Dynamic Programming

Dynamic Programming

- An advanced strategy for designing algorithms is *dynamic programming*
  - A meta-technique, not an algorithm (like divide & conquer)
  - The word “programming” is historical and predates computer programming
- Use when problem breaks down into recurring small sub-problems

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than “*brute-force methods*”, which solve the same subproblems over and over again

Basic Steps

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

Rod cutting

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

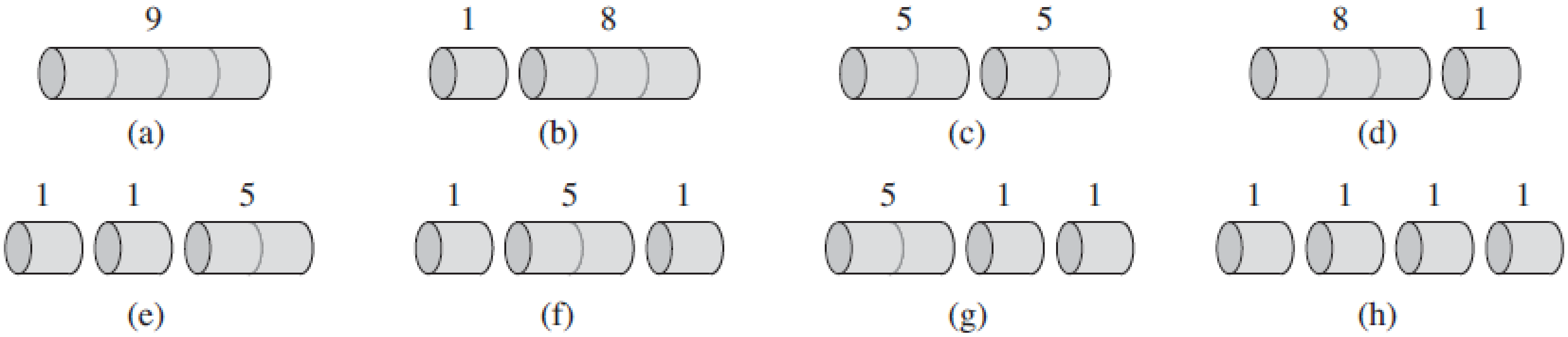


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Rod cutting

an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} \; .$$

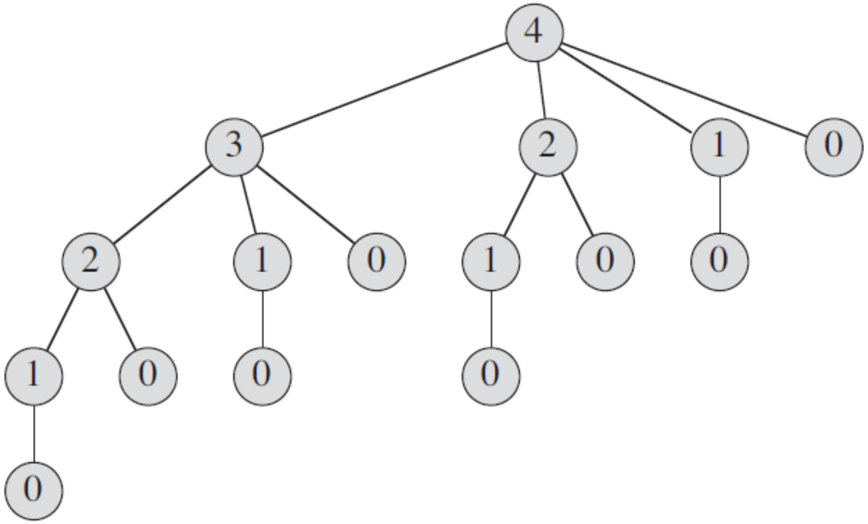
optimal substructure : recurrence function

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right)$$

$$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i}\right)$$

Recursive top-down implementation

```
CUT-ROD$(p, n)$
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```



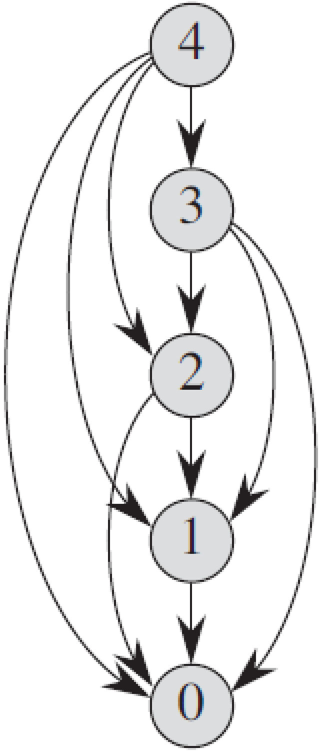
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n) = 2^n$$

Top-down with memoization

```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$
$r[n] = q$
return $q$
```



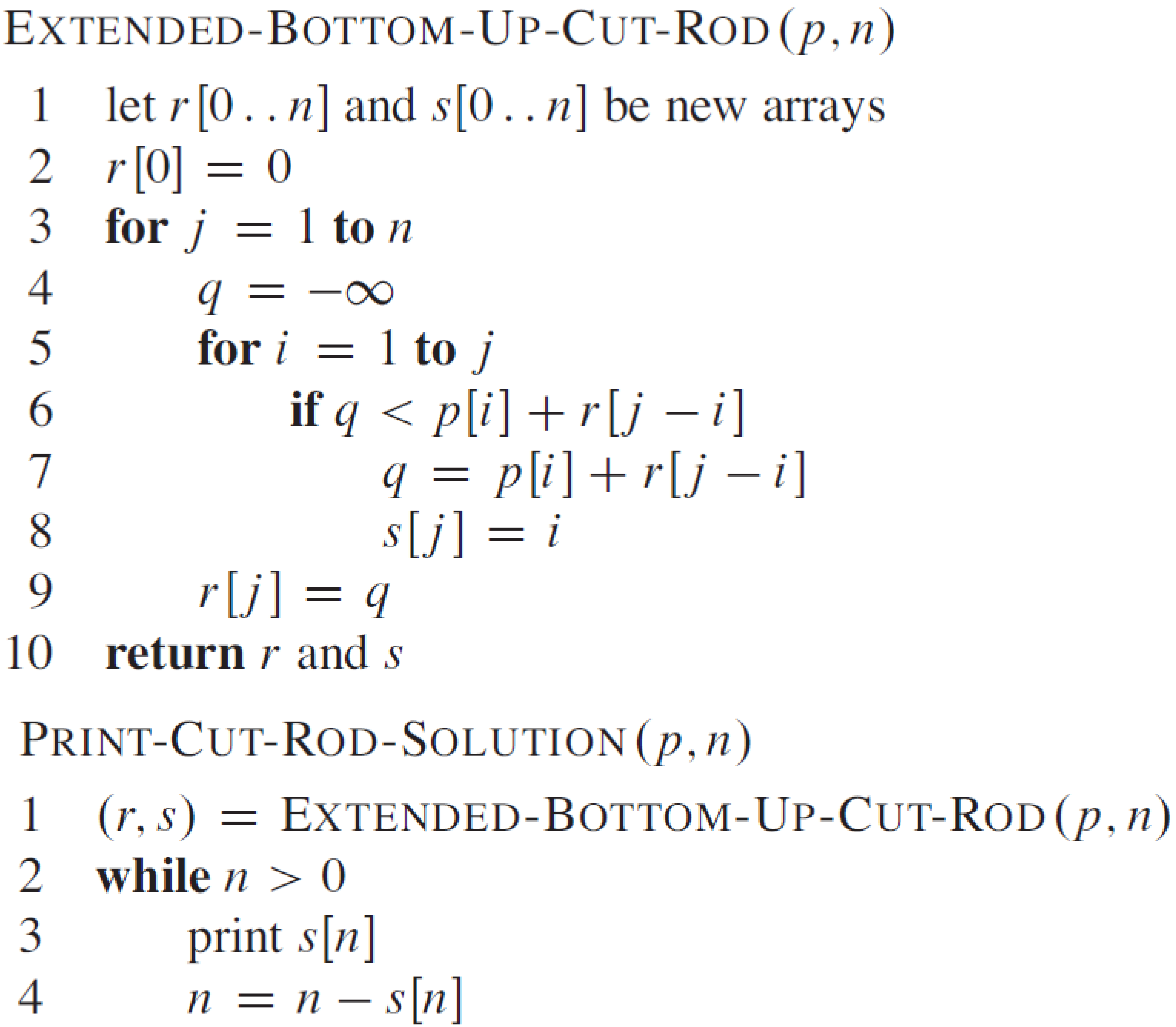
Bottom-up method

```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Bottom-up method: Reconstructing a solution



```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

Matrix-Chain Multiplication

1.The structure of an optimal parenthesization

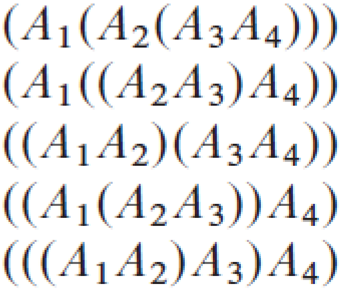
- Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$A_1 \quad \cdot \quad A_2 \quad \cdots \quad A_i \quad \cdot \quad A_{i+1} \quad \cdots \quad A_n$$

$$p_0 \times p_1 \quad p_1 \times p_2 \qquad p_{i-1} \times p_i \quad p_i \times p_{i+1} \qquad p_{n-1} \times p_n$$

- Cost for multiplying two matrices

$$\text{cost}[A \cdot B] = \text{row}_A \, \text{col}_A \, \text{col}_B$$



$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i \, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$p_{i-1}p_kp_j$

$$= A_{i\ldots k} \, A_{k+1\ldots j} \qquad \text{for } i \le k < j$$

m[i, k] m[k+1,j]

- m[i, j] = the minimum number of multiplications needed to compute the product $A_{i\ldots j}$

$$m[i, j] = \underbrace{m[i, k]}_{\substack{\text{min \# of multiplications}\\ \text{to compute } A_{i\ldots k}}} + \underbrace{m[k+1, j]}_{\substack{\text{min \# of multiplications}\\ \text{to compute } A_{k+1\ldots j}}} + \underbrace{p_{i-1}p_kp_j}_{\substack{\text{\# of multiplications}\\ \text{to compute } A_{i\ldots k}A_{k\ldots j}}}$$

3. Compute the Optimal Costs

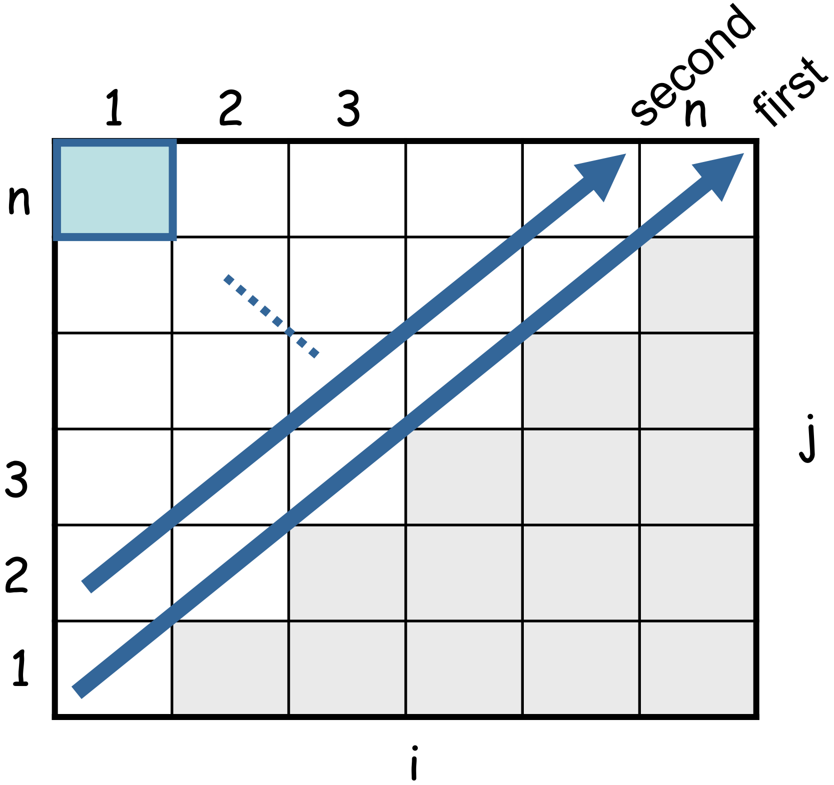
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

m[1..n, 1..n]

- auxiliary table for storing m[i, j]

s[1..n, 1..n]

- auxiliary table that helps reconstruct the optimal solution



4. Construct the Optimal Solution

- Store the optimal choice made at each subproblem
- $s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}
- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = $s[1, n]$

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization

4. Construct the Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

$$A_{1..n} = A_{1..s[1,\,n]} \cdot A_{s[1,\,n]+1..n}$$

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\ A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\ A_{5..6}$

4. Construct the Optimal Solution (cont.)

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A_i"
   else print "("
         PRINT-OPT-PARENS(s, i, s[i, j])
         PRINT-OPT-PARENS(s, s[i, j] + 1, j)
         print ")"
```

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

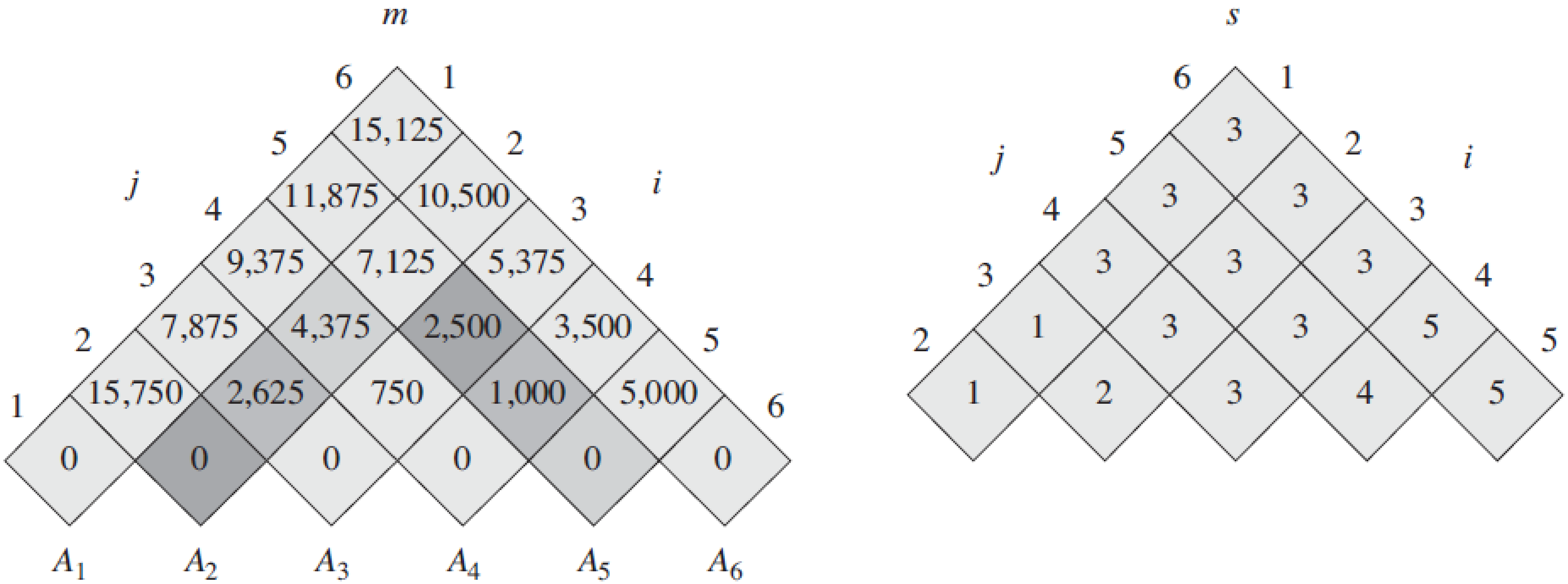


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

Example: $A_1 \cdot \cdot \cdot A_6$ $((A_1(A_2A_3))((A_4A_5)A_6))$

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

s[1..6, 1..6]

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

P-O-P(s, 1, 6)   s[1, 6] = 3

i = 1, j = 6   "("   P-O-P (s, 1, 3)   s[1, 3] = 1

i = 1, j = 3   "("   P-O-P(s, 1, 1)   ⇒ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3   "("   P-O-P (s, 2, 2) ⇒ "A_2"

P-O-P (s, 3, 3) ⇒ "A_3"

")"

")"   …

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

Memoized Matrix-Chain

```
Alg.: MEMOIZED-MATRIX-CHAIN(p)
    n ← length[p] - 1
    for i ← 1 to n
        do for j ← i to n
            do m[i, j] ← ∞
    return LOOKUP-CHAIN(p, 1, n)
```

Initialize the m table with large values that indicate whether the values of m[i, j] have been computed

Top-down approach

Memoized Matrix-Chain

Running time is $O(n^3)$

```
Alg.: LOOKUP-CHAIN(p, i, j)
if m[i, j] < ∞
    then return m[i, j]
if i = j
    then m[i, j] ← 0
    else for k ← i to j - 1
            do q ← LOOKUP-CHAIN(p, i, k) +
                   LOOKUP-CHAIN(p, k+1, j) + $p_{i-1}p_kp_j$
               if q < m[i, j]
                  then m[i, j] ← q
return m[i, j]
```

Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - Some subproblems do not need to be solved

Matrix-Chain Multiplication - Summary

- Both the dynamic programming approach and the memoized algorithm can solve the matrix-chain multiplication problem in $O(n^3)$
- Both methods take advantage of the overlapping subproblems property
- There are only $\Theta(n^2)$ different subproblems
  - Solutions to these problems are computed only once
- Without memoization the natural recursive algorithm runs in exponential time

Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Optimal Substructure - Examples

- Assembly line
  - Fastest way of going through a station j contains:

    the fastest way of going through station $j-1$ on either line
- Matrix multiplication
  - Optimal parenthesization of $A_i \cdot A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains:

    an optimal solution to the problem of parenthesizing $A_{i..k}$ and $A_{k+1..j}$

Discovering Optimal Substructure

1. Show that a solution to a problem consists of making a choice that leaves one or more similar problems to be solved
2. Suppose that for a given problem you are given the choice that leads to an optimal solution
3. Given this choice determine which subproblems result
4. Show that the solutions to the subproblems used within the optimal solution must themselves be optimal
   - Cut-and-paste approach

Parameters of Optimal Substructure

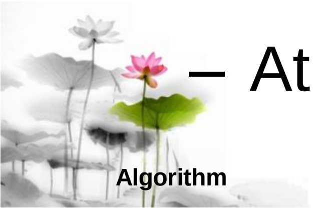
- How many subproblems are used in an optimal solution for the original problem
  - Assembly line: One subproblem (the line that gives best time)
  - Matrix multiplication: Two subproblems (subproducts $A_{i..k}$, $A_{k+1..j}$)
- How many choices we have in determining which subproblems to use in an optimal solution
  - Assembly line: Two choices (line 1 or line 2)
  - Matrix multiplication: $j - i$ choices for k (splitting the product)

Parameters of Optimal Substructure

- Intuitively, the running time of a dynamic programming algorithm depends on two factors:
  - Number of subproblems overall
  - How many choices we look at for each subproblem
- Assembly line
  - $\Theta(n)$ subproblems (n stations)
  - 2 choices for each subproblem

  $\Theta(n)$ overall
- Matrix multiplication:
  - $\Theta(n^2)$ subproblems ($1 \leq i \leq j \leq n$)
  - At most $n-1$ choices

  $\Theta(n^3)$ overall



Longest Common Subsequence

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Ex: x = {A B C B D A B }, y = {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 alg?*

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
- Notice LCS problem has optimal substructure
  - Subproblems: LCS of pairs of *prefixes* of x and y

Finding LCS Length

- Define c[*i*,*j*] to be the length of the LCS of x[1..*i*] and 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?*

LCS Instance

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

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

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

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if |X| = m, |Y| = n, then there are 2^m subsequences of x; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is O(n 2^m)
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: “find LCS of pairs of *prefixes* of X and Y”

LCS Algorithm

- First we’ll find the length of LCS. Later we’ll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

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

LCS recursive solution

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

- We start with *i = j = 0* (empty substrings of x and y)
- Since X$_{0}$ and Y$_{0}$ are empty strings, their LCS is always empty (i.e. *c[0,0] = 0*)
- LCS of empty string and any other string is empty, so for every i and j: *c[0, j] = c[i,0] = 0*

LCS recursive solution

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

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

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

- **Second case:** *x[i] != y[j]*
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

LCS Length Algorithm

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n = length(Y)  // get the # of symbols in Y
for i = 1 to m      c[i,0] = 0      // special case: $Y_0$
for j = 1 to n      c[0,j] = 0      // special case: $X_0$
for i = 1 to m                      // for all $X_i$
    for j = 1 to n                      // for all $Y_j$
        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] )
return c
```

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**

LCS Example (0)

ABCB
BDCAB

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

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

LCS Example (1)

ABCB
BDCAB

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

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

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

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

j		0	1	2	3	4	5
i		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					
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

	j	0	1	2	3	4	5
i		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				
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

	j	0	1	2	3	4	5
i		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	
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

	j	0	1	2	3	4	5
i		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**					
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

	j	0	1	2	3	4	5
i		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**			
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

	j	0	1	2	3	4	5
i		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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		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] )
```

LCS Example (13)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

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	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 (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		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**	**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)

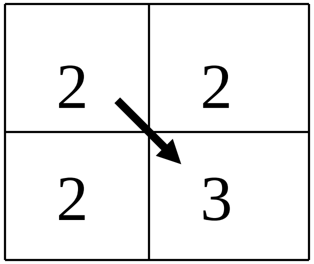
since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence 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

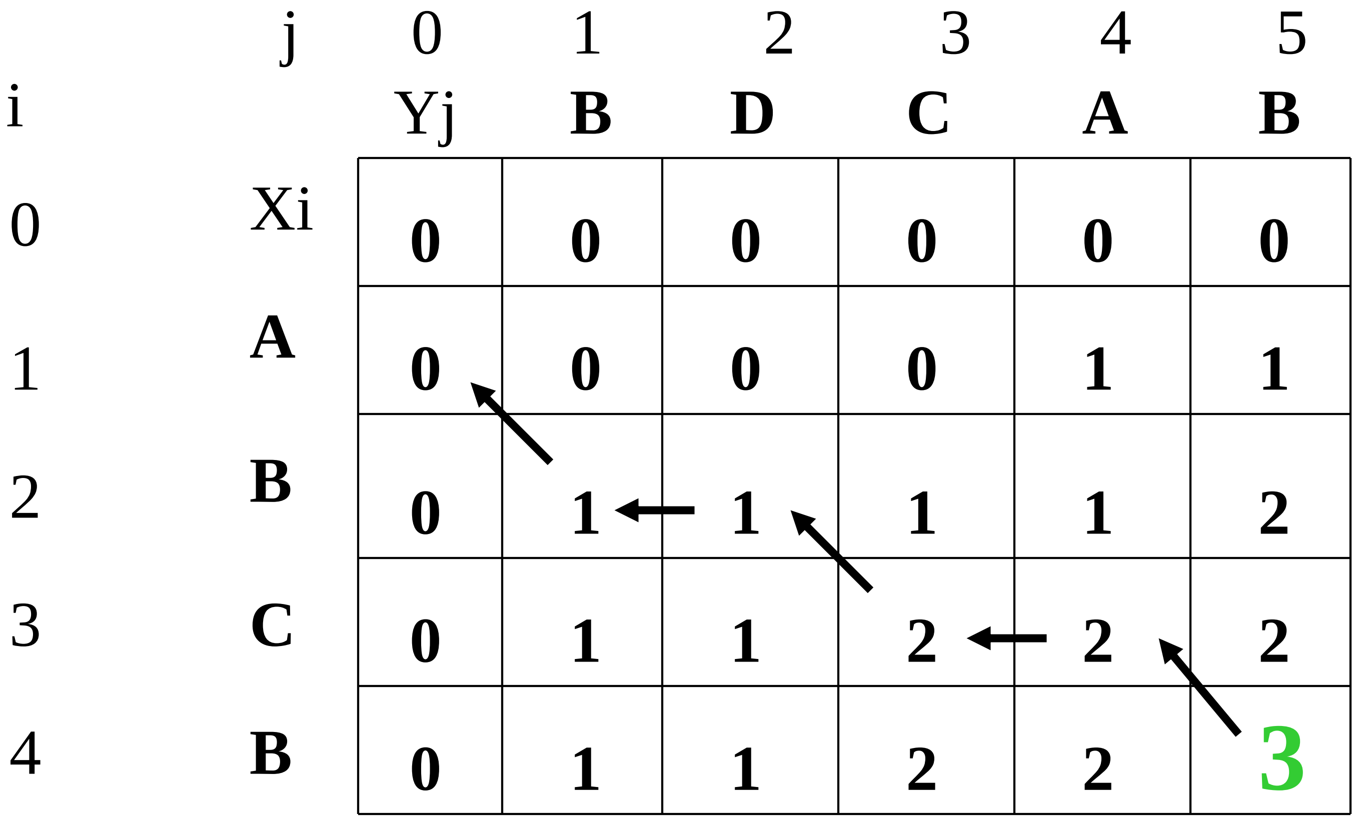
How to find actual LCS - continued

- Remember that

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

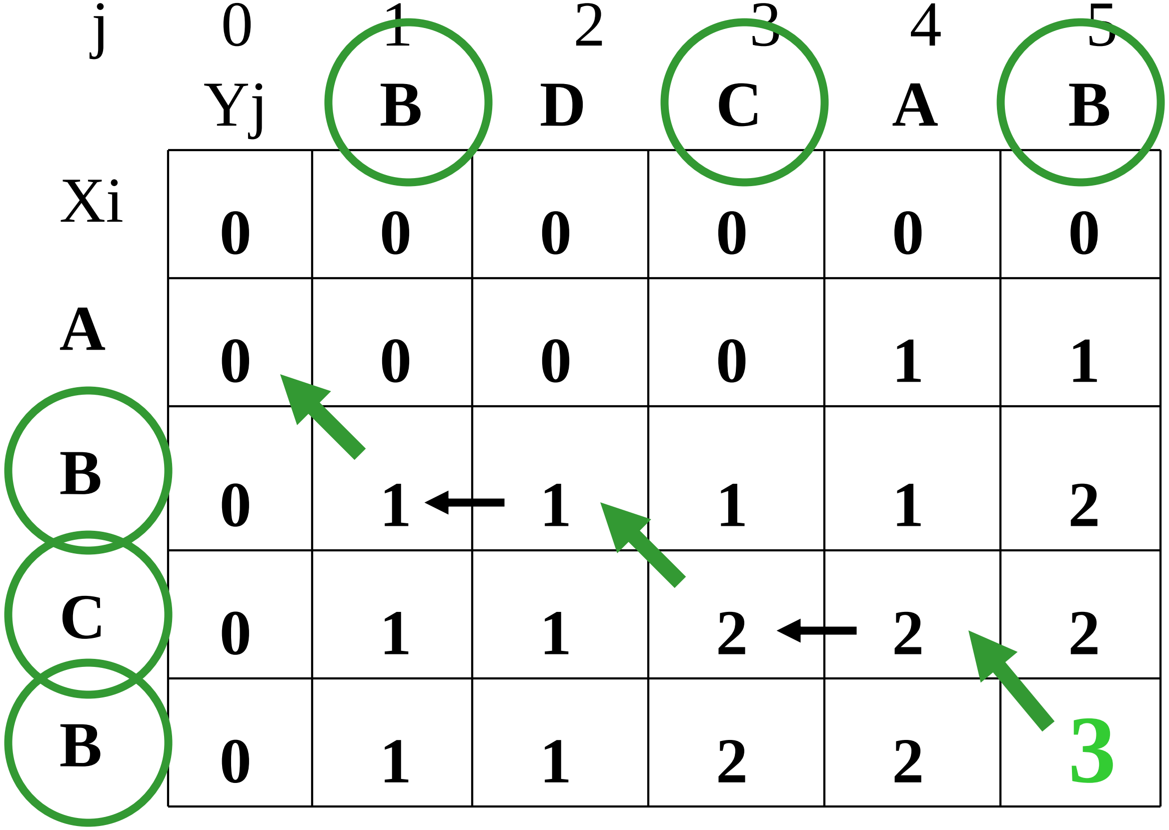
- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS



	j	0	1	2	3	4	5
i		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**	**1**	**2**	**2**	**3**

Finding LCS (2)



	j	0	1	2	3	4	5
i		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	1	2	2	3

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

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

(this string turned out to be a palindrome)

Review: 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

Properties of a 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

Review: Longest Common Subsequence (LCS)

- Problem: how to find the longest pattern of characters that is common to two text strings X and Y
- Dynamic programming algorithm: solve subproblems until we get the final solution
- Subproblem: first find the LCS of *prefixes* of X and Y.
- this problem has *optimal substructure*: LCS of two prefixes is always a part of LCS of bigger strings

Review: Longest Common Subsequence (LCS) continued

- Define X_i, Y_j to be prefixes of X and Y of length i and j; m = |X|, n = |Y|
- We store the length of LCS(X_i, Y_j) in c[i,j]
- Trivial cases: LCS(X_0 , Y_j) and LCS(X_i, Y_0) is empty (so c[0,j] = c[i,0] = 0)
- Recursive formula for c[i,j]:

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

c[m,n] is the final solution

Review: Longest Common Subsequence (LCS)

- After we have filled the array *c[ ]*, we can use this data to find the characters that constitute the Longest Common Subsequence

- Algorithm runs in $O(m*n)$, which is *much* better than the brute-force algorithm: $O(n\ 2^m)$

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

Knapsack problem

There are two versions of the problem:
(1) “0-1 knapsack problem” and
(2) “Fractional knapsack problem”

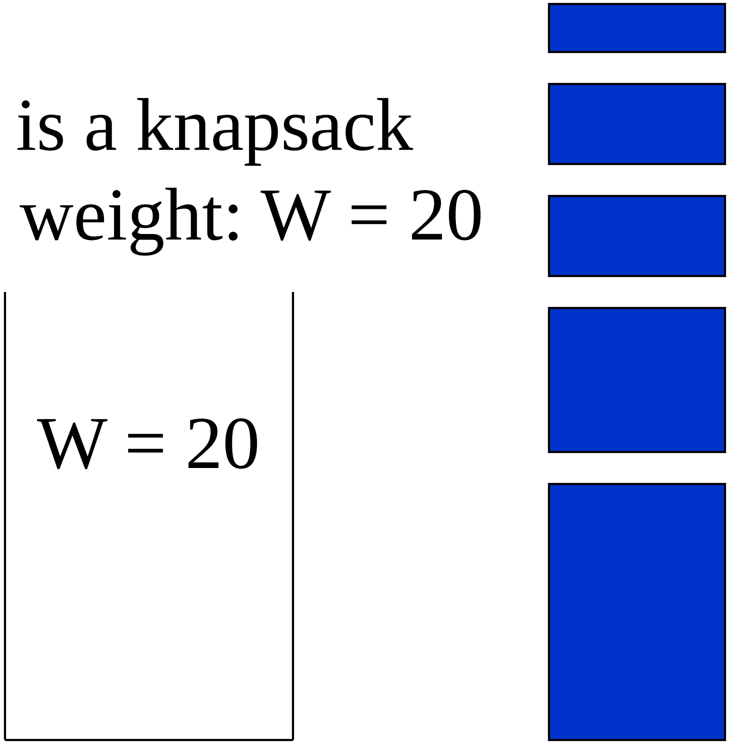
(1) Items are indivisible; you either take an item or not. Solved with *dynamic programming*
(2) Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

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?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



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

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.

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 $O(2^n)$

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

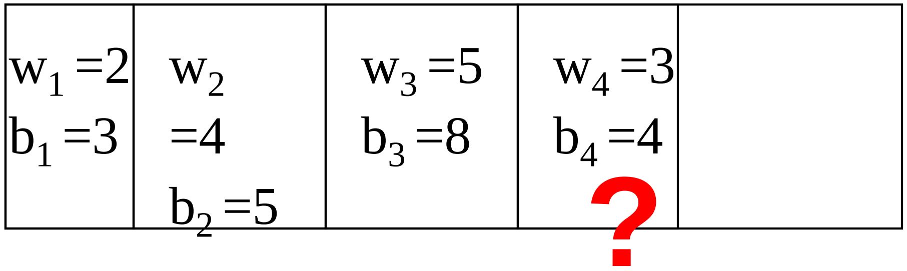
S_k *= {items labeled 1, 2, .. k}*

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = {*items labeled 1, 2, .. 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….

Defining a Subproblem



w_1 =2 b_1 =3

w_2 =4 b_2 =5

w_3 =5 b_3 =8

w_4 =3 b_4 =4

?

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

S_4

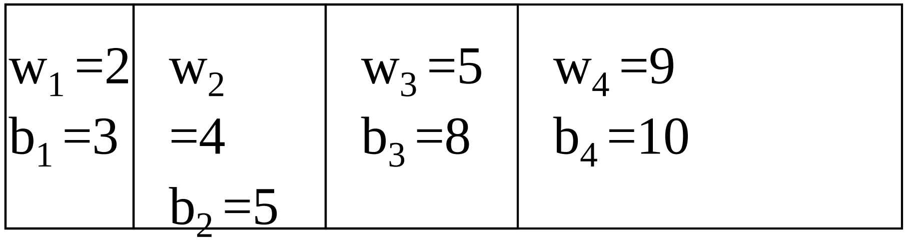
S_5

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20



w_1 =2 b_1 =3

w_2 =4 b_2 =5

w_3 =5 b_3 =8

w_4 =9 b_4 =10

For S_5:

Total weight: 20

total benefit: 26

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

Defining a Subproblem (continued)

- 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]*

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 $S_{k\text{-}1}$ that has total weight w, **or**

2) the best subset of $S_{k\text{-}1}$ that has total weight w-w_k plus the item k

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: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    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
```

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 $O(2^n)$

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)

Example (2)

i W	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
```

Example (3)

i W	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
```

Example (4)

Items:

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

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

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i = -1$

```
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]  // $w_i > w$
```

Example (5)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
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 $B[i,w] = B[i-1,w]$  // $w_i > w$
```

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 $w_i <= w$ // 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 $B[i,w] = B[i-1,w]$  // $w_i > w$
```

Example (7)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

```
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 $B[i,w] = B[i-1,w]$  // $w_i > w$
```

Example (8)

Items:

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

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

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

```
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
```

Example (9)

Items:

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

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			

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

```
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]**  // $w_i > w$
```

Example (10)

Items:

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

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			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
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
```

Example (11)

Items:

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

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			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
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
```

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

Example (13)

Items:

1: (2,3)
2: (3,4)

3: (4,5)

4: (5,6)

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**		

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

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

Example (14)

Items:

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

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		

i=3

b_i=5

w_i=4

w=1..3

```
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
```

Example (15)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

```
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
```

Example (15)

Items:

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

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	

i=3

b_i=5

w_i=4

w=5

w- w_i=1

```
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
```

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 $w_i$ <= 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
```

Example (17)

Items:

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

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

$i=3$

$b_i=5$

$w_i=4$

$w=5$

```
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
```

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
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

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: $\mathbf{O(m*n)}$ vs. $\mathbf{O(n * 2^m)}$
  - 0-1 Knapsack problem: $\mathbf{O(W*n)}$ vs. $\mathbf{O(2^n)}$