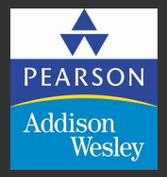


6. Dynamic Programming I

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *dary structure*

Lecture slides by Kevin Wayne

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

Last updated on 1/15/20 6:20 AM

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.

Dynamic programming applications

Application areas.

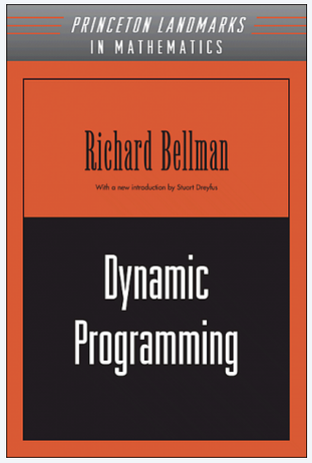
- Computer science: AI, compilers, systems, graphics, theory, ...
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

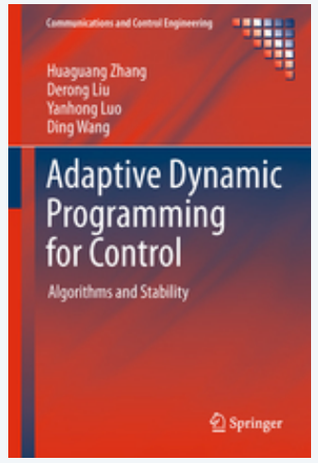
Some famous dynamic pro

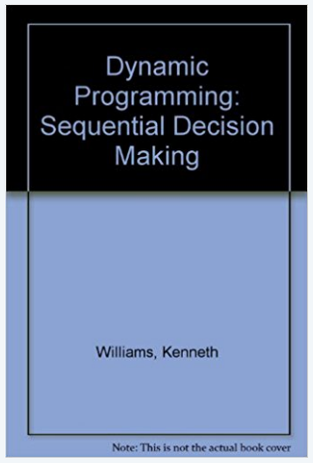
- Avidan–Shamir for seam carving.

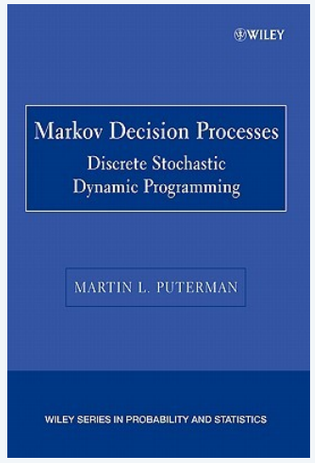
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in TEX.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

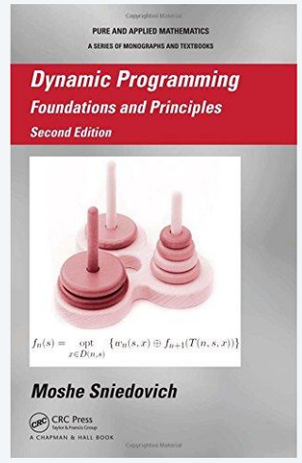
Dynamic programming books

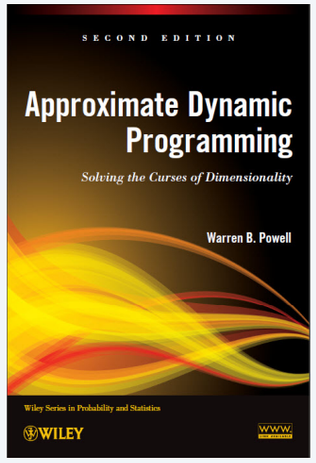


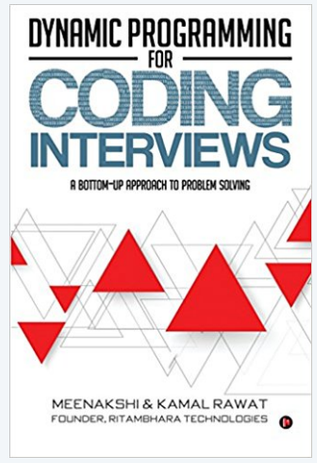


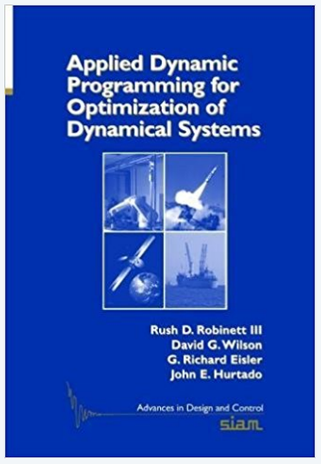


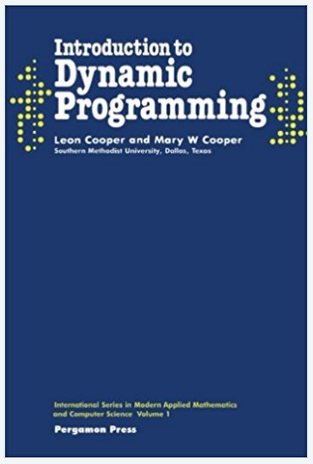


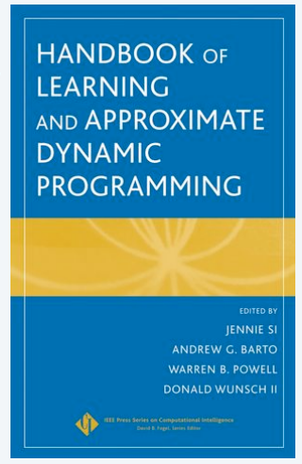


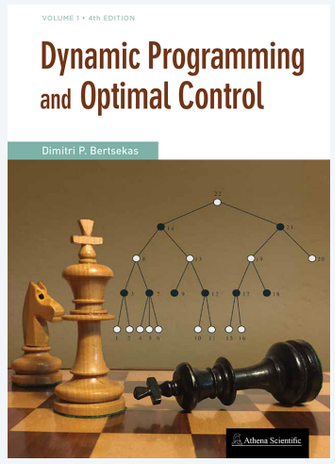


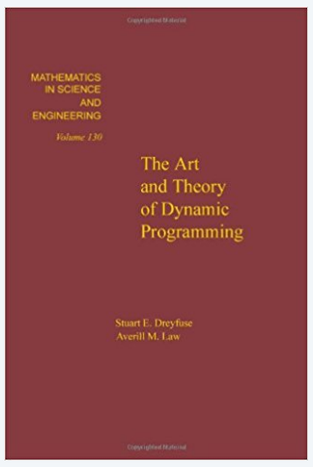


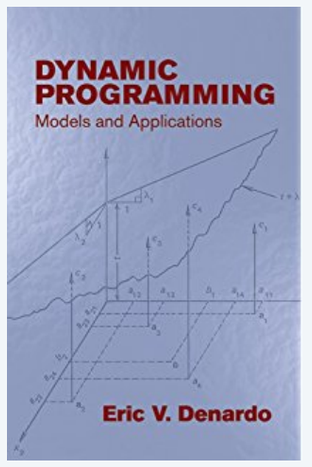


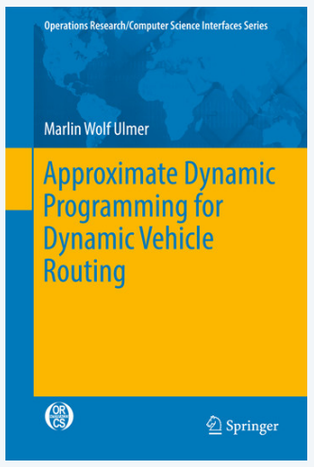


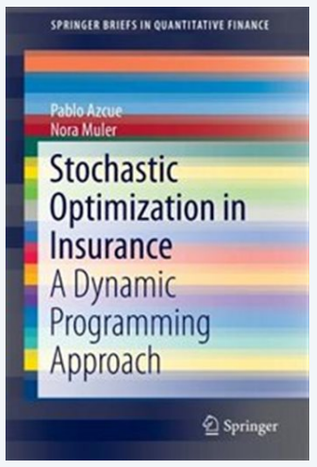


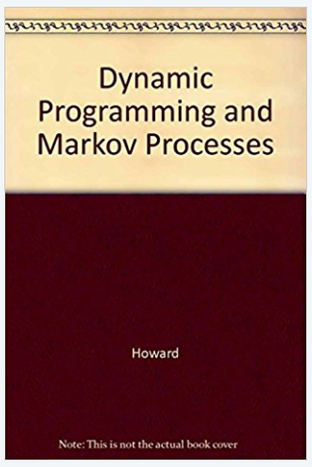


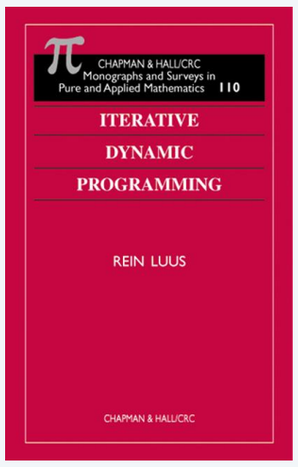


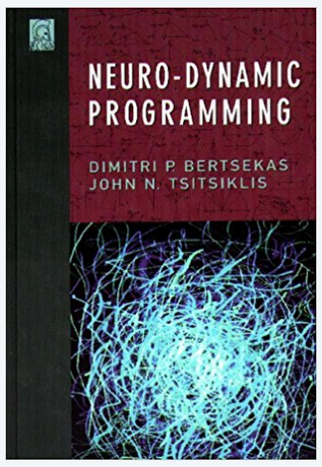


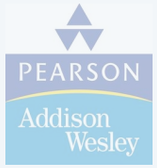












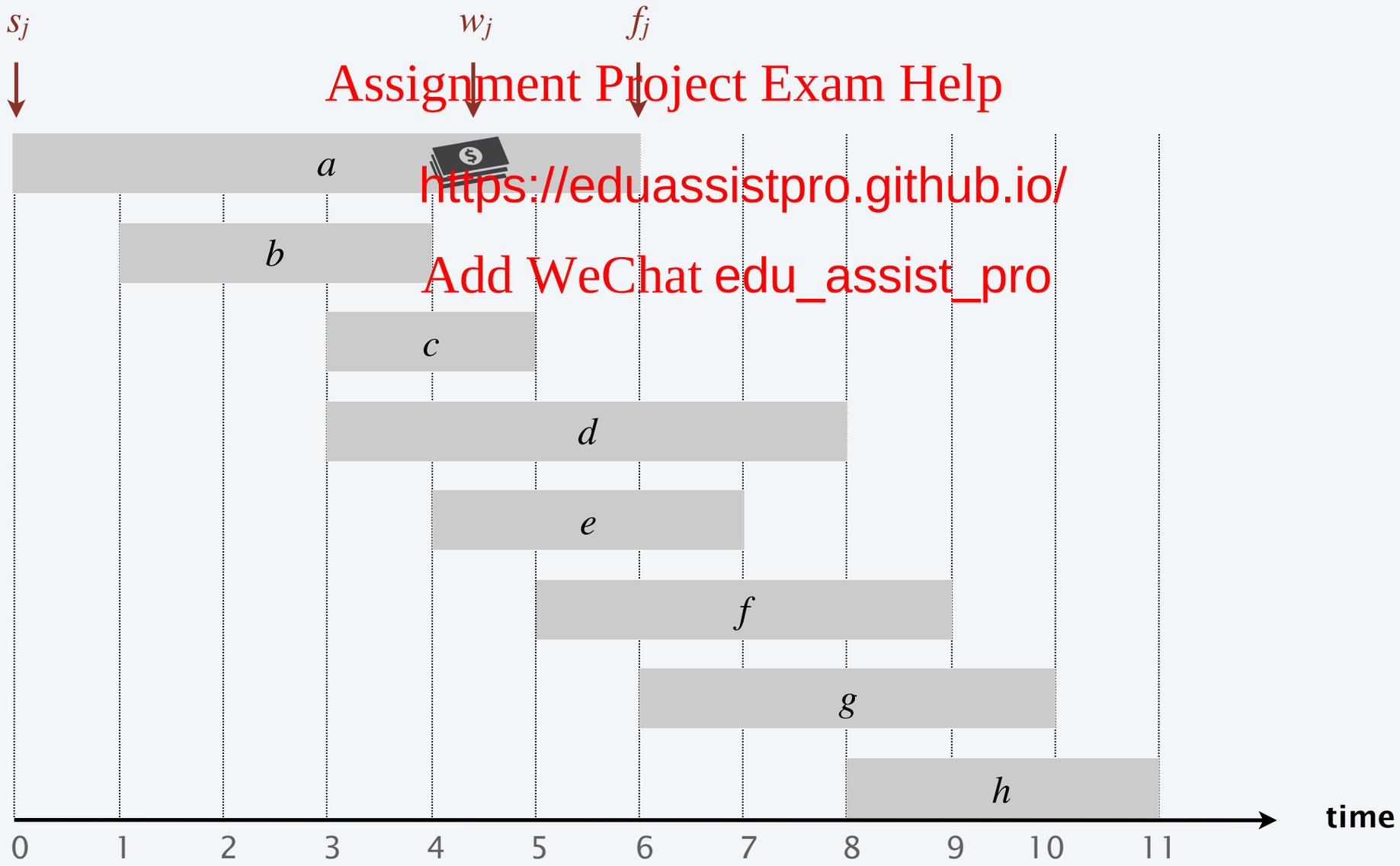
6. DYNAMIC PROGRAMMING I

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

SECTIONS 6.1–6.2

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.



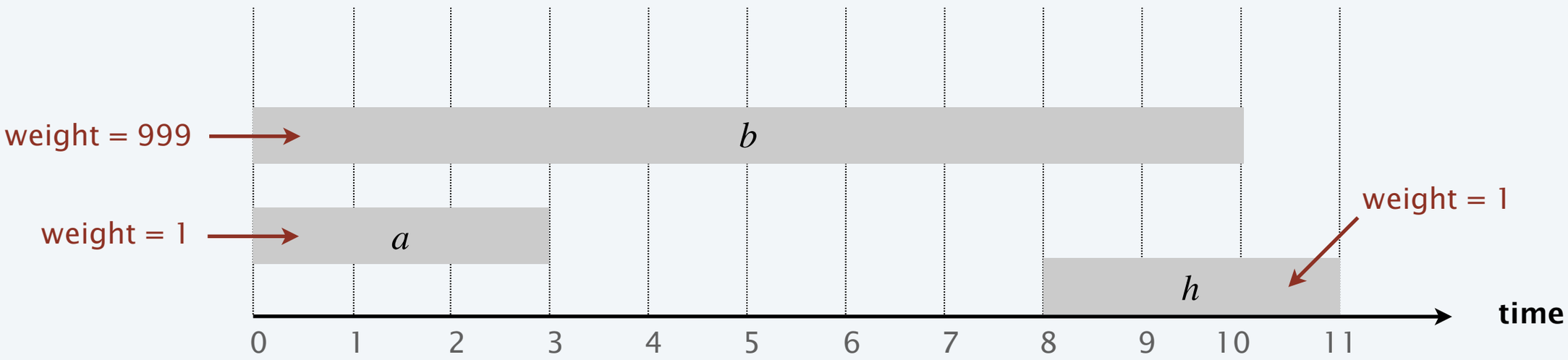
Earliest-finish-time first algorithm

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 can fail spectacularly for weighted version.



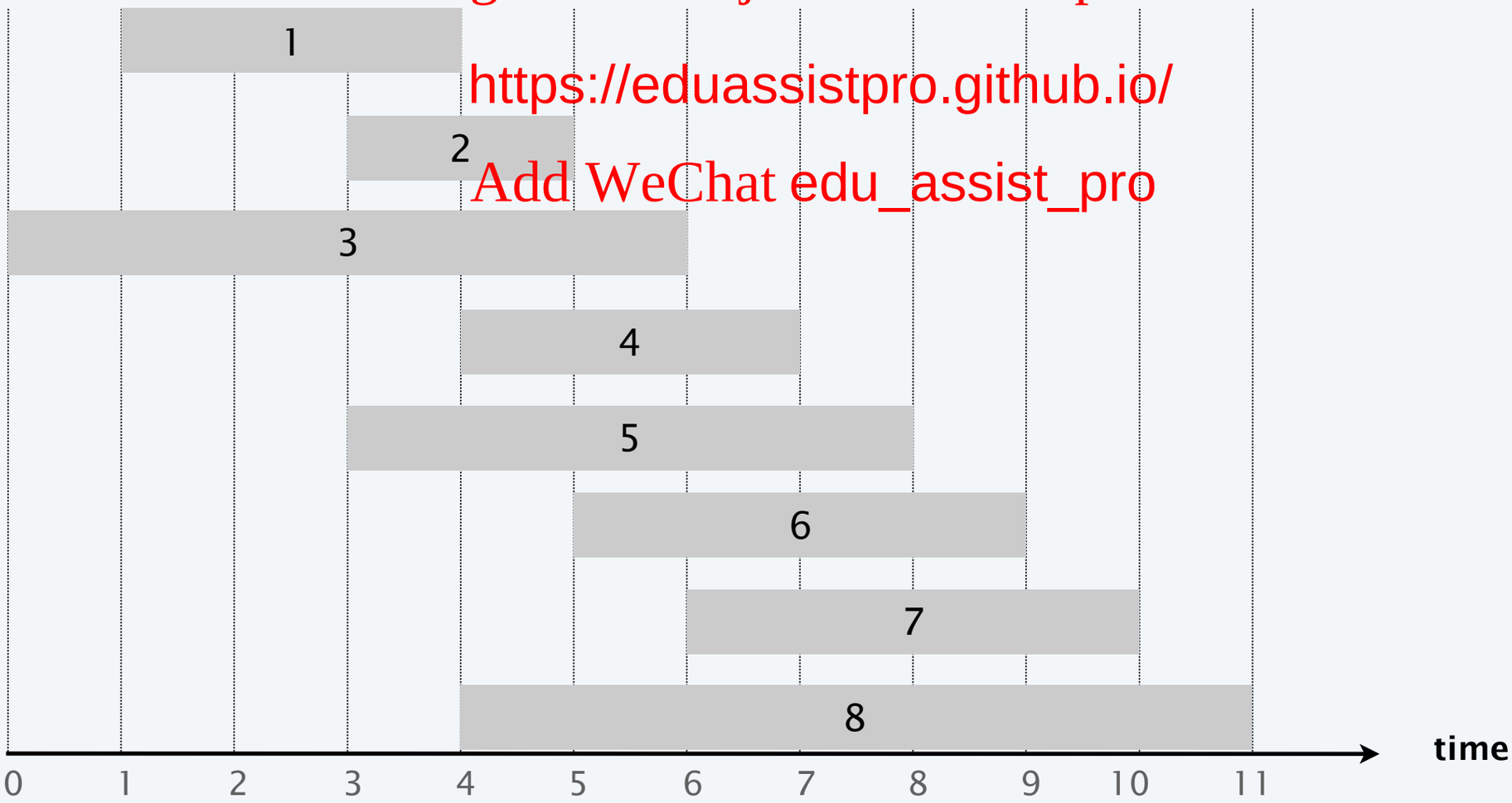
Weighted interval scheduling

Convention. Jobs are in ascending order of finish time: $f_1 \le f_2 \le \ldots \le f_n$.

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 $1, 2, ..., j$.

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

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

- Must include optimal solution to jobs $1, 2, ..., j-1$.

optimal substructure property
(proof via exchange argument)

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

- Collect profit w_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 $1, 2, ..., p(j)$.

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 $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

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

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

RETURN COMPUTE-O

COMPUTE-OPT(j)

IF $(j = 0)$

  RETURN 0.

ELSE

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

Dynamic programming: quiz 1

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

A. $\Theta(n \log n)$

B. $\Theta(n^2)$

C. $\Theta(1.618^n)$

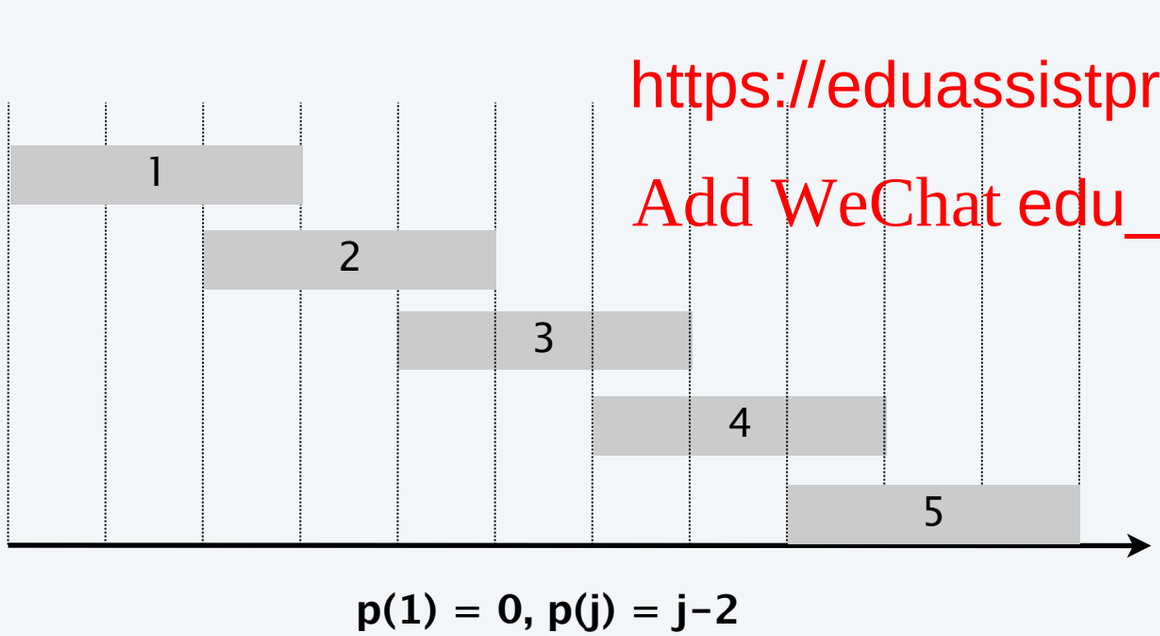
D. $\Theta(2^n)$

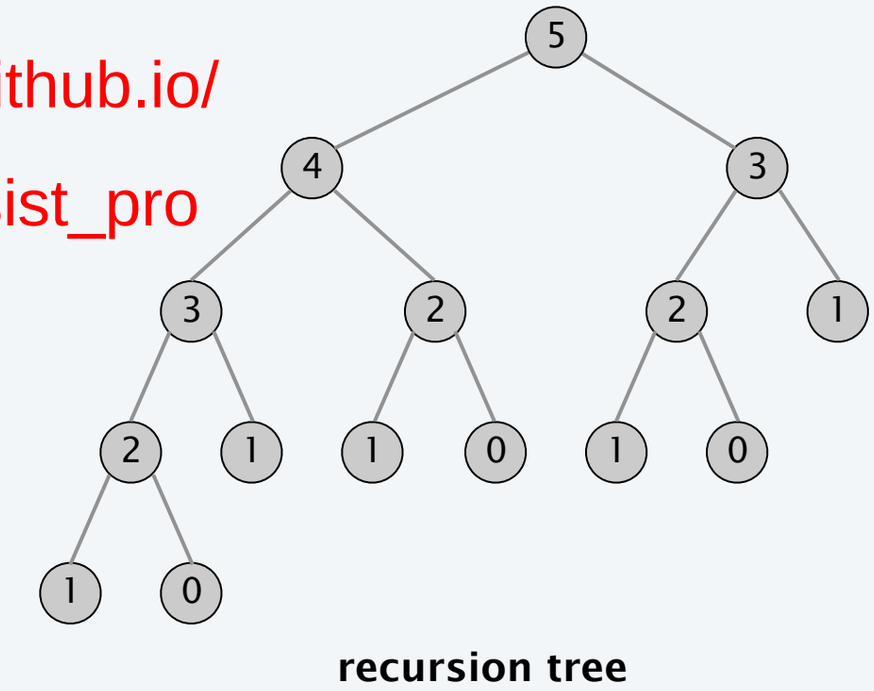

```
COMPUTE-OPT(j)

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.





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$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

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

Compute $p[1], p[2], \ldots, p[$

$M[0] \leftarrow 0$. ← global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{$ M-COMPUTE-OPT$(j-1)$, w_j + 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 initiali
  - (2) calls itself recursive

- Progress measure $\Phi = \#$ initialized entries among $M[1..n]$.
  - initially $\Phi = 0$; throughout $\Phi \leq n$.
  - (2) increases Φ by $1 \Rightarrow \leq 2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. ▪

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

ELSE IF ($w_j + M[p[j]] \geq M[j-1]$)

  RETURN $\{ j \} \cup$ 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($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

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

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

$M[0] \leftarrow 0$.

FOR $j = 1$ TO n

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

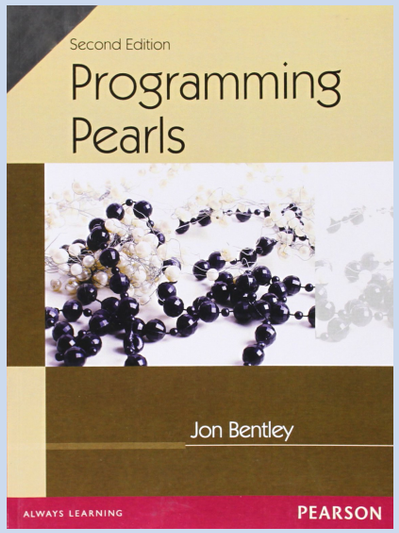
Running time. The bottom-up version takes $O(n \log n)$ time.

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

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



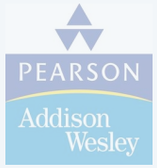
Maximum rectangle problem

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

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

13

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



6. DYNAMIC PROGRAMMING I

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

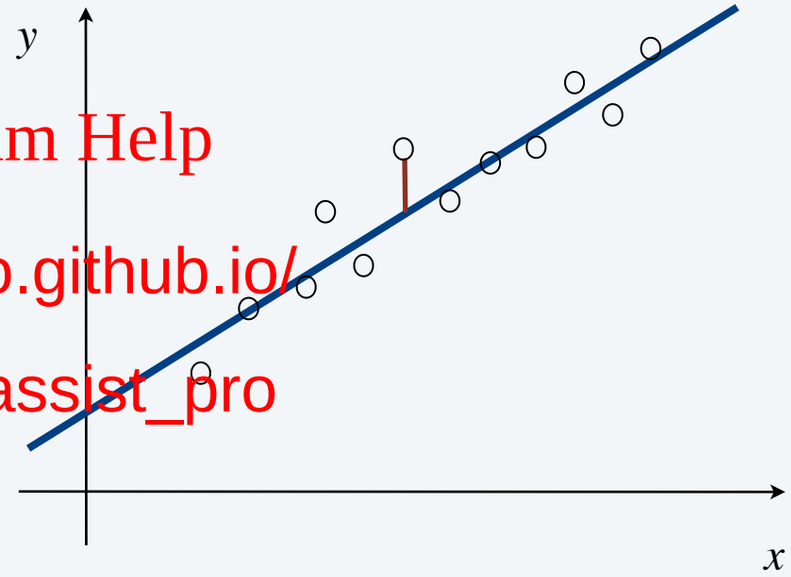
SECTION 6.3

Least squares

Least squares. Foundational problem in statistics.

- 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

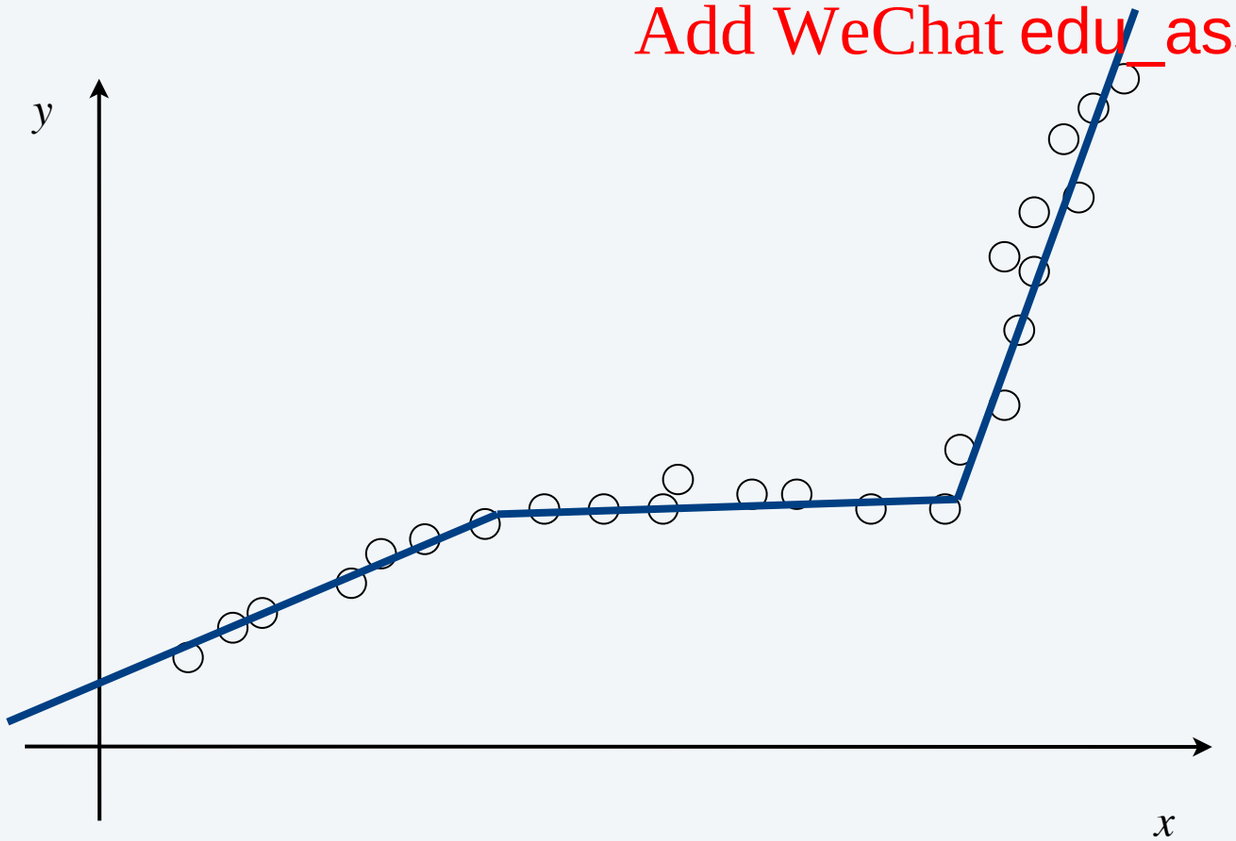
Segmented least squares.

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

goodness of fit

number of lines



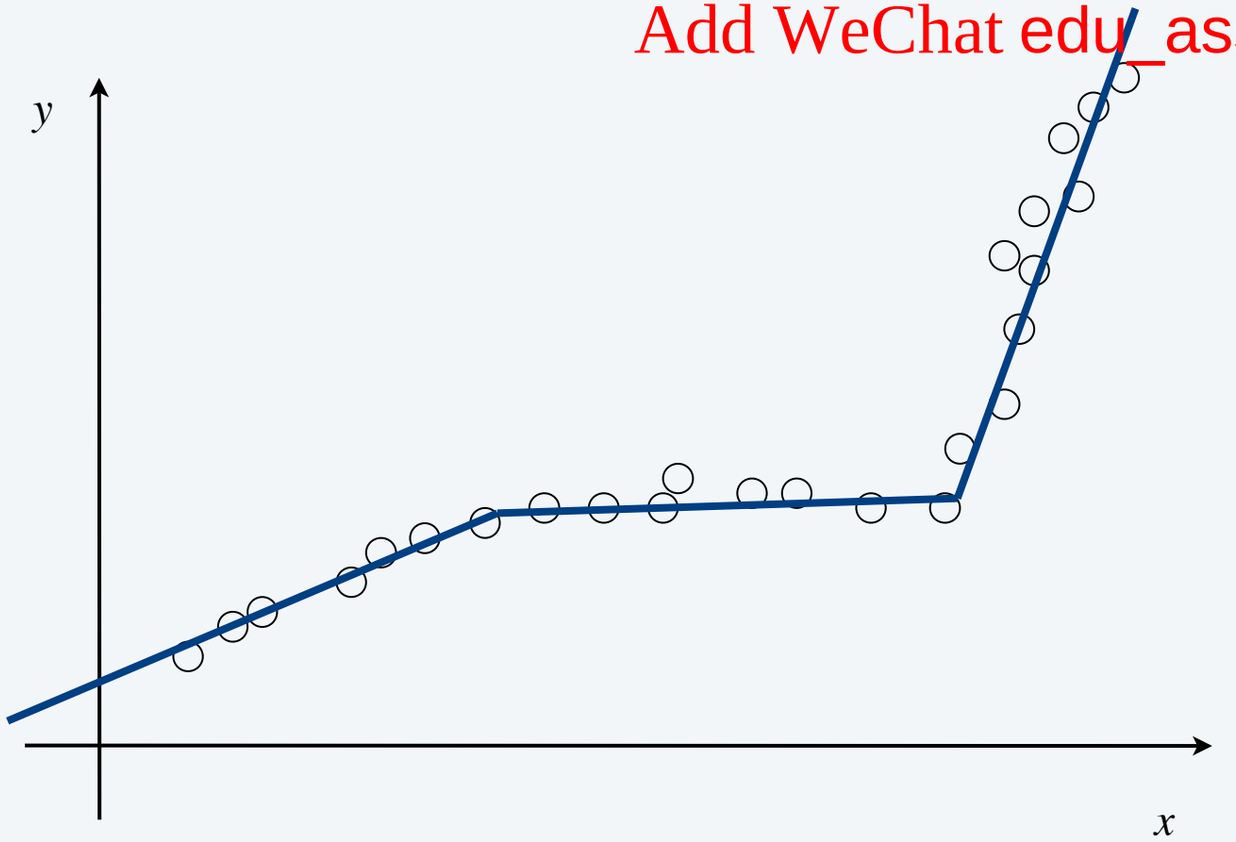
Segmented least squares

Segmented least squares.

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

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

- E = sum of the sums of the
- L = number of lines.



Dynamic programming: multiway choice

Notation.

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

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i \le j$.
- Cost = $e_{ij} + c + OPT(i -$ perty
  meant)

Bellman equation.

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

Segmented least squares algorithm

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

FOR j = 1 TO n

  FOR i = 1 TO j

    Compute the SSE e_{ij} for $p_i, \ldots, p_j$.

$M[0] \leftarrow 0$.

FOR $j = 1$ TO n

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

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.

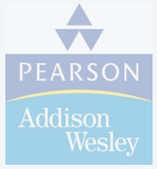
$$a_{ij} = \frac{n\sum_k x_k y_k - \left(\sum_k y_k - a_{ij}\sum_k x_k\right)\sum_k x_k}{n\sum_k x_k^2 - \left(\sum_k x_k\right)^2} \qquad e_{ij} = \sum_k \left(y_k - a_{ij} x_k - b_{ij}\right)^2$$

- $O(n)$ to compute e_{ij}. ▪

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

- For each i : precompute cumulative sums $\sum_{k=1}^{i} x_k, \quad \sum_{k=1}^{i} y_k, \quad \sum_{k=1}^{i} x_k^2, \quad \sum_{k=1}^{i} x_k y_k$.

- Using cumulative sums, can compute e_{ij} in $O(1)$ time.



6. DYNAMIC PROGRAMMING I

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

SECTION 6.4

Knapsack problem

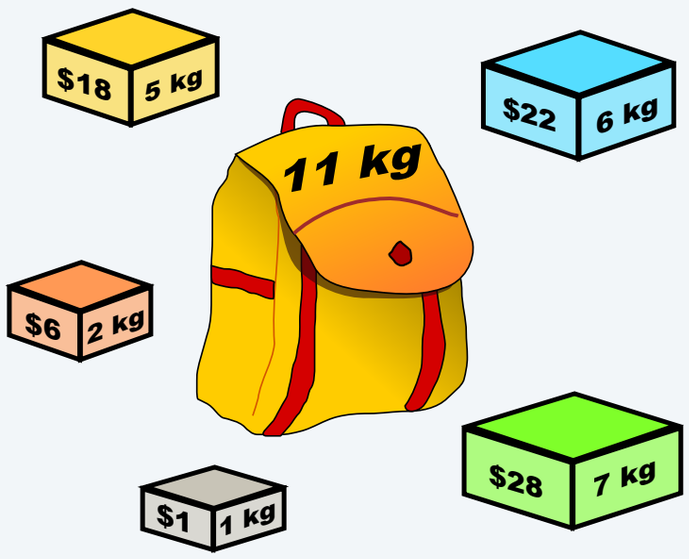
Goal. Pack knapsack so as to maximize total value of items taken.

- 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 integers.



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)

weights and values can be arbitrary positive integers

Dynamic programming: quiz 2

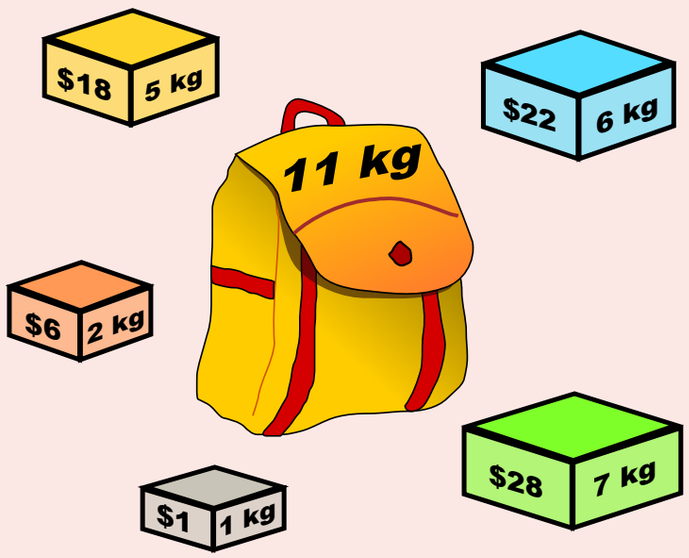
Which algorithm solves knapsack problem?

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.



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)

Dynamic programming: quiz 3

Which subproblems?

A. $OPT(w)$ = optimal value of knapsack problem with weight limit w.

B. $OPT(i)$ = optimal value of knapsack problem with items $1, \ldots, i$.

C. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$ subject to weight li

D. Any of the above.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, 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 $\{ 1, 2, \ldots, i-1 \}$ subject to weight limit w.

Case 2. $OPT(i, w)$ selects it

optimal substructure property
(proof via exchange argument)

- Collect value v_i.
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ 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($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)

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

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

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

subset of items 1, …, i \ limit w	0	1	2	3	4	5	6	7	8	9	10	11
{ }	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, …, 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 $\Theta(n\,W)$ time and $\Theta(n\,W)$ space.

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\,W)$ table entries.
- After computing optimal values, can trace back to find solution: $OPT(i, w)$ takes item i iff ▪

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.

Dynamic programming: quiz 4

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.

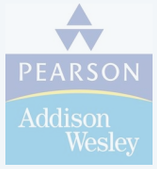
COIN CHANGING

Problem. Given n coin denominations $\{ c_1, c_2, \ldots, c_n \}$ 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, 10

Optimal. 140¢ = 70 + 70.



6. DYNAMIC PROGRAMMING I

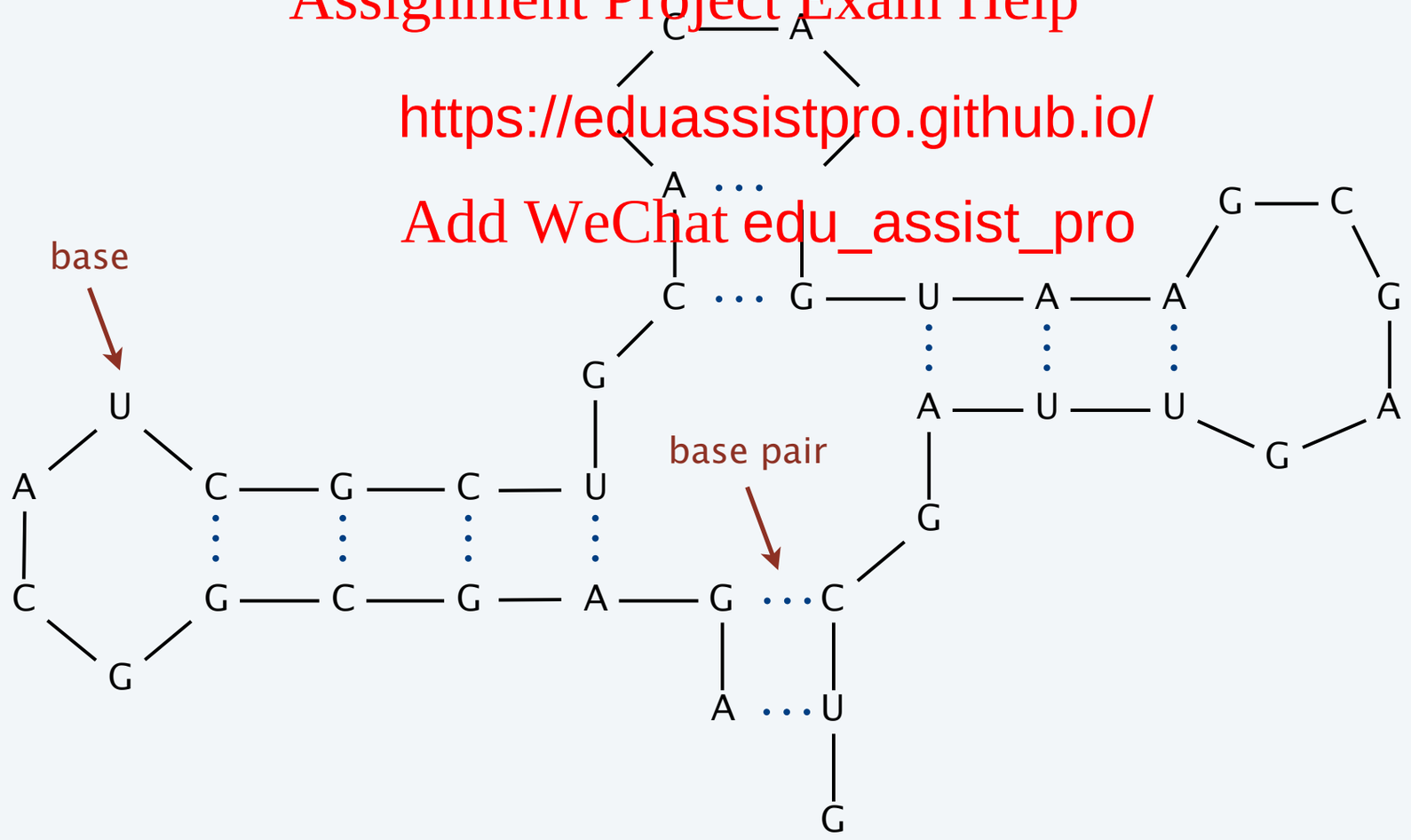
- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

SECTION 6.5

RNA secondary structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet $\{ \mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{U} \}$.

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.

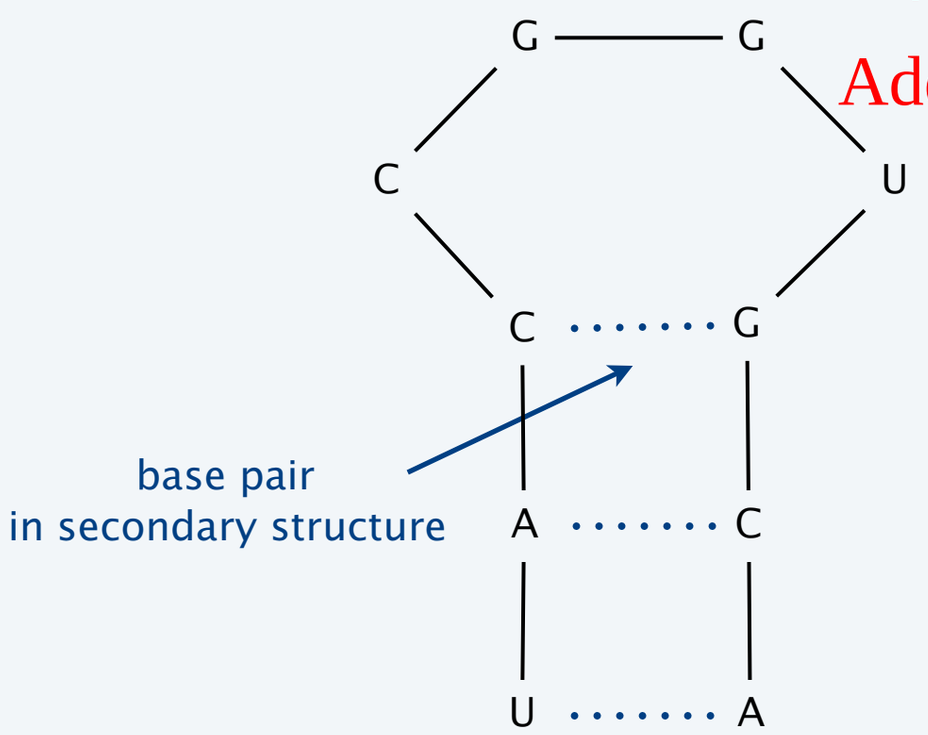


RNA secondary structure for GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

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.



$B = \text{ACGUGGCCCAU}$

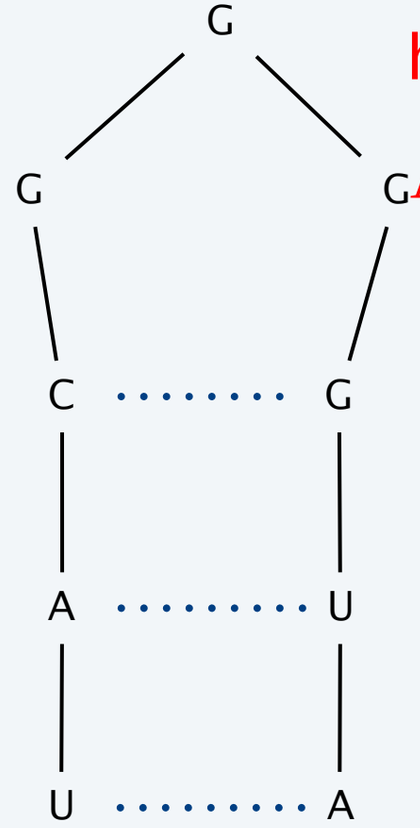
$S = \{ (b_1, b_{10}),\ (b_2, b_9),\ (b_3, b_8) \}$

S is not a secondary structure
(C–A is not a valid Watson–Crick pair)

RNA secondary structure

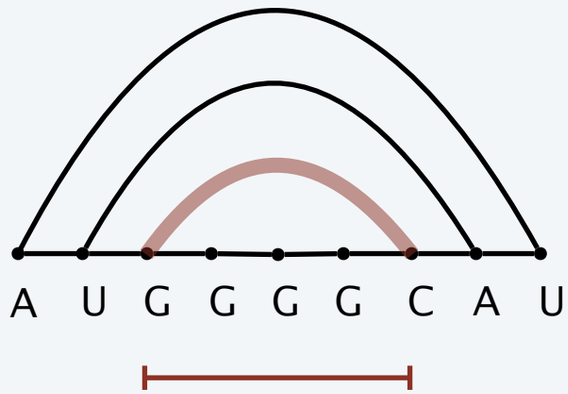
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [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 $i < j - 4$.



$B = \text{AUGGGGCAU}$

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

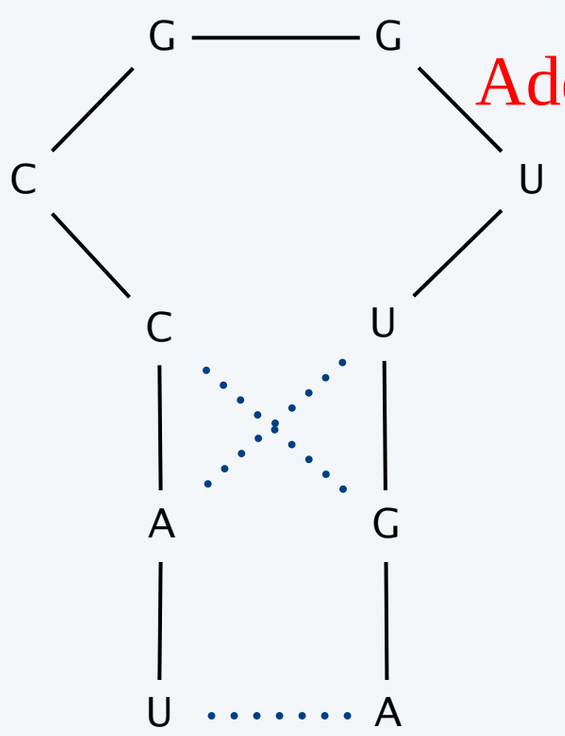


S is not a secondary structure
(≤4 intervening bases between G and C)

RNA secondary structure

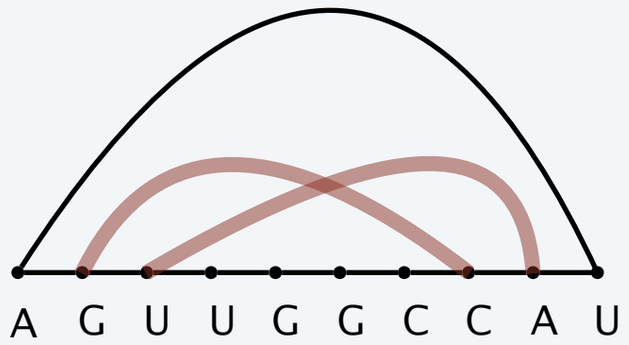
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [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_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.



$B = \text{ACUUGGCCAU}$

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

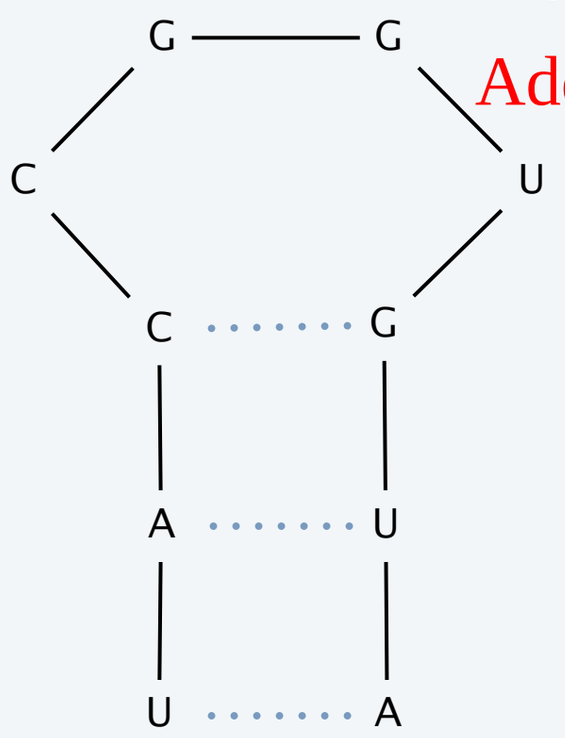


S is not a secondary structure (G–C and U–A cross)

RNA secondary structure

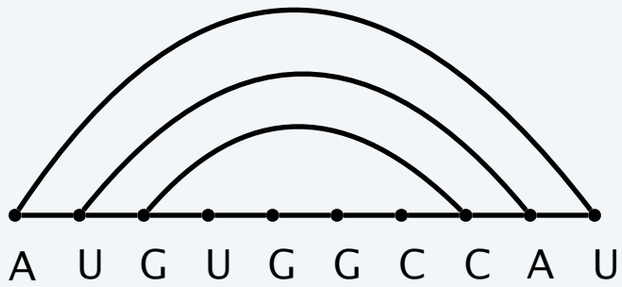
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [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_ℓ) 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_ℓ) 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 ⇒ lower free energy)

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

Dynamic programming: quiz 5

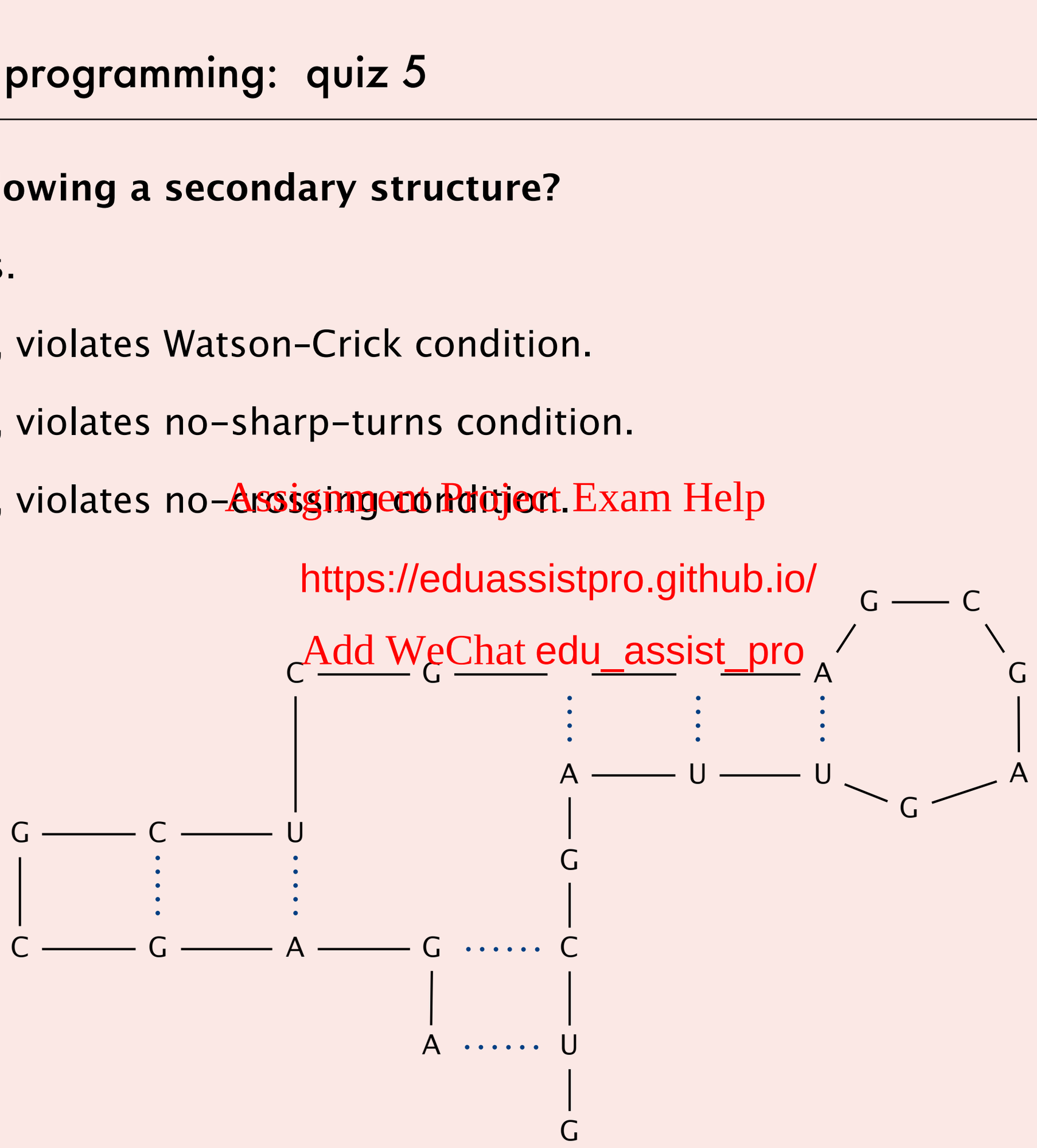
Is the following a secondary structure?

A. Yes.

B. No, violates Watson–Crick condition.

C. No, violates no–sharp–turns condition.

D. No, violates no–crossing condition.



Dynamic programming: quiz 6

Which subproblems?

A. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_1 b_2 \ldots b_j$.

B. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_j b_{j+1} \ldots b_n$.

C. Either A or B.

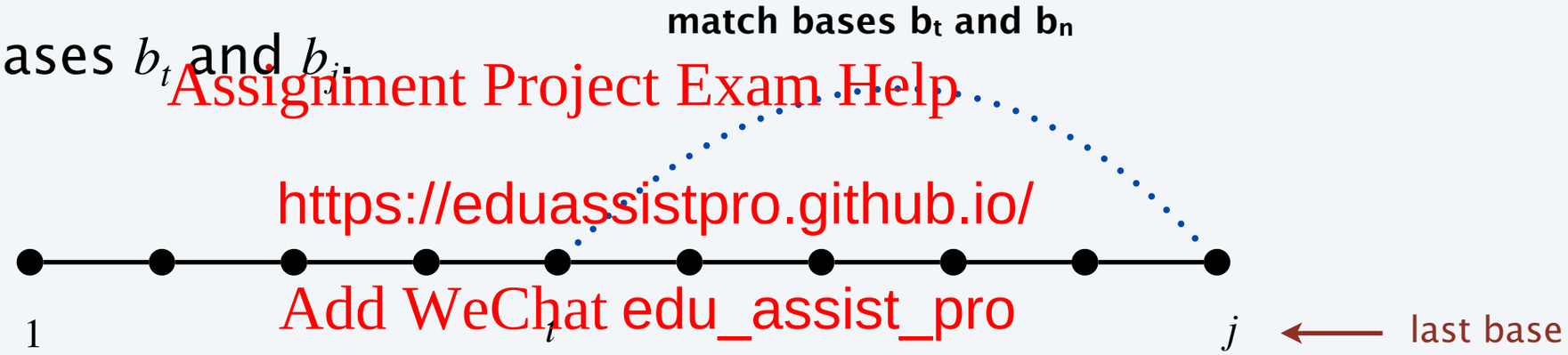
D. Neither A nor B.

RNA secondary structure: subproblems

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

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 $b_1 b_2 \ldots b_{t-1}$. ← $OPT(t-1)$
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{j-1}$. ← 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 $b_i b_{i+1} \ldots b_j$.

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

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

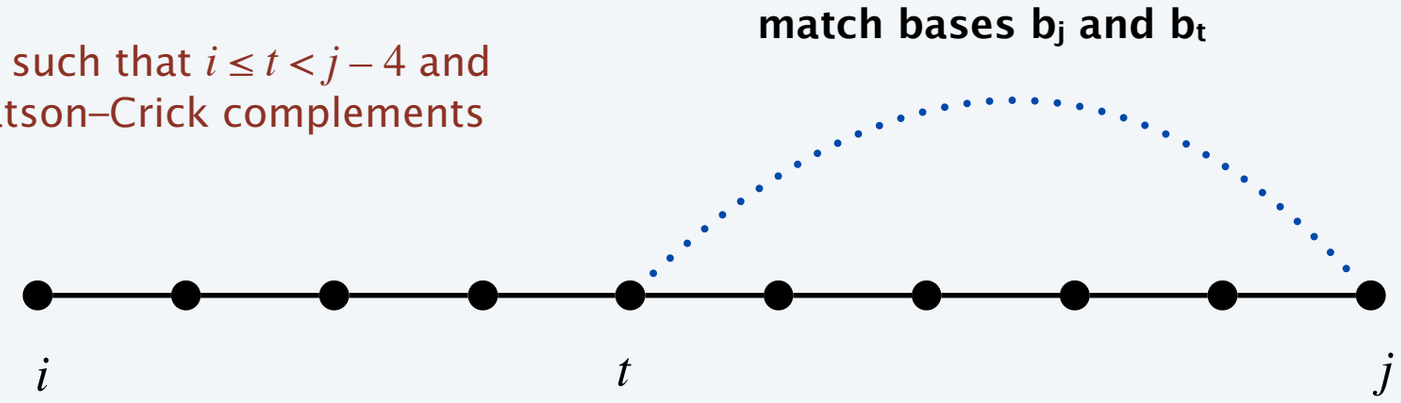
Case 2. Base b_j is not invol

- $OPT(i, j) = \mathrm{OPT}(i, j-1)$.

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



Dynamic programming: quiz 7

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$(n, b_1, \ldots, b_n)$

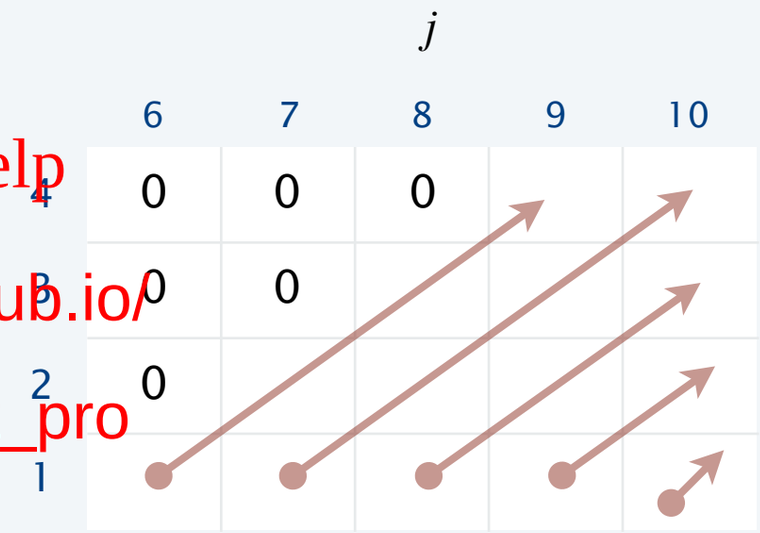
FOR $k = 5$ TO $n - 1$

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

    $j \leftarrow i + k$.

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

RETURN $M[1, n]$.



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.

typically, only a polynomial number of subproblems

- 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.

Techniques.

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

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