CSC373 Worksheet 3 Solution

August 2, 2020

1. Using the following formula

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j} M[i,k] + M[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j \end{cases} \tag{1}$$

we have

C	1	2	3	4	5	6
1	0	350	770	612	1212	1422
2	x	0	840	462	1662	1362
3	x	x	0	252	1092	1098
4	x	x	x	0	1440	936
5	x	x	x	x	0	720
6	x	x	x	x	x	0

And an optimal parenthesization is

My Work:

$(A_1A_2)(A_3A_4)(A_5A_6)$

Correct Solution:

Using the following formula

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j} M[i,k] + M[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j \end{cases} \tag{2}$$

we have

m (i/j)	1	2	3	4	5	6
1	0	350	770	612	1212	1422
2	x	0	840	462	1662	1362
3	x	x	0	252	1092	1098
4	x	x	x	0	1440	936
5	x	x	x	x	0	720
6	x	x	x	x	x	0

s (i/j)	1	2	3	4	5	6
1	0	1	2	1	4	4
2	x	0	2	2	4	4
3	x	x	0	3	4	4
4	x	x	x	0	4	4
5	x	x	x	x	0	5
5	x	x	x	x	x	0

And an optimal parenthesization is

My Work:

$(A_1(A_2(A_3A_4)))(A_5A_6)$

Notes:

- Sequence of Dimensions

  The sequence of dimensions $< p_0 = 5, p_1 = 10, p_2 = 3, p_2 = 12, p_3 = 5, p_4 = 50, p_5 = 6 >$ means there are 6 matrices with dimensions $p_{i-1} \times p_i$

  - $A_1 \rightarrow 5 \times 10$
  - $A_2 \rightarrow 10 \times 3$
  - $A_3 \rightarrow 3 \times 12$
  - $A_4 \rightarrow 12 \times 5$
  - $A_5 \rightarrow 5 \times 50$
  - $A_6 \rightarrow 50 \times 6$
- Dynamic Programming
  - Is applied to optimization problems
  - Applies when the subproblems overlap
  - Uses the following sequence of steps
    1. Characterize the structure of an optimal solution

2. Recursively define the value of an optimal solution
3. Construct an optimal solution from computed information

- Matrix-chain Multiplication
  - Is an optimization problem solved using dynamic programming
  - Goal is to find matrix parenthesis with fewest number of operations

  Example:

  Given chain of matrices $< A, B, C >$, it's fully parenthesized product is:

  * $(AB)C$ needs $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$ operations
  * $A(BC)$ needs $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 27000$ operations

  Thus, $(AB)C$ performs more efficiently than $A(BC)$.

  - Is stated as: given a chain $< A_1, A_2, ..., A_n >$ of n matrices, where for $i = 1, 2, ..., n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 ... A_n$ in a way that minimizes the number of scalar multiplications.
  - Steps

    1. **Check is the problem has Optimal Substructure**

       Let us adopt the notation $A_{i...j}$ where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} ... A_j$.

       Assume the solution has the following parentheses:

       $$(A_{i...k})(A_{k+1...j})$$

       If there is a better way to multiply $(A_{i...k})$, then we would have a more optimal solution.

       This would be a contradiction, as we already stated that we have the optimal solution for $A_{i...j}$.

       Therefore, this problem has optimal substructure.

    2. **Find the Recursive Solution**

       Let $M[i, j]$ be the cost of multiplying matrices from A_i to A_j

       We want to find out at which $'k'$ returns the fewest number of multiplications, or the minimum number of M.

The recursive formula for the cost of multiplying from A_i to A_j is

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j} M[i,k] + M[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j \end{cases} \tag{3}$$

3. **Computing the Estimated Cost**

   * Steps

   1) Fill the table for $i = j$
   2) Fill the table for $i < j$ with a spread of 1
   3) Repeat 2 with the increased value of spread

<u>**Example:**</u>

Given
$< A_1, A_2, A_3, A_4, A_5 >$

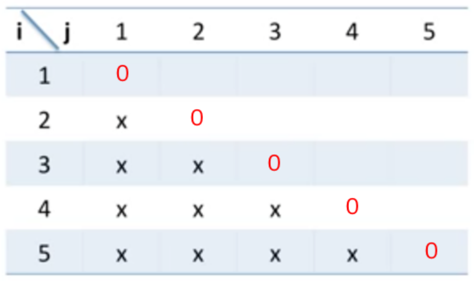
where

* $A_1 \rightarrow 4 \times 10$
* $A_2 \rightarrow 10 \times 3$
* $A_3 \rightarrow 3 \times 12$
* $A_4 \rightarrow 12 \times 20$
* $A_5 \rightarrow 20 \times 7$

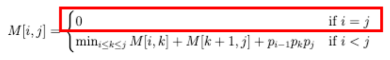
we have:

1) Fill the table for $i = j$

1) i = j



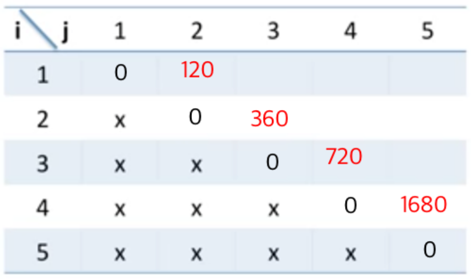
i \ j	1	2	3	4	5
1	0				
2	x	0			
3	x	x	0		
4	x	x	x	0	
5	x	x	x	x	0



$$M[i,j] = \begin{cases} \boxed{0 \qquad \text{if } i = j} \\ \min_{i \le k \le j} M[i,k] + M[k+1,j] + p_{i-1}p_kp_j & \text{if } i < j \end{cases}$$

2) Fill the table for $i < j$ with a spread of 1

2) (i = 1, j = 2), (i = 2, j = 3), (i = 3, j = 4), (i = 4, j = 5)



i \ j	1	2	3	4	5
1	0	120			
2	x	0	360		
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

since

* $i = 1, j = 2$

$$\begin{aligned} M[1,2] &= \min_{1\le k\le 2}(M[1,1] + M[1,2] + p_{i-1}p_kp_j) \quad (4)\\ &= \min_{1\le k\le 2}(0 + 0 + p_0p_1p_2) \quad (5)\\ &= \min_{1\le k\le 2}(0 + 0 + 4\cdot 10\cdot 3) \quad (6)\\ &= 120 \quad (7)\end{aligned}$$

where $p_0 = 3$ is from the dimension 3×10 of A_1, $p_k = 10$ is from the dimension of 3×10 of A_1.

* $i = 2, j = 3$

$$\begin{aligned} M[2,3] &= \min_{2\le k\le 3}(M[2,2] + M[3,3] + p_{i-1}p_kp_j) \quad (8)\\ &= \min_{2\le k\le 3}(0 + 0 + p_1p_2p_3) \quad (9)\\ &= \min_{2\le k\le 3}(0 + 0 + 10\cdot 3\cdot 12) \quad (10)\\ &= 360 \quad (11)\end{aligned}$$

* $i = 3, j = 4$

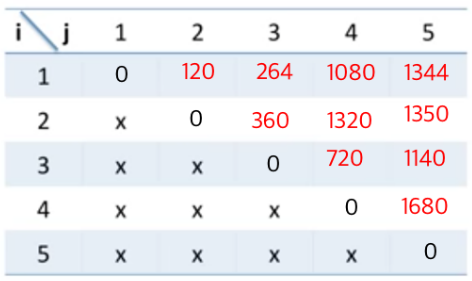
$$\begin{aligned} M[3,4] &= \min_{3\le k\le 4}(M[3,3] + M[4,4] + p_{i-1}p_kp_j) \quad (12)\\ &= \min_{3\le k\le 4}(0 + 0 + p_2p_3p_4) \quad (13)\\ &= \min_{3\le k\le 4}(0 + 0 + 3\cdot 12\cdot 20) \quad (14)\\ &= 720 \quad (15)\end{aligned}$$

* $i = 4, j = 5$

$$\begin{aligned}
M[4,5] &= \min_{4 \le k \le 5} (M[4,4] + M[5,5] + p_{i-1}p_k p_j) && (16)\\
&= \min_{4 \le k \le 5} (0 + 0 + p_3 p_4 p_5) && (17)\\
&= \min_{4 \le k \le 5} (0 + 0 + 12 \cdot 20 \cdot 7) && (18)\\
&= 1680 && (19)
\end{aligned}$$

3) Repeat 2 with the increased value of spread

2) (i = 1, j = 2), (i = 2, j = 3), (i = 3, j = 4), (i = 4, j = 5)



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

* $i = 1, j = 3$

$\underline{k = 1}$

$$\begin{aligned}
M[1,3] &= M[1,1] + M[2,3] + p_{i-1}p_k p_j && (20)\\
&= 0 + 360 + p_0 p_1 p_3 && (21)\\
&= 0 + 360 + 4 \cdot 10 \cdot 12 && (22)\\
&= 0 + 360 + 480 && (23)\\
&= 840 && (24)
\end{aligned}$$

$\underline{k = 2}$

$$\begin{aligned}
M[1,3] &= M[1,2] + M[3,3] + p_{i-1}p_k p_j && (25)\\
&= 120 + 0 + p_0 p_2 p_3 && (26)\\
&= 120 + 0 + 4 \cdot 10 \cdot 12 && (27)\\
&= 120 + 0 + 144 && (28)\\
&= 264 && (29)
\end{aligned}$$

Thus, $\min_{1 \le k \le 3} M[1,3] = 264$.

* $i = 2, j = 4$

  <u>$k = 2$</u>

$$\begin{aligned} M[2,4] &= M[2,2] + M[3,4] + p_{i-1}p_kp_j && (30)\\ &= 0 + 720 + p_1p_2p_4 && (31)\\ &= 0 + 720 + 10 \cdot 3 \cdot 20 && (32)\\ &= 0 + 720 + 600 && (33)\\ &= 1320 && (34)\end{aligned}$$

  <u>$k = 3$</u>

$$\begin{aligned} M[2,4] &= M[2,2] + M[3,4] + p_{i-1}p_kp_j && (35)\\ &= 360 + 0 + p_1p_3p_4 && (36)\\ &= 360 + 0 + 10 \cdot 12 \cdot 20 && (37)\\ &= 360 + 0 + 2400 && (38)\\ &= 2760 && (39)\end{aligned}$$

  Thus, $\min_{2 \leq k \leq 4} M[2,4] = 1320$.

* $i = 3, j = 5$

  <u>$k = 3$</u>

$$\begin{aligned} M[3,5] &= M[3,3] + M[3,5] + p_{i-1}p_kp_j && (40)\\ &= 0 + 1680 + p_2p_3p_5 && (41)\\ &= 0 + 1680 + 3 \cdot 12 \cdot 7 && (42)\\ &= 0 + 1680 + 252 && (43)\\ &= 1932 && (44)\end{aligned}$$

  <u>$k = 4$</u>

$$\begin{aligned} M[3,5] &= M[3,4] + M[5,5] + p_{i-1}p_kp_j && (45)\\ &= 720 + 0 + p_2p_4p_5 && (46)\\ &= 720 + 0 + 3 \cdot 20 \cdot 7 && (47)\\ &= 720 + 420 && (48)\\ &= 1140 && (49)\end{aligned}$$

  Thus, $\min_{3 \leq k \leq 5} M[3,5] = 1140$.

* $i = 2, j = 5$

  <u>$k = 2$</u>

$$\begin{aligned} M[2,5] &= M[2,2] + M[3,5] + p_{i-1}p_kp_j && (50)\\ &= 0 + 1140 + p_1p_2p_5 && (51)\\ &= 0 + 1140 + 10 \cdot 3 \cdot 7 && (52)\\ &= 0 + 1140 + 210 && (53)\\ &= 1350 && (54) \end{aligned}$$

  <u>$k = 3$</u>

$$\begin{aligned} M[2,5] &= M[2,3] + M[4,5] + p_{i-1}p_kp_j && (55)\\ &= 360 + 1680 + p_1p_3p_5 && (56)\\ &= 2040 + 10 \cdot 12 \cdot 7 && (57)\\ &= 2040 + 840 && (58)\\ &= 2880 && (59) \end{aligned}$$

  <u>$k = 4$</u>

$$\begin{aligned} M[2,5] &= M[2,4] + M[5,5] + p_{i-1}p_kp_j && (60)\\ &= 1320 + p_1p_3p_5 && (61)\\ &= 1320 + 10 \cdot 20 \cdot 7 && (62)\\ &= 1320 + 1400 && (63)\\ &= 2720 && (64) \end{aligned}$$

  Thus, $\min_{2 \le k \le 5} M[2,5] = 1350$.

* $i = 1, j = 5$

  <u>$k = 1$</u>

$$\begin{aligned} M[1,5] &= M[1,1] + M[3,5] + p_{i-1}p_kp_j && (65)\\ &= 0 + 1350 + p_0p_1p_5 && (66)\\ &= 0 + 1350 + 4 \cdot 10 \cdot 7 && (67)\\ &= 0 + 1350 + 280 && (68)\\ &= 1630 && (69) \end{aligned}$$

<u>$k = 2$</u>

$$
\begin{aligned}
M[1,5] &= M[1,2] + M[3,5] + p_{i-1}p_kp_j \qquad (70)\\
&= 120 + 1140 + p_0p_2p_5 \qquad (71)\\
&= 120 + 1140 + 4 \cdot 3 \cdot 7 \qquad (72)\\
&= 1260 + 84 \qquad (73)\\
&= 1344 \qquad (74)
\end{aligned}
$$

<u>$k = 3$</u>

$$
\begin{aligned}
M[1,5] &= M[1,3] + M[4,5] + p_{i-1}p_kp_j \qquad (75)\\
&= 264 + 1680 + p_0p_3p_5 \qquad (76)\\
&= 264 + 1680 + 4 \cdot 12 \cdot 7 \qquad (77)\\
&= 1944 + 336 \qquad (78)\\
&= 2280 \qquad (79)
\end{aligned}
$$

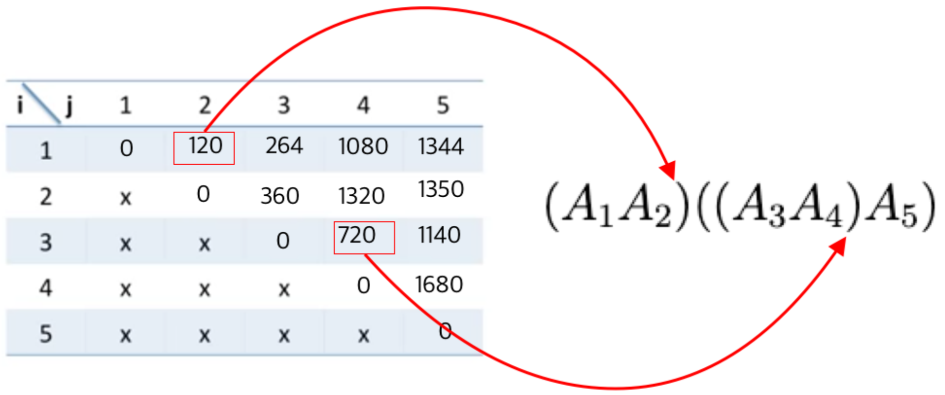
<u>$k = 4$</u>

$$
\begin{aligned}
M[1,5] &= M[1,4] + M[5,5] + p_{i-1}p_kp_j \qquad (80)\\
&= 1080 + 0 + p_0p_4p_5 \qquad (81)\\
&= 1080 + 4 \cdot 20 \cdot 7 \qquad (82)\\
&= 1080 + 560 \qquad (83)\\
&= 1640 \qquad (84)
\end{aligned}
$$

Thus, $\min_{1 \le k \le 5} M[1,5] = 1344$.

4. **Constructing the Optimal Solution (Needs revision)**

3)



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

$$(A_1A_2)((A_3A_4)A_5)$$

So, the optimal solution is $(A_1A_2)((A_3A_4)A_5)$

References:

1) CSBreakdown, Chain Multiplication - Dynamic Programming, link
2) University of Maryland, CMSC351 - Fall 2014 Homework # 4, link

2.
```
procedure MATRIX-CHAIN-MULTIPLY(A,s,i,j)
    if i == j then
        return A[i]
    end if

    if i < j
        a = MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j])
        b = MATRIX-CHAIN-MULTIPLY(A,s,s[i,j] + 1,j)

        return MATRIX-CHAIN-MULTIPLY(a,b)

    end if

end procedure

```

Example:

- *MATRIX-CHAIN-ORDER* computes the table s containing optimal costs
- s table consists of k value at which $m[i, j]$ is minimum!!

  $A_{1..s[1,n]}A_{s[1,n]+1..n}$

- Table of optimal costs m is used with table s to construct solution to matrix-chain multiplication problem

3. First, we need to determine the total number of times $m[i, j]$ is referred in the innermost loop.

   We know from the header that the loop runs from $k = i$ to $k = j - 1$.

   Using this fact, we can write the innermost loop has $j - i = l - 1$ iterations.

   Since $m[i.j]$ is referred twice, the total number of $m[i, j]$ referred in the loop is:

$$(l - 1)2 \tag{1}$$

Second, we need to determine the total number of times $m[i, j]$ is referred in the intermediate loop

We know from the header that the loop runs from $i = 1$ to $i = n - l + 1$.

Using this fact, we can write the intermediate loop runs $n - l + 1$ iterations.

Since each iteration referrs $m[i, j]$ $(l - 1)2$ many times, the total number of times $m[i, j]$ is referred in the itnermediate loop is:

$$(n - l + 1)(l - 1)2 \tag{2}$$

Finally, we need to determine the total number of times $m[i, j]$ is referenced in the outermost loop.

We know from the header that the loop runs from $l = 2$ to n.

Since each iteration referrs $m[i, j]$ $(n - l + 1)(l - 1)2$ many times, the total number of times $m[i, j]$ is referred in the outermost loop is:

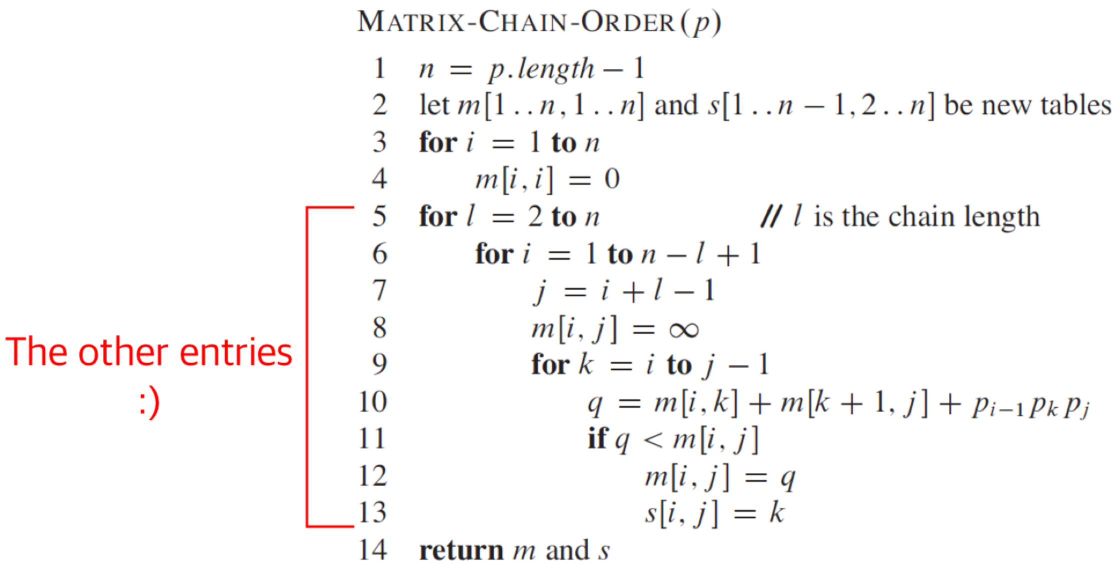
$$
\begin{align}
\sum_{l=2}^{n}(n-l+1)(l-1)2 &= 2\sum_{l'=1}^{n-1}(n-l')(l') \tag{3}\\
&= 2\Big[\sum_{l'=1}^{n-1} nl' - (l')^2\Big] \tag{4}\\
&= 2\Big[n\sum_{l'=1}^{n-1} l' - \sum_{l'=1}^{n-1}(l')^2\Big] \tag{5}\\
&= 2\Big[n\sum_{l'=1}^{n-1} l' - \sum_{l'=1}^{n-1}(l')^2\Big] \tag{6}\\
&= 2\Big[n\sum_{l'=0}^{n-1} l' - \sum_{l'=0}^{n-1}(l')^2\Big] \tag{7}\\
&= 2\Big[\frac{(n-1)n^2}{2} - \frac{(n-1)n(2n-1)}{6}\Big] \tag{8}\\
&= 2\Big[\frac{n^3-n^2}{2} - \frac{(n-1)n(2n-1)}{6}\Big] \tag{9}\\
&= 2\Big[\frac{n^3-n^2}{2} - \frac{(n^2-n)(2n-1)}{6}\Big] \tag{10}\\
&= 2\Big[\frac{n^3-n^2}{2} - \frac{(3n^3-3n^2+n)}{6}\Big] \tag{11}\\
&= 2\Big[\frac{3n^3-3n^2}{6} - \frac{(3n^3-3n^2+n)}{6}\Big] \tag{12}\\
&= 2\Big[\frac{n^3-n}{6}\Big] \tag{13}\\
&= \frac{n^3-n}{3} \tag{14}
\end{align}
$$

Notes:

- Hint from equation $A.3$

  $\sum\limits_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- I feel the need to gain more insight regarding the question's 'other table entries in a call of MATRIX-CHAIN-ORDER'. What does 'other table entries' mean?

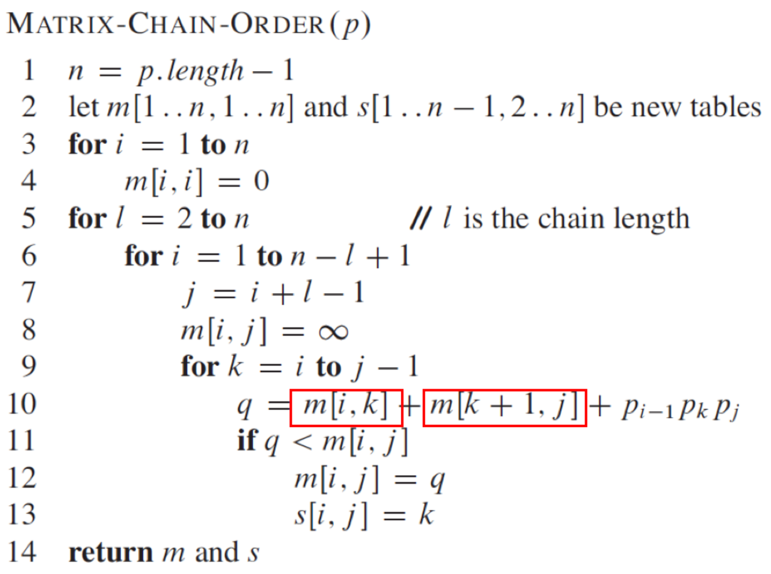
  Answer:



The other entries :)

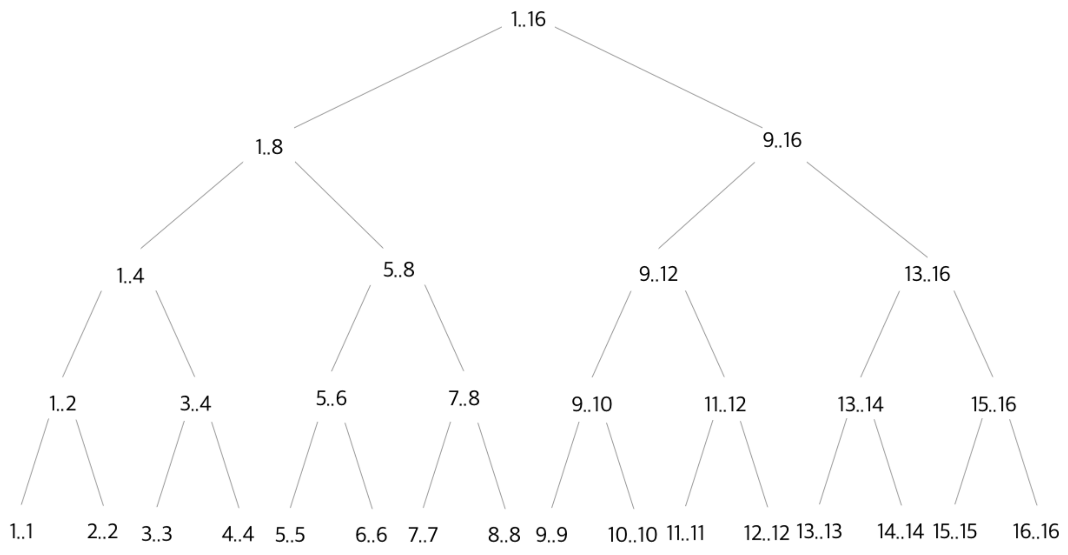
```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n            // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + 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
```

- In the problem '$m[i,j]$ is referenced' refers to $m[i,j]$ used in assignments $q = m[i,j]$...



```
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 = $\boxed{m[i, k]}$ + $\boxed{m[k + 1, j]}$ + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

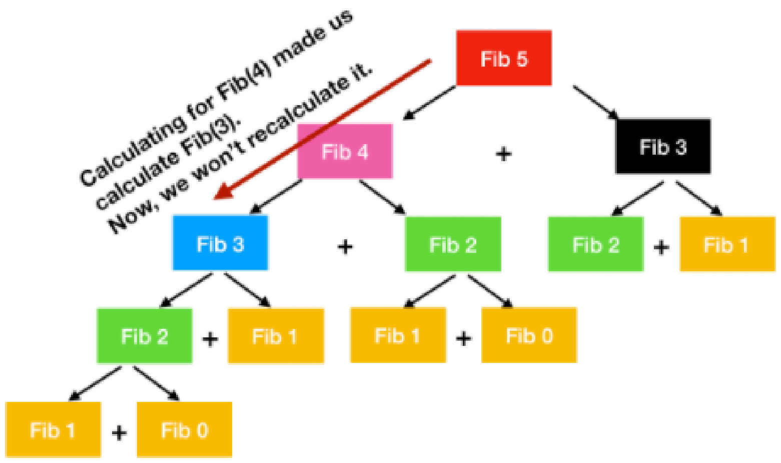
4. **Solution**



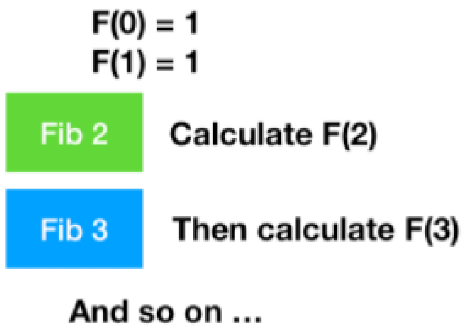
Memoization fails to speed up the algorithm because it lacks overlapping subproblems.

Notes:

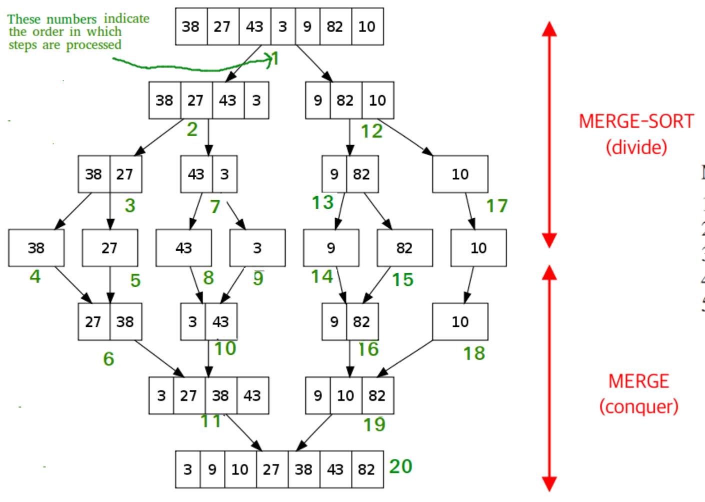
- Elements of Dynamic Programming
  - Optimal Substructure
    - Is the first step to solving dynamic programming problem
    - Exists if an optimal solution to the problem contains within it optimal solution to subproblems
  - Overlapping Subproblems
    - Is the second step to solving dynamic programming
    - Exists when an algorithm revisits the same problem repeatedly
  - Memoization
    - Maintains an entry in a table for the solution to each subproblem
    - Ensures that a method doesn't run for the same inputs more than once
- Top-Down Dynamic Programming
  - Uses recursion
  - Is preferred
    - When all sub-solutions need to be solved
    - Because it's easier


Calculating for Fib(4) made us calculate Fib(3). Now, we won't recalculate it.
Fib 5
Fib 4
+
Fib 3
Fib 3
+
Fib 2
Fib 2
+
Fib 1
Fib 2
+
Fib 1
Fib 1
+
Fib 0
Fib 1
+
Fib 0

- Bottom-Up Dynamic Programming
  - Uses for-loop
  - Is preferred
    - When all sub-solutions need to be solved
    - Because it is sometimes faster (No recursive call and no unnecessary Random Memory access)
  - Is preferred when not all sub-solutions need to be computed



- Merge Sort
  - How it works
    1. Find the middle point to divide the array into two halves
    2. Call mergeSort for first half
    3. Call mergeSort for second half
    4. Merge two halves in sorted order


These numbers indicate the order in which steps are processed
MERGE-SORT (divide)
MERGE (conquer)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

References

1)

5. Yes. This problem does exhibit optimal substructure

Proof. Assume that the optimal structure of $A_iA_{i+1}..A_j$ exists. That is, there exists some $k \in \mathbb{N} - \{0\}$ such that the following parenthesis $(A_{i..k})(A_{k+1..j})$ produces maximum operation cost.

I need to show that the substructures $A_{i..k}$ and $A_{k+1..j}$ are also optimal.

I will do so in parts.

Part 1 (Proving optimal substructure of $A_{i..k}$)

Assume for the sake of contradiction that the substructure $A_{i..k}$ is not optimal.

Then, we can write that there exists some $k' \in \mathbb{N} - \{0\}$ such that $(A_{i...k'})(A_{k'+1..k})$ has larger operation cost than $(A_{i..k})$.

Then, we can write that $((A_{i...k'})(A_{k'+1..k}))(A_{k+1..j})$ has larger operation costs than $(A_{i..k})(A_{k+1..j})$, which contradicts the original assumption that $(A_{i..k})(A_{k+1..j})$ has maximum operation cost.

Thus, $(A_{i..k})$ must be optimal.

Part 2 (Proving optimal substructure of $A_{k+1..j}$)

This proof is nearly verbatim as part 1, where the only difference is using $A_{k+1..j}$ instead of $A_{i..j}$. $\square$

Notes:

- A problem has **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems. [3]
- Showing optimal subtructure for the original Matrix-Chain Multiplication problem

Assume that the optimal structure of $A_iA_{i+1}..A_j$ exists. That is, there exists some $k \in \mathbb{N} - \{0\}$ such that the following parenthesis $(A_{i..k})(A_{k+1..j})$ produces minimum operation cost.

I need to show that the substructures $A_{i..k}$ and $A_{k+1..j}$ are also optimal.

I will do so in parts.

Part 1 (Proving optimal substructure of $A_{i..k}$)

Assume for the sake of contradiction that the substructure $A_{i..k}$ is not optimal.

Then, we can write that there exists some $k' \in \mathbb{N} - \{0\}$ such that $(A_{i...k'})(A_{k'+1..k})$ has smaller operation cost than $(A_{i..k})$.

Then, we can write that $((A_{i...k'})(A_{k'+1..k}))(A_{k+1..j})$ has smaller operation costs than $(A_{i..k})(A_{k+1..j})$, which contradicts the original assumption that $(A_{i..k})(A_{k+1..j})$ has minimum operation cost.

Thus, $(A_{i..k})$ must be optimal.

Part 2 (Proving optimal substructure of $A_{k+1..j}$)

This proof is nearly verbatim as part 1, where the only difference is using $A_{k+1..j}$ instead of $A_{i..j}$.

References:

1) CSBreakdown, Chain Multiplication - Dynamic Programming, link
2) CodeScope, Dynamic Programming, link
3) Wikipedia, Optimal Substructure, link

6. Let $< 2, 10, 20, 5 >$.

Then, we see that $p_0p_1p_3$ is minimum, and by Professor Capulet's claim, splitting at $k = 1$ should result in minimum-cost matrix multiplication.

But we have

m	1	2	3
1	0	400	600
2	x	0	1000
3	x	x	0

s	1	2	3
1	0	1	2
2	x	0	2
3	x	x	0

And the minimum-cost matrix multiplication occurs when $A_{i..j}$ is splitted at $k = 2$.

Thus, Professor Capulet's claim is false.

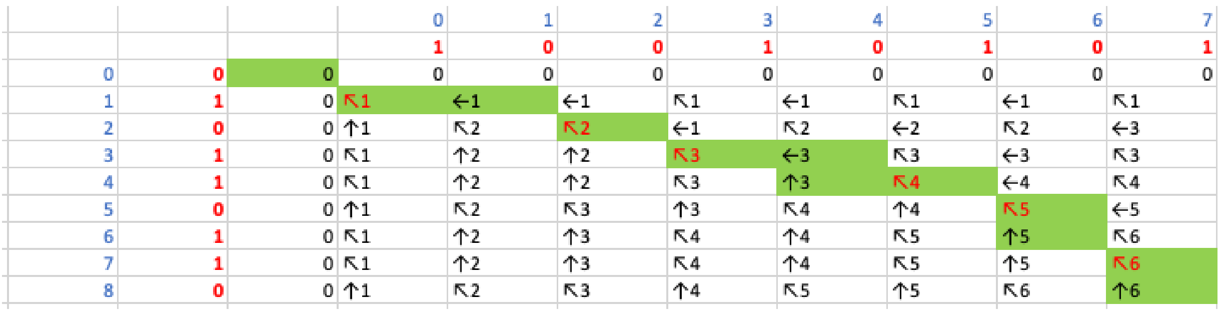
Notes:

- I need to find the matrices $A_i, ..., A_j$ where the total operating cost of Professor Capulet's method is bigger than the properly parenthesized solution
- I feel the need for clarafication regarding the phrase 'always choosing the matrix A_k at which to split the product...' Is k in A_k the same in any matrix multiplications $A_iA_{i+1}...A_j$?

  Answer:

  No. k in A_k is the value that makes $p_{i-1}p_kp_j$ minimum.

7. **Solution:**



			0	1	2	3	4	5	6	7
			1	0	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0
1	1	0	↖1	←1	←1	↖1	←1	↖1	←1	↖1
2	0	0	↑1	↖2	↖2	←1	↖2	←2	↖2	←3
3	1	0	↖1	↑2	↑2	↖3	←3	↖3	←3	↖3
4	1	0	↖1	↑2	↑2	↖3	↑3	↖4	←4	↖4
5	0	0	↑1	↖2	↖3	↑3	↖4	↑4	↖5	←5
6	1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
7	1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
8	0	0	↑1	↖2	↖3	↑4	↖5	↑5	↖6	↑6

So, the LCS of $< 1, 0, 0, 1, 0, 1, 0, 1 >$ and $< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$ is '101101'.

Notes:

- Longest Common Sequence
  - Goal is to find the longest common subsequence in $X =< x_1, x_2, ..., x_n >$ and $Y =< y_1, y_2, ..., y_n >$
  - e.g. Given two substrings $s_1 =< M, U, N, G >$ and $s_2 =< U, N, I, V, E, R, S, I, T, Y >$, the longest substring is '$< U, N >$'.
  - Steps
    1. Verify Optimal Substructure
    2. Create Recursive Solution

$$M[i, j] = \begin{cases} 0 & i = 0, \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j \\ Max(C_{i,j-1}, C_{i-1,j}) & \text{if } i, j > 0 \ a_i \neq b_j \end{cases} \tag{15}$$

    3. Computing the length of an LCS

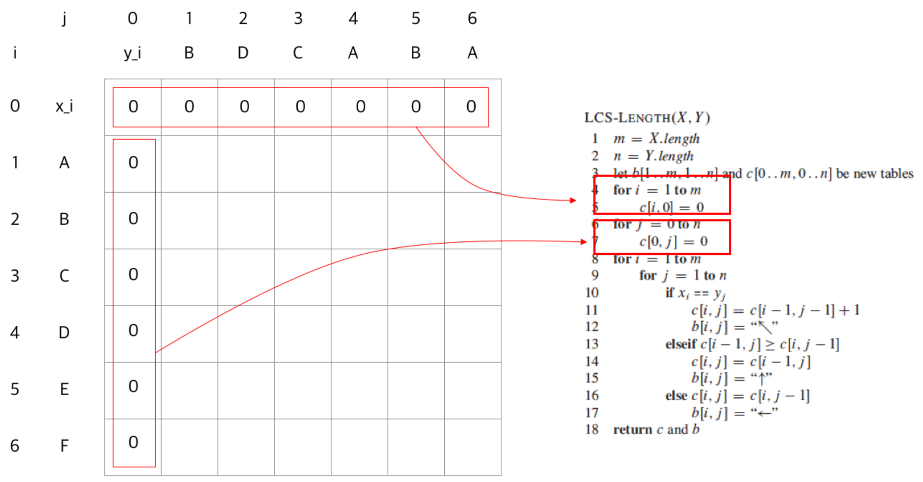
      * Top-Down Solution
      * Bottom-Top Solution

```
    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[i]
                    c[i,j] = c[i-1,j-1] + 1
                    b[i,j] = '↖'

                elseif c[i-1, j] ≥ c[i,j-1]
                    c[i,j] = c[i,j-1]
                    b[i,j] = '↑'

                else
                    c[i,j] = c[i,j-1]
                    b[i,j] = '←'

        return c and b

```

4. Constructing an LCS

Steps:

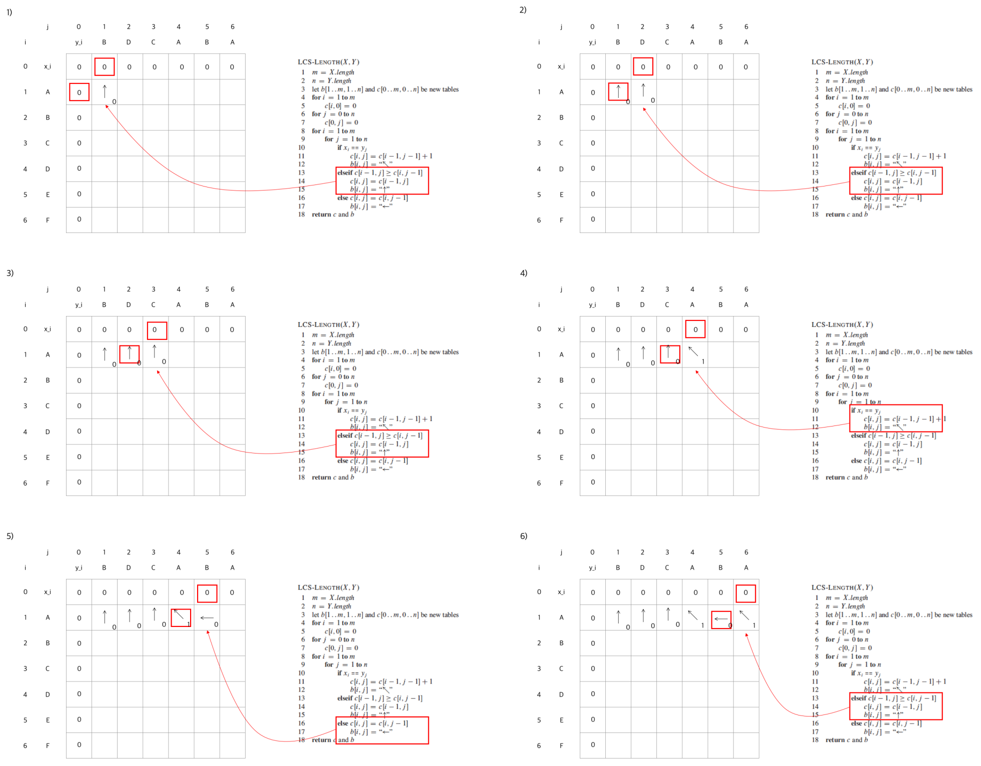
1) Fill $i = 0$ or $j = 0$ with 0



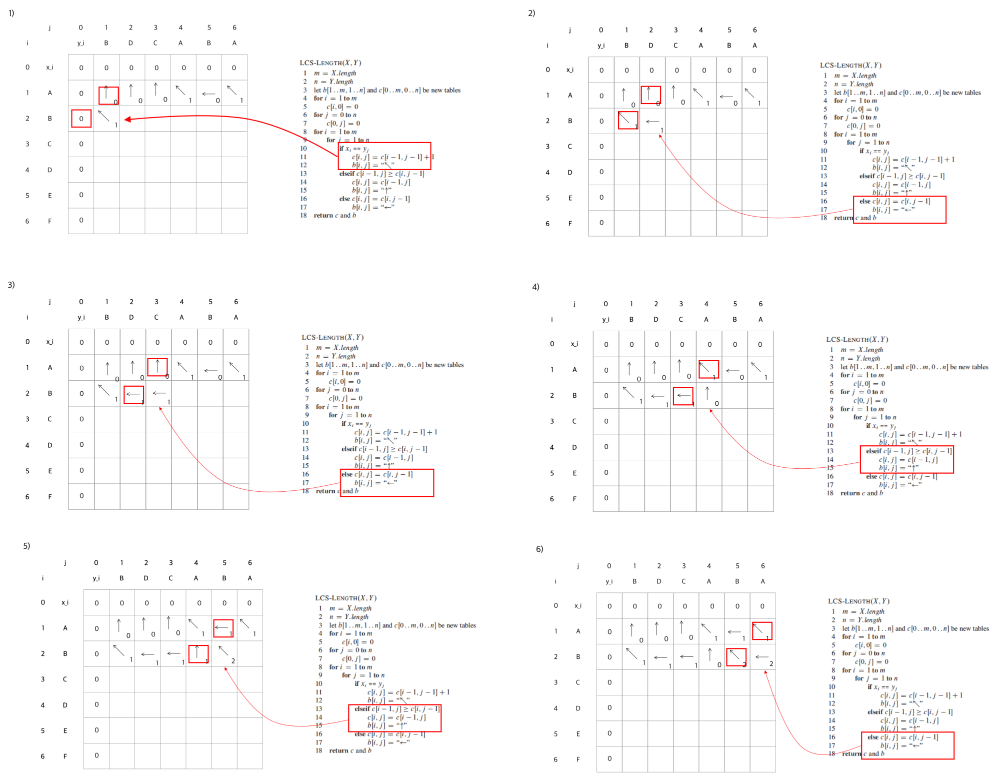
	j	0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	D	0						
5	E	0						
6	F	0						

```
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] = "↖"
        elseif 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
```

2) Compute $i = 1$ (needs revision)



3) compute $i = 2$ (needs revision)

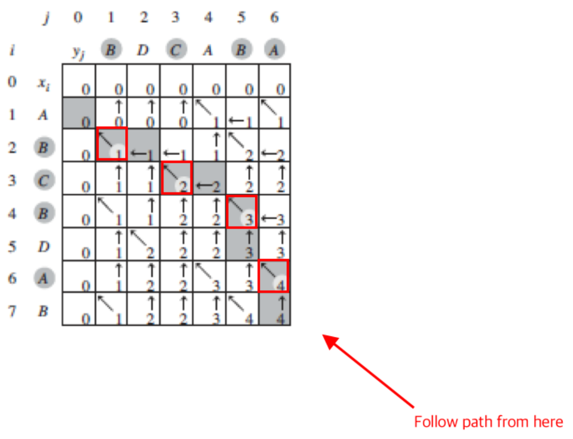


```
LCS-LENGTH(X, Y)
1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i − 1, j − 1] + 1
12             b[i, j] = "↖"
13         elseif c[i − 1, j] ≥ c[i, j − 1]
14             c[i, j] = c[i − 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j − 1]
17             b[i, j] = "←"
18 return c and b
```

4) Compute other entries using the same steps as above

5) Reconstruct LCS

- ∗ Follow the arrows the **lower right-hand corner**
- ∗ Select all '↖'



Here, the result is 'BCBA'

8.

```
PRINT-LCS(c,X,Y,i,J)
    if i == 0 or j == 0
        return
    if X[i] == Y[i]
        PRINT-LCS(c,X,Y,i-1,j-1)
        print X[i]
    elseif c[i-1, j] ≥ c[i, j - 1]
        PRINT-LCS(c,X,Y,i-1,j)
    else
        PRINT-LCS(c,X,Y,i,j-1)

```

9.
```
    LIS(A)

        let B be new array
        j = 0

        for i = 0 to n
            if i = 0
                B[j] = A[i]

            elseif A[i] < max(B)
                REPLACE-NEXT-VALUE-IN-B-BIGGER-THAN-Ai(B,A[i]) #Done using binary search

            elseif A[i] ≤ B[j]
                B[j] = A[i]
            else
                j += 1
                B[j] = A[i]

        return B

```

Correct Solution:

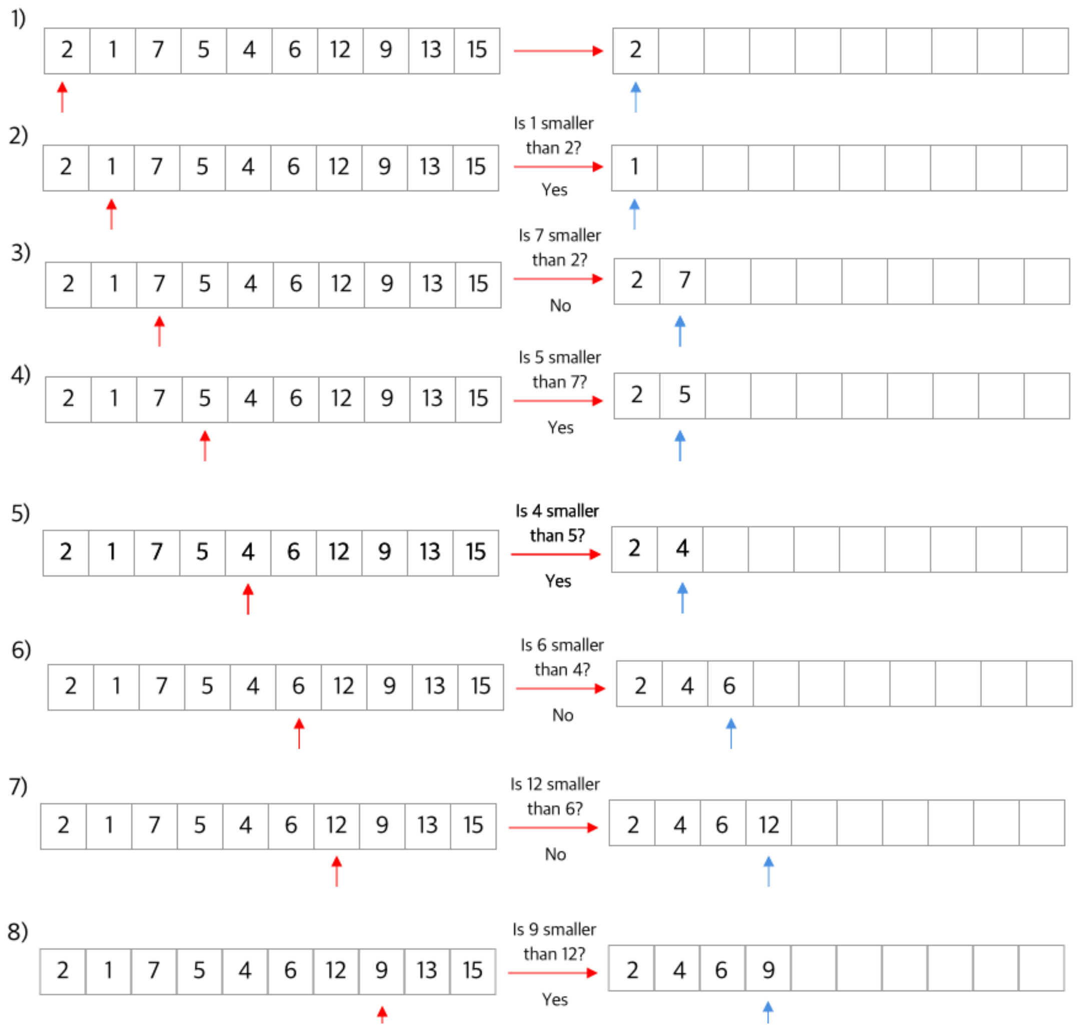
```
    LIS(A)

        let B be new array
        j = 0

        for i = 0 to n
            if i = 0
                B[j] = A[i]

            elseif A[i] ≤ B[j]
                B[j] = A[i]
            else
                j += 1
                B[j] = A[i]

        return B

```

Notes:

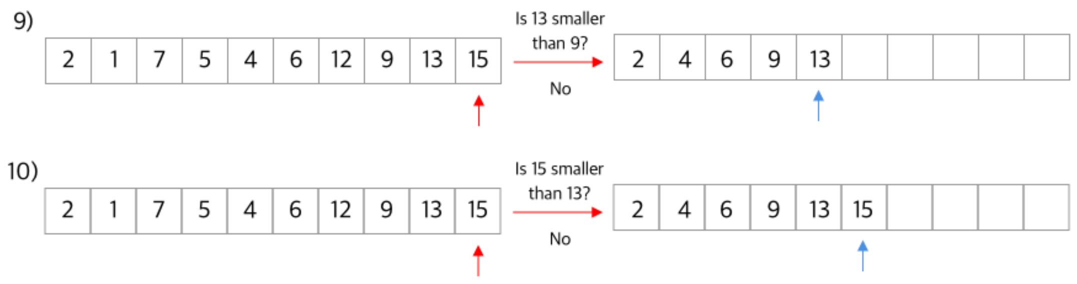
- Longest Increasing Subsequence
  - LIS $\rightarrow$ LCS but a minor variation
    - Is the same as LCS's
      $A =< x_1, x_2, x_3, ..., x_n >$
      $B =< min(A), min(A) + 1, ..., max(A) >$
  - Finds the longest and largest monotonically increasing subset of numbers

  Example:

  $2, 1, 7, 4, 6, 12, 9, 13, 15$

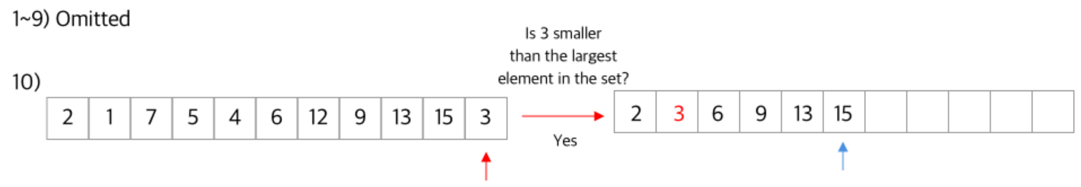


1)
2) Is 1 smaller than 2? Yes
3) Is 7 smaller than 2? No
4) Is 5 smaller than 7? Yes
5) Is 4 smaller than 5? Yes
6) Is 6 smaller than 4? No
7) Is 12 smaller than 6? No
8) Is 9 smaller than 12? Yes



Example 2:

$2, 1, 7, 4, 6, 12, 9, 13, 15, 3$



– Considers values and not indices

  *

Rough Work:

Goal: Find the longest palindromic subsequence, and return its length

Example:

$< A > \rightarrow 1$

$< A, B, A > \rightarrow 3$

$< A, B, B > \rightarrow 2$

$< A, A, B > \rightarrow 2$

Brute Force Method

10. • For each combination of subsequence, check if substring is palinrome
    – If palindrome,

      Check if is the longest

      * If longest, replcae the value of a variable containing the current largest palindromic substring

Improved Solution

- Start at string with full length
- Check if palindrome
  - If palindrome

    Return length
  - if not palindrome
    * Return the length of largest paldromic subsequence by decreasing the RHS bound of the sequence to left by 1

      i.e

      $[A, A, B] \rightarrow [A, A], B \rightarrow$ Return 2
    * Return the length of largest paldromic subsequence by increasing the LHS bound of the sequence to right by 1

      i.e

      $[A, A, B] \rightarrow A, [A, B] \rightarrow$ Return 1

    * If $a \geq b$, return a

      else return b

```
Longest-Palindrome-Subsequence(A,i,j)
    if Palindrome(A,i,J)
        return length(A[i..j])

    a = Longest-Palindrome-Subsequence(A, i, j - 1)
    b = Longest-Palindrome-Subsequence(A, i + 1, j)

    if a $\geq b$
        return a
    else
        return b

```

Here in this case, the algorithm has time complexity of $O(2^n)$.

Improved Solution # 2

- Define 2D table T[i,j]

Let T be 2-D table where $T[i, j]$ contains the length of longest palindromic subsequence $< x_i, ..., x_j >$.

- Define T[i,j] when $i = j$ and $i \neq j$

Now, I need to define $T[i, j]$ when $i = j$ and $i < j$.

When $i = j$, $X[i..j]$ is a part of longest palindromic subsequence with the length of 1.

The same holds for $T[i + 1, j - 1]$.

So, we have $T[i, j] = T[i + 1, j - 1] = 1$.

When $i < j$ and $X[i..j]$ is a part of longest palindromic subsequence, then $T[i, j]$

- Write recursive formula
- Write for-loops

Notes:

- **ith Prefix** of a sequence is a subsequence of a sequence of length i that occurs at the beginning of the sequence.

  Example:

  $X =< A, B, C, B, D, A, B >$

  Then

  $X_4 =< A, B, C, B >$