Date / /
Page No.

Notes

Module-1

Algorithms: It is a set of steps to complete a task.

like make Tea().

Algorithms are we everywhere like showing directions in maps, or next reel on insta or you may like on netflix, etc.

→ It is important subject, also because avg. marks each year in GATE for this is 8.

→ Pagerank algorithm is what google uses to show us the pages.

→ Any algo has ~~to~~ two performance matrices i.e. space & time complexity.

→ We have already saw analyzing time for iterative programs in DS. Let's now analyse for recursive programs.

→
```
func(n) {
    if (n<=1)   return 1;
    return func(n/2)
}
```

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

→
```
fun(n) {
    if (n <=1) return 1;
    return fun(n-1) + n;
}
```

$T(n) = T(n-1) + 1$ → not 'n'.

GO CLASSES

Teacher's Signature _____

Notes

Date / /
Page No.

→ a(n) {
  1 * a(n/2)
}

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

→ a(n) {
  a(n/2) + a(n/2)
}

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

→ we have seen how to write recurrence. The very next part is solving recurrence to give asymptotic time complexity. (we don't need to solve exactly like in DM, we here approximate only).

T(n) : Time take by program

Our Objective is to find T(n)

→ We have three ways to achieve our objective i.e. solving recurrance relation.

a) Iterative / Repeated Substitution method

b) Recursive Tree Method

☆ c) Master's Method (Easiest)

Let's go over each of them one by one.

GO CLASSES

Notes

Date / /
Page No.

★ Iteration Method:

Ex:1 Given: $T(n) = T(n-1)+1$, $T(1)=1$

$T(n) = T(n-1)+1$ — ①

$T(n-1) = T(n-2)+1$ — ②

Using ② in ①

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

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

$\vdots$

$T(n) = T(n-k) \underbrace{+1+1+1+1 \cdots +1}_{k \text{ times}}$

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

as $T(1)=1$

$n-k=1$; $k=n-1$

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

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

$T(n) = n$

$$\boxed{T(n) = \Theta(n)}$$

So, the main goal is substitute repeatedly and observe the series.

GO CLASSES

Notes

Date / /
Page No.

Ex:2 $T(n) = T(n-1) + n$, $T(1) = 1$

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

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

$\vdots$

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

If, $n-k = 1$

then, $k = n-1$

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

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

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

However not needed to be exact

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

Ex:3 $T(n) = \begin{cases} T(n/2) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$

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

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

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

$\vdots$

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

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

$n = 2^k$, $k = \log n$

Teacher's Signature _____

GO CLASSES

Notes

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

↳ G.P. , Do sum, but <u>not</u> required.

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

↳ log n terms

$$= 1 + n\left(\frac{1\,(1-(1/2)^{\log n})}{1 - 1/2}\right) \qquad \left[\left(\tfrac{1}{2}\right)^{\log n} = \frac{1}{2^{\log n}} = \frac{1}{n}\right]$$

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

$$\underline{T(n) = \Theta(n)}$$

⇒ No need to sum G.P. in future, if it is decreasing G.P, its sum is always constant.

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

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

~~$T(n) = 2*$~~

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

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

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

Notes

Date / /
Page No.

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

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

$n = 2^i$ (To make this 1)

$i = \log n$

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

$= n\log n + n \cdot 1$

$= n\log n + n$

$= \Theta(n\log n)$

GO CLASSES

★ Tree Method:

Ex:1 $T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n$

Notes

Date / /
Page No.

Here left branch will reach the 1 earlier at compared to right most. However it do not matter because asymptotic.

for left:

$$\frac{n}{5^k} = 1$$

$$n = 5^k$$

$$k = \log_5 n$$

Similarly for right:

$$k = \log_{5/4} n$$

work done at each step is n, (check sum).

So, min. work done $\geq$ min. steps * work done at each steps

$$\geq \log_5 n * n$$

$$\geq n \log_5 n$$

$$T(n) = \Omega(n \log_5 n)$$

Max. work done $\leq \log_{5/4} n * n$

$$T(n) = O(n \log_{5/4} n)$$

Bu bases donot matter because they are constant.

Ex: $\log_{5/4} n$

$$= \frac{\log_5 n}{\log_5 5/4}$$

Just constant and base changes

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

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

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

GO CLASSES

Notes

Date / /
Page No.

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

So, similar steps to previous, work done at each step is n and no. of steps range from $\log_3 n$ to $\log_{3/2} n$

Crucial step hai

So $T.C = \Theta(n \log n)$

GO CLASSES

Notes

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

Everything is same as previous, the only diff is that here work done at each level/step is not 'n'. It is diff at each level/step. So we will have to sum all works

Level 0 : n^2

level 1 : $\left(\frac{n}{3}\right)^2 + \left(\frac{2n}{3}\right)^2 = \left(\frac{5}{9}\right) n^2$

level 2 : $\left(\frac{n}{3^2}\right)^2 + \left(\frac{2n}{3^2}\right)^2 + \left(\frac{2n}{3^2}\right)^2 + \left(\left(\frac{2}{3}\right)^2 n\right)^2 = \left(\frac{5}{9}\right)^2 n^2$

⋮

level k : $\left(\frac{5}{9}\right)^k n^2$

Notes

$k = \log_2 n$ to $\log_{3/2} n$

↳ Take any no matter.

Total work $T(n) \geq n^2 + \frac{5}{9}n^2 + \left(\frac{5}{9}\right)^2 n^2 + \left(\frac{5}{9}\right)^3 n^3 - - - + \left(\frac{5}{9}\right)^k n^2$

$T(n) \geq n^2 \left(1 + \left(\frac{5}{9}\right)^2 + \left(\frac{5}{9}\right)^3 + - - \left(\frac{5}{9}\right)^k\right)$

↳ Decreasing G.P., so no need to solve. Just put constant

$T(n) \geq n^2 * \text{constant}$

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

$T(n) = O(n^2)$ ⟶ because only k changed, which will chang dec. GP only

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

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

will stop after right unlike others

~~Work done = $1 + 2 + 4 + - - 2^{n/2}$~~

~~$= 2^0 + 2^1 + 2^2 + - - 2^{n/2}$~~

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

Teacher's Signature

Notes

Date / /
Page No.

Lower bound:

$2^0+2^1+2^2+\dots 2^{n/2}$

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

$= \Omega(2^{n/2})$

Upper bound:

$2^0+2^1+2^2+\dots 2^n$

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

$= O(2^n)$

Can't write Θ as cannot ignore constant and both bounds are different.

$2^n > 2^{n/2}$ (Div. by $2^{n/2}$)

$\Rightarrow \underline{\underline{2^{n/2} > 1}}$

GO CLASSES

Ex: $T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + T(\frac{n}{4}) + 1$

Notes

Date / /
Page No.

lower bound ($k = \log_4 n$) ↳ Right.

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

$= 3^{\log_4 n}$

$= n^{\log_4 3}$ $= T.C = \Omega(n^{\log_4 3})$

(Take most largest if inc. G.P. see rule below)

Upper bound ($k = \log_2 n$)

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

$= 3^{\log_2 n}$

$= n^{\log_2 3}$

$T.C. = O(n^{\log_2 3})$

GO CLASSES

Both are very diff. So Θ can't be given.

★ NOTE: Never solve ∞ G.P. or finite G.P. in asymptotic calc. see the below rule.

Let G.P. be

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

Case I: if $c > 1$	Case II: if $c < 1$	Case III: if $c = 1$
i.e. Increasing G.P.	i.e. Dec. G.P.	then all terms are equal.
Choose the max term	choose the min. term	
c^n	1 ↳ constant	$n \cdot 1 = n$

Notes

Ex:5 Solve $T(n) = aT\left(\frac{n}{b}\right) + cn^i$

Total levels = $\log_b n$ both sides.

Work done at each level:

Level 0 : cn^i

1 : $c\left(\frac{n}{b}\right)^i + c\left(\frac{n}{b}\right)^i + \underbrace{\cdots}_{a \text{ times}} c\left(\frac{n}{b}\right)^i = a \cdot c\left(\frac{n}{b}\right)^i$

2 : $c\left(\frac{n}{b^2}\right)^i + c\left(\frac{n}{b^2}\right)^i + \underbrace{\cdots}_{a^2 \text{ times}} c\left(\frac{n}{b^2}\right)^i = a^2 \cdot c\left(\frac{n}{b^2}\right)^i$

k : $a^k \cdot c\left(\frac{n}{b^k}\right)^i$

Sum of work done at all levels

$$= \sum_{k=0}^{\log_b n} a^k \cdot c\left(\frac{n}{b^k}\right)^i = cn^i \sum_{k=0}^{\log_b n} a^k \cdot \left(\frac{1}{b^k}\right)^i$$

and k = no. of level so = $\log_b n$

$$T.C. = cn^i \sum_{k=0}^{\log_b n} \left(\frac{a}{b^i}\right)^k$$

Notes

Date / /
Page No.

Now we have three cases:

Case I: $a < b^i$ then decreasing G.P. $\Theta(n^i)$

Case II: $a = b^i$ then equal to no. of terms $\Theta(n^i)\log_b n$

Case III: $a > b^i$ then ~~equ~~ increasing G.P. $\Theta(n^i)\left(\frac{a}{b^i}\right)^{\log_b n}$

Something similar to this is Master's Theorem.

★ Master's Theorem: Some recurrences were very common, so to avoid these calculation we have a direct theorem. It solves rec. of following form:

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is asym. positive (inc.) function.

There are three cases:

Case I: if $f(n) = O(n^{\log_b a - \epsilon})$, for some constant $\epsilon > 0$, (i.e. This is greater)

then

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

Case II: if $f(n) = \Omega(n^{\log_b a + \epsilon})$, for some constant $\epsilon > 0$, (i.e. This is greater)

then,

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

Case III: if $f(n) = \Theta(n^{\log_b a})$, (i.e. both equal)

then,

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

Teacher's Signature

Notes

→ Do not care about ϵ, it is just to say that comparisons must give polynomial greater or smaller result. The term $f(n)$ or $n^{\log_b a}$, which is polynomially greater, choose that one and if both are equal then do $n^{\log_b a} * \log n$.

→ Polynomial Greater:

1) $n^2 < n^3$

$1 < n$ So polynomial greater

2) $n^2 < n^2 \log n$

$1 < \log n$ Greater, but not polynomially. So can't be solved by master's theorem.

More examaples coming with the questions.

→ We'll first learn to apply this master's theorem and then will see the proof.

Ex:1 $T(n) = 9T\left(\frac{n}{3}\right) + n$

→ $f(n) = n$, $a = 9$ & $b = 3$

$n^{\log_3 9} = n^2$

$n^2 \underset{poly.}{>} n$

n^2 is polynomially greater than n.

So. T.C. = $\Theta(n^2)$

GO CLASSES

Notes

Date / /
Page No.

Ex:2 $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$\Rightarrow n^{\log_4 3}$? $n\log n$ let $\log_4 3 \simeq 0.7$

$n^{0.7}$? $n\log n$

1 ? $n^{0.3}\log n$

So, $n\log n$ is poly. greater than $n^{\log_4 3}$

So $T.C = \underline{\Theta(n\log n)}$

GO CLASSES

Ex:3 $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$

$\Rightarrow n^{\log_2 2}$? $n\log n$

n ? $n\log n$

1 ? $\log n$

No one is polynomially greater, so, Master's Theorem can't be applied (will study more adv. version of master's theorem for solving such recurances).

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

$\Rightarrow n^{\log_2 8}$? n^2

n^3 ? n^2

$n > 1$

So n^3 is poly. greater than n^2

$T.C = \underline{\Theta(n^3)}$

Teacher's Signature

$n^{\epsilon} > (\log n)^{\beta}$

where ϵ is very small & β is very big

Date / /

Page No.

Notes

Ex:5 $T(n) = T\left(\frac{n}{2}\right) + \sqrt{n}$

$n^{\log_2 2}$? $\sqrt{n}$

1 ? $\sqrt{n}$

$1 < \sqrt{n}$

$\therefore$ T.C = $\Theta(\sqrt{n})$

Ex:6 $T(n) = 4T\left(\frac{n}{2}\right) + n^3 \log n$

$n^{\log_2 4}$? $n^3 \log n$

n^2 ? $n^3 \log n$

$1 < n \log n$

So T.C = $\Theta(n^3 \log n)$

Ex:7 $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$

$n^{\log_2 4}$? $n^2 \log n$

n^2 ? $n^2 \log n$

1 ? $\log n$

Can't Apply Master's Th.

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

$n^{\log_2 4}$? n^2

$n^2 = n^2$

So, T.C = $\Theta(n^2 \log n)$

Because Equal.

Que: Polynomially Greater?

a) $n^2 > n \log n$

$\Rightarrow n > \log n$ ✓

b) $n^3 \log n > n^2$

$n \log n > 1$ ✓

c) $n^2 \log n$? n^2

$\Rightarrow \log n$? n ✗

(Not greater)

d) $2^n > n$

$\Rightarrow$ bcz $n^2 > n$

and $2^n > n^2$

So, by transitivity

$2^n > n$ ✓

e) $n! > n^3$

$\Rightarrow n! > n^4 > n^3$

So, by transitivity

$n! > n^3$ ✓

f) $2^n > 1$

$\Rightarrow 2^n > n > 1$ ✓

GO CLASSES

Notes

Date / /
Page No.

Ex: 9 $T(n) = 2T\left(\frac{n}{3}\right) + (\log n)^2$

$n^{\log_3 2} > (\log n)^2$ (bcz. $(\log n)^{\beta} < n^{\epsilon}$, $\epsilon \to 0$, $\beta \to \infty$)

$n^{0.6} > (\log n)^2$

$\therefore$ T.C $= \Theta(n^{\log_3 2})$

→ Now as we know to apply master's theorem. Let's prove this theorem.

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

$\frac{n}{b^k} = 1 \Rightarrow b^k = n \Rightarrow \underline{k = \log_b n}$

Notes

Date / /
Page No.

Total work done =

$$f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right) + \cdots a^k f\left(\frac{n}{b^k}\right)$$

where $k = \log_b n$

See the last term:

$$a^k f\left(\frac{n}{b^k}\right) \Rightarrow a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right)$$

$$= n^{\log_b a} f\left(\frac{n}{n}\right) \qquad [\, n^{\log_n y} = y^{\log_n z} \,]$$

$$= \underbrace{n^{\log_b a}}_{\text{last term}}$$

Total = $f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots n^{\log_b a} \cdot f(1)$

ignore this bcz. some detail is hidden

Now this is G.P., it will be

Case I. Dec. G.P is $f(n) \underset{\text{Asy}}{\overset{\text{Polynomially}}{>}} n^{\log_b a}$

then for dec. G.P. we take $f(n)$

Case II: Inc. G.P. is $f(n) \underset{\text{Asy}}{\overset{\text{Polynomially}}{<}} n^{\log_b a}$

then we take $n^{\log_b a}$ in inc. G.P.

Case III: if $f(n) = n^{\log_b a}$ (mul. with total terms $(= \log_b n)$)

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

GO CLASSES

Notes

Date / /
Page No.

Hence, we have proved the master's theorem. And we can now directly apply on them just take care of polynomially greaterness.

☆ **Generalised Master's Theorem:** We have enhanced the master's theorem for the following situation

$$f(n) \ ? \ f(n)(\log n)^k$$

if we encounter such situation, we couldn't directly apply master's theorem, bcz none is polynomially greater.

By, Gen. Master's theorem in this situation, we can go with $f(n)(\log n)^n$ term by mul. with $\log n$ bcz. of that equality case.

So,

if $f(n) = \Theta(n^{\log_b a} \cdot (\log n)^k)$, $k > 0$

then

$$T.C. = \Theta(n^{\log_b a} \cdot (\log n)^{k+1})$$

Everything else is same.

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

$n^{\log_2 2} \ ? \ n\log n$

↳ Not pol. greater.

Gen. Case:

$T.C = \Theta(n(\log n)^2)$

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

$n^{\log_2 n} \ ? \ n$

Not. Gen. Case.

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

GO CLASSES

Teacher's Signature

Notes

Ex:3 $T(n) = 2T\left(\frac{n}{2}\right) + n(\log n)^{10}$

$\Rightarrow n^{\log_2 2} \;?\; n(\log n)^{10}$

$\Rightarrow$ T.C = $\Theta(n(\log n)^{11})$

Ex:4 $T(n) = 9T\left(\frac{n}{3}\right) + n^2(\log n)^k$

$\Rightarrow n^{\log_3 9} \;?\; n^2(\log n)^k$

$\Rightarrow n^2 \;?\; n^2(\log n)^k$

$\Rightarrow$ T.C = $\Theta(n^2(\log n)^{k+1})$

Ex:5 $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$, $T(2)=1$ $\therefore$ Can't solve by Gen. Master's Theorem

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

Total work:

$$\frac{n}{\log n} + \frac{n}{\log \frac{n}{2}} + \frac{n}{\log \frac{n}{2^2}} + \cdots \frac{n}{\log \frac{n}{2^k}}$$

$$= n \cdot \sum_{i=0}^{\log n - 1} \frac{1}{\log \frac{n}{2^i}} \Rightarrow n \cdot \sum_{i=0}^{\log n - 1} \frac{1}{\log n - \log 2^i}$$

$$\Rightarrow n \cdot \sum_{i=1}^{\log n} \frac{1}{i} = \underline{n \cdot \log\log n}$$

$\therefore \left[\sum_{i=1}^{n} \frac{1}{i} = \log n\right]$ → Remember.

So. $\underline{\underline{T.C = \Theta(n \log\log n)}}$

GO CLASSES

Notes

Date / /
Page No.

Ex:6 $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{(\log n)^2}$, $T(2)=1$

Master's Theorem can't be applied. Solving manually we get:

$$T(n) = n \cdot \sum_{i=1}^{\log n} \frac{1}{i^2}$$

$\left[\because \sum_{i=1}^{\infty} 1/i^2 = \text{constant}\right]$ also if i^3 or i^4 ...

$$= n$$

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

☆ **Extended Master's Theorem:** This is Just Gen. Masters Theorem + Ex:5 + Ex:6.

Writing this for the sake of completion.

The master's theorem applies to recurrance of following form:

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

where $a \geq 1$ & $b > 1$ are constants and $f(n)$ is an asymp. positive function. There are three cases:

1: if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then
$T(n) = \Theta(n^{\log_b a})$

2: if $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, then
$T(n) = \Theta(f(n))$

∴ Same till now.

3: It can further be divided in three cases:

Notes

Date / /
Page No.

3.1 if $f(n) = \Theta(n^{\log_b a} \log^k n)$, with $k \geq 0$, then

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$$

3.2 if $f(n) = \Theta\left(\frac{n^{\log_b a}}{\log n}\right)$, then

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

3.3 if $f(n) = \Theta\left(\frac{n^{\log_b a}}{(\log n)^p}\right)$, with $p \geq 2$, then.

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

That's all.

GO CLASSES

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

$n^{\log_2 4}$? $\frac{n^2}{(\log n)^2}$

By 3.3 T.C.: $\Theta(n^2)$, $p=2$

Ex:2 $T(n) = 9T\left(\frac{n}{3}\right) + (n \log n)^2$

$n^{\log_3 9}$? $n^2 (\log n)^2$

n^2 ? $n^2 (\log n)^2$

By 3.1 T.C. $\Theta(n^2 (\log n)^3)$

Notes

Date / /
Page No.

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

$n^{\log_2 4} \; ? \; \frac{n}{(\log n)^2}$

$n^2 \underset{\text{Poly.}}{>} \frac{n}{(\log n)^2}$

So $\Theta(n^2)$ by **1**

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

$n^{\log_2 1} \; ? \; 1$

$1 \; ? \; 1$

$\Theta(\log n)$ by **3:1**

★ Some problems that even master's theorem cannot solve are:

Ex:1 $T(n) = \sqrt{n}\, T\left(\frac{n}{2}\right) + n$

bcz, $a = \sqrt{n}$ is not a constant

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

bcz $b = \log n$ is not a constant

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

bcz $a = \frac{1}{2}$ is not ≥ 1

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No

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

bcz. $b = \frac{3}{4}$ is not $b > 1$

Ex:5 $T(n) = 3T\left(\frac{n}{2}\right) - n$

bcz. $f(n) = -n$ is not positive.

Ex:6 $T(n) = 2T\left(\frac{n}{2}\right) + n^2 \sin n$

bcz. violates regularity condition.

Ex:7 $T(n) = T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + n$

bcz a, b are not fixed.

GO CLASSES

Notes

Date / /
Page No.

☆ Change of Variable:

Not Even a little trickier to understand.

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

Step 1: Assume $n = 2^m$ (Always assume this), So, $m = \log_2 n$

$$T(2^m) = T((2^m)^{1/2}) + \log_2(2^m)$$

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

Step 2: $T(2^m) = S(m)$

then $S(m/2) = T(2^{m/2})$ (critical to understand)

So,

$$S(m) = S\left(\frac{m}{2}\right) + m$$

Using master's theorem $\Theta(m)$

So,

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

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

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

→ Req. form

Ans

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No

Remember this only

Here is just log of this

If $T(2^m) = S(m)$

then $T(n) = S(\log_2 n)$

because $2^m = n$ and $m = \log_2 n$

(No confusion, Put $m = \log n$)

Similarly:

$$T(2^{(m/2)}) = S\left(\frac{m}{2}\right)$$

$$T(2^{(m^k)}) = S(m^k)$$

$$T((2^m)^k) = T(2^{mk}) = S(mk)$$

$$T\left(\left(\frac{2^m}{k}\right)\right) = S\left(\log\left(\frac{2^m}{k}\right)\right) = S(m - \log k)$$

$\boxed{\text{Remember just that if } T(n) = S(y) \text{ then } y \text{ is } \log n}$

Que: $T(n) = T(\sqrt{n}) + \sqrt{n}$

Step 1: $n = 2^m$

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

$$S(m) = T(2^m)$$

$$S(m) = S\left(\frac{m}{2}\right) + 2^{m/2}$$

By master's theorem

$$S(m) = \Theta(2^{m/2})$$

$$T(2^m) = S(m) = \Theta(2^{m/2})$$

$$T(n) = \Theta(2^{m/2})$$

$$T(n) = \Theta(n^{1/2})$$

$$T(n) = \Theta(\sqrt{n})$$

$T(n) = \Theta(2^{\frac{\log n}{2}})$: $\log T(n) = \frac{\log n}{2}$

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

Can do $m = \log n$) OR

$n = 2^m$

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

Teacher's Signature

Notes

Date / /
Page No.

Ex:3 $T(n) = 2T(\sqrt{n}) + \log n$

$\Rightarrow$ $n = 2^m$

$T(2^m) = 2T(2^{m/2}) + \log_2 2^m$

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

$S(m) = 2S\left(\frac{m}{2}\right) + m$

By master theorem $S(m) = \Theta(m \log m)$

as $m = \log n$

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

GO CLASSES

Ex:4 $T(n) = T(\sqrt{n}) + \log\log n$

$\Rightarrow$

$n = 2^m$

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

$S(m) = S\left(\frac{m}{2}\right) + \log m$

by master's theorem

$S(m) = \Theta((\log m)^2)$

$m = \log n$

$T(2^m) = \Theta((\log m)^2)$

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

Date / /
Page No

DSY Notes

Ex:5 $T(n) = 3T(\sqrt{n}) + 1$ $\Theta((\log_2 n)^{\log_2 3})$

Ex:6 $T(n) = T(\sqrt{n}) + n^2$ $\Theta(n^2)$

Ex:7 $T(n) = 2T(\sqrt{n}) + (\log n)^3$ $\Theta((\log n)^3)$

Ex:8 $T(n) = 4T(\sqrt{n}) + (\log n)^2$ $\Theta((\log n)^2 \cdot \log\log n)$

Ex:9 $T(n) = 12T(n^{1/3}) + (\log n)^2$ $\Theta((\log n)^2)$

Ex:10 GATE CSE 2006

$T(n) = 2T(\sqrt{n}) + 1$ $T(n) = \Theta(\log n)$

$T(1) = 1$

Ex:11 $T(n) = 161^2\, T(n^{1/161}) + 161 \cdot (\log n)^2$ $\Theta((\log n)^2 \log\log n)$

Ex:12 $T(n) = \left(T\left(\frac{n}{161}\right)\right)^{161} \cdot n$

$\Rightarrow$ Take log both sides:

$\log(T(n)) = \log\left(T\left(\frac{n}{161}\right)^{161} \cdot n\right)$

$G(n) = \log(T(n)) = 161 \cdot \log\left(T\left(\frac{n}{161}\right)\right) + \log n$

$G(n) = 161 \cdot G\left(\frac{n}{161}\right) + \log n$ $\because$ bcz $G\left(\frac{n}{161}\right) = \log\left(T\left(\frac{n}{161}\right)\right)$

By master's theorem

$G(n) = \Theta(n)$

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

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

GO CLASSES

Notes

Ex:13 $T(n) = \sqrt{n}\, T(\sqrt{n}) + 100n$

⇒ Divide by n both sides

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 100$$

$G(n) = \frac{T(n)}{n}$, then $G(\sqrt{n}) = \frac{T(\sqrt{n})}{\sqrt{n}}$

$$G(n) = G(\sqrt{n}) + 100$$

now,

$n = 2^m$

$$G(2^m) = G(2^{m/2}) + 100$$

$$S(m) = S\left(\frac{m}{2}\right) + 100$$

$S(m) = \Theta(\log m)$ [by master's theorem]

$$G(2^m) = \Theta(\log m)$$

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

$$\frac{T(n)}{n} = \Theta(\log\log n)$$

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

Ex:14 GATE IT 2008 $T(n) = \sqrt{2}\, T\left(\frac{n}{2}\right) + \sqrt{n}$

GO CLASSES

Part 2 | Module 2: Divide & Conquer Algorithms.

(D&C)

Notes

It is an algorithm design paradigm, where we break a problem into smaller subproblems, solve those subproblems recursively, and then combine the results of those subproblems to get the overall answer.

GO CLASSES

Step 1: Divide into parts. $(n_1, n_2, \ldots, n_k)$

Step 2: Solve each part recursively.

Step 3: Combine part solutions to get original solution

Work is mostly done in either step 1, 2 or 3.

$$\boxed{T(n) = \text{Divide Cost} + T(n_1) + T(n_2) + \ldots + T(n_k) + \text{Combine Cost}}$$

Some algo have high divide cost while some have high combine cost, while some may have both constant or anything. In quick sort, combine cost is $\Theta(1)$ and divide cost is $\Theta(n)$ & in merge sort, " " " $\Theta(n)$ " " " " $\Theta(1)$.

Notes

Date / /
Page No.

Three rules while applying recursion are:

a. always have atleast one case that can be solved w/o using recursion.

b. any recursive call must progress towards a base case.

*c. always assume that the recursive call works, & use this assumption to design your algorithms.

☆ Maximum of an array: Our goal is to input an array and then return its maximum.

w/o Dnc:

```
max = a[0]
for (i = 1 → n-1)
    if (a[i] > max) max = a[i];
```

with Dnc:

a[0 - - - n-1]

a[0, n/2] — Get its max1

a[n/2+1 - - n-1] — Get its max2

max = max(max1, max2)

```
max(a, l, r){
    if (l==r)  return a[l];
    m = (l+r)/2;
    u = max(a, l, m);
    v = max(a, m+1, r);
    return maximum(u, v);
}
```

Some Inbuilt function giving max. of two nos

GO CLASSES

Teacher's Signature

Notes

⇒ Recursively solving problem is somewhat analogus to proving by induction. Both have base case, both assumes (Rec. that solution to subproblem is correct ≈ Induction that hypothesis) and finally both concludes that Big problem can also be solved.

⇒ Time complexity for finding maximum of an array using D&C.

Divide Cost = $O(1)$, Comb. cost = $O(1)$

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

→ because dividing array in two halves

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

By master's theorem.

$\boxed{T(n) = \Theta(n)}$ ✓

This time complexity can also be calculated using the tree method.

Que: Which of the following are correctly implemented to sum array elements using D&C. (MSQ)

⇒ Not writing but all four were correct.

GO CLASSES

Notes

Date / /
Page No.

Ques: Give a function that returns true if value in an array.

⇒ It is asking for searching algo. will do this by divide and conquer instead of linear:

```
search (arr, i, j, val)
{
    if (i == j)    return arr[i] == value ;
    mid = (i+j)/2 ;
    return search (arr, i, mid, val) OR search(arr, mid+1, j, val);
}
```

Ques: Check whether given function sorts array correctly?

```
dumbsort (a, i, j)
{
    if (i==j)  return;
    dumbsort (a, i+1, j);
    //below code is just putting a[i] at correct place
    for (k=i+1 to j) {
        if (a[k-1] > a[k]) {
            swap (a[k-1], a[k])
        }
    }
}
dumbsort (a, 0, n-1);
```

⇒ True. It is dividing array in 1 & n-1 and then combining it by fixing that 1 element.

GO CLASSES

Notes

Date / /
Page No.

Time Complexity of dumbsort:

$T(n) = T(n-1) + n$
↳ Combine Cost (that for loop)

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

★ **Merge Sort:** It is one of the purest application of Divide & Conquer methadology.

It has three simple steps:

→ Divide array into two halves.
→ Recursively sort each half.
→ Merge both halves to make whole array sorted.
↳ Merge subroutine

7	5	2	4	1	6	3	0

→ unsorted array

Step 1. | 7 | 5 | 2 | 4 | and | 1 | 6 | 3 | 0 |

Step 2: Somehow we sorted both halves

| 2 | 4 | 5 | 7 | and | 0 | 1 | 3 | 6 |

Step 3: merge them

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

↳ Array is sorted now

GO CLASSES

Notes

Date / /
Page No.

```
mergesort (a, i, j) {
    if (i==j) return;
    mid = (i+j)/2;
    mergesort (a, i, mid);
    mergesort (a, mid+1, j);

    merge (a, i, mid, j);
}
```

} → Divide

} → Recursively solve

} → Merge both half arrays

→ This merge method is the heart of the merge sort.

Que: If we want to design mergesort such that it works in $\Theta(n \log n)$ time, how much time "merge" should take.

⇒ $T(n) = 2T(\frac{n}{2}) + \text{something}$

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

By master's theorem to make it $n \log n$, something needs to be $O(n)$

So, merge can take upto $O(n)$ time only.

⇒ Merge Algorithm: Heart of mergesort. In this we are given two sorted arrays of size m & n. Our goal is to return a single sorted array of size m+n.

GO CLASSES

Notes

Date / /
Page No

Attempt 1: Append one array to other and then sort that big array using some sorting algo. But sorting would take $[(m+k)\cdot\log(m+k)]$ time and we are bound to have $O(m+k)$ time only to make mergesort work in nlogn time.

* Merge Procedure

Attempt 2: Maintain two pointers at both array, compare these and add smaller to new array updating the respective pointer. If one of the array becomes empty copy the 2nd array as it is now.

~~10~~ ~~15~~ ~~22~~ ~~80~~ ~~5~~ ~~8~~ ~~11~~ ~~70~~

Arr1 (Sorted) Arr2 (Sorted)

5 8 10 11 15 22 70 80.

Final Sorted Array.

Time = $\Theta(m+k)$ here as only $m+k-1$ comparisons.

In worst case we may need to do $m+k-1$ comparisons i.e. till we do not reach till the end of both arrays and only one element is left to copy.

Best case can be achieved if we are able to copy whole subarray w/o any more comparisons. In that case $\min(m,k)$ comparisons need to be done. (Already sorted array gives this best case).

Notes

Date / /
Page No.

Ex: 1 2 3 7 8 9 11 12

Arr 1 Arr 2

1 2 3 7 8 9 11 12

→ Entire Arr 2 is copied w/o comp.

NOTE Merge algo will not return sorted array if the input arrays are not sorted.

```
merge (a, i, mid, j)
{
    p=i , q= mid+1 , k=0 ;
    while (p<= mid && q<=j)
    {
        if (a[p] <= a[q])
        {
            b[k] = a[p] ;
            p++;
        }
        else
        {
            b[k]= a[q];
            q++;
        }
        k++;
    }
    if (p>mid)  copy rem. of 2nd half to b ;
    else if (q>j)  copy rem. of 1st half to b ;
    copy b to a;
}
```

GO CLASSES

Notes

Date / /
Page No

Here b is an auxilary array used to merge.

⇒ No. of comparisons in merge :

Worst Case : $m+k-1$

Best Case : $\min(m,k)$

→ Now we know merge procedure, let's get back to merge sort.

a = [85, 24, 63, 45, 17, 31, 96, 50]

This is how merge sort is working

You already know the seq. of method call :)

	~~(a,0,0)~~, ~~(a,1,1)~~, ~~merge~~
~~(a,0,0)~~	~~(a,1,1)~~, ~~merge~~, ~~a(2,2)~~, ~~(a,3,3)~~, ~~merge~~, ~~(a,4,4)~~ ~~(a,5,5)~~, ~~merge~~,
~~(a,0,1)~~	~~(a,2,3)~~, ~~merge~~, ~~(a,4,5)~~, ~~(a,6,7)~~, ~~merge~~
~~(a,0,3)~~	~~(a,4,7)~~, ~~merge~~
~~(a,0,7)~~	

Notes

for visuals:

Que: Consider the following new variant of mergesort: instead of splitting the list into two parts, we split it into three thirds. Then we recursively sort each third and merge them.

a) Pseudo code for this mergesort:

→ We will use older merge code.

```
mergesort (a, i, j)
{
    if i >= j return;
    k = (i+j)/3 ;   m = 2(i+j)/3 ;
    mergesort (a, i, k) ;
    mergesort (a, k+1, m) ;
    mergesort (a, m+1, j) ;
}
```

Notes

Date / /
Page No.

merge(a, i, k, m);
merge(a, i, m+1, j);
}

b) Total no. of comparisons?

⇒ In worst case,

merge(a, i, k, m); $= \frac{n}{3} + \frac{n}{3} - 1 = \frac{2n}{3} - 1$

merge(a, i, m+1, j) $= \frac{2n}{3} + \frac{n}{3} - 1 = \frac{3n}{3} - 1$

Total $= \frac{5n}{3} - 2$

$= O(n)$

c). Rec. Relation

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

d). Solve above rec. relation:

⇒ By master's theorem

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

Ques: Suppose your smart friend gives you a technique for merging two sorted array into one sorted array in constant time. Now analyze T.C.

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

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

GO CLASSES

Notes

Date / /
Page No.

Qu: T.C. if we divide in ~~two~~ 2 parts instead of dividing in halves. of size 1 & n-1

$\Rightarrow$ $T(n) = T(n-1) + T(1) + \Theta(n)$

$T(n) = \Theta(n^2)$ } Dumb sort

→ We can implement that 3 way partition merge sort using new merge method where we can ~~ea~~ maintain three pointers. Time for this merge will also be $\Theta(n)$.

Qu: Consider an implementation of mergesort that divides the array into k, approximately equal parts instead of usual two equal parts. Worst case running time of this implementation.

$\Rightarrow$

work
n

$\frac{n}{k} * k = n$

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

Note, that we can merge k parts in $\Theta(n \cdot k)$ time.
↳ will see later

$T(n) = kT(\frac{n}{k}) + \Theta(n)$

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

Teacher's Signature _____

Notes

Date / /
Page No

Qu: The classical implementation of mergesort is considered non-adaptive, since the running time is $\Theta(n\log n)$ & do not depend on input array. It can be made adaptive by checking in the merge step whether the largest ~~step~~ element of the first subarray to be merged is smaller than the smallest element of the second subarray. What is the new best case & when this best case is achieved?

→ It is asking us to compare x & y, and if $y > x$ then directly merge. Worst case time will remain same. But, best case will improve.

In best case : $\Theta(1)$
worst case : $\Theta(n)$ } Merge.

Best case

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

worst case

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

Previously $n\log n$ in all cases.

GO CLASSES

Notes

Date / /
Page No.

Que: Consider a version of mergesort in which array is divided in two pieces of random size. What is worst case.

→ Worst case is when always (not once) divided in 1 & n-1 size.

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

$= O(n^2)$

Que: An array A[1...n] is bitonic if there exist a t such that A[1...t] is increasing and A[t+1...n] is decreasing. Give an efficient algo to sort bitonic array A.

→ Ex. of A

1 20 30 40 35 25 19

Step 0: Find t — $O(n)$

Step 1: Reverse A[t+1...n] to A[n...t+1] — $O(n)$

Step 2: Merge A[n...t+1] and A[1...t] — $O(n)$

Total = $O(n) + O(n) + O(n)$

$= O(n)$

Que: An inter Array A is k-even-mixed if there are exactly k even integers in A and the odd integers in A appear in sorted order.

GO CLASSES

Notes

Given a k-even-mixed array A containing n distinct integers for $k = \frac{n}{\log n}$. Tell $O(n)$ algo to sort A.

⇒ Scan through A and put all even integers in order into an array A_E and all odd in another array A_O where $|A_E| = k$ & $|A_O| = n-k$.

A_O is sorted by def. and A_E can be sorted in $O(k \log k)$ times

Given $k = \frac{n}{\log n}$

$$O\left(\frac{n}{\log n} * \frac{\log n}{\log n}\right)$$

$= O(n)$

and now both can be merged (A_O & A_E) in $O(n)$ time.

Total Time $= O(n) + O(n)$

$= O(n)$

→ No. of comparisons if, everywhere is worst case.

$$C(n) = 2C\left(\frac{n}{2}\right) + \underbrace{n-1}_{\text{Merge}}$$

→ No. of comparisons if everywhere is best case.

$$C(n) = 2C\left(\frac{n}{2}\right) + \min\left(\frac{n}{2}, \frac{n}{2}\right)$$

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

GO CLASSES

Notes

★ Merging K sorted arrays into 1 sorted array:

Goal: I/P : K sorted arrays of size $\frac{n}{K}$

O/P : 1 sorte array of size n

There are five diff methods for this!

⇒ Method 1: (Trivial) Append all of the K arrays to form a large array of size 'n' and then apply a sorting algo on that arry.

T.C. = $O(n \log n)$

GO CLASSES

⇒ Method 2: (Successive Merge) In this merge 2 arrays then their merge with 3rd, then their with 4th & so on.

Total work done = $\frac{2n}{k} + \frac{3n}{k} + \cdots + \frac{kn}{k}$

Notes

$= \frac{n}{k} [2+3+\dots+k]$

T.C. = $\Theta(nk)$

⇒ Method 3: (k pointers) In this we maintain k diff pointers & our old approach i.e. find min of all k pointers & update that pointed and add that in new array.

Here comparisons = k for each element.

Total T.C = $\Theta(nk)$

k because to find min. among k elements, we need k comparisons. (k-1 to be precise)

⇒ Method 4: (Using Min Heap) In this build a heap for these k elements. (O(k))

For each insertion, remove min and then update the respective new pointer in the min heap. O(log n) + O(log n)

Repeat this step for all elements.

Notes

(Its actual implementation is not required).

Now remove min and add new.

$T.C = O(k) + O(n*(\log k + \log k))$

initial heap build

Repeat for all n elements

Remove min

Add new key.

$T.C = O(n \log k)$

⇒ Method 5 (Pairwise Merging): Apply merge algo to each pair of array.

Notes

Date / /
Page No

T.C = $n \log k$

because $\log k$ levels and work done at each level is n only.

Ques: If we have $\log n$ sorted list of equal size. Total no. of elements are n. How much min time?

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

$k = \log n$ i.e. no. of list

T.C = $n \log \log n$

T.C = $n * \log k$

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

$= n * (\log n - \log \log n)$

$= n \log n$

GO CLASSES

Notes

Date / /
Page No.

★ Iterative Merge Sort | Straight Two way Merge Sort | Bottom Up :

Merge Subroutine (or Procedure) is common in both recursive & Iterative merge sort.

Here we just need to merge by 1, 2, 4, 8 ...

Take no's in above diagram & apply merge procedure. This is bottom up approach.

Gate CS 99 Que: Use straight two way merge sort algo on :

20, 47, 15, 8, 9, 4, 40, 30, 12, 17

then order of these elements after 2nd pass of algo is :

⇒ 20, 47 15, 8 9, 4 40 30 12 17

1st Pass 20 47 8 15 4 9 30 40 12 17

2nd Pass : 8 15 20 47 4 9 30 40 12 17

Ans

Notes

Date / /
Page No.

Time Complexity: No. of levels * work per level

$\downarrow$ $\log_2 n$ $\qquad$ $\downarrow$ n ↳ as merge takes $(m+k)$

$= n\log n$

↳ same as recursive obviously

So,

Top Down = Split + Combine
$O(1)$ $\qquad$ $O(n\log n)$

Bottom up = Only Combine
$O(n\log n)$

So, both version are same asymptotically.

→ By default merge sort means top down merge sort.

→ Implementation Details (Not for Gate)

```
for (width = 1 ; width < a.length ; width *= 2) {
    int i=0;
    for (i=0; i < a.length ; i=i+(2*width)) {
        int l, m r;
        l=i;
        m= i+width;
        r= i+(2*width)-1;
        merge (a, l, m, r);
    }
}
```

GO CLASSES

Teacher's Signature _____

Notes

Date / /
Page No.

★ Properties of Sorting Algorithm:

⇒ **Stable Sorting:** A sorting algorithm is stable if elements with the same key appear in the output array in the same order as they do in the ~~same~~ input array.

i/p: 3 4_a 5 6 2 4_b 9 8

o/p: 2 3 4_a 4_b 5 6 8 9 Stable

OR

2 3 4_b 4_a 5 6 8 9 Not Stable.

However both of the outputs are sorted

Stability is important because of many application & not writing them.

⇒ **In place Algorithm:** The idea of an in-place algorithm is not unique to sorting, but sorting is most well known case.

A sorting algorithm is said to be an in-place sorting algorithm if the amount of extra space required by the algorithm is $\Theta(1)$. i.e. extra space is bounded by a constant, independent of the size of the array.

(Ignore the $\log n$ extra recursion stack space, Consider do we need another array to copy like in merge sort).

GO CLASSES

Notes

Date / /
Page No.

→ Reversing an array can be inplace or non-inplace algo. Not be inplace if we use strategy like copying reverse in another array and then copying that another in original array.

Inplace Reverse can be implemented by pairwise swap of elements of array. first with last, second with 2nd last & so on.

→ For ~~w~~ recursion stack is not imp., we will take care if we need an extra ~~stack~~ array or not.

→ Merge Sort is <u>not</u> an inplace algo because merge procedure uses auxiliary array & this cannot be avoided efficiently.

→ Merge Sort is stable algorithm because while implementing we can always take the left element.

if (a[p] <= a[q]) → This = is making it stable.

b[k] = a[p];

GO CLASSES

Notes

Date / /
Page No.

★ Maximum element of an array

⇒ **Naive Approach:** (Skewed tournament)

```
max = a[0]
for (i=1 to n-1)
    if (a[i] > max)   max = a[i];
```

No. of comparisons = $n-1$

So, it's a skewed tree

⇒ **Tournament Method:** Finding max & min & 2nd max are same what analogy to finding best player or worst player (loser) in tournament?

In this method of finding max. element compare pairwise

GO CLASSES

Teacher's Signature _____

Notes

At last level we have $\frac{n}{2}$ pairs and for each pair we do one comparison.

Total Comparisons = $\frac{n}{2} + \frac{n}{2^2} + \cdots \frac{n}{2^k}$ $(2^k = n)$

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

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

$$= n-1$$

→ So, in both of the approaches, we need to do $n-1$ comparisons.

★ Maximum & Second Maximum of an array:

Method 1:

```
if (a[0] > a[1])
{   max = a[0];
    smax = a[1];
}
else
{   max = a[1];
    smax = a[0];
}
for (i=2 to n-1) {
    if (a[i] > max) {
        smax = max;
        max = a[i];
    }
```

Notes

```
else
else if (a[i] > smax) {
    smax = a[i];
}
}
```

Total comparisons :

→ Best case : when you never go to else if part.

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

Init.

→ Worst case : when you always go to else part

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

→ Avg. case : Half time go to else half time not

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

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

NOTE : To find 2nd max we always need to know maximum also. For n^{th} max we know n greaters, however seq. not req. which is max greatest or all.

GO CLASSES

Notes

Date / /
Page No.

Method 2 : One thing that can be observed from tournament method is that 2nd largest can only be defeated by largest element i.e. 2nd max is some sibling of maximum.

Ex:

Siblings of 25 are 10, 20, and 15, but 25 defeated them only.

Total no. of siblings :- $\log_2 n$

Now, somehow (implementation may be by having list not required) we can keep track of siblings and then find 2nd max. among those sibling by just finding max of siblings list.

Total Comparisons: $n-1$ for 1st max.

$\log n - 1$ for 2nd max. (among $\log n$ sib)

Total $= n-1 + \log n - 1$

$= n + \log n - 2$

GO CLASSES

Teacher's Signature _____

Notes

Date / /
Page No.

This siblings thing could also be done in skewed tournament, but there n-1 siblings can be there making it inefficient.

Balanced tournament can be viewed as divide & conquer merge sort where on merge we return greater element.

☆ Maximum & Minimum from given array:

Method 1: (Naive)

```
max = min = a[1]
for (i = 2 → n) {
    if (a[i] > max)    max = a[i]
    if (a[i] < min)    min = a[i]
}
```

Comparisons: $(n-1) + (n-1) \Rightarrow 2n-2$ → In each case.

Method 1.1: Can be improved for best case by adding else if instead of if because if if min = if is true no sense in checking 2nd if

```
max = min = a[1]
for (i = 2 → n) {
    if (a[i] > max)    max = a[i]
    else if (a[i] < min)    min = a[i]
}
```

GO CLASSES

Notes

In best case : $(n-1)$ comparison

In worst case : $2(n-1) = 2n-2$ comparisons.

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

Method 2: CLRS method In this we do comparison on pair wise

i.e. we first compare a pair and then greater among pair with max & smaller with min.

```
if (a[1] > a[2])
    max = a[1] ; min = a[2];
else
    max = a[2] ; min = a[1];
for (i=3 to n by 2) {
    if (a[i] > a[i+1]) {
        if (a[i] > max)   max = a[i];
        if (a[i+1] < min)   min = a[i+1];
    }
    else {
        if (a[i] < min).   min = a[i];
        if (a[i+1] > max)   max = a[i+1];
    }
}
```

Basically pick a pair & compare only greater with max & only min with smaller.

GO CLASSES

Notes

Total Comparison: $\underbrace{1}_{\text{Initialisation}} + \underbrace{\left(\frac{n-2}{2}\right)}_{\text{Pairs}} \underbrace{3}_{\text{\# comp. per pair}}$

$$\boxed{\#\text{comparisons}_{\text{even}} = \frac{3n}{2} - 2 \quad \text{in } \underline{\text{all}} \text{ cases}}$$

If 'n' is even

If n is odd then simply during initialisation go from 2 to n instead of 3 to n and set ~~min~~ max = min = a[1]. (No initial comp)

$$\boxed{\#\text{comparisons}_{\text{odd}} = \left(\frac{n-1}{2}\right)3 = \frac{3n}{2} - 1.5}$$

This can be generalised as:

$$\boxed{\#\text{comparison} = \left\lceil \frac{3n}{2} \right\rceil - 2}$$

Que: Let P be no. of comparisons to find min & max using best algorithm:

(a) $P \le \left\lceil \frac{3n}{2} \right\rceil - 2$

(b) $P \le \left\lceil \frac{3n}{2} \right\rceil$

(c) $P \le \left\lfloor \frac{3n}{2} \right\rfloor$

(d) $P \le 10n$

Notes

Date / /
Page No.

Method 3: However same no. of comparison, but still let's see this

In tournament method, one thing that can be noticed is min. will be among those which were not selected initially.

Ex:

Now min is among not selected i.e. (10, 1, 8, 2) only bcz ij

So, in this approach:

Round 1: $\frac{n}{2}$ comparisons

Find max. among winners = $\frac{n}{2} - 1$ comparisons

Find min. among losers = $\frac{n}{2} - 1$ comparisons

$$\text{Total} = \frac{n}{2} + \frac{n}{2} + \frac{n}{2} - 1 - 1$$

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

only works for even n number

GO CLASSES

Notes

If odd then keep 1 element and perform on (n-1) elements and then by max 2 comparisons, we can select min & max from prev. min, prev. max and left out element

$\#$ comparison $= \frac{3}{2}(n-1) - \cancel{2} + \cancel{2}$

$= \frac{3n}{2} - 1.5$

odd.

$\boxed{\left\lceil \frac{3n}{2} \right\rceil - 2}$ → Odd or Even.

GO CLASSES

Method 4: Divide & Conquer method, Just divide array till reach size 1 & then merge by finding smaller & larger of two.

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

↳ 2 comp. each time

& $C(2) = 1$

On solving

$\left\lceil \frac{3n}{2} \right\rceil - 3$

Note: if n is not in power of 2 then

$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2.$

Notes

Date / /
Page No.

Que: Give $n/2$ distinct numbers, you want to find a no. that is neither the min no nor the max. Min comparison?

⇒ Just pick any 3 no. and then by 3 comparisons, we can check which is min or max or not both. So, by this 3 comparisons.

Pyqs

Que: Gate CSE 2007 (Easy), Gate CSE 2014 (Easy), Gate CSE 2021 (Easy)

* Revise Time Complexities Concept *. We'll see some more problems using DnC.

★ **Counting Inversions:** It has various applications like in Movie Recommendation System. Not necessary for us to know the applications

Inversion: Indices $i<j$ form an inversion if $a[i] > a[j]$

Our Goal:

Input: Array

Output: Total No. of inversion in the array.

Ex: 2 4 1 3 5

Pair	Inv.	Pair	Inv.
2,4	X	4,5	X
2,1	✓	1,3	X
2,3	X	1,5	X
2,5	X	3,5	X
4,1	✓		
4,3	✓		

Teacher's Signature _____

GO CLASSES

Notes

So, total = 3 inversions.

Another way to count it is

(Sorted Version)

3 intersections → 3 inversions

Que: Count inversions in following array:

3, 2, 5, 7, 6, 4, 2

⇒ 10 (3-2, 3-2, 5-4, 5-2, 7-6, 7-4, 7-2, 6-4, 6-2, 4-2)

⇒ Maximum no. of Inversions is when each pair we take is an inversion. And total no. of ~~inv.~~ unordered pairs in an array of n elements is $^nC_2$ or $\frac{n(n-1)}{2}$

So, max inv = $\frac{n(n-1)}{2}$

Maximum inversion will be when the array is sorted in descending order.

⇒ Min. no. of inversions are zero, and it is when array is in ascending order.

GO CLASSES

Date / /
Page No.

Notes

Que: Let T[0,.. n-1] be a vector of Integers. Show that if there is an inversion (i,j) in T, then T has at least j-i inversions.

⇒ Let. T =

for (i,j) to be inversion $x > y$.

Now, any element b/w i & j, (say at k), is z.

Now we have two cases: (Exhaustive)

Case 1: $z > x$

then $z > y$ due to transitivity.

now obviously (k,j) is an inversion.

Case 2: $z < x$

then (i,k) is an inversion.

So, in every case elements b/w i & j i.e j-i elements will give inversion.

∴ Hence Proved

Bruteforce Algo to Count Inversion: Pick every possible pair and then check inversion

```
for (i=0 to n-1)
    for (j=i+1 to n-1)
        if (a[i] > a[j])
            inversion++;
```

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

GO CLASSES

Notes

Date / /
Page No.

Time Complexity = $\Theta(n^2)$

⇒ **Using Divide & Conquer:** (Not Good nor required)

Divide in two parts and then after merging count no. of inv. in it ~~of~~ from left + from right. because earlier they can be less but after merge, we may see new inversions. So count after merging. That will always be equal to $\Theta(n^2)$

$$T(n) = \underbrace{2T(n/2) + O(1)}_{\text{Division cost}} + \underbrace{O(n^2)}_{\text{Merge cost}}$$

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

So, no improvement from Bruteforce. Infact no sense of division if we are counting after merging the old way.

* One key observation that can be made is that if we count inversion while merging two sorted array, we can do in $\Theta(n)$ time instead of $\Theta(n^2)$ time.

Ex: ~~3~~ ~~7~~ ~~10~~ ~~14~~ 18 19 (Arr1) ~~2~~ ~~11~~ ~~16~~ ~~17~~ 23 25 (Arr 2)

Now merge arr1 & arr2. If element from ~~left~~ Arr1 is smaller then good, keep moving, but if element from Arr2 is smaller then it's an inversion and obviously it will also be smaller than other (uncompared) elements of Arr1, so, we don't need to compare them with all, just update inversion with # of uncompared elements of Arr1.

#inv = 6 + 3 + 2 + 2 = 13
from 2, 11, 16, 17

2	3	7	10	11	14	16	17	18	19	23	25

Merged Array

GO CLASSES

Notes

So we have 13 inversions here in total in that array.
So what we are doing is

I/P { Sorted Arr 1, Sorted Arr2 } —Algo Merge→ { Sorted Merged Arr + #inversions } O/P

↳ In O(n) time only

This can be implemented in our old algo of merge procedure from merge sort by just adding one more line.

⇒ Counting Inversions using DnC using Merge subrouting:

Now we need to keep track of Inv. from left part, and right part also (i.e. Inversions before it was sorted).

Ex:

Continuing on next page

Notes

Date / /
Page No.

$= 1+3+6 = 10$ Inversions

And by bruteforce [3-2, 3-2, 5-4, 5-2, 7-6, 7-4, 7-2, 6-4, 6-2, 4-2] = #10

T.C = same as Merge Sort = $\Theta(n\log n)$.

Que: An array is bitonic if ----

What will be the T.C. to count all inversions in bitonic array?

⇒ Step 1: Find t if not given $\Theta(n)$

Step 2: Divide array in two parts (Inc & Dec.) by t $\Theta(1)$

Step 3.1: Inc will have 0 inversions $\Theta(1)$

3.2: Dec. will have $\frac{k(k-1)}{2}$ inversion for k elements $\Theta(1)$

Step 4: Reverse the dec. array $\Theta(n)$

Step 5: Apply merge on both arrays. $\Theta(n)$

$$T.C = \Theta(n) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(n) + \Theta(n)$$
$$= \Theta(n)$$

#inversions = $0 + \frac{k(k-1)}{2}$ + merge(left, rev(right)), k = len(right)

Notes

Date / /
Page No

Implementation will have some changes like this in merge procedure:

```
if (a[p] <= a[q]) //No change
{       b[k] = a[p]; p++; k++;
}
else
{
        b[k] = a[q]; q++; k++;
        inversions += (mid - p+1)          // Extra line 1
}
}
{
return inversions;                         //Extra line 2
```

And the main method will be:

```
int ci(a, i, j)
{
        if(i=j) return;
        else
        {       mid = (i+j)/2;
                x = ci(a, i, mid);
                y = ci(a, mid+1, j);
                z = merge(a, i, mid, j);
                return x+y+z;
        }
```

→ #Inversions

Implementation not for Gate, but for me.

GO CLASSES

Notes

Another D&C Problem

★ **Closest Pairs:** In this we are given a list of some points (in 1D or 2D diff problem), we have to tell the pair which is closest. i.e. having least distance.

This problem is never ever asked in Gate. So don't emphasize too much time on this.

⇒ **for 1-D:**

Approach 1: Sort all pairs $\Theta(n\log n)$

Return pair with min. diff $\Theta(n)$

$T.C = n\log n + n$

$= \Theta(n\log n)$

Approach 2: Using D&C. Divide the array in two parts. Do recursively. Now minimum will be either in left or right or border points. So, choose minimum among them.

min(x,y,z) will be any, if x is min from left & y from right & z from dividing border.

$T(n) = 2T(n/2) + \underbrace{O(1)}_{\text{Divide}} + \underbrace{O(1)}_{\text{Merge (choose min)}}$

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

But before this we need to sort the list.

So, $T.C =$ sorting $+ T(n)$

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

$= \Theta(n\log n)$

Teacher's Signature _____

Notes

Date / /
Page No

⇒ **For 2-D:**

I/P: List of 2D coordinates $[(x_1,y_1), (x_2,y_2), (x_3,y_3) \ldots]$

O/P: Two coordinates (x_1,y_1) (x_k,y_k) with minimum distance.

~~Step 1: Sort the list on the basis of x.~~

Approach 1: Compare each coordinate with other coordinate.
↳ Naive T.C: $\frac{n(n-1)}{2} = O(n^2)$

Approach 2: Use DnC.

Step 1: Sort the list.

Step 2: Rec. Divide and find min. in left subarray & right subarry. Say S_L & S_r are min. from left & right. Now we need to calculate border points minimum. In 2-D many points can be at border (maybe all) Then, we do not need to compare all of them we will compare all of one side at border to all within min (S_L, S_R) distance and in this area, at max 6 or 8 (whatever) can be present because if more were there then S_L or S_R would have reduced So, we will only compare $\frac{n}{2} * 8$ only. i.e. 8 with each of one side and get min. of Border So

Step 3: Combine by returning min(S_L, S_R, S_B)

GO CLASSES

Notes

If we compare all with all then

$T(n) = 2T(\frac{n}{2}) + n^2$ → as on merge n^2 comparisons.

$= n^2$

But if we do constant comparison with each of left then

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

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

$T(n) = n\log n$

Total Time = Sorting + $T(n)$

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

$= \Theta(n\log n)$

So, conclusion is limited comparisons need to be done bcz, ltd can be there within range.

GO CLASSES

Notes

★ Exponent of a number:

Method 1: Naive

```
exp(a,n) {
    mul = a;
    for(i=2 to n)
        mul = mul * a;
    return mul;
}
```

multiplication = $n-1 = \Theta(n)$

Method 2: Using Divide & Conquer

2.1

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

$= \Theta(n)$

★ Best

2.2

Note 2

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

$= \Theta(\log n)$

Notes

Date / /
Page No.

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & n \text{ is even} \\ a^{n/2} \cdot a^{n/2} \cdot a & n \text{ is odd} \end{cases}$$

Algo:

```
exp. (a,n) {
    if (n==0) { return 1; }
    if (n==1) { return a; }
    x = exp (a, n/2);
    if (n is even) { return x*x; }
    else           { return x*x*a; }
}
```

$\longrightarrow \Theta(\log n)$

GO CLASSES

★ Matrix Multiplication:

The usual matrix multiplication works as:

$$\underset{A}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}} \cdot \underset{B}{\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}} = \underset{C}{\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}}$$

This — Not this — This — this Not — 4th kr dia.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

mul = n per element in C

total mul $= n * n^2$

$= n^3$

Notes

Date / /
Page No.

Algo:

```
mul(A,B)
{
    for i=1 to n {
    for j=1 to n {
        cij=0
        for k=1 to n {
            cij = cij + aik * bkj;
        }
    }
    }
    return c;
}
```

T.C = $\Theta(n^3)$

This was the naive approach, but matrix multiplication can also be done by DnC.

$$\begin{array}{c} n/2 * n/2 \\ n/2 * n/2 \end{array} \left[\begin{array}{c|c} & \\ \hline & \end{array}\right] \begin{array}{c} \rightarrow n/2 * n/2 \\ \rightarrow n/2 * n/2 \\ n \times n \end{array}$$

We can partition A & B into four $n/2 * n/2$ matrices/partitions as above

$$\underset{A}{\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]} \times \underset{B}{\left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]} = \underset{C}{\left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right]}$$

GO CLASSES

Notes

Date / /
Page No.

So, we can get c in this way also. We will divide recursively. But during merging we will have to do some additions.

Ex:

$$\underset{A_{11}\cdot B_{11}}{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}_{n/2 \times n/2}} + \underset{A_{12}\cdot B_{12}}{\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}_{n/2 \times n/2}} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots \\ & & \\ & & \cdots \end{bmatrix}$$

Now we have $\frac{n^2}{4}$ elements and that much additions we need to do.

Such 4 matrix of size $n/2 \times n/2$ will be formed i.e. C_{11}, C_{12}, C_{21} & C_{22}.

Total additions $= \frac{n^2}{4} \times 4 = n^2$ = Merging Cost.

So, our Recurrence Relation will be:

$$T(n) = \underbrace{8T\left(\frac{n}{2}\right)}_{\text{8 matrix mul of size } n/2} + \underbrace{O(1)}_{\text{Divide cost}} + \underbrace{O(n^2)}_{\text{Merge cost}}$$

By master's theorem $n^{\log_2 8} = \Theta(n^3)$

So, using this we get the same T.C. but atleast we tried.

NOTE: In 1969, Strassen was able to do it in $T(n) = 7T(\frac{n}{2}) + n^2$ i.e. $O(n^{2.808})$. We don't need to know how he did it.

Best known till now is $O(n^{2.376})$.

Strassed did again in 1986 in $O(n^{2.479})$.

$\log_2 7 = 2.808$.

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No

★★ Quick Sort: Another DnC based sorting algorithm.

In this basically we select a pivot element & then using some partition algorithm we divide array so that elements less than pivot go to its left only & that greater than it goes to its right only. This does make pivot element at the right place. We then repeat this process @ left & right subarray.

GO CLASSES

```
quicksort (a, i, j)
{
    if (i=j) return;
    else {
        q= partition (A, i, j);   → Here p will be at correct position
        quicksort (a, i, q-1);    → Here left of p will be sorted
        quicksort (a, q+1, j);
    }                             → Here whole array is sorted.
}
```

Partition is the heart of quick sort & it returns the index of pivot element after partitioning.

, returns 3, choosing 7 as pivot.

Notes

Date / /
Page No.

There are many algorithms to partition an array. We do not need to know them all. Let's see one of them:

left as pivot

```
partition (a, low, high)
{
    i=low+1 ; j=high;   pivot= a[low];
    while (i<=j)
    {
        while (i<=j && a[i] <= pivot ) i++;
        while (i<=j && a[j] > pivot) j--;
        if(i<=j) {
            swap(a[i], a[j]);
            i++; j--;
        }
    }
    swap (a[low], a[j]);
    return j;
}
```

explained

→ So, in the above algo, we keep checking that i should not cross j or vice versa by i<=j. We choose pivot as 1st element. Keep increasing i till we see greater than pivot & then keep decreasing j till we see lesser than pivot & then after encountering, both situation we swap a[i] & a[j], & repeat this till we do not cross i & j. At last swap pivot with a[j] & return j.

You can dry run this.

GO CLASSES

Notes

i/p. a =

7	1	3	12	2	10
0	1	2	3	4	5

low = 0 , high = 5

o/p : a =

0	1	2	3	4	5
1	3	2	7	12	10

returns 3.

Dry run:

0	1	2	3	4	5
~~7~~	1	3	~~12~~	~~2~~	10

2 ~~4~~ 7 12

Pivot = 7,

i = ~~0~~ ~~1~~ ~~2~~ 3

j = ~~5~~ ~~4~~ 3

(will not do this dry run again.)

At any intermediate state of partition algorithm, following invariant holds

P	all are ≤ P	i →	- - - - - -	← j	all are ≥ P.

In CLRS another partition method called Lumoto imp. is used & ans in Hoare imp., but it do not matter for gate.

GO CLASSES

Notes

Ex of Quick Sort. (Any partition, we don't care, and 1st element as pivot).

0	1	2	3	4	5	6
25	7	34	2	70	9	61

Array O/P: 2 7 9 25 34 61 70.

Calling Tree will look like.

GO CLASSES

Notes

Qu Is quick sort stable?

⇒ No, not stable as partition algo can swap anywhere. So, gen. not stable, but can be made stable by adding some complex partition algo, but our answer is that Quick Sort is not stable. (can also take example of then partition)

Qu: Is quick sort inplace?

⇒ Yes as we are not using extra array in partitioning or in sorting.

	Stable	Inplace	Divide cost	Combine ~~Merge~~ cost
Merge	✓	×	↓	↑
Quick	×	✓	↑	↓

GO CLASSES

★ **Analysis of Quick Sort:** $\Theta(n)$ is for partion always in any of the case.

* **Best Case:** If we are lucky then always we will be able to split array in equal halves.

ie.

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{\Theta(n)}_{\text{divide cost}} + \Theta(1)$$

↳ combine cost, nothing to do to combine.

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

Notes

Date / /
Page No.

Que: What if array gets divided as $\frac{n}{100}$ & $\frac{99n}{100}$?

$\Rightarrow$

So lower bound $\Omega' = \Omega(n\log_{100}n)$

upper bound $= O(n\log_{100/99}n)$

but this is same. So

$\Theta(n\log n)$.

So, in general if our array get divided in n/α and $\left(\frac{\alpha-1}{\alpha}\right)n$ parts each time even then we are in best case and the time complexity will be $\Theta(n\log n)$

* **Worst case:** This is when one side of array gets 1 element and the other side gets $(n-1)$ elements

i.e.

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

$= \Theta(n^2)$

GO CLASSES

Notes

Date / /
Page No

In general, if

then $T(n) = T(n-k) + T(k) + O(n)$
$= \underline{O(n^2)}$

So, In worst case (when k & n-k division happen), time complexity of quick sort is $O(n^2)$

In best case (when α & $\frac{d-1}{\alpha}$ division happens), time complexity is $O(n\log n)$

Avg. case time complexity is also $O(n\log n)$
↳ Proved later (optional)

⇒ Intuitively when we divide $\frac{n}{2}, \frac{n}{2}$ parts then at level 1 we position correctly & then at 2nd level 2 are positioned correctly at cost of partition i.e. $O(n)$ & then at level 3, 8 correctly positioned at $O(n)$ only. , whereas, when we divid by 1 & n-1, at each level we only position one element correctly and at same cost per level i.e. $O(n)$. So, we take more time in this case & lesser time in prior case.

Teacher's Signature

Notes

Date / /
Page No.

⇒ Now, this division is solely dependent on pivot, if we always take first or last element as pivot, then already sorted arrays (any order) will be the worst case i/p.

We can just randomise this choosing of pivot so we are less likely to hit worst case and obviously no fixed specific i/p for worst case.

So, in Randomised Quick Sort, we just pick pivot randomly and everything else is same, no change.

However it may suffer from worst case, but we cannot say which i/p can give worst case. (with less prob) $\frac{2}{6!}$ for 6 element array

```
Randomized QS (a,i,j)
{
    if (i=j) return;
    else {
        r = get random no. from i to j;   } pick Randomly
        swap (a[i], a[r])
        q = partition (a,i,j);            } same.
        RQS (a,i,q-1);
        RQS (a,q+1,j);
    }
}
```

It also has the same T.C. i.e. $\Theta(n^2)$ at worst & $\Theta(n\log n)$ at best & avg.

Median: $\underbrace{<m}_{n/2}$ m $\underbrace{>m}_{n/2}$

Notes

Date / /
Page No

Ques: T/F An opponent can force randomized quick sort to run in $\Omega(n^2)$ time by providing any array.

⇒ No, as running time is not dependent on input array, it is now dependent on random function.

Gate CSE 2014. Easy.

Ques: What if we choose middle element i.e. $\left(\frac{i+j}{2}\right)^{th}$ element always as pivot?

⇒ Worst case will be $O(n^2)$ then also, because of obvious reasons. (like first or last element as pivot)

Ques: What if I choose median of array as pivot?

⇒ Median of array can guarantee us that array will be evenly divided. But this will incur us an extra cost of finding median at each step, as it can't be done in $O(1)$ like middle. Cost is $O(n)$. But still $T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{O(n)}_{\text{Divide}} + \underbrace{O(n)}_{\text{Median}} + \underbrace{O(1)}_{\text{Combine}}$

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

So, if we choose median always, we can guarantee $\Theta(n\log n)$ in any case even in worst.

GO CLASSES

Notes

Date / /
Page No.

NOTE Even if median gives worst case as $O(n\log n)$ we don't use it bcz, worst case rarely happens and it's less efficient bcz of maybe the constant we are ignoring.

⇒ If in any que, you are provided some intermediate array, when we are asked what can be partition returning then try to check all indexes such that anything on left is less & on right is more. All those indexes can be possible ans. (Not writing such easy question).

Que: What is the prob. that, with a randomly chosen pivot element, the partition subroutine produces a split in which the size of the smaller of the two subarrays is $\geq n/5$?

⇒

For. the $\geq n/5$, pivot can belong here only. (in actual)

So, P = Choosing $3n/5$ out of n

$$= \frac{\frac{3n}{5}}{n} = \frac{3}{5} = 1 - 2 * \frac{1}{5}$$

Ans

→ In option form.

GO CLASSES

Date / /
Page No.

Notes

Qu T/F?

a). Each element can be choosen as pivot at most once.
⇒ True. (bcz, pivot then don't go to any subarray)

b) Elements are only compared to pivots.
⇒ True, (that's what partition is).

c) Every pair (a,b) in A are compared with each other at most once.
⇒ True, (think of it)

d) i & j are compared iff, i or j is pivot.
⇒ True.

GO CLASSES

★ Avg. Case analysis of Quick Sort. (Optional & skip)

Each permutation of array [1, 2, ---n] has equal prop. i.e. $\frac{1}{n!}$

Let X be the R.V. repr. no. of comparisons performed on a sample array drawn from the sample space, then E[X]?

$$X_{ij} = \begin{cases} 1 & \text{if } i \text{ \& } j \text{ are compared.} \\ 0 & \text{else} \end{cases}$$

Notes

Date / /
Page No.

comparison = $X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$ } all possible pairs.

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}]$$

i.e. probability i & j are compared.

As w.k.t i & j are compared in quick sort iff either i or j is pivot.

$E[X_{ij}]$ = prob. i & j are compared = i or j is pivot

$= \frac{2}{j-i+1}$ → These much nos.

$$E[X_{ij}] = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$= \underline{O(n \log n)}$ → That's all

★ **Select Problem**: I/P: An unsorted array & k.
O/P: k^{th} smallest element of A.

Ex: A = | 7 | 2 | 6 | 9 | 1 | 5 | 4 | 11 |

select(A,1) = 1, select(A,2) = 2, select(A,8) = 11

NOTE: select(A,1) = minimum of A.
select(A,n) = maximum of A.
select(A,n/2) = median of A.

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No

Naive approach can be doing sorting and then returning a[k]
T.C = O(nlogn).

But, can we do better?

→ One thing to observe is in partition algo, we give array & pivot and then get its actual index, but here we want is to give array and actual index and return the element whom that index belong to.

→ We can use above fact and recursively half down the size of problem like binary search. One thing that can be inferred from partitioning is that if pivot's index is lesser then k then go to right sub array only else to left subarray.

Ex: Select (A, 6)

GO CLASSES

Notes

Date / /
Page No.

```
Select (A, i, j, k)
{
    if (i==j)  return A[i];
    m= partition (A, i, j);
    if (m==k)   return  a[m];
    else if (m>k)   return  select (A, i, m-1, k) ;
    else            return  select (A, m+1, j, k) ;
}
```

Analysis:

Worst case : $T(n) = T(n-1) + \Theta(n)$ → Partition

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

Best case $T(n) = T(\frac{n}{2}) + \Theta(n)$ → Not $2T(n/2)$

$= \underline{\Theta(n)}$

NOTE ↔ There is an algo called median of medians (extremely clever) that gives approx. median & we can use that median as pivot. (Irrelevant)

GO CLASSES

Notes

Date / /
Page No

★ Searching:

Linear: I/P: unsorted array, key
O/P: Index of key

Best case = $\Theta(1)$, Avg. case = $\Theta(n/2) = \Theta(n)$, worst case = $\Theta(n)$

Binary Search: I/P: Sorted Array (unsorted won't work), key
O/P: Index of key.

Something similar to B.B.S.T (~~may be combined~~)

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

T.C Best case = $\Theta(1)$, Avg. case = $\Theta(\log n)$, worst case = $\Theta(\log n)$

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

GO CLASSES

Notes

Ex: 2, 3, 5, 10, 12, 15, 20.

Search 12

10

15

12 ✓

DNC finished

★★★ Sorting Algorithms:

★. **Bubble Sort:** It has the simple idea:

#1 Compare pair of adjacent items

#2 swap if items are out of order (order can be anything)
- Ascending
- Decreasing

#3 Repeat 1 & 2 'n' times (or no swaps)

Ex: 8 5 2 6 9 (order: Increasing)

5 8 2 6 9

5 2 8 6 9

5 2 6 8 9

5 2 6 8 | 9 → this part is surely sorted.

One Pass of Bubble sort

Notes

Date / /
Page No.

Ex: 2, 3, 5, 10, 12, 15, 20.

Search 12

10
15
12

DNC finished

Module - 3

★★★ Sorting Algorithms:

★. **Bubble Sort:** It has the simple idea:

#1 Compare pair of adjacent items
#2 swap if items are out of order (order can be anything)
- Ascending
- Decreasing

#3 Repeat 1 & 2 'n' times (or no swaps)

Ex: 8 5 2 6 9 (order - Increasing)

5 8 2 6 9

5 2 8 6 9

5 2 6 8 9

5 2 6 8 | 9 → this part is surely sorted.

One Pass of Bubble sort

Notes

→ Ignore this

5 2 6 8 | 9

2 5 6 8 | 9

2 5 6 8 | 9

2 5 6 | 8 9

← sorted →

Pass 2

2 5 6 | 8 9

2 5 6 | 8 9

2 5 | 6 8 9

Pass 3

2 5 6 8 9

2 | 5 6 8 9

Pass 4

→ Sorted

```
Bubble sort (a)
{   for i = 0 to n-2          # passes
    {
        for j = 0 to n-i
        {
            if (a[j] > a[j+1])   swap (a[j], a[j+1]);
        }
    }
}
```

GO CLASSES

Notes

Date / /
Page No.

T.C = $(n-1)+(n-2)+(n-3) \cdots +1$

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

$= O(n^2)$ → In any case.

Gate CSE 1995.

⇒ Early Termination: We can stop the outer loop (i.e. passes) if there is no swapping inside the inner loop. This will reduce T.C. for best case i.e. already sorted.

```
Bubble Sort (a)
{
    for i=0 to n-2 {
        swapped = false;
        for j = 0 to n-i {
            if (a[j] > a[j+1]) {
                swap (a[j], a[j+1]);
                swapped = true;
            }
        }
        if (swapped == false) break;
    }
}
```

So, this modification will make some changes in T.C

GO CLASSES

Teacher's Signature _____

Notes

	Earlier	Now
Best	$O(n^2)$	$O(n)$ → Already sorted → Not $O(1)$ bcz. at least you'll check if sorted or not.
Worst	$O(n^2)$	$O(n^2)$
Avg.	$O(n^2)$	$O(n^2)$

Que: Is it stable ~~inplace~~? ⇒ Yes. (we'll not swap if equal)

Que: Is it inplace? ⇒ Yes.

★ Insertion Sort: Analogy to adding card in already sorted ~~deck~~ hand of cards.

Ex:

Pass #0

82	50	12	68	17	95	35	23

sorted (S) | current (C) | unsorted (US)

Pass #1

50	82	12	68	—	—	—	23

S | US

Pass #2

12	50	82	68	17	—	—	—	23

S | US.

Pass #3

12	50	68	82	17	95	—	—	—	23

S | US

Pass #4

12	17	50	68	82	95	35	23

S | US

Pass #5

12	17	50	68	82	95	35	23

S | US

Pass #6

12	17	35	50	68	82	95	23

S | US

GO CLASSES

Notes

Date / /
Page No.

12	17	23	35	50	60	82	95

sorted.

#7

The invariant of insertion sort:

```
Insertion Sort (A)
{
    for (i=1 to n-1)
    {
        key = A[i];
        j=i-1;
        while ( j>=0 && A[j] > key)
        {       A[j+1] = A[j];
                j--;
        }
        A[j+1] = key;
    }
}
```

In this we can swap too, but it will increase T.C. w/o any sense. So instead we keep shifting & then place key, at the correct position of left part.

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No.

In best case inner while loop will not run and it will be when the array is already sorted.

T.C: best = $O(n)$
worst = $O(n^2)$
avg = $O(n^2)$

Que: Inplace? Yes.

Que: Stable? Yes (we will not shift if equal)

⇒ Insertion Sort x Binary Search: What if we search by binary search algo in left part instead of linearly search.
Then # comparison = $\log n$.

But since it is an array shifting will take $O(n)$.
So, no benefit.

If we use LL instead of array, then we can avoid shifting time, but the binary search cannot be applied there so searching will take $\Theta(n)$ in this case. So again, no benefit.

⇒ Counting Inversions x Insertion Sort: Relation b/w both is that, if I is the no. of inversion in the array, then Insertion Sort runs in $O(n+I)$ time.

GO CLASSES

Notes

Date / /
Page No.

#times we are shifting element = # inversions of element.
→ obviously.

T.C ≠ O(I) = O(I+n)

bcz in best case I=0, T.C ≠ O(1)
T.C = O(n)

bcz, atleast you will check in best case, if it is sorted.

★ **Selection Sort:** It is called the most natural algorithm bcz if you donot know any other approach, naturally this approach you will use or must have used in bachpan.

#1 find the smallest (or largest) item x.
#2 swap x with first (or last) item.
#3 Repeat #1 & #2 'n' times.

Ex: 29, 10, 14, 37, 13

29, 10, 14, 13 | 37
S

13, 10, 14 | 29, 37
S

13, 10 | 14, 29, 37
S

10, 13, 14, 29, 37 → done.
→ sorted

GO CLASSES

Notes

Date / /
Page No

```
Selection sort (a)
{   n = a.length;
    for (pass=1; pass<=n-1; pass++)
    {
        max = a[0];
        maxIndx = 0;
        for (j=1, j<n; j++)
        {
            if (a[j] > max) {
                ~~swap~~ maxIndx = j;
                max = a[j];
            }
        } ~~swap a[max]~~
        swap (a[maxIndx], a[n-1]);
        n = n-1;
    }
}
```

⇒ Is it stable? ⇒ No, as either may get swapped with max.

⇒ Is it inplace? ⇒ Yes

⇒ Time Complexity: $O(n^2)$.

GO CLASSES

Notes

★ Heapsort:

Unsorted Array $\xrightarrow{\text{Buildheap}}$ $Heap_n$ $\xrightarrow{\text{Del.max}}$ $Heap_{n-1}$ $\xrightarrow{\text{Del.max}}$ $Heap_{n-2}$ - - - $\xrightarrow{\text{Del.max}}$ $Heap_1$ → Sorted Array

$$T.C = \underbrace{O(n)}_{\text{Buildheap}} + \overbrace{O(\log n)}^{\text{Del.max}} + \overbrace{O(\log n-1)}^{\text{Del.max}} + O(\log n-2) - - - + O(\log 1)$$

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

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

$$= n + n\log n$$

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

Ex: 2 7 5 6 4

⇓

○ → Not visible.

Array = 2 4 5 6 7

GO CLASSES

Notes

Date / /
Page No

Is it stable? ⇒ No

Counter Ex: 3a 3b 2 1

⇒ 1, 2, 3b, 3a
↳ unstable

⇒ Is it inplace? Yes.

Summary of Comparison Based Sorting:

	Best	Avg.	Worst	Stable?	Inplace?
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Yes
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Heap	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	No	Yes
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	Yes	No
Quick	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	No	Yes

Quick → most used

GO CLASSES

Notes

→ Very Nice

Theorem: Any comparison based sorting must take at least $n\log n$ time in worst case.

⇒ Our C.B. sorting Algo. works as (in most basic, efficient way)

I/p: A[0], A[1]

→ A[0] > A[1]

Yes → A[1], A[0]

No → A[0], A[1]

Similarly, for three sized i/p:

I/P: a_0, a_1, a_2

$a_0 > a_1$

Yes → $a_1 > a_2$

Yes → a_0, a_1, a_2

No → $a_2 < a_0$

Yes → a_1, a_2, a_0

No → a_1, a_0, a_2

No → $a_2 < a_0$

Yes → a_2, a_0, a_1

$a_1 < a_2$

Yes → a_0, a_1, a_2

No → $a_0\ a_2\ a_1$

n! leaves

This is the best we can do, min. this needs to be done to know the sorted order. Nothing can be skipped.

Notes

Date / /
Page No

for, n sized array, our algo picks out one out of n! possible orderings and, every possible execution of the algorithm is represented as one path from the root to a leaf. (O(depth))

Maximum Leaves we can have if depth is 'd' is 2^d.

So, we can say no. of leaves $\leq 2^{depth}$, las can be less, may be not all nodes reach till depth.

So, $n! \leq 2^{depth}$

$\log n! \leq depth$

T.C = O(depth) ie $O(depth) = \Omega(\log n!) = \Omega(n \log n)$

So. T.C = $\Omega(n \log n)$

So, In worst case we need at least nlogn time to sort an array.

∴ Hence Proved.

So, also we can say merge sort is most optimal bcz. it is also taking nlogn in worst case.

Intuitive

GO CLASSES

Teacher's Signature

Notes

Date / /
Page No.

☆ Linear Time Sorting: (Non-Comparison Based)

In this we have two assumptions:

- keys are Integers
- max value of any key is $\leq k$.

⇒ Counting Sort:

Ex: I/P: 3_a 5_a 2_a 3_b 1 6 5_b 2_b

Bucket:

O/P: 1, 2a, 2b 3a, 3b, 5a, 5b, 6

So, it is stable if bucket follows FIFO (queue).

T.C = $O(k+n)$, whichever is larger

S.C = $O(k)$

where k is maximum key.

if $n \geq k$, then T.C = $O(n)$

GO CLASSES

Notes

Date / /
Page No

★ **Radix Sort:** $k = n^t$, i.e. max value is n^t here.

Idea is: Perform counting sort on l.s.digit first, then perform count sort on next least significant digit & so on.

Ex:

Count sort				
2012	5810	5810	2012	0239
1385	4961	2012	0239	1385
4961	2012 ⇒	0239 ⇒	2373 ⇒	2012
5810 ⇒	2373	4961	1385	2373
2373	1385	2373	5810	4961
0239	0239	1385	4961	5810
Unsorted				sorted

Here, count sort's stability has a major role.

Here, if we have d digit number, then, no. of iterations are d

Each iteration is take $O(n+k)$ where k is 0 to 9 so, $O(n)$

Total Running time $O(dn)$

if $d = \log_{10} n^t$ where n^t is the max value

$O(nd) = O(n \cdot \log_{10} n^t) = O(nt \log n)$ → k is base

GO CLASSES

PAGE NO. :
DATE : / /

Module- 4

☆ Graphs in Algorithms

⇒ These are not those data viz. graphs. These show up all the time like connections in linkedin. or in google pages, neural connections in brain and many more.

→ Graphs are set of vertices and edges b/w them.

$G = (V, E)$

→ Graphs can be directed or undirected.

⇒ **Connectivity in Undirected graph:** Undirected graph is connected if for all pairs of vertices u, v, there exists a path from u to v.

Connected graph

Disconnected Graph

→ An undirected graph is complete a.k.a fully connected if for all pairs of vertices u, v, there exists an edge from u to v.

Ex:

→ A directed graph is strongly connected if there is a path from every vertex to other vertex.

Ex:

PAGE NO. :
DATE : / /

→ A directed graph is weekly connected if there is a path from every vertex to every other vertex, only if we ignore the direction.

Ex:

→ A complete, aka fully connected directed graph has an edge from every vertex to every other vertex.

Ex:

GO CLASSES

★ Storing Graphs (Data structure): We need to store both the set of nodes & the set of edges.

a). Adjacency Matrix: $n \times n$ matrix, where $A_{ij} = 1$ if i, j is an edge. (n is no. of vertices)

Ex:

$$\Rightarrow \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 1 \\ c & 0 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{array}$$

for directed graph:

$$\Rightarrow \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 0 & 0 & 1 \\ b & 1 & 0 & 1 & 0 \\ c & 0 & 0 & 0 & 0 \\ d & 0 & 1 & 1 & 0 \end{array}$$

PAGE NO. :
DATE : / /

→ In case of undirected graph, matrix will be symmetric.

b). **Adjacency List:**

Ex:

⇒

= Vertices

→ 2E if undirected

→ E if directed.

degree(a) + deg(b) ... + deg(n) = 2E if undirected

E if directed. (outdegree)

= Θ(E)

→ Handshaking Lemma

⇒ **Some Time Complexities:**

	Adjacency List	Adjacency Matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if u, v is an Edge	$\Theta(1+\deg(u))$	$\Theta(1)$
List, W's neighbours	$\Theta(1+\deg(W))$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$\Theta(1)$	$\Theta(1)$
Delete edge uv	$\Theta(1+\deg(u)+\deg(v))$	$\Theta(1)$

GO CLASSES

PAGE NO. :
DATE : / /

Listing all neighbours is the most frequent task, & it is efficient for adjacency list, so we prefer adjacency list.

→ Which is faster $O(V^{3/2})$ OR $O(E)$, this thing depends upon the density of the graph i.e. #edges w.r.t #vertices.

⇒ **Dense Graphs:** In dense graph $E \approx |V^2|$, i.e. almost fully connected, each vertice with other vertices is connected via edge.

Ex:

⇒ **Sparse Graphs:** In sparse graphs, $E \approx |V|$, i.e each vertex has only a few edges.

Ex:

So, now coming to above question.

$O(V^{3/2})$ vs $O(E)$

$O(E)$ is $O(V^2)$ in dense & $O(V)$ in sparse, so $O(E)$ is faster in sparse & $O(V^{3/2})$ is faster in dense graph.

PAGE NO. :
DATE : / /

In tree traversal we had a fix order (pre, post, in or any other), obvious start & end.
But, in Graph traversal, we can have any order, and start & end is also not obvious.

There are two types of traversals or search methods:

- Breadth first search (BFS) and Depth first search (DFS)

⇒ Given a graph, how we search naively. we pick any node, put all its neighbours in bag. Now pop any node from the bag & put its neighbours in bag and do this untill nodes are popped from the bag. Now, if we follow a fix sequence for popping, then we can name the traversal, if we pop as LIFO (i.e. stack), then it is DFS and if we pop in FIFO sequence (i.e. bag is queue), then it is BFS.

```
whatever-first-search (G,s):
    put s in bag
    while bag is non-empty:
        Extract v out of bag:
            if v is unmarked:
                mark v
                for each edge (v,w):
                    put w into bag if w is unmarked.
```

⇒ if bag is stack it is DFS, else if bag is queue, it is BFS.

GO CLASSES

PAGE NO. :
DATE : / /

☆ Depth first Search:

Something similar to solving maze, we go any way, explore path if it is blocked we back track.
→ get back to prev. node.

Ex:

DFS(a) will give this depth first tree (DFT), (many DFT possible)

Start time or discovery time is the time we visit vertex v first time (S[v] or d[v]), and finish time is the time we visit vertex v last time (F[v]).

Use conv. S[v]/F[v] on below graph on DFS(a)

PAGE NO. :
DATE : / /

We can draw ~~tree~~ DFT from the S[v]/F[v] format and can do vice versa also.

Que: Run DFS(u) and make DFT_1 & then on DFS(w) to make DFT_2.

$DFT_1 + DFT_2$ = Depth first forest DFF

Writing S[v]/F[v] time format:

NOTE: We can also start from ~~zero~~ one.

PAGE NO. :
DATE : / /

⇒ Implementation of DFS:

```
Visit_DFS (G, a):
    visited[a] = T;
    for each neighbour w of a:
        if (visited[w] == F):
            DFS(G, w);
```

] degree(a) times

→ Here visited array is initialised with false initially. It will just create one DFT. If there is DFT and more DFT, then we need to call it for other unvisited node (will be not from that DFT).

So, DFS(G, a) is called by:

```
DFS(G):
    for each vertex u in vertices:
        visited[u] = false;
    for each vertex u in vertices:
        if (visited[u] = false):
            visit_DFS(G, u);
```

] V times.

⇒ Analysis of DFS:

called exactly once either in DFS or visit_dfs method

Vertex	Adj List	Adj. Matrix
v_1	$1+deg(v_1)$	V
v_2	$1+deg(v_2)$	V
⋮	⋮	⋮
v_n	$1+deg(v_n)$	V
	$V+2E$	V^2

V vertices

PAGE NO. :
DATE : / /

So, DFS T.C $\begin{cases} O(V+E) & \text{Adj. List} \\ O(V^2) & \text{Adj. matrix} \end{cases}$

In CLRS book, instead of visited we have color which can have three values, white if unvisited, black if finished and grey if in stack.

→ we can also traverse graph by DFS using stack instead of using backtracking.

Ex.

GO CLASSES

stack. ~~A~~ ~~E~~ ~~B~~ ~~F~~ ~~G~~ ~~H~~ ~~I~~ ~~C~~ ~~D~~ (visit top most element of stack) LIFO

Order: A E G H I D B C F

★ <u>DFS parenthesis Theorem</u>: In any DFS of a (directed or undirected) graph $G=(V,E)$, for any two vertices u, v, exactly one of the following three conditions holds:

- Intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are ~~intire~~ entirely disjoint ie neither u nor v is a decendent of the other in DFF,

PAGE NO. :
DATE : / /

- the interval $[d[u], f[u]]$ is contained entirely within the interval $(d[v], f[v])$ & u is a decendent of v in a DFT, or

- the interval $[d[v], f[v]]$ is contained entirely within the interval $(d[u], f[u])$ & v is a decendent of u in a DFT.

So,

or

is not possible.

(Similar to balanced pair paranthesis.)

Notation

2/7 ○ v ⇒

Node

Que: Construct the DFT of G which realised d/f intervals:

v_1 : [4/5]
v_2 : [7/8]
v_3 : [12/13]
v_4 : [14/15]
v_5 : [3,6]
v_6 : [2,9]
v_7 : [1,10]
v_8 : [11,16]

→ In our new notation:

Two DFTs.

PAGE NO.:
DATE: / /

Now trees can be drawn easily:

So, DFF and paranthesis notation can be drawn from each other, very easily.

GO CLASSES

Que: Draw intervals from below DFF?

⇒

Que: Which of the following are impossible?

⇒ a) $d(u) < d(v) < f(v) < f(u)$

b) $d(u) < f(u) < d(v) < f(v)$

Correct option ← c) $d(u) < d(v) < f(u) < f(v)$

d) $d(v) < d(u) < f(u) < f(v)$

⇒ c

PAGE NO. :
DATE : / /

★ Edge Classification in DFS:

- DFS
  - Directed
    - tree
    - cross
    - back
    - fwd
  - Undirected
    - tree
    - back

Tree Edge: Edges in the DFS forest.

Fwd Edge: Non-Tree Edge, going from ancestor in tree to descendent
fwd edge$(x,y) \Rightarrow d(x) < d(y) < f(y) < f(x)$

Backward Edge: " " " " " " descendent " " " ancestor.
bwd edge$(x,y) \Rightarrow d(y) < d(x) < f(x) < f(y)$

Cross Edge: " " " $u \to v$, u in neither descendent nor ancestor of v
cross edge$(x,y) \Rightarrow$ intervals $[d(x), f(x)]$ & $[d(y), f(y)]$ are disjoint.

Go for Example

Ex:

Graph.

Back Edge Represents Cycle

PAGE NO.:
DATE: / /

DFS of prev. pg. graph:

DFS

→ Tree Edge
a→b, b→d, etc.

- - - →
→ cross
↳ g→b, h→d, c→d, c→e
→ fwd
↳ a→d, a→f
→ backward
↳ e→a

Now, we can easily classify edges given DFS or Intervals.

→ We have already noted, that all four types of edges are possible for directed graph, but this is not the case case in undirected graph. (No cross & fwd edge in it)

Theorem: In DFS of an undirected graph G, every edge of G is either a tree edge or a back edge.

→ Obviously if there was a cross edge, then that subgraph must have been in left.

There is a path from any ancestor to descendent using tree edge & if there is a back edge, then, there is a back path from descendent to ancestor, and so is a cycle. (Also in undirected)

No. of DFT in undirected graph = No. of connected components in undirected graph

PAGE NO.:
DATE: / /

Ex.

Directed Graph.

DFT. ---> Back Edge
↳ No cross edge.

Fwd edge and back edge are same in undirected graph's DFT, but we call it as back edge only.

Ques: T/F

a). DFS of a directed graph always produces same no. of tree edges.

→ False, it may even have diff no. of DFTs depending upon order & starting vertex. (True, in case of undirected graph).

b). If a dfs produce exactly one back edge on a directed graph then is it possible to have any edge i.e so that removing it from graph will make it acyclic

⇒ True, back edge

⟶ Explained Next Pg.

PAGE NO. :
DATE : / /

Theorem: A graph (directed or undirected) has cycle iff it has a back edge.

Back Edge $\longleftrightarrow$ Cycle.

One back edge can give multiple cycles, but each back edge will give atleast one cycle.

1 back edge $\rightarrow$ atleast one cycles
2 back edges $\rightarrow$ atleast two cycles.

$\rightarrow$ One back edge $\Rightarrow$ 2 cycles.

We can remove the back edge & all the cycles will be removed. (No guarantee of any other single edge other than back edge such that removing it will destroy cycle in the graph, it may or may not destroy cycle).

1 ~~cycle~~ back edge
$\Downarrow$
Atleast one cycle
$\Downarrow$
~~back~~ one edge will surely destroy all cycles (that edge will be back edge), there could be other edges too.

2 back edges
$\Downarrow$
atleast two cycles
$\Downarrow$
One edge may or may not destroy cycles. If one edge destroys then it is not back edge.

GO CLASSES

PAGE NO. :
DATE : / /

Que: T/F If a directed graph is cyclic, but can be made acyclic by removing one edge, then DFS in G will have exactly one back edge.

⇒ False, not necessarily.
↳ It can be fwd edge also, but there it need atleast 2 back edges

☆ Applications of Depth First Search: (167th slide)

In undirected graph:
→ Connected Components, → Articulation Points,
→ Bridges, → Biconnected Comp, etc.

In directed graph:
→ Cyclic/Acyclic Graph → Topological Sort
→ Strongly Connected Comp. , etc

We will go through some of the application of DFS. i.e these can be found in same T.C. as DFS.

a) Detecting Cycles (Directed or Undirected): we can easily check edge types after a DFS and as w.k.t. a graph is cyclic iff it has a back edge. So that's the simplest way to detect cycles using DFS.

GO CLASSES

PAGE NO. :
DATE : / /

It's implementation

```
DFS_visit(u):
    visited[u] = true;
    for all neighbour w of u:
        if visited[w] == true && parent[u] != w:
            return "cycle found";
        if not visited[w]:
            parent[w] = u;
            dfs_visit(w);
```

So, we can determine if any graph (directed or undirected) has a cycle in $O(V+E)$ time.

★ **DAG**: Directed acyclic graph

Ex:

⇒ In this we always have, for any edge $A \to B$

$A \to B$, $f[A] > f[B]$

finish time of A.

Ex:

$f = x$ $f = y$ $f = z$

$x > y > z$

In any order, this is True. [Either choose A, or B directly]

However no relationship b/w start time, but it is for finish time.

GO CLASSES

PAGE NO. :
DATE : / /

It is only true for DAG, in case of cycle in last ex, A will reach from B & A will finish early.

We prefer to write DAGraph in such a fashion that edges are from left to right, such a order of DAG will directly give us topolgical sort i.e. finish order of nodes.

Ex:

Now topological order will be 4, 3, 2, 1 i.e. 4 will finish first & 1 at last.

Note no relation in start time, it is only for finish time.

b). Topological Sort (DAG only): We have already seen it.

Practical Ex:

Now many topological sorts are possible.

→ If there are cycle then topological sort is never possible.

PAGE NO. :
DATE : / /

DAG $\overset{iff}{\longleftrightarrow}$ Topological sort.

Finding Topo. Sort w/o DFS is:

go to a vertex with indegree zero, and for all its neighbours decrease indegree by 1. Repeat this step.

Finding Topo. sort using DFS:

→ Call DFS to compute finishing time f[v] for every vertex.
→ Insert it onto the the front of a list as every vertex is finished.
→ Return the list.

→ In topo. sort all dependents gets finished first & then the node itself.

4. Cut Vertex / Articulation Point (only for undirected) : It is a vertex whose removal from graph disconnects the graph.

Ex:

Here 2 & 6 are articulation points because there removal will disconnect the graph.

GO CLASSES

DFT need not be binary)

PAGE NO. :
DATE : / /

Brute force Approach to find all articulation points: Remove each vertex one by one and add prev. before removing other & keep checking using DFS if graph is still connected or not.

$T.C = O(V * (V+E))$

V vertices

DFS

★ **DFS based approach:** Find DFS of graph and make DFT. Now articulation point can be at internal node, root node or leaf node.

Case I: A.P. is at root

Ex:

Another DFT of same graph

DFT of prev. graph.

*⇒ If node at root has atleast two children then it is A. Point. Here 2 is but not 1.

If root has just one child then root is not A. Point.

(Here 1→4 cross edge is not possible, if it was there then 1 would have been child of 4 instead of 2)

PAGE NO. :
DATE : / /

Case 2: A Point is an Internal Node:

*⇒ Any internal node is an A point iff there is atleast one subtree of it such that none of it's descendents are directly connected to it's ansestors.

Ex:

5 is not an A Point due to this edge.

Here 6 is an A Point.

This is an Apoint (bcz we need anyone subtree only)

It don't have backedge.

It has a back edge

Case 3: A Point is any leaf node: Impossible.

Que: ~~no~~ we choose a leaf node of DFT & run DFS on that leaf node now. How many children do the root have now?

⇒ ~~[illegible]~~ 1 only

GO CLASSES

PAGE NO. :
DATE : / /

We maintain an array called low & start time array to implement this.

* **Que T/F:** u is an Apoint iff in DFT all of its descendent should connect to its ancestor through u only.

→ False, this can connect if we have two subtrees but one atleast should satisfy this.

u is Apoint ⇒ all path go through u only } False

Q) Bridge: Edge whose deletion ~~creates~~ disconnects the graph. It can also be detected using DFS as:

Edge (u,v) is a bridge iff none of the vertices v and its descendents in the DFT has a back edge to u or any of its ancestors. Indeed, this condition means that there is none of the other way from u to v except for edge (u,v).

Ex.

2-5 is Bridge

DFT

GO CLASSES

PAGE NO. :
DATE : / /

★ **Breadth First Search (BFS)** : In this we explore all neighbours of root first & then went for their grand-neighbours.

Ex:

Note that DFS tree tend to be long and skinny while BFS trees are short & fat.

Implementation:

```
BFS(G,v):
    visited[v] = true;
    enque(v);
    while(queue is non-empty):
        u = deque();
        for each neighbour w of q:
            if (visited[w] == false):
                visited[w] = true;
                enque(w);
```

T.C: adj. List = $1 + \deg(v_1) + 1 + \deg(v_2) \ldots + 1 + \deg(v_v)$ (v times)

$= V + E = O(V+E)$

adj matrix = $V + V + \ldots + V = V^2$

GO CLASSES

PAGE NO. :
DATE : / /

★ Applications of BFS:

a) **Shortest Path in unweighted graph:** BFT tells us the shortest path only. (but only if graph is unweighted).

Ex:

DFT don't give us shortest path.

b). **BFS queue state (frontier):** Just by looking at the partial BFT we can say about the queue state:

a). Internal nodes can never be in queue.

b). only last two level leaves could be present.

So, we can say that BFS queue at any step can atmost have leave nodes of last two levels only.

Ex:

✓ are only possible to be in queue state at max.

GO CLASSES

PAGE NO. :
DATE : / /

c) BFT as layers:

NOTE ↔ We cannot have skip level edges in BFS undirected graph & skip level fwd edges are also not allowed in BFS directed graph. tree.

Not Allowed Not Allowed Allowed.

→ We can have within level, cross edges

Allowed. Allowed

GO CLASSES

PAGE NO. :
DATE : / /

We can also have cross edges in continuous levels:

(undirected)

Allowed.

Note that within level cross edge will give odd length cycle (2n+1 nodes) & cross edge in continuous levels will give even length cycles (2n nodes).

Ques: Suppose that vertices v_i & v_j are enqueued during the execution of BFS & that v_i is enqued before v_j. Then: (d(x) means distance of vertex x from src) which can be True?

a) $d[v_i] \leq d[v_j]$

⇒ True.

b) $d[v_i] \leq d[v_j]+1$

⇒ True

Why True? bcz. not necessary to be equal eq. to $d[v_i] < d[v_j]$

Ques: Can you have an edge within a layer?

⇒ Yes, if u & v are neighbours & both also have a parent at level i, then both will be in level i+1, before edge b/w them can be processed.

PAGE NO.:
DATE: / /

d) Bipartite Graph Detection using BFS:

Graph-Theory

Theorem: A graph is bipartite iff there is no odd length cycle.

Basically in bipartite graph vertices can be grouped into two parts and there is no edge b/w vertices of their own grp.

Ex:

Note that tree is always bipartite.

GO CLASSES

Graph ⇒ BFS ⇒ BFT

- → No cross edge ⇒ Only Tree Edge ⇒ Graph is Tree ⇓ Bipartite
- → Cross Edge
  - → B/w levels ⇒ Odd length cycle ⇒ Not Bipartite
  - → Continuous Level ⇒ Even cycle ⇒ Bipartite

So, if BFT of any graph has any cross edge b/w levels then that graph is not bipartite.

e) **No. of connected Components:** Can be easily detected using BFS or DFS.

Just count how many times we need to call.

Module-5 Greedy Algorithms:

PAGE NO.:
DATE: / /

☆☆ Shortest Path Algorithms:

In greedy algorithms we chose what is best at this moment.
Ex: Frog crossing a pond by stepping (jumping max.

Greedy might not work sometimes.
Ex: Change of 30 if Coin of 25 & 10 are available.

So, greedy paradigm says:
Commit to choices one-at-a time,
never look back,
& hope for the best.

⇒ Properties of greedy algorithms:

a) Optimal substructure: The optimal solution contains optimal solutions to subproblems.
(Also common to Dynamic Programming Paradigm)

b) Greedy Choice: A global optimum can be arrived at by selecting a local optimum.

NOTE ⟷ Proving these properties is sometimes hard, so, we won't do it for every problem.

First greedy problem is single source shortest path.

GO CLASSES

PAGE NO. :
DATE : / /

Single Source Shortest Path: Here i/p is a graph with weights ~~mostly~~ (directed or undirected).

and a source.

Output is shortest path from the source to all other nodes.

Attempt 1: Bruteforce Approach

Find all possible paths and then choose the shortest path.

Ex:

All paths from A to E

a)

c)

b)

d)

Here d is the shortest path from A to E.

But main problem here is that paths can be exponential

there 2^k paths are possible.

GO CLASSES

PAGE NO. :
DATE : / /

So brute force approach works but it has exponential T.C.

⇒ **Attempt 2:** Take 2 shortest paths & join them. For s to u find min s to x & min v to u.

Ex:

$s \to x + x \to y + y \to v = \underline{\underline{3}}$

Counter Ex:

Path by our algo = 4 units

actual shortest path = 3

So, this approach is not even working.

⇒ **Attempt 3:** Use BFS

- Will work only for unweighted graph. As same weight of each edge.

⇒ **Attempt 4:** Use BFS + Introduce Dummy nodes to make same edge weight

Ex:

Now apply DFS on it.

$A \to C \to B$
↳ shortest

GO CLASSES

PAGE NO. :
DATE : / /

This approach is correct but not feasible, because if we have one edge of weight 20 & another of 20 crore weight, then we will have one crore dummy vertices, finding DFS on this much vertices will not be feasible.

Attempt 1 (Dijkstra) **Dijkstra Algorithm (Greedy)**

Ex:

Assume in graph S is directly connected to A, B, C only then one thing that we can note is from S→B, there can be path via A which can be less than 2 (1+0.1), so we can't say S→B is final shortest. Similarly S→C is also not determined to be shortest. But S→A cannot be possible to be less than 1 as S→B is 2 and S→C is three provided that there is no negative edge.

So we can say S→A has this path as shortest path only. So we will final A and now will explore it.

Now the minimum is S→B bcz we cannot come to it via A & also not via A→x, A→y or A→z. So final B & explore it.

PAGE NO. :
DATE : / /

Now continue this step to at each pass find one shortest path from all known distances and then explore all paths from it.

This algorithm is none other than Dijkstra Algorithm to find shortest path from single source given by Edger Dijkstra in 1959. It was essentially rediscovery of prim's algorithm.

Easiest algo of it:

#1 pick min node,

#2 relax all outgoing edges from the min node,

#3 Repeat #1, #2 till all nodes are relaxed.

⇒ <u>Relaxing an Edge (U, V)</u>

Relax (U, V, w):

```
if ( d[V] > d[U] + w)
    d[V] = d[U] + w;
    parent[V] = U;
else
    return;
return;
```

So relaxing just updates the distance of node from the source as minimum.

PAGE NO. :
DATE : / /

Dijkstra(G,s):

```
key[v] = ∞ for all vertices of G
key[s] = 0
make min heap Q from key array
explored = { }
while Q is not empty:
    u = Extract Min(Q)
    explored.append(u);
    for each neighbour of u as v with edge w:
        Relax(u, v, w)
```

So, this is implementation of Dijkstra. Here loop is just getting min & relaxing all outgoing edges. (key ↔ d) ↳ Array

Used two terms by mistake

Working Example of Dijkstra:

~~(x) means order of choosing, writing~~

Seq. of finalising: $0 \to 2 \to 1 \to 4 \to 5 \to 3$

PAGE NO. :
DATE : / /

**

Que: Fill the table by stepping through Dijkstra's Algorithm:

Known Vertices A, B, C, G, E, D/F, F/D

Vertex	Priority Queue	Parent
✓A	0	–
✓B	~~∞~~ 1	A
✓C	~~∞~~ ~~3~~ 2	~~A~~ B
✓D	~~∞~~ ~~8~~ 7	~~B~~ E
✓E	~~∞~~ ~~6~~ 5	~~B~~ C
✓F	~~∞~~ ~~10~~ 7	~~A~~ E
✓G	~~∞~~ 3	B

Lowest Cost from A to F 7 ?

$A \xrightarrow{1} B \xrightarrow{1} C \xrightarrow{3} E \xrightarrow{2} F = 7$

Gate CSE 96, 12

GO CLASSES

PAGE NO. :
DATE : / /

★ Does dijkstra work on negative edges?

⇒ In some cases it may work:

Ex:

But, in this case it is not working.

Shortest path in actual will be $A \xrightarrow{(1)} B \to C \to D$ for $A \xrightarrow{(2)} D$.

So, Dijkstra do not work in some of the cases of negative edges, it may work if you are lucky enough.

It is not working bcz as explained initially if we choose S to A bcz it is min, then from nowhere nothing less than it is possible. But if negative edges come our assumption goes wrong & dijkstra do not work.

Dijkstra failed.

PAGE NO. :
DATE : / /

⇒ A naive approach that may come to our mind is we can add some weight to make all edges non negative & apply dijkstra. But this will still be wrong. bcz in shortest path no. of nodes may be more so weight will increase more.

Ex:

⇓ +2

Shortest path changed by making all edges positive for A→B→C to A→C. So, wrong Approach.

Ex-2

⇓ +2

Didn't change bcz you were lucky.

So, this don't work bcz adding a weight to every edge adds more weight to longer paths than shorter paths. (longer & shorter in term of no. of nodes).

So, dijkstra fails for negative edges.

GO CLASSES

PAGE NO. :
DATE : / /

★ Shortest path with a cycle:

→ Cycle can be positive (i.e. weight increases if we take it), then why would I even take it unnecessarily.

Ex:

Shortest path = 10+1+2+3+15 = 31

with one cycle = 10+1+2+3+(-1)+2+1+2+3+15 = 31+8 = 39

with two time = 39+8 = 47

So if cycle is increasing weight, why will I choose it.

→ Cycle can be negative (i.e. weight decreases if we take it), then shortest path do not exist. bcz, we will keep reducing weight by going through cycle various time.

Ex:

w/o cycle = 10+2+3+15 ⇒ 30

One cycle = 10+2+3+(-4)+(-4)+15 = 17

two cycle = 27-3 ⇒ 24

three cycles = 24-3 = 21

So, we will go in cycles & keep reducing weight if cycle is negative. Due to this in negative cycle shortest path do not exist

PAGE NO. :
DATE : / /

NOTE: In case of undirected ~~edge~~ graph, just one negative edge acts like a negative cycle. So undirected graph with -ive edge do not have shortest path.

Ex:

So, from a shortest path algorithm, if there is a negative cycle exist we expect it to detect this negative cycle because shortest path do not exist.

⇒ Note that dijkstra was a greedy algorithm as we are choosing local optimum (minimum edge) always.

⇒ Dijkstra cannot detect negative cycle, indeed, it may even not work if there is a negative edge.
(cycle to dur, edge me hi fail hojaegi).

	-ive cycle	-ive edge or -ive cycle
Dijkstra	X	X
Actual Answer Expected	X	✓

★ Time Complexity of Dijkstra

No. of times we are relaxing an edge : once only.
Time taken for relaxing an edge : ~~constant (O(1))~~ ~~Ther. only~~ ~~if also~~ $\log V$ because we decrease key in heap.
if using heap

GO CLASSES

$\Theta(E+V) \equiv \Theta(E*V)$ bcz E can be $\Theta(1)$ or $\Theta(V^2)$

$\Theta(E+V^2) \equiv O(V^2)$ bcz E can be atmost $O(V^2)$

PAGE NO. :
DATE : / /

No. of time we extract minimum from heap = V times (V vertices)

Time per extraction = log V. (ie using heap)

(Adj. List) & Heap

Total T.C = Edge * T.C. (decreasekey) + Vertices * T.C (Extract min)

$= E \cdot \log V + V \cdot \log V$

for Adj. Matrix with Heap : $E \cdot \log V + V \cdot \log V + V^2$

↳ Extra to find all neighbours V times. (In list it was V+2E)

while Relaxing

$$T.C = \sum_{u \in V} \left[T(\text{remove min}) + \left(\sum_{v \in u.\text{neighbors}} T(\text{decreasekey}) \right) \right]$$

$$= \sum_{u \in V} T(\text{remove min}) + \sum_{u \in V} \sum_{v \in u.\text{neighbors}} T(\text{dec. key})$$

***

Now, there can be various time complexities on the basis of data structures we use to implement dijkstra:

Graph Repr.	Priority Queue D.S.	T.C.
Adjacency List	Heap	$E \log V + V \log V$
" Matrix	"	$E \log V + V \log V + V^2$
" List	Unsorted Array	$V \cdot V + E \cdot 1 = V^2 + E = O(V^2)$
" Matrix	"	$V \cdot V + E \cdot 1 + V^2 = V^2 + E = O(V^2)$
" List	Sorted Array	$V \cdot 1 + E \cdot V = O(EV)$
" Matrix	"	$V \cdot 1 + E \cdot V + V^2 = O(EV + V^2)$
" List	Fibonacci Heap	$V \cdot \log V + E \cdot 1 = O(V \log V + E)$
" Matrix	"	$V \cdot \log V + E \cdot 1 + V^2 = O(V \log V + V^2)$

GO CLASSES

PAGE NO. :
DATE : / /

Fibbonacci Heap is not in our syllabus. It works such that
T(remove min) = O(log V) like min Heap &
T(update key) = O(1) unlike min Heap (log V)

* So, overall best T.C. of Dijkstra is O(V log V + E) for adj list & O(V log V + V²) for adj. matrix by fibonacci heap

** Shortest Path in DAG (Directed Acyclic Graph) :-

Any DAG can be redrawn in new formal where edge go from L to R only.

Ex:

||| ↓ Redrawn

Equivalent topological order of above DAG.

This topological order can be easily found using DFS in O(V+E) time.

By getting finish times

GO CLASSES

PAGE NO. :
DATE : / /

Implementation of shortest path for DAG:

Set $d[s]=0$ & $d[v]=\infty$ for all vertices except s
topologically sort the vertices $(O(V+E))$
for each vertex u in topological order:
    for each neighbour v of u:
        Relax(u,v)

T.C = $O(V+E) + O(V+E)$
↳ sorting ↳ loop
$= O(V+E)$

So, dijkstra says pick minimum to relax outgoing edges, DAG shortest path algo says pick in topological order to relax outgoing edges.

Working Ex: (directing taking topological order)

Order of ~~relax~~ picking: s, t, x, y, z obviously

PAGE NO. :
DATE : / /

→ Not Greedy, Dynamic

☆ **Bellman Ford:** It is also a single source shortest path finding algorithm. It can be called as attempt 6 of that problem as dijkstra also failed for -ive edges & was not able to detect negative cycles.

However, this is not a greedy algorithm, but is the best algo to find shortest path. It is based on a simple theorem we will explore further. (Not Greedy)

Path Relaxation Theorem (Property): If we relax edges in the order of shortest path along with intermixed other relaxations also, then we will get shortest path cost.

Ex: Consider graph & shortest path from s to t. has following relaxation order $s-v_1$, v_1-v_3, v_3-v_2 & v_4-t.

Now if we mix above order with any other relaxations we will still get the shortest path:

$s-v_2$, v_1-v_2, $s-v_1$, v_1-v_4, $s-v_1$, v_1-v_3, v_1-v_3, v_3-v_4, v_3-t, v_4-t, v_3-v_4

PAGE NO. :
DATE : / /

If we have V vertices then shortest path can contain V-1 edges.

Now, if we relax all edges V-1 times in any of the order, we can guarantee that somewhere shortest path order is also come with intermix of relaxations

Ex

Relax all edges 4 times in any order so you will somewhere get shortest path order.

Assume e_6, e_5, e_3, e_1 is shortest path then.

any order	need not be same		
$e_1, e_2, e_3, e_4, \underline{e_6}, e_5, e_7$	$e_7, e_6, e_4, e_3, e_2, e_1, \underline{e_5}$	$e_4, e_2, \underline{e_3}, e_1, e_5, e_6, e_7$	$e_1, \underline{e_2}, e_7, e_5, e_6, e_7, e_3$
1	2	3	4

We got our order.

We can follow any order & diff. orders can be followed in all V-1 passes, but while doing questions we will just follow a fix sequence all times

⇒ So, the core idea behind bellman ford algorithm is that if we relax V-1 times then we are guaranteed to relax in shortest path order for every vertex.

GO CLASSES

PAGE NO. :
DATE : / /

Ex:

Pass 1: A-B, A-C, D-C, B-D, C-B
Pass 2: A-C, A-B, D-C, B-D, C-B
Pass 3: A-C, A-B, D-C, B-D, C-D

} Same will also work i.e we will do same in exam questions

Not changing any weight
(Means already got ans in 2nd pass)

Note that negative edges do not impact shortest path in Bellman-Ford Algorithm. So, it works on -ive edges.

Ex:

Here dijkstra failed, but B-F worked.

★ Detecting -ive Cycles Using Bellman Ford: For this we will run algo V times instead of V-1 times. If the weights are changing at V^{th} time also, then the graph contains negative cycle, else it do not contain -ive cycle.

Obviously, if -ive cycle will be there our weight will keep changing always in each iteration.

PAGE NO. :
DATE : / /

B-F :

Relax all edges V-1 times

Relax once more to check -ive cycle.

we may have to not come till last pass for shortest path

we may even get in lesser passes.

```
B-F (G,s) :
    d[v]=∞ for all vertices v
    d[s]=0;
    for i=1 to n-1:
        for each edge (u,v) & weight w:
            relax(u,v,w)
    for each edge (u,v) & weight w:
        if (d[v] > d[u]+w)
            return false
    return True
```

```
Relax (u,v,w) :
    if (d[v] > d[u]+w)
        d[v] = d[u]+w;
        parent[v] = u
```

T.C = VE + VE

= O(VE) (if Relaxing is taking O(1) using fibonacci heap)

Dijkstra was taking O(V log V + E log V)

GO CLASSES

PAGE NO. :
DATE : / /

Note that after relaxing k times, we have shortest path for all those vertices which have atmost 'k' shortest path length.

We can also implement early termination in B-Ford algorithm like bubble sort using some boolean easily.

Worst case of BF = $O(VE)$
Best case of BF = $O(E)$

★★ Minimum Spanning Trees (MST):

⇒ **Spanning Tree:** A spanning tree of an undirected graph is a connected subgraph which contains all the vertices and has no cycles.

Ex:

A connected undirected graph

Now, some of the spanning trees of the graph are:

etc

GO CLASSES

PAGE NO. :
DATE : / /

In a spanning tree of graph with v vertices, no. of edges is always v-1.

⇒ **Minimum Spanning Tree:** The cheapest way to connect a graph. Spanning tree with min. weighted edges.

Ex:

Some of (all actually) spanning trees:

weight = 74

S. Trees

w = 71

S. Tree 2

w = 72

S. Tree 3

So, we can say 2nd tree is the minimum spanning tree (MST).

GO CLASSES

PAGE NO. :
DATE : / /

⇒ So, a MST of a graph connects all the vertices together while minimizing the no. of edges used & their weights.

Que: What will be the MST of given graph?

⇒ There are various S. Trees possible (exactly - 8 as remove 1 out of 8 edges). But MST will have min weight so remove edge with max weight i.e. f-e. So MST will be.

★ **Applications of MST:** There are several applications of MST:

a) **Communication Networks:** If a company want to have comm b/w its various location with min. cost.

b). **Clustering:** find clusters of nodes (ML related)

Node ↔ Vertices

PAGE NO. :
DATE : / /

c) **Muddy City Problem:** Once, there was a city w/o roads & moving around city became difficult after rains due to muds. So, mayor decided to make roads in some of the streets. If we treat city as graph, then we can get its MST & connect all nodes of city at min. cost.

d) Taxonomy,

etc (Many Application ≃15 in slide)
↳ Not needed for GATE

★ Brute force way to find MST is find all Trees & then choose min. among them. Process is correct but not efficient.

Ex:

Such a graph will have $\Omega(2^n)$ spanning trees atleast. So, T.C. will be exponential.

NOTE ⟷ In a complete graph (K_n) of n vertices, we have n^{n-2} spanning trees.
↳ Cayley's formula

GO CLASSES

PAGE NO. :
DATE : / /

Imp. (By heart these properties)

★ Two important Properties of MSTs are:

a) **Cut Property:** If all edge weights are distinct, then the smallest edge crossing any cut is part of all MSTs.

→ Cut in a graph is a partition of its vertices in two non empty sets.

Ex:

{A} & {B, C, D} are non-empty sets for the cut (in dotted line).

→ Edge (U, V) with U∈S & V∈T is said to cross the cut.

Ex: A–C & A–B are two edges to cross the cut.

→ So, by Cut property A–C edge will surely be part of all MSTs.

Ex:2

GO CLASSES

PAGE NO.:
DATE: / /

Cut giving two sets $S = \{a, b, d, e\}$
$T = \{c, f, g, h, i\}$

So, edges crossing the cut are: a-b, b-h, b-c, c-d, d-f, f-c

By cut property c-d will always be part of all MSTs bcz it is having min weight among all crossing edges.

⇒ Number of possible cuts: Let's have a graph with n vertices, then no. of cuts will be

$v_1, v_2, v_3 \ldots v_n$

all n have 2 choices (to be in S or in T), so 2^n, but both sets are unlabelled bcz cut $[\{v_1, v_2\}, \{v_3 \ldots v_n\}]$ & $[\{v_3 \ldots v_n\}, \{v_1, v_2\}]$ are identical.

So, $\frac{2^n}{2} = 2^{n-1}$

It will also have one possibility of empty set one side & full at other which is not a cut. So remove that.

# of cuts: $\frac{2^n - 2}{2} = \boxed{2^{n-1} - 1}$ Ans.

⇒ If edges are not distinct then cut property says that, smallest edge crossing any cut is part of atleast one (some) of the MST.

PAGE NO. :
DATE : / /

b) **Cycle Property:** If all edges are distinct, then the largest edge of any cycle of graph is never in any MST.

Proof: Suppose e is the heaviest of cycle & e belongs to a MST (T). Now, consider the above MST w/o e (disconnected) (T-e) with two disconnected components. Since Graph has a cycle, there will be another edge to connect components of {T-e}. Let the edge be f. & since e is heaviest, f is lighter than e. So we got a lower cost MST by deleting e from T & adding f in T., which contradicts our assumption that MST had e. So, by contradiction, e can't be in MST.

Ex:

So, by cycle property a-f cannot be in MST.

⇒ So PTR is that one edge is not in MST iff it is the unique heaviest edge in some cycle.

GO CLASSES

PAGE NO.:
DATE: / /

★ **Finding MST:** There are two algorithms (both are greedy) to find MST given graph G.

MST (G, w):
  A = ∅
  ~~find~~ while A doesn't form ST:
    find an edge (u, v) that is safe:
      A = A ∪ {(u, v)}
  return A;

Here safe means, must be included in MST by cut property.

a). **Kruskal's Algorithm:** Take smallest edge, add it iff not forming a cycle.

Ex:

⇒ MST

Seq: h-g, g-f, i-c, a-b ↔ f-c [e-g → No becz of cycle], c-d, a-h, d-e ✓

PAGE NO. :
DATE : / /

Ex :2

→

MST

Kruskal (G) :

Sort edges in increasing order.

$T = \phi$

for each edge e_i in the sorted order :

if $e_i \cup T$ is not making cycle :

$T = T \cup e_i$;

GO CLASSES

PAGE NO.:
DATE: / /

If time to check cycle is α, then
T.C. of Kruskal = $E\log E + E\alpha$
sorting

Now $\alpha = O(1)$ if we use union find (not req. for Gate) &
$\alpha = O(V+E)$ if we use DFS to check cycle (using backedge)

$\boxed{T.C = O(E\log E)}$ → with union find

$T.C = E\log E + EV + E^2$
↳ w/o union find (with DFS)

Working of union find is not req. for Gate so not writing.

NOTE: If asked T.C. somewhere, then we always need to tell by best possible D.S like $E\log E$ for kruskal's & $V\log V + E$ for Dijkstra (fib. heap + adj. list), etc.

Que: If the edge weights are appropriate & Radix Sort can be applied, then T.C. of kruskal's?

⇒ T.C. = sorting time + $E \cdot \alpha$
$= O(E) + E \cdot (O(1))$
↳ Radix sort ↳ Cycle with union-find

$T.C = O(E)$

PAGE NO. :
DATE : / /

CSE 92

Que: Complexity of kruskal's algo for finding MST of an undirected graph containing n vertices & m edge if the edges are sorted is : ____ ?

⇒ $O(m)$

⇒ Can kruskal's handle negative weights?
→ Yes, -ive weight won't impact anything

⇒ If we add constant to each edge, will MST change?
→ No, as smallest weighted edge will still be smallest.

Que: How many MSTs in cycle graph will all edge weights same.

⇒

, Choose any 1 to remove i.e. ${}^4C_1 = 4$

So ${}^nC_{n-1} = {}^nC_1 = n$ MSTs

Ex: 1

1 MST

Ex: 3

2 MSTs

GO CLASSES

PAGE NO. :
DATE : / /

Ex: 3

⇒ 3*3 = 9 MSTs

* TIF : If an edge forms a bridge, then it is part of every spanning tree.

⇒ True, if we have one cut crossing edge, then we will always have to take it to connect both sets.

Qu: # spanning trees if same weight of all edges?

⇒

⇒ ✓ → Bridge

Choices = 4 × 3 × 3 × 4 = 144 MSTs

GO CLASSES

PAGE NO. :
DATE : / /

Qu: # of MSTs?

a)

Now B has three choices, so 3 MSTs.

with weigh = 1+2+4+5+4 = 16

b)

1 Choice = 1 MST

GO CLASSES

PAGE NO. :
DATE : / /

C

⇒

A has two choices : 2 MSTs.

→ Cayley's Formula gives # ST for complete graph K_n : n^{n-2}

Que : T/F :

S1: Min weight edge is in some MST
→ True.

S2: Min. weigh edge is in All MST
→ False

→ Counter Example

S2 will be True if edge weights are distinct.

GO CLASSES

PAGE NO. :
DATE : / /

b) **Prim's Algorithm:** This uses cut property i.e we cut the graph continuously & choose the min edge from edges crossing cut & continue this.

In this we grow a single tree (unlike kruskal's where we grow forest & finally connect them to make tree). We add shortest edge that could grow our tree.

Ex:

Start with any cut:

Cuts:

{A, B}, other

{A, B}, other

{A, B, H}, other or {A, B, C}, other

{A, B, C, I}, other

{A, B, C, I, F}, other

{A, B, C, I, F, G}, other

{A, B, C, I, F, G, H}, other

{A, B, C, I, F, G, H, D}, other

{A, B, C, I, F, G, H, D, E}, {∅}

(MST)

W = 37

So, we always use cut property here.

GO CLASSES

So, cycle is not possible

bcz at any point of time, we are having n vertices & n-1 edges.

⇒ Do we need to detect cycles in prims?

→ No, cycles are not possible in prims.

Proof: Let's have an intermediate state.

as $y > x$, so we bring in by x.
& $a > x$

Now adding any one vertex in a tree can never bring cycle as we are chosing one edge only. So cycles are not possible

Qu: Each of the figures below repr. a partial spanning tree. Determine whether it could be possible obtained from prematurely stopped prim's, kruskal's, both or neither.

a)

⇒ Kruskal's only as we have taken all min edges & not prims bcz we have not growing tree.

b) c) d)

↳ Skipped.

PAGE NO. :
DATE : / /

⇒ How to implement prim's algorithm?

⇒

⇓ After 1 pass

One thing ~~that~~ we need is finding min edge from all the cross edges, but doing so again & again is not efficient. Bcz. if finding min is taking E time, then we need to do it V times i.e. for all vertices

So, T.C = V·E in this case.

But we can implement this using heap. Heap will be made up of all cut crossing edges & we will chose min all the time.

Find min → Add new Entries → Find min → Add new Entries → ...

This is our sequence of steps. So, we can use heap for this efficiency.

PAGE NO. :
DATE : / /

Before moving further in implementation. Let's compare prims & dijkstra. (Revise Dijkstra before seeing this).

In dijkstra also we were taking min & then ~~removing~~ relaxing outgoing edges by adding its weight from src. But here we have no src, we compare it from whole graph 1 & graph 2.

So, basically we add src. weight also in the path in dijkstra. Here, that is not required.

Ex: Initial state.

then in Dijkstra

& in prim's.

So, that's the only diff. b/w dijstra & prim. Infact Dijkstra is the reinvention of prim's algorithm.

So, Implementation of prim's will be exactly same as that of Dijkstra other then two changes:

PAGE NO. :
DATE : / /

Prims (G,S) :

key[V] = ∞ for all v in V

key[S] = 0

T = ∅

make min heap Q from keys

while Q is non-empty :

u = Extract_Min(Q)

T = T ∪ {u}

for each neighbour v of u & weight w :

if v ∈ Q & d[v] > w //1

d[v] = w //2

parent[v] = u

Only 1 & 2 are diff in dijkstra ; there we had d[u]+w instead of just w.

So, both algorithms takes one vertex at a time & relax outgoing edge , main distinction is prim's choose closest vertex from the tree & dijkstra choses closest vertex from the source.

It's T.C. is exactly same as Dijstra

i.e. Adj List = V·T(remove min) + E·T(update key)

Adj Matrix = V·T(remove min) + E·T(update key) + V^2

i.e. Best is $V \log V + E$

GO CLASSES

PAGE NO. :
DATE : / /

Que: Apply Prim's & Dijkstra on Graph:

a). Dijkstra.

b). Prim's

GO CLASSES

Que: If we have MST, then do we have shortest path in same MST?

⇒ No, no sense in relating both.

Ex:

shortest path is A —7— C, but MST is A —5— B —5— C.

So, MST need not give shortest path.

NOTE: Prim's & Kruskal's give the same MSTs always if we have distinct edge weights.

PAGE NO. :
DATE : / /

Qu: Draw MST of following graph using prim's algo & fill the table.

Order vertices A, D, F, B, C, G, E.

Vertex	Distance	Path
A		-
B	2	A
C	15	B
D	5	A
E	2	G
F	7	A
G	3	C

Qu: In this graph, list all edges that must belong to MST regardless of what their value can be:

GO CLASSES

PAGE NO.:
DATE: / /

⇒ a) G-H bcz it is a bridge
b) B-D, if we cut {D} & V-{D}, then by cut property
c) B-E, if we cut {A,B,D} & V-{A,B,D}, " "

Que: Consider an undirected graph where all edges have weight at least 1. If there are exactly two edge with weight 1, then any MST includes both of them?

⇒ True, as kruska's will always select both. (False if 3 were of 1)

Que: Given a graph G, let (u,v) be an edge with min. cost edge b/w $S \subseteq V$ & $V-S$. Then G must includ the edge. (Edges are distinct)

⇒ True, Cut property Definition.

Que: Consider an undirected graph G with unique positive weights. Suppose it has a MST. If we square all the edges of graph. Do we still get same MST

⇒ True, bcz order of weights of edges will still be same. (False, if we don't had positive weights).

CSE 2000

NOTE → Every MST will contain min. weight edge if edges are distinctly weighted.
→ If edge with max. weight is in MST, then its removal must disconnect graph, if edges are distinctly weighted. i.e. E_{max} is bridge.
→ G has a unique MST if weights are distinct.

PAGE NO. :
DATE : / /

Que: If G has a cycle & there is unique edge e which has the min. weight on this cycle, then e must be part of every MST.

⇒ **False.** Not necessarily bcz if e is max of another cycle then e won't be part of any MST, by cycle property.

Ex:

Que: If the edge e is not part of any MST of G, then it must be the max. weight edge on some cycle in G.

⇒ **True**, cycle property.
↳ if not max. weight then we could take other.

Que: If edge weights are +ive, then shortest path b/w two vertices must be part of some MST.

⇒ **false**, Already proved.

Que: Assume we have two graphs G_1 & G_2. Also assume that we have T_1 which is a MST of G_1 & T_2 which is MST of G_2. Now consider a new Graph $G=(V,E)$ such that $V = V_1 \cup V_2$ & $E = E_1 \cup E_2 \cup E_3$ where E_3 is set of all edges that cross the cut (V_1, V_2).

Now if we want to find MST of new graph then can we do:

PAGE NO. :
DATE : / /

a) e_{min} = minimum edge from E_3

b) $T = T_1 + T_2 + e_{min}$

Do this always work?

⇒ Not Necessarily, this will work. (False)

Ex:

Algo will give a-b, c-d, a-c, (10+10+1)

but in actual mst is a-c, c-b, b-d. (1+2+3)

GO CLASSES

PAGE NO. :
DATE : / /

★ Finding missing edge weights given MST:

Case I: weight of edge in MST (x), then answer given in form of max. value of edge, so that it is in MST. ($x \le a$)

Case II: weight of edge not in MST (x), then answer given in form of min. value of edge such that it is not in MST. ($x \ge a$)

The one only way to solve the above problems is via cycle & cut property.

Ques: For the below graph & it's MST, what can be ranged of x?

a)

cut property on E will not work here.

⇒ In this, try to add other vertices not in MST & try to apply cycle theorem. (one by one, not even two together).

Cut property - min of cut is must present.
Cycle property - max of cycle is must absent.

PAGE NO.:
DATE: / /

→ If we add E-F edge we cannot comment anything on value of x
→ If we add C-D edge we can say $x \le 9$, bcz if it was more than 9 then C-D was present in MST instead of x.

→ By D-F, edge $x \le 20$ (Redundant)

So $x \le 9$

b).

x is not in MST.
If we add it in MST, we can say $x \ge 2$.

→ So the crux is that add all non mst edges one by one & then using cycle property, write inequality.

PAGE NO. :
DATE : / /

Que: In below graph, the edge weights of only those edges which are in MST are given. What are the min weight of other edges.

⇒

Min Value: $z = 6$, $x = 9$, $y = 15$ Ans

DIY: TIFR 2014.

★ **Modification of Edges:**

There are 4 possibilities:

a) An edge is in MST & we reduce its weight: Then no change in the MST.

b) An edge is not in MST & we increase its weight: Then no change in the MST.

c) An edge is not in MST & we reduce its weight: MST may change here.

PAGE NO.:
DATE: / /

→ Add this edge to MST. Now we have exactly one cycle. Now, based on cycle property we can decide if still reduced edge is heaviest like before modification or some other edge is heaviest. Now, based on this we can decide new MST using dfs or bfs in $O(n)$

d) An edge is in MST if we increase its weight MST may change.

→ Remove the modified edge from MST. Now we'll have 2 connected components. We can now find the lightest edge that connects both components, i.e. by cut property. Can be done in $O(n)$.

GO CLASSES

GATE CSE 2024

Ques: Let $G=(V,E)$ be a weighted undirected graph & T be a MST of G maintained using adjacency list. Suppose a new edge (u,v) is added to G. The worst case T.C. of determining if T is still an MST is:

⇒ $\Theta(V)$.

We can check it in $\Theta(V+E)$, In MST $E = V-1$

So $\Theta(V+V+1) = \Theta(V)$

CS 2016

Ques: T/F?

a) If e is lightest of cycle, then every MST have e.

⇒ False, e can be heaviest in other cycle.

b) If e is heaviest of cycle, then every MST exclude e.

⇒ True, cycle property.

PAGE NO. :
DATE : / /

9) If e is not part of any MST, then it must be heaviest edge on some cycle of Graph.

→ True

★ **Huffman Encoding:** Another Greedy Algorithm.

Problem was:

I/P: some distribution of characters (frequencies)

O/P: way to encode characters as efficiently as possible.

Ex:

Char.	freq.
a	45 times
b	13 "
c	12 "
d	16 "
e	9 "
f	5 "

Approach 1: Use fixed length code.
↳ working but less efficient

Total Bits = $(45+13+12+16+9+5) * 3$ bits

as 3 are req. for 6 diff. chars.

Approach 2: Variable length code (shorter for common character)

Definitely lesser space than attempt 1, but sometimes decoding may create ambiguous O/Ps which was not an issue in attempt 1 due to fixed length.

GO CLASSES

PAGE NO. :
DATE : / /

Ex: D = 11 & F = 100, E = 111, A = 00

DF $\xrightarrow{\text{Encode}}$ 11100 $\xrightarrow{\text{Decode}}$ EA
$\longrightarrow$ DF } Ambiguity.

So, we cannot just assign codes randomly based on criteria that higher frequency gets lower bits. (greedy approach)

Attempt 3: Use prefix free codes (Huffman Encoding)

⇒ Prefix free Code: If any code is not prefix of another code then code is prefix free.

Ex: 0, 10, 010, etc (all three are prefix free).
1, 01, 001, 0001, 000, etc.

0, 1, 00, 11 is not prefix free.

Que: T/F

a) Every code word in prefix (free not req. to write) code must have same length.
→ False.

b) If all the codes in a prefix code are reversed, the result is still prefix code.
→ False
↳ Counter Ex: 0 → 0
10 → 01
prefix code ⟶ Reverse
↳ Not prefix code.

PAGE NO. :
DATE : / /

→ Huffman was student @ MIT & solved this problem as assignment of optimal code.

Huffman Coding: It says greedily build subtrees by merging starting with the 2 most infrequent letters.

Ex: A:45, B:13, C:02, D:16, E:9, F:5

Now assign 0 & 1 anything but same always to any side.

A:0, D:10, B:110, C:1101, E:1110, F:1111.

Que: Determine huffman code for the alphabets $a_1 \ldots a_5$ with symbol probabilities $P_1 = 0.11$, $P_2 = 0.05$, $P_3 = 0.10$, $P_4 = 0.70$ & $P_5 = 0.04$.

→ Most probable to least probable : a_4, a_1, a_3, a_2, a_5

PAGE NO. :
DATE : / /

$a_4 : 0$, $a_1 : 10$, $a_3 : 110$, $a_2 : 1110$, $a_4 : 1111$

Note: Huffman codes are always prefix free codes. This is bcz all char. are ~~below~~ at leaf. Nothing can be below it.

```
Huffman (C) :
    n = |C| ;
    Q = C ;                          // Min-Heap
    for i=1 to n-1 ;                 // n times.
        x = Extractmin (Q)           // log n
        y = Extract min (Q)          // log n
        z = x + y
        Insert (Q, z)                // log n
```

T.C = O(nlogn) if min heap used.

Qu: which of the following huffman codes are invalid:

a) 0, 11, 101, 100

PAGE NO. :
DATE : / /

So Valid

b): 1, 01, 00, 010

→ Not Valid as not even prefix code.

c) 00, 01, 110, 111

→ Prefix code, but not optimal so, Invalid as huffman can't give this.

Que: Which of three are invalid huffman trees and why?

a)

→ Invalid as no Right child of 17.

PAGE NO. :
DATE : / /

b)

→ Invalid bcz, 8 & 12 would have gone first.

Que: Under a huffman encoding of 'n' symbols with frequencies $f_1, f_2 \ldots f_n$ what is the longest a codeword could possibly be? Give an ex.

→ n-1.

Ex:

$f_n : n,\ f_{n-1} : n,\ f_{n-2} : 2n,\ f_{n-3} : 4n \ldots$

Que: Consider following 5 chars & their frequencies:

$a_1 : 0.4,\ a_2 : 0.2,\ a_3 : 0.2,\ a_4 : 0.1,\ a_5 : 0.1$

a) Draw all possible huffman trees:

GO CLASSES

PAGE NO. :
DATE : / /

Avg. length in all of the cases is same i.e 2.2

Ex: for T_1
$(1*0.4)+(2*0.2)+(3*0.2)+(4*0.1)+(4*0.1) = 2.2 = E(X)$
Same for T_2 & T_3

$Var = E(X^2) - (E(X))^2$
$= E(X^2) - (2.2)^2$
Solve it. , ~~best~~ min var is for T_4 as lengths are
3, 3, 2, 2, 2. (0.16)
↳ Min.

→ for balanced tree we will have minimum variance.

GO CLASSES

PAGE NO. :
DATE : / /

★ Optimal Merge Pattern: Similar to huffman encoding

Here we consider merge cost of two files with size m & n is m+n.

I/P: A set of files of various lengths.
O/P: An optimal sequence of two way merges to obtain a single file

Ex: A: 30 , B: 20 , C: 10

Various merges can be.

A, B, C = 50 + 60 = 110 (A+B = 50 & AB+C = 50+10 = 60)
A, C, B = 40 + 60 = 100
B, A, C = 50 + 60 = 110
B, C, A = 30 + 60 = 90
C, A, B = 40 + 60 = 100
C, B, A = 30 + 60 = 90

Total = 60 + 30 = 90

Distance from leaf node to root = no. of times a file is involved in merging

we want large files to involve less = large files should have less distance.

So, do it similar to huffman encoding.

GO CLASSES

PAGE NO. :
DATE : / /

Gate 99

Que: Min. no. of record movement required to merge five files:
A: 10, B: 20, C: 15, D: 5, E: 25

⇒

Total = 45 + 75 + 30 + 15
= 165 (A)

★ **Interval Scheduling Problem:** Also known as activity selection problem.

I/P: Set of intervals
O/P: Subset of non-overlapping intervals of minimum size.
↳ max size of subset

Ex: Meetings scheduling of VP of a company.

Attempt 1: Pick intervals of min size.
May not work always

Ex: I/P:

works here

Counter Ex:

Not working here

GO CLASSES

PAGE NO. :
DATE : / /

Attempt2: Pick an interval that overlap with min. no. of intervals.

Ex:

Counter Ex:

Not working in this case.

This works in various cases but do not always work. So, not working.

Attempts: (Optimal) Be greedy ie pick the interval with earliest finish time.

Ex:

So this algorithm is giving optimal answer always.

GO CLASSES

PAGE NO. :
DATE : / /

There can be many optimal answers, there can be two cases:

Case 1: Optimal answer is same as algo 3.

Case 2: If diff then:
No. of intervals in optimal = No. of intervals in Algo 3.

(Not proving, why this works).

Que: Does below algo works?

a) Choose the interval that ends last, discards the overlapping intervals & recurse.

⇒ No.

counter ex.

Optimal ans

Actual ans by this algo

b) Choose the interval that starts first, discard all overlapping intervals & recurse.

⇒ No

Counter Ex.

*c) Choose the interval that starts last, discard the overlapping intervals & recurse.

⇒ Yes.

Always works

GO CLASSES

This greedy strategy works. Infact, this is just a time reversed version of the greedy algorithm (attempt 3) we studied.

There are many other variants of the same problem.

★ **Job Scheduling with Deadlines:** In this

→ the resource can be used by atmost one person at a time
→ we can accept only overlap-free request.
→ every request has deadline, profit & a unit length.
→ It has to be done before deadline.
→ Goal is to find a feasible schedule with max. profit

Algo: Greedy approach works here. Choose the most profitable job & schedule it just before its deadline. Do this for all.

Ex:

Job	J_1	J_2	J_3	J_4	J_5	J_6
Deadline	5	3	3	2	4	2
Profit	200	180	190	300	120	100

#4	#1	#3	#5	#2
J_2	J_4	J_3	J_6	J_1

1 2 3 4 5

Total profit = 940 (100 + 180 + 190 + 300 + 120)

PAGE NO. :
DATE : / /

T.C = $O(n \log n)$ + (sort profit wise)
$O(n)$ (Find max deadline & array initialise)
$O(n^2)$ (Start from right of array & search for nearest slot near deadline, done for all jobs)

$= O(n^2)$

☆ **Fractional Knapsack:** Goal is to choose items so that total profit is maximized. Given bag capacity, items & their weights.
Here we need to be greedy. We can be greedy of profit or weight, but it will not work. We need to be greedy for profit per unit of weight.

Attempt 1: Choose highest profit first.
↳ will not work.

Attempt 2: Choose lowest weight first
↳ will not work

(optimal)
Attempt 3: Balancing weight & profit (P/w)
i.e. go in decreasing order of P/w.

GO CLASSES

PAGE NO. :
DATE : / /

Ex:

	P	w	P/w
1	12	4	3
2	32	8	4
3	40	2	20
4	30	6	5
5	50	1	50

Capacity = 10

⇒ fraction array:

1	2	3	4	5
1	1/8	1	1	1

~~10 9 7 10~~

Profit = $50+20+5+\frac{4}{8}\times 1$

⇒ 75.5

Algo:

Sort in dec. order of p_i/w_i ||(nlogn)

Initialise fraction[i]=0;

for each item i in sorted order: || O(n)

if w[i] <= capacity:

fraction[i] = 1;

capacity = capacity - w[i];

else break;

if i<n: || that little part

fraction[i] = capacity / w[i];

T.C = O(nlogn) + O(n)

= O(nlogn)

GO CLASSES

PAGE NO. :
DATE : / /

→ We can solve this problem in O(n) time by avoiding sorting & using median of p_i/w_i & then deciding on the basis of median & capacity & all. (Not req. for gate)

Problems solved by Greedy Algorithms:

a) Dijkstra
* b) Bellman-Ford (Not Greedy)
c) Kruskal's MST
d) Prim's MST
e) Huffman Coding
f) Optimal Merge Pattern
g) Interval scheduling
h) Unit Job scheduling based on deadlines
i). Fractional Knapsack.

GO CLASSES

PAGE NO. :
DATE : / /

Module-6

★ Dynamic Programming

This is another algorithm design paradigm.

* Fibonacci Numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Recursive Eq: $f[i] = f[i-1] + f[i-2]$

$f(0) = 1$

$f(1) = 1$

This can be easily converted into code:

```
int fib(n):
    if (n==0 or n==1) return 1;        //base case
    return fib(n-1) + fib(n-2);        // Recursion
```

So one thing very clearly, we can ~~very easily~~ say is that, very easily we can write code given recursive equation. So our goal here will be to get the recursive equation.

No. of Rec. calls:

Dynamic Programming ≡ Careful Bruteforce

PAGE NO. :
DATE : / /

So at lower bound it will go n/2 times on Right side of tree.

$1 + 2 + 2^2 + 2^3 + 2^4 + \dots + 2^{(n/2 - 1)}$

$O(2^{n/2})$ → Lower Bound. (Last term as ↑ GP)

$1 + 2 + 2^2 + \dots + 2^{n-1}$
$= O(2^n)$ → Upper bound

So we can say $O(2^n)$ recursive calls are there, so it is taking exponential time.

But, unique recursive calls here are only $O(n)$, i.e. $f(n), f(n-1) \dots f(1), f(0)$

Unique are $O(n)$ & total are $O(2^n)$.

So, we can explore carefully by calling each unique only & storing the result we already have in some array.

If Total Rec. Calls >>> Unique Rec. calls, then we can apply dynamic programming.

dp[] is array
↳ Many call it cache.

```
fib(n):
    if (dp[n] != null)      return dp[n];      //called guard
    if (n==0 or n==1)  dp[n]=1;
    else    dp[n] = fib(n-1)+fib(n-2);
    return dp[n];
```

First store, then Return.

PAGE NO.:
DATE: / /

The approach we just wrote is also known as top down approach when we use recursion

Another approach known as bottom up approach can be written as:

```
fib(n):
    dp[0] = dp[1] = 1
    for(i=2 to n):
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
```

Top down approach is easier to code as compared to bottom up in most of the cases (may not feel here).

Bottom up approach is easier to work upon like solving problem by hand in most of the cases.

Top down approach can be more efficient as it fills the array on the basis of values it require, whereas bottom up fills all the values in the array.

Top down approach is recursive & bottom up approach is iterative.

D.P. is called dynamic bcz its founder Richard Bellman worked on it during job thru in RAND corp. & to bluff his company he called it D.P.

He used DP as umbrella for his activities.

GO CLASSES

PAGE NO. :
DATE : / /

★ **Climbing Stairs:** We have staircase with 'n' stairs. We can climb either 1 or 2 stairs at a time. How many ways to climb to the top?

Let $CS(n)$ repr. ways to climb n stairs

$CS(0) = 1$ (No step)

$CS(1) = 1$

$CS(2) = 2$ $[(1,1), (2)]$

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

Now we can easily apply DP on this problem.
↳ similar to fibonacci.

Que: Suppose you have a row of coins with values that are +ive integers $c_1, c_2, \ldots, c_n$.

There values need not be distinct. Our task is to pick up coins that have as much values as possible, subject to the constraint that you don't even pick up two coins that lie beside each other.

for Ex: Given C_1 to C_6 as: $(5, 1, 9, 10, 6, 2)$ (1 2 3 4 5 6). Then max value is 17 using (C_1, C_4, C_6) (5 10 2).

Let $f(n)$ be max total value with n coins with $f(0) = 0$ & $f(1) = c_1$. Rec: relation for $f(n)$ is?

$f(n) = \max(c_n + f(n-2), f(n-1))$

i.e. Choose c_n, then $c_n + f(n-2)$ or
don't choose c_n, then $f(n-1)$
Max of both.

→ Just find Rec. Equation carefully, everything else will be done very easily.

PAGE NO.:
DATE: / /

Que: You are given an integer array, cost where cost[i] is the cost of ith step on a staircase. Once cost paid we can either climb one or two steps.

We can either start from step with index 0 or with index 1.

Rec. Rltn. for the minimum cost to reach the top.

Ex: I/P: cost = [10, 15, 20]

O/P: 15 (Start at idx 1, pay 15, climb 2 steps)

I/P: [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]

O/P: 6

⇒ ans[n] = min. cost to reach n^{th} level

$$\boxed{\begin{array}{l} ans[0] = 10 \\ ans[1] = \min(ans[0], ans[1]) \end{array}}$$

Rec. Rtn.
$$\begin{cases} ans[n] = \min(ans[n-1] + cost[n], ans[n-2] + cost[n]) \\ ans[0] = cost[0]; \\ ans[1] = cost[1]; \end{cases}$$

Que: Consider two teams Ants & Bees playing a series of games until one of the team wins 'n' games.

Assume prob. of ants winning a game is same for each game i.e. p & losing a game is $q = 1-p$. Hence no ties.

Let ants needs i more games to win the series & Bees needs j more games to win the series then P(i, j) gives the prob. of ants winning the series.

PAGE NO.:
DATE: / /

Rec. Relation of P(i,j) is:

⇒ $P(i,j) = p * P(i-1,j) + q * P(i,j-1)$

This is problem where we will be requiring 2D array to store answers solved using DP.

```
fun(i,j):
    if (dp[i][j] != null) return dp[i][j];
    base cases;
    dp[i][j] = ____ //rec. relation
    return dp[i][j];
```

Something like this in top down.

⇒ In bottom up approach we need to decide the order to fill the table.

Ex: $A[i,j] = \min(A[i,j-1], A[i-1,j-1], A[i-1,j+1])$

Here value depends on:

We can fill it row wise only (Not col wise) if doing bottom up.

In top down, we don't care, rec. automatically does it but in bottom up we need to fill it via code.

GO CLASSES

PAGE NO. :
DATE : / /

b). $A(i,j) = \max(A(\min(i,j)-1, \min(i,j)-1), A(\max(i,j)-1, \max(i,j)-1))$

⇒

We can go row wise or column wise here.

c) $A(i,j) = \max(A(i-2, j-2), A(i+2, j+2))$

⇒

Impossible, as cyclic. (Neither row wise nor col. wise)

★ Sometimes top down may save our time, however space complexity remains same.

★ DP ≡ Careful Recursion.

fib w/o D.P = $O(2^n)$
fib with D.P = $O(n)$

So, D.P. saves too much time as solves only unique problems.

GO CLASSES

PAGE NO. :
DATE : / /

★ Elements of Dynamic Programming:

a) Solving smaller subproblems of same kind (Optimal Substructure)
b). Check if same problem is saved several times.
(Overlapping Subproblems)

Now we will see various standard problems that are being solved by DP.

★★ Longest Common Subsequence (LCS):

⇒ Subsequence: A sequence Z is a subsequence of X, if Z can be obtained from X by dropping symbols.

Ex: BDFH is a subseq. of ABCDEFGH
C " " " " "
ABCDEFGH " " " " "
DB is not a " " " " , but BD is.

Que: What are common subseq. of X & Y:
X: ABCDEFGH
Y: ABDFGHI

→ Many common seq. like ABDF, A, B, H, ABDG, etc.
L.C.S. is ABDFGH

GO CLASSES

PAGE NO.:
DATE: / /

I/P: Seq. X & Y.

O/P: Length of L.C.S. & L.C.S

Attempt-1 Naive Algo.

Write all subseq. of X.

Pick one by one & check if it is also subseq. of Y.

If $|X| = m$ & $|Y| = n$,

then possible subseq. of $X = 2^m$. (as each char. have 2 choice)

T.C = $2^m * n$

Checking if subseq. of X is also subseq. of Y is easy, can be done as:

NFA for subseq. of X i.e $c_1 c_2 \ldots c_n$.

Attempt 2: Using Dynamic Programming.

Let lcs[i,j] be length of LCS of X[1...i] & Y[1...j]

i.e. lcs[m,n] represents the answer.

give X[1...m] & Y[1...n]

Then rec. eq. can be given as

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i == y_j \text{ (case 1)} \\ \max \begin{cases} LCS[i-1, j] \\ LCS[i, j-1] \end{cases} & \text{if } x_i \neq y_j \text{ (case 2)} \end{cases}$$

PAGE NO.:
DATE: / /

Case 1: When last char matches

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$

$LCS[4,3] = 1 + LCS[3,2]$

Case 2: When last char are diff.

$X = \langle A, B, D, C \rangle$

$Y = \langle Z, B, E \rangle$

$LCS[4,3] = \max(LCS[3,3], LCS[4,2])$

Note that in Case 2, we can't dec. both of them as match can be found later.

Base Cases:

$LCS[0,j] = 0$ for any j

$LCS[i,0] = 0$ for any i

→ How can we fill table in bottom up?

Both rowwise & column wise will work.

GO CLASSES

PAGE NO.:
DATE: / /

We can easily write code w/o DP as we already have basecases & rec. equations.

Code with DP: **Top Down**

```
LCS(i,j):
    if (dp[i][j] != null)    return dp[i][j];
    if (i==0 or j==0)    dp[i][j] = 0;
    else if (x[i] == y[j]):
        dp[i-1][j-1] = LCS(i-1, j-1);
        dp[i][j] = 1 + dp[i-1][j-1];
    else if (x[i] != y[j]):
        dp[i-1][j] = LCS(i-1, j);
        dp[i][j-1] = LCS(i, j-1);
        dp[i][j] = max(dp[i-1][j], dp[j-1][i]);
    return dp[i][j];
```

↳ Can't solve by hand from this.

Bottom Up (will not write in future)

```
LCS(x,y):
    dp[i,0] = dp[0,j] = 0 for all i & j;
    for i=1 to m:
        for j=1 to n:
            if x[i] == y[j]:
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    return dp[m][n];
```

↳ Ans

PAGE NO. :
DATE : / /

Recursive calls w/o DP

It's a binary tree with m or n levels.

So. T.C = 2^m as no. of nodes = 2^m.

So exponential no. of calls are there

unique calls:

$m \times n$ unique calls only

$O(m*n)$

So as $2^m >>> mn$, DP will work efficiently.

T.C. = $O(m*n)$ $O(mn)$

GO CLASSES

PAGE NO.:
DATE: / /

Note that T.C. in case of DP = (no. of cells in array or array size) * (work per element)

⇒ The LCS can also be done by comparison on 1st element instead of the last always. But it is not considered convenient bcz we will be filling table from last. However same T.C. and S.C.

Rec. eq will be:

$$LCS[i,j] = \begin{cases} 1 + LCS(X[i+1..m], Y[j+1..n]) & \text{if } x_i == y_j \\ \max \begin{cases} LCS(X[i+1..m], Y[j..m]) \\ LCS(X[i..m], Y[j+1..n]) \end{cases} & \text{if } x_i \neq y_j \end{cases}$$

Here LCS[1,1] will give ans.

where LCS[i,j] means LCS from X[i..m] & Y[j..n].

Solving by top down:

X = ABCBDAB

Y = BDCABA

		1	2	3	4	5	6
Base cases		B	D	C	A	B	A
0	0	0	0	0	0	0	0
1 A	0	0	0	0	1	1	1
2 B	0	1	1	1	1	2	2
3 C	0	1	1	2	2	2	2
4 B	0	1	1	2	2	3	3
5 D	0	1	2	2	2	3	3
6 A	0	1	2	2	3	3	4
7 B	0	1	2	2	3	4	4

This element means LCS[6,2]

This is answer

LCS b/w X[1..6] & Y[1..2] only

PAGE NO. :
DATE : / /

So LCS will be of size 4. & we can trace the path for LCS. There can be many LCS as there are many paths.

Ex: BDAB, BCBA, BCAB.

Two paths can also give same LCS.

DIY 2014 Gate.

→ There are many applications of ~~DPA~~ LCS, not listing them.

★ **Matrix Chain Multiplication (MCM):**

$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3)$

Final Matrix after multiplication will be same but no. of computations may differ.

NOTE ⟷ # Multiplications required to multiply matrices of $A_{p \times q}$ & $B_{q \times r}$ are $p \times q \times r$

Ex:

Multiplication per element = ~~p×r~~ q

elements = p×r

multiplications = p×q×r.

GO CLASSES

PAGE NO. :
DATE : / /

Qui: Multiply following matrices & cost of multiplication for diff order of ABC & what is the least cost?

A : 10×5

B : 5×10

C : 10×5

ABC

(AB)C — (10×5×10) + (10×10×5) = 1000

A(BC) — (5×10×5) + (10×5×5) = 500

So A(BC) will give minimum cost.

⇒ **Problem Stmt:**

I/P: A sequence of matrices $A_1, A_2, A_3 \ldots, A_n$.

O/P: full parantheization for product of $A_1, A_2 \ldots A_n$ that minimize # of scalar multiplications.

ABCD

A|BCD — (A((BC)(D))) ((A)(B)(CD))

AB|CD — (AB)(CD)

ABC|D — ((AB)(C)(D)) ((A)(BC)(D))

GO CLASSES

PAGE NO. :
DATE : / /

We have approximately 2^n leaves (exponential), However every leaf gives us unique way to multiply. Then where we can save time if every leaf is unique? We don't need to go till leaf for all parannesiations we can stop in middle like we may have answer of ABC, then don't do again & again.

ways to put bracket = # leaves in tree

To multiply 'n' matrices we require 'n-1' parenthesis

ways to multiply 'n' matrices = # ways to put 'n-1' para.

≡ Stack permutation ≡ Catalan No.

for n matrices = Catalan(n-1) = $\frac{(2(n-1))!}{(n!)(n-1)!} \neq C_n$.
↳ bcz n-1 parenthesis

& we know that $n! > 2^n$
So T.C. of MCM w/o DP = $O(n!)$ which is huge & much more than exponential.

⇒ <u>Recursive Equation for MCM:</u>

Let $m[i,j]$ represent min cost to multiply matrices $A_i \ldots A_j$, where matrix A_k has dimension $p_{k-1} \times p_k$

So, our final answer will be $m[1,n]$.

PAGE NO.:
DATE: / /

$A_i \ldots A_j$

$A_i \mid A_{i+1} \ldots A_j$ $\quad$ $A_i A_{i+1} \mid A_{i+2} \ldots A_j$ $\quad$ - - - - - - $\quad$ $A_i \ldots A_{j-1} \mid A_j$

So, $m[i,j] = \min_{k=i}^{j-1} (A_i \ldots A_k)(A_{k+1} \ldots A_j)$

$= \min_{k=i}^{j-1} (m[i,k] + m[k+1,j] + (P_{i-1} \times P_k \times P_j))$

$(A_i A_{i+1} \ldots A_k)(A_{k+1} \ldots A_j)$

$P_{i-1} \times P_i$ $\quad$ $P_{k-1} \times P_k$ $\quad$ $P_k \times P_{k+1}$ $\quad$ $P_{j-1} \times P_j$

So, cost to multiply $= P_{i-1} \times P_k \times P_j$

$m[i,j] = 0$ if $i=j$ bcz no cost to multiply single matrix

So,

$$m[i,j] = \begin{cases} 0 & \text{if } i=j, \\ \min_{k=i}^{j-1} (m[i,k] + m[k+1,j] + (P_{i-1} \cdot P_k \cdot P_j)) & \text{if } i<j \end{cases}$$

$T.C = m[i,j] = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$

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

Unique problems are C_{n-1} here, but we do not solve each of them.

GO CLASSES

PAGE NO. :
DATE : / /

Let $n=4$, $A_1 A_2 A_3 A_4$

then we have

1, four length subproblem $\{(A_1 A_2 A_3 A_4)\}$
2, 3 length " $\{(A_1 A_2 A_3)(A_2 A_3 A_4)\}$
3, 2 length " $\{(A_1 A_2), (A_2 A_3)(A_3 A_4)\}$
4, 1 length " $\{(A_1)(A_2)(A_3)(A_4)\}$

↳ Base Case 0 multiplic—

No mul. if ~~zero~~ only 1 matrix to multiply.

So, n of 1 length, (n-1) of 2 length, (n-2) of 3 length ... 1 of n length.

Top Down Code is:

```
MCM(i,j):
    if (m[i][j] != null) return m[i][j];

    if (i==j);
        m[i][j]=0;

    else:
        min = INT_MAX;
        for k = i to j-1
            q = MCM(i,k) + MCM(k+1,j) + P_{i-1} · P_k · P_j
            if (q<min):
                min=q;
        m[i][j] = min;
    return m[i][j];
```

GO CLASSES

Just understand T.C. (or remember), a m... solving problem

PAGE NO.:
DATE: / /

T.C using D.P:

$\frac{n^2}{2}$ cells in Array

work per cell is n approx

$T.C = \frac{n^2}{2} \times n = O(n^3)$

Exact T.C $= \sum_{i=1}^{n} i \cdot (n-i)$

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

GO CLASSES

Ex: $A_1: 2\times3$, $A_2: 3\times1$, $A_3: 1\times5$, $A_4: 5\times4$

1,1: 0	2,2: 0	3,3: 0	4,4: 0 → Base cases
1,2: 6	2,3: 15	3,4: 20	
1,3: 16	2,4: 27 → ③		
1,4: 34 → ②			

Min. of 1,1, 1,2, 2+ $2\times3\times1$

Means partition at 2 ← ② ←

Cell repr min mul cost of $A_2 - A_4$

$A_1 A_2 A_3$ 16

min

45 $A_1 | A_2A_3$ $A_1A_2 | A_3$ ← 16

$0+15+(2\cdot3\cdot5)$ $(6+0+(2\times1\times5))$

$A_1A_2A_3A_4$

$A_1 | A_2A_3A_4$ | 34 $A_1A_2A_3 | A_4$

$(0+27+(2\cdot3\cdot4))$ $(6+20+(2\cdot1\cdot4))$ 56

$27+24$ $26+8$

$\Rightarrow 51$ $\Rightarrow 34$

So ans is

$((A_1A_2)(A_3A_4))$

with 34 multiplications

PAGE NO. :
DATE : / /

CSE 2016

Que: $A_1 : 10\times5$, $A_2 : 5\times20$, $A_3 : 20\times10$, $A_4 : 10\times5$

0	0	0	0
1000	1000	1000	
1500 (1,3)	1250 (2,4)		
1500 (1,4)			

① ③ ①

$A_1 A_2 A_3$

$A_1A_2 | A_3$: $(10\times5\times20) + 1000 + 0 = 1000$

$A_1 | A_2A_3$: $(10\times5\times10) + 1000 + 0 = 1500$

$A_2 A_3 A_4$

$A_2 | A_3A_4$: $(5\times20\times5) + 0 + 1000 = 1500$

$A_2A_3 | A_4$: $(5\times10\times5) + 0 + 1000 = 1250$

$A_1A_2A_3A_4$

$A_1 | A_2A_3A_4$: $(10\times5\times5) + 0 + 1250 \Rightarrow 1500$

$A_1A_2 | A_3A_4$: $(10\times20\times5) + 1000 + 1000 \Rightarrow 3000$

$A_1A_2A_3 | A_4$: $(10\times10\times5) + 1500 + 0 \rightarrow 2000$

$\Rightarrow$ So, ans is $(A_1)((A_2A_3)(A_4))$ with 1500 multiplications.

DIY Gate CSE 2011

That's all about MCM T.C. is $O(n^3)$ & unique leaves are C_{n-1}. We are saving time in p/w.

However solving each leave but, we have partial solutions. So saving time there.

GO CLASSES

PAGE NO.:
DATE: / /

★ 0-1 knapsack problem:

I/P: Items with profit & weight & bag capacity

O/P: List of items such that max. profit & items can't be taken in fractions.

As I already know we cannot solve this problem greedily like fractional knapsack. (So not writing that failed attempt).

Let KS[j, x] = optimal value we can fit in knapsack of capacity x with items 1 to j.

Answer is KS[n, capacity]. i.e. max. with bag capacity & all items.

Rec. Rltn is:

$$KS[i, c] = \max \begin{cases} KS(i-1, c-w_i) + P_i & \rightarrow \text{if take } i^{th} \text{ item} \\ KS(i-1, c) & \rightarrow \text{if don't take } i^{th} \text{ item.} \end{cases}$$ **

KS(i, 0) is 0 & KS(<>, c) is 0 → Base cases

when capacity is zero

→ when items are zero

Ex: KS(<x1, ... xi>, cap.)

take xi — don't take xi

P_i + KS(<x1, .. xi-1>, cap-wi) | KS(<x1, .. xi-1>, cap.)

max of both

& so on done recursively.

PAGE NO. :
DATE : / /

Ex:

Item	Weight	Profit
x_1	1	2
x_2	2	2
x_3	3	5.

Capacity = 5

Then by ~~both~~ top down approach:

So ans is 7,

we take x_3, & then either x_2 or x_1

(trace the path)

GO CLASSES

PAGE NO. :
DATE : / /

How many total calls are possible?

⇒ for n items $O(2^n)$ are possible (2 choices each item)

How many unique calls are they?

⇒ $n*c$ i.e. $O(nc)$

so $n*c$

So D.P. is working greatly here.

Also size of table = $n*c$

& work per element = $O(1)$

So T.C = $O(nc)$

Que: what will be the max profit obtained using 0-1 knapsack with capacity of 5?

Item	weight	profit
x_1	3	2
x_2	4	3
x_3	5	4
x_4	6	5.

PAGE NO. :
DATE : / /

Let's solve by bottom up:

Capacity

Bye Case	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	4
4	0	0	0	2	3	4

item

Cell Repr. max profit with 2 items & capacity 4.

Answer

4 profit & path can be tracked for items' list.

$$KS(i,w) = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ KS(i-1,w) & \text{if } w_i > w \\ \max\{KS(i-1,w),\ KS(i-1,w-w_i)+p_i\} & \end{cases}$$

i items & w capacity

T.C. $O(n \cdot w)$

* w can be as large as 2^n, so $O(nw) \equiv O(n \cdot 2^n)$ is not actually polynomial in i/p size. We call these algorithms "pseudo polynomial".

finding a polynomial time algo for knapsack is an open problem.

GO CLASSES

PAGE NO.:
DATE: / /

→ Here topdown can save time as we need not fill all table entries

→ Also we can go from 1st element as told in LCS, but traditionally we go with last as it is more convienient to fill table in that way.

★ **Subset Sum problem:**

Maximum subset sum:

I/P: Given a set of integers & an integer w.

O/P: Find a subset with max. sum w/o exceeding w.

Let MSS(i,w) represent max. subset sum with 1,2 .. i^{th} elements & not exceeding w.

Then rec. relation will be:

$$MSS(i,w) = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ MSS(i-1,w) & \text{if } w_i > w \\ \max\{MSS(i-1,w),\ MSS(i-1,w-w_i)+w_i\} & \end{cases}$$

⇒ **Subset Sum Problem**

I/P: Given a list of positive int a[1..n] & an int t

O/P: True if any subset such that sum is t.

Ex: a = [12, 1, 3, 8, 20, 50]

t = 44 → T

t = 42 → F

t = 43 → T

PAGE NO. :
DATE : / /

$$SS(i,S) = \begin{cases} \text{True} & S=0 \\ \text{False} & i=0,\ S\neq 0 \\ SS(i-1,S) & a_i > S \\ SS(i-1,S-a_i) \text{ OR } SS(i-1,S) & \end{cases}$$

Ex: $a=[10,5,15]$, $S=20$

So True

Also can be done by bottom up by filling table of 2×4 size

(Perfect ex to show top down may save time).

PAGE NO.:
DATE: / /

★ Coin Change Problem:

I/P: Denomination list & amount to return

O/P: Min. no. of notes to return amount.

Greedy cannot solve this problem.

Let $CC(i,s)$ repr. minimum no. of coins to make a change of s amount by $c_1\ c_2 \ldots c_i$ coins.

$$CC(i,s) = \begin{cases} 0 & \text{if } i=0 \text{ or } s=0 \\ CC(i-1,s) & \text{if } c_i > s \\ \min\{1+CC(i,s-c_i),\ CC(i-1,s)\} & \end{cases}$$

★ Bellman Ford was also D.P. problem but was done in greedy with dijkstra.

★ **Floyd-Warshall Algorithm**: It is a all pair shortest path unlike dijkstra & B-F which were single src. shortest path as their source was fixed.

I/P: Graph $G(V,E)$

O/P: shortest path b/w all pair of vertices.

We don't need to know algo. Just know that this can be solved using DP.

GO CLASSES

PAGE NO. :
DATE : / /

T.C = (V^3) for floyd warshall

Algo.		T.C.	$E = V^2$		
	F-W	V^3	V^3	→ Already b/w all pairs V^3	
Single Src	Dijkstra	$(V+E)\log V$	$V^2 \log V$	$\xrightarrow{V \text{ times}}$	$V^3 \log V$
	B-F	VE	V^3	$\xrightarrow{V \text{ times}}$	V^4
			Can't compare		Now Compare

we can't directly compare all pair s.p & single src sp problems. Single src s.p. are eq to all pair s.p if we apply them on all V vertices. Now we can compare so F-W is best here with $O(V^3)$

O/P of this algo is given as:

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

→ Shortest path from V_2 to V_4

Atleast one of the diagonal will be negative iff we have a negative cycle in graph.

F-W is DP problem solved in $O(V^3)$, can detect -ive cycle & can work on -ive edges.

GO CLASSES

PAGE NO. :
DATE : / /

★ **Travelling Salesman Problem:** (TSP) NP-Hard problem (open)
No polynomial time algo. There is exponential time algo present for this.

I/P: Graph & Src
O/P: Min. cost to traverse all cities & come back to src.

Ex:

src = A

ABCDA
ABDCA
|
|
ADBCA

n! combinations

⇒ **Elements of Dynamic Programming:**

a) **Optimal substructure:** Can we divide problem into subproblems?

b) **Overlapping subproblems:** Do we have common subproblems?

	Greedy	DnC	DP
Optimal substructure	✓	✓	✓
Overlapping subproblems	✗	✗	✓
Greedy Choice	✓	✗	✗

GO CLASSES

PAGE NO. :
DATE : / /

- Divide & Conquer.

- Dynamic Programming
-

- Greedy

It works if we have optimal substructure & greedy choice property i.e. making greedy choice does not rule out optimal solution.

GO CLASSES

These notes are the property of GOClasses. Please do not share them without their consent.

For any questions or issues regarding my handwritten notes, feel free to reach out via email at karan757527@gmail.com or DM me on telegram at @karan757527 (or by clicking here).

I would love to know if you liked my notes. Your feedback and appreciation are invaluable, so don't hesitate to share your thoughts. Click here to connect with me on LinkedIn or you can find me as @agrawalkaran.

Wishing you success on your journey ahead!

- Karan Agrawal

GO CLASSES