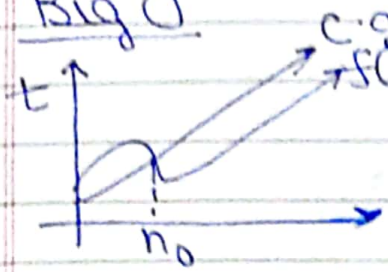
ALGORITHMS

I) Time & Space Complexity:

1) Big O → Worst Case

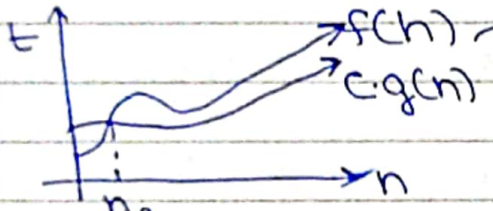


$f(n) = O(g(n))$ (imp)

$f(n) \leq c \cdot g(n)$ for $n \geq n_0$
Where, $n_0 \geq 1, c > 0$

ii) $f(n) = n$, $g(n) \rightarrow n, n^2, n^3, n^4 \cdots$
But, O → Tightest Upper Bound ∴ $O(n)$

2) Big Omega (Ω) → Best Case

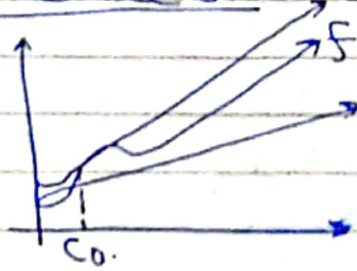


$f(n) = \Omega(g(n))$

$f(n) \geq c \cdot g(n)$, for $n \geq n_0$
Where, $n_0 \geq 1, c > 0$

ii) $f(n) = n$, $g(n) \rightarrow n, \log n, \log(\log n), 1 \cdots$
But, Ω → Tightest lower Bound ∴ $\Omega(n)$

3) Theta (θ) → Avg Case



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

$c_1 g(n) \leq f(n) \leq c_2 g(n), n > n_0$
where, $n_0 \geq 1, c > 0$

eg: $f(n) = 3n^2 + 2n$, → $\theta(n^2)$ $[O(n^2), \Omega(n^2)]$

→ When best & Worst cases are equal, Theta (θ) exists.

* Algo → Iterative : count no. of times loop runs
  → Recursive : analyse recursive function
  To analyse Time

* Time complexity of Iterative Progs

1) for(i=1; i<=n; i++) → O(n)

eg → Pg4 [Gate 1991] (increment by i — inner loop)
calculate loop runs in terms of n.

eg → Pg5 [Q2] [Unrolling loop] (imp)

2) for(i=1; i<=n; i=i*2) → O(log n)

3) A()
{ while(n>1) → n=n/2;
}

n=2 — 1 t
n=4 — 2 t.
n — log n times
∴ O(log n)

eg → Pg7 [Q2] O(n log log n) (imp)

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n} = \log n$$

* Time Complexity of Recursive Progs

A(n)
{ if(n>1)
  return A(n-1)
}

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

- 2 Methods → Back Substitution
  → Tree Method.

eg → Pg8 [Back Substitution Method]
$T(n) = T(n-1) + 1$. (imp)

- Tree Method → $T(n) = 2T(n/2) + n$

eg → Pg9 [$2T(n/2)+1$] → Tree Method

$$1 + 2 + 4 + 8 + \cdots 2^K = \frac{1(2^{K+1}-1)}{2-1}$$ ← Sum of GP. (imp)

* Compare Various functns to Analyse Time

1) Scratch out common terms
2) Log both Sides
3) Substitute Very large Values of n & compare.

Q $2^n > n^2$.
$n, 2\log n$ Sub $n = 2^{100}$
$2^{100} > 200$.

$n > (\log n)^{100}$
$\log n, 100 \log\log n$
Subst. 2^{2300}
$2^{300}, 30000$

Q $n^{\sqrt{n}} > n^{\log n}$.
⇒ $\sqrt{n} \log n, \log n \log n$
⇒ $\frac{1}{2}\log n, \log(\log n)$. $n = 2^{2^{10}}$
⇒ $512 > 10$ ✓

eg → Pg12 [last Q]
[See ∞ range of n] ; $g(n) > f(n)$.

eg → Pg 13 [Q2] (imp)
[Comparing 4 functions, & arrange them]

* MASTERS THEOREM

$T(n) = aT(n/b) + \theta(n^K \log^p n)$
$a \geq 1, b > 1, K \geq 0, p$ is real

1) if $a > b^K$, then $T(n) = \theta(n^{\log_b a})$
2) if $a = b^K$
   a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3) if $a < b^K$
   a) if $p \geq 0$, then $T(n) = \theta(n^K \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^K)$

Note: $\log^2 n = \log(\log n)$
$(\log n)^2 = \log n \cdot \log n$

Q $T(n) = 2^n T(n/2) + n^n$ ✗ a needs to be const

eg → Q5, Q6, Q14.

Q $T(n) = 64 T(n/8) - n^2 \log n$ ✗ (imp)

* Analyse Space Complexity

- how much max extra space required by program in terms of i/p
1) Iterative: if 1D Array (size = n), created inside prog → Space = O(n)
2) Recursive:

eg → Pg 19 [Q2] (Use Tree Method) (imp)

Time → depends on no. of funct^n calls
Space → depth of Stack / depth of Tree

* Amortized Analysis (imp) [Build Heap]

- Instead of computing cost of 1 oper^n, perform series of operations, & compute avg cost.

Techniques → Aggregate
→ Accounting
→ Potential.

* Aggregate Analysis

[Arraylist]

Prob: Insert into Array, if full: double size, & copy ron elements & insert

→ Worst Case = $O(n^2)$
(to insert n elements, n time for each).
→ Amortized Cost = $O(1)$

eg → Pg 21 [Example & Time Analysis] (imp)

II] SORTING TECHNIQUES

1) Insertion Sort

- Pick card, compare & arrange in left hand.
- Total n-1 Passes
- 1+2+3+.....n-1] Comp.

Time: [O(n²) → Worst Case (Rev Sorted)
O(n) → Best Case (Sorted)]

Space: O(1)

(i) If Binary Search used: (Time) (imp)
No. of comp = O(log n)
Movements/Shift = O(n). Decrease comparison
∴ T = O(n²)

(ii) Doubly Linked list
No of comp = O(n) Decrease movement
Movement = O(1)
∴ T = O(n²)

2) Merge Sort Algorithm [Divide-Conquer]

- Heart of MS: Merge Algo.
- To merge 2 sorted lists: n & m
  T = O(n+m), S = O(n+m)
  (n+m) comp. & (n+m) copy.
- ∞ is added at the end of sublists, to copy the other list, if 1 list is over.

eg → Pg 24 [Why ∞ & Merge] (imp)

* Merge Sort [Out of Place / Stable]

Divide & Conquer: Divide Prob into many SP, Solve the SP first, & then combine.

eg → Pg 25 [Working of MS] (imp)

⇒ Space complexity

- Merge Procedure: O(n) — Extra Array.
- Recursive calls:
  [Size of Stack = depth of Tree = O(log n)]
  ∴ Space = O(n + log n) = O(n)

⇒ Time Complexity: T(n) = 2T(n/2) + n
(Partition always at middle)

T = Θ(n log n)
(Best = Avg = Worst Case)

eg → Pg 26 [n/log n lists — O(1)]

Note: To compare two n-length string, # comp = O(n)

eg → Pg 27

3) Quick Sort Algorithm [Divide-Conquer]

- Heart of QS: Partitioning Procedure.

Note: if i/p is small, QS better than MS.

* Partitioning Procedure → T = Θ(n)
1. Choose last element as pivot
2. i = -1, j = 0
3. if A[j] > pivot, → inc j.

3. if A[j] <= pivot, inc i ; & swap A[i], A[j]
4. At last, exchange A[i+1] with pivot.

∴ T = Θ(n) → In any Array input.
[Partitn Algo]

* Quick Sort

eg → Pg 29 [Code]

- Depending upon the Partitioning idx, the SubProblems/ time complexity vary. (imp)

⇒ Space Complexity
- Partitioning Algo: O(1)
- Recursion:
  No. of Stack Records = ht of Tree → O(logn) : Best Case
  → O(n) : Worst Case

⇒ Time Complexity
T(n) = n + T(n−K) + T(K)
if K = middle → T(n) = 2T(n/2) + n : O(nlogn)
K = last/first → T(n) = T(n−1) + n : O(n²)

Partn at middle: O(nlogn): Best Case
Partn at last/first: O(n²) : Worst Case

(ii) Ascending / Descending / Array with Eq, elem.
[T = O(n²), ht of Tree = O(n), Skewed]
Tree → n / 0, n−1 / 0, n−2
Work/ Level = n
Levels = n
∴ T = O(n²)

(iii) Split input into ratio (eg: 1:9, 1:99, 1:999)
T = O(nlogn)

eg → Pg 32 [Split i/p into 1:9 ratio] or Select 1/10th Smallest elem as Pivot.

(iv) Alternate Best & Worst Case.
[Worst] n / 0, n−1
[Best] (imp) (n−2)/2, (n−2)/2
∴ T(n) = cn + cn + 2T((n−2)/2)
T(n) ⇒ O(nlogn)

eg → Pg 33 [Median 9]

* HEAPS

- The DS, as an i/p to an algorithm, affects the time complexity of Algo.

⇒

	Insert	deleteMin	findMin
MinHeap	O(logn)	O(logn)	O(1)

* Heap: Almost complete binary Tree [no Gaps in Array]
- Implemented as an Array. Visualized as Tree

100(i) / 10, 20 / 1, 4, 3 [Max Heap]

1	2	3	4	5	6
100	10	20	1	4	3

parent(i) = ⌊i/2⌋ [Parent of i]
left(i) = 2i
right(i) = 2i+1 (imp)

eg → Pg 35 [Q1] (Is it a maxHeap, & HeapSize)

Array in desc Order: maxHeap
Array in asc Order: minHeap (imp)

* Properties of a complete Binary tree

(ii) ht of a node: longest edge to leaf
(iii) ht of tree = ht of root
(iii) Given ht: , max nodes → $[h \Rightarrow 2^{h+1}-1]$

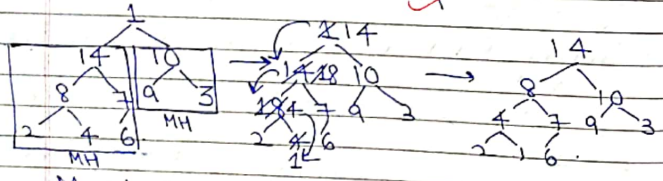
(iv) Ternary Tree: h → $(3^{h+1}-1)/2$

(iii) if n nodes
$n \xrightarrow{\text{min ht}} \lfloor \log n \rfloor$

(v) In a CBT, leaves start from index value $\{\lfloor \frac{n}{2} \rfloor + 1 \text{ to } n\}$ (imp)

(iii) every leaf is a heap. #leaves = n/2

* MaxHeapify: If left & right SubTree is max Heap, what to do with root so as to make the tree → maxHeap.



· Max dist root travel = O(h) = O(logn).

∴ MaxHeapify: T = O(logn) = O(h)
S = O(logn)

imp → depends on level of node, where called.

* Build Max Heap

- Non leaf ∈ $\lfloor n/2 \rfloor$ to 1.
- Apply MaxHeapify one by one, on the non leaf Nodes.

eg → Pg 40 [eg1] (imp)

NOTE: Max no. of nodes, at height h: $\lceil \frac{n}{2^{h+1}} \rceil$ in Heap (CBT)

∴ Build Heap = MaxHeapify on non leaf Nodes

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O(n)$$

∴ Build Heap → Time Complexity = O(n)
Space Complexity = O(logn) (imp)
Max. Space taken by any heapify call

(i) Extract Max / Delete Max — findMax = O(1)

(ii) Delete the root of maxHeap, & return it.
(iii) Replace root with last element, & dec HeapSize.
(iii) ∵ L & R are maxHeaps, apply maxHeapify.

T = O(logn)
S = O(logn)

(2) Increase Key (MaxHeap)

· IncreaseKey(A, 5, 300), [index 5 → 300]
· Worst Case: if leaf node value travels to root.
∴ O(h) = O(logn) (imp)

T = O(logn)

3) Decrease Key (MaxHeap)
- if root value is decreased
- L & R are maxHeap, so MaxHeapify
∴ $T = O(h) = O(\log n)$

4) Insert element (MaxHeap)
- Insert -∞ at the last position.
- IncreaseKey (-∞, 300) ; & 300 travels up.
∴ $T = O(\log n)$

- findMin / deleteAny / Search : O(n) (imp)
[MaxHeap]

4) Heap Sort [InPlace Sorting]

(i) Build MaxHeap.
(ii) Exchange root with last element, & dec Heap size.
(iii) Apply maxHeapify & repeat(ii).
→ reverse sorted from behind.
$T = O(n \log n)$ (imp)
∴ atleast n/2 times, the ht will be log n.

5) Bubble Sort

→ Heaviest element comes to the right after 1st pass. [Stable / Adaptive / InPlace]
→ n-1 Passes.
$n-1 + n-2 + \cdots + 1 = O(n^2)$
eg → Pg 43 [Code]

- Time complexity { $O(n^2)$: Reverse Sorted. / $O(kn)$: (if K heaviest elements are unordered, $K \ll n$) $\Rightarrow O(n)$

6) Bucket Sort
Sort a large set of floating no.'s in range 0 to 1.
No. of buckets = Base = 10.
Space Complexity:
$S = O(n+k)$; n nos, k buckets. (imp)
Time complexity: { $T = O(n)$ [Uniform dist] / $T = O(n^2)$ [All map into same bucket] }
DS used → HashTable + Chaining.
eg → Pg 46 [Q]

7) Counting Sort
→ Range must be known, & keys should be in that range (K)
$S = O(K)$ (imp)
$T = O(n+K)$
→ for every no. in range, maintain a cell in array.
→ traverse the array, & count occurs of num, & put in cell.
eg → Pg 47 (imp)
* Disadvantg: [• Range must be known / • [1, 1000, 2, 3] (ArraySize = 1000)]

8) Radix Sort

1) Each key in Array should be / made a d-digit number.

2. Digits are numbered from 1 to d (right to left)
   eg: 3 5 3
3. for i = 1 to d:
   // Use Stable Sort Algo to sort A on digit i
   [Counting Sort]

eg → Pg 48 [Example] ← imp
(When sorted on Unit digit, the 1 digit no. will be sorted)

* Time Complexity: $T = \log_b u \left(O(n+b)\right)$, $S = O(b)$ ← imp
  u: largest no.
  $\log_b$: no. of digits
  $O(n+b)$: Time of Counting Sort.
  $O(b)$: Space used by counting Sort.
  b → Base
  n → No of elem

9) Selection Sort
- find the smallest element in each Pass & place in beginning
  $T = O(n^2)$, $S = O(1)$ → $n-1+n-2+n-3+\cdots 1 = O(n^2)$ ($n-1$)
- No of Swaps = $O(n)$ - Max

III] GREEDY TECHNIQUE

- Used for Optimization Problem (min/max)
  Optimization Problem → Greedy (want solve all, some)
  → DP (solve all, but some in exponential time)
  → $O(nK)$
  → $O(2^n)$

1) Fractional Knapsack Problem
→ sort n objects in P/W ratio [decreasing order] → $(n \log n)$ (imp)
→ check one by one, if it could fit in KS [full/fraction] → (n)
  $T = O(n \log n)$
  $S = O(b)$ → [P/W array]

eg → Pg 52, Algo → Call

* Greedy KS using Max Heap:
  - Build max Heap (P/W): $O(n)$
  - delete max n times: $O(\log n) \times h$ (atleast n/2 times)
  ∴ $T = O(n \log n)$ [with Array / with Heap]

2) Huffman Codes → [min no of bits, to save a file]
→ Used to compress a file.
→ if 4 char: 2 bits needed. (Fixed Size encoding)
  ∴ Total 8 bits
  but, Huffman Code: (Variable Size encoding)
→ If occurrences/frequencies of chars are not same, HC is used. Highest freq char gets the least no. of bits. (imp)
→ HC = Prefix Codes [Code can't be prefix of any other code] eg: 0, 10, 110, 111.

External Path Length = No. of bits to save file
Σ(freq × Path length)

eg → Pg 56 (imp)

ALGO

1) Make a min Heap with c.[freq of char] : O(n)
2) for i to n:
   allocate new node z (Tree Node)
   z.left = extractmin(Q) } O(2 log n)
   z.right = extractmin(Q)
   z.freq = x.freq + y.freq } O(log n)
   Insert(Q, z)

(1) Heap :. T = O(n log n) ← imp
S = O(n) → Space req to store tree.

(2) [Sort Algo] Sort → n log n
extract min → O(n) [Shift] } O(n²) ∵ (n-1) times
insert → O(n).
T = O(n²)

Note: If problem: pulling out min, & put back: Heap (imp) only pull out min: Sorting Algo.

eg → Pg 58 [Prob in Huffman coding]

(3) Job Sequencing with Deadlines

1. Arrange the jobs in dec order of Profits : O(n log n)
2. Take each job, & Place it as close to its deadline (in Gantt Chart) : O(n²)

(imp) T = O(n²)
S = { O(n) : if n << max Deadline(d)
    { O(d) : if n > d

eg → Pg 61 [Q 2]

(4) Optimal Merge Patterns

→ Merge files with sorted records, so as to minimise the no of record movements. (imp)
External Path = No. of record movements
Length

Algo (i) Create min Heap : O(n)
(ii) Take 2 min, merge it, put back : O(n log n)
∴ T = O(n log n)

→ Largest record files will be moved less, nearer to root. (imp)

eg → Pg 63 [Q 7]

* Intro to Spanning Trees

- Simple Graph : no loops / ll edges.
- Null Graph : no edges
  E = O(V²) → Complete Graph (worst case)
  V = O(log E)

→ Spanning Tree : Min no of edges (tree) required to connect all Vertices (n) → (n-1) edges.

⇒ No of Spanning trees → n^{n-2} STs (imp)
via Kn

* Kirchoff's theorem

→ To find no. of STs for a Not Complete Graph
(i) Construct Adjacency Matrix
(ii) Diagonal Os → degree of nodes
(iii) Non-diagonal 1s → "-1"
(iii) Non-diagonal Os → "0".
No. of STs = Cofactor of any element
= $(-1)^{i+j} M_{ij}$

eg → Pg 65 [Q 7] (imp)

* MST: ST with min cost out of the n^{n-2} STs [Brute Force]

(5) Prim's MST Algorithm

→ Start a vertex (u), select its nxt nbr (v).
→ From (u,v), select the nxt nbr from the unvisited nodes & so on. (imp)
→ Tree goes one edge a time, & we never get a cycle here (∵ node selected from UV nbrs)
→ if edge wts not distinct: more than 1 MST.
eg → Pg 69 [Gate-10] (Adj. Matrix)
eg → Pg 70 [O should be log]

(i) Prims implementation (Without MinHeap)

· DS: boolean visited[V], weights[V], parent[V]
→ Run loop V times
{ · Pick vertex with minWeight from Unvisited (min), & Mark it as visited
· explore all its unvisited nbrs
→ decide whether to update Parent & weight
if weights[i] > edges[min][i]
// update
}
eg → Pg 71 [Algo + Example] : T = O(V²), S = O(V)

(ii) Prims implementation (with MinHeap)

eg → Algo [Code] (imp)
G: Graph, r: root; & pg: Pg 71

· Build Heap: O(V)
· extract min: O(V log V)
· decrease Key: (DK carried out Edge times).
∴ O(E log V)
· T = O(V log V + E log V + V). (imp)
T = O(E log V) [With Min Heap]
T = O(V log V + E) [with Fibanacci Heap]
O(E log V) ⟷ O(V²)

(1) Dense Graph: E = O(V²)
- O(V² log V) > O(V²)
- choose without Heap
(2) Sparse Graph: E = O(V) (imp)
- O(V log V) < O(V²)
- choose with Heap implementation

(6) Kruskals MST Algorithm

→ We may get a forest, we need to check if the edges dont form a cycle (while adding)

* Disjoint Sets

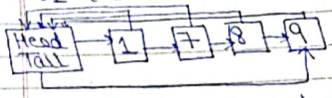
Sets which are disjoint to each other.
⇒ Operations on Disjoint Set:
(i) Create set(x): Create set with item(x) [Choose any item of a set as representative]
(ii) Union: S1 US2
(iii) Find set(x): Find set of x.

· DS for Disjoint Sets → Linked list imp.
→ Forest/Tree imp.

1) Linked list implementation

imp

$S_1 = \{1, 7, 8, 9\}$

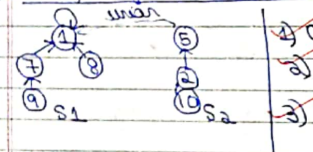


1) Create Set: O(1)
2) Find Set(7): O(1)
∵ Pointer to 7
3) Union: O(n)
∵ Point all elem of next list to head.

∴ By Amortized Analysis, with LL implement^n
$T = \Theta(n)$ [Merge n disjoint sets, union]

2) Tree / Disjoint Forest Implementation

imp



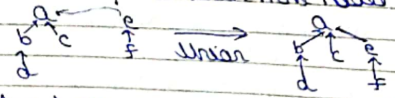
1) Create Set: O(1)
2) Find Set(x): O(d)
[d = depth]
3) Union(x, y): O(1)

* To improve FindSet opⁿ, = restrict depth
  - Union By Rank
  - Path compression

(i) Union By ~~Rank~~ Rank

- Make root of tree with less nodes point to the root of tree with more nodes

eg:

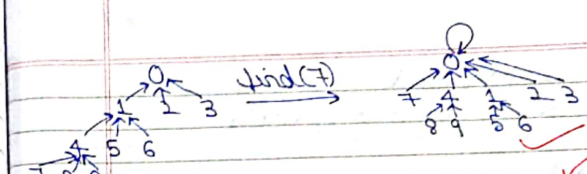


imp

Max depth possible with n nodes = log n
∴ Find Set ≈ O(log n) [Union By Rank]

(ii) Path Compression

- whenever we perform find(x), make every node visited in the path point to the root. Next, find(x): O(1)



imp

∴ If we use Union By Rank + Path compression,

Create Set: O(1)	: Disjoint Set DS (tree imp)
Find Set(x): O(1)	
Union(x, y): O(1)	

5) Connected components using Disjoint Set DS

- For dynamic Graphs: Disjoint Set > BFS/DFS (better)

eg → Pg 84 [Find no of Connect Comp]

∴ $T = O(E + V)$
$S = O(V)$

imp

6) Kruskal's MST Algorithm (Using Disjoint Sets)

Algo → Pg 84

imp

$T = O(E \log E) / \Theta(E \log V)$
$S = O(V)$

7) Dijkstra Algorithm

- Single Source Shortest Path Algorithm
- It doesn't have the capacity to detect (-ve) edge cycles, so it is not allowed to work with -ve edge Graph (no trust on answer)

eg → Pg 87 [Algo] (Dijkstra with Heap)

imp

extract min (V times): $O(V \log V)$
decrease Key (E times) / Relaxation: $O(E \log V)$

∴ $T = O(E \log V)$
$S = O(V)$

(i) If Graph is disconnected / Graph has -ve edge cycle ⇒ Shortest Path not possible.

(B) Bellman Ford Algorithm

- Single Source Shortest Path Algo
- can detect -ve edge cycle

Graph → Bellman Ford → No (if -ve cycle) / Yes (SP)

- But, Slower than Dijkstra.

⇒ Intuition: If G has n vertices SP won't have more than n-1 edges.

(i) List the edges (in some order)

(ii) Relax all the edges (n-1) times. After 1st relaxation, SP contains at most 1 edge (imp)

(iii) Relax 1 more time, if value of any node changes ⇒ -ve cycle exists.

∴ T = O(E × V-1) × O(1) → relaxation time [∵ array]

$T = O(VE)$

if, Sparse Graph: $T = O(n^2)$
Dense Graph: $T = O(n^3)$

IV] DYNAMIC PROGRAMMING

dynamically decide → storing in table

* Fibonacci:

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

$T = O(2^n)$; (without DP)

∵ ht of recursion tree = n, # / rec calls = 2^n.

$T = O(n)$, $S = O(n)$ (with DP)

(A) Matrix Chain Multiplication Problem

$A_{p \times q}$, $B_{r \times q}$

⇒ A·B → Total mul = $p \times r \times q$, Size = $p \times q$, (C)

Problem: Parenthesise mult. of n Matrices in a way, so as to min # multiplications (imp)

* For n Matrices → no. of parenthesising ways = $K_{n-1} = \frac{2nCn}{n+1}$ (n=4, subst 3 in formula)

⇒ Requirements for DP:

(i) Optimal Substructure: Main Problem must be divided into Small Problem, & conquered it

(ii) Recursive formula.

(iii) Overlapping / dependent SubProblems.

* $A_1 A_2 A_3 \cdots A_n$ ← Matrix
$p_0 \times p_1$, $p_1 \times p_2$, $p_{n-1} \times p_n$ ← dimension [array]

* $A_i A_{i+1} A_{i+2} \cdots A_j \rightarrow (A_i A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$ + ($P_{i-1} \times P_k$, $P_k \times P_j$, $P_{i-1} P_k P_j$)

⇒ Recursive eqn: (imp)

$$m[i,j] = \begin{cases} 0, & i = j \\ \min\{m[i,K] + m[K+1,j] + P_{i-1} P_K P_j\}, & i \le K \le j-1 \end{cases}$$

[min # mul to mult $A_i \cdots A_j$]

eg → Recursion Tree [Pg 94]

$T = O(2^n)$ // using Recursion without DP.

Lots of overlapping SPs.

→ No. of Unique SubProblems = $O(n^2)$ (imp)

eg → Pg95 [Q]

No. of unique function calls = n^2 [Lcs(i,j)]
Time to compute 1 funct call = O(n). ∵ K∈(i,j)

T = $O(n^2 \times n) = O(n^3)$ (imp)
S = $O(n^2)$

- Bottom up DP (Iterative - Tabular)
- Top down DP (Memoization - Recursive)
  [Almost Space, bit more due to Stack recursion]

③ Longest Common SubSequence

Q. RAVINDRA / AJAY — LCS = {AA} = 2 — m = 5, n = 4 : 2^m SS

(i) Brute force: $O(2^m \times n)$

* X: $x_1 x_2 x_3 \cdots x_n$
  Y: $y_1 y_2 y_3 \cdots y_m$

Recursive Eqn: (imp)

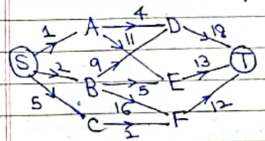
$$C[i,j] = \begin{cases} 0 & ; \ i=0 \text{ or } j=0 \\ 1 + C[i-1, j-1] & ; \ x_i = y_i \\ \max(C[i-1,j], C[i,j-1]) & ; \ x_i \neq y_i \end{cases}$$

* Recursion tree (in worst case)

eg → Pg99 [Recursion tree]
Depth of tree = n + m
(ii) $T = O(2^{n+m})$ // recursion without DP. (imp)
∴ lots of overlapping SPs
→ No. of unique SPs: O(n×m)
Time to compute 1 SP: O(1)
(iii) ∴ T = O(mn), S = O(mn) // with DP.

eg → Pg101 [Bottom up DP - tabular] (imp)
(Path tracing from bottom most cell).

③ Multi Stage Graph → within a stage, no edges; edges only from S_i to S_{i+1}.



Problem: Find Shortest Path from S to D.

- Dijkstra takes O(ElogV) ∵ compute SP from Source to all Vertices ∈ D. ⇒ Too Slow.

$$M(S,1) = \min \begin{cases} S\text{-}A + m(A,2) \\ S\text{-}B + m(B,2) \\ S\text{-}C + m(C,2) \end{cases}$$

(node, Stage no.)

m(S,1): SP from node S [stage 1] to T.
(Recursive eqn)
Vertices in next stage.

eg → Pg103 [Recursion tree]
depth = O(n) (imp)
(ii) T = O(Kn) // if K vertices in each stage.
// without DP.
∴ lots of overlapping SPs ⇒ DP.

eg → Pg103
∴ No. of unique SPs = O(n)
Time req to compute 1 funct call = O(n)
∵ $T[i] = \min_{j=i+1 \text{ to } n} \{ cost[i,j] + T[j] \}$ (imp)

T = O(E+V)
S = O(V)

(ii) T = $O(n^2)$
S = O(n)
[1D Array]

∵ $E = n^2$ → T = O(E), S = O(V)
// with DP.

④ 0/1 KnapSack

KS → Fractional KS [Ritz] → Greedy
KS → 0/1 KS [Pants] → DP.

eg → Why Greedy fails in 0/1 KS → Pg 105 [capacity = w] (imp)

(i) Brute Force: → T = O(2^n)
[for every obj include/exclude]

* Recursive Eqn: (imp)

$$KS(i,w) = \begin{cases} 0; & i=0 \text{ or } w=0 \\ KS(i-1,w); & w_i > w \\ \max\begin{pmatrix} P_i + KS(i-1, w-w_i) \\ KS(i-1,w) \end{pmatrix} & \begin{matrix} \text{// include} \\ \text{// exclude} \end{matrix} \end{cases}$$

i: ith object, w: tot. capacity left

eg → Recursion tree [Pg 106]

(ii) Recursion without DP.
depth = n.
∴ T = O(2^n), S = O(n)
lot of overlapping SPs, ∴ DP.

eg → Pg 106 [Bottom up DP]
No of Unique SPs = O(n × W)
Time req to compute 1 SP: O(1)
Table Size = O(n·w)
(iii) ∴ T = O(n+w), S = O(n+w) // with DP. (imp)
but in case, w ≈ 2^n → T = exponent.
depends on w: Not Polynomial Algo.
nP complete Problem.
(Lower time bnd, no proof that lower time exist)

⑤ SubSet Sum Problem (imp)

Problem: Is there any SubSequence in a given Set, whose sum is S.

(i) Brute Force → T = O(2^n) #SubSequences

* Recursive eqn: (imp)

$$SS(i,S) = \begin{cases} \text{True}, & S=0 \\ \text{False}, & i=0, S \neq 0 \\ SS(i-1,S); & S < a_i \quad \text{[exclude]} \\ SS(i-1, S-a_i) \vee SS(i-1,S) & \\ \text{[include]} \qquad \text{[exclude]} & \end{cases}$$

i: ith value, S: sum

eg → Pg 109 [Recursion Tree]
depth = n.

(ii) Recursion without DP → T = O(2^n), S = O(n)

* Lots of overlapping SPs.
* No. of Unique SPs = O(n × S)
∴ Time required to compute 1 SP: O(1)
eg → Pg 109 [Bottom up DP] (imp)
T = O(n·S), S = O(nS) // with DP.
if S ≈ 2^n, why to use DP?, use Brute force
∴ nP complete Problem (like 0/1 KS)

⑥ Travelling Salesman Problem

Problem: The Salesman has to visit every Vertex & return back to source (in min cost) or Find the min cost Hamiltonian cycle. (imp)

(ii) Brute Force → $T = O(n!) = \boxed{T = O(n^n)}$
∵ $(n-1!/2)$ Hamiltonian cycles.

eg → Pg 111 [TSP example]

* Recursive eqn: (imp)

$$T(i,S) = \begin{cases} \min_{j \in S} \left[C[i,j] + T(j, S-\{j\}) \right] & ; S \neq \phi \\ C[i,1] & ; S = \phi \end{cases}$$

(i: set of vertices unvisited; j: ∀ in set; 1: source)

* Lots of overlapping SPs in Recursion tree.
* No. of unique SPs = $O(n \cdot 2^n)$
Time req. to compute 1 SP = $O(n)$
∴ $\boxed{T = O(n^2 2^n)}$ // with DP (imp)
$\boxed{S = O(n \cdot 2^n)}$

· nP complete Problem. → $(n-1)2^n$.

$$(n-1) + (n-1)^{n-2}C_{n-3} + (n-1)^{n-2}C_{n-4} + \cdots + (n-1)^{n}C_{2}$$

(7) Floyd Warshall

- All Pairs Shortest Path Problem.
(i) Run Dijkstra for every node : $O(VE \log V)$
$\approx O(V^3 \log V)$
(ii) Run BF for every node : $O(VE) \neq O(V^4)$
∴ Too Slow → Floyd Warshall

* Recursive eqn: (imp)

$$d_{ij}^{K} = \begin{cases} \min\left(d_{ij}^{K-1}, d_{iK}^{K-1} + d_{jK}^{K-1}\right) & ; K \neq 0 \\ W_{ij} & ; K = 0 \end{cases}$$

· D_0: path allowed (with no intermediate vertex)
· D_1: SP using 1 as intermediate V.
· D_2: SP using 1,2 as intermediate V.
↳ compute D_2 using D_1.

eg → Pg 115 [Bottom up DP]

$O(n)$ # Tables = # Vertices (∵ SP with all V intermediate)
· To compute 1 table → $O(n^2)$ Funct calls.
· each funct call : $O(1)$
∴ $\boxed{T = O(n^2 \times n) = O(n^3)}$ (imp)
$\boxed{S = O(n^2)}$
∵ Max Space only 2 Matrix at a time.

* NP Completeness

Problem → Tractable: if there exists at least one Poly algo. $O(n^K)$ / fast algo / real time S/w solutions.
Intractable: not tractable / slow, can't solve in Poly time, but can give approx results in real time.

UP, DNP, P

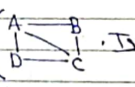
⇒ Many probs look tractable at surface but are not

Tractable	Intractable
- Shortest Path (imp)	- Longest Path
- Euler Tour	- Min Cost Hamiltonian Cycle
- 2CNF	- 3CNF $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee \bar{x_2})$

* Optimizn → Decision Problem.
[Work on Decision Problem (Y/N) b4 Opt Prob.
(i) TSP → Is any SP covering all Vertices of length atmost K?
(ii) 01KS → Is there any sol? whose Profit is atleast K?
(If you cant solve DP, OP no chance)
⇒ We use only DP for NP-Completeness (imp)

• if OP is easy → DP is easy
• if DP is hard → OP is hard

* Verification Algo



· Is G Hamiltonian? A-B-C-D-A
DP. Soln.

VA takes O(n) time [check V & edges]

* P: Set of Decision Problems for which there is Polynomial time algo to solve them.
* NP: Set of DPs whose solution can be verified in Poly time / O(nK) VA.
→ P: Greedy, DnC
NP (imp)

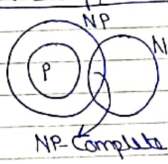
• if all NP-P Problems can be solved in O(nK), P=NP (If NP Hard Conc.)
• if atleast 1 Prob of NP-P cant be solved in Poly time P⊂NP
Rest below P & NP not known yet.

till today, noone could solve it Poly time / proved noone can solve in polll

* Poly time reduction Algo
A —Poly→ B
α β
(i) if β is easy, α is also easy (imp)
(ii) if α is hard, β is also hard.
Eg → Pg 8 [Q-A, B]

easy = O(nK) solved

* NP-Hard: If every Problem in NP can be poly time reducible to a Problem 'A', then 'A' is NP-Hard. (imp)
→ if 'A' can be solved in P, every Prob in NP is P.
∴ P=NP
* NP-Complete: If it is NP & NP Hard.
* Our Belief / not Proof



* if NP-Complete, can be solved in Poly time, than ∀ NP problem in P ∴ P=NP (imp)
* if NP-Complete, proved to never been solved. P⊂NP

Task as a researcher.
* Status of NP = unknown (nothing but Solvable time, only nK VA).
* if A=NP Hard, & A —Poly→ B, then B=NP Hard.
* if B is already in NP, ⇒ B=NP Complete.
* Well Known NP Complete.

Circuit-SAT → SAT → 3NFSAT → Clique → Vertex Cover → HC → TSP
3NFSAT → Subset Sum.