Chapter-15

Dynamic programming

Cormen et al.

Divide-and-Conquer	Dynamic programming
similarity	
① Solves problems by combining the solutions to subproblems.	① Solves problems by combining the soln to subproblems.
Differences	
② Subproblems are independent	② Subproblems are not independent
③ It solves some subproblems many times (more work)	③ It solves every subproblem only once. (less work)

Dynamic programming is used/applied to optimization problems.

Steps in Dynamic Programming/Development of DP.

① Characterize the structure of an optimal soln.
② Recursively define the value of an optimal soln
③ Compute the value of an optimal soln in bottom-up fashion
④ Construct an optimal soln from computed information.

→ It finds the value
→ It shows the complete soln.

① Assembly-line scheduling

Description - It consist of

(i) Assembly lines e.g. line 1, line 2

(ii) Stations e.g. station 1, 2, ...

Let $S_{i,j}$ = jth station on line i

let $i = 1, 2$

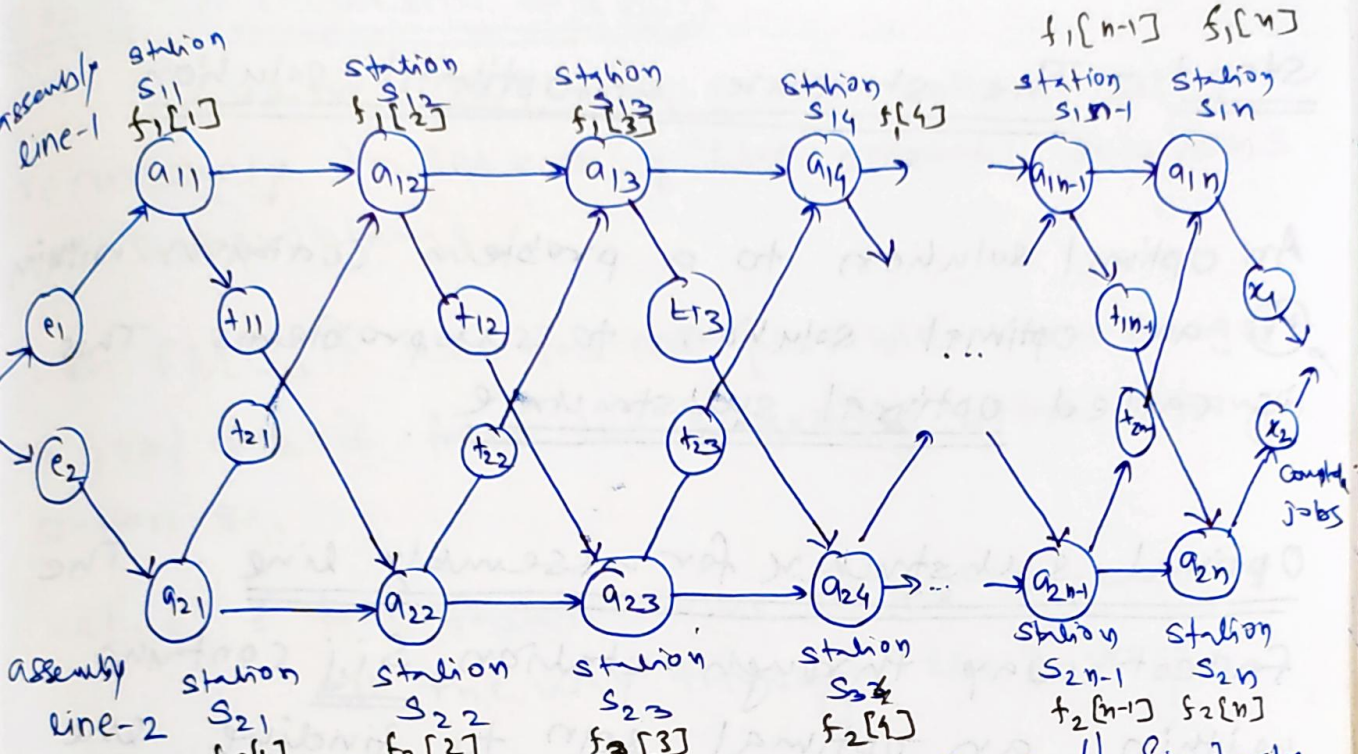
$j = 1, 2, \ldots, n$

$a_{i,j}$ = the assembly time required at station $S_{i,j}$.

The stations $S_{i,j}$ (for $i=1,2$) perform same function except it takes different amount of time to perform the same function.

At any time, the job can be transfered between any line in the next station.

Let $t_{i,j}$ is the time to transfer the job from the assembly line i to station $S_{i,j}$



Where e_1 and e_2 are times to schedule jobs on assembly line 1 & 2 respectively.

Similarly x_1 & x_2

Problem Defⁿ - Determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto/job.

"Brute force" methods is to list all stations and then check one by one

The no of way to list station = $2^n = \Omega(2^n)$

and to compute time is $\Theta(n)$

Step 1 - The structure of an optimal solution

An optimal solution to a problem contains within it an optimal solution to subproblems. This is called optimal substructure

Optimal substructure for assembly line - The fastest way through station $S_{i,j}$ contains within an optimal soln to finding the fastest way through either station $S_{1,j-1}$ or $S_{2,j-1}$

More specific

The fastest way through station $S_{1,j}$ is either

(i) the fastest way through $S_{1,j-1}$
(ii) the fastest way through $S_{2,j-1}$

The fastest way through station $S_{2,j}$ is either

(i) the fastest way through $S_{2,j-1}$
(ii) the fastest way through $S_{1,j-1}$

Step-2:- A recursive solution

Define the value of an optimal soln recursively in terms of the optimal solutions to subproblems.

Let $f_i[j]$ = the fastest possible time to get a job/chasis from the starting point through station $S_{i,j}$

Let f^* = the fastest time to get a chassis all the way through the factory.

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

Let $l_i[j]$ = line no. (1 or 2) whose station $j-1$ is used in a fastest way through station $S_{i,j}$

$j = 2$ to n
$i = 1$ or 2

Let l^* = the line no whose station n is used in a fastest way through entire factory.

Step 3 - Compute the value of an optimal soln

(i.e. compute the value of fastest time.)

We can solve the previous definition recursively in $\Omega(2^n)$ time.

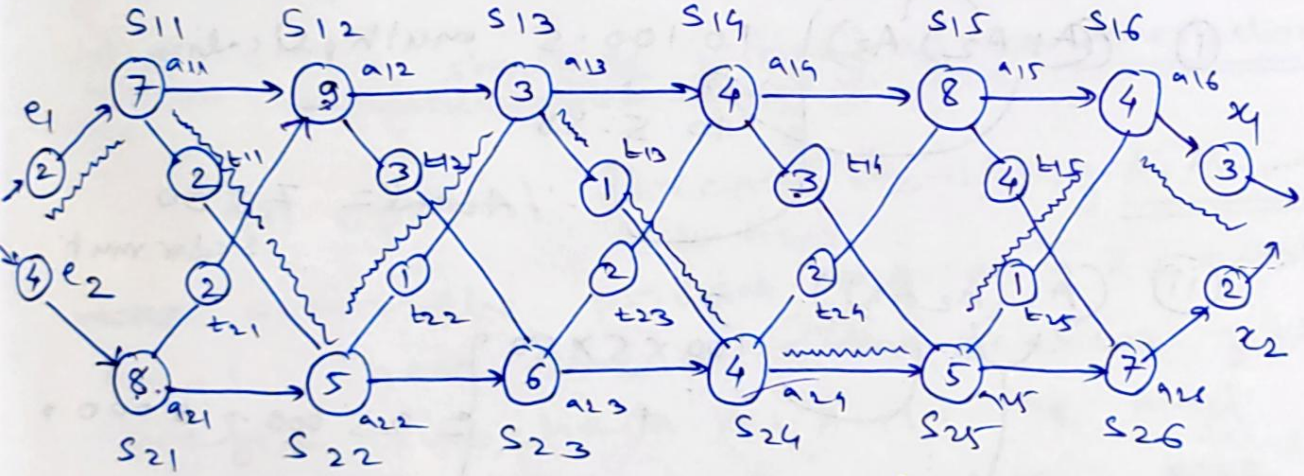
Iteratively we can solve above problem.

```
FASTest_Way (a, t, e, x, n)
1. f1[1] ← e1 + a1,1
2. f2[1] ← e2 + a2,1
3. for j ← 2 to n
4. ——→ do if f1[j-1] + a1,j ≤ f2[j-1] + t2,j-1 + a1,j
5. ——→ then f1[j] ← f1[j-1] + a1,j
6. ——————→ l1[j] ← 1
7. ——————→ else f1[j] ← f2[j-1] + t2,j-1 + a1,j
8. ——————→ l1[j] ← 2
9. ——→ if f2[j-1] + a2,j ≤ f1[j-1] + t1,j-1 + a2,j
10. ——→ then f2[j] ← f2[j-1] + a2,j
11. ——————→ l2[j] ← 2
12. ——→ else f2[j] ← f1[j-1] + t1,j-1 + a2,j
13. ——————→ l2[j] ← 1
14. if f1[n] + x1 ≤ f2[n] + x2
15.     then f* = f1[n] + x1
16.          l* = 1
17.     else f* = f2[n] + x2
18.          l* = 2
```

Step 4 - Construct the fastest time through all stations
(An optimal solⁿ)

```
print_stations (l, n)
1. i ← l*
2. print "line" i ", station" n
3. for j ← n down to 2
4.       do i ← li[j]
5.          print "line" i ", station" j-1
```

Example



j →	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j →	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 2$

detailed

$l^* = 1$

	1	2	3	4	5	6
$f_1[j]$	2+7=9	9+9=18 / 12+2+9=23	18+3=21 / 16+1+3=20	20+4=24 / 22+2+4=28	24+8=32 / 25+2+8=35	32+4=36 / 30+1+4=35
$f_2[j]$	4+8=12	9+2+5=16 / 16+5=21	18+3+6=27 / 16+6=22	20+1+4=25 / 22+4=26	24+3+5=32 / 25+5=30	32+4+7=43 / 30+7=37

$s_0[1] \Rightarrow S_{11}$
S_{22}
S_{13}
S_{24}
S_{25}
S_{16}

(2) Matrix-chain multiplication

```
Matrix-Multiplication (A, B)
for i ← 1 to rows[A]
    for j ← 1 to columns[B]
        C[i,j] ← 0
        for k ← 1 to columns[A]
            C[i,j] ← C[i,j] +
                A[i,k] [illegible]
```

Let $\langle A_1 A_2 A_3 \rangle$ be chain of matrices

A_1 : 10×100, A_2 : 100×5, A_3 : 5×50

(i) $((A_1 A_2) A_3)$ $10 \cdot 100 \cdot 5$ multiplications

$+$

$10 \cdot 5 \cdot 50$

$\therefore$ total = 7500 scalar multi

(ii) $(A_1 (A_2 A_3))$

$100 \times 5 \times 50$

$+$

$10 \times 100 \times 50$

$= 25000 + 50000$

$= 75000$

Matrix-chain multiplication problem :

Given $\langle A_1 A_2, \ldots A_n \rangle$ chain of n matrices where each A_i has $p_{i-1} \times p_i$ dimension

$\forall i = 1, 2, \ldots n$

i.e. $A_1 \rightarrow p_0 p_1$

$A_2 \rightarrow p_1 p_2$ etc.

no. of possible parenthesizations of n matrices = P(n)

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\, P(n-k) & \text{if } n \geq 2 \end{cases}$$

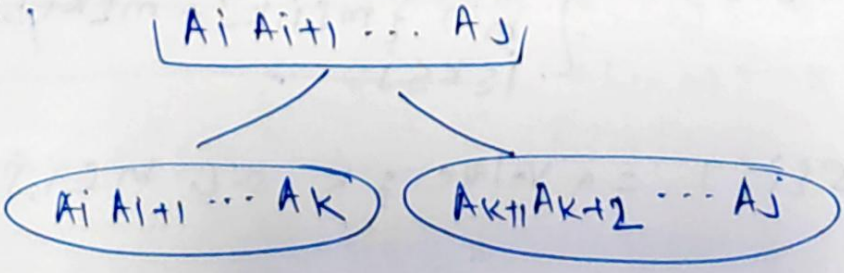
Soln is $\Omega(2^n)$ to the above recurrence soln.

∴ Brute-force method would take $\Omega(2^n)$ time to check all parenthesizations.

Step 1 - The structure of an optimal parenthesization

An optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the prefix subchain $A_i A_{i+1} \cdots A_k$ and this subchain $A_{k+1} \cdots A_j$ is within optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.

i.e.,

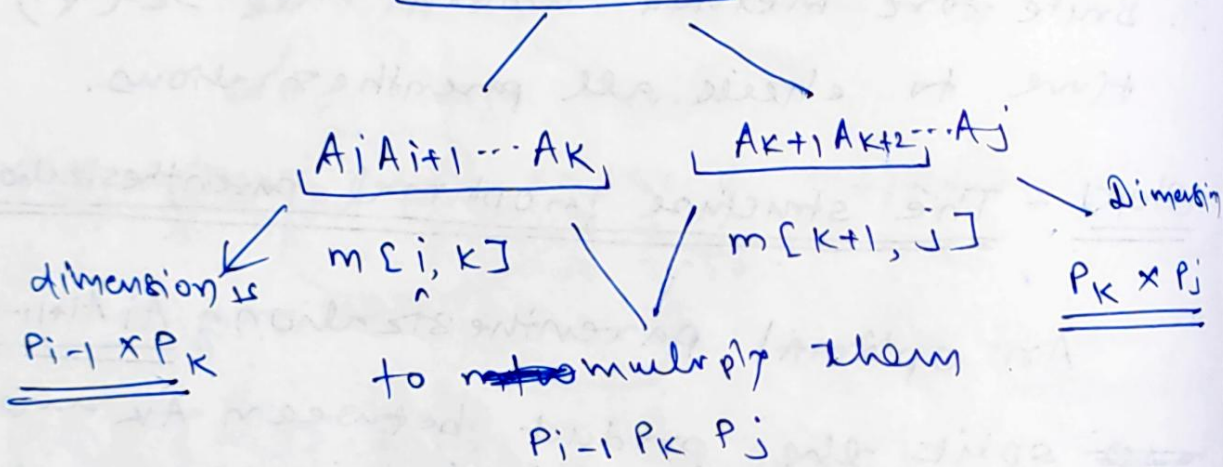


Step 2 - A recursive soln

Let m[i,j] be the minimum number of scalar multiplications needed to compute $A_{i..j}$

Now let k s.t.:

$$i \le k \le j$$



$$\therefore m[i,j] = m[i,k] + m[k+1,j] + P_{i-1} P_k P_j$$

Therefore the recursive definition is

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k \le j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \} & \text{if } i<j \end{cases}$$

$s[i,j]$ = a value of k s.t. $m[i,j]$ minimum. $i \le k \le j$

Step 3 - Computing the optimal costs

A recursive algorithm may encounter each subproblems many times in different branches of its recursion tree. So instead of computing recursively, we perform the third step of dynamic programming and compute the optimal soln using tabular method in bottom-up fashion.

```
Matrix-chain-order(p)
1.  n ← length[p] - 1
2.  for i ← 1 to n
3.      do m[i,i] ← 0
4.  for l ← 2 to n   (l is chain length)
5.      do for i ← 1 to n-l+1
6.          do j ← i+l-1   → {j = i+1, i+2, i+3, ..., i+n-1
7.              m[i,j] ← ∞
8.              for k ← i to j-1
9.                  do q ← m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
10.                 if q < m[i,j]
11.                 then m[i,j] ← q
12.                     s[i,j] ← k
13. return m and s.
```

Step-4 - Constructing an optimal solution

The final matrix multiplication in Comp

$A_{1..n}$ optimally is

$A_{1..S[1,n]}\ A_{S[1,n]+1..n}$

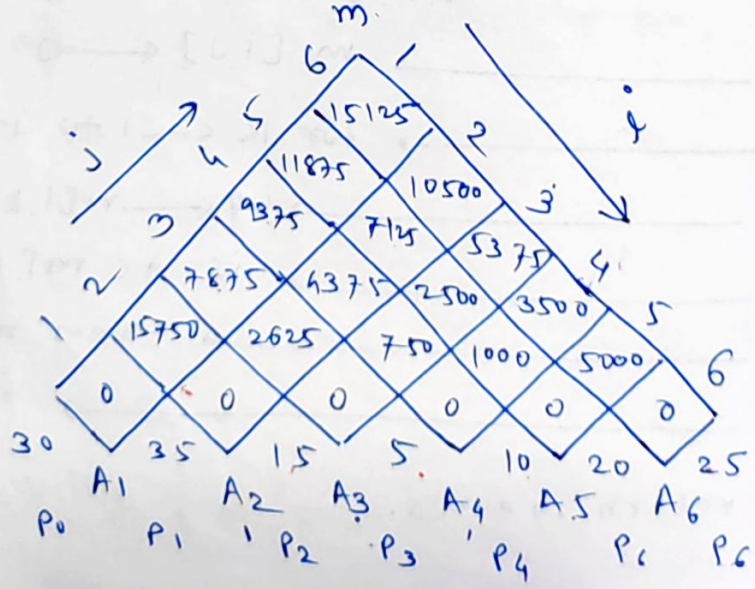
Print_Optimal_Parens (S, i, j)

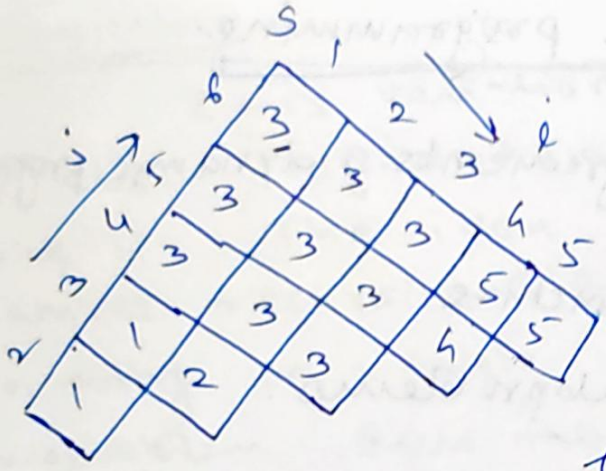
1. if i = j
2. → then print "A"i
3. → else print "("
4. ——→ Print_Optimal_Parens (S, i, S[i,j]
5. ——→ Print_optimal_Parens (S, S[i,j]+1, j
6. ——→ Print ")"

Example - $P_0\ P_1\ P_2\ P_3\ P_4\ P_5\ P_6$

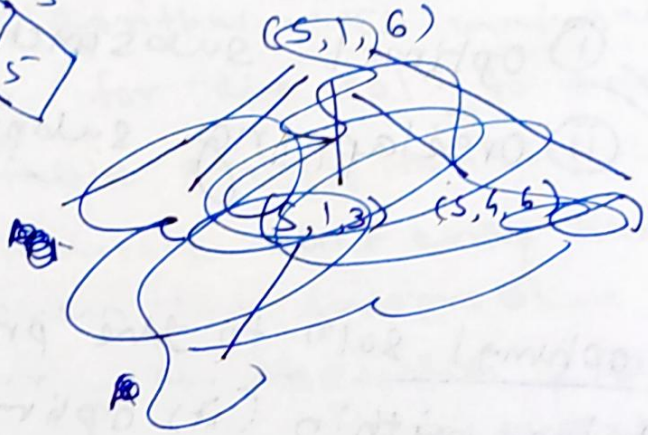
i.e. 30 35 15 5 10 20 25

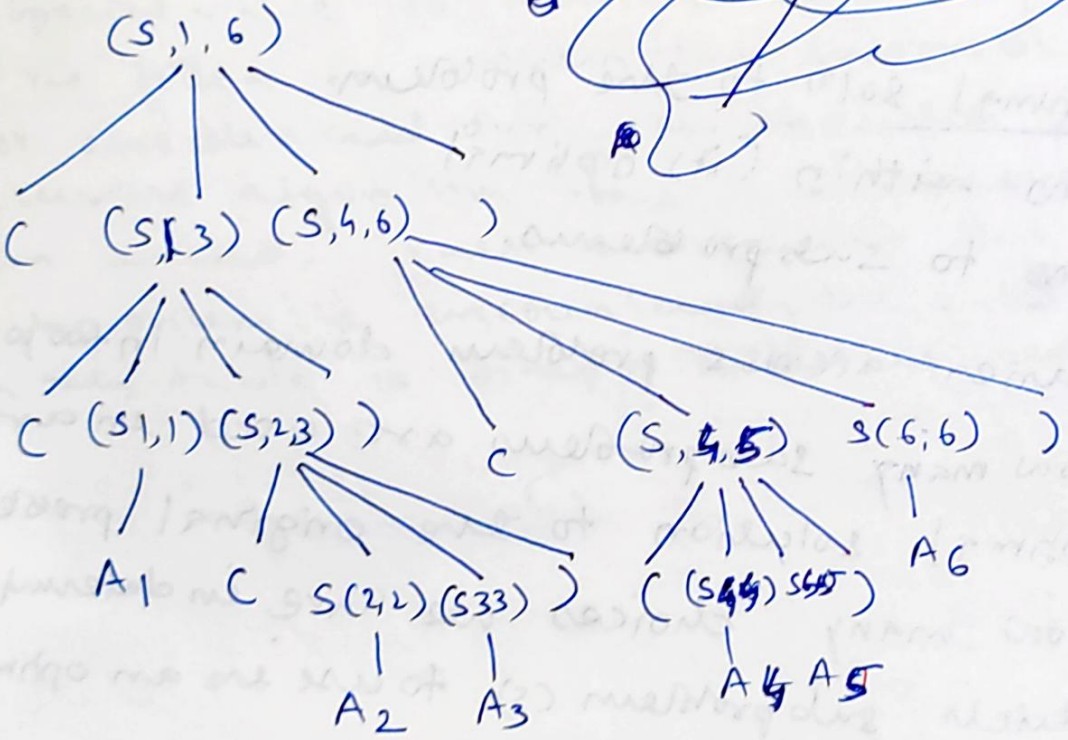
A_1 30×35
A_2 35×15
A_3 15×5
A_4 5×10
A_5 10×20
A_6 20×25





Call Print_Optimal_Parens(S, 1, 6)





$$\therefore \quad ((A_1(A_2A_3))((A_4A_5)A_6))$$

Elements of Dynamic programming

The two key ingredients of dynamic programming are

① optimal substructure
② Overlapping subproblems.

① an optimal soln to the problem contains within (it) optimal solutions to subproblems.

It varies across problem domain in two ways

(i) how many subproblems are used in an optimal solution to the original problem and (ii) how many choices we have in determining which subproblem (s) to use in an optimal solution.

② Overlapping subproblems

When total no. of distinct subproblems is a polynomial in the input size. A recursive algorithm revisits the same problem over and over again, we say the optimization problem has overlapping subproblems.

Memoization -

It is variation of dynamic programming in which top-down approach is used to solve it. The idea is to memoize the natural recursive algorithm. It maintains an entry in a table for the soln to each subproblem. Each table entry initially contains a special value to indicate that entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its soln is computed and then stored. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.

Longest common subsequence

Defⁿ - Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ and another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X s.t. $\forall$ all $j = 1, 2, \ldots, k$ we have $x_{i_j} = z_j$

e.g. $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A\ B\ C\ B\ D\ A\ B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$

(1) All characters in Z must be in order of its positions in X

Defⁿ - Given two sequences X and Y, we say that a sequence Z is a **common subsequence** of X and Y, if Z is **a subsequence of both X & Y**

e.g. if $X = \langle A, B\ C\ B\ D\ A\ B \rangle$

$Y = \langle B\ D\ C\ A\ B\ A \rangle$

Common subsequences are $\langle B\ C\ A \rangle$

$\langle B\ C\ B\ A \rangle$

& $\langle B\ D\ A\ B \rangle$

Defⁿ - Given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ & $Y = \langle y_1, y_2, \ldots y_n \rangle$ and wish to find the longest maximum length common subsequence of X & Y (LCS)

Step-1 - Chacterizing a LCS

Defn - **ith prefix of Sequence X** - It is ith prefix sequence of X denoted by X_i

e.g. if $X = \langle A\ B\ C\ B\ D\ A\ B \rangle$

$X_4 = \langle A\ B\ C\ B \rangle$

$X_0 = \phi$.

It is "i" number prefix symbol of X.

Optimal substructure of LCS

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

1. If $x_m = y_n$ and $z_k = x_m = y_n$ implies z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$ and $z_k \neq x_m$ implies Z is an LCS of X_{m-1} & Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$ implies Z is an LCS of X & Y_{n-1}

Step-2 - Recursive Soln

Let $c[i,j]$ = the length of an LCS of the sequences X_i and Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

Step 3 - Computing the length of an LCS

c[i,j] is an entry of table c[0..m, 0..n]

b[1..m, 1..n] = is used to simplify construction of an optimal soln.

```
LCS_Length(X, Y)
1.  m ← length[X]
2.  n ← length[Y]
3.  for i ← 1 to m
4.      do c[i,0] ← 0
5.  for j ← 1 to n
6.      do c[0,j] ← 0
7.  for i ← 1 to m
8.      do for j ← 1 to n
9.          do if xi = yj
10.             then c[i,j] ← c[i-1,j-1]+1
11.                 b[i,j] ← "↖"
12.         else if c[i-1,j] ≥ c[i,j-1]
13.             then c[i,j] ← c[i-1,j]
14.                 b[i,j] ← "↑"
15.         else c[i,j] ← c[i,j-1]
16.             b[i,j] ← "←"
17. return c and b.
```

Step-4

Constructing an LCS (Longest Common Subsequence)

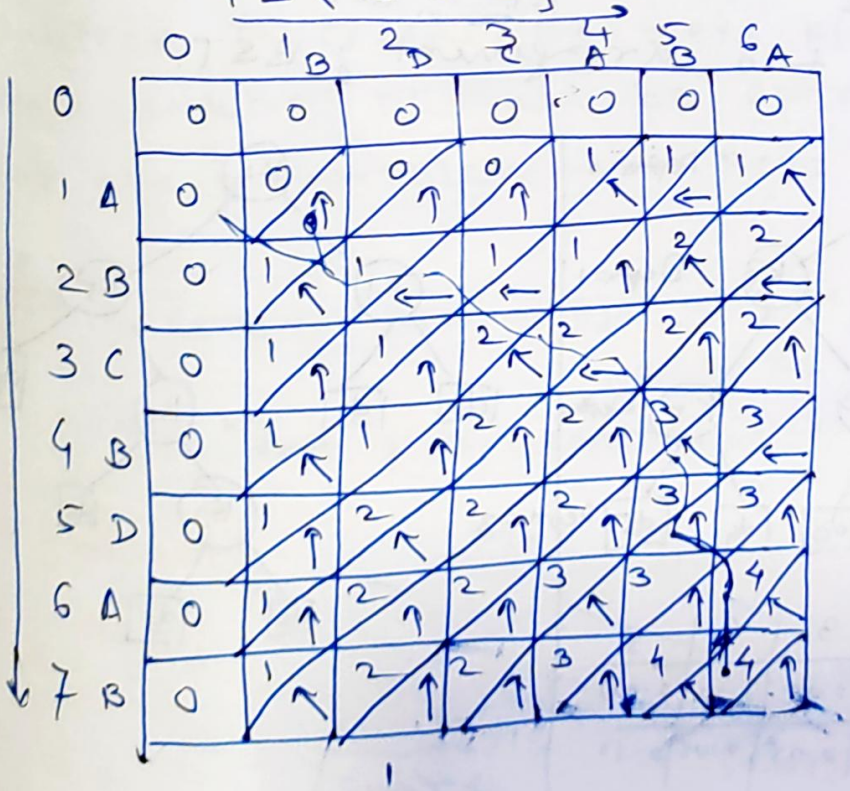
```
Print_LCS(b, X, i, j)
1. if i=0 or j=0
2.    then return
3. if b[i,j] = "↖"
4.    then Print_LCS(b, X, i-1, j-1)
5.         Print xi
6. else-if b[i j] = "↑"
7.         then Print LCS(b, X, i-1, j)
8. else Print_LCS(b, X, i, j-1).
```

Example

1 2 3 4 5 6 7

X = < A B C B D A B >

Y = < B D C A B A



Sol'n

1 + 2 + 3 + 4 + 6

B C B A