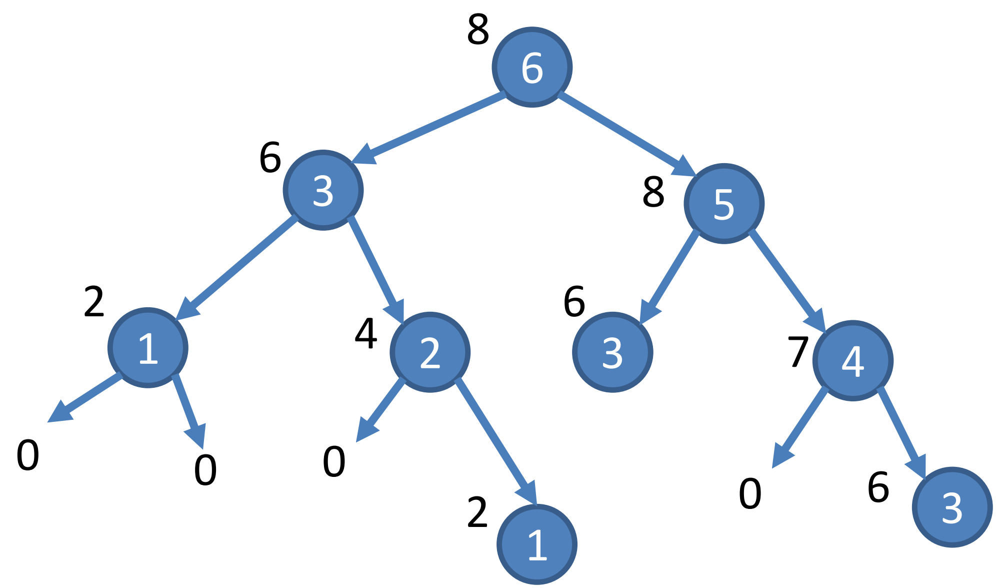
Dynamic Programming

Farrukh Salim Shaikh

Weighted Interval Scheduling (WIS)

WIS(n)

```
M[0] ← 0
for j ←1 to n
        if w_j + M[p(j)] > M[j - 1]
                M[j] = w_j + M[p(j)]
                pred[j] = p(j)
        else
                M[j] = M[j - 1]
                pred[j] = j – 1
return M[n]
```



WIS-FIND-SOLUTION(j)

```
j ← n
while j > 0
        if pred[j] = p(j)
                Output j
        j ← pred[j]
```

Index(i)	Interval	p(i)
1	v1=2	0
2	v2=4	0
3	v3=4	1
4	v4=7	0
5	v5=2	3
6	v6=1	3

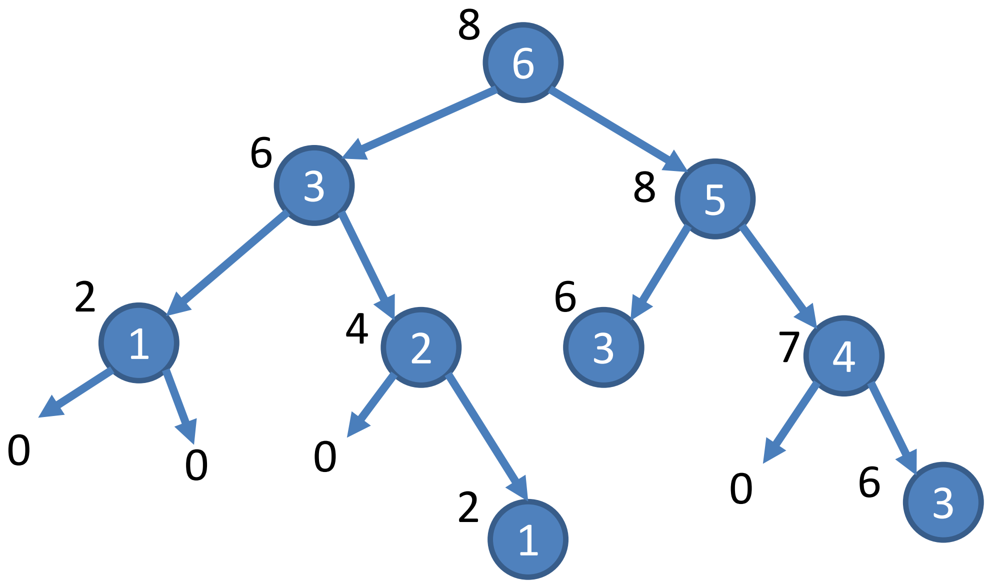
Weighted Interval Scheduling (WIS)

WIS(n)

M[0] ← 0
for j ←1 to n
  if w_j + M[p(j)] > M[j - 1]
    M[j] = w_j + M[p(j)]
    pred[j] = p(j)
  else
    M[j] = M[j - 1]
    pred[j] = j – 1
return M[n]

WIS-FIND-SOLUTION(j)

j ← n
while j > 0
  if pred[j] = p(j)
    Output j
  j ← pred[j]

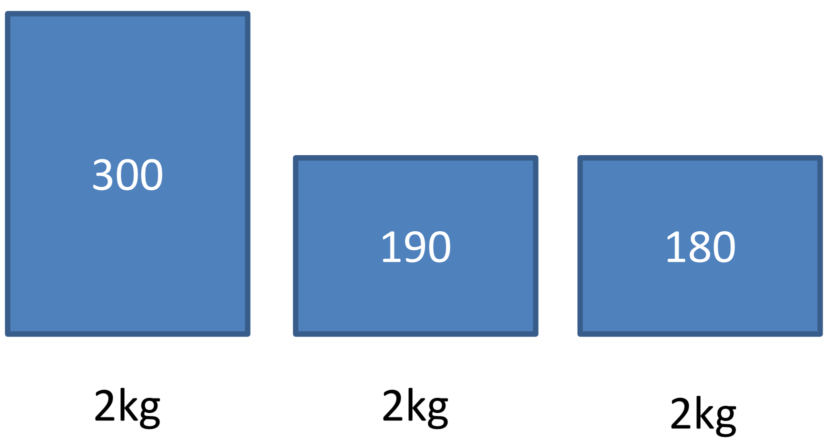


	pred	p	M
6	5	3	8
5	3	3	8
4	0	0	7
3	1	1	6
2	0	0	4
1	0	0	2
0	0	0	0

0/1 Knapsack problem

0/1 Knapsack problem

- Given a set 'S' of 'n' items,
- such that each item i has a positive value v_i and positive weight w_i
- The goal is to find maximum-benefit subset that does not exceed the given weight W.



Maximum weight:
W = 4kg

Optima Solution:
Item B and C

Benefit:
370

Developing a Recursive Solution

- Let 𝓋 be an optimal solution
- Note that presence of item i in 𝓋, does not preclude any other item j.
- Item n (last one) either belongs to 𝓋, or it doesn't.
  - If $n \in$ 𝓋:
    - Optimal solution contains 'n',
    - plus optimal solution of other n – 1 items,
    - But with a reduced maximum weight of $W - w_n$
  - If $n \notin$ 𝓋:
    - Optimal solution if for n – 1 items,
    - with maximum allowed weight W remain unchanged.

Developing a Recursive Solution

- $w_n > W \Rightarrow n \notin \mathcal{v}$
  - $\mathcal{v}(n,W) = \mathcal{v}(n - 1, W)$

- Otherwise, n is either $\in \mathcal{v}$ or $\notin \mathcal{v}$
- If $n \in \mathcal{v}$:
  - $\mathcal{v}(n,W) = v_n + \mathcal{v}(n - 1, W - w_n)$
- If $n \notin \mathcal{v}$:
  - $\mathcal{v}(n,W) = \mathcal{v}(n - 1, W)$

- $\mathcal{v}(n,W) = \mathrm{MAX}(v_n + \mathcal{v}(n - 1, W - w_n), \mathcal{v}(n - 1, W))$

Developing a Recursive Solution

- ʋ$(n,W) = \text{MAX}(v_n +$ ʋ$(n - 1, W - w_n),$ ʋ$(n - 1, W)\ )$

- $$\text{OPT}\ (j,w) = \begin{cases} \text{OPT}\ (j - 1, w) & \text{if } w_j > w \\ \text{MAX}(v_j + \text{OPT}(j - 1, w - w_j), \\ \text{OPT}\ (j - 1, w)\ ) & \text{otherwise} \end{cases}$$

- j is in optimal solution iff.

$$v_j + \text{OPT}(j - 1, w - w_j) \geq \text{OPT}\ (j - 1, w)$$

A recursive algorithm

Knapsack(j,w)
  If j = 0 or w = 0
    return 0
  else if $w_j > w$
    return Knapsack(j – 1, w)
  else
    return MAX(vj + Knapsack(j – 1, w – w_j),
               Knapsack(j – 1, w))

- The initial call is Knapsack(n, W)

A recursive algorithm

```
M-Knapsack(j,w)
  If j = 0 or w = 0
        return 0
  else if M[j,w] is empty
        M[j,w] ← MAX(v_j + Knapsack(j – 1, w – w_j),
                     Knapsack(j – 1, w))
return M[j,w]
```

- This is an example of pseudo-polynomial problem, since it depends on another parameter W that is independent of the problem size.

Dynamic Programming Algorithm

```
Knapsack(j,w)
 for i ← 0 to n          M[i,0] ← 0
 for w ← 0 to W          M[0,w] ← 0

 for j ← 1 to n
        for w ← 0 to W
               if v_j + M[j – 1, w – w_j] ≥ M[j – 1, w]
                      M[j,w] = v_j + M[j – 1, w – wj]
               else
                      M[j,w] = M[j – 1, w))
return M[n,W]
```

Knapsack(j,w)

for i $\leftarrow$ 0 to n $\quad$ M[i,0] $\leftarrow$ 0

for w $\leftarrow$ 0 to W $\quad$ M[0,w] $\leftarrow$ 0

for j $\leftarrow$ 1 to n

for w $\leftarrow$ 0 to W

if $\mathbf{v_j + M[j - 1, w - w_j] \geq M[j - 1, w]}$

$M[j,w] = v_j + M[j - 1, w - wj]$

else

$M[j,w] = M[j - 1, w))$

return M[n,W]

- Complexity = O(nW) (Right?)

Example

- Let W = 9
- wi = {2, 3, 4, 5}
- vi = {3, 4, 5, 7}

- M[j,w] ← MAX($\mathbf{v_j + M[j-1, w-w_j]}$, M[j – 1, w])

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0	0	3	4	5	7	8	10	11	12
5	4	3	0	0	3	4	5	7	8	9	9	12
4	3	2	0	0	3	4	4	7	7	7	7	7
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

Example (contd…)

			Weight									
vi	wi	Index	0	1	2	3	4	5	6	7	8	9
7	5	4	0	0	3	4	5	7	8	10	11	12
5	4	3	0	0	3	4	5	7	8	9	9	12
4	3	2	0	0	3	4	4	7	7	7	7	7
3	2	1	0	0	3	3	3	3	3	3	3	3
		0	0	0	0	0	0	0	0	0	0	0

- More than one optimal solutions might be possible, but DP provides only one optimal solution at a time

Complete DP algorithm

```
Knapsack(j,w)
 for i ← 0 to n          M[i,0] ← 0    s[i] ← 0
 for w ← 0 to W          M[0,w] ← 0

 for j ← 1 to n
         for w ← 0 to W
                 if vj + M[j – 1, w – wj] ≥ M[j – 1, w]
                         M[j,w] = M[j – 1, w – wj]
                         s[j] = 1
                 else
                         M[j,w] = M[j – 1, w))
return M[n,W]
```

Complete DP algorithm

```
Knapsack-Find-Solution
CW = W
for (i = n downto 1)
        if (s[i,CW] == 1)
                output i
                CW = CW – w[i]
```

Complete DP algorithm - II

```
Knapsack(j,w)
 for i ← 0 to n                M[i,0] ← 0        s[i] ← 0
 for w ← 0 to W   M[0,w] ← 0

 for j ← 1 to n
          for w ← 0 to W
                    if vj + M[j – 1, w – wj] ≥ M[j – 1, w]
                         M[j,w] = M[j – 1, w – wj]
                         s[j] = 1
                    else
                         M[j,w] = M[j – 1, w))
CW = W
for (i = n downto 1)
          if (s[i,CW] == 1)
                    output i
                    CW = CW – w[i]
return M[n,W]
```

Longest Common Subsequence

- 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 and Y.

- For example: if X = {A,B,C,B,D,A,B} and Y = {B,D,C,A,B,A}, the sequence {B,C,A} is a common subsequence of both X and Y.

- The sequence {B,C,A} is not a longest common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence {B,C,B,A}, which is also common to both X and Y, has length 4.

- The sequence {B,C,B,A} is an LCS of X and Y, as is the sequence {B,D,A,B}, since X and Y have no common subsequence of length 5 or greater.

- Applications include Biological applications that often need to compare the DNA of two (or more) different organisms. We can say that two DNA strands are similar if one is a substring of the other.

Longest Common Subsequence

- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- In LCS, we have to find the Longest Common Subsequence that is in the same relative order.
- String of length n has 2^n different possible subsequences.

Longest Common Subsequence

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Recurrence Relation for LCS

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

- Case 1: Base Case
- Case 2: If the two alphabets match, add 1 with the value of the diagonal
- Case 3: If the two alphabets do not match, select value from top or left (which ever is greater)

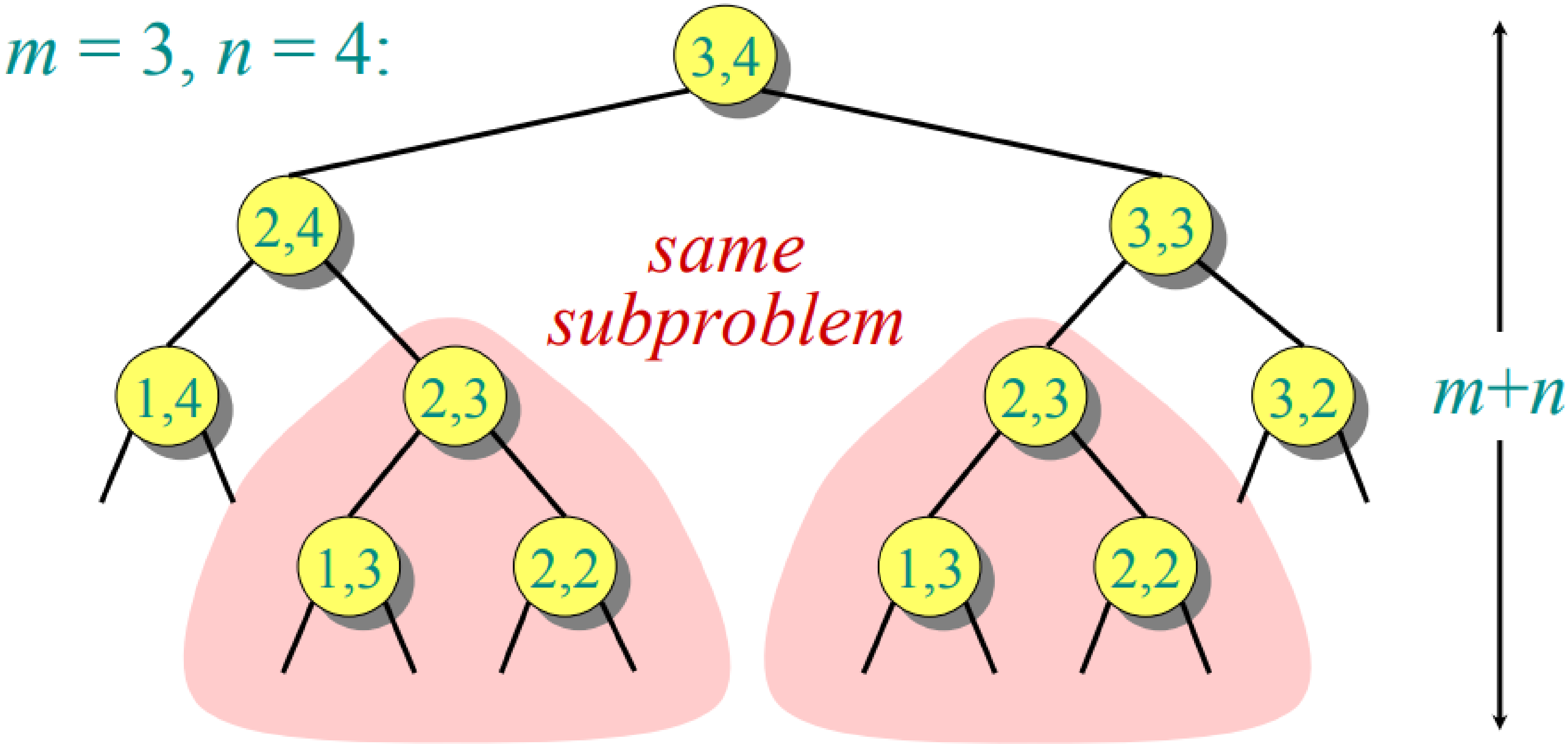
Algorithm for LCS

```
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
```

- Complexity = O(nm) (Right?)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Recursion Tree



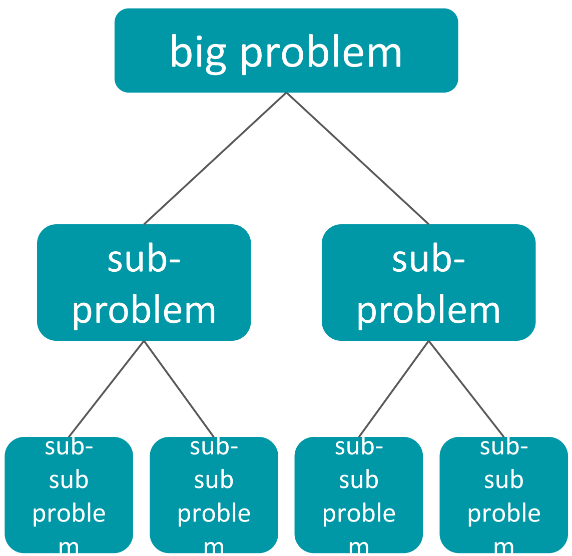
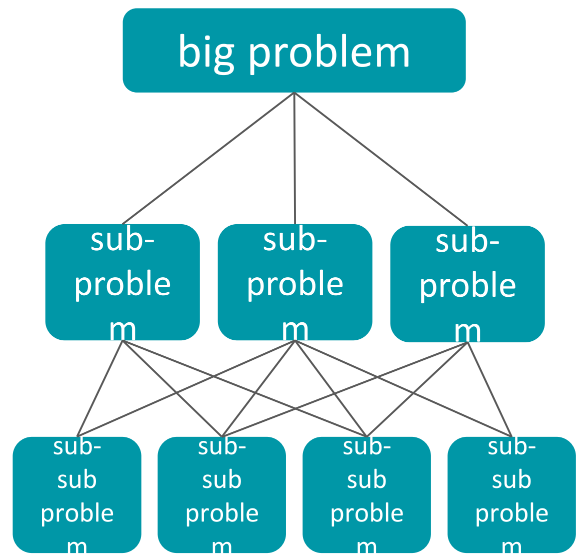
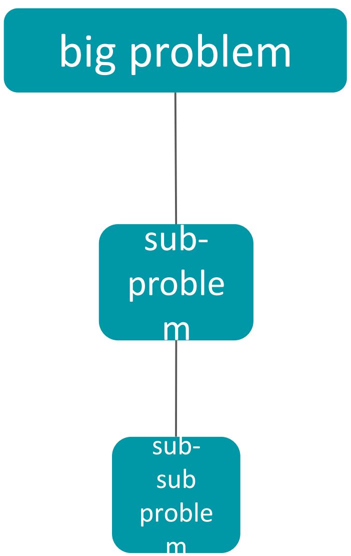
Example # 2

i	j 0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0	0	0	0	0	0	0	0	0	0	0	0
1 p	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 r	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 s	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 i	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 d	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 e	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 n	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 t	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Running Time and Memory O(mn)
Output: Priden

Matrix Chain Multiplication

D&C vs. DP vs. GREEDY

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING	GREEDY
		

Steps for DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

Matrix-chain Multiplication

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- Matrix multiplication is associative. So parenthesizing does not change result. There are 5 possible ways:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Matrix Multiplication

A 2 * **3**

2	3	4
5	6	7

B **3** * 4

2	3	4	5
1	1	1	1
2	2	2	2

A * B 2 * 4

2*2 + 3*1 + 4*2			

❖**Number of multiplication operations performed while multiplying two matrices**

(# Rows of 1st matrix) x (# columns of 1st matrix) x (# Columns of 2nd matrix)

OR

(# Rows of 1st matrix) x (# rows of 2nd matrix) x (# Columns of 2nd matrix)

<u>For example:</u>

<u>No. of multiplication operations when A and B are multiplied = **2 x 3 x 4** = 24 operations</u>

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)

Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

1. **for** $i \leftarrow 1$ **to** p
2.   **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4.   **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Total Number of Multiplications = p*q*r

Matrix-chain Multiplication ...contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Matrix chain multiplication: Dynamic Programming

- DP breaks the problem into subproblems whose solutions can be combined to solve the global subproblem.

A_1	A_2	A_3	A_4	=	$A_{1..4}$
4*5	5*2	2*8	8*7		4*7

- Chain of matrices = A = $\{A_1,A_2,A_3,A_4\}$; index starts from 1
- Dimensions of matrices = P = {4,5,2,8,7} ; index starts from 0

- Let $A_{i..j}$ be the result of matrices i through j.
- It is easy to see that $A_{i..j}$ is a $p_{i-1} * p_j$ matrix.

Matrix chain multiplication: Dynamic Programming

- Let $1 \leq i < j \leq n$

- Let m[i,j] is minimum number of multiplications from i to j, using recursive formulations as:

- If i = j, then m[i,i] = 0 //Only one matrix, diagonal entries

- If i < j, find the product $A_{i..j}$

- This can be split by k, $i \leq k < j$, so final product: $A_{i..k} * A_{k+1..j}$

Matrix chain multiplication: Dynamic Programming

- The optimum time to compute $A_{i..k}$ is m[i,k] and optimum time for $A_{k+1..j}$ is m[k+1,j]

- Since $A_{i..k}$ is a p_{i-1} * p_k matrix and Ak+1..j is $p_{\mathbf{k}}$ * p_j matrix, the number of multiplications will be p_{i-1} * p_k * pj.

$$m[i,i] = 0$$
$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

Matrix chain multiplication: Dynamic Programming

We do not want to calculate m entries recursively. So how should we proceed? We will fill m along the diagonals. Here is how. Set all $m[i, i] = 0$ using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are $m[1, 2], m[2, 3], m[3, 4], \ldots, m[n-1, n]$. $m[1, 2]$, for example is

$$m[1, 2] = m[1, 1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2$$

For example, for m for product of 5 matrices at this stage would be:

$m[1,1]$	$\leftarrow m[1,2]$ $\downarrow$			
	$m[2,2]$	$\leftarrow m[2,3]$ $\downarrow$		
		$m[3,3]$	$\leftarrow m[3,4]$ $\downarrow$	
			$m[4,4]$	$\leftarrow m[4,5]$ $\downarrow$
				$m[5,5]$

Matrix chain multiplication: Dynamic Programming

Next, we compute cost of multiplication for sequences of three matrices. These are $m[1,3], m[2,4], m[3,5], \ldots, m[n-2,n]$. $m[1,3]$, for example is

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 \end{cases}$$

We repeat the process for sequences of four, five and higher number of matrices. The final result will end up in $m[1,n]$.

Matrix chain multiplication: Dynamic Programming

Example: Let us go through an example. We want to find the optimal multiplication order for

$$\underset{(5\times 4)}{A_1} \cdot \underset{(4\times 6)}{A_2} \cdot \underset{(6\times 2)}{A_3} \cdot \underset{(2\times 7)}{A_4} \cdot \underset{(7\times 3)}{A_5}$$

We will compute the entries of the m matrix starting with the base condition. We first fill that main diagonal:

0				
	0			
		0		
			0	
				0

- Here A = { A1,A2,A3,A4,A5} ; starts from index 1
- and P = {5,4,6,2,7,3}; starts from index 0

Matrix chain multiplication: Dynamic Programming

Next, we compute the entries in the first super diagonal, i.e., the diagonal above the main diagonal:

$$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2 = 0 + 0 + 5 \cdot 4 \cdot 6 = 120$$
$$m[2,3] = m[2,2] + m[3,3] + p_1 \cdot p_2 \cdot p_3 = 0 + 0 + 4 \cdot 6 \cdot 2 = 48$$
$$m[3,4] = m[3,3] + m[4,4] + p_2 \cdot p_3 \cdot p_4 = 0 + 0 + 6 \cdot 2 \cdot 7 = 84$$
$$m[4,5] = m[4,4] + m[5,5] + p_3 \cdot p_4 \cdot p_5 = 0 + 0 + 2 \cdot 7 \cdot 3 = 42$$

The matrix m now looks as follows:

0	120			
	0	48		
		0	84	
			0	42
				0

Matrix chain multiplication: Dynamic Programming

We now proceed to the second super diagonal. This time, however, we will need to try two possible values for k. For example, there are two possible splits for computing $m[1,3]$; we will choose the split that yields the minimum:

$$m[1,3] = m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 == 0 + 48 + 5 \cdot 4 \cdot 2 = 88$$

$$m[1,3] = m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 120 + 0 + 5 \cdot 6 \cdot 2 = 180$$

$$\text{the minimum } m[1,3] = 88 \text{ occurs with } k = 1$$

Matrix chain multiplication: Dynamic Programming

Similarly, for $m[2,4]$ and $m[3,5]$:

$$m[2,4] = m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 = 0 + 84 + 4 \cdot 6 \cdot 7 = 252$$
$$m[2,4] = m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 = 48 + 0 + 4 \cdot 2 \cdot 7 = 104$$
$$\text{minimum } m[2,4] = 104 \text{ at } k = 3$$

$$m[3,5] = m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 = 0 + 42 + 6 \cdot 2 \cdot 3 = 78$$
$$m[3,5] = m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 = 84 + 0 + 6 \cdot 7 \cdot 3 = 210$$
$$\text{minimum } m[3,5] = 78 \text{ at } k = 3$$

Matrix chain multiplication: Dynamic Programming

With the second super diagonal computed, the m matrix looks as follow:

0	120	88		
	0	48	104	
		0	84	78
			0	42
				0

Matrix chain multiplication: Dynamic Programming

We repeat the process for the remaining diagonals. However, the number of possible splits (values of k) increases:

$$m[1,4] = m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 104 + 5 \cdot 4 \cdot 7 = 244$$
$$m[1,4] = m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 120 + 84 + 5 \cdot 6 \cdot 7 = 414$$
$$m[1,4] = m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 88 + 0 + 5 \cdot 2 \cdot 7 = 158$$
$$\text{minimum } m[1,4] = 158 \text{ at } k = 3$$

$$m[2,5] = m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 = 0 + 78 + 4 \cdot 6 \cdot 3 = 150$$
$$m[2,5] = m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 = 48 + 42 + 4 \cdot 2 \cdot 3 = 114$$
$$m[2,5] = m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 = 104 + 0 + 4 \cdot 7 \cdot 3 = 188$$
$$\text{minimum } m[2,5] = 114 \text{ at } k = 3$$

Matrix chain multiplication: Dynamic Programming

0	120	88	158	
	0	48	104	114
		0	84	78
			0	42
				0

That leaves the $m[1,5]$ which can now be computed:

$$
\begin{aligned}
m[1,5] &= m[1,1] + m[2,5] + p_0 \cdot p_1 \cdot p_5 = 0 + 114 + 5 \cdot 4 \cdot 3 = 174 \\
m[1,5] &= m[1,2] + m[3,5] + p_0 \cdot p_2 \cdot p_5 = 120 + 78 + 5 \cdot 6 \cdot 3 = 288 \\
m[1,5] &= m[1,3] + m[4,5] + p_0 \cdot p_3 \cdot p_5 = 88 + 42 + 5 \cdot 2 \cdot 3 = 160 \\
m[1,5] &= m[1,4] + m[5,5] + p_0 \cdot p_4 \cdot p_5 = 158 + 0 + 5 \cdot 7 \cdot 3 = 263 \\
&\text{minimum } m[1,5] = 160 \text{ at } k = 3
\end{aligned}
$$

Matrix chain multiplication: Dynamic Programming

Matrix “m”

We thus have the final cost matrix.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

- **Matrix “s”**

and the split k values that led to a minimum m[i, j] value

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Matrix chain multiplication: Dynamic Programming

- Matrix “m” top right value is minimum cost for multiplying five matrices and Matrix ”s” is used to put parenthesis or order in which they will be multiplied to get that minimum cost.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Based on the computation, the minimum cost for multiplying the five matrices is 160 and the optimal order for multiplication is

$$((A_1(A_2A_3))(A_4A_5))$$

Matrix chain multiplication: Dynamic Programming

- **How to put parenthesis by "s" matrix :**
- To find order or parenthesis, start from top right value s[1,5]. At s[1,5], we have k=3 where row value is A1 and column value is A5 so this means that divide A1,A2,A3,A4,A5 into (A1A2A3)(A4A5).
- Now it has been divided into two parts. First part is from 1 to 3 and second from 4 to 5. So look for s[1,3] and s[4,5].
- At s[1,3], we have k=1 where row value is A1 and column value is A3 so this means that divide A1,A2,A3 into (A1)(A2A3) so the complete becomes (A1(A2A3))(A4A5)
- At s[4,5], we have k=4 where row value is A4 and column value is A5 so this means that divide A4,A5 into (A4)(A5) which wont make any effect so the complete becomes (A1(A2A3))(A4A5)
- **Matrix "s"**

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Matrix chain multiplication: Dynamic Programming

Here is the dynamic programming based algorithm for computing the minimum cost of chain matrix multiplication.

```
MATRIX-CHAIN(p, N)
for i = 1, N
do m[i, i] ← 0
for L = 2, N
do
    for i = 1, n − L + 1
    do j ← i + L − 1
        m[i, j] ← ∞
        for k = 1, j − 1
        do t ← m[i, k] + m[k + 1, j] + p_{i−1} · p_k · p_j
            if (t < m[i, j])
              then m[i, j] ← t;  s[i, j] ← k
```

Analysis: There are three nested loops. Each loop executes a maximum n times. Total time is thus $\Theta(n^3)$.

Matrix chain multiplication: Dynamic Programming

The s matrix stores the values k. The s matrix can be used to extracting the order in which matrices are to be multiplied. Here is the algorithm that caries out the matrix multiplication to compute $A_{i..j}$:

```
MULTIPLY(i, j)
if (i = j)
   then return A[i]
   else  k ← s[i, j]
         X ← MULTIPLY(i, k)
         Y ← MULTIPLY(k + 1, j)
         return X · Y
```