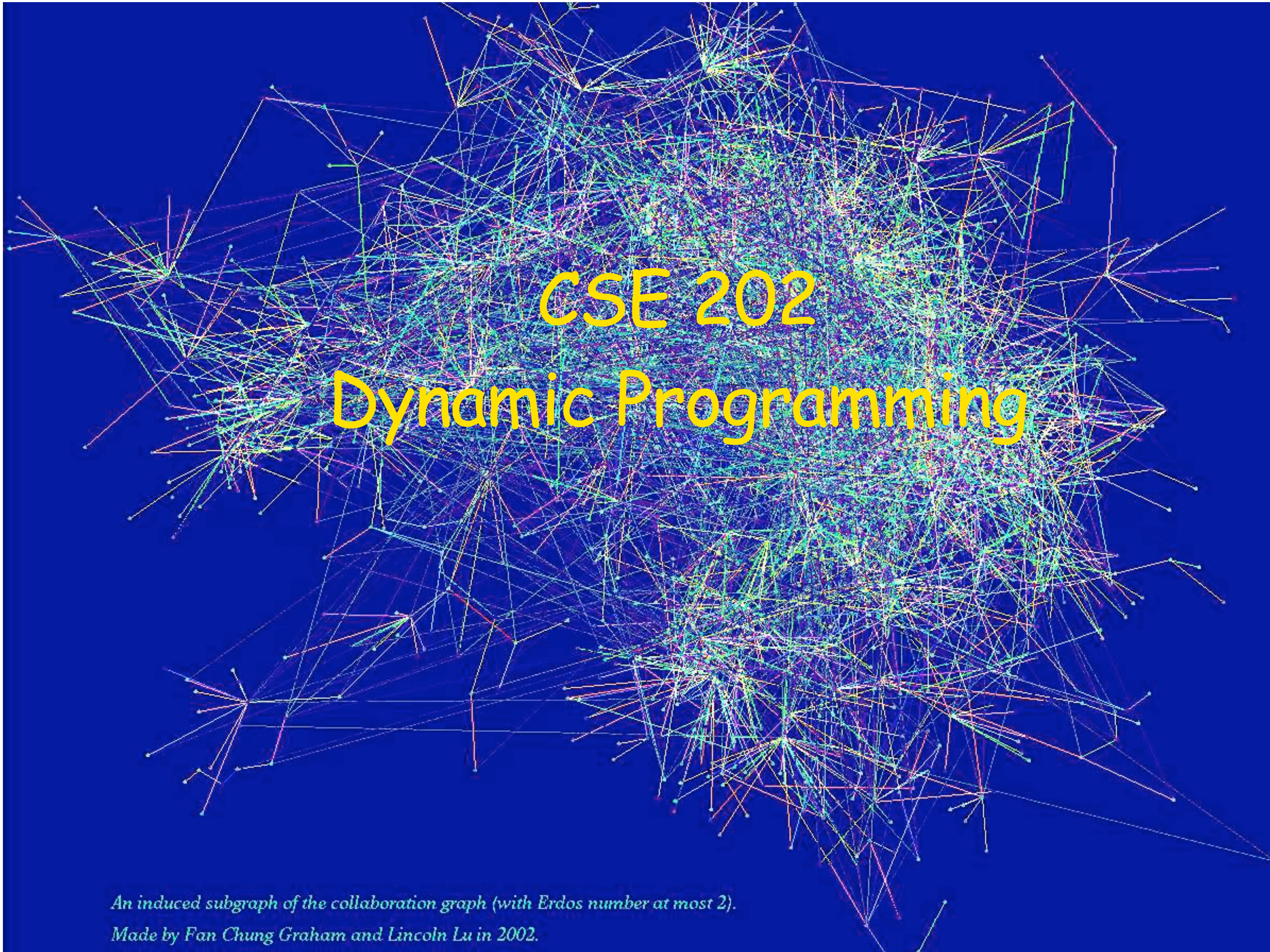
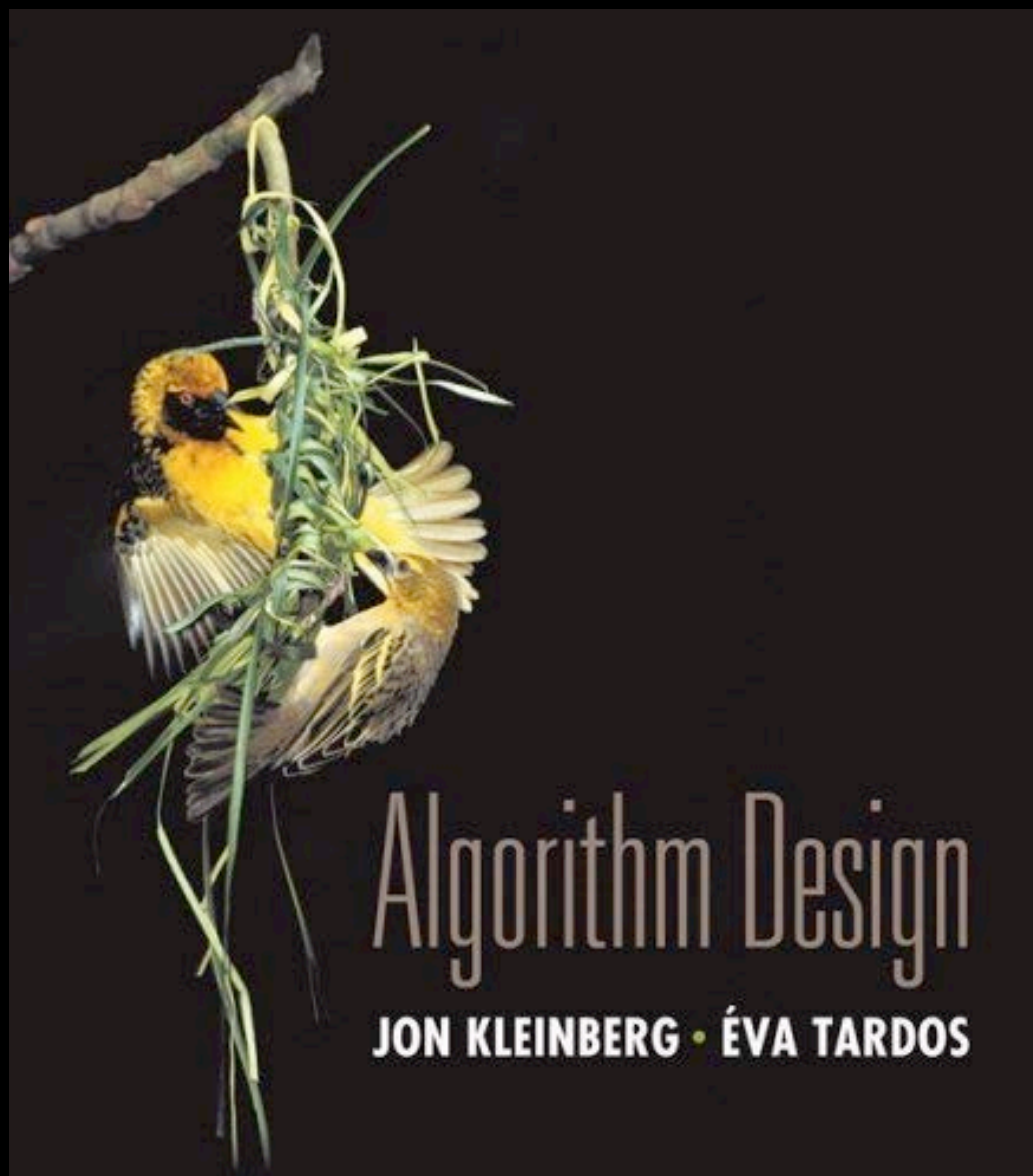
CSE 202
Dynamic Programming



An induced subgraph of the collaboration graph (with Erdos number at most 2).
Made by Fan Chung Graham and Lincoln Lu in 2002.



Chapter 6

Dynamic Programming

PEARSON Addison Wesley

Slides by Kevin Wayne.

Algorithm Design Paradigms

- **Exhaustive Search**
- **Greedy Algorithms:** Build a solution incrementally piece by piece
- **Divide and Conquer:** Divide into parts, solve each part, combine results
- **Dynamic Programming:** Divide into subtasks, perform subtask by size. Combine smaller subtasks to larger ones

Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Richard E. Bellman

From Wikipedia, the free encyclopedia

Richard Ernest Bellman (August 26, 1920 – March 19, 1984) was an applied mathematician, celebrated for his invention of dynamic programming in 1953, and important contributions in other fields of mathematics.

Richard E. Bellman	
Born	August 26, 1920 New York City, New York
Died	March 19, 1984 (aged 63)
Fields	Mathematics and Control theory
Alma mater	Princeton University University of Wisconsin–Madison Brooklyn College
Known for	Dynamic programming

Contents [show]

Biography [edit]

Bellman was born in 1920 in New York City, where his father John James Bellman ran a small grocery store on Bergen Street near Prospect Park in Brooklyn. Bellman completed his studies at Abraham Lincoln High School in 1937[1], and studied mathematics at Brooklyn College where he received a BA in 1941. He later earned an MA from the University of Wisconsin–Madison. During World War II he worked for a Theoretical Physics Division group in Los Alamos. In 1946 he received his Ph.D. at Princeton under the supervision of Solomon Lefschetz.[2]

He was a professor at the University of Southern California, a Fellow in the American Academy of Arts and Sciences (1975), and a member of the National Academy of Engineering (1977).

He was awarded the IEEE Medal of Honor in 1979, "for contributions to decision processes and control system theory, particularly the creation and application of dynamic programming". His key work is the Bellman equation.

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

Dynamic Programming (DP): A Simple Example

Problem: Compute the n-th Fibonacci number 1, 1, 2, 3, 5, 8, 13, 21, ...

Recursive Solution

```
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

Running Time:

$T(n) = T(n-1) + T(n-2) + 1$

Dynamic Programming (DP): A Simple Example

Problem: Compute the n-th Fibonacci number   1, 1, 2, 3, 5, 8, 13, 21, ...

Recursive Solution

```
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

Running time: $O(c^n)$

Running Time:

$T(n) = T(n-1) + T(n-2) + 1$

$T(n) = O(c^n)$

$c^2 - c - 1 = 0$ $c = 1.618...$

Dynamic Programming (DP): A Simple Example

Problem: Compute the n-th Fibonacci number   1, 1, 2, 3, 5, 8, 13, 21, ...

Recursive Solution

```
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

Running time: $O(c^n)$

Running Time:

$$T(n) = T(n-1) + T(n-2) + 1$$

$$T(n) = O(c^n)$$

$c^2 - c - 1 = 0$ $c = 1.618...$

Dynamic Programming Solution

```
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Dynamic Programming (DP): A Simple Example

Problem: Compute the n-th Fibonacci number

Recursive Solution

```
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

Running time: $O(c^n)$

Running Time:

$T(n) = T(n-1) + T(n-2) + 1$

$T(n) = O(c^n)$

Dynamic Programming Solution

```
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Running time: $O(n)$

Running Time:

$T(n) = O(n)$

Why does DP do better?

Problem: Compute the n-th Fibonacci number

Recursive Solution

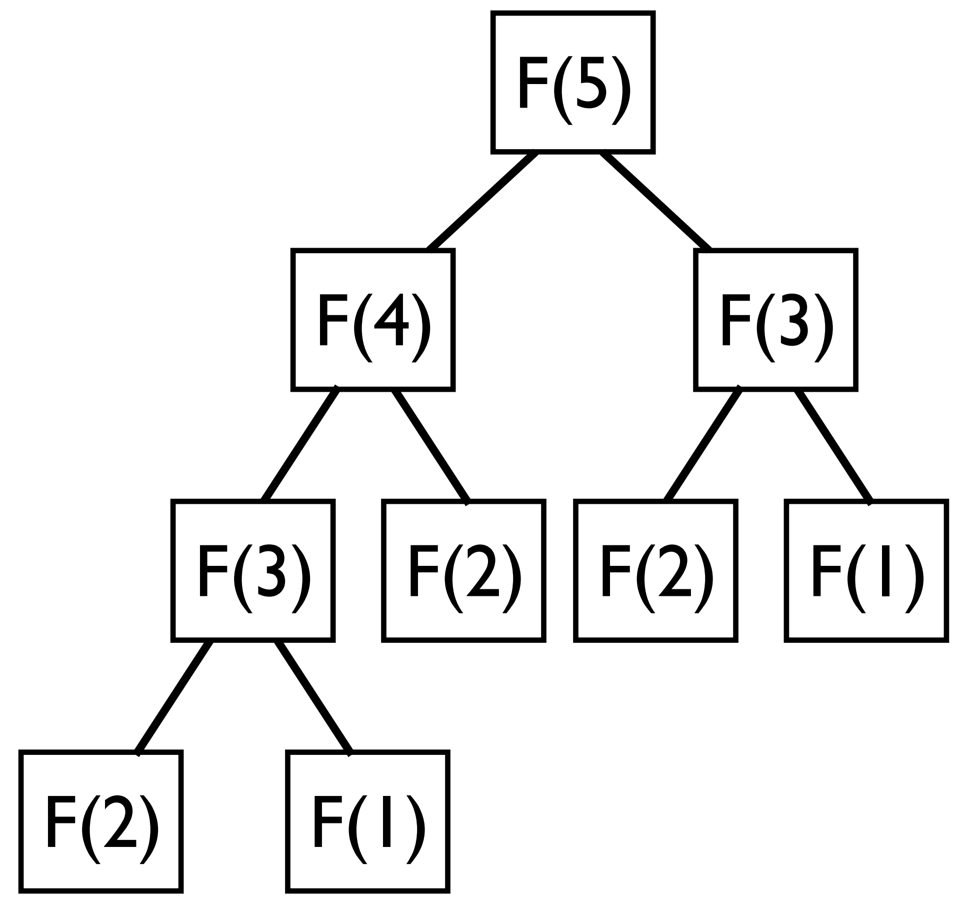
```
function Fib1(n)
if n = 1 return 1
if n = 2 return 1
return Fib1(n-1) + Fib1(n-2)
```

Running time: $O(c^n)$

Dynamic Programming Solution

```
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Running time: $O(n)$



Recursion Tree

Dynamic Programming

Main Steps:

1. Divide the problem into **subtasks**
2. Define the subtasks **recursively** (express larger subtasks in terms of smaller ones)
3. Find the **right order** for solving the subtasks (but do not solve them recursively!)

Dynamic Programming

Dynamic Programming Solution

```
function Fib2(n)
Create an array fib[1..n]
fib[1] = 1
fib[2] = 1
for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
return fib[n]
```

Running time: O(n)

Main Steps:

1. Divide the problem into **subtasks**: compute fib[i]

2. **Define the subtasks recursively** (express larger subtasks in terms of smaller ones)

3. Find the **right order** for solving the subtasks (i = 1,..,n)

Dynamic Programming

- String Reconstruction
- ...

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

Example:

x = anonymousarrayofletters : **True**

x = anhuymousarrayofhetters : **False**

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

Example:

x = anonymousarrayofletters : **True**

x = anhuymousarrayofhetters : **False**

STEP 1: Define subtask

S(k) = True if x[1..k] is a valid sequence of words

False otherwise

Output of algorithm = S(n)

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

Example:
x = anonymousarrayofletters : **True**
x = anhuymousarrayofhetters : **False**

STEP 1: Define subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

Output of algorithm = S(n)

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S																								

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S																								

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S																								

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S																								

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T																						

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T																					

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T																				

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T																			

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F																		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F																	

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F																

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F															

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T														

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T													

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F												

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F										

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T									

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F								

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T							

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F						

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F					

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T				

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F			

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word
Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n) [Do not solve recursively!]

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
False otherwise

STEP 2: Express Recursively

S(k) = True iff $\exists$ j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Algorithm:

```
S[0] = true
for k = 1 to n:
  S[k] = false
  for j = 1 to k:
    if S[j-1] and dict(x[j..k])
      S[k] = true
```

Reconstructing Document:

Define array D(1,..n):
If S(k) = true, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True,
and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1				-	-	-	-			-	-	-		-		-	-		-	-		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1			-	-	-	-			-	-	-		-		-	-		-	-		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1	2		-	-	-	-			-	-	-		-		-	-		-	-		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words

= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1	2	3	-	-	-	-			-	-	-		-		-	-		-	-		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is $j < k$ s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1	2	3	-	-	-	-	1		-	-	-		-		-	-		-	-		

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words

= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1	2	3	-	-	-	-	1	10	-	-	-	10	-	15	-	-	17	-	-	17	17

String reconstruction

Given: document x[1..n] : an array of characters
dictionary function dict(w): returns true if w is a valid word

Is x a sequence of valid words ?

STEP 1: Define Subtask

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

STEP 2: Express Recursively

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

STEP 3: Order of Subtasks

S(1), S(2), S(3), ..., S(n)

Reconstructing Document:

Define array D(1,..n):
If S(k) = True, then D(k) = starting position of the word that ends at x[k]

Reconstruct text by following these pointers.

x		A	N	O	N	Y	M	O	U	S	A	R	R	A	Y	O	F	L	E	T	T	E	R	S
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
S		T	T	T	T	F	F	F	F	T	T	F	F	F	T	F	T	F	F	T	F	F	T	T
D		1	1	2	3	-	-	-	-	1	10	-	-	-	10	-	15	-	-	17	-	-	17	17

How to Write a Dynamic Programming Solution

1. Define the subproblem (in words)

S(k) = True if x[1..k] is a valid sequence of words
= False otherwise

2. Write down recurrence relation

S(k) = True iff there is j < k s.t. S(j) is True, and x[j+1..k] is a valid word

3. Base case, Final solution, Order

Solution: S(n), Base Case: S(0)=0,
Evaluation Order: S(1),..,S(n)

4. Correctness Proof (by induction)

5. Running time analysis (usually easy, but not always)

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1									
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1									
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1									
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

$$S(i,j) = S(i-1,j-1) + 1, \text{ if } x[i] = y[j]$$
$$= \max(S(i-1,j), S(i,j-1)), \text{ ow}$$

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0								
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

		A	**C**	**T**	**G**	**G**	**C**	**T**	**A**	**G**
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0						
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1					
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2									
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0							
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1						
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1					
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1			
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2		
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3									
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

$$S(i,j) = S(i-1,j-1) + 1, \text{ if } x[i] = y[j]$$
$$= \max(S(i-1,j), S(i,j-1)), \text{ ow}$$

STEP 3: Order of subtasks

Row by row, top to bottom

		A	**C**	**T**	**G**	**G**	**C**	**T**	**A**	**G**
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0							
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1						
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2					
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2	2				
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

$$S(i,j) = S(i-1,j-1) + 1, \text{ if } x[i] = y[j]$$
$$= \max(S(i-1,j), S(i,j-1)), \text{ ow}$$

STEP 3: Order of subtasks

Row by row, top to bottom

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2	2	2			
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2	2	2	2		
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2	2	2	2	2	
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

Example:

x = A,C,G,T,A,G
y = G,T, C,C,A,C

LCS(x, y) = G,T,A

		A	C	T	G	G	C	T	A	G
	0	1	2	3	4	5	6	7	8	9
G	1	0	0	0	1	1	1	1	1	1
T	2	0	0	1	1	1	1	2	2	2
G	3	0	0	1	2	2	2	2	2	3
A	4									
C	5									
A	6									
G	7									
T	8									
T	9									

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Algorithm:

```
for i = 0 to n: S[i,0] = 0
for j = 0 to m: S[0,j] = 0
for i = 1 to n:
  for j = 1 to m:
    if x[i] = y[j]:
      S[i,j] =
        S[i-1,j-1] + 1
    else:
      S[i,j] = max{
        S[i-1,j], S[i,j-1]}
return S[n,m]
```

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Running Time: O(mn)

Algorithm:

```
for i = 0 to n: S[i,0] = 0
for j = 0 to m: S[0,j] = 0
for i = 1 to n:
  for j = 1 to m:
    if x[i] = y[j]:
      S[i,j] =
        S[i-1,j-1] + 1
    else:
      S[i,j] = max{
        S[i-1,j], S[i,j-1]}
return S[n,m]
```

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

Running Time: O(mn)

How to reconstruct the actual subsequence?

Algorithm:

```
for i = 0 to n: S[i,0] = 0
for j = 0 to m: S[0,j] = 0
for i = 1 to n:
  for j = 1 to m:
    if x[i] = y[j]:
      S[i,j] =
        S[i-1,j-1] + 1
    else:
      S[i,j] = max{
        S[i-1,j], S[i,j-1]}
return S[n,m]
```

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

$$S(i,j) = S(i-1,j-1) + 1, \text{ if } x[i] = y[j]$$
$$= \max(S(i-1,j), S(i,j-1)), \text{ ow}$$

STEP 3: Order of subtasks

Row by row, top to bottom

To reconstruct LCS:

Define L(i, j):

$$L(i, j) = (i - 1, j - 1), \text{ if } x[i] = y[j]$$
$$= (i - 1, j), \quad \text{ow if } S(i-1,j) > S(i, j-1)$$
$$= (i, j - 1), \quad \text{ow}$$

Reconstruct LCS by following the L(i,j) pointers, starting with L(m,n)

Longest Common Subsequence (LCS)

Problem: Given two sequences x[1..m] and y[1..n], find their longest common subsequence

STEP 1: Define subtasks

S(i,j) = Length of LCS of x[1..i] and y[1..j]

Output of algorithm = S(n,m)

STEP 2: Express recursively

S(i,j) = S(i-1,j-1) + 1, if x[i] = y[j]
= max(S(i-1,j), S(i,j-1)), ow

STEP 3: Order of subtasks

Row by row, top to bottom

To reconstruct LCS:

Define L(i, j):

L(i, j) = (i - 1, j - 1), if x[i] = y[j]
= (i - 1, j), ow if S(i-1,j) > S(i, j-1)
= (i, j - 1), ow

Reconstruct LCS by following the L(i,j) pointers, starting with L(m,n)

Running Time: O(mn)

Dynamic Programming vs Divide and Conquer

Divide-and-conquer

A problem of size n is decomposed into a few subproblems which are significantly smaller (e.g. n/2, 3n/4,...)

Therefore, size of subproblems decreases geometrically.

eg. n, n/2, n/4, n/8, etc

Use a recursive algorithm.

Dynamic programming

A problem of size n is expressed in terms of subproblems that are not much smaller (e.g. n-1, n-2,...)

A recursive algorithm would take exp. time.

Saving grace: in total, there are only polynomially many subproblems.

Avoid recursion and instead solve the subproblems one-by-one, saving the answers in a table, in a clever explicit order.

DP: Common Subtasks

Case 1: Input: $x_1, x_2,..,x_n$ Subproblem: $x_1, .., x_i$.

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \; x_7 \quad x_8 \quad x_9 \quad x_{10}$

DP: Common Subtasks

Case 1: Input: $x_1, x_2,..,x_n$ Subproblem: $x_1, .., x_i$.

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$

Case 2: Input: $x_1, x_2,..,x_n$ and $y_1, y_2,..,y_m$ Subproblem: $x_1, .., x_i$ and $y_1, y_2,..,y_j$

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$

$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$

DP: Common Subtasks

Case 1: Input: $x_1, x_2, .., x_n$ Subproblem: $x_1, .., x_i$.

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6}\ x_7 \quad x_8 \quad x_9 \quad x_{10}$

Case 2: Input: $x_1, x_2, .., x_n$ and $y_1, y_2, .., y_m$ Subproblem: $x_1, .., x_i$ and $y_1, y_2, .., y_j$

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6}\ x_7 \quad x_8 \quad x_9 \quad x_{10}$

$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5}\ y_6 \quad y_7 \quad y_8$

Case 3: Input: $x_1, x_2, .., x_n$. Subproblem: $x_i, .., x_j$

$x_1 \quad x_2\ \boxed{x_3 \quad x_4 \quad x_5 \quad x_6}\ x_7 \quad x_8 \quad x_9 \quad x_{10}$

DP: Common Subtasks

Case 1: Input: $x_1, x_2,..,x_n$ Subproblem: $x_1, .., x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \; x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Case 2: Input: $x_1, x_2,..,x_n$ and $y_1, y_2,..,y_m$ Subproblem: $x_1, .., x_i$ and $y_1, y_2,..,y_j$

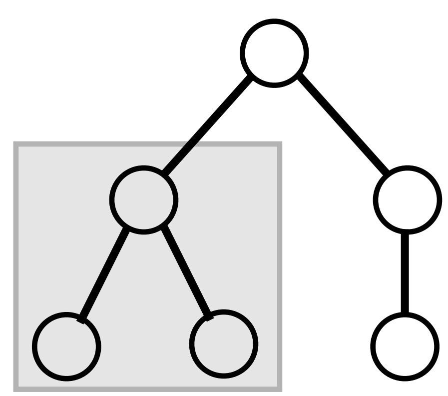
$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \; x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \; y_6 \quad y_7 \quad y_8$$

Case 3: Input: $x_1, x_2,..,x_n$. Subproblem: $x_i, .., x_j$

$$x_1 \quad x_2 \; \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \; x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Case 4: Input: a rooted tree. Subproblem: a subtree

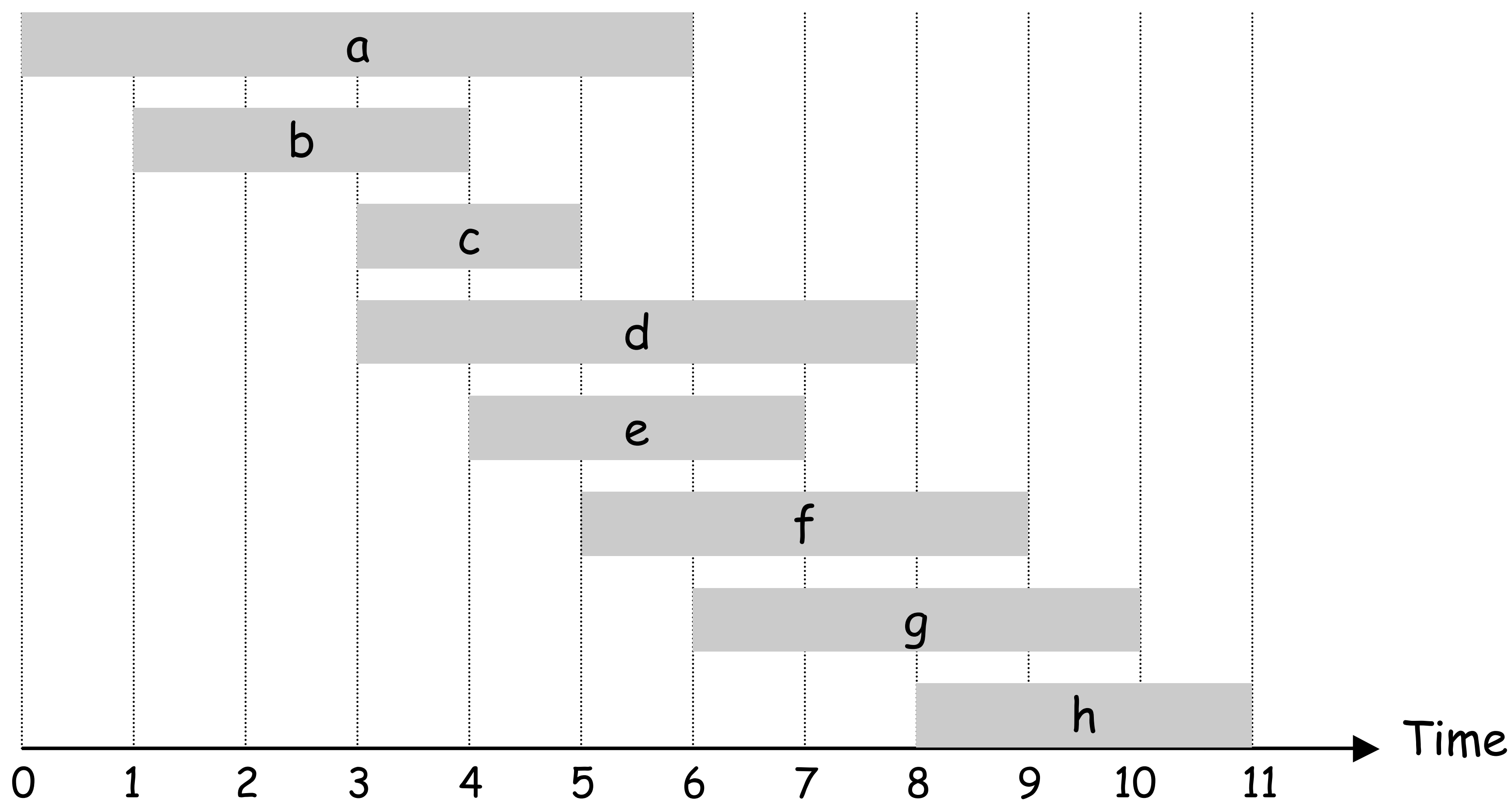


6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

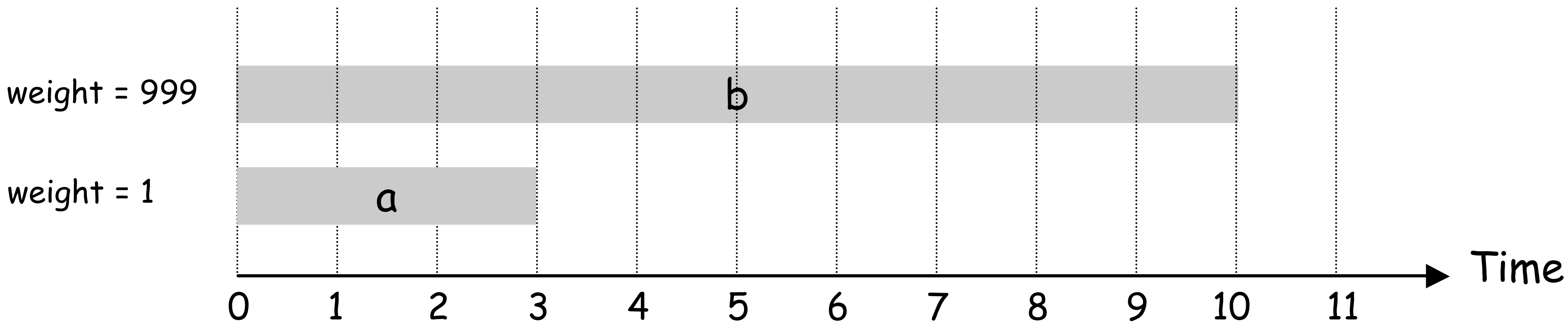


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

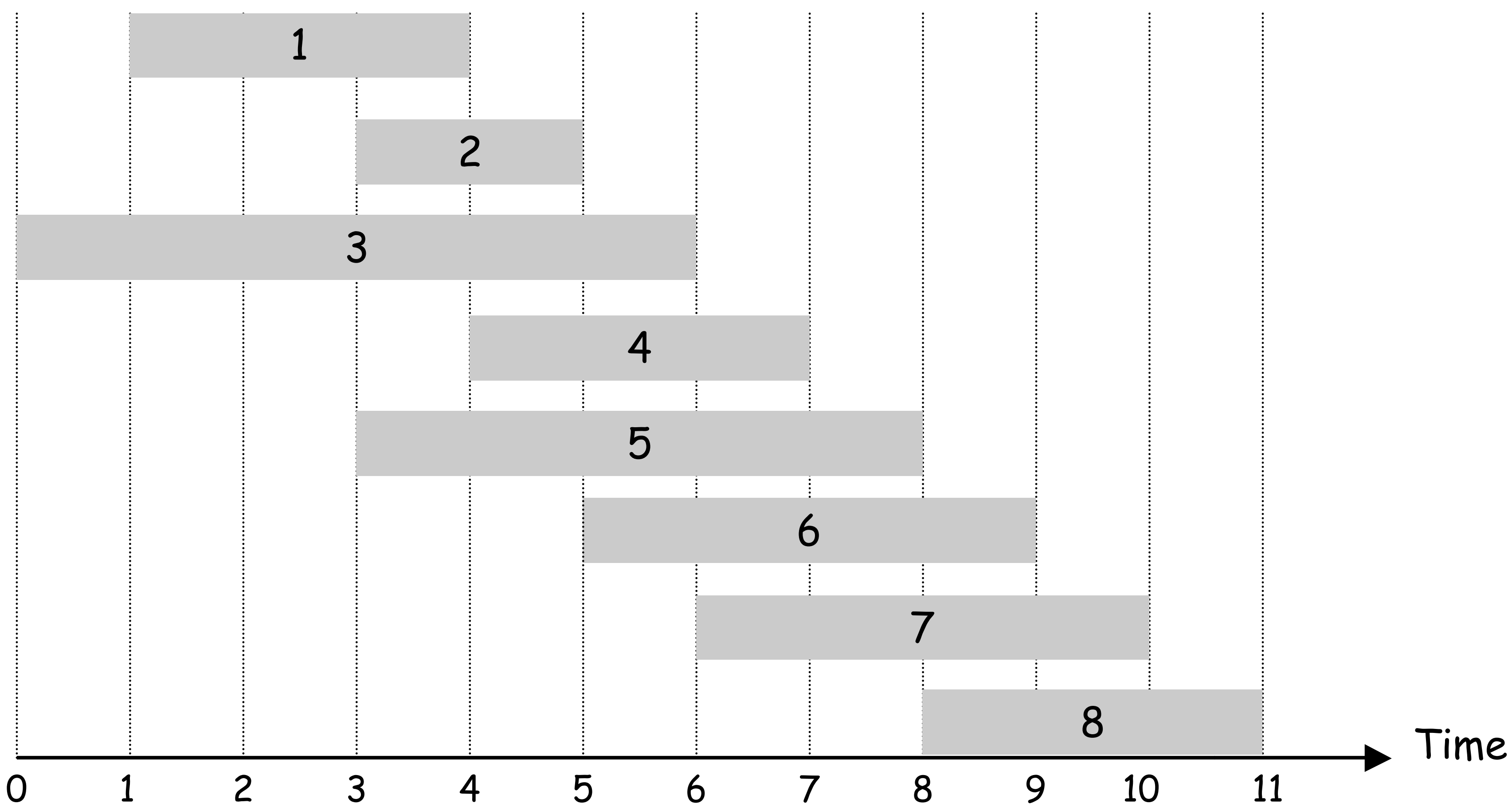
Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

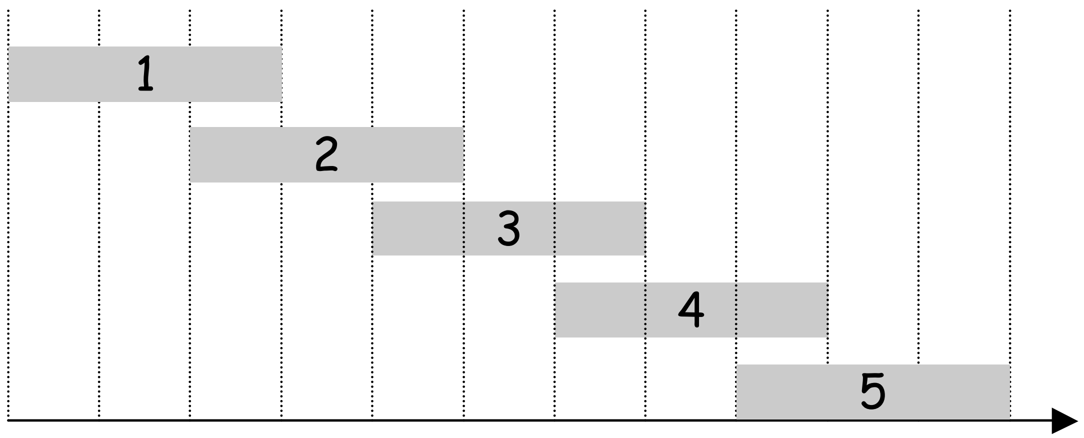
Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

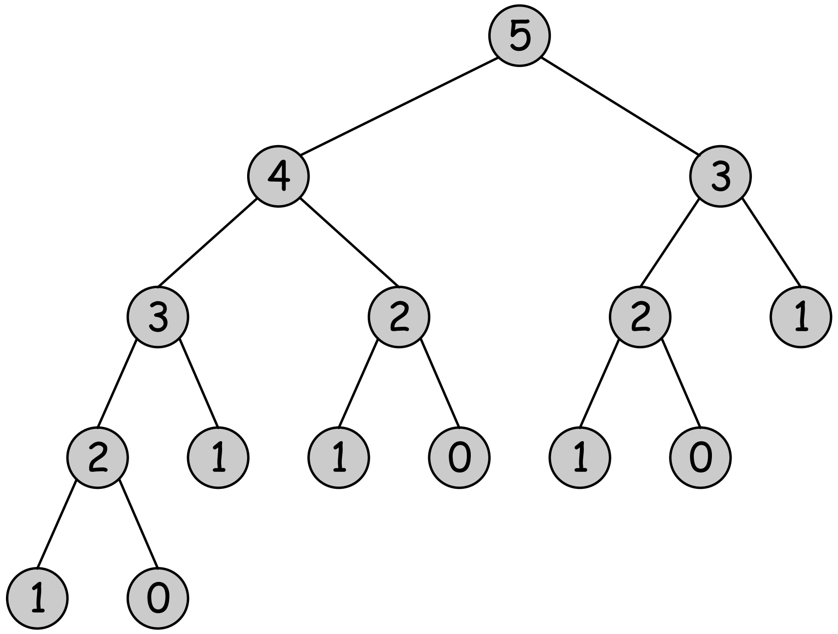
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty    ← global array
M[j] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing $p(\cdot)$: O(n) after sorting by start time.

- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \le n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.

- Overall running time of `M-Compute-Opt(n)` is O(n). ▪

Remark. O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

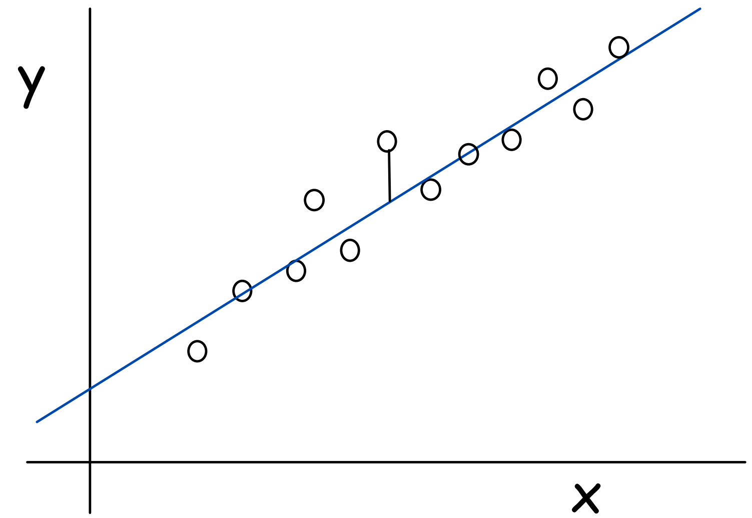
6.3 Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line y = ax + b that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solution. Calculus $\Rightarrow$ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

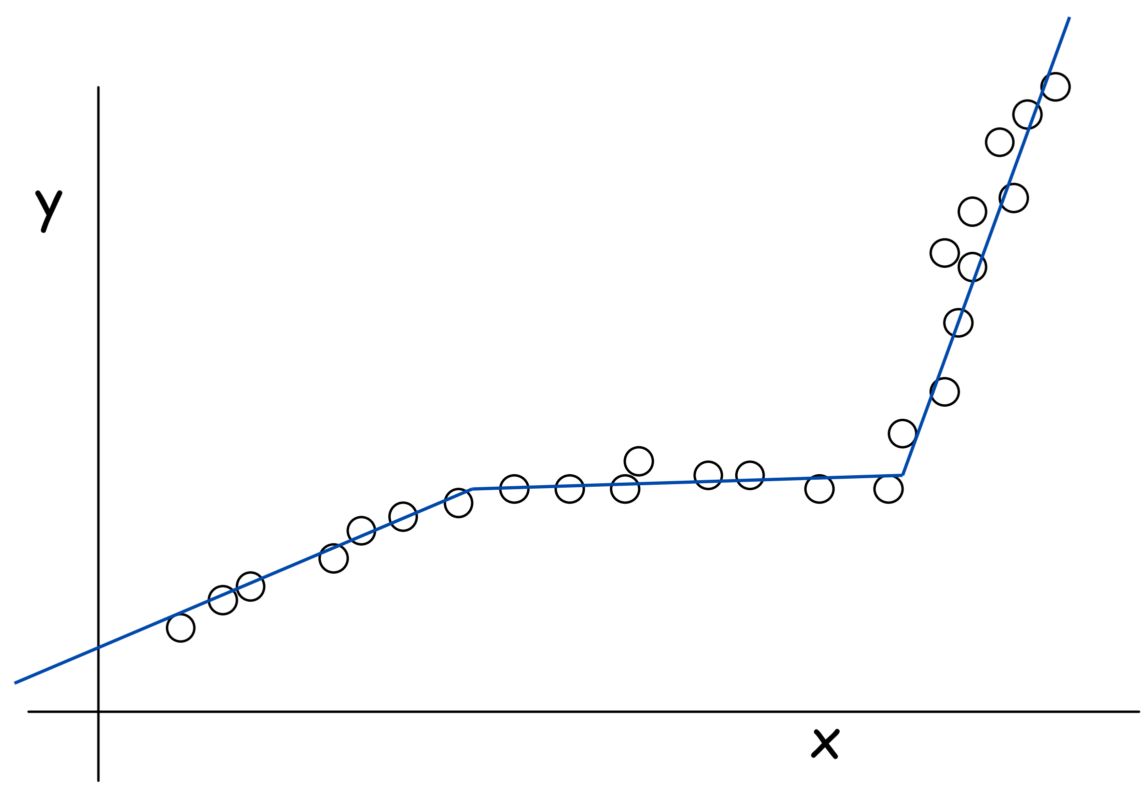
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What's a reasonable choice for $f(x)$ to balance accuracy and parsimony?

goodness of fit

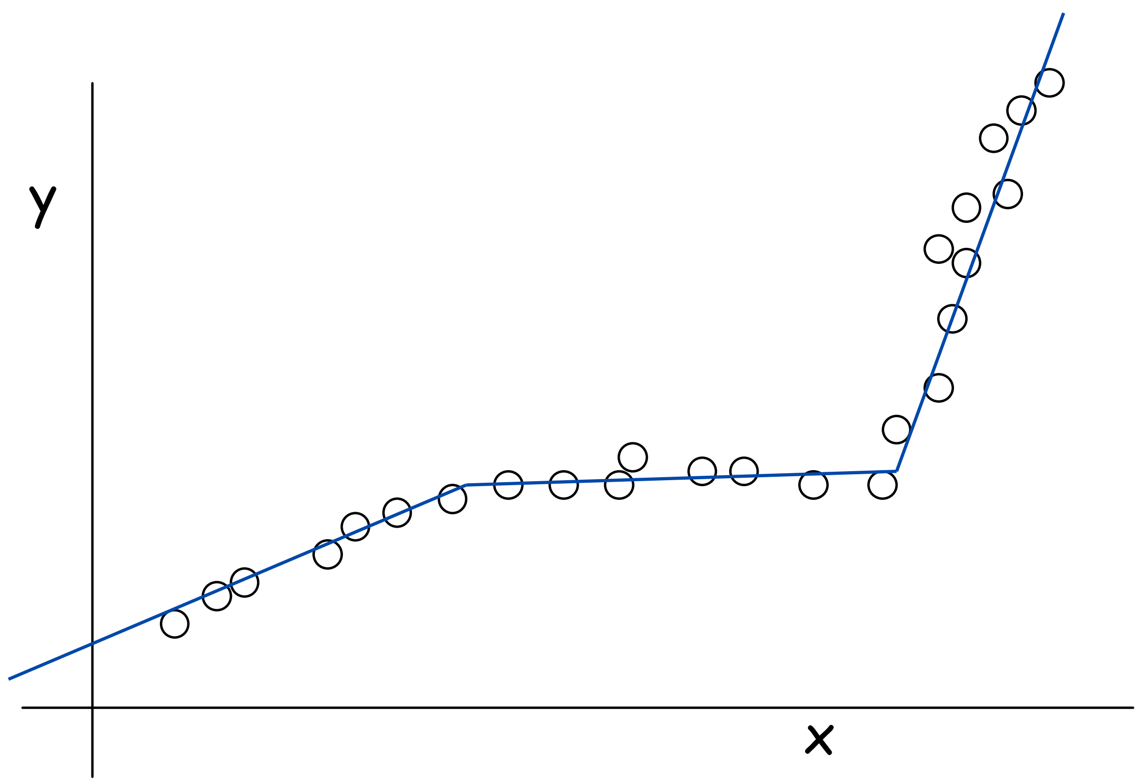
number of lines



Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors E in each segment
  - the number of lines L
- Tradeoff function: $E + c L$, for some constant $c > 0$.



Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = minimum cost for points p_1, p_{i+1} , . . . , p_j.
- e(i, j) = minimum sum of squares for points p_i, p_{i+1} , . . . , p_j.

To compute OPT(j):

- Last segment uses points p_i, p_{i+1} , . . . , p_j for some i.
- Cost = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{ \; e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

Running time. $O(n^3)$. ← can be improved to $O(n^2)$ by pre-computing various statistics

- Bottleneck = computing $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ per pair using previous formula.