M, 09/23/19

Fall'19 CSCE 629

Analysis of Algorithms

Fang Song
Texas A&M U

Lecture 10

- Dynamic programming history
- Weighted interval scheduling

Credit: based on slides by K. Wayne

Dynamic Programming history

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

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.

- Etymology
  - Dynamic programming = planning over time (by filling a table)
  - 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 solutions, each has a value
Goal: a solution w. optimal (min or max) value

- Areas
  - Computer science: theory, graphics, AI, compilers, systems, …
  - Bioinformatics
  - Operations research, information theory, control theory
- Some 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 actually have many ideas wrapped …

Fibonacci sequence

Def. Fibonacci sequence

- Input. n
- Output. a_n

0,1,1,2,3,5,8,13,21,34, …

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

Leonardo of Pisa (Fibonacci)
$1170 - 1250$

- A simple recursive alg.

$\underline{\boldsymbol{Fib(n)}}$
1. If $n = 0$, return 0
2. If $n = 1$, return 1
3. return $\boldsymbol{Fib}(n-1) + \boldsymbol{Fib}(n-2)$

- Correctness ✓
- Running time?

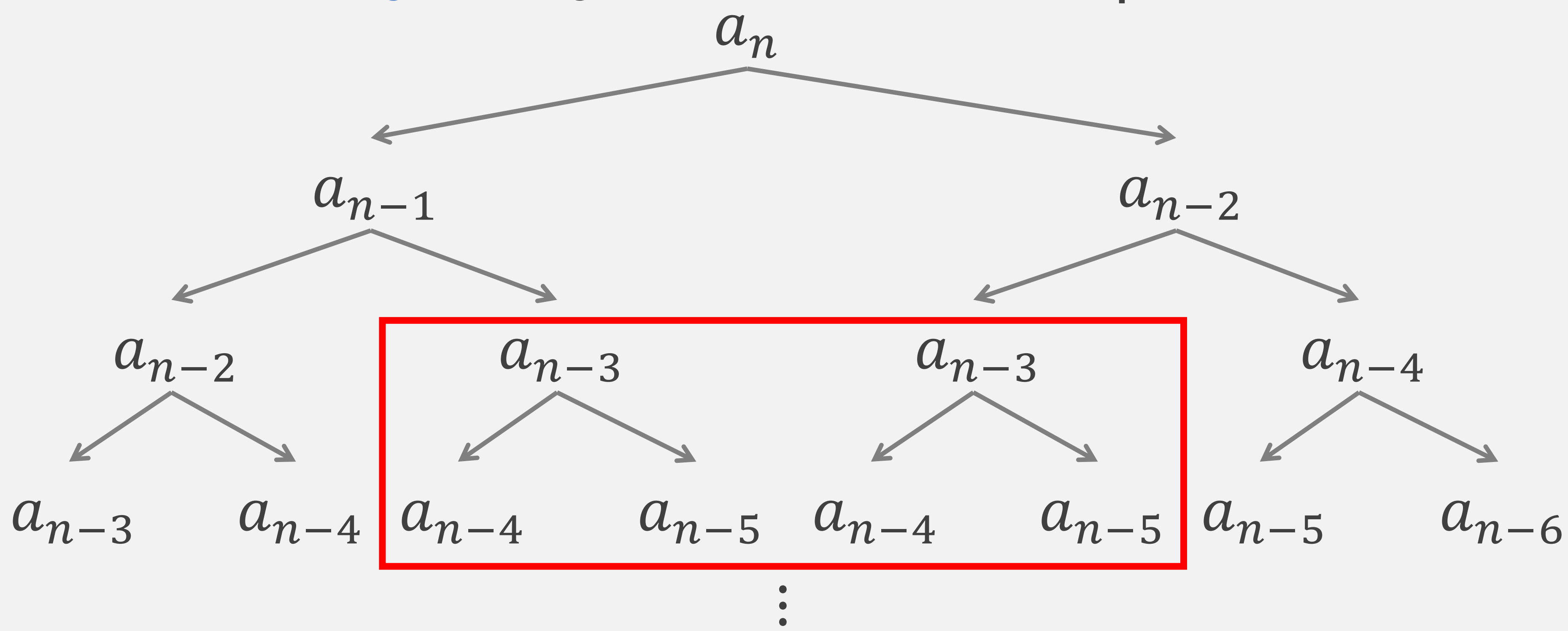
$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Exercise. Show that $T(n) = 2^{O(n)}$

- Can we do better?

A "dumb" recursion

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



Why recursion in divide-&-conquer works great?

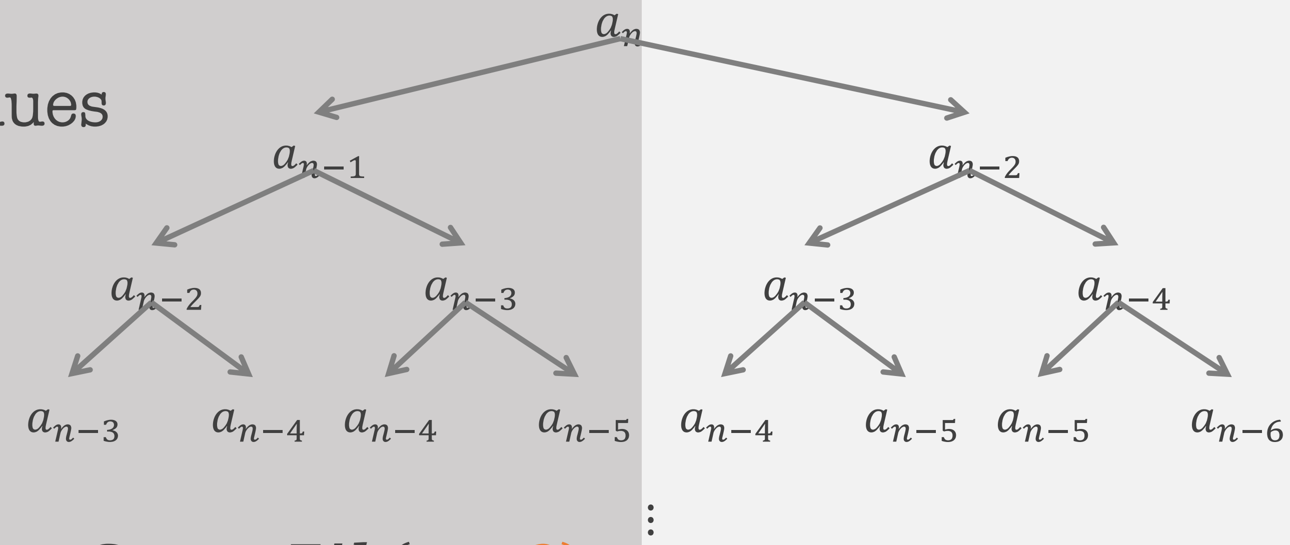
☺ independent & significantly smaller subproblems

A “smart” recursion by memoization

SmartFib(n)

// $\boldsymbol{a}[\mathbf{0}, \dots, \boldsymbol{n}]$ store subproblem values
  from recursive calls

1. If $n = 0$, return 0
2. If $n = 1$, return 1
3. Else
   If $\boldsymbol{a}[\boldsymbol{n}]$ not defined
     $a[n] \leftarrow \boldsymbol{SmartFib}(n-1) + \boldsymbol{SmartFib}(n-2)$
   return $\boldsymbol{a}[\boldsymbol{n}]$



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

Fill it deliberately

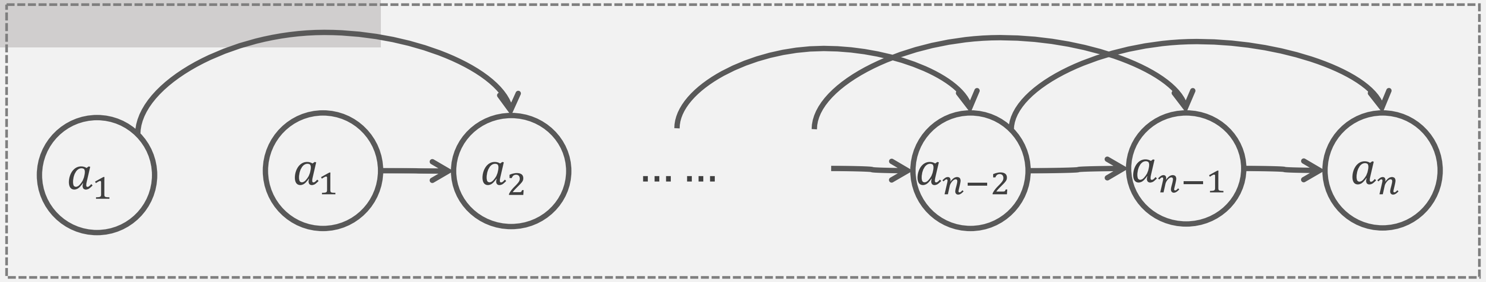
IterFib(n)

// $a[0, \ldots, n]$ store subproblem values

1. $a[0] \leftarrow 0$
2. $a[1] \leftarrow 1$
3. For $i = 2, \ldots, n$

   $a[n] \leftarrow a(n-1) + a(n-2)$
4. return $a[n]$

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

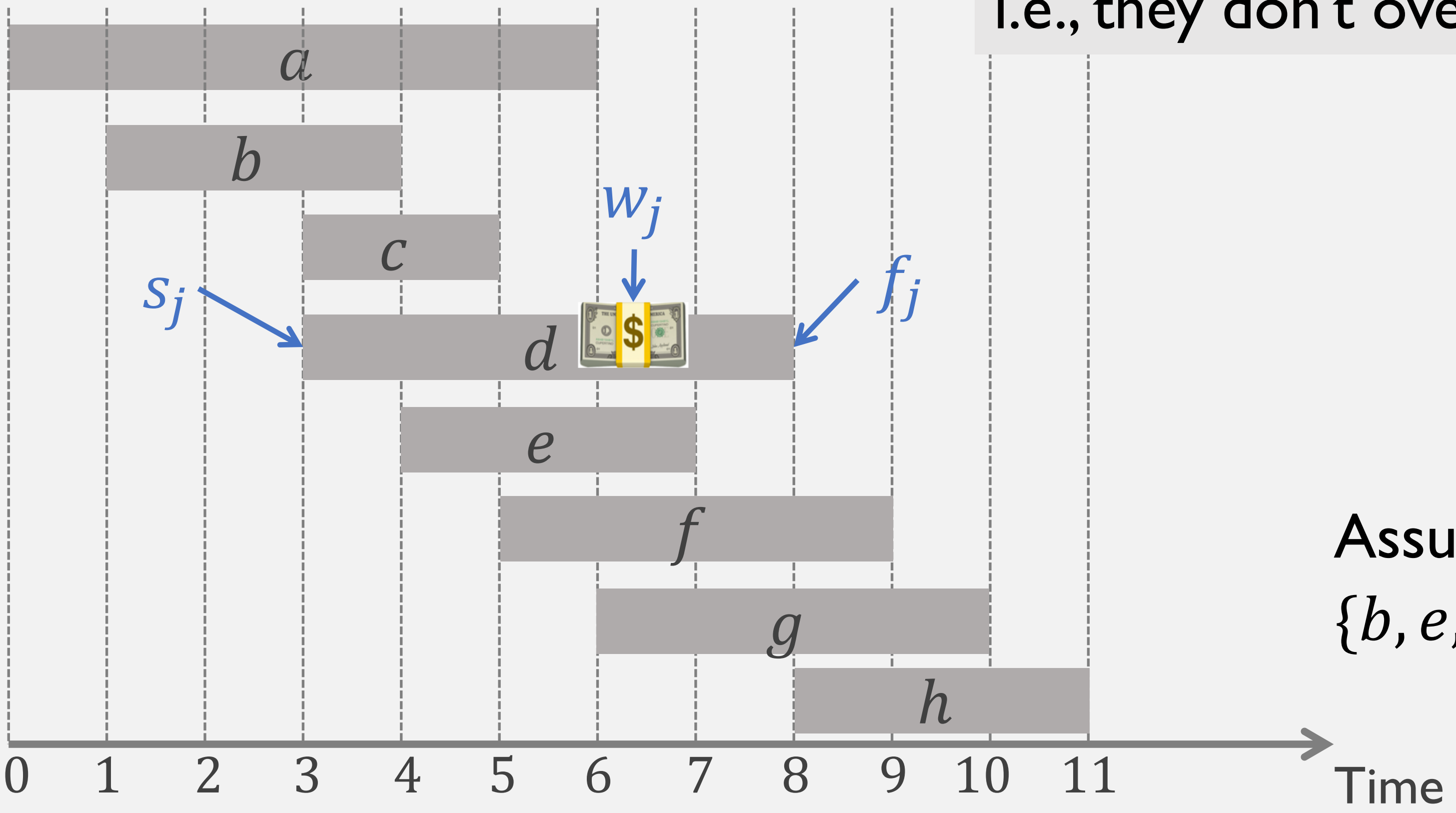


- $O(n)$ additions.
- Space for storing $O(n)$ integers
  - Can you save space?

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

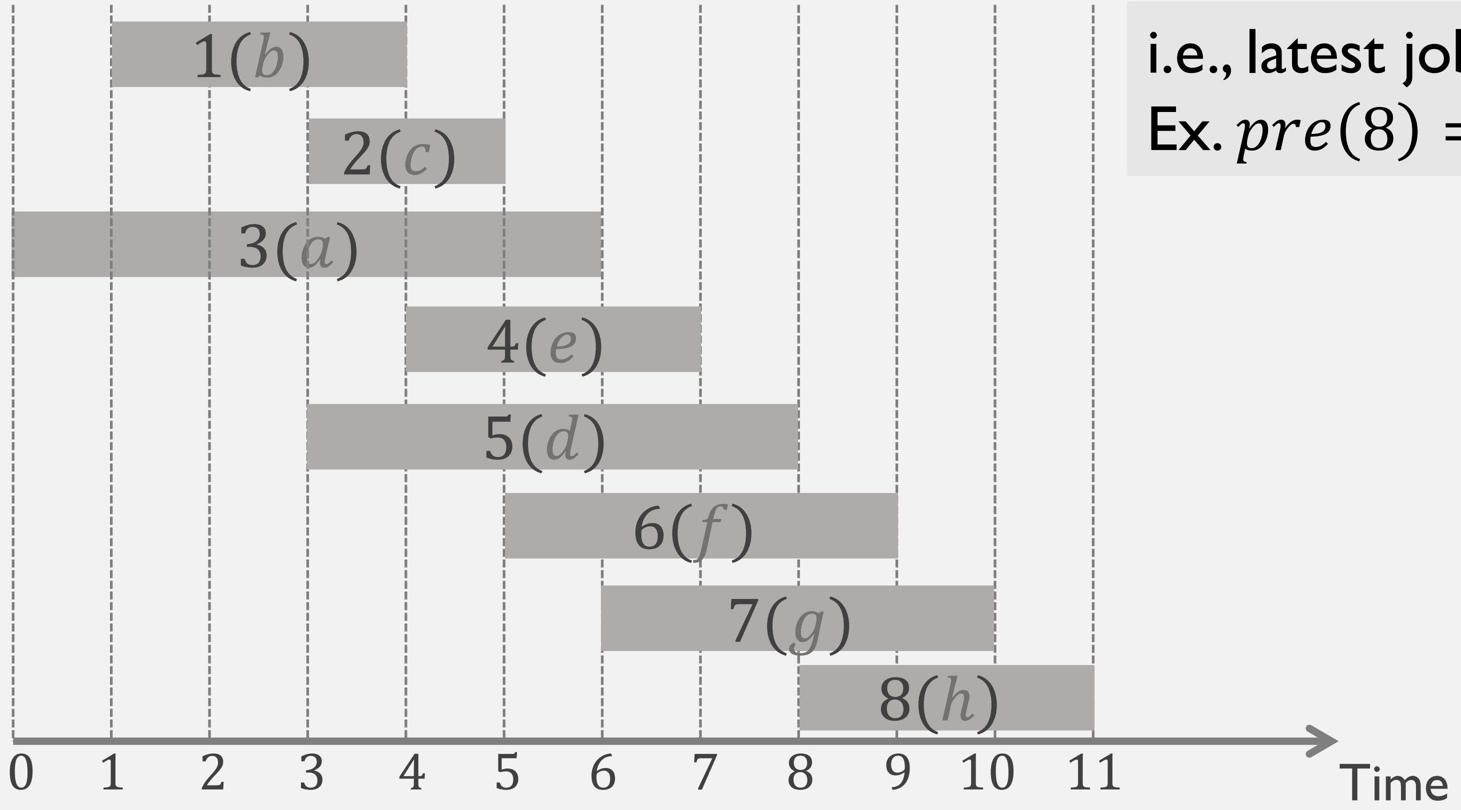


Assuming all $w_j = 1$,
$\{b, e, h\}$ is an optimal soln.

Weighted interval scheduling

Notation. Label jobs by finishing time $f_1 \leq f_2 \leq \cdots \leq f_n$

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



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

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

Forming the recursion for optimal solution

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

$\mathrm{OPT}(n)$: value of optimal soln. to initial problem

Case1. $\mathrm{OPT}(j)$ does NOT select job j

- Must include optimal solution to subproblem consisting of remaining compatible jobs $1,2,\ldots,j-1$

Case2. $\mathrm{OPT}(j)$ 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 compatible jobs $1,2,\ldots,pre(j)$

$$\mathrm{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\underbrace{\mathrm{OPT}(j-1)}_{\text{Case 1}}, \underbrace{w_j + \mathrm{OPT}(pre(j))}_{\text{Case 2}}\} & \text{otherwise} \end{cases}$$

"Dumb" recursion

- Input. n, $s_1, \dots, s_n$, $f_1, \dots, f_n$, $w_1, \dots, w_n$
- Output. $\mathrm{OPT}(n)$

// Sort by finishing time so that $f_1 \leq f_2 \leq \cdots \leq f_n$
// Compute $pre(1), pre(2), \dots, pre(n)$
ComputeOPT(j)
1. If $j = 0$
   return 0
2. Else
   return $\max\{\boldsymbol{ComputeOPT}(j-1), w_j + \boldsymbol{ComputeOPT}\big(pre(j)\big)\}$

- Running time $\boldsymbol{ComputeOPT}(n)$?
  - !!! Exponential(n)

"Smart " recursion by memoization

Memoization. Store results of subproblems in a table; lookup as needed.

// Sort by finishing time so that $f_1 \le f_2 \le \cdots \le f_n$
// Compute $pre(1), pre(2), \dots, pre(n)$
// $M[0, \dots, n]$ store subproblem values; $M[0] = 0$ others initialize to NULL
$\underline{\boldsymbol{M\text{−}ComputeOPT}(j)}$
1. $M[1] = 0$
2. If $M[j] = \text{NULL}$
   $\boldsymbol{M}[\boldsymbol{j}] = \max\{\boldsymbol{M\text{−}ComputeOPT}(j-1), w_j + \boldsymbol{M\text{−}ComputeOPT}\big(pre(j)\big)\}$
3. return $M[j]$

- Running time $\boldsymbol{M\text{−}ComputeOPT}(n)$?

Bottom-up dynamic programming

// Sort by finishing time so that $f_1 \leq f_2 \leq \cdots \leq f_n$
// Compute $pre(1), pre(2), \ldots, pre(n)$ → $O(n\log n)$
// $M[0, \ldots, n]$ store subproblem values; initialize to 0

IterM–ComputeOPT(n)

previously computed values

1. For $j = 1, \ldots, n$

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

2. return $M[n]$

$\Big\}\ O(n)$

- Running time? $O(n\log n)$
- How to find a optimal solution (rather than just its value)?
- What lessons we've learned?