CS3000: Algorithms & Data
Jonathan Ullman

Lecture 5:
- Dynamic Programming:
  Fibonacci Numbers, Interval Scheduling

Jan 22, 2020

Dynamic Programming

- Don't think too hard about the name
  - *I thought dynamic programming was a good name. It was something not even a congressman could object to. So I used it as an umbrella for my activities.* -Bellman

- Dynamic programming is careful recursion
  - Break the problem up into small pieces
  - Recursively solve the smaller pieces
  - **Key Challenge:** identifying the pieces

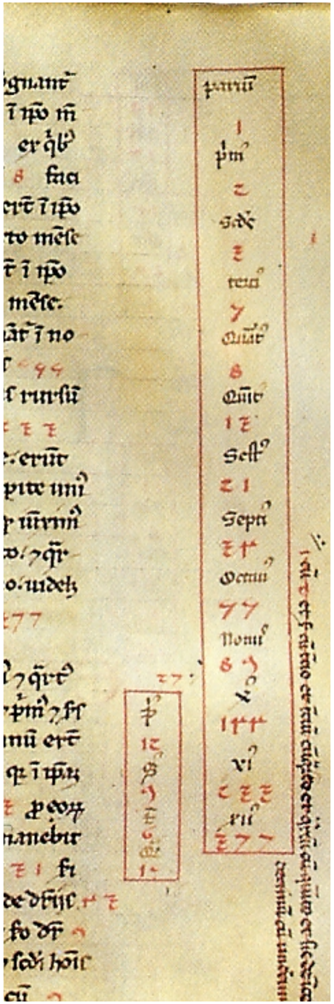
Warmup: Fibonacci Numbers

Fibonacci Numbers

F(0) F(1)

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

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



Fibonacci Numbers: Take I

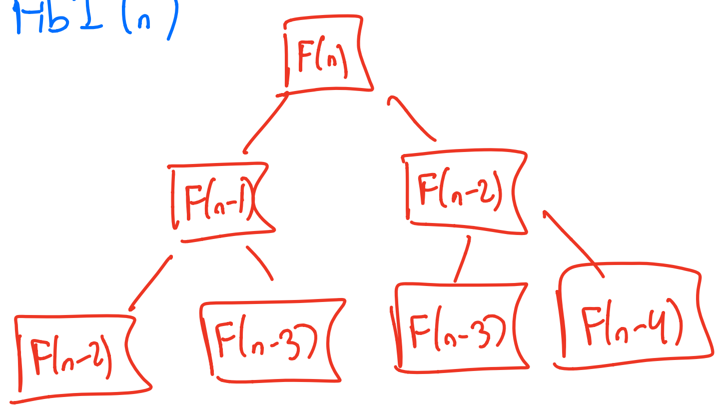
```
FibI(n):
  If (n = 0): return 0
  ElseIf (n = 1): return 1
  Else: return FibI(n-1) + FibI(n-2)
```

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

$C(n)$: # of calls made by FibI(n)

$C(n) = C(n-1) + C(n-2)$

$C(n) = F(n) \approx 1.62^n$



Fibonacci Numbers: Take II

"Memoization" "Top-Down Dynamic Programming"

```
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]
```

FibII(0) FibII(1)

- How many recursive calls does **FibII(n)** make?
  - We fill n-1 new elements of M
  - Each pair of recursive calls fills one elt
  - ⇒ At most 2(n-1) calls

$O(n)$

Fibonacci Numbers: Take III

M | 0 | 1 | 1 | 2 | 3 | 5 | 8 | ... | F(n-1) | F(n)

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

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

"Bottom-Up Dynamic Programming"

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

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

Dynamic Programming: Interval Scheduling

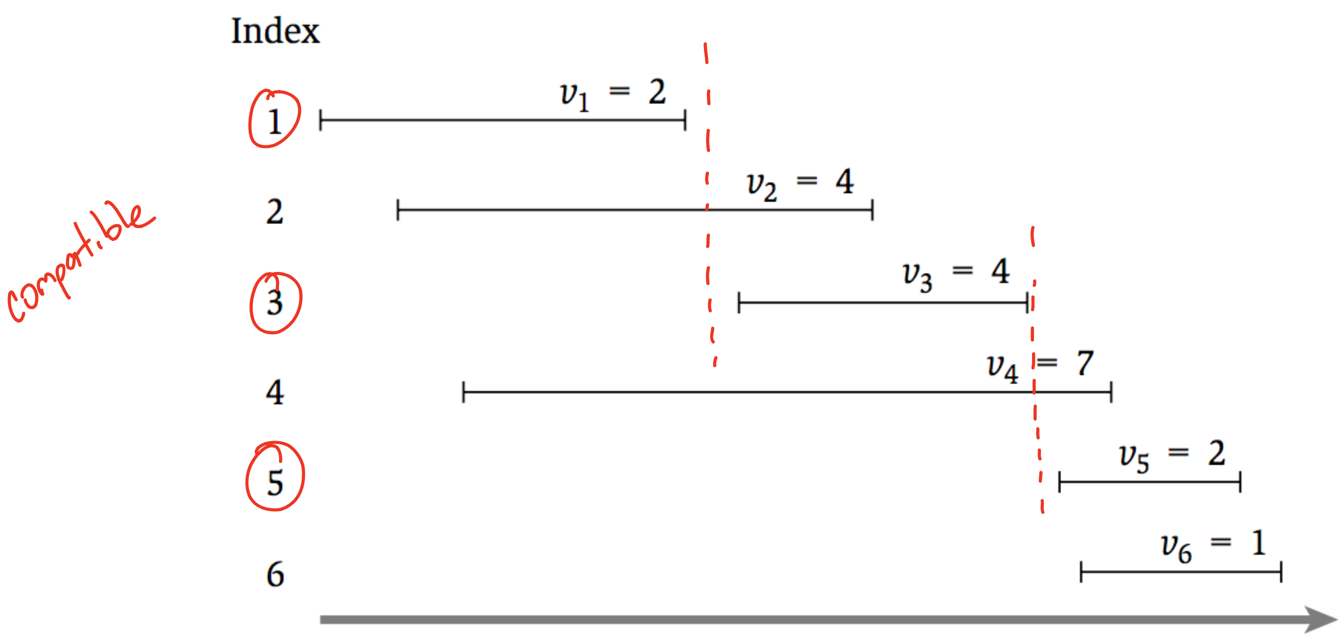
Interval Scheduling

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

- **Input:** n intervals (s_i, f_i) each with value v_i > 0
    - 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

value({1, 3, 5}) = 2 + 4 + 2 = 8



Possible Algorithms

- Choose intervals in decreasing order of v_i

Possible Algorithms

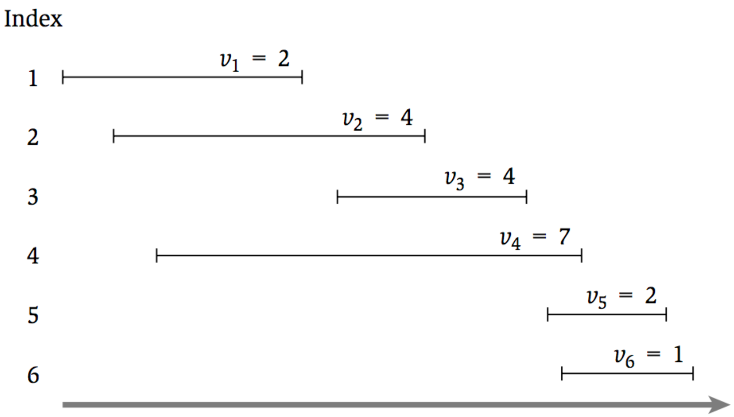
- Choose intervals in increasing order of s_i

Possible Algorithms

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

A Recursive Formulation

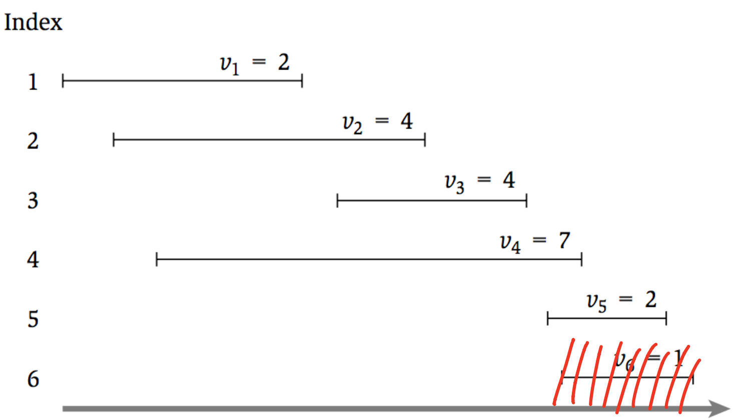
- Let O be the **optimal** schedule
- **Bold Statement:** O either contains the last interval or it does not.



A Recursive Formulation

- Let O be the **optimal** schedule
- **Case 1:** Final interval is not in O (i.e. $6 \notin O$)

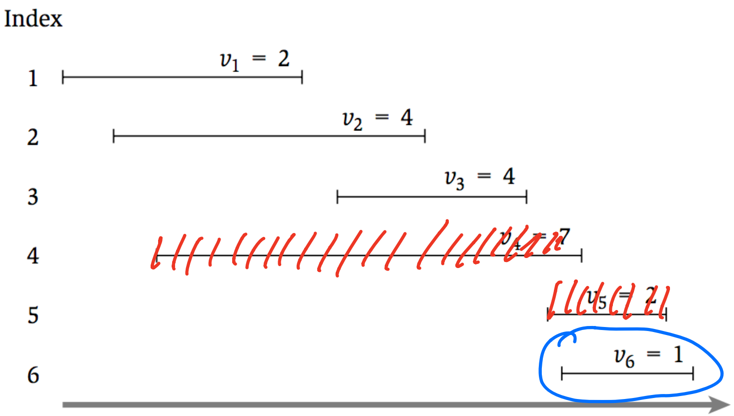
The O must be the optimal schedule for $\{1,2,3,4,5\}$



A Recursive Formulation

- Let O be the **optimal** schedule
- **Case 2:** Final interval is in O (i.e. $6 \in O$)

O must be $\{6\}$ + [the optimal schedule for $\{1, 2, 3\}$]



A Recursive Formulation

- Let O_i be the **optimal schedule** using only the intervals $\{1, \ldots, i\}$
- **Case 1:** Final interval is not in O ($i \notin O$) $[O_i = O_{i-1}]$
  - 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 $f_j < s_i$ $[O_i = \{i\} + O_{p(i)}]$
  - Then O must be i + the optimal solution for $\{1, \ldots, p(i)\}$

Any of $\{1, 2, \ldots, p(i)\}$ are compatible with i

If $value(O_{i-1}) > v_i + value(O_{p(i)})$

then $O_i = O_{i-1}$

If $v_i + value(O_{p(i)}) > value(O_{i-1})$

then $O_i = \{i\} + O_{p(i)}$

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 solution for $\{1, \ldots, p(i)\}$

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

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)`**?

Can be exponential in n.

Interval Scheduling: Take II ("Top Down")

M[i] stores 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]
```

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

$O(n)$ (2 recursive calls / array elt)

× (n-1 array elts)

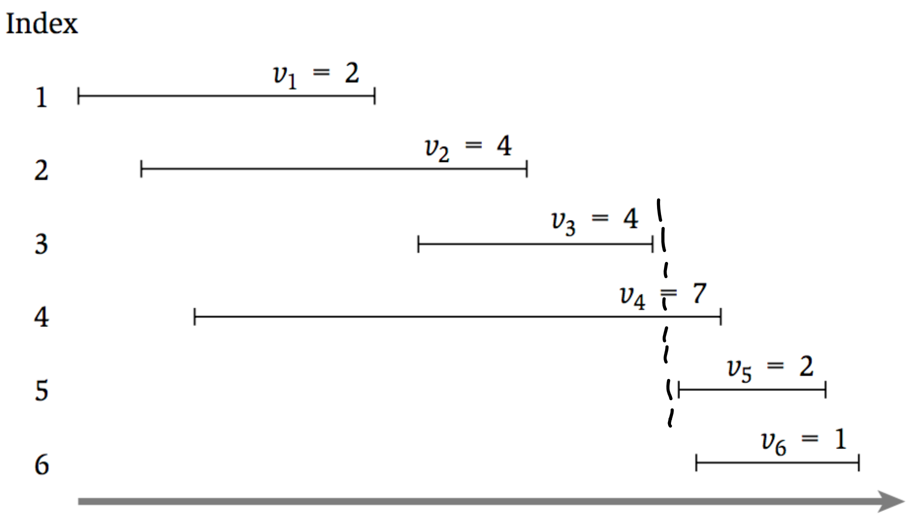
Interval Scheduling: Take III

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

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

O(n) time

Interval Scheduling: Take II



$p(1) = 0$

$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$

$M[2] = \max\{M[1], v_2 + M[0]\}$
$= \max\{2, 4+0\} = 4$

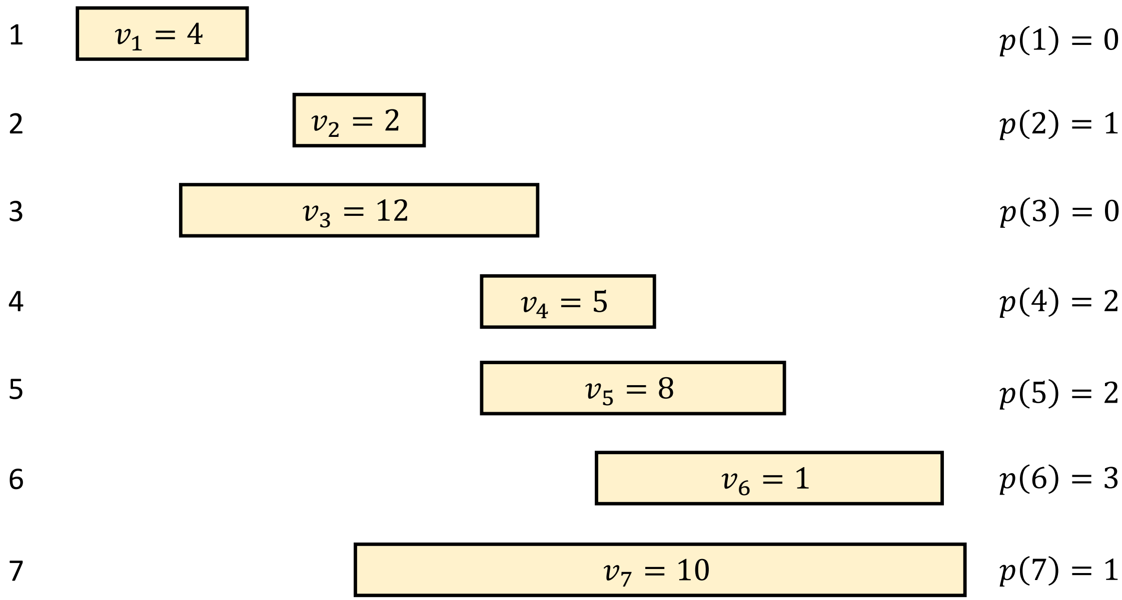
$M[3] = \max\{M[2], v_3 + M[1]\}$
$= \max\{4, 4+2\} = 6$

	yes	yes	yes	yes	yes	no
M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	2	4	6	7	8	8

$M[6] = \max\{M[5], v_6 + M[3]\}$
$= \max\{8, 1+6\}$

value of the optimal schedule

Now You Try



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

Finding the Optimal Schedule

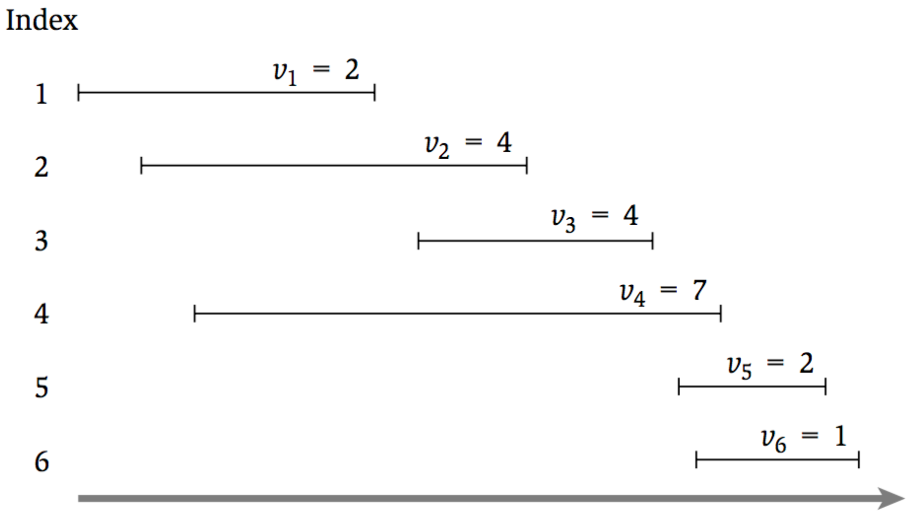
Does this go both ways?

- 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 $(i \notin O)$ $\Leftrightarrow$ $OPT(i) = OPT(i-1)$
- **Case 2:** Final interval is in O $(i \in O)$ $\Leftrightarrow$ $OPT(i) = v_i + OPT(p(i))$

Be careful: Both could be true (if there are multiple opts

- $OPT(i) = \max\{\underbrace{OPT(i-1)}_{\text{If this is "the" max then } O_i = O_{i-1}}, \underbrace{v_i + OPT(p(i))}_{\text{If this is "the" max then } O_i = \{i\} + O_{p(i)}}\}$

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 (vn + 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)`**?

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**