Objectives

- Dynamic Programming
  - ➢ Weighted Interval Scheduling

Review: Algorithmic Paradigms

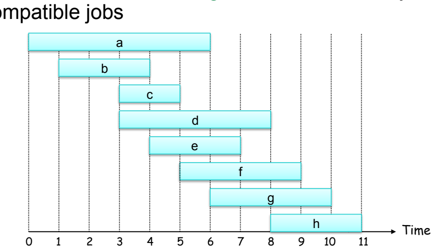
- Greedy. Build up a solution incrementally, myopically optimizing some local criterion
- Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem
- **Dynamic programming.** Break up a problem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

Review: 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

Review: 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

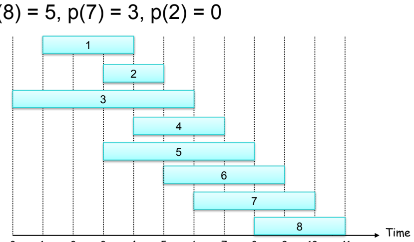


Weighted Interval Scheduling

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

- Notation. 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

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

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

Dynamic Programming: Binary Choice

- Notation. 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

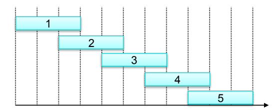
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Choose the better of the two solutions

Weighted Interval Scheduling: Recursive Algorithm

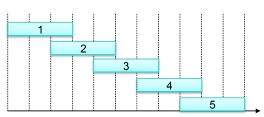
```
Input: n jobs (associated start time s_j, finish time f_j, and value v_j)

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

Compute p(1), p(2), …, p(n)

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

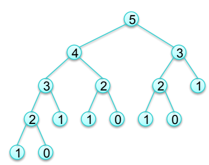
What is the run time?
(Trace for *n* = 5)



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

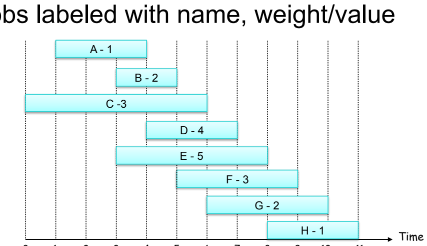
Weighted Interval Scheduling: Memoization

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

```
Input: n jobs (associated start time 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 = 2 to n
   M[j] = empty        ← global array
M[1] = 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]
```


Example

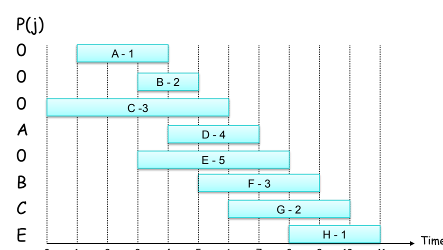
- Jobs labeled with name, weight/value



M

0	A	B	C	D	E	F	G	H
0								

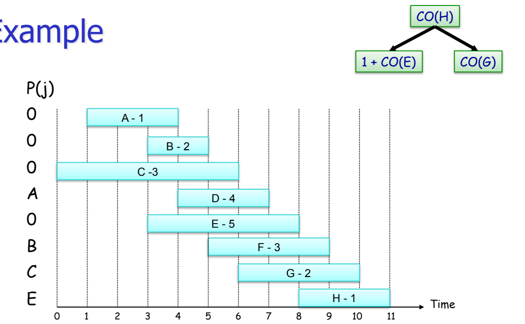
Example



M

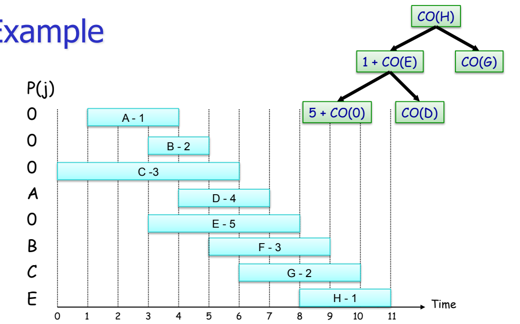
0	A	B	C	D	E	F	G	H
0								

Example



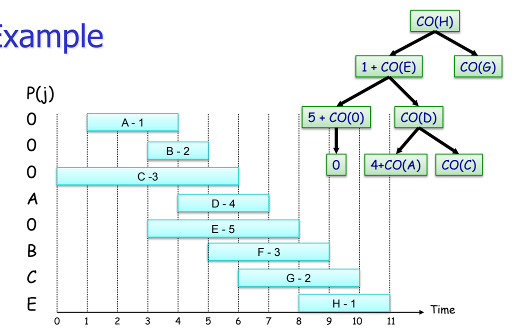
M	0	A	B	C	D	E	F	G	H
	0								

Example



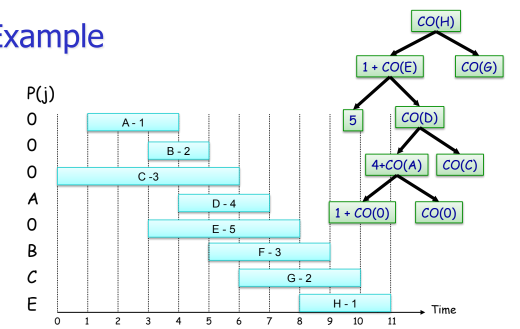
M	0	A	B	C	D	E	F	G	H
	0								

Example



M	0	A	B	C	D	E	F	G	H
	0								

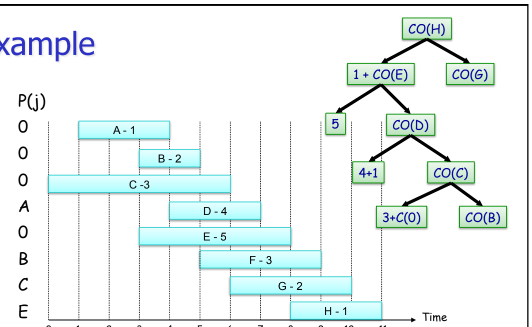
Example



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

Mar 10, 2010 L CSCI211 - Sprenkle 16

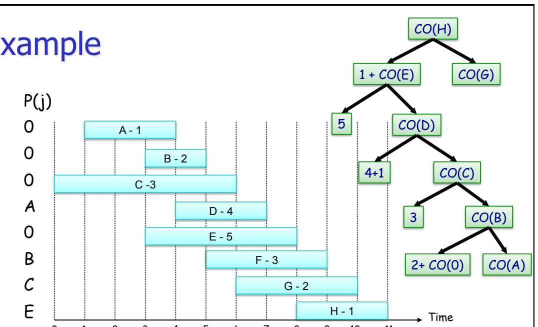
Example



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

Mar 10, 2010 L CSCI211 - Sprenkle 17

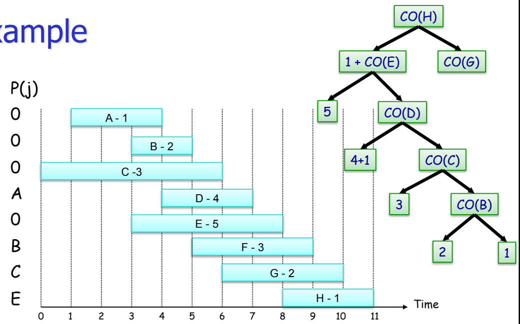
Example



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

Mar 10, 2010 L CSCI211 - Sprenkle 18

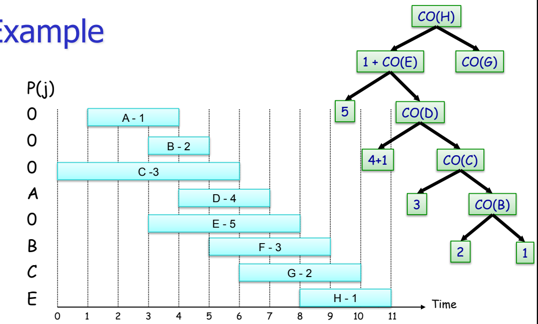
Example



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

L

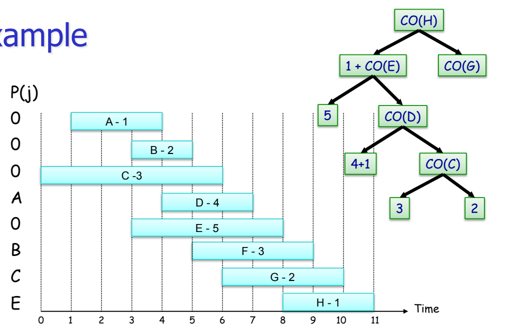
Example



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

L L

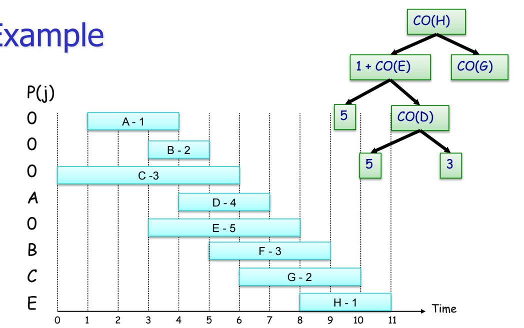
Example



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

L L L

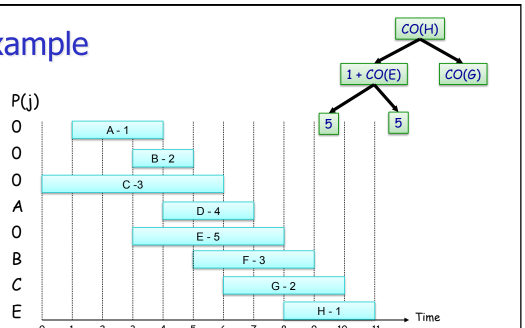
Example



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

L L L L

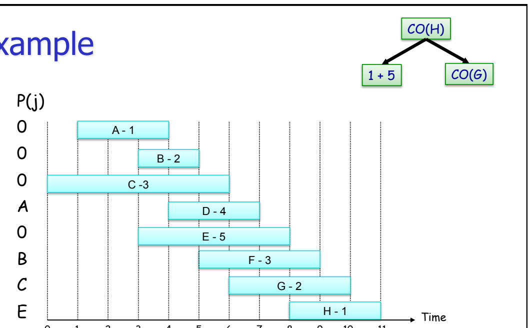
Example



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

L L L L L/R

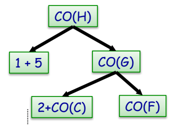
Example

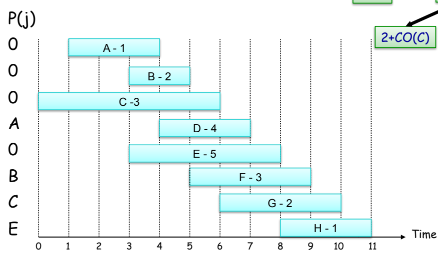


M	0	A	B	C	D	E	F	G	H
	0	**1**	**2**	**3**	**5**	**5**			

L L L L L/R

Example

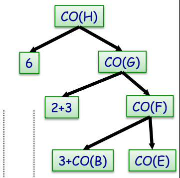


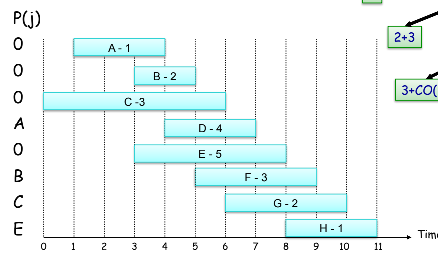


M

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

Example

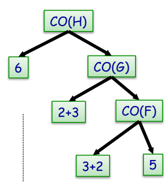


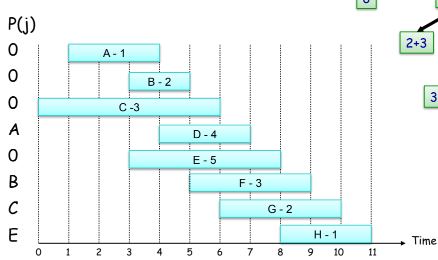


M

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

Example

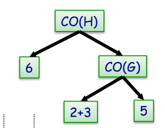


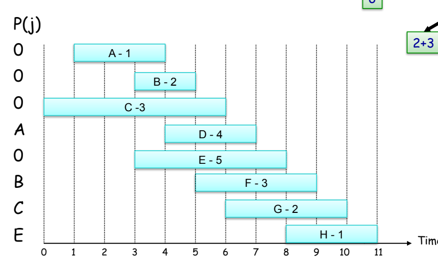


M

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

Example

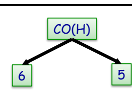


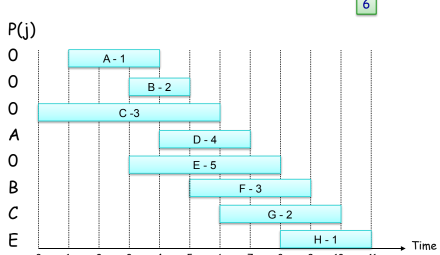


M

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 s_j, finish time f_j, and value v_j)

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$
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(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 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                                  O(n log 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]
```


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 v_j + 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 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]
```

Iterative Solution

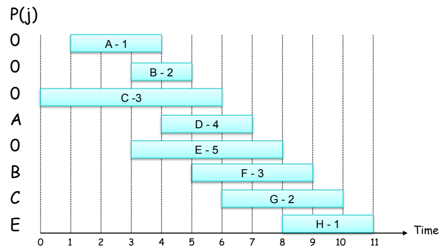
- Build up solution from subproblems instead of breaking down

```
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)

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

Runtime?

- Typically, approach we'll take

Example: Iteratively

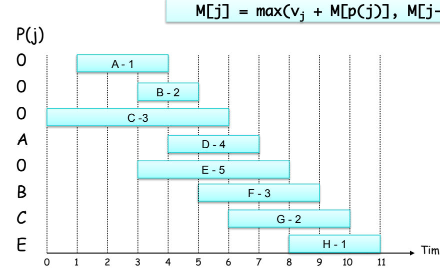


M	0	A	B	C	D	E	F	G	H
	0								

Mar 10, 2010 CSCI211 - Sprenkle 37

Example: Iteratively

M[j] = max(v$_j$ + M[p(j)], M[j-1])

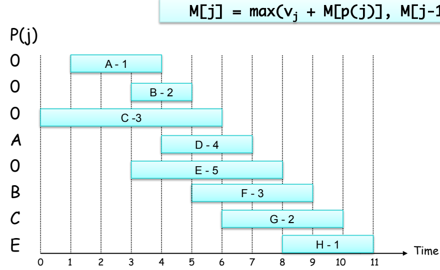


M	0	A	B	C	D	E	F	G	H
	0								

Mar 10, 2010 CSCI211 - Sprenkle 38

Example: Iteratively

M[j] = max(v$_j$ + M[p(j)], M[j-1])

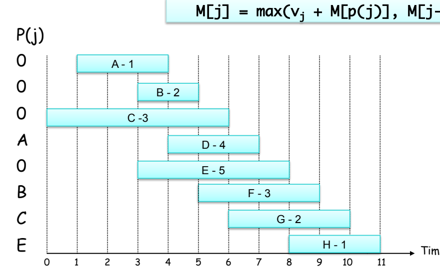


M	0	A	B	C	D	E	F	G	H
	0	1							

Mar 10, 2010 CSCI211 - Sprenkle 39

Example: Iteratively

M[j] = max(v$_j$ + M[p(j)], M[j-1])

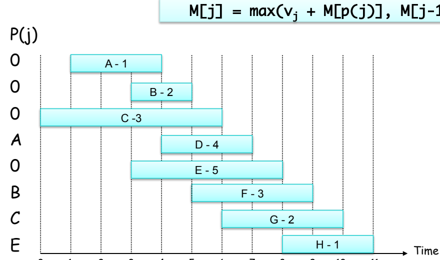


M	0	A	B	C	D	E	F	G	H
	0	1							

Mar 10, 2010 CSCI211 - Sprenkle 40

Example: Iteratively

M[j] = max(v$_j$ + M[p(j)], M[j-1])

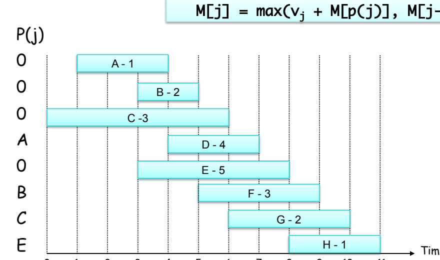


M	0	A	B	C	D	E	F	G	H
	0	1	2						

Mar 10, 2010 CSCI211 - Sprenkle 41

Example: Iteratively

M[j] = max(v$_j$ + M[p(j)], M[j-1])

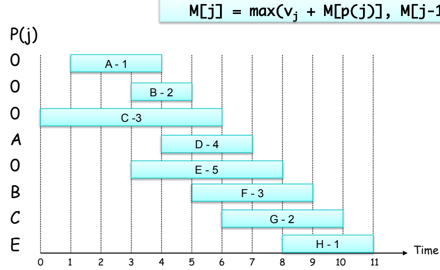


M	0	A	B	C	D	E	F	G	H
	0	1	2	3					

Mar 10, 2010 CSCI211 - Sprenkle 42

Example: Iteratively

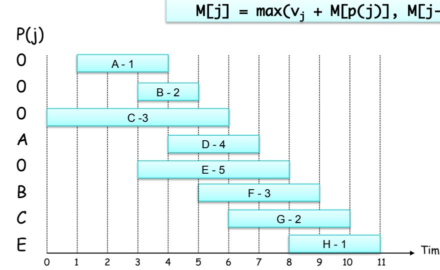
$M[j] = \max(v_j + M[p(j)], M[j-1])$



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

Example: Iteratively

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

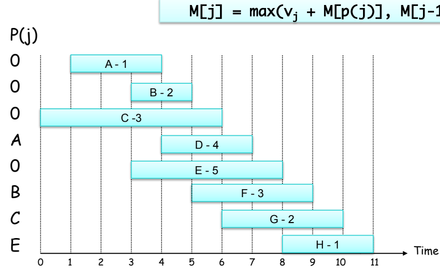


M	0	A	B	C	D	E	F	G	H
	0	**1**	**2**	**3**	**5**				

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