CS101 Algorithms and Data Structures

Dynamic Programming

Textbook Ch 15

Fibonacci numbers

Consider this function:

```
double F( int n ) {
    return ( n <= 1 ) ? 1.0 : F(n - 1) + F(n - 2);
}
```

The run-time of this algorithm is

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & n \le 1 \\ \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1) & n > 1 \end{cases}$$

Fibonacci numbers

Consider this function:

```
double F( int n ) {
    return ( n <= 1 ) ? 1.0 : F(n - 1) + F(n - 2);
}
```

The runtime is similar to the actual definition of Fibonacci numbers:

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & n \le 1 \\ \mathrm{T}(n-1) + \mathrm{T}(n-2) + \Theta(1) & n > 1 \end{cases} \qquad F(n) = \begin{cases} 1 & n \le 1 \\ F(n-1) + F(n-2) + 1 & n > 1 \end{cases}$$

$\mathrm{T}(n) = \mathrm{O}(2^n)$

Fibonacci numbers

Problem:

- To calculate $F(44)$, it is necessary to calculate $F(43)$ and $F(42)$
- However, to calculate $F(43)$, it is also necessary to calculate $F(42)$
- It gets worse, for example
  - $F(40)$ is called 5 times
  - $F(30)$ is called 620 times
  - $F(20)$ is called 75 025 times
  - $F(10)$ is called 9 227 465 times
  - $F(0)$ is called 433 494 437 times

Surely we don't have to recalculate $F(10)$ almost ten million times…

Fibonacci numbers

Here is a possible solution:

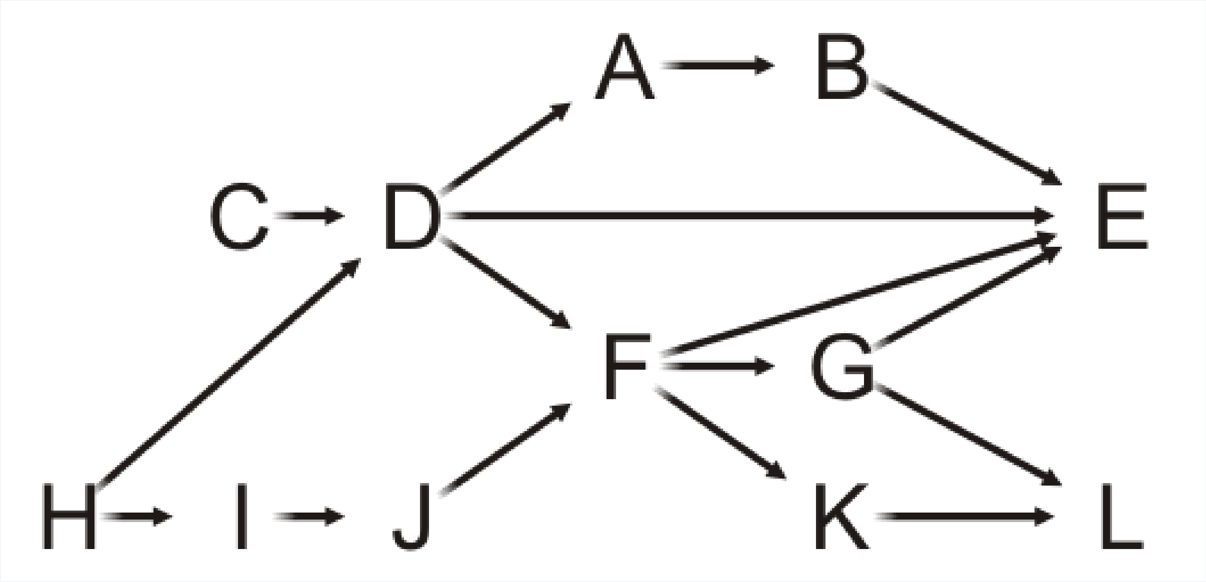
- To avoid calculating values multiple times, store intermediate calculations in a table
- When storing intermediate results, this process is called *memoization*
  - The root is *memo*
- We save (*memoize*) computed answers for possible later reuse, rather than re-computing the answer multiple times

Connected

Determining if two vertices are connected in a DAG, we could implement the following:

```
bool Weighted_graph::connected( int i, int j ) {
    if ( adjacent( i, j ) ) {
        return true;
    }

    for ( int v : neighbors( i ) ) {
        if ( connected( v, j ) ) {
            return true;
        }
    }
    return false;
}
```



- What are the issues with this implementation?

Dynamic programming

In solving optimization problems, the top-down approach may require repeatedly obtaining optimal solutions for the same sub-problem

- Mathematician Richard Bellman initially formulated the concept of dynamic programming in 1953 to solve such problems
- This isn’t new, but Bellman formally defined this process

Dynamic programming

Dynamic programming is distinct from divide-and-conquer, as the divide-and-conquer approach works well if the sub-problems are essentially unique

- Storing intermediate results would only waste memory

If sub-problems re-occur, the problem is said to have *overlapping sub-problems*

Algorithmic paradigms

Greedy. 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

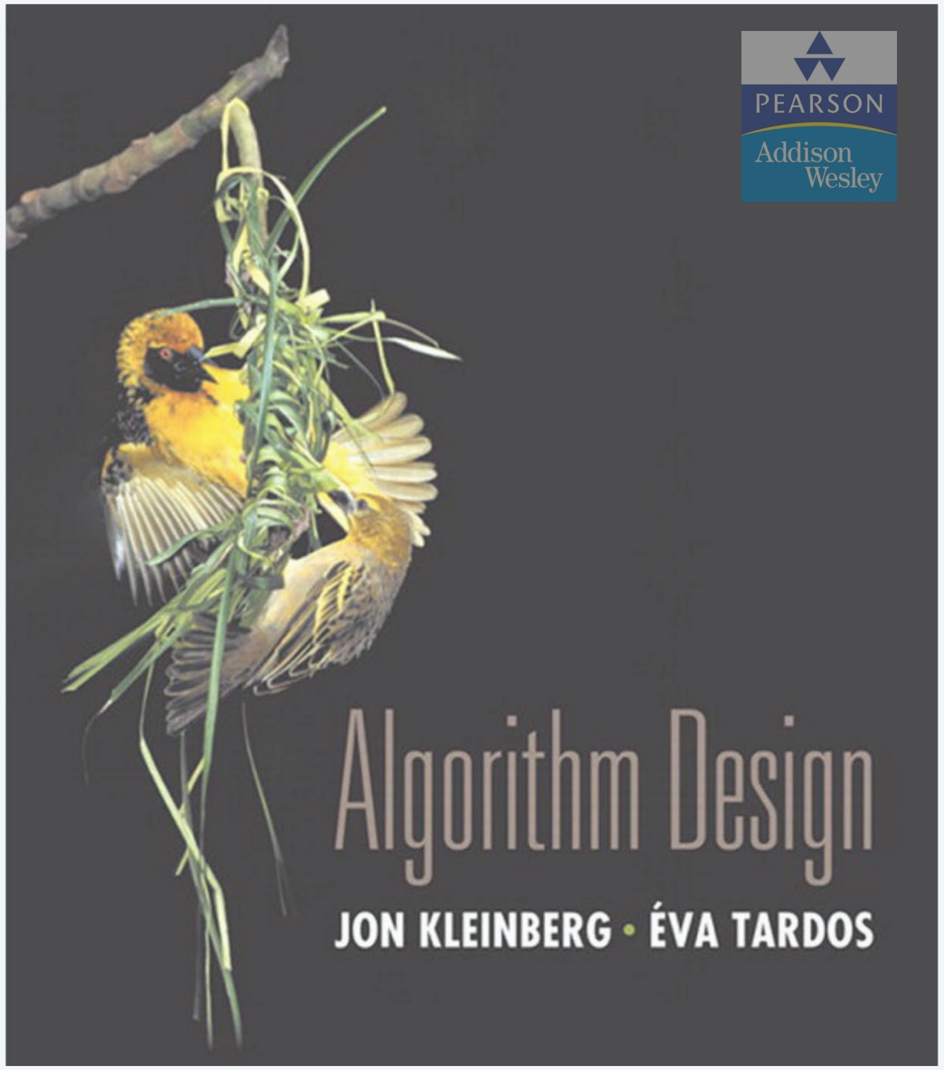
Dynamic programming applications

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.



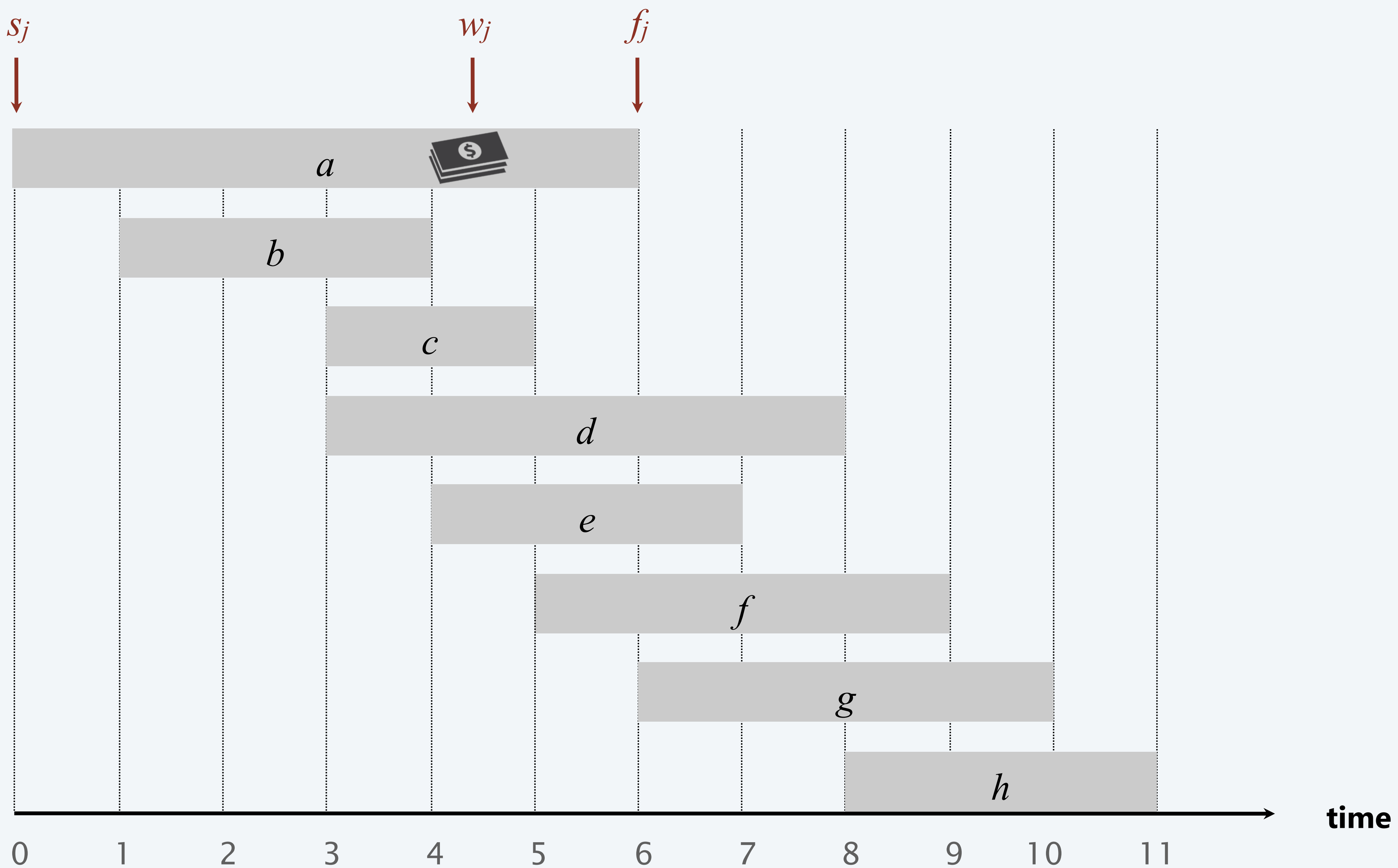
SECTIONS 6.1–6.2

DYNAMIC PROGRAMMING

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

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.



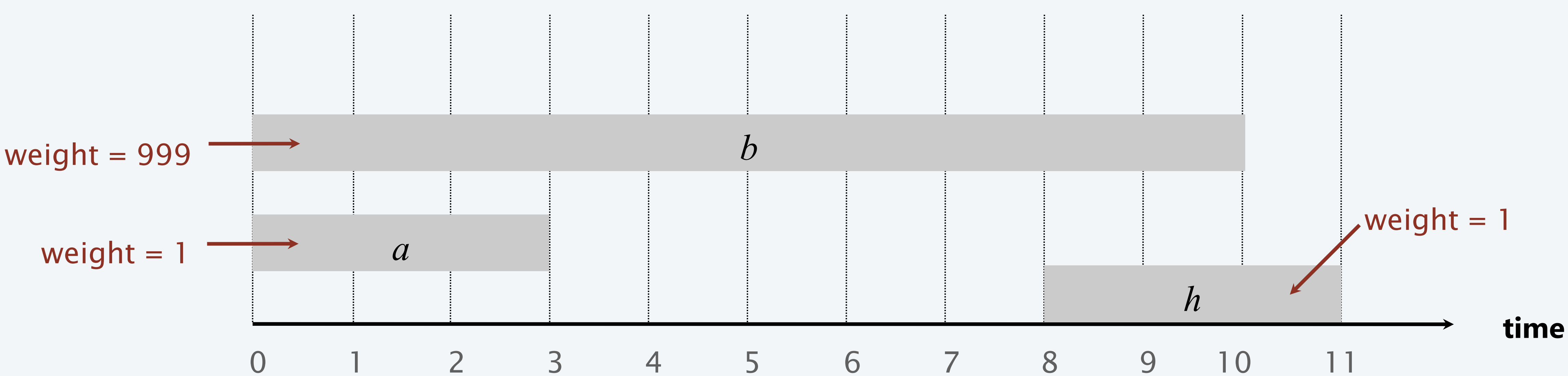
Earliest-finish-time first algorithm

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.



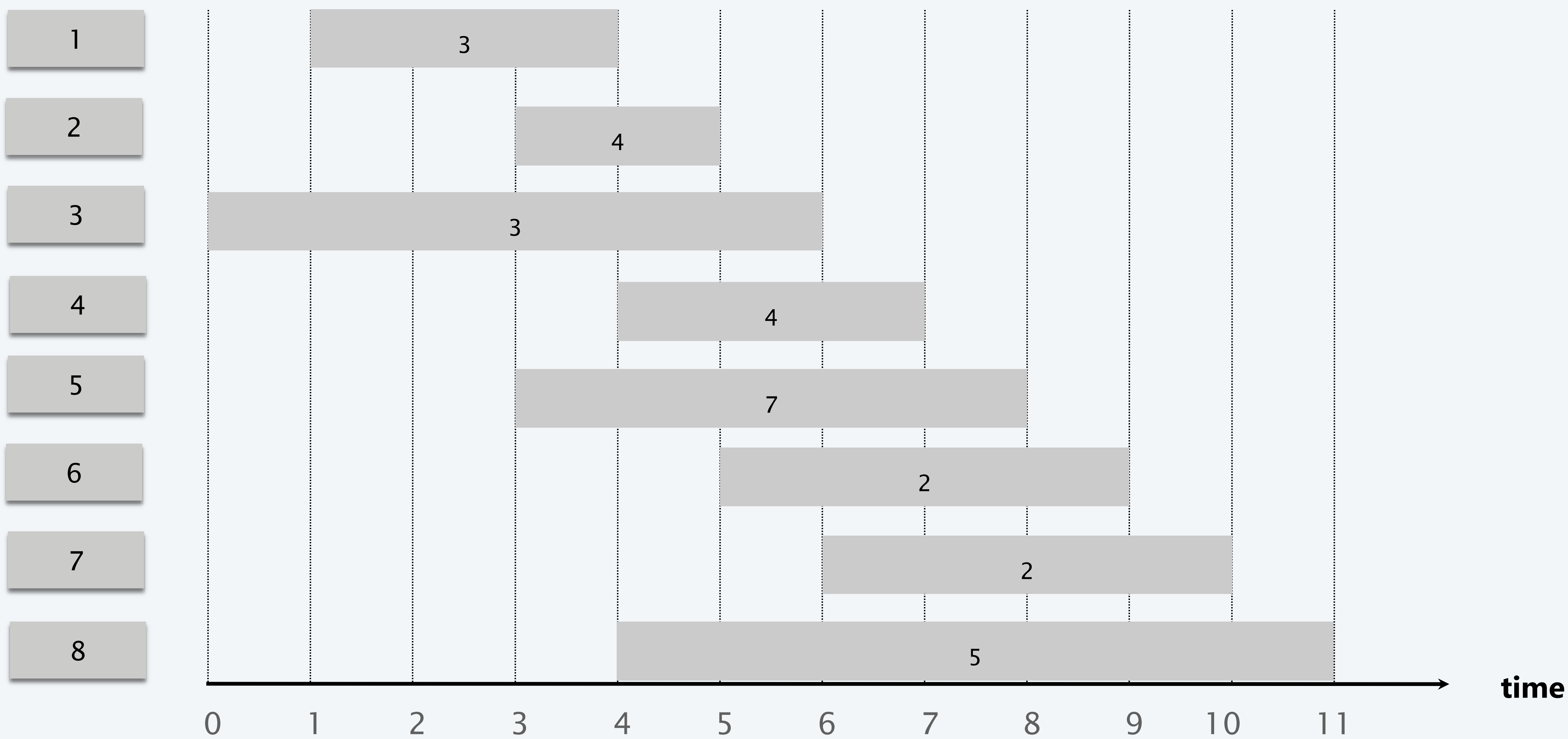
Weighted interval scheduling

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



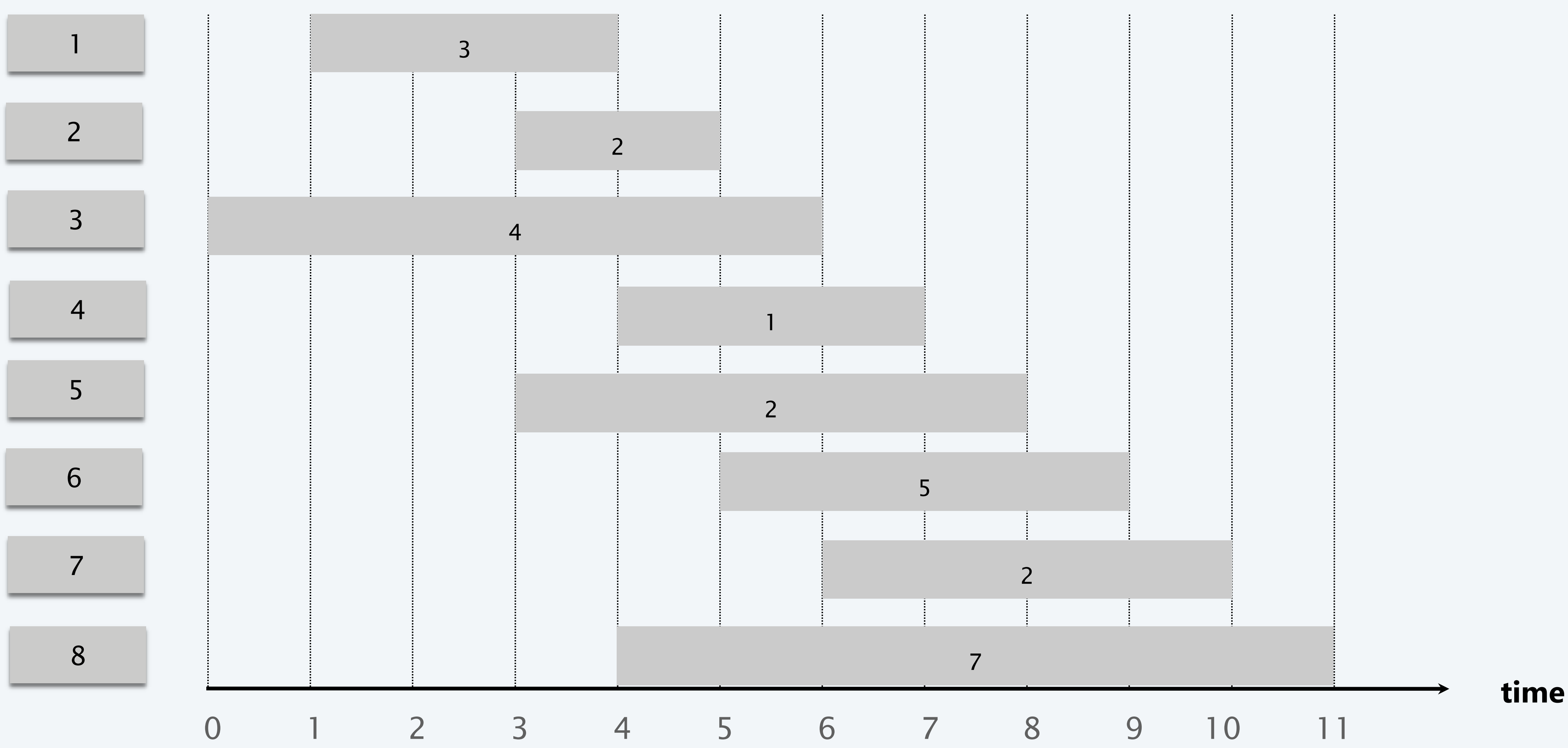
Weighted interval scheduling

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($p[j]$) }.

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.

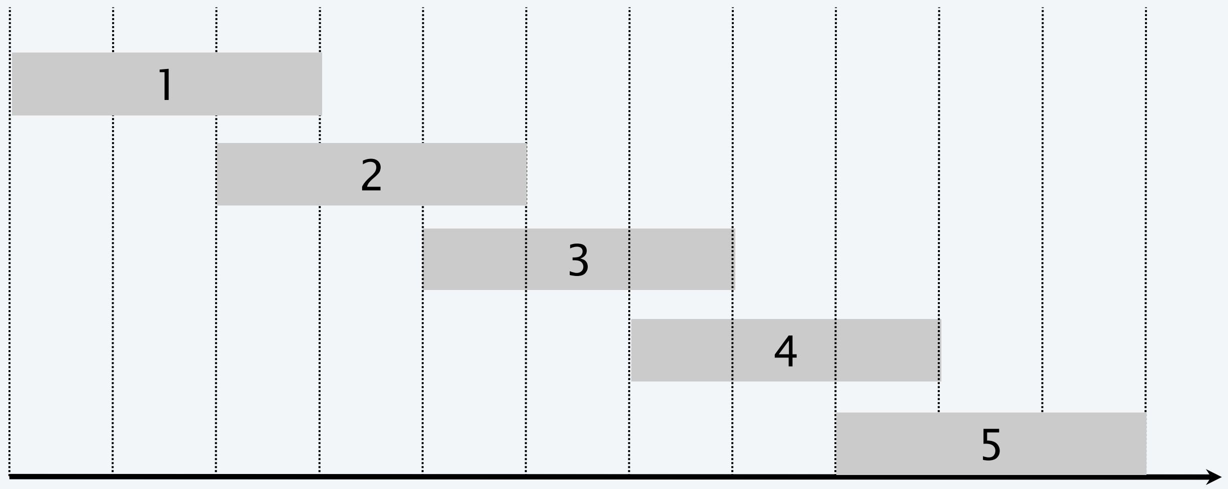
ELSE

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

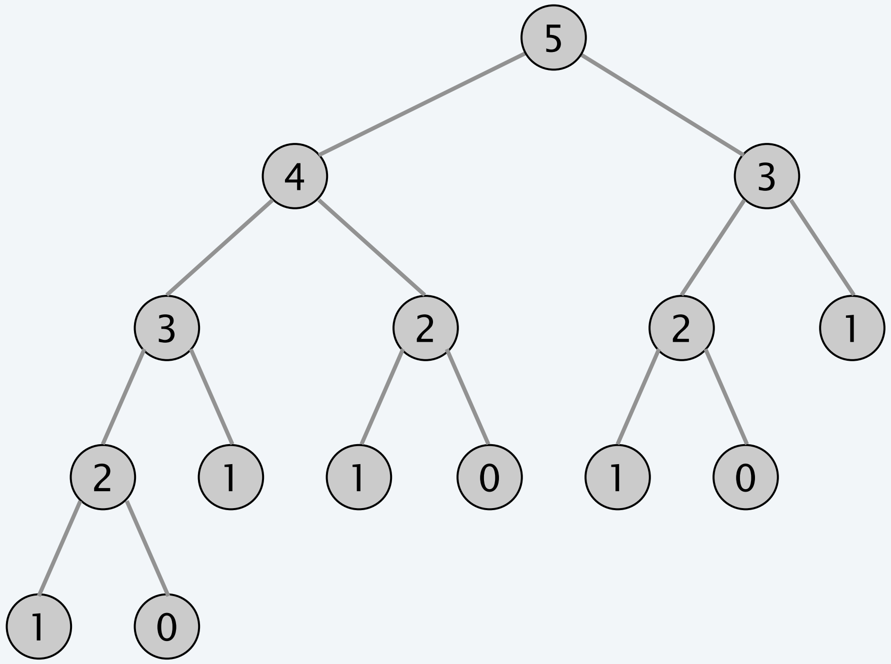
Weighted interval scheduling: brute force

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.



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$)$

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

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 $\Phi \leq n$.
  - 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]] > 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, …, s_n, f_1, …, f_n, w_1, …, 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

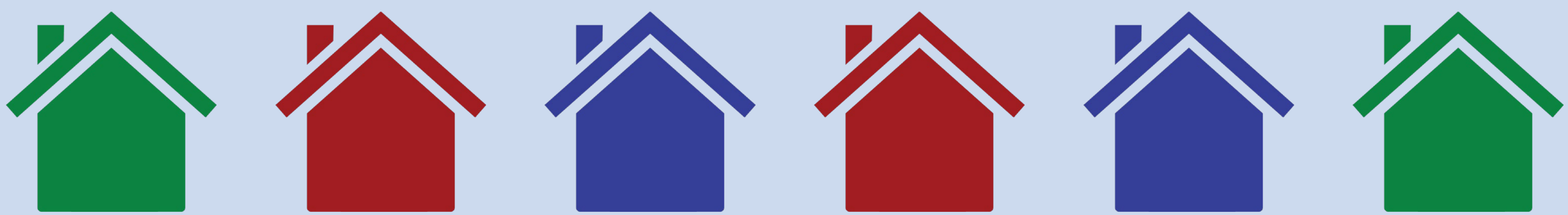
$\quad M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}$.

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

House coloring problem

Goal. Paint a row of n houses red, green, or blue so that

- No two adjacent houses have the same color.
- Minimize total cost, where $cost(i, color)$ is cost to paint i given color.



	A	B	C	D	E	F
	7	6	7	8	9	20
	3	8	9	22	12	8
	16	10	4	2	5	7

cost to paint house i the given color

HOUSE COLORING PROBLEM

Subproblems.

- $R[i]$ = min cost to paint houses $1, \ldots, i$ with i red.
- $G[i]$ = min cost to paint houses $1, \ldots, i$ with i green.
- $B[i]$ = min cost to paint houses $1, \ldots, i$ with i blue.
- Optimal cost = $\min \{ R[n], G[n], B[n] \}$.

Dynamic programming equation.

- $R[i+1] = cost(i{+}1, red) + \min \{ B[i], G[i] \}$
- $G[i+1] = cost(i{+}1, green) + \min \{ R[i], B[i] \}$
- $B[i+1] = cost(i{+}1, blue) + \min \{ R[i], G[i] \}$

overlapping subproblems

Running time. $O(n)$.

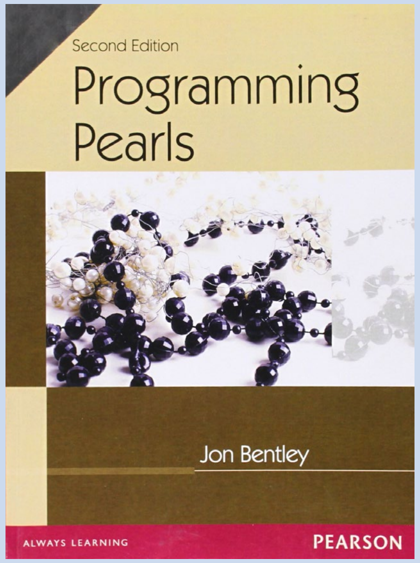
Maximum subarray problem

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

12	5	−1	31	−61	$\boxed{59}$	$\boxed{26}$	$\boxed{-53}$	$\boxed{58}$	$\boxed{97}$	−93	−23	84	−15	6

187

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



Kadane's algorithm

Def. $OPT(i)$ = max sum of any subarray of x whose rightmost index is i.

Goal. $\max_i OPT(i)$

Bellman equation.

$$OPT(i) = \begin{cases} x_1 & \text{if } i = 1 \\ \max\{\, x_i,\ x_i + OPT(i-1) \,\} & \text{if } i > 1 \end{cases}$$

take only element i

take element i together with best subarray ending at index $i - 1$

Running time. $O(n)$.

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, …

Bentley's algorithm

Assumption. Suppose you knew the left and right column indices j and j'.

$$A = \begin{array}{c} \begin{array}{ccccccc} & & j & & j' & & \end{array} \\ \begin{bmatrix} -2 & 5 & 0 & -5 & -2 & 2 & -3 \\ 4 & -3 & -1 & 3 & 2 & 1 & -1 \\ -5 & 6 & 3 & -5 & -1 & -4 & -2 \\ -1 & -1 & 3 & -1 & 4 & 1 & 1 \\ 3 & -3 & 2 & 0 & 3 & -3 & -2 \\ -2 & 1 & -2 & 1 & 1 & 3 & -1 \\ 2 & -4 & 0 & 1 & 0 & -3 & -1 \end{bmatrix} \end{array} \qquad x = \begin{bmatrix} -7 \\ 4 \\ -3 \\ 6 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$

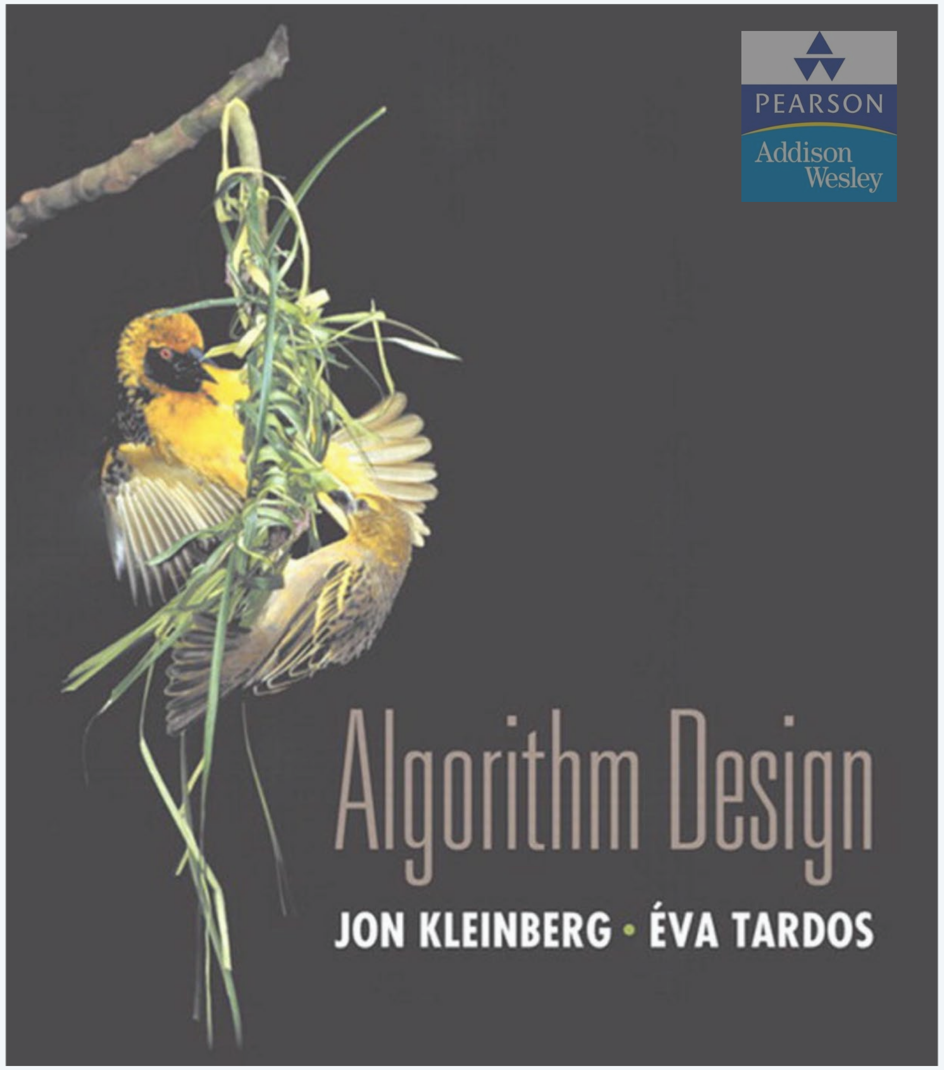
$0 - 5 - 2$

solve maximum subarray problem in this array

An $O(n^3)$ algorithm.

- Precompute cumulative row sums $S_{ij} = \sum_{k=1}^{j} A_{ik}$.
- For each $j < j'$:
  - define array x using row-sum differences: $x_i = S_{ij'} - S_{ij}$
  - run Kadane's algorithm in array x

Open problem. $O(n^{3-\varepsilon})$ for any constant $\varepsilon > 0$.



SECTION 6.3

DYNAMIC PROGRAMMING

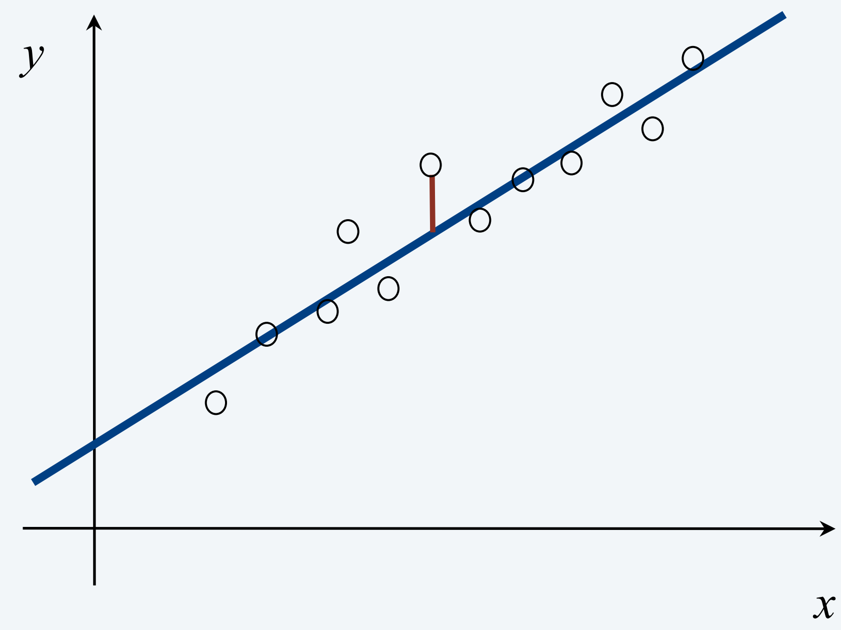
- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*

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

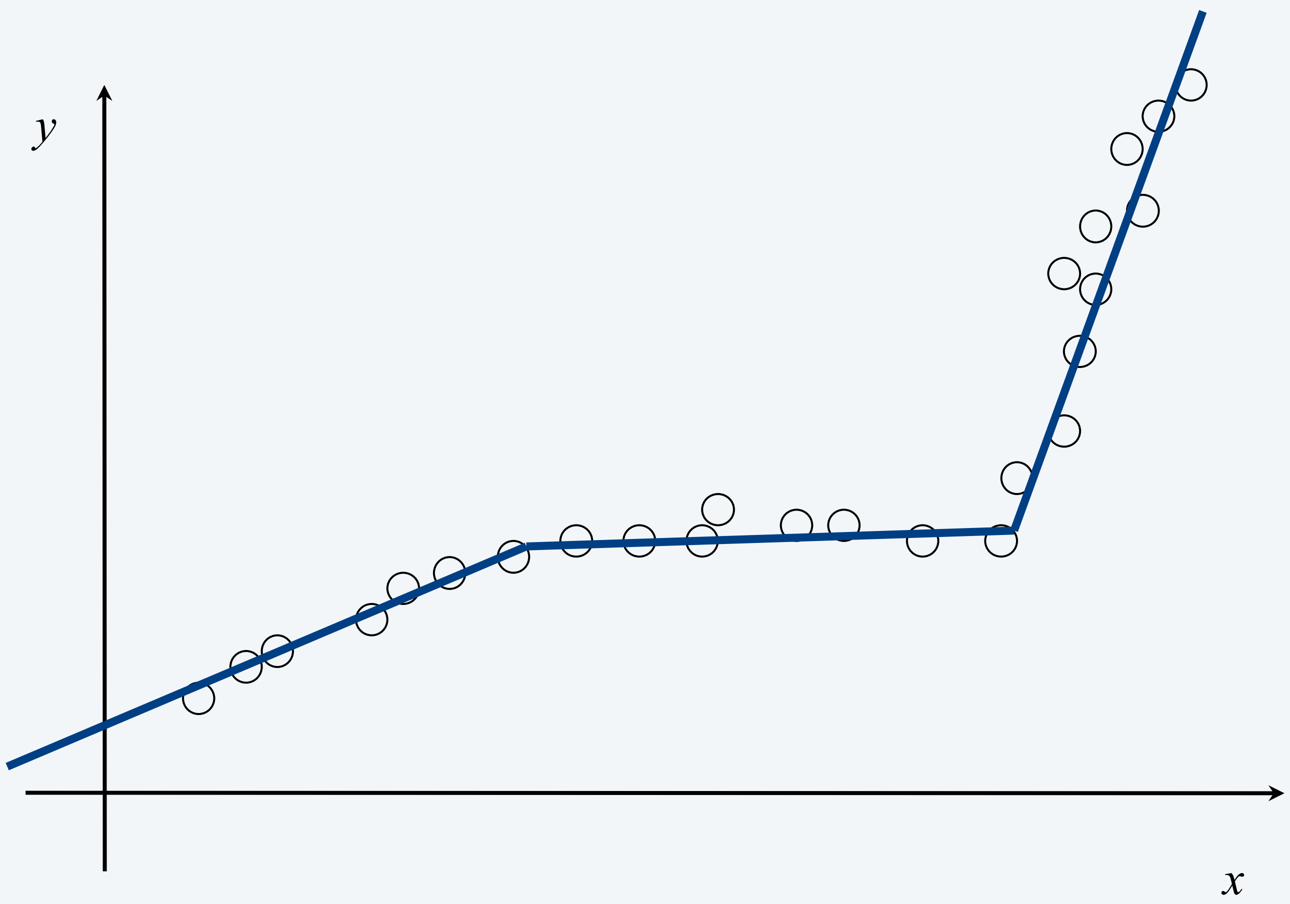
Segmented least squares.

- 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



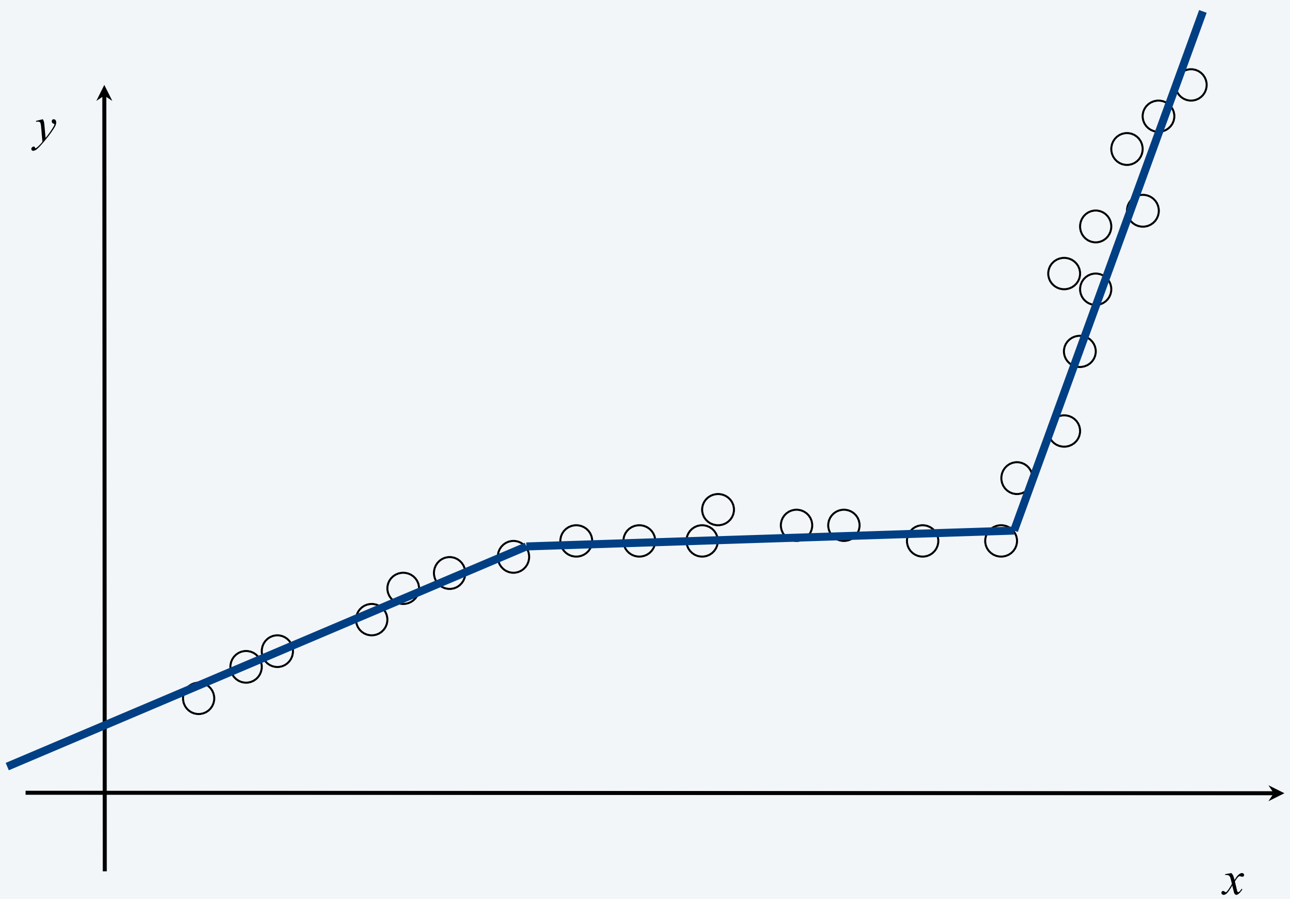
Segmented least squares

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 squared errors in each segment.
- 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 points $p_i, p_{i+1}, \ldots, p_j$ by a single segment .

To compute $OPT(j)$:

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

optimal substructure property
(proof via exchange argument)

Bellman equation.

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

Segmented least squares algorithm

SEGMENTED-LEAST-SQUARES(n, p_1, …, 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}, …, p_j.

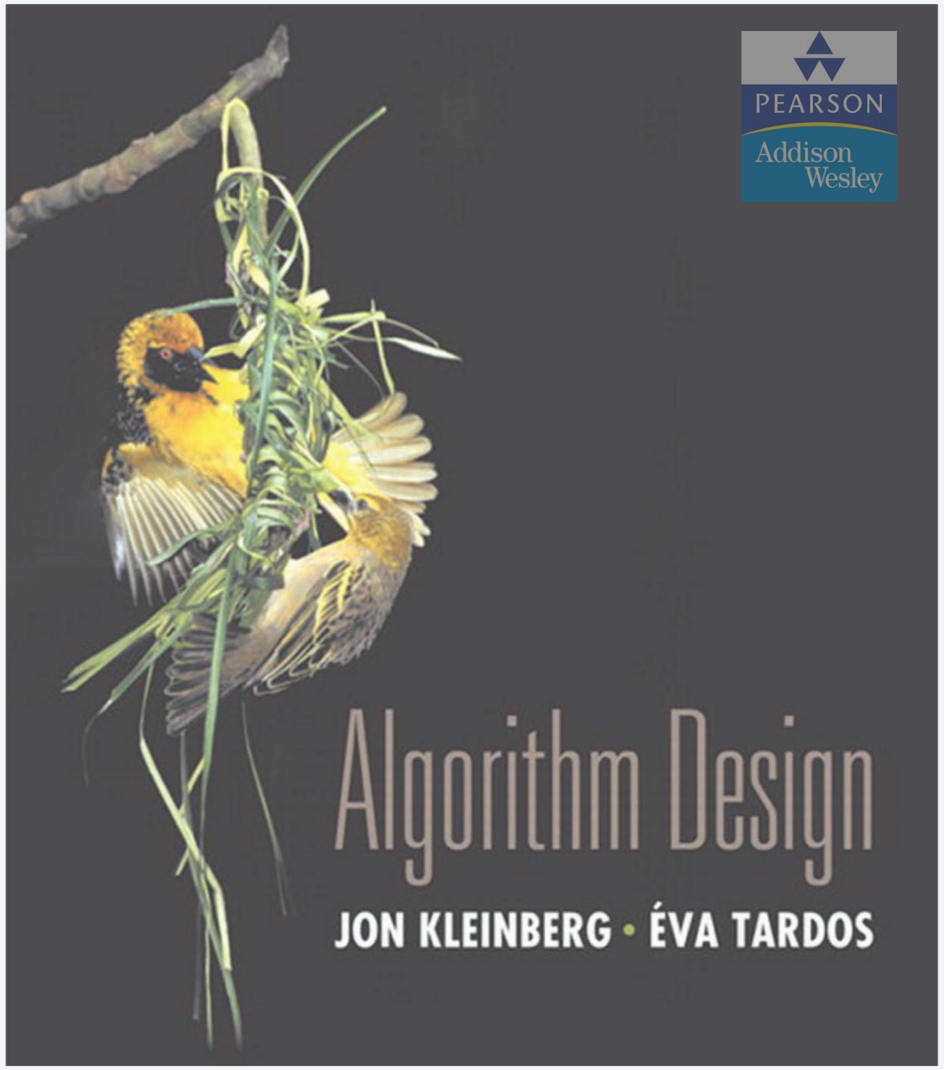
$M[0] \leftarrow 0$.

FOR $j = 1$ TO n

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

previously computed value

RETURN $M[n]$.



SECTION 6.4

DYNAMIC PROGRAMMING

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

Knapsack problem

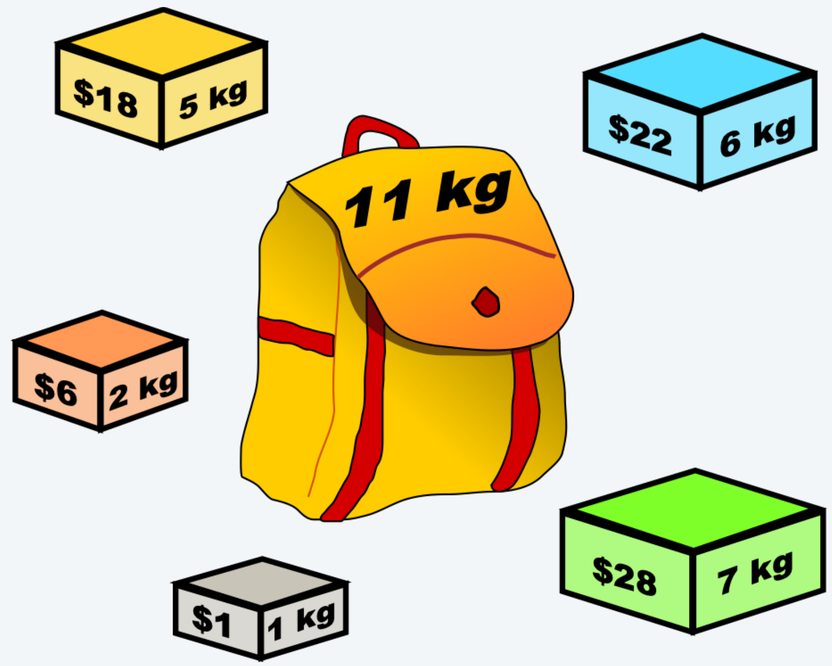
Goal. Pack knapsack so as to maximize total value.

- 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	US\$1	1 kg
2	US\$6	2 kg
3	US\$18	5 kg
4	US\$22	6 kg
5	US\$28	7 kg

knapsack instance
(weight limit W = 11)

Dynamic programming: quiz 2

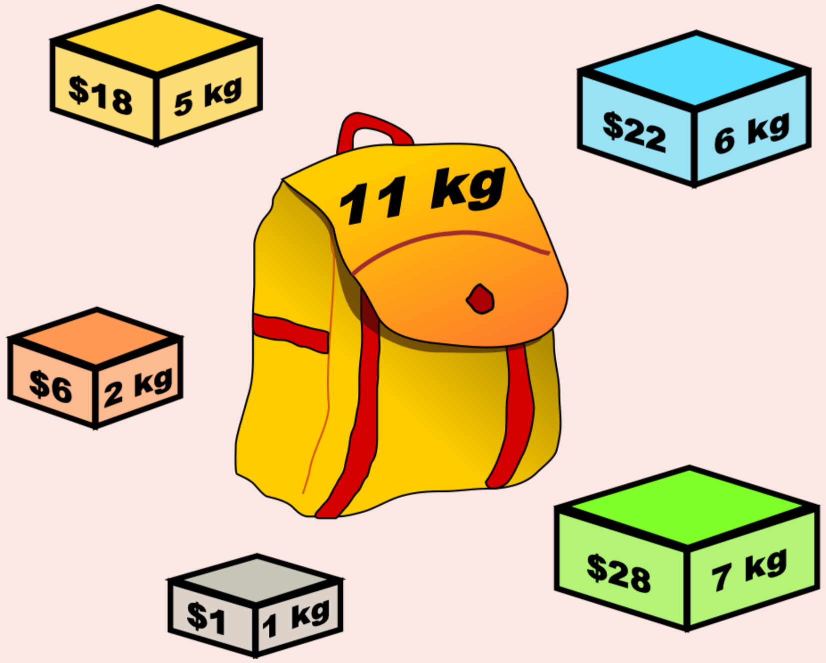
Which algorithm solves knapsack problem?

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. Dynamic programming.



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

knapsack instance
(weight limit W = 11)

Which subproblems?

- *A.* $OPT(w)$ = max-profit with weight limit w.
- *B.* $OPT(i)$ = max-profit subset of items $1, \ldots, i$.
- *C.* $OPT(i, w)$ = max-profit subset of items $1, \ldots, i$ with weight limit w.
- D. Any of the above.

Dynamic programming: false start

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

Goal. $OPT(n)$.

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

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

optimal substructure property
(proof via exchange argument)

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

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing which 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 $1, \ldots, i$ 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.

- 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.

optimal substructure property
(proof via exchange argument)

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	US$1	1 kg
2	US$6	2 kg
3	US$18	5 kg
4	US$22	6 kg
5	US$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 $M[i, w] > M[i-1, w]$. ▪

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.

Coin changing

Def. $OPT(v)$ = min number of coins to make change for v.

Goal. $OPT(V)$.

Multiway choice. To compute $OPT(v)$,

- Select a coin of denomination c_i for some i.
- Select fewest coins to make change for $v - c_i$.

optimal substructure property (proof via exchange argument)

Bellman equation.

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{ 1 + OPT(v - c_i) \} & \text{otherwise} \end{cases}$$

Running time. $O(n\,V)$.

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.

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