CS 7800: Advanced Algorithms

Dynamic Programming !

Class 4: ~~Greedy Algorithms II~~

- Fibonacci Numbers
- Weighted Interval Scheduling

Jonathan Ullman

September 16, 2025

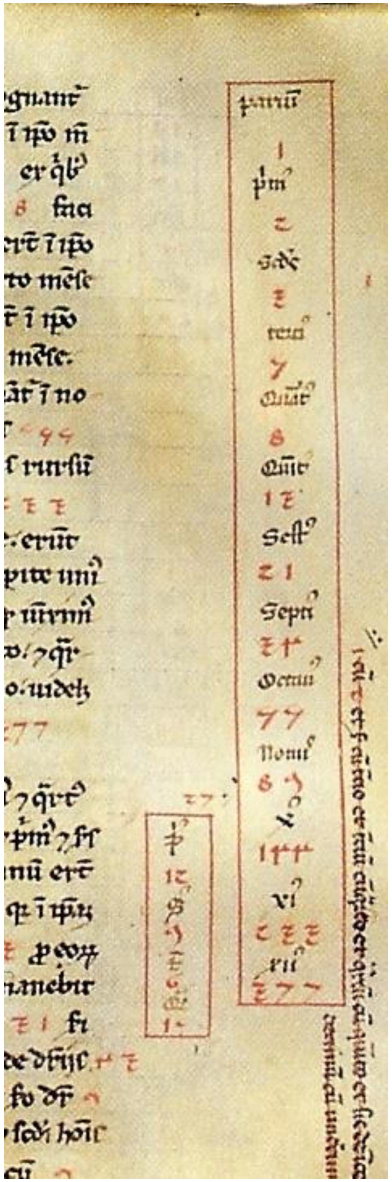
Fibonacci Numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

- $F(1) = 0$
- $F(2) = 1$
- $\boxed{F(n) = F(n-1) + F(n-2)}$

Defined by a recursive algorithm
"recurrence"

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

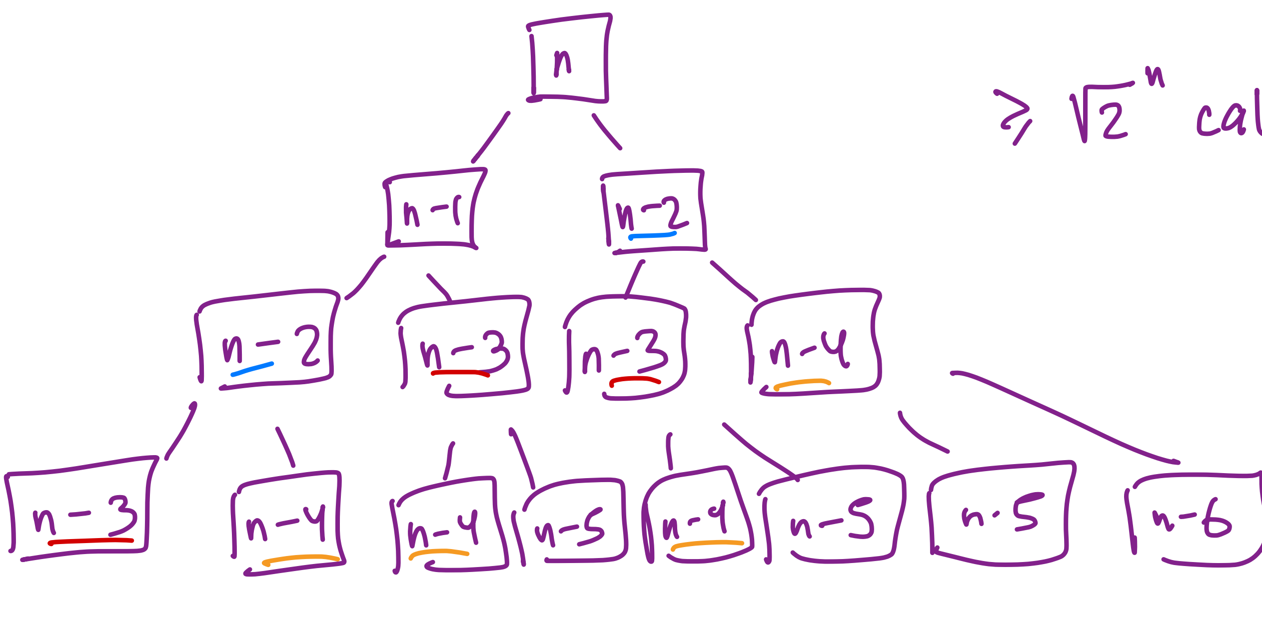


Fibonacci's
Liber Abaci (1202)

Fibonacci Numbers I

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

What is the running time of **`FibI`**?



$\geq \sqrt{2}^n$ calls $\approx 1.41^n$

Fibonacci Numbers II ("Top Down")

"Memoization"

```
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]
```

What is the running time of **FibII**?

- $O(1)$ in each call, excluding time in recursive calls

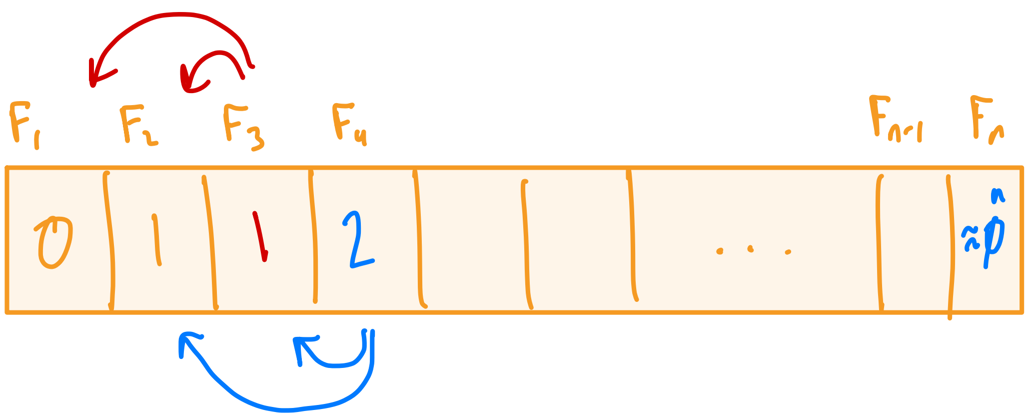
X - Total # of calls is at most $\underbrace{2}_{\text{2 calls per array elt}}\underbrace{(n-2)}_{\text{n-2 array entries filled}}$

- $O(n)$ time overall

Fibonacci Numbers III (“Bottom Up”)

```
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 running time of **`FibIII`**? $O(n)$ time



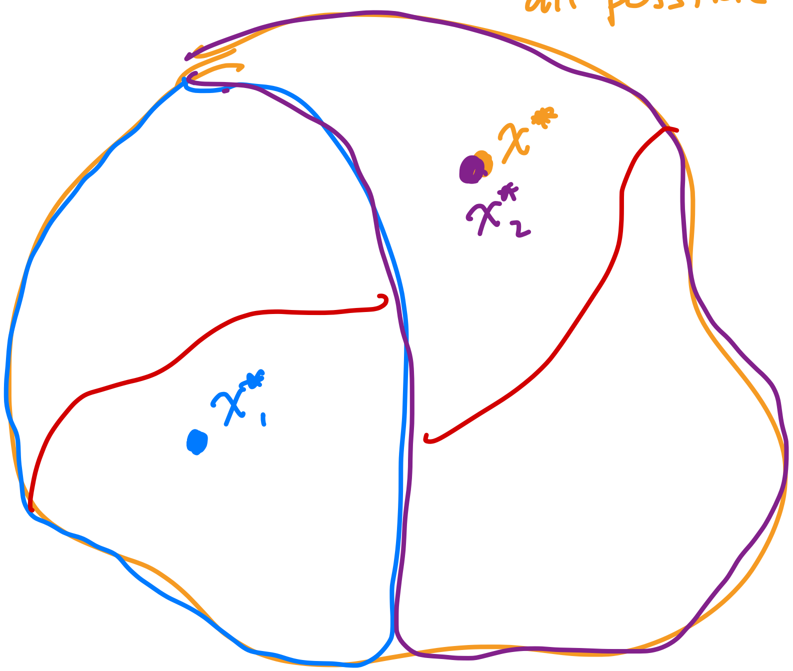
Fibonacci Numbers Recap

- Can compute $F(n)$ in $O(n)$ time*

e.g. write an interval scheduling prob of size n in terms of a small number of smaller problems

- $\boxed{\text{F(n) is defined as a recursive function}}$
  - Reduces F(n) to a small number of subproblems
  - Naively solving the recurrence is slooooow
  - Can cleverly avoid solving subproblems twice

OK, so what is dynamic programming?



$\boxed{\text{find } x \in \mathcal{X} \text{ maximizing } f(x)}$

split $\mathcal{X}$ into $\mathcal{X}_1, \mathcal{X}_2$

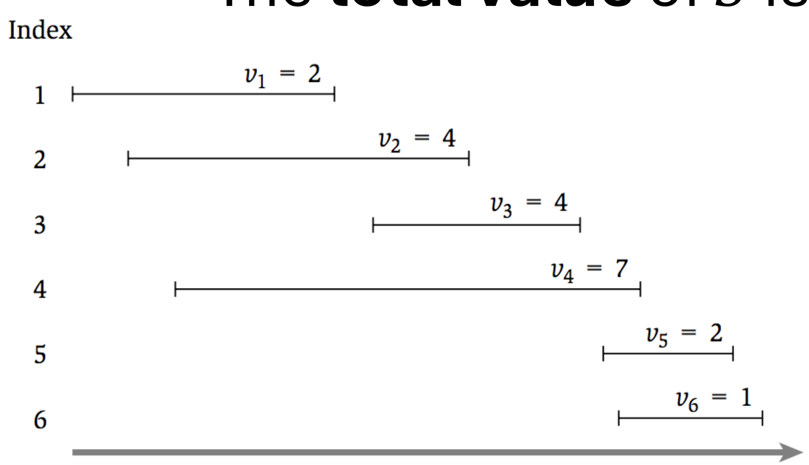
$$\max_{x \in \mathcal{X}} f(x) = \max\left\{\max_{x_1 \in \mathcal{X}_1} f(x_1), \max_{x_2 \in \mathcal{X}_2} f(x_2)\right\}$$

Suggests a recursive algorithm:

1. Optimizing over $\mathcal{X}_1, \mathcal{X}_2$ should be an instance of the same problem
2. $\mathcal{X}_1, \mathcal{X}_2$ are from a small set of subproblems

Weighted Interval Scheduling

- **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 $S \subseteq \{1, \ldots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$



Finding the Recurrence

Warm-up: just find the value of the optimal schedule

Idea: Split all solutions $\mathcal{X}$ into

$\mathcal{X}_1 = \{$all solutions not incl. $n\}$

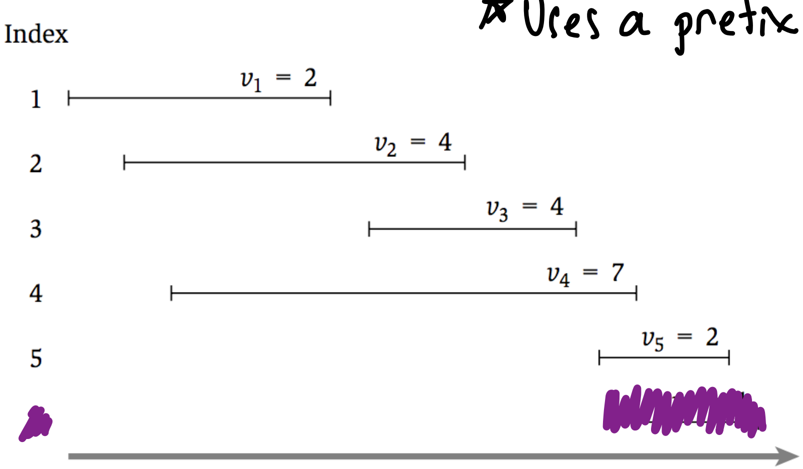
$\mathcal{X}_2 = \{$all solutions incl. $n\}$

Case 1: solutions not including n

$\mathcal{X}_1 = \{$all compatible schedules using $1, \ldots, n-1\}$

★ A smaller WIS problem ("subproblem")

★ Uses a prefix of the intervals



Finding the Recurrence

Warm-up: just find the value of the optimal schedule

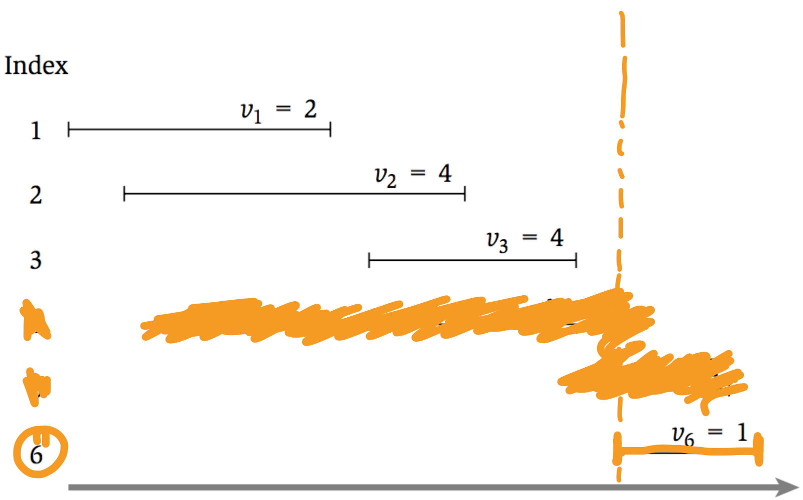
Idea: Split all solutions $\mathcal{X}$ into $\mathcal{X}_1 = \{\text{all solutions not incl. } n\}$

$\mathcal{X}_2 = \{\text{all solutions incl. } n\}$

Case 2: Solutions including n

$\mathcal{X}_2 = \{$ all schedules of the form $\{6\} \cup \{$a compatible schedule among $1,2,3\}\}$

$= \{$ all schedules of the form $\{n\} \cup \{$compatible schedule among $1, \ldots, p_n\}\}$



Let p_i be the last interval that finishes before i starts

- $p_6 = 3$

Finding the Recurrence

Warm-up: just find the value of the optimal schedule

Idea: Split all solutions $\mathcal{X}$ into

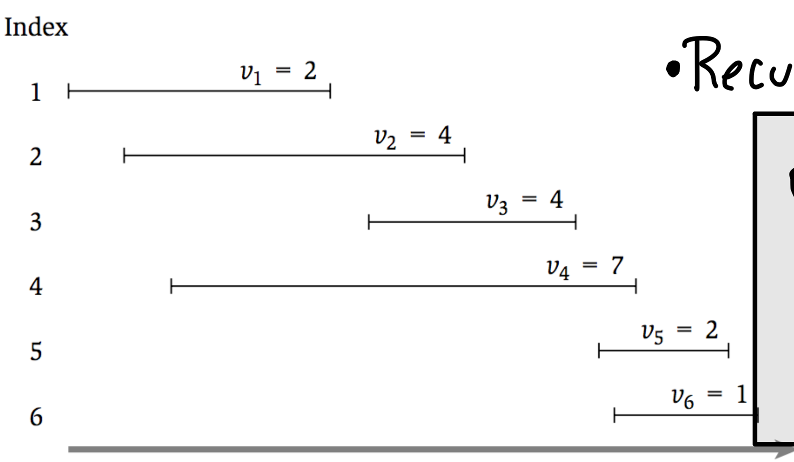
$\mathcal{X}_1 = \{$all solutions not incl. $n\}$

$\mathcal{X}_2 = \{$all solutions incl. $n\}$

- Subproblems: "Solve WIS on a prefix of the intervals"

$$\boxed{OPT(i) = \text{the value of the optimal schedule on intervals } 1, \ldots, i}$$

- Goal: Compute $OPT(n)$



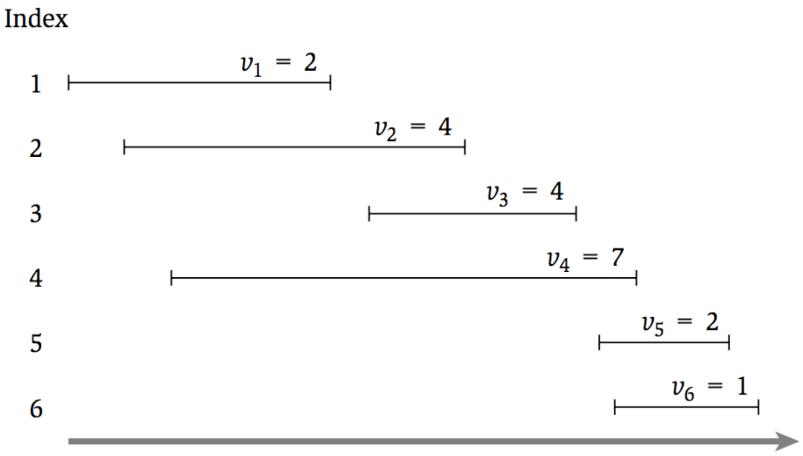
- Recurrence:

$$OPT(n) = \max\{OPT(n-1),\ v_n + OPT(p_n)\}$$

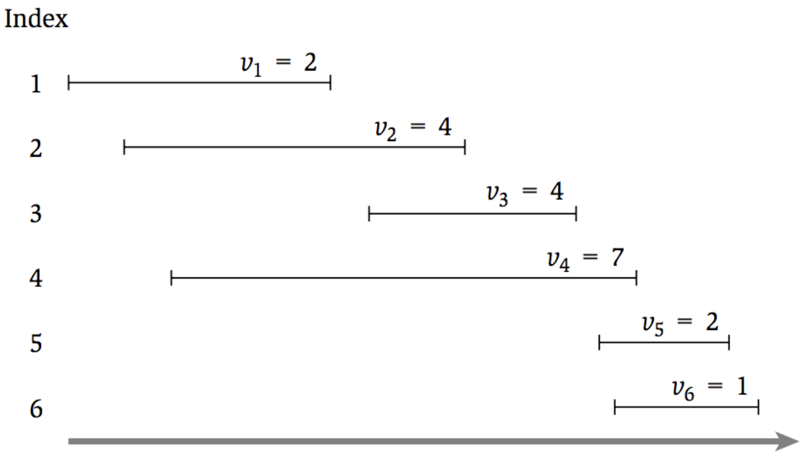
$$OPT(0) = 0$$

$$OPT(1) = v_1$$

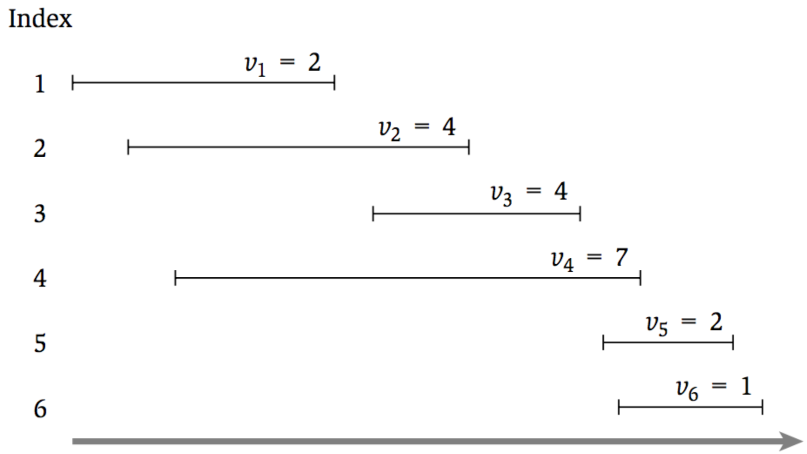
Finding the Recurrence



Finding the Recurrence



Finding the Recurrence



Interval Scheduling I

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

only difference with Fibonacci

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

can be as big as 2^n

Interval Scheduling II (Top Down)

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

What is the running time of **FindValueII(n)**?

$O(1)$ time per call, excluding recursive calls

fill n-1 values

$\times$ 2 call per value

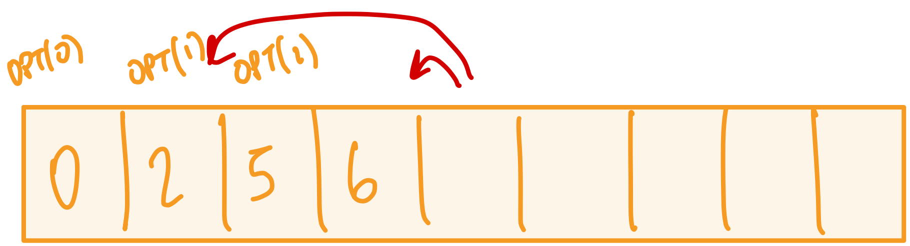
$2(n-1)$ recursive calls

$= O(n)$ time total

Interval Scheduling III (Bottom Up)

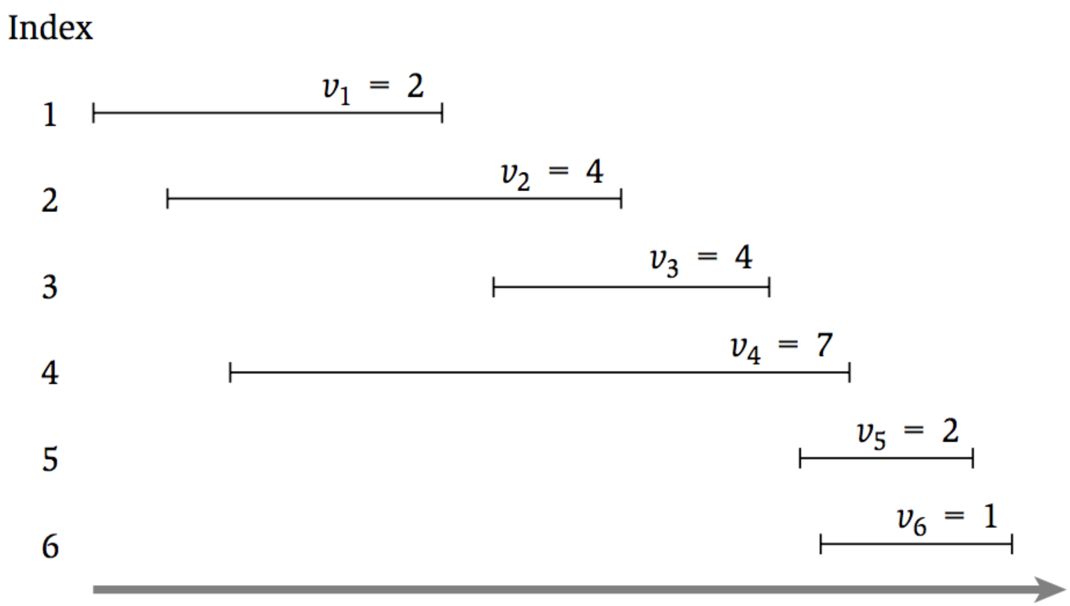
```
// All inputs are global vars
FindValIII (n):
  M[0] ← 0, M[1] ← v₁
  for (i = 2,…,n):
    M[i] ← max{M[i-1], vᵢ + M[pᵢ]}
  return M[n]
```

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



O(n) time

Interval Scheduling III (Bottom Up)

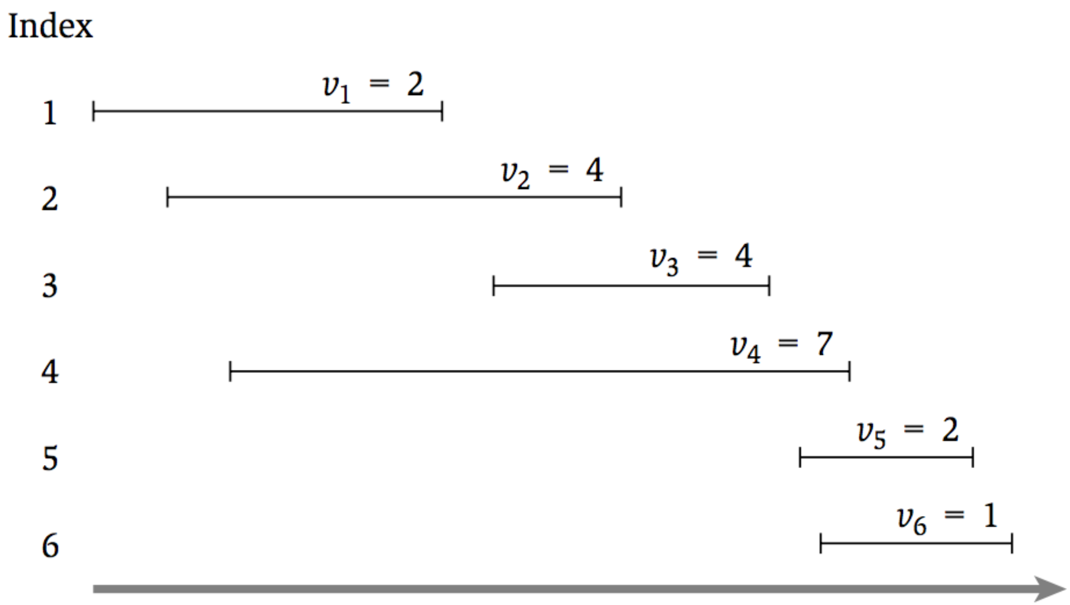


Fill in the table OPT[0] OPT[1] OPT[2] ... OPT[n]

for this small instance

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

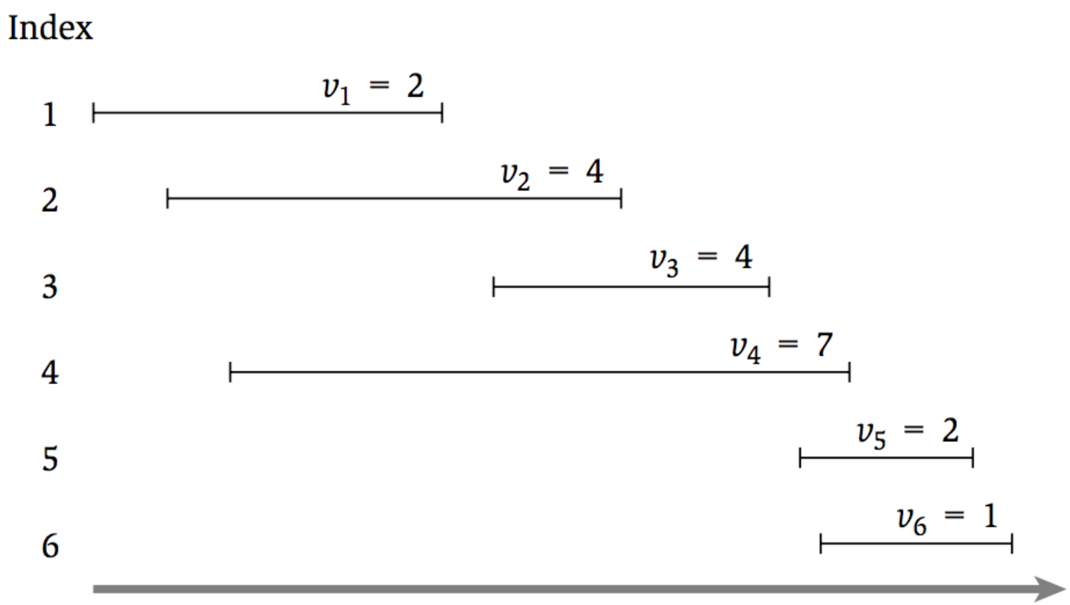
Finding the Optimal Solution

Finding the Optimal Solution

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

What is the running time of **`FindOPT (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

Weighted Interval Scheduling Recap

- There is an $O(n \log n)$ algorithm for the weighted interval scheduling problem
  - Generalizes the greedy alg for the unweighted version
  - Our first example of dynamic programming

- **Dynamic Programming Recipe:**

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

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

  (3) design an algorithm to **efficiently solve** the recurrence

  (4) if needed, recover the **actual solution** at the end