Portland State University

W'21 CS 584/684
Algorithm Design & Analysis

Fang Song

Lecture 7

- Dynamic programming
- Weighted interval scheduling

Credit: based on slides by K. Wayne

Historic note on dynamic programming

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.

- **Richard Bellman**
  - DP [1953] @RAND
  - B-Ford algorithm for general shortest path (stay tuned!)
  - Curse of dimensionality
  - …

- **Etymology**
  - Dynamic programming = planning over time
  - Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Reference: Bellman, R. E. Eye of the Hurricane, An Autobiography.

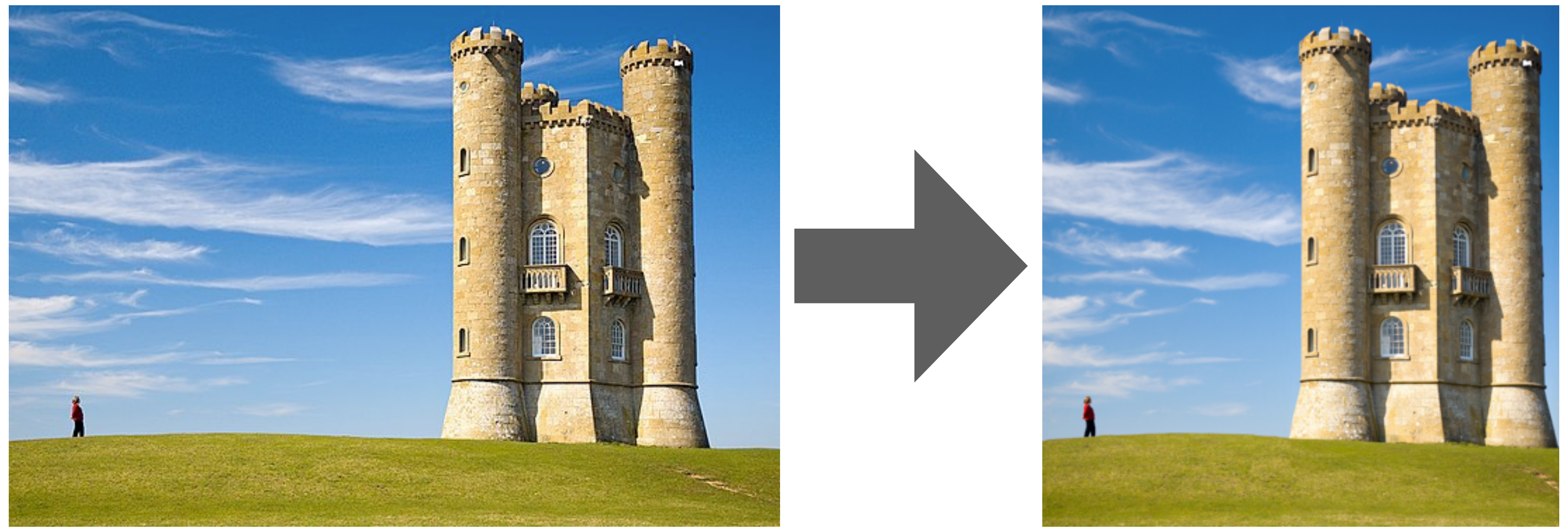
Dynamic programming applications

Indispensable technique for optimization problems.

- Many soln's, each has a value.
- Find a solution with optimal (min or max) value

- **Areas**
  - Computer science: theory, graphics, AI, compiler, systems, ...
  - Bioinformatics
  - Operations, information theory, control theory.

- **Famous DP algorithms**
  - Avidan–Shamir for seam carving.
  - Unix `diff` for comparing two files.
  - Viterbi for hidden Markov models.
  - Knuth–Plass for word wrapping text in TeX.
  - Cocke–Kasami–Younger for parsing context-free grammars.



Dynamic programming

- Break up a problem into a series of overlapping subproblems.

- There is an ordering on the subproblems, and a relation showing how to solve a subproblem given answers to “smaller” subproblems (i.e., those appear earlier in the ordering).

An implicit DAG: nodes = subproblems, edges = dependencies

Our examples on shortest path in DAGs and longest increasing subsequence (i.e., longest path in DAGs) have packed many ideas …

Fibonacci sequence

Leonardo of Pisa (Fibonacci)
1170 - 1250

- **Definition.**

0,1,1,2,3,5,8,13,21,34,…

$$a_0 = 0, \quad a_1 = 1, \quad a_2 = 1$$
$$a_n = a_{n-1} + a_{n-2}$$

Input: n.

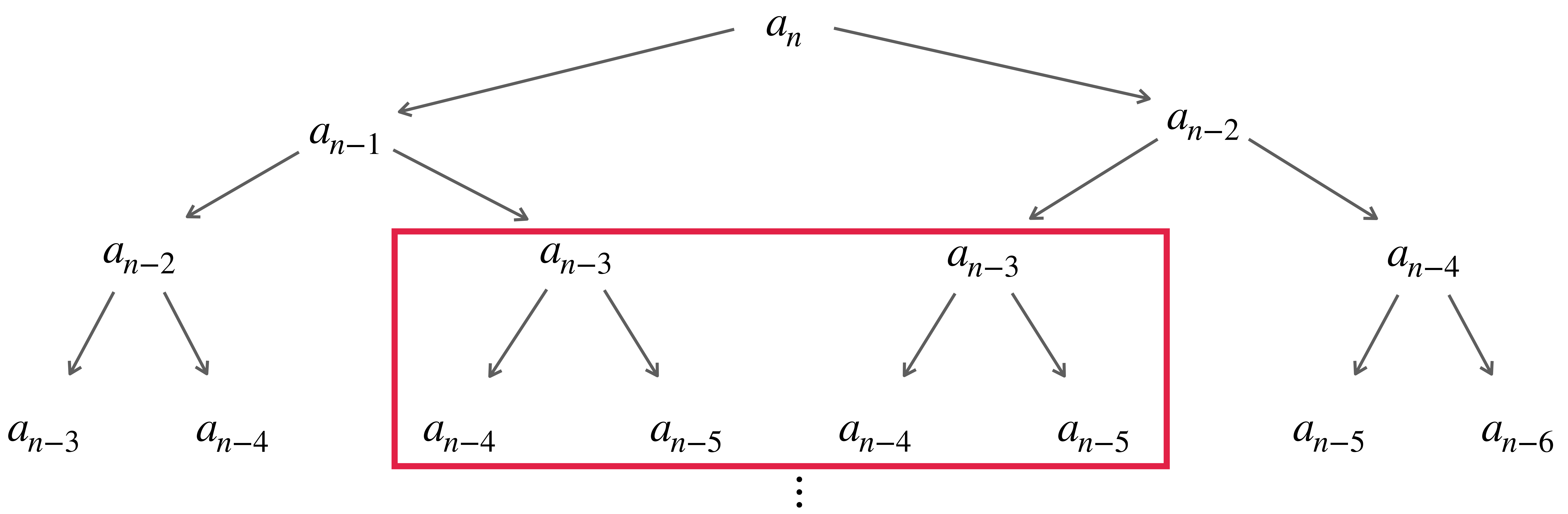
Output: a_n.

Fib(n): // A simple recursive algorithm
1. If $n = 0$, return 0
2. If $n = 1$, return 1
3. Return $Fib(n-1) + Fib(n-2)$

- **Correctness.**
- **Running time.**
  - $T(n) = T(n-1) + T(n-2) + \Theta(1)$ [Exercise. Show that $T(n) = 2^{O(n)}$.]
- **Can we do better?**

What we did? A "wasteful" recursion

- Lots of redundancy! Only $n-1$ distinct subproblems.

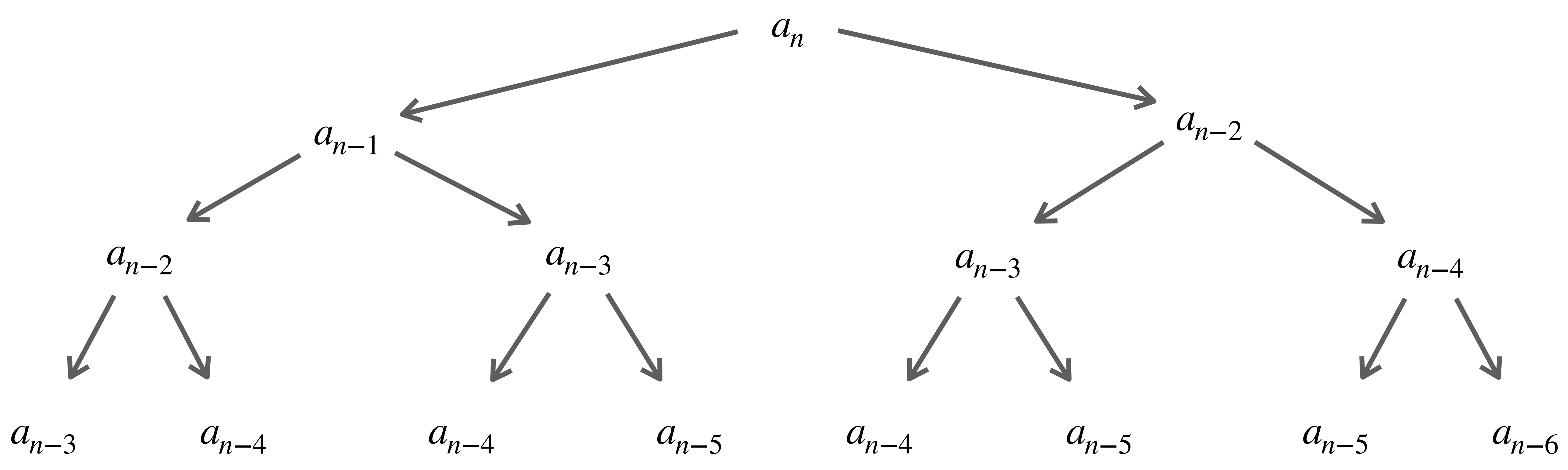


Why recursion in divide-&-conquer works great?

☺ independent & significantly smaller subproblems

A “smart” recursion by memoization

SmartFib(n):
// $a[0,\ldots,n]$ store subproblem values from recursive calls
1. If $n = 0$, return 0
2. If $n = 1$, return 1
3. Else
   If $a[n]$ not defined
      $a[n] \leftarrow$ SmartFib$(n-1)$ + SmartFib$(n-2)$
   Return $a[n]$



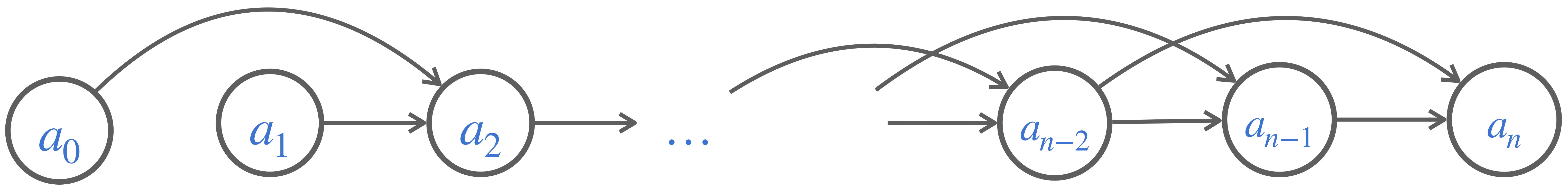
- Running time. Linear $O(n)$.
- Track the recursion tree
  - Fill up $a[\ldots]$ bottom up.

Fill it deliberately

IterFib(n):
// $a[0,\ldots,n]$ store subproblem values
1. $a[0] \leftarrow 0$
2. $a[1] \leftarrow 1$
3. For $i = 2,\ldots,n$
   $a[n] = a[n-1] + a[n-2]$
4. Return $a[n]$

- $O(n)$ additions.
- Space for storing $O(n)$ integers.
  - Can we save space further?

- DP is about smart recursion (i.e., without repetition) top-down.
- Usually easy to express by building up a table iteratively bottom-up.

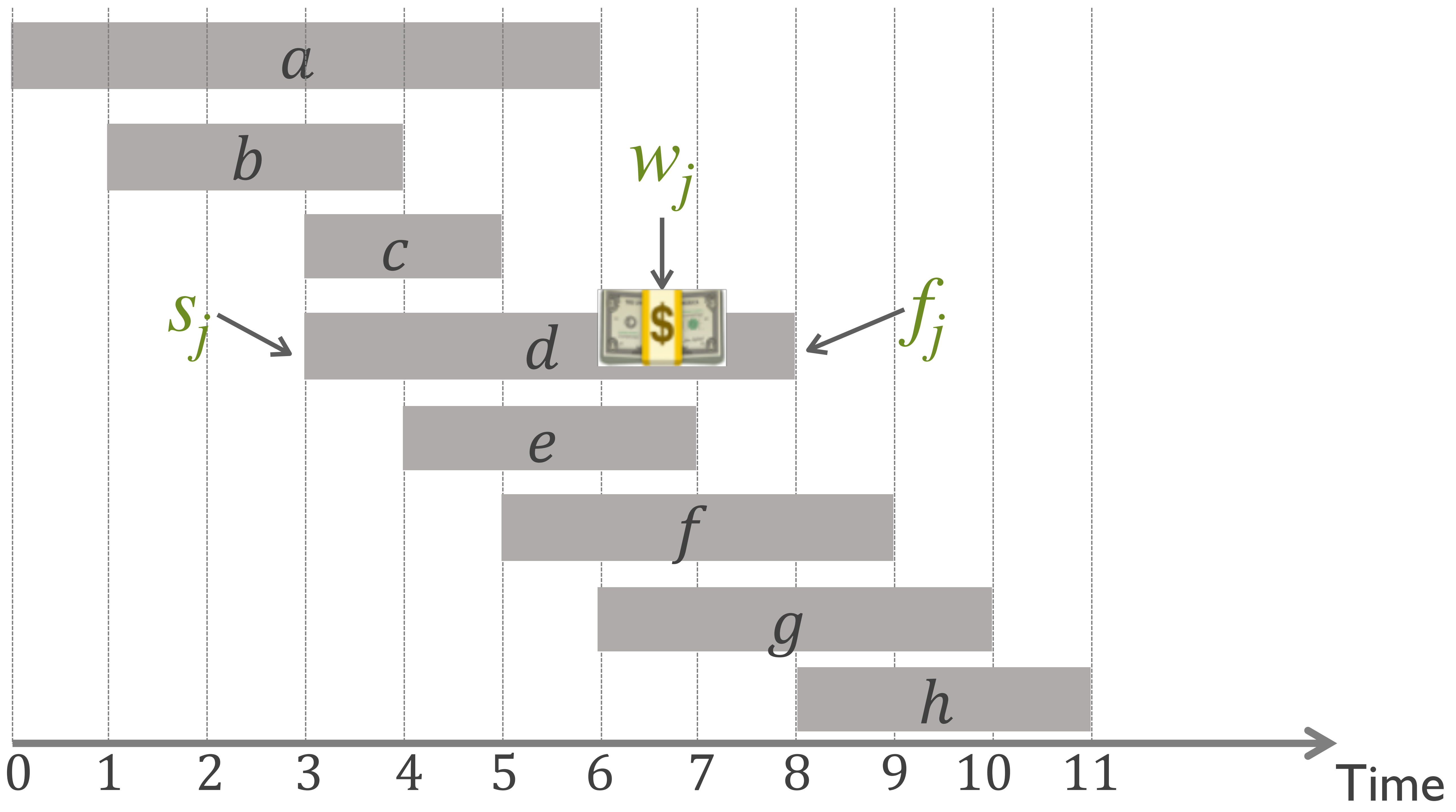


Weighted interval scheduling

Input: n jobs; job j starts at s_j, finishes at f_j, weight w_j.

Output: subset of mutually compatible jobs of maximum weight.

[i.e., they don't overlap]

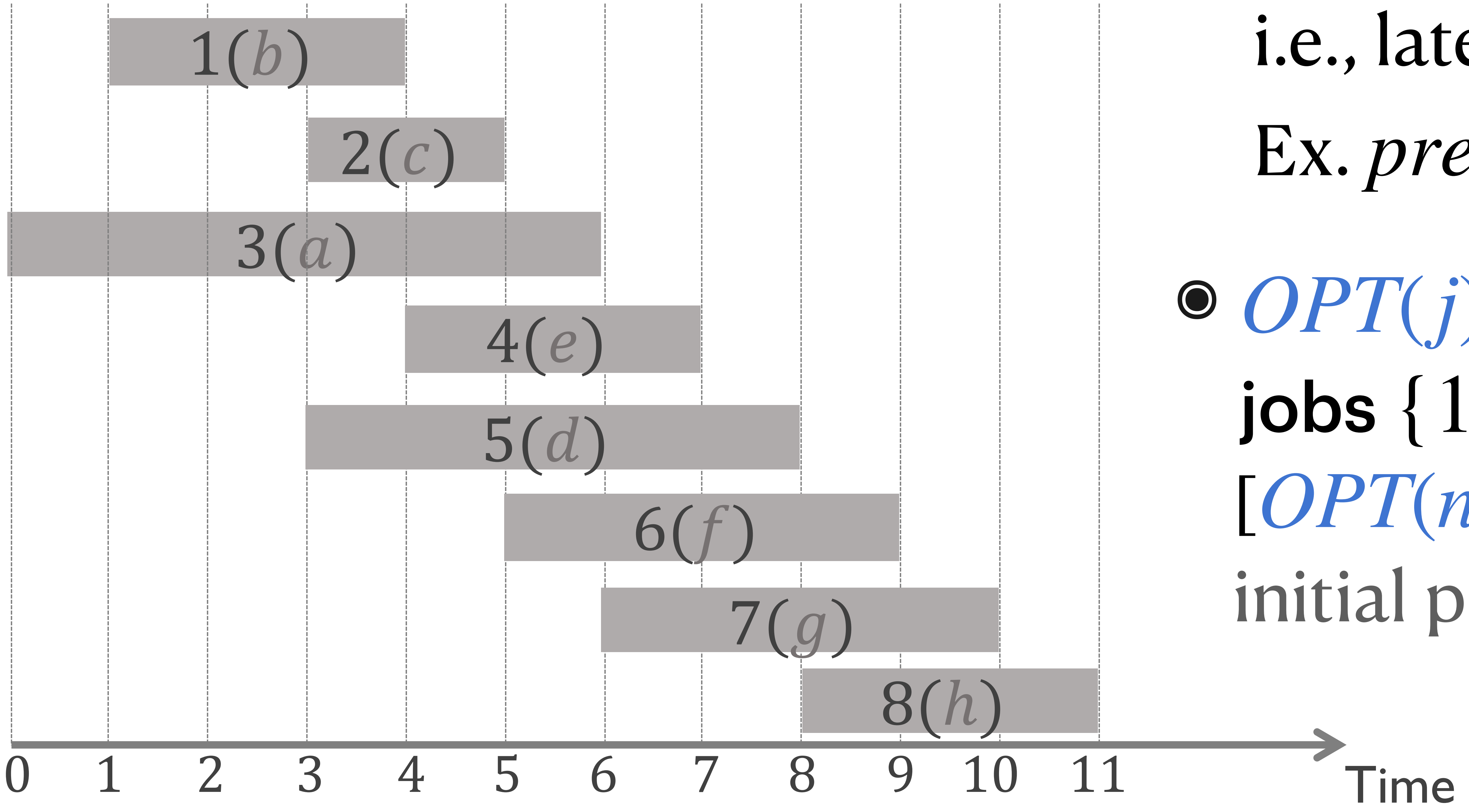


Assuming all $w_j = 1$,

$\{b, e, h\}$ is an optimal soln.

Weighted interval scheduling cont'd

- Label jobs by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$.



- Def. $pre(j)$ = largest index $i < j$ such that i is compatible with j.

  i.e., latest job before j & compatible with j.

  Ex. $pre(8) = 5, pre(7) = 3, pre(2) = 0$

- $OPT(j)$ = value of optimal solution to jobs $\{1,2,\ldots,j\}$.

  [$OPT(n)$: value of optimal solution to initial problem]

Forming the recursion for optimal solution

1. **Case 1.** $OPT(j)$ **does NOT select job** j**.**
   - Must include optimal solution to subproblem consisting of remaining compatible jobs $1,2,\ldots,j-1$: $OPT(j-1)$.

2. **Case 2.** $OPT(j)$ **selects job** j**.**
   - Collect profit w_j; exclude incompatible jobs $\{pre(j)+1, pre(j)+2, \ldots, j-1\}$
   - Include optimal solution to subproblem of remaining $1,2,\ldots,pre(j)$: $OPT(pre(j))$.

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max\{\underbrace{OPT(j-1)}_{\text{case1}}, \underbrace{w_j + OPT(pre(j))}_{\text{case2}}\}, & \text{otherwise} \end{cases}$$

"Wasteful" recursion

Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$.

Output: $OPT(n)$.

ComputeOPT(j):
// sort by finishing time so that $f_1 \leq f_2 \leq \ldots \leq f_n$
// compute $pre(1), pre(2), \ldots, pre(n)$
1. If $j = 0$, return 0
2. Else return
   $\max\{\text{ComputeOPT}(j-1), w_j + \text{ComputeOPT}(pre(j))\}$

- **Running time?**
  - Exponential(n) :(

"Smart" recursion by memoization

- Memoization. Store results of subproblems; lookup as needed.

M-computeOPT(j):
// sort by finishing time so that $f_1 \leq f_2 \leq \ldots \leq f_n$
// compute $pre(1), pre(2), \ldots, pre(n)$
// $M[0,\ldots,n]$ store subproblem values; $M[0] = 0$, others init to NULL

1. $M[1] = 0$
2. If $M[j] = NULL$
   $M[j] = \max\{\text{M-computeOPT}(j-1), w_j + \text{M-computeOPT}(pre(j))\}$
3. Return $M[j]$

- Running time M-computeOPT(n)?

Bottom-up dynamic programming

IterM-computeOPT(n):
// sort by finishing time so that $f_1 \leq f_2 \leq \ldots \leq f_n$ — $O(n \log n)$
// compute $pre(1), pre(2), \ldots, pre(n)$
// $M[0,\ldots,n]$ store subproblem values; init to 0
1. For $j = 1,\ldots,n$ — $O(n)$

   $M[j] = \max\{M[j-1], w_j + M[pre(j)]\}$
2. Return $M[n]$

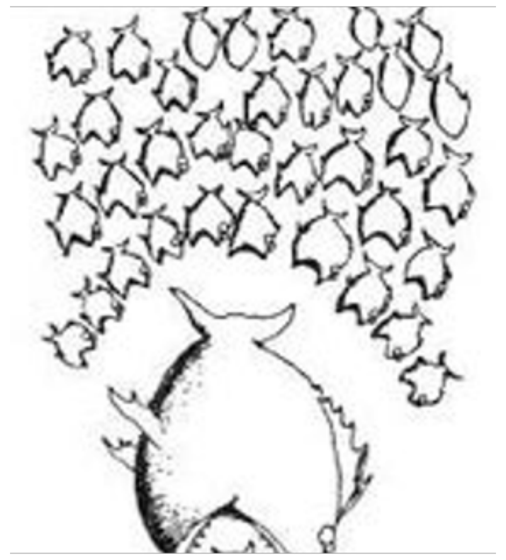
Previously computed values

- Running time IterM-computeOPT(n): $O(n \log n)$
- How to find an optimal solution, in addition to its value?
- What lessons we've learned?

Essence of dynamic programming

- Break up a problem into a series of overlapping subproblems.
- There is an ordering on the subproblems, and a relation showing how to solve a subproblem given answers to "smaller" subproblems.

Top-down



- DP is about smart recursion (i.e. without repetition) by momoization.

- Usually easy to express by building up a table iteratively.

Credit: Mary Wootters

Bottom up

A recipe for DP

1. **Formulate the problem recursively (key step).**
   a. Specification. Describe what problems to solve (not how).
   b. Recursion. Give a recursive formula for the whole problem in terms of answers to smaller instances of the same problem.
   c. Step back and double check.
2. **Build solutions to your recurrence (kinda routine).**
   a. Identify subproblems.
   b. Choose a memoization data structure.
   c. Identify dependencies and find a good order (DAG in topological order).
   d. Write down your algorithm.
   e. Analyze time. Find possible improvement if possible.

We usually go with bottom-up approach in this class.

Scratch