Dynamic Programming

- Algorithm design technique
  - Many apparently exponential optimization problems have polynomial solutions using DP
- Has been described as *controlled brute force*, or *divide-and-conquer with memory*
- Title refers not to computer programming but to the process of gradually (i.e., dynamically) filling a table in a systematic way.

On the Origins of Dynamic Programming

The 1950s were not good years for mathematical research. We had an interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'*. I'm not using the term lightly. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

Richard Bellman, on the origin of his term 'dynamic programming' (1984)

Example: Fibonacci by DAC

Problem: Given n, find the nth Fibonacci number

```
FIB($n$)
1  if $n < 2$
2      then return $n$
3      else  return FIB($n-1$) + FIB($n-2$)
```

Time? $T(n) = T(n-1) + T(n-2) + 1$

$\geq 2T(n-2) + 1 \geq 2^{n/2}$

Exercise. How many calls to $Fib(n-k)$ are generated by $Fib(n)$?

Exercise. Find an exact solution to the recurrence for $T(n)$

4

Fibonacci by Memo(r)ization

- Can speed up the algorithm by storing the results of our recursive calls and reusing them when needed again
- This technique is called ***memoization***

```
FIB($n$)
1  if $n < 2$
2      then $f \leftarrow n$
3      else  $f \leftarrow$ FIB($n-1$) + $memo[n-2]$
4  $memo[n] = f$
5  return $f$
```

Time? $T(n) = T(n-1) + 1 = O(n)$

5

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: HIEROGLYPHOLOGY and MICHELANGELO

- Applications
  - Text processing: edit distance, diff command
  - DNA comparison

6

A Brute Force Algorithm

- For every subsequence s of x determine if s is a subsequence of y. Keep 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?
- Running time?

  $O(n\ 2^m)$

7

Notation

- LCS(x,y) is really a set of strings, all of the same length

  Example: x = AGCGTAG, y = 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

8

A Simplification

- First, compute |LCS(x,y)| only. Then extend algorithm to find actual string.
- Solve smaller problem on *prefixes* of x and y.

  Let $c[i, j]$ =| LCS(x[1:i], y[1:j]) |

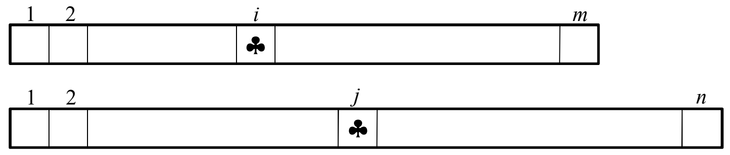
  Then |LCS(x,y) | = $c[m, n]$

Theorem.

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

9

Proof ($x[i] = y[j]$)

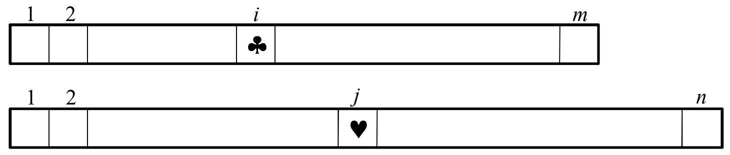


- Let $c[i,j] = k$ and $z[1{:}k] = \text{LCS}(x[1{:}i], y[1{:}j])$
- What is $z[k]$?

  $z[k] = x[i] = y[j]$, 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, $z[1{:}k-1] = \text{LCS}(x[1{:}i-1], y[1{:}j-1])$ (cut-and-paste argument)
- $c[i-1,j-1] = k-1$, which $\Rightarrow c[i,j] = c[i-1,j-1]+1$

Proof ($x[i] \neq y[j]$)



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

  if $z[k] \neq x[i]$

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

  if $z[k] \neq y[j]$

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

  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 = LCS(x, y), then *any* prefix of z is a LCS of *some* prefix of x and *some* prefix of y

This suggests an obvious strategy: DAC

12

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 $c[i, j]$.
```

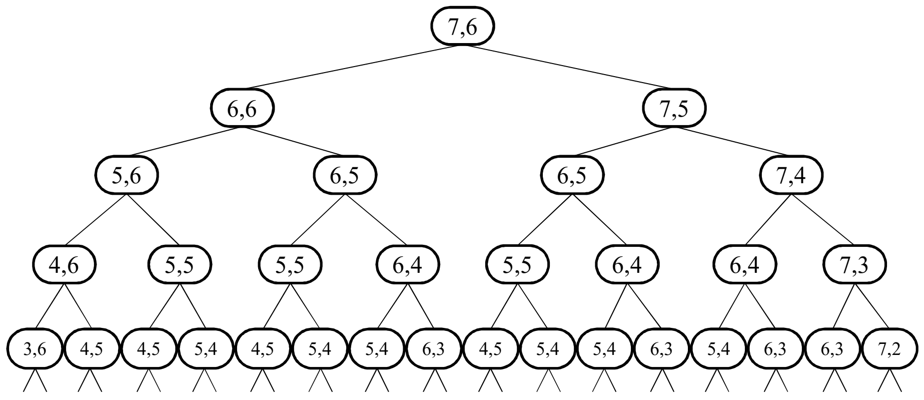
- Worst case time?

  Always take the **else** clause (line 5)

  Happens when $x[i] \neq y[j]$, for all i, j

13

Recursion Tree ($m = 7$, $n = 6$)



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

$\Rightarrow$ exponential time

14

DP Hallmark 2

Overlapping subproblems. A recursive solution contains a *small number* of distinct problem instances repeated *many* times.

How many distinct subproblems does LCS have?

There are *mn* subproblems but exponential number of generated instances!

This suggests storing solutions to subproblems, in case they are needed later.

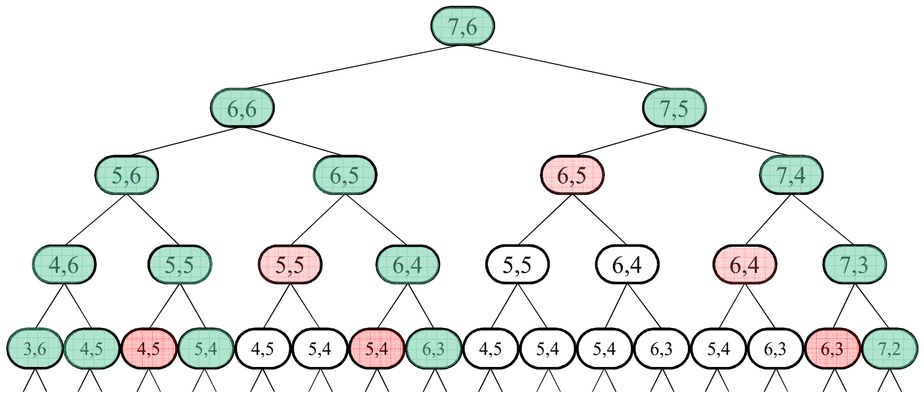
15

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? $\Theta(mn)$

Recursion Tree ($m = 7$, $n = 6$)



- How many green nodes are there?
- How many red nodes?
- What do they mean?

Dynamic Programming

- Concentrates on the bookkeeping portion of memoized DAC.
- Fills memo table "bottom up", from previously solved to new unsolved subproblems.

Key idea: There is a collection of subproblems that *can be ordered* in a list, and a recursive formula that expresses the solution to any subproblem in terms of the solution to subproblems that appear earlier in the list.

18

Steps of a Dynamic Programming Solution

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

19

DP Algorithm for LCS Length

```
LCS-LENGTH($x, y, m, n$)
for $i \leftarrow 1$ to $m$
        do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $m$
        do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$
        do for $j \leftarrow 1$ to $n$
                do if $x[i] == y[j]$
                        then $c[i, j] \leftarrow c[i-1, j-1] + 1$
                        else  if $c[i-1, j] \geq c[i, j-1]$
                                then $c[i, j] \leftarrow c[i-1, j]$
                                else  $c[i, j] \leftarrow c[i, j-1]$
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	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 = Θ(*mn*)
- Space = Θ(*mn*) but can reduced to Θ(min{*m*,*n*})

Steps of a Dynamic Programming Solution

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

37

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

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

38

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

39

Matrix Chain 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?

40

Standard Matrix Multiplication

```
MATRIX-MULTIPLY($A, B$)
if $columns[A] \neq rows[B]$
    then error
for $i \leftarrow 1$ to $rows[A]$
    do for $j \leftarrow 1$ to $columns[B]$
        do $C[i, j] \leftarrow 0$
            for $k \leftarrow 1$ to $columns[A]$
                do $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
```

- If A is $p \times q$ and B is $q \times r$ computing C requires pqr multiplications ($\Theta(n^3)$ for square $n \times n$ matrices)

Does it matter?

- <u>Example</u>: compute ABC where
  A is 60×5, B is 5×100, C is 100×10

  1. Computing $(AB)C$ requires 90,000 mults
  2. Computing $A(BC)$ requires 8,000 mults

- <u>Goal</u>: For each subchain $M_{ij} = M_i \times \ldots \times M_j$ find optimal parentheses placement

$$M_{ij} = (M_i \times \cdots \times M_k)(M_{k+1} \times \cdots \times M_j)$$

Solving Matrix Chain by 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 optimal solution

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

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

43

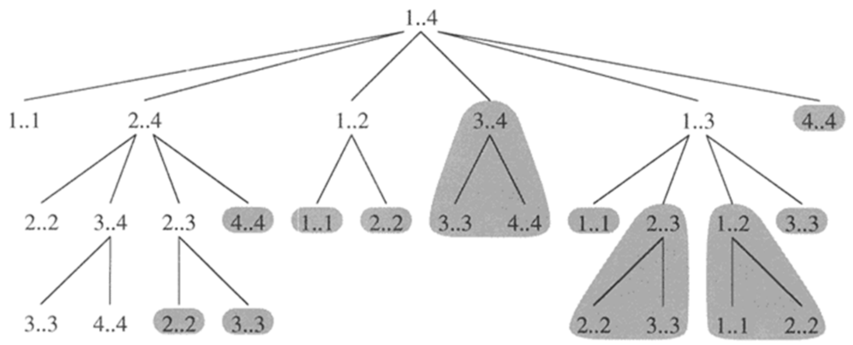
Divide-and-Conquer

```
DAC-MATRIX-CHAIN($p, i, j$)
1  if $i == j$
2      then return 0
3  $m[i,j] \leftarrow \infty$
4  for $k \leftarrow i$ to $j-1$
5      do $q \leftarrow$ DAC-MATRIX-CHAIN($p, i, k$)
              +DAC-MATRIX-CHAIN($p, k+1, j$) $+ p_{i-1}p_k p_j$
6          if $q < m[i,j]$
7              then $m[i,j] \leftarrow q$
8  return $m[i,j]$
```

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

44

Recursion Tree for $n = 4$



- Each node contains the parameters *i* and *j*
- Shaded subtrees represent redundant computations that could be replaced by a table lookup

45

```
LookUpChain($p$, $i$, $j$)
1.  if $m[i, j] < \infty$ then return $m[i, j]$
2.  if $i = j$ then $m[i, j] = 0$
3.  else for $k \leftarrow i$ to $j-1$ do
4.      $q \leftarrow$ LookUpChain($p$, $i$, $k$) +
                LookUpChain($p$, $k$+1, $j$) + $p_{i-1}p_kp_j$
5.      if $q < m[i, j]$ then $m[i, j] \leftarrow q$
6.  return $m[i, j]$
7. end
```

```
MEMOIZED-MATRIX-CHAIN($p$)
1 $n \leftarrow length[p] - 1$
2 for $i \leftarrow 1$ to $n$
3     do for $j \leftarrow i$ to $n$
4            do $m[i, j] \leftarrow \infty$
5 return LOOKUP-CHAIN($p$, 1, $n$)
```

46

Solving Matrix Chain by DP

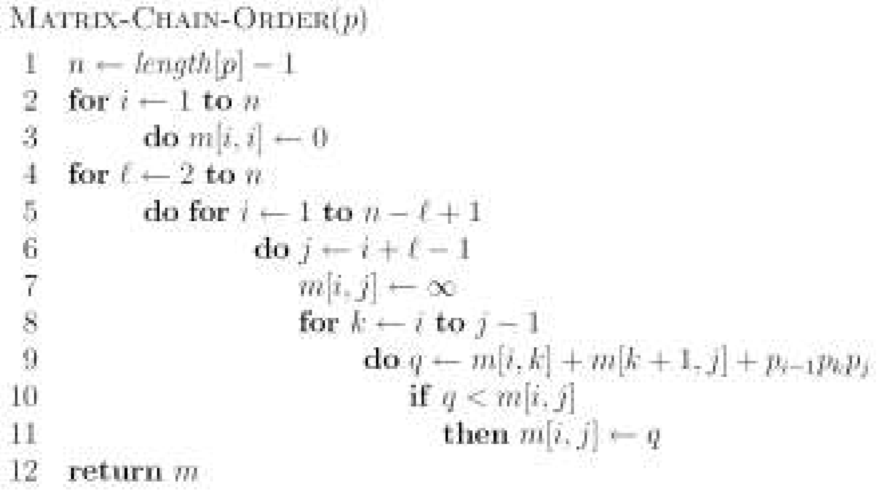
1. Characterize the structure of an optimal solution
   $M_{ij} = (M_i \times \ldots \times M_k)\ (M_{k+1} \times \ldots \times M_j)$

2. Recursively define the value of optimal solution
   $m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

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

47

3. Compute Solution Bottom Up



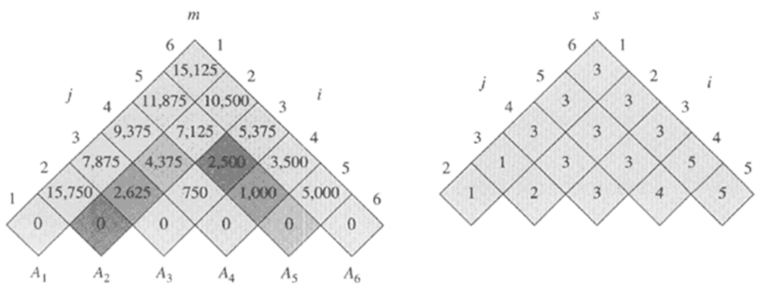
```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
        do m[i, i] ← 0
for ℓ ← 2 to n
        do for i ← 1 to n − ℓ + 1
                do j ← i + ℓ − 1
                    m[i, j] ← ∞
                    for k ← i to j − 1
                        do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                            if q < m[i, j]
                               then m[i, j] ← q
return m
```

- Time? $\Theta(n^3)$

48

4. Construct Solution

```
MATRIX-CHAIN-ORDER($p$)
$n \leftarrow length[p] - 1$
for $i \leftarrow 1$ to $n$
        do $m[i,i] \leftarrow 0$
for $\ell \leftarrow 2$ to $n$
        do for $i \leftarrow 1$ to $n - \ell + 1$
                do $j \leftarrow i + \ell - 1$
                    $m[i,j] \leftarrow \infty$
                    for $k \leftarrow i$ to $j - 1$
                        do $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
                            if $q < m[i,j]$
                                then $m[i,j] \leftarrow q$
                                     $\boxed{s[i,j] \leftarrow k}$
return $m$ and $s$
```



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

$$m[2,5] = \min\begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 = 13{,}000 \\ m[2,3] + m[4,5] + p_1p_3p_5 = 7{,}125 \\ m[2,4] + m[5,5] + p_1p_4p_5 = 11{,}375 \end{cases}$$

Printing Expression with Parentheses

```
PRINT-OPTIMAL-PARENS($s, i, j$)
1  if $i = j$
2      then print "$A$"$_i$
3      else print "("
4           PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
5           PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
6           print ")"
```

51

Multiplying the Chain

```
CHAIN-MULTIPLY($A, i, j$)
1  if $i == j$
2      then return $A_i$
3  if $j == i + 1$
4      then return $A_1 \times A_2$
5  $k \leftarrow s[i, j]$
6  $A \leftarrow$ CHAIN-MULTIPLY($i, k$)
7  $B \leftarrow$ CHAIN-MULTIPLY($k + 1, j$)
8  return $A \times B$
```

52