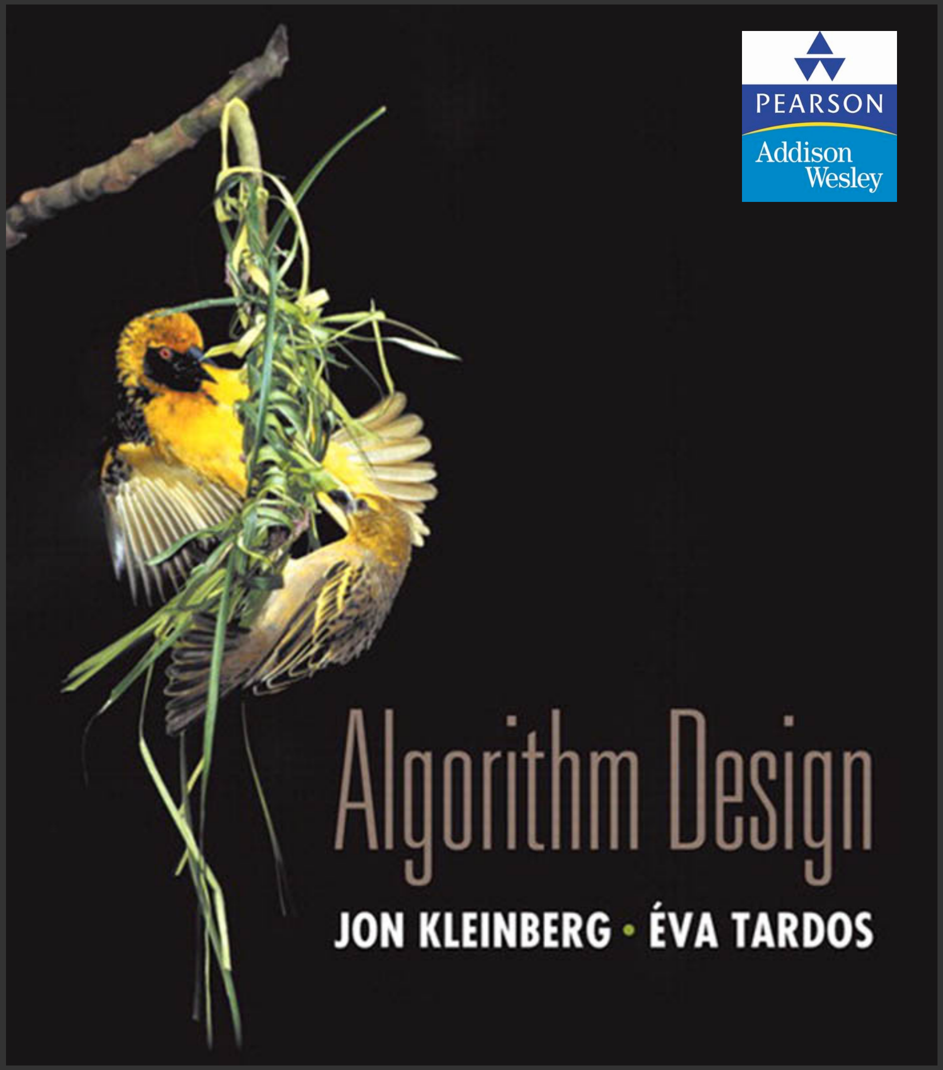


6. Dynamic Programming I

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

Lecture slides by Kevin Wayne

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

Last updated on 4/4/18 5:30 AM

Algorithmic paradigms

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

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.

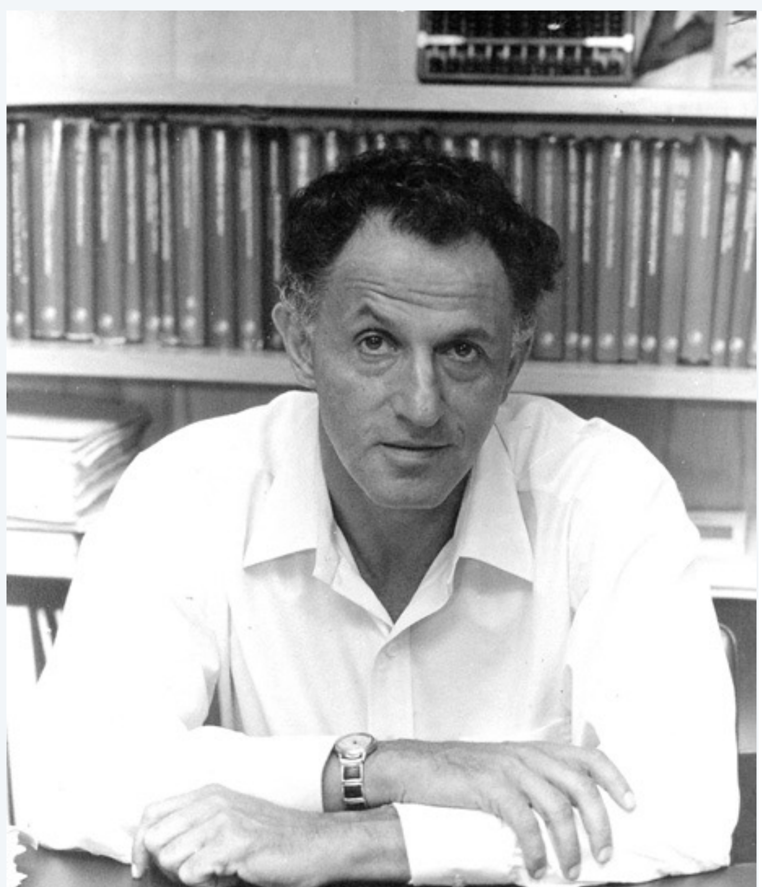
fancy name for
caching intermediate results
in a table for later reuse

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

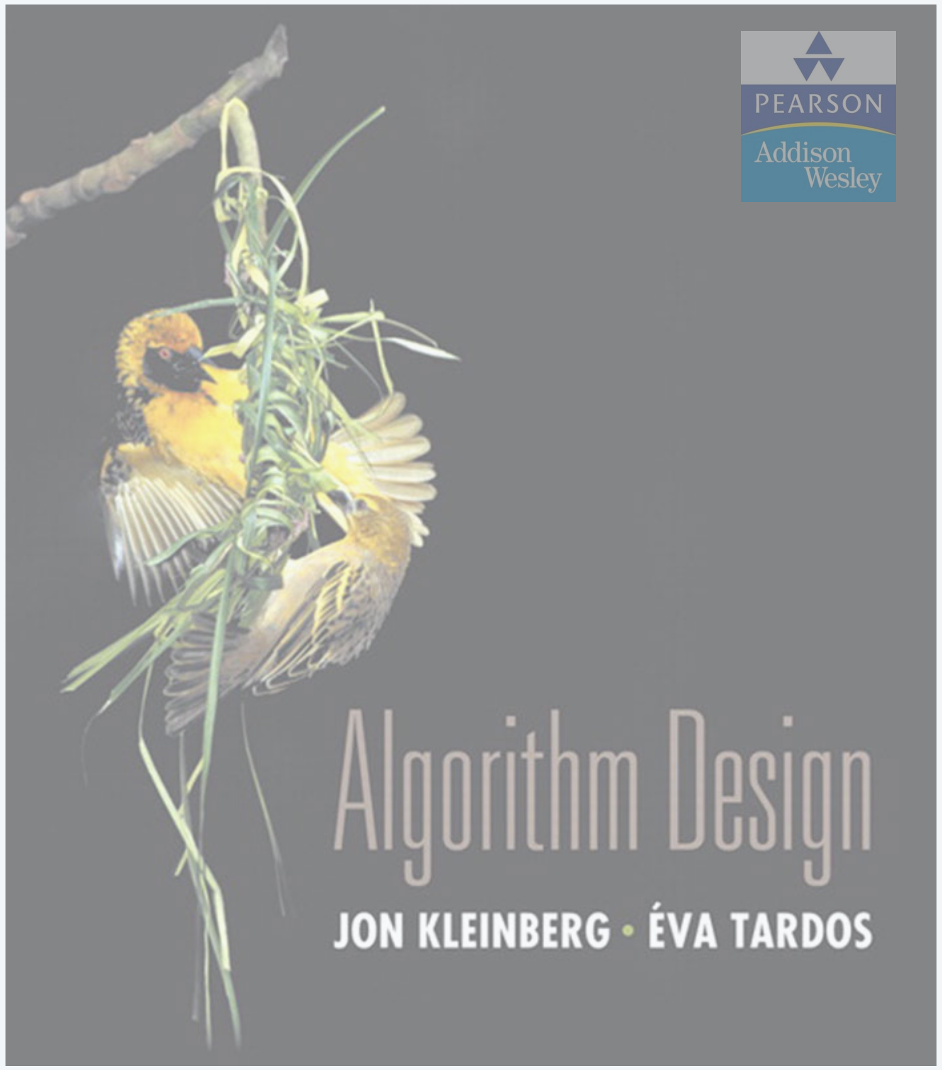
Application areas.

- 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 $\TeX$.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

Dynamic programming books



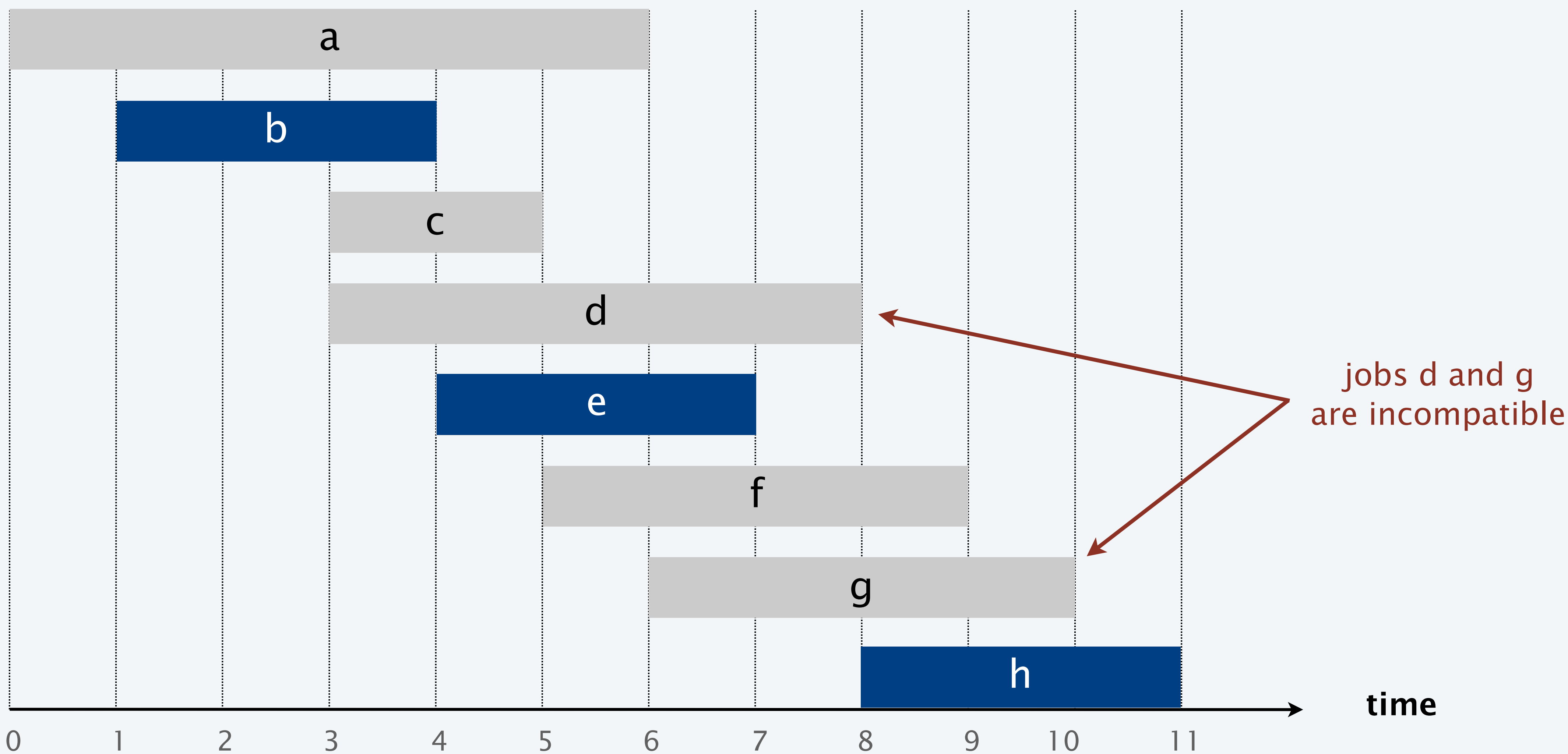
SECTION 4.1

4. GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Interval scheduling

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



Greedy algorithms I: quiz 2

Consider jobs in some order, taking each job provided it's compatible with the ones already taken. Which rule is optimal?

A. [Earliest start time] Consider jobs in ascending order of s_j.

B. [Earliest finish time] Consider jobs in ascending order of f_j.

C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. None of the above.

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

  IF job j is compatible with S

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

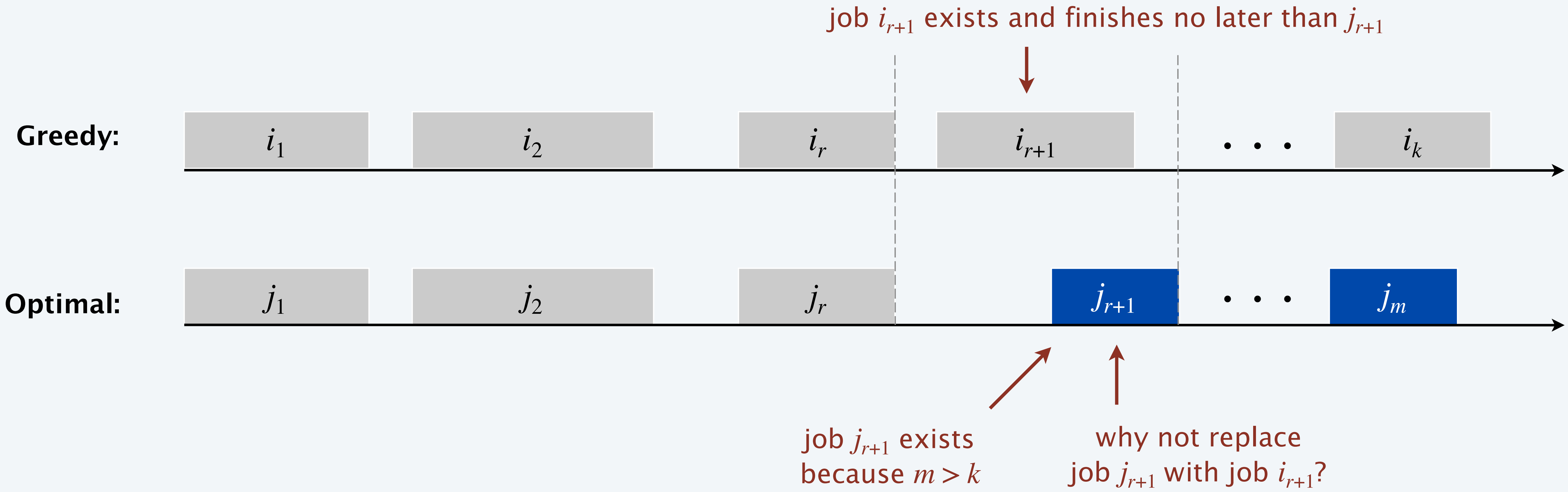
- Keep track of job j^* that was added last to S.
- Job j is compatible with S iff $s_j \ge f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
- Let j_1, j_2, ... j_m denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, ..., i_r = j_r$ for the largest possible value of r.

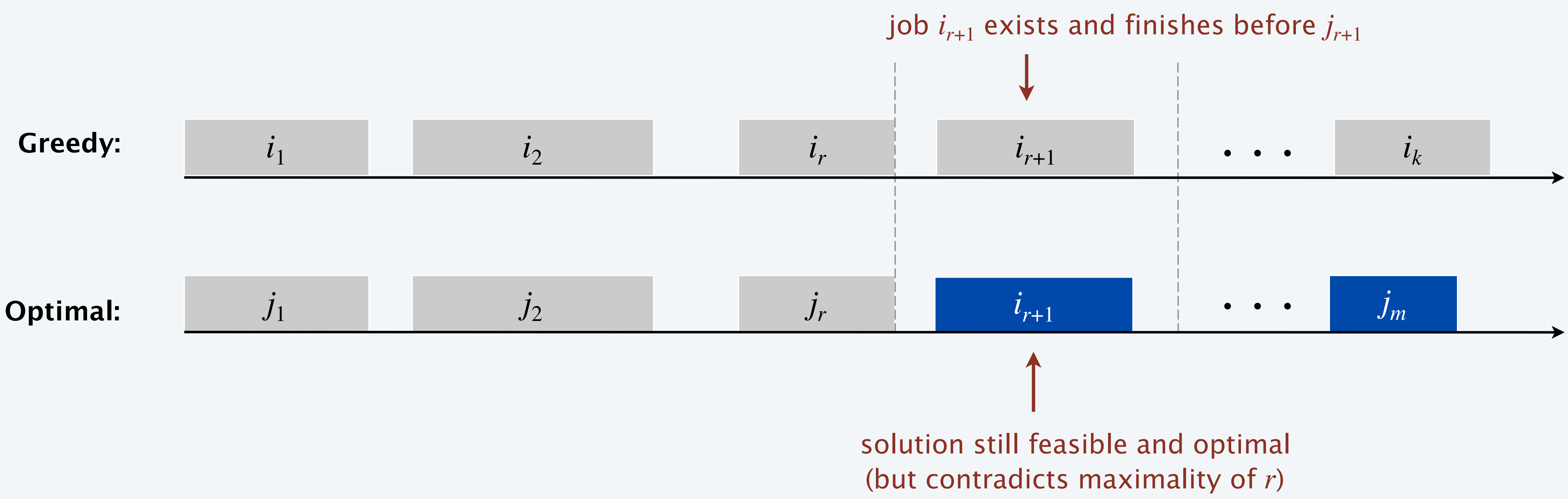


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
- Let j_1, j_2, ... j_m denote set of jobs in an optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



Greedy algorithms I: quiz 3

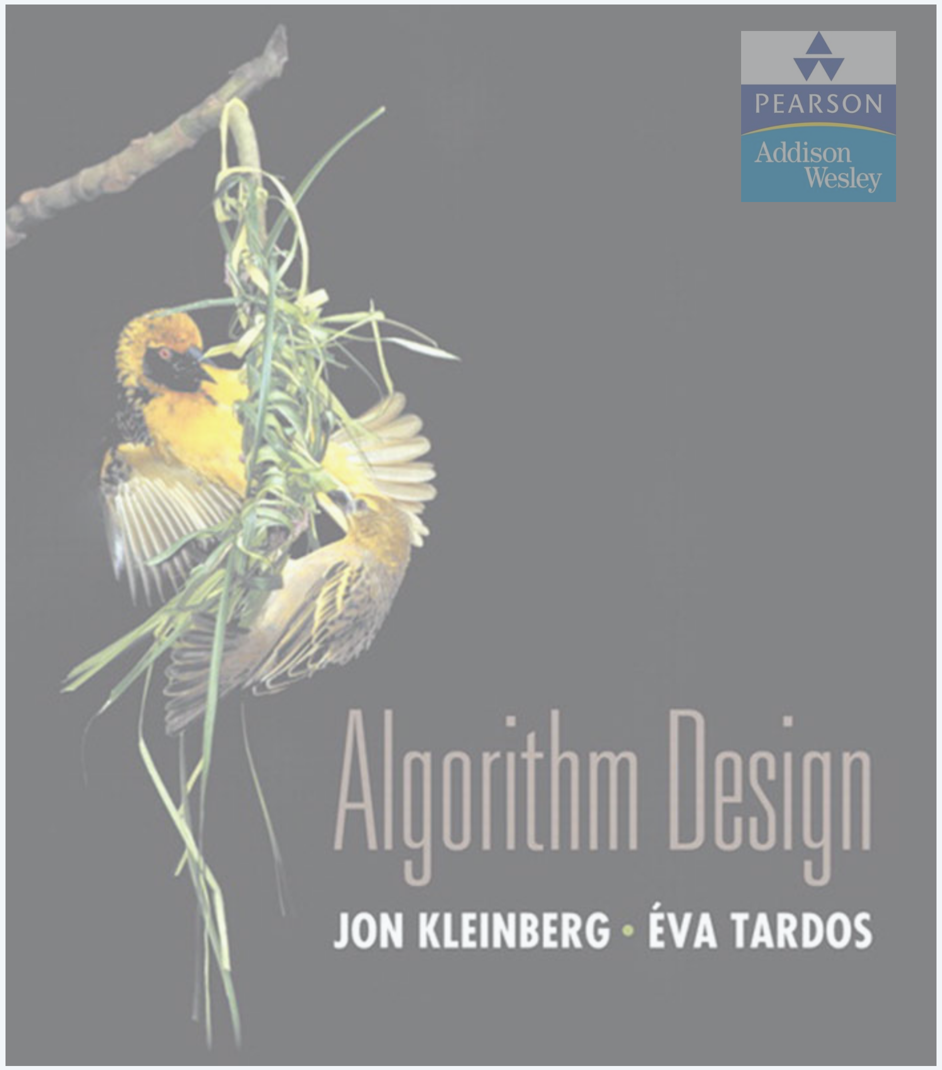
Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is the earliest-finish-time-first algorithm still optimal?

A. Yes, because greedy algorithms are always optimal.

B. Yes, because the same proof of correctness is valid.

C. No, because the same proof of correctness is no longer valid.

D. No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.



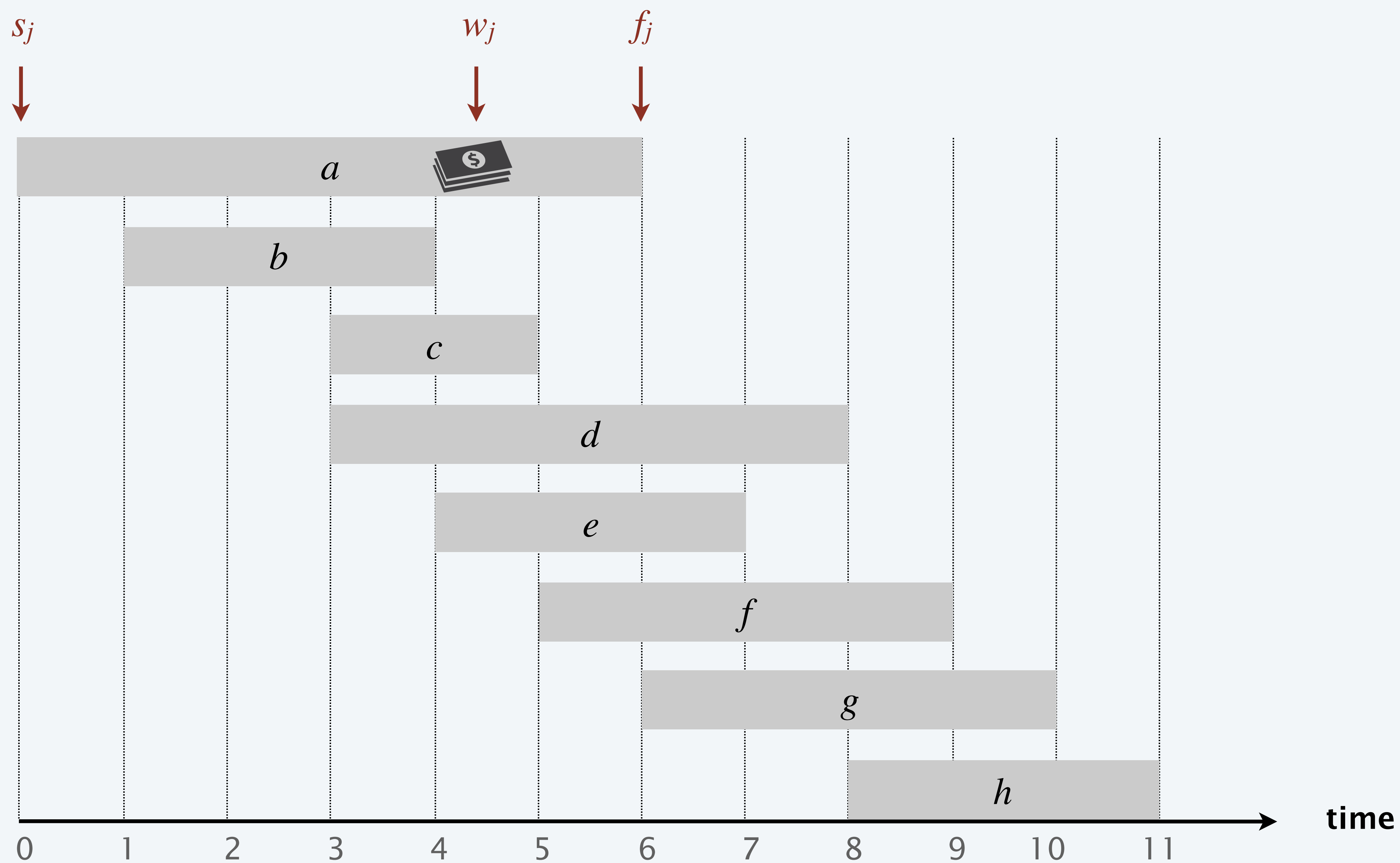
SECTIONS 6.1–6.2

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.



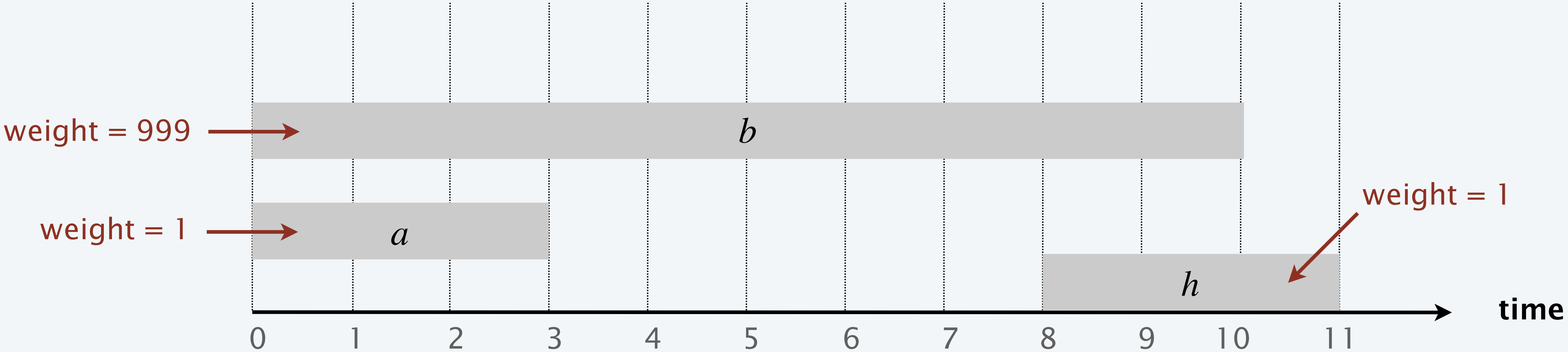
Earliest-finish-time first algorithm

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.



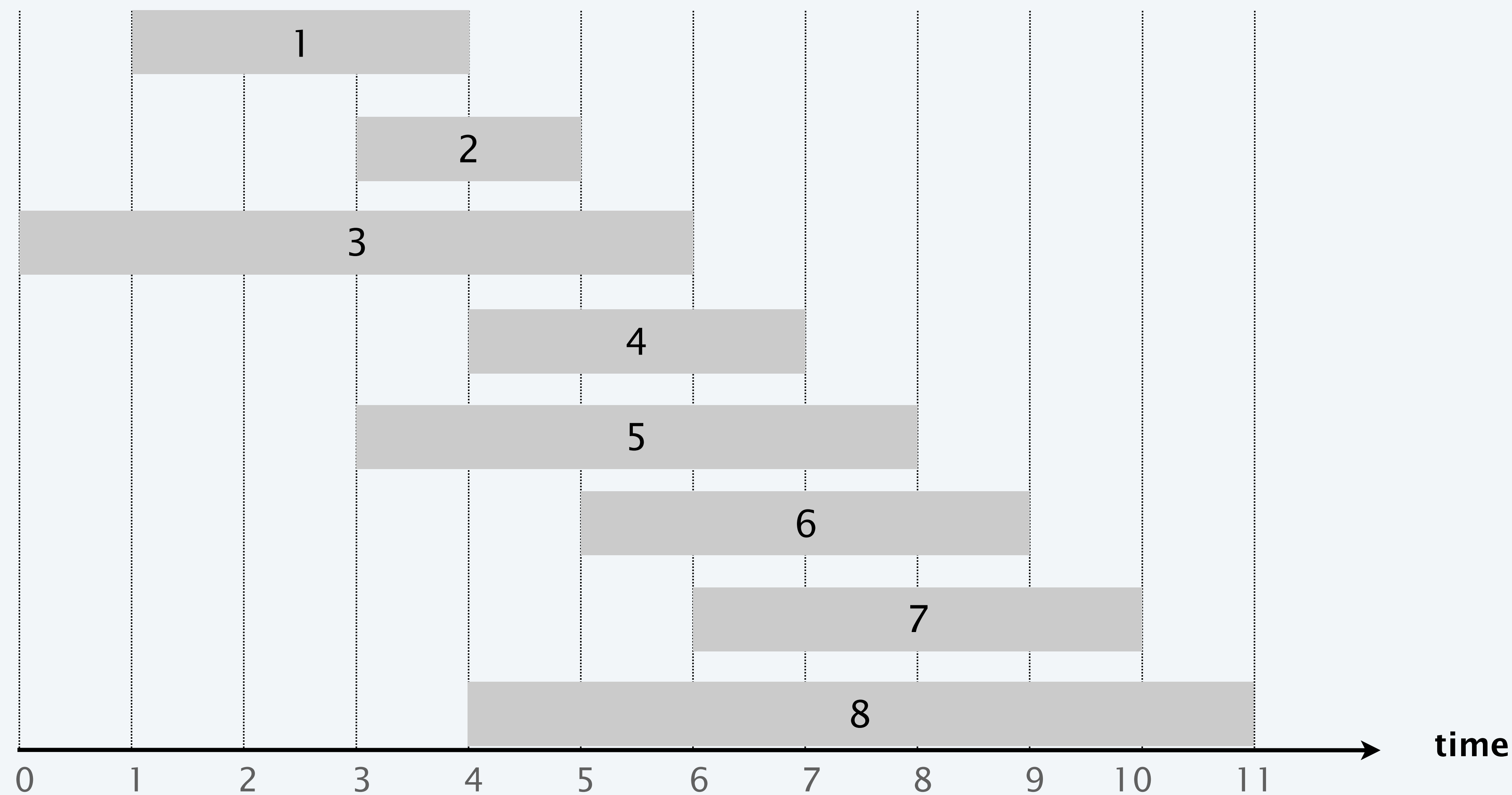
Weighted interval scheduling

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

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

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

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 $f_1 \le f_2 \le \ldots \le f_n$.

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

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

  RETURN 0.

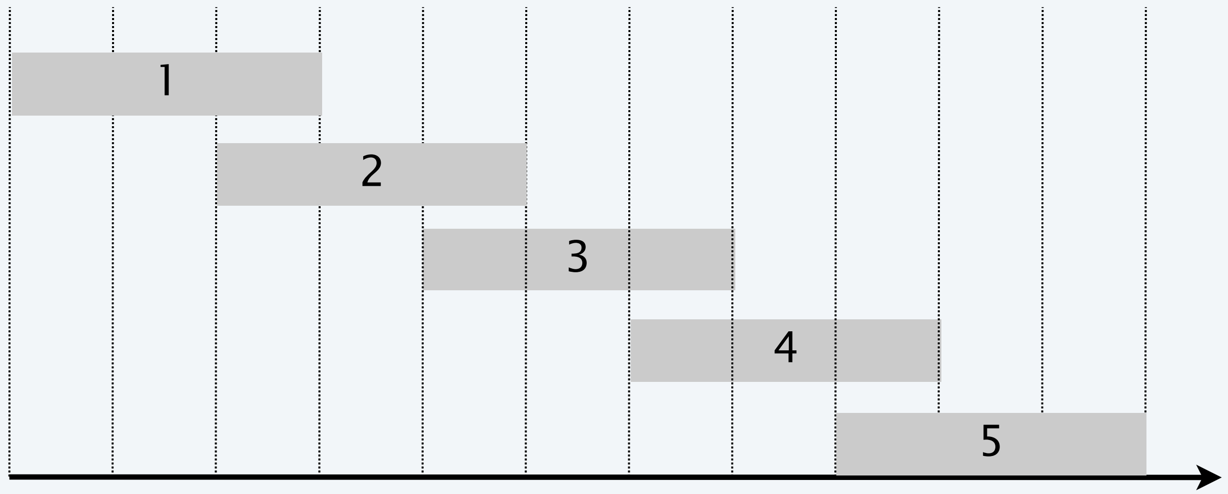
ELSE

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

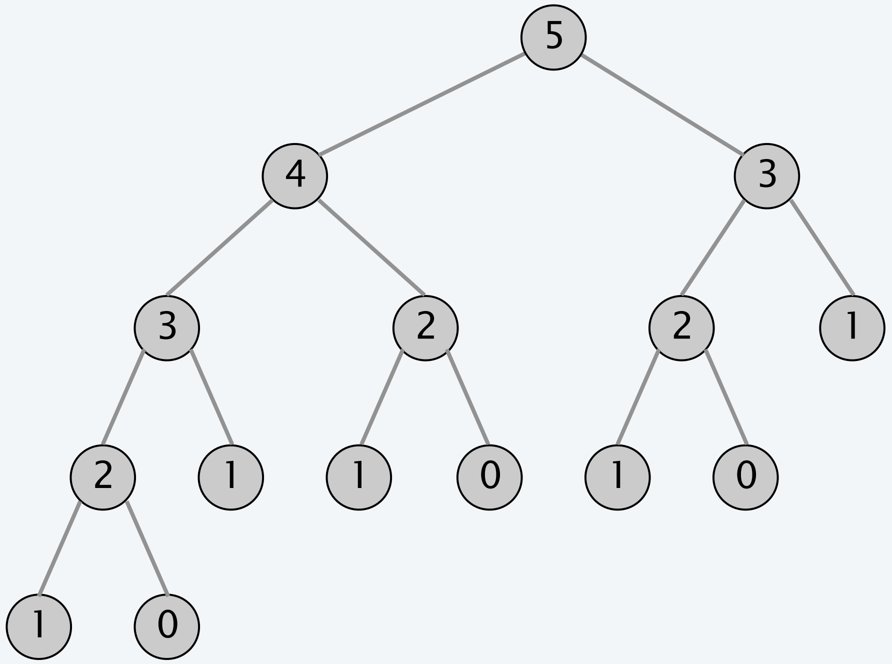
Weighted interval scheduling: brute force

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.



p(1) = 0, p(j) = j−2



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

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.

$M[0] \leftarrow 0.$ ← global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$\quad M[j] \leftarrow \max \{$ 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 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 $\Phi \le n$.
  - (2) increases Φ by $1 \Rightarrow \le 2n$ 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 ∅.
ELSE IF (w_j + M[p[j]] > M[j–1])
   RETURN { j } ∪ 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 $f_1 \le f_2 \le \ldots \le f_n$.

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

$M[0] \leftarrow 0$.

previously computed values

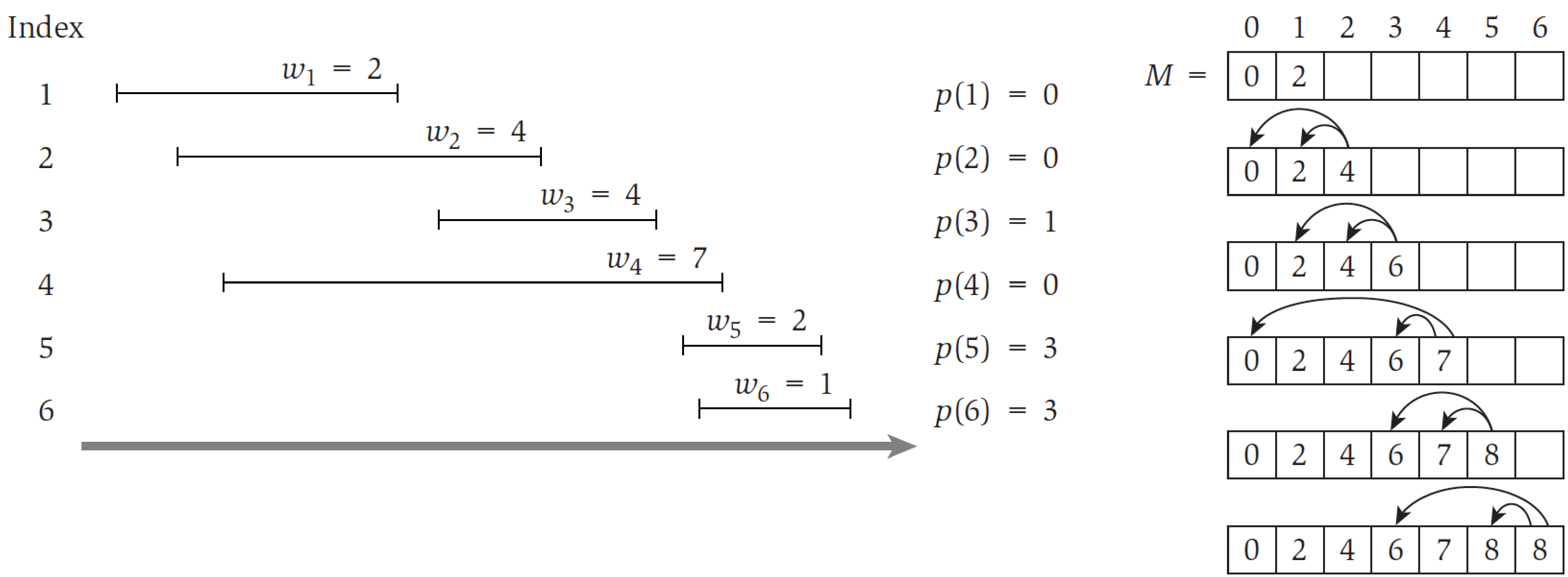
FOR $j = 1$ TO n

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

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

Weighted interval scheduling: bottom-up dynamic programming

Bottom-up dynamic programming. Unwind recursion.



$M[0] \leftarrow 0.$

For $j = 1$ **to** n

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

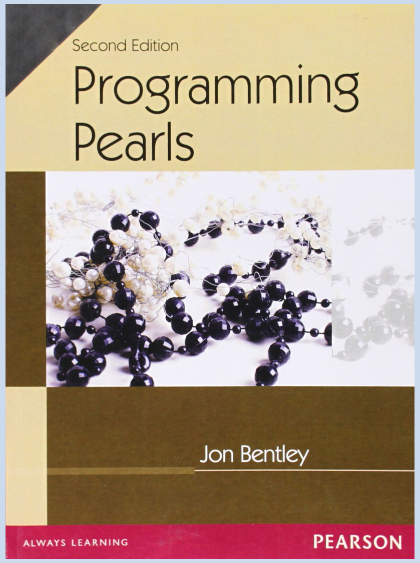
Maximum subarray problem

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, ...



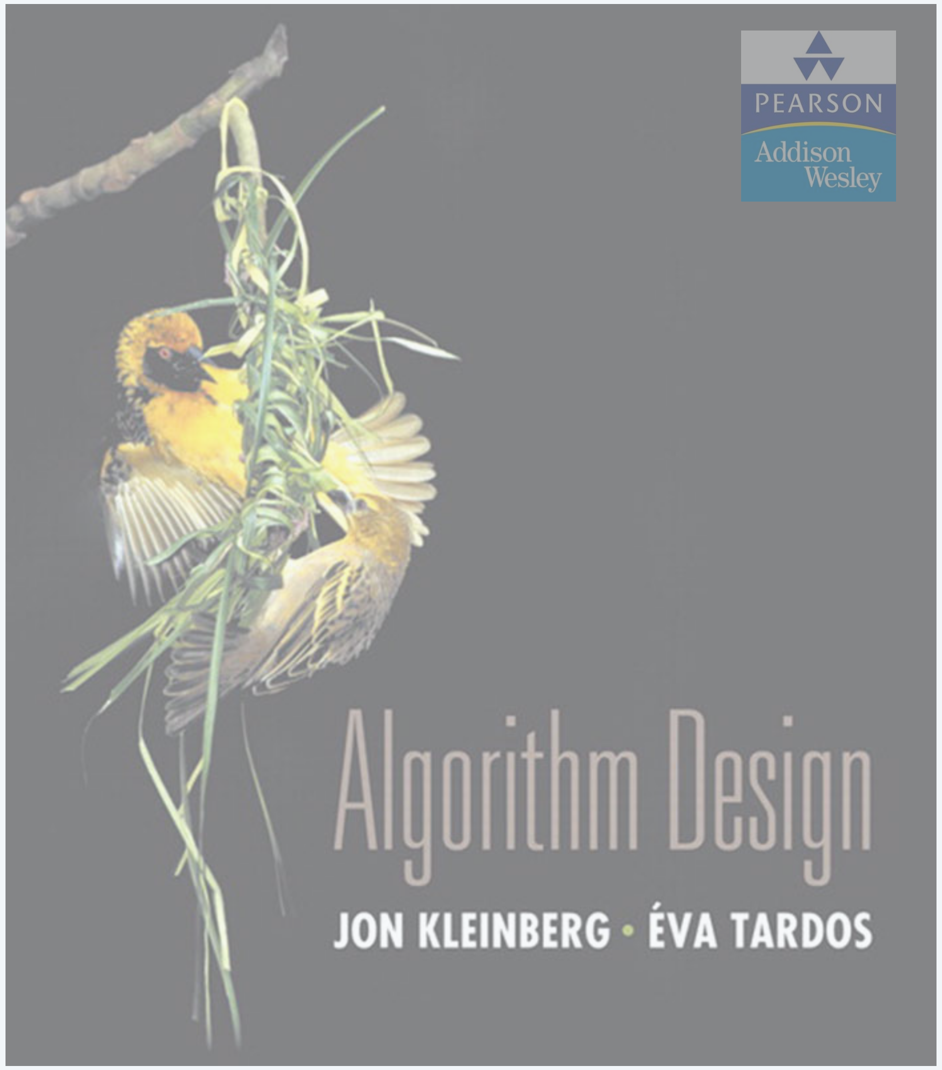
Maximum rectangle problem

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 6.4

6. DYNAMIC PROGRAMMING I

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

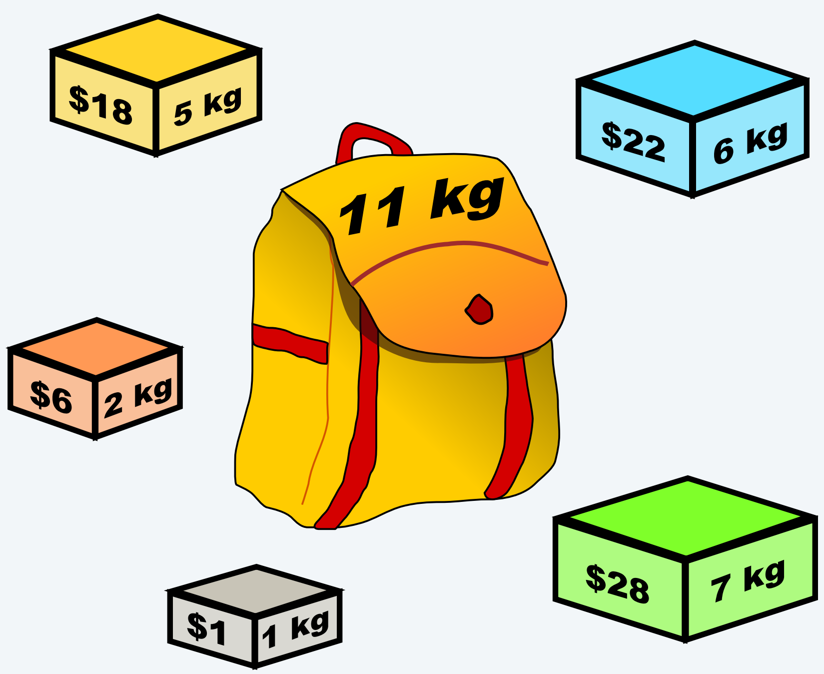
Knapsack problem

Goal. Pack knapsack so as to maximize total value.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Knapsack has weight capacity of W.

Assumption. All input values are integral.

Ex. $\{1, 2, 5\}$ has value \$35 (and weight 10).
Ex. $\{3, 4\}$ has value \$40 (and weight 11).



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: adding a new variable

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

Goal. $OPT(n, W)$.

possibly because $w_i > w$

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

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ using 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 $\{ 1, 2, \ldots, i-1 \}$ using this 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 $M[n, W]$.

---

previously computed values

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

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

subset of items 1, …, i	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) = max-profit subset of items 1, …, i with 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.

weights are integers between 1 and W

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 $M[i, w] > M[i-1, w]$. ▪

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, 100, 350, 1295, 1500 }.
Optimal. 140¢ = 70 + 70.

4. Greedy Algorithms I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Coin changing

Goal. Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount to customer using fewest coins.

Ex. 34¢.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.

Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $0 < c_1 < c_2 < \ldots < c_n$.

$S \leftarrow \varnothing$. ← multiset of coins selected

WHILE $(x > 0)$

  $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$.

  IF no such k, RETURN *"no solution."*

  ELSE

    $x \leftarrow x - c_k$.

    $S \leftarrow S \cup \{ k \}$.

RETURN S.

Is the cashier's algorithm optimal?

- **A.** Yes, greedy algorithms are always optimal.
- **B.** Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.
- **C.** Yes, because of special properties of U.S. coin denominations.
- **D.** No.

Cashier's algorithm (for arbitrary coin denominations)

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Optimal: 140¢ = 70 + 70.

A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.

- Cashier's algorithm: 15¢ = 9 + ?.
- Optimal: 15¢ = 7 + 8.

Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4.
Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.
Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.
Pf.
- Recall: ≤ 1nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.

dollars (100¢) | **quarters** (25¢) | **dimes** (10¢) | **nickels** (5¢) | **pennies** (1¢)

Optimality of cashier's algorithm (for U.S. coin denominations)

Theorem. Cashier's algorithm is optimal for U.S. coins { 1, 5, 10, 25, 100 }.

Pf. [by induction on amount to be paid x]

- Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm. ▪

k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \le 4$	–
2	5	$N \le 1$	4
3	10	$N + D \le 2$	4 + 5 = 9
4	25	$Q \le 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

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 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.