HW1 due 5/21
HW2 out 5/21, due 5/28
Q2 out 5/21, due 5/24 NOON

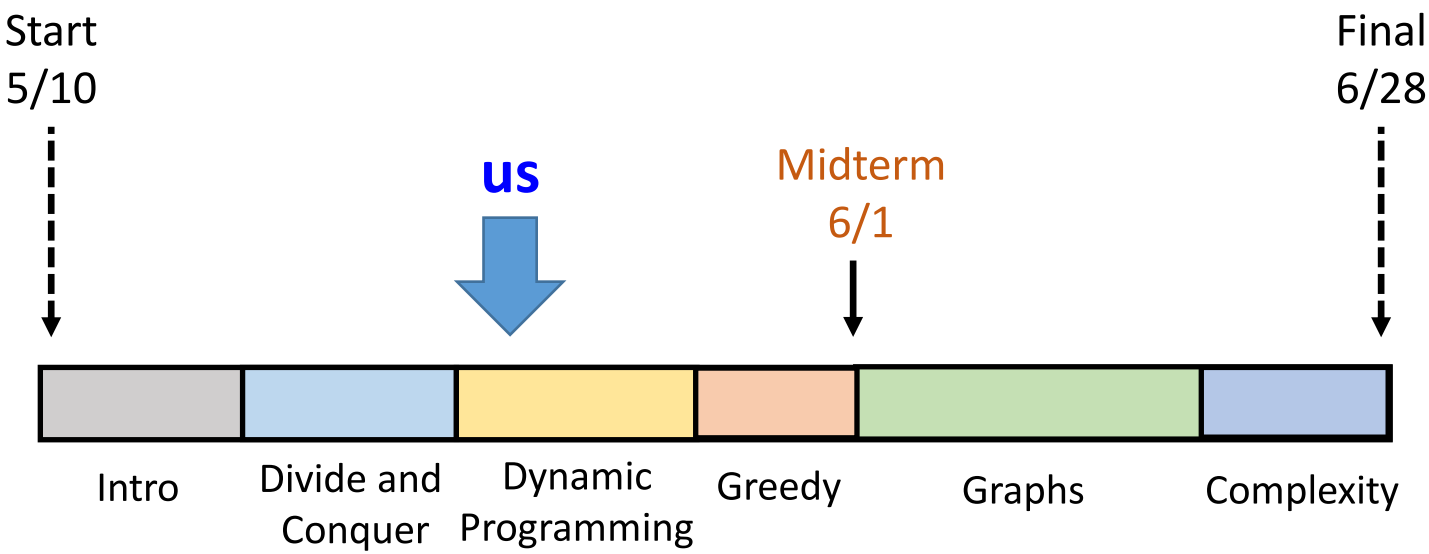
CS3000: Algorithms & Data
Drew van der Poel

Lecture 8

- Dynamic Programming: Fibonacci Numbers
- Dynamic Programming: Weighted Interval Scheduling

May 20, 2021

Outline



Last class: divide and conquer: Binary Search + Selection

Next class: dynamic programming: Knapsack

Dynamic Programming

- Don't think too hard about the name
  - *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.* –Richard Bellman
- Dynamic programming is careful & smarter recursion
  - Break the problem up into small pieces & recursively solve (like Divide & Conquer)
  - Reuse solutions as necessary when subproblems repeat
  - Don't combine solutions (like in D&C)
  - Often the only poly. time algorithm (D&C doesn't work)

- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection
- Alg. techniques: divide & conquer, **dynamic programming**
- Analysis: asymptotic analysis, recursion trees, Master Thm.
- Proof techniques: (strong) induction, contradiction

Intro: Fibonacci Numbers

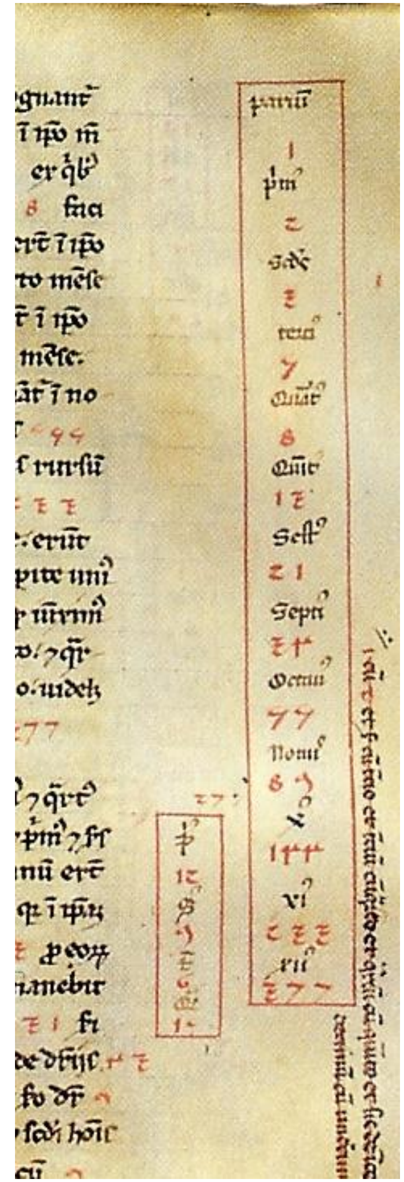
Fibonacci Numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- $F(1) = 0, F(2) = 1,$
  $F(n) = F(n-1) + F(n-2)$

  $F(3) = F(2) + F(1) = 1 + 0 = 1$

- $F(n) \to \phi^n \approx 1.62^n$
- $\phi = \left(\frac{1+\sqrt{5}}{2}\right)$ is the golden ratio



Fibonacci's *Liber Abaci* (1202)

Fibonacci Numbers: Take I

i	1	2	3	4	5	6
$Fib(i)$	0	1	1	2	3	5
$T(i)$	1	1	3	5	9	15

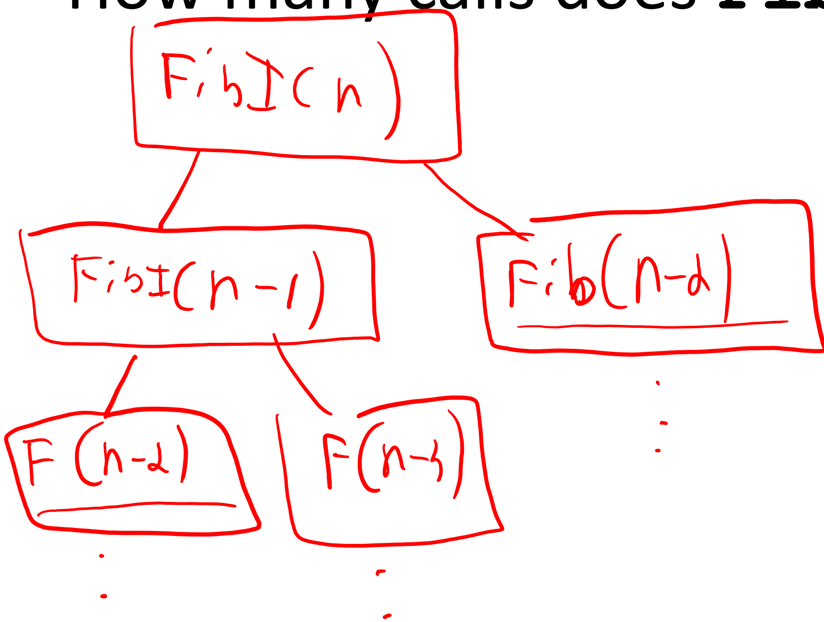
+1

```
FibI(n):
  If (n = 1): return 0
  ElseIf (n = 2): return 1
  Else: return FibI(n-1) + FibI(n-2)
```

$+T(n-1)$ $\quad +T(n-2)$

$T(3) = 3 = 2F(4) - 1 = 2 \cdot 2 - 1 = 3$

- How many calls does **FibI(n)** make?



- $T(n) = \#\ of\ calls\ by\ FibI(n)$

$T(1) = 1, \quad T(2) = 1$

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

$T(n) = 2F(n+1) - 1$

$\approx 2 \cdot 1.62^{n+1} - 1 = \Theta(1.62^n)$

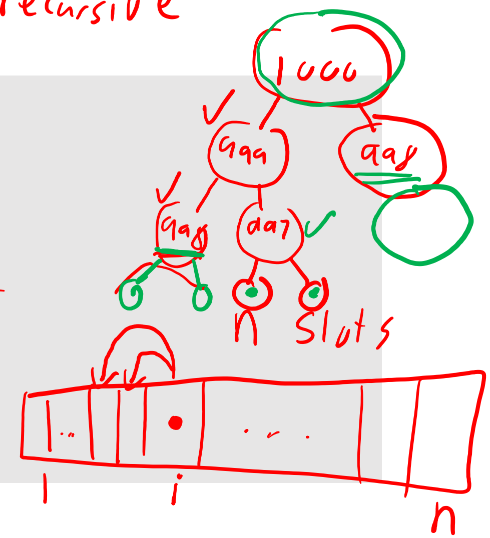
Fibonacci Numbers: Take II ("Top down")

"memoization"

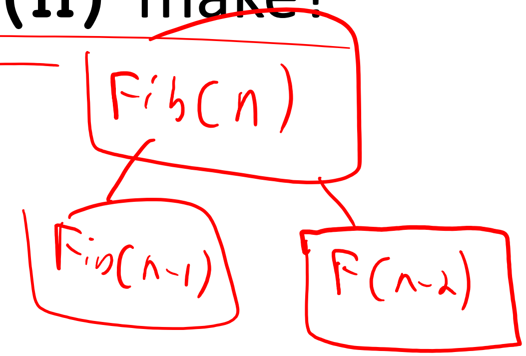
↳ recursive

DP Table of size n →

```
M ← empty array, M[1] ← 0, M[2] ← 1
FibII(n):
  If (M[n] is not empty): return M[n]
  ElseIf (M[n] is empty):
    M[n] ← FibII(n-1) + FibII(n-2)
    return M[n]
```



- How many recursive calls does **FibII(n)** make?



n-2 elems. filled

each fill → requires 2 rec. calls ✓

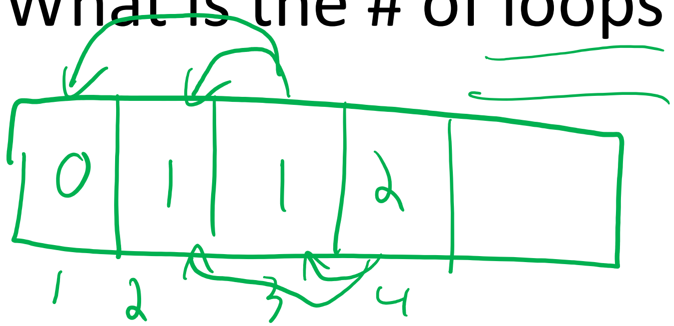
TOTAL: $2 \cdot (n-2) + 1 = \Theta(n)$

Fibonacci Numbers: Take III ("Bottom up")

↳ iterative

```
FibIII(n):
  M[1] ← 0, M[2] ← 1
  For i = 3,…,n:
    M[i] ← M[i-1] + M[i-2]
  return M[n]
```

- What is the # of loops of **FibIII(n)**?



n−2 loops

→ $\Theta(n)$

Fibonacci Numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$
- $F(n) = F(n-1) + F(n-2)$

- Solving the recurrence recursively takes $\Omega(1.62^n)$ time
  - Problem: Recompute the same values $F(i)$ many times
- Two ways to improve the running time
  - Remember values you've already computed ("top down")
  - Iterate over all values $F(i)$ ("bottom up")

- **Fact:** Fastest algorithms solve in logarithmic time

Dynamic Programming Recipe

- **Recipe:**
  - (1) identify a set of **subproblems**
  - (2) relate the subproblems via a **recurrence**
  - (3) find an **efficient implementation** of the recurrence (top down or bottom up) → fills in DP table
  - * (4) **reconstruct the solution** from the DP table

$F(i) \quad i \in [1, n]$

$F(i) = F(i-1) + F(i-2)$

Dynamic Programming: Weighted Interval Scheduling

Weighted Interval Scheduling

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …
- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \dots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$

- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, **weighted interval scheduling**

- Alg. techniques: divide & conquer, dynamic programming

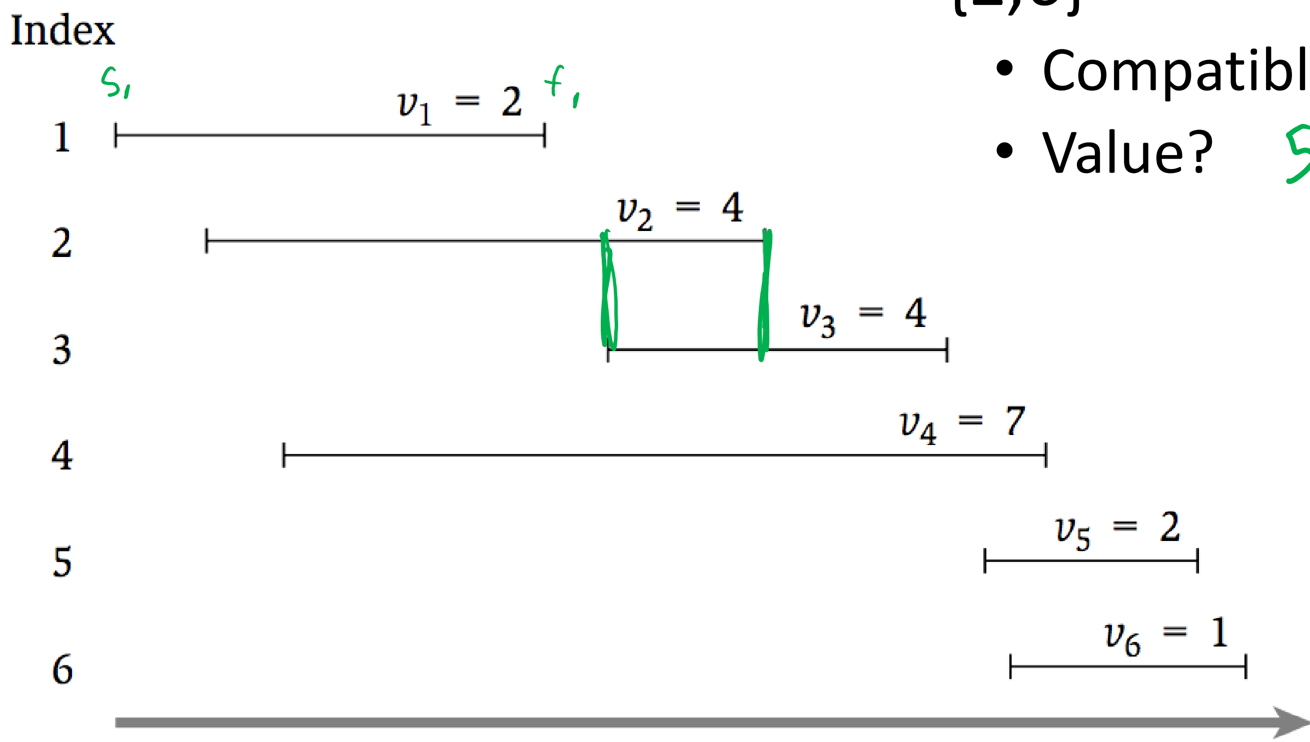
- Analysis: asymptotic analysis, recursion trees, Master Thm.

- Proof techniques: (strong) induction, contradiction

Interval Scheduling

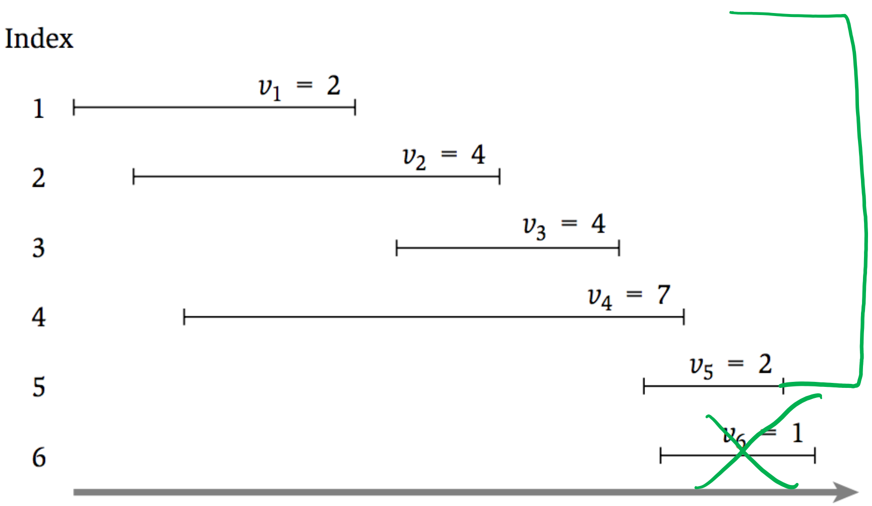
$n = 6$

- {2,3}
  - Compatible? no
  - Value? 8
- {2,6}
  - Compatible? yes
  - Value? 5



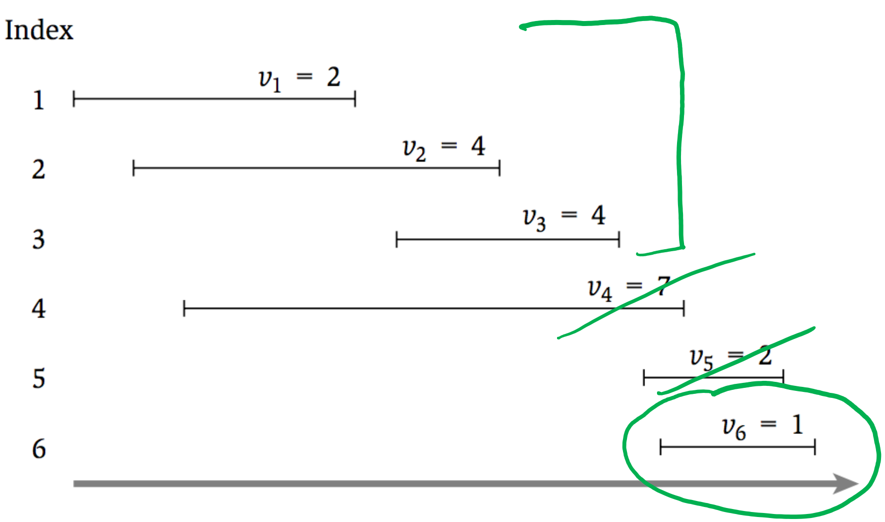
A Recursive Formulation

- Let O be the **optimal** schedule
- **Case 1:** Final interval is not in O (i.e. $6 \notin O$)
  - Then O must be the optimal solution for $\{1, \ldots, 5\}$



A Recursive Formulation

- Let O be the **optimal** schedule
- **Case 2:** Final interval is in O (i.e. $6 \in O$)
  - Then O must be $\{6\}$ + the optimal solution for $\{1, \dots, 3\}$



A Recursive Formulation

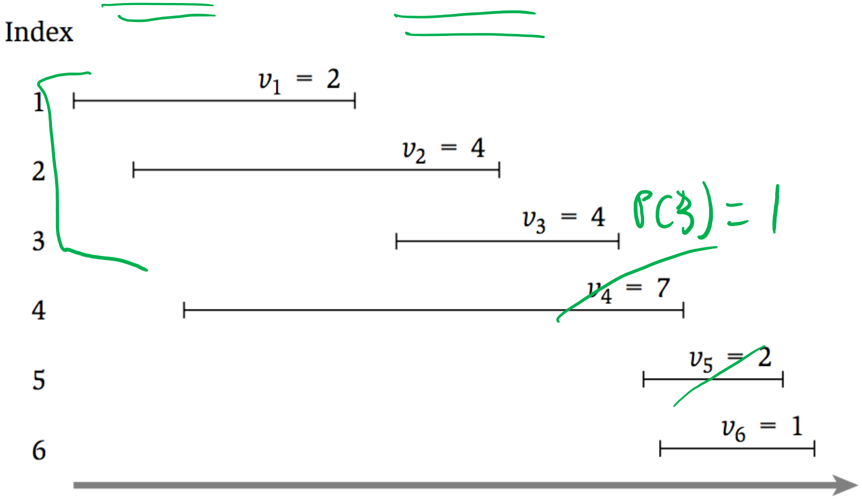
Which is better?

- the optimal solution for $\{1, \ldots, 5\}$
- $\{6\}$ + the optimal solution for $\{1, \ldots, 3\}$

Case 1
6 is out

Case 2
6 is in

$\{6\} \cup O_3$



$p(3) = 1$

Subproblems:
Opt. soln. for $\{1, \ldots, i\}$

$1 \le i \le n$

A Recursive Formulation: Subproblems

- ***Subproblems:*** Let O_i be the **optimal schedule** using only the intervals $\{1, \ldots, i\}$
- **Case 1:** Final interval is not in O_i $(i \notin O_i)$
  - Then O_i must be the optimal solution for $\{1, \ldots, i-1\}$
    - $O_i = O_{i-1}$
- **Case 2:** Final interval is in O_i $(i \in O_i)$
  - Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  - $P(6) = 3$ Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O_i must be i + the optimal solution for $\{1, \ldots, p(i)\}$
    - $O_i = \{i\} + O_{p(i)}$ $\cup$

relevant subproblems: O_{i-1}, $O_{p(i)}$

$\max\left(O_{i-1},\ \{i\} \cup O_{p(i)}\right)$

A Recursive Formulation: Subproblems & Recurrence

a value, not a set of intervals

- **Subproblems:** Let $OPT(i)$ be the ***value* of the optimal schedule** using only the intervals $\{1, \ldots, i\}$ $(OPT(i) = value(O_i))$

- **Case 1:** Final interval is not in O_i $(i \notin O_i)$
  - Then O_i must be the optimal solution for $\{1, \ldots, i-1\}$
- **Case 2:** Final interval is in O_i $(i \in O_i)$
  - Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  - Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O_i must be i + the optimal solution for $\{1, \ldots, p(i)\}$

$2 \leq i \leq n$ i out i in

- $OPT(i) = \max\{OPT(i-1), v_i + OPT(p(i))\}$

- $OPT(0) = 0, OPT(1) = v_1$

recurrence

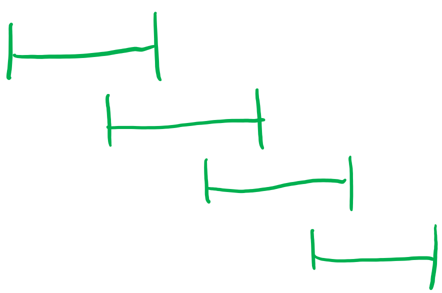
Dynamic Programming Recipe

- **Recipe:**
  - (1) identify a set of **subproblems**
  - (2) relate the subproblems via a **recurrence**
  - (3) find an **efficient implementation** of the recurrence (top down or bottom up)
  - (4) **reconstruct the solution** from the DP table

Interval Scheduling: Straight Recursion Homie

```
// All inputs are global vars
FindOPT(n):
  if (n = 0): return 0
  elseif (n = 1): return v1
  else:
    return max{FindOPT(n-1), vn + FindOPT(p(n))}
```

- What is the worst-case running time of **`FindOPT(n)`** (how many recursive calls)?



exponential

Interval Scheduling: Top Down

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← v1
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← max{FindOPT(n-1), vn + FindOPT(p(n))}
    return M[n]
```

- What is the running time of **`FindOPT(n)`**?

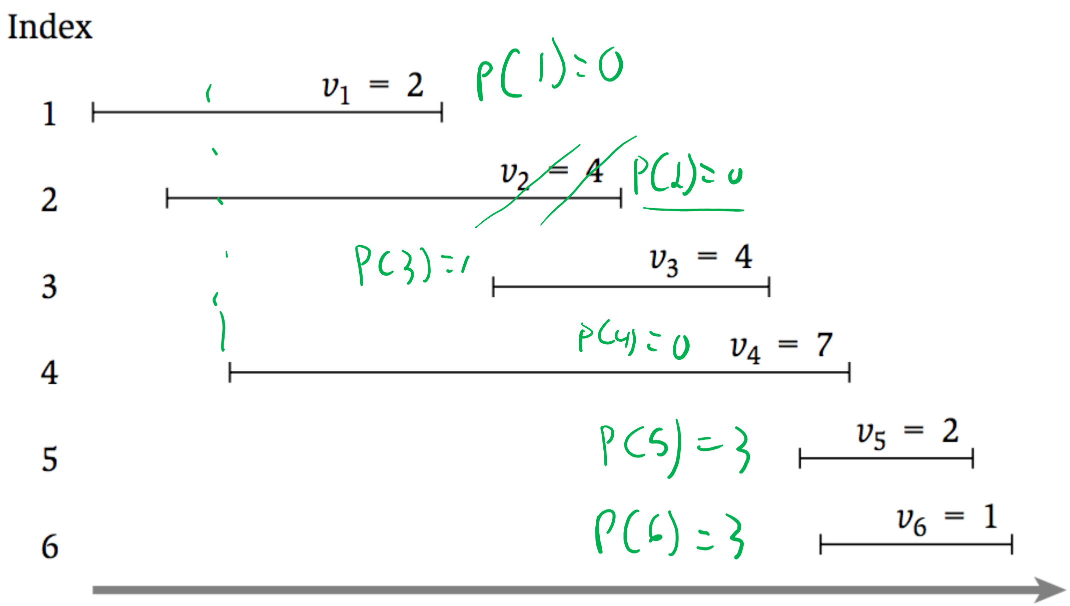
n-1 elems. filled

each fill → 2 rec. calls

Total: $(n-1)\,2 \to \Theta(n)$

Interval Scheduling: Top Down

$n = 6$



$FO(6): FO(5)$ vs. $FO(3) + v_6$

$8 > 6 + 1$

$FO(5): FO(4)$ vs. $FO(3) + v_5$

$7 < 6 + 2$

$FO(4): FO(3)$ vs. $FO(0) + v_4$

$6 < 0 + 7$

$FO(3): FO(2)$ vs. $FO(1) + v_3$

$4 < 2 + 4$

$FO(2): FO(1)$ vs. $FO(0) + v_2$

$2 < 0 + 4$

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

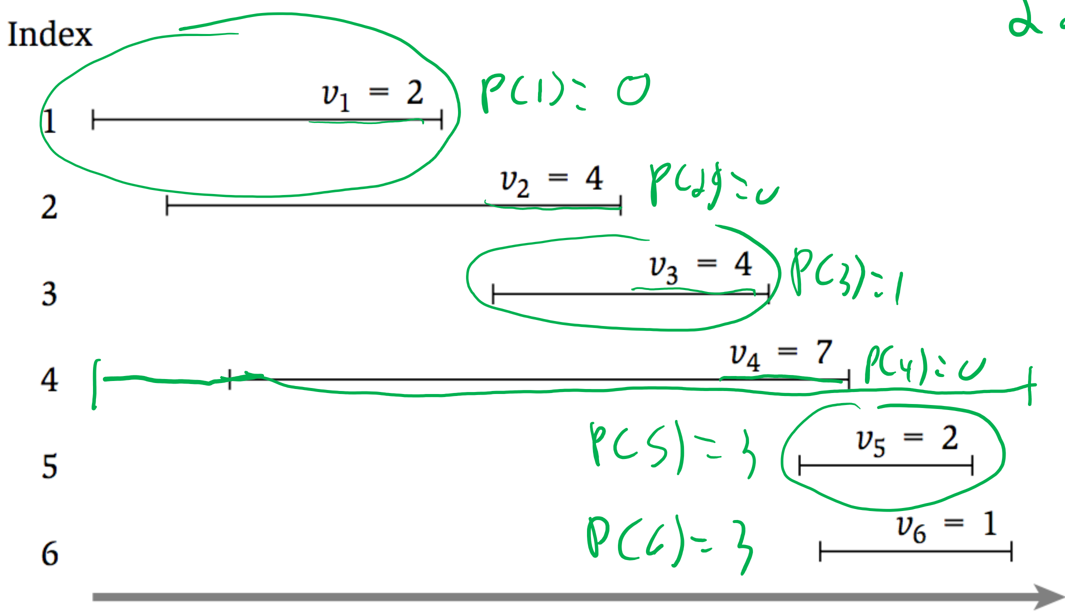
Interval Scheduling: Bottom Up

```
// All inputs are global vars
FindOPT(n):
  M[0] ← 0, M[1] ← v₁
  for (i = 2,…,n):
    M[i] ← max{M[i-1], vᵢ + M[p(i)]}
  return M[n]
```

- What is the running time of **FindOPT(n)**?

n-1 loops → $\Theta(n)$ total r/t

Interval Scheduling: Bottom Up



2: $M[1]$ vs. $v_2 + M[0]$

$2 < 4 + 0$

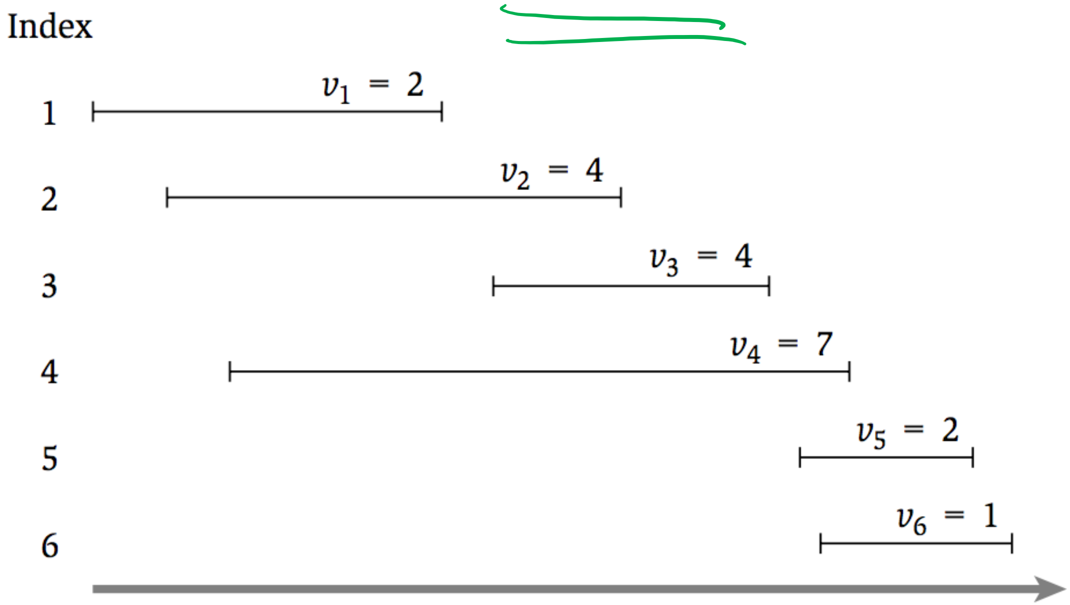
3: $M[2]$ vs. $v_3 + M[1]$

$4 < 4 + 2$

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Finding the Optimal Solution

- But we want a schedule, not a value!



M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Dynamic Programming Recipe

- **Recipe:**

  (1) identify a set of **subproblems**

  (2) relate the subproblems via a **recurrence**

  (3) find an **efficient implementation** of the recurrence (top down or bottom up)

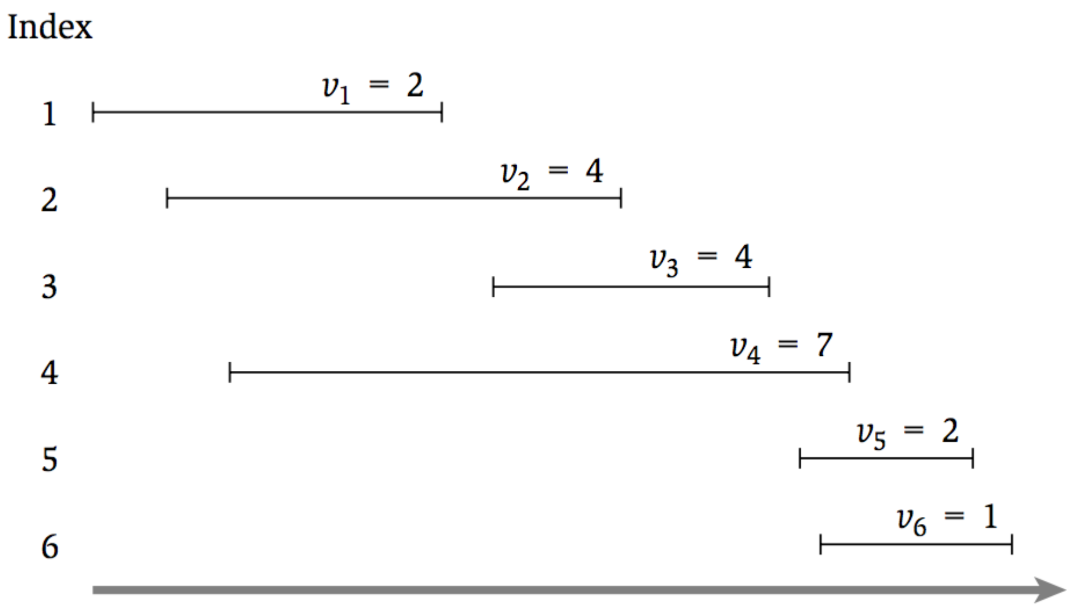
  (4) **reconstruct the solution** from the DP table

Finding the Optimal Solution

```
// All inputs are global vars
FindSched(M,n):
  if (n = 0): return ∅
  elseif (n = 1): return {1}
  elseif (vn + M[p(n)] > M[n-1]):
    return {n} + FindSched(M,p(n))
  else:
    return FindSched(M,n-1)
```

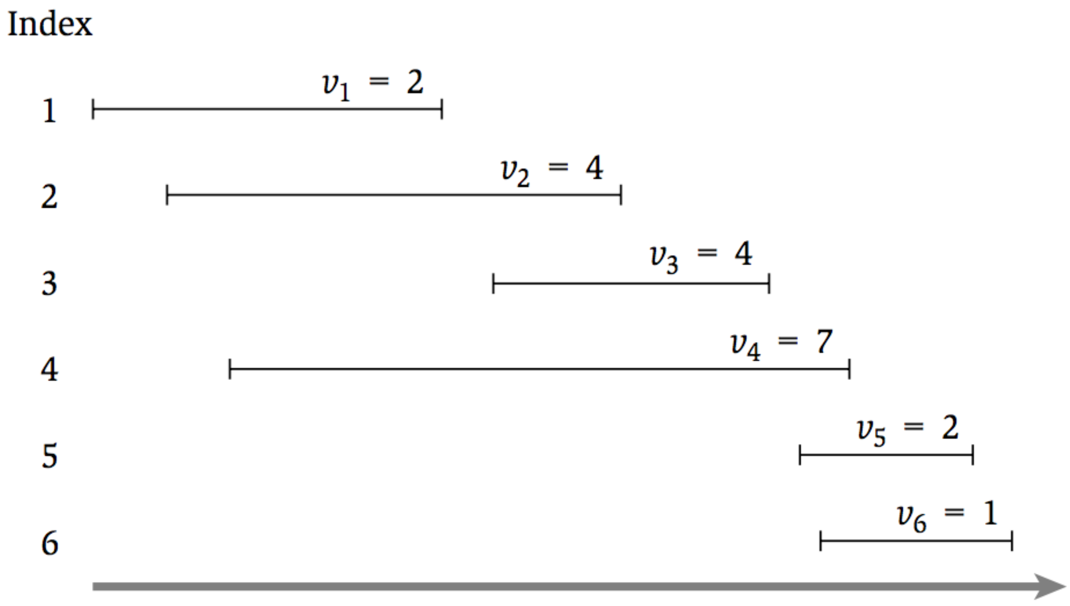
- What is the running time of **`FindSched(n)`**?

Finding the Optimal Solution



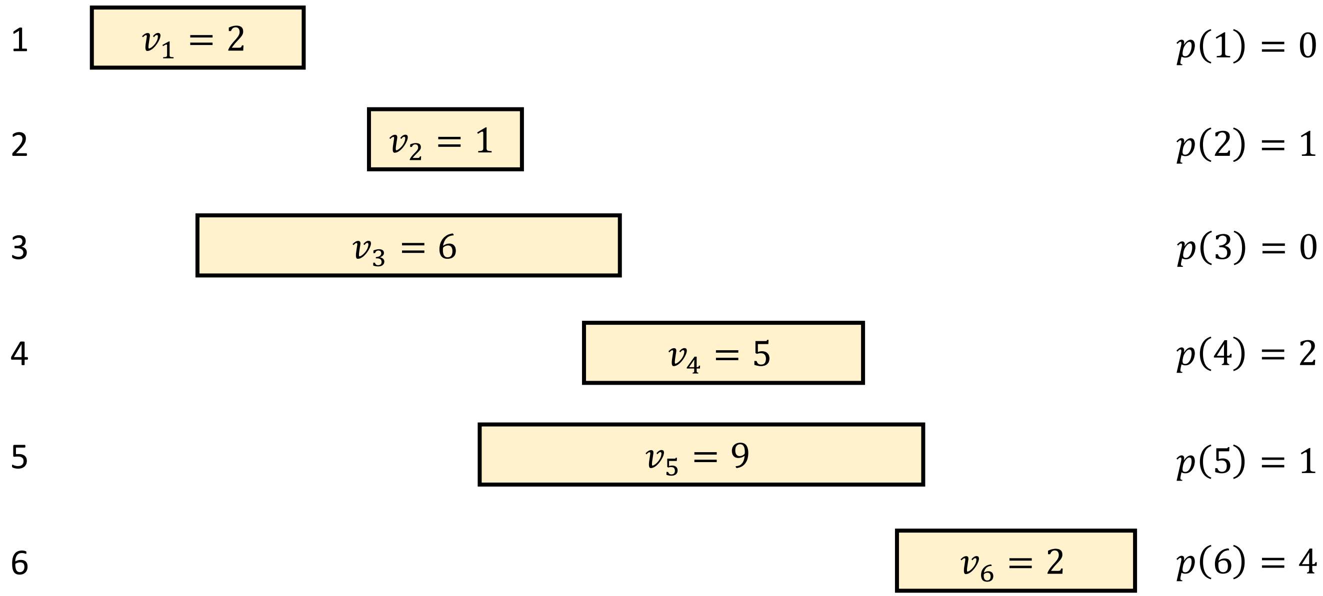
M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

How much space is used?



M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

Now You Try



M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]

Dynamic Programming Recap

- Express the optimal solution as a **recurrence**
  - Identify a small number of subproblems
  - Relate the optimal solution on subproblems

- Efficiently solve for the **value** of the optimum
  - Simple implementation is exponential time, but top-down and bottom-up are linear time
    - Top-Down: recursive, store solution to subproblems
    - Bottom-Up: iterate through subproblems in order

- Find the **solution** using the table of **values**