28/05/25

Dynamic Programming

- Top down
  - Memorization
- Bottom-up
  - Tabulation

→ Memorization helps by storing the results of Expensive func^n calls and reusing them when same func^n/ Same input occur again. This avoids the redundant calculations and making the performance of the code Efficient. Memorization is used to speed up the computer programs by Eliminating the repetitive computation of func^n calls that process the same input. It is a specific form of ~~cat~~ caching technique i.e used In dynamic programming where the purpose of caching is to improve the performance of our program and keep the data accessible that can be used later. It basically Stores the previously calculated result of Subproblem and reuses the stored result for the same subproblem. So, Memorization is mainly used to solve the recursive problems which are involving overlapping subproblems.

Memorization consists of 3 types:
- 1 Arguement
- 2 Arguement
- 3 Arguement memorization

Recursion (n^{th} fibonacci program)

-> main ():
- n = 5
- result = n^{th}fibonacci (n) — passing 5 in this method

-> nthfibonacci (5):

if (n <= 1) {

return n

}

return n^{th}fibonacci (n-1) + nthfibonacci (n-2);

= ~~4~~ + 3

= 4

n^{th} fibonacci (4) + n^{th} fibonacci (3)

-> n^{th} fibonacci (3) + n^{th} fibonacci (2)

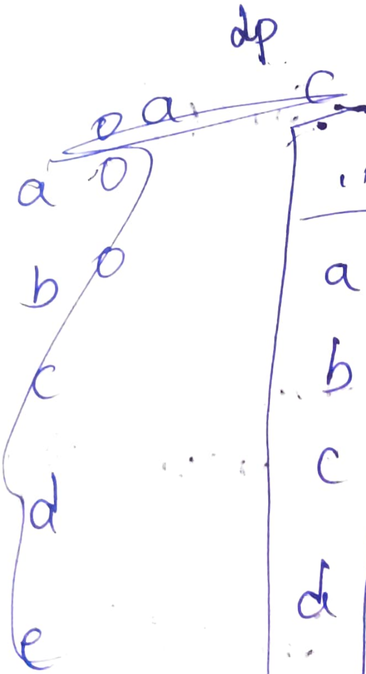
Tabulation

Tabulation is a process to divide problems into subproblems.
Tabulation creates a table and fills up ~~same~~ one row at a time.
Tabulation begins with resolving the smallest subproblems first and ~~bri~~ brings up towards the largest subproblem using the results of smallest problems

LCS Tabulation

m = 5 dp[6][4]
n = 3

dp table



		a	c	e
""	0	0	0	0
a	0	1	1	1
b	0	1	1	1
c	0	1	2	2
d	0	1	2	2
e	0	1	2	3

→ loop over i from 1 to m, and j from 1 to n:-

return dp[5][3] = 3;
LCS length: 3

Quicksort

-) arr = {10, 7, 8, 9, 1, 5}

-) first we call quicksort (arr, 0, 5)

Pivot = 5

-) Partitioning:

10 > 5 -) Skip

7 > 5 -) skip

8 > 5 -) skip

9 > 5 -) skip

1 > 5 -) swap (10, 1)

After Partitioning:

• arr = [1, 7, 8, 9, 10, 5]

swap pivot with 7 (arr [1])

arr = [1, 5, 8, 9, 10, 7]

-) Partition index = 1

-) Now two recursive calls:

• quicksort (arr, 0, 0) -) return (single element)

• quicksort (arr, 2, 5)

-) Second call: quicksort (arr, 2, 5)

Pivot = 7

-) Partitioning:

• 8 > 7 -) Skip

• 9 > 7 -) skip

• 10 > 7 -) skip

No swaps, place pivot before 8

- arr : [1, 5, 7, 9, 10, 8]

- pi : 2

calls :
- quicksort(arr, 2, 1) → return
- quicksort(arr, 3, 5)

Final sorted array : [1, 5, 7, 8, 9, 10]

T.C : O(n log n)

W.C : O(n^2)

LCS Memoization

S1 = "abcde"

S2 = "ace"

LCS of "abcde" and "ace" is "ace".

So, LCS length = 3

→ func^n : lcs(s1, s2, m, n, memo)

→ recursion + memoization:

- m = current index of s1
- n = current index of s2
- memo[m][n] = stores previously calculated LCS values for (m, n)

→ m = 5 (length of "abcde"), n = 3 (length of ace)

→ characters at s1[4] = 'e', s2[2] = 'e' → match memo table

m/n	0	1	2	3
0	0	0	0	0
1	0	1		
2	0	1		
3	0		2	
4	0	1	2	
5	0			3

O/P:-

LCS Length: 3

$T.C = O(m*n)$

$S.C = O(m*n)$

0/1 Knapsack problem

$m[i, w] = Max(M[i-1, w]\ m[i-1, w + w[i]] + P(i))$

p_i	w_i
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1									
2									
3									

→ 1st if Condition:-

negative values won't come.

$n = 4$

$4 - 1 = 3$

$w(3) \Rightarrow 4$

Values $(n-1)$

$4-1$

$= 3$

6 +

include 6 + knapsack (weights values, Capacity - w - n-1)

$5 - w(n-1) = 0$

weights = | 2 | 3 | 4 | 5 |
0 1 2 3

values = | 3 | 4 | 5 | 6 |
0 1 2 3

capacity = 5, n = 4

memo

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0	0				
3	0					6
4	0					

1FC knapsack(w, v, 5, 4, memo) calls

5 ≤ 5

include = value(3)

= 6 + knapsack(w, v, 5-5, 3, m)

= 6

~~2FC knapsack(w, v, 0, 3, m)~~

1FC

exclude = knapsack(w, v, 5, 3, m)

2FC knapsack(w, v, 5, 3, m)

if (4 ≤ 5

include = 4 + k(w, v, 1, 2, m)

~~3rd FC~~

knap(w, v, 1, 2, me).

if (~~w[1]~~ 3 ≤ 1

else

memo[2][1] = knapsack(w, v, 1, 1, me);

~~4FC~~

knapsack(w, v, 1, 1, memo)

if (2 ≤ 1

5FC

else

memo(1)[1] = knapsack(w, v, 1, 0, n)

knapsack(w, v, 5, 2, mem)

2FC

exclude = knapsa

3rd FC.

knapsack(w, v, 5, 2, m)

3

knapsack(w, v, 2, 1,

if (3 ≤ 5

7

include = 4 + ~~knk~~

exll

knapsack(w, v, 2, 1, m)

2 ≤ 2

4FC

5FC

knapsack if (w(0)

knap(w, v, 0, 0, mem)

include 3 ~~2~~ +

0

excl 0

→ Activity Selection problm is a classic example of Greedy alg

It involves selecting maximum no of activities that dony/don't overlap given list of activities with start & finish times.

Greedy → Always pick the activity that finishes the Earliest & compatible with previously selected activities.

Activity	Start time	finish time
A_1	1	4
A_2	3	5
A_3	0	6
A_4	5	7
A_5	8	9
A_6	5.	9

1.) Sort finish time

A_1 A_2 A_4 A_5 A_6 A_3

2.) Apply greedy Selection $A_1 \rightarrow$ finish time = 4

A_2 start $= 3 < 4 \rightarrow$ skip

A_4 start $= 5 \geq 4 \rightarrow$ Select A_4

A_5 start $= 8 \geq 7 \rightarrow$ Select A_5

$\boxed{A_1\ A_4\ A_5}$

Appln's: Job Scheduling
Task Scheduling

→ Fractional knapsack ⇒ can be solved by greedy method. Items can be broken
Solved efficiently using greedy alg
Ex: filling a bag with grains, oil or gold dust

Ex:

A Item	B value	C weight
1	60	10
2	100	20
3	120	30

knapsack capacity $\boxed{W=50}$

⇒ ratios: 6, 5, 4

Take A, B & 2/3rd of C

1) ratio $= \frac{V}{W} = \frac{60}{10} = 6, 5, 4$
A B C

Total value $= 60 + 100 + 120 \times 2/3 = 240$

2) Sort the items based on ratio.
3) Take the highest ratio & add to knapsack until we can't add next item
4) At the end add next item as much (fraction) as we can

A
cap left = 40
Value = 60

B
cap left = 20
Value = 160

C
Cap. left = 0
Value = 240
Now Take 2/3rd of C
$W: 2/3 * 30 = 20$
$V: 2/3 * 120 = 80$

Final Total $= 60 + 100 + 80 = 240$
knapsack is Exactly full $= 10 + 20 + 20 = 50$ kg