Dynamic Programming :-

- Typically apply to optimization problems
- A powerful technique to solve many diff. problems in poly time, for which a naive approach would take an exponential time.
- It is a general approach to solving problems
- ~~Similar, to~~ In DP, the subproblems can overlap.

- Basic Idea :-
  * Break a problem in a reasonable number of subproblems (may be n^2) s.t. the optimal sol^n to the smaller subproblems can be used to generate the op. sol^n for the whole problem.
  * We ask the same question to the subproblems over and over again, but instead of solving each subproblem, we solve it once and reuse the sol^n.

Rod Cutting :-

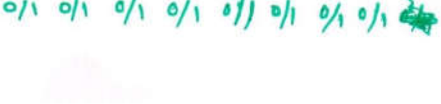
- Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue (r_n) obtainable by cutting the rod & selling the pieces.
  - If the price p_n for a rod of length n is large enough, an optimal sol$^{\underline{n}}$ may not require cutting at all.
- Ex:- If $n = 4$, How many diff. ways we can the rod of length 4 inches (including no cut at all). The cost ~~t~~able p_i is as follows:-

rod length l_i	1	2	3	4
cost c_i	1	5	8	9

	Diff. ways of cutting the rod of 4 inches	Cost
(a)	$1 + 3 = 4$	9
(b)	$2 + 2 = 4$	10
(c)	$3 + 1 = 4$	9
(d)	$1 + 1 + 2 = 4$	7
(e)	$1 + 2 + 1 = 4$	7
(f)	$2 + 1 + 1 = 4$	7
(g)	$1 + 1 + 1 + 1 = 4$	4

- optimal sol$^{\underline{n}}$ is :- cut the rod into two pieces each of 2 inches, to get max profit of <u>10</u>.

- We can cut up a rod of length n in 2^{n-1} diff ways. why ??
  → Note, for each $0 \leq i \leq n$, we have an independent option of cutting, or not cutting



- Despite the exponentially large possibility space, we can use DP to solve the problem in $\Theta(n^2)$.

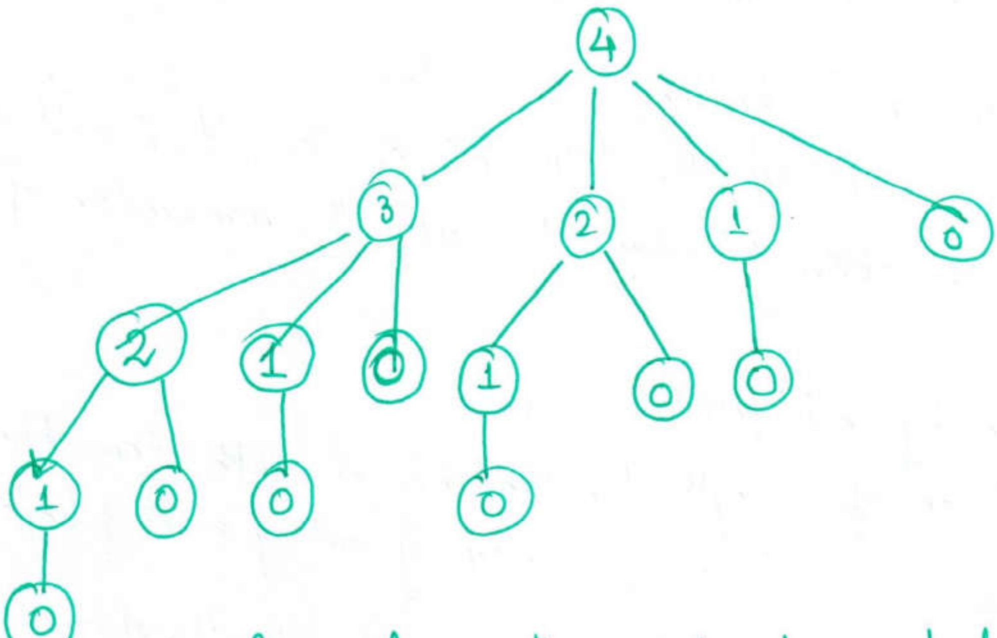
- We can decompose the problem as follows:-
  - First, cut a piece off the left end of the rod, & sell it.
  - Then, find the optimal way to cut the remainder of the rod.
  - That is, we try all possible cases:-
    - Cut a piece of length 1, combine it with the optimal way of cutting a rod of length $(n-1)$.
    - Cut a piece of length 2, combine it with the optimal way of cutting a rod of length $(n-2)$.
    - so, on.
  - We try all the possible lengths & then pick the best one ... AIRTEL !!
  - We obtain

$$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i} \right)$$

Naive Algo:- Recursive Top-Down implementation:-

```
CUT_ROD(p, n)
    if n == 0
        return 0
    q = -∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n-i))
    return q.
```

- following tree shows the recursive calls made for $n=4$:



- Path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting the rod of length n.
- In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

- from the ~~recurrence~~ recursion tree, we see that we are actually doing a lot of extra work.
  - computing the same thing again & again
  - like fibonacci numbers (recursive implementation)

— x —

Memoization (Top down approach) :-

- Idea :- store the result of recursive calls, reuse it if needed.
  - subproblems

```
MEMOIZED-CUT-ROD (p, n)
    Let r[0... n] be a new array
    for i = 0 to n                    ← initialize a new auxilary array
        r[i] = -∞
    return MEMOIZED-CUT-ROD-AUX (p, n, r)

MEMOIZED-CUT-ROD-AUX (p, n, r)        - memoized version of CUT-ROD algo.
    if r[n] ≥ 0
        return r[n]                   → checks to see if the desired value is already known
    if n == 0
        q = 0;
    else q = -∞
        for i = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n-i, r));
    r[n] = q
    return q.
```

= Runtime :- $\Theta(n^2)$

- Solves each subproblem exactly once.

—x—

Bottom up approach

- Proactively compute the soln for smaller rod first; assuming that they will be used later to solve for large rod.

- Algo:-

```
BOTTOM-UP-CUT-ROD (p,n)
    let r[0...n] be a new array
    r[0] = 0
    for j = 1 to n
        q = -∞
        for i = 1 to j
            q = max(q, p[i] + r[j-i])
        r[j] = q
    return r[n]
```

- Time complexity = $\Theta(n^2)$; double "for" loop.

- No recursion
- The ~~problem~~ Algo solves subproblems of size $j = 0, 1, \ldots, n$ in that order.

- **Example**:- Assume that we have a rod of size 4. (say)

Prices of diff sizes of rod is shown below:-

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

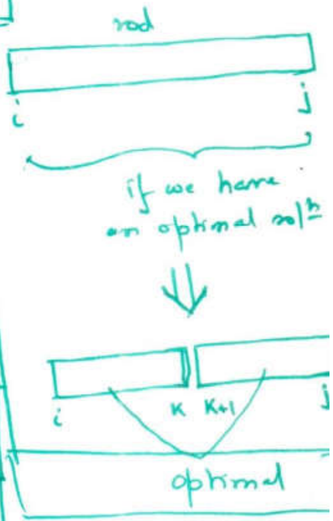
- if the rod is size 4, then possible combinations are :-
  - $1+1+1+1 = 4 \longrightarrow$ cost $= 4$.
  - $2+2 = 4 \longrightarrow$ cost $= 10$ ⟵ most value
  - $1+3 = 4 \longrightarrow$ cost $= 8$
  - $1+1+2 = 4 \longrightarrow$ cost $= 7$
  - $4 = 4 \longrightarrow$ cost $= 9$
- Remaining ways of cutting the rod is some permutation of the above.

—— x ——

Optimal substructure of rod-cutting problem :-

Substructure :- optimal sol^n to the problem incorporate optimal sol^n to related subproblems, which can be solved independently.

⟶ $\boxed{\text{Proof by contradiction}}$



—— x ——

Sol^n

In the algo., $r[i]$ is the price of the optimal cut until i.

i	0	1	2	3	4	5	6	7	8
p_i	0	1	5	8	9	10	17	17	20
r_i	0	1	5	8	10	13	17	18	22

$$r[1] = \max(-\infty, 1 + \cancel{r[0]}^{\,0}) = 1$$

$$r[2] = \max \begin{cases} p[1] + r[2-1] = 2 \\ p[2] + r[2-2] = 5 \end{cases}$$

$$r[3] = \max \begin{cases} p[1] + r[2] = 1+5 = 6 \\ p[2] + r[1] = 5+1 = 6 \\ p[3] = 8 \end{cases}$$

$$r[4] = \max \begin{cases} p[1] + r[3] = 9 \\ p[2] + r[2] = 10 \\ p[3] + r[1] = 9 \\ p[4] = 9 \end{cases}$$

$$c[5] = \max \begin{cases} p[1] + r[4] = 11 \\ p[2] + r[3] = \underline{13} \leftarrow \\ p[3] + r[2] = 13 \\ p[4] + r[1] = 10 \\ p[5] = 10 \end{cases}$$

$$c[6] = \max \begin{cases} p[1] + r[5] = 14 \\ p[2] + r[4] = 15 \\ p[3] + r[3] = 16 \\ p[4] + r[2] = 14 \\ p[5] + r[1] = 11 \\ p[6] + r[0] = 17 \leftarrow \end{cases}$$

no cut
optimal for rod of size 6

— x —

Matrix-chain Multiplication :-

→ Assume we have two matrices $A_{4\times 6}$ and $B_{6\times 3}$

→ So, if we multiply the two :-

$$C_{4\times 3} = A_{4\times 6} \times B_{6\times 3}$$

→ Total number of multiplications :- $4 \times 6 \times 3 = 72$

→ Now, consider the problem of multiplying the following :-

A_1	A_2	A_3
10×100	100×5	5×50.

- If we multiply these matrices in the following order.

(a) $((A_1 A_2) A_3)$ ⟶ # of multiplications

$(10 \times 100 \times 5) = 5000$

$(10 \times 5 * 50) = 2500$

$= 7500$ mult.

(b) $(A_1 (A_2 A_3))$ ⟶ $100 \times 5 \times 50 + 10 \times 100 \times 50$

$= 25{,}000 + 50{,}000$

$= 75{,}000$ multi

→ **Matrix Chain Multiplication Problem :-**

Given a chain of n matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$.

→ full paranthesize the product $A_1, A_2, \ldots, A_n$ s.t. it minimizes the total number of multiplication.

Optimal Substructure for Matrix chain Multiplication.

Lemma.

Suppose, we have an optimal solⁿ for ~~$A_{i..j}$~~ multiplying matrices from $A_i, A_{i+1}, \ldots, A_j$.

Also assume that that solⁿ has the following parenthesis

$$\left(A_i A_{i+1} \cdots A_k\right)\left(A_{k+1} \cdots A_j\right)$$

Then, the way we parenthesize the prefix subchain $A_i \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be the optimal parenthesization of $A_i A_{j+1} \cdots A_k$.

Proof: – By contradiction. – If not then we get a better solⁿ for $A_i A_{i+1} \cdots A_j$ $\rightarrow$ contradiction.

– Homework

Recursive Solution :-

- Let $m[i,j]$ be the min number of multiplications required to to compute $A_{i \cdots j}$
- We define $m[i,j]$ recursively, as follows :-

$$m[i,j] = 0 \quad \text{if } i = j$$

$$m[i,j] = \min_{i \le k < j} \{ \underbrace{m[i,k]}_{\text{cost of multiplying } A_{i\ldots k}} + \underbrace{m[k+1,j]}_{\text{cost of multiplying } A_{k+1\cdots j}} + \underbrace{p_{i-1} p_k p_j}_{\text{cost of multiply the subproblem } A_{i\ldots k}\, A_{k+1\cdots j}} \} \quad \text{if } i < j$$

- ~~Since~~, We donot know the value of 'k'.
- There are only $j-i$ possible values of k, i.e., $k = i, i+1, \ldots, j-1$.
- We need to check them all to find the best.

So,

$$m[i,j] = \begin{cases} 0 & \text{if } i=0 \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i \leq j \end{cases}$$

Example:- Consider the following matrices :-

A_1	A_2	A_3	A_4	A_5
4×10	10×3	3×12	12×20	20×7
$p_0 \times p_1$	$p_1 \times p_2$	$p_2 \times p_3$	$p_3 \times p_4$	$p_4 \times p_5$

- We start with the case when $i=j$.
- Then we compute $m[i,j]$ for $i<j$ where the diff between i & j is 1.
- Next, we do the same for when i & j has a spread of 2
- and so on.

i \ j	1	2	3	4	5
1	0	120	264	1080	1344
2	x	0	360	1320	1350
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

cases where $|i-j|=2$

cases where $|i-j|=1$

cases when $i=j$.

cases when $i>j$

spread $(i<j)$ of 1

$m[1,2] = \min\limits_{1 \le k < 3} \{ m[1,1] + m[2,2] + 4 \times 10 \times 3 = 0+0+4 \times 10 \times 3 = 120.$

$m[2,3] = \min\limits_{2 \le k < 3} \{ m[2,2] + m[3,3] + 10 \times 3 \times 12 = 0+0+360 = 360$

$m[3,4] = \min \{ 0+0+3 \times 12 \times 20 = 720$

$m[4,4] = \min\limits_{4 \le k < 5} \{ 0+0+12 \times 20 \times 7 = 1680$

Spread of 2 :-

$$m[1,3] = \min_{1 \le k < 3} \begin{cases} m[1,1] + m[2,3] + 4 \times 10 \times 12 = 840 \\ m[1,2] + m[3,3] + 4 \times 3 \times 12 = 264. \end{cases}$$

$$m[2,4] = \min_{2 \le k < 4} \begin{cases} m[2,2] + m[3,4] + 10 \times 3 \times 20 = 0 + 720 + 10 \times 3 \times 20 \\ \qquad = 1320. \\ \underbrace{m[2,3]}_{360} + \underbrace{m[4,4]}_{0} + 10 \times 12 \times 20 = \underline{2760} \end{cases}$$

$$m[3,5] = \min_{3 \le k < 5} \begin{cases} m[3,3] + m[4,5] + 3 \times 12 \times 7 = 1932 \\ m[3,4] + m[5,5] + 3 \times 20 \times 7 = \underline{\underline{1140}}. \end{cases}$$

Spread of 3

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} m[1,1] + m[2,4] + 4 \times 10 \times 20 = 2120 \\ m[1,2] + m[3,4] + 4 \times 3 \times 20 = 1080 \\ m[1,3] + m[4,4] + 4 \times 12 \times 20 = \underline{1224} \end{cases}$$

$$m[2,5] = \min_{2 \le k < 5} \begin{cases} m[2,2] + m(3,5] + 10 \times 3 \times 7 = 1350 \\ m[2,3] + m(4,5) + 10 \times 12 \times 7 = 2880 \\ m(2,4) + m(5,5) + 10 \times 20 \times 7 = 2720. \end{cases}$$

Similarly, spread of 4

$$m[1,5] = \min_{1 \le k < 5} \begin{cases} m[1,1] + m[2,5] + 4 \times 10 \times 7 = 1630 \\ m[1,2] + m[3,5] + 4 \times 3 \times 7 = 1344 \\ m[1,3] + m[4,5] + 4 \times 12 \times 7 = 2280 \\ m[1,4] + m[5,5] + 4 \times 20 \times 7 = \underline{1640} \end{cases}$$

- So we can multiply A_1 to A_5 in 1344 multiplications.
- In which order ??
  - → we need to keep track of the 'k' value.

for example :-

- when we solve for $m[1,5]$, we find that the optimal sol^n is for $\underline{k=2}$.

$$\boxed{\text{i.e., } m[1,5] = m[1,2] + m[3,5] + p_0 p_2 p_5}$$

So, we put a bracket between A_2 & A_3, i.e.,

$$(A_1 \; A_2)\;(A_3 \; A_4 \; A_5)$$

- Then, we look for subproblems $m[1,2]$ & $m[3,5]$ used for solving $m[1,5]$.

- Can we put a bracken in subproblem $m[1,2]$.
  → no, only two matrices.

- what about $m[3,5]$.
  - see, the sol^n to find we find a min fat $k=4$.
  - we have :-

$$- \quad (A_1 \; A_2)\,((A_3 \; A_4)\; A_5)$$

- Check the above sol^n :-

$$\underbrace{\underbrace{(A_1 \; A_2)}_{120}\underbrace{\underbrace{((A_3 \; A_4)}_{720}\; A_5)}_{420}}_{84}$$

⇒ Total multiplication

$\underline{1344}$

- **Algo** :-

```
MATRIX-CHAIN-ORDER (p)
    n = p.length - 1
    Let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables.
    for i = 1 to n
        m[i,i] = 0
    for l = 2 to n                // l is the chain length.
        for i = 1 to n-l+1
            j = i + l - 1
            m[i,j] = ∞
            for k = i to j-1
                q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
                if q < m[i,j]
                    m[i,j] = q
                    s[i,j] = k
    return m and s.
```

- Runtime is $O(n^3)$.

Algo is nested three deep, and each loop indexed (l, i, and k) takes on at most $n-1$ values

- space complexity is $\Theta(n^2)$.

Longest Common Subsequence :-

- A subsequence of a string S, is a set of characters, that appear in left-to-right order, but not necessarily consecutively.

  Example :- ACTTGCG

  - ACT, ATTC, T, ACTTGC are all subsequences
  - TTA is not a subsequence.

- A common subsequence of two strings is a subsequence that appears in both strings.
- A longest common subsequence is a common subsequence of maximum length.
- Example :-

  S_1 = AAACCGTGAGTTATTCGTTCTAGAA

  S_2 = CACCCCTAAGGTACCTTTGGTTC

  LCS is ACCTAGTACTTTG

- Has applications in many areas including biology.
- Brute-force Soln :- Try all possible subsequence from one string, and search for matches in the other string.
  - # of ~~abc~~ ~~substring~~ subsequence of a string is 2^m. (m is the length of the string.
  - Exponential number of possible subsequence.

Optimal substructure of an LCS :-

Theorem :- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, in this case x_m & y_n cannot both be in the LCS. Thus, either x_m is not part of the LCS, or y_n is not part of the LCS (or possibly both are not part of LCS).
   (a) if $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
   (b) if $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof (1) :- If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length k+1.
→ contradiction to our assumption that Z (of length k) is the LCS of X and Y.

Thus, we must have $z_k = x_m = y_n$.

Next :- the prefix Z_{k-1} is a length (k-1) common subsequence of X_{m-1} and Y_{n-1}. We have to show that it is LCS.

Suppose, for the purpose of contradiction that there is a common subsequence W of X_{m-1} and Y_{n-1} with length greater than k-1.

Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y, whose length is greater than k, which is a contradiction.

Proof 2 :- If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y.

If there were a common sub-seq. W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

Proof 3 :- Proof is symmetric to 2.

— x —

Recursive Solution :-

$X = \langle x_1, x_2, \ldots, x_m \rangle$
$Y = \langle y_1, y_2, \ldots, y_n \rangle$

- from the above theorem, to find the LCS of
  - we should either examine one subproblem, i.e., if $x_m = y_n \longrightarrow$ we find LCS of X_{m-1} & Y_{n-1}
  - OR, examine two subproblems i.e., if $x_m \neq y_n \longrightarrow$ we find LCS of X_{m-1} and Y and LCS of X and Y_{n-1} and find the max LCS of the two

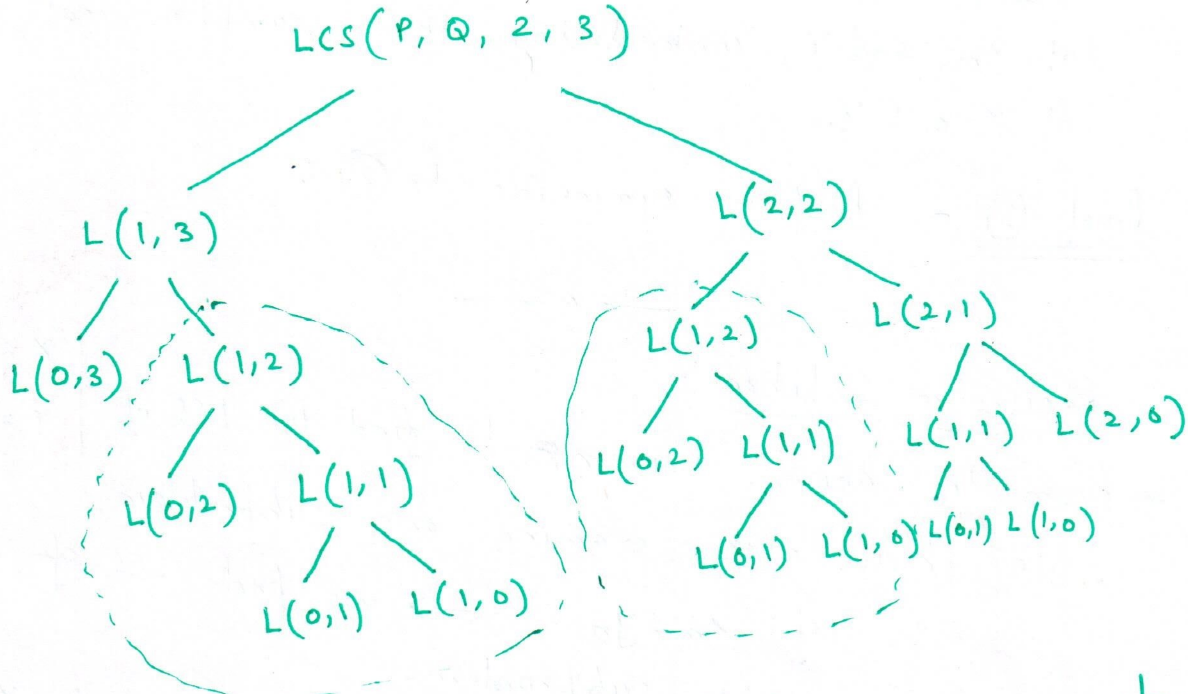
- The optimal substructure of LCS problem gives the following recursive formula:-

$$\underbrace{c[i,j]}_{\text{The length of an LCS of the sequences } X_i \,\&\, Y_j} = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \;\&\; x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j>0 \;\&\; x_i \neq y_j \end{cases}$$

Overlapping subproblems :-

Ex :- Let's consider two strings

$P =$ "A A" $\qquad Q =$ "B B B"



- We could easily write an exponential-time recursive algo to compute the length of an LCS of two ~~subse~~ sequences.
- However, LCS has only $\Theta(m.n)$ distinct subproblems.
  - so use dynamic programming. to compute the solution bottom-up.
- Memoization can improve the performance. to $\Theta(m.n)$.

// initialize arr[n][m] to undefined.

LCS-Memoized(X, Y}, n, m) {

~~m = X.length;~~

~~n = Y.length;~~

if arr[n][m] != undefined

return arr[n][m].

if (n == 0) or (m == 0) return 0 ;

else if (X[n] == Y[m]) return 1+ LCS-Memoized(X, Y, n-1, m-1)

else return max(LCS-Memo(X, Y, n-1, m), LCS-Memo(X, Y, n, m-1))

}

$\boxed{\Theta(m.n)}$

LCS using Dynamic Programming (Bottom-up):

Algo:-

```
LCS-LENGTH (X, Y)
    m = X.length
    n = Y.length
    let b[1..m, 1..n] and c[0..m, 0..n] be new tables.

    for i = 1 to m
        c[i, 0] = 0

    for j = 0 to n
        c[0, j] = 0

    for i = 1 to m
        for j = 1 to n
            if x_i == y_j
                c[i, j] = c[i-1, j-1] + 1
                b[i, j] = "↖"
            else if c[i-1, j] ≥ c[i, j-1]
                c[i, j] = c[i-1, j]
                b[i, j] = "↑"
            else c[i, j] = c[i, j-1]
                b[i, j] = "←"
    return c and b.
```

Run-time complexity is $\Theta(m.n)$

↳ each table entry takes $\Theta(1)$ to compute.

Example :-

Let $X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$.

find LCS.

	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

- if $x_i = y_j$ $\longrightarrow$ top left diagonal $+1$.
- else $\longrightarrow$ max of (top and left).

- Constructing LCS :- (Algo. from the book)
  - whenever we encounter "$\nwarrow$" $\Rightarrow$ $x_i = y_j$ is an element in LCS.
  - We encounter the elements of LCS in reverse order.
  - Move from bottom rigth (i.e. 4) to top-left (0) and print the char corresponding to "$\nwarrow$".
  - So LCS :- $\langle B\ C\ B\ A \rangle$
  - The procedure takes $O(m+n)$ time.

as it decrements one of i & j in each recursive call

Bellman-ford Algorithm

- Richard Bellman and Lester ford, Jr., ~~published the algo~~ worked on the same problem, separately and published it in 1958 and 1956, respectively.

- Solves the sortest-path problem in the general case in which edge weights may be negative.

- Algorithm returns a boolean value indicating whether or not there is a -ve weight cycle that is reachable from the source.
  - In such a case, (i.e., -ve weight cycles) the algo indicates no solution. Else, algo. produces the shortest path & their weights.

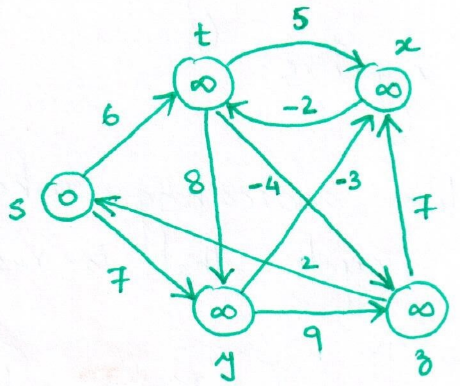
- Algo:-

```
BELLMAN-FORD (G, w, s)
    INITIALIZE (G, s);
    for i = 1 to |V| - 1                      ...... O(V)
        for each edge (u,v) ∈ E   } .... O(E)
            RELAX (u, v, w)
    for each edge (u,v) ∈ E
        if v.d > u.d + w(u,v)
            return FALSE
    return TRUE
```

```
RELAX (u, v, w)
    if (d[v] > d[u] + w(u,v))
        d[v] =
        π[v] = u
```

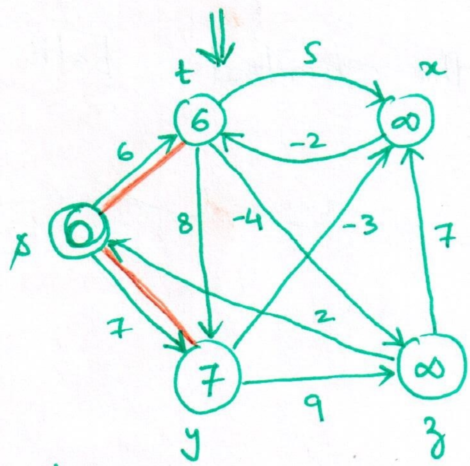
- Total complexity : $O(VE + E) = O(VE)$

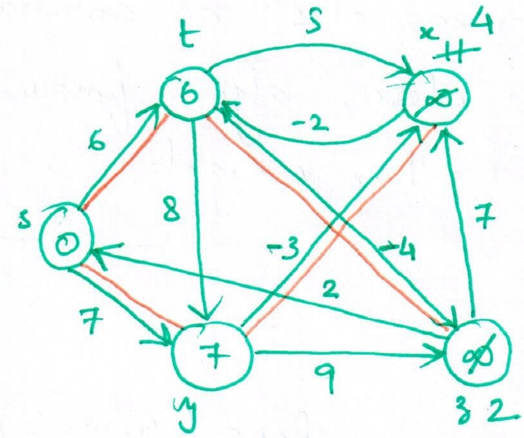
- It is slower than Dijkstra's algorithm for the same problem, but ~~more~~ can handle -ve weight edges in a graph.
- follows Dynamic Programming :- - Try out all possible solⁿ & pick the best one.
  - Relax all edges. (for V-1 times)
- Example :-



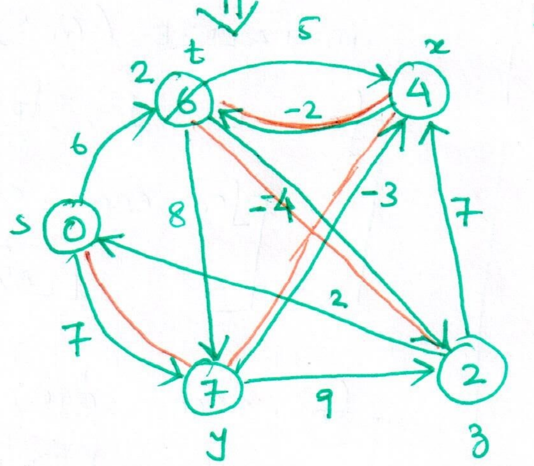
(a) Initialize(G, s)

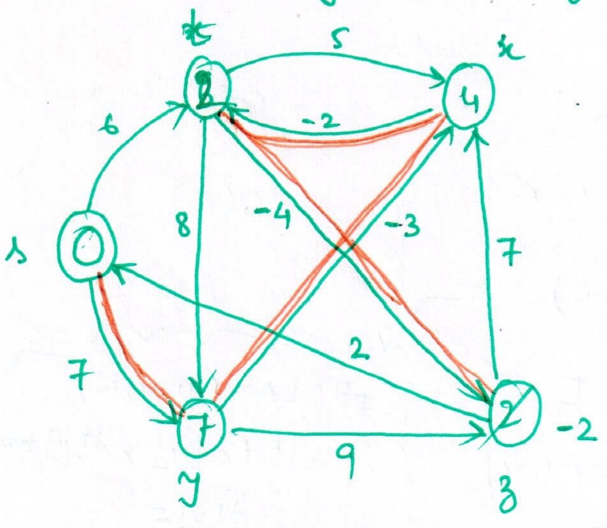
- Relax all edges in the following sequence.
- Keep the edges starting from s at the last, (to help the iteration)





$(t, x) = 5$
$(t, y) = 8$
$(t, z) = -4$
$(x, t) = -2$
$(y, x) = -3$
$(y, z) = 9$
$(z, x) = 7$
$(z, s) = 2$
$(s, t) = 6$
$(s, y) = 7$





- # iterations $= V - 1 = 5 - 1 = 4$.

$\therefore$ Complexity $= O(VE)$.

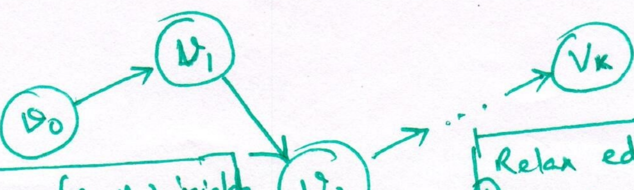
Lemma:- Let $G(V, E)$ be a weighted, directed graph with source 's' and weight function $w: E \rightarrow \mathbb{R}$, and assume G contains no negative-weight cycles that are reachable from 's'.

Then, after $|V|-1$ iteration of the for loop of BF-Algo, we have $d[v] = \delta(s, v)$ for all vertices v that are reachable from 's'.

Corollary:- ~~If a value for each vertex $v \in V$, there is a~~ ~~p~~ If the value $d[v]$ fails to converge after $|V|-1$ passes, then $\exists$ a −ve edge cycle reachable from s.

Proof (of Lemma):-

- Use path relaxation property.
- Consider any vertex v, that is reachable from 's'.
- and let, $p = \langle v_0, v_1, \ldots, v_K \rangle$ where $v_0 = s$ and $v_K = v$. be any shortest path from s to v.
- As, shortest path are simple (no loop), p has at most $|V|-1$ edges, i.e., $k \leq |V| - 1$.
  otherwise we will have loop
- Each of the $|V|-1$ iterations of the "for" loop in BF-algo relaxes all $|E|$ edges.



Thus, relaxing (v_0, v_1) yields $\delta(s, v_1)$ after the first pass

(*) At some pt in iter 1, we will Relax edge (v_0, v_1).

(*) Due to optimal substructure, if $(v_0 \ldots v_K)$ is the SP then v_0, v_1 is also a SP.

Idea of BF:- At each pass we move closer to v_K, while constructing the SP.

i.e., Iter. 1 $\Rightarrow \delta(s, v_1)$

Iter 2 $\Rightarrow \delta(s, v_2)$

... so on ...

- Thus, after 1st iteration of all edges E, we have $d[v_1] = \delta(s, v_1)$, as we will relax edge $(v_0, v_1) \in E$ in this pass.

- Similarly in the second iteration we get $d[v_2] = \delta(s, v_2)$ as we relax the edge (v_1, v_2).

- After k passes, $d[v_k] = \delta(s, v_k)$
- So, after $|V|-1$ passes, all reachable vertices will have a δ value.

—x—

<u>Proof (Corollary)</u>:

- If $\exists$ an edge in the graph that can be relaxed after $|V|-1$ passes...

  It implies that the current shortest path from s to v is not simple, as we have a repeating vertex.

- found a cycle of $-ve$ weight.

—x—