Dynamic Programming
CISC5835, Algorithms for Big Data
CIS, Fordham Univ.

Instructor: X. Zhang

Rod Cutting Problem

- A company buys long steel rods (of length n), and cuts them into shorter one to sell
  - integral length only
  - cutting is free
  - rods of diff lengths sold for diff. price, e.g.,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Best way to cut the rods?
  - n=4: no cutting: $9, 1 and 3: 1+8=$9, 2 and 2: 5+5=$10
  - n=5: ?

Rod Cutting Problem Formulation

- Input:
  - a rod of length n
  - a table of prices p[1…n] where p[i] is price for rod of length i
- Output
  - determine maximum revenue r_n obtained by cutting up the rod and selling all pieces
- Analysis solution space (how many possibilities?)
  - how many ways to write n as sum of positive integers?
    - 4=4, 4=1+3, 4=2+2
  - # of ways to cut n: $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$.

Rod Cutting Problem Formulation

- // return r_n: max. revenue
- int Cut_Rod (int p[1…n], int n)

- Divide-and-conquer?
  - how to divide it into smaller one?
  - we don't know we want to cut in half…

Rod Cutting Problem

- // return r_n: max. revenue for rod of length n
- int Cut_Rod (int n, int p[1…n])

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Start from small
  - n=1, r_1=1 //no possible cutting
  - n=2, r_2=5 // no cutting (if cut, revenue is 2)
  - n=3, r_3=8 //no cutting
  - r_4=9 (max. of p[4], p[1]+r_3, p[2]+r_3, p[3]+r_1)
  - r_5 = max (p[5], p[1]+r_4, p[2]+r_2, p[3]+r_2, p[4]+r_1)
  - …

Rod Cutting Problem

- // return r_n: max. revenue for rod size n
- int Cut_Rod (int n, int p[1…n])

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Given a rod of length n, consider first rod to cut out
  - if we don't cut it at all, max. revenue is p[n]
  - if first rod to cut is1: max. revenue is p[1]+r_{n-1}
  - if first rod to cut out is 2: max. revenue is p[2]+r_{n-2}, …
  - max. revenue is given by maximum among all the above options
- r_n = max (p[n], p[1]+r_{n-1}, p[2]+r_{n-2}, …, p[n-1]+r_1)

Optimal substructure

- // return r_n: max. revenue for rod size n
- int Cut_Rod (int n, int p[1…n])

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- r_n = max (p[n], p[1]+r_{n-1}, p[2]+r_{n-2}, …, p[n-1]+r_1)
- **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solutions to problems of smaller size (1, 2, 3, … n-1).

Rod Cutting Problem

- // return r_n: max. revenue for rod size n
- int Cut_Rod (int p[1…n], int n)

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

  - r_n = max (p[n], p[1]+r_{n-1}, p[2]+r_{n-2}, …, p[n-1]+r_1)

```
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
```

Recursive Rod Cutting

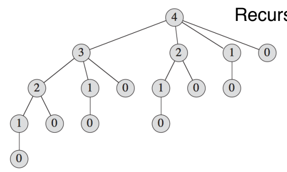
```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Running time T(n)

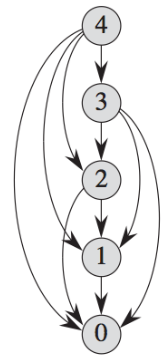
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

Closed formula: $T(n)=2^n$

Recursive calling tree: n=4



Subproblems Graph



- Avoid recomputing subproblems again and again by storing subproblems solutions in memory/table (hence "programming")
  - trade-off between space and time
- Overlapping of subproblems

Dynamic Programming

- Avoid recomputing subproblems again and again by storing subproblems solutions in memory/ table (hence "programming")
  - trade-off between space and time
- Two-way to organize
  - top-down with memoization
    - Before recursive function call, check if subproblem has been solved before
    - After recursive function call, store result in table
  - bottom-up method
    - Iteratively solve smaller problems first, move the way up to larger problems

Memoized Cut-Rod

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array      // stores solutions to all problems
for i = 0 to n
    r[i] = −∞                   // initialize to an impossible negative value
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)   // A recursive function
if r[n] ≥ 0                     // If problem of given size (n) has been
    return r[n]                 solved before, just return the stored result
if n == 0
    q = 0
else q = −∞
    for i = 1 to n              // same as before…
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Memoized Cut-Rod: running time

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array   // stores solutions to all problems
for i = 0 to n
    r[i] = −∞                // initialize to an impossible negative value
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)    // A recursive function
if r[n] ≥ 0              // If problem of given size (n) has been
    return r[n]          solved before, just return the stored result
if n == 0
    q = 0
else q = −∞
    for i = 1 to n       // same as before…
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Bottom-up Cut-Rod

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array     // stores solutions to all problems
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j                       // Solve subproblem j, using
        q = max(q, p[i] + r[j − i])      solution to smaller subproblems
    r[j] = q
return r[n]
```

Running time: 1+2+3+..+n-1=O(n²)

Bottom-up Cut-Rod (2)

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

What if we want to know who to achieve r[n]?
i.e., how to cut?
i.e., n=n_1+n_2+…n_k, such that p[n_1]+p[n_2]+…+p[n_k]=r_n

Recap

- We analyze rod cutting problem
- Optimal way to cut a rod of size n is found by
  - 1) comparing optimal revenues achievable after cutting out the first rod of varying len,
    - This relates solution to larger problem to solutions to subproblems
  - 2) choose the one yield largest revenue

maximum (contiguous) subarray

- Problem: find the contiguous subarray within an array (*containing at least one number*) which has largest sum (midterm lab)
  - If given the array [-2,1,-3,4,-1,2,1,-5,4],
  - contiguous subarray [4,-1,2,1] has largest sum = 6
- Solution to midterm lab
  - brute-force: n^2 or n^3
  - Divide-and-conquer: T(n)=2 T(n/2)+O(n), T(n)=nlogn
  - Dynamic programming?

Analyze optimal solution

- Problem: find contiguous subarray with largest sum
- Sample Input: [-2,1,-3,4,-1,2,1,-5,4] (array of size n=9)
- How does solution to this problem relates to smaller subproblem?
  - If we divide-up array (as in midterm)
  - [-2,1,-3,4,-1,2,1,-5,4] //find MaxSub in this array

  [-2,1,-3,4,-1]   [2,1,-5,4]

  still need to consider subarray that spans both halves

  This does not lead to a dynamic programming sol.
- Need a different way to define smaller subproblems!

Analyze optimal solution

- Problem: find contiguous subarray with largest sum

A	-2	1	-3	4	-1	2	1	-5	4
Index	1	2	3	4	5	6	7	8	9

- MSE(k), max. subarray ending at pos k, among all subarray ending at k (A[i…k] where i<=k), the one with largest sum
  - MSE(1), max. subarray ending at pos 1, is A[1..1], sum is -2
  - MSE(2), max. subarray ending at pos 2, is A[2..2], sum is 1
  - MSE(3) is A[2..3], sum is -2
  - MSE(4)?

Analyze optimal solution

A	-2	1	-3	4	-1	2	1	-5	4
Index	1	2	3	4	5	6	7	8	9

- MSE(k) and optimal substructure
  - MSE(3): A[2..3], sum is -2 (red box)
  - MSE(4): two options to choose
    - (1) either grow MSE(3) to include pos 4
      - subarray is then A[2..4], sum is MSE(3)+A[4]=-2+A[4]=2
    - (2) or start afresh from pos 4
      - subarray is then A[4…4], sum is A[4]=4 (better)
    - Choose the one with larger sum, i.e.,
      - **MSE(4) = max (A[4], MSE(3)+A[4])**

How a problem's optimal solution can be derived from optimal solution to smaller problem

Analyze optimal solution

A	-2	1	-3	4	-1	2	1	-5	4
Index	1	2	3	4	5	6	7	8	9

MSE(4)=4, array is A[4…4]

- MSE(k) and optimal substructure
  - Max. subarray ending at k is the larger between A[k…k] and Max. subarray ending at k-1 extended to include A[k]

    MSE(k) = max (A[k], MSE(k-1)+A[k])
  - **MSE(5)= , subarray is**
  - **MSE(6)**
  - **MSE(7)**
  - **MSE(8)**
  - **MSE(9)**

Analyze optimal solution

A	-2	1	-3	4	-1	2	1	-5	4
Index	1	2	3	4	5	6	7	8	9

- Once we calculate MSE(1) … MSE(9)
  - MSE(1)=-2, the subarray is A[1..1]
  - MSE(2)=1, the subarray is A[2..2]
  - MSE(3)=-2, the subarray is A[2..3]
  - MSE(4)=4, the subarray is A[4…4]
  - … MSE(7)=6, the subarray is A[4…7]
  - MSE(9)=4, the subarray is A[9…9]
- What's the maximum subarray of A?
  - well, it either ends at 1, or ends at 2, …, or ends at 9
  - Whichever yields the largest sum!

Idea to Pseudocode

A	-2	1	-3	4	-1	2	1	-5	4
Index	1	2	3	4	5	6	7	8	9

- Calculate MSE(1) … MSE(n)
  - MSE(1)= A[1]
  - MSE(i) = max (A[i], A[i]+MSE(i-1));
- Return maximum among all MSE(i), for i=1, 2, …n

```
(int, start,end) MaxSubArray (int A[1…n])
{
   // Use array MSE to store the MSE(i)
   MSE[1]=A[1];
   max_MSE = MSE[1];

  for (int i=2;i<=n;i++)
  {
      MSE[i] = ??

      if (MSE[i] > max_MSE) {
          max_MSE = MSE[i];
          end = i;
      }
  }
  return (max_MSE, start, end)
}
```

Practice:
1) fill in ??
2) How to find out the starting index of the max. subarray, i.e., the start parameter?

Running time Analysis

```
int MaxSubArray (int A[1…n], int & start,
    int & end)
{
   // Use array MSE to store the MSE(i)
   MSE[1]=A[1];
   max_MSE = MSE[1];

  for (int i=2;i<=n;i++)
  {
      MSE[i] = ??

      if (MSE[i] > max_MSE) {
          max_MSE = MSE[i];
          end = i;
      }
  }
  return max_MSE;
}
```

- It's easy to see that running time is O(n)
  - a loop that iterates for n-1 times
- Recall other solutions:
  - brute-force: n^2 or n^3
  - Divide-and-conquer: nlogn
- Dynamic programming wins!

What is DP? When to use?

- We have seen several optimization problems
  - brute force solution
  - divide and conquer
  - dynamic programming
- To what kinds of problem is DP applicable?
  - **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solution to problem of smaller size (1, 2, 3, … n-1).
  - **Overlapping subproblems**: small subproblem space and common subproblems

Optimal substructure

- **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solution to problem of smaller size (1, 2, 3, … n-1).
- Rod cutting: find r_n (max. revenue for rod of len n)

Sol to problem instance of size n

Sol to problem instance of size n-1, n-2, … 1

r_n = max (p[1]+r_{n-1}, p[2]+r_{n-2}, p[3]+r_{n-3},…, p[n-1]+r_1, p[n])

  - A recurrence relation (recursive formula)

- => Dynamic Programming: Build an optimal solution to the problem from solutions to subproblems
  - We solve a range of sub-problems as needed

Optimal substructure in Max. Subarray

- **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solution to problem of smaller size (1, 2, 3, … n-1).
  - Max. Subarray Problem:

-2	1	-3	4	-1	2	1	-5	4
1	2	3	4	5	6	7	8	9

  - MSE(i) = max (A[i], MSE(i-1)+A[i])

Max. Subarray Ending at position i is the either the max. subarray ending at pos i-1 extended to pos i; or just made up of A[i]

  - Max Subarray = max (MSE(1), MSE(2), …MSE(n))

Overlapping Subproblems

- space of subproblems must be "small"
  - total number of distinct subproblems is a polynomial in input size (n)
  - a recursive algorithm revisits same problem repeatedly, i.e., optimization problem has ***overlapping subproblems***.
- DP algorithms take advantage of this property
  - solve each subproblem once, store solutions in a table
  - Look up table for sol. to repeated subproblem using constant time per lookup.
- In contrast: divide-and-conquer solves new subproblems at each step of recursion.

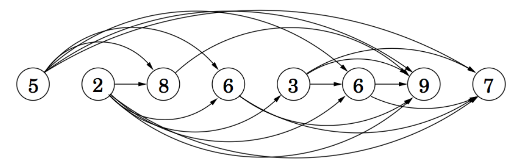
Longest Increasing Subsequence

- Input: a sequence of numbers given by an array a
- Output: a longest **subsequence** (a subset of the numbers taken in order) that is increasing (ascending order)
- Example, given a sequence
  - 5, 2, 8, 6, 3, 6, 9, 7
  - There are many increasing subsequence: 5, 8, 9; or 2, 9; or 8
  - The longest increasing subsequence is:
    2, 3, 6, 9 (length is 4)

LIS as a DAG

- Find longest increasing subsequence of a sequence of numbers given by an array **a**

  5, 2, 8, 6, 3, 6, 9, 7



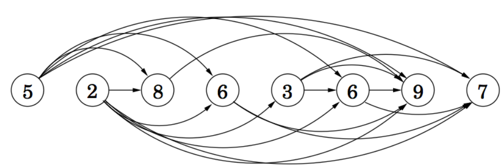
Observation:

- If we add directed edge from smaller number to larger one, we get a DAG.
- A path (such as 2,6,7) connects nodes in increasing order
- LIS corresponds to longest path in the graph.

Graph Traversal for LIS

- Find longest increasing subsequence of a sequence of numbers given by an array **a**

  5, 2, 8, 6, 3, 6, 9, 7

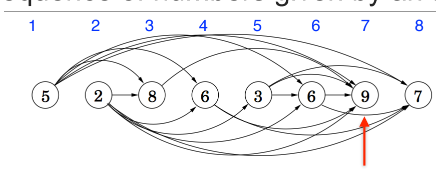


Observation:

- LIS corresponds to longest path in the graph.
- Can we use graph traversal algorithms here?
  - BFS or DFS?
  - Running time

Dynamic Programming Sol: LIS

- Find Longest Increasing Subsequence of a sequence of numbers given by an array **a**



Let L(n) be the length of LIS ending at n-th number

L(1) = 1, LIS ending at pos 1 is 5

L(2) = 1, LIS ending at pos 2 is 2

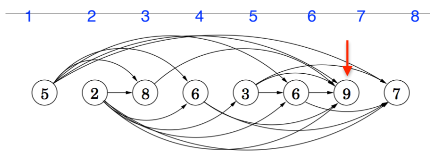
L(7)= // how to relate to L(1), …L(6)?

- Consider LIS ending at a[7] (i.e., 9). What's the number before 9?

  …. ? ,9

Dynamic Programming Sol: LIS

- Given a sequence of numbers given by an array **a**



Let L(n) be length of LIS ending at n-th number

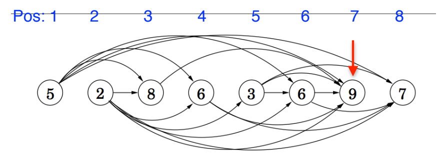
Consider **all increasing subsequence** ending at a[7] (i.e., 9).

- What's the number before 9?
  - It can be either NULL, or 6, or 3, or 6, 8, 2, 5 (all those numbers pointing to 9)
  - If the number before 9 is 3 (a[5]), what's max. length of this seq? L(5)+1 where the seq is 3, 9

LIS ending at pos 5

Dynamic Programming Sol: LIS

- Given a sequence of numbers given by an array **a**



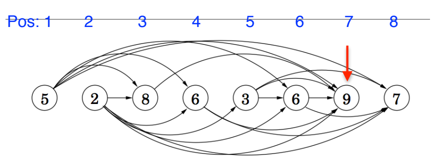
Let L(n) be length of LIS ending at n-th number

Consider **all increasing subsequence** ending at a[7] (i.e., 9).

- It can be either NULL, or 6, or 3, or 6, 8, 2, 5 (all those numbers pointing to 9)
  - L(7)=max(1, L(6)+1, L(5)+1, L(4)+1, L(3)+1, L(2)+1, L(1)+1)
  - L(8)=?

Dynamic Programming Sol: LIS

- Given a sequence of numbers given by an array **a**



Let L(n) be length of LIS ending at n-th number.

Recurrence relation:

$$L(j) = 1 + \max\{L(i) : (i,j) \in E\}$$

Note that the i's in RHS is always smaller than the j

- How to implement? Running time?
- LIS of sequence = Max (L(i), 1<=i<=n) // the longest among all

Next, two-dimensional subproblem space

i.e., expect to use two-dimensional table

Longest Common Subseq.

- Given two sequences

  $X = \langle x_1, x_2, \dots, x_m \rangle$

  $Y = \langle y_1, y_2, \dots, y_n \rangle$

  find ***a*** maximum length common subsequence (LCS) of X and Y
- *E.g.:*

  $X = \langle A, B, C, B, D, A, B \rangle$
- Subsequence of X:
  - A subset of elements in the sequence taken in order but not necessarily consecutive

    $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc

Example

$X = \langle A, B, C, B, D, A, B \rangle$ $X = \langle A, B, C, B, D, A, B \rangle$

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

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)
- BCBA = LCS(X,Y): functional notation, but is it not a function
- $\langle B, C, A \rangle$, however is not a LCS of X and Y

Brute-Force Solution

- Check every subsequence of *X[1 . . m]* to see if it is also a subsequence of *Y[1 .. n].*
- There are 2^m subsequences of *X* to check
- Each subsequence takes O(n) time to check
  - scan *Y* for first letter, from there scan for second, and so on
- Worst-case running time: $O(n2^m)$
  - Exponential time too slow

Towards a better algorithm

Simplification:

1. Look at length of a longest-common subsequence
2. Extend algorithm to find the LCS itself later

Notation:

- Denote **length of a sequence *s*** by |*s*|
- Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$ we define the ***i*-th prefix of *X* as** (for i = 0, 1, 2, …, m)

  $X_i = \langle x_1, x_2, \dots, x_i \rangle$
- Define:

  $c[i, j] = |\, LCS(X_i, Y_j) = |LCS(X[1..i], Y[1..j])|$:

  the length of a LCS of sequences $X_i = \langle x_1, x_2, \dots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \dots, y_j \rangle$
- $|LCS(X,Y)| = c[m,n]$ //this is the problem we want to solve

Find Optimal Substructure

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$
- To find LCS (X,Y) is to find c[m,n]

c[i, j] = | LCS (X_i, Y_j) |
//length LCS of i-th prefix of X and j-th prefix of Y
// X[1..i], Y[1..j]

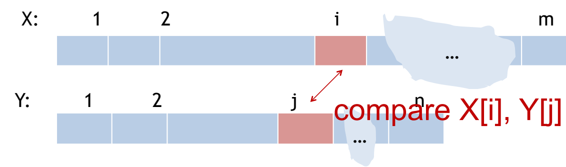
- How to solve c[i,j] using sol. to smaller problems?
  - what's the smallest (base) case that we can answer right away?
  - How does c[i,j] relate to c[i-1,j-1], c[i,j-1] or c[i-1,j]?

Recursive Formulation

Base case: c[i, j] = 0 if i = 0 or j = 0
LCS of an empty sequence, and any sequence is empty

General case:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise (i.e., if } X[i] \neq Y[j]) \end{cases}$$



Recursive Solution. Case 1

Case 1: X[i] ==Y[j]

e.g.: $X_4 = \langle A, B, D, E \rangle$

$Y_3 = \langle Z, B, E \rangle$

- Choice: include one element into common sequence (E) and solve resulting subproblem

c[4, 3] = $\underline{c[4 - 1, 3 - 1]}$ + 1

LCS of $X_3 = \langle A, B, D \rangle$ and $Y_2 = \langle Z, B \rangle$

- Append X[i] = Y[j] to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and Y_{j-1}

Recursive Solution. Case 2

Case 2: X[i] ≠ Y[j]

e.g.: $X_4 = \langle A, B, D, G \rangle$

$Y_3 = \langle Z, B, D \rangle$

Either the G or the D
is not in the LCS
(they cannot be both in LCS)

$$c[i, j] = \max \{ \underbrace{c[i-1, j]}_{\text{If we ignore last element in Xi}}, \underbrace{c[i, j-1]}_{\text{If we ignore last element in Yj}} \}$$

- Must solve two problems
  - find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
  - find a LCS of X_i and Y_{j-1} : $X_i = \langle A, B, D, G \rangle$ and $Y_{j-1} = \langle Z, B \rangle$

Recursive algorithm for LCS

```
// X, Y are sequences, i, j integers
//return length of LCS of X[1…i], Y[1…j]
LCS(X, Y, i, j)
    if i==0 or j ==0
        return 0;
    if X[i] == Y[ j] // if last element match
    then
         c[i, j] ←LCS(X, Y, i–1, j–1) + 1
    else
       c[i, j] ←max{LCS(X, Y, i–1, j),
                    LCS(X, Y, i, j–1)}
```


Optimal substructure & Overlapping Subproblems

- A recursive solution contains a "small" number of distinct subproblems repeated many times.
  - e.g., C[5,5] depends on C[4,4], C[4,5], C[5,4]
  - Exercise: Draw there subproblem dependence graph
    - each node is a subproblem
    - directed edge represents "calling", "uses solution of" relation
- Small number of distinct subproblems:
  - total number of distinct LCS subproblems for two strings of lengths *m* and *n* is *mn*.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS(X, Y, i, j)
   if c[i, j] = NIL   // LCS(i,j) has not been solved yet
        then if x[i] = y[j]
                then c[i, j] ←LCS(x, y, i–1, j–1) + 1
                else c[i, j] ←max{LCS(x, y, i–1, j),
                                  LCS(x, y, i, j–1)}
```

Same as before

Bottom-Up

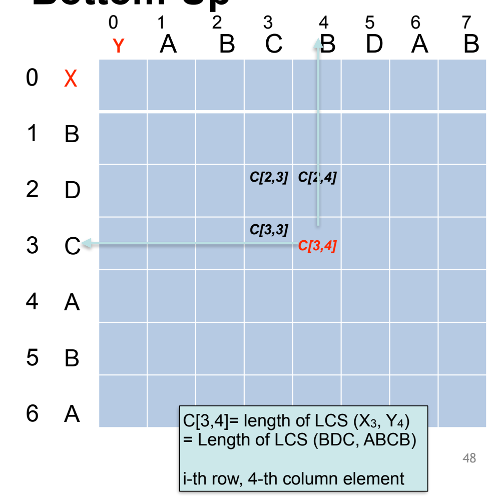
Initialization: base case
c[i,j] = 0 if i=0, or j=0

//Fill table row by row
// from left to right

```
for (int i=1; i<=m;i++)
   for (int j=1;j<=n;j++)
     update c[i,j]
```

return c[m, n]

Running time = Θ(mn)



Dynamic-Programming Algorithm

Reconstruct LCS tracing backward:

how do we get value of C[i,j] from? (either C[i-1,j-1]+1, C[i-1,j], C[i, j-1)

as red arrow indicates…

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Output B Output C Output B Output A

Matrix

Matrix: a 2D (rectangular) array of numbers, symbols, or expressions, arranged in rows and columns.

e.g., a 2 × 3 matrix (there are two rows and three columns)

$$\begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}.$$

Each element of a matrix is denoted by a variable with two subscripts, $a_{2,1}$ element at second row and first column of a matrix A.

an m × n matrix A:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Matrix Multiplication

Matrix Multiplication:

Dimension of A, B, and A x B?

Matrix A, Matrix B, Product

$$\begin{bmatrix} 1 & 4 & 6 & 10 \\ 2 & 7 & 5 & 3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 6 \\ 2 & 7 & 5 \\ 9 & 0 & 11 \\ 3 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 93 & 42 & 92 \\ 70 & 60 & 102 \end{bmatrix}$$

© mathwarehouse.com

$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

Total (scalar) multiplication: 4x2x3=24

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

Total (scalar) multiplication: $n_2 x n_1 x n_3$

Multiplying a chain of Matrix

Given a sequence/chain of matrices, e.g., A_1, A_2, A_3 there are different ways to calculate $A_1A_2A_3$

1. $(A_1A_2)A_3)$

2. $(A_1(A_2A_3))$

Dimension of A_1: 10 x 100

A_2: 100 x 5

A_3: 5 x 50

all yield the same result

But not same efficiency

Matrix Chain Multiplication

Given a chain <A_1, A_2, … A_n> of matrices, where matrix A_i has dimension p_{i-1}x p_i, find optimal fully parenthesize product $A_1A_2…A_n$ that minimizes number of scalar multiplications.

Chain of matrices <A_1, A_2, A_3, A_4>: five distinct ways

A_1: p_1 x p_2 **A_2**: p_2 x p_3 **A_3**: p_3 x p_4 **A_4**: p_4 x p_5

$(A_1(A_2(A_3A_4)))$ ← # of multiplication: $p_3p_4p_5$+ $p_2p_3p_5$+ $p_1p_2p_5$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$ Find the one with minimal multiplications?

$(((A_1A_2)A_3)A_4)$

Matrix Chain Multiplication

- Given a chain <A_1, A_2, … A_n> of matrices, where matrix A_i has dimension p_{i-1}x p_i, find optimal fully parenthesize product $A_1A_2…A_n$ that minimizes number of scalar multiplications.
- Let m[i, j] be the minimal # of scalar multiplications needed to calculate $A_iA_{i+1}…A_j$ (m[1…n]) is what we want to calculate)
- Recurrence relation: how does m[i…j] relate to smaller problem
  - First decision: pick k (can be i, i+1, …j-1) where to divide $A_iA_{i+1}…A_j$ into two groups: $\underline{(A_i…A_k)}(A_{k+1}…A_j)$
  - $(A_i…A_k)$ dimension is p_{i-1} x p_k, $(A_{k+1}…A_j)$ dimension is p_k x p_j

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

Summary

- Keys to DP
  - Optimal Substructure
  - overlapping subproblems
- Define the subproblem: r(n), MSE(i), LCS(i,j) LCS of prefixes …
- Write recurrence relation for subproblem: i.e., how to calculate solution to a problem using sol. to smaller subproblems
- Implementation:
  - memoization (table+recursion)
  - bottom-up table based (smaller problems first)
- Insights and understanding comes from practice!