Objectives

- Dynamic Programming
  - Wrap up weighted interval scheduling

Review: Weighted Interval Scheduling

- Jobs have start time, end time, value/weight
  - Goal: schedule compatible jobs with maximum weight
- What was the key insight to solving the weighted interval scheduling problem?

Binary decision:
- Optimal solution for jobs i through j includes j or doesn't

- How do we pick the solution?

Choose the larger value of
- [choose j and the best solution of compatible jobs] OR
  [best solution if don't pick j]

Why doesn't greedy work?

Weighted Interval Scheduling: Memoization Analysis

Costs?

```
Input: n jobs (associated start time sj, finish time fj, and
value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(n)

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
```


Weighted Interval Scheduling: Memoization Analysis

```
Input: n jobs (associated start time sj, finish time fj, and
value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn    O(n log n)
Compute p(1), p(2), …, p(n)   O(n log n);
                              O(n) w/ some preprocessing
for j = 1 to n
   M[j] = empty   O(n)
M[0] = 0

M-Compute-Opt(n)   O(n)

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
```


Weighted Interval Scheduling: Running Time

- Claim. Memoized version of algorithm takes O(n log n) time
  - Sort by finish time: O(n log n)
  - Computing p(⋅): O(n log n)
  - `M-Compute-Opt(j)`: each invocation takes O(1) time and either
    - (i) returns an existing value `M[j]`
    - (ii) fills in one new entry `M[j]` and makes two recursive calls
  - Progress measure Φ = # nonempty entries of `M[]`
    - (i) initially $\Phi = 0$, throughout $\Phi \le n$
    - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls
  - Running time of `M-Compute-Opt(n)` is O(n). ▪
- Remark. O(n) if jobs are pre-sorted by start and finish times

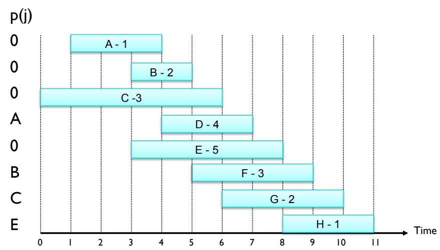
Weighted Interval Scheduling: Finding a Solution

- Dynamic programming algorithms compute ***optimal value***
- What if we want the ***solution*** itself (**not** simply the value)?
- Do some post-processing
  - Looking at M, how do we know which set of intervals were chosen?

M

0	A	B	C	D	E	F	G	H
0	**1**	**2**	**3**	**5**	**5**	**5**	**5**	**6**

Towards Finding a Solution



M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5	5	5	5	6

Weighted Interval Scheduling: Finding a Solution

- Dynamic programming algorithms compute ***optimal value***
- What if we want the ***solution*** itself (**not** simply the value)?
- Do some post-processing

```
M-Compute-Opt(n)
Find-Solution(n)

def Find-Solution(j):
   if j = 0:
      output nothing
   elif v_j + M[p(j)] > M[j-1]:
      print j
      Find-Solution(p(j))
   else:
      Find-Solution(j-1)
```

Runtime?

O(n)

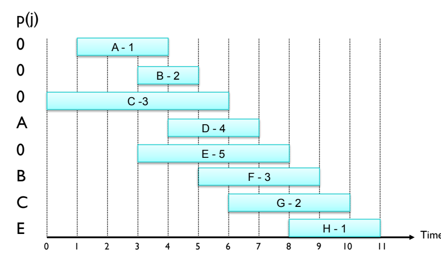
Turning it Around…

- We solved the Fibonacci problem as both recursive/ memoized and an **iterative** algorithm

Can we write this algorithm as an **iterative** solution?

```
Input: n jobs (associated start time s_j, finish time f_j, and
value v_j)

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
```

Towards Iterative Solution…



M	0	A	B	C	D	E	F	G	H

Iterative Solution

- Build up solution from subproblems instead of breaking down

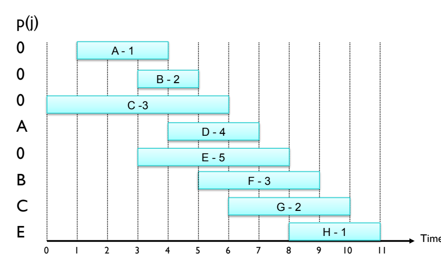
```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

M[0] = 0
for j = 1 to n
   M[j] = max(v_j + M[p(j)], M[j-1])
```

Runtime?

O(n)

- Typically, we'll take iterative approach

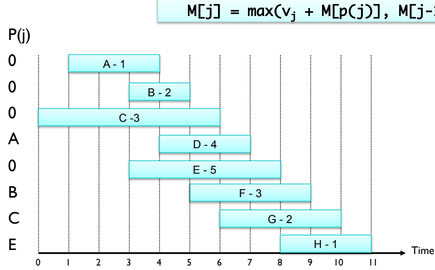
Example: Iteratively



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

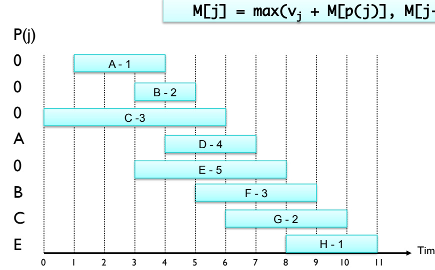
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

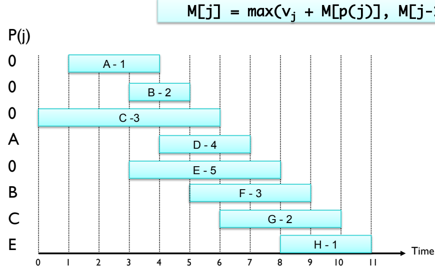
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0	1							

Example: Iteratively

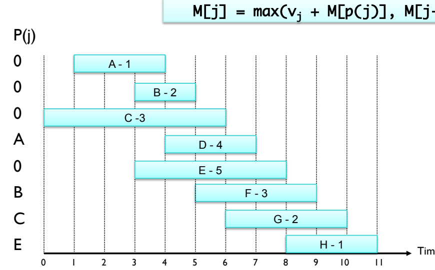
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0	1							

Example: Iteratively

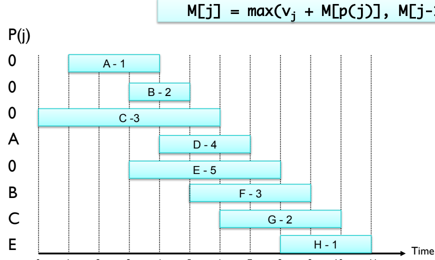
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0	1	2						

Example: Iteratively

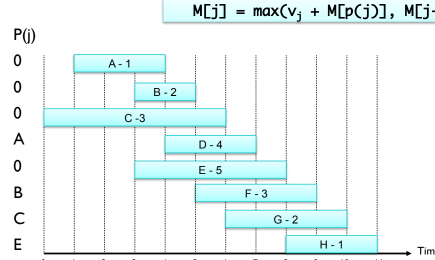
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0	1	2	3					

Example: Iteratively

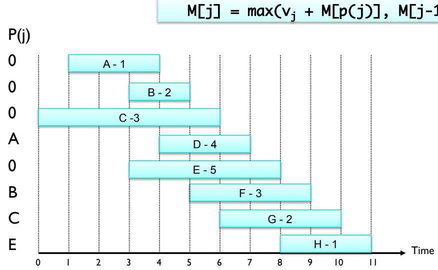
`M[j] = max(v_j + M[p(j)], M[j-1])`



M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5				

Example: Iteratively

M[j] = max(v_j + M[p(j)], M[j-1])

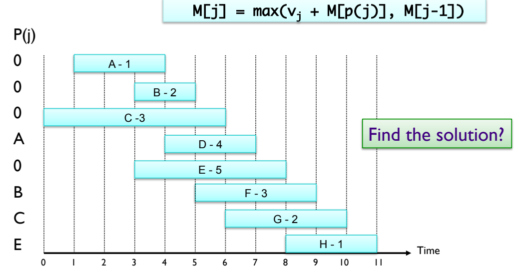


M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5				

And so on….

Example: Iteratively

M[j] = max(v_j + M[p(j)], M[j-1])



M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5	5	5	5	6

Summary: Properties of Problems for DP

- Polynomial number of subproblems
- Solution to original problem can be easily computed from solutions to subproblems
- Natural ordering of subproblems, easy to compute recurrence

Dynamic Programming Process

1. Determine optimal substructure of problem
   - Define the recurrence relation
2. Define algorithm to find the ***value*** of optimal solution
3. Optionally, change algorithm to an ***iterative*** rather than recursive solution
4. Define algorithm to find ***optimal solution***
5. Analyze running time of algorithms

Map to weighted interval scheduling problem

Assignments

- Continue reading Chapter 6
- PS7 due Friday
- Exam 2 handed out Friday