Algo 5N

Analysis: Priori (Absolute analysis) not depends on compiler lang & CPU ← uses asymptotic notations
Posteriori (relative analysis) depends ...

1. TO find Time complexity, focus on loops and recursive function calls.

2. +, − :- n/a *//% :- log n power = log log n

$i=3, i=i^{10}, i\le n$ → $3^{10^k}=n$

$i>3, i=i^{1/10}$ → $n^{1/10^k}=3$

$\log_{10}\log_3$

3. ~~In Recursive program, for TC focus on else part.~~

4. Finding TC means, TC of best algo in worst case.

5. In GP, if $r<1$, $S_n=O(1)$ (decreasing GP)

6. If can't compare fn directly, compare its closest upper bound. Ex. To compare n!, compare n^n.

7. O :- $f(n) \le c\cdot g(n)$ $\}\ \forall n, n\ge n_0$
   Ω :- $f(n) \ge c\cdot g(n)$ $\}\ c>0, n_0>0$
   O and Ω both possible, then Θ also possible.

8. $1/n < 1 < \log(\log n) < \log n < (\log n)^2 < \sqrt{n} < n < n\log n < n^2 < c^n < n^n$
   ($n\log n$ is polynomial not log function)

9. $2^n < n! < n^n$;

10. $n! = O(n^n)$; $\log n! = \Theta(n\log n)$

11. Depth of stack :- no of stack units used = level of tree

12. Recurrence Relation solving :-
    Substitution method, Recursive tree method, master theorem.

13. Master Theorem :- $T(n) = aT(n/b) + f(n)$
    $a\ge 1, b>1$ (constant)
    Ex: $T(n) = 2^n T(n/2) + n^n$ is not applicable (f(n) not allowed)

* $a^{\log_b n} = n^{\log_b a}$ * $\log_b a = \frac{\log a}{\log b}$

* if $a > b$ then $\log a > \log b$

(i) compare $f(n)$ & $n^{\log_b a}$, bigger (polynomial times) or bigger one is TC.

(ii) If equal, $TC = O(n^{\log_b a} * (\log n)^k)$, $k \geq 0$

or log times
right ($n^{\log_b a}$) / smaller → make right equal then add $(\log n)$

NOTE: $f(n) = n$; $n^{\log_b a} = n \log n$ → master theorem is not applicable here as right is log times bigger.

14. Solving RR using master theorem containing root. $\sqrt{n}$

Type: $T(n) = cT(\lceil n \rceil) + f(n)$

(i) Assume $n = 2^k$

(ii) Assume $T(2^k) = S(k)$

(iii) Replace $S(k)$ by $T(2^k)$

(iv) Replace k by $\log_2 n$

NOTE:- If option does not contain O, Ω or Θ, we have to give exact result. (check by substituting value).

* Things to Remember :-

(i) $a^{\log_b n} = n^{\log_b a}$

(ii) $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \log_2(n)$

NOTE:- While computing TC and SC we don't consider decreasing functions.

(iii) In $\log_b a$, on increasing base 'b' value decreases. on increasing a value increases

* a^n & b^n are asymptotically not equal if $c \neq b$ $\overset{c}{=} 2^n, (\sqrt{2})^n$, etc

(iv) logs with different base are asymtotically equal. i.e. $(\log_2 n) = O(\log_8 n)$

* $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

* $e \approx 2.7$

* sometimes compare f" by substituting some big values.

(v) TO compare 2 fu" by taking Log, first eliminate common thing. (Try to avoid log).

(vi) If O & Ω both not possible, fu" are not comparable.

15. TO convert AP-GP series into GP, multiply series by r value of GP then subtract.

16. $n^3 = O(n^3)$ Tightest upper bound
$= \Omega(n^3)$ Tightest lower bound

17. If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$ False
it fails if we take $f(n) = 2n$, $g(n) = n$

18. $f(n) = O(f(n/2))$ False (Ex: $f(n) = 2^n$)

19) $f(n)$, $(f(n))^2$ are not comparable
O fails for decreasing f" & Ω fails for inc. f".

20. $\log^* n$ is no. of times log applied on n to make it 1.
Ex: $n = 2^{2^2}$ $\log^* n = 3$

21. * means "how many times" to make 1.
$f(n) = n/2 \Rightarrow f^*(n) = \log_2 n$
$f(n) = n+2 \Rightarrow f^*(n) = n/2$

22. Compare log fu" by substituting $\log n$.
Ex: $(\log(\log n))! < \log(n!) < (\log n)! < (\log n)^{\log n}$
$= n \log n$
Let $\log n = k \Rightarrow n = 2^k$
$(\log k)!$ (Put $n = 2^k$ to compare this) $nk = 2^k k$ $k!$ k^k

* $T(n) = T(n-1) + \sqrt{n}$
$T(n) = 1 + \sqrt{2} + \cdots + \sqrt{n} = \sum_{n=1}^{n} \sqrt{n} = \int_{x=0}^{n} \sqrt{x}\,dx = \left[\frac{x^{3/2}}{3/2}\right]_0^n = O(n^{3/2})$

Extra

1) S_n of GP $= \frac{a(r^n-1)}{(r-1)}$ $r>1$; $\frac{a(1-r^n)}{(1-r)}$ $r<1$

$T_n = a r^{n-1}$

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

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

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

3) $2^2+4^2+6^2+\cdots+(2n)^2 = 2^2\left[1^2+2^2+3^2+\cdots+n^2\right]$

* If fu^n are $n^{1+\sin n}$, $n^{1+\cos n}$ compare power by drawing graph.

4) ${}^nC_K + {}^nC_{K+1} = {}^{n+1}C_{K+1}$

NOTE: $T(n) = n^2 + n + T(n-1)$
Asymptotically take it as
$T(n) = n^2 + T(n-1)$

Q. TC of $B(i) = 1/i$ ← its decreasing f^n its max value is 1 so take $O(1)$

```
for(i=1; i<=n; i++)        } → O(n)
{ B(i); }                  ← O(1)
```

$\therefore TC = O(n)$

* If not mentioned in ques TC of BC or WC, find for both & choose option satisfying both.

* $n^2 = O(1.000001^n)$

NOTE: If $f(n) > g(n)$ $\forall n$ then $g(n) = o(f(n))$ False
This is absolute comparison, can't use asymptotic for it.
Ex: $f(n) = 2n+1$, $g(n) = 2n$ & $f(n) > g(n)$ $\forall n$
but $g(n) \neq o(f(n))$

DAC

1. Finding max and min
2. Power of an element
3. Binary search
4. Merge sort
5. Quick sort
6. Selection procedure (Kth smallest)
7. Strassen matrix multiplication
8. counting inversions
9. Finding max contiguous subarray sum.

⇒ Finding Max & Min :-

1) Without DAC — Take max, min 1st ele & simply traverse the array.
   (i) TC = O(n) (EC)
   (ii) SC = O(1)
   iii) No of comparison = 1(n-1) BC
   = 1.5(n-1) AC
   = 2(n-1) WC

2) With DAC
   (i) RR:- $T(n) = 2T(n/2) + C$
   (ii) TC = O(n) (EC)
   (iii) SC = O(log n)
   iv) No of comparison = 1.5n - 2 Every case
   ↳ RR:- $C(n) = \begin{cases} 2C(n/2) + 2 & \text{if } n > 2 \\ 1 & \text{if } n = 2 \\ 0 & \text{if } n = 1 \end{cases}$

* If in ques algo is not mentioned, go with DAC.

* Function call in recursive program is Preorder & Fu" execution is Postorder.

* Which is good? ← DAC

		comparision
Best case comparison	without DAC	(n-1)
Avg " "	with DAC	1.5(n) - 2
worst " "	with DAC	1.5n - 2

3) If we divide array in 3 equal parts
   (i) RR: $T(n) = 3T(n/3) + C$
   (ii) TC = O(n)
   (iii) Stack space = $O(\log_3 n)$
   iv) RR for comparison :- $C(n) = 3C(n/3) + 4$

2. ⇒ Power of an element :- (a^n)

1. without DAC

$TC = O(n)$

if $n = 2^k$; $TC = O(\log n)$ → $p = p * p$

3^8
$3 \times 3 = 9$
$9 \times 9 = 81$
$81 \times 81 =$ Ans

2. with DAC $a^n = a^{n/2} \cdot a^{n/2}$ (let $n = 2^k$) if not 2^k then some constant adjustment

(i) RR :- $T(n) = T(n/2) + c$

(ii) $TC = O(\log n)$ Every case

(iii) stack space $= O(\log n)$

(iv) RR for no of multiplication :- $P(n) = P(n/2) + 1$

(v) No of multiplications $= O(\log_2 n)$

➤ Linear Search :-

(i) $TC = O(n)$ WC
$O(1)$ BC
$O(n)$ AC
} ⇒ $TC = O(n) = \Omega(1)$

3. ⇒ Binary search :- (sorted array)

1. With DAC.

(i) RR :- $T(n) = T(n/2) + c$

(ii) $TC = O(\log n)$ WC, AC
$= O(1)$ BC
} ⇒ $O(\log n) = \Omega(1)$

(iii) $SC = O(\log n)$ WC, AC
$= O(1)$ BC
} ⇒ $SC = O(\log n) = \Omega(1)$

* Tail recursion

2. Without DAC (non recursive)

(i) $TC = O(\log n)$ AC
WC $= \Omega(1)$ BC

(ii) $SC = O(1)$ EC

* Avg no. of comparison $= \log_2 n$

* For small value of n calculate manually.

n = 10 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Avg no of comp $= \frac{1 \times 1 + 2 \times 3 + 4 \times 5 + 3 \times 7}{10} = 4.8$

* If in ques SC is asked and algo is not given, consider best one i.e. non recursive one.

4. ⇒ Merge Sort :- (Outplace & stable)

➤ Merge Algo :- (Sorted subarrays) (size of subarray)

(i) Min no. of comparison (BC) = min (m, n)

(ii) Max no. of comparison (WC) = (m+n-1)

(iii) No. of moves (in BC & WC) = (m+n)

(iv) TC = O(m+n) (for outplace algo) (common algo)

(v) TC = O(mn) (for inplace algo)

↳ Compare, swap, sort 2nd subarray.

➤ Merge sort :

(i) RR :- $T(n) = 2T(n/2) + n$

(ii) TC = O(n log n) (Every case)

(iii) SC = Stack space + Meage algo = log n + n = O(n)

* Sorting algo are of two types :-

(i) Inplace :- Quick sort (takes < log n extra space)

(ii) Outplace :- Merge sort (takes > log n extra space)
count, Radix sort

* If sorted array is given (in ques), think of Binary search and Merge algo.

* A list of n strings each of length n is to be sorted in lexicographic order using merge sort. TC ⇒ $O(n^2 \log n)$

Date: / /

➤ Fibinocci series :-

(i) RR :- $T(n) = T(n-1) + T(n-2) + C$

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

(iii) Stack space = $O(n)$

➤ Fibinocci Merge sort :-

Division done as fibinocci no.

TC = $O(n f(n))$ Every com

n = 8
f(6) = 8
f(6)
f(5) f(4)
f(4) f(3)

➤ we may use modified Merge algo to find A∪B and A∩B. (A & B are sorted arrays)

TC = $O(m+n)$ EC.

➤ Stable :- Merge, bubble, insertion sort, count, Radix sort

Not stable :- Quick, selection, heap sort

* Stable sorting algo :- the order of repeatitive elements in i/p and o/p same

➤ Straight 2-way Merge sort :- combining 2-2 groups

↳ TC = $O(n \log n)$

Straight 3-way Merge sort :- Combining 3-3 grps

NOTE :- (completely or almost sorted array)

i) Bigger size array → Merge sort

ii) Smaller size array → Insertion sort

iii) less no of swap → Selection sort

iv) Random array / unsorted array → Quick sort

v) Range given → count sort

* If we convert tail recursion into non recursive program, stack space is saved.
* changing recursive program into non recursive → TC does not change

5. ⇒ **Quick sort:** (Best sorting algo)

(i) It is **inplace** and **not stable**.

➤ **Partition algo:** returns index of pivot element

(i) Normally **1st element** is taken as pivot. If pivot is selected randomly, it's called randomized quick sort.

(ii) TC = O(n) EC SC = O(1)

➤ **Quick sort:**

starting index p, pivot index m, last index q

(i) RR:- $T(n) = n + T(m-p) + T(q-m)$

in BC & AC balanced partitioning

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

TC = O(n log n)

Stack = O(log n)

in WC unbalanced partitioning

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

$TC = O(n^2)$

Stack = O(n)

(ii) Quick sort is not tail recursion in BC & AC but it's tail recursion in WC.

➤ **Optimized quick sort:** Consider it in exam.

(i) After partitioning, handling smaller part by recursion and bigger part by loop

(ii) stack space: O(1) BC (unbalanced partition)
O(log n) WC, AC (Balanced partition)

(iii) TC = O(n log n) BC, AC
$O(n^2)$ WC

(iv) RR in general: $T(n) = n + T(\alpha n) + T((1-\alpha)n)$
$0 \le \alpha \le 1$

(smaller ele in any order) pivot (Largh ele in any order)

pivot ele is min or max always if array is sorted or almost sorted

(V) Quick sort will give worst case if partitioning is unbalanced ie. constant no. of element on one side and remaining on other side.

* In randomized quick sort, worst case (in sorted array everytime selecting smallest or largest ele) is rare, so performance better.

* n^{th} smallest ele as pivot → will go to n^{th} position from starting
n^{th} largest ele as pivot → will go to n^{th} position from last.
median element → $n/2^{th}$ smallest/largest ele will go to $n/2^{th}$ position.

* Using Median of 5 algo, median of unsorted array can be found in O(n) time.

* If selecting median as pivot

$$T(n) = n + n + T(n/2) + T(n/2)$$
$$= n + 2T(n/2)$$
$$= O(n \log n) \quad (EC)$$
$$\text{Stack} = O(\log n)$$

* Finding median of sorted array = O(1)
" " unsorted array = O(n)

6. ⇒ Counting no of inversions : same as merge sort

inversions is max if array is in desc order

(1) Inversion : if $i < j$ and $a[i] > a[j]$

(2) Brute force approach : TC = $O(n^2)$ linear search for each element

Q 

If subarray a has '100' inversions with subarray b. If we sort b, then also inversions btw a & b will be 100.

* Expected no. of inversions in a randomly chosen permutation of $1 \cdots n$ = $^nC_2 * 0.5 = \frac{n(n-1)}{4}$

∵ we have $^nC_2$ pairs & probability of each pair being inverted is $\frac{1}{2}$

(3) Using DAC: modify merge algo in merge sort.

(i) Small prblm: 1 ele → 0 inversion

(ii) No. of inversion of a node = sum of no. of inv. of children + no. of inv. b/w children.

(iii) While merging if smaller ele comes from right side, add no. of ele in left side to total no. of inversions.

(iv) RR :- $T(n) = 2T(n/2) + n \Rightarrow TC = \Theta(n \log n)$

(v) Stack space = $O(\log n)$

(vi) SC = $n + \log n = O(n)$
↑ array to merge

7.

⇒ Selection procedure: (Partition algo + Binary search)

(1) To find K^{th} smallest element in an array.

➢ Approach 1: Sort then select. $TC = O(n \log n)$

➢ Approach 2: K passes of selection sort. $TC = O(kn)$

➢ Using DAC:

(i) Algo: m = partition (After partition m^{th} smallest ele is at index m)
if $K < m$ go left
if $K > m$ go right

(ii) RR :- $$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ n + T(m-p) \text{ or } T(q-m) & \text{else} \end{cases}$$

p = starting index, m = pivot index, q = last index

* Best algo of selection procedure: select median as pivot
$T(n) = n + n + T(n/2) = O(n)$

Date: / /

BC, AC
Balanced partition
$T(n) = n + T(n/2)$
$TC = O(n)$

WC
Unbalanced partition (sorted or almost sorted array)
$T(n) = n + T(n-1)$
$TC = O(n^2)$

(iii)

TC =			in 1st partition k^{th} min found	SC =	
	$O(n)$	BC		$O(1)$	BC
	$O(n)$	AC		$O(\log n)$	AC
	$O(n^2)$	WC		$O(n)$	WC

* In randomized selection procedure, pivot ele is selected randomly (less chance of WC).

⇒ Strassen's matrix multiplication :-

(1) Matrix addition : $C_{n \times n} = A_{n \times n} + B_{n \times n}$
$TC = O(n^2)$

(2) Multiplication without DAC :

(i) $C_{m \times n} = A_{m \times p} * B_{p \times n}$
#columns in A = # rows in B

(ii) C_{ij} = (i^{th} row of A) * (j^{th} column of B)

(iii) For 1 ele of C, p multiplications needed.
For mn elements, (mnp) multipl. n

(iv) $TC = O(mnp) = O(n^3)$ (if $m = n = p$)

(3) using DAC :
(i) Divide order of A & B by 2 and solve.
If order not divisible by 2, add row/column of 0.

(ii) $T(n) =$ 8 * matrix multi of order $n/2$ + 4 * matrix addition of order $n/2$

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

RR: $T(n) = \begin{cases} 8T(n/2) + n^2 & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}$ TC = $O(n^3)$

(4) Acc" to Strassen's, $T(n) = 7T(n/2) + 18(n/2)^2$

$T(n) = 7T(n/2) + n^2$

↳ $TC = O(n^{\log_2 7}) = O(n^{2.81})$

g.

⇒ Maximum contiguous subarray sum :

(1) Algo : • Split from middle
• Recursively calculate max from left & right part.
• calculate max across left & right part. ⇒ O(n)
• Return max of three

(2) RR:- $T(n) = 2T(n/2) + n$

(3) $TC = O(n \log n)$ EC

$SC = O(\log n)$ EC

*Best algo : Kadane's algo :

• Initialize ; max_so_far = INT_MIN ; max_ending_here = 0
• Loop for each element of the array
- max_ending_here += arr[i]
- if (max_so_far < max_ending_here)
max_so_far = max_ending_here
- if (max_ending_here < 0) max_ending_here = 0
• Return max_so_far

TC = O(n) SC = O(1)

NOTE: By default, sorting means ascending order
Heap means maxheap as it gives asc order

Sorting Techniques

Date: / /

1. ⇒ Selection sort :

- It needs $(n-1)$ iteration (passes)
- Total comparison $= 1+2+\cdots+(n-1) = O(n^2)$
- Total swaps $= (n-1) = O(n)$ Each pass, 1 swap
- TC $= \underset{\text{comparison}}{O(n^2)} + \underset{\text{swap}}{O(n)} = O(n^2)$
- NOT stable and inplace.

2. ⇒ Insertion sort :

- Stable and inplace
- TC = $O(n + \text{no. of inversions})$
  - if atmost n inversions ⇒ TC = $O(n)$
  - if exactly 1 " ⇒ $O(n)$ WC $O(1)$ BC (1st 2 el inversion)
  - exactly $\frac{n(n-1)}{2}$ inv ie array in des order to sort in asc order TC = $O(n)$
- Best case : array is in increasing order
  worst case : array is in decreasing order.
- Best case : NO. of comparisons = $(n-1)$
  NO. of swaps = 0
  TC = $O(n)$
- Worst case : NO. of comparisons = NO. of swaps $= \frac{n(n-1)}{2} = O(n^2)$
  TC = $O(n^2)$
- Average case : TC $= \underset{\text{Half array Best case}}{n/2} * O(1) + \underset{\text{Half array worst case}}{n/2} * O(n) = O(n^2)$

NOTE: For best case of sorting, remember insertion sort.

- NO. of inversions in the array = NO. of swaps in insertion sort.
- If array having n elements, have at most n inversions, TC of insertion sort $= O(n)$
  Ex: 10, 20, 30, 40, 50, 5 (almost sorted)
  1 or 2 el are unsorted

NOTE: Array representation is better for almost complete BT; for remaining trees, LL is better.

* For heap consider array representation. If type of tree is not mentioned then consider LL representation.

- If we do Binary search instead of linear search in worst case,

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

  # swaps = $O(n^2)$

  TC remains same = $O(n^2)$

3. ⇒ Bubble sort:

- Stable and inplace
- It needs $(n-1)$ iterations.
- Total comparisons = $1+2+\cdots+(n-1) = \frac{(n-1)n}{2} = O(n^2)$
- Total swaps = 0 BC (array in asc order)

  $O(n^2)$ WC, AC
- TC = $O(n^2)$ EC

  ↳ becoz of comparisons
- SC = $O(1)$

consider in exam

* Modified Bubble sort: count #swaps in each iteration. If #swaps = 0 then stop.

  TC = $O(n)$ BC (in 1 or 2 pass array is sorted)

  $O(n^2)$ WC, AC

4. ⇒ Heap sort:

- Heap is a almost complete BT (height balanced) (height = log n)
- Maxheap tree will give ascending order, (Default)

  Minheap tree will give descending order.
- Inplace and not stable.
- Building heap from array, TC = $O(n)$ (Build heap method)
- Algo: • construct Min/Max heap using buildheap method
  • Perform n deletion

TC = $O(n) + O(n \log n) = O(n \log n)$ WC, AC (when all ele same)

$= O(n)$ BC

If TC is less than O(nlogn) then it is wrong algo.
↳ Best Time complexity it can have is O(nlogn)

Comparison based sorting: selection, insertion, bubble, merge, quick, heap.
NOT comparison based sorting: count, radix sort
↳ Linear time sorting.

5. ⇒ Counting sort: traverse array & put ele in buckets → O(n); traverse buckets & print → O(max(arr)) = K
- i/p: Given 'n' ele in range [1 - K]
- Stable and Outplace
- TC = O(n+K)

6. ⇒ Radix sort:
- i/p: Given 'n' ele in range [1 - n^t]
- Algo: Perform counting sort on each digit starting from LSB.
- Stable and Outplace
- TC = O(dn) = O($\log_K n^t$ n) = O(tn) ← if we take base K=n then (digits)

↖ Here K is base ie constant

7. ⇒ Wave sort:
- arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
- Algo:
```
for(i=0 ; i<n ; i+=2)
{  if(i>0 && arr[i] < arr[i-1])
       swap(arr[i], arr[i-1])
   if(i<n-1 && arr[i] < arr[i+1])
       swap(arr[i], arr[i+1])
}
```
even ele >= left & right side
- TC = O(n)

1. Knapsack problem
2. Job sequencing with deadline
3. Huffman coding
4. Optimal merge pattern
5. Minimum cost spanning tree
6. Single source shortest path

Greedy Technic

(1) Mostly, given n i/ps we have to find a subset which satisfies condition & optimizes our goal.

2) Solution space: set of all possible solutions
Feasible sol": set of sol" that satisfies our condition
Optimal sol": sol" which optimizes our goal.
↳ need not be unique.

1. ⇒ Fractional knapsack problm:

(i) Prblm: Given 'n' objects with their weight 'w' and price 'P', a knapsack of capacity 'C'. Find max profit

(ii) Algo:
- Traverse array to find rate (P/w) of each item
- Sort array of P/w in descending order (MERGE SORT)
- Traverse sorted array of P/w to take items in knapscak as per its capacity, till C becomes 0

(iii) TC = O(n) + O(nlogn) + O(n) = O(nlogn) EC

(iv) If we use Quick sort, TC in WC = $O(n^2)$
in BC, AC = O(nlogn)

2. ⇒ Job sequencing with deadline:

(i) Prblm: Given 'n' jobs each having 1 unit runtime, their profit & deadline. We have to schedule jobs to CPU inorder to maximize profit.

(ii) Running time (1 unit) ≤ Deadline ≤ n

(iii) #jobs we can execute = max (Deadline)

(iv) Algo:
- Find max of deadline. Make an array "sol" of size = max
- In the array 'sol', start from last and do linear search for each index to find optimal job for that index.

NOTE: In array 'sol', slot i can have jobs with deadline $\geq i$, choose optimal one.

$TC = \overset{\text{finding max}}{O(n)} + n * O(n) = O(n^2)$ EC

(v) Better Approach: consider it for ques.
- first sort profit in descending order then follow above procedure

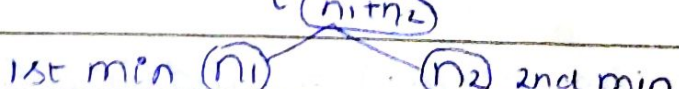
$TC = \overset{\text{sorting}}{O(n \log n)} + \overset{\text{finding max}}{O(n)} + n * O(n) = O(n^2)$ WC
$n * O(1) = O(n \log n)$ BC AC

3.
⇒ Huffman encoding: Optimal prefix notation/ Data compression technique

(i) Prblm: Given a message composed of 'n' distinct character with frequency of each character. Encode characters to minimize message size.

(ii) char having high frequency are represented in less no. of bits.

(iii) Algo:
- Build minheap of frequency of char.
- Delete twice to get 2 minimum.
- Do



- Add new node t in minheap.
- Give left edge label 0 & right edge 1.

(iv) If all char occur with freq $< \frac{1}{3}$, then it's guaranteed to be no codeword of length 1.
- If some codeword occur with freq $> \frac{2}{5}$, it's guaranteed to be a codeword of length 1.
- With n symbols, max codeword len could be (n-1)

(v) char with high freq are at top of tree, char with low freq are at bottom of tree.

(v) $TC = O(n) + (n-1)\,O(3\log n) = O(n \log n)$
(Build heap; each time 1 ele deleted & at last 1 ele left so repeated (n-1) times; 2 deletion +1 insertion)

(vi) 1 char Huffman code can't be prefix of other char.

(vii) By default
Huffman 2-way encoded tree: taking 2 min as children
Huffman 3-way encoded tree: take 3 min as children.
↳ label Left:0, Mid:1, Right:2

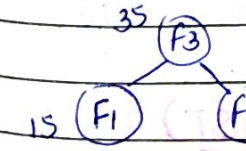
(viii) By default, order of labeling edge is
Left → 0, Right → 1
If ans does not match try, Left → 1, Right → 0

(ix) Decode an encoded message via tree.

(x) If there's not much difference in freq, we will get almost balanced tree.

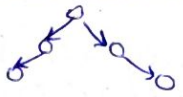
4.
⇒ Optimal merge pattern:

(i) Pblm: Given 'n' files with no. of records, we have to find sequence to merge them to have minimum no of moves.

(ii)


No. of moves = 15 + 20 = 35 $(m+n)$
Max no of comparison = $(15+20-1) = 34$ $(m+n-1)$
Min no of comparison = 15 $\min(m,n)$

(iii) Merging 2-2 files → 2 way merging (By default)
" 3-3 " → 3 way merging

* MST of directed graph: From source all vertices should be reachable 

Date: / /

(iv) Algo: same as huffman encoding.
Difference is just that, after choosing 2 min, order does not matter which one to keep left & right.

(v) Min no of moves = sum of internal nodes

(vi) $TC = O(n) + (n-1)O(3\log n) = O(n \log n)$

Basics of graph:

(i) $E \leq \frac{V(V-1)}{2} \Rightarrow E = O(V^2) \Rightarrow \log E = O(\log V)$

(ii) Adjacency matrix representation: ← good for almost complete graph
- Space complexity = $O(V^2)$
- To check adjacency of 2 vertex $\Rightarrow TC = O(1)$
- To find degree of a vertex $\Rightarrow TC = O(V)$ scan a row
- To find total edges $\Rightarrow TC = O(V^2)$

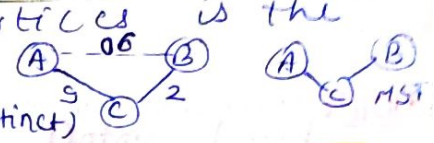
(iii) Adjacency list representation: (By default)
- Space complexity = $O(V+2E) = O(V+E)$
- To check adjacency of 2 vertex $\Rightarrow TC = O(V)$ WC, $O(1)$ BC
- To find degree of a vertex $\Rightarrow TC = O(V)$ WC, $O(1)$ BC } = O(deg)
- To find no of edges $\Rightarrow TC = O(V+2E)$

(iv) In a V-vertex simple graph, min no. of edges required to make it connected = $(V-1)$

Minimum cost spanning trees (MST):

(1) Spanning tree: Subgraph connecting all vertices with minimum no of edges $(V-1)$.
(2) If a graph with $(V-1)$ edges is connected → has no cycle (it is tree
disconnected → has cycle.

NOTE: In MST, the path blw any 2 vertices is the shortest path. False.



NOTE: If edge e has max weight of some cycle True then every MST excludes e. (if edge wrist is distinct)
. If e has min weight of some cycle, then every MST includes it. False

(3) NO. of spanning trees in complete graph (K_n) = n^{n-2}

(4) There are 2 greedy algo to construct MST:
- Prims (vertex by vertex approach)
- Kruskal's (edge by edge approach)

• SCV edge with min weight such that one vertex is in S & other in (V-S) must be in MST.

→ 1st & 2nd min edge must be part of MST
atleast one of 3rd & 4th min must be part of MST

➤ Kruskal algo:

(i) Algo:
- construct minheap of weight of edges.
- select min edge, add to MST if no cycle.
- Repeat till all vertices are covered ie. (V-1) edges are included in MST.

(ii) For checking cycle, we use disjoint set data structure.

Algo:
- initially all vertex are leader of there own.
- If we are considering edge a–b and leader of a & b r different then no cycle. Grp ab & make any one leader
- If both has same leader then it forms cycle.

* check for leader using 'find' operation and merge two grps using 'union' operation.
* If we simply do union ⇒ TC = O(V)
* If we implement disjoint set DS via union by rank → find & union takes ⇒ TC = O(logV)
* If we implement via union by rank & path compression find & union takes ⇒ TC = O(1) ← By default consider it.

(iii) TC: (Build heap, delete from heap, check cycle)
- BC: $TC = E + (V-1)\log E + (V-1)O(1) = O(E + V\log E) = O(E + V\log V)$
- WC: $TC = E + E\log E + E\,O(1) = O(E\log E) = O(E\log V) = O(V^2 \log V)$
- For array: $TC = E\log E + E\,O(1) = E\log E$ (Default) (sort, check cycle) — If weight is in range apply radix sort, TC = O(E)

(5) Kruskal algo may generate disconnected graph in blw but prims always generate connected.

More than 1 MST → guaranteed edge weight repeated

(6) If edge weights are not distinct, more than 1 MST may be possible.

All edge weight distinct → Unique MST.

➤ Prims Algo:

(i) Algo:

- Initialize min = ∞, parent = NULL for all vertices.
- Make minheap of 'min' of vertices.
- Start with any vertex & make its min = 0.
- Select minimum & update min and parent of its adjacent vertices if min can be reduced.

*decrease key takes O(logn) time in heap. (WC) BC O(1)

(ii) TC (graph represented by Adj list & using minheap)

$TC = \underbrace{V}_{\text{Build heap}} + \underbrace{V\log V}_{\text{deletion from heap}} + \underbrace{2E}_{\text{checking adjacent}} + \underbrace{2E\log V}_{\text{decrease key}} = O[(V+E)\log V]$

∵ in connected graph $E \geq (V-1)$

$TC = O(E\log V)$

* If any vertex has min = ∞, parent = NULL till last, means graph is disconnected.

NOTE: By default consider adj list representation of graph.

(iii) TC (graph represented by adj matrix & using Minheap)

$TC = V + V\log V + V^2 + 2E\log V = O(V^2 + E\log V)$

(iv) TC (using adj list & normal array)

$TC = V^2 + 2E + 2E*O(1) = O(V^2)$

(v) TC (using adj matrix & array)

$TC = V^2 + V^2 + 2E*O(1) = O(V^2)$

NOTE: In undirected graph, Σdegree = 2E
" directed " Σdegree = E

NOTE: If all edge weights are distinct & every edge weight is increased by the same value then MST does not change, shortest path blw any pair of vertex may change.

(vi) TC (using adj list & sorted array)
TC = V*O(1) + 2E + 2E*O(V) = O(EV)
(min is initially sorted)

(vii) TC (using adj matrix & sorted array)
TC = V*O(1) + V² + 2E*O(V) = O(EV)

(viii) TC (using adj list & sorted doubly LL)
TC = V*O(1) + 2E + 2E*O(V) = O(EV)

6.
⇒ Single source shortest path :

(1) Prblm: we have to find shortest (min cost) path from given source vertex to every other vertex

(2) There are two algos:
- Dijkstra → vertex by vertex (same as prims)
- Bellmon ford → Edge by edge

(3) min of vertex 'x' represents min cost path from source to x

➤ Dijkstra:

(i) Algo is same as prims
When we select min 'x', its adjacent 'y' will be updated if y.min > (x.min + xy)

(ii) TC = V + V log V + 2E + 2E log V = O[(V+E) log V]
(Build heap; deletion from heap; adjacent scores; decrease key)
If graph is connected E ≥ (V-1) = O(E log V)

(iii) It works good with +ve edge weight, it may fail with -ve edge weight.

(4) Blw two vertices more than one shortest path may be possible, no matter edges are distinct or repeated.

Date: / /

➤ Bellman Ford:

(i) Algo:
- Initialize min = ∞, parent = NULL of all vertices (source vertex min = 0).
- Give edge no. to all edges (in any order).
- Goto all edges sequentially and do decrease key if possible ← DO it (V-1) times because max path length possible is (V-1)

(ii) Here we r using array (not minheap).

(iii) TC = V * E * O(1) (decrease key in array) = O(VE)

(iv) It handles -ve edge weight perfectly.

(v) Perform last step once more Vth time. If value of vertex B, C changes by -ϑ from previous step then there is -ve edge weight cycle having sum -ϑ & B, C are either part of it or dependent on it.
↳ DO min of B & C = -∞

(vi) It can find -ve edge weight cycle if its reachable.

(5) Which one is better?
- +ve edge weight graph → Dijkstra
- -ve edge weight graph → Bellman ford
- unweighted or equally weighted graph → BFS
  ↳ take weight = 1
  ↳ TC = O(V+E)

1. Fibonacci series
2. 0/1 Knapsack
3. Sum of subsets
4. Matrix chain multiplication
5. Longest common subsequence
6. Travelling salesperson
7. All pair shortest path (Floyd, Warshall)
8. Optimal cost binary search tree
9. Length of longest increasing sequence

Dynamic Programming

1. ⇒ Fibonacci series:

1) $f(n) = \begin{cases} n & \text{if } n \le 1 \\ f(n-1) + f(n-2) & \text{else} \end{cases}$

➤ Recursive program:

i) TC = $O(2^n) = \Omega(2^{n/2})$; SC = Stack = $O(n)$

ii) $T(n) = \underbrace{T(n-1) + T(n-2)}_{\text{no. of fu call}} + \underbrace{C}_{\text{↳ 1 fu cost}}$

iii) TO calculate $f(n)$, no. of f" call made = 2^n

iv) " " $f(n)$ " - distinct f" call = $(n+1)$

2) f(n) once calculated, store it for further use, its DP.
↑ we will use array to have random access

Size of array = No. of distinct f" calls

➤ Dynamic Programming:

i) TC = $\underbrace{(n+1)}_{\text{no. of distinct f" calls}} * \underbrace{O(1)}_{\text{↳ each f" cost}} = O(n)$

ii) SC = $\underbrace{n}_{\text{stack}} + \underbrace{(n+1)}_{\text{array}} = O(n)$

iii) Can't write recurrance relation here. (For conditional recursion)

2. ⇒ Longest Common Subsequence: (LCS)

1) No. of subsequence possible of a string of length $n = 2^n$

Ex: "abc": ϕ, a, b, c, ab, ac, bc, abc

2) Prblm: Given 2 arrays X & Y of length m & n, we have to give length of LCS.

LCS(m, n)

NOTE: Recursive f" calling is Preorder
f" execution is Postorder

Ex: Str1: A L G O R I T H M
Str2: G L O B A L I S M

cancel letters that r not common then find LCS directly.
LCS = 4

Date: / /

(3) Brute force approach: LCS(m,n)

if m=0 or n=0 return 0
if X[m] == Y[n] return 1+LCS(m-1,n-1)
if X[m] ≠ Y[n] return max[LCS(m-1,n), LCS(m,n-1)]

(i) In best case (X,Y are same) only if part executed, we get unary tree of height min(m,n).

(ii) In worst case, (no ele in common) else part executed, we get binary tree of height (m+n)
↳ Total f" call = 2^{m+n}

(iii) TC in WC: $T(m,n) = T(m-1,n) + T(m,n-1) + C$
$TC = 2^{m+n}$, Stack = SC = O(m+n)

TC in BC: $T(m,n) = T(m-1,n-1) + C$
$TC = O(\min(m,n))$, $SC = O(\min(m,n))$

(4) No. of distinct f" calls in f(m,n) = (m+1)(n+1)
= mn

Each f" cost is constant.

(5) with DP:

(i) TC = (no. of distinct f" calls) * (cost of 1 f" call) = $\begin{cases} O(\min(m,n)) & BC \\ O(mn) & WC \end{cases}$

(ii) Make a 2D array of size (m+1) * (n+1).

(iii) SC = (m+n) + mn = O(mn) EC
stack array

(6) Non recursive approach of DP:

- Fill the table from bottom to top
- TC = O(mn), SC = O(mn)

3. ⇒ 0/1 knapsack:

(1) Prblm: Given 'n' items with weight & Profit, a knapsack with capacity 'm', find max Profit.

(2) Brute force approach: f(m, n)

if m=0 or n=0 return 0

BC if weight[n] > m return f(m, n-1)

WC else return max[f(m - weight[n], n-1) + profit[n] , f(m, n-1)]

(i) TC = O(2^n) WC Stack = O(n) WC, BC

O(n) BC

(3) No. of distinct f^n calls in f(m, n) = (m+1)(n+1) = mn

Make a 2D array of size (m+1)(n+1).

(4) With DP :

(i) TC = mn * O(1) = O(mn) ≈ O(2^n) (m may be large; bcoz of less repetitive f^n call)

(ii) SC = n (stack) + mn (array) = O(mn)

(5) With DP also TC can't be reduced from 2^n, so its NP complete problem.

4. ⇒ Sum of subsets : (similar to 0/1 knapsack)

(1) Prblm : Given a set having 'n' +ve integers & an integer 'm'. Find any subset whose sum is 'm'.

(2) No. of possible subsets of a set having n ele = 2^n

(3) Algo without DP: f(m, n)

if m=0 return ϕ

if n=0 error

if set[n] > m return f(m, n-1)

else

{ return f(m - set[n], n-1) ∪ set[n] (take set[n])

return f(m, n-1) (don't take)

}

TC = O(2^n) WC SC = O(n)
O(1) BC set[n] = m

(4) With DP:
TC = O(mn) WC SC = n + mn = O(mn)

(5) Bcoz of less repetition, algo with DP has TC O(mn) ≈ O(2^n). Thus, its NP complete problem.

⇒ Matrix chain Multiplication: (Optimal parenthesis prblm)

(1) $A_{m \times n} \times B_{n \times p} = C_{m \times p}$ for 1 etc of C, n multiplications required.
Total no of multiplication required = mnp

(2) Matrix multiplication is associative but not commutative.

(3) Prblm: We need to find best way to multiply n matrices inorder to have minimum multiplications
ABC = (AB)C = A(BC)

(4) n ← (ABCD) → Here finding min of (n-1) children takes O(n) time ie 1 f^n cost is O(n)
n-1 ← (A)(BCD) (AB)(CD) (ABC)(D)
n^2+n ← (B)(CD) (BC)(D) (A)(B) (C)(D) (A)(BC) (AB)C
It is a n-ary tree (max n children) with n level.

TC (without DP) = n + n^2 + ... + n^n = O(n^n)
Stack = O(n)

(5) f(1, n) → M1 × M2 × ... × Mn
(1 will grow to n; n will decrease to 1)
No. of distinct f^n calls = n^2 Each f^n cost is O(n)

NOTE: For ordering 4 matrices we require 3 bracket ().
No. of possible ordering for n matrices is C_{n-1} ← catalon no. $\frac{2^k C_k}{k+1}$

(6) $f(1,4) = \min \begin{cases} f(1,1) + f(2,4) + \text{combining time} \\ f(1,2) + f(3,4) + \text{"} \\ f(1,3) + f(4,4) + \text{"} \end{cases}$

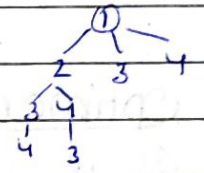
(7) TC (with DP) = $O(n^2) * O(n) = O(n^3)$
stack = $O(n)$
SC = $\underset{\text{stack}}{n} + \underset{\text{array}}{n^2} = O(n^2)$

(8) In DP, we need 2D array of size $n \times n$

⇒ Travelling salesman prblm:

1) Prblm: Given n interconnected cities, salesman has to start from any city & return back to same city covering all with minimum distance.

(2) $f(i, S) = \min_{k \in S} \{ ik + f(k, S - \{k\}) \}$
(starting city, set of remaining city)



Similarly we need to check with all starting point

(3) TC (Brute force approach) = $O(n!)$
check all n! permutations to find min.

4) It is a n-ary tree with n levels. 1 f" cost = $O(n)$. (finding min of n children)
No. of distinct f" calls in $f(i,S) = n * 2^{n-1}$

5) TC = No. of distinct f" calls * 1 f" cost = $n * 2^{n-1} * n = O(n^2 2^n)$

6) It is a NP Hard problem.

7. ⇒ All pair shortest path (Floyd Worshall algo)

(1) Prblm : Matrix A^0 gives distance blw vertices. our task is to find min distance blw each pair of vertices.

(2) considering vertex 1 as intermediate vertex, build A^1. (diagonal ele, 1st row, 1st column ele will not change)
$A^1[2,3] = \min\{A^0[2,3], A^0[2,1] + A^0[1,3]\}$
similarly consider all vertex 2,3,4 as intermediate. (let there be 4 vertex). A^4 gives final result

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

(4) TC = $n * n^2 = O(n^3)$

8. ⇒ Optimal binary search tree: (Weighted BST)
It is a BST that minimizes expected search cost. (no of comparisons required to search a key).

Ex: Keys = {10, 12} freq = {34, 50}