CS3000: Algorithms & Data
Paul Hand

Lecture 9:

- Dynamic Programming
- Interval Scheduling

Feb 6, 2019

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

Dynamic Programming: Interval Scheduling

Interval Scheduling

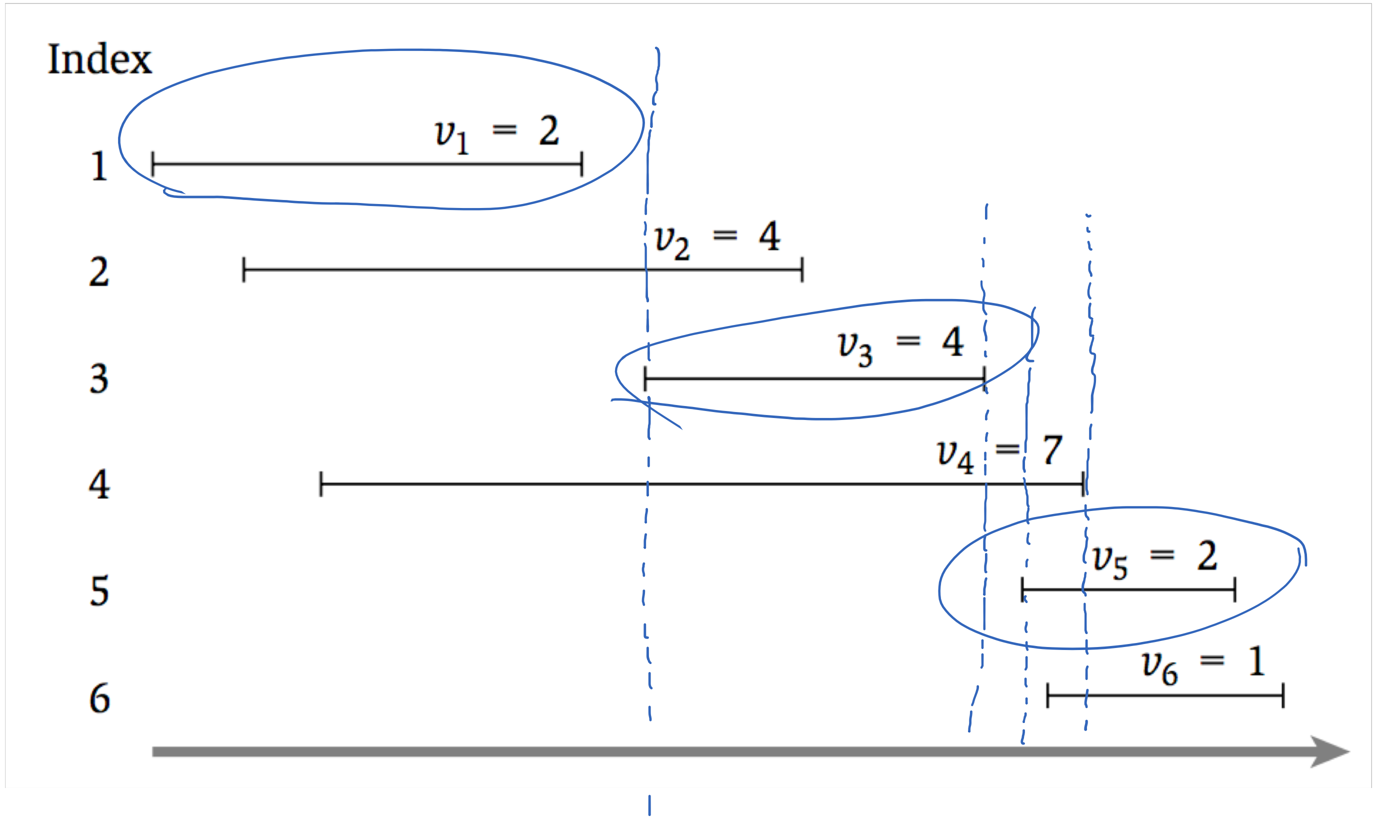
- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …

start time — finish time of i^{th} request

- **Input:** n intervals (s_i, f_i) each with value v_i — what they pay
  - 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 $\sum_{i \in S} v_i$

Interval Scheduling:

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 $\sum_{i \in S} v_i$



Activity: Find the schedule that maximizes the total value of the intervals.

1 & 3 & 5

Why is this the best? Justify.

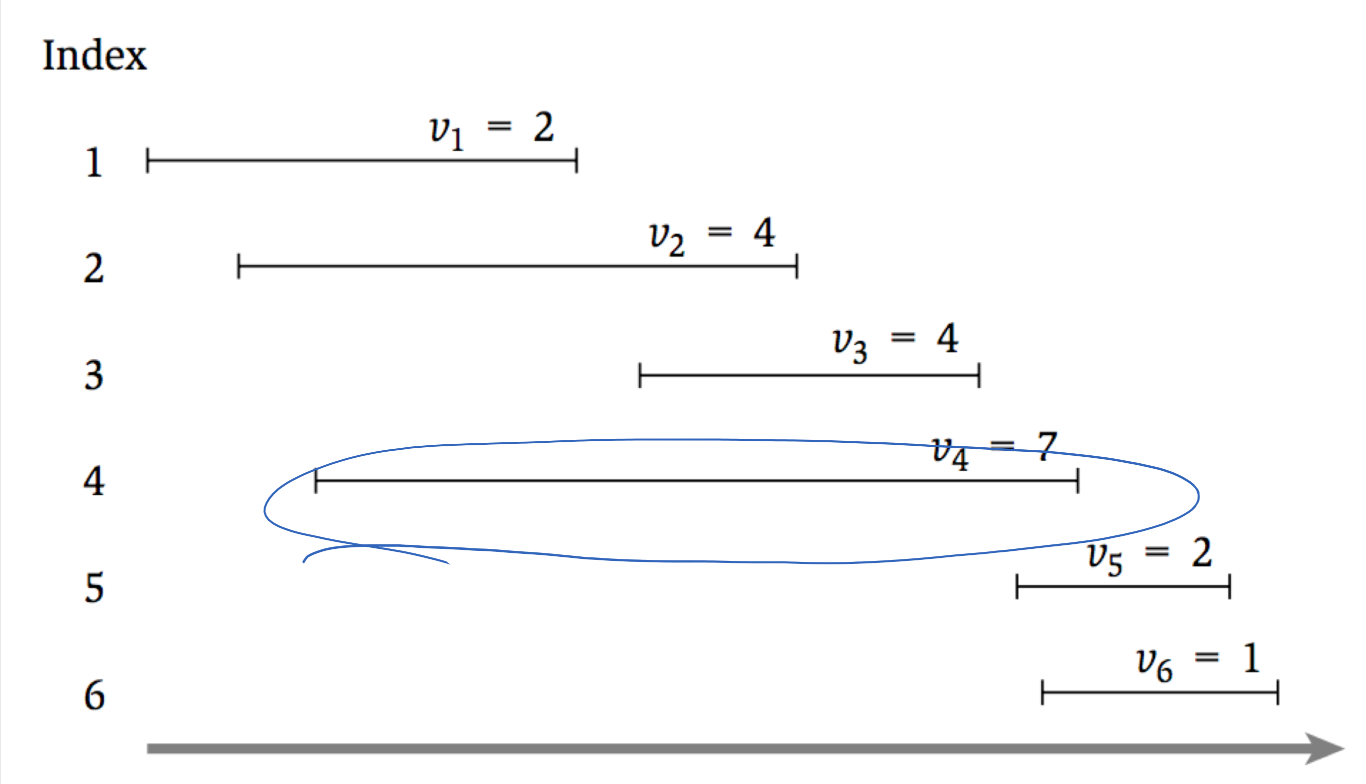
Either 4 is in schedule or it is not.

repeat

max val is 7

Possible Algorithms

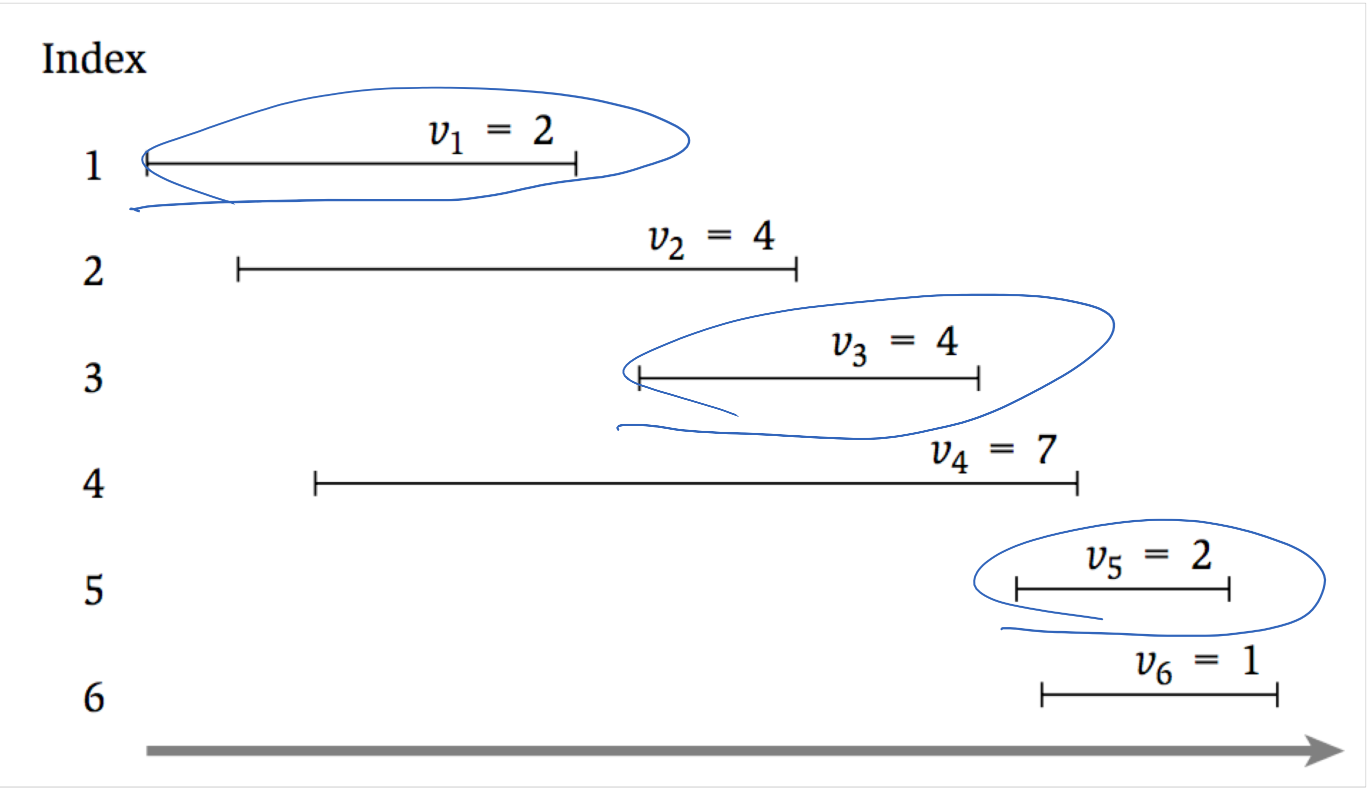
- Choose intervals in decreasing order of v_i



Sched is 4. Value is 7

Possible Algorithms

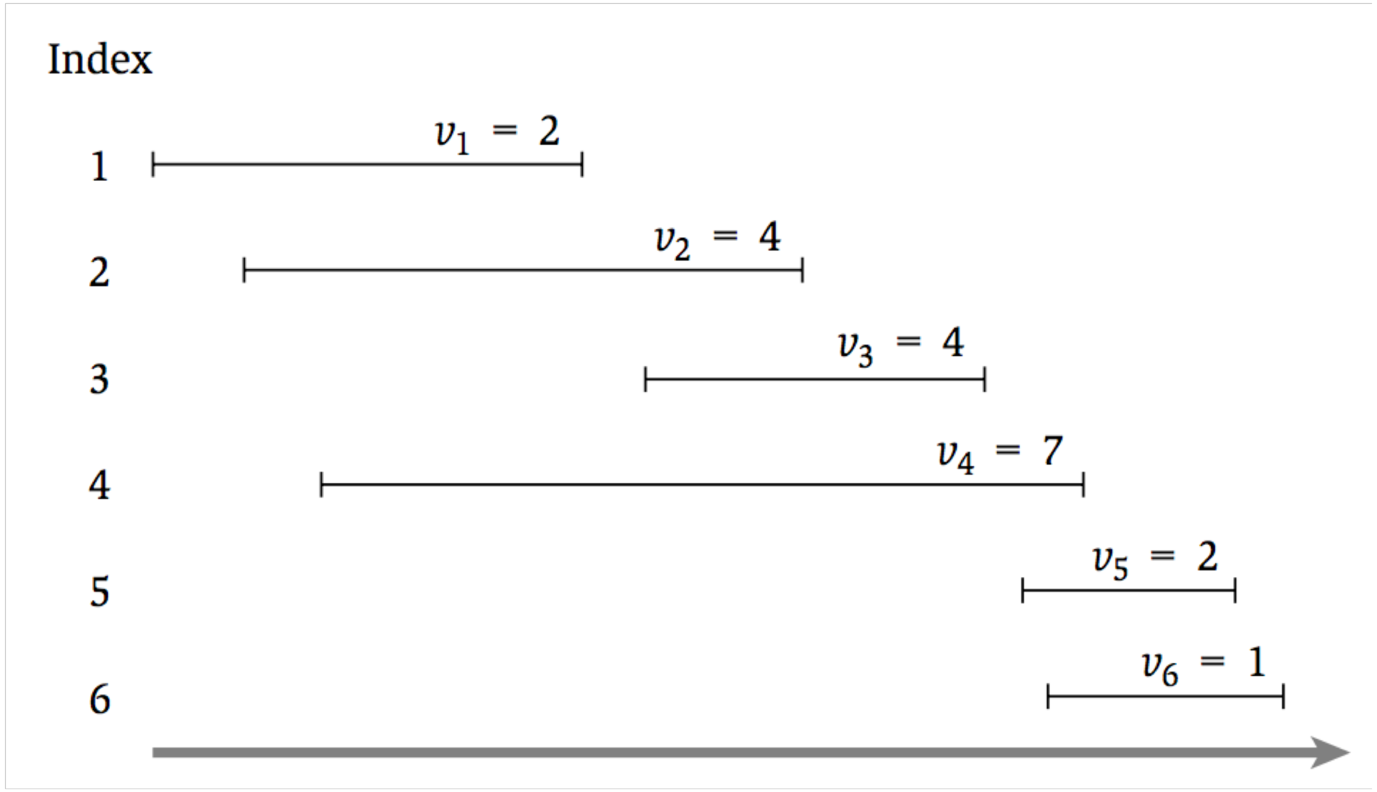
- Choose intervals in increasing order of s_i



happens to work
in this case

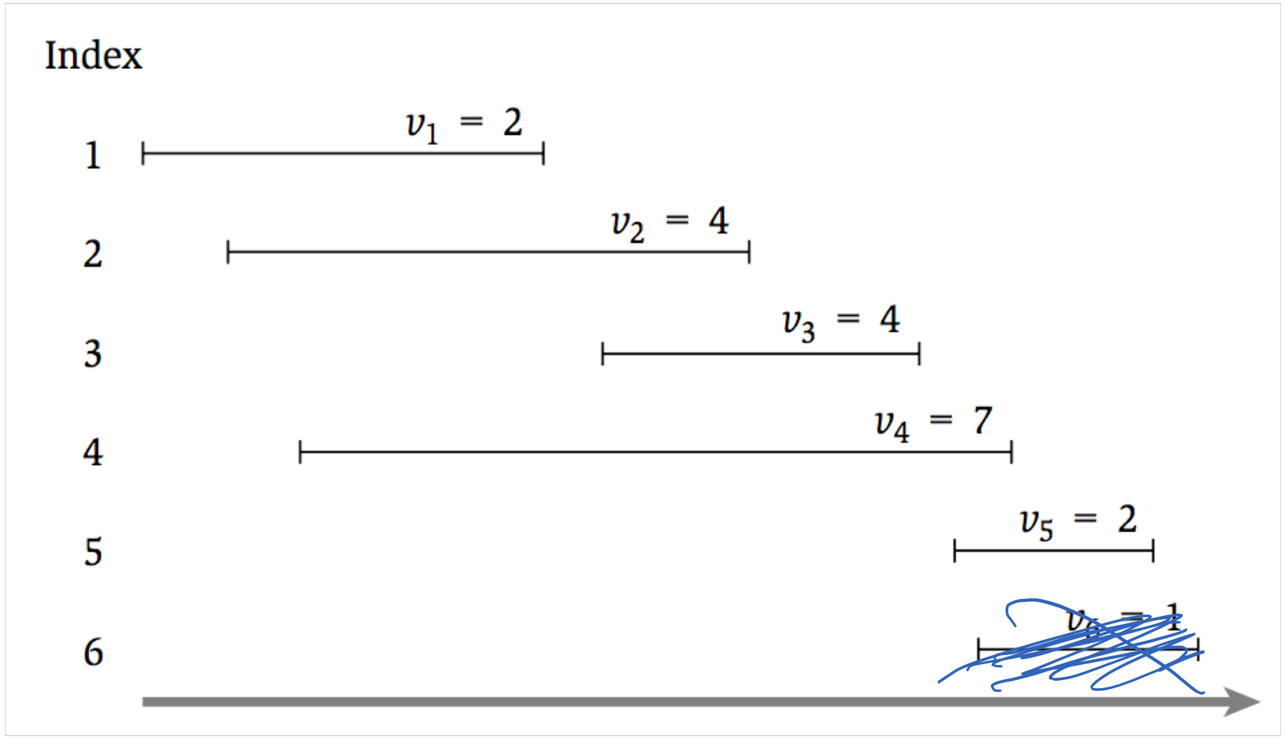
Possible Algorithms

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



A Recursive Formulation

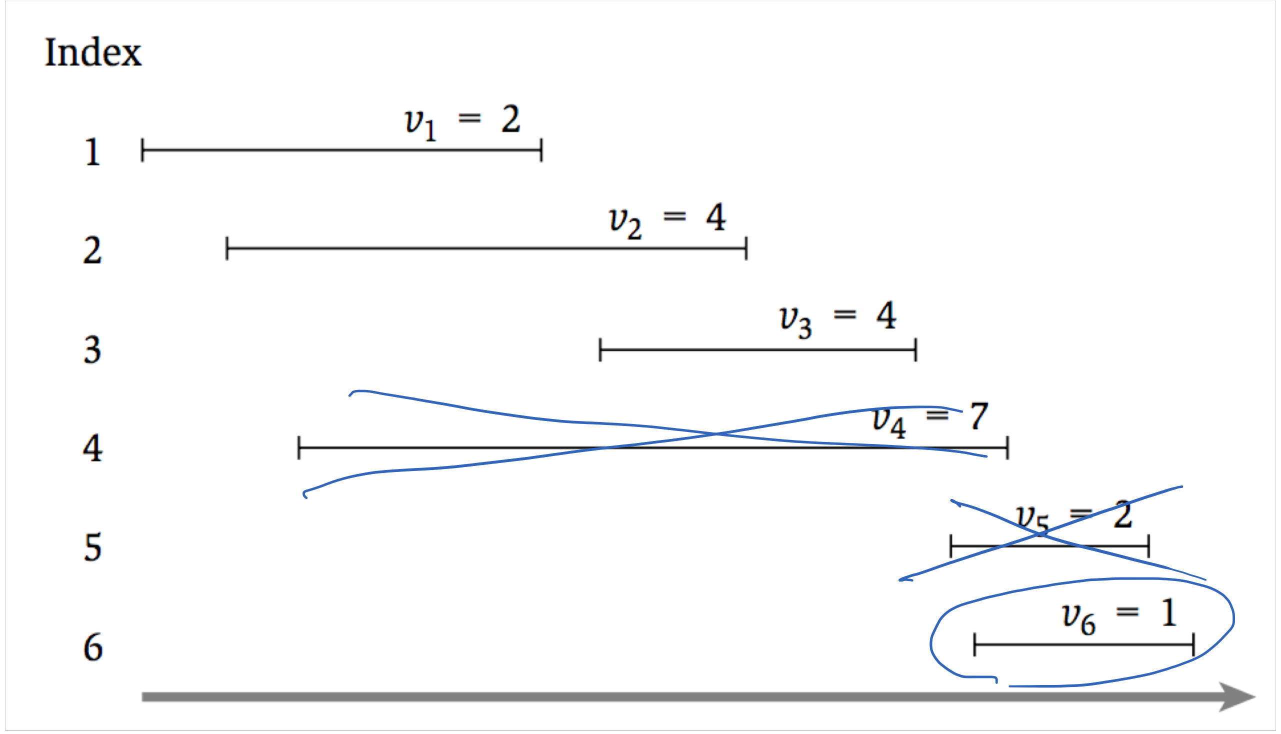
- Let O be the **optimal** schedule
- **Case 1:** Final interval is not in O (i.e. $6 \notin O$)
  - Then O must be the optimal solution for $\{1, \ldots, 5\}$



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 schedule for $\{1, \ldots, 3\}$

because 4&5 conflict with 6



A Recursive Formulation

Notation
$O = O_n$

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

i will conflict with ~~of~~ any item j bigger than $p(i)$

A Recursive Formulation

- 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 ($i \notin O$)
  - Then O must be the optimal solution for $\{1, \ldots, i-1\}$
- **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 schedule for $\{1, \ldots, p(i)\}$

latest (in terms of finish time) interval compatible with i

- $OPT(i) = \max\{\underbrace{OPT(i-1)}_{\text{case 1}}, \underbrace{v_i + OPT(p(i))}_{\text{case 2}}\}$ ← recurrence
- $OPT(0) = 0, OPT(1) = v_1$

How do we solve this recurrence?

Interval Scheduling: Take I

assume p is computed

```
// 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))}
```

worst case   # of recursive calls

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

worst case: $p(n) = n-1$.

Call FindOPT(n-1) twice

$O(2^n)$

Interval Scheduling: Take II

Memoization

$M(i) = OPT(i)$

```
// 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]
```

max ind compatible w/ interval n

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

$\approx 2n$

F(n)

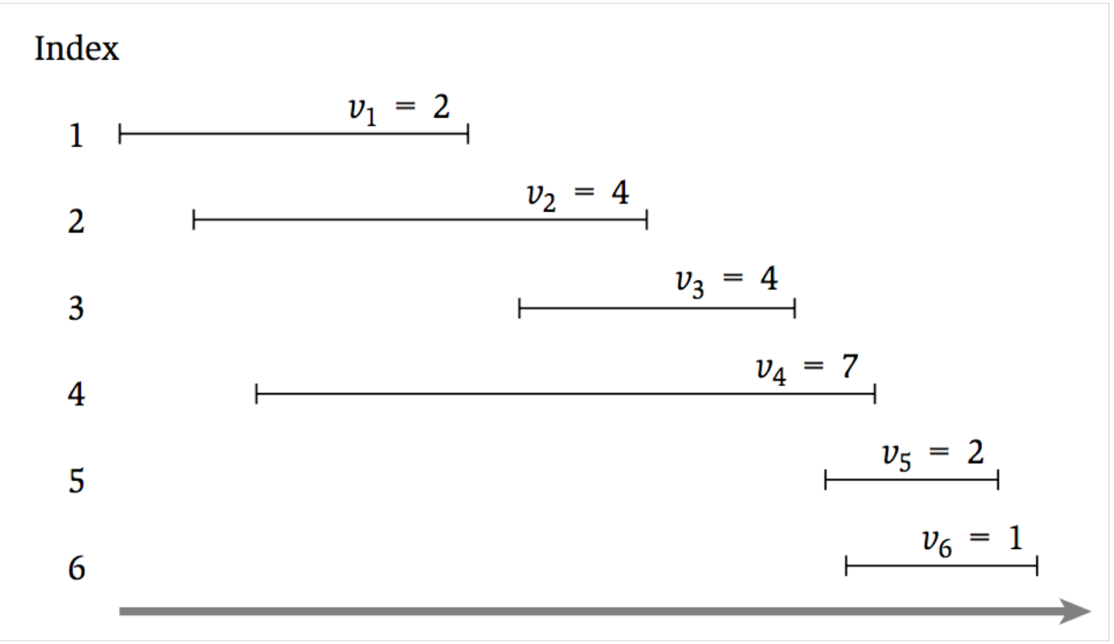
F(n-1) F(...)

F(n-2) ...

F(1)

because of memoization recursion tree has width of 2 at each layer

Interval Scheduling: Take II




```
// 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]
```

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

P gives indices

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

M gives optimal value

max(M[1], 4+ M[0])

max(M[2], 4+ M[1])

Interval Scheduling: Take III

Bottom-up approach

```
// All inputs are global vars
FindOPT(n):
  M[0] ← 0, M[1] ← 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)**?

$O(n)$

Can interval scheduling problem be solved in $O(n)$ time?

Sort: $n \log n$

compute p.

Finding the Optimal Solution /Schedule (not just value)

- 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 ($i \in O$)

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

Interval Scheduling: Take III

assume you have M

M= {1,3,5}

```
// All inputs are global vars
FindSched(M,n):
  if (n = 0): return Ø
  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)
```

Schedule containing only first interval

case 2

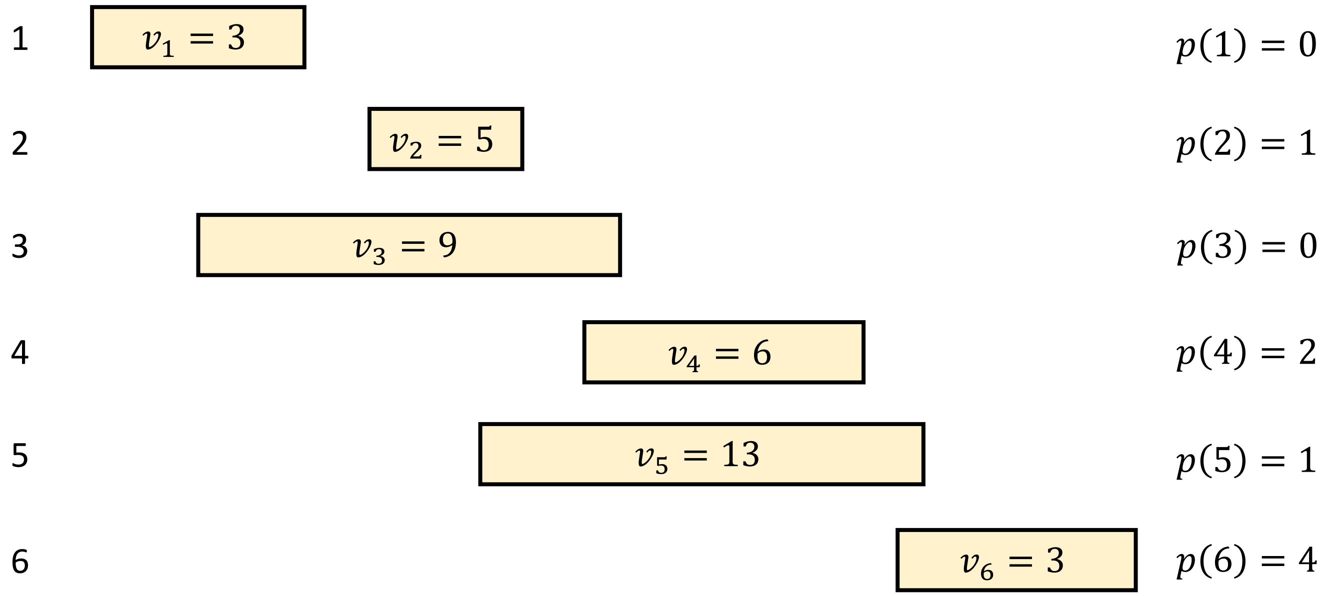
case 1

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

O(n).

To ponder: can you find opt sched & value together?

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**