

# COMP319 Algorithms

## Lecture 9

# Dynamic Programming

Instructor: Gil-Jin Jang

School of Electronics Engineering, Kyungpook National University

Textbook Chapter 15

# Table of Contents

- Prerequisites
- Dynamic programming: conditions
- Longest common subsequence (LCS)
- 0-1 Knapsack problem
- Greedy algorithms

# PREREQUISITES

Divide and Conquer

Brief description and comparison of:

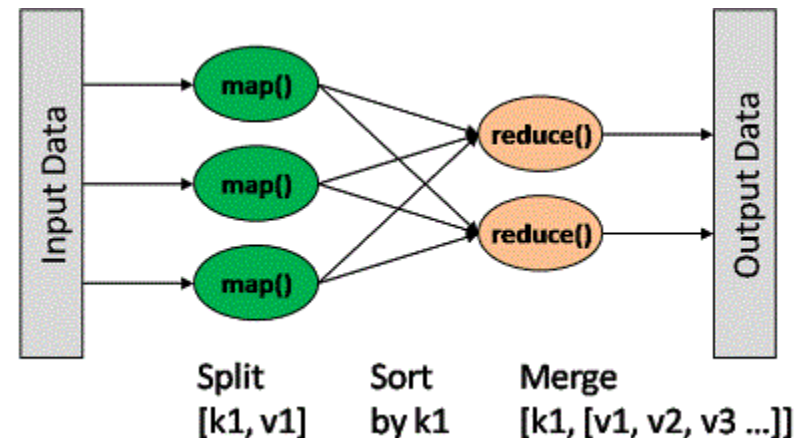
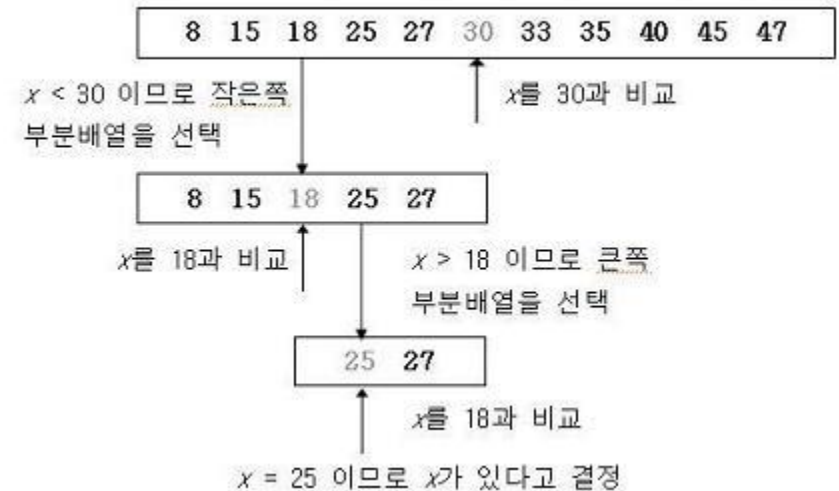
- Linear Programming
- Quadratic Programming
- Dynamic Programming

# Divide and Conquer (분할정복)

- To solve a large problem with many factors, it is helpful to divide it into smaller **subproblems**

- merge sort, quick sort
- binary search (이분검색)
- MapReduce (for parallel processing)

\* key  $x$ 의 값이 25일 때



# Divide and Conquer Analysis

- General solution

- Often followed by a recursive solution (재귀용법)

$$T(n) = 2T(n/2) + O(f(n)) \in O(f(n) \cdot \log_2 n) = O(f(n) \cdot \lg n)$$

$$T(n) = kT(n/k) + O(f(n)) \in O(f(n) \cdot \log_k n) = O(f(n) \cdot \lg n)$$

- Algorithm design points:

- How to define subproblems

- 해결가능한 분할 방법이 있어야 한다

- How to guarantee BALANCED division

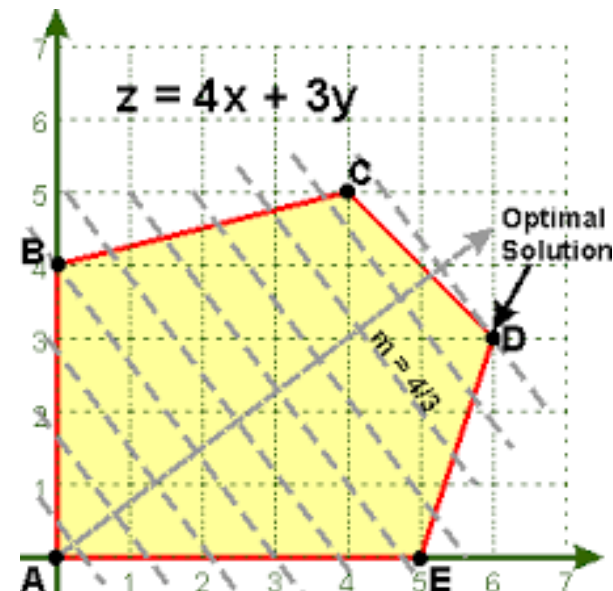
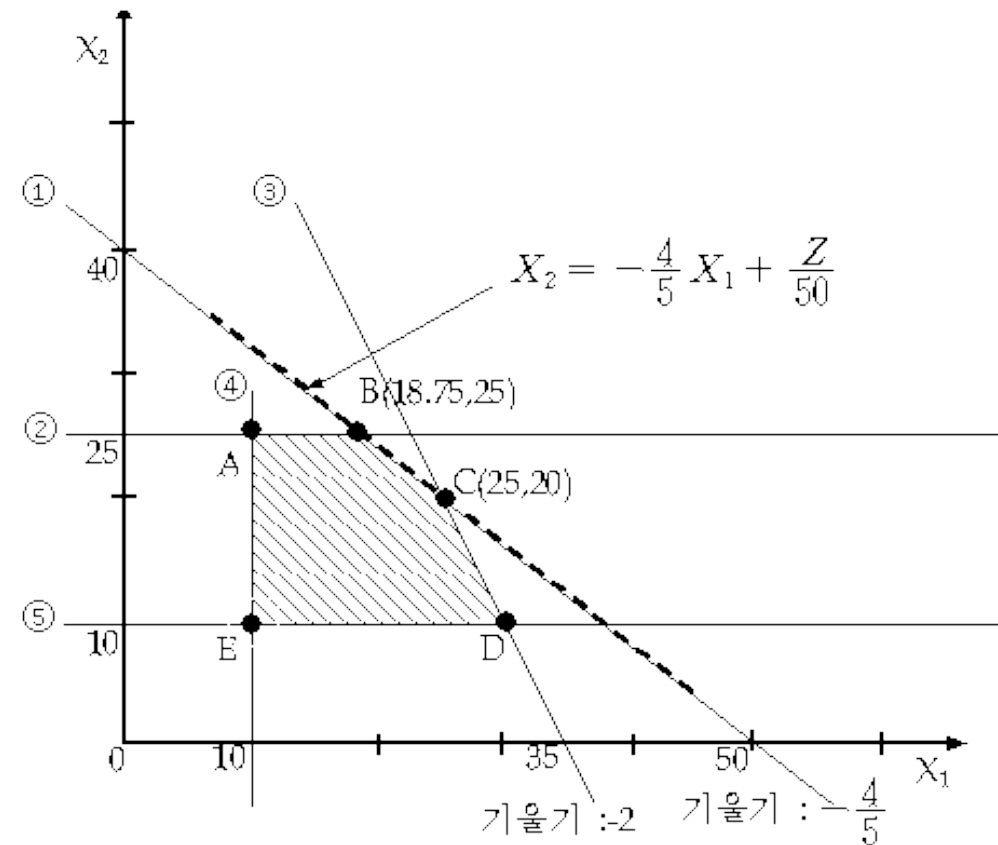
- 균형되게 분할하여야 깊이를 줄일 수 있다

# What is Programming (계획법)?

- In mathematics or economics, a set of procedure to find an optimal (min or max) solution **with constraints**
  - 조건부 최적화(최소화/최대화) 방법
- Some well-known programming
  - Linear programming
  - Quadratic programming
  - Integer programming
  - **Dynamic programming** .

# Linear Programming (선형 계획법)

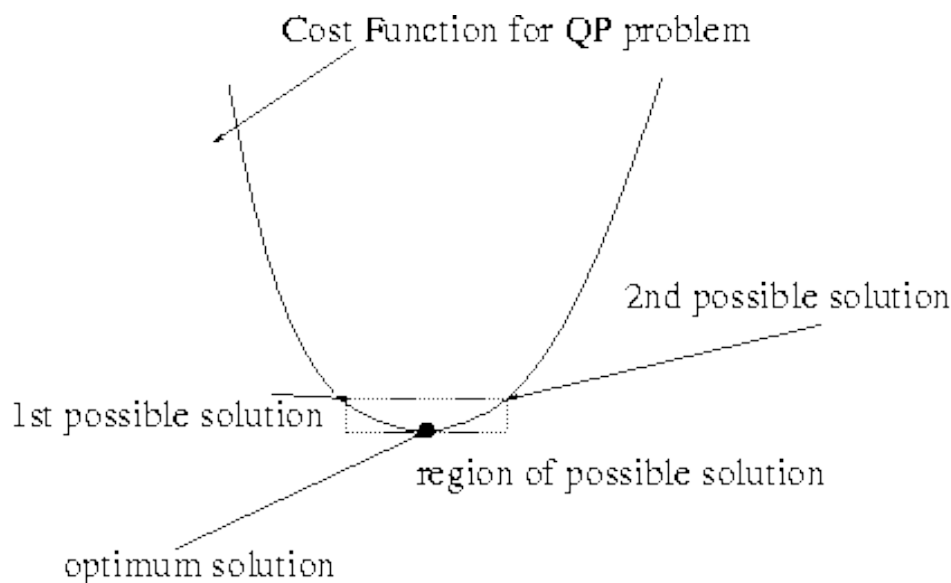
- SIMPLEX:** Finding a optimal with LINEAR constraints



# Quadratic Programming (이차 계획법)

- **CONVEX optimization:**  
Quadratic cost and  
quadratic / linear  
constraints

$$\begin{aligned}
 &\text{minimize} && \frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x} + \mathbf{x}^T \mathbf{f} \\
 &\text{subject to} && \sum_{i \in I_k} x_i = b_k, \quad k \in S_{equ} \\
 &&& \sum_{i \in I_k} x_i \leq b_k, \quad k \in S_{neq} \\
 &&& x_i \geq 0, \quad i \in I.
 \end{aligned}$$

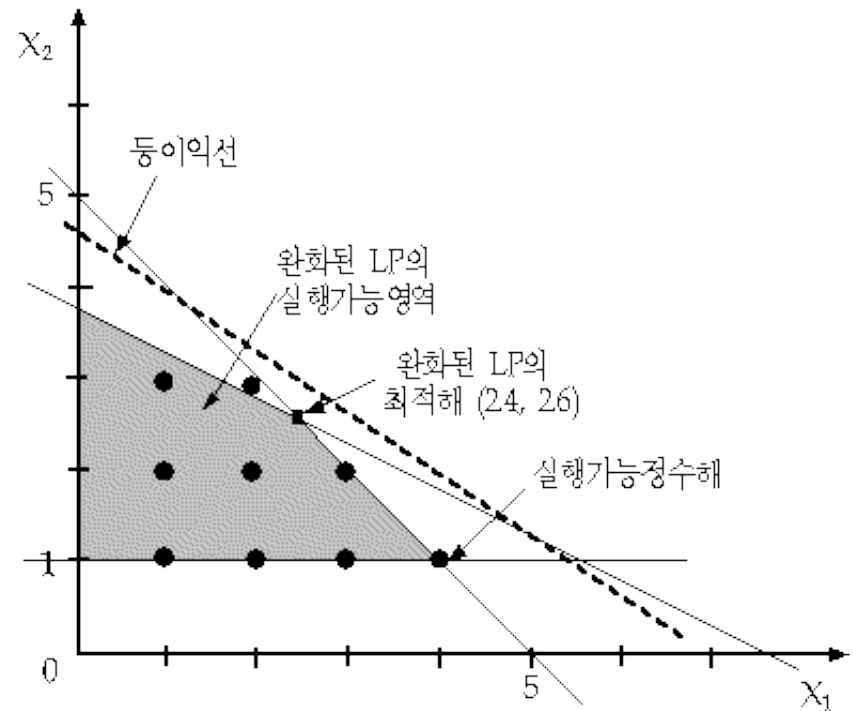


$$\begin{aligned}
 &\min_{\mathbf{x}} \quad 0.5x_1^2 + 0.5x_2^2 - 2x_1 - 2x_2 \\
 &\text{subject to:} \quad -x_1 + x_2 \leq 2 \\
 &\quad \quad \quad x_1 + 3x_2 \leq 5 \\
 &\quad \quad \quad x_1^2 + x_2^2 - 2x_2 \leq 1 \\
 &\quad \quad \quad x_1^2 + x_2^2 - x_1 + 2x_2 \leq 1.2 \\
 &\quad \quad \quad 0 \leq \mathbf{x}
 \end{aligned}$$



# Integer Programming (정수 계획법)

- Only integer solutions are accepted
  - $O(N^2)$  or  $O(k^N)$  by exhaustive search
  - 가능한 모든 경우를 탐색하는 경우의 수



# DYNAMIC PROGRAMMING

Overlapping Subproblems

Optimal Substructure

Longest Common Subsequence

# Dynamic Programming

- Another strategy for designing algorithms is *dynamic programming*
  - A metatechnique, not an algorithm (like divide & conquer)
  - The word PROGRAMMING is historical and predates computer programming
- Similarly to divide-and-conquer, use when problem breaks down into recurring small subproblems
  - The parent problem is dependent on the previous, small subproblems  
(과거의 해를 활용하여 현재의 문제해결)

# Properties: 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 (overlapping subproblems)

# Set 1. Overlapping Subproblems

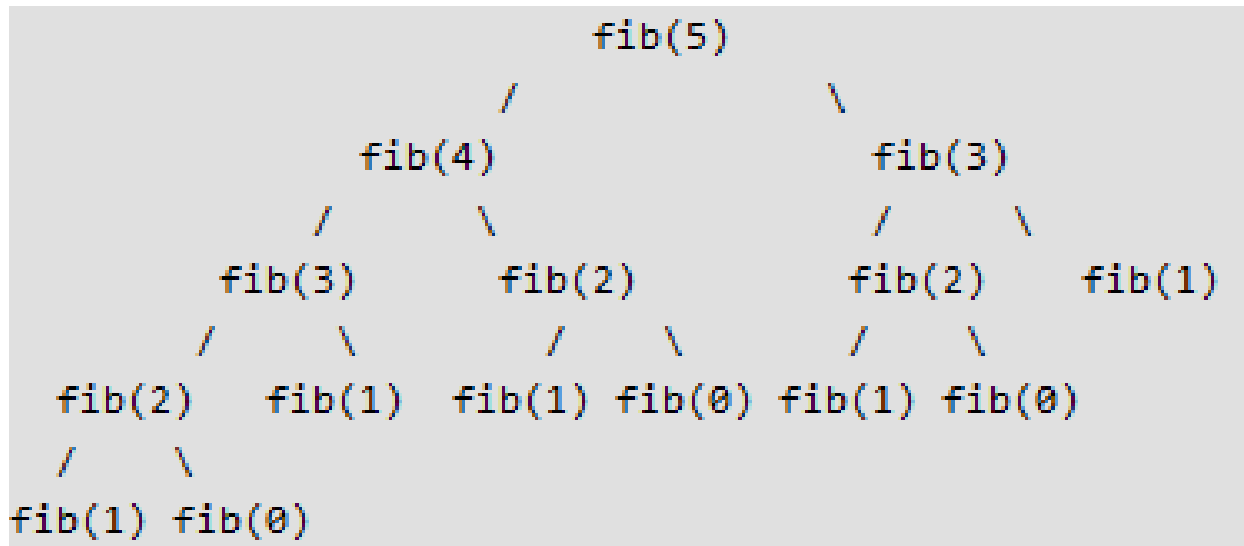
- When the subproblems overlap, DP stores the subproblem solutions in the table before use
  - 겹치게 사용되는 같은 subproblem들의 해를 미리 표에 저장하여 다시 연산을 하지 않는다
  - DP is not applicable when no overlapping subproblems
- Examples
  - Non-DP: binary search – subproblems do not overlap
  - DP: Fibonacci sequence
$$f(a, b) = f(a - 1, b) + f(a, b - 1), \quad a \geq 1, b \geq 1$$
$$f(0, 0) = f(n, 0) = f(0, n) = 1, \quad n \geq 1$$

<http://doctormaroo.tistory.com/1>

<http://www.geeksforgeeks.org/dynamic-programming-set-1/>

```
int fib(int n)
{
    if ( n <= 1 ) return n;
    else return fib(n-1) + fib(n-2);
}
```

**Good:** Easy to understand  
**Bad:** Recursive function calls consumes call stack in the system memory, and function switch time as well.



Significant amount of overlaps (redundancy)

<http://doctormaroo.tistory.com/1>

<http://www.geeksforgeeks.org/dynamic-programming-set-1/>

# A. Memorization (Top-down, 하향식)

- Memorization:  
Whenever `fib(n)` is computed, store the value in a table
- Re-use:  
When `fib(n-1)` or `fib(n-2)` is requested, check the table first

```
#include<stdio.h>
#define NIL -1
#define MAX 100
int lookup[MAX];

/* Initialize Search Table */
void initialize() {
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* Memorized Fibonacci */
int fib(int n) {
    if (lookup[n] == NIL) {
        if (n <= 1) lookup[n] = n;
        else
            lookup[n] = fib(n-1) + fib(n-2);
    }
    return lookup[n];
}
```

<http://doctormaroo.tistory.com/1>

<http://www.geeksforgeeks.org/dynamic-programming-set-1/>

## B. Tabulation (Bottom-up, 상향식)

- Memorization: fill the table when requested
  - 요구될 때 채운다
- Tabulation: fill the values first, and return the solution as the last filled value
  - 미리 사용될 값을 **예측**하고 채워 나간다
  - Good FILL STRATEGY is needed

```
/* With Table, no recursion */
#include<stdio.h>

int fib(int n) {
    int f[n+1];
    int i;
    f[0] = 0;    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}

int main ()
{
    printf("Fibonacci number is %d\n",
        fib(9));
}
```

<http://doctormaroo.tistory.com/1>

<http://www.geeksforgeeks.org/dynamic-programming-set-1/>



## Set 2. Optimal Substructure

- A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems
  - How many subproblems are used in an optimal solution.
  - How many choices in determining subproblems
  - Running time depends roughly on  $(\text{\#subprob}) \times (\text{\#choices})$
- Dynamic programming uses optimal structure in a **bottom up** manner:
  - Find optimal solutions to subproblems.
  - Choose which to use in optimal solution to the problem.

# LCS: LONGEST COMMON SUBSEQUENCE

# Longest Common Subsequence (LCS)

- Given two sequences  $\mathbf{x}[1..m]$  and  $\mathbf{y}[1..n]$ , find the longest subsequence which occurs in both

$\mathbf{X}$  =    A   **B**            **C**            **B** D   **A**   B

$\mathbf{Y}$  =            **B** D   **C** A   **B**            **A**

- [B C] and [B A] are both subsequences of both  $\mathbf{X}$  and  $\mathbf{Y}$ 
    - LCS? –the longest one among all the subsequences*
- Brute-force** (*non-systematic*) algorithm: For every subsequence of  $\mathbf{x}$ , check if it's a subsequence of  $\mathbf{y}$ 
  - How many subsequences of  $\mathbf{x}$  are there?*
  - What will be the running time of the brute-force algorithm?*

# Brute-Force 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  $O(n 2^m)$ 
  - *there exists  $2^m$  subsequences of  $x$  to check against  $n$  elements of  $y$ :  $\sim O(n 2^m)$*
- Brute-force: we can reduce the search entries by choosing minimum of the two sequences, but still exponential complexity

$$\sum_{l=1}^k \binom{k}{l} = 2^k, \quad k = \min(m, n)$$

# LCS Algorithm

- LCS problem has optimal substructure:
  - Subproblems: find LCS of pairs of prefixes of **x** and **y**
  - Solutions of the above subproblems are parts of the final one.
- Simplify the subproblem:
  - Only consider the problem of finding the length of LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS

# 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:
- **Case 1:**  $x[i] = y[j]$ 
  - 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  $Y_{j-1}$  , **plus 1**



# LCS recursive solution

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

- **Case 2:  $x[i] \neq y[j]$** 
  - As symbols don't match, our solution is not improved, and the length of  $\text{LCS}(X_i, Y_j)$  is the same as before, i.e., maximum of  $\text{LCS}(X_i, Y_{j-1})$  and  $\text{LCS}(X_{i-1}, Y_j)$

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

# LCS Length Algorithm

LCS-Length( $X$ ,  $Y$ )

```
1. m = length(X)           // get # symbols in X
2. n = length(Y)           // get # 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  $X_i$ 
    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])
6. return c
```

*Why not use recursive function? -- redundant*

# 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?*

$LCS(X, Y) = BCB$

$X = A \mathbf{B} \quad \mathbf{C} \quad \mathbf{B}$

$Y = \quad \mathbf{B} \mathbf{D} \mathbf{C} \mathbf{A} \mathbf{B}$

# LCS Example (0)

*ABCB*  
*BDCAB*

		<i>j</i>	0	1	2	3	4	5
			<i>Y<sub>j</sub></i>	<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>								
0	<i>X<sub>i</sub></i>							
1	<b>A</b>							
2	<b>B</b>							
3	<b>C</b>							
4	<b>B</b>							

$X = ABCB; \quad m = |X| = 4$

$Y = BDCAB; \quad n = |Y| = 5$

Allocate array **c[5, 4]**

# LCS Example (1)

*ABCB*  
*BDCAB*

		<i>j</i>	0	1	2	3	4	5
			<i>Y<sub>j</sub></i>	<b><i>B</i></b>	<b><i>D</i></b>	<b><i>C</i></b>	<b><i>A</i></b>	<b><i>B</i></b>
<i>i</i>	<i>X<sub>i</sub></i>	0	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
1	<b><i>A</i></b>	<b>0</b>						
2	<b><i>B</i></b>	<b>0</b>						
3	<b><i>C</i></b>	<b>0</b>						
4	<b><i>B</i></b>	<b>0</b>						

```

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

# LCS Example (2)

**A**BCB  
**B**DCAB

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0				
2	<b>B</b>		0					
3	<b>C</b>		0					
4	<b>B</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)

**A**BCB  
**B**DC**A**B

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0		
2	<b>B</b>		0					
3	<b>C</b>		0					
4	<b>B</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)

**A**BCB  
BDC**A**B

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	
2	<b>B</b>		0					
3	<b>C</b>		0					
4	<b>B</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)

**A**BCB  
BDCAB

		<i>j</i>	0	1	2	3	4	5
			<i>Y<sub>j</sub></i>	<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>	0	0	0	0	0	0	0
1	<b>A</b>	0	0	0	0	0	1	→ 1
2	<b>B</b>	0						
3	<b>C</b>	0						
4	<b>B</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)

**A****B****C****B**  
**B****D****C****A****B**

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1				
3	<b>C</b>		0					
4	<b>B</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</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1	1	1	1	
3	<b>C</b>		0					
4	<b>B</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</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
<b>2</b>	<b>B</b>		0	1	1	1	1	<b>2</b>
3	<b>C</b>		0					
4	<b>B</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**  
**BD CAB**

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1	1	1	1	2
3	<b>C</b>		0	↓	↓			
				1	→	1		
4	<b>B</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</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1	1	1	1	2
3	<b>C</b>		0	1	1	2		
4	<b>B</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</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1	1	1	1	2
3	<b>C</b>		0	1	1	2	2	2
4	<b>B</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**

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<b>A</b>		0	0	0	0	1	1
2	<b>B</b>		0	1	1	1	1	2
3	<b>C</b>		0	1	1	2	2	2
4	<b>B</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**

		<i>j</i>	0	1	2	3	4	5
		<i>Y<sub>j</sub></i>		<i>B</i>	<i>D</i>	<i>C</i>	<i>A</i>	<i>B</i>
<i>i</i>	<i>X<sub>i</sub></i>							
0			0	0	0	0	0	0
1	<i>A</i>		0	0	0	0	1	1
2	<i>B</i>		0	1	1	1	1	2
3	<i>C</i>		0	1	1	2	2	2
4	<i>B</i>		0	1	1	2	2	

Diagram illustrating the Longest Common Subsequence (LCS) example. The table shows the dynamic programming table with indices *i* and *j*. The sequence *X* is *ABCB* and the sequence *Y* is *BDCAB*. The table entries represent the length of the LCS for subproblems. The current state is at *i*=4, *j*=4, where the value is 2. Arrows indicate the backtracking path from (4,4) to (3,3) and then to (2,2). A circle highlights the cell (4,0) containing the value 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 (15)

*ABCB*  
*BDCAB*

		<i>j</i>	0	1	2	3	4	5
			<i>Y<sub>j</sub></i>	<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
<i>i</i>	<i>X<sub>i</sub></i>	0	0	0	0	0	0	0
1	<b>A</b>	0	0	0	0	0	1	1
2	<b>B</b>	0	1	1	1	1	1	2
3	<b>C</b>	0	1	1	2	2	2	2
4	<b>B</b>	0	1	1	2	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 **BACKTRACK** how it was acquired:

2	2
2	3



For example, here  
$$c[i, j] = c[i - 1, j - 1] + 1 = 2 + 1 = 3$$
  
Path:  $(i, j) \rightarrow (i - 1, j - 1)$

# How to find actual LCS - continued

*Remember that:*

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

- 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
$i$		$Y_j$		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
		$X_i$						
0		<b>0</b>	0	0	0	0	0	0
1	<b>A</b>	0	0	0	0	0	1	1
2	<b>B</b>	0	1	1	1	1	1	2
3	<b>C</b>	0	1	1	2	2	2	2
4	<b>B</b>	0	1	1	2	2	2	<b>3</b>

# Finding LCS (2)

		$j$	0	1	2	3	4	5
		$Y_j$		<b>B</b>	<b>D</b>	<b>C</b>	<b>A</b>	<b>B</b>
$i$	$X_i$							
0		0	0	0	0	0	0	0
1	<b>A</b>	0	0	0	0	0	1	1
2	<b>B</b>	0	1	1	1	1	1	2
3	<b>C</b>	0	1	1	2	2	2	2
4	<b>B</b>	0	1	1	2	2	2	<b>3</b>

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

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

# 0-1 KNAPSACK PROBLEM



# 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  $W$ .
- Consider weights of items as well as their value.

<i>Item #</i>	<i>Weight</i>	<i>Value</i>
1	1	8
2	3	6
3	5	5

# Knapsack problem formulation

- There are two versions of the problem:
- (1) “0-1 knapsack problem”
  - Items are indivisible; you either take an item or not. Solved with *dynamic programming*
- (2) “Fractional knapsack problem”
  - Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

# 0-1 Knapsack problem

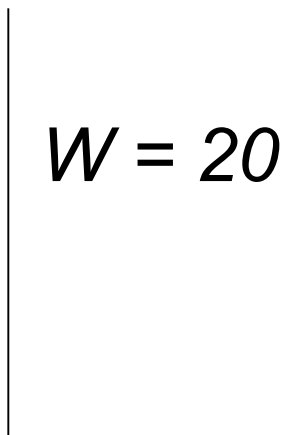
- Given a knapsack with maximum capacity  $W$ , and a set  $S$  consisting of  $n$  items
- Each item  $i$  has some weight  $w_i$  and benefit value  $b_i$  (all  $w_i$ ,  $b_i$  and  $W$  are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
  - It is called a “0-1” problem, because each item must be entirely accepted or rejected.





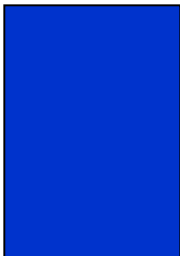
# 0-1 Knapsack problem: a picture

$$T^* = \arg \max_T \sum_{i \in T} b_i$$

subject to  $\sum_{i \in T} w_i \leq W$

Max weight:  
 $W = 20$



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

# 0-1 Knapsack: brute-force approach

- 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  $O(2^n)$  - Can we do better?

If items are labeled  $1..n$ , then a subproblem would be to find an optimal solution for

$$S_k = \{\text{items labeled } 1, 2, \dots, k\}$$
- Question: can we describe the final solution ( $S_n$ ) in terms of subproblems ( $S_k$ )?

# 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

$w_1=2$	$w_2=4$	$w_3=5$	$w_4=9$
$b_1=3$	$b_2=5$	$b_3=8$	$b_4=10$

**For  $S_5$ :**

Total weight: 20

total benefit: 26

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

$S_4$

$S_5$

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

# Alternate Formulation of Subproblems

- Add another parameter:  $\underline{w}$ , which will represent the exact weight for each subset of items
- Let  $X_k^w$  be the best subset of  $S_k$  that has maximum total weight  $w$ , then  $X_k^w$  is either of:
  - 1)  $X_{k-1}^w$ : the best subset of  $S_{k-1}$  with total weight  $w$
  - 2)  $X_{k-1}^{w-w_k} \cup \{item_k\}$ : the best subset of  $S_{k-1}$  with total weight  $w - w_k$  plus the item  $k$

# Alternate Formulation of Subproblems

- $X_k^w = X_{k-1}^w$  or  $X_{k-1}^{w-w_k} \cup \{item_k\}$ 
  - To determine which option to take, we need to know the benefits of the above
- Recursive formula for subproblems:
  - Let  $B_k^w$  be the total benefit of  $X_k^w$ , then the subproblem then will be to compute  $B_k^w$
  - Recursive formula for subproblems:

$$B_k^w = \begin{cases} B_{k-1}^w & \text{if } w_k > w \\ \max \left\{ B_{k-1}^w, B_{k-1}^{w-w_k} + b_k \right\} & \text{otherwise} \end{cases}$$



# Recursive Formula

$$B_k^w = \begin{cases} B_{k-1}^w & \text{if } w_k > w \\ \max \left\{ B_{k-1}^w, B_{k-1}^{w-w_k} + b_k \right\} & \text{otherwise} \end{cases}$$

- The best subset of  $S_k$  that has the total weight  $w$ , either contains item  $k$  or not.
  - **Case 1**:  $w_k > w$ . Item  $k$  cannot be part of the solution; if it is added, regardless of the other items, the total weight becomes larger than  $w$ .
  - **Case 2**:  $w_k \leq w$ . Then the item  $k$  can be in the solution, and choose the case with greater benefit.

# 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  $w_i \leq w$  // consider item i
            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  $B[i, w] = B[i-1, w]$  //  $w_i > w$ 

```

Denote  $B[k, w]$  for  $B_k^w$

# 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     $O(W)$   
  ...
```

*What is the running time of this algorithm?*

*→  $O(n*W)$*

*Remember that the brute-force algorithm  
takes  $O(2^n)$*

# Example (1)

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

*Run the algorithm on the following data:*

$n = 4$  (# of elements)

$W = 5$  (max weight)

*Elements*

(weight, benefit):

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

*for  $w = 0$  to  $W$*

*$B[0,w] = 0$*

# Example (2)

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

$n = 4$  (# of elements)

$W = 5$  (max weight)

Elements

(weight, benefit):

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

for  $i = 0$  to  $n$

$$B[i,0] = 0$$

# Example (3)

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

# Example (4)

	<i>i</i>	0	1	2	3	4
<i>w</i>	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  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (5)

$w$	$i$	0	1	2	3	4
0		0	0	0	0	0
1		0	0			
2		0	3			
3		0	<b>3</b>			
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  $B[i, w] = B[i-1, w]$  //  $w_i > w$



# Example (6)

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	<b>3</b>			
5		0				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

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

# 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	<b>3</b>			

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

# Example (8)

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

# Example (9)

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

# 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		
4		0	3			
5		0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

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

# 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	<b>4</b>		
5		0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

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

# Example (12)

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	<b>7</b>		

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

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

# 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 → 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  $w_i \leq w$  // 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  $B[i, w] = B[i-1, w]$  //  $w_i > w$



# 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	
5		0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i = 0$

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

# 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  $w_i \leq w$  // 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  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (16)

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	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

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

# 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	<b>7</b>

$i=3$

$b_i=5$

$w_i=4$

$w=5$

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

# 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, backtracking is necessary
  - See the LCS algorithm

# 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 vs. naïve algorithm):
  - LCS:  $O(mn)$  vs.  $O(n2^m)$
  - 0-1 Knapsack problem:  $O(Wn)$  vs.  $O(2^n)$

# GREEDY ALGORITHMS

# Review: Dynamic Programming

- Break problem down into recurring subproblems
- Basic idea:
  - **Optimal substructure**: optimal solution to problem consists of optimal solutions to subproblems
  - **Overlapping subproblems**: few subproblems in total, many recurring instances of each
  - Solve **bottom-up**, building a table of solved subproblems that are used to solve larger ones
- Variations:
  - “Table” could be 3-dimensional, triangular, a tree, etc.



# Review: Memorization

- *Memorization* is another way to deal with overlapping subproblems
  - After computing the solution to a subproblem, store it in a table
  - Subsequent calls just do a table lookup
- Can modify recursive algorithm to use memorization:
  - There are  $mn$  subproblems
  - *How many times is each subproblem wanted?*
  - *What will be the running time for this algorithm? The running space? – usually proportional to the number of subproblems ( $O(mn)$ )*

# Review: Dynamic Programming

- *Dynamic programming*: build table bottom-up
  - Same table as memorization, but instead of starting at  $(m,n)$  and recursing down, start at  $(1,1)$
- *Least Common Subsequence*: LCS easy to calculate from LCS of prefixes
  - There are few subproblems in total
  - And many recurring instances of each  
(unlike divide & conquer, where subproblems unique)
  - *How many distinct problems exist for the LCS of  $x[1..m]$  and  $y[1..n]$ ? → A:  $mn$*

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

# Greedy Algorithms

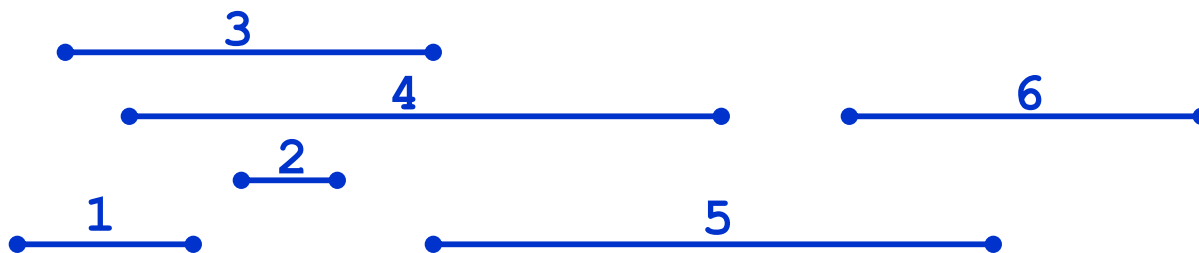
- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Walking to the Corner
    - Playing a bridge hand
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works, or gives us near-optimal solution
- Dynamic programming can be overkill (too much); greedy algorithms tend to be easier to code

# Activity-Selection Problem

- Problem: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

# Activity-Selection

- Formally:
  - Given a set  $S$  of  $n$  activities
    - $s_i$  = start time of activity  $i$
    - $f_i$  = finish time of activity  $i$
  - Find max-size subset  $A$  of compatible activities



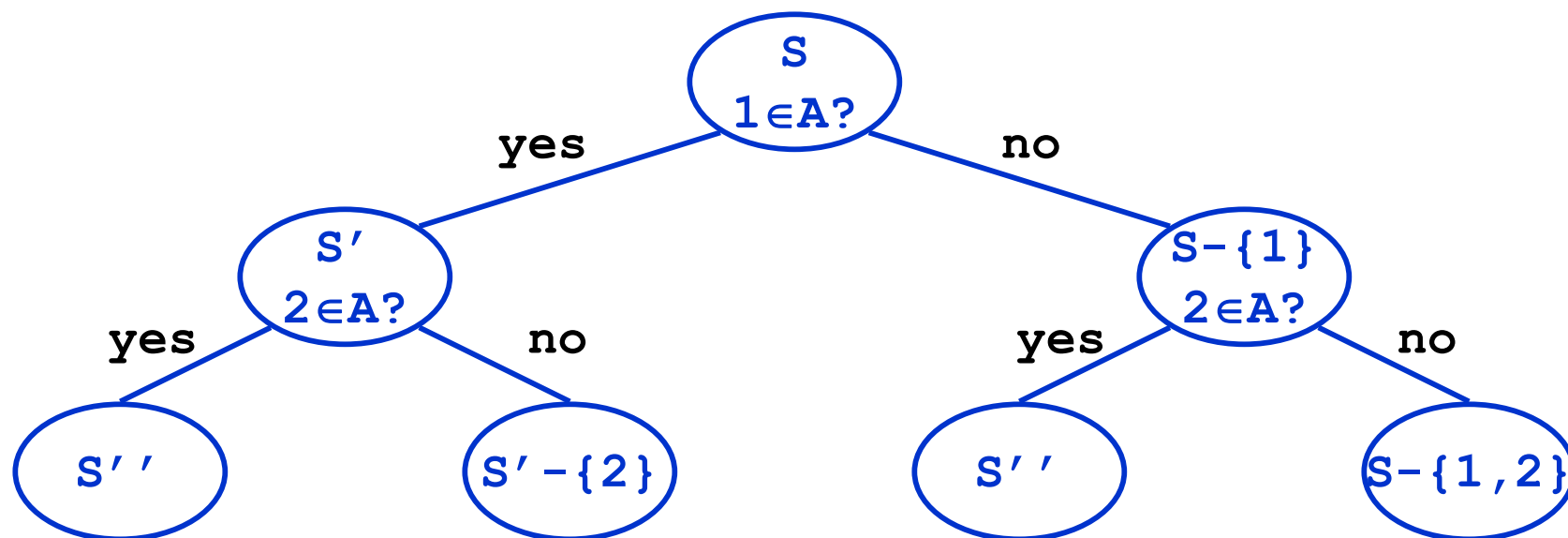
- Assume that  $f_1 \leq f_2 \leq \dots \leq f_n$

# Activity Selection: Optimal Substructure

- Let  $k$  be the minimum activity in  $A$  (i.e., the one with the earliest finish time).
- Then  $A - \{k\}$  is optimal to  $S' = \{i \in S: s_i \geq f_k\}$ 
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in  $S$  compatible with #1
  - Proof: if we could find optimal solution  $B'$  to  $S'$  with  $|B'| > |A - \{k\}|$ ,
    - Then  $B' \cup \{k\}$  is compatible
    - And  $|B' \cup \{k\}| > |A|$

# Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



# Greedy Choice Property

- Dynamic programming? Memorize? Yes, but...
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice  $\Rightarrow$  globally optimal solution
  - Theorem 17.1: if  $S$  is an activity selection problem sorted by finish time, then  $\exists$  optimal solution  $A \subseteq S$  such that  $\{1\} \in A$ 
    - Sketch of proof: if  $\exists$  optimal solution  $B$  that does not contain  $\{1\}$ , can always replace first activity in  $B$  with  $\{1\}$  (*Why?*). Same number of activities, thus optimal.



# Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

# Review: The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

# Review: The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among  $n$  items, where the  $i$ th item worth  $v_i$  dollars and weighs  $w_i$  pounds
  - Carrying at most  $W$  pounds, maximize value
    - Note: assume  $v_i$ ,  $w_i$ , and  $W$  are all integers
    - “0-1” b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

## Review: The Knapsack Problem and Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most  $W$  pounds
  - *If we remove item  $j$  from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most  $W - w_j$  that thief could take from museum, excluding item  $j$

# Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*

## The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - As you have seen, however, it can be solved with dynamic programming

2019-10-29

# NEXT TOPICS

Graph algorithms

Breadth-first search vs Depth-first search

Minimum Spanning Tree: Prim's Algorithm