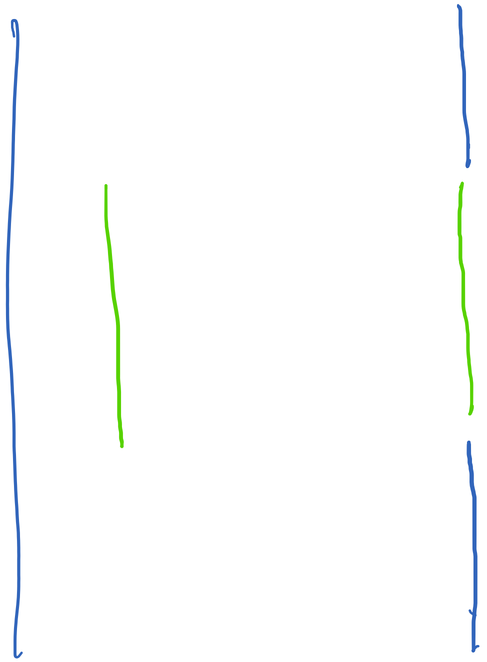


(index, x, ymin, ymax)

CS3000: Algorithms & Data
Paul Hand

Paul Hand

Lecture 8:

- Path Counting
- Dynamic Programming
- Fibonacci Numbers
- Interval Scheduling

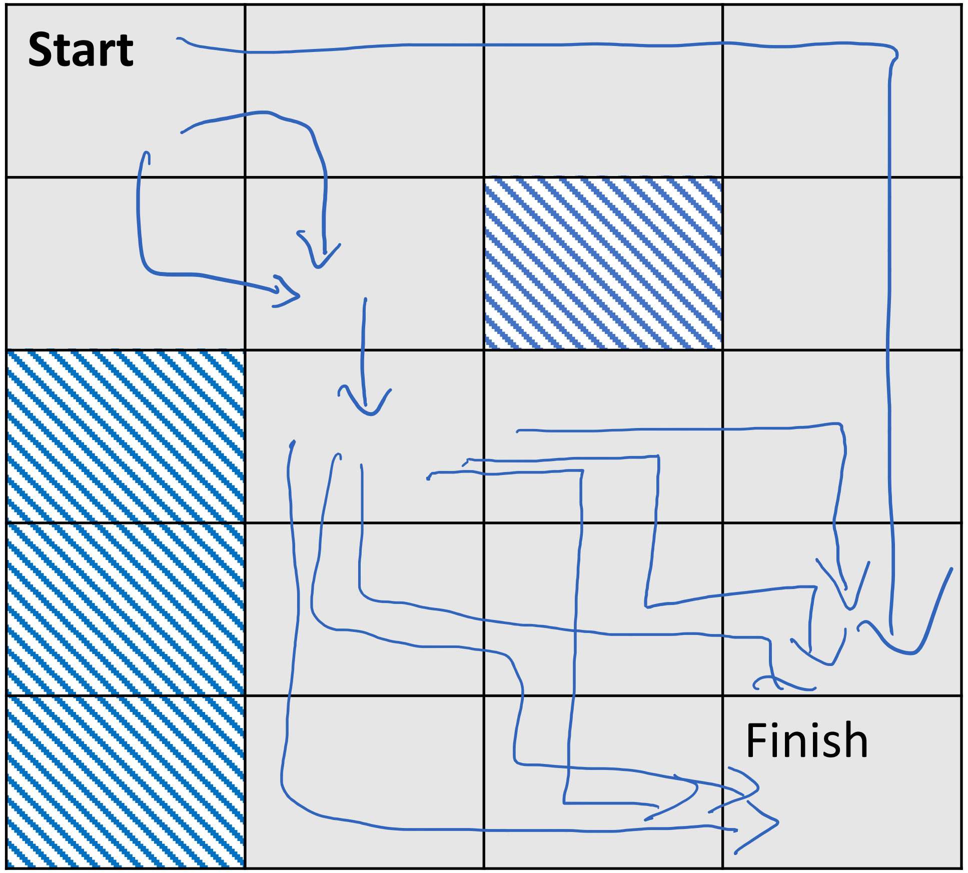
Feb 4, 2019

Warmup: Path Counting

Activity:

Agent can only move right or down. (no diagonal movement)

How many ways can it get to the finish?



1 + 2 · 6

(# of ways to get to (2,2))

(# ways to solve bottom right 3x3 block)

Activity:

Agent can only move right or down.

How many ways can it get to the finish?

Starting pt

NumPaths(X,y):

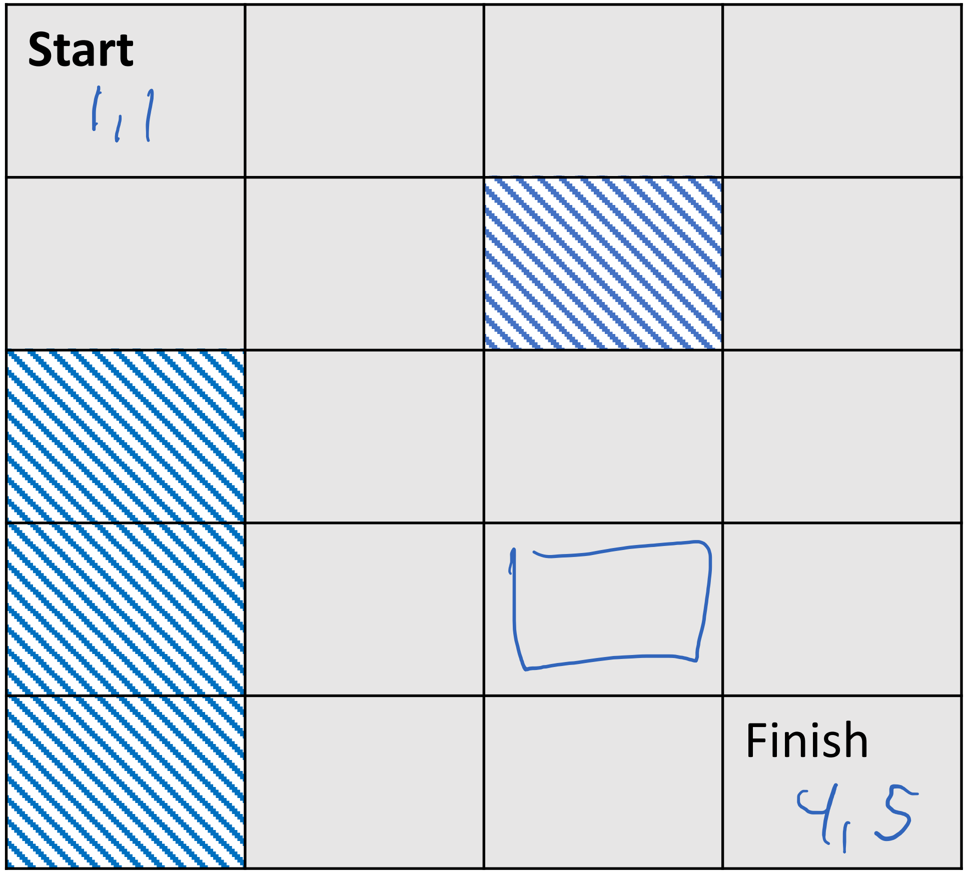
If (X,y)==(4,5) return 1

If not(valid(X,y)):

return 0

return NumPath(X+1, y)

+ NumPaths(X,y+1)



Toolkit

ValidSquare(X,y)

return T if valid

F if not

Why is this wasting resources?

Compute same thing many times!

NumPaths(4,3)

Activity:

Agent can only move right or down.

How many ways can it get to the finish?

~~Write an algorithm~~.

NumPaths(X, y)

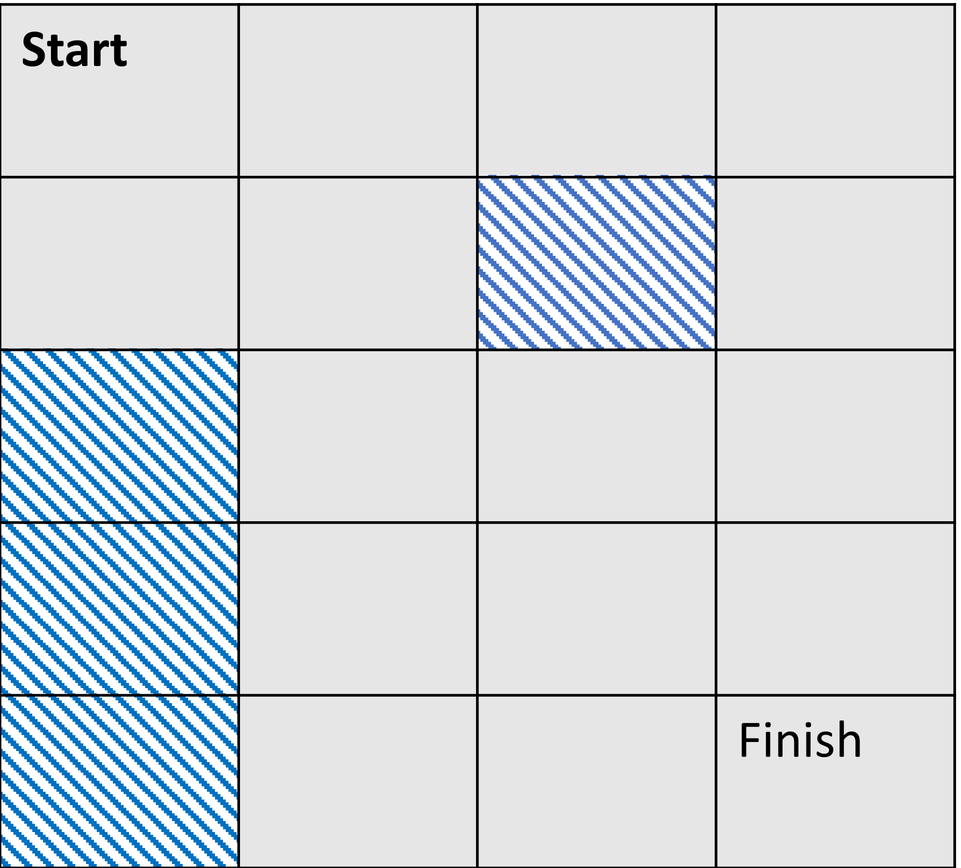
Start 5	(12) 3	(10) 1	(8) 1
(13) 2	(11) 2		(5) 1
	(9) 2	(6) 1	(3) 1
	(7) 1		(1) 1
	(4) 1	(2) 1	Finish

Activity:

Agent can only move right or down.

How many ways can it get to the finish?

Write an algorithm.



Dynamic Programming

Dynamic Programming

Dynamic programming is careful recursion

- Break the problem up into small pieces
- Recursively solve the smaller pieces
- Store outcomes of smaller pieces that get called multiple times
- **Key Challenge:** identifying the pieces

Warmup: Fibonacci Numbers

Fibonacci Numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$
- $F(n) = F(n-1) + F(n-2)$
- $F(n) \to \phi^n \approx 1.62^n$
- $\phi = \left(\frac{1+\sqrt{5}}{2}\right)$ is the golden ratio

There is an exact formula

How do we compute n^{th} Fib #.

Fibonacci Numbers: Take I

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

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

Pay for all of the calls here

recursive calls in FibI(n): X_n

$X_n = X_{n-1} + X_{n-2}$ — Same formula for Fib #

$X_n \approx 1.62^n$

exponentially slow

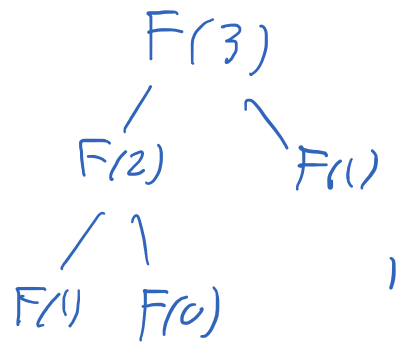
Fibonacci Numbers: Take II

"Once you've computed something, remember it"

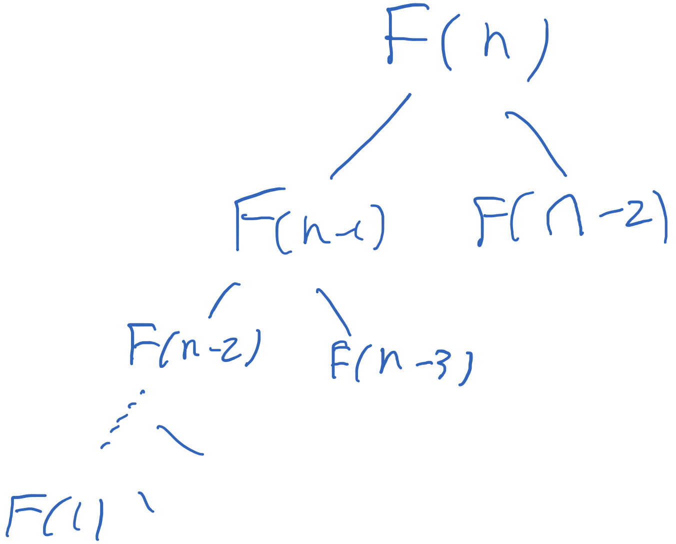
```
M ← empty array, M[0] ← 0, M[1] ← 1
FibII(n):
  If (M[n] is not empty): return M[n]
  ElseIf (M[n] is empty):
    M[n] ← FibII(n-1) + FibII(n-2)
    return M[n]
```

Store Fib #s already computed

M[n] – nth Fib #



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



n layers
2 items per layer

$\Theta(n)$ 2n NOT 1.62^n !!!

Fibonacci Numbers: Take III

build up Fib #s from smallest to largest

not recursive

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

| 0 | 1 | 1 | 2 | 3 | 5 | | | | | |

- What is the running time of **FibIII(n)**? (This is tricky)

n additions

adding two ~~digits~~ #s with k digits takes $\Theta(k)$ time

n^{th} Fib # has $\approx n$ digits

n adds, worst case n digits → n^2 ops

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 $\approx 1.62^n$ 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") ← memoized recursion
  - Iterate over all values $F(i)$ ("bottom up")

- **Fact:** Can solve even faster using Karatsuba's algorithm!

What is the tradeoff?

“When you gain something, you usually lose something too”

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

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

Time: 1.62^n
Space: n

gain: improved run time
lose: store in memory M

time-space tradeoff

Time $O(n^2)$
Space $O(n^2)$

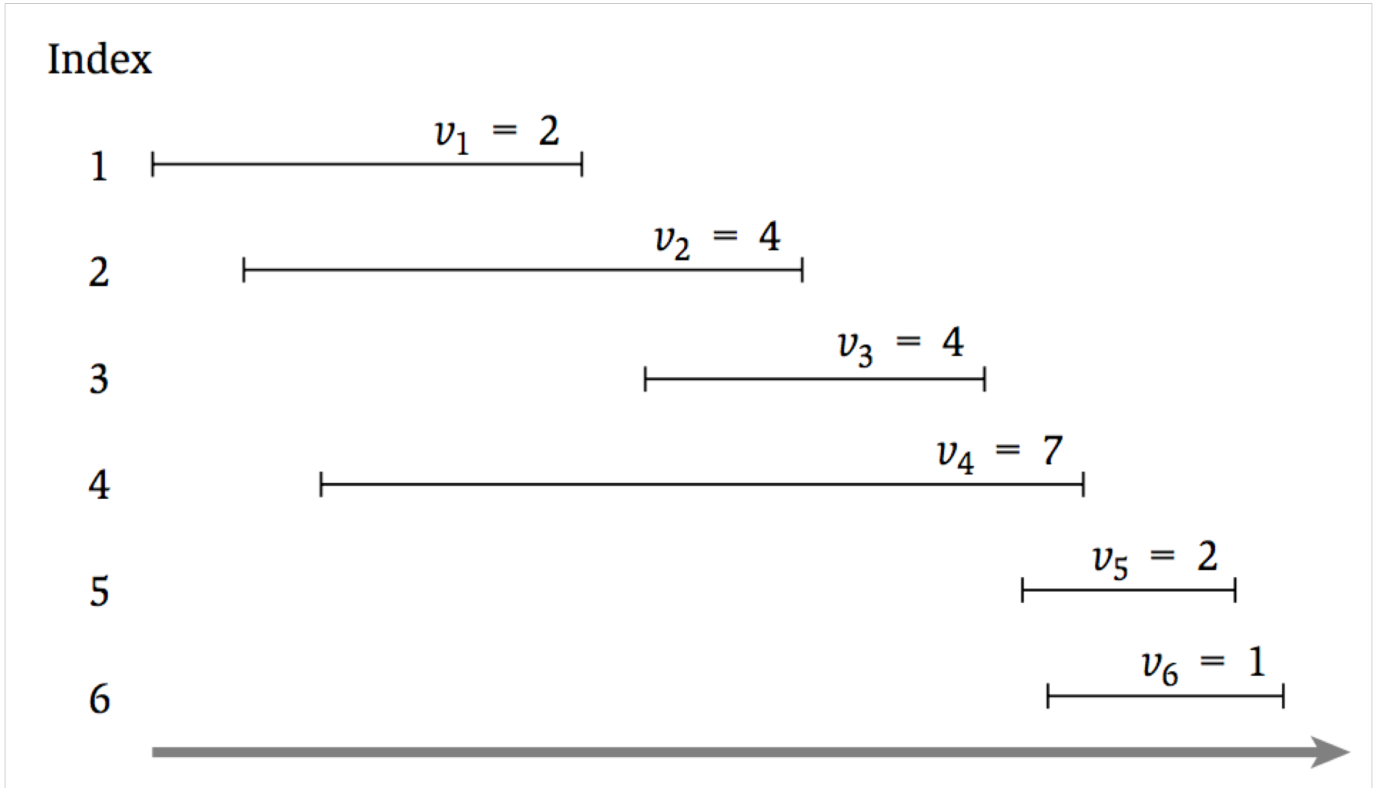
Dynamic Programming: Interval Scheduling

Interval Scheduling

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

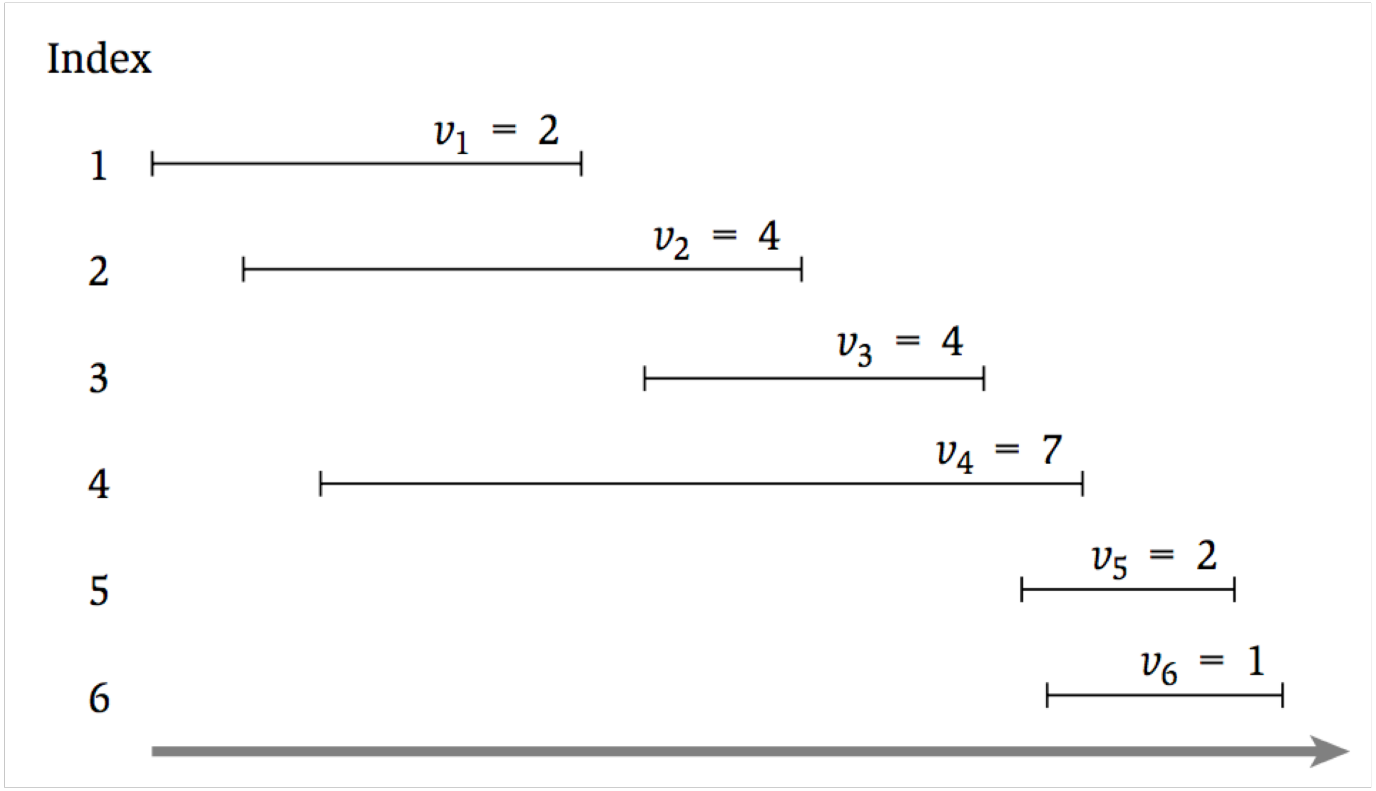
- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S maximizing the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \dots, n\}$
  - A schedule S is **compatible**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

Interval Scheduling



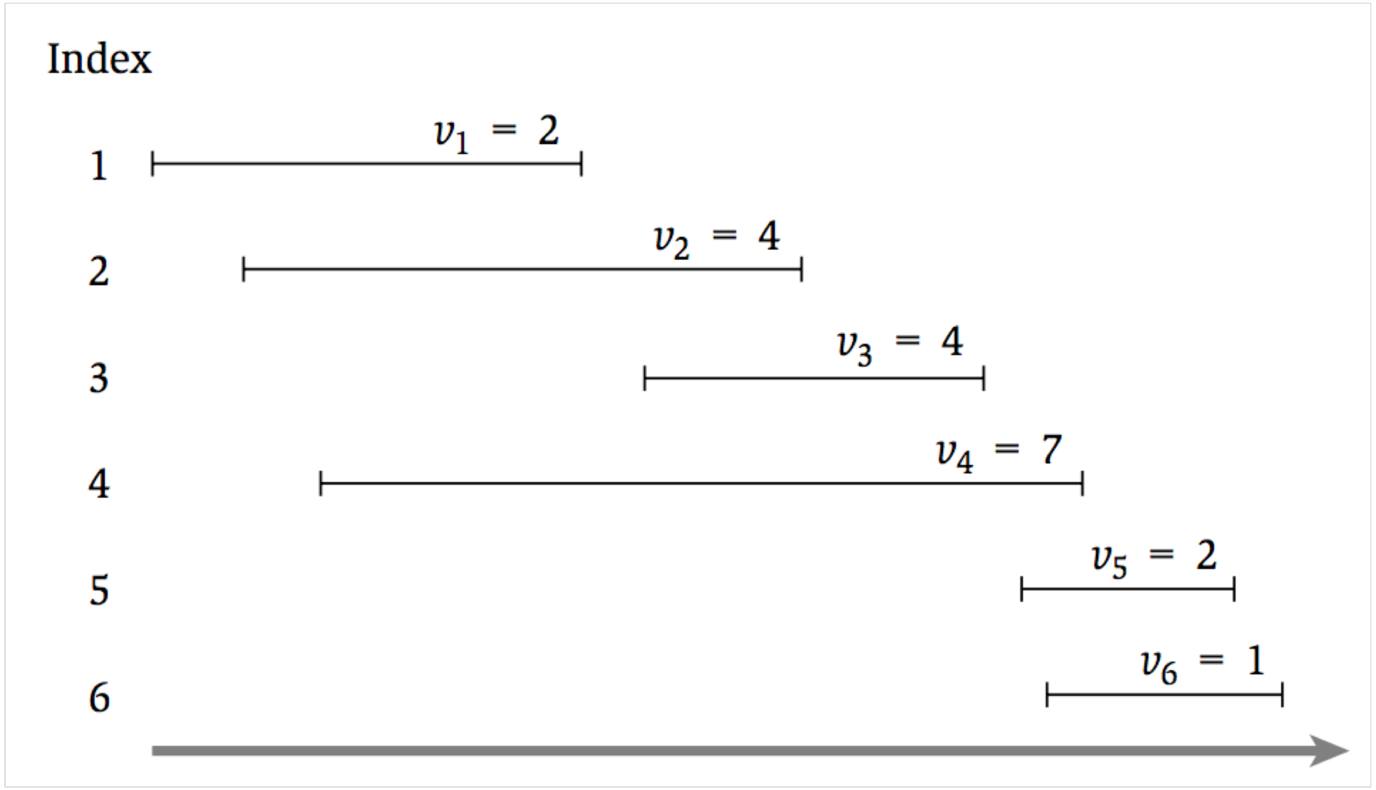
Possible Algorithms

- Choose intervals in decreasing order of v_i



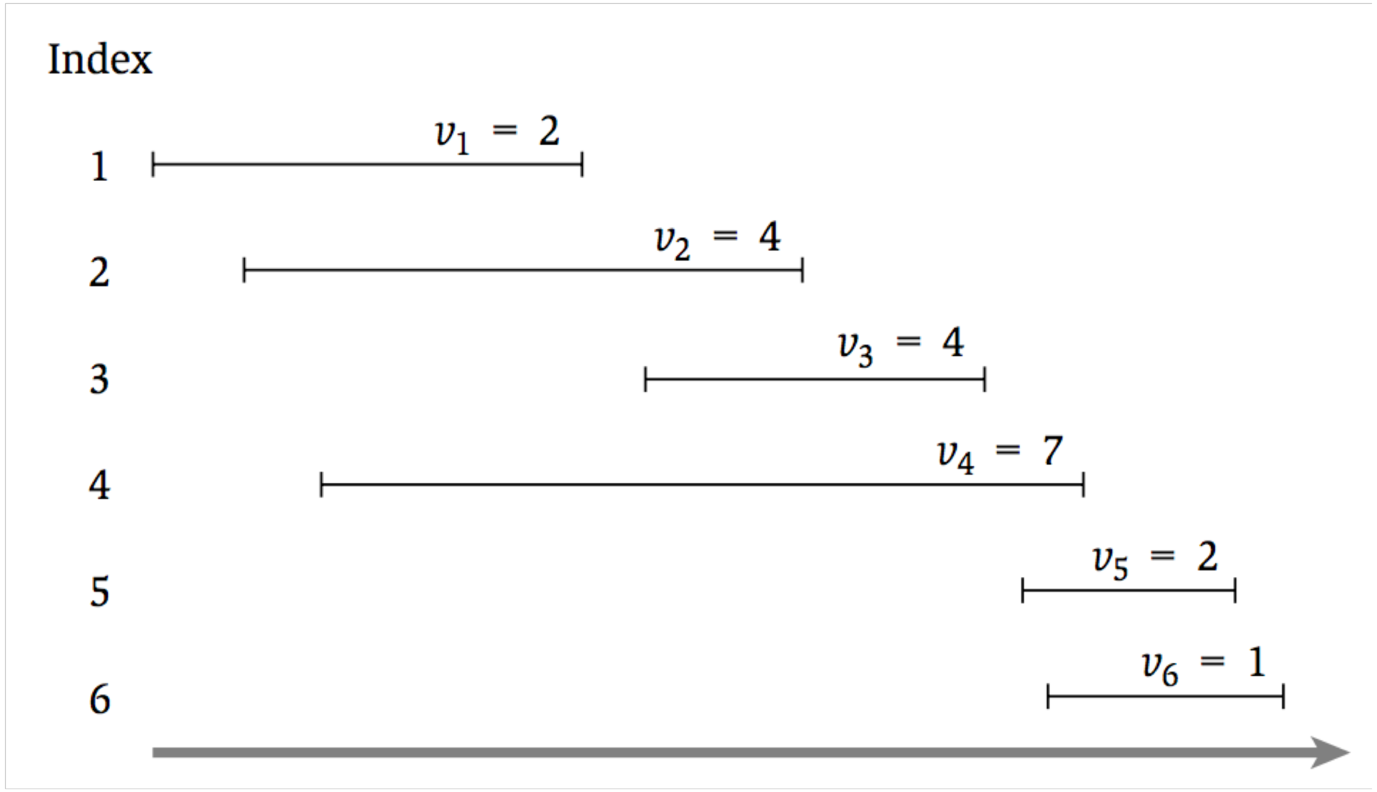
Possible Algorithms

- Choose intervals in increasing order of s_i



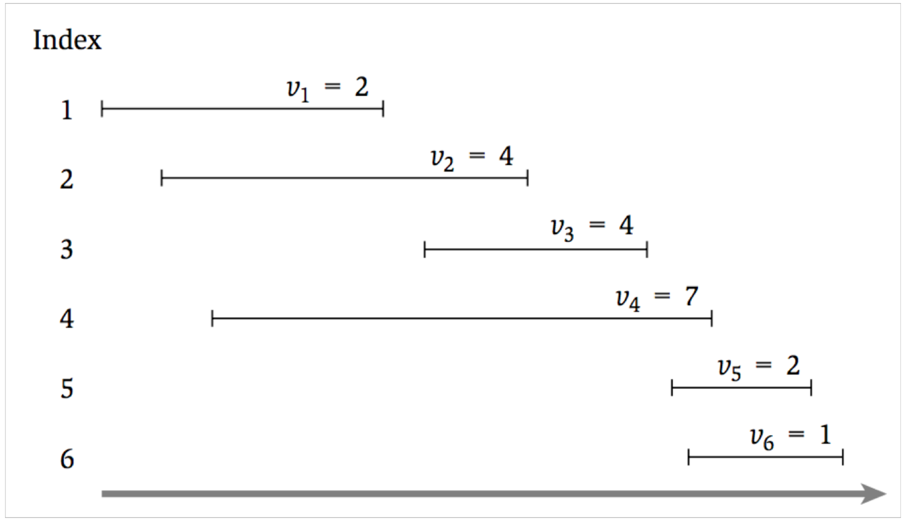
Possible Algorithms

- Choose intervals in increasing order of $f_i - s_i$



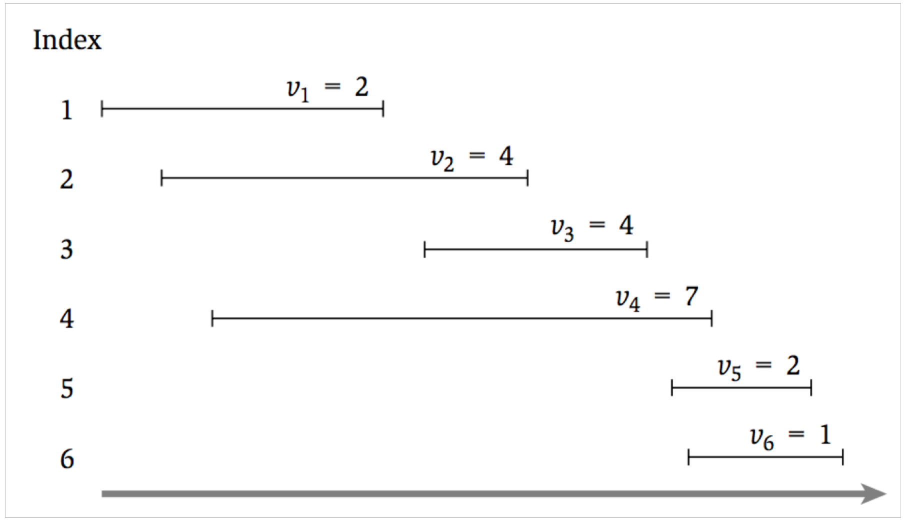
A Recursive Formulation

- 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

- Let O_i be the **optimal schedule** using only the intervals $\{1, \dots, i\}$
- **Case 1:** Final interval is not in O ($i \notin O$)
  - Then O must be the optimal solution for $\{1, \dots, i-1\}$
- **Case 2:** Final interval is in O ($i \in O$)
  - 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 must be i + the optimal solution for $\{1, \dots, p(i)\}$

A Recursive Formulation

- Let $OPT(i)$ be the **value of the optimal schedule** using only the intervals $\{1, \dots, i\}$
- **Case 1:** Final interval is not in O ($i \notin O$)
  - Then O must be the optimal solution for $\{1, \dots, i-1\}$
- **Case 2:** Final interval is in O ($i \in O$)
  - 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 must be i + the optimal solution for $\{1, \dots, p(i)\}$

- $OPT(i) = \max\{OPT(i-1), v_n + OPT\big(p(i)\big)\}$
- $OPT(0) = 0, OPT(1) = v_1$

Interval Scheduling: Take I

```
// 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))}
```

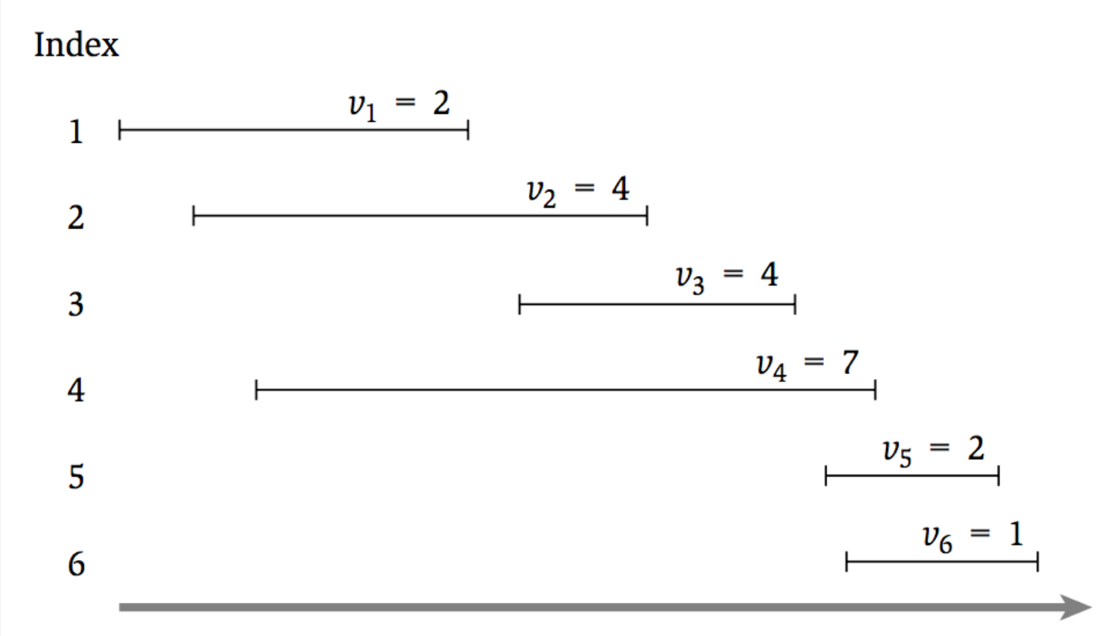
- What is the running time of **`FindOPT(n)`**?

Interval Scheduling: Take II

```
// All inputs are global vars
M ← empty array, M[0]←0, M[1]←1
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: Take II



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

Interval Scheduling: Take III

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

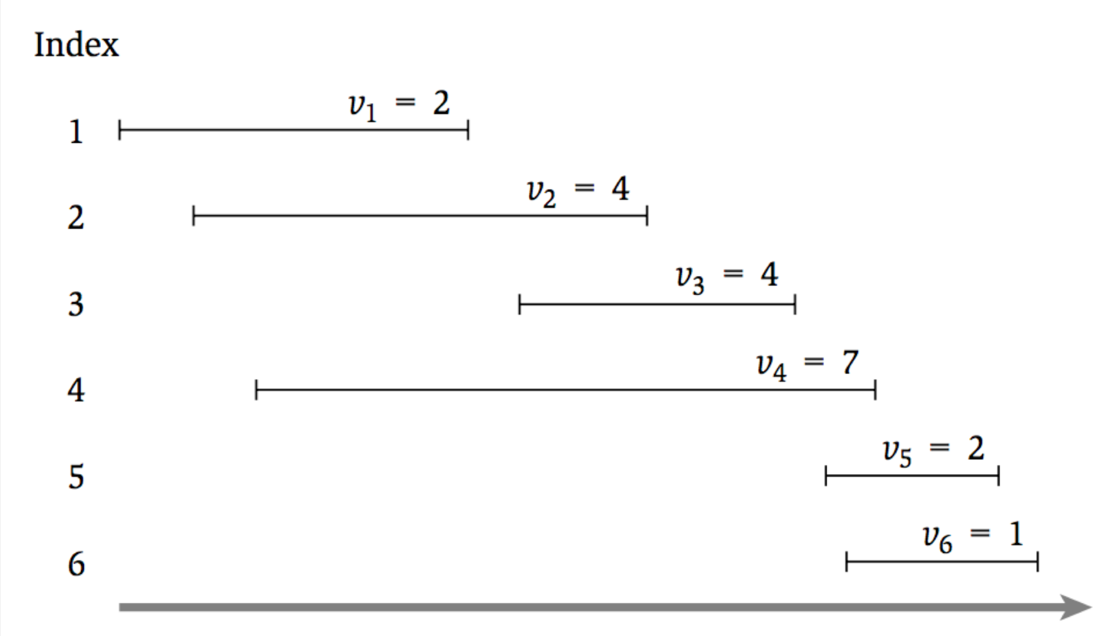
- What is the running time of **`FindOPT(n)`**?

Finding the Optimal Solution

- Let $OPT(i)$ be the **value of the optimal schedule** using only the intervals $\{1, \ldots, i\}$
- **Case 1:** Final interval is not in O ($i \notin O$)
- **Case 2:** Final interval is in O ($i \in O$)

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

Interval Scheduling: Take II



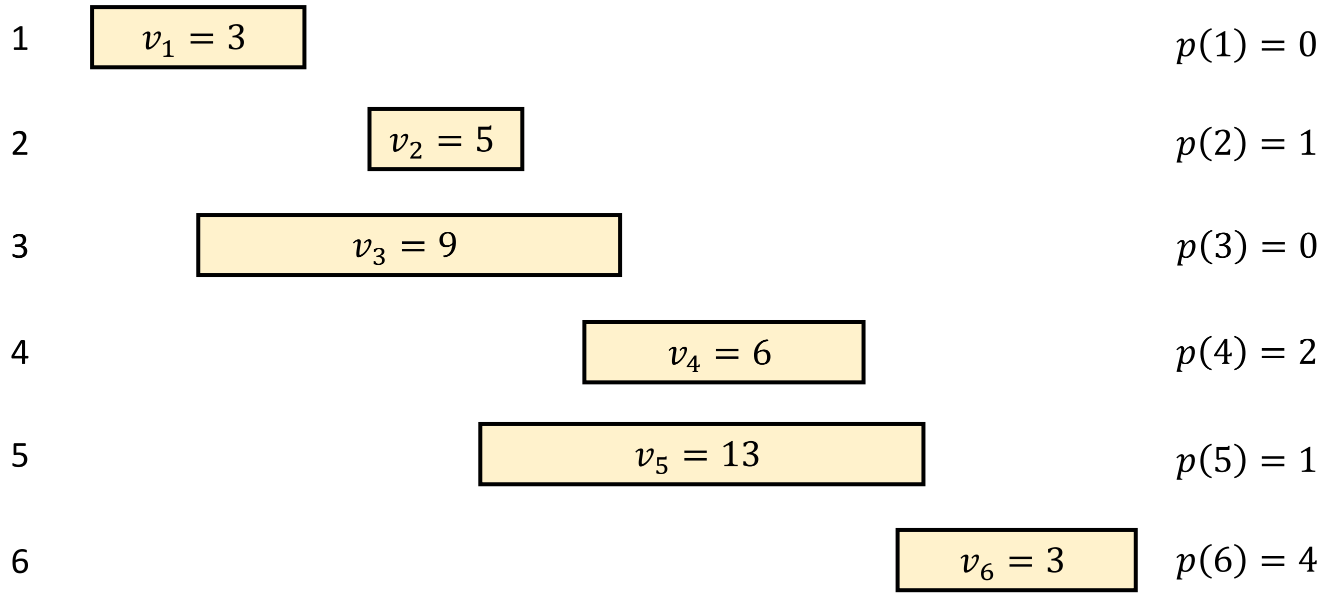
M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]

Interval Scheduling: Take III

```
// 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)`**?

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
  - Top-Down: store solution to subproblems
  - Bottom-Up: iterate through subproblems in order

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