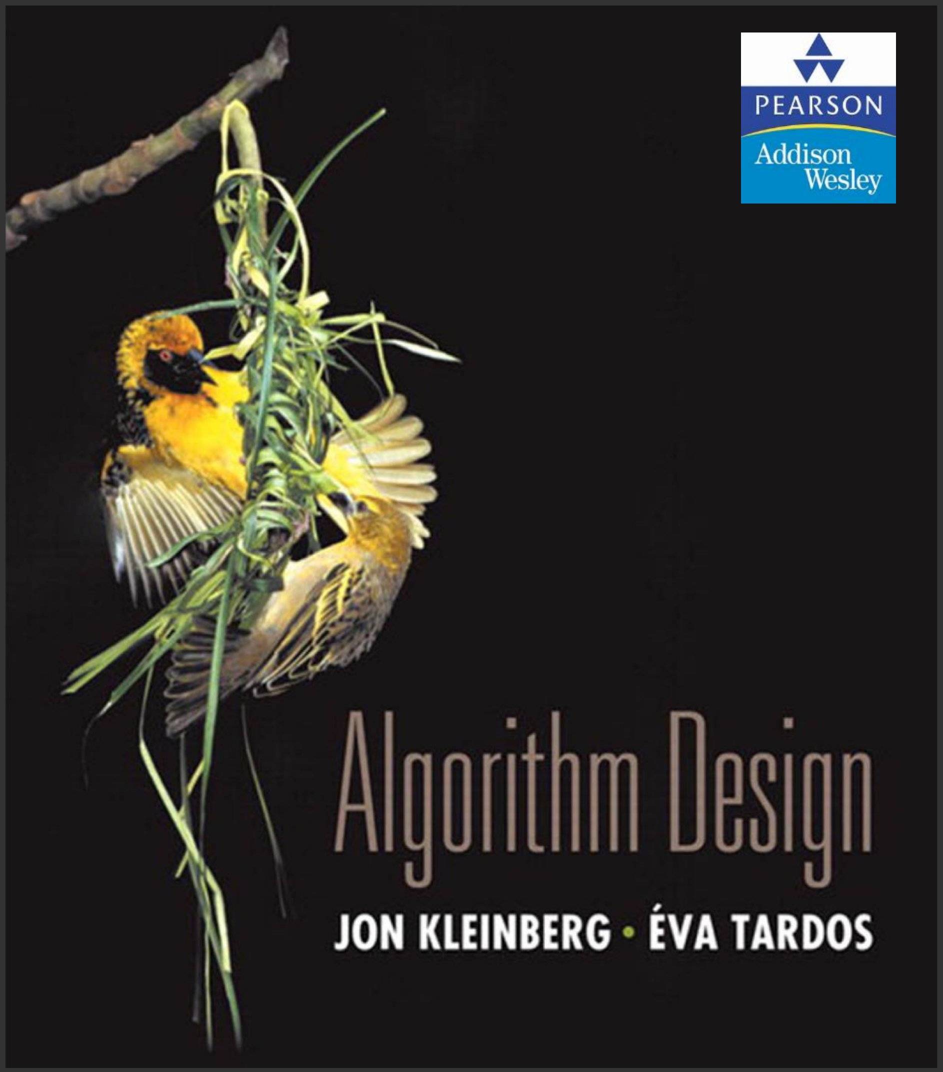


6. Dynamic Programming I

- *weighted interval scheduling (& memoization)*
- *Longest Increasing Subsequence*
- *House Coloring problem (exercise)*

Lecture slides by Kevin Wayne

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

Algorithmic paradigms

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

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

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

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

Dynamic programming history

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

Etymology.

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

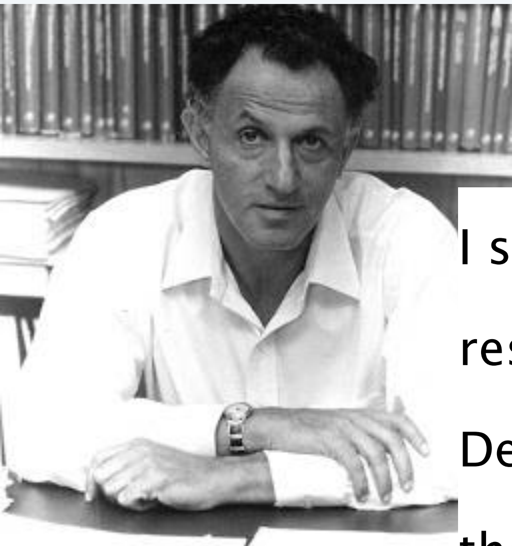
THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

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

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

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



I spent the Fall quarter (of 1950) at RAND. [..] The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. [...] The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. [...] I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

Eye of the Hurricane: An Autobiography

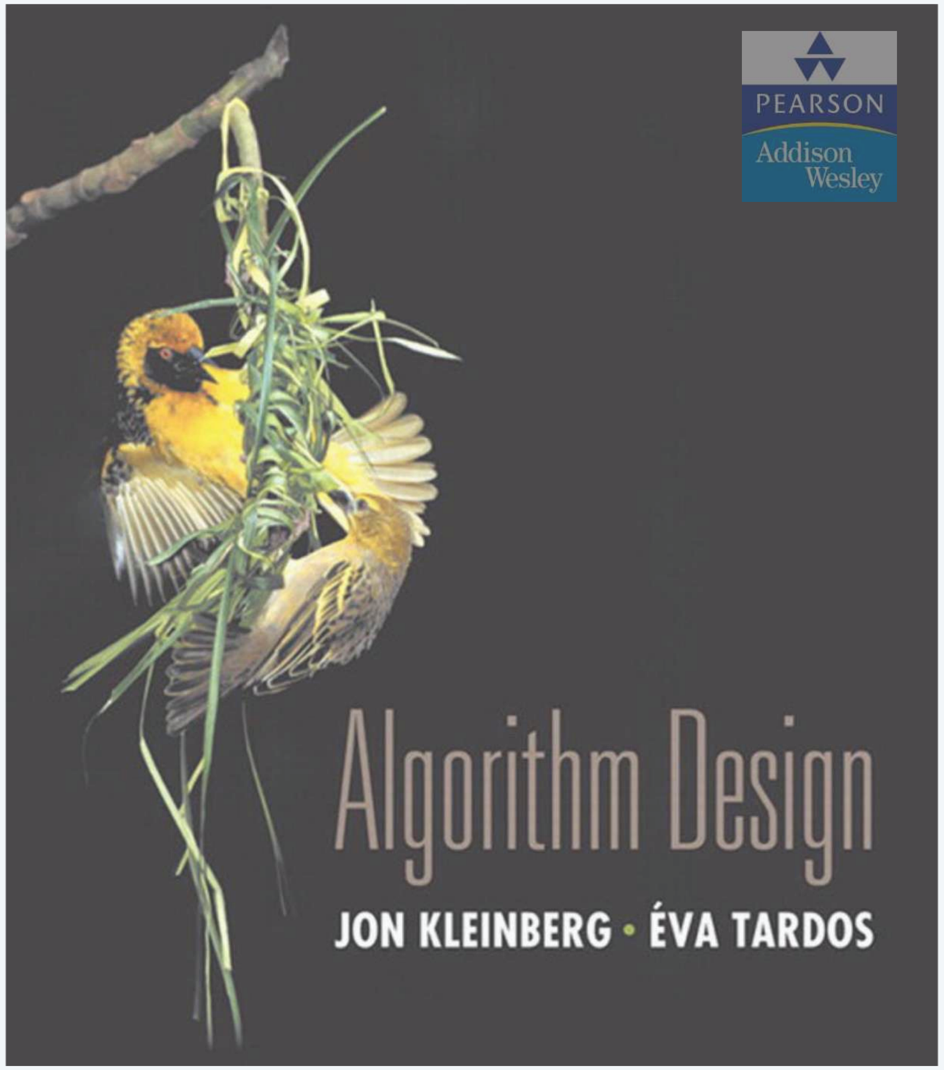
Dynamic programming applications

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.



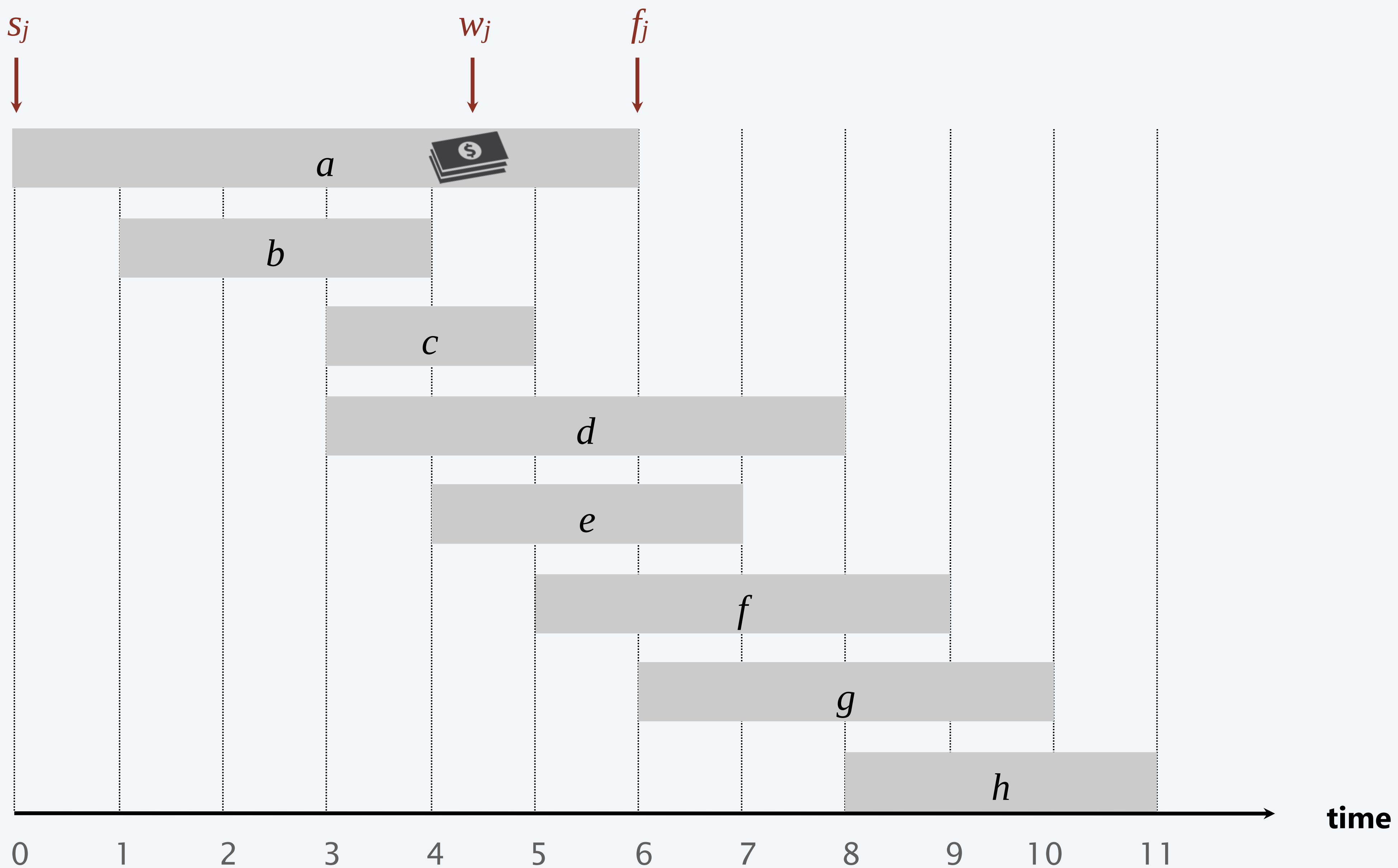
SECTIONS 6.1–6.2

6. DYNAMIC PROGRAMMING I

- *weighted interval scheduling (& memoization)*

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.



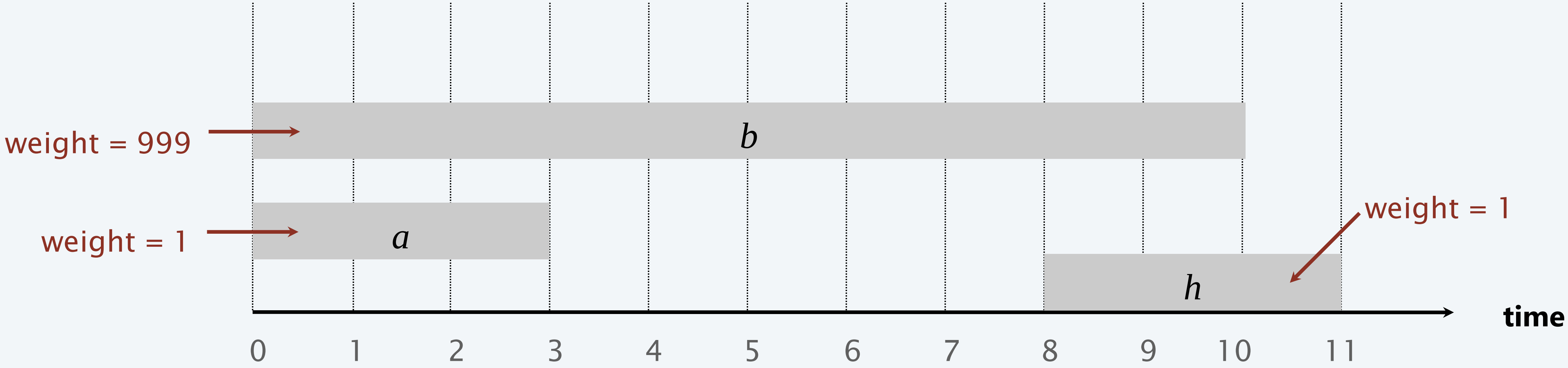
Earliest-finish-time first algorithm

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.



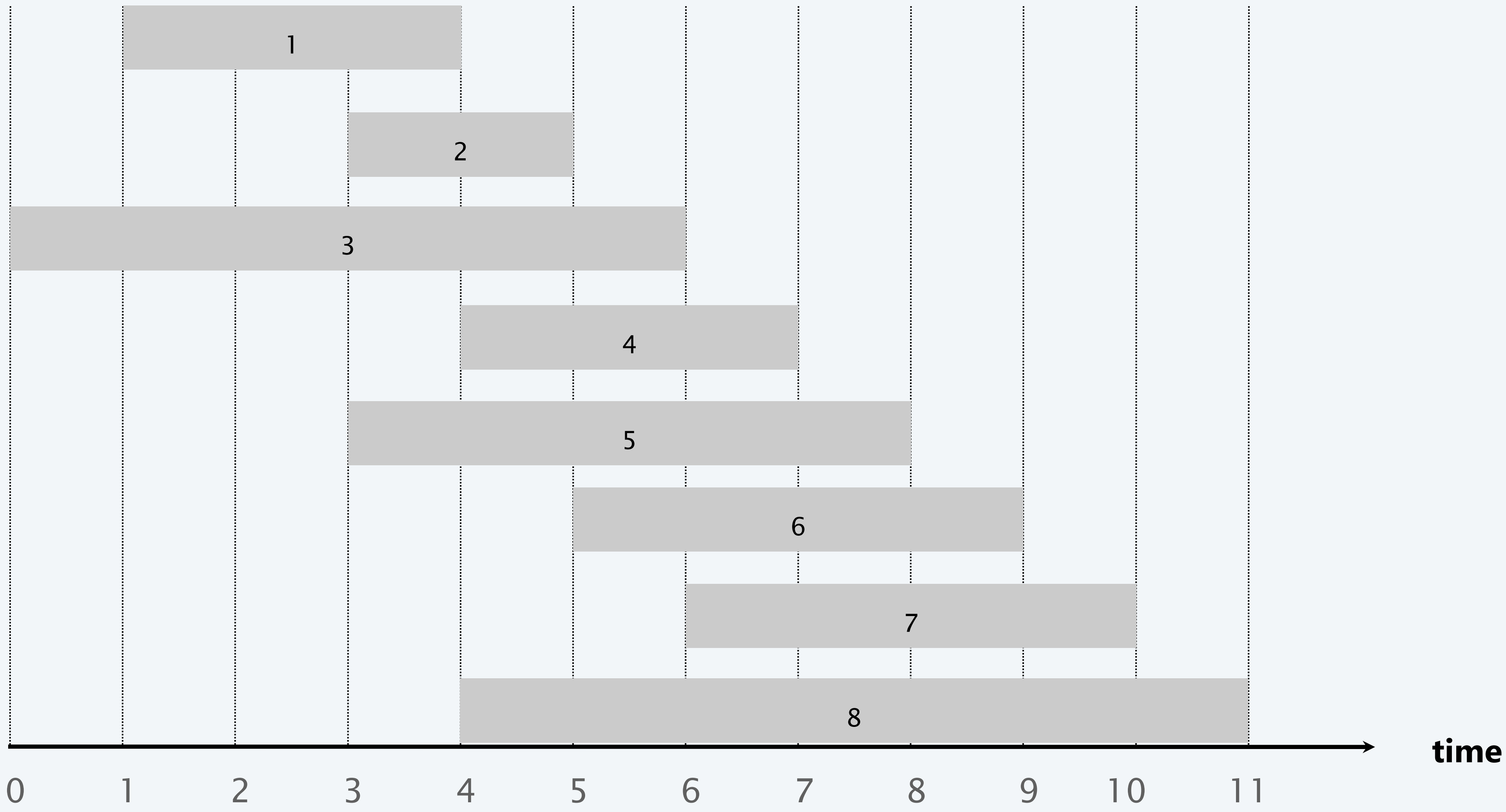
Weighted interval scheduling

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 rightmost 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: bottom-up dynamic programming

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

$M[0] \leftarrow 0$.

previously computed values

FOR $j = 1$ TO n

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

RETURN $M[n]$.

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

- Sort by finish time: $O(n \log n)$ via mergesort.
- Compute $p[j]$ for each j : $O(n \log n)$ via binary search.
- FOR cycle takes $O(n)$ time

Weighted interval scheduling: turning back to recursion

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-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

  RETURN 0.

ELSE

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

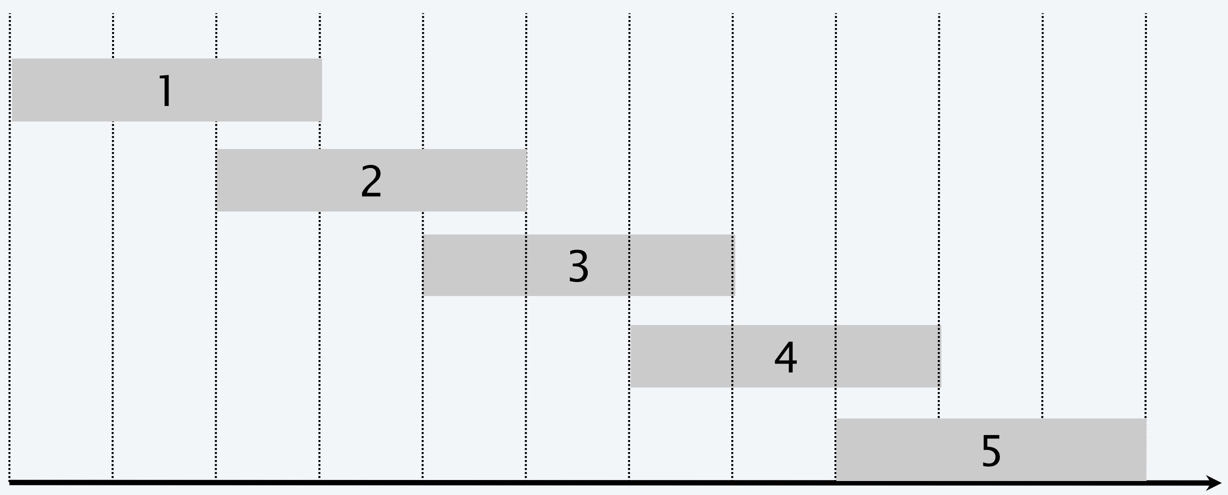
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n-1) + \Theta(1) & \text{if } n > 1 \end{cases}$$

$T(n)=\Theta(2^n)$

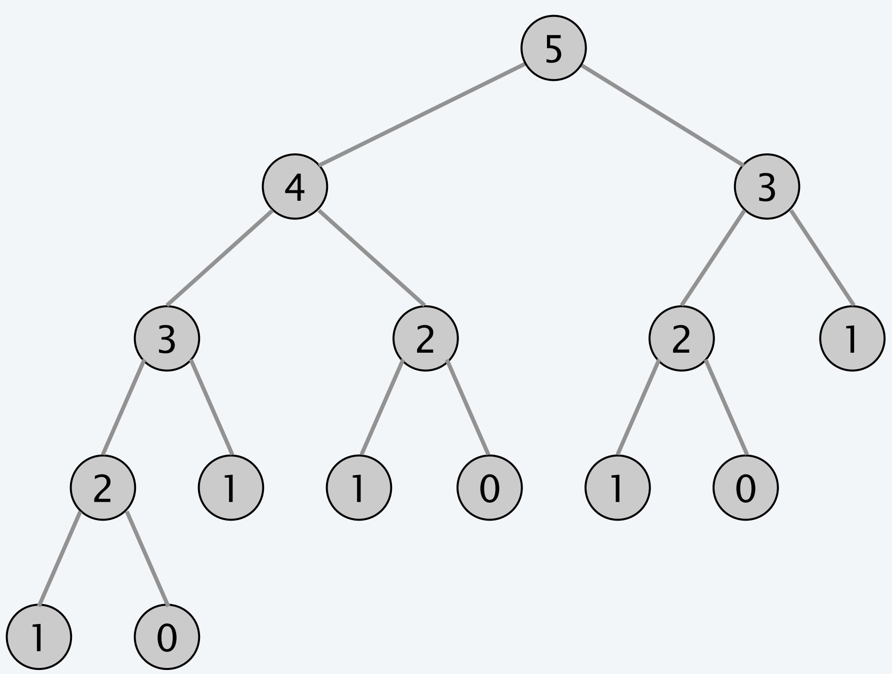
Weighted interval scheduling: turning back to recursion

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

$$T(n)=T(n-1)+T(n-2)+\Theta(1)$$

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

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)

$M[j] \leftarrow \max \{$ 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 $\Phi \leq n$.
  - (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?

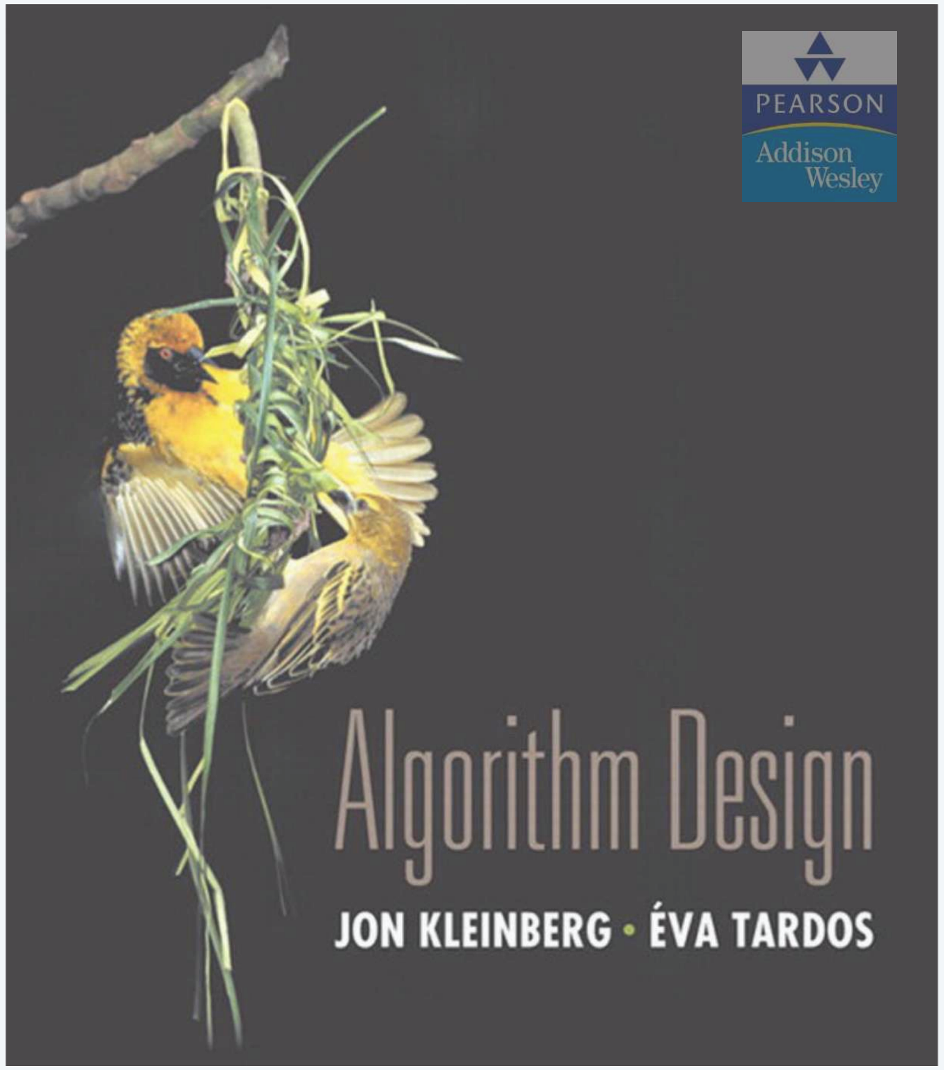
A. Make a second pass by calling FIND-SOLUTION(n).

```
FIND-SOLUTION(j)
_______________________________________________
IF (j = 0)
    RETURN ∅.
ELSE IF (w_j + M[p[j]] > 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 $\leq n \Rightarrow O(n)$.

Memoization (top-down)	VS	Table-based (bottom-up)
• Top-Down approach (more intuitive)		• Harder to grasp
• Easier to index subproblems by other objects (e.g., sets).		• Need to index subproblems with integers
• Only computes necessary subproblems		• Always computes all subproblems
• Function calls overhead		• No recursion. More cache efficient.
• Time complexity is harder to analyze		• Time complexity is easy to analyze
		• Short and clean code



6. DYNAMIC PROGRAMMING I

- *Longest Increasing Subsequence*

Drink as much as possible

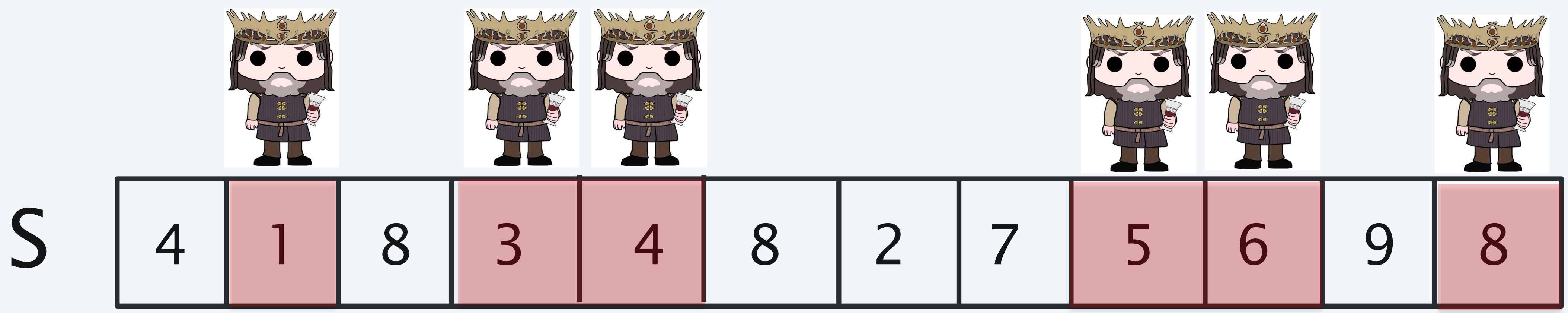
Robert wants to drink as much as possible

- Robert walks through the streets of King's Landing and encounters n taverns $t_1, t_2, \ldots, t_n$, in order
- When Robert encounters a tavern t_i, he can either stop for a drink or continue walking
- The wine served in tavern t_i has strength s_i (the higher, the stronger)
- The strength of Robert's drinks must increase over time
- Goal: Compute the maximum number of drinking stops of Robert

An example

S	4	1	8	3	4	8	2	7	5	6	9	8

An example



optimal solution: 6

This is a problem known as

Longest Increasing Subsequence

A DP algorithm: first attempt

- Subproblem definition:

  OPT[i]: length of the LIS of S[1],...,S[i]

- Base case:

  OPT[1]= 1

- Solution:

  OPT[n]

- Recursion formula:

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] <u>that ends with S[i]</u>

S	4	1	8	3	4	8	2	7	5	6	9	8

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],…,S[i] <u>that ends with S[i]</u>

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4				

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] that ends with S[i]

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4				

Possible lengths: 3

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] <u>that ends with S[i]</u>

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4				

Possible lengths: 3

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] that ends with S[i]

	1	2	3	4	5	6	7	8	9	10	11	12
S	4	1	8	3	4	8	2	7	5	6	9	8
OPT	1	1	2	2	3	4	2	4				

Possible lengths: 3 4

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] that ends with S[i]

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4				

Possible lengths: 3 4 3

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] that ends with S[i]

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4				

Possible lengths: 3 4 3 2 2

A DP algorithm: second attempt

Tip: sometimes adding constraints to subproblems can help!

OPT[i]: length of the LIS of S[1],...,S[i] <u>that ends with S[i]</u>

S	4	1	8	3	4	8	2	7	5	6	9	8
	1	2	3	4	5	6	7	8	9	10	11	12
OPT	1	1	2	2	3	4	2	4	4			

Possible lengths: 3 4 3 2 2

OPT[9]=4

A DP algorithm: second attempt

- Subproblem definition:

  OPT[i]: length of the LIS of S[1],...,S[i] <u>that ends with S[i]</u>

- Base case:

  OPT[1]= 1

- Solution:

  $\max_{i=1,2,..n} \text{OPT}[i]$

- subproblem order:

  OPT[1], OPT[2],..., OPT[n]

- Recursion formula:

$$\text{OPT}[i] = 1 + \max\left\{0, \max_{\substack{j=1,2,\ldots,i-1 \\ \text{st } S[j]<S[i]}} \text{OPT}[j]\right\}$$

Longest Increasing Subsequence

LIS(S[1:n])

OPT[1]=1

FOR i = 2 TO n

$$\text{OPT}[i] = 1 + \max\left\{0, \max_{\substack{j=1,2,\ldots,i-1 \\ \text{st } S[j]<S[i]}} \text{OPT}[j]\right\}$$

RETURN max_i OPT[i].

Running time.

- each OPT[i] is computed in $O(i)=O(n)$ time.
- $O(n^2)$ 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

Subprojects.

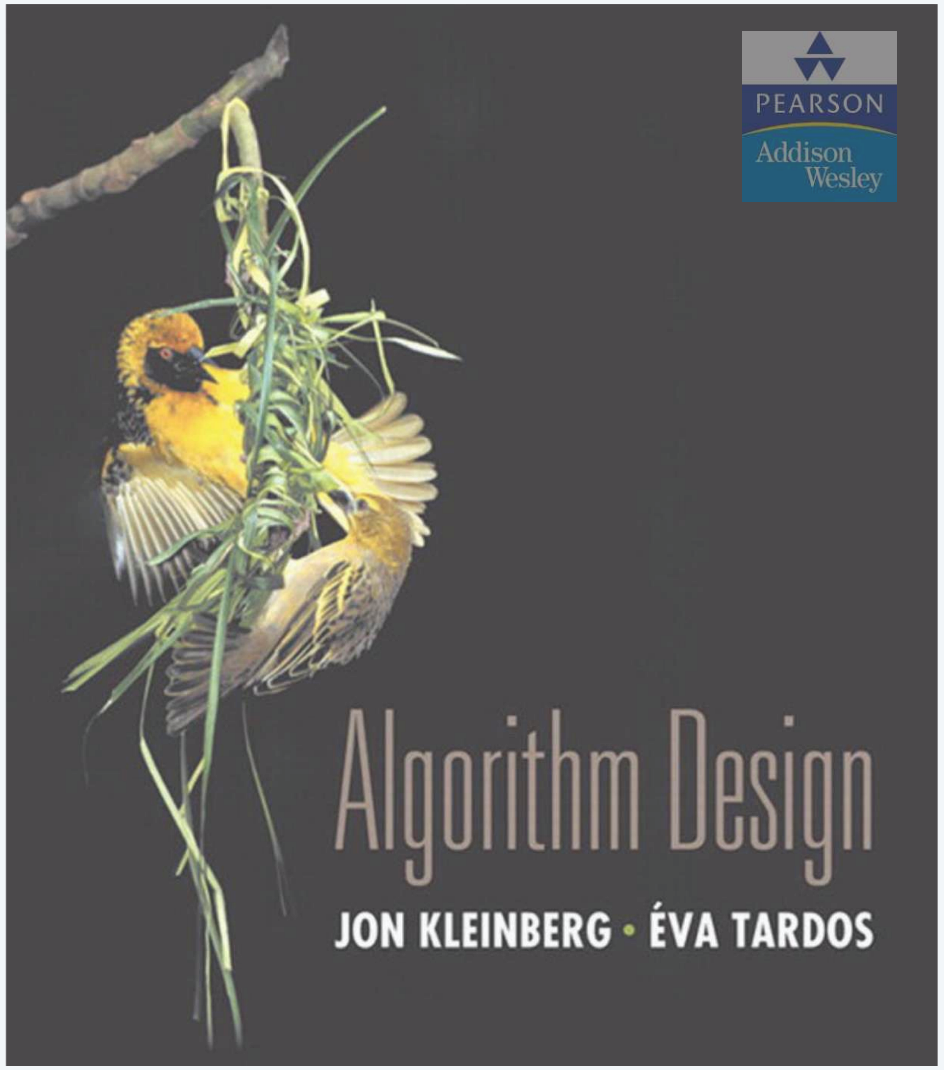
- $R[i]$ = min cost to paint houses $1, \ldots, i$ with i red.
- $G[i]$ = min cost to paint houses $1, \ldots, i$ with i green.
- $B[i]$ = min cost to paint houses $1, \ldots, i$ with i blue.
- Optimal cost = $\min \{ R[n], G[n], B[n] \}$.

Dynamic programming equation.

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

overlapping subproblems

Running time. $O(n)$.



SECTION 6.3

6. DYNAMIC PROGRAMMING I

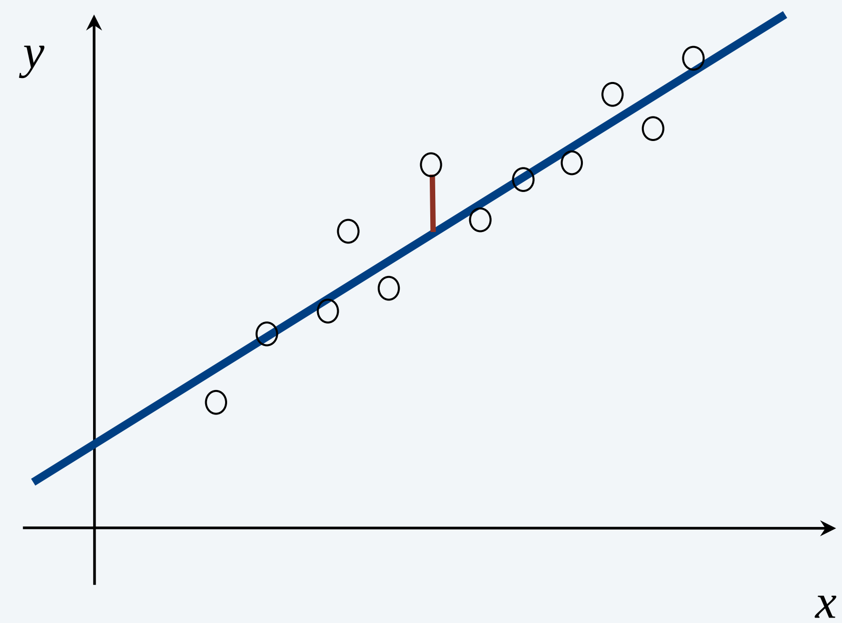
- *segmented least squares*

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

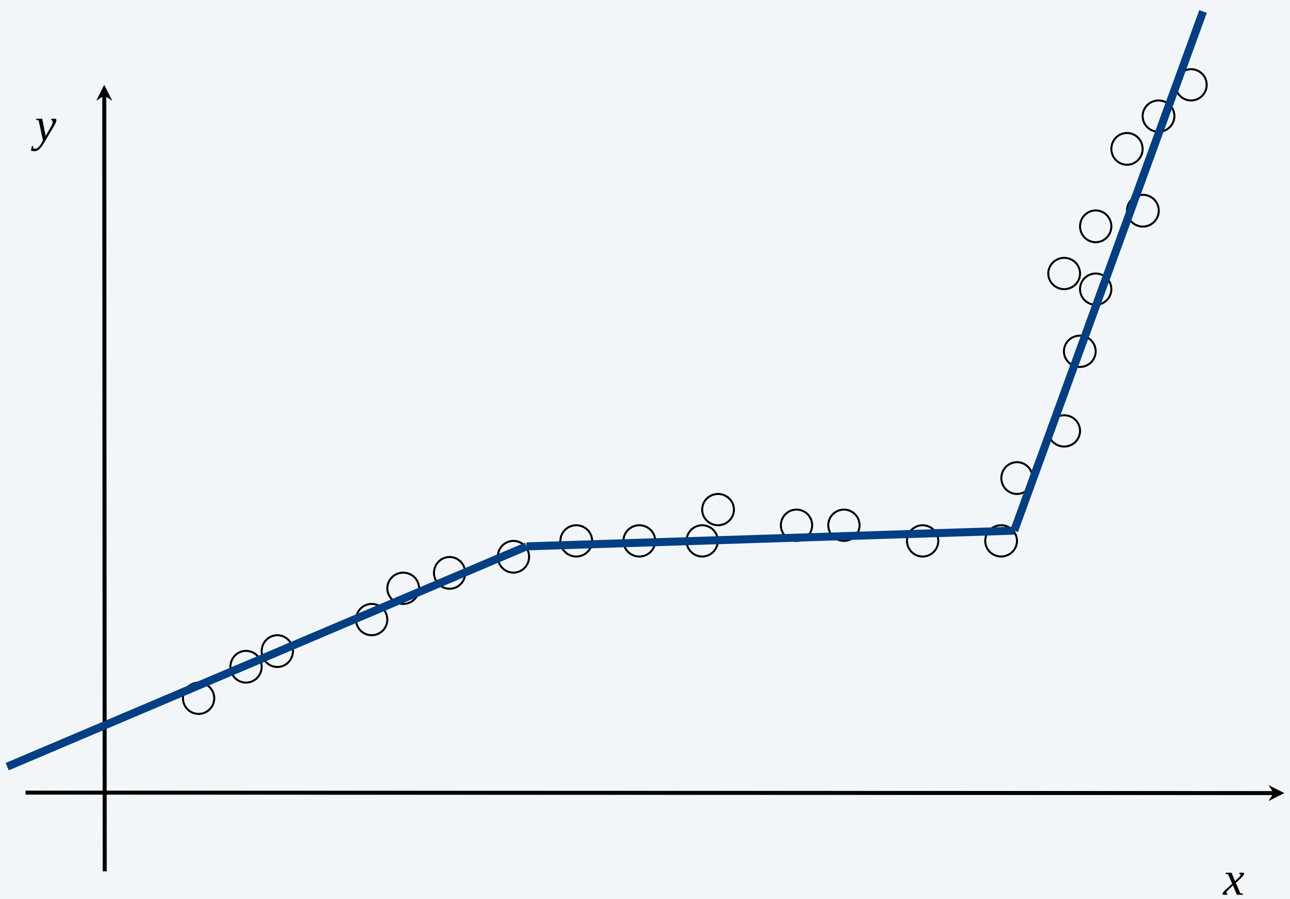
Segmented least squares.

- 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



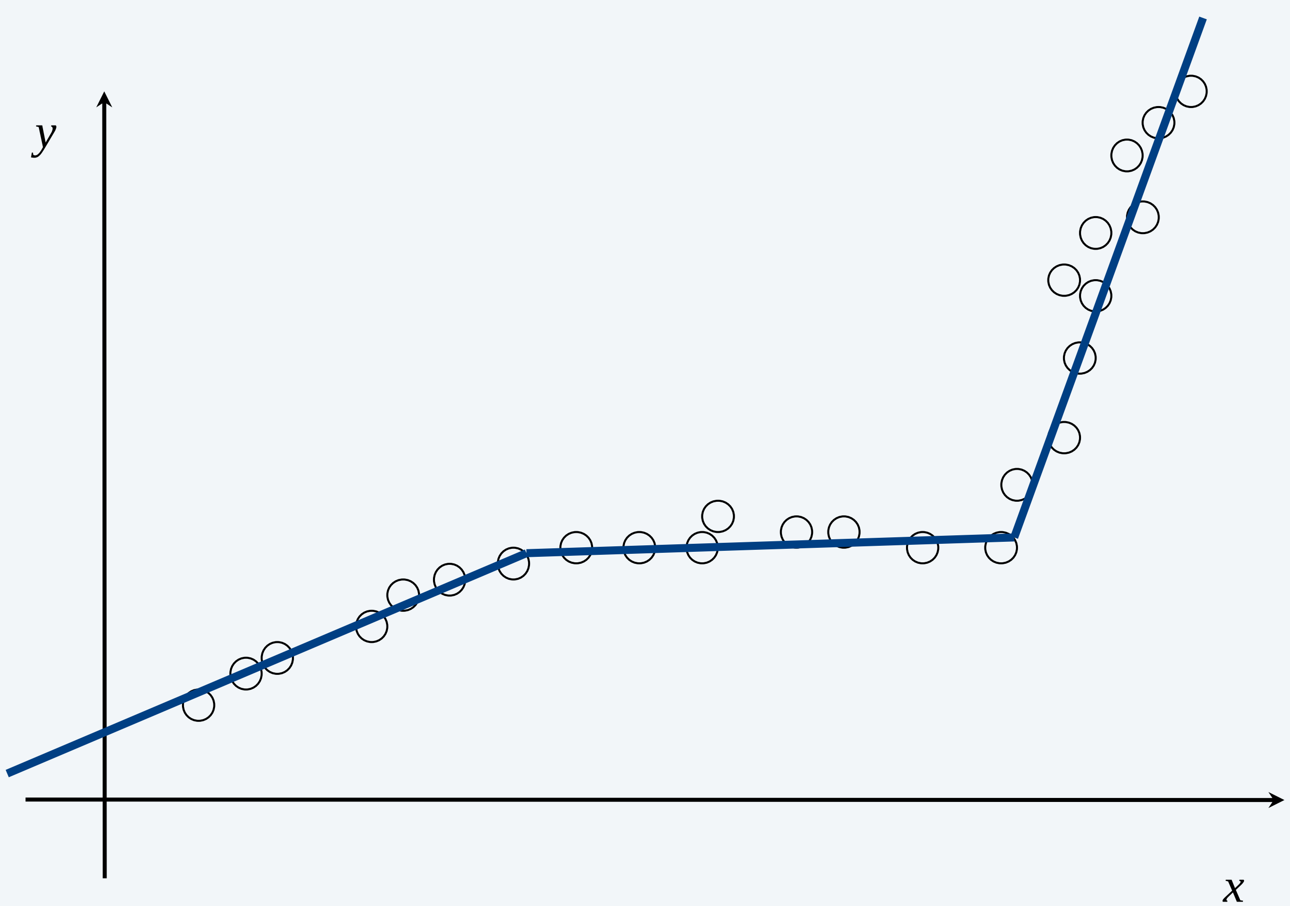
Segmented least squares

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 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 $i \leq j$.
- 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

  previously computed value

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

RETURN $M[n]$.

Segmented least squares analysis

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

Pf.

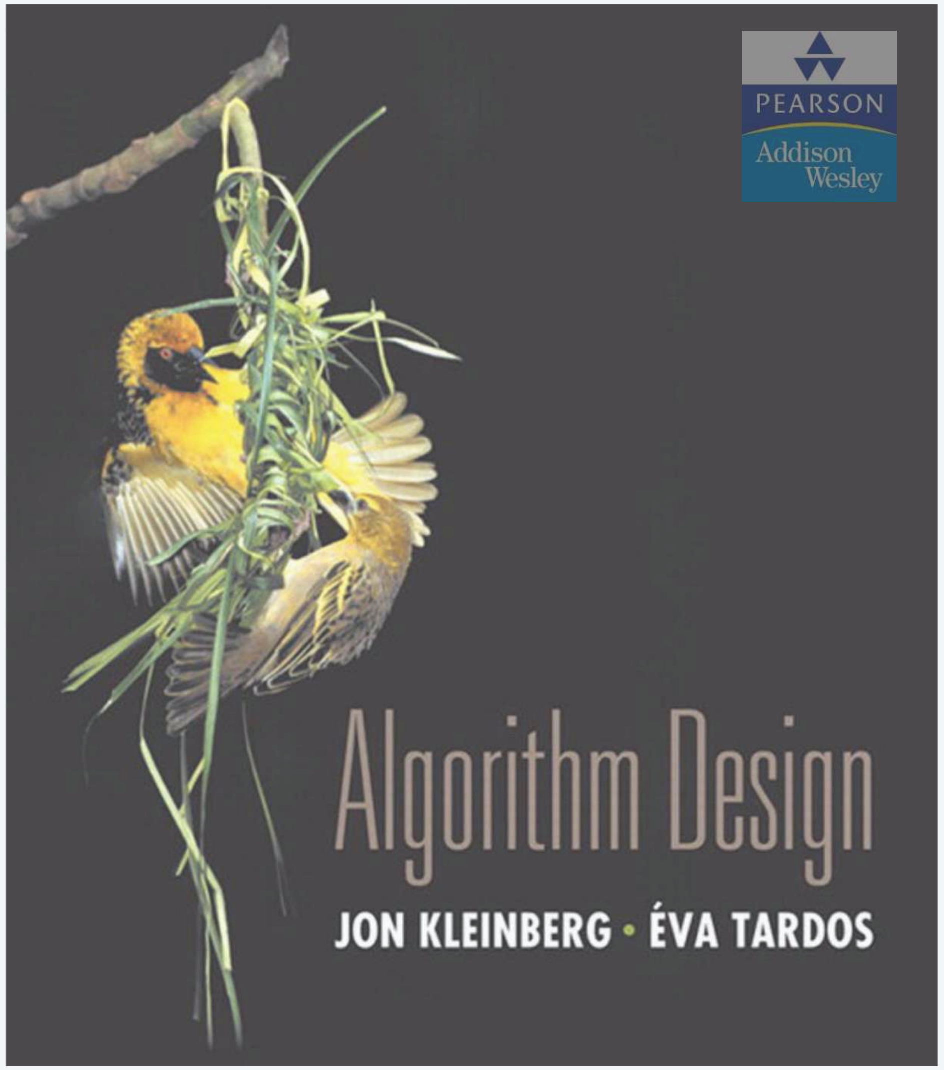
- Bottleneck = computing SSE e_{ij} for each i and j.

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

- $O(n)$ to compute e_{ij}. ▪

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

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



SECTION 6.4

6. DYNAMIC PROGRAMMING I

- *knapsack problem*

Knapsack problem

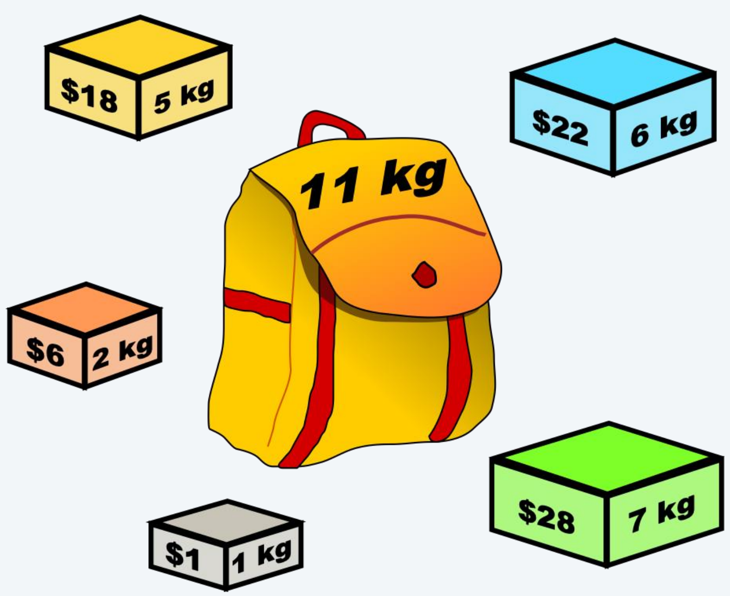
Goal. Pack knapsack so as to maximize total value of items taken.

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

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

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

Assumption. All values and weights are integral.



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

weights and values can be arbitrary positive integers

knapsack instance (weight limit W = 11)

Dynamic programming: false start

Def. $OPT(i)$ = optimal value of knapsack problem with 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: two variables

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

Goal. $OPT(n, W)$.

possibly because $w_i > w$

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

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i - 1 \}$ subject to 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 $\{ 1, 2, \ldots, i - 1 \}$ subject to 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, …, w_n, v_1, …, v_n)

FOR $w = 0$ TO W

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

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

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

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

previously computed values

RETURN $M[n, W]$.

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

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	1 USD	1 kg
2	6 USD	2 kg
3	18 USD	5 kg
4	22 USD	6 kg
5	28 USD	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) = 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.

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 $M[i, w] > M[i-1, w]$. ▪

Remarks.

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

Is the running time of the DP algorithm for the knapsack problem polynomial?

- No, because $\Theta(n\ W)$ is not a polynomial function of the input size.
- It is *pseudo-polynomial*.

Pseudo-polynomial algorithm: an algorithm whose running time is polynomial in the values of the input (e.g. the largest integer present in the input).

- efficient when numbers involved in the input are reasonably small (e.g., in the knapsack problem when w_i are small)
- not necessary polynomial in the input size (number of bits required to represent the input)