Divide and Conquer: Recap

- Handle base case with small inputs
- Divide problem into smaller part(s)
  - May require careful thought, take time in the algorithm
- Recurse in appropriate smaller part(s)
- Combine the solutions returned in recursive calls
  - May require careful thought, take time in the algorithm
- Prove correctness, often using induction
- Establish recurrence relation for running time
- Solve recurrence using one of:
  - Master Theorem
  - Recursion Tree
  - Formulate a conjecture and prove by induction

CS5800: Algorithms

Dynamic Programming
a. Fibonacci Series

Fibonacci Numbers

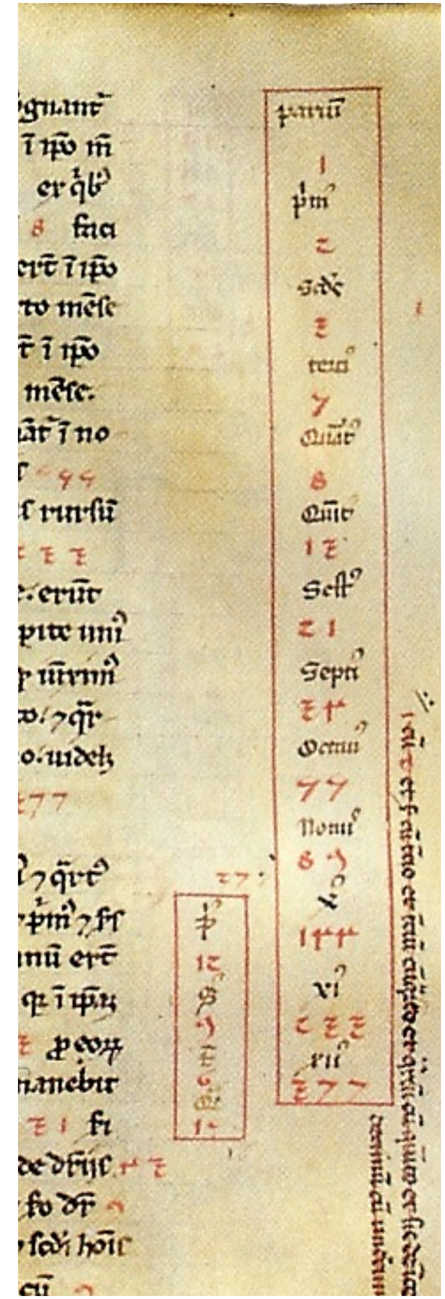
F(1) F(2)

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

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

- $F(n) \to \phi^n \approx 1.62^n$ asymptotically

↑ out of the scope of this course



Fibonacci's *Liber Abaci* (1202)

Fibonacci Numbers: Take I

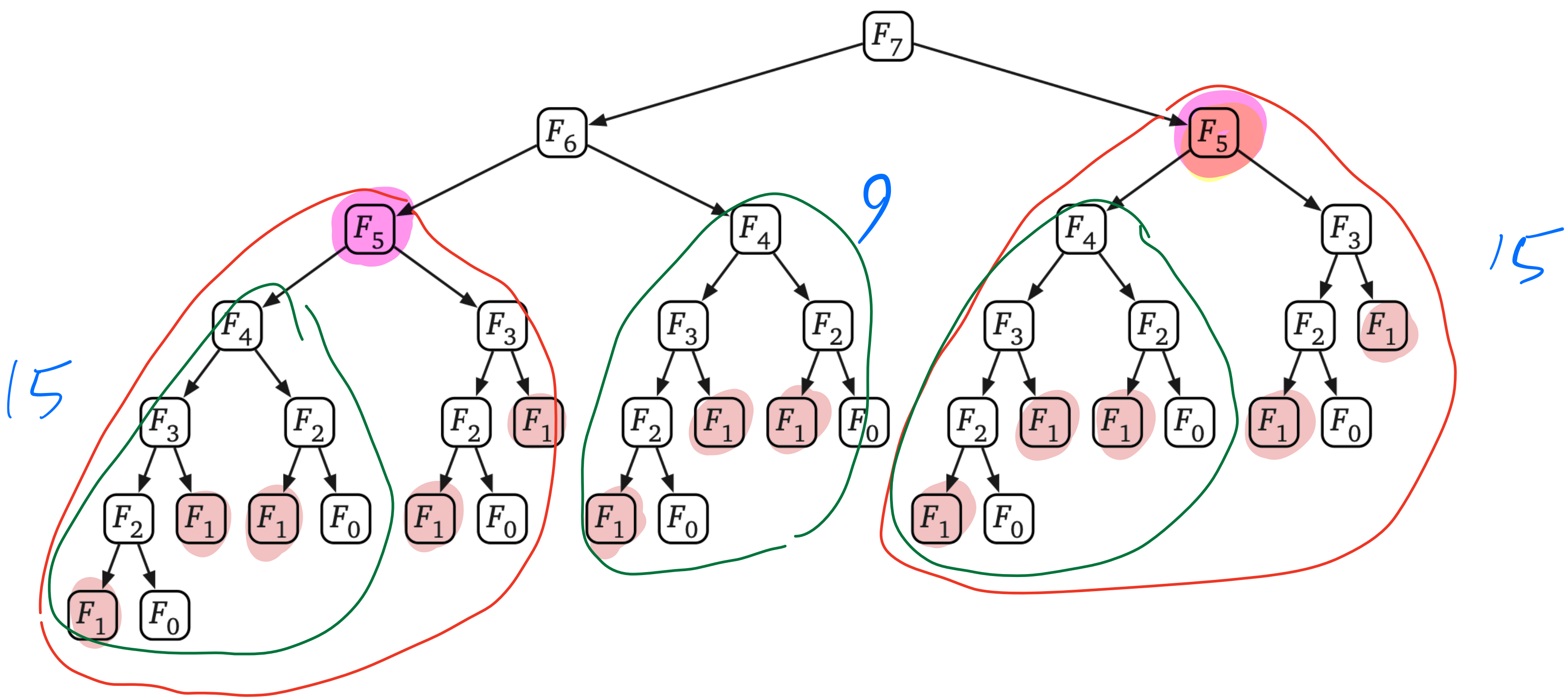
```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

Fibonacci Numbers: Take I

T(5) = 15

T(7) = 41

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```



Fibonacci Numbers: Take I

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```

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

Let $T(n)$ = total # of rec calls to RecFibo when RecFibo(n) called

$T(n) = T(n-1) + T(n-2) + 1$ $T(0) = 1$ $T(1) = 1$

F: 0 1 1 2 3 5 8 13 21

T: 1 1 3 5 9 15 25 41

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

$\Rightarrow T(n) = \Theta(1.62^n)$

Fibonacci Numbers: Memo(r)ization

def F[n]

```
MemFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Base cases

global array

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

Fibonacci Numbers: Memo(r)ization

claim: every element F[i]

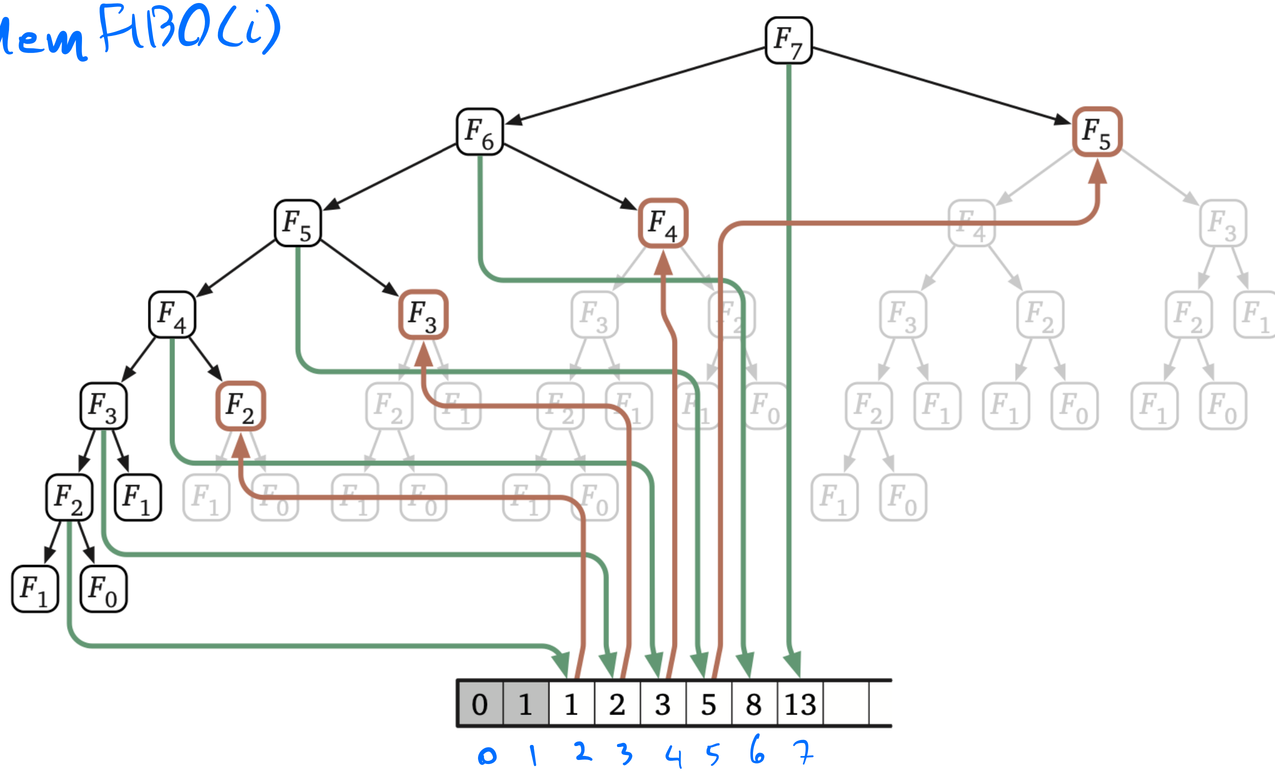
is accessed at most 3 times.

F[i] is accessed by the first calls

to MemFIBO(i+1), MemFIBO(i+2)

MemFIBO(i)

```
MEMFIBO(n):                                         n=7
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined                 n=6            n=5
          F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
      return F[n]
```



Fibonacci Numbers: Bottom up

<u>ITERFIBO(n):</u>

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

O(1)

for $i \leftarrow 2$ to n

$F[i] \leftarrow F[i-1] + F[i-2]$

O(n)

return $F[n]$

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

Fibonacci Numbers

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

- 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 → $O(n)$
  - Remember values you've already computed ("top down") ← Memoization
  - Iterate over all values $F(i)$ ("bottom up") ← Dynamic Programming

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

Dynamic Programming Recipe

- **Recipe:**

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

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

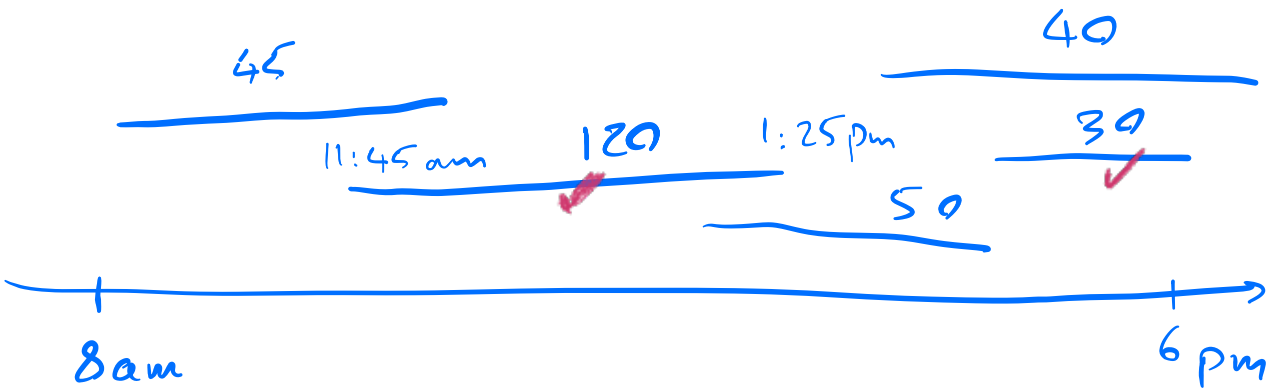
  (3) find an **efficient implementation** of the recurrence (top down or bottom up)

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

Dynamic Programming

a. Fibonacci Series

b. Weighted Interval Scheduling



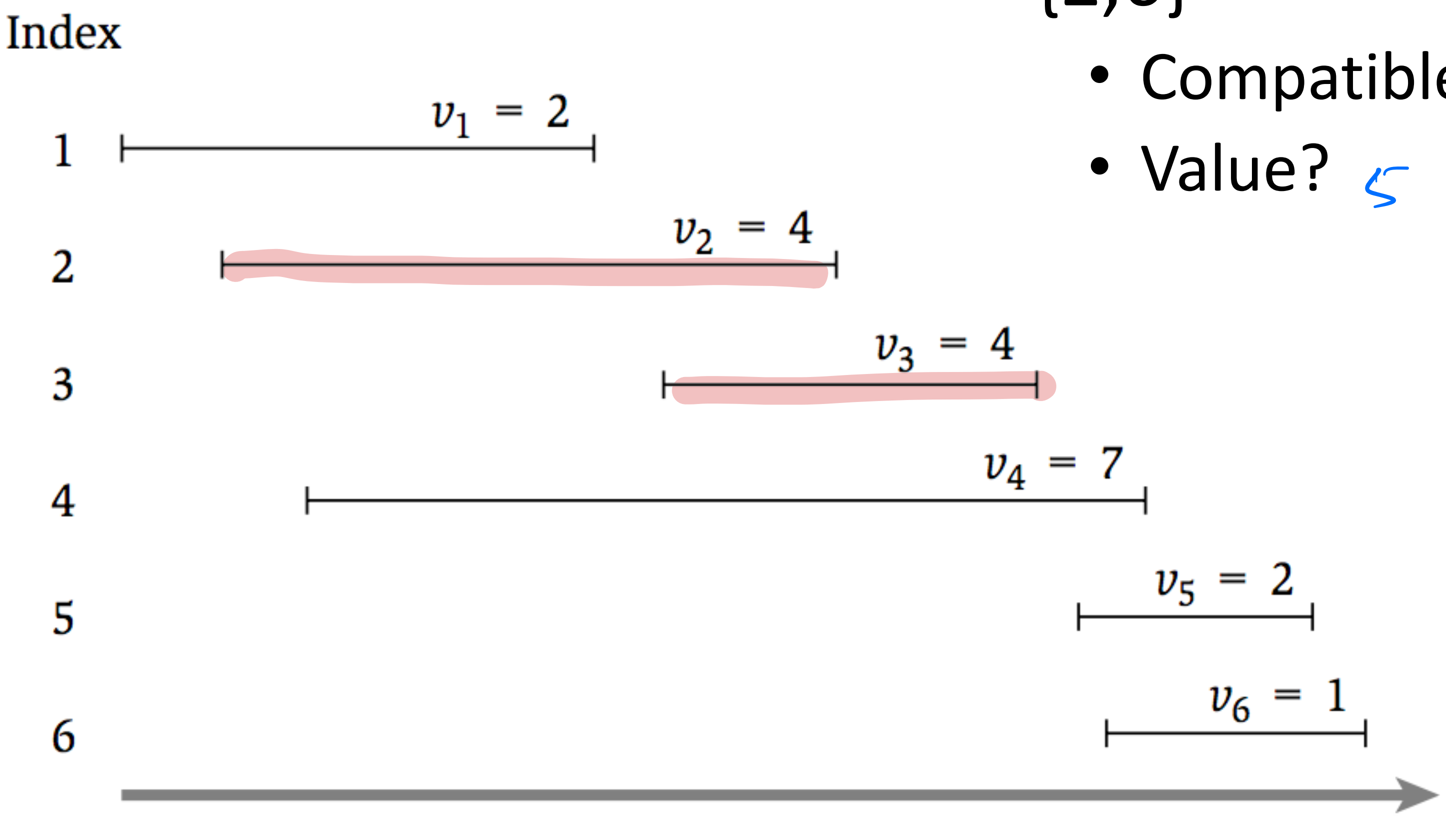
Weighted Interval Scheduling

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …

start of ith interval — finish time of interval i

- **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 $\sum_{i \in S} v_i$

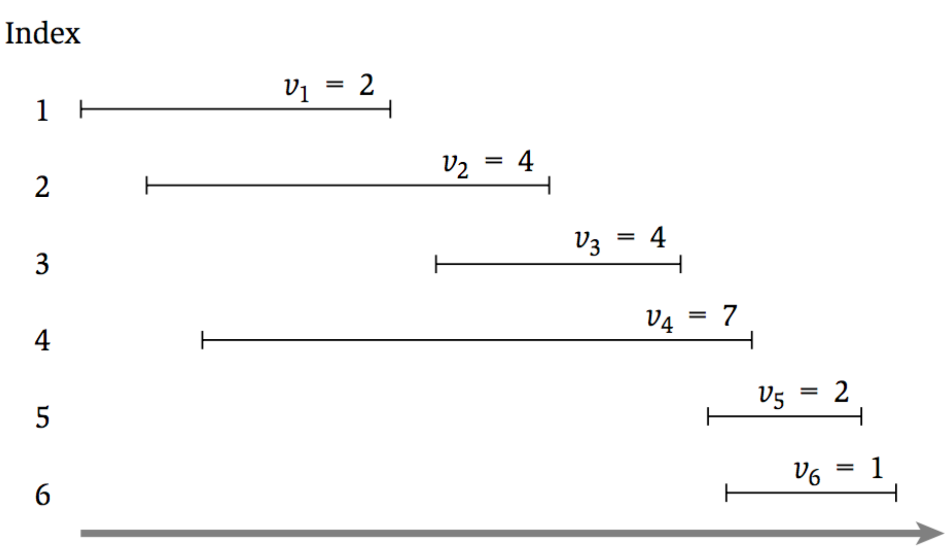
Interval Scheduling



- {2,3}
  - Compatible? NO
  - Value? 8
- {2,6}
  - Compatible? YES
  - Value? 5

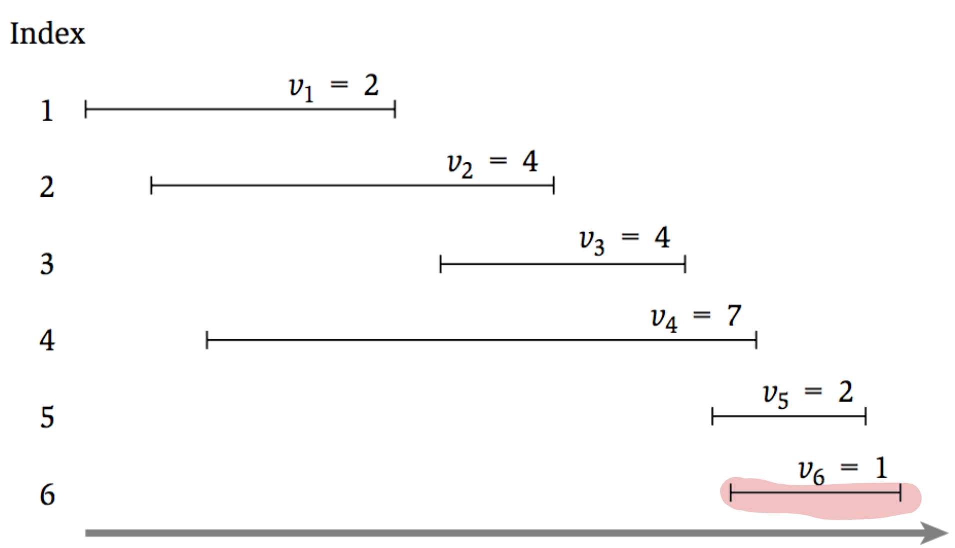
A Recursive Formulation

- 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

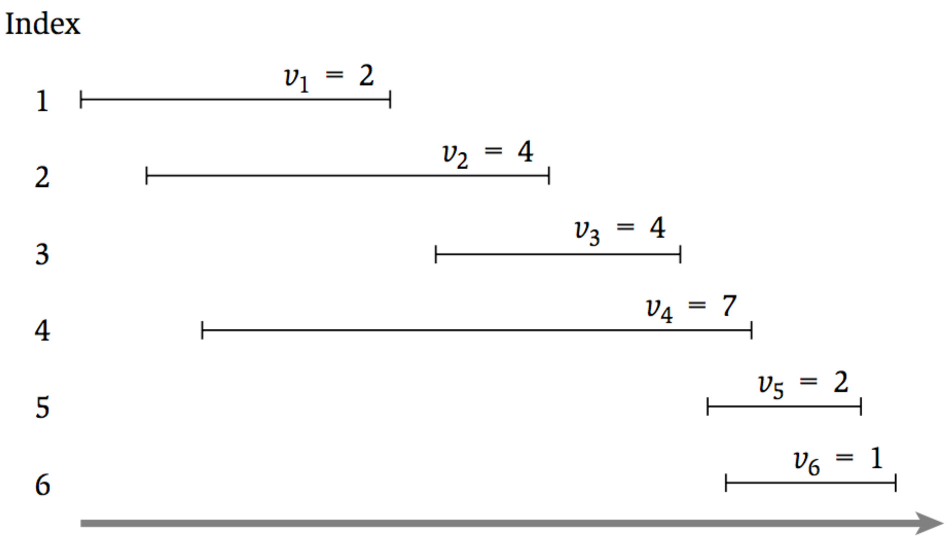
- Let O be the **optimal** schedule
- **Case 2:** Final interval is in O (i.e. $6 \in O$)
  - Then O must be $\{6\}$ + the optimal solution for $\{1, \ldots, 3\}$



A Recursive Formulation

Which is better?

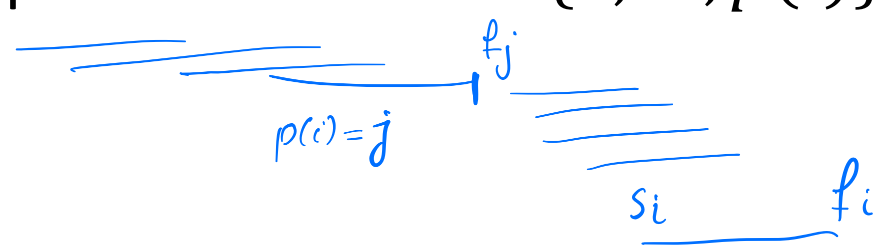
- the optimal solution for $\{1, \ldots, 5\}$
- $\{6\}$ + the optimal solution for $\{1, \ldots, 3\}$



A Recursive Formulation: Subproblems

Final solution = O_n

- ***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 $f_1 < f_2 < \cdots < f_n$
  - Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O_i must be i + the optimal solution for $\{1, \ldots, p(i)\}$
    - $O_i = \{i\} + O_{p(i)}$



A Recursive Formulation: Subproblems & Recurrence

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

- **Case 1:** Final interval is not in O_i $(i \notin O_i)$
  - Then O_i must be the optimal solution for $\{1, \dots, i-1\}$
- **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 $\{1, \dots, p(i)\}$

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

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

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

```
// All inputs are global vars
FindOPT(n):
  if (n = 0): return 0
  elseif (n = 1): return v_1
  else:
    return max{FindOPT(n-1), v_n + FindOPT(p(n))}
```

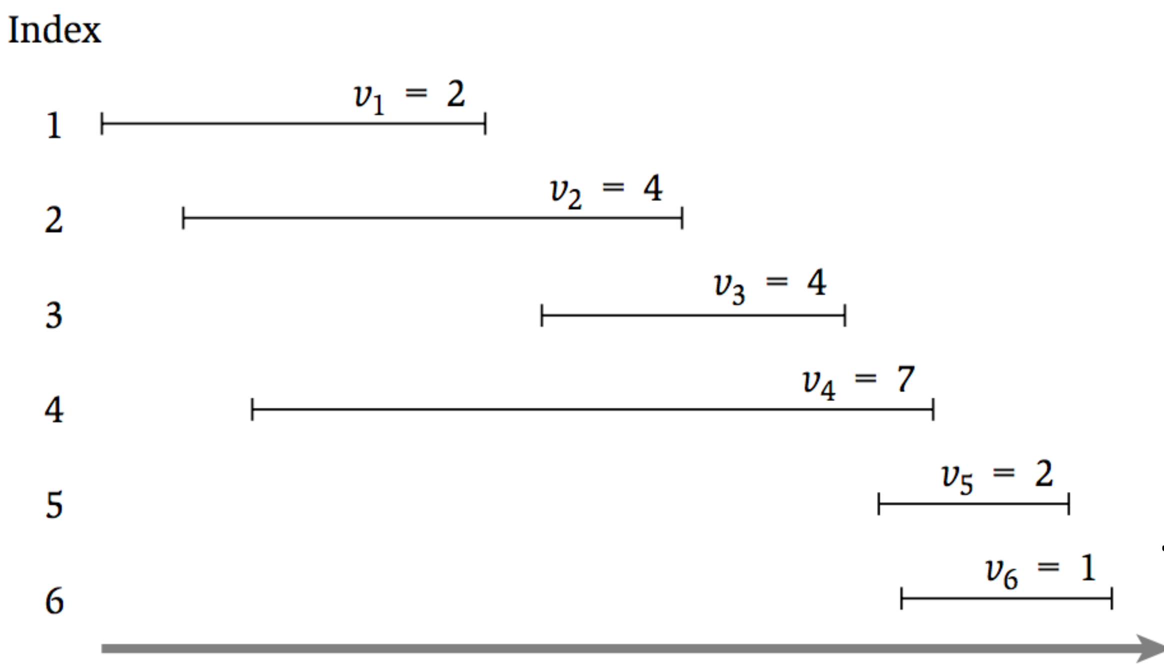
- What is the worst-case running time of **`FindOPT(n)`** (how many recursive calls)?

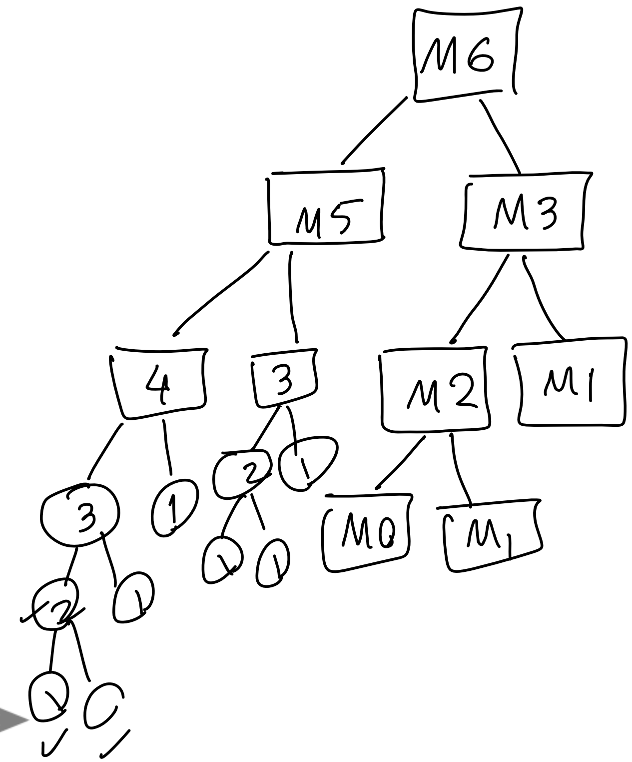
Interval Scheduling: Top Down

```
// 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)`**?

Interval Scheduling: Top Down





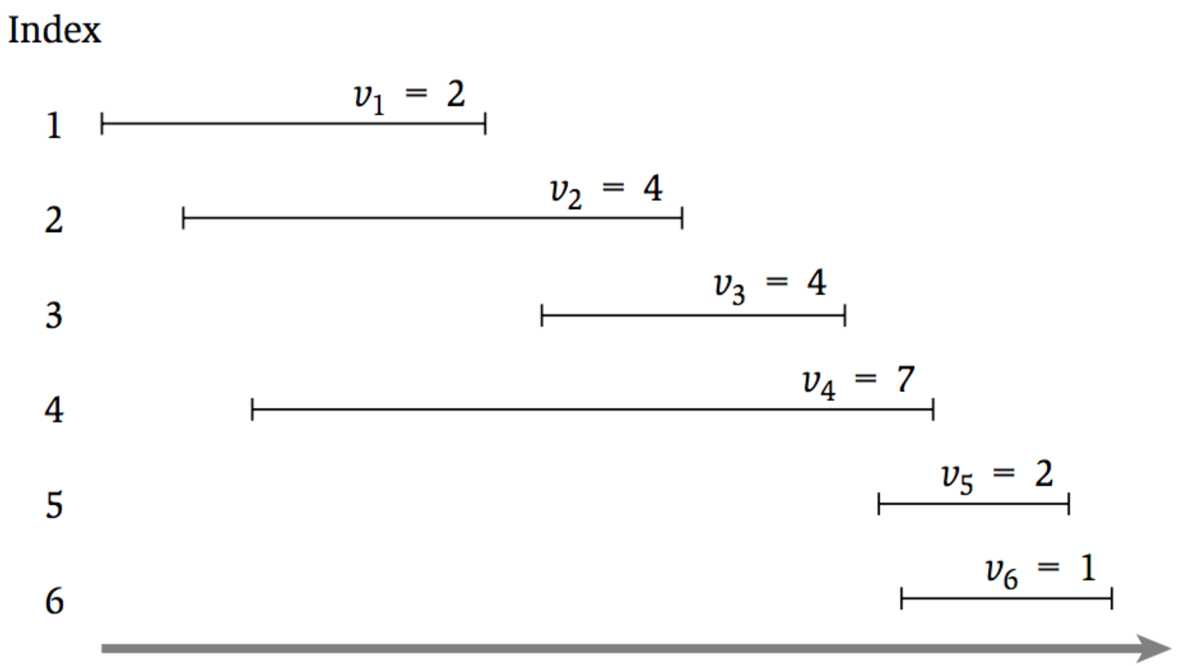
M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2					

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)`**?

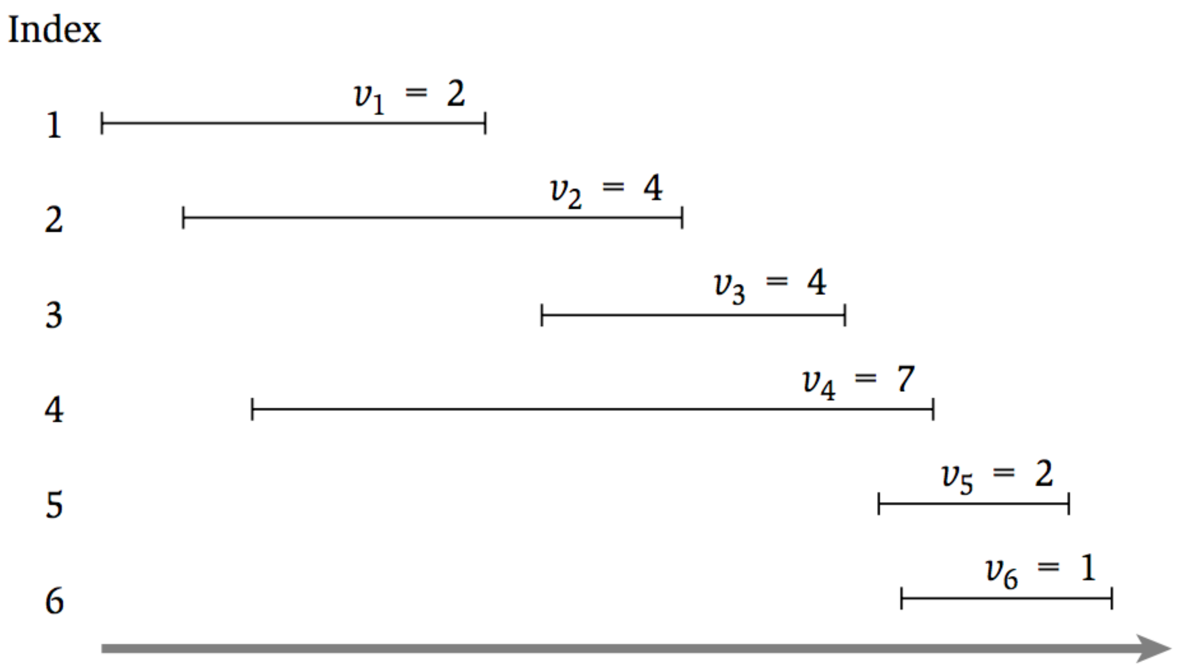
Interval Scheduling: Bottom Up



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

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)

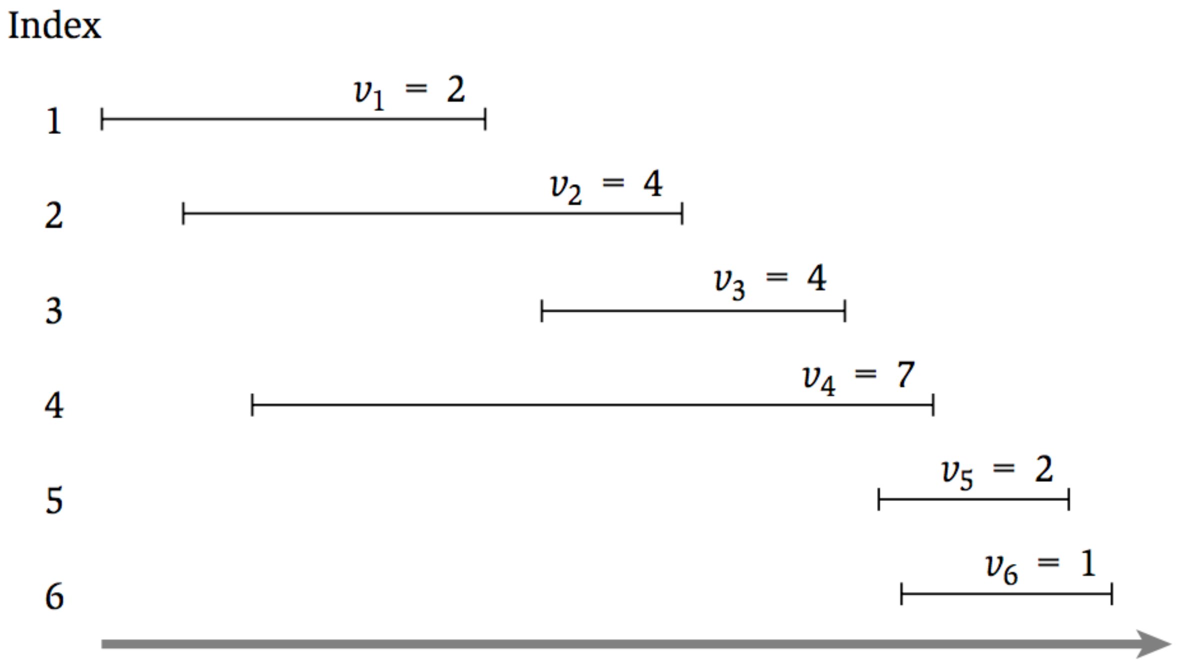
  (4) **reconstruct the solution** from the DP table

Finding the Optimal Solution

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

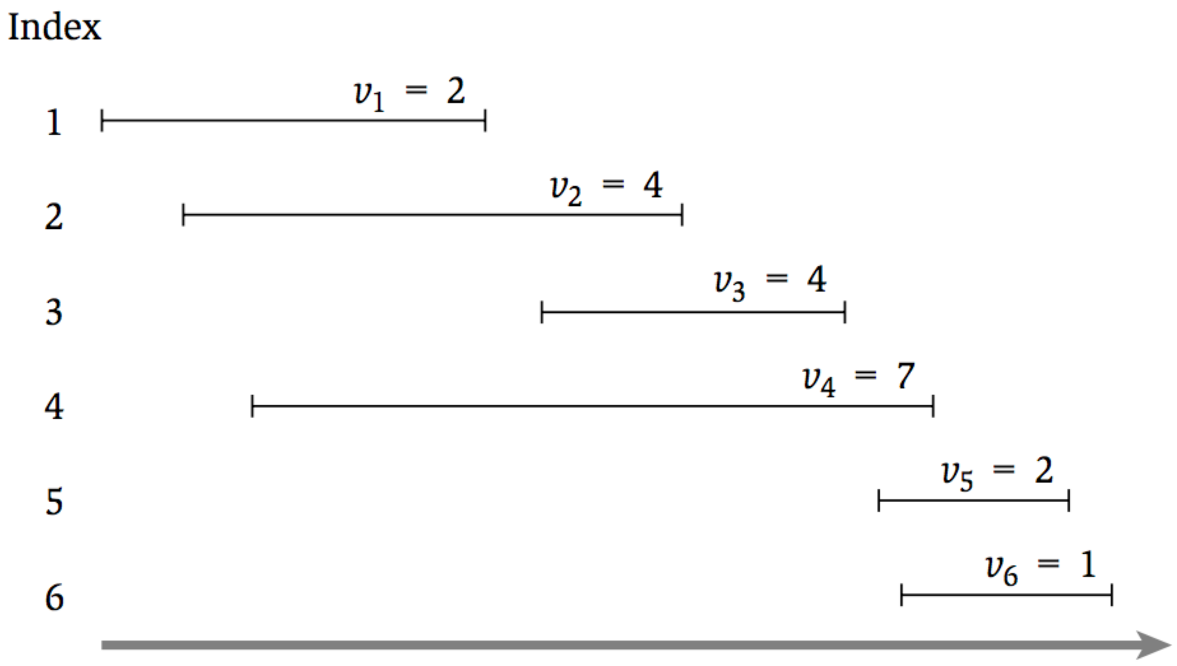
- What is the running time of **`FindSched(n)`**?

Finding the Optimal Solution



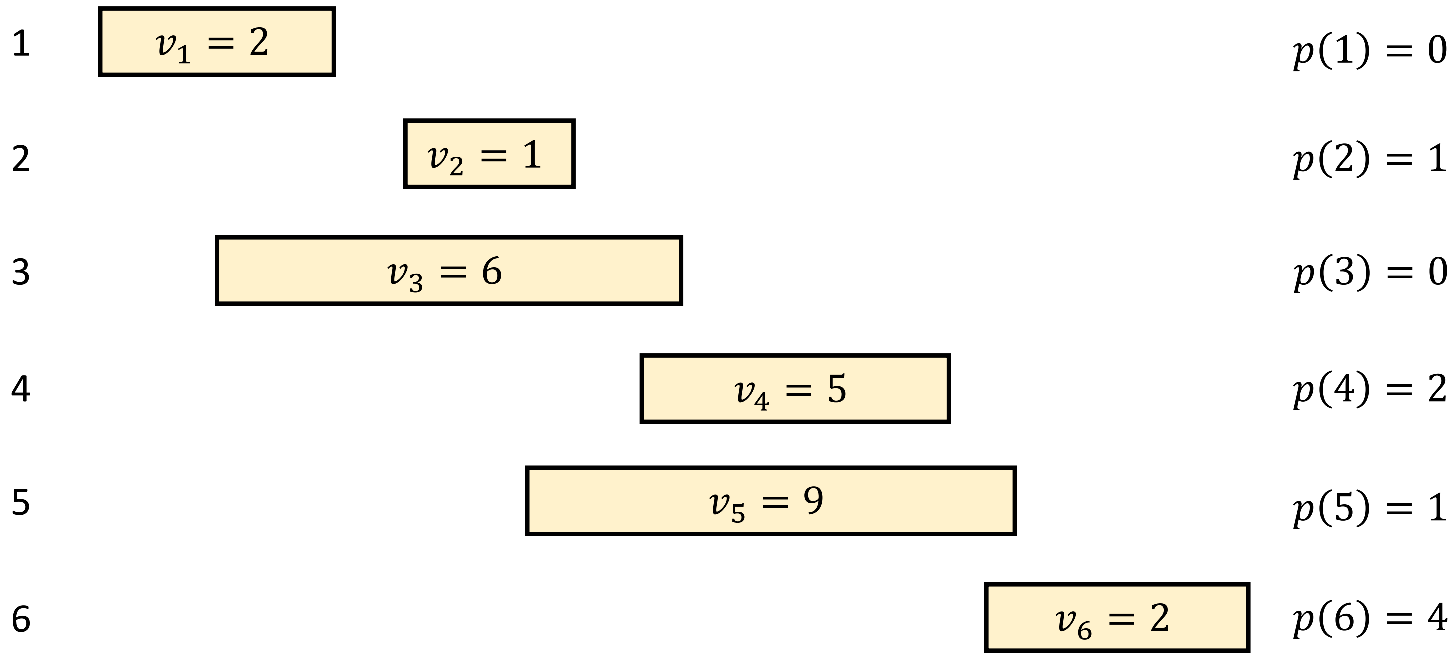
M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

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**