Objectives

- Introduction to Dynamic Programming
  - Fibonacci
  - Weighted interval scheduling

Review

- What was the key to improving the runtime of integer and matrix multiplication operations?

Algorithmic Paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion
- Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem
- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems

Dynamic Programming History

- Richard Bellman pioneered systematic study of dynamic programming in 1950s
- Etymology
  - Dynamic programming = planning over time
    - Not our typical use of "programming"
  - Then-Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

WARMUP: FIBONACCI SEQUENCE

How Would You Solve the Fibonacci Sequence?

- Input: the number of Fibonacci numbers, x
- Output: display the list of the first x Fibonacci numbers

Sequence:

- $F_0=F_1=1$
- $F_n=F_{n-1}+F_{n-2}$

Soln 1: Using a List

- Typical Solution:

```
fibs = []              # create an empty list
fibs.append(1)         # append the first two Fib numbers
fibs.append(1)

for x in xrange(2, N):
       newfib = fibs[x-1]+fibs[x-2]

       fibs.append(newfib)

print fibs             # print out the list
```

Building up solution

Running time? Space cost?

Do we need a whole list?

Soln 2: Using Three Variables

- Only need the solutions to the last two problems (F[k-1], F[k-2])

```
lastNum = 1
twoAgo = 1
print(twoAgo, lastNum, end="")

for n in xrange(2, N):

    nthNum = twoAgo + lastNum
    print(nthNum, end="")

    twoAgo = lastNum
    lastNum = nthNum
```


Soln 3: Recursion

```
def fibonacci(n):
   return fibonacci(n-1) + fibonacci(n-2)
```

- What is the running time of this algorithm?

Dynamic Programming
Memoization Process

- Create a table with the possible inputs
- If the value is in the table, return it, without recomputing it
- Otherwise, call function recursively
  - Add value to table for future reference

How can we apply this template to our Fibonnaci problem?

Memoization Example: Fibonacci

```
memoized_fibonacci(n):
    for i = 1 to n:
       results[i] = -1     # -1 means undefined

    return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
    if results[n] != -1:   # value is defined
       return results[n]
    if n == 0:
       val = 1
    elif n == 1:
       val = 1
    else:
       val = memoized_fib_recurs(results, n-2)
       val = val + memoized_fib_recurs(results, n-1)
    results[n] = val
    return val
```

Runtime?

O(n)

Memoization Example: Fibonacci

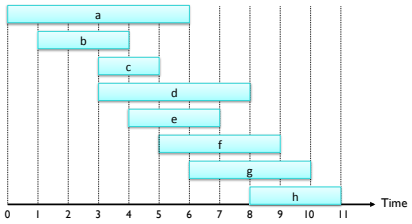
Alternative version…

```
memoized_fibonacci(n):
    for i = 1 to n:
       results[i] = -1 # -1 means undefined
    results[1] = 1
    results[2] = 1

    return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
    if results[n] != -1: # value is defined
       return results[n]

    val = memoized_fib_recurs(results, n-2)
    val = val + memoized_fib_recurs(results, n-1)
    results[n] = val
    return val
```

WEIGHTED INTERVAL SCHEDULING

Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are compatible if they don't overlap
- **Goal**: find maximum weight subset of mutually compatible jobs



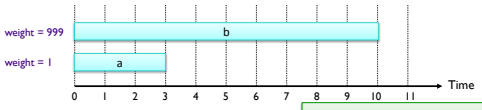
Unweighted Interval Scheduling Review

- Recall. Greedy algorithm works if all weights are 1 (or equivalent).
  - Consider jobs in ascending order of finish time
  - Add job to subset if it is compatible with previously chosen jobs

What happens to Greedy algorithm if we add weights to the problem?

Limitation of Greedy Algorithm

- Recall. Greedy algorithm works if all weights are 1 (or equivalent).
  - Consider jobs in ascending order of finish time
  - Add job to subset if it is compatible with previously chosen jobs
- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed



Any other greedy approaches?

Limitations of Greedy Algorithms

- Need to consider weight
  - No greedy algorithm works
- Need a more complex algorithm to solve problem

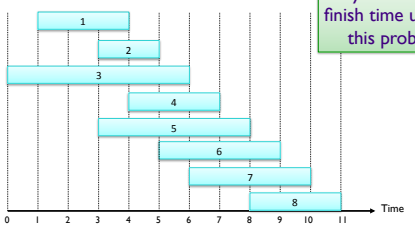
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$

Def. p(j) = largest index $i < j$ such that job i is compatible with j

Ex: p(8) = 5, p(7) = 3, p(2) = 0

Why is ordering by finish time useful in this problem?



Dynamic Programming

- Assume we have an optimal solution
- **OPT(j)** = **value** of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*

What is something *obvious/trivial* we can we say about the optimal solution with respect to job *j*?

Dynamic Programming: Binary Choice

- OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*
  - Case 1: OPT selects job *j*
  - Case 2: OPT does not select job *j*

Explore both of these cases…
- What jobs are in OPT? Which are not?

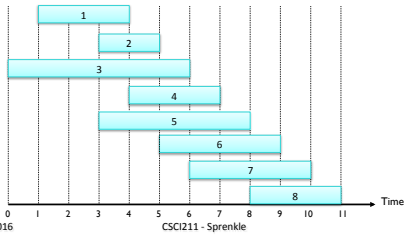
Keep in mind our definition of p

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$

Def. p(j) = largest index $i < j$ such that job i is compatible with j

Ex: p(8) = 5, p(7) = 3, p(2) = 0



Dynamic Programming: Binary Choice

- OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*
  - Case 1: OPT selects job *j*
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
  - Case 2: OPT does **not** select job *j*
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

optimal substructure

Formulate OPT(j) as a recurrence relation

Dynamic Programming: Binary Choice

- OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*
  - Case 1: OPT selects job *j*
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
  - Case 2: OPT does **not** select job *j*
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

Formulate OPT(j) in terms of smaller subproblems
Which should we choose?

Two options: Opt(j) = v_j + Opt(p(j))
Opt(j) = Opt(j-1)

Dynamic Programming: Binary Choice

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., *j*
  - Case 1: OPT selects job *j*
    - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
  - Case 2: OPT does **not** select job *j*
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

$$\text{Opt}(j) = \begin{cases} 0 & j=0 \\ \max\{ v_j + \text{Opt}(p(j)), \text{Opt}(j-1) \} & \text{Otherwise} \end{cases}$$

Basecase
Choose the "better" of the two solutions

Weighted Interval Scheduling: Recursive Algorithm

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

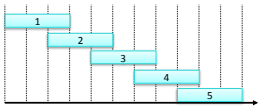
Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$

Compute p(1), p(2), …, p(n) — Closest compatible job

```
Compute-Opt(j):
   if j = 0
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

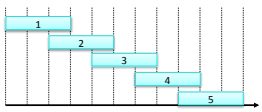
Picks j — Doesn't pick j

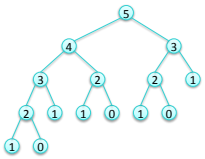
What is the runtime? (Trace for n = 5)



Weighted Interval Scheduling: Brute Force

- Observation. Redundant sub-problems ⇒ exponential algorithms
- Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.





p(1) = 0, p(j) = j-2

Weighted Interval Scheduling: Memoization

- Store results of each sub-problem in a cache; lookup as needed.

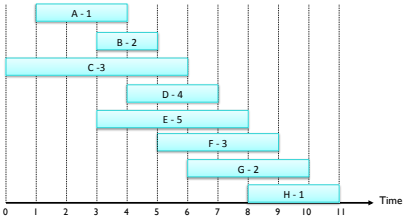
Input: n jobs (associated start time s_j, finish time f_j, and value v_j)

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), …, p(n)

```
for j = 1 to n
   M[j] = empty        <- global array
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]

M-Compute-Opt(n)       <- Call function with initial input
```


Example

- Jobs labeled as name – weight

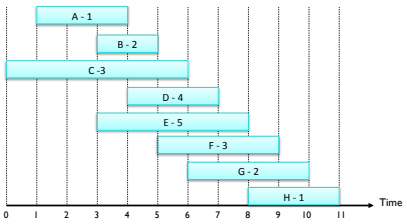


M	0	A	B	C	D	E	F	G	H
	0								

Example

What is the value of p for each job?

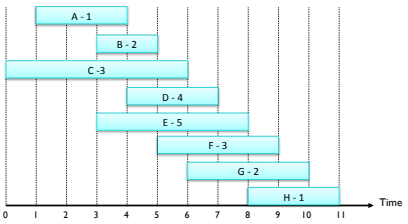
- Jobs labeled as name – weight



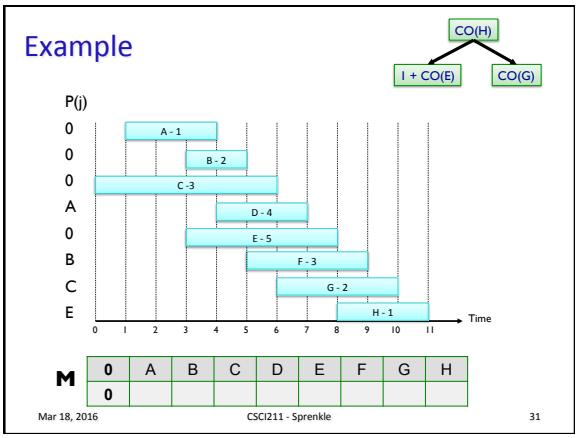
M	0	A	B	C	D	E	F	G	H
	0								

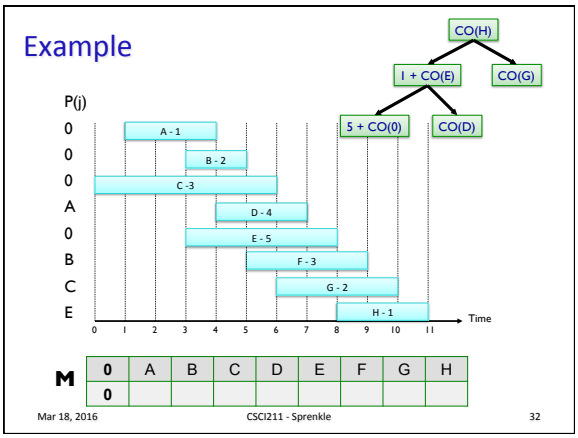
Example

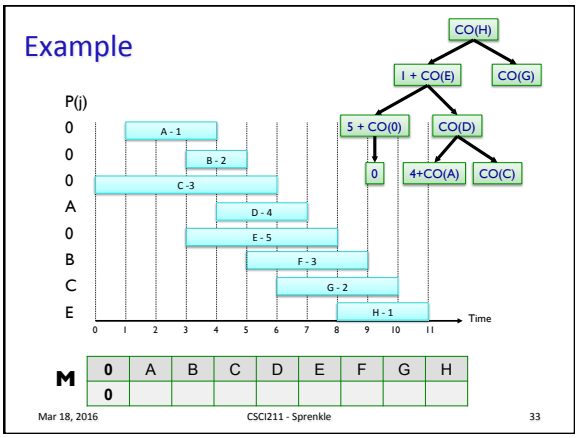
P(j)
0
0
0
A
0
B
C
E

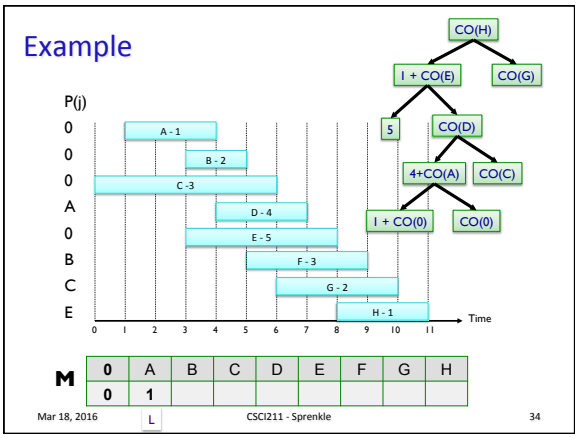


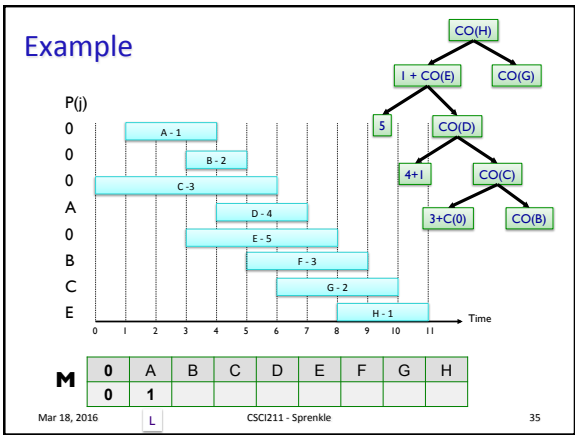
M	0	A	B	C	D	E	F	G	H
	0								

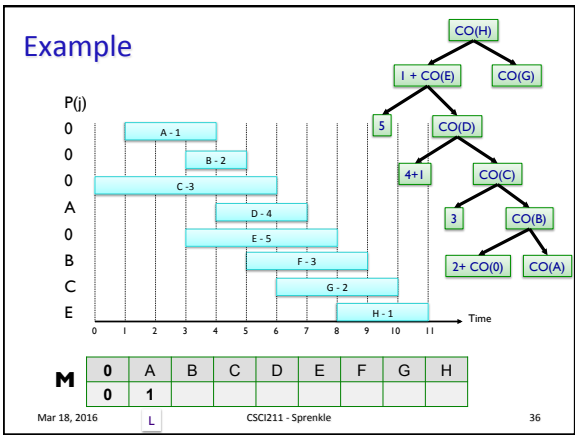

Example
CO(H)
I + CO(E)
CO(G)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0
Mar 18, 2016
CSCI211 - Sprenkle
31

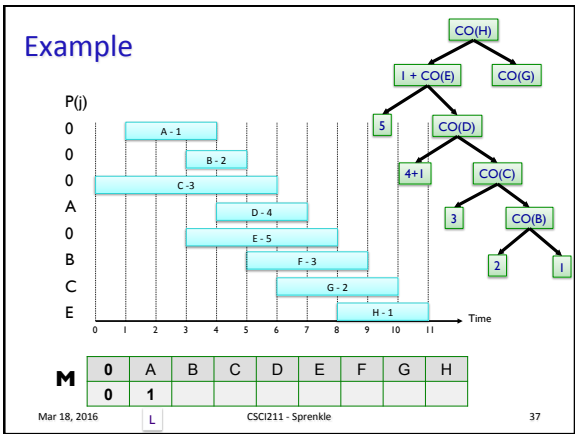

Example
CO(H)
I + CO(E)
CO(G)
5 + CO(0)
CO(D)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0
Mar 18, 2016
CSCI211 - Sprenkle
32

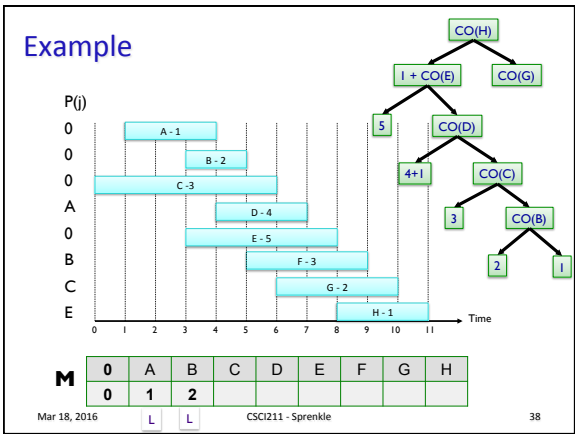

Example
CO(H)
I + CO(E)
CO(G)
5 + CO(0)
CO(D)
0
4+CO(A)
CO(C)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0
Mar 18, 2016
CSCI211 - Sprenkle
33

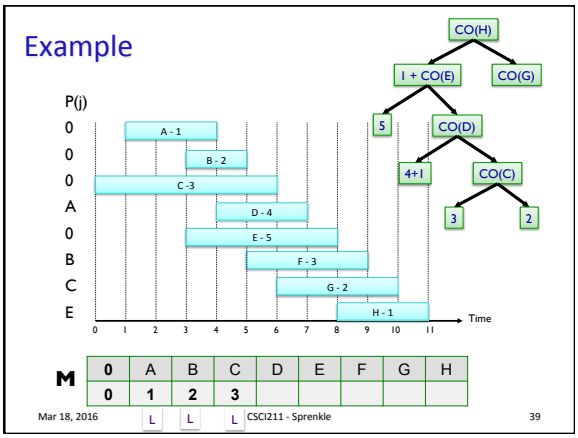

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+CO(A)
CO(C)
I + CO(0)
CO(0)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0 1
Mar 18, 2016
L
CSCI211 - Sprenkle
34

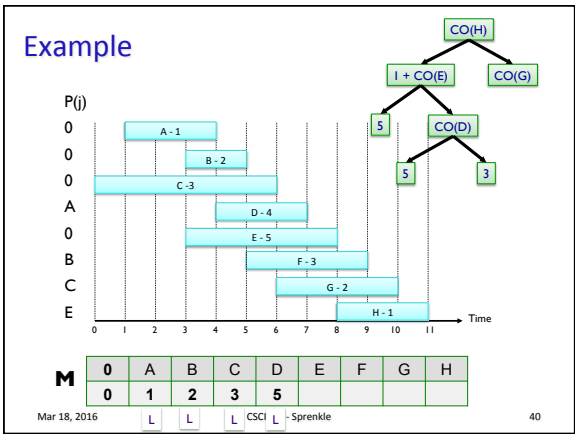

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+I
CO(C)
3+C(0)
CO(B)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0 1
Mar 18, 2016
L
CSCI211 - Sprenkle
35

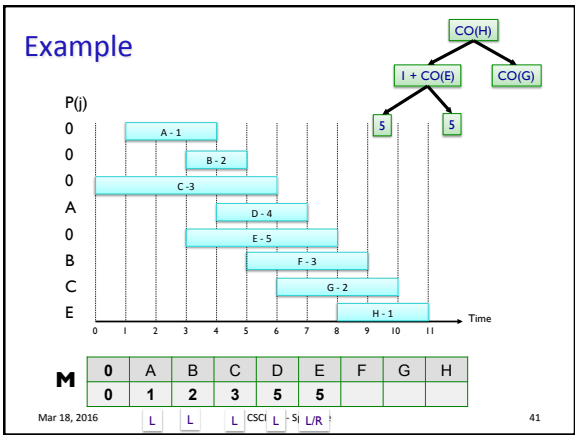

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+I
CO(C)
3
CO(B)
2+ CO(0)
CO(A)
P(j)
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
M
0 A B C D E F G H
0 1
Mar 18, 2016
L
CSCI211 - Sprenkle
36

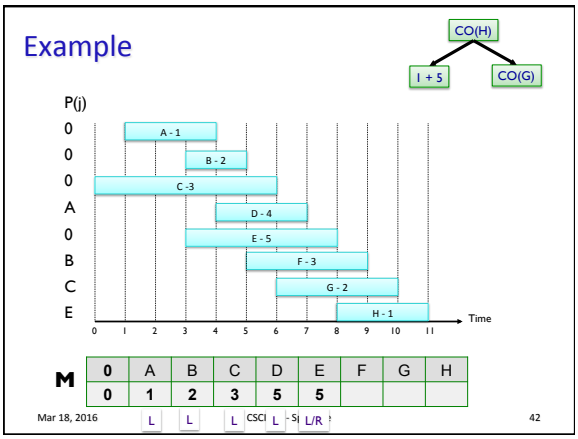

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+I
CO(C)
3
CO(B)
2
I
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1
Mar 18, 2016
L
CSCI211 - Sprenkle
37

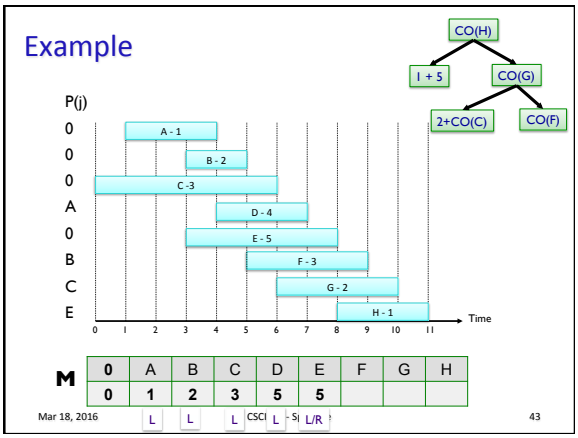

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+I
CO(C)
3
CO(B)
2
I
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1 2
Mar 18, 2016
L
L
CSCI211 - Sprenkle
38

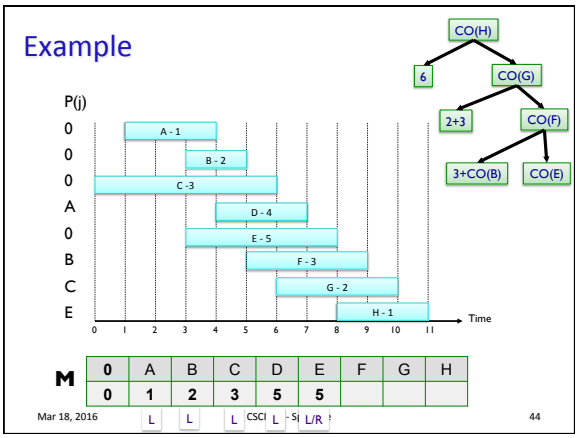

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
4+I
CO(C)
3
2
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1 2 3
Mar 18, 2016
L
L
L
CSCI211 - Sprenkle
39

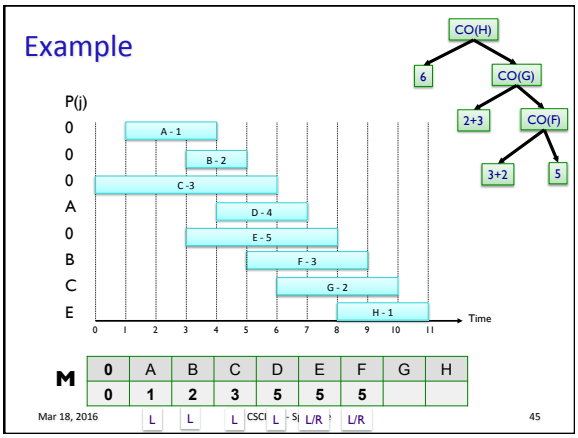

Example
CO(H)
I + CO(E)
CO(G)
5
CO(D)
5
3
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1 2 3 5
Mar 18, 2016
L
L
L
CSCI
L
- Sprenkle
40

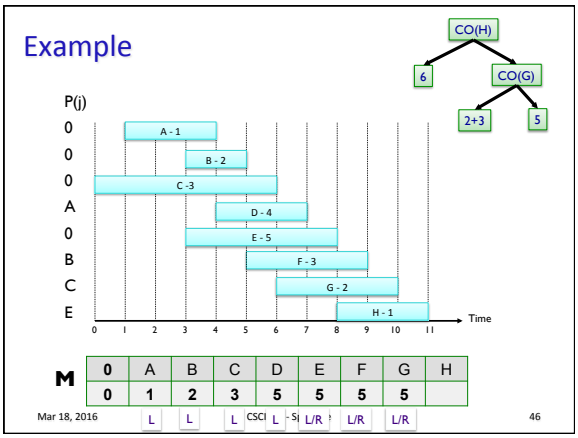

Example
CO(H)
I + CO(E)
CO(G)
5
5
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1 2 3 5 5
Mar 18, 2016
L
L
L
CSCI
L
L/R
41

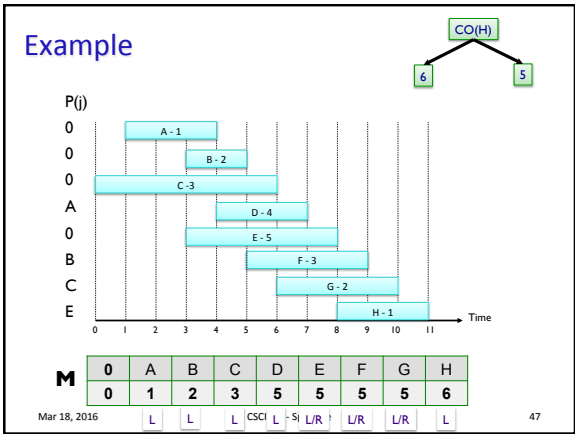

Example
CO(H)
I + 5
CO(G)
P(j)
0
0
0
A
0
B
C
E
A - 1
B - 2
C -3
D - 4
E - 5
F - 3
G - 2
H - 1
Time
0 1 2 3 4 5 6 7 8 9 10 11
M
0 A B C D E F G H
0 1 2 3 5 5
Mar 18, 2016
L
L
L
CSCI
L
L/R
42











Weighted Interval Scheduling: Memoization Analysis

Costs?

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

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$
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]

M-Compute-Opt(n)
```

Weighted Interval Scheduling: Memoization Analysis

```
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   O(n log n)
Compute p(1), p(2), …, p(n)  O(n log n);

for j = 1 to n
   M[j] = empty    O(n)
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]

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


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 Φ = 0, throughout Φ ≤ n
    - (ii) increases Φ by 1 ⇒ 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

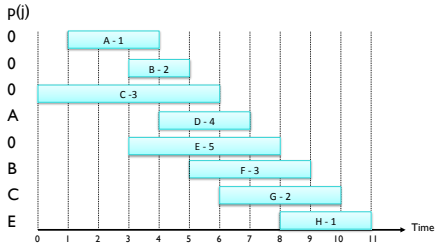
Weighted Interval Scheduling: Finding a Solution

- Dynamic programming algorithms compute ***optimal value***
- What if we want the ***solution*** itself?
  - **Not** simply the optimal 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)

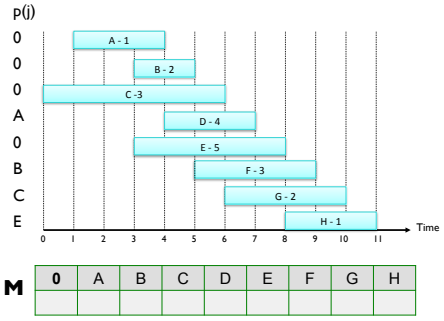
Turning it Around…

- 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]
M-Compute-Opt(n)
```

Towards Iterative Solution…



M	0	A	B	C	D	E	F	G	H

Iterative Solution

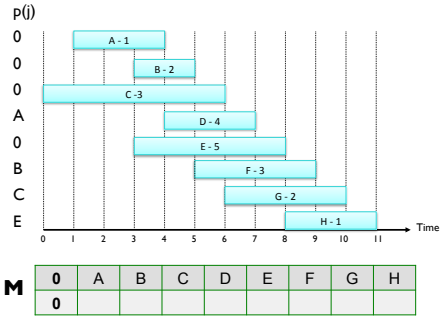
- Build up solution from subproblems instead of breaking down

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)
M[0] = 0
for j = 1 to n
    M[j] = max(vj + M[p(j)], M[j-1])
```

Runtime? O(n)

- Typically, we'll take iterative approach

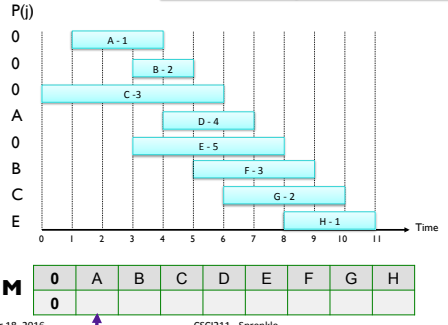
Example: Iteratively



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

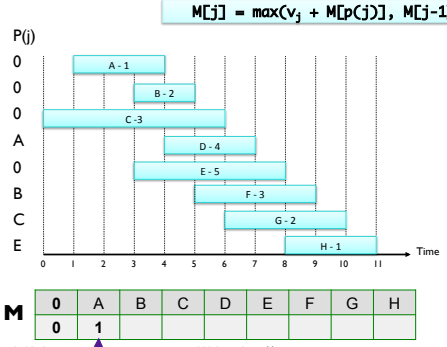
$M[j] = \max(v_j + M[p(j)], M[j-1])$



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

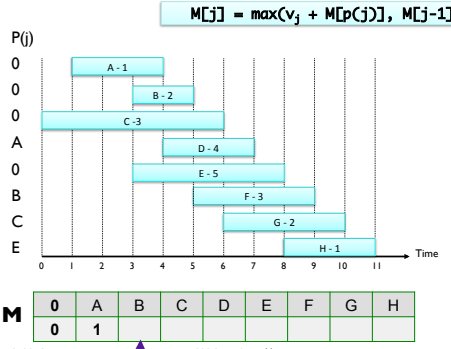
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

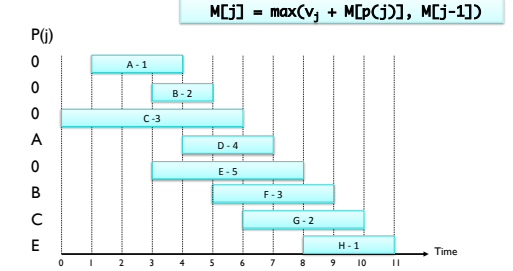
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

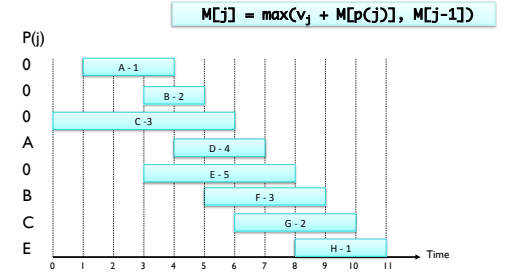
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

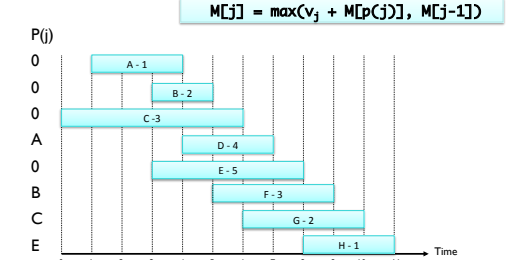
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

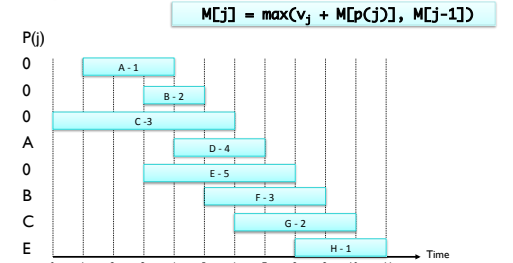
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

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

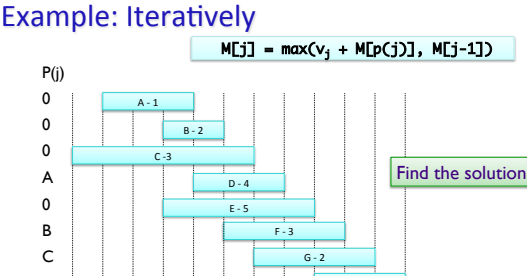


M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5				

And so on….

Example: Iteratively

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



Find the solution?

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