Chained Matrix Multiplication

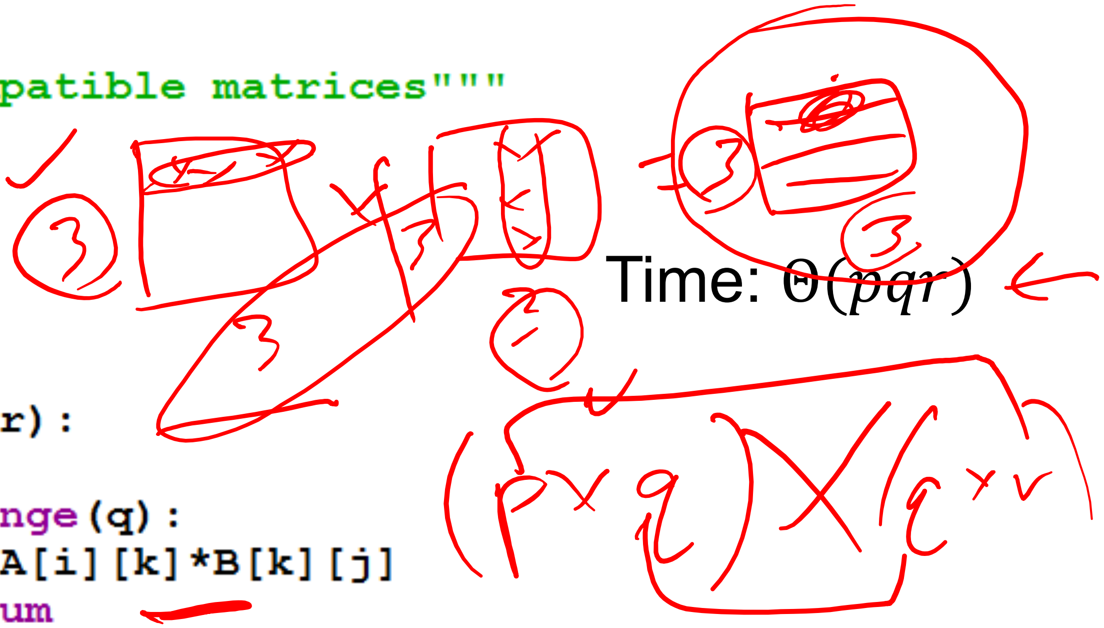
Chained Matrix Multiplication

- Given a chain of matrices $A_1, A_2, \ldots, A_n$, compute the product $A = A_1 \times A_2 \times \ldots \times A_n$ using a minimum number of scalar multiplications
- Matrix A_i has dimension $p_{i-1} \times p_i$
- Amounts to finding the best placement of parentheses.
  - $((A_1 \times A_2) \times A_3)$ vs. $(A_1 \times (A_2 \times A_3))$
  - How many ways for n $=$ 4?
  - How many ways for arbitrary n?

Standard Matrix Multiplication

```
def newMatrix(m,n,val=0,setRandom=False):
    """create m by n matrix initialized to val"""
    if setRandom:
        return [[random()for x in range(n)] for x in range(m)]
    else: return [[val for x in range(n)] for x in range(m)]
```

```
def mulMatrix(A,B):
    """multiply two compatible matrices"""
    p = len(A)
    q = len(A[0])
    r = len(B[0])
    assert q == len(B)
    C = newMatrix(p,r)
    for i in range(p):
        for j in range(r):
            sum = 0
            for k in range(q):
                sum += A[i][k]*B[k][j]
            C[i][j] = sum
    return C
```



Time: $\Theta(pqr)$

- If A is $p \times q$ and B is $q \times r$ computing C requires pqr multiplications.

Does It Matter?

- Given the following matrices:
  - A is 20×2
  - B is 2×30
  - C is 30×12
  - D is 12×8
- Specifically, how many multiplications does it take to compute $A \times B \times C \times D$?
  - First thing you should ask is “can it even be done?”

- Note that matrix multiplication is an associative operation, meaning that the order in which we multiply doesn’t matter.

  A(B(C D)) or (A B)(C D) or A((B C)D)) or ((A B)C)D or (A(B C))D

- However, each of these has a different number of multiplications.
  - A(B(CD)) = $(30 \times 12 \times 8) + (2 \times 30 \times 8) + (20 \times 2 \times 8) = 3{,}680$
  - (AB)(CD) = $(20 \times 2 \times 30) + (30 \times 12 \times 8) + (20 \times 30 \times 8) = 8{,}880$
  - A((BC)D)) = $(2 \times 30 \times 12)+(2 \times 12 \times 8) + (20 \times 2\times8) = 1{,}232$
  - ((AB)C)D = $(20 \times 2 \times 30) + (20 \times 30 \times 12) + (20 \times 12 \times 8) = 10{,}320$
  - (A(BC))D = $(2 \times 30 \times 12) + (20 \times 2 \times 12) + (20 \times 12 \times 8) = 3{,}120$
- Obviously, there is an optimal solution.
  - How do we solve for it?

END

Chained Matrix Multiplication DAC

Recursive Definition

- At the top level, given four matrices, there are three ways of parenthesizing this set into two subsets: (A1) (A2 A3 A4) or (A1 A2) (A3 A4) or (A1 A2 A3) (A4).
- The best way of parenthesizing for this set of four is given by the following: Best(firstSet) + Best(secondSet) + amount to multiply resulting in two matrices.
- Note that, for this example problem, we would need to do the computation three times and pick the optimal one.
- This is simply a recursive definition of the problem.

Solving Matrix Chain by DAC

1. Characterize the structure of an optimal solution.

$$M_{ij} = (M_i \times \cdots \times M_k)(M_{k+1} \times \cdots \times M_j), \text{ some } i \le k < j$$

2. Recursively define the value of an optimal solution.

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

3. Compute the value of an optimal solution top down using recursion.
4. Construct an optimal solution from the information computed in 3.

Divide and Conquer

```
def DAC_MC(p,i,j):
    """Minimum #multiplications to multiply chain M_i to M_j"""
    if i == j:
        return 0
    m[i][j]=math.inf
    for k in range(i,j):
        q = DAC_MC(p,i,k) + DAC_MC(p,k+1,j)+p[i]*p[k+1]*p[j+1]
        if q < m[i][j]:
            m[i][j] = q
    return m[i][j]
```

Time?

$$T(n) = 1 + \sum_{k=1}^{n-1} T(k) + T(n-k) = 1 + 2\sum_{k=1}^{n-1} T(k)$$

Repeated Work

- This approach will compute the correct answer, but it has tons of repeated work. For example:
  - MC(2, 5) takes the min of
    - MC(2,2) + MC(3,5) and ✓
    - MC(2,3) + MC(4,5) and ✓
    - MC(2,4) + MC(5,5) ✓
  - But then MC(2,4) needs:
    - MC(2,2) + MC(3,4) and
    - MC(2,3) + MC(4,4)
  - You can see the repeated work (in red) and this is just the tip of the iceberg.
    - Turns out that this is an exponential algorithm because of the repeated work

$(A_2) \times (A_3 \times A_4 \times A_5)$

Matrix Chain With Memoization

```
def Memo_MC(p):
    m = [[math.inf for x in range(len(p)-1)] for x in range(len(p)-1)]
    return LookUpCost(p,0,len(p)-2)

def LookUpCost(p,i,j):
    if m[i][j]<math.inf: return m[i][j]
    if i == j: m[i][j]=0
    else:
        for k in range(i,j):
            q = LookUpCost(p,i,k)+LookUpCost(p,k+1,j)+p[i]*p[k+1]*p[j+1]
            if q < m[i][j]: m[i][j] = q
    return m[i][j]
```

END

Chained Matrix Multiplication Using Dynamic Programming

Theory

Steps of a Dynamic Programming Solution

- Characterize the structure of an optimal solution.
- Recursively define the value of optimal solution.
- Compute the value of optimal solution **bottom up.**
- Construct an optimal solution from the information computed in 3.

Solving Matrix Chain Using DP

1. Characterize the structure of an optimal solution.

$$M_{ij} = (M_i \times \cdots \times M_k)(M_{k+1} \times \cdots \times M_j), \text{ some } i \le k < j$$

2. Recursively define the value of an optimal solution.

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

3. Compute the value of an optimal solution bottom up.
4. Construct an optimal solution from the information computed in 3.

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min\begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = \quad 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

m+*s*	0	1	2	3	4	5
0	0					
1			×	+		
2			0			
3				0	×	
4					+ 0	
5						0

s	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min\begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = \quad 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

m+s	0	1	2	3	4	5
0	0	15,750				
1		0	2,625			
2			0	750		
3				0	1,000	
4					0	5,000
5						0

s	0	1	2	3	4	5
0		0				
1			1			
2				2		
3					3	
4						4
5						

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = \quad 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

m+s	0	1	2	3	4	5
0	0	15,750	7,875			
1		0	2,625	4,375		
2			0	750	2,500	
3				0	1,000	3,500
4					0	5,000
5						0

s	0	1	2	3	4	5
0		0	0			
1			1	2		
2				2	2	
3					3	4
4						4
5						

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min\begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

m+s	0	1	2	3	4	5
0	0	15,750	7,875	9,375		
1		0	2,625	4,375	7,125	
2			0	750	2,500	5,375
3				0	1,000	3,500
4					0	5,000
5						0

s	0	1	2	3	4	5
0		0	0	2		
1			1	2	2	
2				2	2	2
3					3	4
4						4
5						

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min\begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = \quad 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

m+s	0	1	2	3	4	5
0	0	15,750	7,875	9,375	11,875	
1		0	2,625	4,375	7,125	10,500
2			0	750	2,500	5,375
3				0	1,000	3,500
4					0	5,000
5						0

s	0	1	2	3	4	5
0		0	0	2	2	
1			1	2	2	2
2				2	2	2
3					3	4
4						4
5						

Matrix	Size
A_0	30×35
A_1	35×15
A_2	15×5
A_3	5×10
A_4	10×20
A_5	20×25

$$m[1,4] = \min\begin{cases} m[1,1] + m[2,4] + p_1 p_2 p_5 = 13{,}000 \\ m[1,2] + m[3,4] + p_1 p_3 p_5 = \quad 7{,}125 \\ m[1,3] + m[4,4] + p_1 p_4 p_5 = 11{,}375 \end{cases}$$

(0)((1)(2))((3)(4))(5))

m+*s*	0	1	2	3	4	5
0	0	15,750	7,875	9,375	11,875	15,125
1		0	2,625	4,375	7,125	10,500
2			0	750	2,500	5,375
3				0	1,000	3,500
4					0	5,000
5						0

s	0	1	2	3	4	5
0		0	0	2	2	0
1			1	2	2	2
2				2	2	2
3					3	4
4						4
5						

Chained Matrix Multiplication Using Dynamic Programming

Code

3. Compute Solution Bottom up

```
def DP_MC(p):
    for i in range(len(p)-1):
        m[i][i]=0
    for chainLength in range(2,len(p)-1):
        for i in range(len(p)-chainLength):
            j = i + chainLength - 1
            m[i][j] = math.inf
            for k in range(i,j):
                q = m[i][k]+m[k+1][j]+p[i]*p[k+1]*p[j+1]
                if q < m[i][j]:
                    m[i][j] = q
    return m[0][len(p)-2]
```

Time? $\Theta(n^3)$

4. Construct Solution

- When we find a k that is best, record it

```
for k in range(i, j):
    q = ……
    if (q < m[i][j]):
        m[i][j] = q
        traceback[i][j] = k
```

END

Longest Common Subsequence

Steps of a Dynamic Programming Solution

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

Longest Common Subsequence

- Given strings *x*[1:*m*] and *y*[1:*n*] find a longest subsequence common to both
- Subsequence need not be contiguous

Example

- *Input:* HIEROGLYPHOLOGY and MICHELANGELO
- *Output:* HELLO
- Applications
  - Text processing: edit distance, diff command
  - DNA comparison

A Brute Force Algorithm

- For every subsequence s of *x*, determine if *s* is a subsequence of *y*. Keep the longest match.
  - How do you find if s is a subsequence of y?
- How long does it take to find *s* in *y*[1:*n*]?
- How many subsequences does *x*[1:*m*] have?
- What is the running time?
  - $O(n\ 2^m)$

Notation

- LCS(x,y) is really a set of strings, all of the same length.
    - **Example:** $x = \text{AGCGTAG}$, $y = \text{GTCAGA}$
    - LCS(x,y) = {GTAG, GCAG, GCGA}
- Here, we use LCS(x,y) to denote any longest common string of x and y.
- $|s|$ denotes the length of string s. |LCS(x,y)| = 4.

A Simplification

- First, compute |LCS(x,y)| only. Then extend the algorithm to find actual string.
- Solve the smaller problem on *prefixes* of x and y.
  - Let $c[i,j]$ = |LCS($x[1:i]$, $y[1:j]$)|
  - Then |LCS(x,y)| = $c[m,n]$

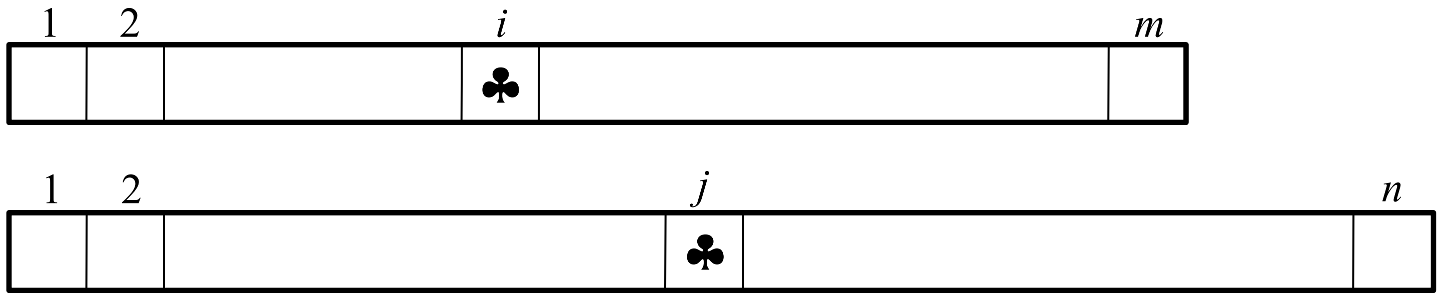
A G A T

A G T T

Claim.

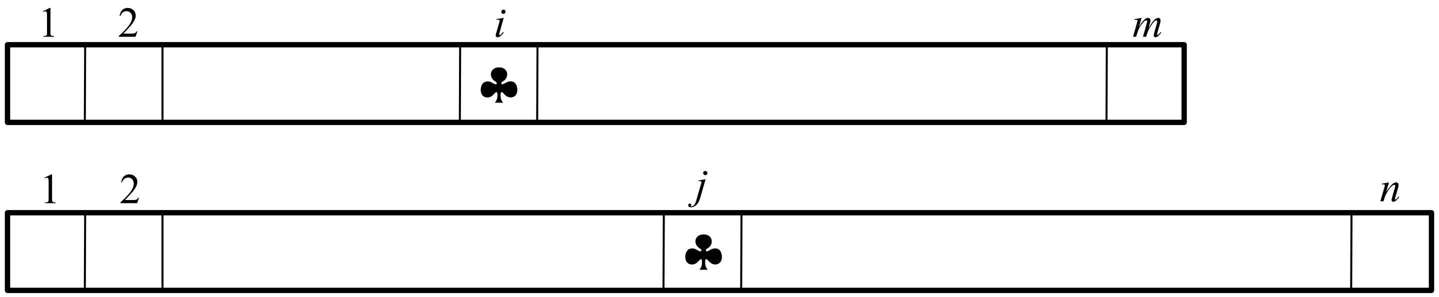
$$c[i,j] = \begin{cases} c[i-1, j-1] + 1, & \text{if } x[i] = y[j] \\ \max\{c[i, j-1], c[i-1, j]\}, & \text{otherwise} \end{cases}$$

Case 1: ($x[i] = y[j]$)



- Let c[i,j] = k and z[1:k] = LCS(x[1:i], y[1:j])
- What is $z[k]$?
  - $z[k] = x[i] = y[j]$, as otherwise can make z longer!
- $z[1:k-1]$ is a CS of x[1:i – 1] and $y[1:j-1]$
  - In fact, $z[1:k-1]$ = LCS(x[1:I – 1], $y[1:j-1]$) (cut-and-paste argument)
- c[i – 1, j – 1] = k – 1, which $\Rightarrow$ $c[i,j]$ = c[I – 1, j – 1] + 1

Case 2: ($x[i] \neq y[j]$)



- Again, let $z[1:k]$ = LCS($x[1:i]$, $y[1:j]$)
- We know that $z[k] \neq x[i]$ **or** $z[k] \neq y[j]$ (or both)

If $z[k] \neq x[i]$,

then LCS($x[1:i]$, $y[1:j]$) = LCS($x[1:i-1]$, $y[1:j]$)

If $z[k] \neq y[j]$,

then LCS($x[1:i]$, $y[1:j]$) = LCS($x[1:i]$, $y[1:j-1]$)

Thus, $c[i,j] = \max\{c[i-1, j], c[i, j-1]\}$

DP Hallmark 1

- *Optimal substructure:* An optimal solution to a problem instance is made up of optimal solutions to subproblem instances.
  - If $z = \text{LCS}(x, y)$, then *any* prefix of z is the LCS of some prefix of x and *some* prefix of y.

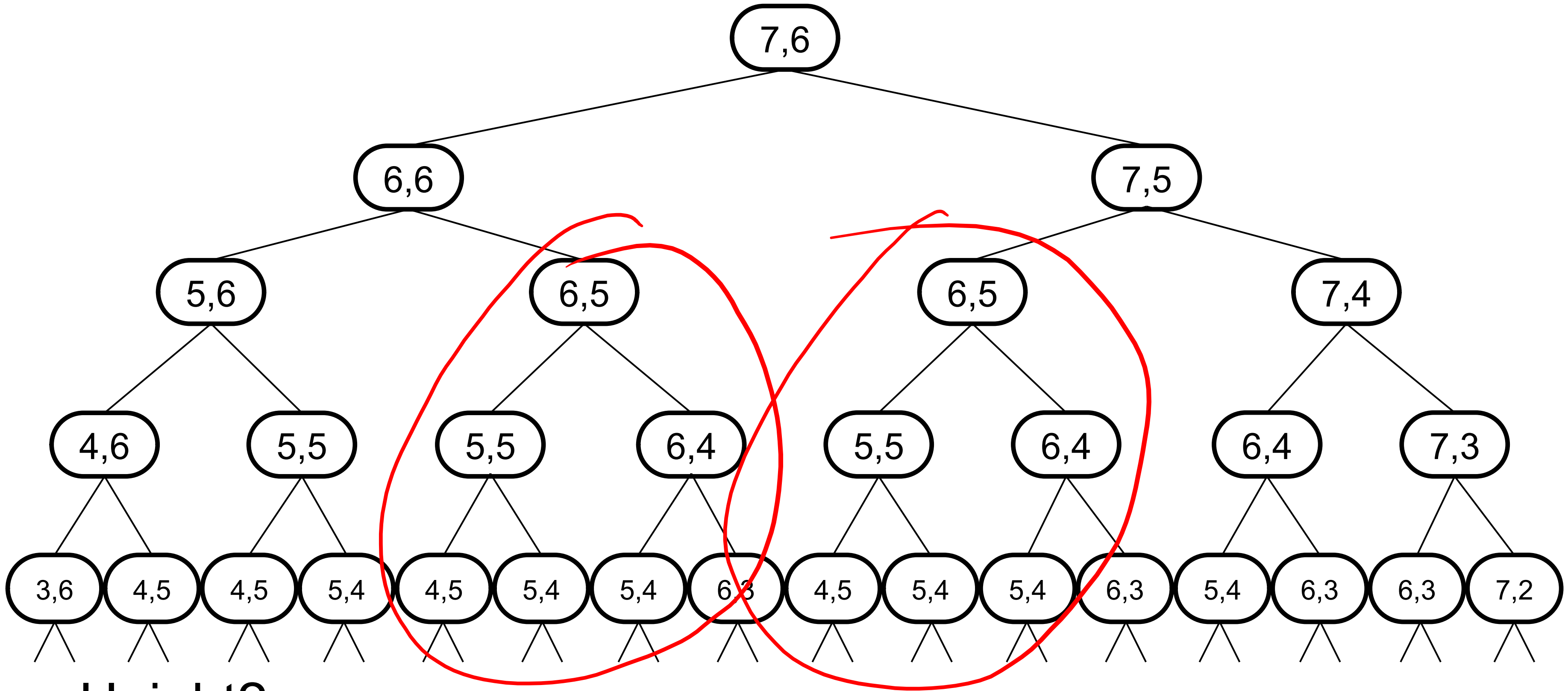
This suggests a strategy: DAC.

A DAC Algorithm

```
LCS($x, y, i, j$)
1  if $i = 0$ or $j = 0$
2      then return 0
3  if $x[i] = y[j]$
4      then $c[i, j] \leftarrow$ LCS($x, y, i - 1, j - 1$) + 1
5      else $c[i, j] \leftarrow \max\{$LCS($x, y, i - 1, j$), LCS($x, y, i, j - 1$)$\}$
6  return $c[i, j]$.
```

- Worst-case time?
  - Always take the **else** clause (line 5)
  - Happens when $x[i] \neq y[j]$, for all i, j

Recursion Tree (m = 7, n = 6)



- Height? $m + n$
- Number of nodes? $> 2^{\min\{m,n\}}$ $\Rightarrow$ exponential time

END

Longest Common Subsequence Using Dynamic Programming

DP Hallmark 2

- *Overlapping subproblems.* A recursive solution contains a *small number* of distinct problem instances repeated *many* times.
- How many distinct subproblems does the LCS have?
- There are *mn* subproblems but an exponential number of generated instances!
- This suggests storing solutions to subproblems, in case they are needed later.

Memoized DAC

```
LCS($x, y, i, j$)
1  if $i = 0$ or $j = 0$
2      then return 0
3  if $c[i, j] =$ NIL
4      then if $x[i] = y[j]$
5              then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
6              else $c[i, j] \leftarrow \max\{$LCS$(x, y, i-1, j)$, LCS$(x, y, i, j-1)\}$
7  return $c[i, j]$.
```

Time? $\Theta(mn)$

Space? $\Theta(mn)$

Steps of a Dynamic Programming Solution

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

Question: What is the correct bottom-up order for the LCS?

$$c[i,j] = \begin{cases} c[i-1,j-1]+1, & \text{if } x[i] = y[j] \\ \max\{c[i,j-1], c[i-1,j]\}, & \text{otherwise} \end{cases}$$

DP Algorithm for LCS Length

```
LCS-LENGTH($x, y, m, n$)
1   for $i \leftarrow 1$ to $m$
2         do $c[i, 0] \leftarrow 0$
3   for $j \leftarrow 0$ to $m$
4         do $c[0, j] \leftarrow 0$
5   for $i \leftarrow 1$ to $m$
6         do for $j \leftarrow 1$ to $n$
7               do if $x[i] == y[j]$
8                     then $c[i, j] \leftarrow c[i-1, j-1] + 1$
9                     else  if $c[i-1, j] \geq c[i, j-1]$
10                              then $c[i, j] \leftarrow c[i-1, j]$
11                              else  $c[i, j] \leftarrow c[i, j-1]$
12  return $c$
```

j

i

	ε	A	G	C	G	T	A	G
ε	0	0	0	0	0	0	0	0
G	0							
T	0							
C	0							
A	0							
G	0							
A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0							
2	T	0			X	X			
3	C	0			X	[illegible]			
4	A	0							
5	G	0							
6	A	0							

Fill the table from top to bottom and, within each row, from left to right.

(3,4) (2,3)

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0						
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1					
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1				
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**			
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1	1	1	1		
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1	1	1	1	1	
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0							
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1	1	1	1	1	1
2	T	0	0						
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1	1	1	1	1	1
2	T	0	0	1					
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0	0	1	1				
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	1	1	1	1	1	1
2	T	0	0	1	1	1			
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0	0	1	1	1	**2**		
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0	0	1	1	1	**2**	2	
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0	0	1	1	1	**2**	2	2
3	C	0							
4	A	0							
5	G	0							
6	A	0							

Fill the rest of the table!

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	**A**	**G**	**C**	**G**	**T**	**A**	**G**
0	**ε**	0	0	0	0	0	0	0	0
1	**G**	0	0	**1**	1	**1**	1	1	**1**
2	**T**	0	0	1	1	1	**2**	2	2
3	**C**	0	0	1	**2**	2	2	2	2
4	**A**	0	**1**	1	2	2	2	**3**	3
5	**G**	0	1	**2**	2	**3**	3	3	**4**
6	**A**	0	**1**	2	2	3	3	**4**	4

- Time = $\Theta(mn)$
- Space = $\Theta(mn)$ but can reduced to $\Theta(\min\{m,n\})$

Example: x = AGCGTAG, y = GTCAGA

		0	1	2	3	4	5	6	7
		ε	A	G	C	G	T	A	G
0	ε	0	0	0	0	0	0	0	0
1	G	0	0	**1**	1	**1**	1	1	**1**
2	T	0	0	1	1	1	**2**	2	2
3	C	0	0	1	**2**	2	2	2	2
4	A	0	**1**	1	2	2	2	**3**	3
5	G	0	1	**2**	2	**3**	3	3	**4**
6	A	0	**1**	2	2	3	3	**4**	4

GTAG

- What is a longest common substring?
- How do you find others?

Reconstructing the LCS

```
LCS($x, y, m, n$)
1  $(i, j) \leftarrow (m, n)$
2  while $i \neq 0$ and $j \neq 0$
3      do if $x[i] == y[j]$
4          then Output $x[i]$
5              $(i, j) \leftarrow (i-1, j-1)$
6          else if $c[i-1, j] > c[i, j-1]$
7              then $(i, j) \leftarrow (i-1, j)$
8              else $(i, j) \leftarrow (i, j-1)$
```

Traverse in $O(m + n)$ time.

END