Dynamic Programming

Because sometimes greed fails..

Slides adapted from Ran Libeskind-Hadas and David Kauchak

Greedy Algorithms

- The idea:
  - When we have a choice to make, make the one that looks the best *right now!*
  - Make a *locally optimal choice* in hopes of a *globally optimal solution*
  - Don't always generate optimal solutions, but can.
    - General characteristics of when greed is good (optimal)

Greedy Strategy

- How do we find greedy strategies that work?

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always *safe*.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

When is greed good?

- No general way to tell whether a problem can be solved optimally using a greedy algorithm
- Two key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem

Making….

Making Change!

```
change(42, [1, 5, 10, 25, 50])
change(42, [1, 5, 21, 28])  # greed?
change(amount, coins[1:i])
  if amount = 0: return 0
  elif i == 0: return Infinity
  elif coinValue[i] > amount:
     return change(amount, coins[1:i-1])
  else:
     useIt = 1 + change(amount-coinValue[i],
                        coins[1:i])
     loseIt = change(amount, coins[1:i-1])
     return min(useIt, loseIt)
```

Back to Greed!

Denominations $1, b, b^2, \ldots, b^k$ (b integer ≥ 2)

Observation 1: It is never optimal to use b or more coins of denomination 1 or b or … b^{k-1}.

Observation 2: with at most $(b-1)$ of each of coins $1, b, \ldots, b^i$ we can make at most…

(At home) Worksheet!

change(amount) # greedy version for coins $1, b, b^2 \ldots, b^k$

- **Choose the largest coin b^i that doesn't exceed amount**
- **Recurse on change(amount-b^i)**

Proving Correctness

Proof by strong induction on amount, n
(not on our coin set $1, b, b^{2,} \ldots, b^k$)

Basis: n = 0

Induction hypothesis: Assume that the greedy algorithm uses the optimal number of coins for any amount from 0 to n.

Induction step: Consider an amount n+1. The greedy algorithm uses the largest coin b^i (i between 0 and k) that doesn't exceed n+1. We **first** claim that this choice is safe in the sense that there exists an optimal solution that uses a b^i coin.

Consider an optimal solution S (a multiset of coins) for amount n+1.

If S contains b^i then our claim is true. If not, then S must make up at least b^i from smaller coins 1, b, …, b^{i-1}.

But, by Observation 1, since S is optimal, it uses no more than b-1 of each of these coins.

By Observation 2, we can't make up b^i using at most b-1 of each of these coins. So, S *must* contain coin b^i.

Now, the remaining amount $(n+1)-b^i$ must be made up using the least number of coins. But, our algorithm recurses on this amount and, by the induction hypothesis, it uses the least number of coins for that amount since $(n+1)-b^i$ is between 0 and n. Q.E.D.

Greedy Recap

- The idea:
  - When we have a choice to make, make the one that looks the best *right now!*
  - Make a *locally optimal choice* in hopes of a *globally optimal solution*
- Key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value
  - Minimization and maximization

Dynamic programming

One of the most important algorithm tools!

Very common interview question

Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems

AND

the sub-problems are overlapping

Where did "dynamic programming" come from?

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very 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; I'm using it precisely. 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? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

Richard Bellman On the Birth of Dynamic Programming

Stuart Dreyfus

http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

Fibonacci numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, …

What is the recurrence for the n^{th} Fibonacci number?

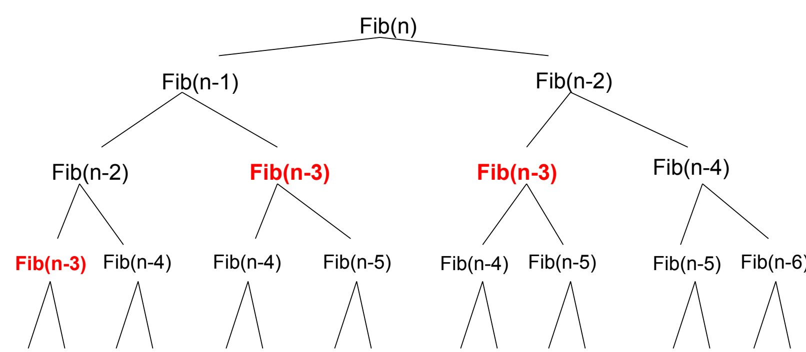
F(n) = F(n-1) + F(n-2)

The solution for n is defined with respect to the solution to smaller problems (n-1 and n-2)

Fibonacci: a first attempt

```
FIBONACCI(n)
  if n = 1 or n = 2
          return 1
  else
          return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

A lot of repeated work!



Identifying a dynamic programming problem

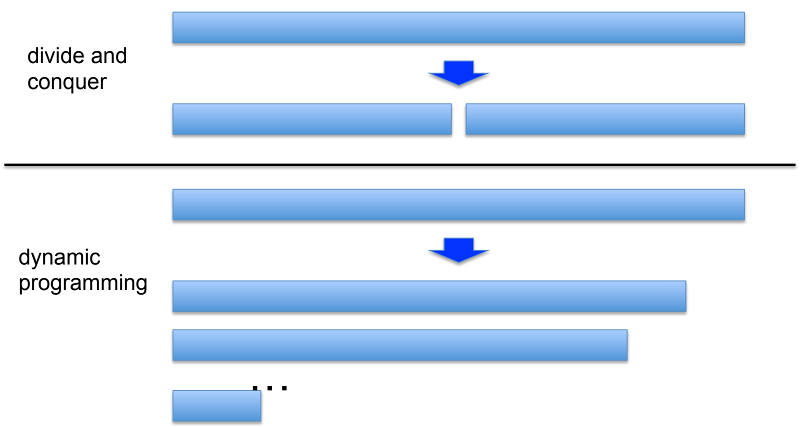
The solution can be defined with respect to solutions to subproblems

The subproblems created are *overlapping*, that is **we see the same subproblems repeated**

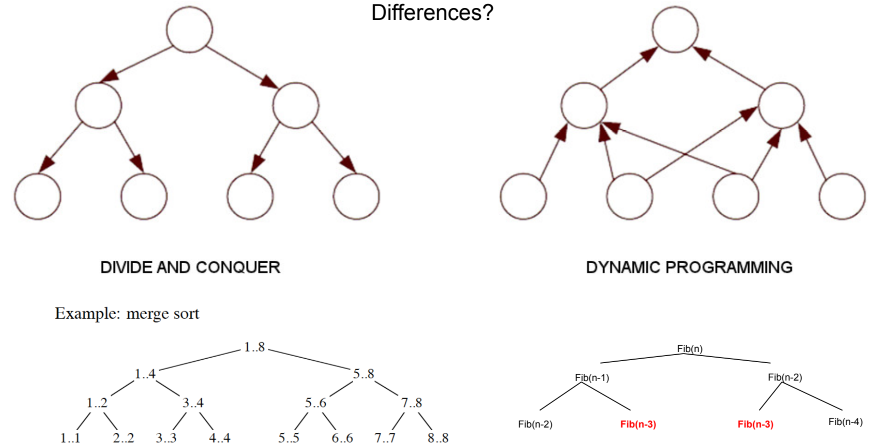
The Goal

- Solve each subproblem once
- Save solution in a table and refer back any time we revisit the subproblem
- "Store, don't recompute" → Time-memory trade-off
- Two basic approaches: top-down with memoization and bottom up

Overlapping sub-problems



Differences?



Creating 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, typically in bottom-up fashion
4. Construct an optimal solution from computed information

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

Creating a dynamic programming solution

Step 1: Identify a solution to the problem with respect to **smaller** subproblems (pretend like you have a solver, but it only works on smaller problems):

- F(n) = F(n-1) + F(n-2)

Step 2: **bottom up** - start with solutions to the smallest problems and build solutions to the larger problems

use an array to store solutions to subproblems


```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```


Important Questions to ask about the DP Table:

- **Meaning?**
  - What do the cells mean?
- **Want?**
  - What cell do you want?
- **Easy?**
  - What cells can you fill out (easily)?
- **Rule?**
  - What rule helps fill out other cells?

The DP table should include the possible inputs to the recursive call


```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```



```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```


Is it correct?

Running time?

Longest common subsequence (LCS)

For a sequence X = $x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices ($i_1, i_2, \ldots, i_k$) where
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

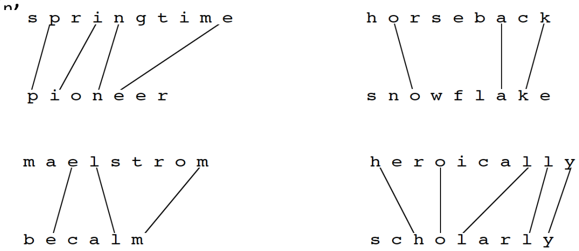
AADAA

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences X = $x_1, x_2, \ldots, x_n$ and
Y = $y_1, y_2, \ldots, y_n$,

Examples:



LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences X = $x_1, x_2, \ldots, x_n$ and
Y = $y_1, y_2, \ldots, y_n$,

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences X = $x_1, x_2, \ldots, x_n$ and
Y = $y_1, y_2, \ldots, y_n$,

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

Assume you have a solver for smaller problems

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Whiteboard: Specify the solution to this problem as a combination of subproblems

Hint: Is the last character part of the LCS?

Step 1: Define the problem with respect to subproblems

X = A B C B D A A

LCS

Y = B D C A B A

The characters are part of the LCS

What is the recursive relationship?

If they're the same

$$LCS(X,Y) = LCS(X_{1...n-1}, Y_{1...m-1}) + x_n$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

LCS

Y = B D C A B A

If they're different

$$LCS(X,Y) = LCS(X_{1...n-1}, Y)$$

$$LCS(X,Y) = LCS(X, Y_{1...m-1})$$

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & if\ x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

(for now, let's just worry about counting the length of the LCS)

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & if\ x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1...j}, Y_{1...k})$

two different indices

Step 2: Build the solution from the bottom up

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & if\ x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1...j}, Y_{1...k})$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & if\ x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & if\ x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

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

Worksheet:

Meaning?
- What do the cells mean?

Want?
- What cell do you want?

Easy?
- What cells can you fill out (easily)?

Rule?
- What rule helps fill out other cells? (whiteboard)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	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	B	0						
5	D	0						
6	A	0						
7	B	0						

Need to initialize values within 1 smaller in either dimension.

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		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	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		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

Where's the final answer?

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

Running time?

Keeping track of the solution

Our LCS algorithm only calculated the length of the LCS between X and Y

What if we wanted to know the actual sequence?

Keep track of this as well…

```
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]        ↖
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]                ↑
                else
                        c[i, j] ← c[i, j − 1]                ←
return c[m, n]
```

$$LCS[i,j] = \begin{cases} 1 + LCS[i,j] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1] & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	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

We can follow the arrows to generate the solution

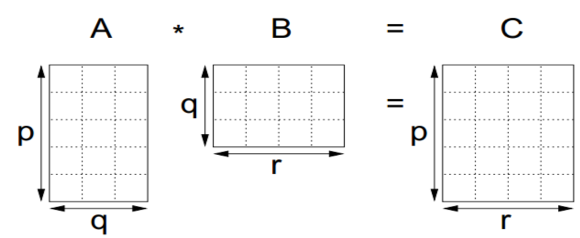
BCBA

Elements of a DP (revisited)

- Optimal substructure
  - A solution to a problem consists of making a choice/computation that will lead to an optimal solution
  - Given this choice/computation, determine which subproblems arise and how to characterize the resulting space of subproblems.
  - Solutions to the sub-problems used within the optimal solution must themselves be optimal. Otherwise, we'd see "cut-and-paste" error:
    - Suppose that one of the subproblem solutions is not optimal
    - Cut it out
    - Paste in an optimal solution
    - Get a better solution to the original problem. Contradicts the optimality of problem solutions
- Overlapping subproblems

How does this differ from greedy?

Matrix Multiplication



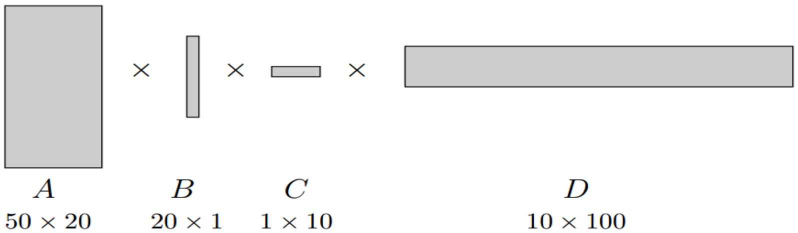
Multiplication time = pqr

Fig. 7: Matrix Multiplication.

http://www.cs.sunysb.edu/~jgao/CSE548-fall07/David-mount-DP.pdf

Matrix Multiplication

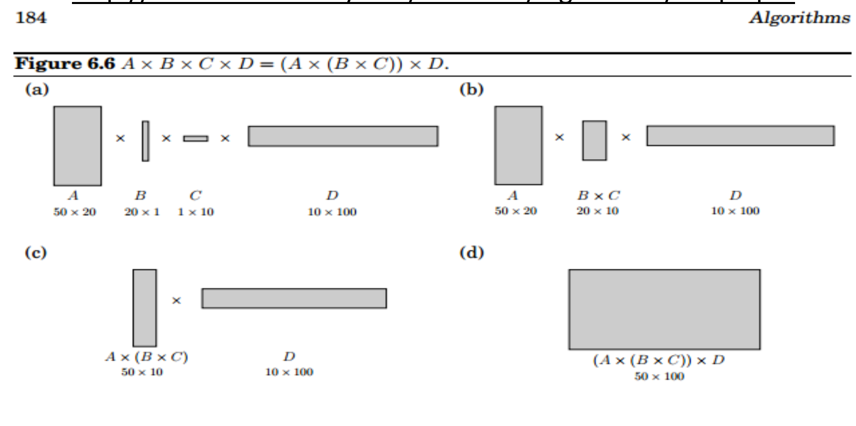
http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf



How many possible orderings of multiplication?

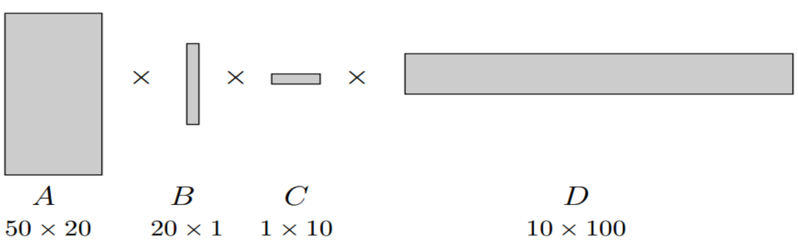
Matrix Multiplication

http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf



Order Matters

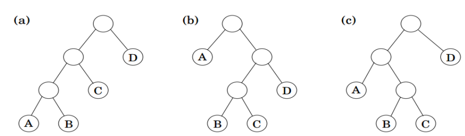
http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

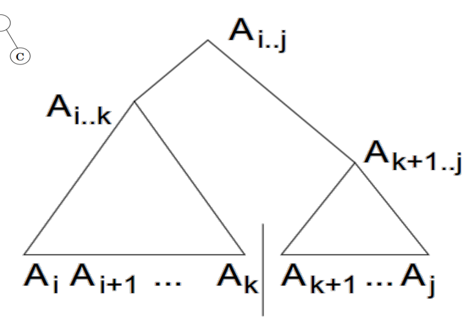


Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120,200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60,200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

Which parentheses match to each diagram?

$((A \times B) \times C) \times D$ $\quad (A \times (B \times C)) \times D$ $\quad A \times ((B \times C) \times D)$





http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf
http://www.cs.sunysb.edu/~jgao/CSE548-fall07/David-mount-DP.pdf

Matrix Multiplication Revisited

what's with all this revisiting business?!

Ran++ Conventions

M_1 M_2 M_3

10 × 100   100 × 5   5 × 50

```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

In general…

M_1 M_2 M_n

$p_0 \times p_1$ $p_1 \times p_2$ $p_{n-1} \times p_n$

$[M_i, \ldots, M_j]$

minMults (matrices [i:j])
returns the min # of total
mults required

Matrix Multiplication Revisited

what's with all this revisiting business?!

M_1 M_2 M_3

10 × 100   100 × 5   5 × 50

```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

```
minMults(matrices[i:j])
  if i
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?

  return best
```

Matrix Multiplication Revisited

what's with all this revisiting business?!

M_1 M_2 M_3

10 × 100   100 × 5   5 × 50

```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?

  return best
```

Matrix Multiplication Revisited

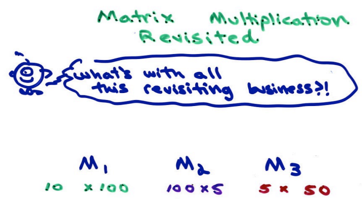
what's with all this revisiting business?!

M_1 M_2 M_3

10 × 100   100 × 5   5 × 50

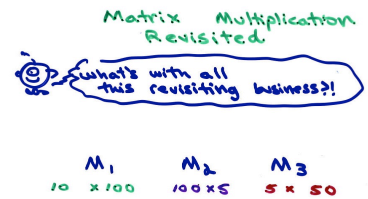
```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]

  return best
```




```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]

  return best
```




```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

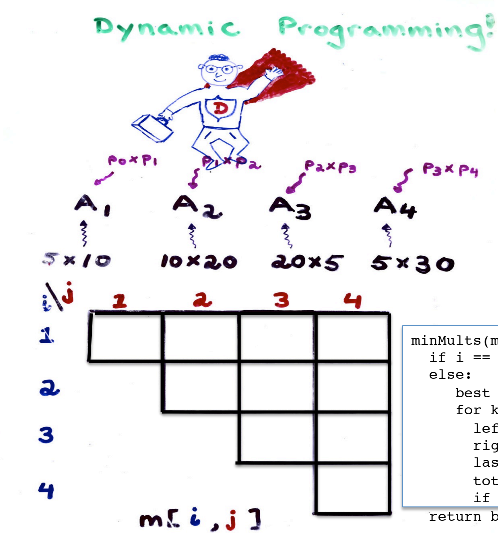

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```

Meaning?
- What do the cells mean?

Want?
- What cell do you want?

Easy?
- What cells can you fill out (easily)?

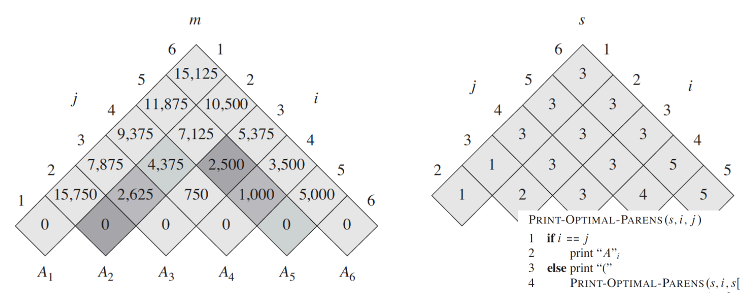
Rule?
- What rule helps fill



Fill this in in your notes!

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```

Another example: (CRLS)




```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

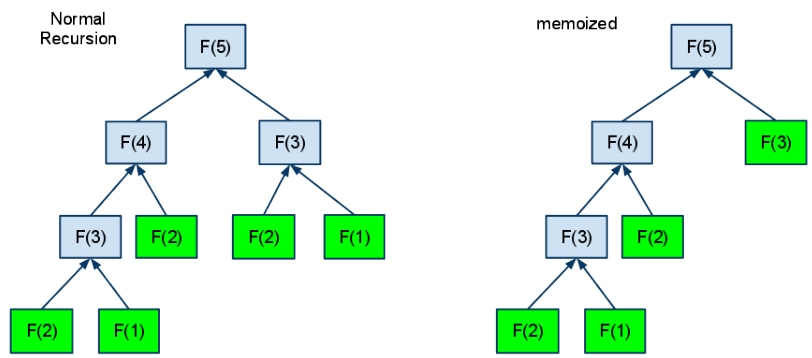
Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

Top-down Alternative: Memoization

- Memoizing is remembering what we have computed previously.
- Solve recursively (top-down)
  - "Store, don't recompute"
  - Make a table indexed by subproblem
  - When solving subproblem (top-down):
    - Look-up in table
    - If answer is there, use it
    - Else, compute answer and store it.
- Bottom-up DP goes a step further: first determine the order in which the table would be accessed, and fill it in that way

The top-down approach



http://www.agillo.net/getting-groovy-with-fibonacci/

Quick summary

- Step 1: Define the problem with respect to subproblems
  - We did this for divide and conquer too. What's the difference?
  - You can identify a candidate for dynamic programming if there is **overlap** or **repeated work** in the subproblems being created
- Step 2: build the solution from the bottom up
  - Build the solution such that the subproblems referenced by larger problems are already solved
  - Memoization is also an alternative