CS 7800: Advanced Algorithms

Class 4: Greedy Algorithms II

- Fibonacci Numbers
- Weighted Interval Scheduling

Jonathan Ullman

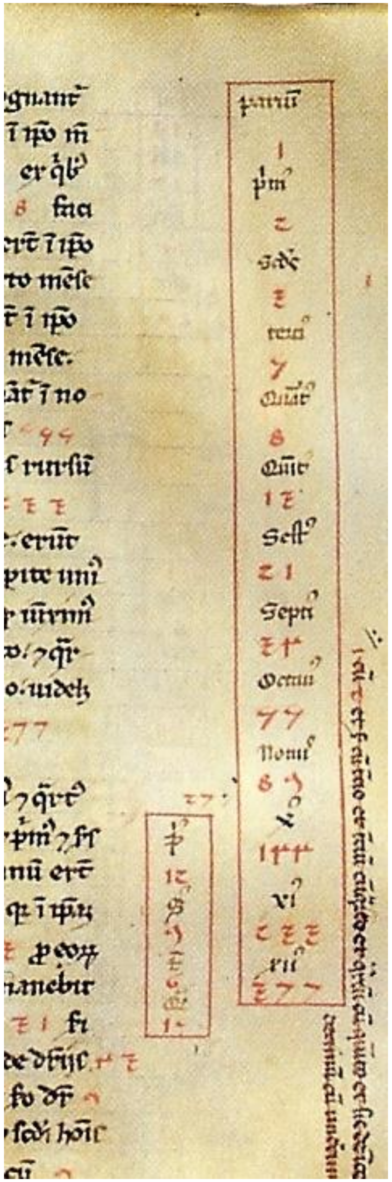
September 16, 2025

Fibonacci Numbers

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

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

- $F(n) \to \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1.62^n$ 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`**?

Fibonacci Numbers II (“Top Down”)

```
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`**?

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`**?

Fibonacci Numbers Recap

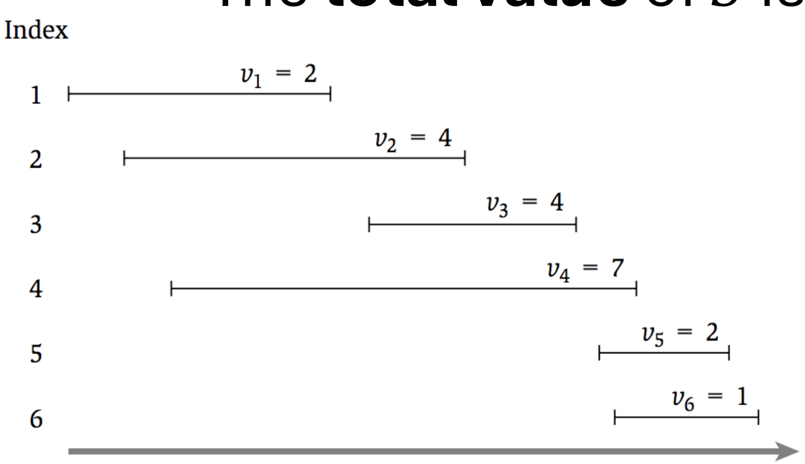
- Can compute $F(n)$ in $O(n)$ time*

- F(n) is defined as a recursive function
  - Reduces F(n) to a small number of subproblems
  - Naively solving the recurrence is slooooooow
  - Can cleverly avoid solving subproblems twice

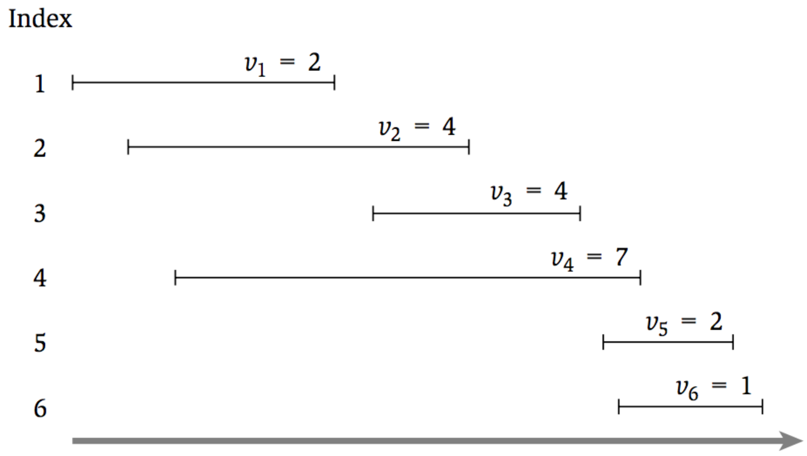
OK, so what is dynamic programming?

Weighted Interval Scheduling

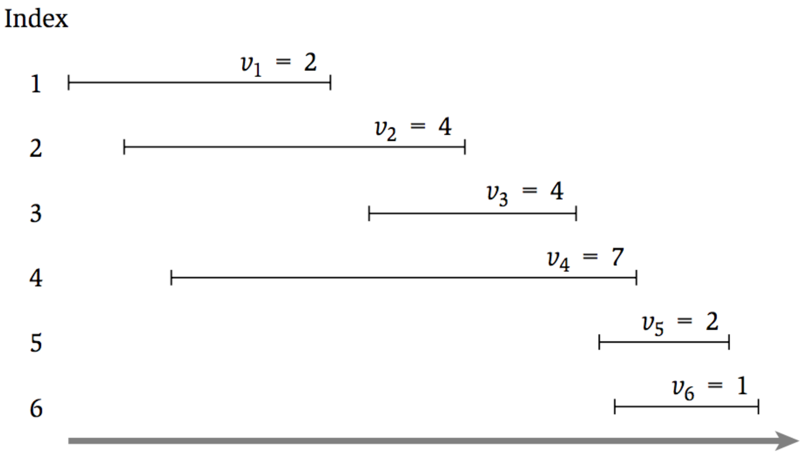
- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \ldots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$



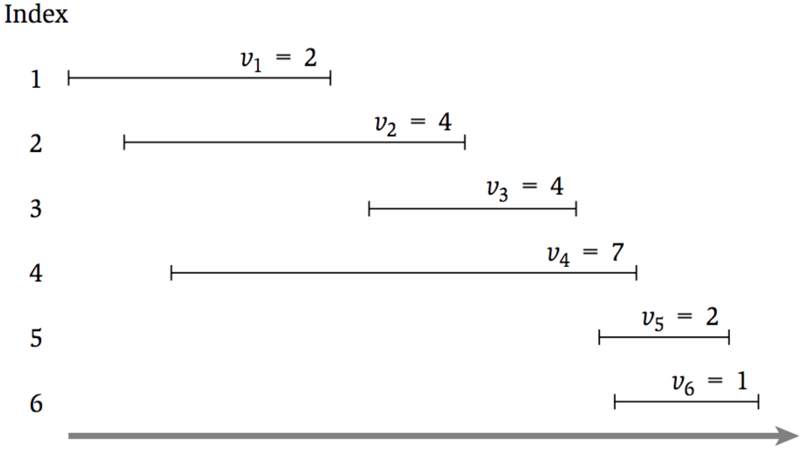
Finding the Recurrence



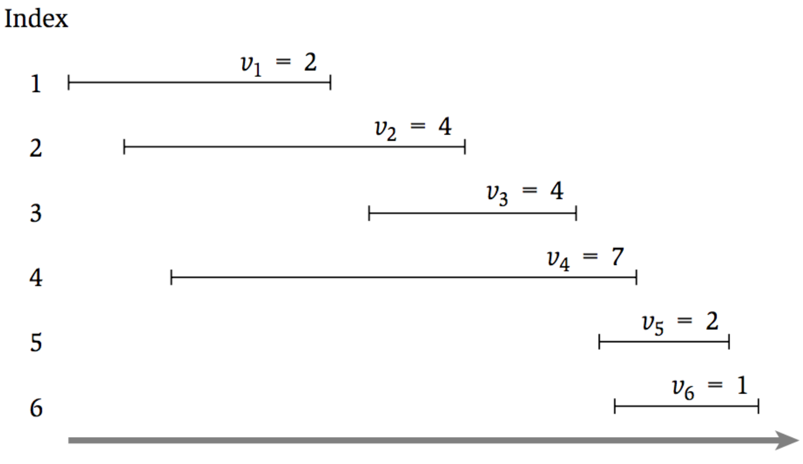
Finding the Recurrence



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 v_1
  else:
    return
    max{FindValI(n-1), v_n + FindValI(p_n)}
```

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

Interval Scheduling II (Top Down)

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← v_1
FindValII(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← max{FindValII(n-1), v_n + FindValII(p_n)}
    return M[n]
```

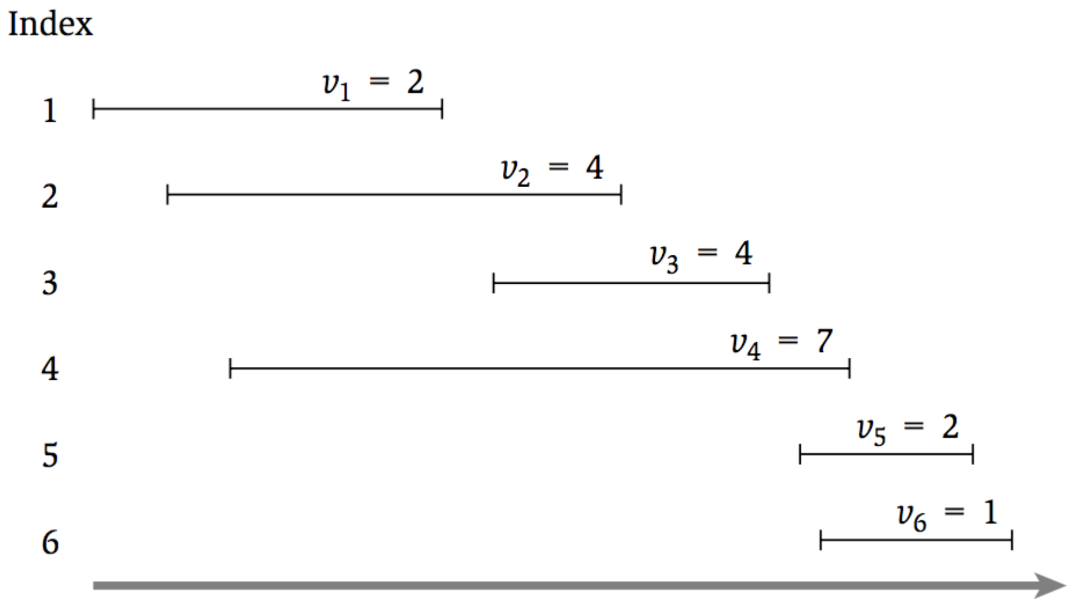
What is the running time of **`FindValueII(n)`**?

Interval Scheduling III (Bottom Up)

```
// All inputs are global vars
FindValIII (n):
  M[0] ← 0, M[1] ← v_1
  for (i = 2,…,n):
    M[i] ← max{M[i-1], v_i + M[p_i]}
  return M[n]
```

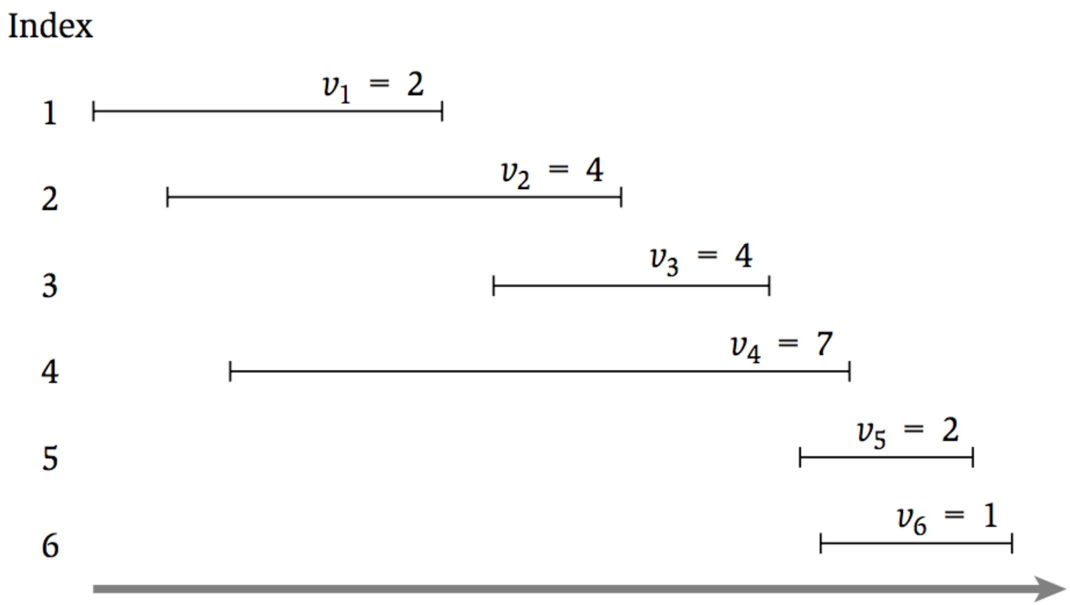
What is the running time of **`FindValueIII(n)`**?

Interval Scheduling III (Bottom Up)



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

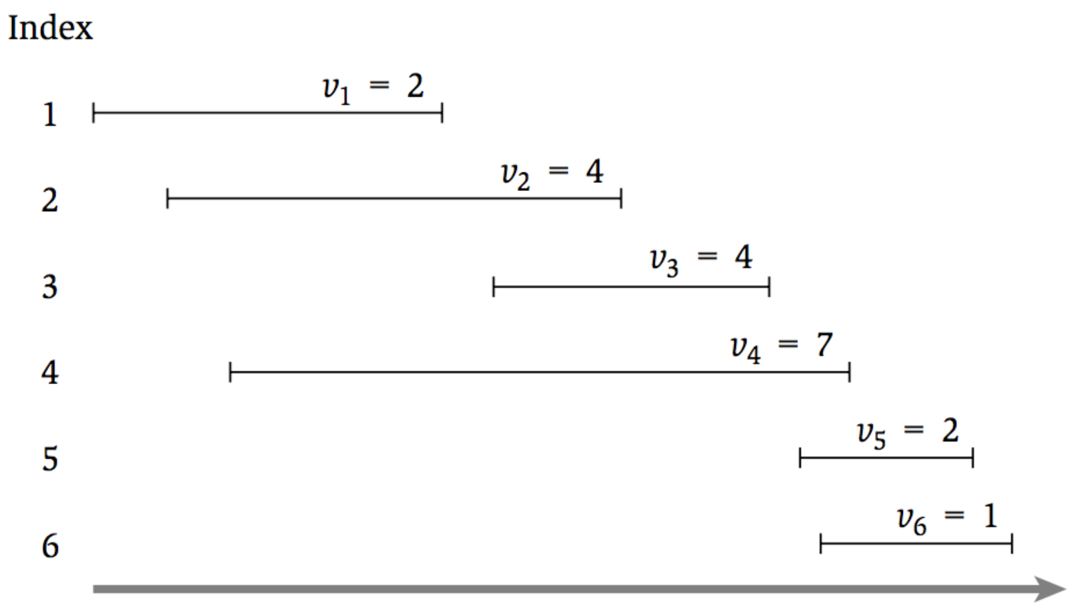
Finding the Optimal Solution

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