

## 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

Copyright © 2013 Kevin Wayne

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


# Algorithmic paradigms

---

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

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

**Dynamic programming.** Break up a problem into a series of **overlapping** subproblems, and build up solutions to larger and larger subproblems.



fancy name for  
caching away 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 was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.



### 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

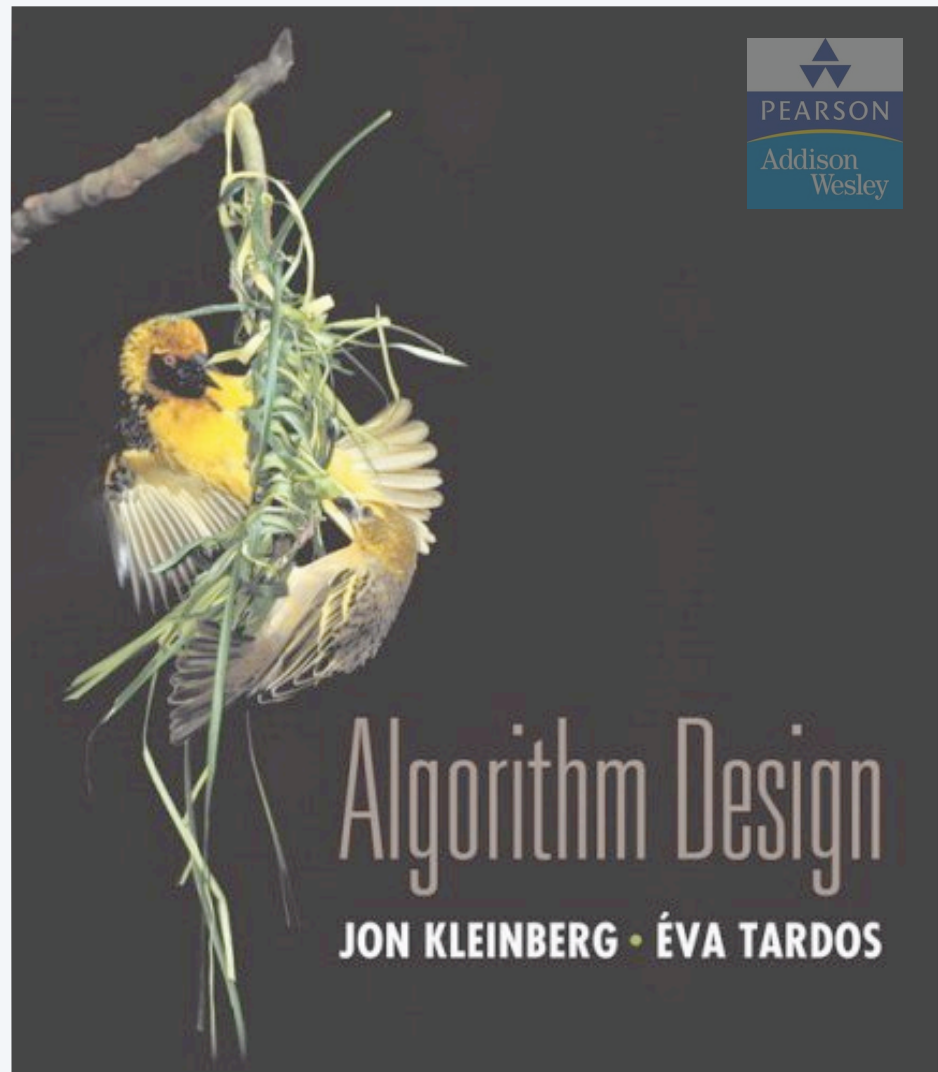
---

## Areas.

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

## Some famous dynamic programming algorithms.

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



## SECTION 6.1–6.2

# 6. DYNAMIC PROGRAMMING I

---

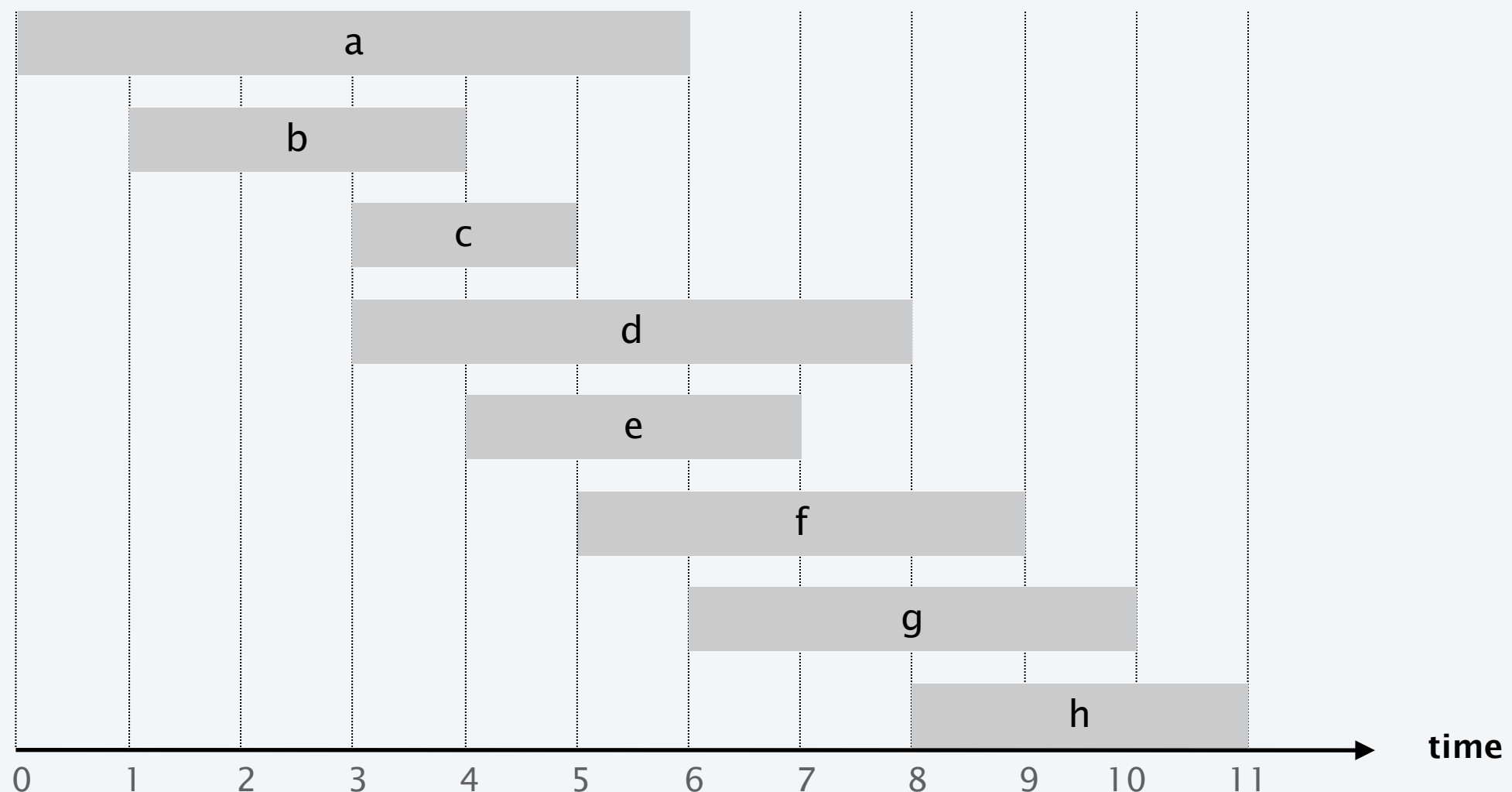
- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ *RNA secondary structure*

# 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.

Can be solved with greedy ordering schedules



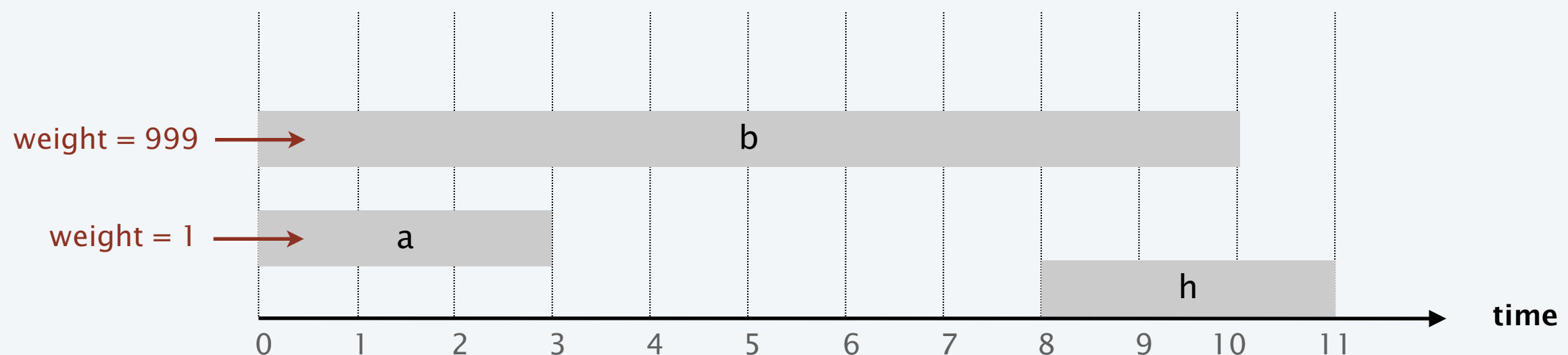
# Earliest-finish-time first algorithm

## 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.

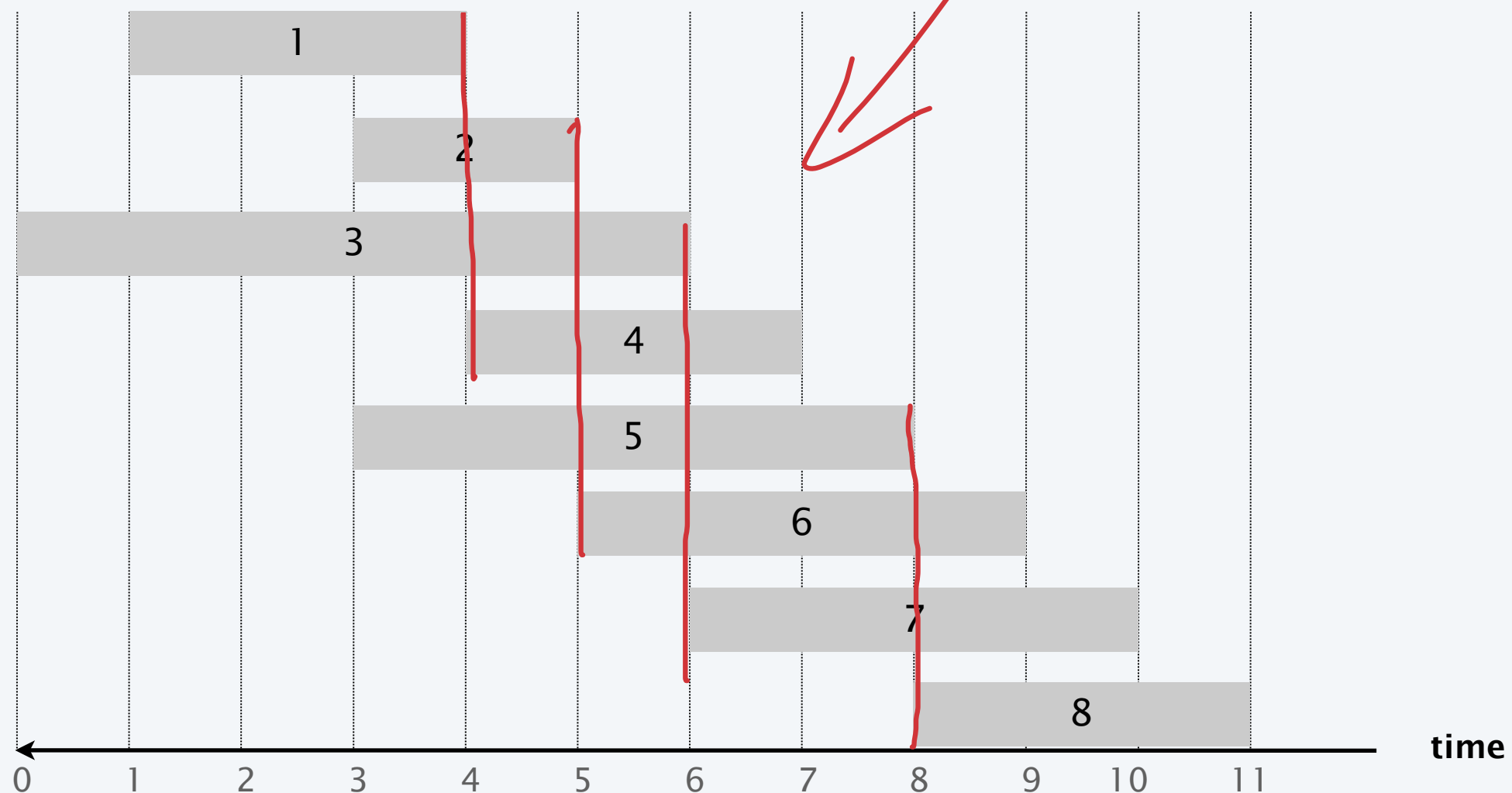


# Weighted interval scheduling

**Notation.** Label jobs by finishing 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.**  $p(8) = 5, p(7) = 3, p(2) = 0$ .





# Dynamic programming: binary choice

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

**Case 1.**  $OPT$  selects job  $j$ .

- Collect profit  $v_j$ .
- 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)

**Case 2.**  $OPT$  does not select job  $j$ .

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

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ \underbrace{v_j + OPT(p(j))}_{\text{now + compatible}}, \underbrace{OPT(j-1)}_{\text{prec.}} \right\} & \text{otherwise} \end{cases}$$

# Weighted interval scheduling: brute force

---

Input:  $n, s[1..n], f[1..n], v[1..n]$

Sort jobs by finish time so that  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

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

*→ predecessors (linear time) compatible.*

Compute-Opt( $j$ )

if  $j = 0$

return 0.

else

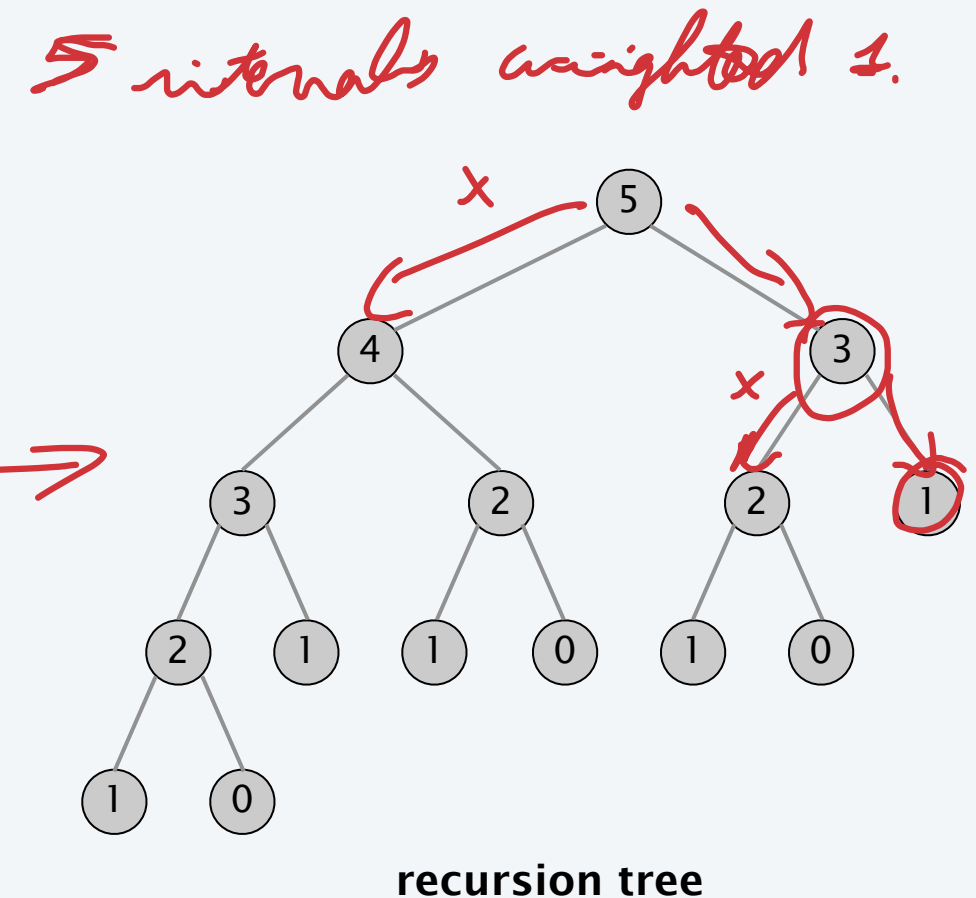
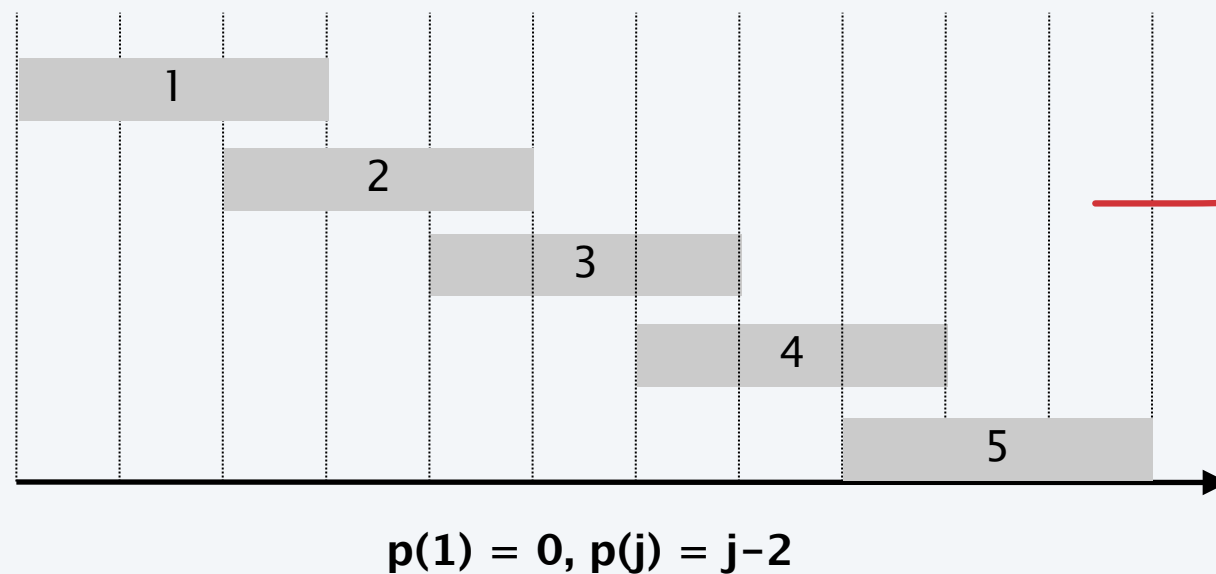
*[* return  $\max(v[j] + \text{Compute-Opt}(p[j]), \text{Compute-Opt}(j-1))$ . *]*

*N.B.*

# Weighted interval scheduling: brute force

**Observation.** Recursive algorithm fails spectacularly because of redundant subproblems  $\Rightarrow$  exponential algorithms.

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



# Weighted interval scheduling: memoization

*Path.!!*

Memoization. Cache results of each subproblem; lookup as needed.

Input:  $n$ ,  $s[1..n]$ ,  $f[1..n]$ ,  $v[1..n]$

*Value of interval.*

Sort jobs by finish time so that  $f[1] \leq f[2] \leq \dots \leq f[n]$ .

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

for  $j = 1$  to  $n$

$M[j] \leftarrow \text{empty}.$

$M[0] \leftarrow 0.$

M-Compute-Opt( $j$ )

1 if  $M[j]$  is empty

    2  $M[j] \leftarrow \max(v[j] + \text{M-Compute-Opt}(p[j]), \text{M-Compute-Opt}(j - 1)).$

6 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  $p(\cdot)$  :  $O(n \log n)$  via 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  $\Phi = \#$  nonempty entries of  $M[]$ .**
  - initially  $\Phi = 0$ , throughout  $\Phi \leq n$ .
  - (ii) increases  $\Phi$  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 presorted by start and finish times.

# Weighted interval scheduling: finding a solution

---

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

A. Make a second pass.

```
Find-Solution(j)
if j = 0
    return  $\emptyset$ 
else if (v[j] + M[p[j]] > M[j-1])
    return {j}  $\cup$  Find-Solution(p[j]).
else
    return Find-Solution(j-1).
```

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

# Weighted interval scheduling: bottom-up

---

Bottom-up dynamic programming. Unwind recursion.

**BOTTOM-UP** ( $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ )

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

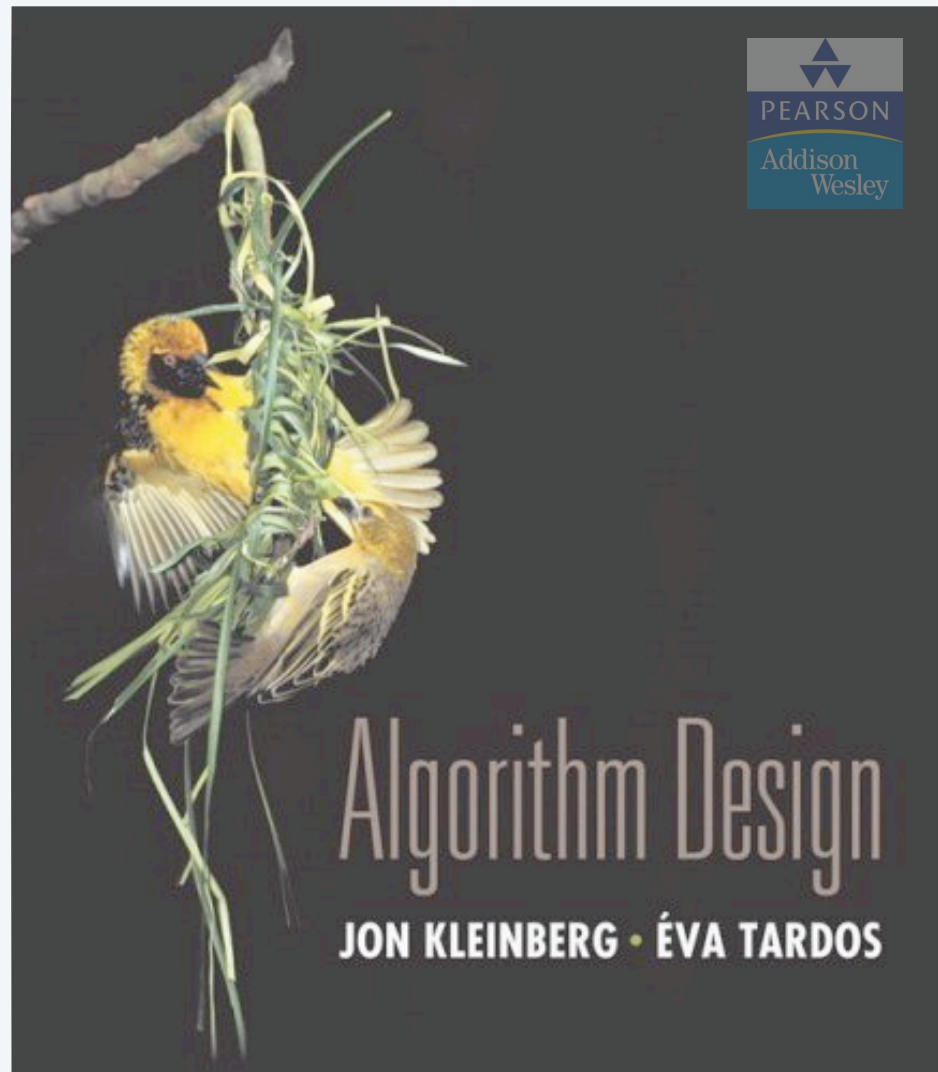
Compute  $p(1), p(2), \dots, p(n)$ .

$M[0] \leftarrow 0.$

FOR  $j = 1$  TO  $n$

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

*Index of Compatible.*



## SECTION 6.3

# 6. DYNAMIC PROGRAMMING I

---

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

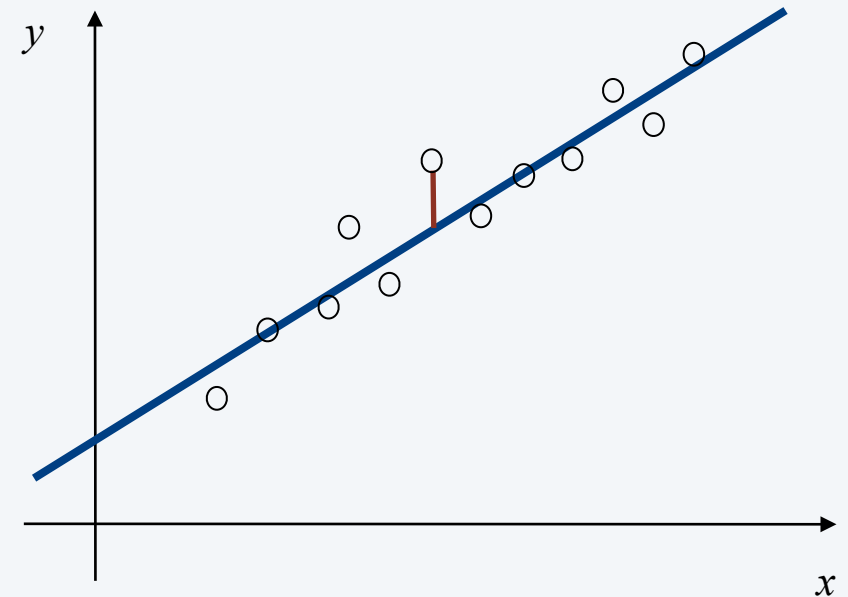


# Least squares

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

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

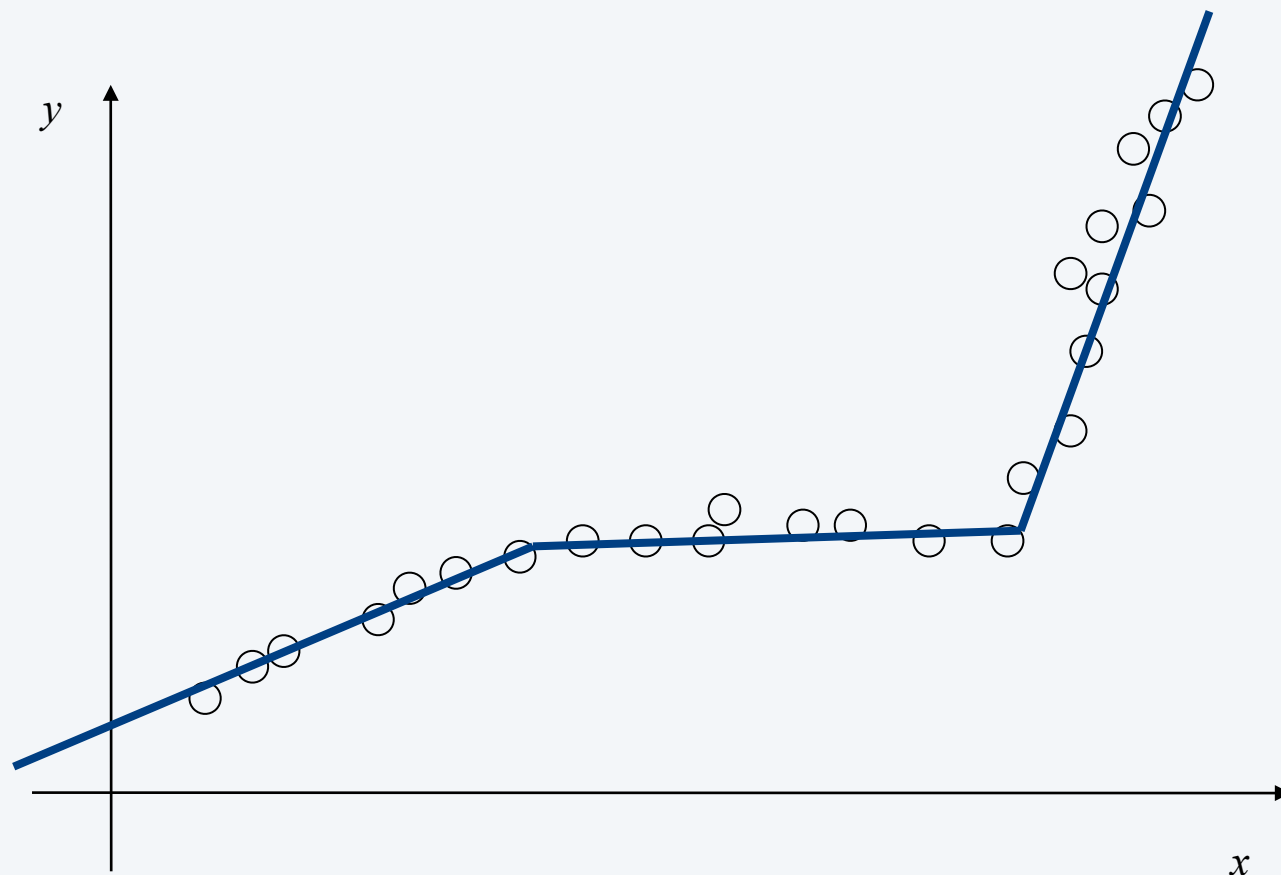
# Segmented least squares

## 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

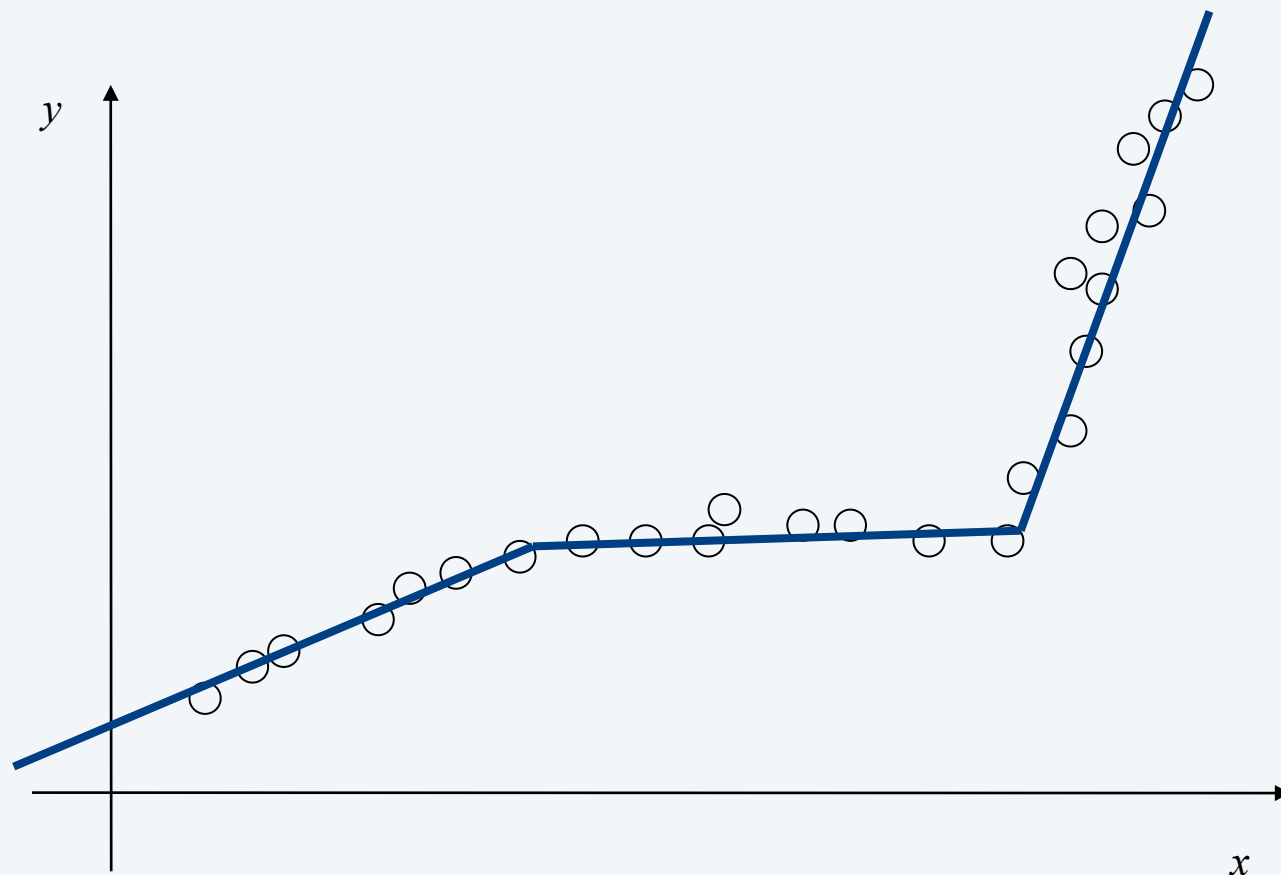


# Segmented least squares

---

Given  $n$  points in the plane:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  with  $x_1 < x_2 < \dots < x_n$  and a constant  $c > 0$ , find a sequence of lines that minimizes  $f(x) = E + cL$ :

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



# Dynamic programming: multiway choice

## Notation.

- $OPT(j)$  = minimum cost for points  $p_1, p_2, \dots, p_j$ .
- $e(i, j)$  = minimum sum of squares 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  $i$ .
- Cost =  $e(i, j) + c + OPT(i - 1)$ . ← optimal substructure property (proof via exchange argument)

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

# 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 least squares  $e(i, j)$  for the segment  $p_i, p_{i+1}, \dots, p_j$ .

$M[0] \leftarrow 0$ .

FOR  $j = 1$  TO  $n$

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

RETURN  $M[n]$ .

---

# Segmented least squares analysis

---

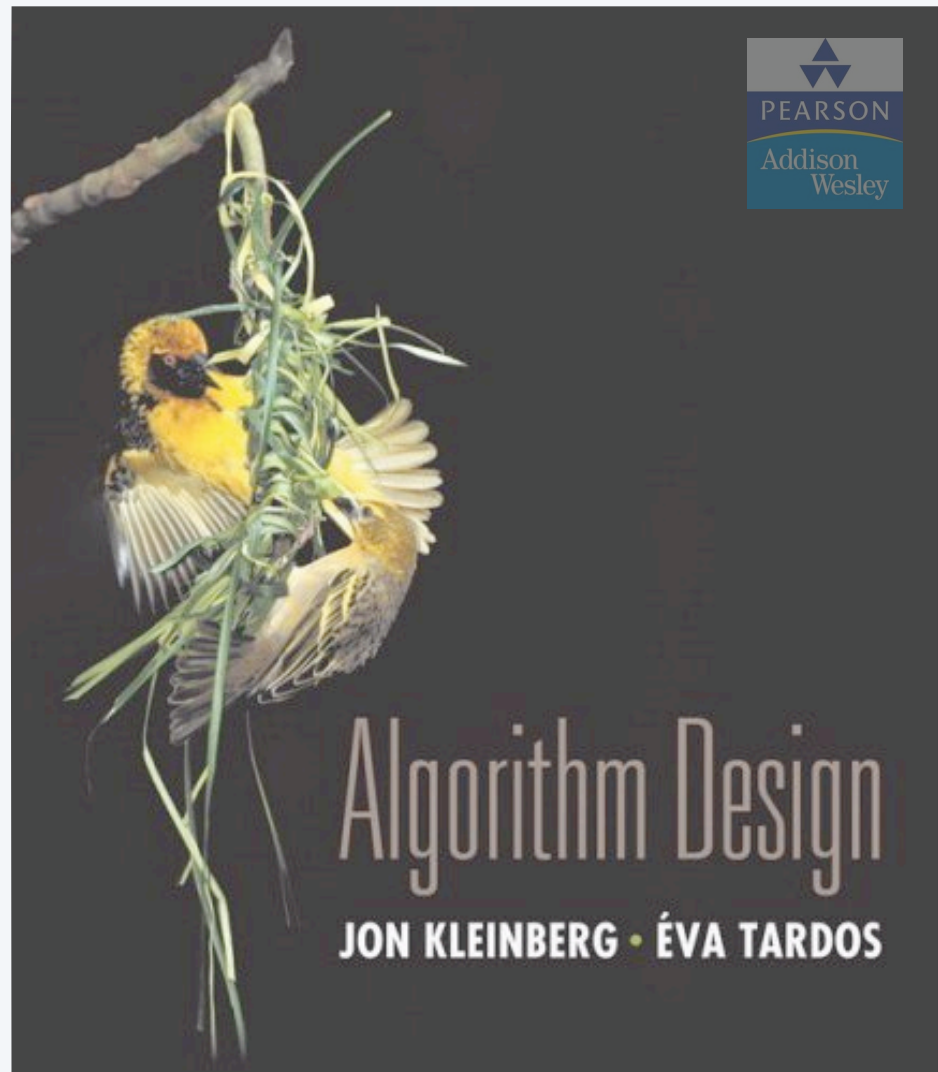
**Theorem.** [Bellman 1961] The dynamic programming algorithm solves the segmented least squares problem in  $O(n^3)$  time and  $O(n^2)$  space.

**Pf.**

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

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

**Remark.** Can be improved to  $O(n^2)$  time and  $O(n)$  space by precomputing various statistics. How?



## SECTION 6.4

# 6. DYNAMIC PROGRAMMING I

---

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

# Knapsack problem

---

- Given  $n$  objects and a "knapsack."
- Item  $i$  weighs  $w_i > 0$  and has value  $v_i > 0$ .
- Knapsack has capacity of  $W$ .
- Goal: fill knapsack so as to maximize total value.

Ex.  $\{1, 2, 5\}$  has value 35.

Ex.  $\{3, 4\}$  has value 40.

Ex.  $\{3, 5\}$  has value 46 (but exceeds weight limit).

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

knapsack instance  
(weight limit  $W = 11$ )

Greedy by **value**. Repeatedly add item with maximum  $v_i$ .

Greedy by **weight**. Repeatedly add item with minimum  $w_i$ .

Greedy by **ratio**. Repeatedly add item with maximum ratio  $v_i / w_i$ .

Observation. None of greedy algorithms is optimal.



# Dynamic programming: false start

---

Def.  $OPT(i) = \max$  profit subset of items  $1, \dots, i$ .

Case 1.  $OPT$  does not select item  $i$ .

- $OPT$  selects best of  $\{1, 2, \dots, i-1\}$ .

optimal substructure property  
(proof via exchange argument)

Case 2.  $OPT$  selects item  $i$ .

- Selecting item  $i$  does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before  $i$ ,  
we don't even know if we have enough room for  $i$ .

Conclusion. Need more subproblems!

## Dynamic programming: adding a new variable

---

Def.  $OPT(i, w)$  = max profit subset of items  $1, \dots, i$  with weight limit  $w$ .

Case 1.  $OPT$  does not select item  $i$ .

- $OPT$  selects best of  $\{1, 2, \dots, i-1\}$  using weight limit  $w$ .

Case 2.  $OPT$  selects item  $i$ .

- New weight limit =  $w - w_i$ .
- $OPT$  selects best of  $\{1, 2, \dots, i-1\}$  using this new weight limit.

↖ ↘ optimal substructure property  
(proof via exchange argument)

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

# Knapsack problem: bottom-up

---

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  $w = 1$  TO  $W$

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

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

RETURN  $M[n, W].$

---

# Knapsack problem: bottom-up demo

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

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

		weight limit w											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items 1, ..., 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) = max profit subset of items 1, ..., i with weight limit w.

# Knapsack problem: running time

---

**Theorem.** There exists an algorithm to solve the knapsack problem with  $n$  items and maximum weight  $W$  in  $\Theta(nW)$  time and  $\Theta(nW)$  space.

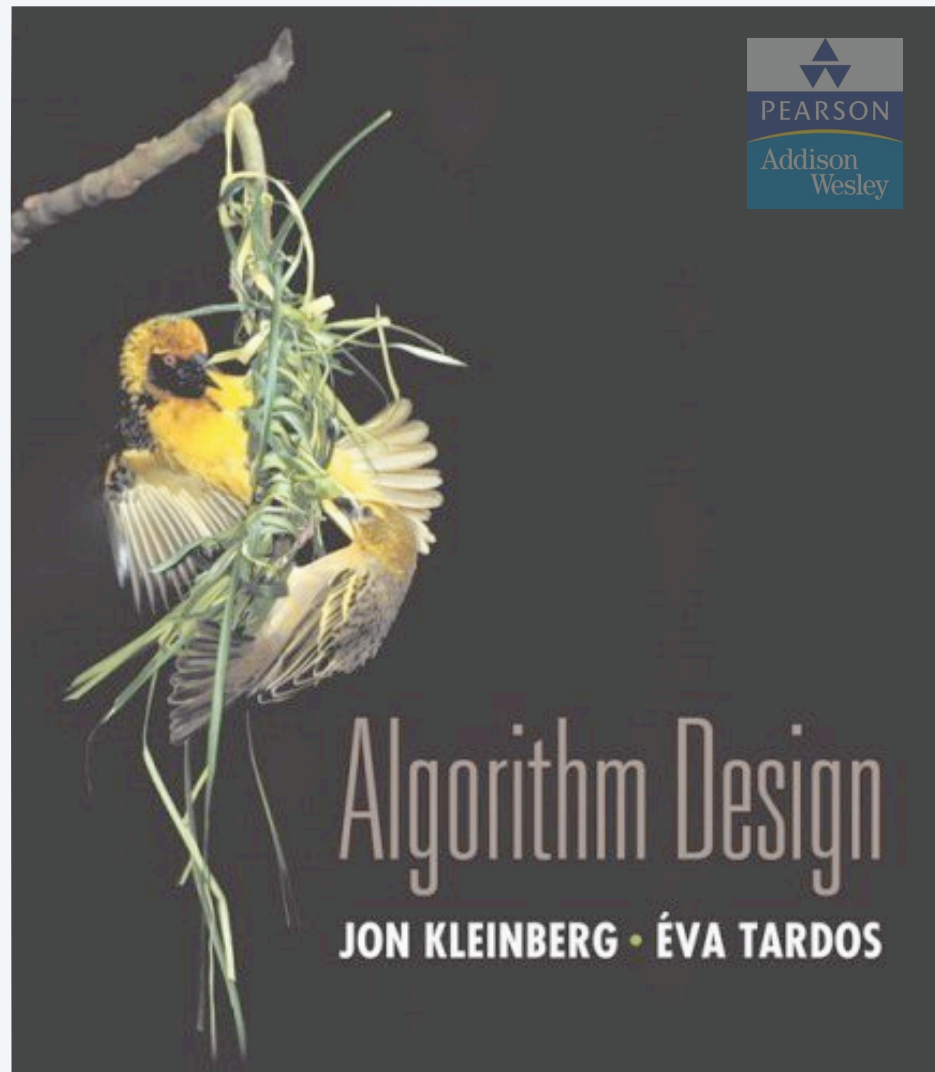
**Pf.**

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

weights are integers  
between 1 and  $W$

**Remarks.**

- Not polynomial in input size! ← "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [ CHAPTER 8 ]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [ SECTION 11.8 ]



## SECTION 6.5

# 6. DYNAMIC PROGRAMMING I

---

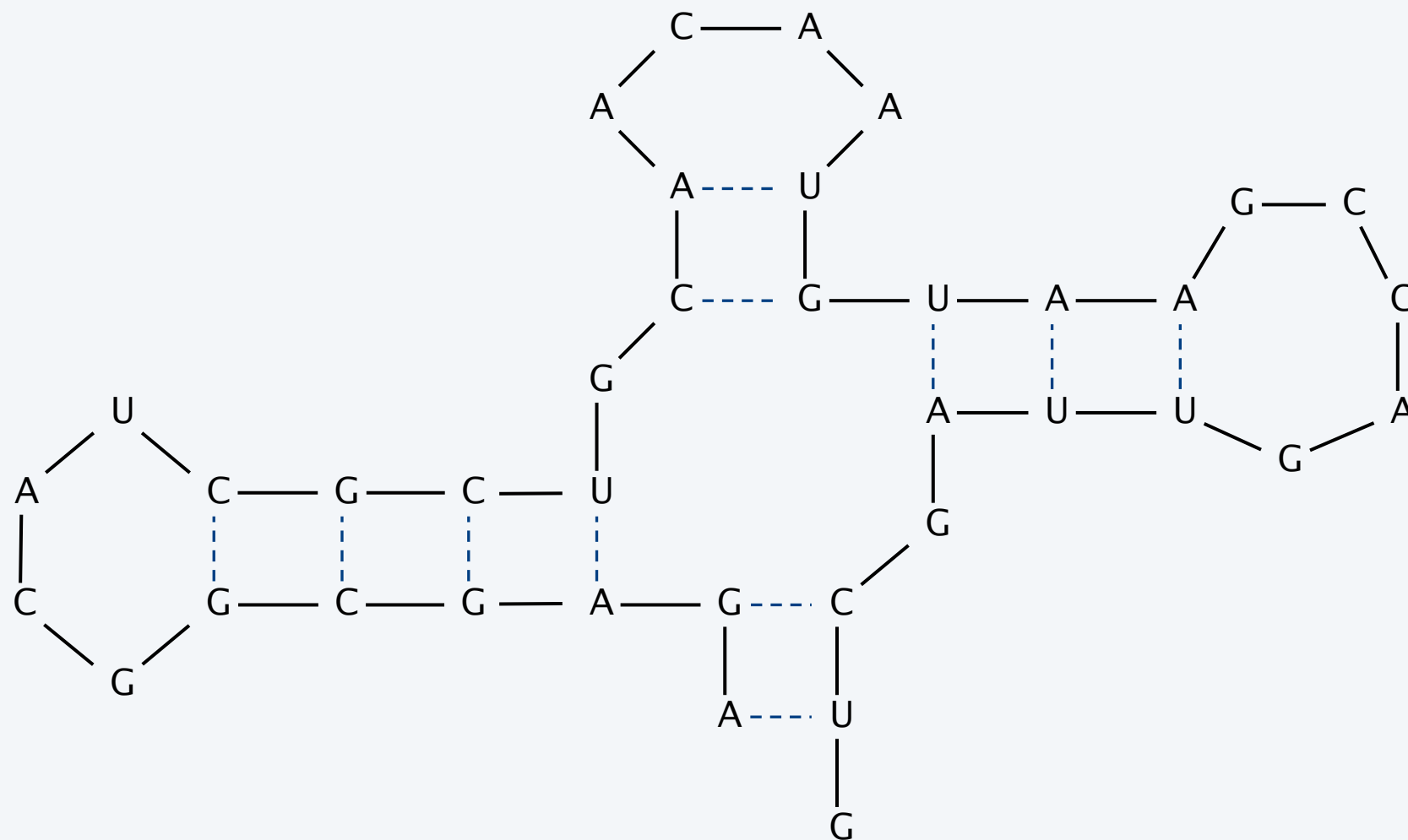
- ▶ *weighted interval scheduling*
- ▶ *segmented least squares*
- ▶ *knapsack problem*
- ▶ *RNA secondary structure*

# 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.
- [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.** Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy.

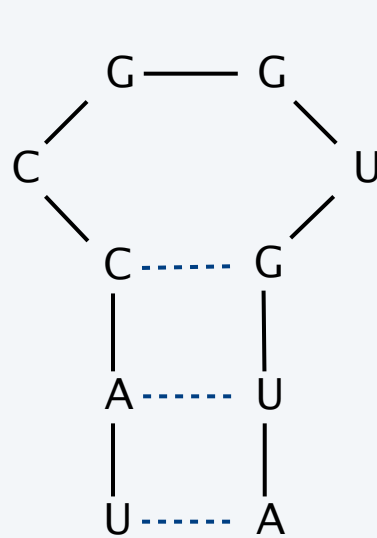
 approximate by number of base pairs

**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.

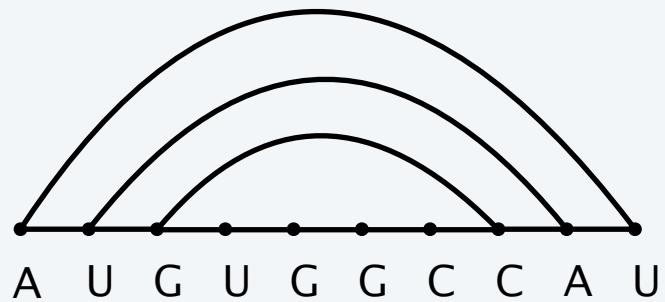
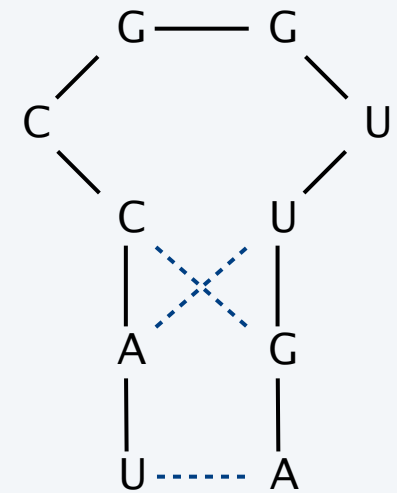
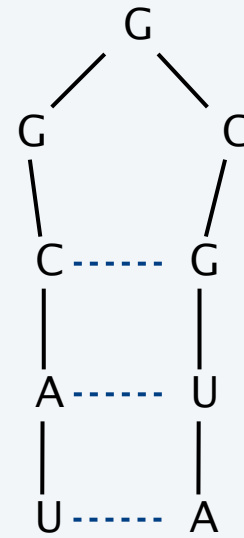


# RNA secondary structure

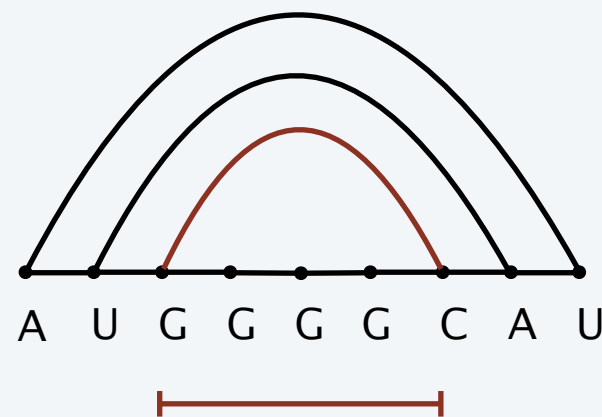
## Examples.



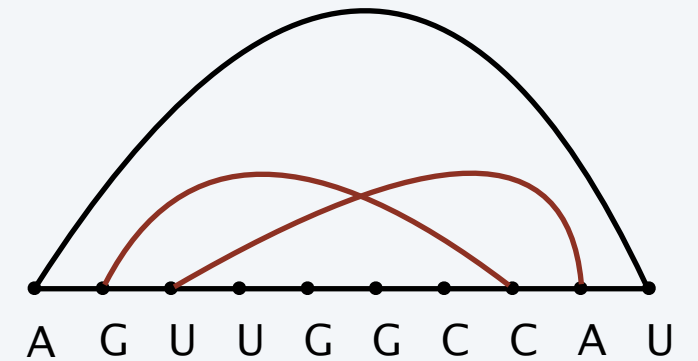
base pair



ok



sharp turn  
( $\leq 4$  intervening bases)



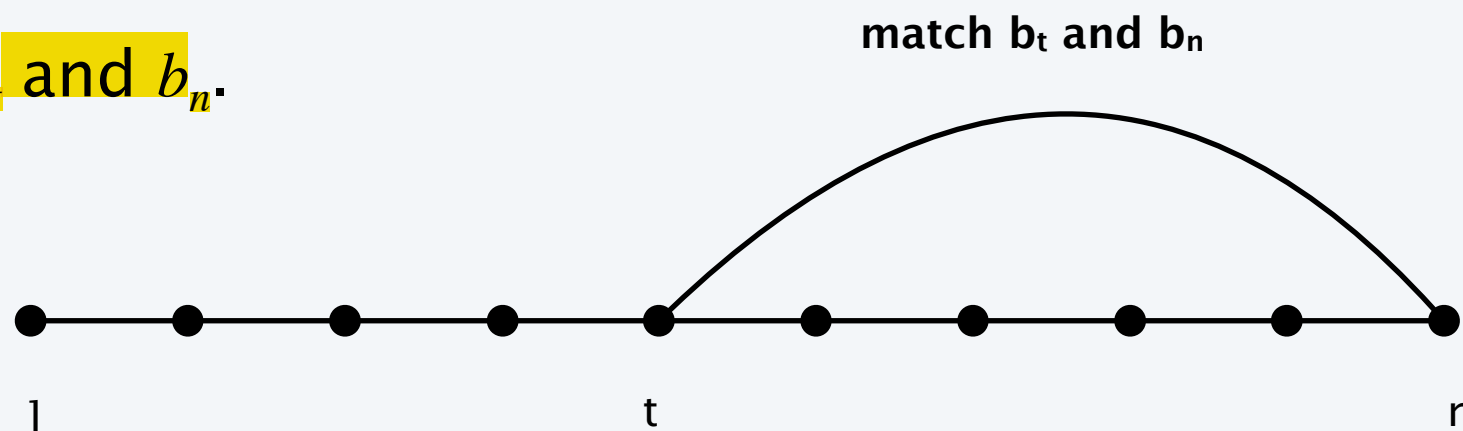
crossing

# RNA secondary structure: subproblems

---

First attempt.  $OPT(j)$  = maximum number of base pairs in a secondary structure of the substring  $b_1 b_2 \dots b_j$ .

Choice. Match  $b_t$  and  $b_n$ .



Difficulty. Results in two subproblems but one of wrong form.

- Find secondary structure in  $b_1 b_2 \dots b_{t-1}$ .  $\longleftarrow OPT(t-1)$
- Find secondary structure in  $b_{t+1} b_{t+2} \dots b_{n-1}$ .  $\longleftarrow$  need more subproblems

# Dynamic programming over intervals

---

**Notation.**  $OPT(i, j)$  = 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  $i \leq t < j - 4$ .

- Noncrossing constraint decouples resulting 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

# Bottom-up dynamic programming over intervals

Q. In which order to solve the subproblems?

A. Do shortest intervals first.

```
RNA( $n, b_1, \dots, b_n$ )
```

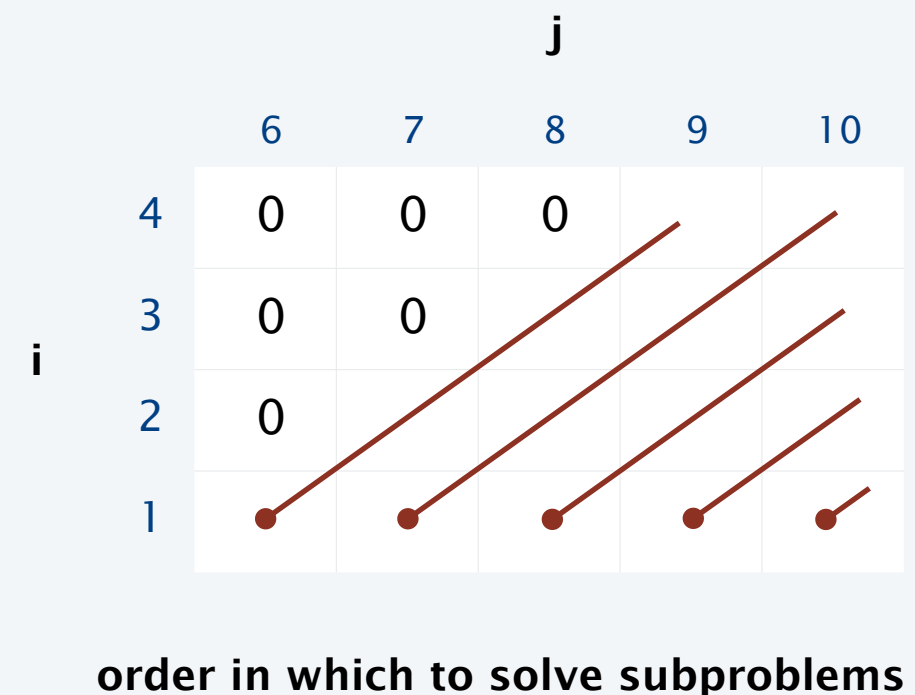
```
FOR  $k = 5$  TO  $n - 1$ 
```

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

```
     $j \leftarrow i + k.$ 
```

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

```
RETURN  $M[1, n].$ 
```



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

# Dynamic programming summary

---

## Outline.

- Polynomial number of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from smallest to largest, with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

## Techniques.

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

Top-down vs. bottom-up. Different people have different intuitions.