Table 2

↓	0	1	2	3	4	5	6	7	8	9	10	→ Table 1
0		W	L	W	L	W	L	W	L	W	L	
1	W	W	W	W	W	W	W	W	W	W	W	
2	L	W	L	W	L	W	L	W	L	W	L	
3	W	W	W	W	W							
4	L	W	L	W	L							
5	W	W	W									
6	L	W	L									
7	W	W	W									
8	L	W	L									
9	W	W	W									
10	L	W	L									

(0,3) → P1 : 2 coins
P2 : 1 coin

(0,4) → P1 : 2 coins
P2 : 2 coins

(0,5) → P1 : 2 coins
P2 : 2 coins
P1 : 1 coin

(1,2) → P1 : 1 coin (table 2)
P2 : 1 coin
P1 : 1 coin

Every player wants to win. So he will not choose in such a way that he will lose.

So if one table contains 1 coin and
i) another table contains odd no. coins → P1 should first pick up 2 coins, one from each table in order to win

ii) another table contains ~~odd~~ even no. of coins, then P1 should first pick up 1 coin from the table which has only 1 coin, ~~first~~ in order to win

So ~~pto p1~~ in this situation, p1 will always try to keep even no. of coins in the other table in order to win.

(2, 2) → P1 : 2 coins, P2 : 2 coins

or

P1 : 1 coin → table 1 (same table)
P2 : 1 coin → table 1

P1 : 1 coin
P2 : 1 coin

~~To compute (i,j)~~

Let no. of coins in table 1 = i
no. of coins in table 2 = j
→ (i,j)

So for P1, he has 3 options:

i) Move 1 coin from table 1 → (i-1, j)
ii) Move 1 coin from table 2 → (i, j-1)
iii) Move 2 coins → (i-1, j-1)

~~The~~ To compute the entry in (i,j), we should know the values in (i-1, j), (i, j-1) and (i-1, j-1)

If (i-1, j)
(i, j-1)
(i-1, j-1)
→ W (P1 wins)

then from (i,j), after performing any of the 3 options, P1 will change the ~~table~~ state to winning state (W) ~~and so, P2 will win.~~

Prime Merit
Page No.
Date

Scanned with CamScanner

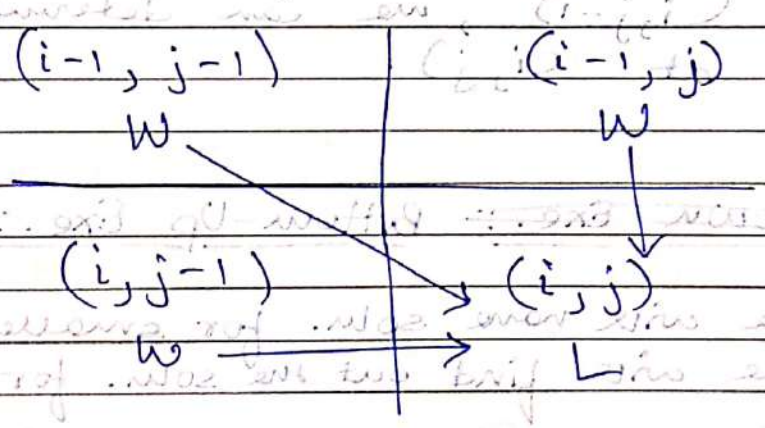
From this state, when P2 starts doing his move, P2 will always win.

Strategy : P1 will force P2 to lose the match.

If $(i-1, j-1) = L$
$(i-1, j) = W$
$(i, j-1) = W$

In this case P1 will move 2 coins from either tables so as to change the table state to losing state for P2 $(i-1, j-1)$

So if all the 3 choices are winning strategies, then P1 will lose. If any one of them is a losing strategy, then P1 will win.



So for (3, 3) → (2, 2) is L
so P1 will win
by removing 2 coins.

So by fill row 0 and col 0, the whole table can be filled up.

Prime Merit
Page No.
Date

Scanned with CamScanner

Strategy:

If any of the tables contain odd no. of coins → P1 will win by removing coins in such a manner that both the tables should contain even no. of coins each.

If both the tables contain even no. of coins → P2 will win because by doing any move, P1 will not be able to keep even no. of coins in each table.

In the same way for games no. 2 & 3 we can determine which player will win by making a table and following the same principle, i.e., by knowing the entries in $(i-1, j-1)$, $(i-1, j)$ and $(i, j-1)$, we can determine the entry at (i, j).

~~Top-Down Exe.:~~ Bottom-Up Exe.:

Here, we will have soln. for smaller values and we will find out the soln. for larger values.

Eg.: The above games follow ~~top-down~~ bottom-up approach.

~~Bottom-Up Exe.:~~ Top-Down Exe.:

Here, the soln. for smaller ~~prob~~ values prob. is not given. So to compute the ~~soln~~ soln. for larger values

probs., we need to first calc. the soln. for smaller value probs.

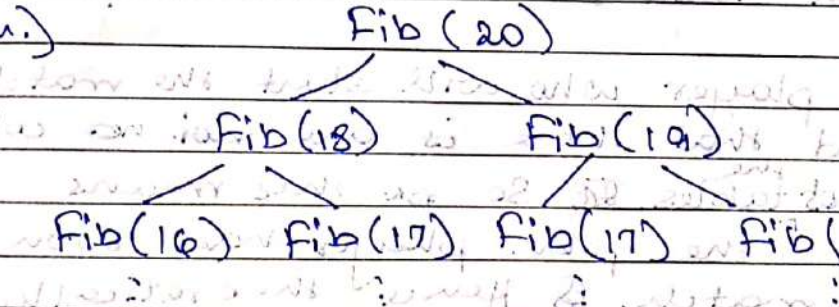
Eg.: In the above game, to compute $C(i,j)$, we need to compute $(i-1,j-1)$, $(i-1,j)$ and $(i,j-1)$.

To compute $(i,j-1)$, we also need $(i-1,j-1)$
To compute $(i-1,j)$, we also need $(i-1,j-1)$.

So certain computations will be carried out no. of times when we go for top-down approach. (from larger value to smaller value). It takes a lot of time because the smaller values are computed no. of times.

Eg 9) Use recursive func. to find the 20th fibo. no.

soln.)



~~So~~ ~~For~~ Here, this follows top-down approach because the smaller values are not known.

So maybe the 8th fibo. no. might be computed for 30 times.
4th fibo. no. → computed for 200 times

But every time the ~~same~~ 5th fibo. no. will have the same value irres. of the no. of times it is calc.

Prime Mark
Page No.
Date

Scanned with CamScanner

So to avoid this, we can use an array to find the ith fibo. no.. Store the first and second fibo. no. as 1, 1, resp. and go for a bottom-up approach. Find fib(3), fib(4), ..., fib(20). So here, every time we calc. fibo(i), fib(i-1) and fib(i-2) are alrady available. In this case, none of the fibo. nos. will be calc. more than once.

So bottom-up approach of exe. gives a faster exe. in comparison ~~to~~ to top-down approach.

Dynamic prog. uses bottom-up approach.

Q) Who will win the game if there are no coins in table 1 and table 2?

Soln.) The player who will start the match, will find that there ~~is~~ are ~~not~~ no coins in any of the tables. ~~Tf~~ So ~~so~~ this means possibly the prev. player had won the match. ~~Q~~ Hence, theoretically, player 2 will win the match.

Dynamic Prog.

Here, some recursive strategy/approach is used to find the soln. for smaller probs. Every soln of these smaller probs. will be kept in the mem. for later use.

Prime Merit
Page No.
Date

Scanned with CamScanner

Mem. will not be reqd. if we go for top-down approach. But for bottom-up approach, there must be some mem. where the soln. values for the lower i/p probs. are available. Then by using them, we can find the soln. for probs. to with larger i/p.

Problems:

exam

1) Longest Common Subsequence (LCS)

It is done b/w. 2 or more no. of strings.

substring – Any no. of consecutive symbols/characters in a string.

Eg.: String – ABCBDAB
Substring – AB, ABC, ABCB, DAB, BDA

subsequence – Symbols/characters will appear one after another in the string (not necessarily consecutive). This means, i^{th} and $(i+1)^{th}$ char. in subseq. must be present in the string where i^{th} char should appear first, and then $(i+1)^{th}$, but they might not be consecutive. So from a string by removing 0 or more no. of symbols, we can have a subseq.

Eg: String – ABCBDAB
Subseq. – C, ACA, CDA, CBA, CDB, BBB

Every substring is a subseq. but ~~vice versa~~ every subseq. is not necessarily a substring.

Prime Merit
Page No.
Date

Scanned with CamScanner

For a pair of strings :

$X = \{A\ B\ C\ B\ D\ A\ B\} \rightarrow [1 \ldots m]$
$Y = \{B\ D\ C\ A\ B\ A\} \rightarrow [1 \ldots n]$ } m & n no. of symbols.

First

i) ~~First~~ find the subseq. that is common to both the strings.

∴ Subseq. = BD, BCA, AA, DAB
(2) (3) (2) (3) → length

ii) Find the length of the subseq. (no. of symbols present in the subseq.)

iii) Find the longest subseq. that is common to both.

A prob. might have more than one optimal soln., but everytime the length should be the same.

X = A B C B D A B
Y = B D C A B A

Subseq. = B C B A

Soln. to LCS:

X, Y → strings ~~of so~~ containing a, b, c, d where first char. has an index = 1.

same length

1 2 3 ..... i-1 i
X = b a a c b d c c b d d a b
Y = c c a b b b d c a a c d b
1 2 3 ..... j-1 j

Find the soln. from 1 to i^{th} pos. of ~~X~~ X and j^{th} pos. of Y. Later on extend it to n.

Prime Merit
Page No.
Date

Scanned with CamScanner

Find the suboptimal soln. upto $(i-1)$ & $(j-1)$ and add 1 to the length of the suboptimal soln. if $x[i] == y[j] \rightarrow$ optimal soln.

If $x[i] \neq x[j]$:

~~Case~~ i) Find the optimal soln. till i and $(j-1)$
~~Case~~ ii) Find the optimal soln. till $(i-1)$ & j.

~~Among the~~ Find the optimal b/w ~~these~~ above 2. whichever is max. $\rightarrow$ optimal soln.

$x = baacbdccbd$ $\quad$ $x = baacbdccbd$
$y = ccabbbd$ $\quad$ $y = ccabbbdc$

max.

Let the optimal length of the LCS upto the chars. i and j in the strings x and y, resp. be $c[i,j]$.

$x \rightarrow 100$ chars. and $y \rightarrow 200$ chars.
We ~~will~~ may not find the optimal soln. upto 100 and 200, instead upto say 50 and 80, resp.
(But always find from 1 to 50, not 10 to 50 — should always start from 1).

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & , \text{ if } x[i] == y[j] \\ \max(c[i,j-1], c[i-1,j]) & , \text{ otherwise} \end{cases}$$

If $c[i,j] \rightarrow$ optimal,
then $c[i-1,j-1] \rightarrow$ optimal because it is recursive in nature.

Prime Merit
Page No.
Date

Scanned with CamScanner

Once the optimal soln. ~~upto i,j~~ from i to j is found out, we can find the soln. from m to n, where m → length of X
n → length of Y

Every ~~recurs~~ dynamic prog. ~~prob~~ soln. will start with an initial value. → like recursion

→ $c[0,0] = 0$

~~Here~~ If length of ~~X, Y~~ X & Y = 0 (null), then length of subseq. = 0. (string = ϕ)

If either X or Y is ~~(null)~~ null, then length of subseq. = 0

→ $c[0,j] = c[i,0] = 0$

Eg:) X = A B C B
Y = B D C A B] → length of X, Y may not be same

m = |X| = 4
n = |Y| = 5

x_i \ y_j	B	D	C	A	B
0 A					
1 B					
2 C					
3 A					
4 B					

Prime Merit
Page No.
Date

Scanned with CamScanner

$c[i,j]$:

		0	1	2	3	4	5
x_i \ y_j			B	D	C	A	B
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

~~for i = 0 and~~

for 0th row and 0th col., all values are 0 → initialization.

if $(x_i == y_j)$

$c[i,j] = c[i-1,j-1] + 1$

else

$c[i,j] = \max(c[i-1,j], c[i,j-1])$

For $c[1,1]$: $A \neq B \rightarrow 0$

$c[1,2]$: $A \neq D \rightarrow 0$

$c[1,4]$: $A = A \rightarrow c[0,3] + 1 = 1$

$c[1,5]$: $A \neq B \rightarrow \max(c[1,4], c[0,5])$

$= 1$

$c[2,5]$: $B = B \rightarrow c[1,4] + 1$

$= 2$

So for X and Y of length 4 & 5, resp., length of LCS = 3.

For X and Y of length upto 2 & 5 resp., length of LCS = 2

To find the chars. of the subseq., we need to perform a backtracking method (indicated by arrows — from which cell the value is obtained).

Once '0' is reached — no backtracking is reqd.

Prime Merit
Page No.
Date

Scanned with CamScanner

Keep note of the arrows that are obtained from corner. → matching chars.

Corners: c[2, 1], c[3, 3], c[4, 5]

↓ ↓ ↓

B C B] → either row/col.

~~Sub~~ LCS = BCB

Algo:

```
LCS_length (X, Y)
{
    m = length(X);
    n = length(Y);
    for i = 0 to m do
        c[i, 0] = 0;
    for j = 0 to n do
        c[0, j] = 0;
    for i = 1 to m do
    {
        for j = 1 to n do
        {
            if (X[i] = = Y[j])
                c[i, j] = c[i-1, j-1] + 1;
            else
                c[i, j] = max(c[i-1, j],
                              c[i, j-1]);
        }
    }
    return c;
}
```

→ initialization

Prime Merit
Page No.
Date

Scanned with CamScanner

Exe. Time:

i loop → m
j loop → n

∴ ~~Exe~~ $\boxed{\text{Exe. Time} = O(mn)}$

To keep a track of the corner arrows ~~change~~ → same algo., only change if+else

```
if (X[i] == Y[j])
{
    c[i,j] = c[i-1, j-1] + 1;
    p[i,j] = 'c';          → c: corner
}
else
{
    c[i,j] = max(c[i-1,j], c[i,j-1]);
    if (c[i,j] == c[i-1,j])
        p[i,j] = 'T';      → T: top
    else
        p[i,j] = 'L';      → L: left
}
```

for getting the ~~subse~~ chars. of the LCS, read the position matrix and wherever there is a corner (c), note down the chars. The result obtained from all the corner chars. → LCS.

p[] → position matrix

Exe. Time of traceback algo. to find LCS:

$\boxed{\text{Exe. Time} = O(m+n)}$

Worst case: No corner ele. found → traverse whole row then whole col.

Best case: All matching ele. → traverse only diag. ele.

~~while LCS_characters (p[], x, y)~~
~~{~~
~~LCS = 0, c =~~

```
LCS_characters (p[], x, y)
{
    m = length(x);
    n = length(y);
    c = p[m][n];
    ~~seq = "";~~ char seq[];
    while (c != '\0')
    {   seq + = c;
        if (c == 'C')
            c = p[i-1][j-1];
        else if (c == 'T')
            c = p[i-1][j];
        else if (c == 'L')
            c = p[i][j-1];

        ~~seq + = c;~~
    }
    return seq;
}
```

```
LCS_characters (p[], x, y)
{
    i = m = length(x);
    j = n = length(y);
    c = p[i, j];
    char seq[];

    while ( c != '\0')
    {
        ~~seq += x[i-1];~~

        if ( c == 'c')
        {
            seq += x[i-1];
            c = p[--i, --j];
        }
        else if (c == 'T')
            c = p[--i, j];
        else if (c == 'L')
            c = p[i, --j];
    }

    reverse (seq);
    return seq;
}
```

These algo. can also be ~~used~~ extended further to find the LCS b/w more than 2 strings also.

Prime Mental
Page No.
Date

Scanned with CamScanner

2) Shortest Path Problems:

Type 3 Using Floid Warshall's Algo.:

Here, we have to find the shortest path b/w every pair of vertices using DP approach.

Given a weighted directed graph, G, find the min-weight path for every pair of vertices. Weight of path is sum of edges.

Assume that we have already determined the path from vertices 1 to k-1 (may not be the shortest path) ~~throu~~ with the help of some calc. Next step, we will ~~calc. the dist~~ include the vertice 'k' in our calc.

Let source = i, dest. = j
inter. vertice = ~~keep~~ 0, 1, 2, ..., k-1
if source = 0

We have to find out if by including 'k', whether the path is the shortest one or not.
(may not be shortest)

The prev. calc. shortest dist. obtained by including vertex 'k-1' is compared with ~~that of~~ the ~~shortest~~ dist. obtained by including vertex 'k'.

Scanned with CamScanner

$$a^k[i,j] = \min\left(a^{k-1}[i,j],\ a^{k-1}[i,k] + a^{k-1}[k,j]\right)$$

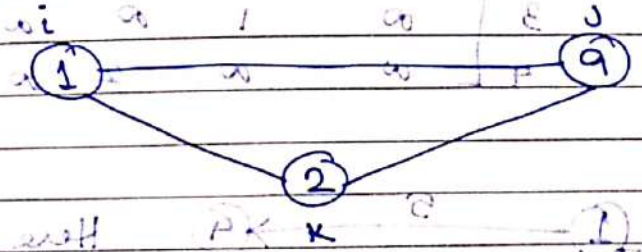
prev. dist. (may not be shortest)

new dist. (i to k + k to j)

If new dist. shorter, update $a^k[i,j]$ with the new dist., else keep it as it is.
↳ shortest dist. will be through k

$a^k[i,j] \rightarrow$ shortest dist. b/w i to j through k.

Eg:



Vertices : 1, 2, 3, ..., 9.

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

cost matrix — given (direct dist. is given — w/o using any inter. vertex)

For k=2: $a^2[1,9] = \min\left(a^1[1,9],\ a^1[1,2] + a^1[2,9]\right)$

directly obtained from cost matrix (direct dist.)

indirect dist.

⋮

For k = 5

⋮

For k = 8

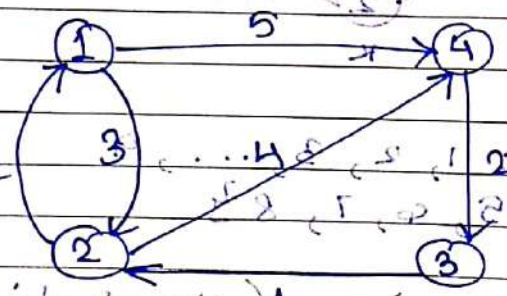
Once the soln. is obtained through 5, next we will find it through 6 and so on.

If the calc. is done upto $K = 5$, then it means we have already compared the direct distance through 2, 3, 4, 5 and shortest dist. have been determined among them. But we are yet to determine the shortest dist. through 6, 7, 8.

Eg.:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \infty & 3 & \infty & 5 \\ 2 & 2 & \infty & \infty & 4 \\ 3 & \infty & 1 & \infty & \infty \\ 4 & \infty & \infty & 2 & \infty \end{array}$$

weighted cost/matrix



Here, (1,1), (2,2), ..., (4,4) = ∞ → self edge

The cost matrix is considered as a^0 matrix.

So from a^1, find a^2.

From a^2, find a^3, and so on.

$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & \infty & 1 & \infty & \infty \\
4 & \infty & \infty & 2 & \infty
\end{array}
\longrightarrow
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & \infty & 1 & \infty & \infty \\
4 & \infty & \infty & 2 & \infty
\end{array}
$$

a^0 $\qquad$ a^1

$$a^1[2,3] = \min(a^0[2,3],\ a^0[2,1] + a^0[1,3])$$
$$= \min(\infty,\ 2+\infty)$$
$$= \infty$$

$$a^1[2,4] = \min(a^0[2,4],\ a^0[2,1] + a^0[1,4])$$
$$= \min(4,\ 2+5)$$
$$= 4$$

$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & \infty & 1 & \infty & \infty \\
4 & \infty & \infty & 2 & \infty
\end{array}
\longrightarrow
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & 3^2 & 1 & \infty & 5^2 \\
4 & \infty & \infty & 2 & \infty
\end{array}
$$

a^1 $\qquad$ a^2

inter.

$3^2 \rightarrow$ value $= 3$, vertex $= 2$

↳ does not mean 3.

$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & 3^2 & 1 & \infty & 5^2 \\
4 & \infty & \infty & 2 & (\infty)
\end{array}
\longrightarrow
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & \infty & 3 & \infty & 5 \\
2 & 2 & \infty & \infty & 4 \\
3 & 3^2 & 1 & \infty & 5^2 \\
4 & 5^3 & 3^3 & 2 & \infty
\end{array}
$$

a^2

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

a^3 $\qquad$ a^4

Prev. matrix → ~~no~~ cells having no superscript → cells will have value 0 in the superscript matrix.

Prime Merit
Page No.
Date

~~Suppose there are 100 vertices. If 1 to 50 the shortest dist. is 5000 and through inter. vertex 6~~

Super-script matrix

(In algo.-
i) dist./cost matrix
ii) super-script matrix)
OR
(~~structure~~ matrix containing str. ele - dist, superscript pos)

	1	2	3	4	5	6	7	8	9	10
1			10			3			6	0
2										
3										
4										
5										
6										
7										
8										
9										
10			7							

Q) Suppose ~~there~~ there are 10 vertices. If the shortest distance from vertex 1 to vertex 9 is through vertex 6, then is it necessary that the shortest dist. will be from vertex 1 to vertex 6 and then from vertex 6 to vertex 9?

Soln.) (1, 9) → ~~sho~~ → shortest dist. from vertex 1 to 9 is through inter. vertex 6. (one of the inter. vertex is 6, but not necessarily the only inter. vertex).

Now, find (1, 6) → 3
Now, find (1, 3) → 10
Now, find (1, 9) → 0 (0 means direct path)
non-zero value - indirect path)

Scanned with CamScanner

Now, find (10, 3)
Now, find (10, 7)

In this way the shortest path is determined.

a^1 is computed from a^0. While computing a^2, we could have computed from a^0, but we compute it from a^1. So this means we are taking the shortest path which has already been calc. (a^1) to find a^2. Hence, a ~~situation will never occur where the dist. calc. through 1 was more~~

~~Algo. : (Floid Warshall's APSP)~~

~~FWAPSP (w) // w → cost matrix~~
~~{~~
~~for each edge (u, v) do // initialization~~
~~a[u][v] = w(u, v) of a^0.~~
~~}~~

No. of times matrix is computed = no. of vertices.

So every ele. of matrix will be computed for 'no. of vertices' times.

During computation, in every iteration, each and every ele. of the matrix is computed but only few are updated.

Prime Merit
Page No.
Date

Algo. – Floid Warshall's APSP

```
FWAPSP (w)    // w -> cost matrix
{
  for each edge (u,v) do   // initialization
      a[u][v] = w(u,v);
                 -> weight of edge (u,v)
  for k from 1 to |V| do   // kth iteration
    for i from 1 to |V| do
      for j from 1 to |V| do
        if (a[i][j] >> a[i][k] +
                       a[k][j])
          a[i][j] = a[i][k] +
                    a[k][j];
          pos. matrix -> p[i][j] = k;
          superscript matrix
}
```

Exe. Time :

$\boxed{T(n) = \Theta(n^3)}$ -> n = no. of vertices

Every ele. is calc. every time. 'n²' ele. are computed for 'n' times. -> $\Theta(n^3)$

exactly (not max.)

Scanned with CamScanner

5) Write an algo. to find the path of ~~the~~ this shortest path algo.

3) <u>Matrix Chain Multiplication:</u>

A chain of matrices are there, suppose 'n' matrices are there:

$A_1, A_2, \ldots, A_n$ with size

$(p_0 \times p_1), (p_1 \times p_2), \ldots, (p_{n-1} \times p_n) \rightarrow$ size

Each multiplication in a matrix $\rightarrow$ scalar mult.

5) No. of scalar mult. reqd. in $A_{5\times6} \times A_{6\times2}$

Soln.) resultant matrix size = 5×2

For each ele. in the resultant matrix, we mult. one of the rows of the first matrix with the first col. of 2nd matrix.

Each row in first matrix has 6 ele.
Each col. in second matrix has 6 ele.

$\therefore$ No. of mult. reqd. to get first ele. in resultant matrix = 6

$\Rightarrow$ For every ele. in resultant matrix, it requires 6 mult.

$\therefore$ No. of ele. in resultant matrix = $5 \times 2 = 10$

$\therefore$ Total no. of scalar mult. $= 10 \times 6$
$= 60$ (Ans)

Scanned with CamScanner

Prime Merit
Page No.
Date

#) For $A_{m \times n} \times A_{n \times p}$

No. of scalar mult. $= m \times n \times p$

row of matrix 1 — col. of matrix 1 — col. of matrix 2

$\therefore A_i \times A_{i+1} \rightarrow$ no. of scalar mult.

$P_{i-1} \times P_i \quad P_i \times P_{i+1} \qquad = P_{i-1} \times P_i \times P_{i+1}$

Eg: $A_1 \times A_2 \times A_3 \times A_4$

$(3 \times 5) \quad (5 \times 4) \quad (4 \times 2) \quad (2 \times 5) \rightarrow$ size

~~No. of co~~

Comb. to multiply more than 2 matrices:

i) $((A_1 \times A_2) \times A_3) \times A_4$

ii) $(A_1 \times A_2) \times (A_3 \times A_4)$

~~iii) $A_1 \times (A_2 \times A_3)$~~

iii) $A_1 \times ((A_2 \times A_3) \times A_4)$

etc.

For 2 matrices $\rightarrow$ 1 comb.

For 10 matrices $\rightarrow$ huge no. of comb.

~~No. of scalar mult.:~~

~~i) $((A_1 \times A_2) \times A_3) \times A_4 = (3 \times 5 \times 4) +$~~

Scanned with CamScanner

No. of scalar mult.

i) $((A_1 \times A_2) \times A_3) \times A_4$

$A_1 \times A_2 \rightarrow 3 \times 5 \times 4$ (scalar mult.)
$(3 \times 4) \rightarrow$ (size)

$(A_1 \times A_2) \times A_3 \rightarrow 3 \times 4 \times 2$ (scalar mult.)
$(3 \times 2) \rightarrow$ (size)

$((A_1 \times A_2) \times A_3) \times A_4 \rightarrow 3 \times 2 \times 5$ (scalar mult.)
size $(3 \times 5) \rightarrow$ size

$\therefore$ Total no. of scalar mult.
$= (3 \times 5 \times 4) + (3 \times 4 \times 2) + (3 \times 2 \times 5)$
$= 114$

ii) $(A_1 \times (A_2 \times A_3)) \times A_4 \rightarrow$ optimal

$A_2 \times A_3 \rightarrow 5 \times 4 \times 2$
(5×2)

$A_1 \times (A_2 \times A_3) \rightarrow 3 \times 5 \times 2$
(3×2)

$(A_1 \times (A_2 \times A_3)) \times A_4 \rightarrow 3 \times 2 \times 5$
(3×5)

Total no. of scalar mult.
$= (5 \times 4 \times 2) + (3 \times 5 \times 2) + (3 \times 2 \times 5)$
$= 100$

Scanned with CamScanner

iii) $(A_1 \times A_2) \times (A_3 \times A_4)$

$A_1 \times A_2 \rightarrow 3 \times 5 \times 4$
(3×4)

$A_3 \times A_4 \rightarrow 4 \times 2 \times 5$
(4×5)

$(A_1 \times A_2) \times (A_3 \times A_4) = 3 \times 4 \times 5$
(3×5)

$\therefore$ Total no. of scalar mult.
$= (3 \times 5 \times 4) + (4 \times 2 \times 5) + (3 \times 4 \times 5)$
$= 160$

~~So~~ It is seen that some of the mult. seq. takes less no. of scalar mult. than others. So we should find out which order of mult. to carry out to get the optimal no. of scalar mult.

~~From the above~~ Integer mult. takes a lot less time than floating point mult. in a comp.

Mult. are used in many app. like image processing. So by reducing the no. of scalar mult. computation time is also reduced.

Here, we are not interested to find out (matrix) mult. we want to find out the order in which matrices need to be multiplied if we have 'n' no. of matrices so as to get least no. of scalar mult.

Scanned with CamScanner

Let $c[i,j] \rightarrow$ minimum no. of scalar mult. reqd (optimal cost) to mult. $A_i, \ldots, A_j$.

In DP, when we find the optimal soln. for large prob., its soln to smaller prob should also be optimal $\rightarrow$ suboptimal soln. (bottom-up)

So to find suboptimal soln, find ~~a~~ ~~mult~~ optimal soln (min. cost) from $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, \ldots, A_j$. Then these 2 matrices will be multiplied again to get optimal soln.

$k \rightarrow$ inter. pos.

$$A_i \times A_{i+1} \times \ldots \times A_k \rightarrow R$$

$$\downarrow \qquad \downarrow \qquad \downarrow \qquad \downarrow$$

$$(i-1 \times i) \quad (i \times i+1) \quad (k-1 \times k) \quad (i-1 \times k)$$

$$A_{k+1} \times \ldots \times A_j \rightarrow R$$

$$\downarrow \qquad \downarrow \qquad \downarrow$$

$$(k \times k+1) \quad (j-1 \times j) \quad (k \times j)$$

$$R_{(i-1 \times k)} \times R_{(k \times j)} = R_{(i-1 \times j)}$$

$$\Big\downarrow \text{ no. of scalar mult}$$

$$((i-1) \times k \times j)$$

Prime Merit
Page No.
Date

Generalised Optimal Cost:

Matrices $A_i, \ldots, A_j$

$$c[i,j] = \begin{cases} \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + r_{i-1} r_k r_j\}, & 1 \le i < j \le n \\ 0, & j = i, 0 \le i \le n \end{cases}$$

Choose k starting from i to j-1 and whichever is ~~min~~ giving min. cost → optimal soln.

If i = j → only 1 matrix is there. then, no mult. possible → 0 (initial/ boundary cond.)

Eg.: $A_1 \times A_2 \times A_3 \times A_4 \times A_5$

(5×3) (3×2) (2×4) (4×5) (5×6)

$p_0 \times p_1$ $p_1 \times p_2$ $p_2 \times p_3$ $p_3 \times p_4$ $p_4 \times p_5$

Find the order of chain of matrix mult.

super-script → k	1	2	3	4	5
1	0	$5\times3\times2 = 30^1$	70^2	120^2	190^2
2		0	$3\times2\times4 = 24^2$	70^2	136^2
3			0	$2\times4\times5 = 40^3$	100^4
4				0	$4\times5\times6 = 120^4$
5					0

Scanned with CamScanner

No. of matrices = 2 : (Diff. of 2)

$A_1 \times A_2 : k = 1$
$A_2 \times A_3 : k = 2$
$A_3 \times A_4 : k = 3$
$A_4 \times A_5 : k = 4$
$\rightarrow$ unique answer

Diff. of 3 :

$A_1 \times A_2 \times A_3 \quad : \quad k = 1, 2$
$A_2 \times A_3 \times A_4 \quad : \quad k = 2, 3$
$A_3 \times A_4 \times A_5 \quad : \quad k = 3, 4$

For $k = 1$: $A_1 \times (A_2 \times A_3)$

$A_2 \times A_3 = 3 \times 2 \times 4 = 24$
(3×4)

$A_1 \times (A_2 \times A_3) = 5 \times 3 \times 4 = 60$

$\therefore$ Total $= (24 + 60) = 84$

For $k = 2$: $(A_1 \times A_2) \times A_3$

$A_1 \times A_2 = 5 \times 3 \times 2 = 30$
(5×2)

$(A_1 \times A_2) \times A_3 = 5 \times 2 \times 4 = 40$

$\therefore$ Total $= 30 + 40$
$= 70$

For $A_1 \times A_2 \times A_3 \rightarrow$ optimal $= 70$, $k = 2$

Scanned with CamScanner

Prime Merit
Page No.
Date

For $A_2 \times A_3 \times A_4$:

For $k = 2$: $A_2 \times (A_3 \times A_4)$

$= (3 \times 2 \times 5) + (2 \times 4 \times 5)$

$= 30 + 40$

$= 70$

For $k = 3$: $(A_2 \times A_3) \times A_4$

$= (3 \times 2 \times 4) + (3 \times 4 \times 5)$

$= 24 + 60$

$= 84$

$A_3 \times A_4 \times A_5$

$k = 3$: $A_3 \times (A_4 \times A_5)$

$= (2 \times 4 \times 6) + 120$

$= 48 + 120$

$= 168$

$k = 4$: $(A_3 \times A_4) \times A_5$

$= 40 + (2 \times 5 \times 6)$

$= 40 + 60$

$= 100$

Scanned with CamScanner

Prime Merit
Page No.
Date

~~A_1 & A_2~~ Diff. of 4:

$A_1 \times A_2 \times A_3 \times A_4$

$k = 1$: $A_1 \times (A_2 \times A_3 \times A_4)$
$= (5 \times 3 \times 5) + 70$
$= 75 + 70$
$= 145$

$k = 2$: $(A_1 \times A_2) \times (A_3 \times A_4)$
$= 30 + 40 + (5 \times 2 \times 5)$
$= 70 + 50$
$= 120$

$k = 3$: $(A_1 \times A_2 \times A_3) \times A_4$
$= 70 + (5 \times 4 \times 5)$
$= 70 + 100$
$= 170$

$A_2 \times A_3 \times A_4 \times A_5$:

$k = 2$: $A_2 \times (A_3 \times A_4 \times A_5)$
$= (3 \times 2 \times 6) + 100$
$= 36 + 100 = 136$

$k = 3$: $(A_2 \times A_3) \times (A_4 \times A_5)$
$= 24 + 120 + (3 \times 4 \times 6)$
$= 144 + 72 = 216$

$k = 4$: $(A_2 \times A_3 \times A_4) \times A_5$
$= 70 + (3 \times 5 \times 6) = 70 + 90$
$= 160$

Scanned with CamScanner

Prime Merit
Page No.
Date

Diff. of 5:

Here $A_1 \times A_2 \times A_3 \times A_4 \times A_5$

$k = 1$: $A_1 \times (A_2 \times A_3 \times A_4 \times A_5)$
$= (5 \times 3 \times 6) + 136$
$= 90 + 136$
$= 226$

$k = 2$: $(A_1 \times A_2) \times (A_3 \times A_4 \times A_5)$
$= 30 + 100 + (5 \times 2 \times 6)$
$= 130 + 60$
$= 190$

$k = 3$: $(A_1 \times A_2 \times A_3) \times (A_4 \times A_5)$
$= 70 + 120 + (5 \times 4 \times 6)$
$= 190 + 120$
$= 310$

$k = 4$: $(A_1 \times A_2 \times A_3 \times A_4) \times A_5$
$= 120 + (5 \times 5 \times 6)$
$= 120 + 150$
$= 270$

So for multiplying $A_1, A_2, \ldots, A_5$, the least no. of scalar mult. reqd is 178 i.e., when $k = 2$.

~~Resultant seq. =~~

∴ Resultant Seq.

$= (A_1 \times A_2) \times (A_3 \times A_4 \times A_5)$

$= ((A_1 \times A_2) \times (A_3 \times A_4)) \times A_5$

Here, when we compute the values of the matrix, it is done diagonal wise (not rowwise / colwise)

Algorithm:

Size of matrix = $n \times n$

$n \rightarrow$ no. of matrices to be multiplied

Only the upper triangular matrix is calc. All the main diagl elem are initialized to zero

	1	2	3	4	5
1	0	A	B	C	D
2		0	A	B	C
3			0	A	B
4				0	A
5					0

It. 1) $i = 1 \rightarrow$ compute A

(1, 2), (2, 3), (3, 4), (4, 5)

It. 2) $i = 2 \rightarrow$ compute B

(1, 3), (2, 4), (3, 5)

It. 3) $i = 3 \rightarrow$ compute C
(1, 4), (2, 5)

It. 4) $i = 4 \rightarrow$ compute D
(1, 5)

So no. of iterations is from 1 to (n-1)

Outer loop $\rightarrow$ i
Inner loop $\rightarrow$ for accessing ele.

Inside every iteration, the first ele. always has row = 1 and the last ele. has row = n-i

$\therefore$ No. of ele. accessed in inner loop = $n-i$

Inside every iteration, the ele. accessed have col = row + i

Outer loop $i \rightarrow$ 1 to n-1
Inner loop $j \rightarrow$ 1 to n-i
$\hookrightarrow$ ele. = $(j, j+i)$

<u>Computation of $m[i, j]$:</u>

For i to j:
i) Compute i to k, $\forall i \leq k < j$
ii) Compute k+1 to j.

Multiply the results of these 2 inter matrices.

Scanned with CamScanner

```
for k = i to j-1
{
    t = m[i,k] + m[k+1, j] +
        p[i-1]p[k]p[j] ;   //p -> size
                             matrix
    if ( t < m[i,j])
    {
        m[i,j] = t;
        s[i,j] = k;   //s -> superscript
                        matrix
    }
}
```

<u>Computation of m[j, j+i]:</u>

```
for k = j to j+i-1
{
    t = m[j,k] + m[k+1, j+i] +
    p[j-1][k][j+i] ;
    if ( t < m[j, j+i])
    {
        m[j, j+i] = t;
        s[j, j+i] = k;
    }
}
```

Prime Merit
Page No.
Date

Algo.:

```
MATRIX_CHAIN_ORDER (p[], n, m[], s[])
{
    for i = 1 to n do    //initialization
        m[i, i] = 0;

    for i = 1 to n-1 do
    {
        for j = 1 to n-i do
        {
            m[j, j+i] = ∞;  //to store
                smallest value, initialize
                to ∞

            for k = j to j+i-1 do
            {
                t = m[j, k] + m[k+1, j+i]
                    + p[j-1]p[k]p[j+i];

                if (t < m[j, j+i])
                {
                    m[j, j+i] = t;
                    s[j, j+i] = k;
                }
            }
        }
    }
}
```

Input - [p[] → size matrix (containing matrix dim.)
n → no. of matrices]

result - [m[] → minimum cost table
s[] → split / superscript table]

Scanned with CamScanner

Exe. Time:

$$\boxed{T(n) = \theta(n^3)}$$

p[8] Size matrix can also be taken a 2D array where ~~every row~~ no. of rows will be the no. of matrices and each row will contain 2 values — dim. of each matrix (row, col.).

Size matrix = p[n][2]

Categorization of Problems:

Here, we will not write the algo. to the probs. The algo. is given, we will only ~~do~~ categorize the prob. to which category it belongs to. The categorization is done based upon their exe. time.

Categories: i) P
ii) NP
iii) NP Complete
iv) NP Hard

The probs. which have soln. in algorithm analysis can be categorized into any 1 of the 4 cat.

Scanned with CamScanner

Prime Merit
Page No.
Date

iii) NP - Completeness:

Intractable probs. - The probs. whose exe. ~~time~~ cannot be kept track of.
Eg: If an algo. takes > 100 yrs for exe. → cannot see the result ~~exe. can exe.~~
→ exe. cannot be kept track of.

Tractable probs. - Algos. whose exe. time is very fast, like secs, mins, hrs can be kept track of their exe.

Some algos. have ~~their exe. time~~ polynomial time exe. and some take super poly. time exe.

Poly. time exe. → order : $O(n^k)$
k = positive integer
Eg. : $O(n^2)$ (sorting), $O(n^3)$ (matrix chain mult.), $O(1)$, $O(n \log n)$ (sorting)

$n \rightarrow$ no. of i/p.

$O(n^2)$ colloquially means that 'n^2' (no. of st. would be executed. But actually it means that the no. of st. to be executed can be $an^2 + bn + c$ ~~(a, b, c consts)~~

for $n = 10,000$, $n^2 = 10^8$. So if millions of instr. are executed using this algo., then it would possibly complete its exe. in 10 - 20 secs for ~~n = 10,00~~ 10,000 no. of i/ps.
Because of poly. exe. time → tractable

Poly. solvable algos. are normally tractable.

Eg. If $k = 100 \rightarrow O(n^{100})$ and $n = 1000$.

No. of st. executed $= 1000^{100}$ → requires possible 500 yrs for computation → not tractable though its exe. is poly.

However, there are hardly any algos. which are poly. solvable and its computation time is $< n^{100}$.

Theoretically, for $n^k \rightarrow k$ can be any +ve const. But if the algo. is polynomially solvable, the order is found out be max. n^2 or n^3 (not even more than that.). So this is the reason due to which we say that poly. solvable algos. are tractable.

Super poly. exe. time:

The exe. time of these algo. is exp. × poly.

Eg.: $O(n^0 \cdot 2^n)$, $O(n \cdot 2^n)$, $O(n^n)$, $O(n!)$

If exe. time $= O(2^n)$ and $n = 100$ (moderate size of i/p.)

If we ~~exec~~ execute 2^{100} no. of st. in a super computer, its exe. time will be more than 100 yrs.

So super poly. solvable algos. exe. are normally intractable in nature.

Eg.: exe. time is in the range of 100-200 yrs.

Eg.: $n = 2/5/6/10 \rightarrow 2^2 / 2^5 / 2^6 / 2^{10}$

In such cases, the exe. time may be less than 1sec. → tractable.

But in real life, super poly. solvable algos. have high values of 'n' → intractable

Eg.: In travelling salesperson (TSP) prob. → $n = 50/60$.

i) P: (Polynomial)

P class of probs. are solvable in poly. time in a deterministic machine.

Every machine that is seen in the real world today → deterministic

deterministic → o/p is very much predicted based upon the i/p.

Non-deterministic machines are a theoretical concepts (not available in real world).

Eg.: To find the largest no. from 'n' nos.

$T(n) = \theta(n)$ → deterministic

$T(n) = \theta(1)$ → non-deterministic

↳ can find the largest in 1 attempt

So for non-deterministic :
selection sort / bubble sort = $O(n)$ → because in every iteration, it can find the next largest el. in $O(1)$ and in 'n' iterations, the array would be sorted

~~So~~ In non-deterministic machine, super poly. solvable algos. exe. time can be reduced to $O(n^2)$ or $O(n^3)$ → theoretical approach

ii) NP : (Non-deterministic Polynomial)

This is the class of probs. which are solved in poly. time in a non-deterministic machine.

↳ det. is not used because non-det. machines do not exist.

Q) Full form of NP.

Ans) Non-deterministic Polynomial
(not Non Polynomial)

Decision Probs. :

These probs. have a soln. in 0 or 1 / true or false

Eg: Given a graph, does any path exist b/w the source node A and the dest. node P, ? Soln. is given in yes/no (not the dist. b/w the 2 nodes) of path

Every optimization prob. has a corres. decision prob.

~~Eg.: Knapsack prob. — do you have a soln.~~ → optimization prob.

Eg.: Knapsack prob. — max. cap = 100
→ Do you have a soln. where the profit ≥ 500? (decision prob.)

Eg.: Single source multiple dest. (optimization prob.)
→ Starting from vertex 1, do you have a shortest path to the vertex 5 where the dist. < 500? (decision prob.)

Eg.: Activity selection (optimization prob.)
→ Do you have a soln. where in a single pandal, I can have min. 6 activities? (decision prob.)

iii) NP Complete:

NP Completeness is ~~defin~~ discussed directly w.r.t. decision probs. ~~and~~ and not w.r.t. optimization probs. (But our probs. were optimization probs.)

Relation b/w optimization prob. to decision prob. as long as the ~~so~~ ~~comp~~ computation time is concerned:

1) Knapsack Prob.: If we find out the soln. to this prob., its corres. decision prob. can be found in $\Theta(1)$ amt. of time.

2) Shortest path :- After using Dijkstra ~~Bellman~~ or some other algo. to find the shortest path, the decision prob. soln. can be found in $O(1)$ time.

```
if (dist <= 500)
    return: true;
else
    return false;
```

So if we have the soln. to the optimization prob., we can quickly get the soln. to its corres. decision prob.

So finding the soln. to the optimization prob. and the decision prob. are not far ahead of each other (very close)

⇒ If soln. to optimization prob. is simple, then to decision prob. is also simple.

⇒ If soln. to decision prob. is hard, then the soln. to its corres. opt. prob. ought to be hard, because their ~~rele~~ relation is very close.

If the opt. prob. is solved in P → decision prob. is solved in P

If the decision prob. takes super poly. time → opt. prob. will also take super poly. time.

Scanned with CamScanner

Prime Merit
Page No.
Date

Verification of a Prob.:

Here, the soln. need to be verified whether it is correct or not. (we do not need to find the soln.)

Eg: Knapsack Prob.: a, b, c, d, e, f, g, h
→ decision prob. (not opt. → NP complete)

Once it is established that the decision prob. is in a certain class, its opt. prob. is also in the same class because the are time is not diff. b/w opt. & decision prob.

Suppose soln. : c, p, q, w
(full) (full) (full) (half)

If (profit of c × c) + (profit of p × p) + (profit of q × q) + (profit of w × w/2) ≥ 500 → soln. is verified to be true

< 500 → soln. is verified to be false

If 'n' no. of items are there, every time take the profit of each item and find the soln. → verification is done in poly. time.

Scanned with CamScanner

Eg.: Shortest Path Prob. : 1, 2, 3, ..., 10

~~Soln from 1 to 10 : 1, 2, 5, 3, 9, 6, 4, ...~~

decision prob.: Do you have a path existing from 1 to 10 where the cost is less than 1000 ?

Soln. : 1, 2, 4, 6, 5, 9, 4, 8, 7, 10

Verify if the soln. is correct

Add all the cost of the edges.

If 'n' no. of vertices are there, we need to add 'n-1' no. of edges. In reading 'n-1' no. of cost from cost matrix, computation time is ~~O(n)~~ order of 'n'.

If total cost < 1000 → soln is verified to be true

≥ 1000 → soln is verified to be false.

~~Here, we~~ If we can verify the soln. whether it is correct or not in poly. time, the prob. belongs to NP.

used Def. of NP : If the soln. is verified in poly. time, in deterministic machine, then it belongs to NP.

This means P probs. → solved quickly (opt. prob. directly & verified)

NP probs. → verified quickly but might not be solved quickly (decision prob. directly (opt. → indirectly))

Prime Merit
Page No.
Date

Imp!

Reducibility :

The crux of NP : completeness is reducibility

Eg: Prob. whose soln. exists : $ax^2 + bx + c = 0$ → Prob. Y
Prob. whose soln. is unknown : $5x + 6 = 0$ → Prob. X

$5x + 6 = 0$ can be recast into a prob. $ax^2 + bx + c = 0$ with $a = 0$.
$\Rightarrow 0x^2 + 5x + 6 = 0$

~~So if the recasting time of a prob. is poly. and~~

If a recasting is done from prob. X to Prob. Y and if Y is solved, then there exists a soln. of X

If X to Y conversion time is poly. (P) and soln. of Y is in poly. time (P), then X will also be solved in poly. time (P) (P + P = P → addition of 2 poly. is a poly., mult. of 2 poly. is also poly.)

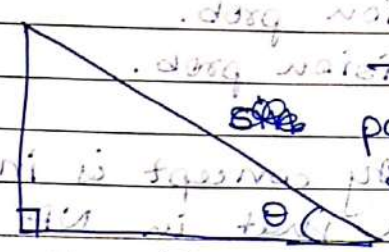
$\Rightarrow$ X is no harder to solve than Y if Y is solved in poly. time because X → Y is done in poly. time

Y is solved → X is solved
Y is hard → X is hard

Eg.: $\sqrt{17^2 + 19^2}$ → Prob. X

Pythagoras Th. → Prob. Y

Recasting X to Y.



→ Draw on graph paper and measure the diagonal.

$= \sqrt{17^2 + 19^2}$

$\tan\theta = \frac{17}{19} \approx 1$

$\Rightarrow \theta = 45^\circ$

$\therefore$ hypotenuse $= \frac{17}{\sin 45^\circ}$

$\Rightarrow \sqrt{17^2 + 19^2} = 17\sqrt{2} = 24$

If the recasting is done in poly. (P) time and the right ∠ed Δ is solved in poly. (P) time, then the ori. prob. is also solved in poly. time.

This is known as reducibility where a given prob. is reduced in poly. time into another prob. whose soln is determined in poly. time, such that the soln. to the given prob. is found in poly. time

When we recast a prob. X to prob. Y, both of them should be of the same type

Eg: X → searching, X → boolean,
Y → searching, Y → boolean

X → decision prob.
Y → decision prob.

Reducibility concept is introduced because it is seen that in NP complete, all the probs. are related. Till now, no poly. soln. exists for a NP complete prob. All of them are solved in super poly. time. Eg.: $O(n^2 2^n)$, $O(n 2^n)$, $O(2^n)$.

Eg: Travelling Salesperson Prob. (TSP):

Diff. cities of India are interconnected. We start our visit from Delhi. We need to cover the rest of the cities of India and return back to Delhi by covering minimum of distance.

~~Suppose, the soln. : $O(n^2 2^n)$~~

Suppose, there exists a soln. to TSP which is of order $n^2 2^n$. But it is not proved ~~known~~ ~~whether~~ that it is the lower bound exe. time of TSP. This means that in future, there might be some soln. whose exe. time can have an order $< n^2 2^n$.

Eg.: For a comparison based sorting, lower bound = $n \log n$ → proved. (No one can find an algo. with exe. time $< n \log n$).

Scanned with CamScanner

So till now, whatever NP complete probs. we have, their exe. times are not in P. But ~~in future~~ no one has given a proof that it cannot be solved in P.

Q) Do NP complete problems belong P?

Ans) Till now, it does not. In future, if any of the NP complete probs. belong to P, ~~then~~ from then onwards every NP complete probs. will be solved in poly. time because they are computationally related.

All NP complete probs. are computationally so related that if one of the NP complete prob. is solved in poly. time, then every NP complete prob. will be solved in poly. time.

$P \leq_p Q$ → Soln. to prob. P will not be taking more than a poly. time soln. to prob. Q.
↳ (recasting time)
→ whatever time it takes to solve Q, P will be solved by max. another poly. times.

iii) NP Complete: (NPC)
→ decision prob.

Theory 1: A prob. P is NP, complete if:

i) $P \in NP$ → P must be verified in poly. time
ii) $R \leq_p P \ \forall R \in NP$ → there must be a known NP prob. R and we must recast ~~from R to P in poly. time.~~

So if there exists a soln. to R, there will be a soln. to P.

from R to P in poly. time. ~~If the~~ The recasting should be done for all R ∈ NP (for all NP probs. that exist) each in poly. time

Once it is proved that P is NP complete and it is hard prob., takes a lot of time of computation → many ppl might not run it in their comp. expecting their answer after 1 or 2 days (because NP-complete takes a lot of time for a moderate value of n — super poly.)

If $P \leq_p Q$ and P is NPC, Q is also NPC

~~iv) NP-Hard:~~

Theory 2: Prob. P is NPC if:

i) $P \in NP$

ii) $R \leq P$ for any $R \in NPC$ → recasting from R to P is done in poly. time and for any R ~~(known NPC prob.)~~ R belongs to NPC

R → known NPC prob.

Eg: Out of 1000 NP prob., 100 prob. belong to NPC. Out of this 100 prob., any one can be chosen as for R. Whichever prob. is chosen, if that is recasted to a new prob. P, in poly. time → $P \in NPC$

Using Theory 2 → easier to prove
Using Theory 1 → used to prove first NPC prob.

Cook's theorem proves the first NPC prob. ~~which is a~~, i.e., the circuit satisfiability prob.

iv) NP - Hard :

A prob. is NP-Hard if:

Theory 1: $R \leq_p P$ $\forall R \in NP$ → If R can be recast to P in poly. time for all R belongs to NP

So every NPC prob. must satisfy NP Hard prop. and NP prop.

Theory 2: $R \leq_p P$ for any $R \in NPC$

Q) Which is a really hard prob., NPC or NP Hard?

Soln.) NP Hard — If the ~~soln~~ prob. ∈ NP, atleast we can verify the soln. in poly. time, ~~which happens in c~~ So atleast in case of NPC, the soln. can be verified in poly. time (quickly). But in NP Hard, the soln. cannot be verified in poly. time. (Given a soln., it might take 200 yrs to verify it ~~— so hard~~) — so finding the actual soln. would take a lot more time). First cond. of NPC → easy cond.

Scanned with CamScanner

Prime Merit
Page No.
Date

Examples of NPC:

1) Graph coloring (reg. allocation)
2) Hamiltonian cycle
3) Hamiltonian path
4) Knapsack Prob.
5) Travelling salesperson prob.
6) Job scheduling with penalties
7) Etc.

Steps to prove a prob P to be NPC:

Assume that the first NPC prob. exists.

1) Pick up a known NPC prob. Q.

2) Reduce Q to P (known to unknown).
   - Do a transformation from Q to P.
     Since P & Q are decision probs.,
     P → 'yes' then Q → 'yes' or
     P → 'no' then Q → 'no'.
     So if we get a soln. to Q, we get a soln. for P.
   - Prove that ~~the soln~~ it runs in poly. time

NP Hard

3) Prove $P \in NP$

Scanned with CamScanner

Prime Merit
Page No.
Date

Derivation of NPC prob.:

Circuit_SAT
↓
Formula_SAT → boolean
↓
3_CNF_SAT
↓
Hamiltonian Cycle → graph
↓
TSP

1) Circuit_SAT:

It is a comb. of AND, OR, NOT gates ~~Every circuit has some i/p~~ It is a boolean circuit which has i/p – x_1, x_2, x_3.

I/p x_1	x_2	x_3	O/p Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0 → circuit is not satisfiable
1	0	0	0
1	0	1	0 (whatever i/p is given o/p = 0)
1	1	0	0

No. of comb. of i/p $= 2^3 = 8$

The circuit satisfiability testing is always done for designing of CPU.

Scanned with CamScanner

Even for smaller circuits, this test is done

~~So their s~~

There should exist atleast one i/p comb. for which the o/p should be 1. circuit is satisfiable (keep the ckt. in the bigger circuitry).

If not satisfiable → discard the circuit by conn. it with ground supply.

This prob. was initially proved by Cook to be NPC.

2) <u>Formula – SAT:</u>

Assuming that ~~the circuit~~ circuit-SAT belongs to NPC.

For every circuit, a boolean formula can be written. Eg: $Y = (x_1 + x_2 + x_3).(x_1 + \bar{x}_2 + x_3)$

So if a ckt is satisfiable → formula is also satisfiable.

Eg.: $Y = (x_1 + \bar{x}_2 + x_3).(\bar{x}_1 + x_2 + x_3).(\bar{x}_1 + \bar{x}_2 + x_3)$

Assume that there is a ckt. which has this formula.

~~Step 1) Formula – SAT belongs to NPC.~~

Formula_SAT is NPC?

Step 1) Formula_SAT $\in$ N.P.
Step 2) Circuit_SAT $\leq_p$ formula_SAT

We have to prove if Formula_SAT is NPC prob. ~~and~~ Given a prob., prove whether it is satisfiable. A formula is satisfiable if for any ~~given~~ i/p ~~cond.~~ comb., the o/p is 1.

For $x_1 = 1, x_2 = 1, x_3 = 0$. $\rightarrow$ soln. given (o/p = 1)

Step 1) Verify the soln. to prove if it is NP.

Subs. the values in the given eqn., we can verify the soln. in poly. time because the no. of terms is poly. $\longrightarrow$ NP.

Step 2) Reducing Circuit_SAT to Formula_SAT:

Every ckt. has a corres. formula. So for this given ckt. formula, we have a corres. ckt. consisting of AND, OR, NOT gates. To convert a ckt. to formula we need poly. time.

So if the formula is satisfiable, the ckt. is satisfiable and vice versa.

$\therefore$ Because circuit_SAT $\in$ NPC, now formula_SAT also belongs to NPC.

Scanned with CamScanner

3) 3_CNF_SAT: ~~(Third Conj.~~

3_CNF_SAT → 3rd Conjunctive Normal Form Satisfiability Prob.

CNF and decision tree normal form are given by mathematicians only.
→ SOP and POS.

Here, every term contains 3 vars. (3 CNF)
Eg.: (Formula_SAT example.)

If every term contains 4 vars → 4 CNF

Imp. 2 CNF → does not ∈ to NPC
↳ belong to P

3 CNF → ∈ NPC onwards

If there is a circuit, it can have a 3 CNF. If circuit → satisfiable, 3 CNF → satisfiable. Given a formula with 3 CNF, we can verify that the soln. belongs to NPC.

4) Hamiltonian Cycle:

Given a graph, start visiting from any one of the vertices and cover rest of the vertices exactly once and return back to the same vertex. → Hamiltonian cycle exists and graph is called Hamiltonian graph.

Scanned with CamScanner

Not all graphs contain a hamiltonian cycle.
(undir. → many times, dir. → not have sometimes)

Decision Prob. : ~~Does whether~~ Given a graph, does it ~~has a~~ have a hamiltonian cycle or not ?

~~Step 1)~~ Check for NPC. (For this prob., till now no soln. exists that is poly. solvable ∴ all are super poly.)

Step 1) Hamiltonian cycle ∈ NP
Step 2) 3-CNF-SAT $\leq_p$ Hamiltonian cycle, such that if the formula is satisfiable, then ~~a~~ the ~~cycle~~ graph has a Hamiltonian cycle. (reverse is also true).

Step 1) Let a graph have 10 vertices: 1, 2, 3, ..., 10.
source = 1 ~~destn = 1~~

Let soln. : 1, 4, 3, 7, 8, 5, 9, 6, 10, 2, 1

Using the cost matrix, find if there is a cost for (1, 4), (4, 3), (3, 7), ..., (2, 1).

If for any of the ~~cost~~ edges, the cost is ∞ → soln. is not correct. → $\Theta(n^2)$
We also have to check that the vertices given in the soln. are unique. → $O(n^2)$ or $O(n \log n)$ → sorting
→ ~~each takes~~ $O(n^2)$. → poly. time.

∴ The prob. is verified in poly. time.
→ prob. ∈ NP.

If a graph has a Hamiltonian cycle
→ no. of vertices = n
no. of edges = n

If a graph has no cycle:
→ no. of vertices = n
no. of edges = n-1

Step 2) 3_CNF is reduced to a graph in poly. time. (From a ~~3CS~~ 3 CNF boolean expn. → graph)

If for ~~an~~ one i/p comb. the formula is not satisfiable, then the graph does not have a Hamiltonian cycle.

The i/p. comb. for which the formula is satisfiable, for that same i/p. comb. the graph has a Hamiltonian cycle. → NPC.

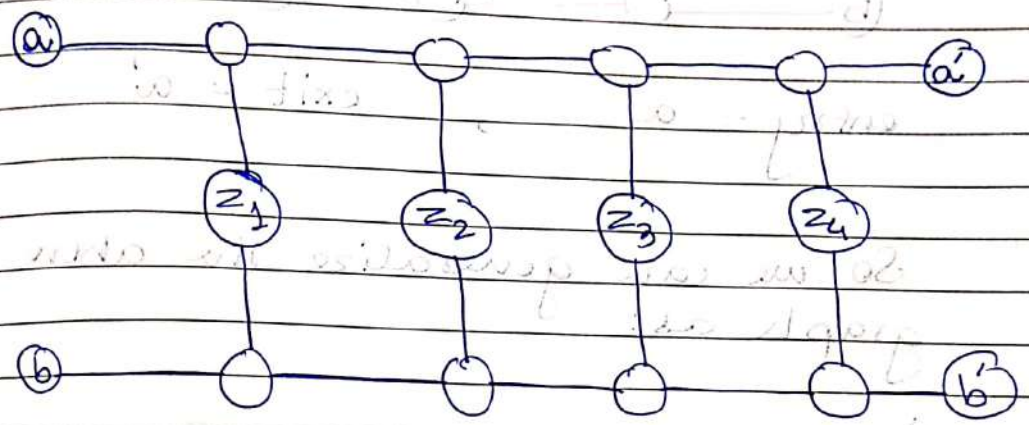
<u>Conversion of 3 CNF to graph:</u>

Widget A → part of a graph - subgraph (given a big graph, take a part of that graph)

Widget B → other part of the graph

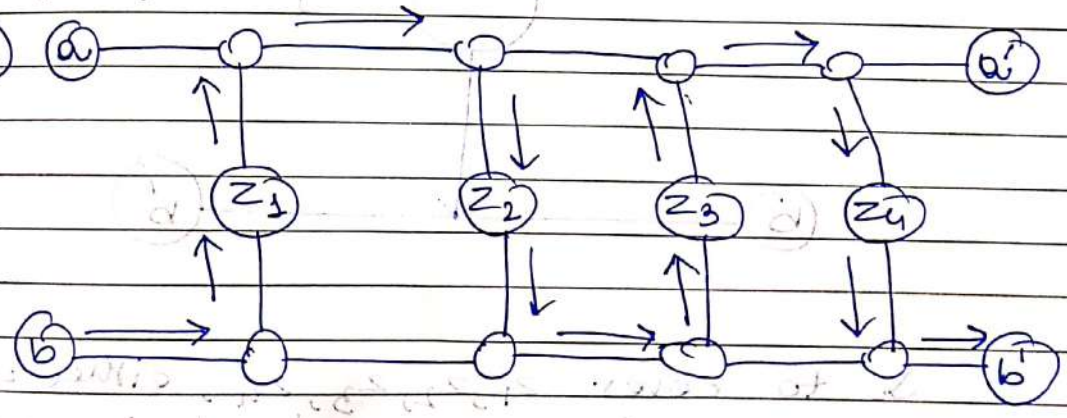
Eg.: Vertices a, b, a', b' are caught from a big graph → ~~subgraph~~ part of a graph

We would like to visit vertices z_1, z_2, z_3 and z_4 exactly once. ~~ex~~ – other vertices are removed.



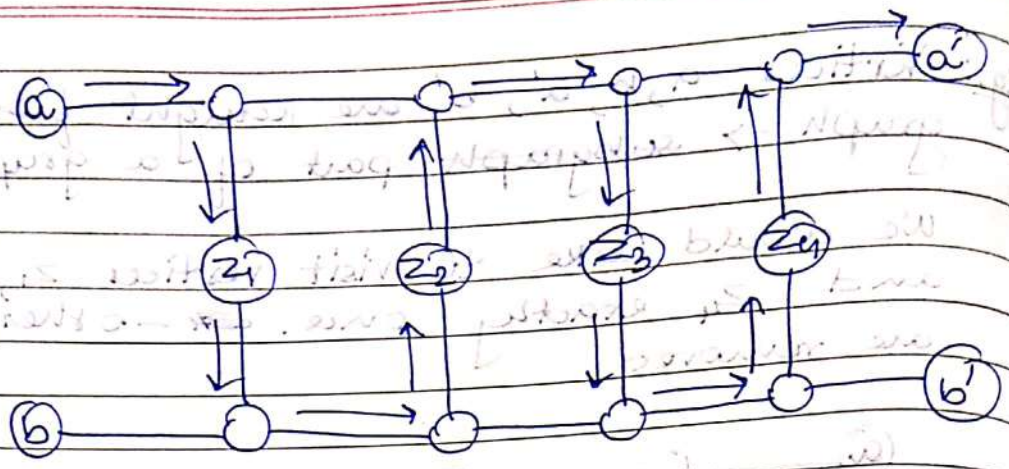
From the graph, we can enter this subgraph either at ~~a, b,~~ $a / b / a' / a'$.

Soln. 1)



entry – b , exit = b'.

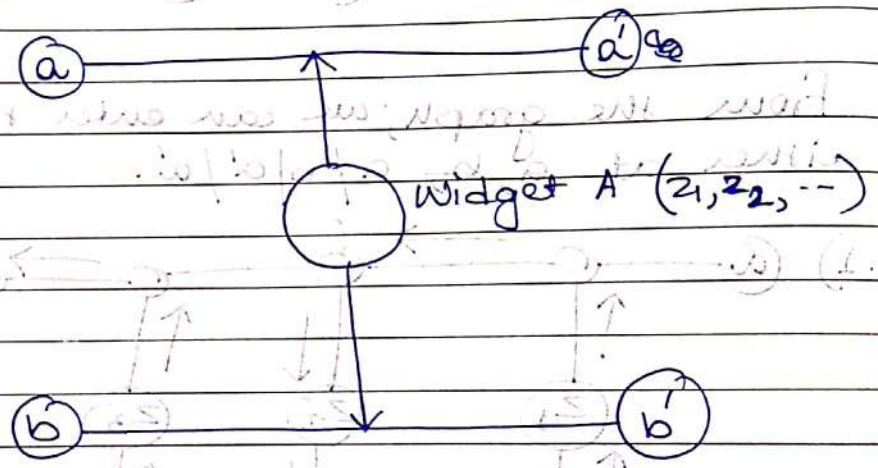
Soln. 2)



entry - a , exit = a'

So we can generalize the above graph as:

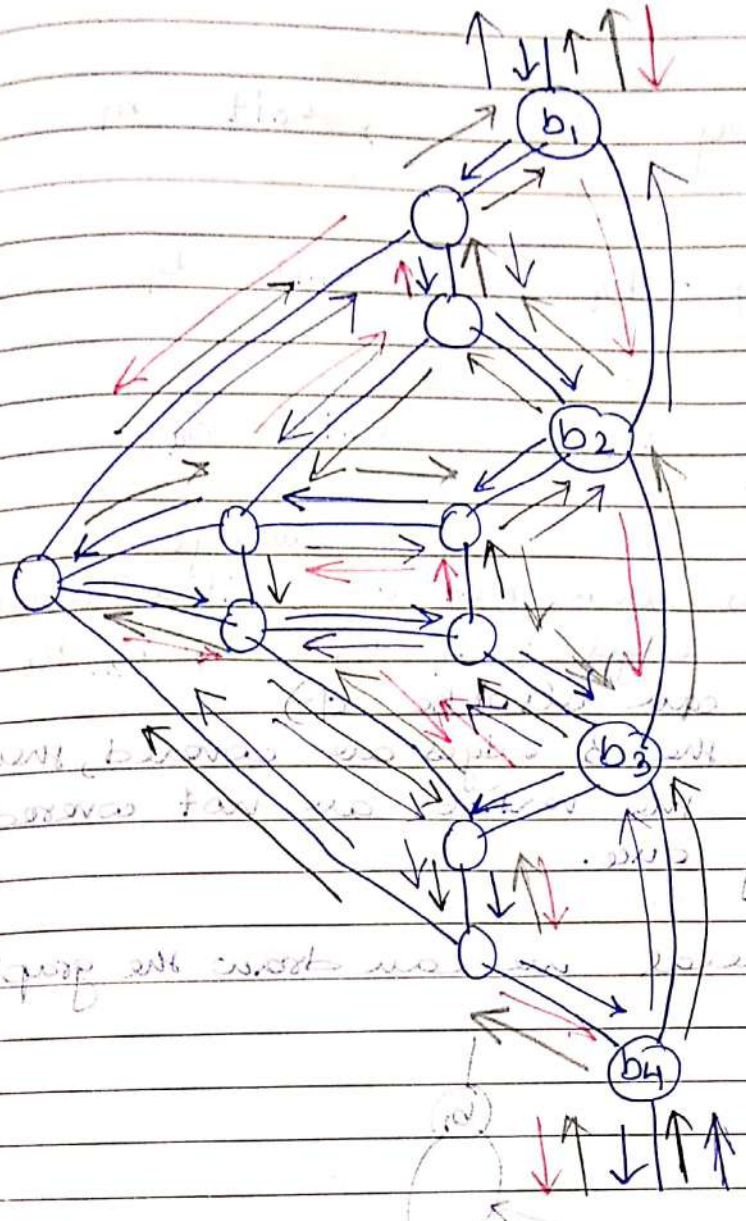
Imp!



So to cover z_1, z_2, z_3, z_4, either we cover a, a' or b, b', but never both.

If a widget is conn. b/w 2 edges we will cover either of the edges, not both, while covering the vertices atleast once.

Eg:



Cover vertices b_1, b_2, b_3, b_4 ~~atleast~~ along with the rest of the vertices atleast once. We can enter the graph either from b_1 or b_4.

Soln. 1) → (blue arrows)

entry – b_1 , exit – b_4

Soln. 2) → (black arrows)

entry – b_4, exit – b_1

Scanned with CamScanner

Prime Merit
Page No.
Date

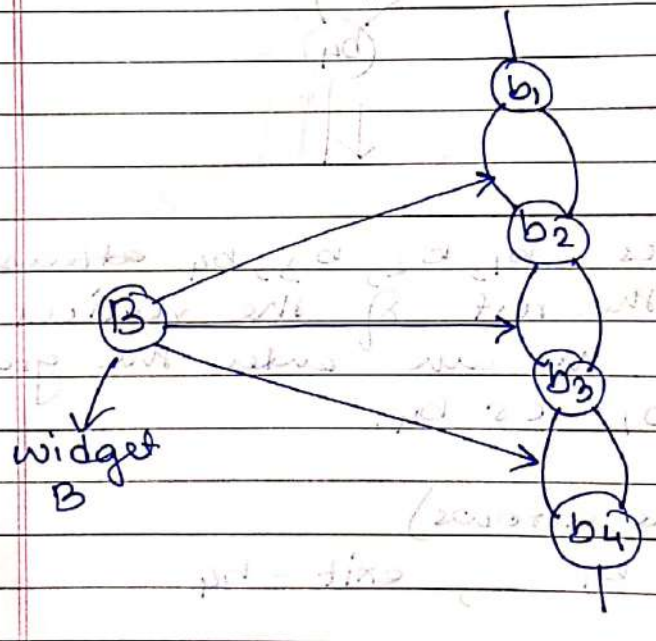
Soln. 3) → (pencil arrows)
entry - b_4 , exit - b_1

Soln. 4) → (blue arrows)
entry - b_4 , exit - b_1

Soln. 5) → (red arrows)
entry - b_1 , exit - b_4

∴ It is seen that we always visit a subset of the edges : $b_1 - b_2$, $b_2 - b_3$, $b_3 - b_4$
(subset can also be ϕ)
If all the 3 edges are covered, then rest of the vertices are not covered exactly once.

In general we can draw the graph as:



If widget B is conn to one side, all the other side edges will never get covered together. — a subset might get covered.

Scanned with CamScanner

covered. ~~If one~~ Only one of the edges b/w $b_1 - b_2$ ~~and $b_2 - b_3$~~ would get covered, not both. Same for $b_2 - b_3$ & $b_3 - b_4$.

At any point of time if we find out that all the RHS edges of b_1, b_2, b_3, b_4 are covered together, it means that all vertices of widget B are not covered.

Eg: $Y = \overbrace{(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)}^{\text{term}} \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + x_2 + \bar{x}_3)$ → 3 CNF (POS)

Convert formula → graph.

Step 1

Soln.) For every i/p var., there will be 2 vertices in the RHS of the graph.

$x_1 \rightarrow x_1', x_1''$

$x_2 \rightarrow x_2', x_2''$ → not complement

$x_3 \rightarrow x_3', x_3''$

∴ No. of vertices in the RHS = 2 (no. of i/p. × 2)

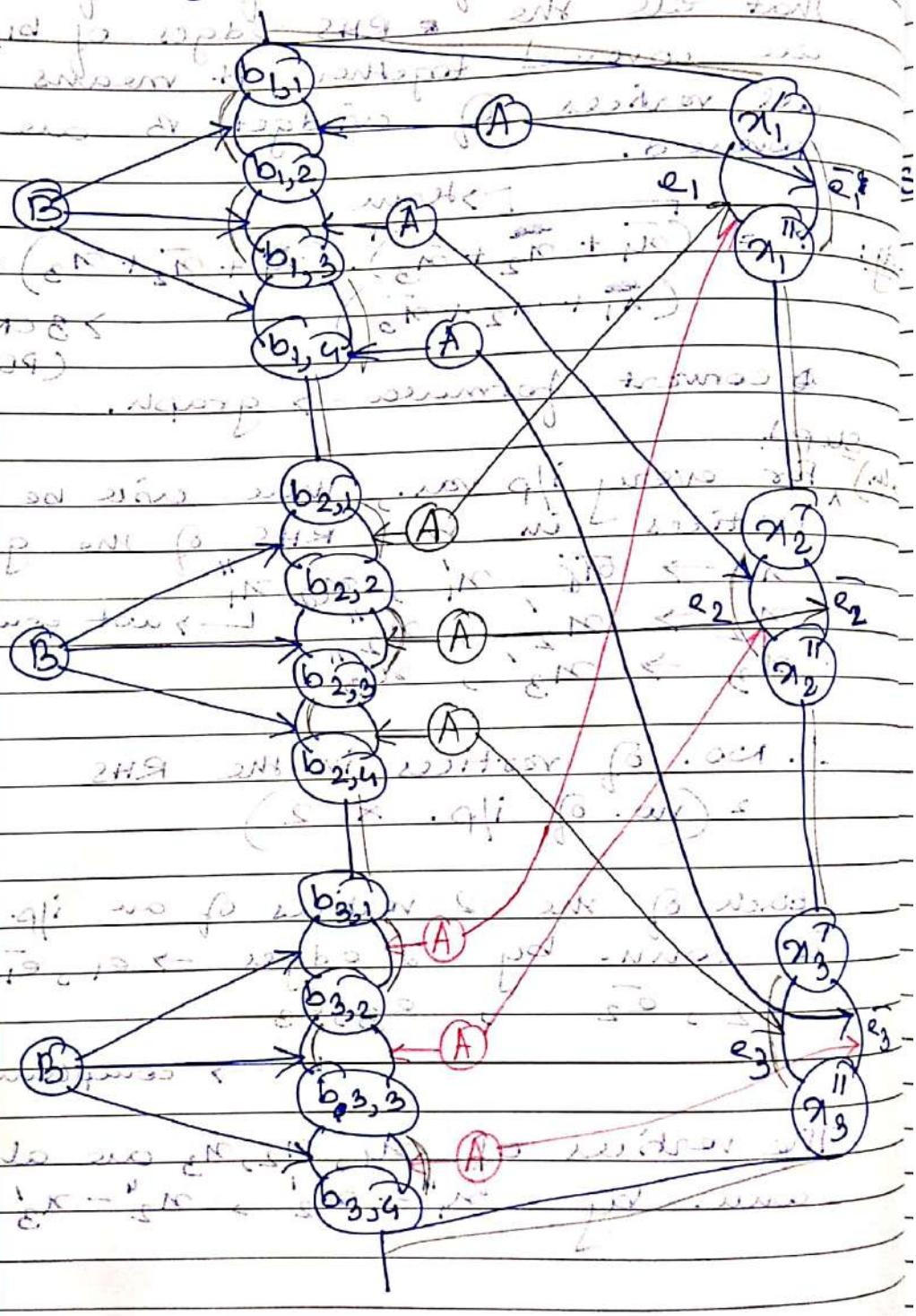
Each of the 2 vertices of an i/p var. is conn. by 2 edges → $e_1, \bar{e}_1$, $e_2, \bar{e}_2$, $e_3, \bar{e}_3$

→ complement

The vertices of x_1, x_2, x_3 are also conn. by $x_1'' - x_2'$, $x_2'' - x_3'$

Step 2) For every term, there will be a widget B. in the LHS.

The LHS and RHS of the graph are also coin.



Step 3) For each var. in a term, there will be a widget A coin.

Term 1 := $(\bar{x}_1 + x_2 + \bar{x}_3)$

$\bar{x}_1 \to e_1'$, $x_2 \to e_2$, $\bar{x}_3 \to \bar{e}_3$

Step 4) Let $x_1 = 0$, $x_2 = 1$, $x_3 = 1$

If for this value, the formula is satisfiable, the graph will have a Hamiltonian cycle.

$x_1 = 0 \to \bar{e}_1$
$x_2 = 1 \to e_2$
$x_3 = 1 \to e_3$
] → determines path

For each each widget A, if $\bar{e}_i$ is down, then the RHS of $b_{i,j}$ will be down.

If e_i is down, the LHS of $b_{i,j}$ will be down.

For each widget A, if its RHS is down, its LHS will not be down and vice versa.

Following these steps, we get a cycle in the graph (pencil marks).
So this graph has a Hamiltonian cycle w/o violating the property of widget B and the formula is satisfiable for the given i/p values.

Scanned with CamScanner

Property of gadget B: All of the 3 edges in each side will not be covered at once. (only a subset will be ~~covered~~ covered each time.) → All vertices are visited exactly once.

~~Eg: $x_4 =$~~

Eg: $x_1 = 1$, $x_2 = 0$, $x_3 = 1$

~~In the same way, the formula is satisfiable for the given i/p values → Hamiltonian~~

For the given values of i/p., the formula is not satisfiable → No Hamiltonian cycle exists in the graph.

(Graph - ~~gg~~ Hamiltonian.docx)

Here, the prop. of gadget B → unsatisfied

The conversion of formula to graph is done in poly. time and if the formula is satisfied for a given comb. of i/p., the graph has a Hamiltonian cycle for that comb. of i/p. Hence, it is a NPC prob.

Prime Merit
Page No.
Date

5). <u>Travelling Salespersons Problem (TSP):</u>

In a graph, a salesperson will start its visit from one of the vertex and visit/cover all vertices of the graph and returns back to the source vertex by covering min. dist. → optimization prob.

By using DP, soln. → $O(n^2 2^n)$.

<u>Decision prob.</u>: Does the graph have a TSP whose cost is 0?

Soln.) ~~The cost~~ In the graph, if there is an edge present b/w 2 vertices, convert the cost ~~=0~~ to 0. If there is no edge, the cost will remain as ∞. ~~or~~ So the cost matrix will be changed. (cost will not be 2, 5, 10, ... It can only be 0/∞.)

To prove TSP is NPC:

Step 1) <u>TSP ∈ NP</u>:

Let vertices = 1, 2, 3, ..., 10 ~~soln~~
source = 1
Soln. = 1, 5, 4, 8, 10, 3, ..., 1

Check 1 → 5 : 0 ✓
Check 5 – 4 : 0 ✓
Check 4 – 8 : ∞ ✗ → soln. is incorrect.

If all edges = 0, ~~so~~ soln. is correct.
~~So soln. is ve~~

Scanned with CamScanner

So the soln. is verified in poly. time.
→ NP.

Step 2) $\underset{(G)}{\text{Hamiltonian}} \leq_p \underset{(G')}{\text{TSP}}$

Recast G to G'. If G' has a TSP, G should have a Hamiltonian cycle.

Cost matrix is changed!

$$cost[i,j] = \begin{cases} 0, & \text{if edge exists b/w } i \text{ \& } j \\ 1, & \text{if no edge exists b/w } i \text{ \& } j \end{cases}$$

→ changing of G to G'. (G' has the cost matrix where each value can be either 0 or 1).

To convert the prev. cost matrix to a new cost matrix, we need $\Theta(n^2)$ time if there are 'n' no. of vertices.
→ poly. time.

If graph has a TSP whose cost = 0, then the graph has a Hamiltonian cycle.

If TSP cost > 0 → no Hamiltonian cycle.

TSP decision prob. can also be changed as:

Q) Does the graph have a TSP whose cost is 'k', where 'k' no. of vertices are there?

Soln) If 5 vertices are there, no. of edges in TSP = 5.
(n vertices → n edges, for TSP)

Cost matrix:

$$cost(i,j) = \begin{cases} 1, & \text{if edge exists b/w } i,j \\ 0, & \text{if no edge exists b/w } i,j \end{cases}$$

If there is a TSP with cost = k → Hamiltonian cycle exists.

If cost < k → some 0s were added
↓
no edge exists at some i,j
↓
No Hamiltonian cycle

~~A graph can have multiple Hamiltonian cycle.~~

A graph can have only 1 Hamiltonian cycle.

Scanned with CamScanner