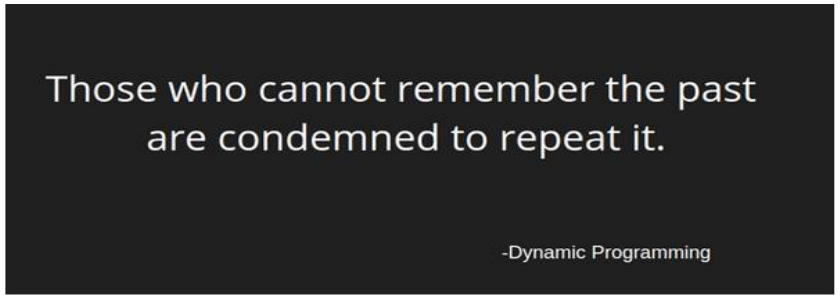
CSE 221 Algorithms: Ch5

Instructor: Saraf Anika
CSE, SoSET, EDU
Fall 2021

Dynamic Programming(DP) Intro

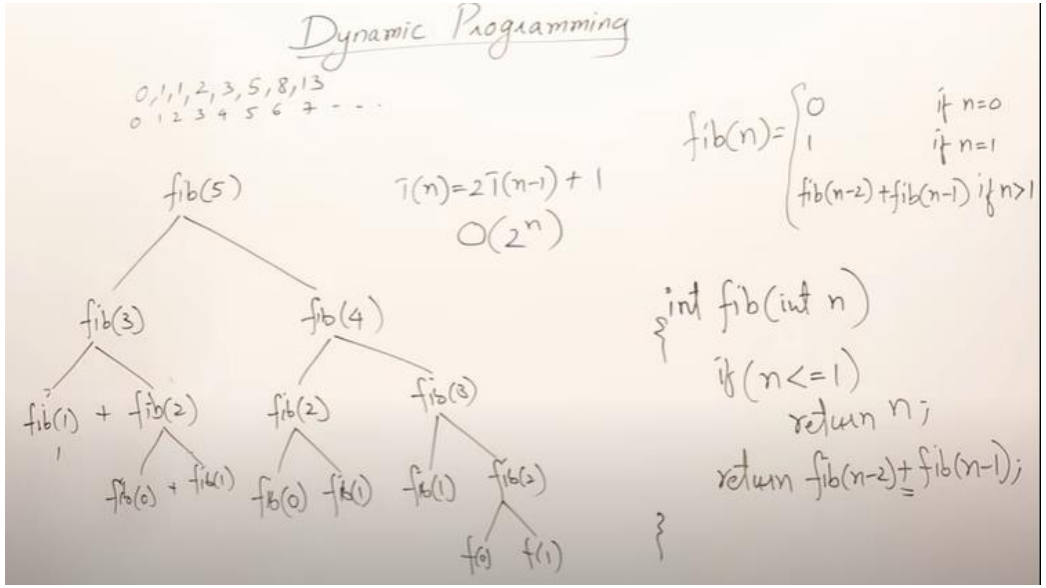


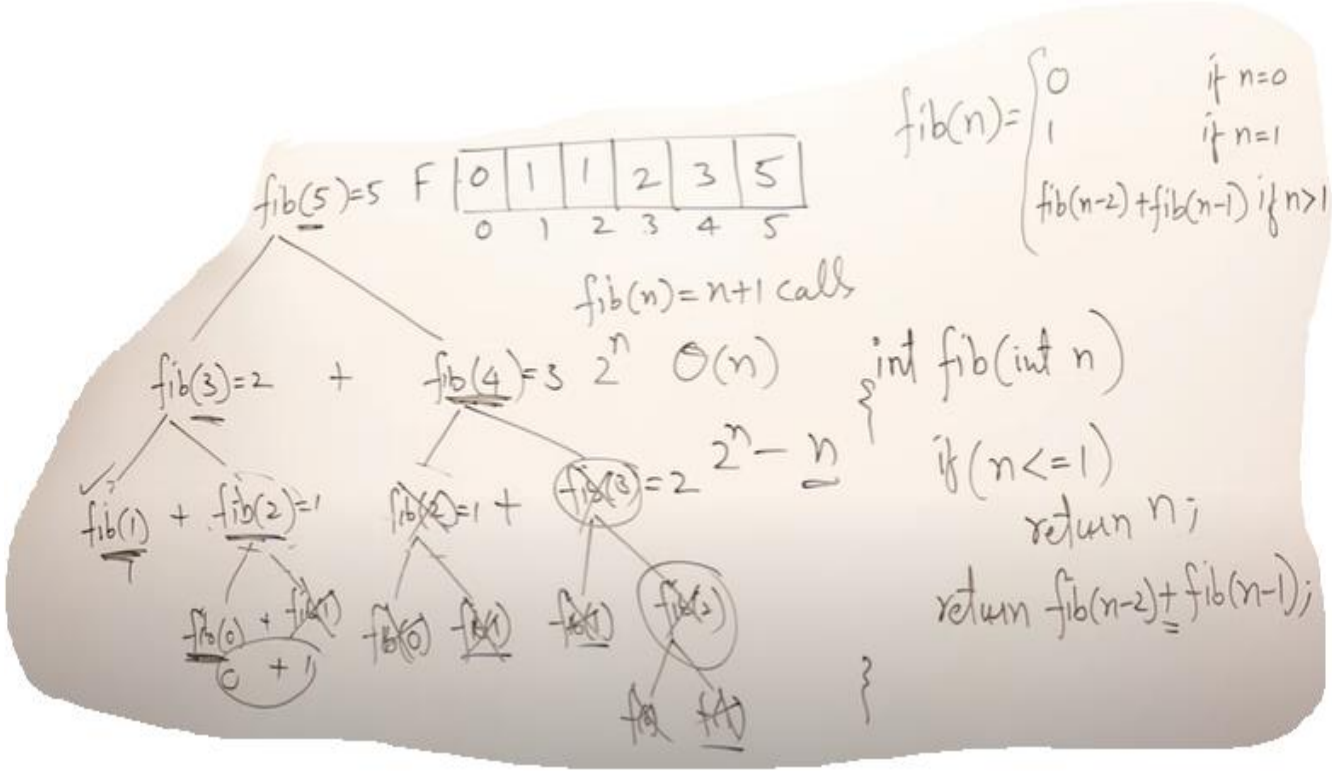
- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its sub-problems.
- Let's take the example of the **Fibonacci numbers**. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8, and they continue on from there.
- If we are asked to calculate the nth Fibonacci number, we can do that with the following equation

Dynamic Programming(DP)

- **Top-down with Memoization :**

In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called Memoization.





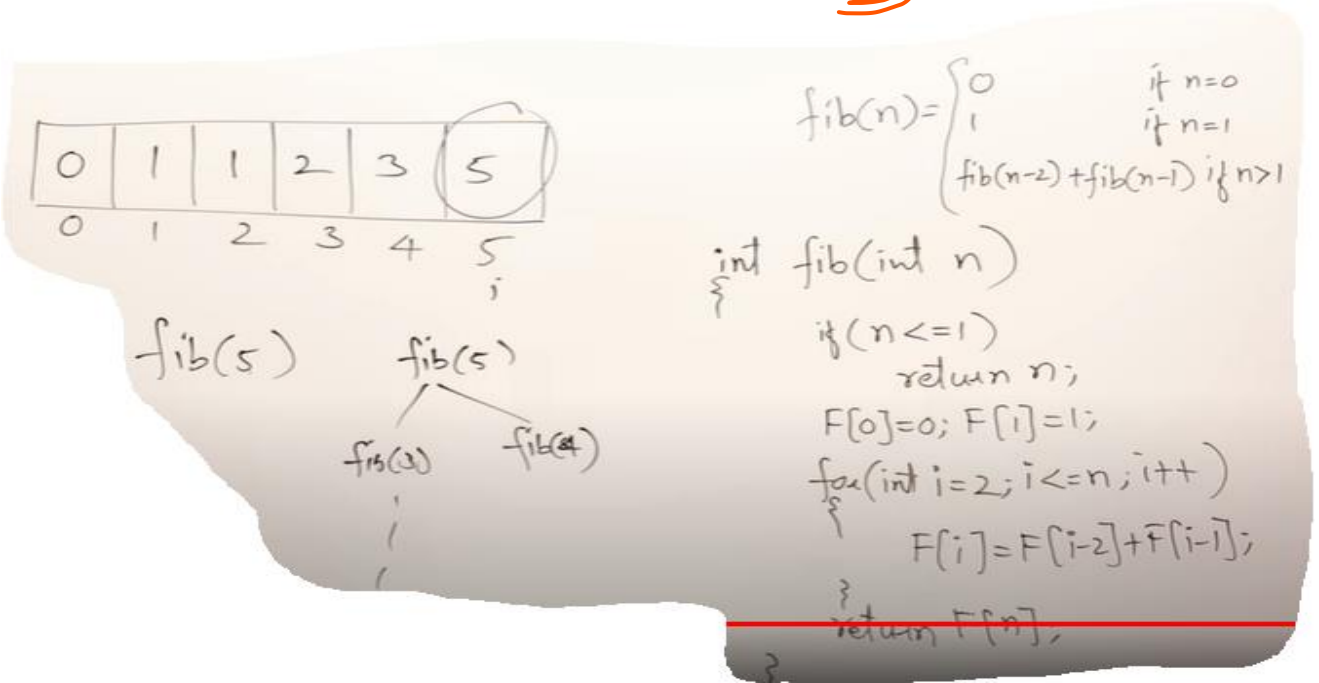
Dynamic Programming(DP)

- **Bottom-up with Tabulation:**

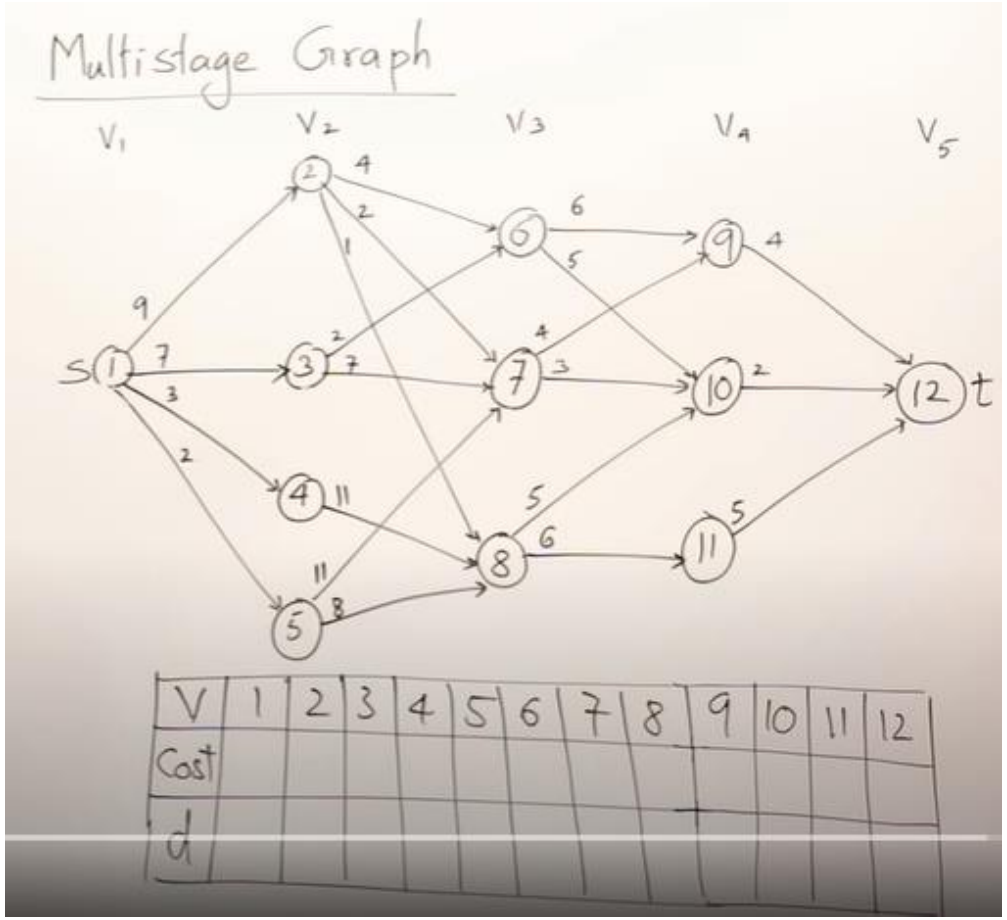
Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Tabulation is the opposite of Memoization, as in Memoization we solve the problem and maintain a map of already solved sub-problems. In other words, in memoization, we do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Tabulation VS Memorization



Multistage Graph(Bottom-Up Approach)



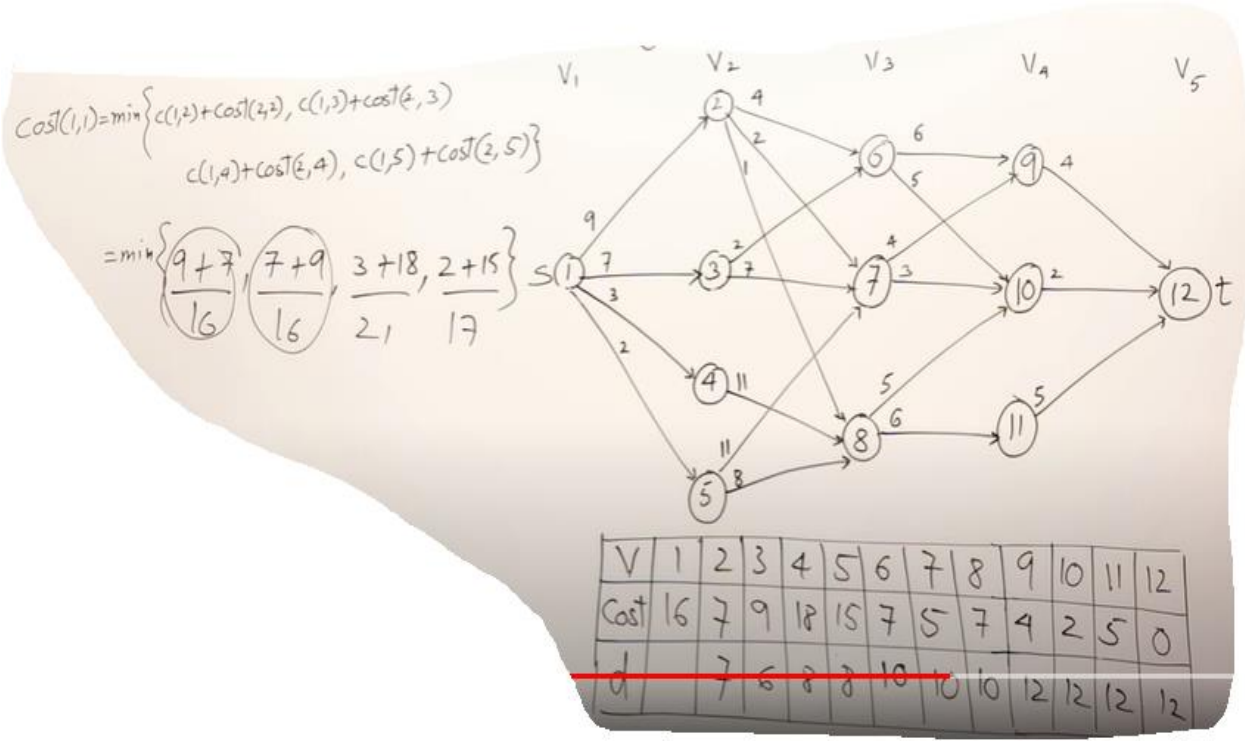
$$Cost(3,6) = \min\{c(6,9) + cost(4,9),\ c(6,10) + cost(4,10)\}$$

$$= \min\{6+4,\ 5+2\} = 7$$

$$Cost(1,1) = \min\{c(1,2) + cost(2,2),\ c(1,3) + cost(3,3),\ c(1,4) + cost(3,4),\ c(1,5) + cost(2,5)\}$$

$$= \min\{9+7,\ 7+9,\ 3+18,\ 2+15\}$$

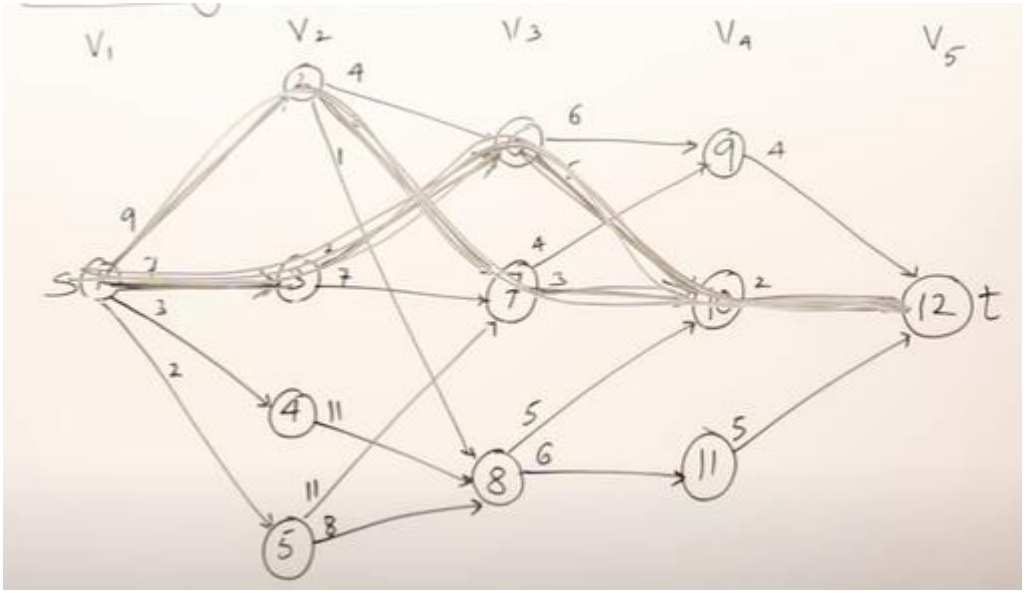
$$= \min\{16,\ 16,\ 21,\ 17\}$$



Multistage Graph(Bottom-Up Approach)

stage, vertex no

$$Cost(i,j) = \min_{\substack{\langle j,l\rangle \in E \\ l \in V_{i+1}}} \{ c(j,l) + Cost(i+1,l) \}$$



stage, vertex

$d(1,1) = 2$

$d(2,2) = 7$

$d(3,7) = 10$

$d(4,10) = 12$

stage, vertex

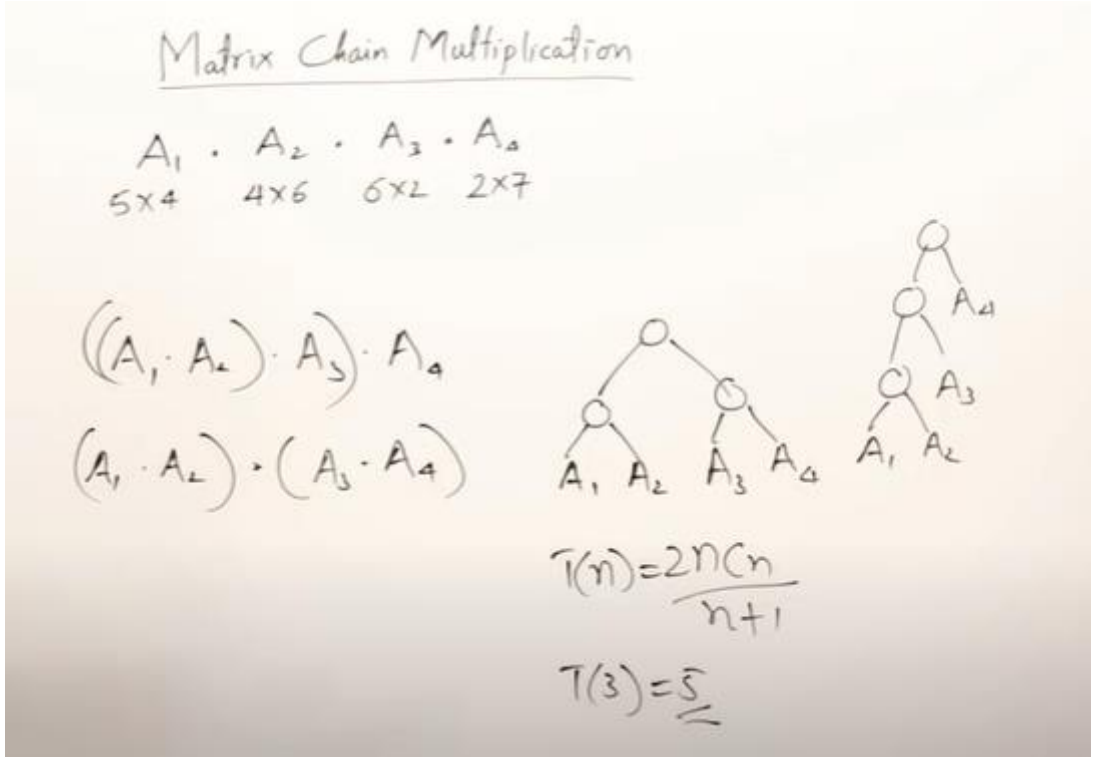
$d(1,1) = 3$

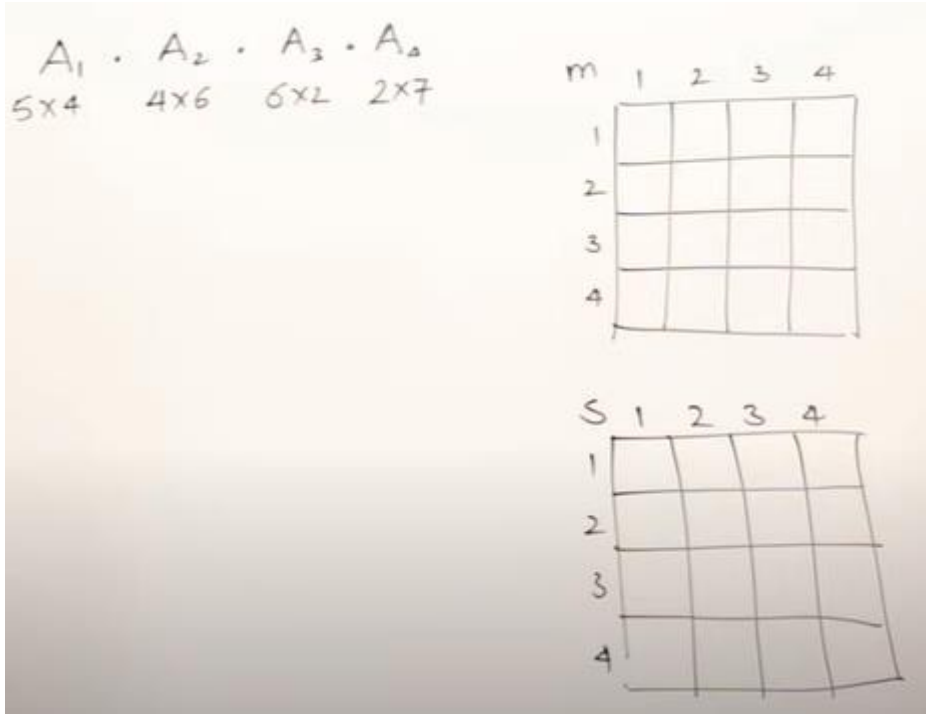
$d(2,3) = 6$

$d(3,6) = 10$

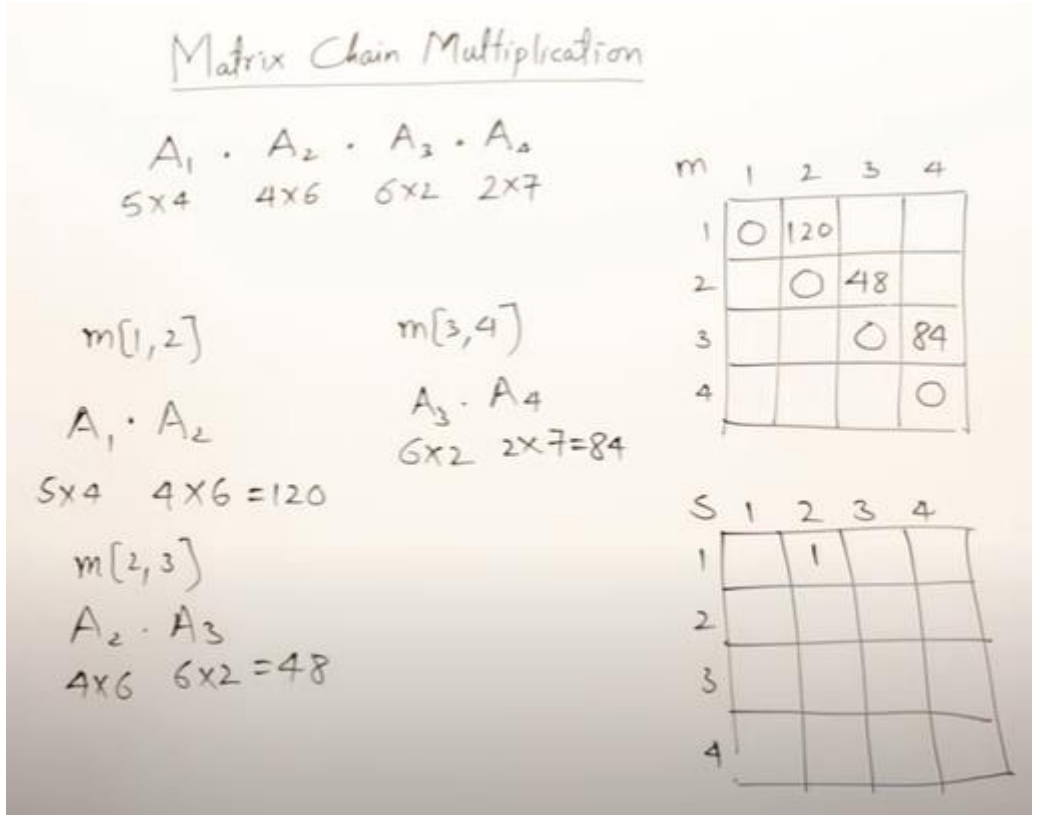
$d(4,10) = 12$

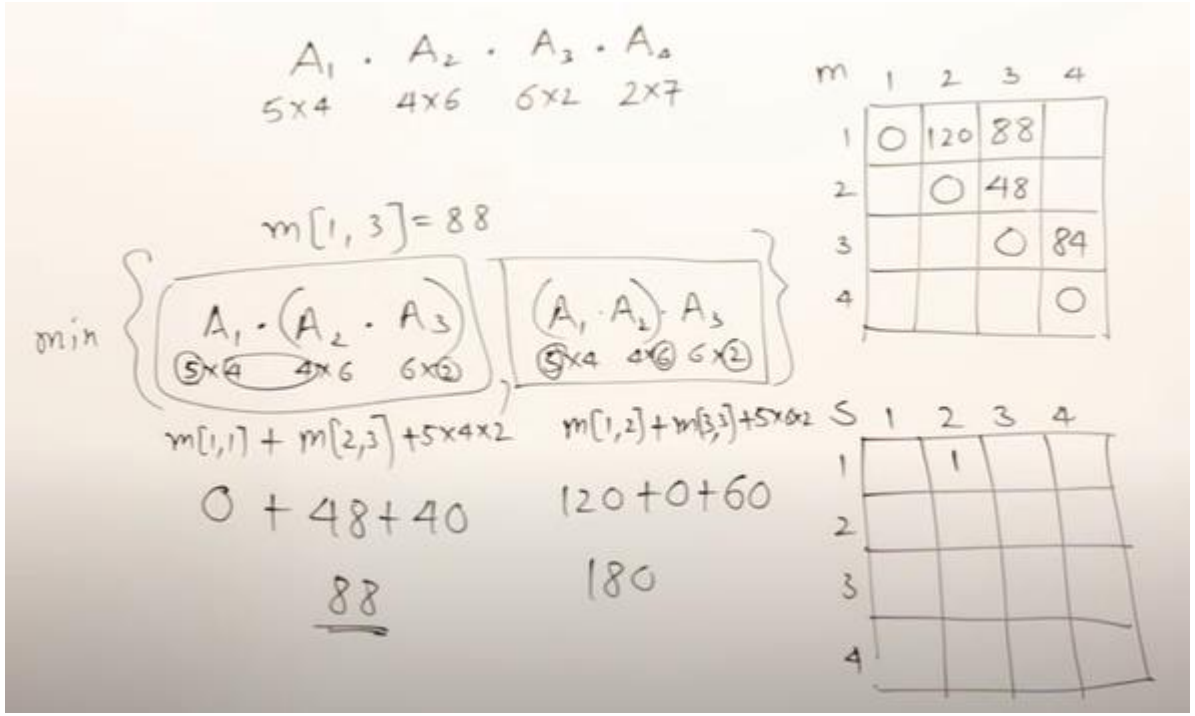
Matrix Chain Multiplication



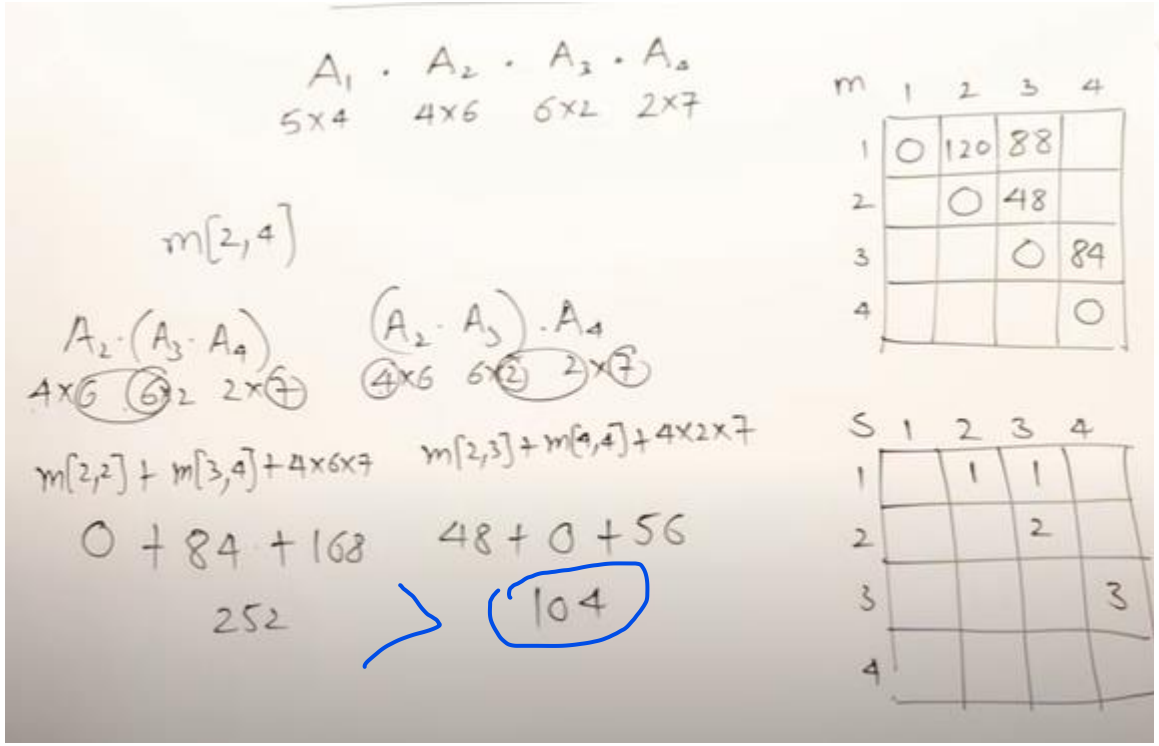


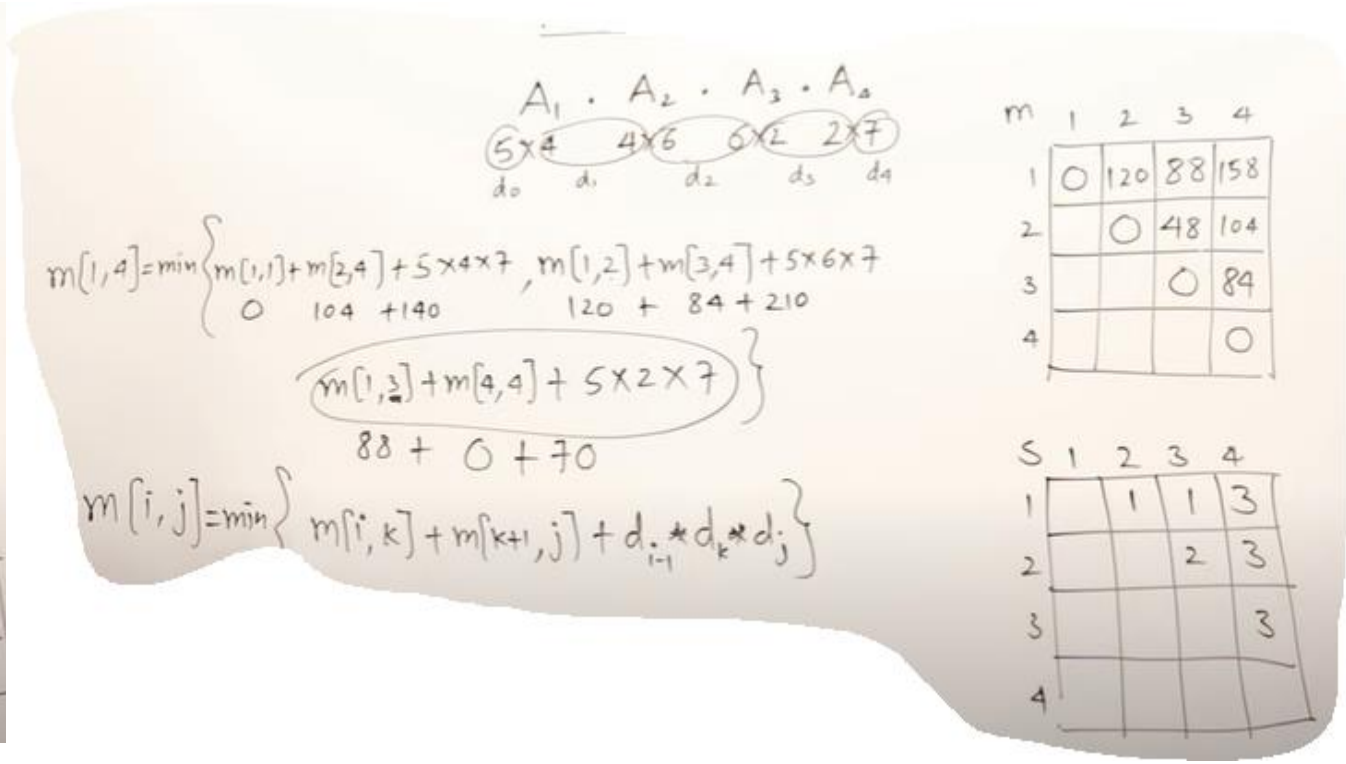
Matrix Chain Multiplication



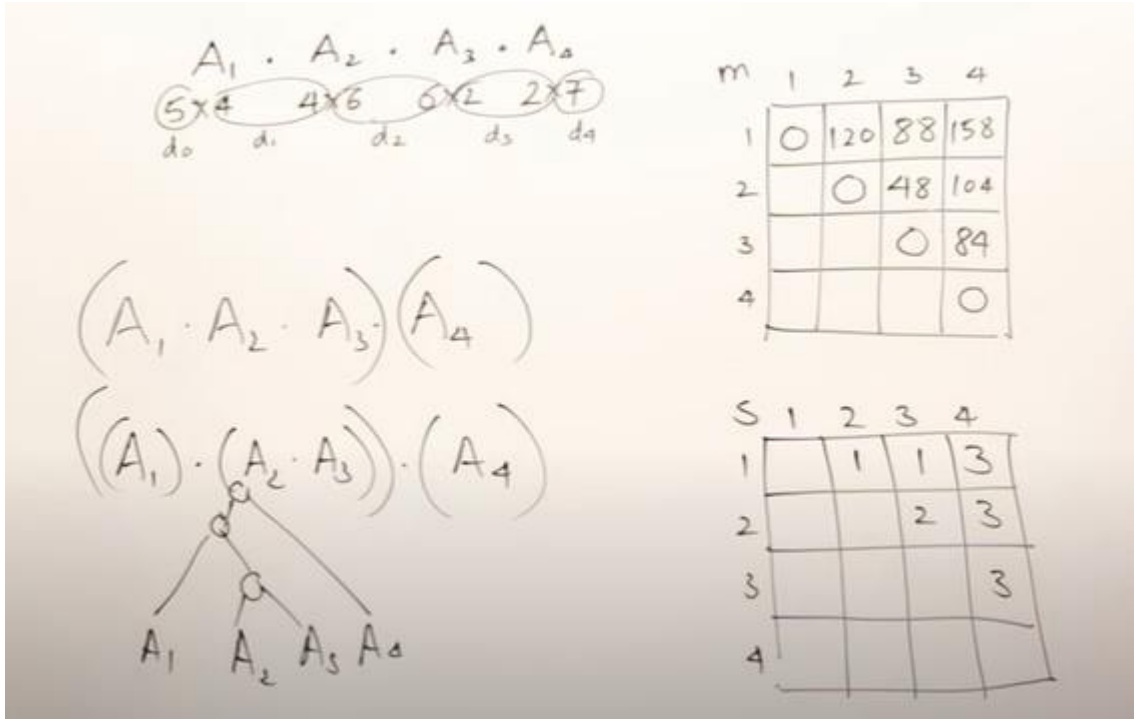


Matrix Chain Multiplication





Matrix Chain Multiplication



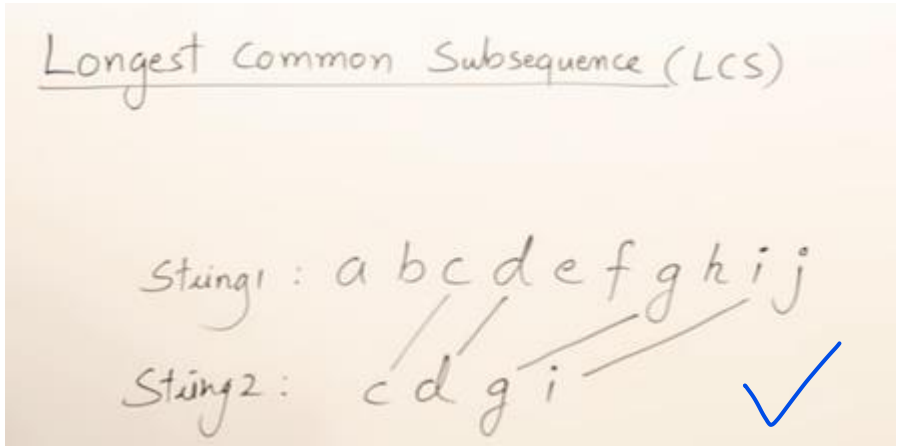
Time Complexity:

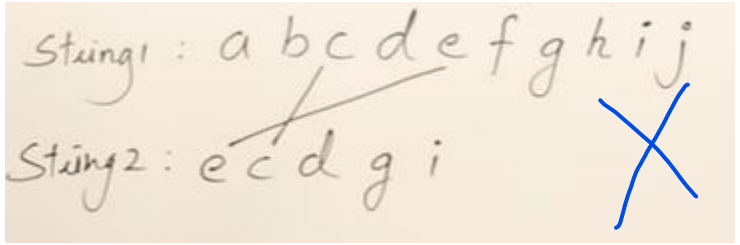
$$\frac{n(n-1)}{2} = n^2 \times n = n^3$$

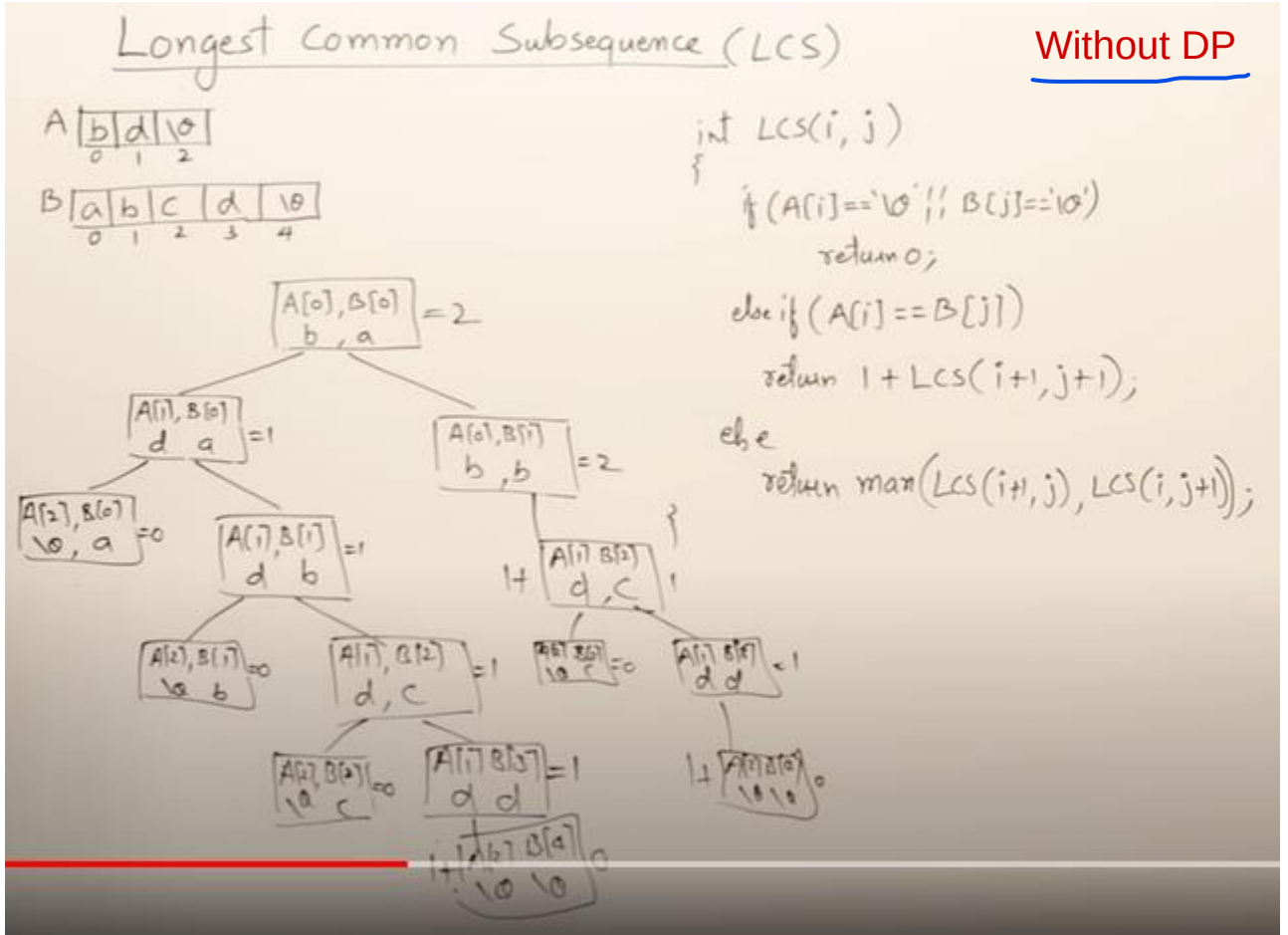
$$O(n^3)$$

LCS

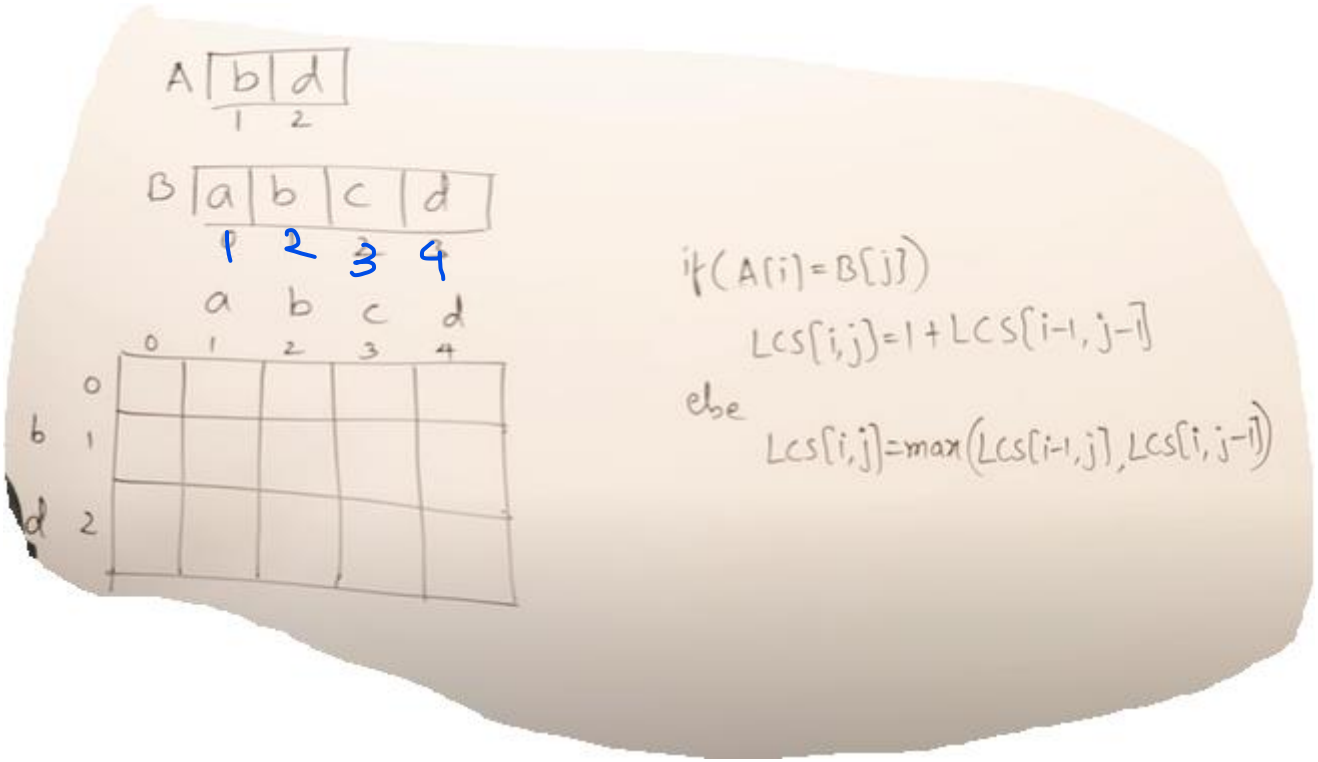
Longest Common Subsequence (LCS)

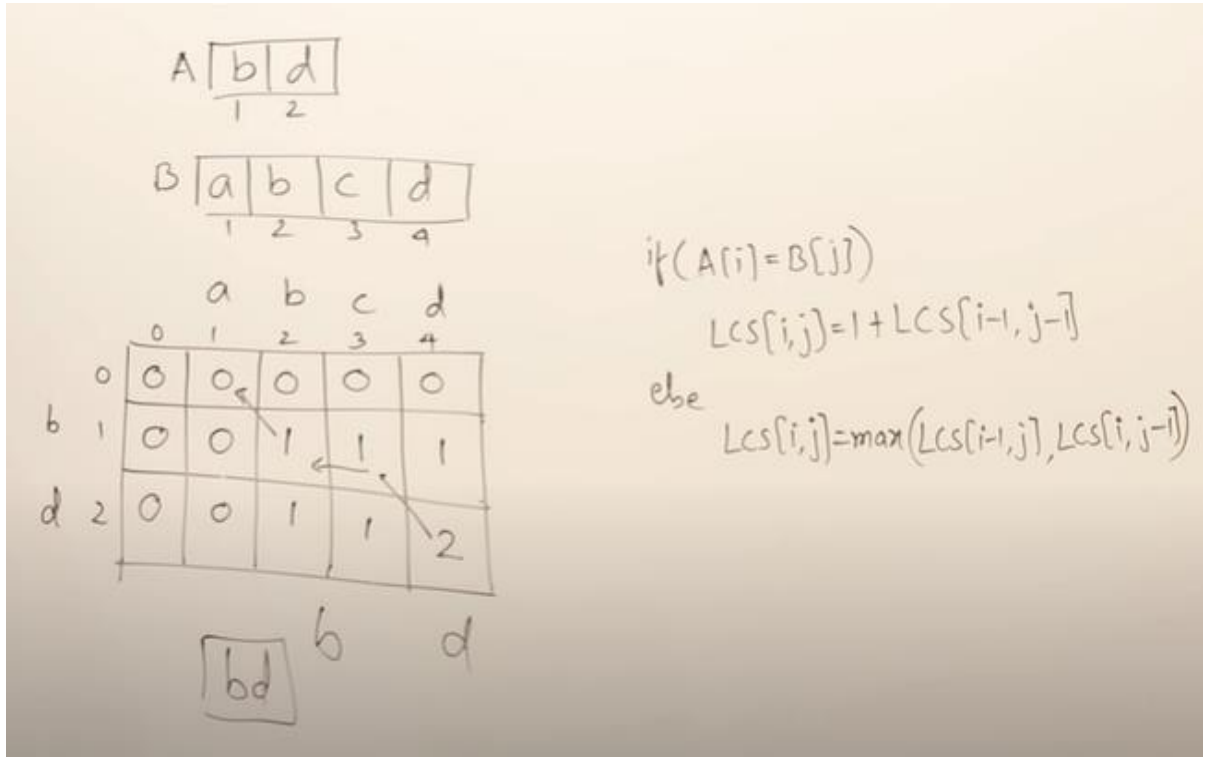




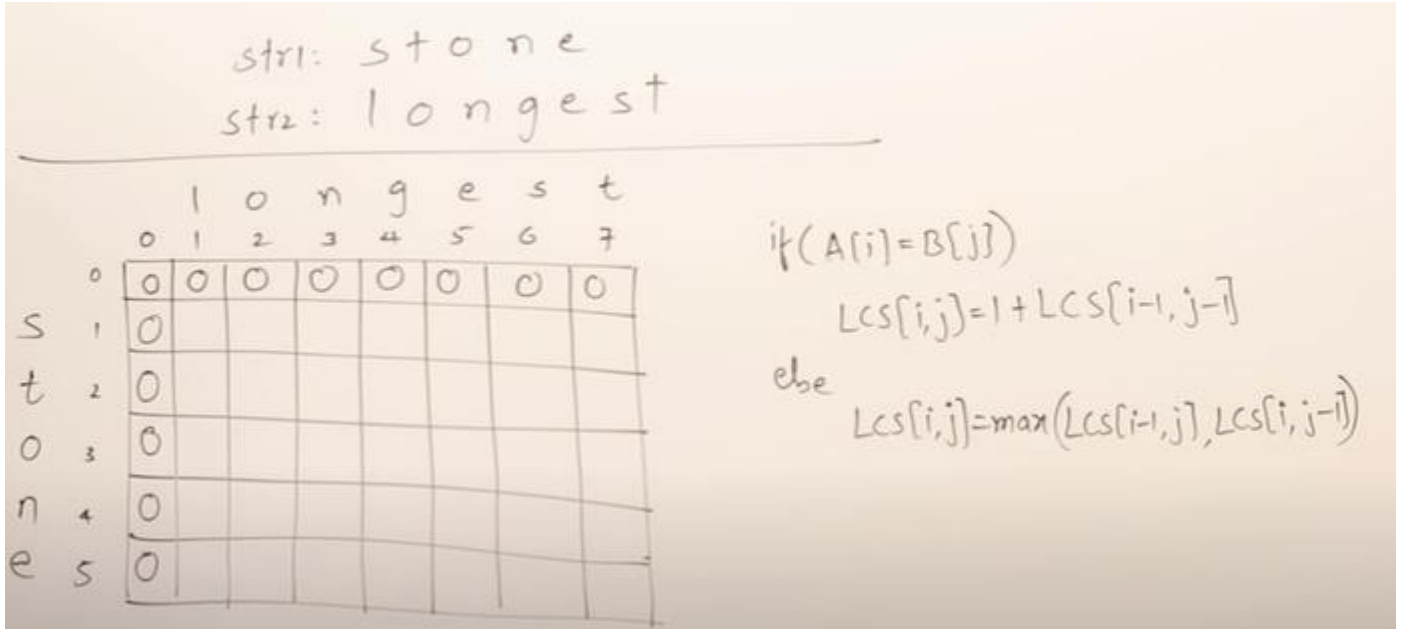


LCS

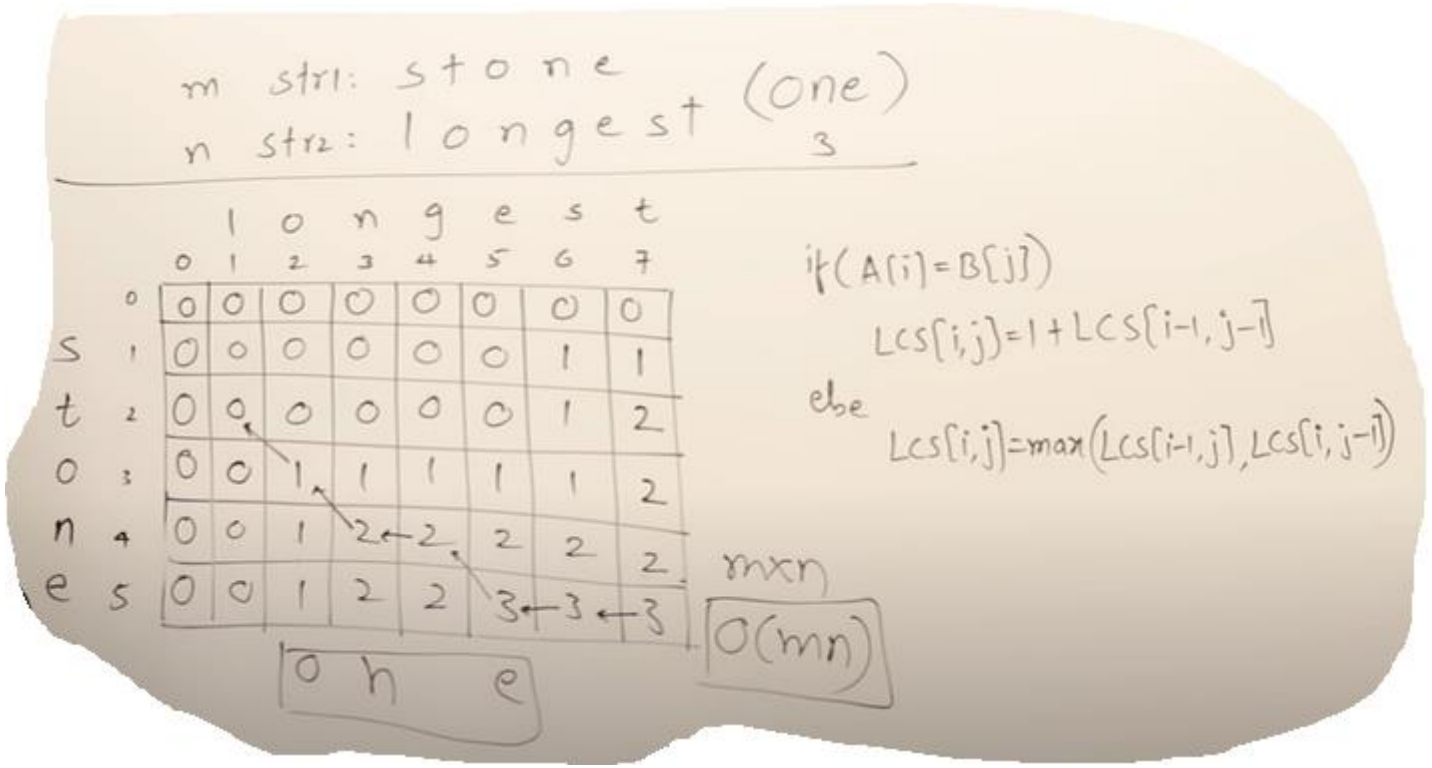

A b d
1 2
B a b c d
1 2 3 4
a b c d
0 1 2 3 4
0
b 1
d 2
if(A[i]=B[j])
LCS[i,j]=1+LCS[i-1,j-1]
else
LCS[i,j]=max(LCS[i-1,j],LCS[i,j-1])


A b d
1 2
B a b c d
1 2 3 4
a b c d
0 1 2 3 4
0 0 0 0 0 0
b 1 0 0 1 1 1
d 2 0 0 1 1 2
b d
bd
if(A[i]=B[j])
LCS[i,j]=1+LCS[i-1,j-1]
else
LCS[i,j]=max(LCS[i-1,j],LCS[i,j-1])

LCS


str1: stone
str2: longest
l o n g e s t
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0 0
s 1 0
t 2 0
o 3 0
n 4 0
e 5 0
if(A[i]=B[j])
Lcs[i,j]=1+Lcs[i-1,j-1]
else
Lcs[i,j]=max(Lcs[i-1,j],Lcs[i,j-1])

LCS


m str1: stone
n str2: longest (one)
3
longest
0 1 2 3 4 5 6 7
s t o n e
if(A[i]=B[j])
LCS[i,j]=1+LCS[i-1,j-1]
else
LCS[i,j]=max(LCS[i-1,j], LCS[i,j-1])
mxn
O(mn)
one

0/1 Knapsack Problem: 2 Methods (DP)

Breakdown of the formula:

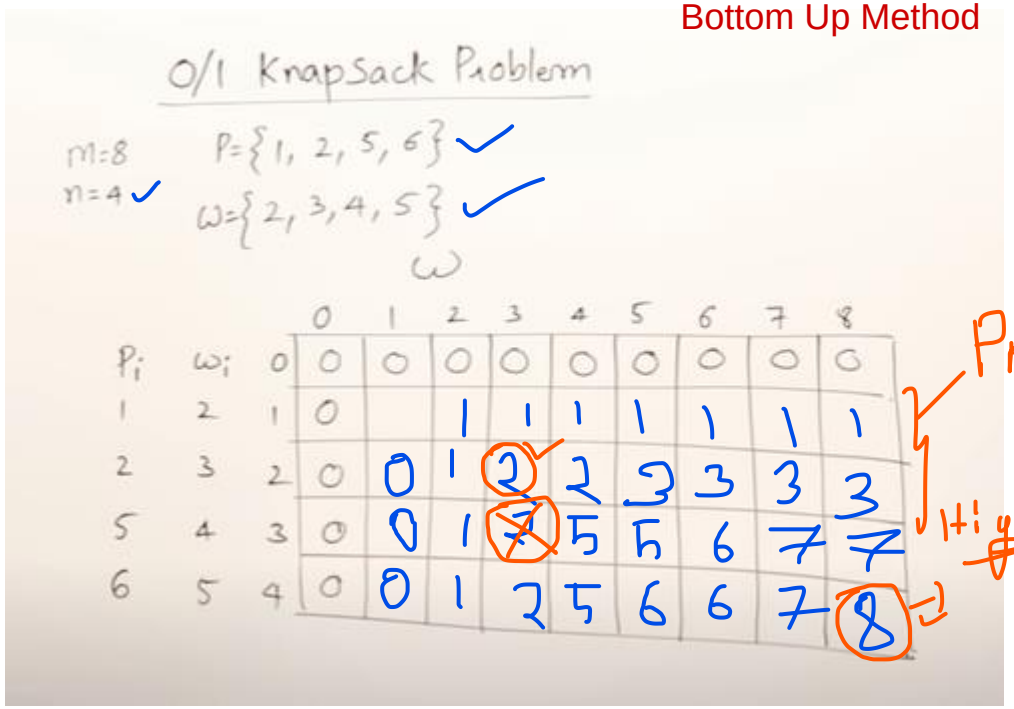
⭐V: DP table which stores values of subproblems

⭐V[i, w]: Maximum profit by considering the first 'i' elements in a bag of weight 'w'

⭐V[i-1, w]: Case when the current object is not included(~0) and the bag with current weight 'w' must be filled with the maximum profit possible (stored at 'i-1')

⭐V[i-1, w-w[i]] + p[i]: Case when the current object is included(~1) and the remaining part of the bag 'w-w[i]' must be filled with the maximum profit possible (stored at 'i-1')

Bottom Up Method

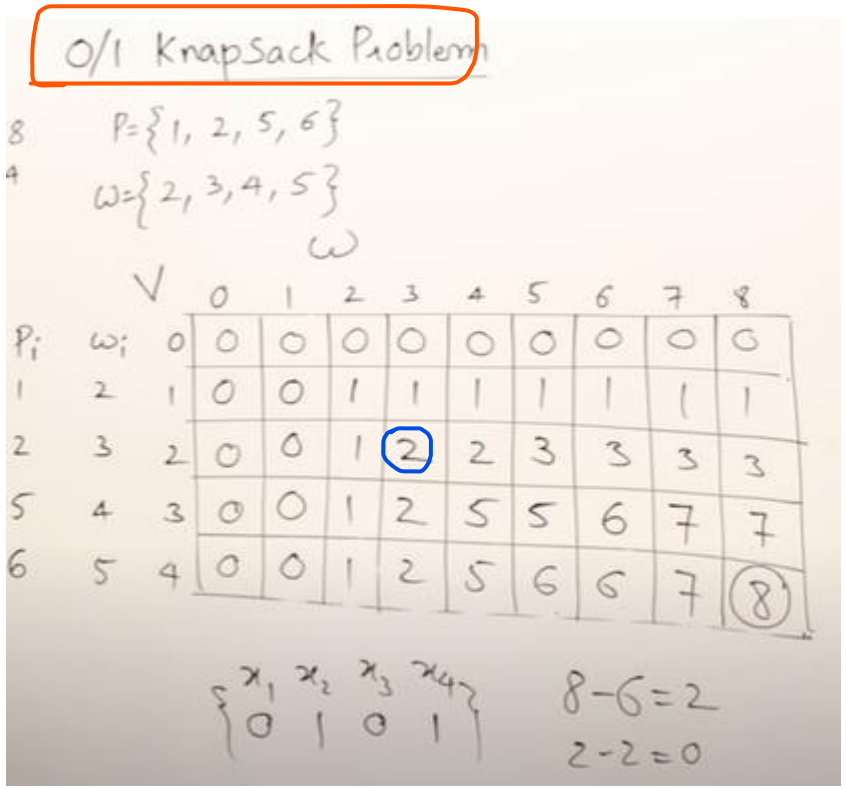


Formula, (Bottom Up Method)

$V[i,j] = \max\{V[i-1,w], V[i-1, w-w[i]+P[i]] \}$

Where w is the values of the rows on the top of the table, w[i] is the associated weights given in the question.

0/1 Knapsack Problem: 2 Methods



SET METHOD: