Graduate Algorithms

CS673-2016F-09

Dynamic Programming

David Galles

Department of Computer Science
University of San Francisco

09-0: Recursive Solutions

- Divide a problem into smaller subproblems
- Recursively solve subproblems
- Combine solutions of subproblems to get solution to original problem

Some Examples:

- Mergesort
- Quicksort
- Selection

09-1: Recursive Solutions

- Occasionally, straightforward recursive solution takes too much time
  - Solving the same subproblems over and over again
- Canonical example: Fibonacci Numbers

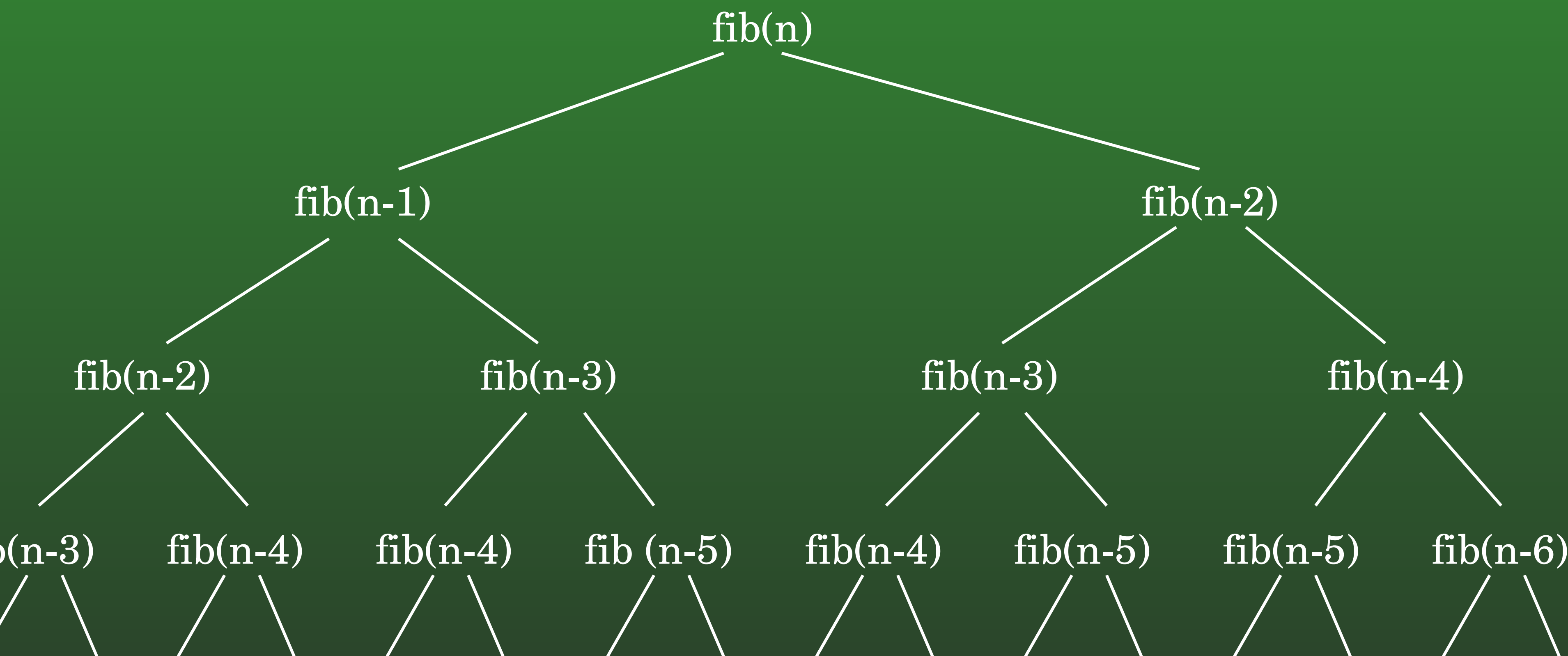
$$
\begin{aligned}
F(0) &= 1 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2)
\end{aligned}
$$

09-2: Fibonacci

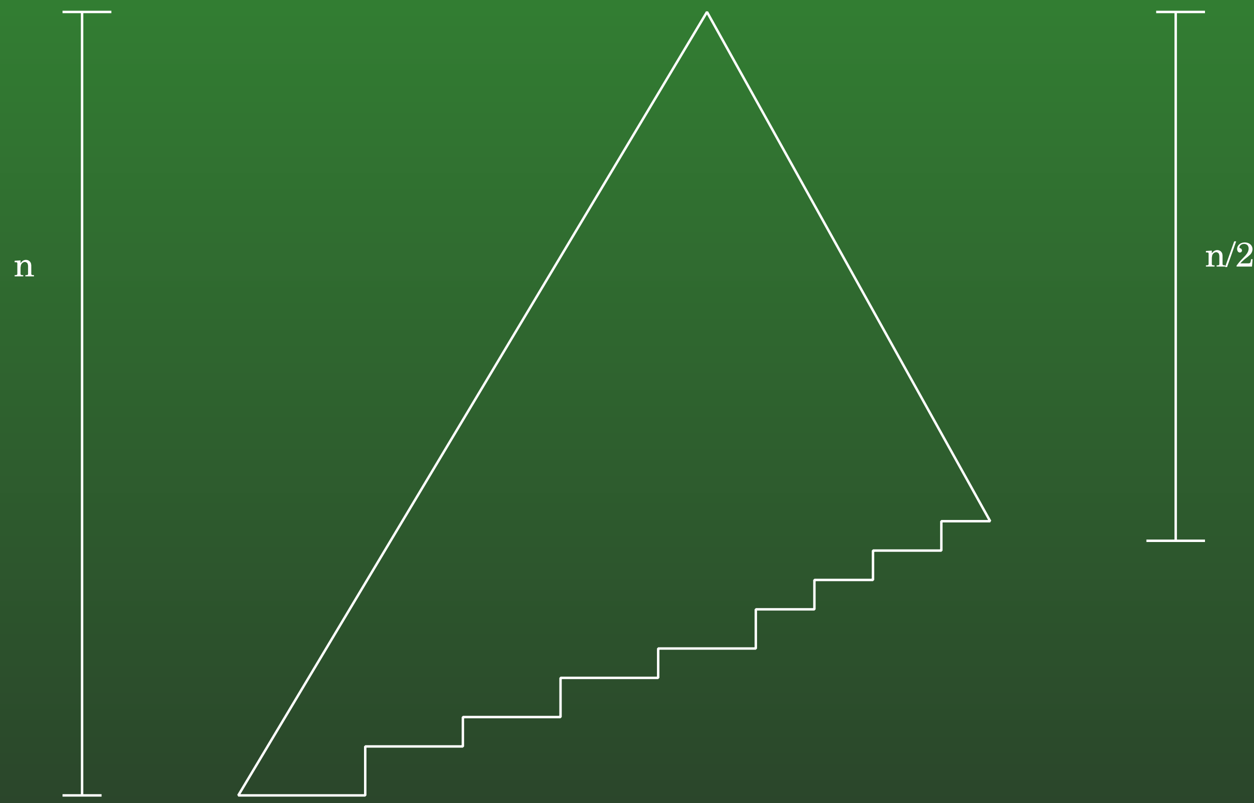
```
fib(n)
   if (n < 2)
       return 1
   return fib(n-1) + fib(n-2)
```

- How much time does this take?
  - Tight bound is a little ugly, but can we get loose upper/lower bounds?
  - Polynomial, quadratic, exponential?

09-3: Fibonacci

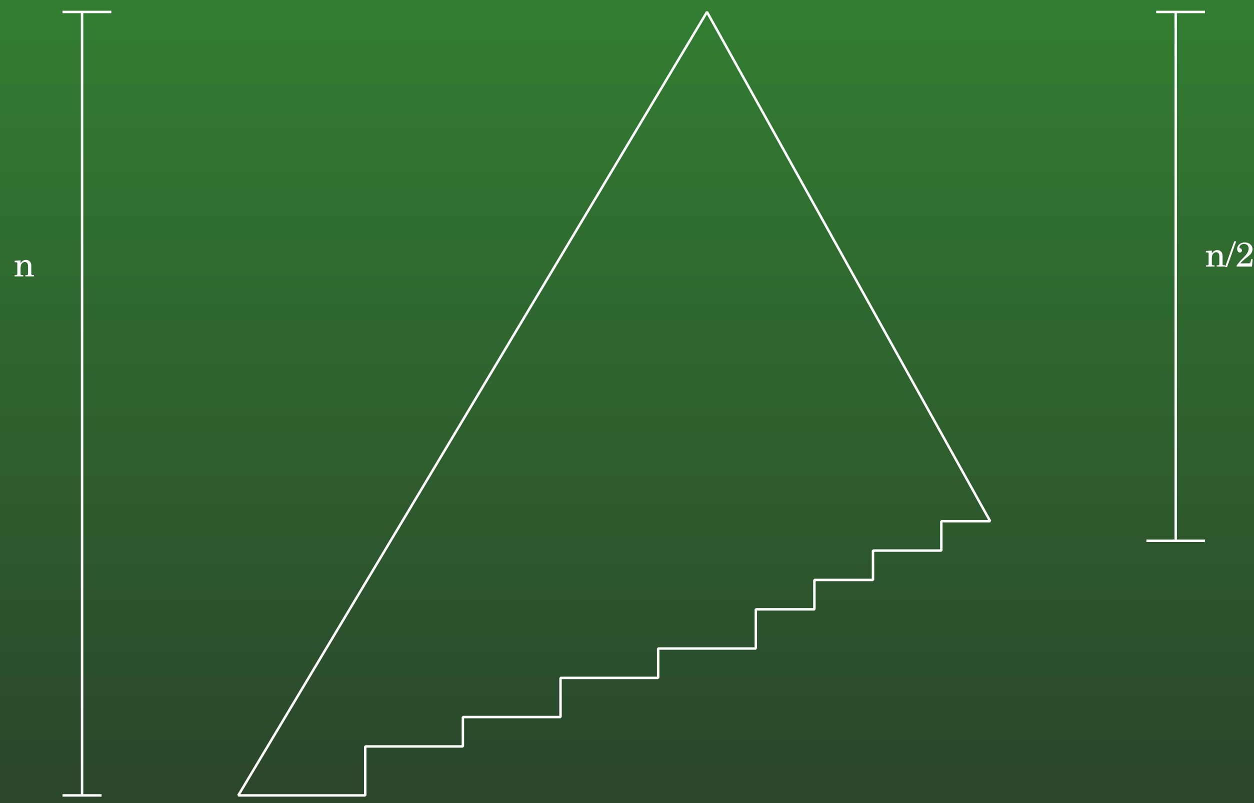


09-4: Fibonacci



- How many leaves are in this tree?

09-5: Fibonacci



- How many leaves are in this tree?
    - $2^{\frac{n}{2}} <$ # of leaves $< 2^n$

09-6: Fibonacci

- Straightforward implementation requires exponential time
- How many *different* subproblems are there, for finding `fib(n)` ?

09-7: Fibonacci

- Straightforward implementation requires exponential time
- How many *different* subproblems are there, for finding `fib(n)` ?
  - (n - 1)
- Takes so much time because we are recalculating solutions subproblems over and over and over again.
- What if we stored solutions to subproblems in a table, and only recalculated if the values were not in the table?

09-8: Fibonacci

								. . .
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	

09-9: Fibonacci

1	1							. . .
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	

- First, fill in the “base cases” of the recursion

09-10: Fibonacci

1	1	2	3	...				...
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	

- First, fill in the “base cases” of the recursion
- Next, fill in the rest of the table
  - Pick the order to fill the table entries in carefully
  - Make sure that answers to subproblems that you need are already in the table

09-11: Fibonacci

```
Fibonacci(n)
    T[0] = 1
    T[1] = 1
    for i = 2 to n do
        T[i] = T[i-1] + T[i-2]
    return T[n]
```

09-12: Fibonacci

- Time required to calculate fib(n) using this method?
- Space required to calculate fib(n) using this method?

09-13: Fibonacci

- Time required to calculate fib(n) using this method?
  - $\Theta(n)$
- Space required to calculate fib(n) using this method?
  - $\Theta(n)$
- Can we do better?

09-14: Fibonacci

- Time required to calculate fib(n) using this method?
  - $\Theta(n)$
- Space required to calculate fib(n) using this method?
  - $\Theta(n)$
- Can we do better?
  - Only need to store last two numbers, not the entire table

09-15: Fibonacci

```
fib(n)
   prev = 1
   prev_prev = 1
   for i = 2 to n do
       prev = prev + prev_prev
       prev_prev = prev - prev_prev
   return prev
```


Dynamic Programming

- Simple, recursive solution to a problem
- Straightforward implementation of recursion leads to exponential behavior, because of repeated subproblems
- Create a table of solutions to subproblems
- Fill in the table, in an order that guarantees that each time you need to fill in a square, the values of the required subproblems have already been filled in

09-17: Winning a Series

- Two teams, A and B
  - A has a probability of p of winning any one game
  - B has a probability $q = 1 - p$ of winning any game
- A need to win i more games to win the series
- B needs to win j more games to win the series

What is the probability that A will win the series?

09-18: Winning a Series

- $P(i, j)$ = Probability that team A will win the series, given that A needs i more wins, and j needs j more wins
  - $P(0, k) = 1$, for all $k > 0$
  - $P(k, 0) = 0$ for all $k > 0$
  - $P(i, j) =$?

09-19: Winning a Series

- $P(i, j)$ = Probability that team A will win the series, given that A needs i more wins, and j needs j more wins
  - $P(0, k) = 1$, for all $k > 0$
  - $P(k, 0) = 0$ for all $k > 0$
  - $P(i, j) = p * P(i-1, j) + q * P(i, j-1)$

09-20: Winning a Series

```
AWinProbability(i,j)
   if (i=0)
      return 1
   if (j=0)
      return 0
   return p*AWinProbability(i-1,j) +
          q*AWinProbability(i,j-1)
```

- Running time is exponential (why?)
- How many *unique* subproblems are there?

Winning a Series

$$\begin{array}{ccccccc}
 & P(0,1) & P(0,2) & P(0,3) & P(0,4) & \ldots & P(0,n) \\
P(1,0) & P(1,1) & P(1,2) & P(1,3) & P(1,4) & \ldots & P(1,n) \\
P(2,0) & P(2,1) & P(2,2) & P(2,3) & P(2,4) & \ldots & P(2,n) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
P(n,0) & P(n,1) & P(n,2) & P(n,3) & P(n,4) & \ldots & P(n,n)
\end{array}$$

- Total subproblems = n^2, assuming we start with each team needing to win n games

09-22: Winning a Series

- Start with a table with the "base cases" filled in:

	0	1	2	3	4	5	6	7
0		1	1	1	1	1	1	1
1	0							
2	0							
3	0							
4	0							
5	0							
6	0							
7	0							

- In which order should we fill the rest of the table elements?

09-23: Winning a Series

- Start with a table with the “base cases” filled in:

	0	1	2	3	4	5	6	7
0		1	1	1	1	1	1	1
1	0							
2	0							
3	0							
4	0							
5	0							
6	0							
7	0							

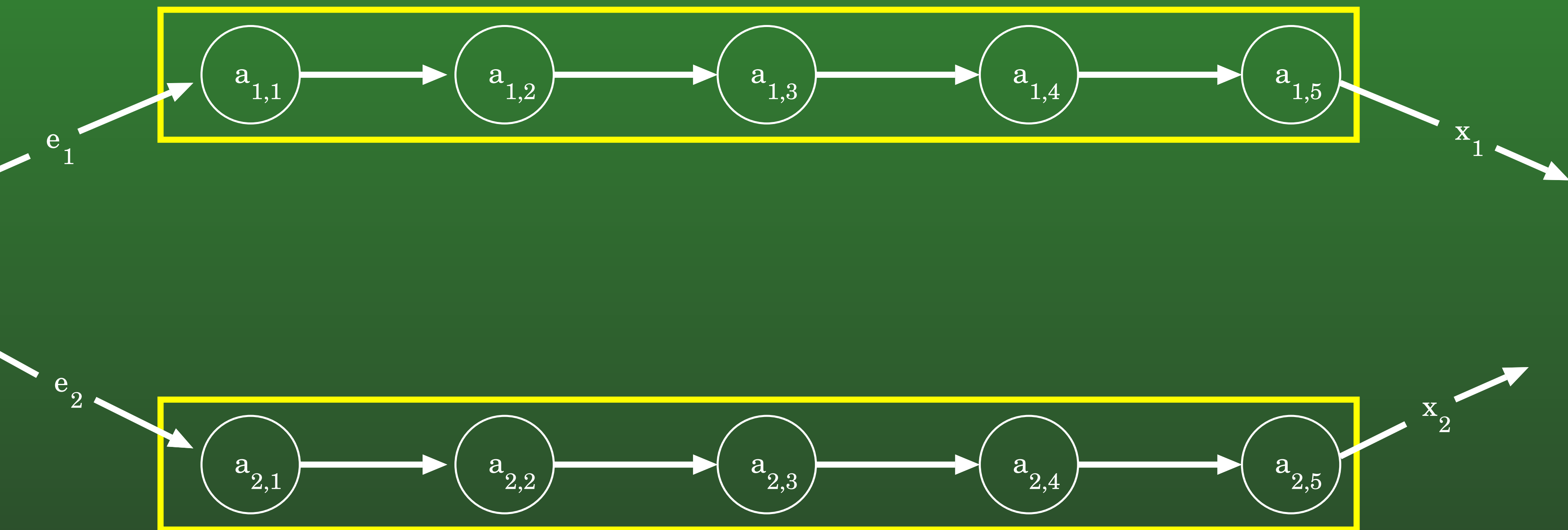
- Fill in the rest of the table elements in this order

09-24: Winning a Series

```
for i=0 to n do
   T[0,i] = 1
   T[i,0] = 0
for i = 1 to n do
   for j = 1 to n do
      T[i,j] = p*T[i-1,j] + q*T[i,j-1]
```

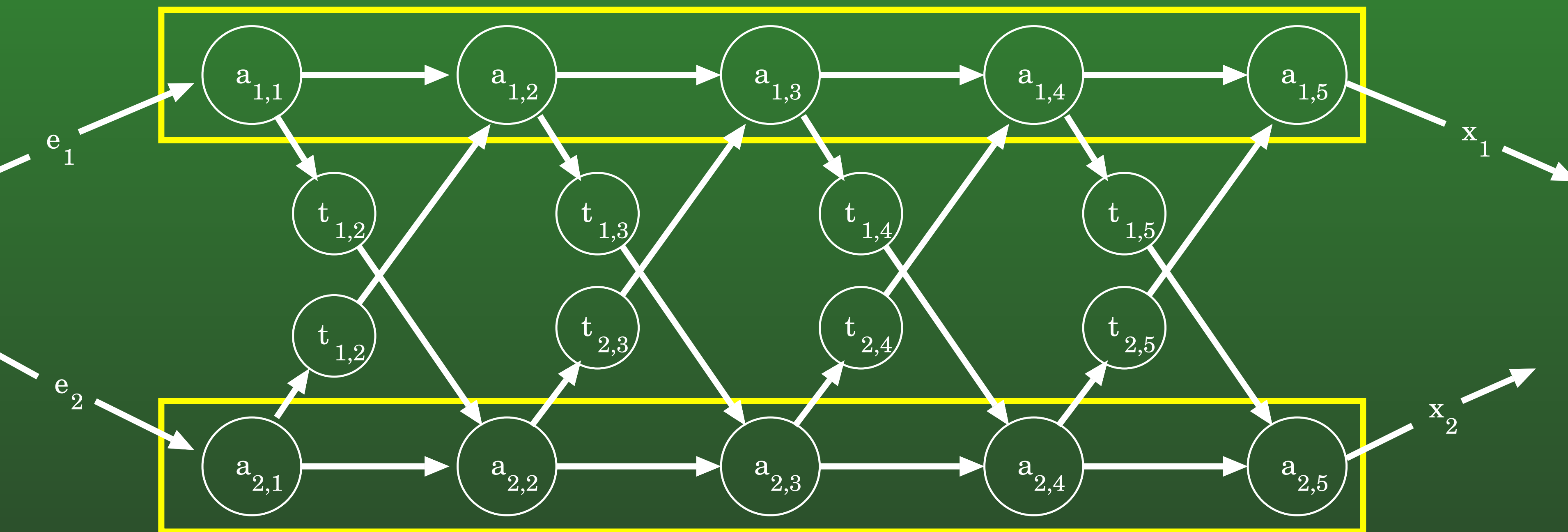
- We can read the final answer off the table

09-25: Assembly Line Scheduling



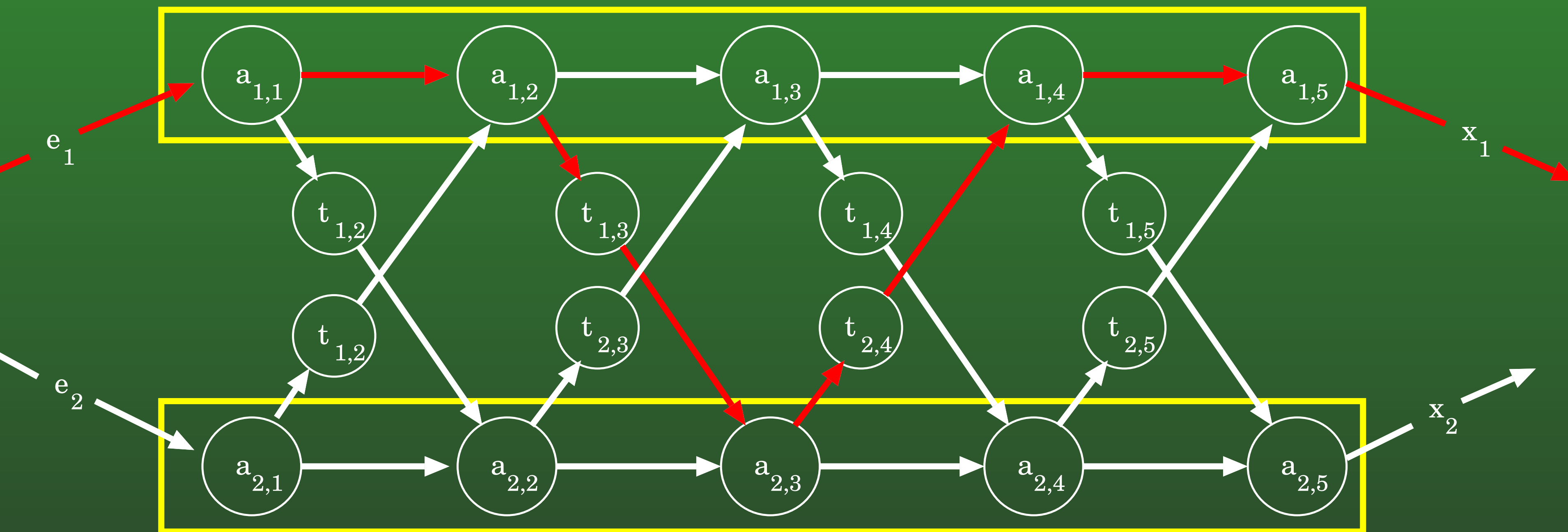
- Two different assembly lines
- Time to build a car on line 1: $e_1 + \sum_{i=1}^{n} a_{1,i} + x_1$
- Time to build a car on line 2: $e_2 + \sum_{i=1}^{n} a_{2,i} + x_2$

09-26: Assembly Line Scheduling



- Rush orders can use both lines
- Time penalty for switching

Assembly Line Scheduling



- Rush orders can use both lines
- Time penalty for switching
- $e_1 + a_{1,1} + a_{1,2} + t_{1,3} + a_{2,3} + t_{2,4} + a_{a,4} + a_{1,5} + x_1$

09-28: Assembly Line Scheduling

- How fast can we build a single car?
  - Try all possible paths through the assembly line
  - Exponential in the number of stations
  - Many repeated subproblems

09-29: Assembly Line Scheduling

- First step:
  - Find a “brute force” recursive solution, which examines all possible paths.
  - Give a simple recursive function that calculates the fastest path through the assembly line
    - What is the base case?
    - How can we make the problem smaller?
    - HINT: You can use two mutually recursive functions if you want to
- Second step
  - Use a table to avoid recalculating values

09-30: Assembly Line Scheduling

Two assembly lines, n stations in each line

- $f_1(k) =$ fastest time to complete the car, if we start at station k of assembly line 1.
- $f_2(k) =$ fastest time to complete the car, if we start at station k of assembly line 2.
- Base cases:

$NOTE:$ The text defines $f_i(k)$ = fastest time to get to station k on line i. The switch in the lecture notes is intentional, to show two ways of solving the problem.

Assembly Line Scheduling

Two assembly lines, n stations in each line

- $f_1(k) =$ fastest time to complete the car, if we start at station k of assembly line 1.
- $f_2(k) =$ fastest time to complete the car, if we start at station k of assembly line 2.
- Base cases:
  - $f_1(n) = a_{1,n} + x_1$
  - $f_2(n) = a_{2,n} + x_2$

09-32: Assembly Line Scheduling

Two assembly lines, n stations in each line

- $f_1(k) =$ fastest time to complete the car, if we start at station k of assembly line 1.
- $f_2(k) =$ fastest time to complete the car, if we start at station k of assembly line 2.
- Recursive cases:
  - $f_1(k) = a_{1,k} + \min(f_1(k+1), f_2(k+1) + t_{1,k+1})$
  - $f_2(k) = a_{2,k} + \min(f_2(k+1), f_1(k+1) + t_{2,k+1})$

09-33: Assembly Line Scheduling

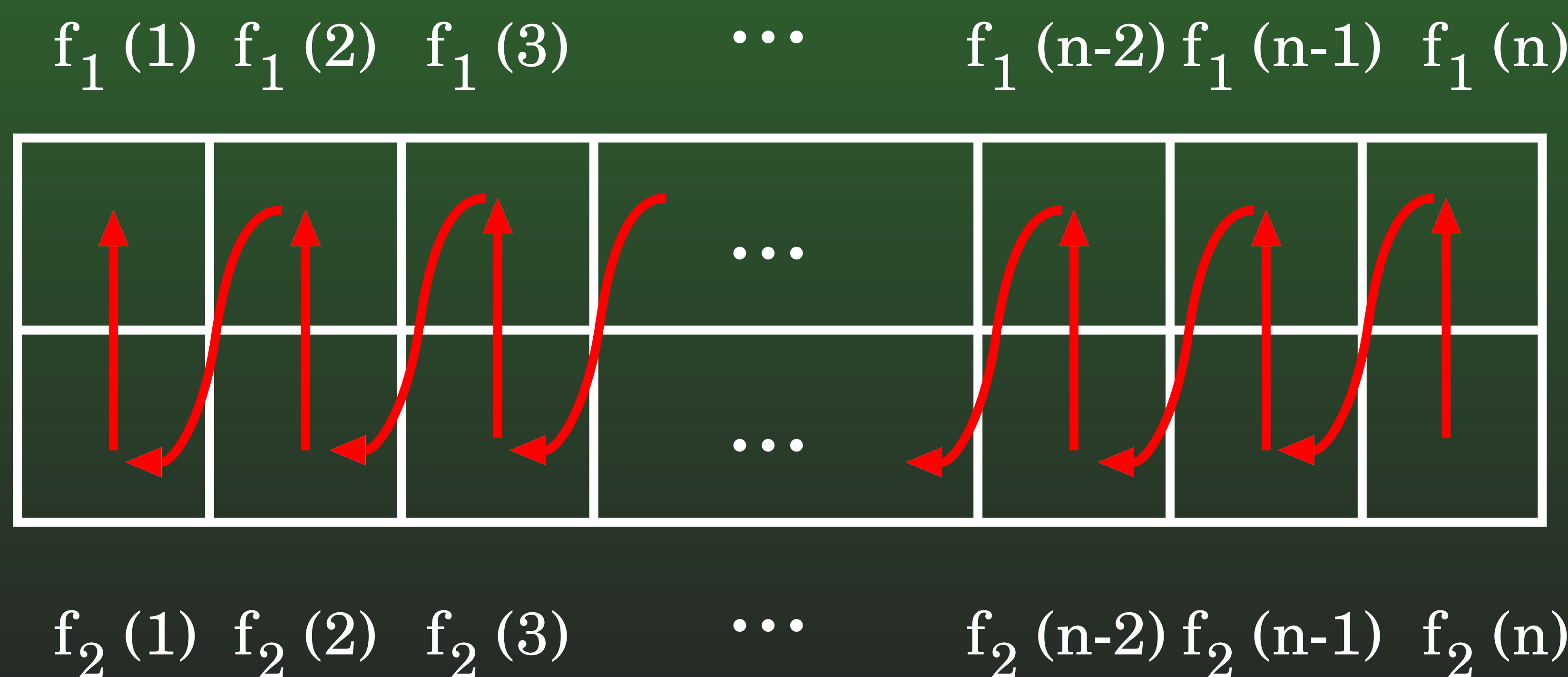
- We can now define a table:

$$T[i,j] = f_i(j)$$

$f_1(1)$	$f_1(2)$	$f_1(3)$	$\cdots$	$f_1(n-2)$	$f_1(n-1)$	$f_1(n)$
			$\cdots$			
			$\cdots$			
$f_2(1)$	$f_2(2)$	$f_2(3)$	$\cdots$	$f_2(n-2)$	$f_2(n-1)$	$f_2(n)$

09-34: Assembly Line Scheduling

- $T[1,n] = a_{1,n} + x_n$, $T[2,n] = a_{2,n} + x_n$
- $T[1,j] = a_{i,j} + \min(T[1,j+1], t_{1,j+1} + T[2,j+1])$
- $T[2,j] = a_{i,j} + \min(T[2,j+1], t_{2,j+1} + T[1,j+1])$



09-35: Assembly Line Scheduling

- Once we have the table T, what is the fastest we can get a car through?

09-36: Assembly Line Scheduling

- Once we have the table T, what is the fastest we can get a car through?

$$\min(T[1,1] + e_1, T[2,1] + e_2)$$

- How can we modify the algorithm to calculate the optimal path as well?

09-37: Assembly Line Scheduling

- Once we have the table T, what is the fastest we can get a car through?

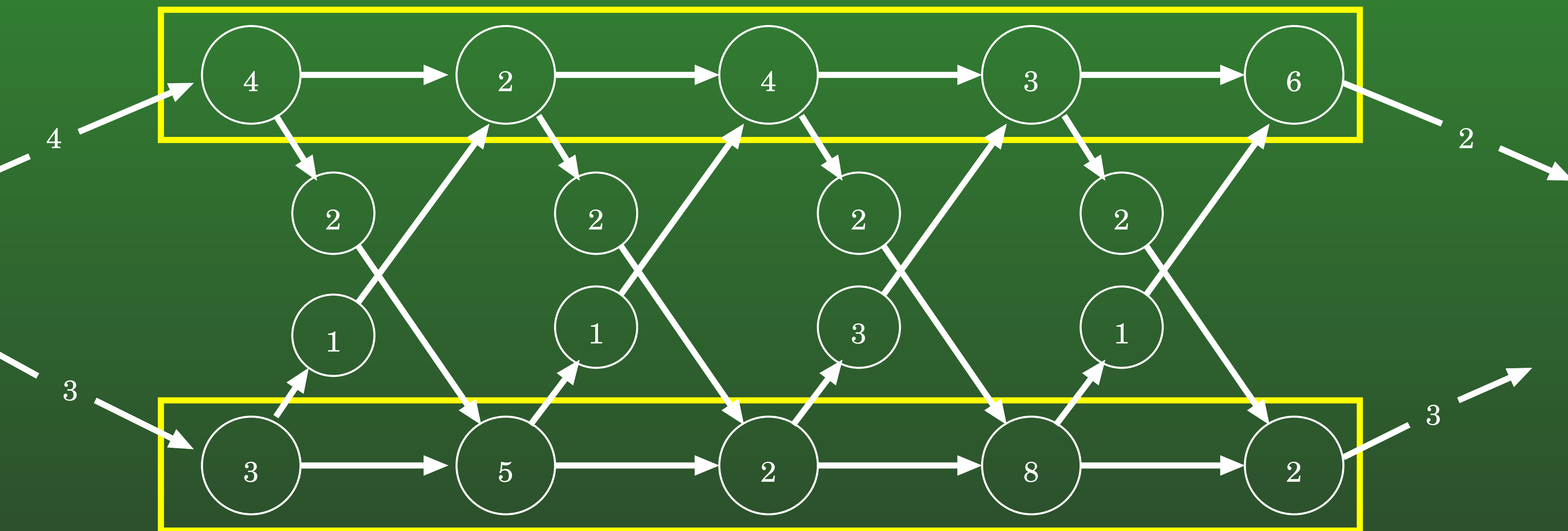
$$\min(T[1,1] + e_1, T[2,1] + e_2)$$

- How can we modify the algorithm to calculate the optimal path as well?

$$P[1,k] = \begin{cases} 1 & \text{if } T[1,k+1] \leq t_{1,k+1} + T[2,k+1] \\ 2 & \text{otherwise} \end{cases}$$

$$P[2,k] = \begin{cases} 2 & \text{if } T[2,k+1] \leq t_{2,k+1} + T[1,k+1] \\ 1 & \text{otherwise} \end{cases}$$

09-38: Assembly Line Scheduling



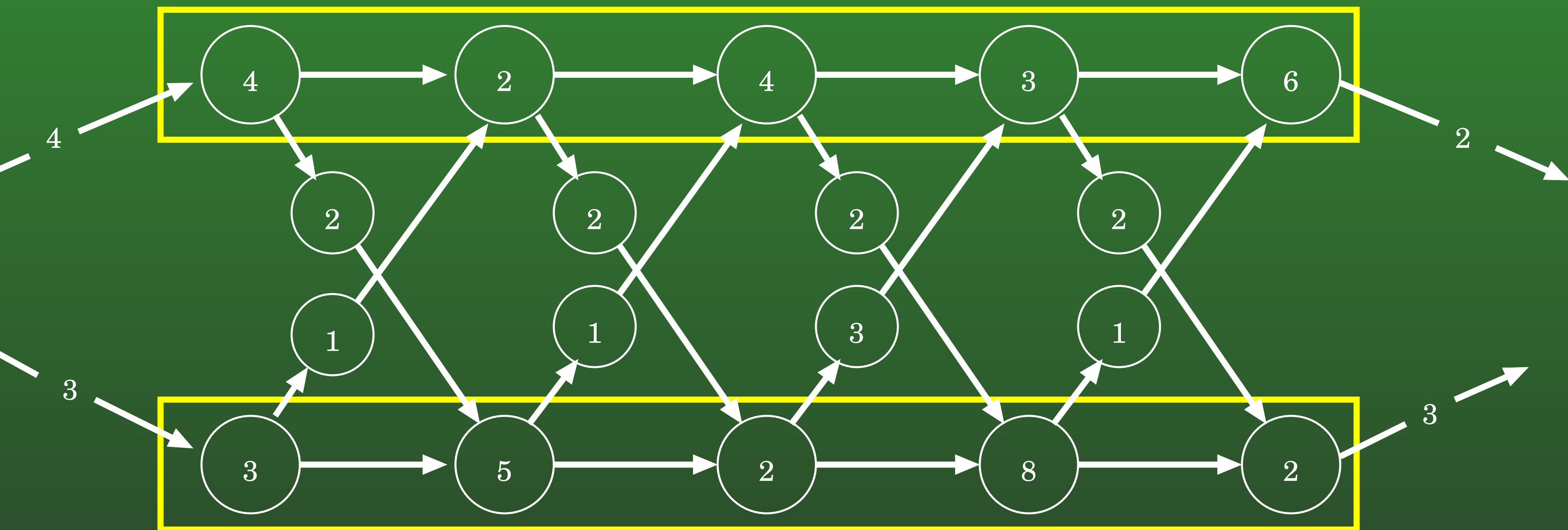
T

	1	2	3	4	5
1					
2					

P

	1	2	3	4	5
1					
2					

Assembly Line Scheduling



T		1	2	3	4	5
	1	20	16	14	10	8
	2	20	20	15	13	5

P		1	2	3	4	5
	1	1	1	1	2	-
	2	(1,2)	(1,2)	2	2	-

09-40: Assembly Line Scheduling

FastestBuild(a, t, e, x, n)
    $T[1, n] \leftarrow a1, n + x_1$
    $T[2, n] \leftarrow a1, n + x_2$
    for $j \leftarrow n - 1$ to 1 do
        if $T[1, j+1] \leq T[2, j+1] + t_{1,j+1}$ then
            $T[1, j] = T[1, j+1] + a_{1,j}$
            $P[1, j] = 1$
        else
            $T[1, j] = T[2, j+1] + t_{1,j+1} + a_{1,j}$
            $P[1, j] = 2$
        if $T[2, j+1] \leq T[1, j+1] + t_{2,j+1}$ then
            $T[2, j] = T[2, j+1] + a_{2,j}$
            $P[2, j] = 2$
        else
            $T[2, j] = T[1, j+1] + t_{2,j+1} + a_{2,j}$
            $P[2, j] = 1$
    if $T[1, 1] + e_1 > T[2, 1] + e_2$ then
        cost = $T[1, 1] + e_1$
    else
        cost = $T[2, 1] + e_2$

Making Change

- Problem:
  - Coins: 1, 5, 10, 25, 50
  - Smallest number of coins that sum to an amount X?
- How can we solve it?

09-42: Making Change

- Problem:
  - Coins: 1, 4, 6
  - Smallest number of coins that sum to an amount X?
- Does the same solution still work? Why not?

09-43: Making Change

- Problem:
  - Coins: d_1, d_2, d_3, $\ldots$, d_k
    - Can assume $d_1 = 1$
  - Value X
  - Find smallest number of coins that sum to X
- Solution:

09-44: Making Change

- Problem:
  - Coins: d_1, d_2, d_3, . . ., d_k
    - Can assume $d_1 = 1$
  - Value X
  - Find smallest number of coins that sum to X
- Solution:
  - We can pick any coint to start with: d_1, d_2, . . ., d_k
  - We then have a smaller subproblem: Finding change for the remainder after using the first coin

Making Change

- Problem:
  - Coins: $d_1, d_2, d_3, \ldots, d_k$
    - Can assume $d_1 = 1$
  - Value X
  - Find smallest number of coins that sum to X
- Solution:
  - $C[n]$ = smallest number of coins required for amount n
  - What is the base case?
  - What is the recursive case?

09-46: Making Change

- $C[n]$ = smallest number of coins required for amount n
  - Base Case:

$$C[0] = 0$$

  - Recursive Case:

$$C[n] = \min_{i, d_i <= n} (C[X - d_k] + 1)$$

09-47: Making Change

0	1	2	3	4	5	6	7	8	...	n
									...	

$$d_1 = 1, d_2 = 4, d_3 = 6$$

09-48: Making Change

0	1	2	3	4	5	6	7	8	...	n
0	1	2	3	1	2	1	2	2	...	

$$d_1 = 1, d_2 = 4, d_3 = 6$$

09-49: Making Change

```
C[0] = 0
for i = 1 to n do
   if (i >= d[0])
      C[i] = 1 + C[i - d[0]]
   for j=1 to numdenominations - 1 do
     if (d[j] >= i)
        C[i] = min(C[i], C[i-d[j]] + 1)
```

09-50: Making Change

- What's the required space, if total = n and # of different kinds of coins = m?
- What's the running time, if total = n and # of different kinds of coins = m?

09-51: Making Change

- What's the required space, if total = n and # of different kinds of coins = m?
  - $\Theta(n)$
- What's the running time, if total = n and # of different kinds of coins = m?
  - $O(m * n)$ (each square can take up to time $O(m)$ to calculate)

09-52: Making Change

- Given the table, can we determine the optimal way to make change for a given value X? How?

0	1	2	3	4	5	6	7	8	...	n
0	1	2	3	1	2	1	2	2	...	

$$d_1 = 1, d_2 = 4, d_3 = 6$$

09-53: Matrix Multiplication

- Quick review (on board)
  - Matrix A is $i \times j$
  - Matrix B is $j \times k$
  - # of scalar multiplications in $A * B$?

09-54: Matrix Multiplication

- Quick review (on board)
  - Matrix A is $i \times j$
  - Matrix B is $j \times k$
  - # of scalar multiplications in $A * B$?
    - $i * j * k$

09-55: Matrix Chain Multiplication

- Multiply a chain of matrices together
  - $A * B * C * D * E * F$
- Matrix Multiplication is associative
  - $(A * B) * C = A * (B * C)$
  - $(A * B) * (C * D) = A * (B * (C * D)) = ((A * B) * C) * D = A * ((B * C) * D) = (A * (B * C)) * D$

09-56: Matrix Chain Multiplication

- Order Matters!
- $A : (100 \times 100), B : (100 \times 100),$
  $C : (100 \times 100), D : (100 \times 1)$
  - $((A * B) * C) * D$ Scalar multiplications:
  - $A * (B * (C * D))$ Scalar multiplications:

Matrix Chain Multiplication

- Order Matters!
- $A : (100 \times 100), B : (100 \times 100),$
  $C : (100 \times 100), D : (100 \times 1)$
  - $((A * B) * C) * D$ Scalar multiplications: 2,010,000
  - $A * (B * (C * D))$ Scalar multiplications: 30,000

09-58: Matrix Chain Multiplication

- Matrices $A_1, A_2, A_3 \ldots A_n$
- Matrix A_i has dimensions $p_{i-1} \times p_i$
- Example:
  - $A_1 : 5 \times 7$, $A_2 : 7 \times 9$, $A_3 : 9 \times 2$, $A_4 : 2 \times 2$
  - $p_0 = 5, p_1 = 7, p_2 = 9, p_3 = 2, p_4 = 2$
  - How can we break $A_1 * A_2 * A_3 * \ldots * A_n$ into smaller subproblems?
  - Hint: Consider the last multiplication

Matrix Chain Multiplication

- $M[i, j]$ = smallest # of scalar multiplications required to multiply $A_i * \ldots * A_j$
- Breaking $M[1, n]$ into subproblems:
  - Consider last multiplication
  - (use whiteboard)

Matrix Chain Multiplication

- $M[i, j]$ = smallest # of scalar multiplications required to multiply $A_i * \ldots * A_j$
- Breaking $M[1, n]$ into subproblems:
  - Consider last multiplication:
  - $(A_1 * A_2 * \ldots * A_k) * (A_{k+1} * \ldots * A_n)$
  - $M[1, n] = M[1, k] + M[k + 1, n] + p_0 p_k p_n$
  - In general, $M[i, j] = M[i, k] + M[k + 1, j] + p_{i-1} p_k p_j$
    - What should we choose for k? which value between i and $j - 1$ should we pick?

09-61: Matrix Chain Multiplication

- Recursive case:

$$M[i,j] = \min_{i \leq k < j}(M[i,k] + M[k+1,j] + p_{i-1} * p_k * p_j$$

- What is the base case?

09-62: Matrix Chain Multiplication

- Recursive case:

$$M[i,j] = \min_{i \leq k < j}(M[i,k] + M[k+1,j] + p_{i-1} * p_k * p_j$$

- What is the base case?

$$M[i,i] = 0$$

for all i

Matrix Chain Multiplication

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

- In what order should we fill in the table? What to we need to compute $M[i,j]$?

	1	2	3	4	5	6	7	8
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

09-64: Matrix Chain Multiplication

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

- In what order should we fill in the table? What to we need to compute $M[i,j]$?

	1	2	3	4	5	6	7	8
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

Matrix Chain Multiplication

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

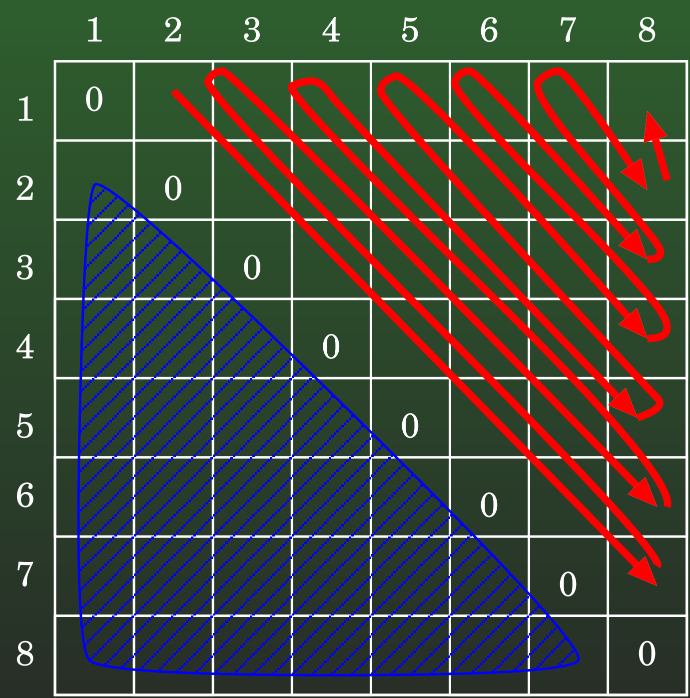
- What about the lower-left quadrant of the table?

	1	2	3	4	5	6	7	8
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

09-66: Matrix Chain Multiplication

$$M[i,j] = \min_{i \leq k < j}(M[i,k] + M[k+1,j] + p_{i-1} * p_k * p_j$$

- What about the lower-left quadrant of the table?



Not Defined

09-67: Matrix Chain Multiplication

```
Matrix-Chain-Order(p)
    n ← # of matrices
    for i ← 1 to n do
        M[i, i] ← 0
    for l ← 2 to n do
        for i ← 1 to n − l + 1
            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
                    S[i, j] = k
```

09-68: Dynamic Programming

- When to use Dynamic Programming
  - Optimal Program Substructure
    - Optimal solution needs to be constructed from optimal solutions to subproblems
    - Often means that subproblems are independent of each other

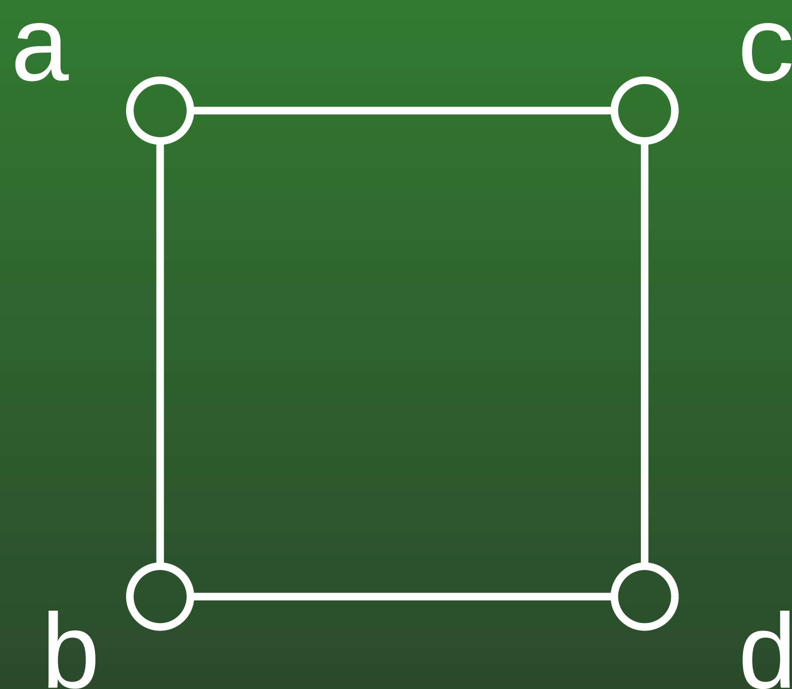
Dynamic Programming

- Optimal Program Substructure
  - Example: Shortest path in an undirected graph
    - let $p = x_1 \to x_2 \to \ldots \to x_k \to \ldots \to x_n$ be a shortest path from x_1 to x_n.
    - The subpath $x_1 \to x_k$ of p is a shortest path from x_1 to x_k
    - The subpath $x_k \to x_n$ of p is a shortest path from x_k to x_n
  - How would we prove this? (on board)

09-70: Dynamic Programming

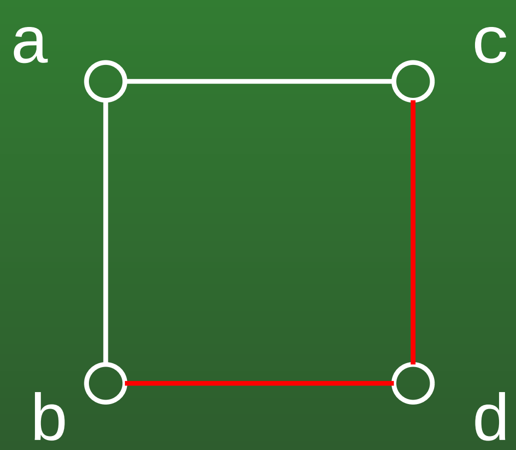
- Optimal Program Substructure
  - Example: Longest simple path (path without cycles) in an undirected graph
    - let $p = x_1 \rightarrow x_2 \rightarrow \ldots x_k \rightarrow \ldots \rightarrow x_n$ be a longest simple path from x_1 to x_n.
    - Is the subpath $x_1 \rightarrow x_k$ of p is the longest simple path from x_1 to x_k?
    - Is the subpath $x_k \rightarrow x_n$ of p is the longest simple path from x_k to x_n?

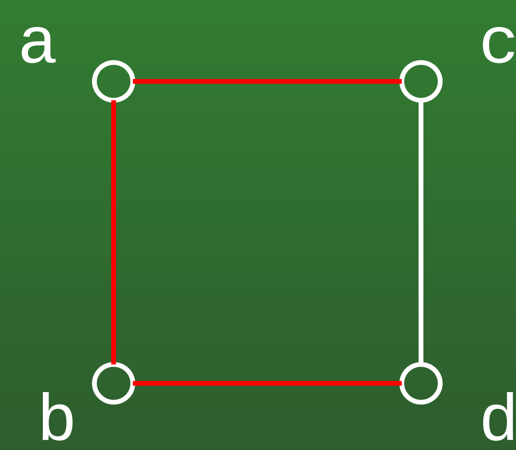
09-71: Dynamic Programming



- Longest simple path from $b \to c$?
- Longest simple path from $a \to c$ or $d \to c$?

09-72: Dynamic Programming

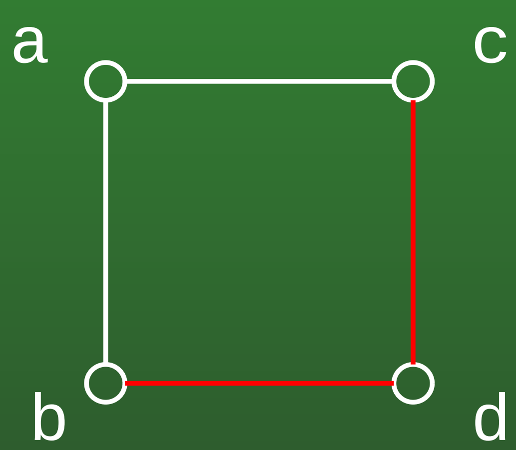


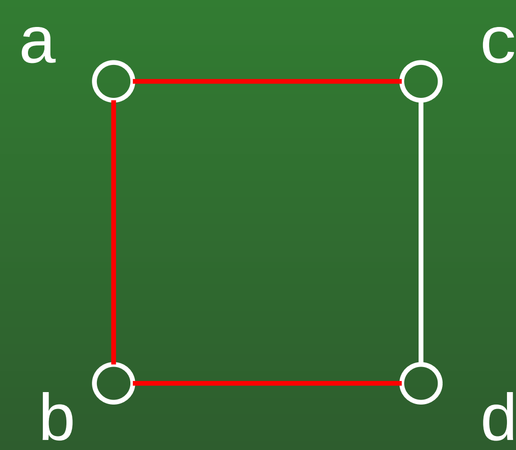


`ongest Path from b->c`   `Longest Path from d->c`

- Why isn’t the optimal solution composed of optimal solutions to subproblems?

09-73: Dynamic Programming





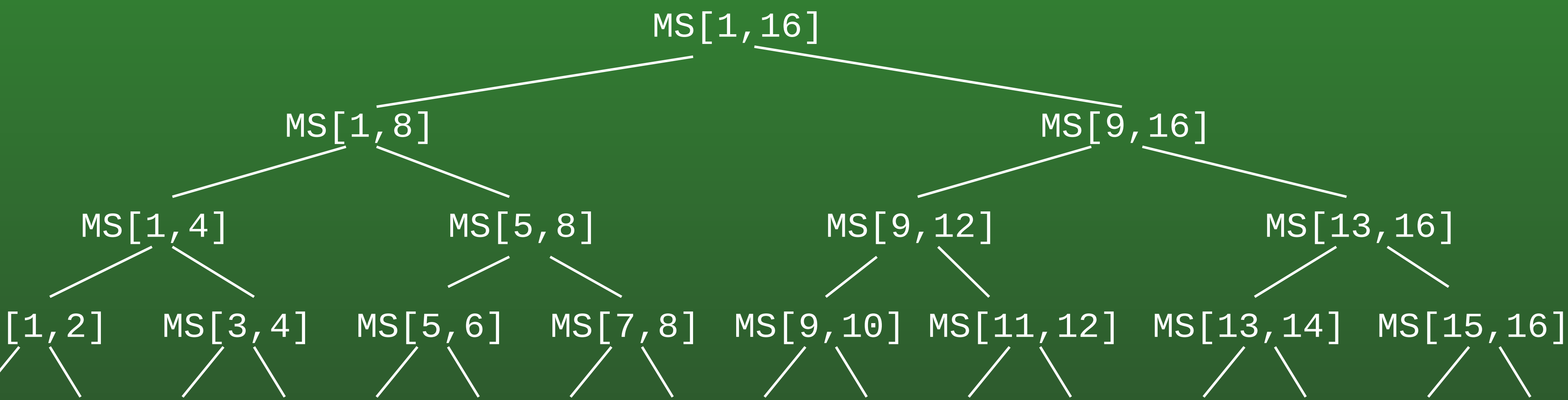
`ongest Path from b->c`   `Longest Path from d->c`

- Why isn’t the optimal solution composed of optimal solutions to subproblems?
  - Subproblems interfere with each other
  - Subproblems are not independent

09-74: Dynamic Programming

- When to use Dynamic Programming
  - Optimal Program Substructure
  - Repeated Subproblems
    - Solving the exact same problem more than once
    - Fibonacci, assembly line, matrix chain multiplication

09-75: Dynamic Programming



- Mergesort
  - No repeated subproblems
  - Dynamic Programming *not* good for “Divide & Conquer” algorithms

09-76: Sequence/Subsequence

- Sequence: Ordered list of elements
- Subsequence: Sequence with some elements removed
- Example: $< A, B, B, C, A, B >$
  - $< A, C, B >$
  - $< A, B, A >$
  - $< A, C, A, B >$

09-77: LCS

- Longest Common Subsequence
- $< A, B, C, B, D, A, B >$, $< B, D, C, A, B, A >$
  - $< B, C, A >$. Common subsequence, not longest
  - $< B, C, B, A >$ Longest Common subsequence
    - Is it unique?

09-78: LCS

- $< A, B, C, B, D, A, B >$, $< B, D, C, A, B, A >$
  - $< B, C, A >$. Common subsequence, not longest
  - $< B, C, B, A >$ Longest Common subsequence
    - $< B, C, A, B >$ also a LCS
    - Also consider $< A, A, A, B, B, B >$, $< B, B, B, A, A, A >$

09-79: LCS

- Finding a LCS:
  - $X =< x_1, x_2, x_3, \ldots, x_m >$
  - $Y =< y_1, y_2, y_3, \ldots, y_n >$
  - $x_m = y_n$
- What can we say about $Z =< z_1, \ldots, z_k >$ the LCS of X and Y?

LCS

- Finding a LCS:
  - $X =< x_1, x_2, x_3, \ldots, x_m >$
  - $Y =< y_1, y_2, y_3, \ldots, y_n >$
  - $x_m = y_n$
- What can we say about $Z =< z_1, \ldots, z_k >$ the LCS of X and Y?
  - $z_k = x_m = y_n$
  - $< z_1, \ldots z_{k-1} >$ is a LCS of $< x_1, \ldots, x_{m-1} >$ and $< y_1, \ldots, y_{n-1} >$

09-81: LCS

- Finding a LCS:
  - $X =< x_1, x_2, x_3, \ldots, x_m >$
  - $Y =< y_1, y_2, y_3, \ldots, y_n >$
  - $x_m \neq y_n$
- What can we say about $Z =< z_1, \ldots, z_k >$ the LCS of X and Y?

09-82: LCS

- Finding a LCS:
  - $X =< x_1, x_2, x_3, \ldots, x_m >$
  - $Y =< y_1, y_2, y_3, \ldots, y_n >$
  - $x_m \neq y_n$
- What can we say about $Z =< z_1, \ldots, z_k >$ the LCS of X and Y?
  - Z is a subsequence of $< x_1, \ldots, x_{m-1} >$, $< y_1, \ldots, y_n >$ or
  - Z is a subsequence of $< x_1, \ldots, x_m >$, $< y_1, \ldots, y_{n-1} >$

LCS

- How can we set up a recursive function that calculates the length of a LCS of two sequences?
  - If two sequences end in the same character, the LCS contains that character
  - If two sequences have a different last character, the length of the LCS is either the length of the LCS we get by dropping the last character from the first sequence, or the last character from the second sequence

09-84: LCS

- $C[i, j]$ = length of LCS of first i elements of X, and first j elements of Y
  - What is the base case?
  - What is the recursive case?

09-85: LCS

- $C[i, j]$ = length of LCS of first i elements of X, and first j elements of Y
  - Base Case:
    - C[i,0] = C[0,j] = 0
  - Recursive Case: $C[i, j]$
    - If $x_i = y_j$, $C[i, j] = 1 + C[i - 1, j - 1]$
    - If $x_i \neq y_j$, $C[i, j] = \max(C[i - 1, j], C[i, j - 1])$

09-86: LCS

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

09-87: LCS

	j	0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i		0	0	O	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

- How can we extract the sequence from this table?

09-88: LCS

	j	0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i		0	0	O	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

- Can we save space if we just want the length of the longest sequence?

09-89: LCS

LCS(X, Y)
    $m \leftarrow$ length(X)
    $n \leftarrow$ length(Y)
    for $i \leftarrow 1$ to m do
        $C[i, 0] \leftarrow 0$
    for $j \leftarrow 1$ to n do
        $C[j, 0] \leftarrow 0$
    for $i \leftarrow 1$ to m do
        for $j \leftarrow 1$ to n do
            if $X_i = Y_j$
                $C[i, j] = 1 + C[i-1, j-1]$
            else
                $C[i, j] = \max(C[i-1, j], C[i, j-1])$

09-90: Memoization

- Sometimes calculating the order to fill the table can be difficult
- Sometimes you do not need to calculate the entire table to get the answer you need
- Solution: “Memoize”recursive solution
  - Set table entries to ∞
  - At beginning of each recursive call, check to see if result is in table. If so, stop and return that result
  - When a value is returned, store it in the table as well

09-91: Memoization

```
Matrix-Chain-Order(p)
    n ← # of matrices
    for i ← 1 to n do
        for j ← 1 to n do
            M[i, j] ← ∞
    return MMChain(1, n)

MMChain(i, j)
    if M[i, j] ≠ ∞
        return M[i, j]
    if i = j
        M[i, j] = 0
        return 0
    for k ← i to j − 1 do
        if MMChain(i, j) < MMChain(i, k) + MMChain(k + 1, j) + p_{i−1} * p_k * p_j
            M[i.j] ← M[i, k] + M[k + 1, j] + p_{i−1}i * p_k * p_j
            S[i, j] ← k
    return M[i, j]
```

09-92: Memoization

- Use a hash table to implement memoization as well
  - Works for arbitrary function parameters

```
f(i,j,k)
   create a hash key out of i,j,k
   if key is in the hash table,
       return associated value
   <calculate f value>

   insert f value into the hash table
   return f value
```