

Dynamic Programming

Chapter 7, Algorithm Design

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c31>

Dynamic Programming

- ◆ Another general method for designing algorithms
 - » **A design technique, not an algorithm (like divide & conquer)**
 - » **The word “programming” is historical and predates computer programming**
- ◆ Used when problem breaks down into recurring repeated small sub-problems

dynprog - 2

Weighted Interval Scheduling

- ◆ A weighted interval is represented by $x = (s, f, v)$, where s : start time, f : finish time, and v : weight.
- ◆ The WIS problem: Given a set of weighted intervals, choose a set of non-overlapping intervals such that the total weight is maximal.
- ◆ The Greedy Approach: works when all the weights are the same.
- ◆ The Brute-Force Approach: Check for all non-overlapping subsets of the intervals.

dynprog - 3

Weighted Interval Scheduling

- ◆ Let the intervals be denoted by $S = \{x_1, x_2, \dots, x_n\}$, or simply, $S = \{1, 2, \dots, n\}$, and sorted by finish time.
- ◆ Let $p(j)$ be the largest interval before j such that i and j are disjoint. If no such intervals, $p(j) = 0$.
- ◆ Divide One Less Approach:
 - » If n is in the optimal solution X , then
 - no intervals inside $(p(n), n)$ can be in the solution.
 - X contains an optimal solution of intervals from 1 to $p(n)$.
 - » If n is not in X , then X is an optimal solution for intervals $S - \{n\}$.

dynprog - 4

A Recursive Algorithm

- ◆ Let $Opt(i)$ denote the total weights of an optimal solution of the first i intervals.
- ◆ $Opt(n) = \max(Opt(n-1), Opt(p(n))+v_n)$
- ◆ Algorithm:

```
Compute-Opt(n)
  If (n = 0) return 0
  a := Compute-Opt(p(n)) + v_n
  b := Compute-Opt(n-1)
  return max(a, b)
```

Complexity of Compute-Opt(n)?

dynprog - 5

An Improvement

- ◆ $Opt(n) = \max(Opt(n-1), Opt(p(n))+v_n)$
- ◆ Algorithm:

```
Compute-Opt(n)
  For j := 0 to n do M[j] := -1
  return Compute-Opt2(n)

Compute-Opt2(n)
  If (n = 0) return 0
  If (M[n] != -1) return M[n]
  a := Compute-Opt(p(n)) + v_n
  b := Compute-Opt(n-1)
  M[n] := max(a, b)
  return M[n]
```

Complexity of Compute-Opt(n)?

dynprog - 6

An Interactive Algorithm

- ♦ $\text{Opt}(n) = \max(\text{Opt}(n-1), \text{Opt}(p(n)) + v_n)$
- ♦ Algorithm:

```
Compute-Opt(n)
M[0] := 0
For j := 1 to n do
  a := Compute-Opt(p(j)) + v_j
  b := Compute-Opt(j-1)
  M[j] := max(a, b)
return M[n]
```

Complexity of Compute-Opt(n)?

dynprog - 7

How to Find the Solution Set

```
Find-Solution(j)
If (j > 0 and v_j + M[p(j)] >= M[j-1])
  return Find-solution(p(j)) U { j }
Else
  return Find-solution(j - 1)
```

dynprog - 8

Properties of a typical problem that can be solved with dynamic programming

- ♦ **Simple Subproblems**
 - » We should be able to break the original problem to smaller subproblems that have the same structure
- ♦ **Optimal Substructure of the problems**
 - » The solution to the problem must be a composition of subproblem solutions
- ♦ **Subproblem Overlap**
 - » Optimal subproblems to unrelated problems can contain subproblems in common

dynprog - 9

Segmented Least Squares

- ◆ Given a set P of n points in the plane, $P = \{ (x_1, y_1), \dots, (x_n, y_n) \}$, assuming $x_1 < x_2 < \dots < x_n$
- ◆ Given a line L defined by $y = ax + b$, the error of L with respect to P is
- ◆ $\text{Error}(L, P) = \sum_{i=1..n} (y_i - ax_i - b)^2$.
- ◆ How to find L to minimize $\text{Error}(L, P)$?

dynprog - 10

Knapsack Problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

dynprog - 11

Knapsack problem

There are two versions of the problem:

- (1) "0-1 knapsack problem" and
- (2) "Fractional knapsack problem"

- (1) Items are indivisible; you either take an item or not. Solved with *dynamic programming*
- (2) Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

dynprog - 12

0-1 Knapsack problem

- ◆ Given a knapsack with maximum capacity W , and a set S consisting of n items
- ◆ Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- ◆ Problem: How to pack the knapsack to achieve maximum total value of packed items?

dynprog - 13

0-1 Knapsack problem: a picture

	Weight	Benefit value
Items	w_i	b_i
	2	3
	3	4
	4	5
	5	8
	9	10

This is a knapsack
Max weight: $W = 20$

$W = 20$

dynprog - 14

0-1 Knapsack problem

- ◆ Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$
- ◆ The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.
- ◆ Just another version of this problem is the “*Fractional Knapsack Problem*”, where we can take fractions of items.

dynprog - 15

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- ◆ Since there are n items, there are 2^n possible combinations of items.
- ◆ We go through all combinations and find the one with the most total value and with total weight less or equal to W
- ◆ Running time will be $O(2^n)$

dynprog - 16

0-1 Knapsack problem: brute-force approach

- ◆ Can we do better?
- ◆ Yes, with an algorithm based on dynamic programming
- ◆ We need to carefully identify the subproblems

Let's try this:

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, \dots, k\}$

dynprog - 17

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, \dots, k\}$

- ◆ This is a valid subproblem definition.
- ◆ The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- ◆ Unfortunately, we can't do that. Explanation follows....

dynprog - 18

Defining a Subproblem

$w_1=2$	$w_2=4$	$w_3=5$	$w_4=3$	
$b_1=3$	$b_2=5$	$b_3=8$	$b_4=4$	

Max weight: $W = 20$
For S_4 :
 Total weight: 14;
 total benefit: 20

$w_1=2$	$w_2=4$	$w_3=5$	$w_5=9$
$b_1=3$	$b_2=5$	$b_3=8$	$b_5=10$

For S_5 :
 Total weight: 20
 total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4 (bracketed items 1-4)
 S_5 (bracketed items 1-5)

Solution for S_4 is not part of the solution for S_5 !!!

dynprog - 19

Defining a Subproblem

- ◆ As we have seen, the solution for S_4 is not part of the solution for S_5
- ◆ So our definition of a subproblem is flawed and we need another one!
- ◆ Let's add another parameter: w , which will represent the exact weight for each subset of items
- ◆ The subproblem then will be to compute $B[k, w]$

dynprog - 20

Recursive Formula for subproblems

- ◆ Recursive formula for subproblems:

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{else} \end{cases}$$

- ◆ It means, that the best subset of S_k that has total weight w is one of the two:
 - 1) the best subset of S_{k-1} that has total weight w ,
or
 - 2) the best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

dynprog - 21

Recursive Formula

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{else} \end{cases}$$

- ◆ The best subset of S_k that has the total weight w , either contains item k or not.
- ◆ First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- ◆ Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value

dynprog - 22

0-1 Knapsack Algorithm

```
for w := 0 to W do B[0,w] := 0
for i := 0 to n do B[i,0] := 0
  for w = 0 to W
    if w_i <= w // item i can be part of the solution
      if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w-w_i]
      else B[i,w] = B[i-1,w]
    else B[i,w] = B[i-1,w] // w_i > w
```

dynprog - 23

Running time

```
for w = 0 to W      O(W)
  B[0,w] = 0
for i = 0 to n      Repeat n times
  B[i,0] = 0
  for w = 0 to W    O(W)
    < the rest of the code >
  What is the running time of this algorithm?
  O(n*W)
  Remember that the brute-force algorithm
  takes O(2^n)
```

dynprog - 24

Example

Let's run our algorithm on the following data:

$n = 4$ (# of elements)

$W = 5$ (max weight)

Elements (weight, benefit):

(2,3), (3,4), (4,5), (5,6)

dynprog - 25

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for $w = 0$ to W
 $B[0,w] = 0$

dynprog - 26

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

for $i = 0$ to n
 $B[i,0] = 0$

dynprog - 27

Example (4)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0				
3	0				
4	0				
5	0				

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=1$
 $b_i=3$
 $w_i=2$
 $w=1$
 $w-w_i=-1$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 28

Example (5)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0				
4	0				
5	0				

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=1$
 $b_i=3$
 $w_i=2$
 $w=2$
 $w-w_i=0$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 29

Example (6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0				
5	0				

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=1$
 $b_i=3$
 $w_i=2$
 $w=3$
 $w-w_i=1$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 30

Example (7)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=1$
 $b_i=3$
 $w_i=2$
 $w=4$
 $w-w_i=2$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 31

Example (8)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=1$
 $b_i=3$
 $w_i=2$
 $w=5$
 $w-w_i=2$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 32

Example (9)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=2$
 $b_i=4$
 $w_i=3$
 $w=1$
 $w-w_i=-2$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 33

Example (10)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	→ 3		
3	0	3			
4	0	3			
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=2$
 $b_i=4$
 $w_i=3$
 $w=2$
 $w-w_i=-1$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 34

Example (11)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3			
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=2$
 $b_i=4$
 $w_i=3$
 $w=3$
 $w-w_i=0$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 35

Example (12)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3			

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=2$
 $b_i=4$
 $w_i=3$
 $w=4$
 $w-w_i=1$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 36

Example (13)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=2$
 $b_i=4$
 $w_i=3$
 $w=5$
 $w-w_i=2$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 37

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	→ 0	
2	0	3	3	→ 3	
3	0	3	4	→ 4	
4	0	3	4		
5	0	3	7		

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=3$
 $b_i=5$
 $w_i=4$
 $w=1..3$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 38

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7		

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=3$
 $b_i=5$
 $w_i=4$
 $w=4$
 $w-w_i=0$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 39

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ 7	

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=3$
 $b_i=5$
 $w_i=4$
 $w=5$
 $w-w_i=1$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 40

Example (16)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	→ 0
2	0	3	3	3	→ 3
3	0	3	4	4	→ 4
4	0	3	4	5	→ 5
5	0	3	7	7	

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=3$
 $b_i=5$
 $w_i=4$
 $w=1..4$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 41

Example (17)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7	→ 7

Items:
 1: (2,3)
 2: (3,4)
 3: (4,5)
 4: (5,6)

$i=3$
 $b_i=5$
 $w_i=4$
 $w=5$

if $w_i \leq w$ // item i can be part of the solution
 if $b_i + B[i-1, w-w_i] > B[i-1, w]$
 $B[i, w] = b_i + B[i-1, w-w_i]$
 else
 $B[i, w] = B[i-1, w]$
 else $B[i, w] = B[i-1, w]$ // $w_i > w$

dynprog - 42

Comments

- ◆ This algorithm only finds the max possible value that can be carried in the knapsack
- ◆ To know the items that make this maximum value, an addition to this algorithm is necessary

dynprog - 43

More problems

Optimal BST: Given sequence $K = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i , build a binary search tree (BST) with **minimum expected search cost**.

Matrix chain multiplication: Given a sequence of matrices $A_1 A_2 \dots A_n$, with A_i of dimension $m_i \times n_i$, insert parenthesis to minimize the total number of scalar multiplications.

Minimum convex decomposition of a polygon,
Hydrogen placement in protein structures, ...

dynprog - 44

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either **top-down** with caching or **bottom-up** in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

dynprog - 45

Optimal Binary Search Trees

♦ **Problem**

- » Given sequence $K = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i .
- » Want to build a binary search tree (BST) with **minimum expected search cost**.
- » Actual cost = # of items examined.
- » For key k_i , cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T .

dynprog - 46

Expected Search Cost

$E[\text{search cost in } T]$

$$\begin{aligned}
 &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i \\
 &= \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^n p_i \\
 &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i \quad (15.16)
 \end{aligned}$$

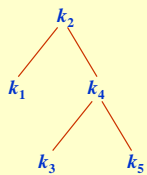
Sum of probabilities is 1.

dynprog - 47

Example

- ♦ Consider 5 keys with these search probabilities:

$$p_1 = 0.25, p_2 = 0.2, p_3 = 0.05, p_4 = 0.2, p_5 = 0.3.$$



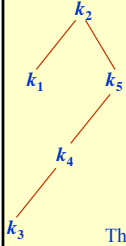
i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, $E[\text{search cost}] = 2.15$.

dynprog - 48

Example

- ♦ $p_1 = 0.25, p_2 = 0.2, p_3 = 0.05, p_4 = 0.2, p_5 = 0.3$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i)p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, $E[\text{search cost}] = 2.10$.

This tree turns out to be optimal for this set of keys.

dynprog - 49

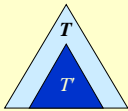
Example

- ♦ **Observations:**
 - » Optimal BST **may not** have smallest height.
 - » Optimal BST **may not** have highest-probability key at root.
- ♦ Build by exhaustive checking?
 - » Construct each n -node BST.
 - » For each, assign keys and compute expected search cost.
 - » But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

dynprog - 50

Optimal Substructure

- ♦ Any subtree of a BST contains keys in a contiguous range k_i, \dots, k_j for some $1 \leq i \leq j \leq n$.

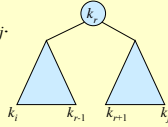


- ♦ If T is an optimal BST and T contains subtree T' with keys k_i, \dots, k_j , then T' must be an optimal BST for keys k_i, \dots, k_j .
- ♦ **Proof:** Cut and paste.

dynprog - 51

Optimal Substructure

- ♦ One of the keys in k_i, \dots, k_j , say k_r , where $i \leq r \leq j$, **must be the root** of an optimal subtree for these keys.
- ♦ Left subtree of k_r contains k_i, \dots, k_{r-1} .
- ♦ Right subtree of k_r contains k_{r+1}, \dots, k_j .



- ♦ **To find an optimal BST:**
 - » Examine all candidate roots k_r , for $i \leq r \leq j$
 - » Determine all optimal BSTs containing k_i, \dots, k_{r-1} and containing k_{r+1}, \dots, k_j

dynprog - 52

Recursive Solution

- ♦ Find optimal BST for k_i, \dots, k_j , where $i \geq 1, j \leq n, j \geq i-1$.
When $j = i-1$, the tree is empty.
- ♦ Define $e[i, j]$ = **expected search cost of optimal BST for k_i, \dots, k_j** .
- ♦ If $j = i-1$, then $e[i, j] = 0$.
- ♦ If $j \geq i$,
 - » Select a root k_r , for some $i \leq r \leq j$.
 - » Recursively make an optimal BSTs
 - for k_i, \dots, k_{r-1} as the left subtree, and
 - for k_{r+1}, \dots, k_j as the right subtree.

dynprog - 53

Recursive Solution

- ♦ When the OPT subtree becomes a subtree of a node:
 - » Depth of every node in OPT subtree goes up by 1.
 - » Expected search cost increases by

$$w(i, j) = \sum_{l=i}^j p_l \quad \text{from (15.16)}$$

- ♦ If k_r is the root of an optimal BST for k_i, \dots, k_j :
 - » $e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$
 $= e[i, r-1] + e[r+1, j] + w(i, j)$. (because $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$)
- ♦ But, we don't know which one is k_r . Hence,

$$e[i, j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \end{cases}$$

dynprog - 54

Computing an Optimal Solution

For each subproblem (i, j) , store:

- ♦ expected search cost in a table $e[1..n+1, 0..n]$
 - » Will use only entries $e[i, j]$, where $j \geq i-1$.
- ♦ $root[i, j]$ = root of subtree with keys k_p, \dots, k_j , for $1 \leq i \leq j \leq n$.
- ♦ $w[1..n+1, 0..n]$ = sum of probabilities
 - » $w[i, i-1] = 0$ for $1 \leq i \leq n$.
 - » $w[i, j] = w[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$.

dynprog - 55

Pseudo-code

OPTIMAL-BST(p, q, n)

```

1. for  $i \leftarrow 1$  to  $n+1$ 
2.   do  $e[i, i-1] \leftarrow 0$ 
3.   do  $w[i, i-1] \leftarrow 0$ 
4. for  $l \leftarrow 1$  to  $n$ 
5.   do for  $i \leftarrow 1$  to  $n-l+1$ 
6.     do  $j \leftarrow i+l-1$ 
7.       do  $e[i, j] \leftarrow \infty$ 
8.         do  $w[i, j] \leftarrow w[i, j-1] + p_j$ 
9.           for  $r \leftarrow i$  to  $j$ 
10.            do  $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$ 
11.              if  $t < e[i, j]$ 
12.                then  $e[i, j] \leftarrow t$ 
13.                  do  $root[i, j] \leftarrow r$ 
14. return  $e$  and  $root$ 
    
```

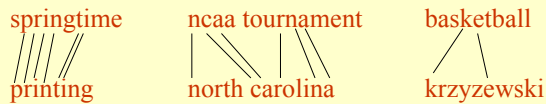
Consider all trees with l keys.
 Fix the first key.
 Fix the last key.
 Determine the root of the optimal (sub)tree

Time: $O(n^3)$

dynprog - 56

Longest Common Subsequence (LCS)

♦ **Problem:** Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, find a common subsequence whose length is maximum.



Subsequence need not be consecutive, but must be in order.

dynprog - 57

Longest Common Subsequence (LCS)

- ◆ Given two sequences $x[1..m]$ and $y[1..n]$, find the longest subsequence which occurs in both
- ◆ Ex: $x = \text{“A B C B D A B”}$, $y = \text{“B D C A B A”}$
- ◆ “B C” and “A A” are both subsequences of both
What is the LCS?
- ◆ Brute-force algorithm: For every subsequence of x , check if it’s a subsequence of y .
 - » *How many subsequences of x are there?*
 - » *What will be the running time of the brute-force algorithm?*

dynprog - 58

LCS Algorithm

- ◆ Brute-force algorithm: 2^m subsequences of x to check against n elements of y : $O(n 2^m)$
- ◆ We can do better: For now, let’s only worry about the problem of finding the *length* of LCS
 - » When finished we will see how to backtrack from this solution back to the actual LCS
- ◆ Using “Divide One Less”

dynprog - 59

Finding LCS Length

- ◆ Define $c[i,j]$ to be the length of the LCS of $X_i = x[1..i]$ and $Y_j = y[1..j]$
 - » *What is the length of LCS of x and y ?*

- ◆ Theorem:

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

- ◆ *What is this really saying?*

dynprog - 60

LCS Length Algorithm 1

LCS-Length(m, n)

1. If (m = 0 or n = 0) **return** 0
2. If (A[m] = A[n])
 return LCS-Length(m-1, n-1)+1
3. a := LCS-Length(m, n-1);
4. b := LCS-Length(m-1, n);
5. **return** max(a, b);

What's the complexity?

dynprog - 61

LCS Length Algorithm 1

LCS-Length(m, n)

For i := 0 to m For j := 0 to n do C[i,j] := -1
return LCS-Length2(m, n)

LCS-Length2(i, j)

1. If (i = 0 or j = 0) **return** 0
2. If C[i,j] != -1 **return** C[i, j];
3. If (A[i] = A[j]) C[i,j] := LCS-Length(i-1, j-1)+1
4. else a := LCS-Length2(i, j-1);
5. b := LCS-Length2(i-1, j);
6. C[i,j] := max(a, b);
7. **return** C[i,j]

What's the complexity?

dynprog - 62

LCS Example

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

- ♦ X = ABCB
- ♦ Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB
X = A **B** C **B**
Y = **B** D C A **B**

dynprog - 63

Properties of a typical problem that can be solved with dynamic programming

- ♦ **Simple Subproblems**
 - » We should be able to break the original problem to smaller subproblems that have the same structure
- ♦ **Optimal Substructure of the problems**
 - » The solution to the problem must be a composition of subproblem solutions
- ♦ **Subproblem Overlap**
 - » Optimal subproblems to unrelated problems can contain subproblems in common

dynprog - 64

LCS Length 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: Y0
4. for j := 1 to n c[0,j] := 0 // special case: X0
5. for i := 1 to m // for all Xi
6.   for j := 1 to n // for all Yj
7.     if ( Xi = Yj )
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
```

dynprog - 65

LCS Example (0)

		<div style="display: flex; justify-content: space-between; align-items: center;"> ABCB BDCAB </div>						
		j	0	1	2	3	4	5
		Y _j	B	D	C	A	B	
i	X _i							
0								
1	A							
2	B							
3	C							
4	B							

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

dynprog - 66

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i	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

dynprog - 67

LCS Example (2)

ABCB
BDCAB

	j	0	1	2	3	4	5
i	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])

dynprog - 68

LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i	Y _j	B	D	C	A	B	
0	X _i	0	0	0	0	0	0
1	A	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])

dynprog - 69

LCS Example (4)

ABCB
BDCAB

j	0	1	2	3	4	5
	Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0
1	A	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])

dynprog - 70

LCS Example (5)

ABCB
BDCAB

j	0	1	2	3	4	5
	Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0
1	A	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])

dynprog - 71

LCS Example (6)

ABCB
BDCAB

j	0	1	2	3	4	5
	Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0
1	A	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])

dynprog - 72

LCS Example (7)

ABCB
BDCAB

j	0	1	2	3	4	5
Y _j	B	B	D	C	A	B
i	X _i	0	0	0	0	0
0	A	0	0	0	0	1
2	B	0	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])

dynprog - 73

LCS Example (8)

ABCB
BDCAB

j	0	1	2	3	4	5
Y _j	B	D	C	A	B	B
i	X _i	0	0	0	0	0
0	A	0	0	0	1	1
2	B	0	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])

dynprog - 74

LCS Example (10)

ABCB
BDCAB

j	0	1	2	3	4	5
Y _j	B	D	C	A	B	
i	X _i	0	0	0	0	0
0	A	0	0	0	1	1
2	B	0	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])

dynprog - 75

LCS Example (11) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

dynprog - 76

LCS Example (12) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

dynprog - 77

LCS Example (13) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

dynprog - 78

LCS Example (14)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	1	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	2

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

dynprog - 79

LCS Example (15)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	1	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	3

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

dynprog - 80

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

dynprog - 81

How to Find Actual LCS

- ◆ So far, we have just found the *length* of LCS, but not LCS itself.
- ◆ We want to modify this algorithm to make it output LCS of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

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

dynprog - 82

How to find actual LCS

- ◆ Remember that

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$
- ◆ So we can start from $c[m,n]$ and go backwards
- ◆ Whenever $c[i,j] = c[i-1, j-1] + 1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- ◆ When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

dynprog - 83

Finding LCS

		j					
		0	1	2	3	4	5
		Yj	B	D	C	A	B
i	Xi	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

dynprog - 84

Finding LCS (2)

	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
i	Xi	0	0	0	0	0	0
0	A	0	0	0	0	1	1
1	B	0	1	1	1	1	2
2	C	0	1	1	2	2	2
3	B	0	1	1	2	2	3
4							

LCS (reversed order): **B C B**
 LCS (straight order): **B C B**
 (this string turned out to be a palindrome)

dynprog - 85

Other sequence questions

- ♦ **Edit distance:** Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, what is the minimum number of deletions, insertions, and changes that you must do to change one to another?
- ♦ **Protein sequence alignment:** Given a score matrix on amino acid pairs, $s(a,b)$ for $a,b \in \{\Lambda\} \cup A$, and 2 amino acid sequences, $X = \langle x_1, \dots, x_m \rangle \in A^m$ and $Y = \langle y_1, \dots, y_n \rangle \in A^n$, find the alignment with lowest score...

dynprog - 86

Matrix Chain-Products

- ♦ Review: Matrix Multiplication.
 - » $C = A * B$
 - » A is $d \times e$ and B is $e \times f$
 - » $O(def)$ time

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] * B[k, j]$$

dynprog - 87

Matrix Chain-Products

- ♦ **Matrix Chain-Product:**
 - » Compute $A=A_0*A_1*...*A_{n-1}$
 - » A_i is $d_i \times d_{i+1}$
 - » Problem: How to parenthesize?
- ♦ Example
 - » B is 3×100
 - » C is 100×5
 - » D is 5×5
 - » $(B*C)*D$ takes $1500 + 75 = 1575$ ops
 - » $B*(C*D)$ takes $1500 + 2500 = 4000$ ops

dynprog - 88

An Enumeration Approach

- ♦ **Matrix Chain-Product Alg.:**
 - » Try all possible ways to parenthesize
 $A=A_0*A_1*...*A_{n-1}$
 - » Calculate number of ops for each one
 - » Pick the one that is best
- ♦ Running time:
 - » The number of parenthesizations is equal to the number of binary trees with n leaf nodes (Why?)
 - » This is **exponential!**
 - » It is called the Catalan number, and it is almost 4^n .
 - » This is a terrible algorithm!

dynprog - 89

A Greedy Approach

- ♦ Idea #1: repeatedly select the product that uses (up) the most operations.
- ♦ Counter-example:
 - » A is 10×5
 - » B is 5×10
 - » C is 10×5
 - » D is 5×10
 - » Greedy idea #1 gives $(A*B)*(C*D)$, which takes $500+1000+500 = 2000$ ops
 - » $A*((B*C)*D)$ takes $500+250+250 = 1000$ ops

dynprog - 90

Another Greedy Approach

- ♦ Idea #2: repeatedly select the product that uses the fewest operations.
- ♦ Counter-example:
 - » A is 101×11
 - » B is 11×9
 - » C is 9×100
 - » D is 100×99
 - » Greedy idea #2 gives $A*((B*C)*D)$, which takes $109989+9900+108900=228789$ ops
 - » $(A*B)*(C*D)$ takes $9999+89991+89100=189090$ ops
- ♦ The greedy approach is not giving us the optimal value.

dynprog - 91

A "Recursive" Approach

- ♦ Define **subproblems**:
 - » Find the best parenthesization of $A_i * A_{i+1} * \dots * A_j$.
 - » Let $N_{i,j}$ denote the number of operations done by this subproblem.
 - » The optimal solution for the whole problem is $N_{0,n-1}$.
- ♦ **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
 - » There has to be a final multiplication (root of the expression tree) for the optimal solution.
 - » Say, the final multiply is at index i : $(A_0 * \dots * A_i) * (A_{i+1} * \dots * A_{n-1})$.
 - » Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
 - » If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

dynprog - 92

A Characterizing Equation

- ♦ The global optimal has to be defined in terms of optimal subproblems, depending on the position of the final multiply.
- ♦ Let us consider all possible places for that final multiply:
 - » Assume that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
 - » So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- ♦ Note that subproblems are not independent--the **subproblems overlap**.

dynprog - 93

A Recursive Algorithm

- Create a recursive procedure from the recursive formula.
- Because of overlapping subproblems, the complexity is high.
- Running time: $\Omega(2^n)$

Algorithm *matrixChain(S)*
Input: sequence S of n matrices to be multiplied
Output: number of operations in an optimal parenthesization of S
Return `matrixChainRecur(S, 1, n)`;

Procedure *matrixChainRecur(S, i, j)*
 If $(i = j)$ return 0
 $N_{ij} \leftarrow +\infty$
for $k \leftarrow i$ **to** $j-1$ **do**
 $N_{ik} \leftarrow \text{matrixChainRecur}(S, i, k)$
 $N_{k+1,j} \leftarrow \text{matrixChainRecur}(k+1, j)$
 $N_{ij} \leftarrow \min\{N_{ij}, N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$
return N_{ij}

dynprog - 94

A Recursive Algorithm with Memory

- Each N_{ij} needs to be computed once.
- If we store them, then we need only constant time to lookup.
- Running time: $O(n^3)$

Algorithm *matrixChain(S)*
for $i \leftarrow 1$ **to** $n-1$ **do**
 for $j \leftarrow i+1$ **to** n **do** $N_{ij} \leftarrow +\infty$
Return `matrixChainRecurMemo(S, 1, n)`;

Procedure *matrixChainRecurMemo(S, i, j)*
 If $(i = j)$ return 0
 If $N_{ij} < +\infty$ return N_{ij}
for $k \leftarrow i$ **to** $j-1$ **do**
 $N_{ik} \leftarrow \text{matrixChainRecurMemo}(S, i, k)$
 $N_{k+1,j} \leftarrow \text{matrixChainRecurMemo}(k+1, j)$
 $N_{ij} \leftarrow \min\{N_{ij}, N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$
return N_{ij}

dynprog - 95

A Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,... subproblems, and so on.
- Running time: $O(n^3)$

Algorithm *matrixChain(S)*:
Input: sequence S of n matrices to be multiplied
Output: number of operations in an optimal parenthesization of S

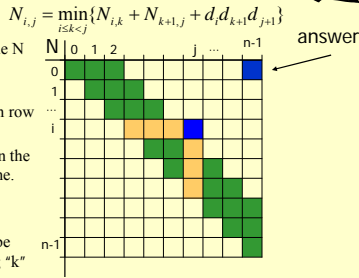
for $i \leftarrow 1$ **to** $n-1$ **do**
 $N_{ii} \leftarrow 0$
for $b \leftarrow 1$ **to** $n-1$ **do**
 for $i \leftarrow 0$ **to** $n-b-1$ **do**
 $j \leftarrow i+b$
 $N_{ij} \leftarrow +\infty$
 for $k \leftarrow i$ **to** $j-1$ **do**
 $N_{ij} \leftarrow \min\{N_{ij}, N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

dynprog - 96

A Dynamic Programming Algorithm Visualization



- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes $O(n)$ time.
- Total run time: $O(n^2)$
- Getting actual parenthesization can be done by remembering "k" for each N entry



dynprog - 97

Examples of Dynamic Programming

» 0-1 Knapsack

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{else} \end{cases}$$

» Longest common subsequences:

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

» Optimal binary search tree:

$$e[i, j] = \begin{cases} 0 & \text{if } j = i - 1 \\ \min_{i \leq r < j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \end{cases}$$

» Matrix chain multiplication

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

dynprog - 98

Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

dynprog - 99

Optimal Substructure

- ♦ Optimal substructure varies across problem domains:
 - » 1. *How many subproblems* are used in an optimal solution.
 - » 2. *How many choices* in determining which subproblem(s) to use.
- ♦ Informally, running time depends on (# of subproblems overall) × (# of choices).
- ♦ How many subproblems and choices do the examples considered contain?
- ♦ Dynamic programming uses optimal substructure **bottom up**.
 - » *First* find optimal solutions to subproblems.
 - » *Then* choose which to use in optimal solution to the problem.

dynprog - 100

Optimal Substructure

- ♦ Does optimal substructure apply to all optimization problems? **No**.
- ♦ Applies to determining the **shortest path** but **NOT** the **longest simple path** of an unweighted directed graph.
- ♦ Why?
 - » **Shortest path has independent subproblems.**
 - » Solution to one subproblem does not affect solution to another subproblem of the same problem.
 - » **Subproblems are not independent in longest simple path.**
 - Solution to one subproblem affects the solutions to other subproblems.
 - » **Example:**

dynprog - 101

Overlapping Subproblems

- ♦ The space of subproblems must be “small”.
- ♦ The **total number of distinct subproblems is a polynomial in the input size**.
 - » A recursive algorithm is exponential because it solves the same problems repeatedly.
 - » If divide-and-conquer is applicable, then each problem solved will be brand new.

dynprog - 102

Conclusion

- ♦ Dynamic programming is a useful technique of solving certain kind of problems
- ♦ When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- ♦ Running time (Dynamic Programming algorithm vs. naïve algorithm):
 - » LCS: $O(m*n)$ vs. $O(n * 2^m)$
 - » 0-1 Knapsack problem: $O(W*n)$ vs. $O(2^n)$

dynprog - 103

Summary: Dynamic programming

- ♦ DP is a method for solving certain kind of problems
- ♦ DP can be applied when the solution of a problem includes solutions to subproblems
- ♦ We need to find a recursive formula for the solution
- ♦ We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- ♦ In the end we'll get the solution of the whole problem

dynprog - 104
