CS4800: Algorithms & Data
Jonathan Ullman

Lecture 7:

- Dynamic Programming: Fibonacci Numbers, Interval Scheduling

Jan ~~26~~, 2018
30

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."
    –Richard Bellman

- Dynamic programming is careful recursion
  - Identify a "small" number of "subproblems"
  - Relate these problems via a recurrence
  - Carefully solve the recurrence

sounds similar to D&C, but is different in practice

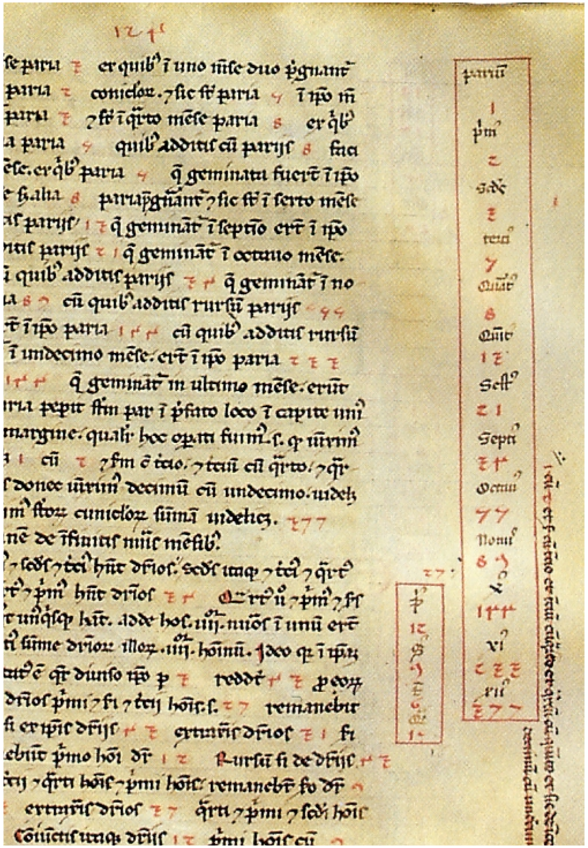
Warmup: Fibonacci Numbers

Fibonacci Numbers

recurrence is given "for free"

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

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



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Fibonacci Numbers Alg I

FibI(n):

  If $n = 0,1$: return n // Base Cases

  Else: return FibI$(n-1)$ + FibI$(n-2)$ //Recursion

- How many total recursive calls does this algorithm make when computing $F(n)$?

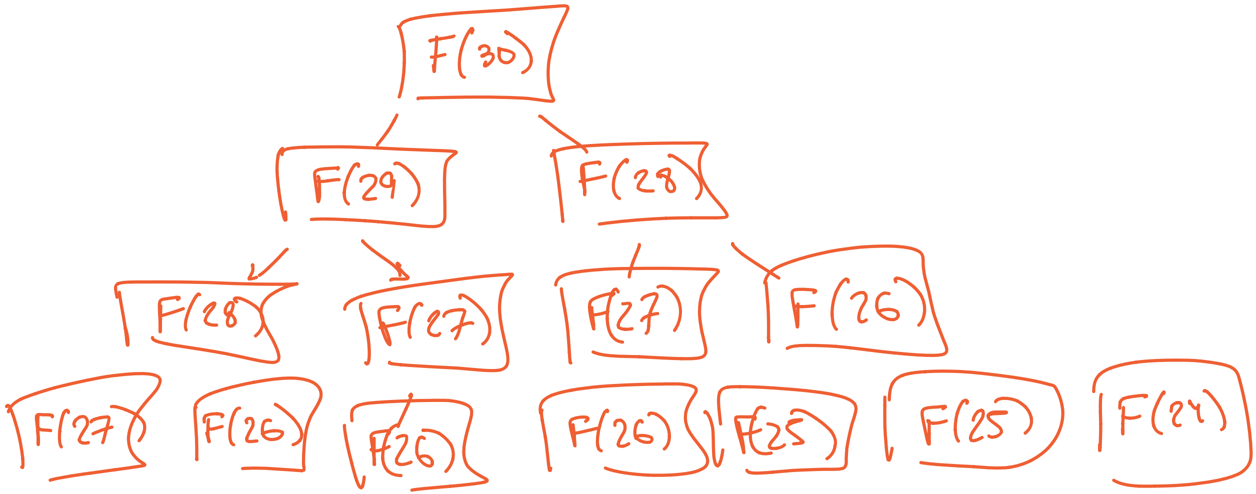
If $R(n)$ is the # of recursive calls to FibI on input n

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

$R(n) = F(n) = 1.62^n$ (over a million calls to compute $F(30)$)

Fibonacci Numbers Alg I

FibI(n):
    If $n = 0,1$: return n
    Else: return FibI($n-1$) + FibI($n-2$)



Fibonacci Numbers Alg II ("Memoization")

Store solutions to problems you have already seen

Let M be an array (initialize each cell to "empty")
FibII(n):
  If $n = 0{,}1$: return n // Base case
  Else if $M[n]$ is not empty: return $M[n]$ // Checking for a solution
  Else:
    $M[n] \leftarrow \text{FibII}(n-1) + \text{FibII}(n-2)$
    return $M[n]$

already know the answer

do not know answer

- How many total recursive calls does this algorithm make when computing $F(n)$?

$M[2] \ldots M[n]$

- M only has n-1 elements at any point in the algorithm
- Every time we make two recursive calls, we fill one elem of M.
- $R(n) \leq 2(n-1)$ WAY less than 1.62^n recursive calls

Fibonacci Numbers Alg II

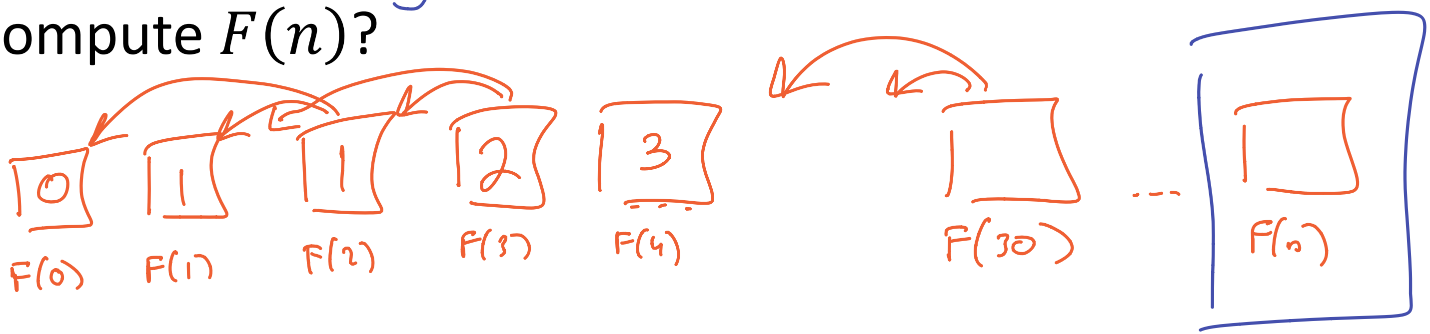
Fibonacci Numbers Alg III ("Bottom-Up / Iterative")

FibIII(n):
  $M[0] \leftarrow 0, M[1] \leftarrow 1,$
  For $i = 3, \ldots, n$:
    $M[i] \leftarrow M[i-1] + M[i-2]$
  return $M[n]$

Fill in the table in the "correct" order

$O(n)$ additions × $O(n)$ time per addition $= O(n^2)$ time

- How many additions does this algorithm take to compute $F(n)$?



Fibonacci Numbers Alg III

FibIII(n):

  $M[0] \leftarrow 0, M[1] \leftarrow 1,$

  For $i = 3, \ldots, n$:

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

  return $M[n]$

Summary

- The "obvious" recursive algorithm for Fibonacci numbers takes exponential time

- Problem: recursing on the same problem many times
  - Idea I: Remember solutions (aka "top-down") "memoization"
  - Idea II: Solve subproblems in order (aka "bottom-up")

"iterative"

- Dynamic programming is careful recursion
  - Identify a "small" number of "subproblems"
  - Relate these problems via a recurrence
  - Carefully solve the recurrence

Weighted

Interval Scheduling

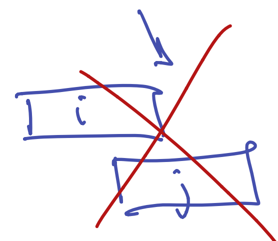
Interval Scheduling

assume all s_i, f_i are distinct

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, the Large Hadron Collider, etc…

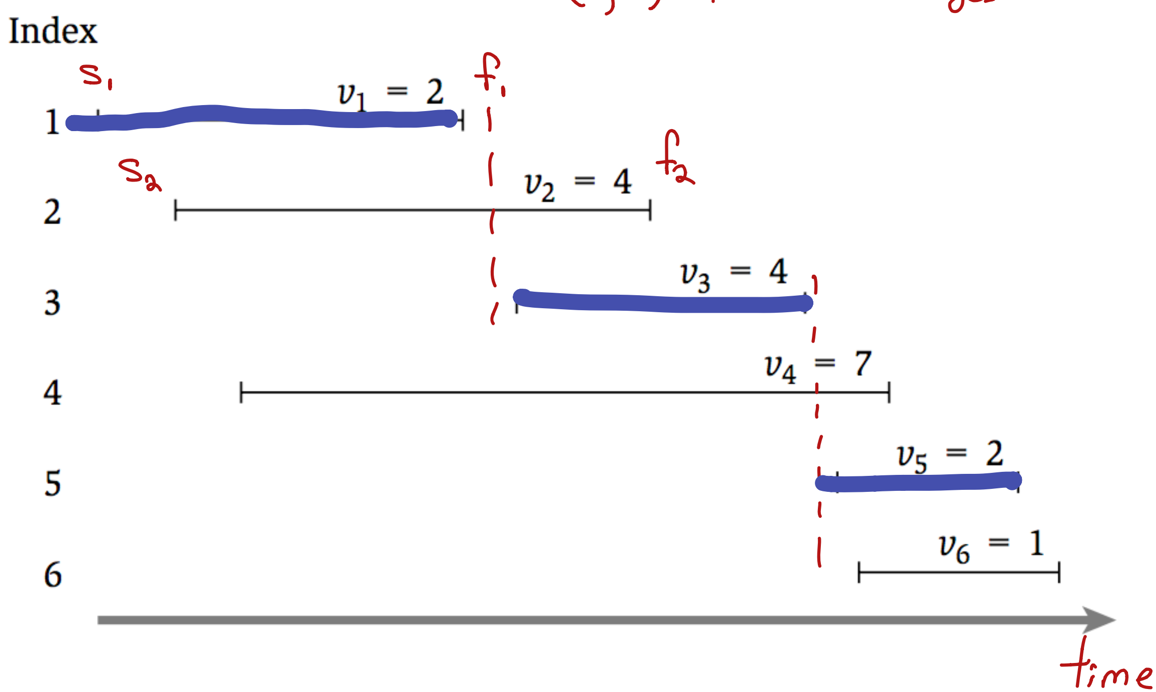
(start time, finish time, value)

- Input: a n intervals (s_i, f_i) with value v_i
- Output: a compatible schedule S with the largest possible total value
  - A schedule is a subset of intervals $S \subseteq \{1, \dots, n\}$
  - A schedule S is compatible if no two $i, j \in S$ overlap
  - The total value of S is $\sum_{i \in S} v_i$



Interval Scheduling

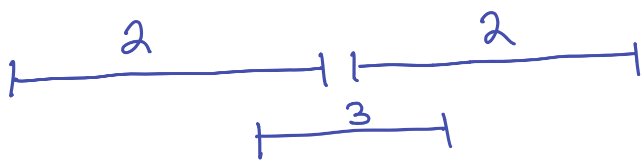
schedules	compatible?	value
{1, 4, 5, 6}	No	12
{1, 6}	yes	3
{1, 3, 5}	yes	8



Possible Algorithms

- Choose most valuable interval first

  - take most valuable
  - eliminate conflicts

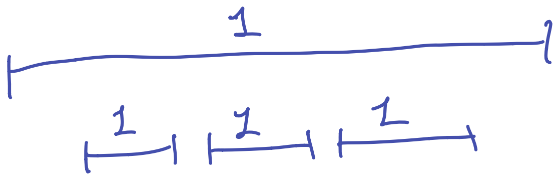


this rule gives value 3

opt value is 4

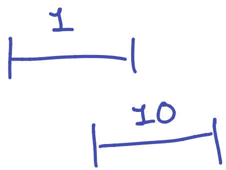
Possible Algorithms

- Choose interval with earliest start time first



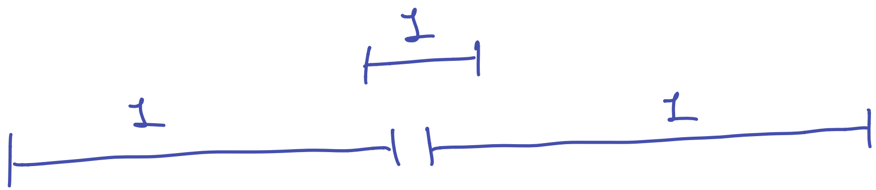
Possible Algorithms

- Choose interval with earliest finish time



Possible Algorithms

- Choose shortest interval first



The optimal set of intervals requires some "global reasoning"

A Recursive Algorithm

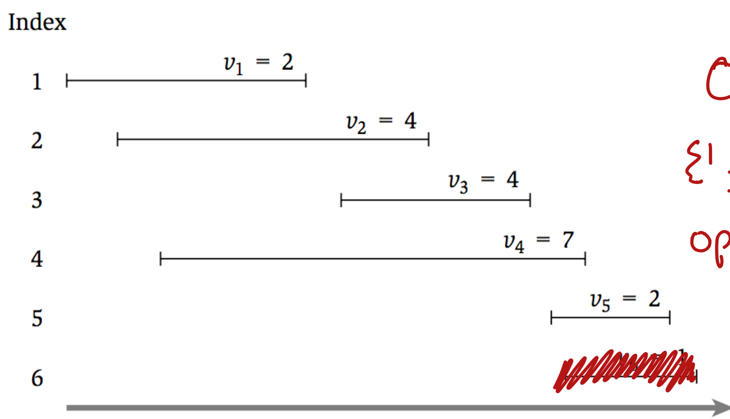
Step 1: Posit an optimal solution and think about it.

Step 2: Classify O

O either does or does not contain 6.

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

Why? If there were a better solution O' containing only intervals in {1, ..., 5} then O' is better than O.

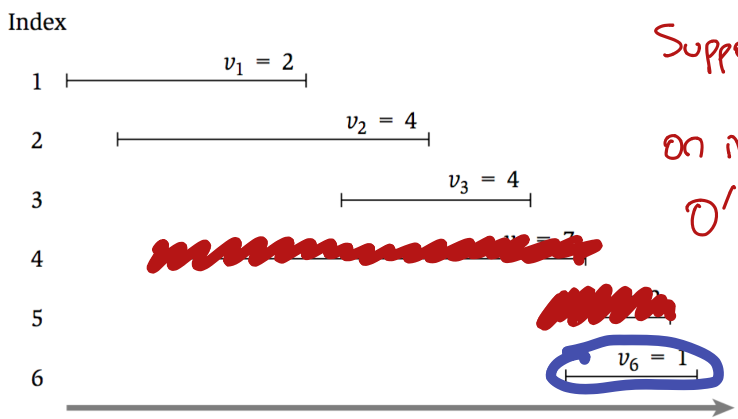


Opt solution for {1, ..., 6} is an opt solution for a smaller problem

A Recursive Algorithm

O is one of two types of solutions, each of which contains an opt soln to a smaller problem.

- Let O be the **optimal** solution
- Case I: Last interval is not in O ($6 \notin O$)
  - Then O must be the optimal solution for $\{1, \ldots, 5\}$
- Case 2: Last interval is in O ($6 \in O$)
  - Then O must be [the optimal solution for $\{1, \ldots, 3\}$] + {6}



Suppose $O = \{2, 6\}$

on intervals 1, 2, 3, $O' = \{1, 3\}$ beats $\{2\}$

A Recursive Algorithm

Step 0 of the alg

Assume that the intervals are sorted by f, so $f_1 < f_2 < \dots < f_n$.

- Let O be the **optimal** solution
- Case I: Last interval is not in O ($n \notin O$)
  - Then O must be the optimal solution for $\{1, \dots, n-1\}$
- Case 2: Last interval is in O ($n \in O$)
  - Sort intervals by end time so $f_1 < f_2 < \cdots < f_n$
  - [Let $p(i)$ be the last interval not overlapping i]
  - Then O must be [the optimal solution for $\{1, \dots, p(n)\}$] $\cup \{n\}$

Let $OPT(i)$ be the value of the optimal schedule using $\{1, \dots, i\}$

$OPT(0) = 0$

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

Recursive Algorithm I

Assume you have precomputed these.

FindOPT(i):
  If $i = 0$: return 0
  Else: return $\max\{v_i + \text{FindOPT}\big(p(i)\big), \text{FindOPT}(i-1)\}$

- Claim: FindOPT(i) computes $OPT(i)$ $\forall i = 0, 1, \ldots, n$

Proof: Base case $(i=0)$: easy

Inductive Step: if FindOPT(j) = OPT(j) $\forall j < i$

then $\text{FindOPT}(i) = \max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i-1)\}$

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

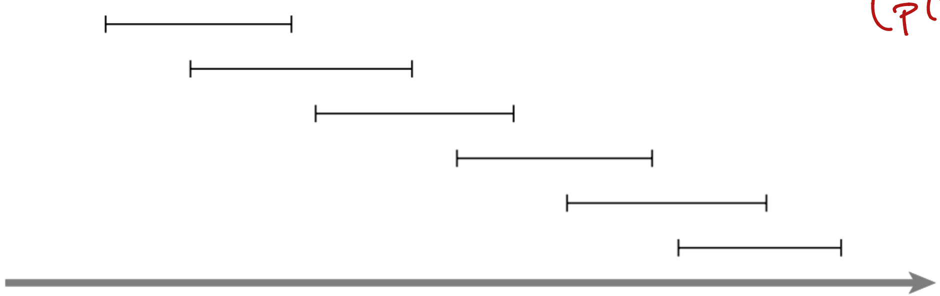
$= \text{OPT}(i)$

Recursive Algorithm I

FindOPT(i):
  If $i = 0$: return 0   FindOPT$(i-2)$
  Else: return $\max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i-1)\}$

- Sad Fact: FindOPT(n) might run in $\geq F(n)$ time

$(p(i) = i-2)$



Makes at least 1.62^n recursive calls.

Recursive Algorithm II: Memoization

M is an array to hold subproblems we've solved. (Initially empty)

MFindOPT(i):

  If $i = 0$: return 0

  ElseIf $M[i]$ is not empty: return $M[i]$

  Else:

    $M[i] \leftarrow \max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i-1)\}$

    return $M[i]$

Mediocre

- ~~Happy~~ fact: $\text{MFindOPT}(n)$ runs in time $O(n)$

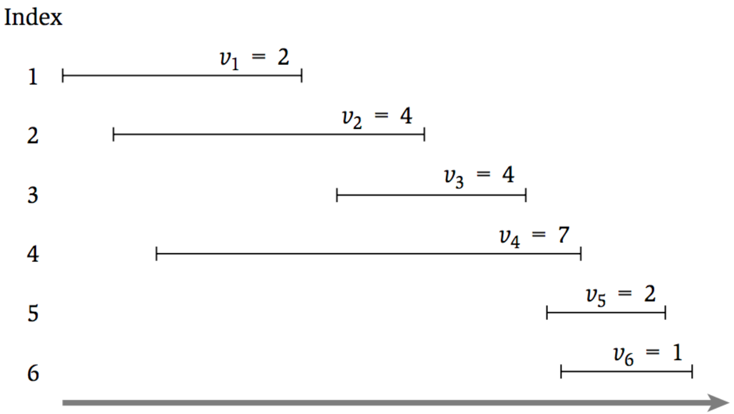
  - Only n elems of M get filled in
  - Every <u>two</u> recursive calls fills <u>one</u> elem

  $\Longrightarrow$ At most 2n recursive calls $T(n) = O(n)$

  also time for sorting

Finding the Solution

Finding the Solution



- $v_6 + OPT(3) = 7$
- $OPT(5) = 8$

- We know $6 \notin O$!

Finding the Solution

```
FindSOL(i):
    If i = 0: return ∅
    Else:
        If v_i + M[p(i)] ≥ M[i − 1]:
            return: {i} ∪ FindSOL(p(i))
        Else
            return: FindSOL(i − 1)
```

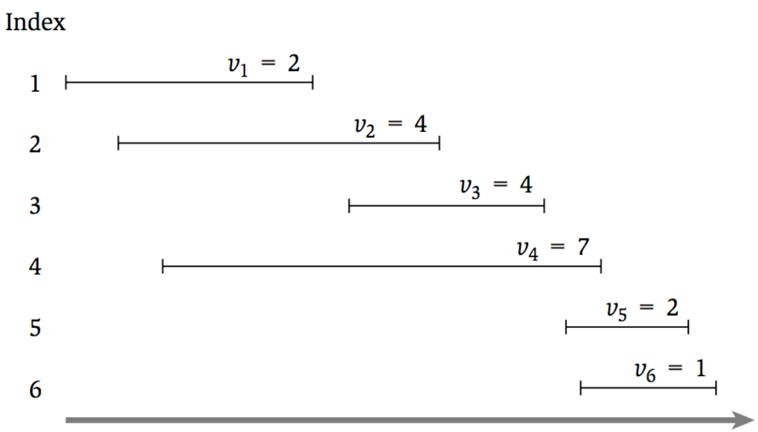
Recursive Algorithm III: Bottom-Up

IterFindOPT(i):

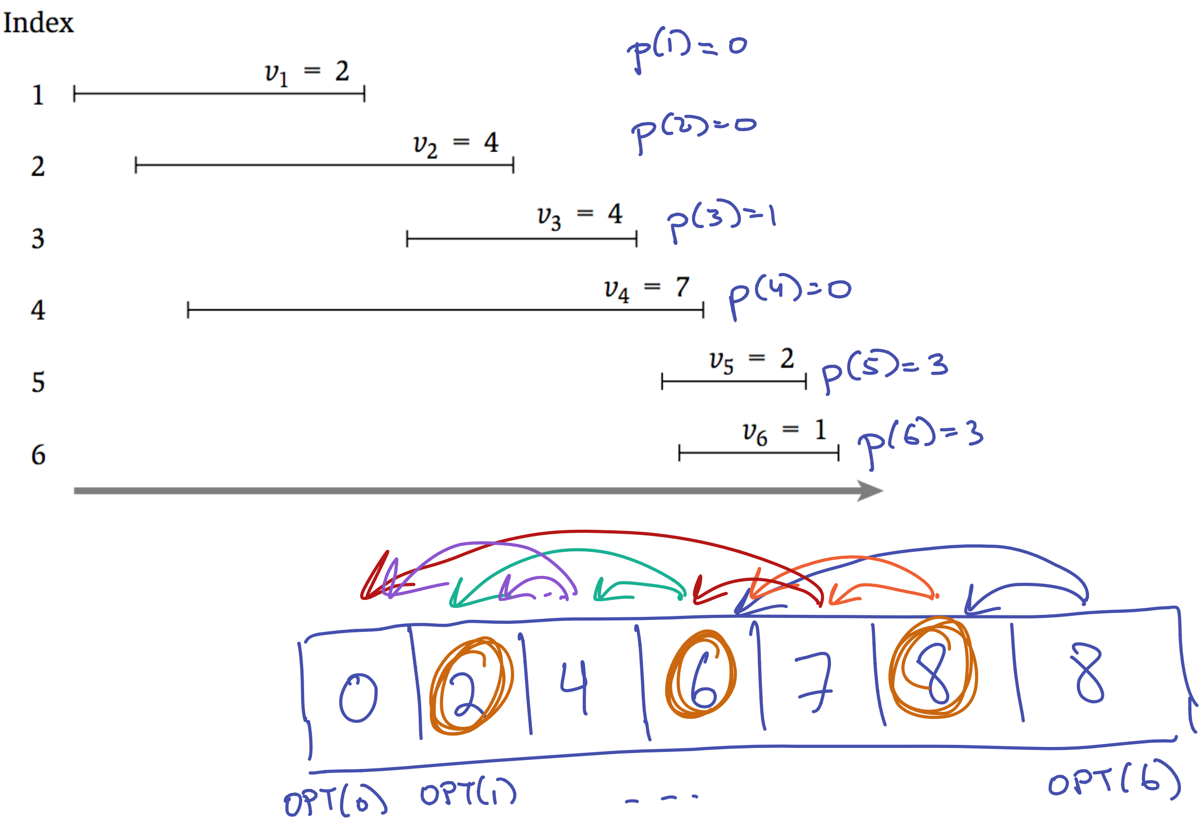
$M[0] \leftarrow 0$

For $i = 1, \dots, n$:

$M[i] \leftarrow \max\{v_i + \text{FindOPT}\big(p(i)\big), \text{FindOPT}(i-1)\}$



Recursive Algorithm III: Bottom-Up



Now You Try

1	$v_1 = 3$	$p(1) = 0$
2	$v_2 = 5$	$p(2) = 1$
3	$v_3 = 9$	$p(3) = 0$
4	$v_4 = 6$	$p(4) = 2$
5	$v_5 = 13$	$p(5) = 1$
6	$v_6 = 3$	$p(6) = 4$

Dynamic Programming Recap

- What did we do:
  - Identified a “small” number of “subproblems”
  - Related these problems via a recurrence
  - Carefully solved the recurrence

- Defining the subproblems and finding a recurrence can be challenging