Objectives

- Dynamic Programming
  - Fibonacci Sequence
  - Weighted Interval Scheduling

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"
  - 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)
print fibs[0], fibs[1],
for x in xrange(2, N):
        newfib = fibs[x-1]+fibs[x-2]
        print newfib,
        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,

for n in xrange(2, N):

    nthNum = twoAgo + lastNum
    print nthNum,

    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 j = 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 == 1:
       val = 1
    elif n == 2:
       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

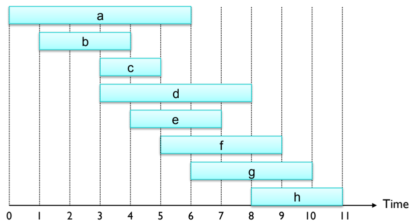
Alternative version…

```
memoized_fibonacci(n):
    for j = 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



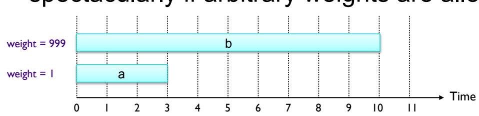
Unweighted Interval Scheduling Review

- 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.
  - 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

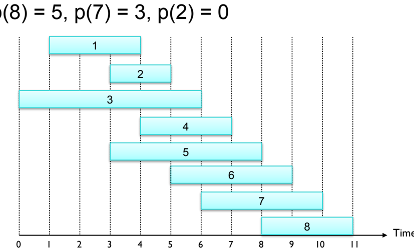


Weighted Interval Scheduling

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

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

- 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* we can we say about the optimal solution with respect to job *j*?

Dynamic Programming: Binary Choice

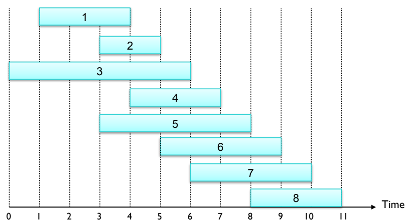
- OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*
  - Case 1: OPT selects job *j*
  - Case 2: OPT does not select job *j*

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: $f_1 \le f_2 \le \ldots \le f_n$
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

optimal substructure

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

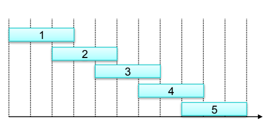
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1)\} & \text{otherwise} \end{cases}$$

Basecase
Choose the "better" of the two solutions

Weighted Interval Scheduling: Recursive Algorithm

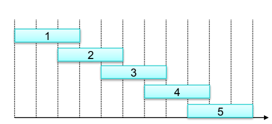
```
Input: n jobs (associated start time sj, finish time fj, and value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn

Compute p(1), p(2), …, p(n)

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

What is the run time?
(Trace for *n* = 5)

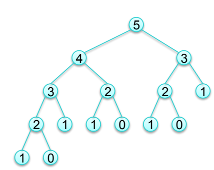


Weighted Interval Scheduling: Brute Force

- 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

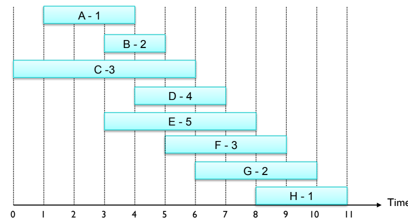
- Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
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 = 2 to n
   M[j] = empty        ← global array
M[1] = 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]
```


Example

What is the value of p for each job?

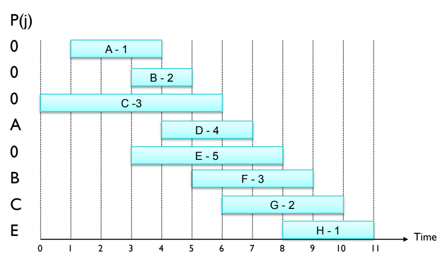
- Jobs labeled with name – weight



M

0	A	B	C	D	E	F	G	H
0								

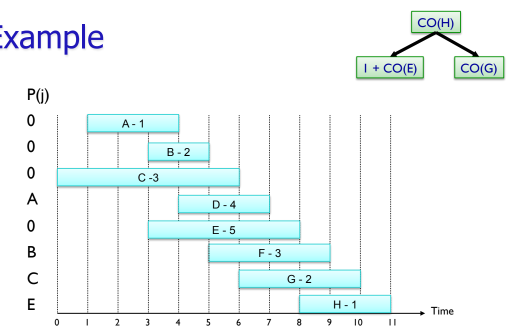
Example



M

0	A	B	C	D	E	F	G	H
0								

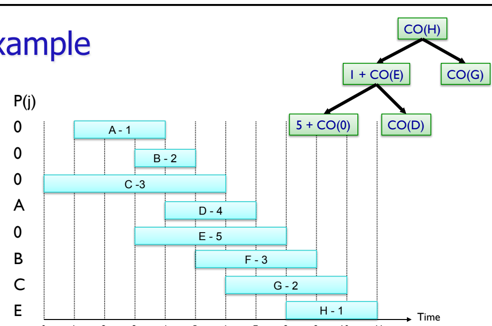
Example



M

0	A	B	C	D	E	F	G	H
0								

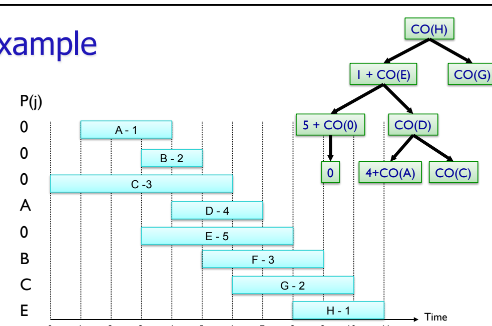
Example



M

0	A	B	C	D	E	F	G	H
0								

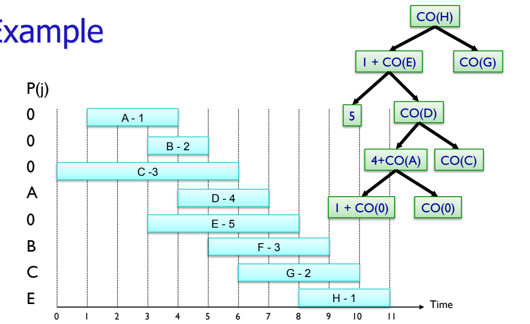
Example



M

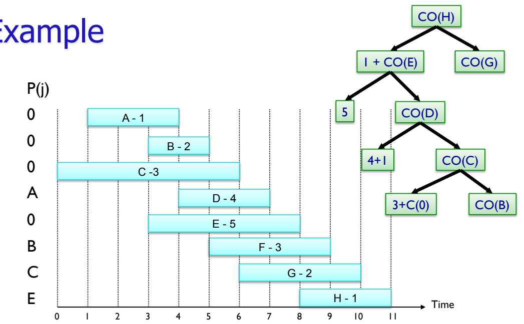
0	A	B	C	D	E	F	G	H
0								

Example



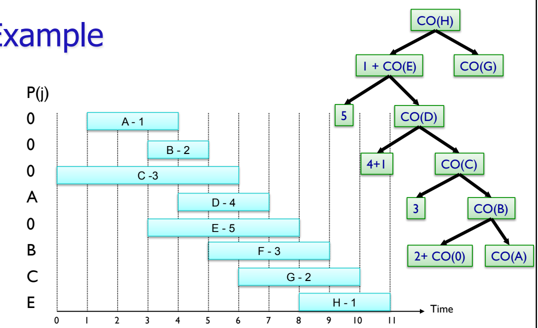
M	0	A	B	C	D	E	F	G	H
	0	1							

Example



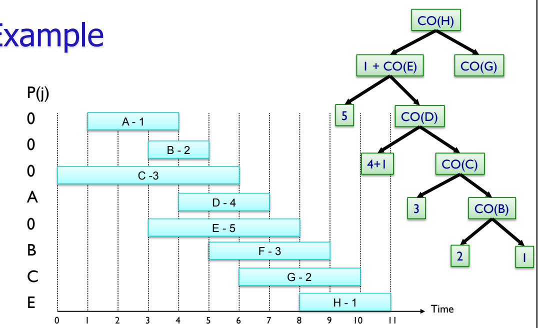
M	0	A	B	C	D	E	F	G	H
	0	1							

Example



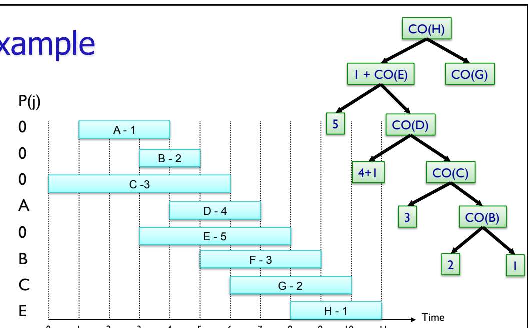
M	0	A	B	C	D	E	F	G	H
	0	1							

Example



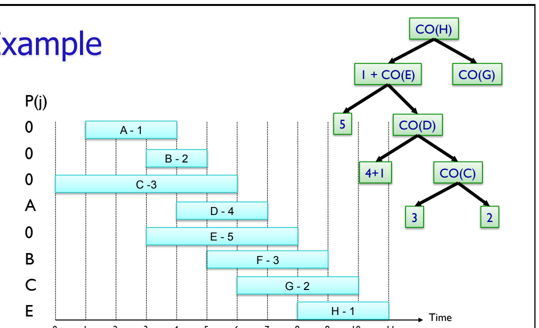
M	0	A	B	C	D	E	F	G	H
	0	1							

Example



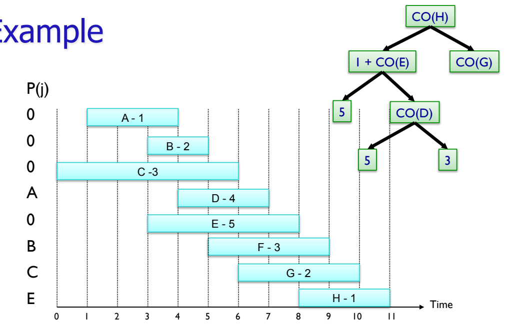
M	0	A	B	C	D	E	F	G	H
	0	1	2						

Example



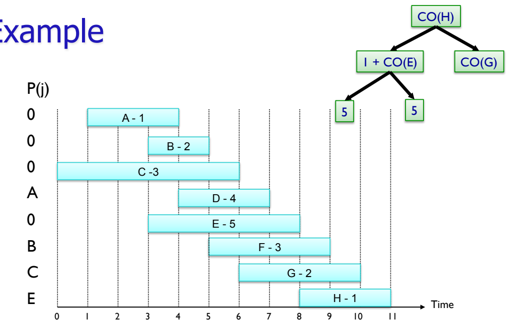
M	0	A	B	C	D	E	F	G	H
	0	1	2	3					

Example



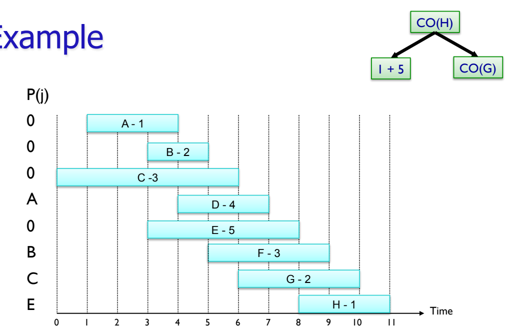
	0	A	B	C	D	E	F	G	H
M	0	1	2	3	5				

Example



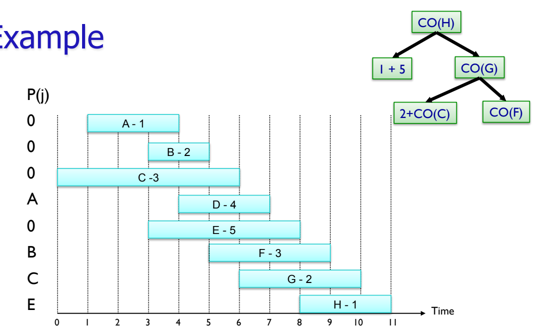
	0	A	B	C	D	E	F	G	H
M	0	1	2	3	5	5			

Example



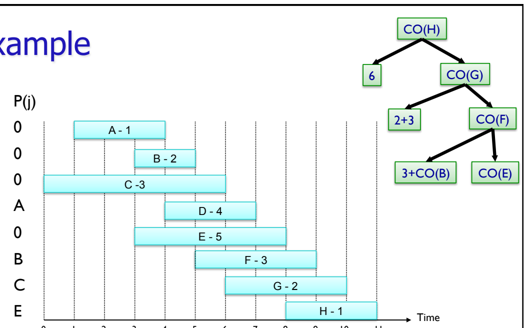
	0	A	B	C	D	E	F	G	H
M	0	1	2	3	5	5			

Example



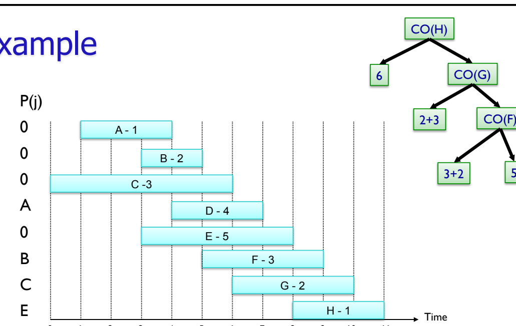
	0	A	B	C	D	E	F	G	H
M	0	1	2	3	5	5			

Example



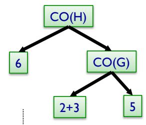
	0	A	B	C	D	E	F	G	H
M	0	1	2	3	5	5			

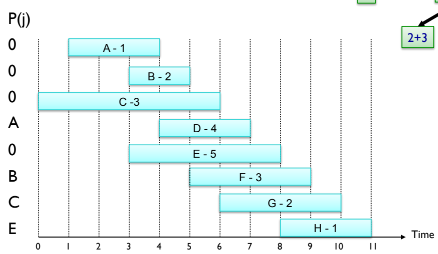
Example



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

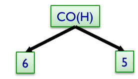
Example

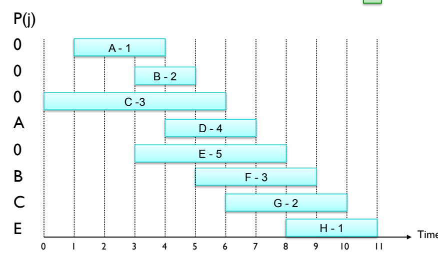




M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5	5	5	5	
		L	L	L	L	L/R	L/R	L/R	

Example





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

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)

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) after sorting by start time
  - `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
  - Overall 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
		L	L	L	L	L/R	L/R	L/R	L

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

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

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

Runtime?

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


Iterative Solution

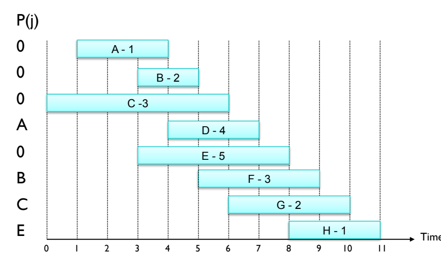
- Build up solution from subproblems instead of breaking down

```
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)

Iterative-Compute-Opt:
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
```

Runtime?

- Typically, approach we'll take

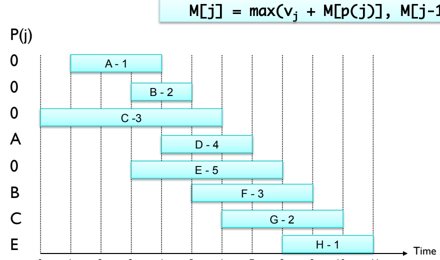
Example: Iteratively



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

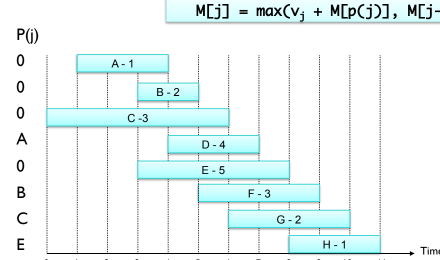
$M[j] = \max(v_j + M[p(j)], M[j-1])$



M	0	A	B	C	D	E	F	G	H
	0								

Example: Iteratively

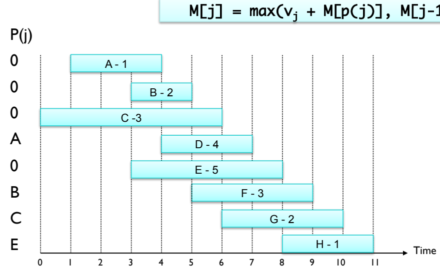
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

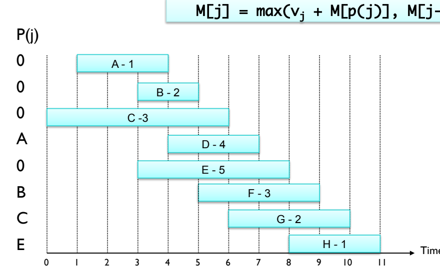
`M[j] = max(v_j + M[p(j)], M[j-1])`



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

Example: Iteratively

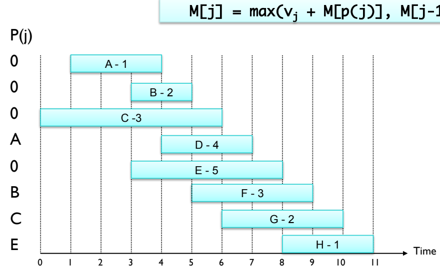
`M[j] = max(v_j + M[p(j)], M[j-1])`



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

Example: Iteratively

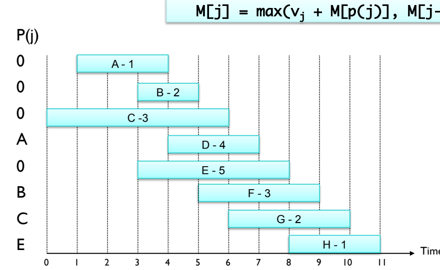
`M[j] = max(v_j + M[p(j)], M[j-1])`



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

Example: Iteratively

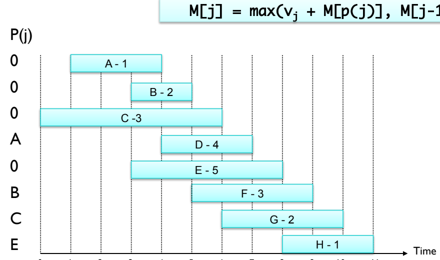
`M[j] = max(v_j + M[p(j)], M[j-1])`



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])`

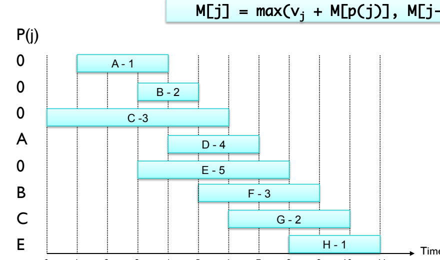


M	0	A	B	C	D	E	F	G	H
	0	1	2	3	5				

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

Assignments

- Finish reading Chapter 5, start Chapter 6
  - 5.5
  - 6 – front matter, 6.1
- PS6 due Friday