COMP319 Algorithms 1
Lecture 12
Dynamic Programming

Instructor: Gil-Jin Jang

Definition of Dynamic Programming

Longest common subsequence (LCS)

0-1 Knapsack problem

Textbook Chapter 15

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Table of Contents

Divide and Conquer

Brief description and comparison of:

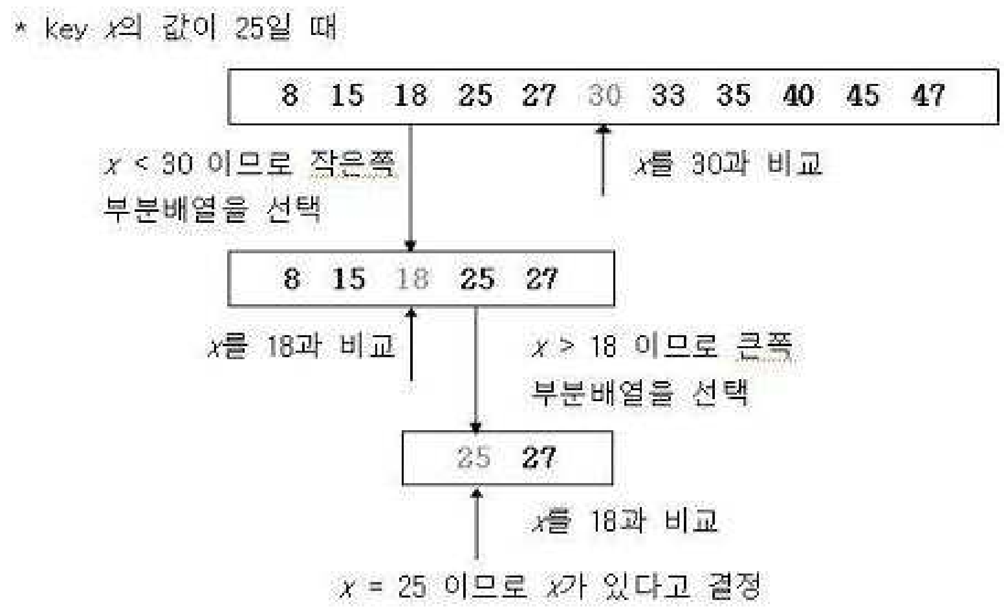
- Linear Programming
- Quadratic Programming
- Dynamic Programming

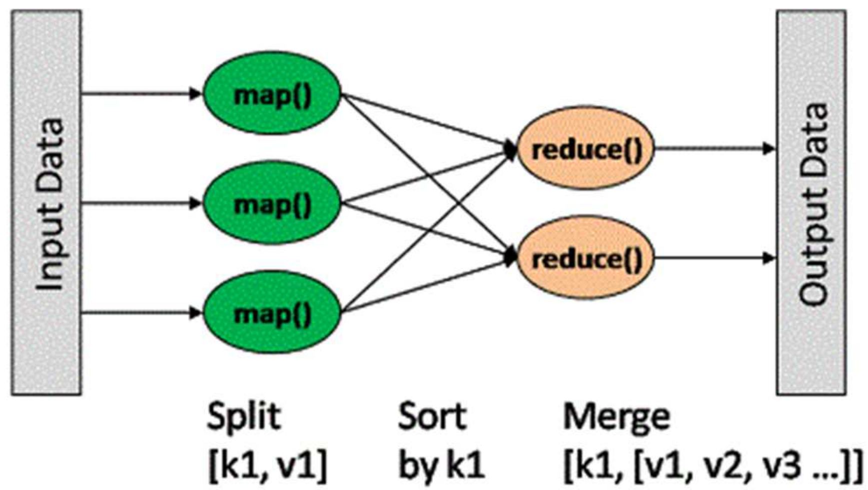
PREREQUISITES

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Divide and Conquer (분할정복)

- To solve a large problem with many factors, it is helpful to divide it into smaller **<u>subproblems</u>**
  - merge sort, quick sort
  - binary search
  - MapReduce (for parallel processing)





Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Divide and Conquer Analysis

- General solution
  - Often followed by a *recursive* solution

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

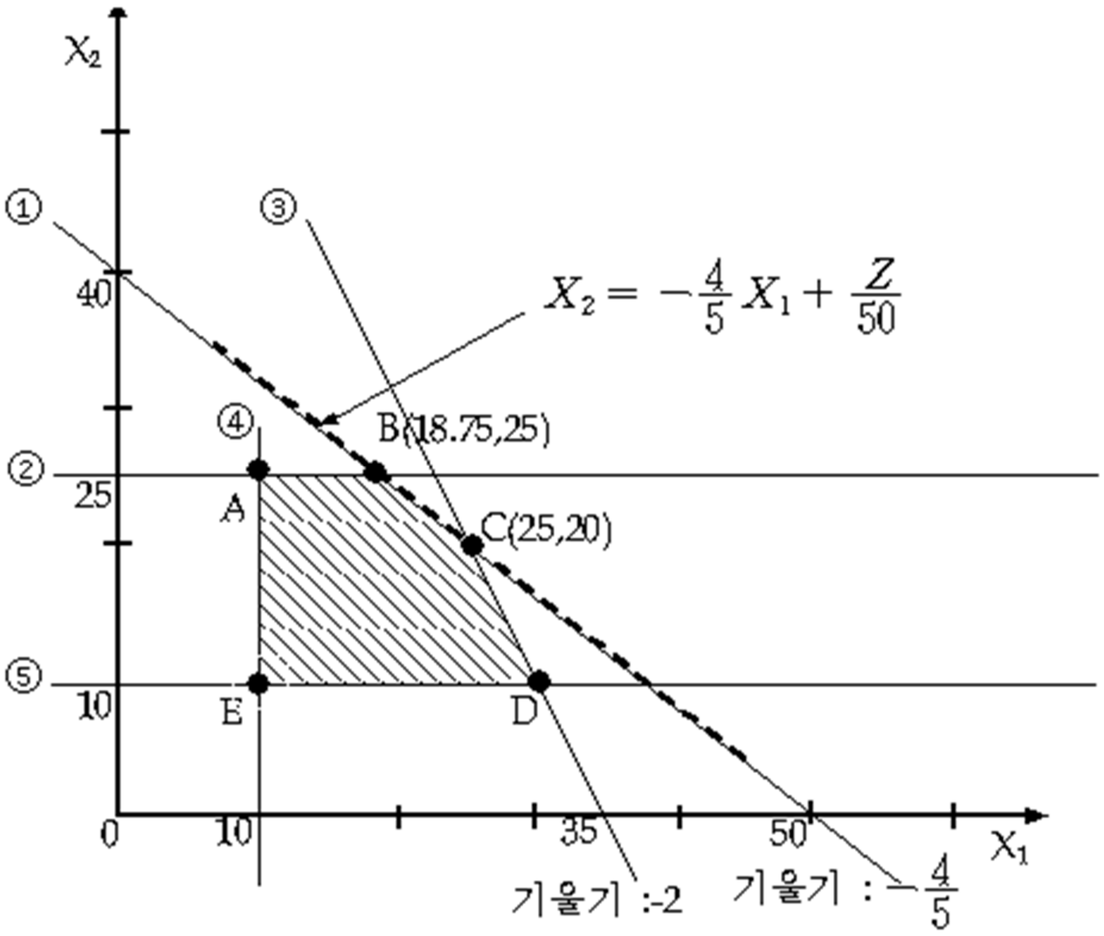
- Algorithm design points:
  - How to define subproblems
    - Subproblems should be solvable
  - How to guarantee BALANCED division
    - Balanced division can reduce the recursion depth

What is Programming?

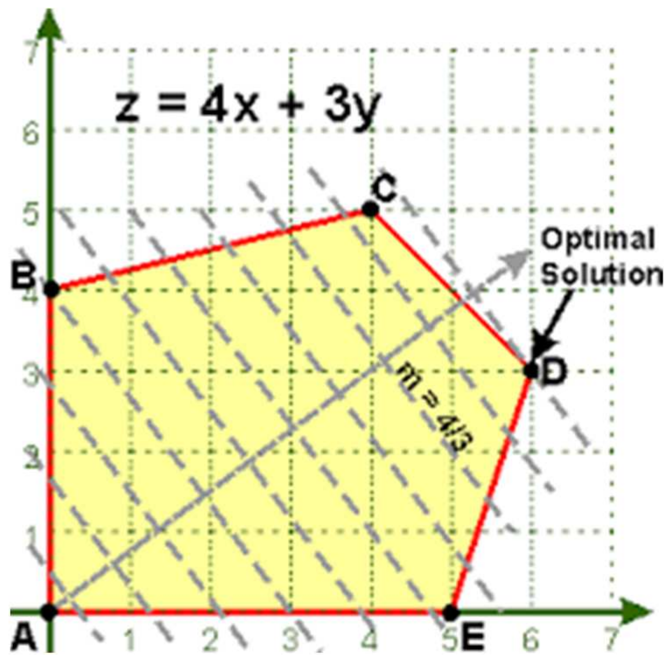
- In mathematics or economics, a set of procedure to find an optimal (min or max) solution ***with constraints***
  - Constrained optimization (minimization/maximization)
- Some well-known programming
  - Linear programming
  - Quadratic programming
  - Integer programming
  - **Dynamic programming** .

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Linear Programming



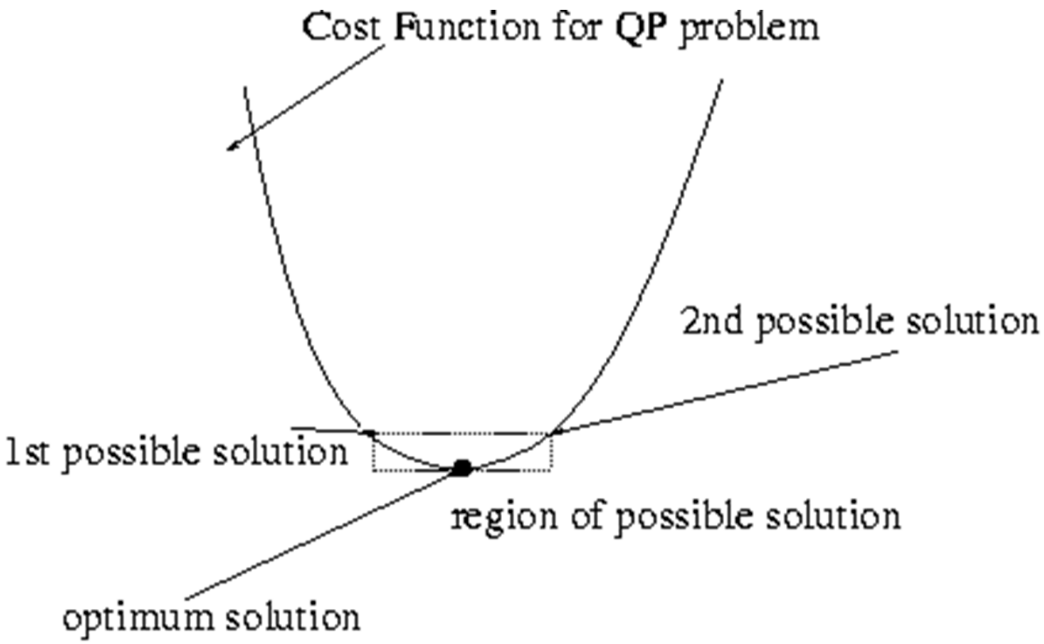
- **SIMPLEX**: Finding a optimal with LINEAR constraints



Quadratic Programming

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

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

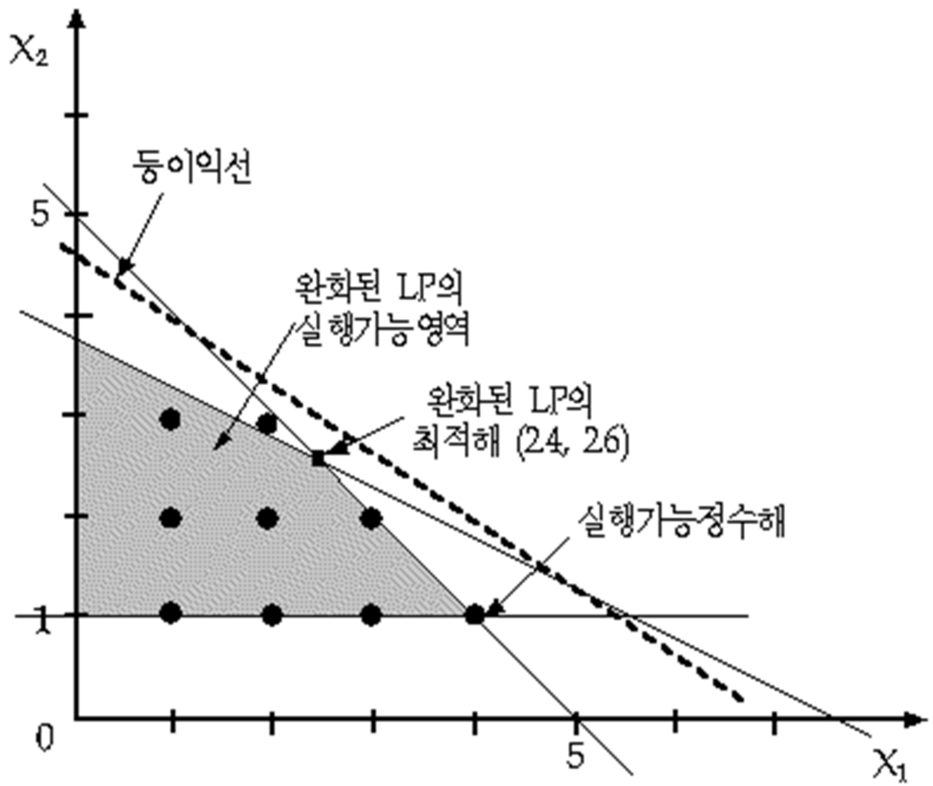


$$
\begin{aligned}
\min_{\mathrm{x}} \; & 0.5x_1^2 + 0.5x_2^2 - 2x_1 - 2x_2 \\
\text{subject to:} \quad & -x_1 + x_2 \le 2 \\
& x_1 + 3x_2 \le 5 \\
& x_1^2 + x_2^2 - 2x_2 \le 1 \\
& x_1^2 + x_2^2 - x_1 + 2x_2 \le 1.2 \\
& 0 \le \mathrm{x}
\end{aligned}
$$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Integer Programming

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



Overlapping Subproblems

Optimal Substructure

Longest Common Subsequence

DYNAMIC PROGRAMMING

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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
  - Solving orders are important

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**)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Set 1. Overlapping Subproblems

- When the subproblems overlap, DP ***stores*** the subproblem solutions in the ***table before*** use
  - DP is not applicable when no overlapping subproblems
- Examples
  - Non-DP: binary search – subproblems do not overlap
  - DP: Fibonacci sequence

$$f(a,b) = f(a-1,b) + f(a,b-1), \qquad a \geq 1, b \geq 1$$
$$f(0,0) = f(n,0) = f(0,n) = 1, \qquad n \geq 1$$

http://doctormaroo.tistory.com/1
http://www.geeksforgeeks.org/dynamic-programming-set-1/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

```
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.

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \       /    \       /     \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Significant amount of overlaps (redundancy)

http://doctormaroo.tistory.com/1
http://www.geeksforgeeks.org/dynamic-programming-set-1/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

A. Memorization (Top-down)

- 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/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

B. Tabulation (Bottom-up)

- Memorization: fill the table when requested
  - 요구될 때 채운다
- Tabulation: fill the values first, and return the solution as the last filled value
  - **Prediction** of the solutions to use is crucial
  - 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/

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 (#subprob) x (#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.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Prerequisites

Requirements for dynamic programming

Longest common subsequence

0-1 Knapsack problem

LCS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Longest Common Subsequence (LCS)

- Given two sequences **x**[1..*m*] and **y**[1..*n*], find the longest subsequence which occurs in both

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

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

  - [B C] and [B A] are both subsequences of both **X** and **Y**
    - *LCS? –the longest one among all the subsequences*
- **Brute-force** (*non-systematic*) 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?*

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**: ~ O(n 2^m)*
- Brute-force: we can reduce the search entries by choosing minimum of the two sequences, but still exponential complexity

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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: $c[0, j] = c[i,0] = 0$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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_{i-1} , **plus 1**

LCS recursive solution

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

- **Case 2:** $x[i] \neq y[j]$
  - 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 , $Y_{j\text{-}1}$) and LCS($X_{i\text{-}1}$,Y_j)

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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: Y0
4. for j = 1 to n
      c[0,j] = 0              // special case: X0
5. for i = 1 to m             // for all Xi
    for j = 1 to n            // for all Yj
     if Xi == Yj  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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Example (2)

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				
2	B		0					
3	C		0					
4	B		0					

```
if ( Xi == Yj )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ( Xi == Yj )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ( Xi == Yj )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Example (6)

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				
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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Example (7)

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	
3	C		0					
4	B		0					

```
if ( Xi == Yj )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ( Xi == Yj )
     c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Example (10)

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			
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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ( Xi == Yj )
      c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ( Xi == Yj )
     c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Example (15)

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	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] )
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

LCS Algorithm Running Time

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

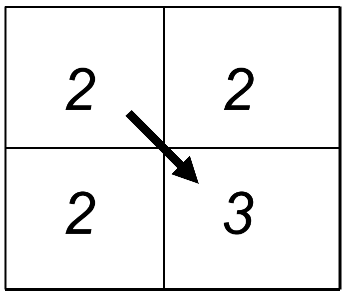
`O(m*n)`

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 ***<u>BACKTRACK</u>*** how it was acquired:



For example, here

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

Path: $(i,j) \to (i-1, j-1]$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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)

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	1	2	2	3

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

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

(this string turned out to be a palindrome)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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
- To know the items that make this maximum value, backtracking is necessary
- Running time (Dynamic programming vs. naïve algorithm):
  - LCS: $O(mn)$ vs. $O(n2^m)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Prerequisites

Requirements for dynamic programming

Longest common subsequence (LCS)

0-1 Knapsack problem

0-1 KNAPSACK PROBLEM

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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.

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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*.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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.

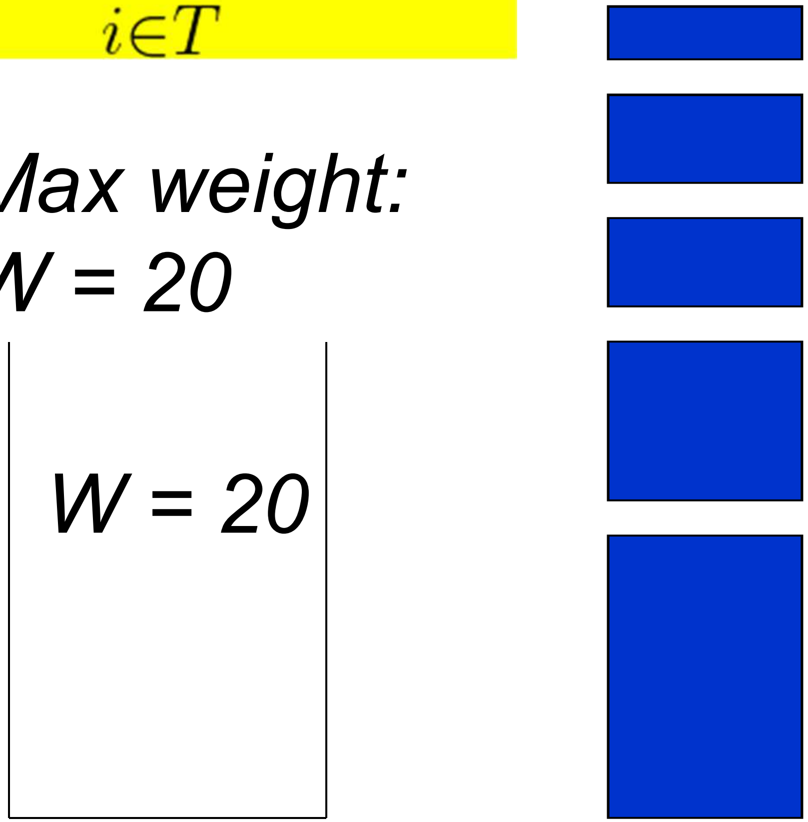
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

0-1 Knapsack problem: a picture

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

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

Max weight:
W = 20



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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Defining a Subproblem

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

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

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Alternate Formulation of Subproblems

- As we have seen, the solution for S_4 is not part of the solution for S_5
  - The definition of a subproblem is not satisfied
- Alternate formulation
  - Add another parameter: ***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:

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

0-1 Knapsack Algorithm

Denote B[k,w] for B_k^w

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
      if wi <= w // consider item i
           if bi + B[i-1,w-wi] > B[i-1,w]
                B[i,w] = bi + B[i-1,w- wi]
           else
                B[i,w] = B[i-1,w]
      else B[i,w] = B[i-1,w]   // wi > w
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 $O(2^n)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Example (1)

W \ i	0	1	2	3	4
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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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)

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 $w_i <= w$ *// 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]** *//* $w_i > w$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Example (4)

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				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Example (5)

Items:

$\boxed{1: (2,3)}$

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				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 $w_i <= w$ *// 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 $\mathbf{B[i,w] = B[i-1,w]}$ *//* $w_i > w$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 $w_i <= w$ *// 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]** *//* $w_i > w$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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	**4**		
5	0	3			

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 $w_i <= w$ *// 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]** *//* $w_i > w$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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 $w_i <= w$ *// 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 (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**	

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

Items:

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

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

END OF LECTURE 12

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I