Agenda for today

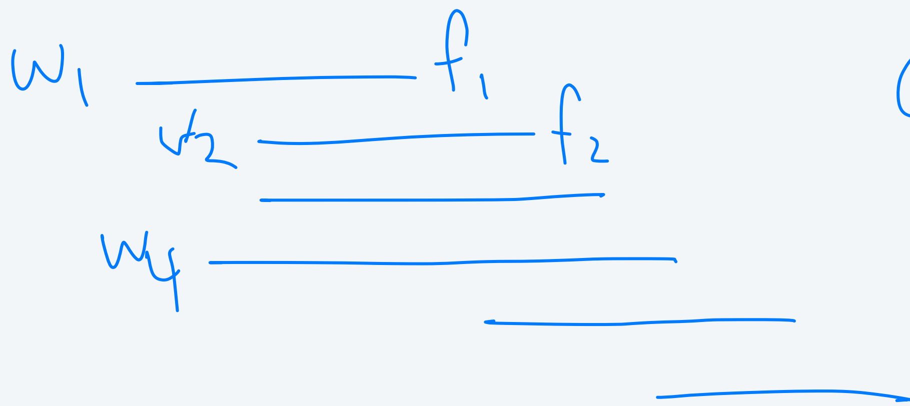
Reminder about memoization and writing efficient algorithms using recurrence relations

Backtracking to find optimal choices (not just value)

Non-recursive versions of dynamic programming algorithms

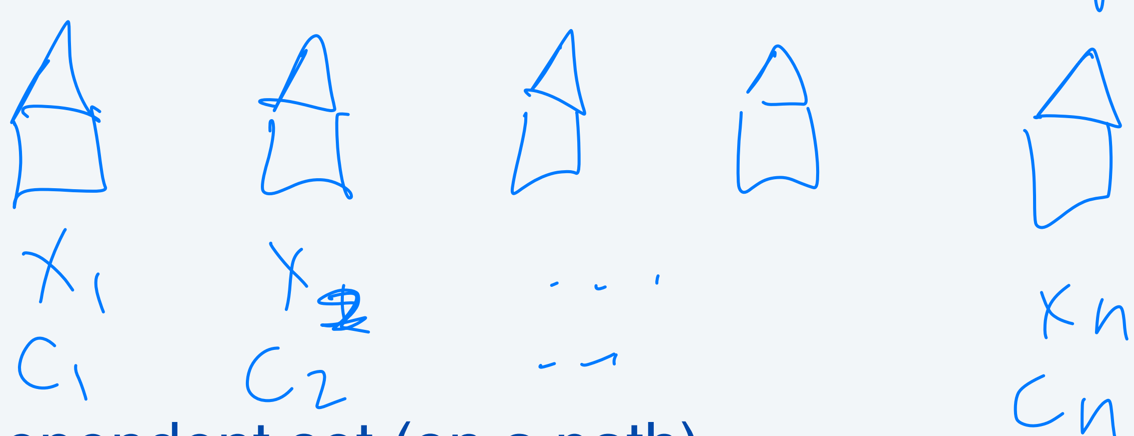
We have seen 3 problems

Weighted interval



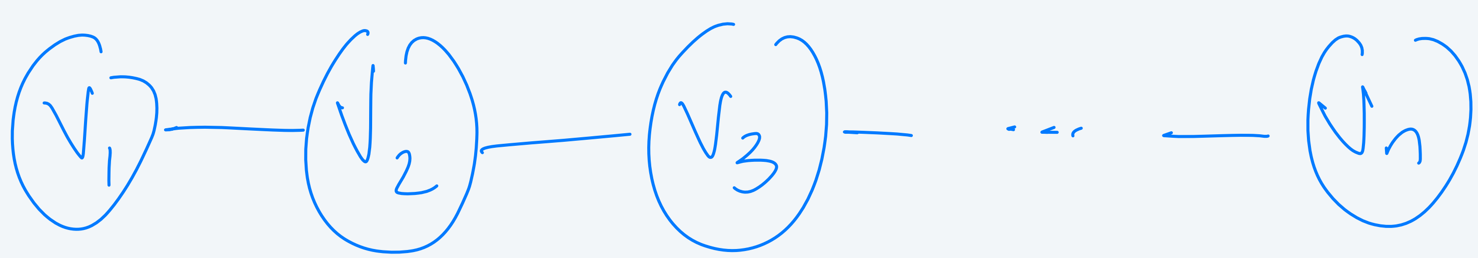
$$opt(j) = \begin{cases} 0 & \text{if } j=0 \\ \max\{opt(j-1), \; w_j + opt(p(j))\} & \text{if } j>0 \end{cases}$$

Max candy



$$opt(j) = \begin{cases} 0 & \text{if } j=0 \\ \max\{opt(j-1), \; c_j + opt(p(j))\} & \text{if } j>0 \end{cases}$$

Independent set (on a path)



$$opt(j) = \begin{cases} 0 & \text{if } j=0 \\ v_1 & \text{if } j=1 \\ \max\{opt(j-1), \; v_j + opt(j-2)\} & \text{if } j>1 \end{cases}$$

What does an algorithm look like for weighted interval problem?

SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \le f_2 \le \ldots \le f_n$. $n \log n$

Compute $p[1], p[2], \ldots, p[n]$ via binary search. $n \log n$

RETURN COMPUTE-OPT(n). $\sim 2^n$

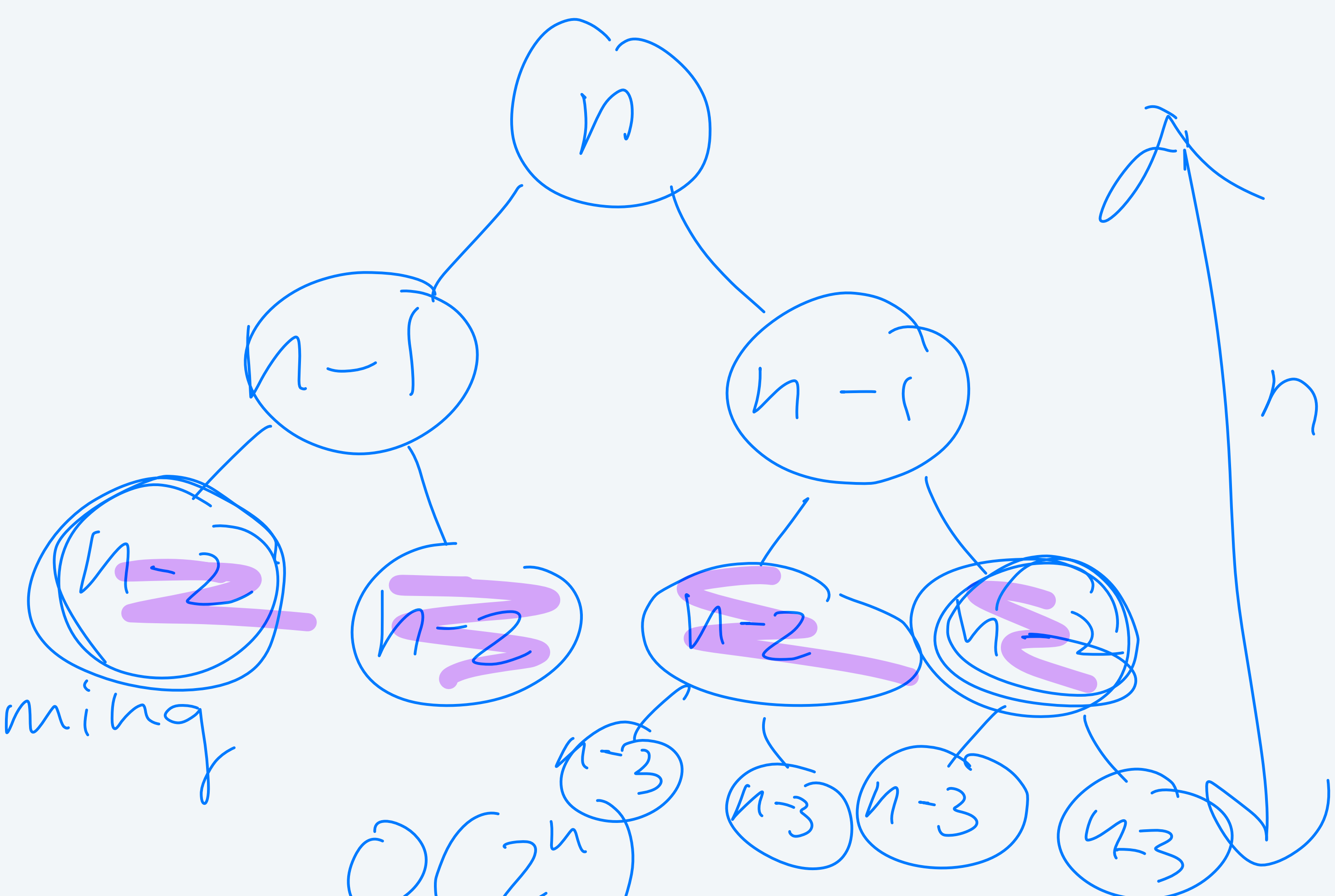
COMPUTE-OPT(j)

IF $(j = 0)$

  RETURN 0.

ELSE

  RETURN max {COMPUTE-OPT$(j-1)$, w_j + COMPUTE-OPT$(p[j])$ }.



$O(2^n)$

Dynamic Programming

Memoization

Caching

What does an algorithm look like for weighted interval problem?

SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

M-SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$. — n log n

Compute $p[1], p[2], \ldots, p[n]$ via binary search. n log n

M[0] = 0 (global) — n

M-Compute-Opt(n) — O(n)

return M[n] — constant

COMPUTE-OPT(j)

IF $(j = 0)$

RETURN 0.

ELSE

RETURN max {COMPUTE-OPT$(j-1)$, w_j + COMPUTE-OPT$(p[j])$ }.

M-COMPUTE-OPT(j)

if M[j] is already filled:

return M[j]

else:

M[j] = max {M-comp-opt(j-1), w_j + M-comp-opt(p[j])}

fill M

overall

O(n log n)

What does an algorithm look like for weighted interval problem?

SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF $(j = 0)$

    RETURN 0.

ELSE

    RETURN max {COMPUTE-OPT$(j-1)$, w_j + COMPUTE-OPT$(p[j])$ }.

$O(2^n)$

M-SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

M[0] = 0 (make this global)

COMPUTE-OPT(n).

Return M[n]

M-COMPUTE-OPT(j)

IF (M[j] ~~uninitialized~~ initialized)

    RETURN M[j]

ELSE

    M[j] = max {COMPUTE-OPT$(j-1)$, w_j + COMPUTE-OPT$(p[j])$ }.

$O(n \log n)$ (overall, because of sorting—M-Compute-Opt is $O(n)$)

What does an algorithm look like for max candy problem?

M-MAX-CANDY $(n, x_1, \ldots, x_n, c_1, \ldots, c_n)$

Sort houses by distance and renumber so that $x_1 \leq x_2 \leq \ldots \leq x_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

M[0] = 0 (make this global)

COMPUTE-OPT(n).

Return M[n]

only do 1 and 2 !!!

M-COMPUTE-OPT(j)

IF (M[j] ~~uninitialized~~ initialized)

RETURN M[j]

ELSE

M[j] = max {COMPUTE-OPT($j-1$), c_j + COMPUTE-OPT($p[j]$) }.

your turn w/ indep. set

notice that your input is sorted!

What does an algorithm look like for max candy problem?

M-MAX-CANDY $(n, x_1, \ldots, x_n, c_1, \ldots, c_n)$

Sort houses by distance and renumber so that $x_1 \leq x_2 \leq \ldots \leq x_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

M[0] = 0 (make this global)

COMPUTE-OPT(n).

Return M[n]

Runtime?

M-COMPUTE-OPT(j)

IF (M[j] uninitialized)

    RETURN M[j]

ELSE

    M[j] = max {COMPUTE-OPT$(j-1)$, c_j + COMPUTE-OPT$(p[j])$ }.

Your turn, with independent set on a path

Notice that the input is already sorted

What does an algorithm look like for independent set problem?

M-INDEPENDENT-SET ($v_1, \ldots, v_n$)

M[0] = 0 (make this global)

M[1] = ~~…~~ V_1

O(n)

M- COMPUTE-OPT(n).

O(n)

Return M[n]

constant

M-COMPUTE-OPT(j)

IF (M[j] ~~uninitialized~~ initialized)

RETURN M[j]

ELSE

M[j] = max {COMPUTE-OPT(j – 1), v_j + COMPUTE-OPT(j-2) }.

need to make sure j>1

M-compute-opt(n) going to use <2n rec. calls

Non-recursive algorithms

M-SCHEDULE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

M[0] = 0 (make this global)

COMPUTE-OPT(n).

Return M[n]

M-COMPUTE-OPT(j)

IF (M[j] uninitialized)

RETURN M[j]

ELSE

M[j] = max {COMPUTE-OPT($j-1$), w_j + COMPUTE-OPT($p[j]$) }.

NR-SCHEDULE$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

$M[0] \leftarrow 0$.

FOR $j = 1$ TO n

$M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}$.

memoized weighted interval scheduling

Non-recursive algorithms

NR-SCHEDULE($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$. ✓ — $n \log n$

Compute $p[1], p[2], \ldots, p[n]$. ✓ — $n \log n$

$M[0] \leftarrow 0$.

FOR $j = 1$ TO n — n

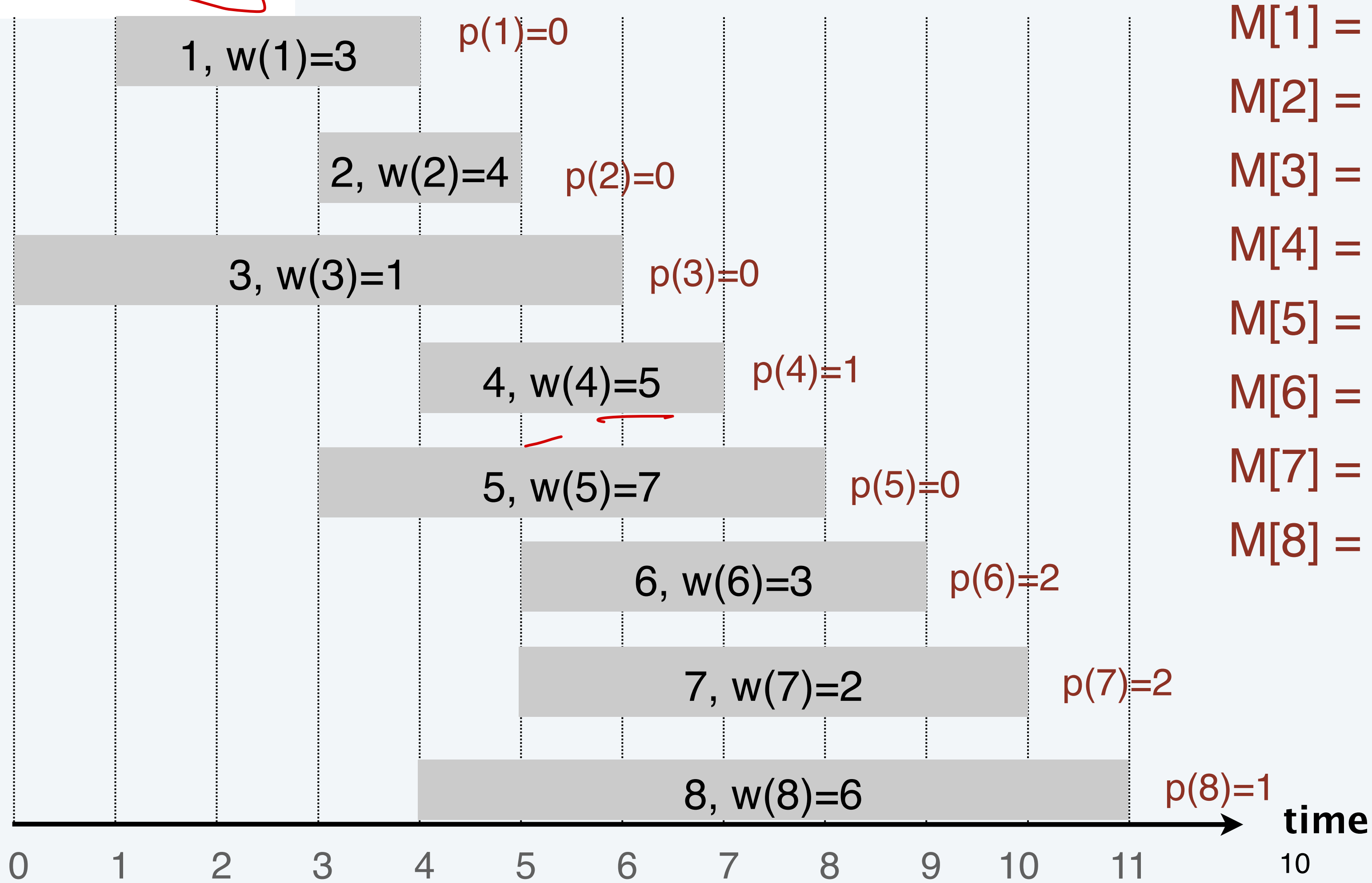
  $M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}$. — n

Let's trace through

$j=1$ $\max\{ M[0], w_1 + M[0] \}$

$j=2$ $\max\{4, 3\}$

$j=3$ $\max\{4, 1\}$



M[0] = 0
M[1] = 3
M[2] = 4
M[3] = 4
M[4] =
M[5] =
M[6] =
M[7] =
M[8] =

Non-recursive algorithms

M-MAX-CANDY $(n, x_1, \ldots, x_n, c_1, \ldots, c_n)$

Sort houses by distance and renumber so that $x_1 \leq x_2 \leq \ldots \leq x_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

M[0] = 0 (make this global)

COMPUTE-OPT(n).

Return M[n]

M-COMPUTE-OPT(j)

IF (M[j] uninitialized)

  RETURN M[j]

ELSE

  M[j] = max {COMPUTE-OPT($j-1$), c_j + COMPUTE-OPT($p[j]$) }.

IS $v_1, v_2, \ldots, v_n$

~~NR-MAX-CANDY~~ $(\cancel{n, x_1, \ldots, x_n, c_1, \ldots, c_n})$

~~Sort houses by distance and renumber so that~~ $\cancel{x_1 \leq x_2 \leq \ldots \leq x_n}$.

~~Compute~~ $\cancel{p[1], p[2], \ldots, p[n]}$ ~~via binary search.~~

$M[0] \leftarrow 0$.

$M[1] \leftarrow v_1$

FOR $j = \cancel{1}\ 2$ TO n

  $M[j] \leftarrow \max \{ M[j-1], \cancel{c_j + M[p[j]]}\ v_j + M[j-2] \}$.

return $M[n]$

~~Runtime?~~

Your turn

Weighted interval scheduling: finding a solution

FIND-SOLUTION(j)

IF ($j = 0$)

RETURN $\varnothing$.

ELSE IF ($w_j + M[p[j]] > M[j-1]$) 6 + 3   8

RETURN $\{ j \} \cup$ FIND-SOLUTION($p[j]$).

ELSE

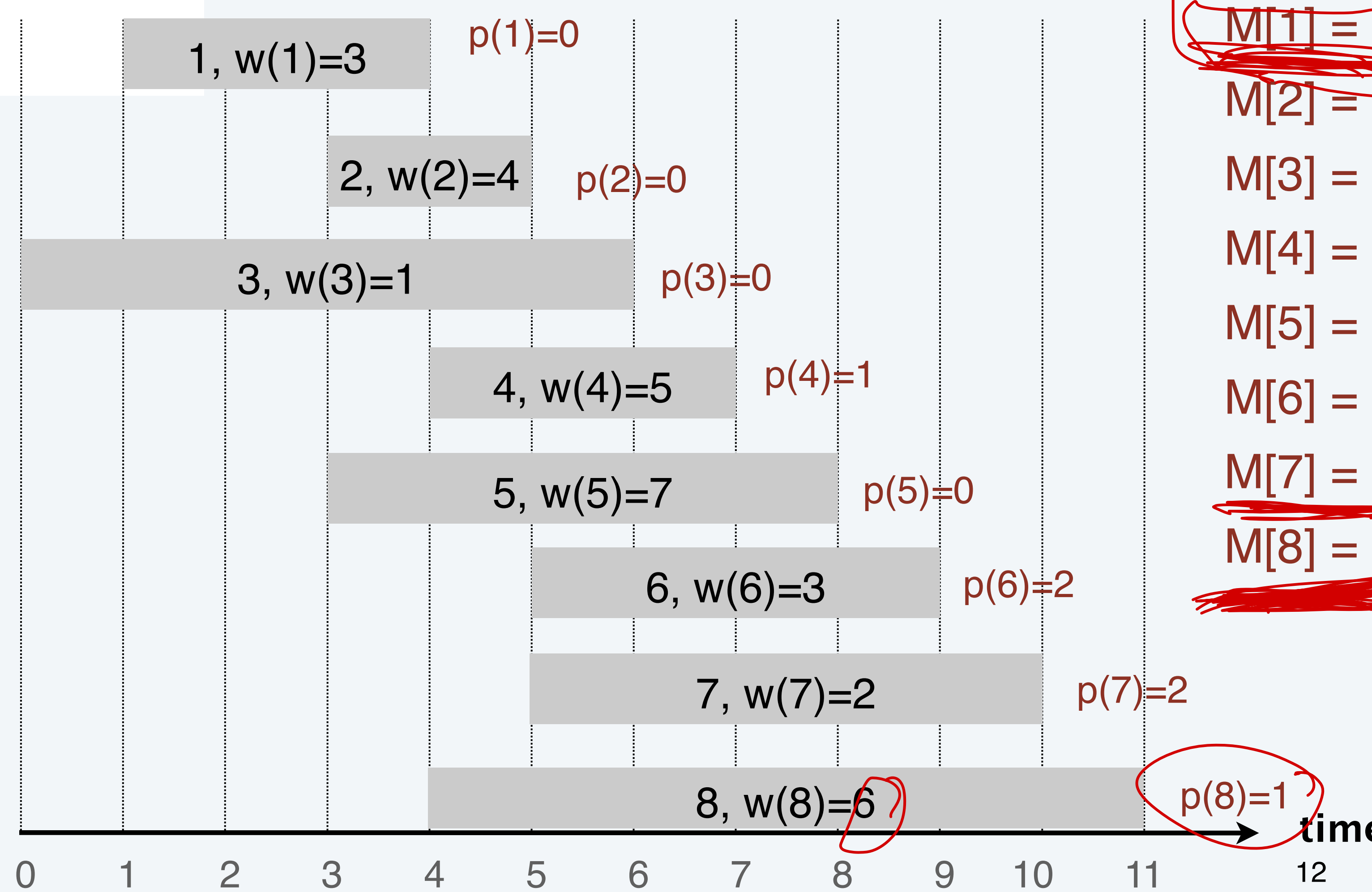
RETURN FIND-SOLUTION($j-1$).

find-solution(8)

{8} ∪ find-sol(1)

{8, 1}

M[0] = 0
M[1] = 3
M[2] = 4
M[3] = 4
M[4] = 8
M[5] = 8
M[6] = 8
M[7] = 8
M[8] = 9



find-solution(n). let's trace through

your turn

Independent set on a path: finding a solution

Can there be more than one optimal set of intervals?

FIND-SOLUTION(j)

IF ($j = 0$)

  RETURN $\varnothing$.

ELSE IF ($w_j + M[p[j]] > M[j-1]$)

  RETURN $\{ j \} \cup$ FIND-SOLUTION($p[j]$).

ELSE

  RETURN FIND-SOLUTION($j-1$).

1. Yes
2. No

Can there be more than one optimal set of intervals?

$\textsc{Find-Solution}(j)$

$\textsc{If}\ (j = 0)$

$\quad\textsc{Return}\ \varnothing.$

$\textsc{Else if}\ (w_j + M[p[j]] > M[j-1])$

$\quad\textsc{Return}\ \{\,j\,\} \cup \textsc{Find-Solution}(p[j]).$

$\textsc{Else}$

$\quad\textsc{Return}\ \textsc{Find-Solution}(j-1).$

1. Yes
2. No

With table: which one does this algorithm find?

Memoization allowed us to go from $O(2^n)$ to $O(n)$...

Can we memoize merge sort? (with table)

mergesort(L):

```
  L_1 = first half of L
  L_2 = first half of L
  sorted_L_1 = mergesort(L_1)
  sorted_L_2 = mergesort(L_2)
  return merged L_1 and L_2
```

Memoization allowed us to go from $O(2^n)$ to $O(n)$...

Can we memoize merge sort?

mergesort(L):

```
  L_1 = first half of L
  L_2 = first half of L
  sorted_L_1 = mergesort(L_1)
  sorted_L_2 = mergesort(L_2)
  return merged L_1 and L_2
```

No-the key was overlapping subproblems