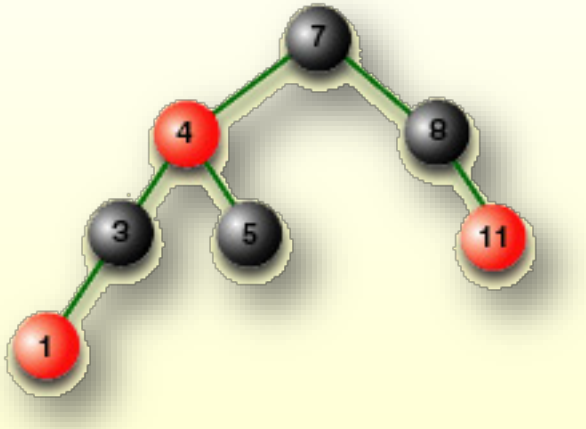
CS161:
Design and Analysis of Algorithms



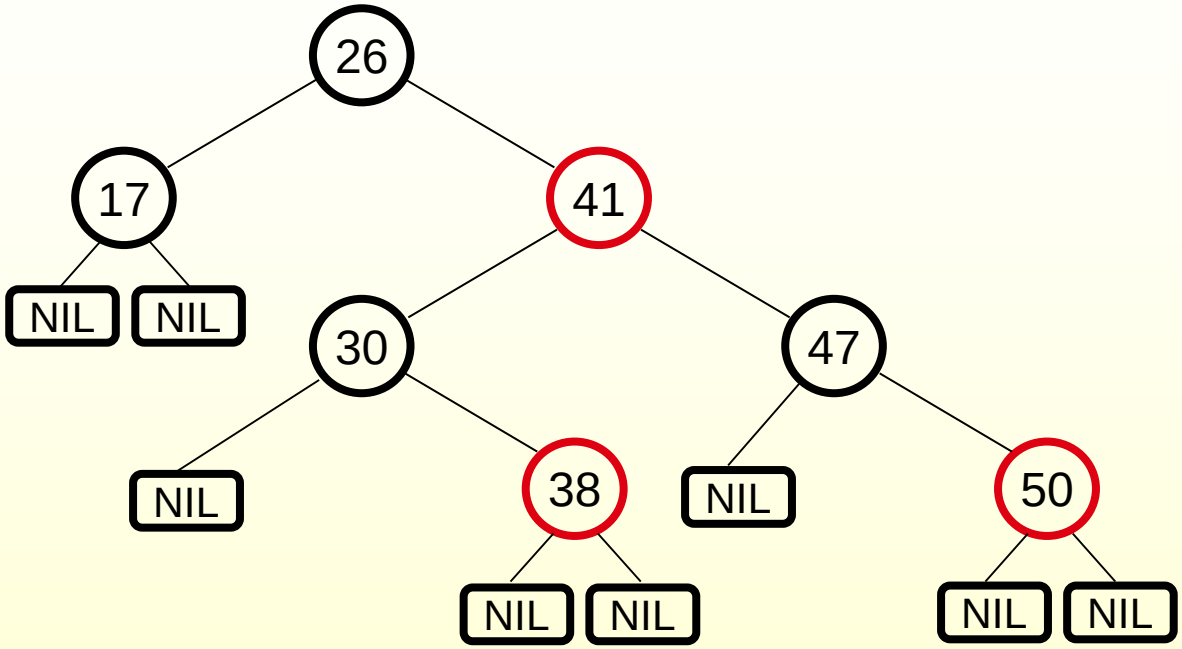
Lecture 12
Leonidas Guibas

Outline

- Review of last lecture: Balance trees, red-black trees
- Today: Dynamic programming
  - optimization problems
  - recursive vs iterative solutions
  - memoization

Slides modified from
- http://www.cs.virginia.edu/~luebke/cs332/
- http://www.cs.unc.edu/~plaisted/

Example: RED-BLACK-TREE



- For convenience we use a sentinel NIL[T] to represent all the null nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

Red-Black-Trees Properties

(**Satisfy the binary search tree property**)

1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf

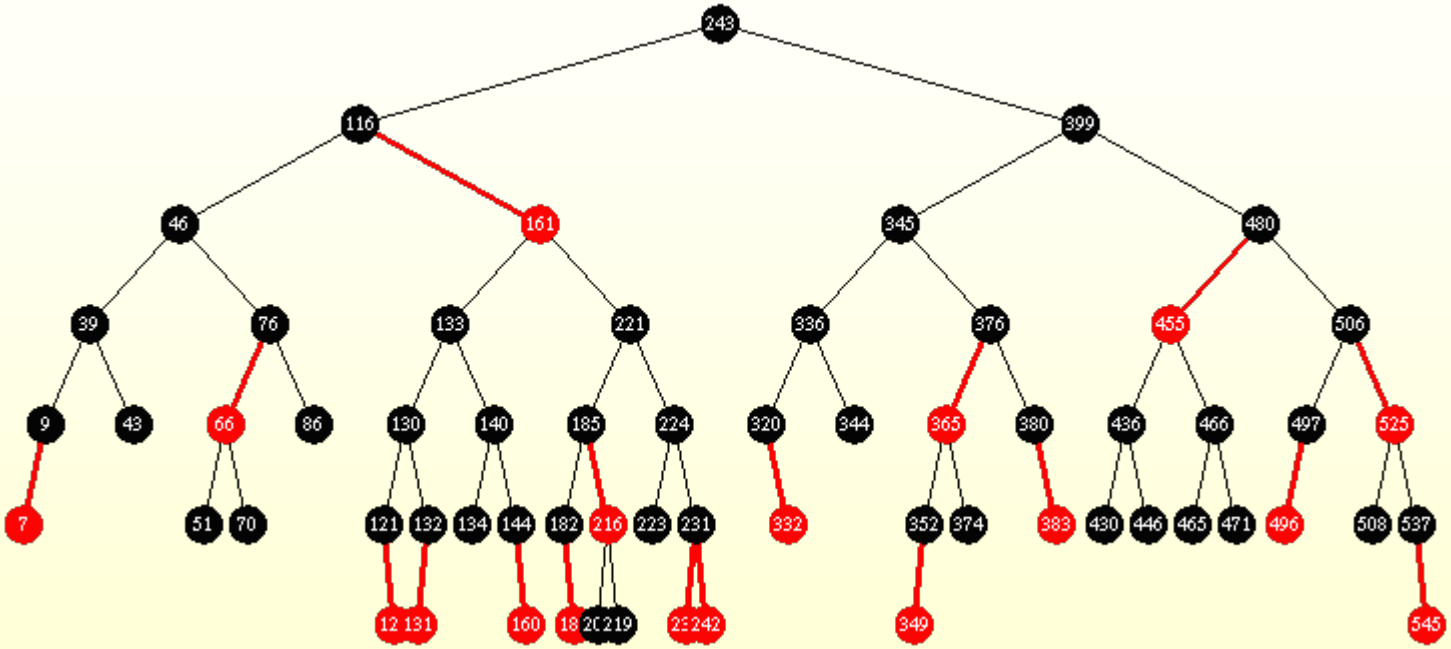
   local
5. For each node, all paths from that node to descendant leaves contain the same number of **black** nodes

   global

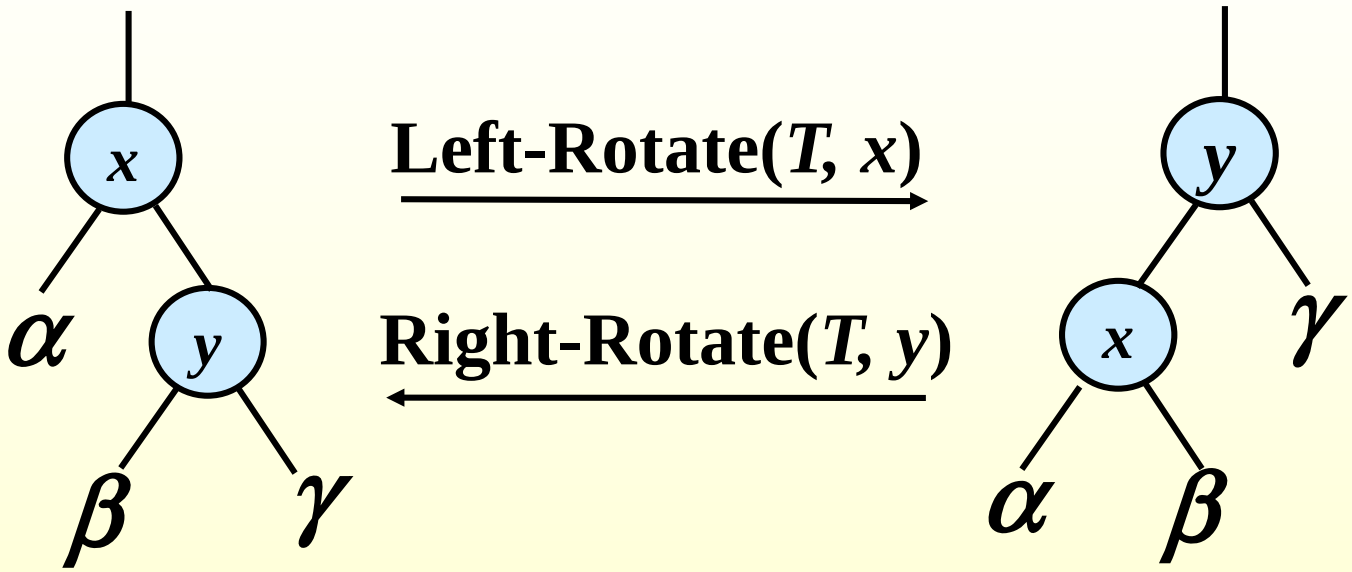
Important Property of Red-Black-Trees

A red-black tree with n internal nodes has height <u>at most</u> $2 \lg(n + 1)$

- Need to prove two claims first …

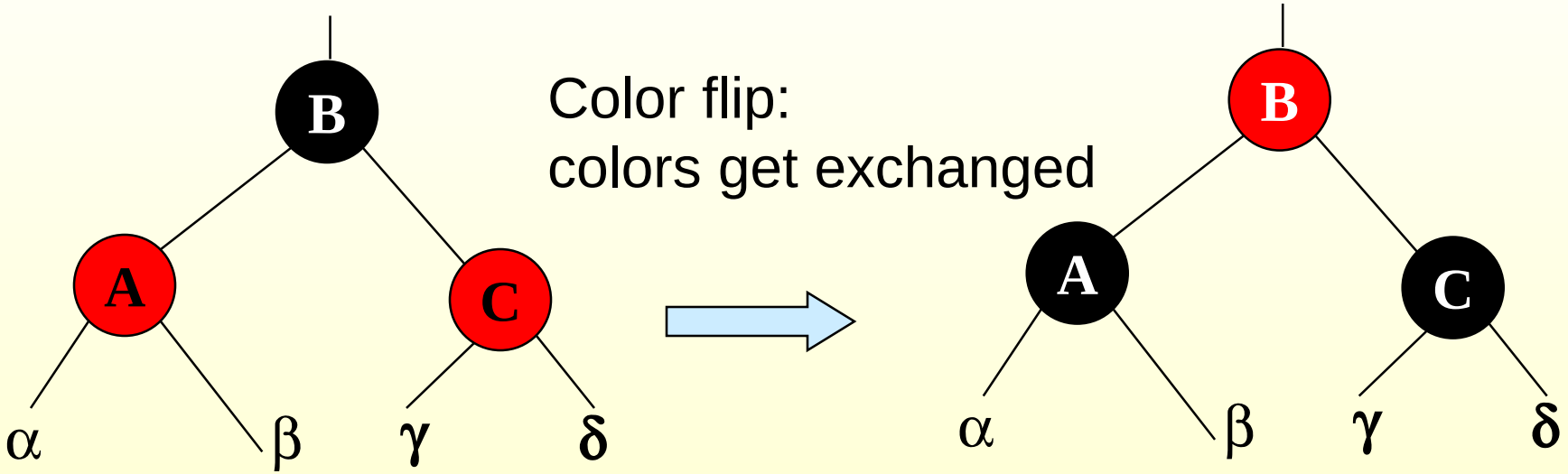

243
116
399
46
161
345
480
39
76
133
221
336
376
455
506
9
43
66
86
130
140
185
224
320
344
365
380
436
466
497
525
7
51
70
121
132
134
144
182
216
223
231
332
352
374
383
430
446
465
471
496
508
537
160
349
545

Rotations: Local Tree Rearrangements



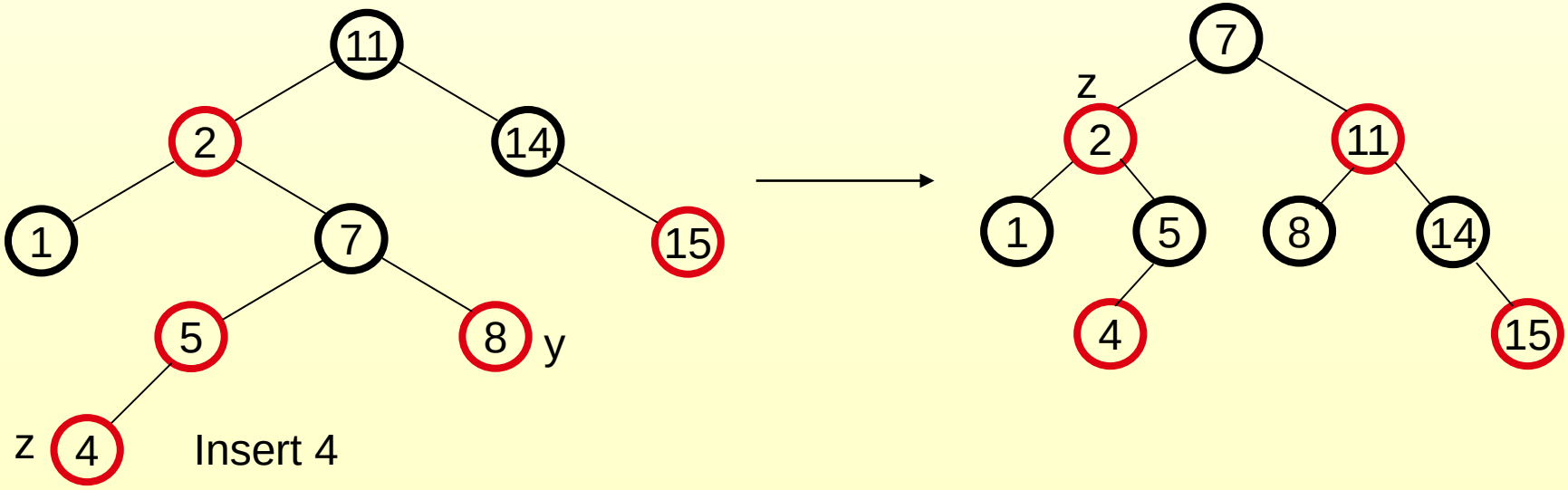
Internal tree rebalancing operations

Another Crucial Operation: Color Flip



Insertion and Deletion

- Insertions and deletions in O(lg n time) [downwards pass]
- Do them as in normal BST, then fixup the RB invariants [upwards pass]
- Only flip propagates, then 1-3 rotations fix the tree



Dynamic Programming

- The word “programming” is historical and predates computer programming
- Relates to certain “tabular” approaches (e.g., also in “linear programming”)
- A very powerful, general tool for solving certain optimization problems
- Once understood it is relatively easy to apply

A Motivating Example: Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

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

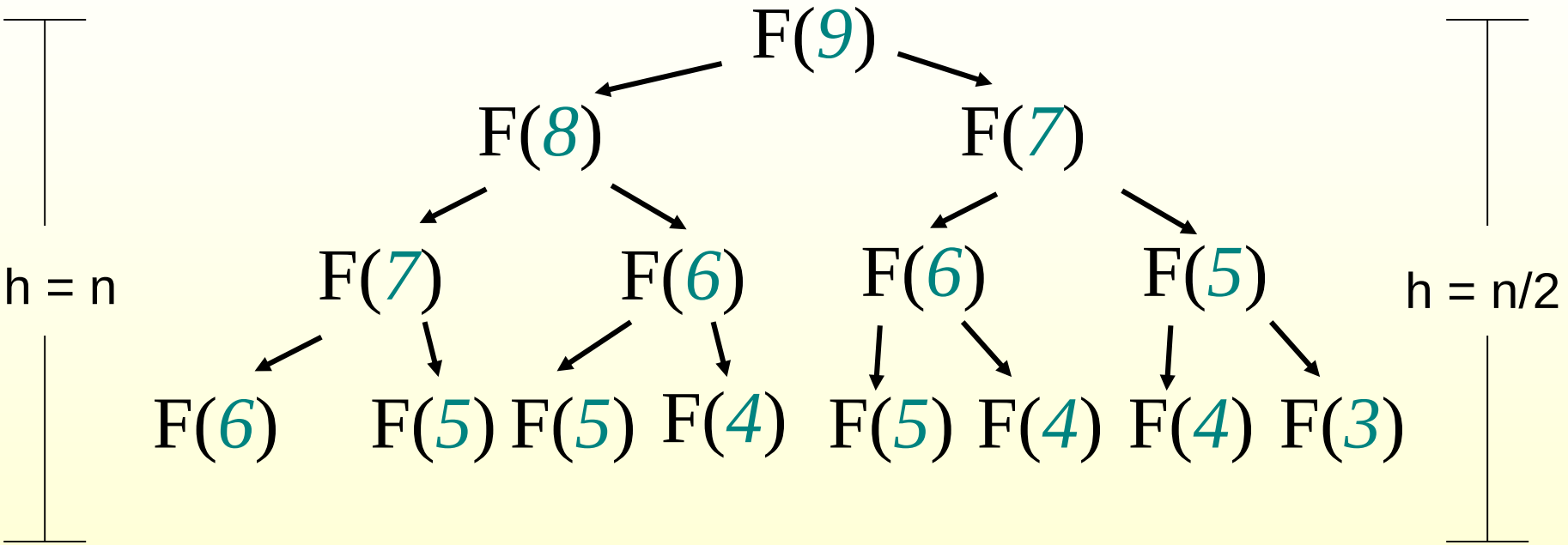
- How to compute F(n)?

A Recursive Algorithm

```
function fib(n)
        if (n == 0 or n == 1)  return 1;
        else  return fib(n-1) + fib(n-2);
```

What is the time complexity?

Recursion Tree



- Time complexity between $2^{n/2}$ and 2^n

$$T(n) = F(n) = O(1.6^n) = O(\varphi^n)$$

golden ratio

An Iterative Algorithm

```
function Fib(n)
        F[0] = 1;   F[1] = 1;
        for i = 2 to n
                F[i] = F[i-1] + F[i-2];
        Return F[n];
```

Time complexity? O(n)

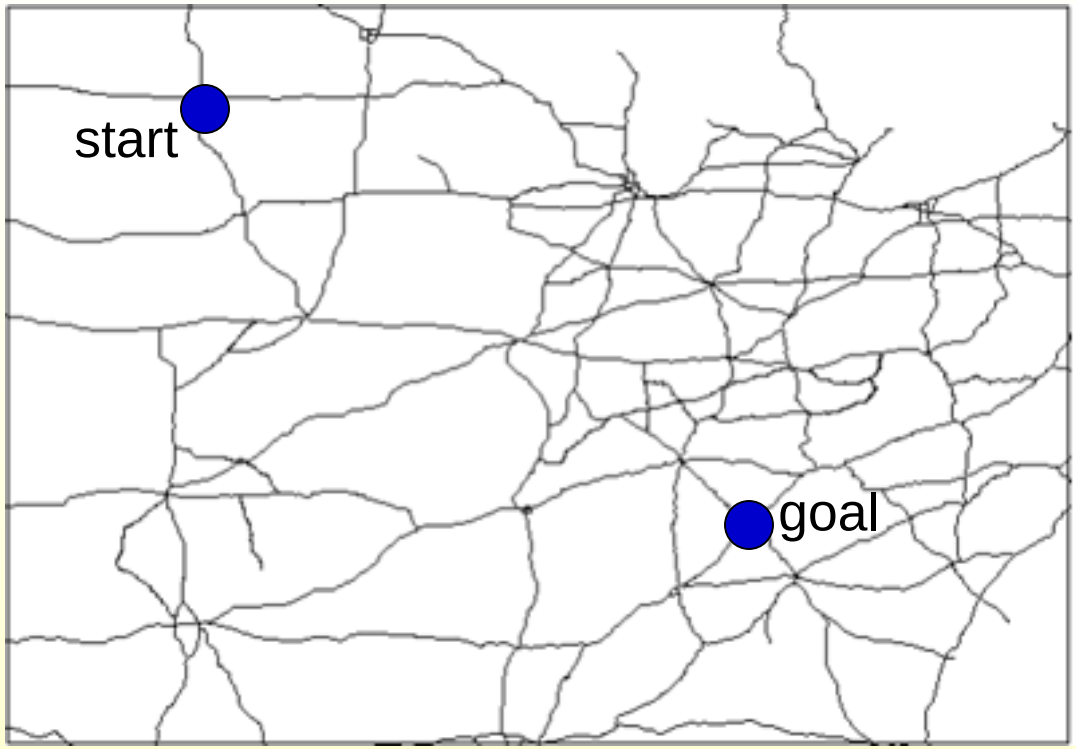
Recursion vs. Iteration

- Problem with the recursive Fib algorithm:
  - The same subproblem is being solved many times …
- Solution: avoid solving the same subproblem more than once

  (1) pre-compute all subproblems that may be needed later

  (2) Compute on demand, but memorize the solution to avoid recomputing
- Can one always speedup a recursive algorithm by making it an iterative algorithm?
  - E.g., Merge Sort
  - No, since there is no overlap between the two sub-problems

Another Motivating Example: Shortest Path Problem



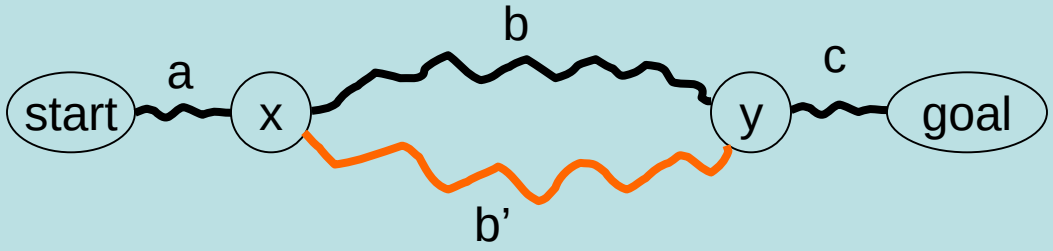
- Find the shortest path from start to goal
- An important graph optimization problem

Possible Approaches

- Brute-force algorithm
  - Enumerate all possible paths and compare
  - How many?
- Greedy algorithm
  - Always use the edge that takes you closest to the goal
  - Does not necessarily lead to the optimal solution
- Can you think of a recursive strategy?

Optimal Substructure

- Claim: if a path start→goal is optimal, any sub-path x→y, where x, y is on the optimal path, is also optimal.



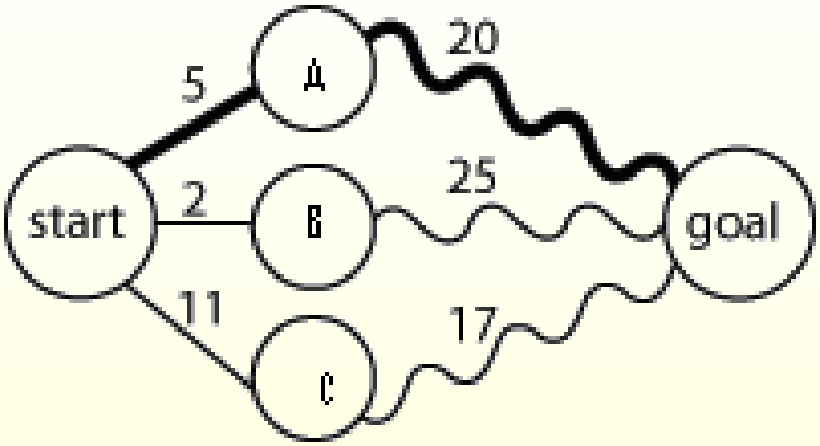
a + b + c is shortest

b' < b

a + b' + c < a + b + c

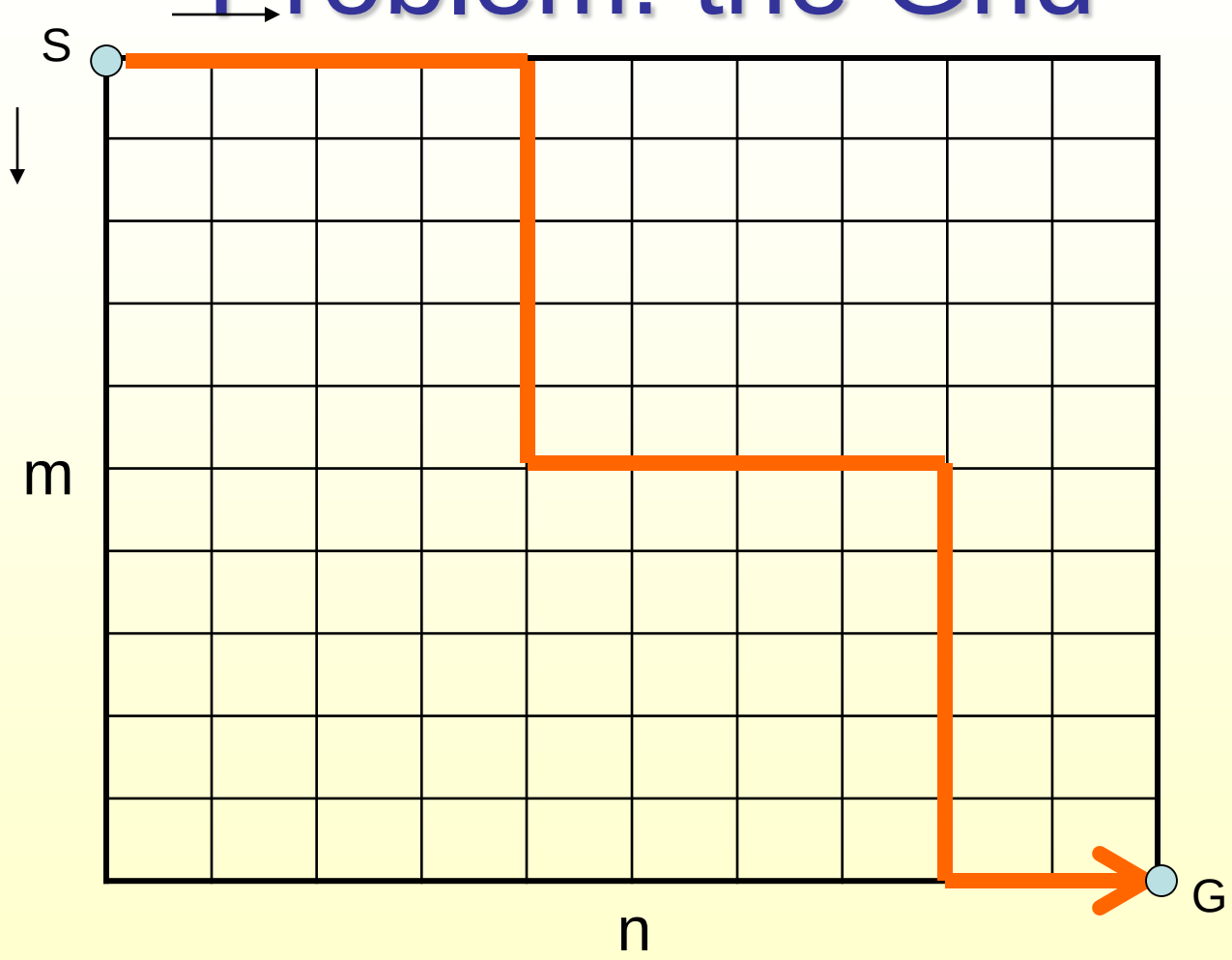
- Proof by contradiction
  - If the subpath b between x and y is not the shortest
  - we can replace it with a shorter one, b'
  - which will reduce the total length of the new path
  - the optimal path from start to goal is not the shortest => contradiction!
  - Hence, the subpath b must be the shortest among all paths from x to y

Shortest Path Problem



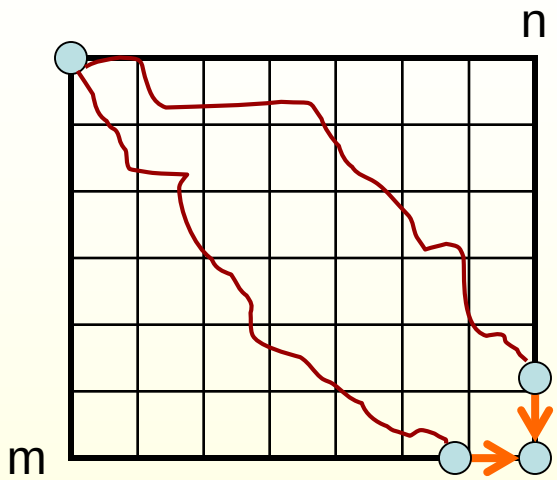
$$SP(start, goal) = \min \begin{cases} dist(start, A) + SP(A, goal) \\ dist(start, B) + SP(B, goal) \\ dist(start, C) + SP(C, goal) \end{cases}$$

A Special Shortest Path Problem: the Grid



Each edge has a length (cost). We need to get to G from S. Can only move to the right or bottom. Aim: find a path with the minimum total length

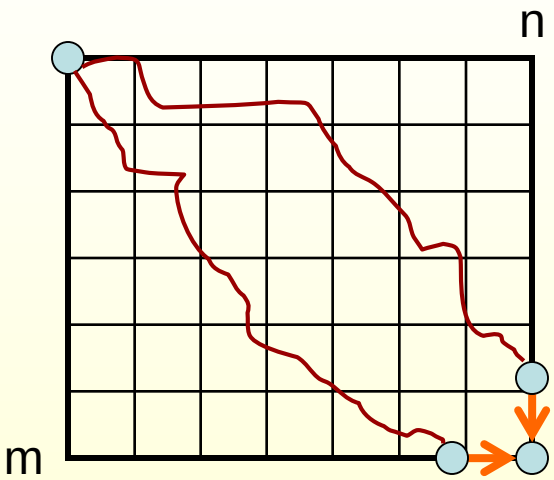
Recursive Solution



- Suppose we've found the shortest path
- It must use one of the two edges:
  - (m, n-1) to (m, n) Case 1
  - (m-1, n) to (m, n) Case 2
- If case 1
  - find shortest path from (0, 0) to (m, n-1)
  - SP(0, 0, m, n-1) + dist(m, n-1, m, n) is the overall shortest path
- If case 2
  - find shortest path from (0, 0) to (m-1, n)
  - SP(0, 0, m, n-1) + dist(m, n-1, m, n) is the overall shortest path
- We don't know which case is true
  - But if we've found the two paths, we can compare
  - Actual shortest path is the one with shorter overall length

Recursive Solution

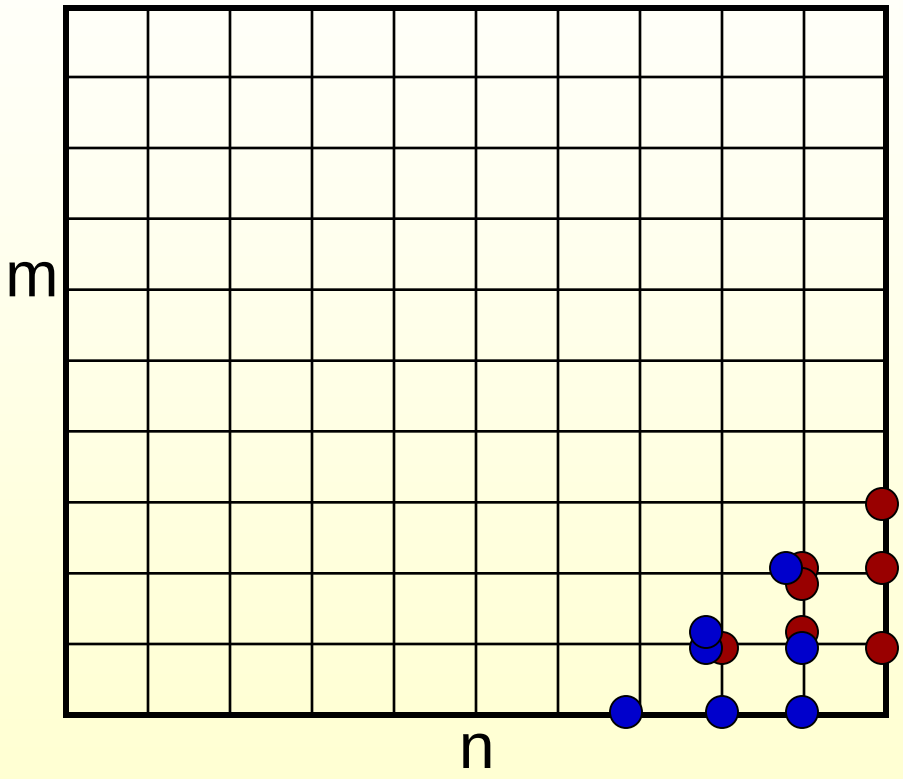
Let $F(i, j) = SP(0, 0, i, j)$. => $F(m, n)$ is length of SP from (0, 0) to (m, n)



$$F(m, n) = min \begin{cases} F(m\text{-}1, n) + dist(m\text{-}1, n, m, n) \\ F(m, n\text{-}1) + dist(m, n\text{-}1, m, n) \end{cases}$$

Generalize

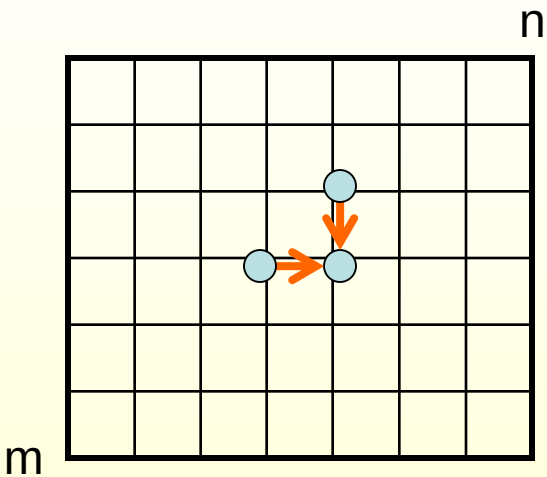
$$F(i, j) = min \begin{cases} F(i\text{-}1, j) + dist(i\text{-}1, j, i, j) \\ F(i, j\text{-}1) + dist(i, j\text{-}1, i, j) \end{cases}$$



- To find shortest path from (0, 0) to (m, n), we need to find shortest path to both (m-1, n) and (m, n-1)
- If we use recursive call, some subpaths will be recomputed for many times
- Strategy: solve subproblems in certain order so that redundant computation can be avoided

Dynamic Programming

Let $F(i, j) = SP(0, 0, i, j)$. => $F(m, n)$ is length of SP from (0, 0) to (m, n)

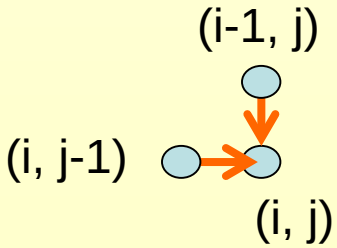


$$F(i, j) = min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

Boundary condition: $i = 0$ or $j = 0$.
Easy to figure out.

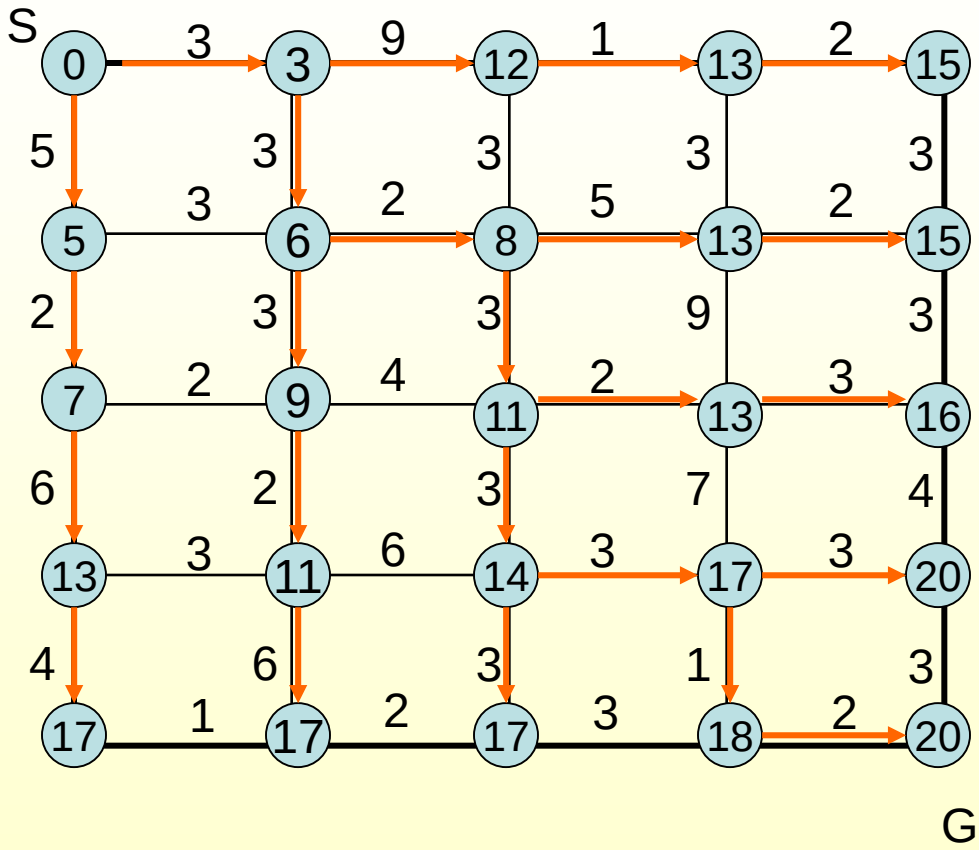
$i = 1 .. m, j = 1 .. n$

Number of unique subproblems = m * n



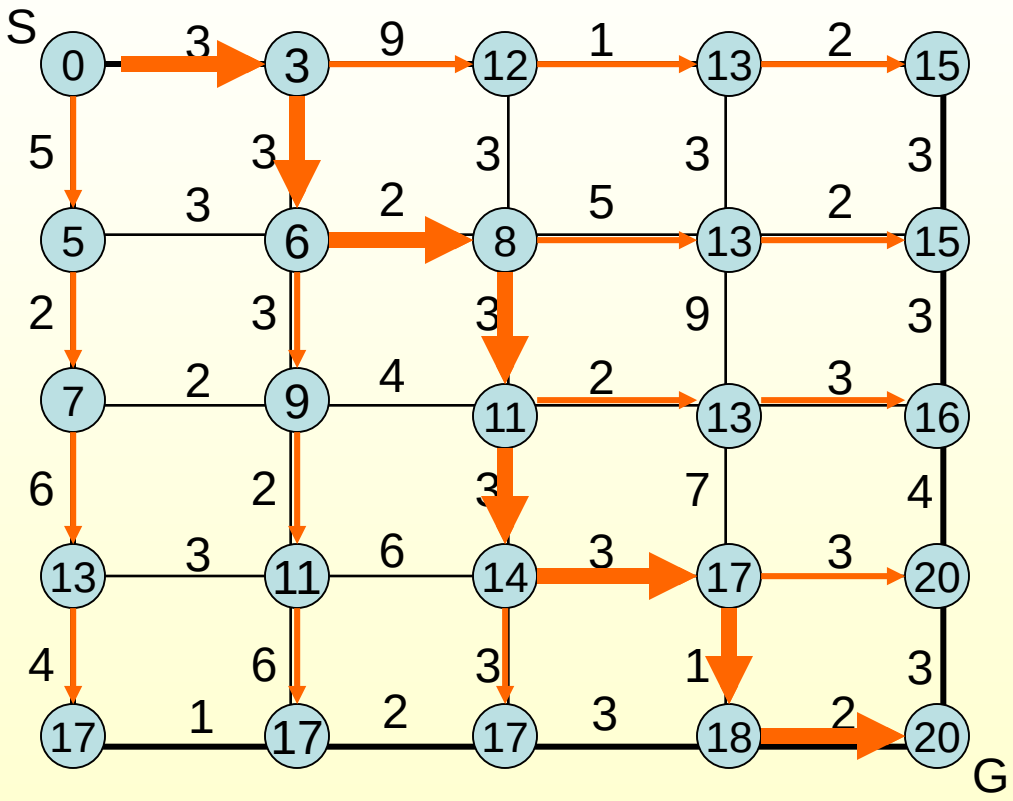
Data dependency determines order to compute: left to right, top to bottom

Dynamic Programming Illustration



$$F(i, j) = min \begin{cases} F(i\text{-}1, j) + dist(i\text{-}1, j, i, j) \\ F(i, j\text{-}1) + dist(i, j\text{-}1, i, j) \end{cases}$$

Trace Back



Shortest path has length 20
What is the actual shortest path?

Elements of Dynamic Programming

- Optimal sub-structures
  - Optimal solutions to the original problem contains optimal solutions to sub-problems
- Overlapping sub-problems
  - Some sub-problems appear in many solutions
- Memorization and reuse
  - Carefully choose the order that sub-problems are solved, such that each sub-problem will be solved for at most once and the solution can be reused

Two Steps to Dynamic Programming

- Formulate the solution as a recurrence relation on solutions to subproblems.
- Specify an order to solve the subproblems so you always have what you need (solved subproblem).
  - Bottom-up
    - Tabulate the solutions to all subproblems before they are used
  - What if you cannot determine the order easily, of if not all subproblems are needed?
  - Top-down
    - Compute when needed.
    - Remember the ones you've computed

Question

- If instead of shortest path, we want (simple) longest path, can we still use dynamic programming?
  - Simple means no cycle allowed
  - Not in general, since no optimal substructure
    - Even if s…m…t is the optimal path from s to t, s…m may not be optimal from s to m
  - Yes for acyclic graphs

Example Problems Solvable by Dynamic Programming

- Sequence alignment problem (several different versions)
- Shortest path in general graphs
- Knapsack (several different versions)
- Scheduling
- Matrix products
- etc.

- We discuss alignment, matrix products, and (certain types of) knapsack problems
- Shortest path in graph algorithms

Sequence Alignment

- Compare two strings to see if they are similar
  - We need to define similarity
  - Very useful in many many applications
  - Comparing DNA sequences, text articles, source code, etc.
  - Example: Longest Common Subsequence problem (LCS)

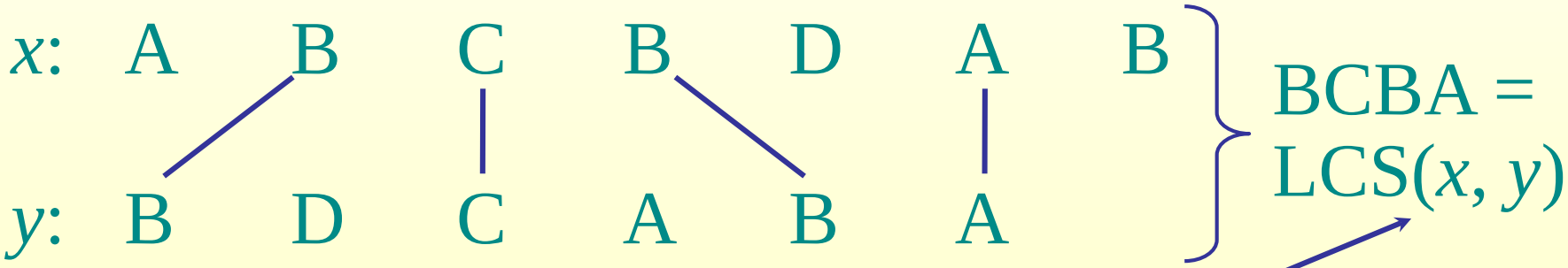
Common Subsequence

- A subsequence of a string is the string with zero or more chars left out
- A common subsequence of two strings:
  - A subsequence of both strings
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both common subsequences of x and y

Longest Common Subsequence

- Given two sequences $x[1\,..\,m]$ and $y[1\,..\,n]$, find a longest subsequence common to them both.

"a" *not* "the"



functional notation,
but not a function

Brute-Force LCS Algorithm

Check every subsequence of $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$.

Analysis

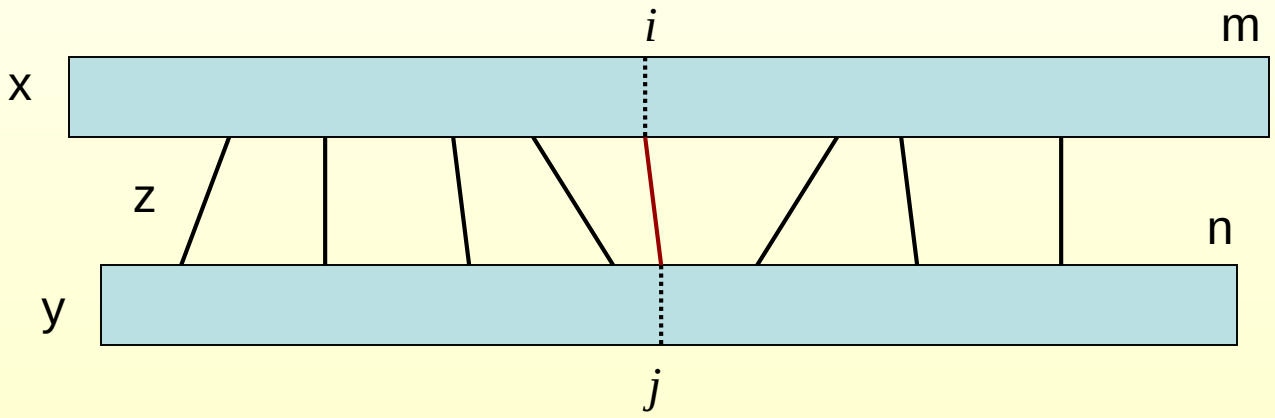
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm: a DP strategy

- Key: optimal substructure and overlapping sub-problems
- First we find the length of LCS. Later we modify the algorithm to find LCS itself.

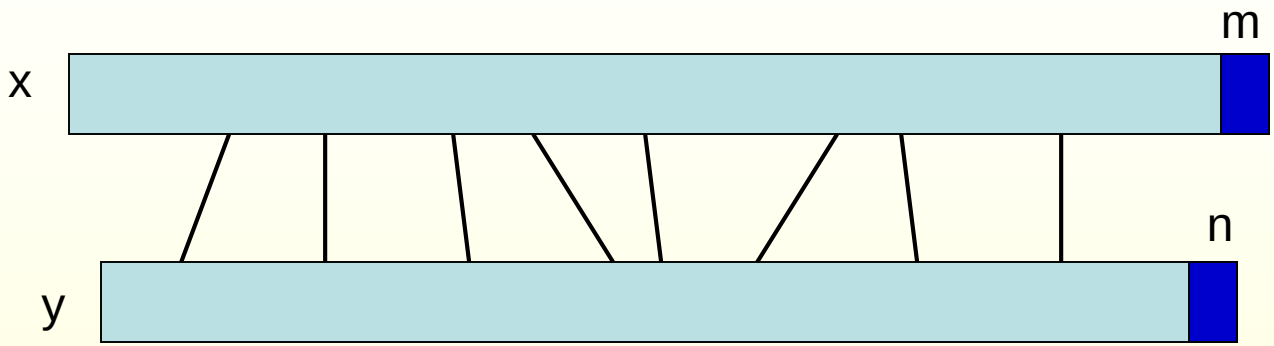
Optimal Substructure

- Notice that the LCS problem has *optimal substructure*: parts of the final solution are solutions of subproblems.
  - If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.



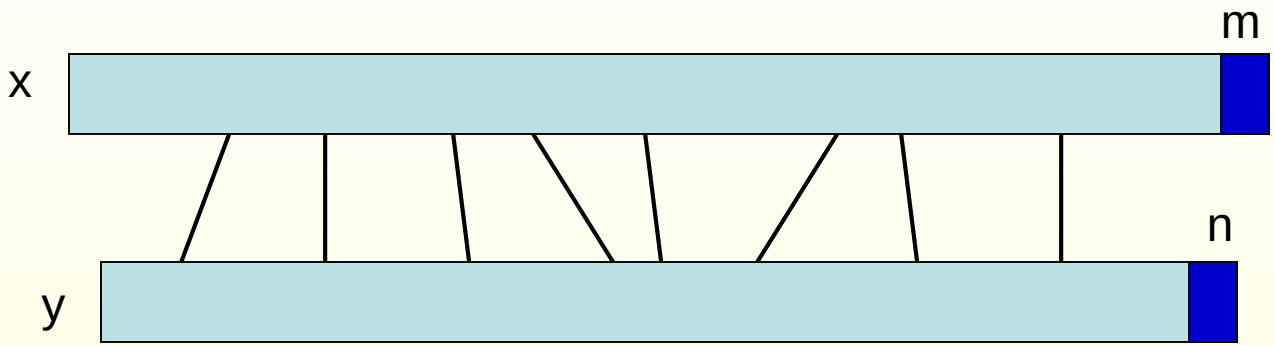
- Subproblems: “find LCS of pairs of *prefixes* of x and y”

Recursive Thinking



- Case 1: $x[m]=y[n]$. There is **an** optimal LCS that matches $x[m]$ with $y[n]$. → Compute LCS $(x[1..m-1], y[1..n-1])$
- Case 2: $x[m]\neq y[n]$. At most one of them is in LCS
  - Case 2.1: *x[m] not in LCS* → Compute LCS $(x[1..m-1], y[1..n])$
  - Case 2.2: *y[n] not in LCS* → Compute LCS $(x[1..m], y[1..n-1])$

Recursive Thinking



- Case 1: $x[m]=y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) \,||\, x[m]$

Reduce both sequences by 1 char

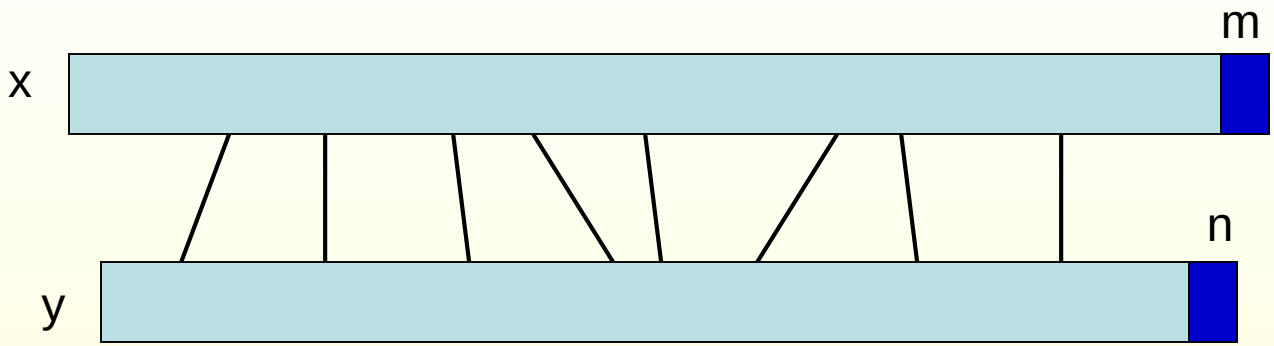
concatenate

- Case 2: $x[m] \neq y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n])$ or

$LCS(x[1..m], y[1..n-1])$, whichever is longer

Reduce one sequence by 1 char

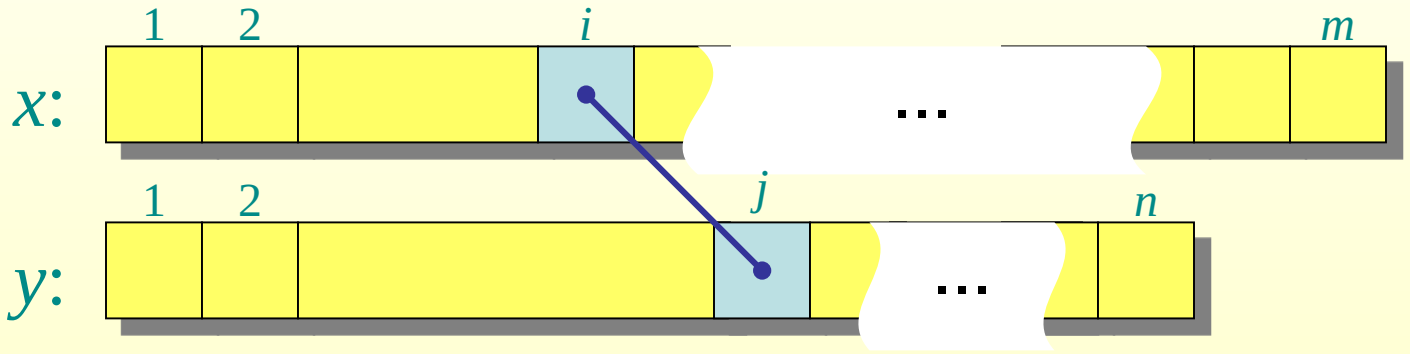
Finding Length of LCS



- Let *c[i, j]* be the length of LCS(*x[1..i], y[1..j]*)
  => *c[m, n]* is the length of LCS(*x*, *y*)
- If $x[m] = y[n]$
  $c[m, n] = c[m\text{-}1, n\text{-}1] + 1$
- If $x[m] != y[n]$
  $c[m, n] = max\{ c[m\text{-}1, n], c[m, n\text{-}1] \}$

Generalize: Recursive Formulation

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

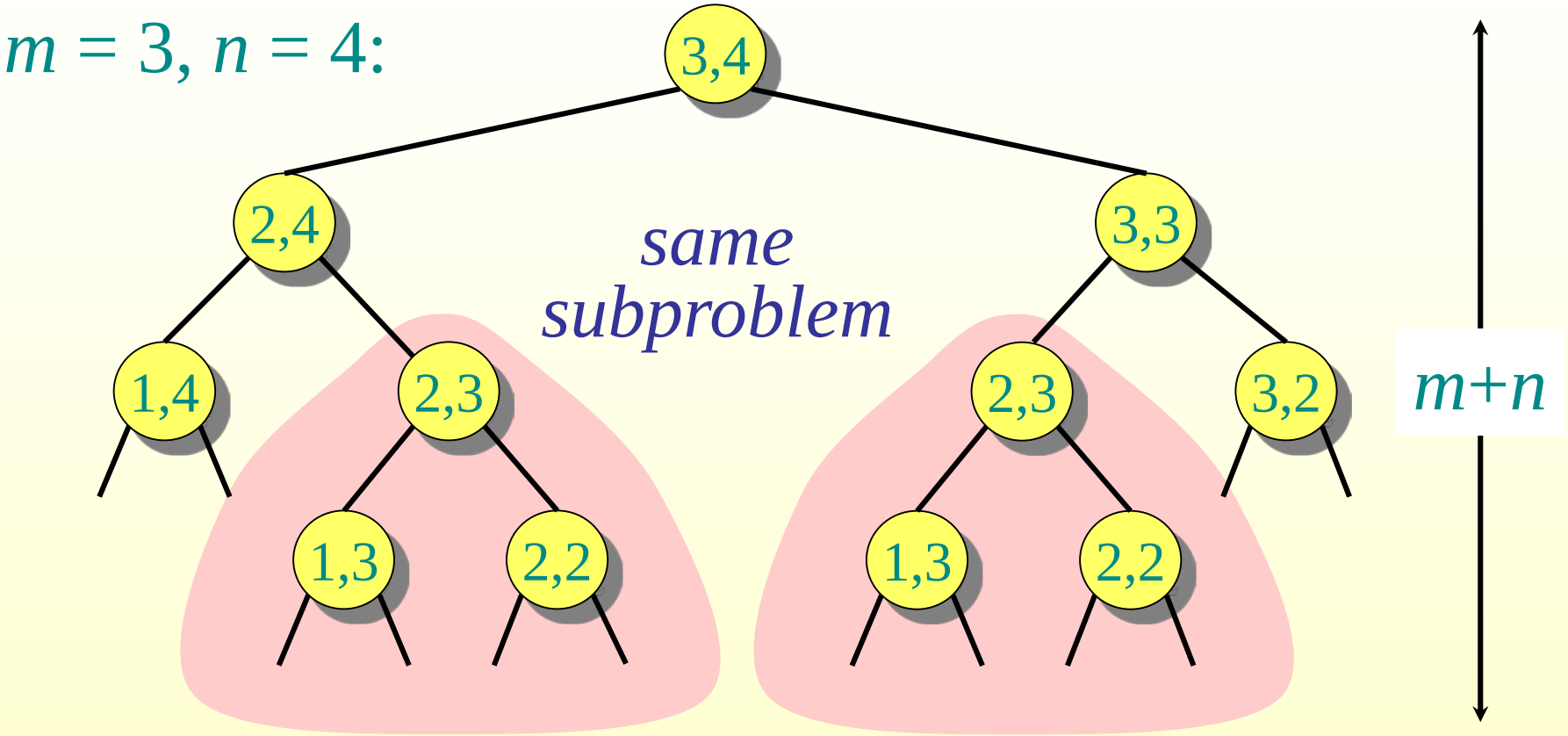


Recursive Algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i–1, j–1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i–1, j)$,
      LCS$(x, y, i, j–1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion Tree

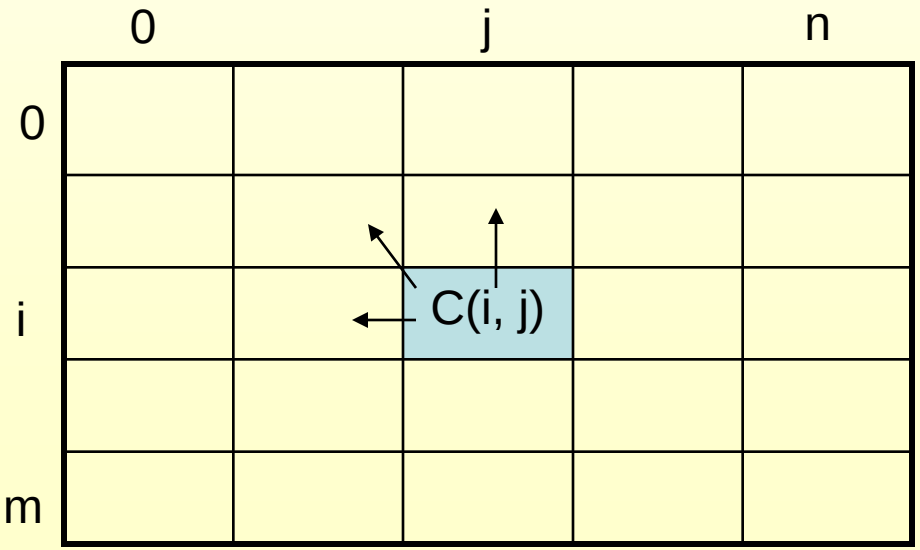


Height = $m + n$ ⇒ work potentially exponential, but we're solving subproblems already solved!

DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: *m* * *n*

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



DP Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m  c[i,0] = 0   // special case: Y[0]
4. for j = 1 to n   c[0,j] = 0   // special case: X[0]
5. for i = 1 to m                // for all X[i]
6.      for j = 1 to n                    // for all Y[j]
7.              if ( X[i] == Y[j])
8.                      c[i,j] = c[i-1,j-1] + 1
9.              else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]							
1	**A**							
2	**B**							
3	**C**							
4	**B**							

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**					
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB

BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**		
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB

BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB

BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

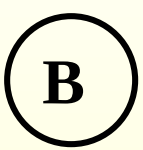
i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**				
3	**C**		**0**					
4	**B**		**0**					

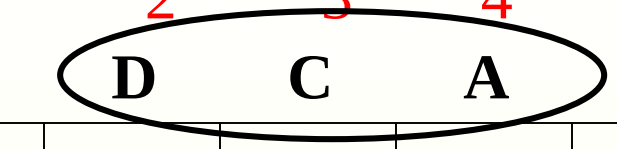
if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					





if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (9)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**		
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB

BDCAB

i		j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**	
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	**3**

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

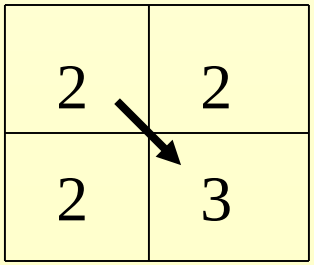
since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to Find Actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was computed:

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

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.



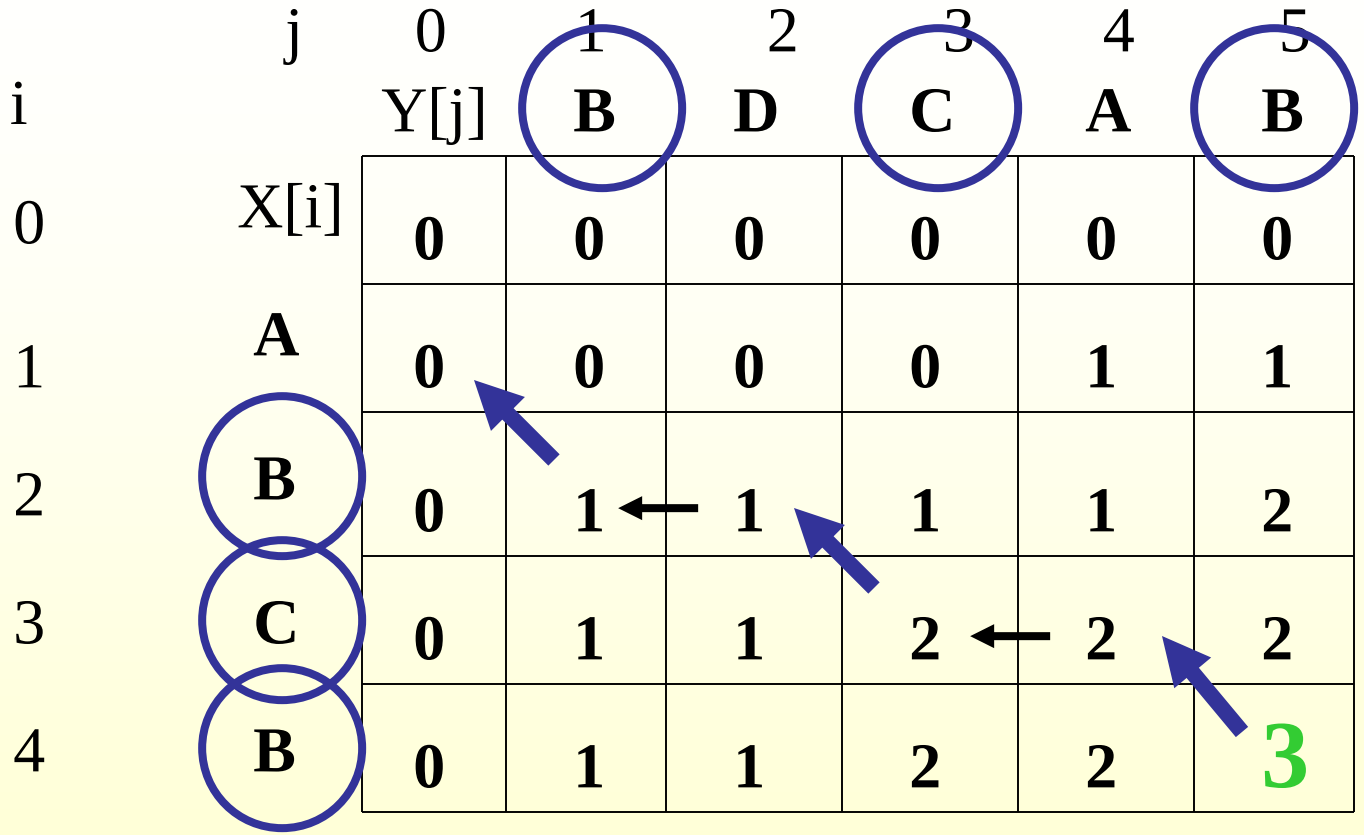
For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Time for trace back: $O(m+n)$.

Finding LCS (2)

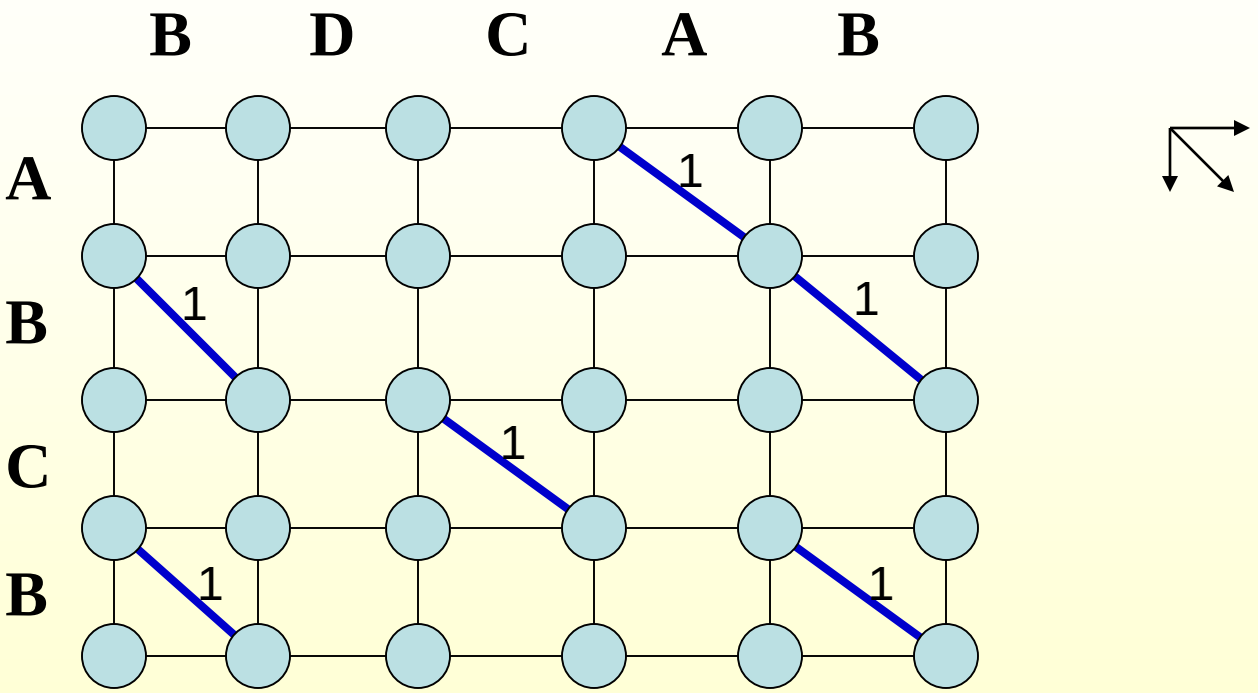


LCS (reversed order): **B C B**

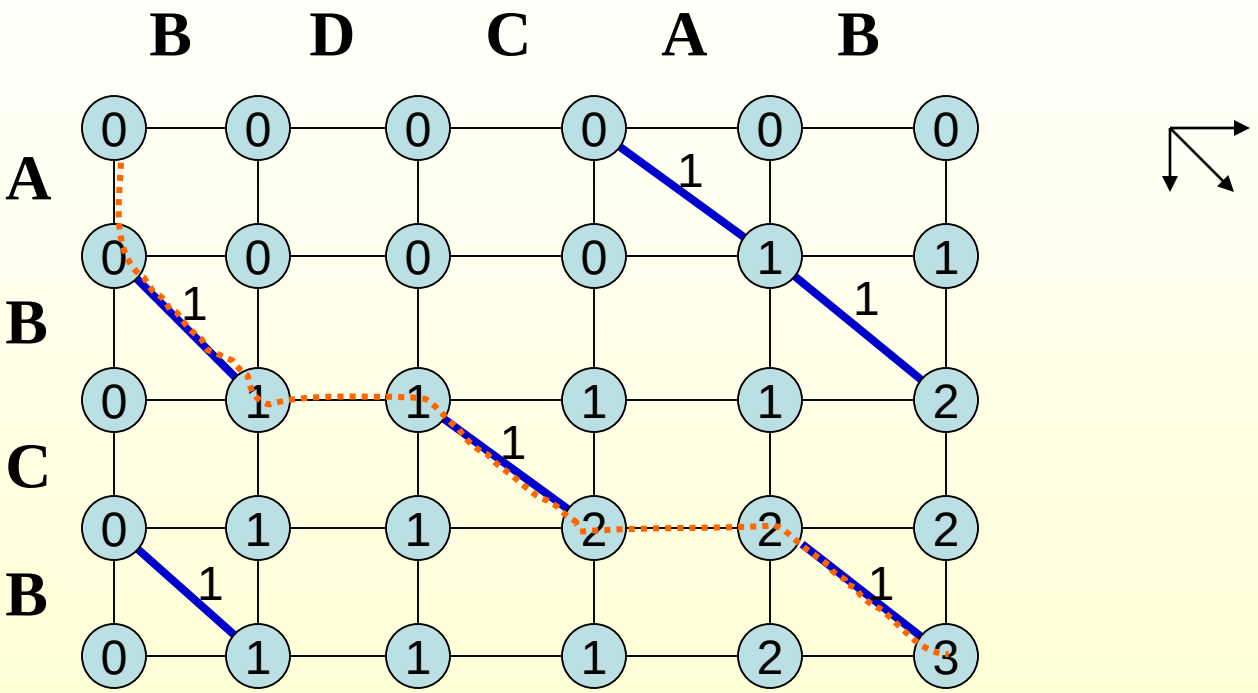
LCS (straight order): **B C B**

(this string turned out to be a palindrome)

LCS as a Longest Path Problem



LCS as a Longest Path Problem



Matrix-Chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied
  - We want to compute the product $A_1A_2\ldots A_n$
- There are many possible ways (parenthesizations) for computing the product – matrix multiplication is associative

Matrix-Chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways to group:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-Chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- How do we multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

```
1.  for i ← 1 to p
2.      for j ← 1 to r
3.          C[i, j] ← 0
4.          for k ← 1 to q
5.              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute C.
Number of scalar multiplications = *pqr*

Matrix-Chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    } Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    } Total: 75,000

Matrix-Chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i=1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ so that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i\ A_{i+1}\ \ldots\ A_j$
  - An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where$1 \le k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

…contd

- **Key observation**: parenthesizations of the subchains $A_1A_2…A_k$ and $A_{k+1}A_{k+2}…A_n$ must also be optimal if the parenthesization of the chain $A_1A_2…A_n$ is optimal (why?)

- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach

…contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach

…contd

- $A_{i..j} = (A_i A_{i+1} \cdots A_k) \cdot (A_{k+1} A_{k+2} \cdots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ *for* $i \le k < j$
- $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

Dynamic Programming Approach

…contd

- But… optimal parenthesization occurs at one value of k among all possible $i \le k < j$
- Check all these and select the best one

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

Dynamic Programming Approach

…contd

- To keep track of how to construct an optimal solution, we use a table *s*
- $s[i, j]$ = value of k at which $A_i A_{i+1} \ldots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Determine the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

MATRIX-CHAIN-ORDER($p[\,]$, n)

```
for i ← 1 to n
    m[i, i] ← 0
for l ← 2 to n
    for i ← 1 to n-l+1
            j ← i+l-1
            m[i, j] ← ∞
            for k ← i to j-1
                    q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                    if q < m[i, j]
                            m[i, j] ← q
                            s[i, j] ← k
return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
  - Each entry $s[i, j]=k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost

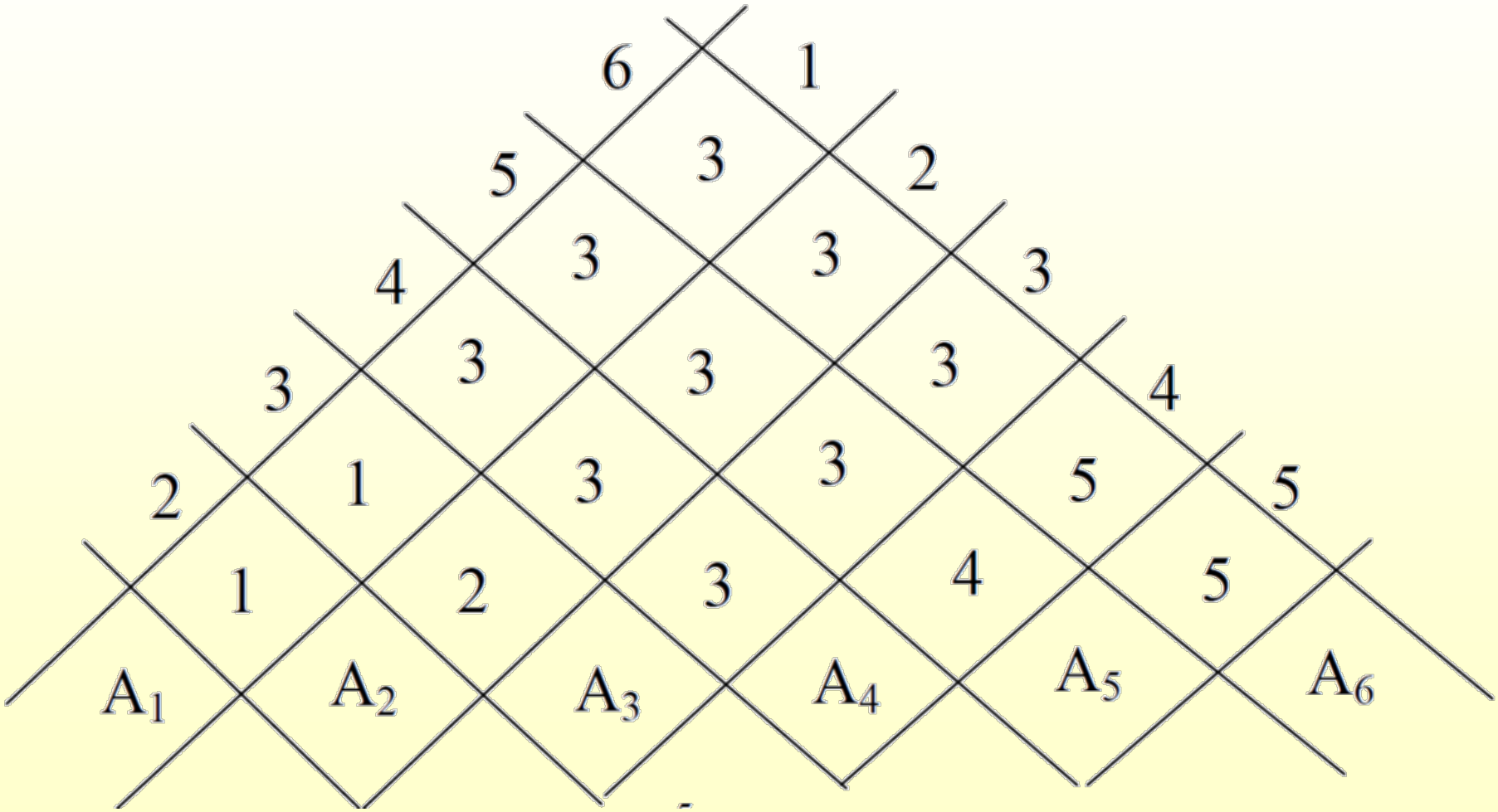
Example

- Show how to multiply this matrix chain optimally

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

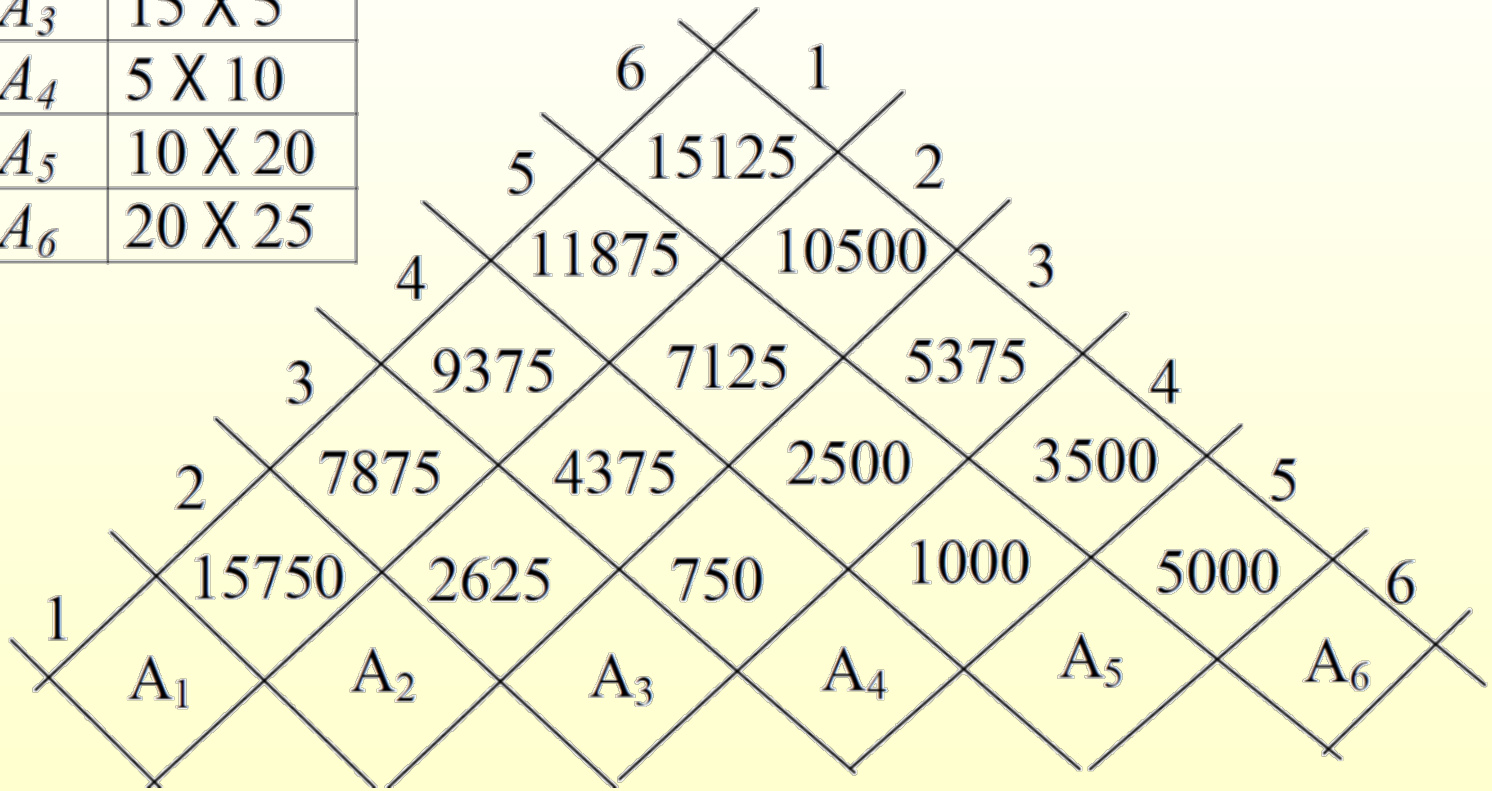
- Solution
  - Minimum cost 15,125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4\,A_5)A_6))$

Principal split points k i.e. $A_i \ldots A_j = (A_i \ldots A_k)(A_{k+1} \ldots A_j)$:



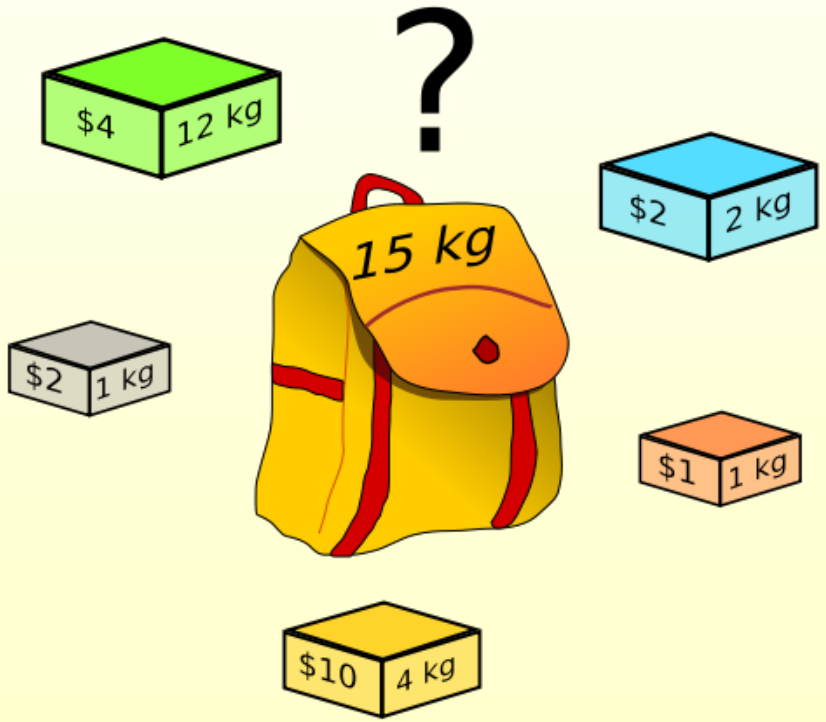
mat	dim
A_1	30 X 35
A_2	35 X 15
A_3	15 X 5
A_4	5 X 10
A_5	10 X 20
A_6	20 X 25

Here A_i is of size $p_{i-1}p_i$ the product $A_i \ldots A_k$ is of size $p_{i-1}p_k$ and the product $A_{k+1} \ldots A_j$ of size $p_{k+1}p_j$



Knapsack Problem

- Each item has a value and a weight
- Objective: maximize value
- Constraint: knapsack has a weight limitation



Three versions:

0-1 knapsack problem: take each item or leave it

Fractional knapsack problem: items are divisible

Unbounded knapsack problem: unlimited supplies of each item.

Which one is easiest to solve?

We study the 0-1 problem today.

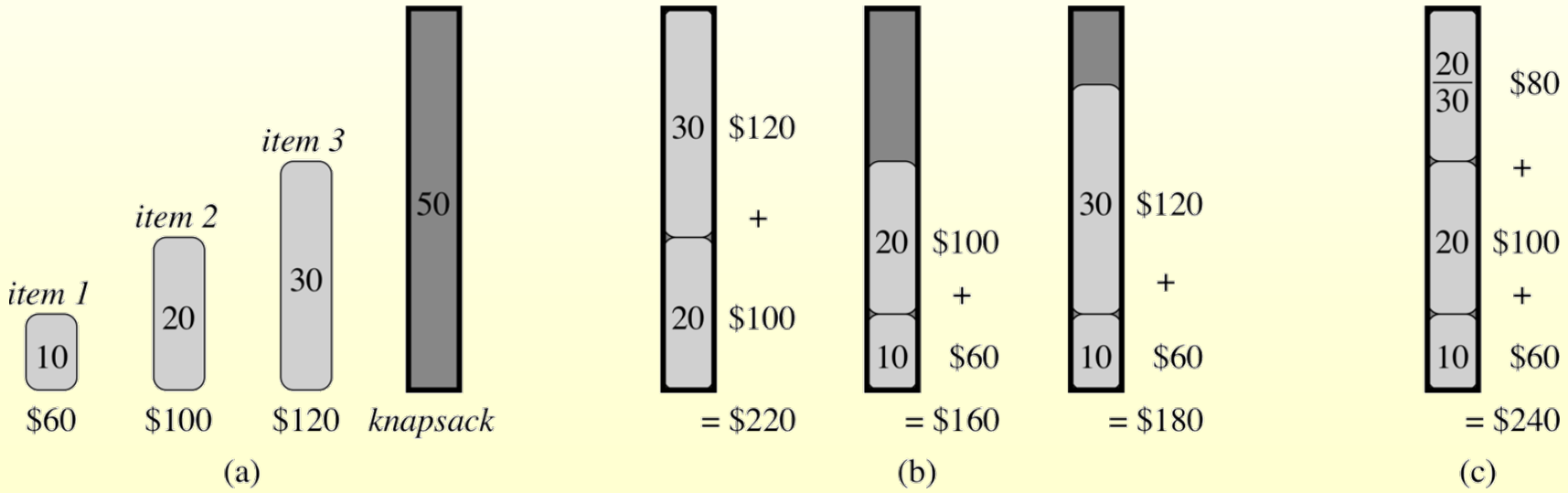
Formal Definition (0-1 problem)

- Knapsack has weight limit W
- Items labeled $1, 2, \ldots, n$ (arbitrarily)
- Items have weights $w_1, w_2, \ldots, w_n$
  - Assume all weights are integers
  - For practical reason, only consider $w_i < W$
- Items have values $v_1, v_2, \ldots, v_n$
- Objective: find a subset of items, S, such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximal among all such (feasible) subsets

Naïve Algorithms

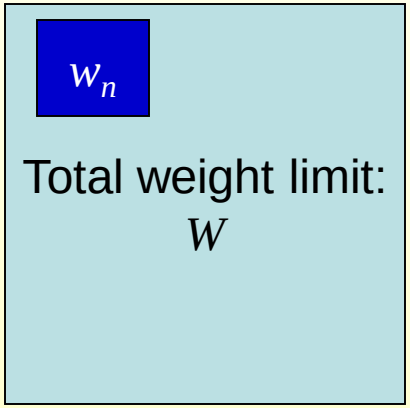
- Enumerate all subsets.
  - Optimal. But exponential time
- Greedy 1: take the item with the largest value
  - Not optimal
  - Give an example
- Greedy 2: take the item with the largest value/weight ratio
  - Not optimal
  - Give an example

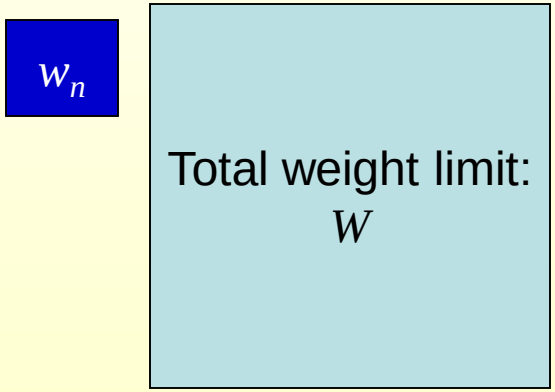
Greedy Not Always Good



A DP Algorithm

- Suppose we have found the optimal solution S
- Case 1: item n is included
- Case 2: item n is not included





Find an optimal solution using items $1, 2, \ldots, n\text{-}1$ with weight limit $W - w_n$

Find an optimal solution using items $1, 2, \ldots, n\text{-}1$ with weight limit W

Recursive Formulation

- Let $V[i, w]$ be the optimal total value when items $1, 2, \ldots, i$ are considered for a knapsack with weight limit w

  => $V[n, W]$ is the optimal solution

$$V[n, W] = max \begin{cases} V[n\text{-}1, W\text{-}w_n] + v_n \\ V[n\text{-}1, W] \end{cases}$$

Generalize

$$V[i, w] = \begin{cases} max \begin{cases} V[i\text{-}1, w\text{-}w_i] + v_i & \text{item } i \text{ is taken} \\ V[i\text{-}1, w] & \text{item } i \text{ not taken} \end{cases} \\ V[i\text{-}1, w] \quad \text{if } w_i > w & \text{item } i \text{ not taken} \end{cases}$$

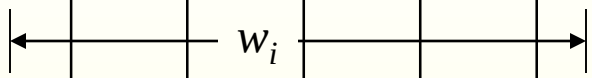
Boundary condition: $V[i, 0] = 0, V[0, w] = 0$. Number of sub-problems = ?

Example

- n = 6 (# of items)
- W = 10 (weight limit)
- Items (weight, value):

2	2
4	3
3	3
5	6
2	4
6	9

			w	0	1	2	3	4	5	6	7	8	9	10
i	w_i	v_i		0	0	0	0	0	0	0	0	0	0	0
1	2	2		0										
2	4	3		0										
3	3	3		0		$V[i-1, w-w_i]$					$V[i-1, w]$			
4	5	6		0							$V[i, w]$			
5	2	4		0										
6	6	9		0										



$$V[i, w] = \begin{cases} \max \begin{cases} V[i-1, w-w_i] + v_i & \text{item } i \text{ is taken} \\ V[i-1, w] & \text{item } i \text{ not taken} \end{cases} \\ V[i-1, w] \quad \text{if } w_i > w & \text{item } i \text{ not taken} \end{cases}$$

			w	0	1	2	3	4	5	6	7	8	9	10
i	w_i	v_i												
				0	0	0	0	0	0	0	0	0	0	0
1	2	2		0	0	2	2	2	2	2	2	2	2	2
2	4	3		0	0	2	2	3	3	5	5	5	5	5
3	3	3		0	0	2	3	3	5	5	6	6	8	8
4	5	6		0	0	2	3	3	6	6	8	9	9	11
5	2	4		0	0	4	4	6	7	7	10	10	12	13
6	6	9		0	0	4	4	6	7	9	10	13	13	15

$$V[i, w] = \begin{cases} \max \begin{cases} V[i\text{-}1, w\text{-}w_i] + v_i & \text{item } i \text{ is taken} \\ V[i\text{-}1, w] & \text{item } i \text{ not taken} \end{cases} \\ V[i\text{-}1, w] \quad \text{if } w_i > w & \text{item } i \text{ not taken} \end{cases}$$

i	w_i	v_i	w 0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0
1	2	2	0	0	2	2	2	2	2	2	2	2	2
2	4	3	0	0	2	2	3	3	5	5	5	5	5
3	3	3	0	0	2	3	3	5	5	6	6	8	8
4	5	6	0	0	2	3	3	6	6	8	9	9	11
5	2	4	0	0	4	4	6	7	7	10	10	12	13
6	6	9	0	0	4	4	6	7	9	10	13	13	15

Optimal value: 15

Item: 6, 5, 1

Weight: 6 + 2 + 2 = 10

Value: 9 + 4 + 2 = 15

Time Complexity

- Θ (nW)
- Polynomial?
  - Pseudo-polynomial
  - Works well if W is small
- Consider following items (weight, value):

  (10, 5), (15, 6), (20, 5), (18, 6)
- Weight limit 35
  - Optimal solution: item 2, 4 (value = 12). Iterate: 2^4 = 16 subsets
  - Dynamic programming: fill up a 4 x 35 = 140 table entries
- What's the problem?
  - Many entries are unused: no such weight combination
  - Top-down may be better

Use DP to Solve New Problems

- Directly map a new problem to a known problem
- Modify an algorithm for a similar task
- Design your own
  - Think about the problem recursively
  - Optimal solution to a larger problem can be computed from the optimal solution of one or more subproblems
  - These sub-problems can be solved in certain manageable order
  - Works nicely for naturally ordered data such as strings, trees, some special graphs
  - Trickier for general graphs
- The text book has some very good exercises.