

## 6. DYNAMIC PROGRAMMING I

---

- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ *RNA secondary structure*

Lecture slides by Kevin Wayne

Copyright © 2005 Pearson–Addison Wesley

<http://www.cs.princeton.edu/~wayne/kleinberg-tardos>

# Algorithmic paradigms

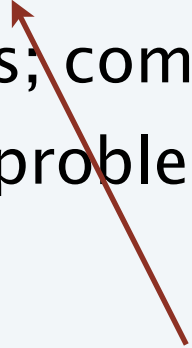
---

**Greed.** Process the input in some order, myopically making irrevocable decisions.

**Divide-and-conquer.** Break up a problem into **independent** subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of **overlapping** subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

fancy name for  
caching intermediate results  
in a table for later reuse



# Dynamic programming history

---

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

## Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a “dynamic” adjective to avoid conflict.



## THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time  $t$  is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

# Dynamic programming applications

---

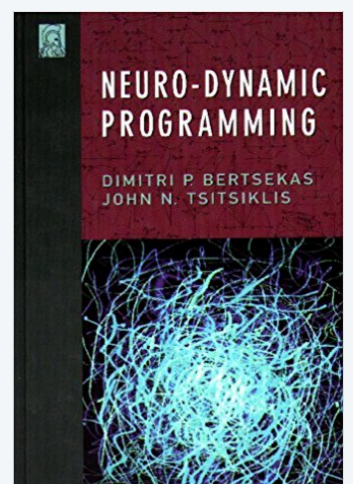
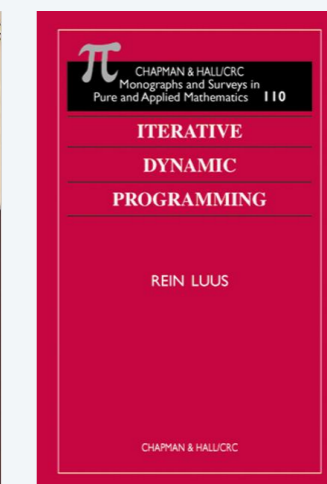
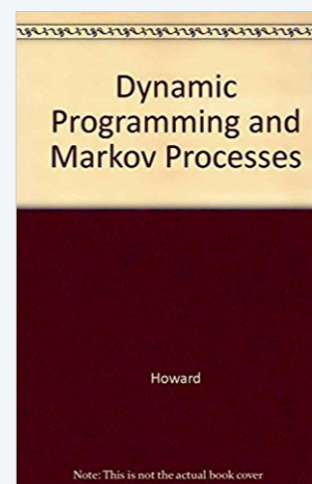
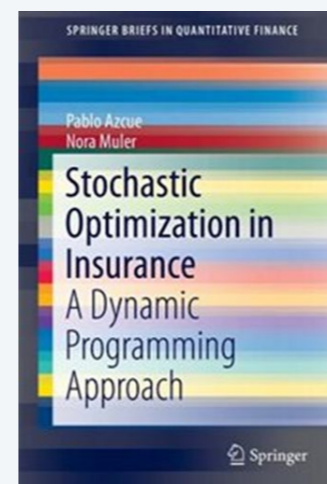
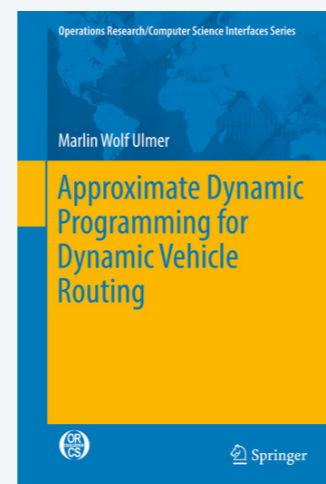
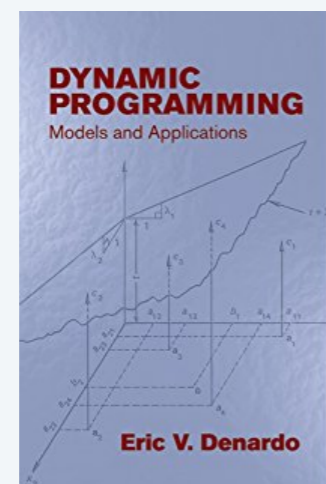
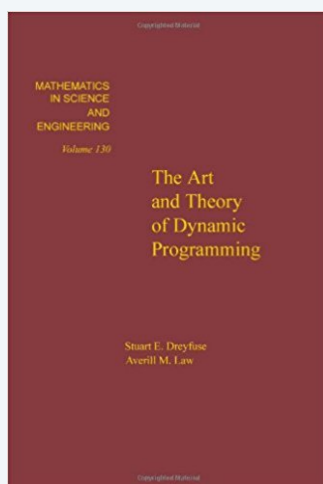
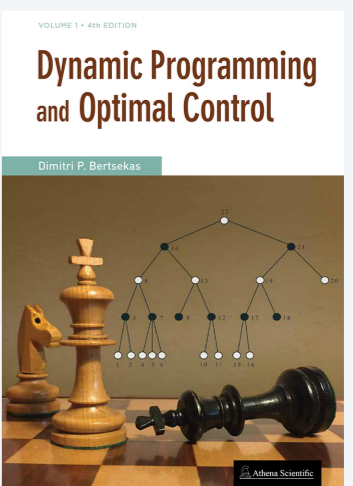
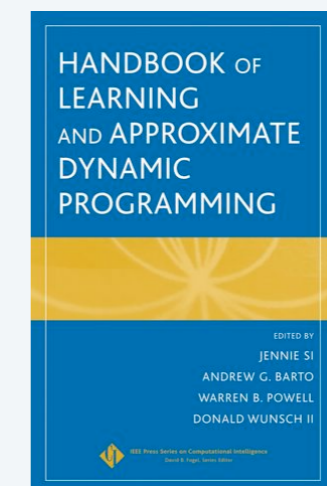
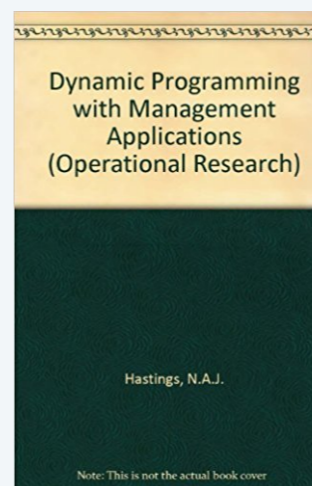
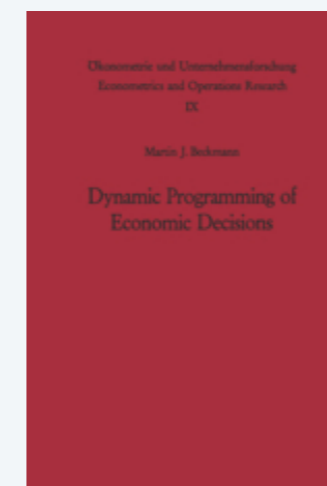
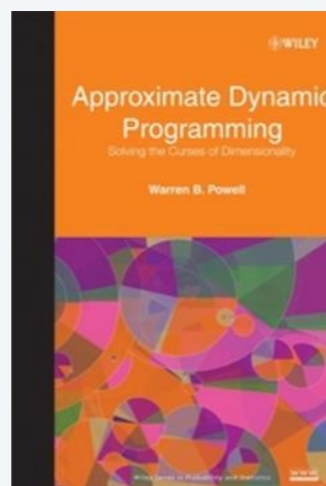
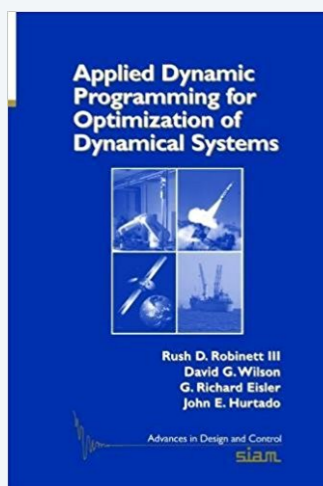
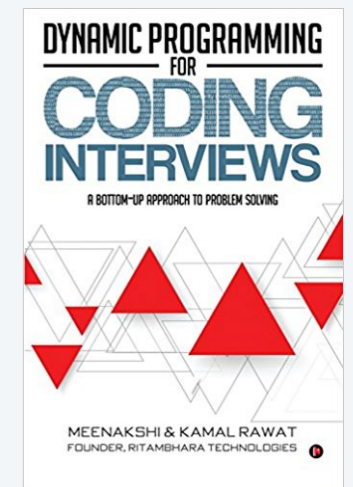
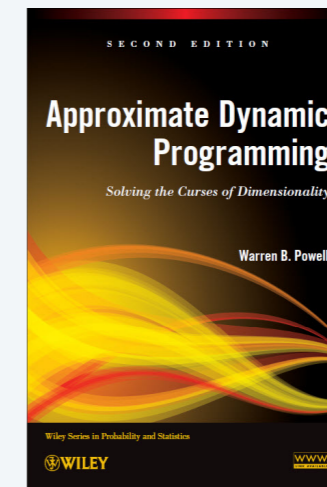
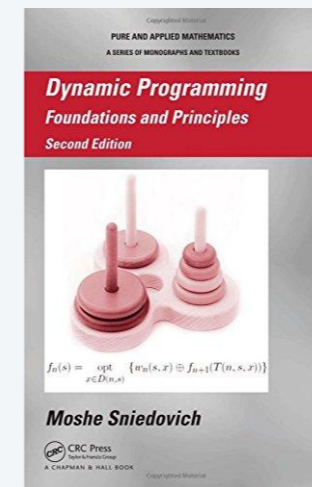
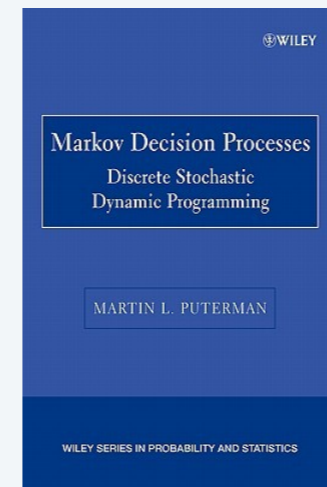
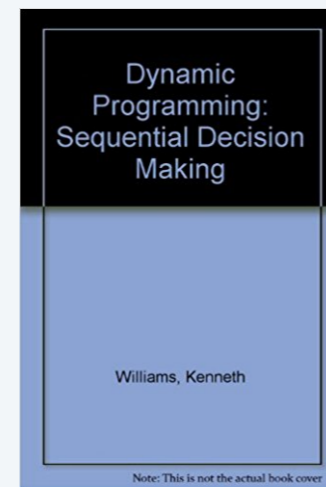
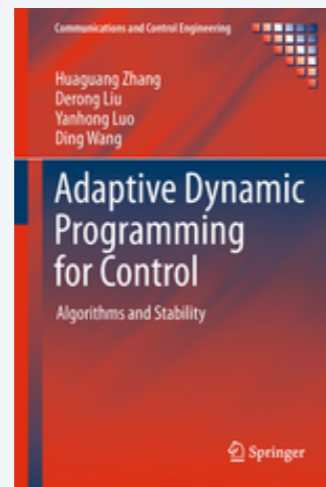
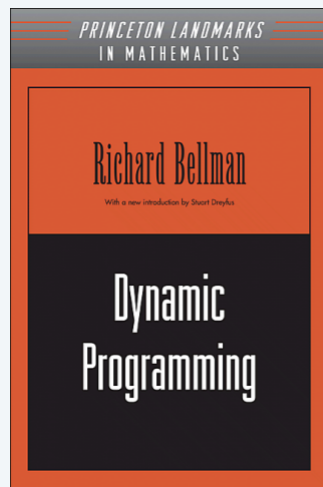
## Application areas.

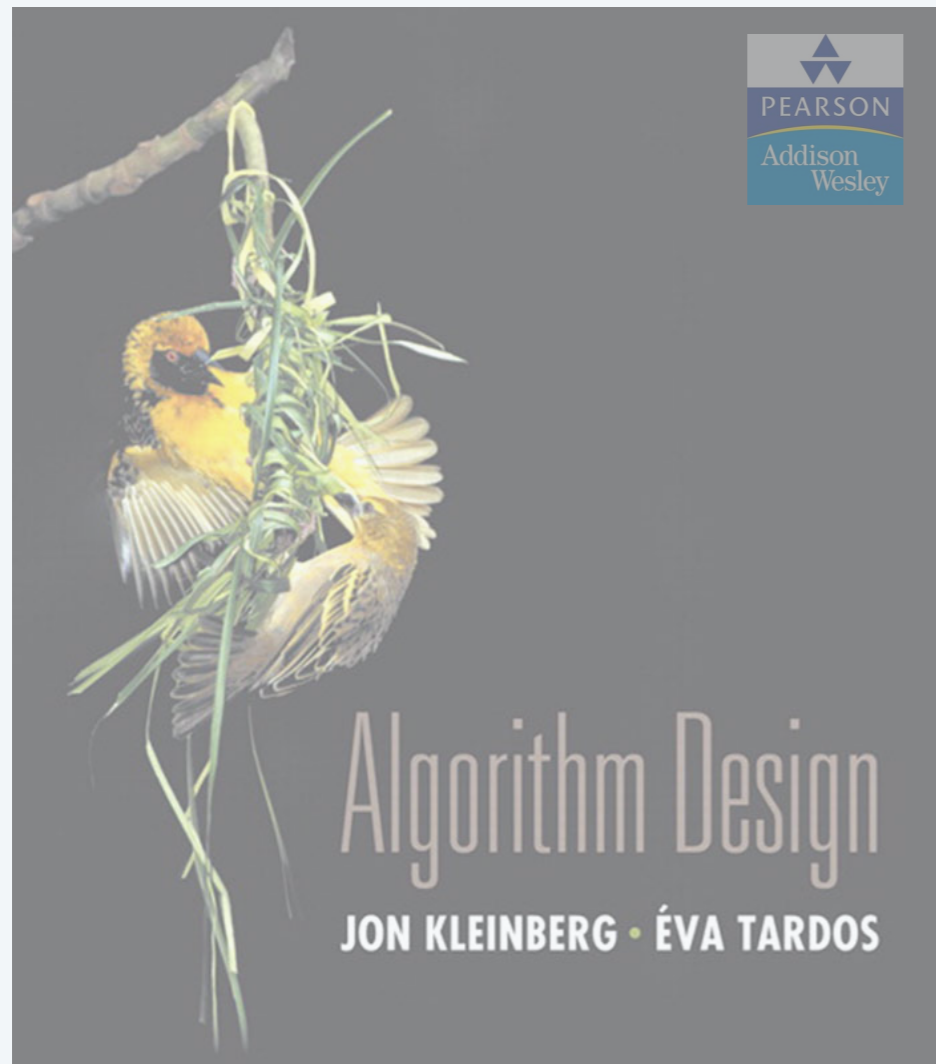
- Computer science: AI, compilers, systems, graphics, theory, ....
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

## Some famous dynamic programming algorithms.

- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in  $T_{E}X$ .
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

# Dynamic programming books





SECTIONS 6.1–6.2

## 6. DYNAMIC PROGRAMMING I

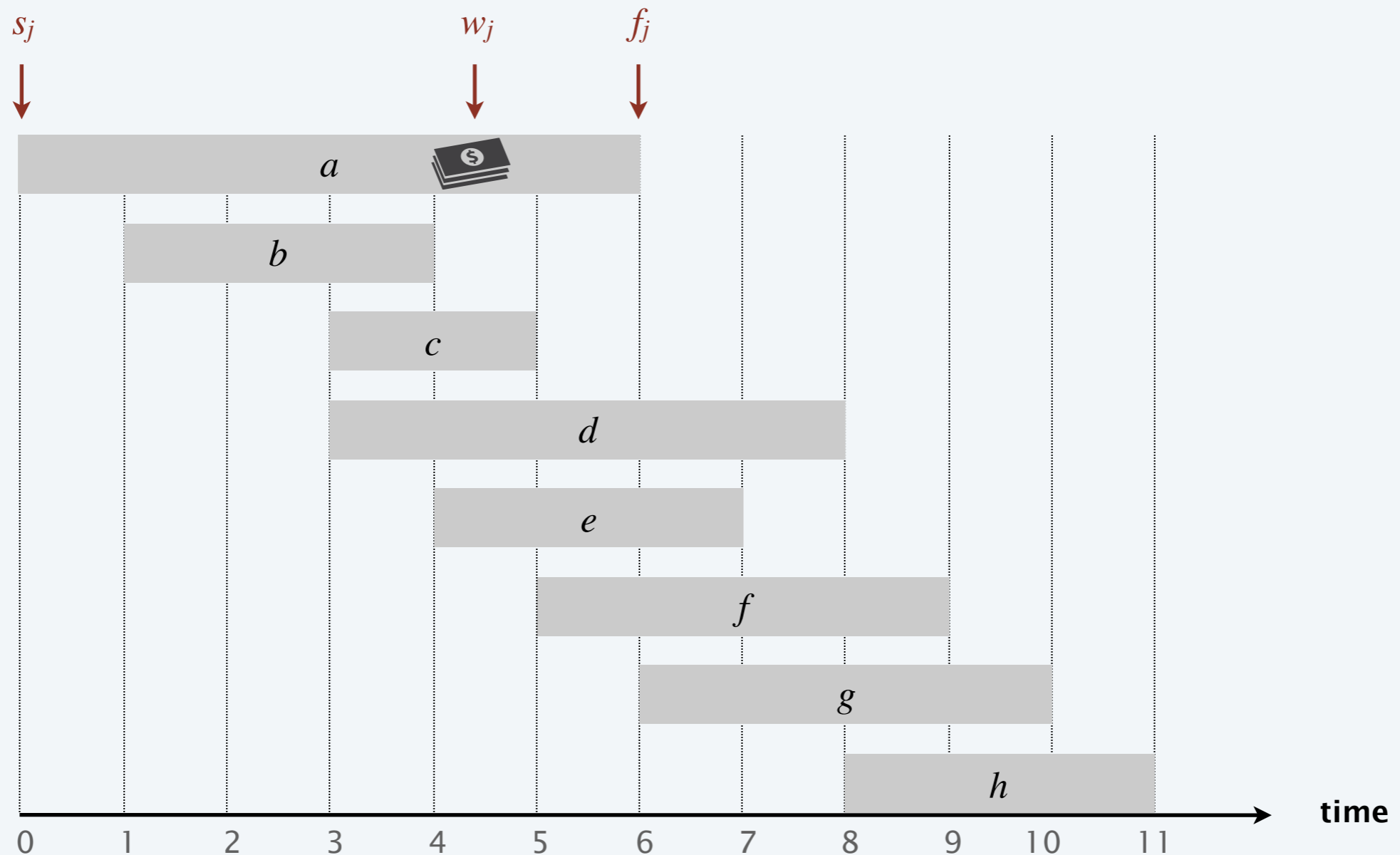
---

- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ *RNA secondary structure*

# Weighted interval scheduling

---

- Job  $j$  starts at  $s_j$ , finishes at  $f_j$ , and has weight  $w_j > 0$ .
- Two jobs are **compatible** if they don't overlap.
- Goal: find max-weight subset of mutually compatible jobs.



# Earliest-finish-time first algorithm

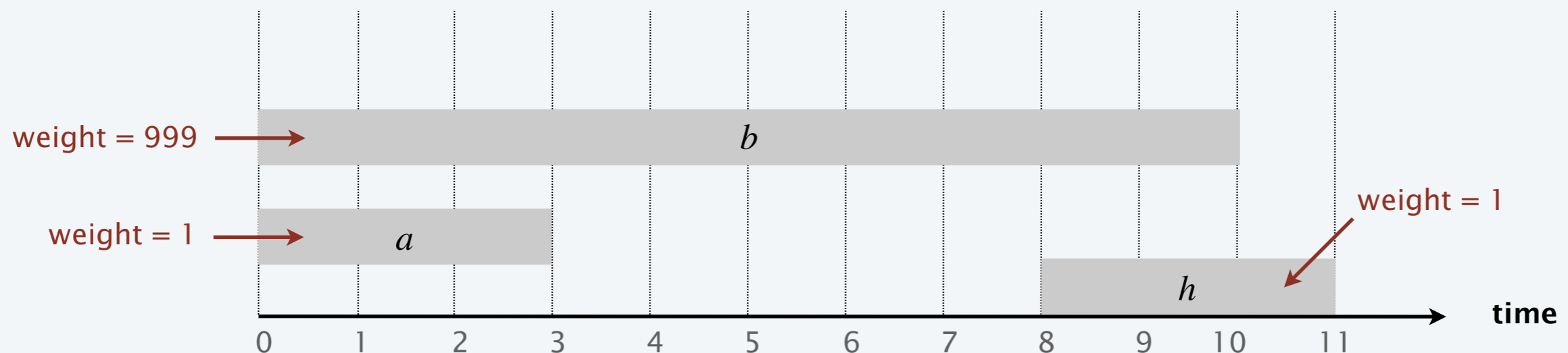
---

## Earliest finish-time first.

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

**Recall.** Greedy algorithm is correct if all weights are 1.

**Observation.** Greedy algorithm fails spectacularly for weighted version.






# Weighted interval scheduling

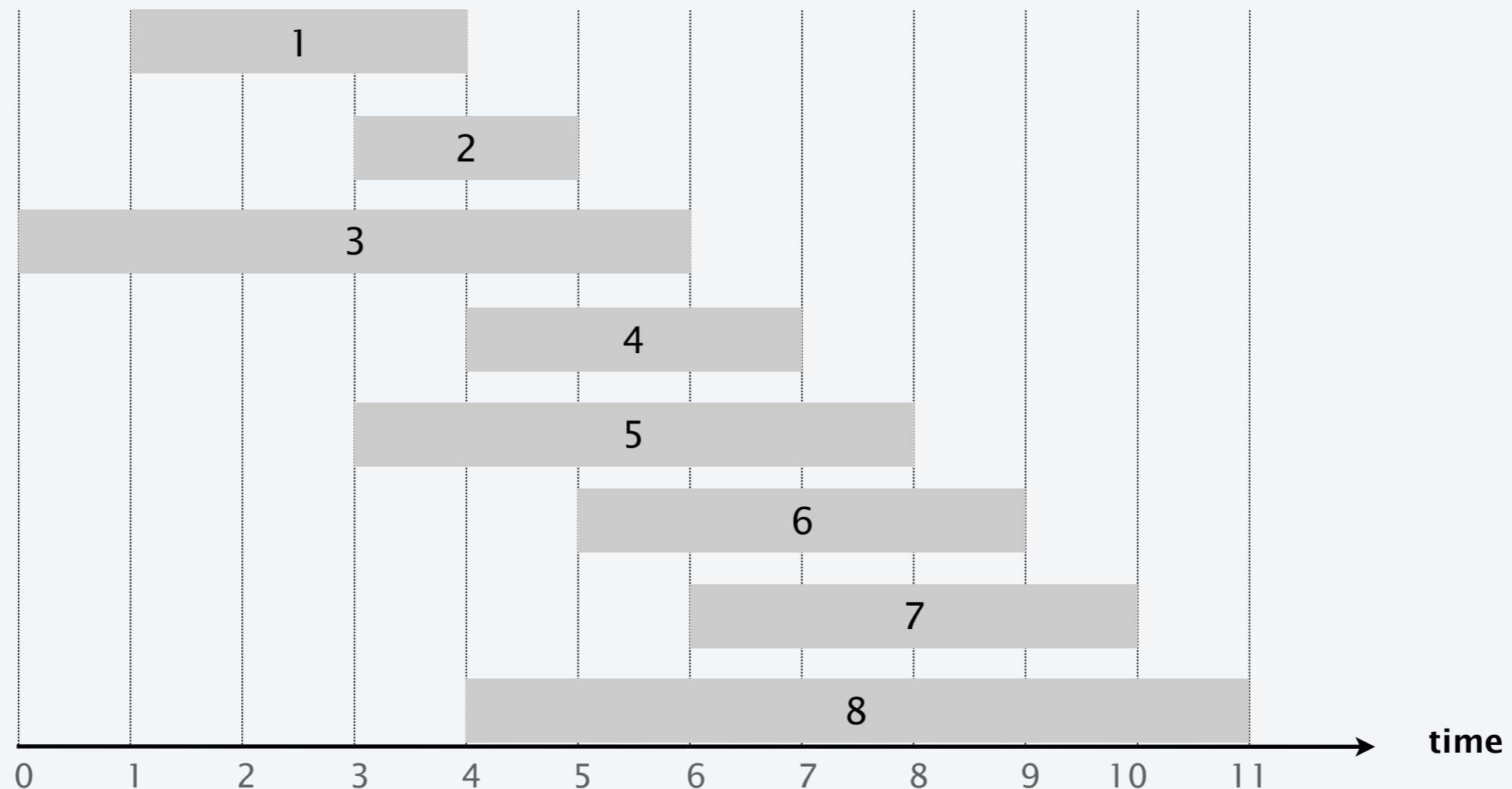
---

**Convention.** Jobs are in ascending order of finish time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Def.**  $p(j)$  = largest index  $i < j$  such that job  $i$  is compatible with  $j$ .

**Ex.**  $p(8) = 1, p(7) = 3, p(2) = 0$ .

  $i$  is leftmost interval that ends before  $j$  begins



# Dynamic programming: binary choice

---

**Def.**  $OPT(j)$  = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs  $1, 2, \dots, j$ .


**Goal.**  $OPT(n)$  = max weight of any subset of mutually compatible jobs.

**Case 1.**  $OPT(j)$  does not select job  $j$ .

- Must be an optimal solution to problem consisting of remaining jobs  $1, 2, \dots, j - 1$ .

**Case 2.**  $OPT(j)$  selects job  $j$ .

- Collect profit  $w_j$ .
- Can't use incompatible jobs  $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$ .
- Must include optimal solution to problem consisting of remaining compatible jobs  $1, 2, \dots, p(j)$ .

 optimal substructure property  
(proof via exchange argument)

**Bellman equation.** 
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ OPT(j - 1), w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$$

# Weighted interval scheduling: brute force

---

**BRUTE-FORCE** ( $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$ )

---

Sort jobs by finish time and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Compute  $p[1], p[2], \dots, p[n]$  via binary search.

**RETURN** COMPUTE-OPT( $n$ ).

**COMPUTE-OPT**( $j$ )

---

**IF** ( $j = 0$ )

**RETURN** 0.

**ELSE**

**RETURN**  $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$ .



What is running time of `COMPUTE-OPT( $n$ )` in the worst case?

- A.  $\Theta(n \log n)$
- B.  $\Theta(n^2)$
- C.  $\Theta(1.618^n)$
- D.  $\Theta(2^n)$

```
COMPUTE-OPT( $j$ )
```

```
IF ( $j = 0$ )
```

```
    RETURN 0.
```

```
ELSE
```

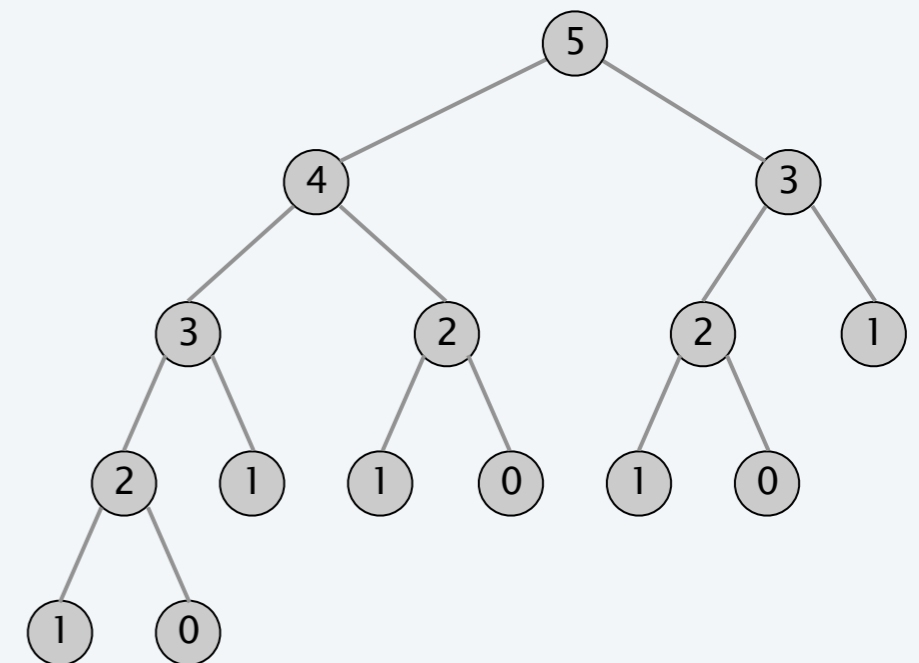
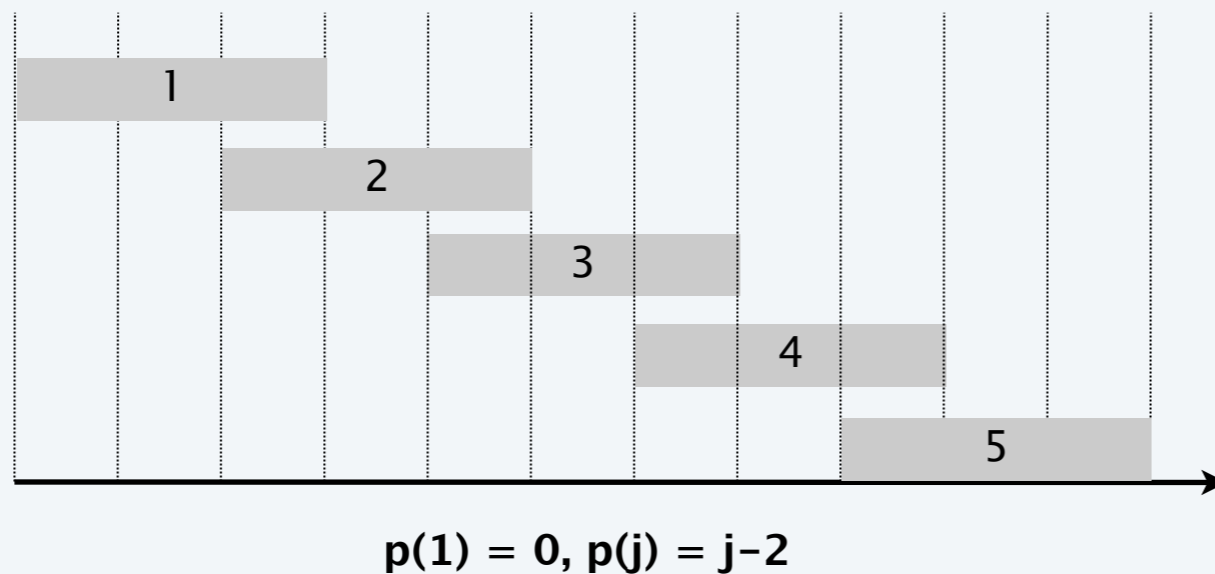
```
    RETURN max {COMPUTE-OPT( $j-1$ ),  $w_j +$  COMPUTE-OPT( $p[j]$ ) }.
```

# Weighted interval scheduling: brute force

---

**Observation.** Recursive algorithm is spectacularly slow because of overlapping subproblems  $\Rightarrow$  exponential-time algorithm.

**Ex.** Number of recursive calls for family of “layered” instances grows like Fibonacci sequence.



recursion tree

# Weighted interval scheduling: memoization

---

## Top-down dynamic programming (memoization).

- Cache result of subproblem  $j$  in  $M[j]$ .
- Use  $M[j]$  to avoid solving subproblem  $j$  more than once.

**TOP-DOWN**( $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$ )

---

Sort jobs by finish time and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Compute  $p[1], p[2], \dots, p[n]$  via binary search.

$M[0] \leftarrow 0.$   global array

**RETURN** M-COMPUTE-OPT( $n$ ).

**M-COMPUTE-OPT**( $j$ )

---

**IF** ( $M[j]$  is uninitialized)

$M[j] \leftarrow \max \{ \text{M-COMPUTE-OPT}(j-1), w_j + \text{M-COMPUTE-OPT}(p[j]) \}.$

**RETURN**  $M[j]$ .

# Weighted interval scheduling: running time

---

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

**Pf.**

- Sort by finish time:  $O(n \log n)$  via mergesort.
- Compute  $p[j]$  for each  $j$ :  $O(n \log n)$  via binary search.
- M-COMPUTE-OPT( $j$ ): each invocation takes  $O(1)$  time and either
  - (1) returns an initialized value  $M[j]$
  - (2) initializes  $M[j]$  and makes two recursive calls
- Progress measure  $\Phi = \#$  initialized entries among  $M[1..n]$ .
  - initially  $\Phi = 0$ ; throughout  $\Phi \leq n$ .
  - (2) increases  $\Phi$  by 1  $\Rightarrow \leq 2n$  recursive calls.
- Overall running time of M-COMPUTE-OPT( $n$ ) is  $O(n)$ . ■

Those who cannot remember the  
past are condemned to repeat it.

- Dynamic Programming



# Weighted interval scheduling: finding a solution

---

Q. DP algorithm computes optimal value. How to find optimal solution?

A. Make a second pass by calling FIND-SOLUTION( $n$ ).

```
FIND-SOLUTION( $j$ )
```

---

```
IF ( $j = 0$ )
```

```
    RETURN  $\emptyset$ .
```

```
ELSE IF ( $w_j + M[p[j]] > M[j-1]$ )
```

```
    RETURN  $\{j\} \cup \text{FIND-SOLUTION}(p[j])$ .
```

```
ELSE
```

```
    RETURN FIND-SOLUTION( $j-1$ ).
```

$$M[j] = \max \{ M[j-1], w_j + M[p[j]] \}.$$

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

# Weighted interval scheduling: bottom-up dynamic programming

---

Bottom-up dynamic programming. Unwind recursion.

**BOTTOM-UP**( $n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$ )

---

Sort jobs by finish time and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

Compute  $p[1], p[2], \dots, p[n]$ .

$M[0] \leftarrow 0$ .

previously computed values

**FOR**  $j = 1$  **TO**  $n$

$M[j] \leftarrow \max \{ M[j-1], w_j + M[p[j]] \}$ .

---

**Running time.** The bottom-up version takes  $O(n \log n)$  time.

# MAXIMUM SUBARRAY PROBLEM

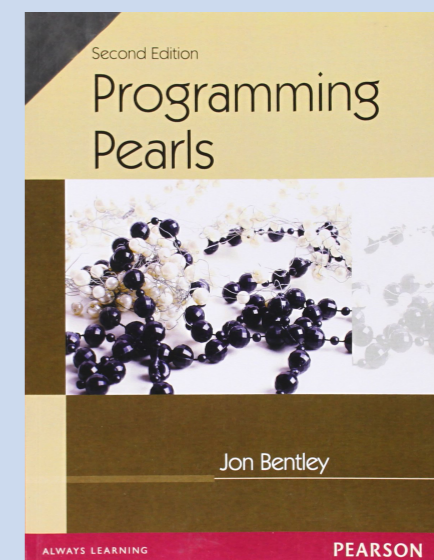


**Goal.** Given an array  $x$  of  $n$  integer (positive or negative), find a contiguous subarray whose sum is maximum.

12	5	-1	31	-61	59	26	-53	58	97	-93	-23	84	-15	6
----	---	----	----	-----	----	----	-----	----	----	-----	-----	----	-----	---

187

**Applications.** Computer vision, data mining, genomic sequence analysis, technical job interviews, ....



# MAXIMUM RECTANGLE PROBLEM

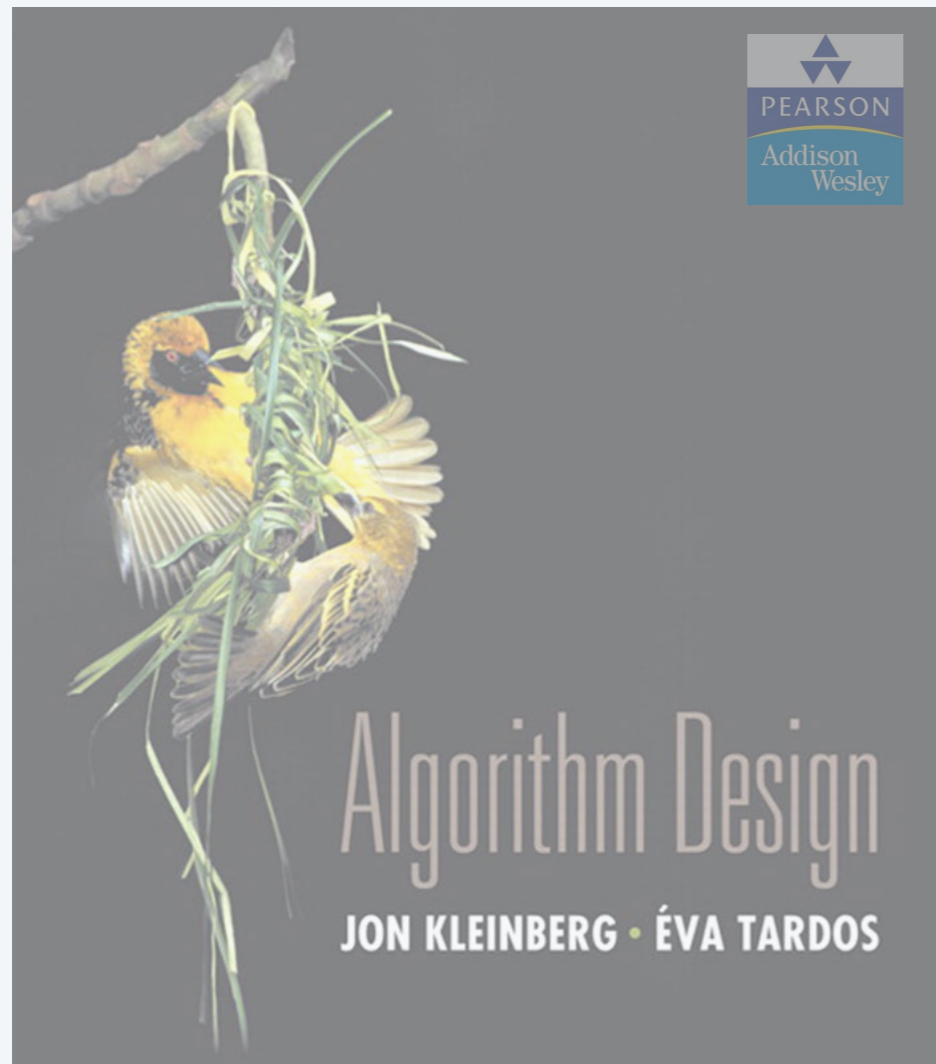


**Goal.** Given an  $n$ -by- $n$  matrix  $A$ , find a rectangle whose sum is maximum.

$$A = \begin{bmatrix} -2 & 5 & 0 & -5 & -2 & 2 & -3 \\ 4 & -3 & -1 & 3 & 2 & 1 & -1 \\ -5 & 6 & 3 & -5 & -1 & -4 & -2 \\ -1 & -1 & 3 & -1 & 4 & 1 & 1 \\ 3 & -3 & 2 & 0 & 3 & -3 & -2 \\ -2 & 1 & -2 & 1 & 1 & 3 & -1 \\ 2 & -4 & 0 & 1 & 0 & -3 & -1 \end{bmatrix}$$

13

**Applications.** Databases, image processing, maximum likelihood estimation, technical job interviews, ...



## SECTION 6.3

# 6. DYNAMIC PROGRAMMING I

---

- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ *RNA secondary structure*

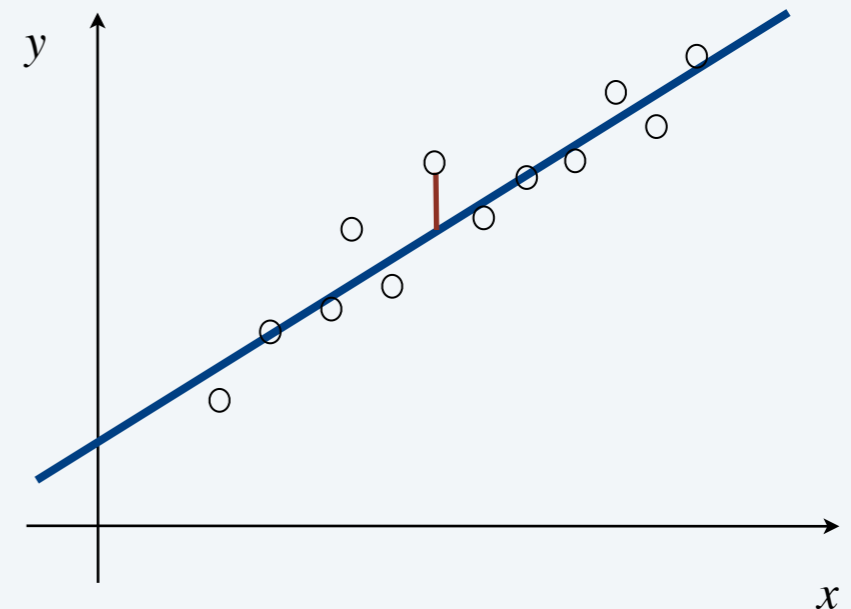
# Least squares

---

**Least squares.** Foundational problem in statistics.

- Given  $n$  points in the plane:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ .
- 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

---

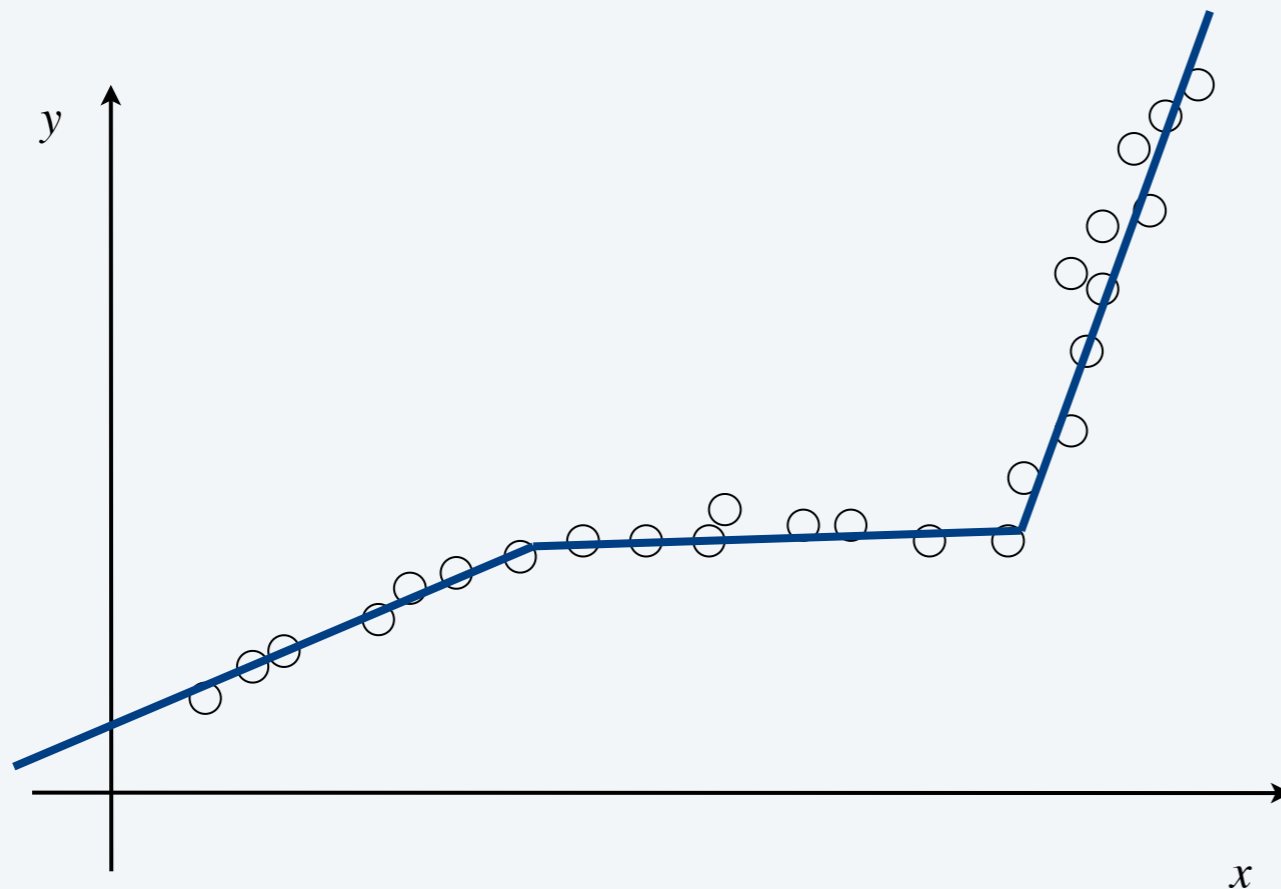
## Segmented least squares.

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

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

↑  
goodness of fit

↑  
number of lines



# Segmented least squares

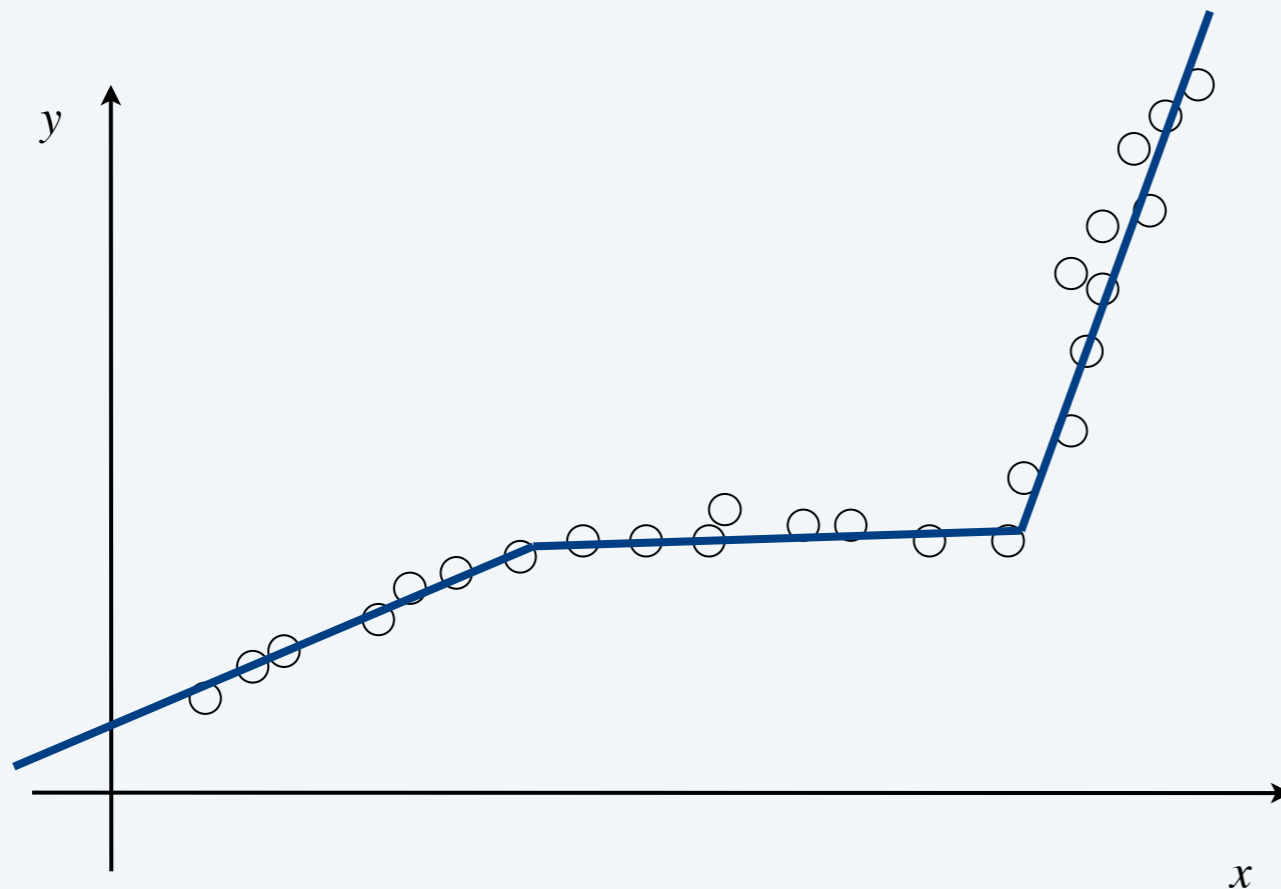
---

## Segmented least squares.

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

**Goal.** Minimize  $f(x) = E + cL$  for some constant  $c > 0$ , where

- $E$  = sum of the sums of the squared errors in each segment.
- $L$  = number of lines.





# Dynamic programming: multiway choice

---

## Notation.

- $OPT(j)$  = minimum cost for points  $p_1, p_2, \dots, p_j$ .
- $e_{ij}$  = SSE for for points  $p_i, p_{i+1}, \dots, p_j$ .

## To compute $OPT(j)$ :

- Last segment uses points  $p_i, p_{i+1}, \dots, p_j$  for some  $i \leq j$ .
- Cost =  $e_{ij} + c + OPT(i - 1)$ . ← optimal substructure property  
(proof via exchange argument)

## Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i - 1) \} & \text{if } j > 0 \end{cases}$$

# Segmented least squares algorithm

---

SEGMENTED-LEAST-SQUARES( $n, p_1, \dots, p_n, c$ )

---

FOR  $j = 1$  TO  $n$

    FOR  $i = 1$  TO  $j$

        Compute the SSE  $e_{ij}$  for the points  $p_i, p_{i+1}, \dots, p_j$ .

$M[0] \leftarrow 0$ .

FOR  $j = 1$  TO  $n$

$M[j] \leftarrow \min_{1 \leq i \leq j} \{ e_{ij} + c + M[i-1] \}$ .

previously computed value



RETURN  $M[n]$ .

---

# Segmented least squares analysis

---

**Theorem.** [Bellman 1961] DP algorithm solves the segmented least squares problem in  $O(n^3)$  time and  $O(n^2)$  space.

**Pf.**

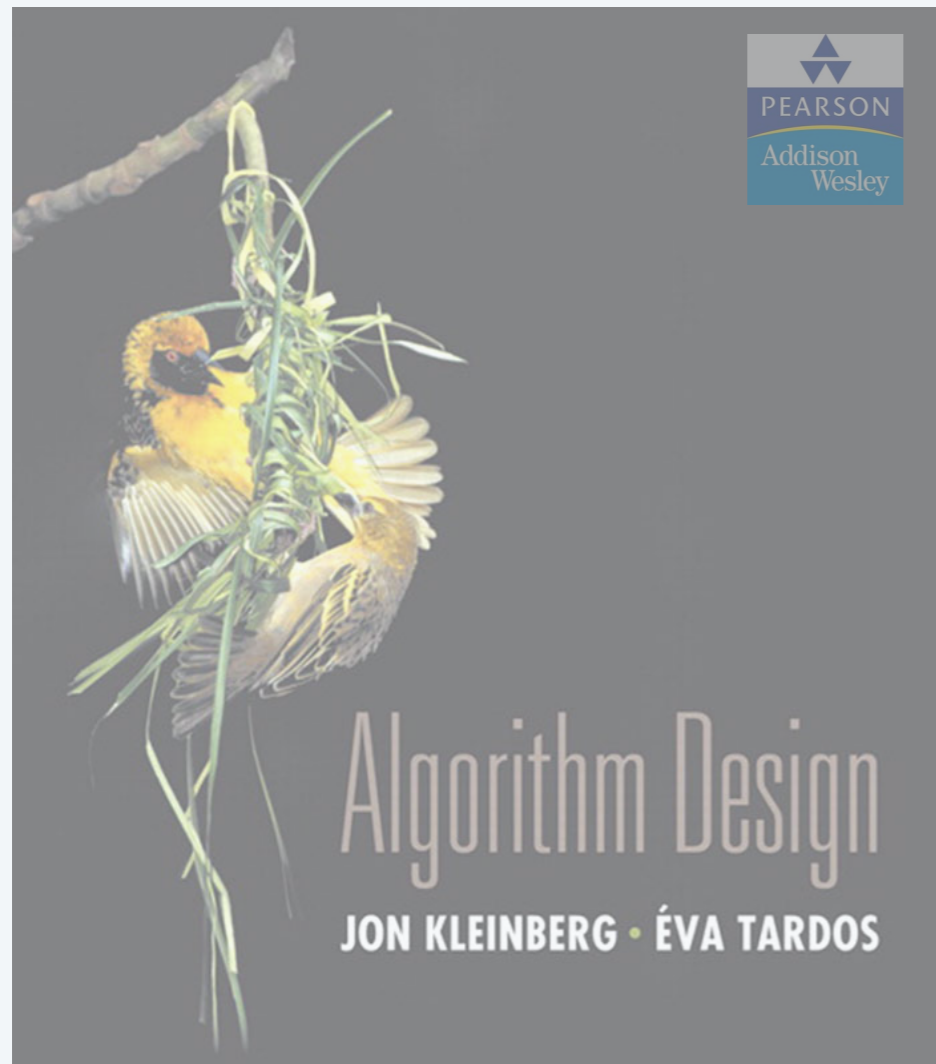
- Bottleneck = computing SSE  $e_{ij}$  for each  $i$  and  $j$ .

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$  to compute  $e_{ij}$ . ■

**Remark.** Can be improved to  $O(n^2)$  time.

- For each  $i$ : precompute cumulative sums  $\sum_{k=1}^i x_k$ ,  $\sum_{k=1}^i y_k$ ,  $\sum_{k=1}^i x_k^2$ ,  $\sum_{k=1}^i x_k y_k$ .
- Using cumulative sums, can compute  $e_{ij}$  in  $O(1)$  time.



## SECTION 6.4

# 6. DYNAMIC PROGRAMMING I

---

- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ ***knapsack problem***
- ▶ *RNA secondary structure*

# Knapsack problem

---

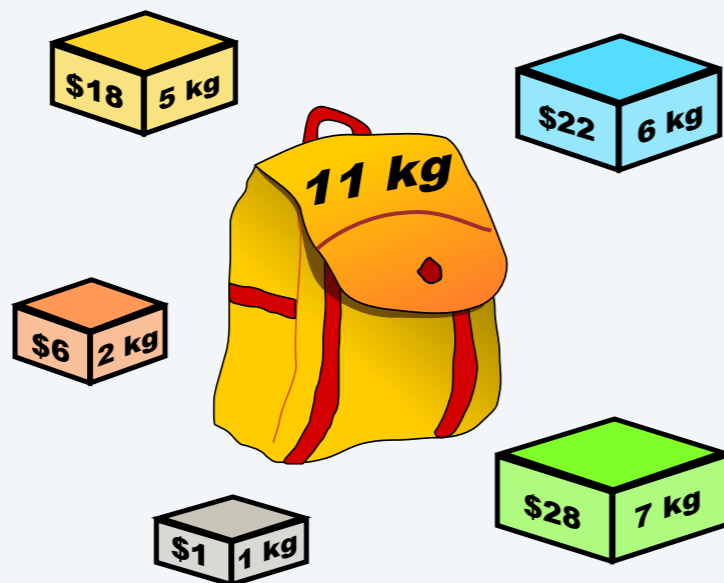
**Goal.** Pack knapsack so as to maximize total value of items taken.

- There are  $n$  items: item  $i$  provides value  $v_i > 0$  and weighs  $w_i > 0$ .
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of  $W$ .

**Ex.** The subset  $\{1, 2, 5\}$  has value \$35 (and weight 10).

**Ex.** The subset  $\{3, 4\}$  has value \$40 (and weight 11).

**Assumption.** All values and weights are integral.



Creative Commons Attribution-Share Alike 2.5  
by Dake

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

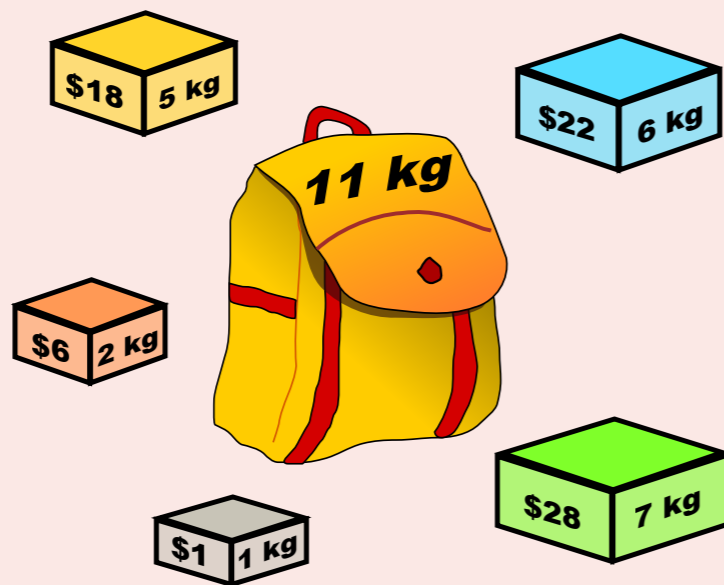
weights and values  
can be arbitrary  
positive integers

knapsack instance  
(weight limit  $W = 11$ )



## Which algorithm solves knapsack problem?

- A. Greedy-by-value: repeatedly add item with maximum  $v_i$ .
- B. Greedy-by-weight: repeatedly add item with minimum  $w_i$ .
- C. Greedy-by-ratio: repeatedly add item with maximum ratio  $v_i / w_i$ .
- D. None of the above.



Creative Commons Attribution-Share Alike 2.5  
by Dake

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance  
(weight limit  $W = 11$ )



## Which subproblems?

- A.**  $OPT(w)$  = optimal value of knapsack problem with weight limit  $w$ .
- B.**  $OPT(i)$  = optimal value of knapsack problem with items  $1, \dots, i$ .
- C.**  $OPT(i, w)$  = optimal value of knapsack problem with items  $1, \dots, i$  subject to weight limit  $w$ .
- D.** Any of the above.

# Dynamic programming: two variables

---

**Def.**  $OPT(i, w)$  = optimal value of knapsack problem with items  $1, \dots, i$ , subject to weight limit  $w$ .

**Goal.**  $OPT(n, W)$ .

**Case 1.**  $OPT(i, w)$  does not select item  $i$ .

possibly because  $w_i > w$

- $OPT(i, w)$  selects best of  $\{ 1, 2, \dots, i - 1 \}$  subject to weight limit  $w$ .

**Case 2.**  $OPT(i, w)$  selects item  $i$ .

optimal substructure property  
(proof via exchange argument)

- Collect value  $v_i$ .
- New weight limit =  $w - w_i$ .
- $OPT(i, w)$  selects best of  $\{ 1, 2, \dots, i - 1 \}$  subject to new weight limit.

**Bellman equation.**

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{ OPT(i - 1, w), v_i + OPT(i - 1, w - w_i) \} & \text{otherwise} \end{cases}$$



# Knapsack problem: bottom-up dynamic programming

---

**KNAPSACK**( $n, W, w_1, \dots, w_n, v_1, \dots, v_n$ )

**FOR**  $w = 0$  **TO**  $W$

$M[0, w] \leftarrow 0.$

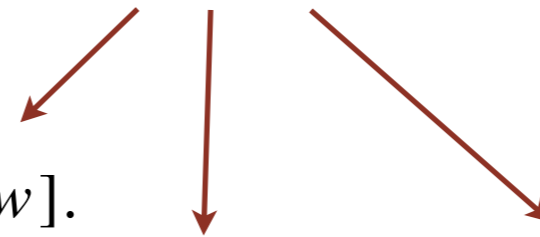
**FOR**  $i = 1$  **TO**  $n$

**FOR**  $w = 0$  **TO**  $W$

**IF** ( $w_i > w$ )  $M[i, w] \leftarrow M[i-1, w].$

**ELSE**  $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}.$

previously computed values



**RETURN**  $M[n, W].$

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{ OPT(i - 1, w), v_i + OPT(i - 1, w - w_i) \} & \text{otherwise} \end{cases}$$

# Knapsack problem: bottom-up dynamic programming demo

$i$	$v_i$	$w_i$
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\} & \text{otherwise} \end{cases}$$

		weight limit $w$											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items $1, \dots, i$	{ }	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

$OPT(i, w)$  = optimal value of knapsack problem with items  $1, \dots, i$ , subject to weight limit  $w$

# Knapsack problem: running time


---

**Theorem.** The DP algorithm solves the knapsack problem with  $n$  items and maximum weight  $W$  in  $\Theta(nW)$  time and  $\Theta(nW)$  space.

**Pf.**

- Takes  $O(1)$  time per table entry.
- There are  $\Theta(nW)$  table entries.
- After computing optimal values, can trace back to find solution:  
 $OPT(i, w)$  takes item  $i$  iff  $M[i, w] > M[i - 1, w]$ . ■

weights are integers  
between 1 and  $W$



**Remarks.**

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.



**Does there exist a poly-time algorithm for the knapsack problem?**

- A.** Yes, because the DP algorithm takes  $\Theta(n W)$  time.
- B.** No, because  $\Theta(n W)$  is not a polynomial function of the input size.
- C.** No, because the problem is **NP**-hard.
- D.** Unknown.

# COIN CHANGING



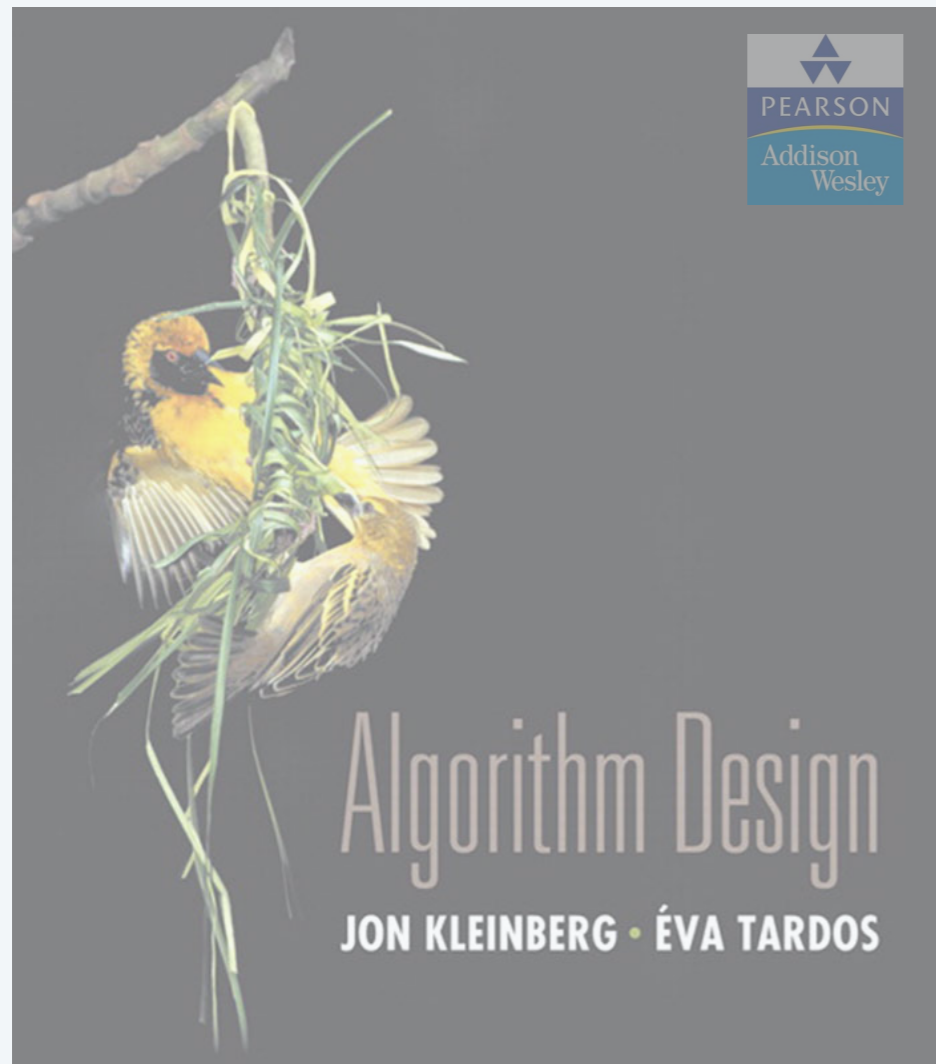
**Problem.** Given  $n$  coin denominations  $\{c_1, c_2, \dots, c_n\}$  and a target value  $V$ , find the fewest coins needed to make change for  $V$  (or report impossible).

**Recall.** Greedy cashier's algorithm is optimal for U.S. coin denominations, but not for arbitrary coin denominations.

**Ex.**  $\{1, 10, 21, 34, 70, 100, 350, 1295, 1500\}$ .

**Optimal.**  $140\text{¢} = 70 + 70$ .





## SECTION 6.5

# 6. DYNAMIC PROGRAMMING I

---

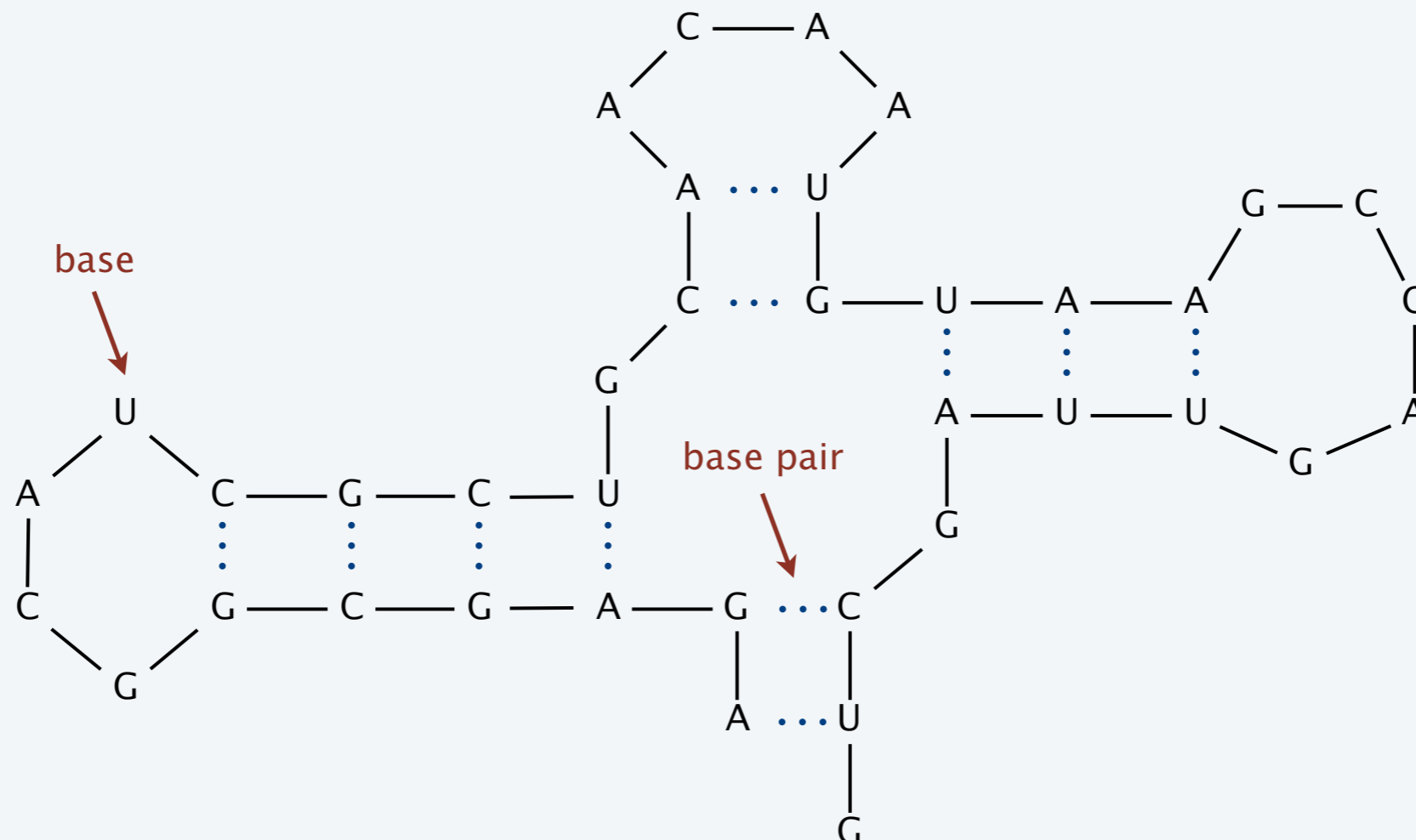
- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ ***RNA secondary structure***

# RNA secondary structure

---

**RNA.** String  $B = b_1b_2\dots b_n$  over alphabet  $\{A, C, G, U\}$ .

**Secondary structure.** RNA is single-stranded so it tends to loop back and form **base pairs** with itself. This structure is essential for understanding behavior of molecule.

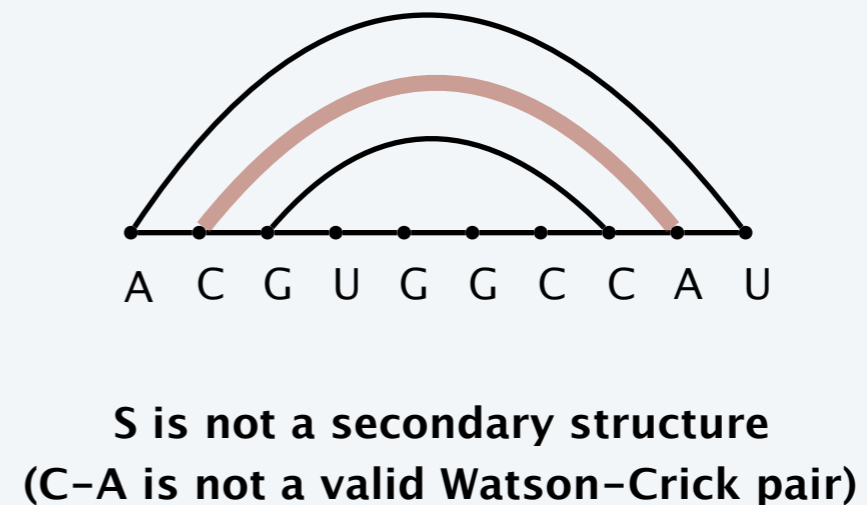
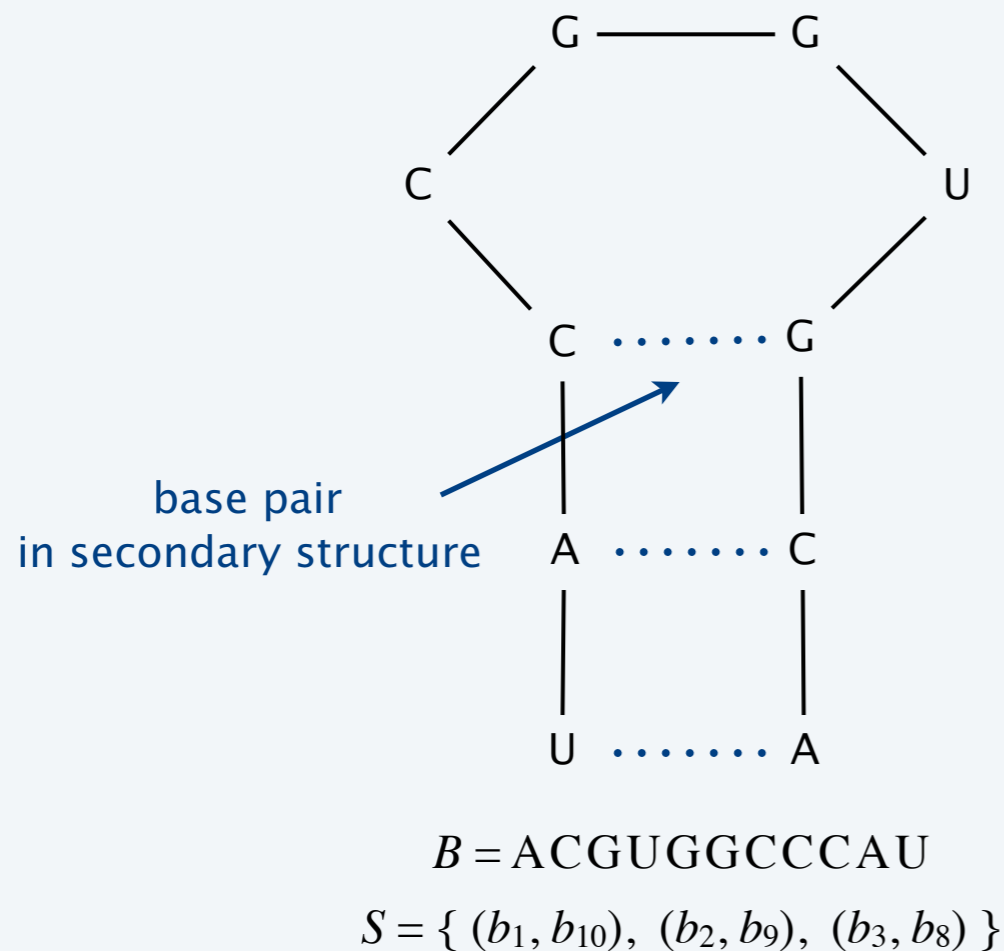


RNA secondary structure for GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

# RNA secondary structure

**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

- [Watson–Crick]  $S$  is a matching and each pair in  $S$  is a Watson–Crick complement: A–U, U–A, C–G, or G–C.

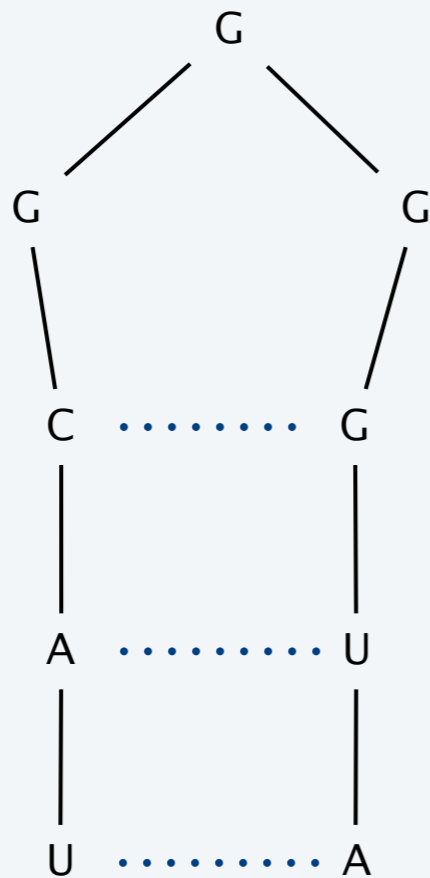




# RNA secondary structure

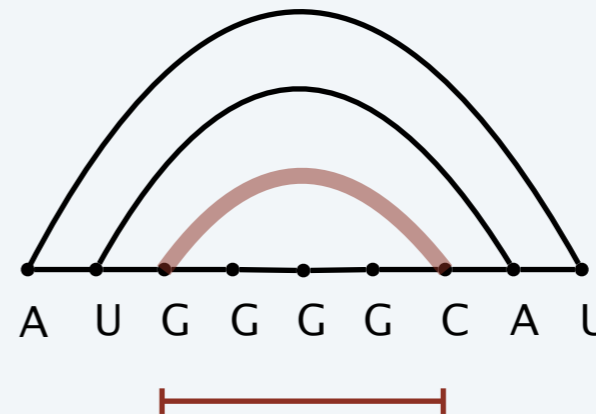
**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

- [Watson–Crick]  $S$  is a matching and each pair in  $S$  is a Watson–Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .



$B = \text{AUGGGGCAU}$

$S = \{ (b_1, b_9), (b_2, b_8), (b_3, b_7) \}$



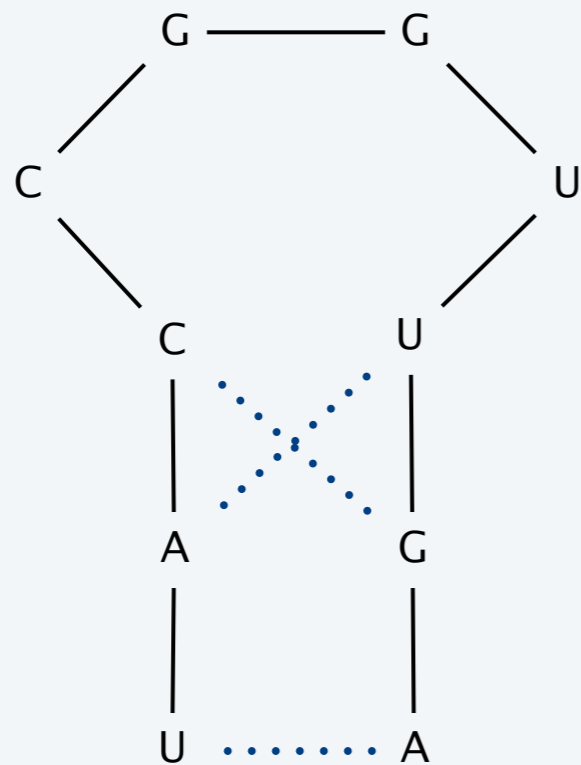
**$S$  is not a secondary structure**  
( $\leq 4$  intervening bases between G and C)

# RNA secondary structure

---

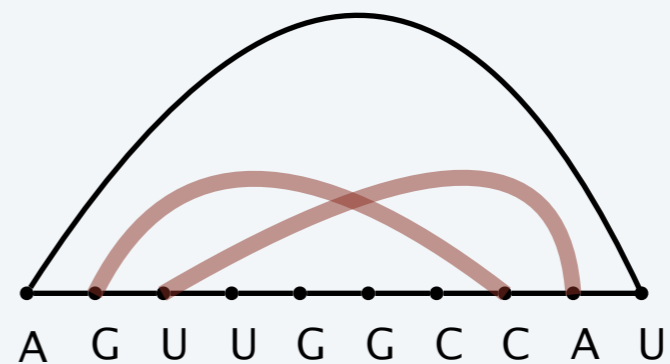
**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

- [Watson–Crick]  $S$  is a matching and each pair in  $S$  is a Watson–Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .
- [Non-crossing] If  $(b_i, b_j)$  and  $(b_k, b_\ell)$  are two pairs in  $S$ , then we cannot have  $i < k < j < \ell$ .



$B = \text{ACUUGGCCAU}$

$S = \{ (b_1, b_{10}), (b_2, b_8), (b_3, b_9) \}$



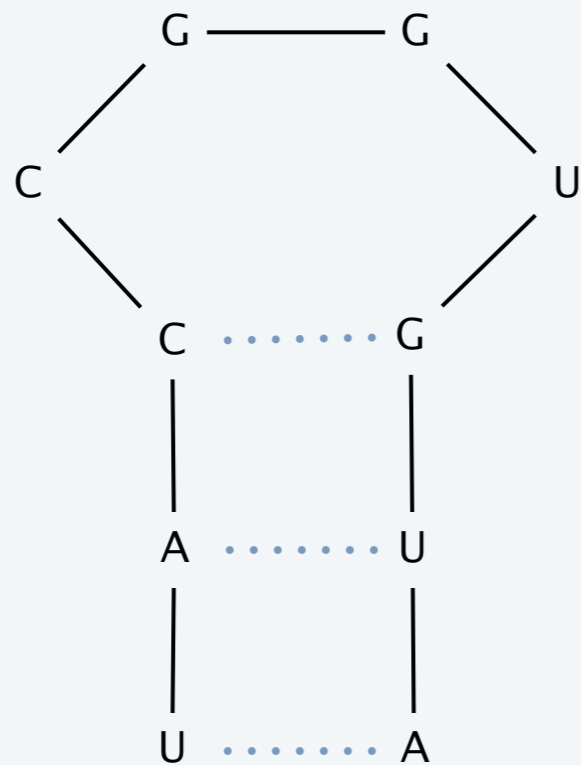
**S is not a secondary structure**  
(G–C and U–A cross)

# RNA secondary structure

---

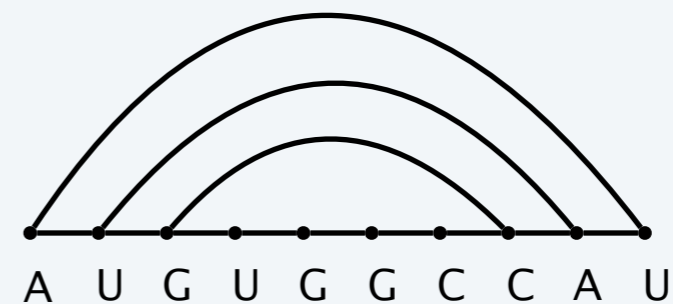
**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

- [Watson–Crick]  $S$  is a matching and each pair in  $S$  is a Watson–Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .
- [Non-crossing] If  $(b_i, b_j)$  and  $(b_k, b_\ell)$  are two pairs in  $S$ , then we cannot have  $i < k < j < \ell$ .



$B = \text{AUGUGGCCAU}$

$S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$



**$S$  is a secondary structure  
(with 3 base pairs)**


# RNA secondary structure

---

**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

- [Watson–Crick]  $S$  is a matching and each pair in  $S$  is a Watson–Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .
- [Non-crossing] If  $(b_i, b_j)$  and  $(b_k, b_\ell)$  are two pairs in  $S$ , then we cannot have  $i < k < j < \ell$ .

**Free-energy hypothesis.** RNA molecule will form the secondary structure with the minimum total free energy.

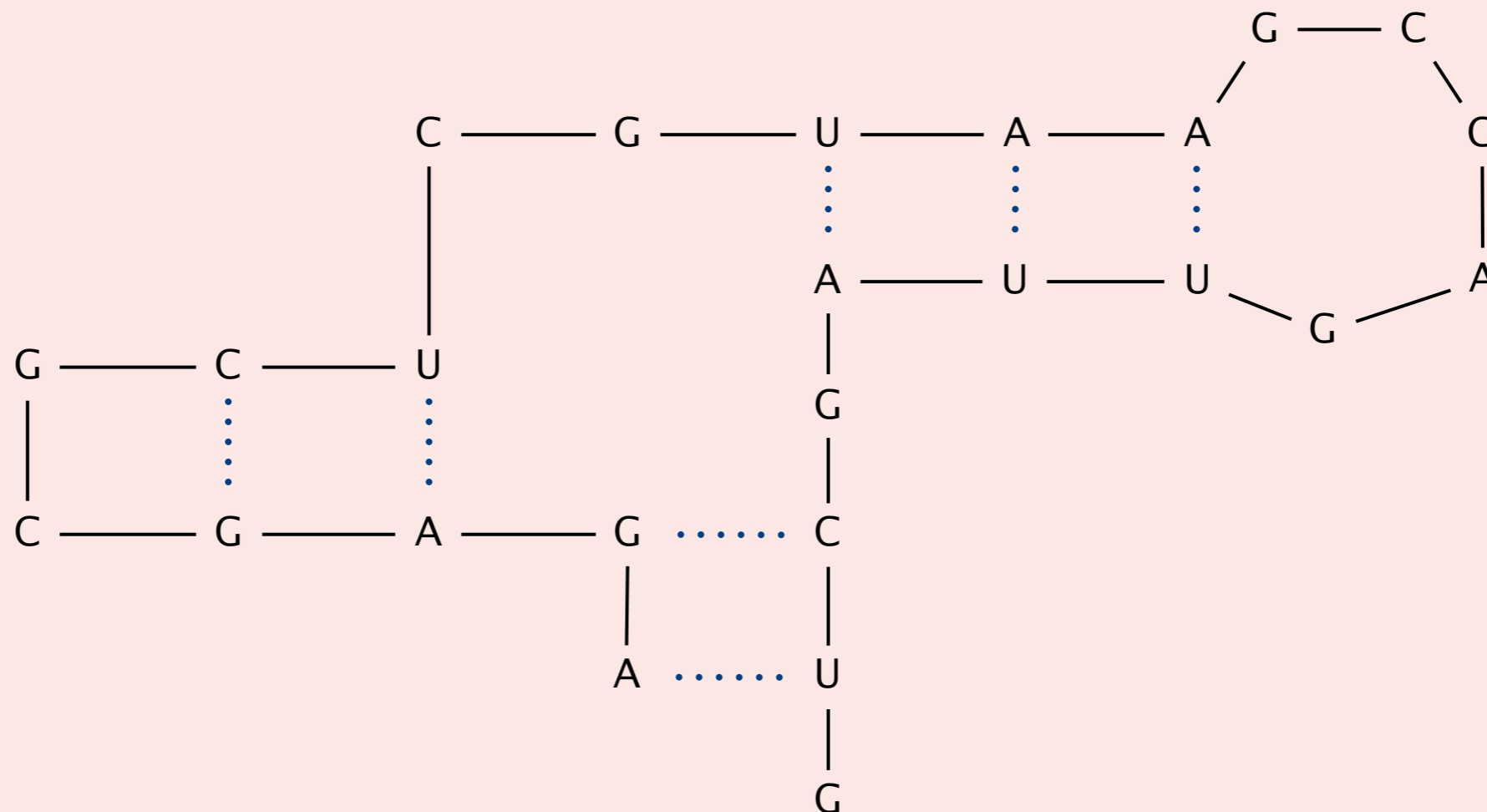
  
approximate by number of base pairs  
(more base pairs  $\Rightarrow$  lower free energy)

**Goal.** Given an RNA molecule  $B = b_1 b_2 \dots b_n$ , find a secondary structure  $S$  that maximizes the number of base pairs.



Is the following a secondary structure?

- A. Yes.
- B. No, violates Watson–Crick condition.
- C. No, violates no-sharp-turns condition.
- D. No, violates no-crossing condition.





## Which subproblems?

- A.**  $OPT(j) = \max$  number of base pairs in secondary structure of the substring  $b_1b_2 \dots b_j$ .
- B.**  $OPT(j) = \max$  number of base pairs in secondary structure of the substring  $b_jb_{j+1} \dots b_n$ .
- C.** Either A or B.
- D.** Neither A nor B.

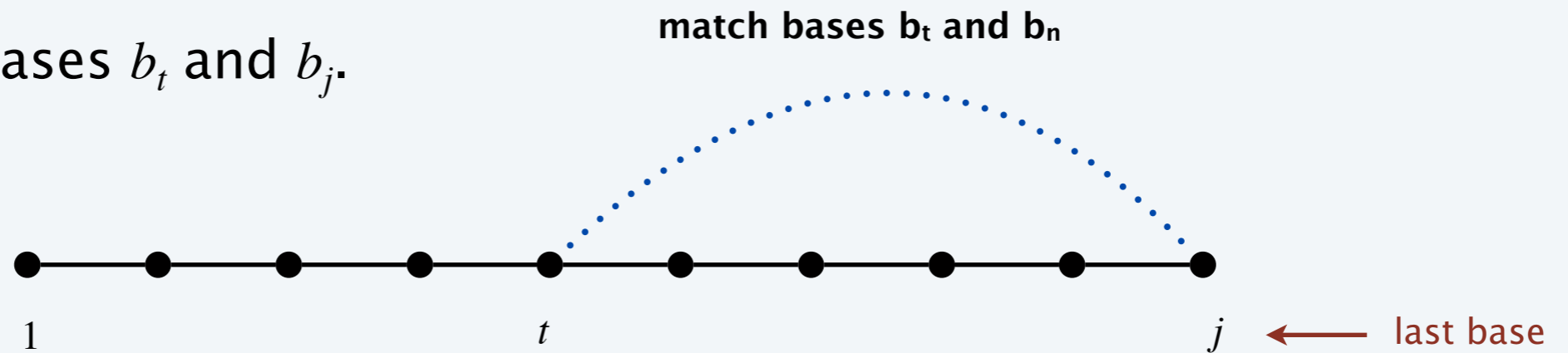
# RNA secondary structure: subproblems

---

**First attempt.**  $OPT(j)$  = maximum number of base pairs in a secondary structure of the substring  $b_1b_2 \dots b_j$ .

**Goal.**  $OPT(n)$ .

**Choice.** Match bases  $b_t$  and  $b_j$ .



**Difficulty.** Results in two subproblems (but one of wrong form).

- Find secondary structure in  $b_1b_2 \dots b_{t-1}$ . ←  $OPT(t-1)$
- Find secondary structure in  $b_{t+1}b_{t+2} \dots b_{j-1}$ . ← need more subproblems (first base no longer  $b_1$ )

# Dynamic programming over intervals

---

**Def.**  $OPT(i, j)$  = maximum number of base pairs in a secondary structure of the substring  $b_i b_{i+1} \dots b_j$ .

**Case 1.** If  $i \geq j - 4$ .

- $OPT(i, j) = 0$  by no-sharp-turns condition.

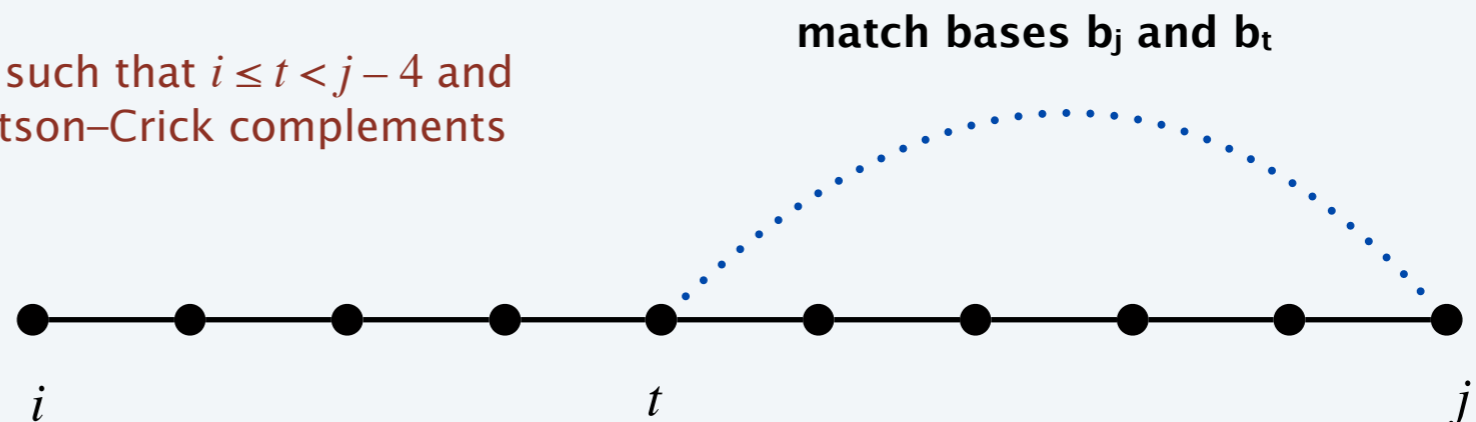
**Case 2.** Base  $b_j$  is not involved in a pair.

- $OPT(i, j) = OPT(i, j - 1)$ .

**Case 3.** Base  $b_j$  pairs with  $b_t$  for some  $i \leq t < j - 4$ .

- Non-crossing condition decouples resulting two subproblems.
- $OPT(i, j) = 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \}$ .

take max over  $t$  such that  $i \leq t < j - 4$  and  $b_t$  and  $b_j$  are Watson-Crick complements







In which order to compute  $OPT(i, j)$  ?

- A. Increasing  $i$ , then  $j$ .
- B. Increasing  $j$ , then  $i$ .
- C. Either A or B.
- D. Neither A nor B.

# Bottom-up dynamic programming over intervals

Q. In which order to solve the subproblems?

A. Do shortest intervals first—increasing order of  $|j - i|$ .

RNA-SECONDARY-STRUCTURE( $n, b_1, \dots, b_n$ )

FOR  $k = 5$  TO  $n - 1$

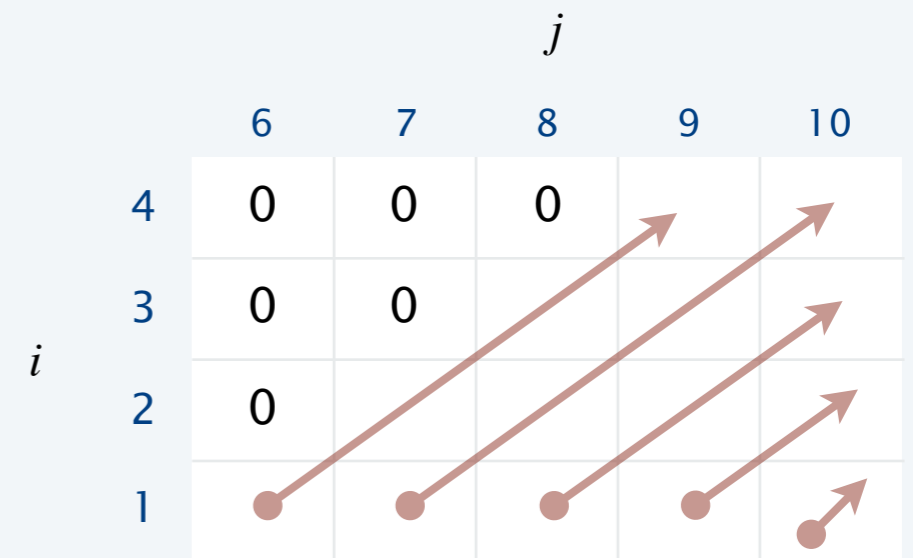
FOR  $i = 1$  TO  $n - k$

$j \leftarrow i + k.$

Compute  $M[i, j]$  using formula.

RETURN  $M[1, n]$ .

all needed values  
are already computed



order in which to solve subproblems

**Theorem.** The DP algorithm solves the RNA secondary structure problem in  $O(n^3)$  time and  $O(n^2)$  space.


# Dynamic programming summary

---

## Outline.

- Define a collection of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from “smallest” to “largest” that enables determining a solution to a subproblem from solutions to smaller subproblems.

typically, only a polynomial  
number of subproblems



## Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Intervals: RNA secondary structure.

Top-down vs. bottom-up dynamic programming. Opinions differ.