Recap of weighted interval scheduling

Compute-Opt(n):
if M[n] is non-null, return M[n],
else // This is the first time we've been
      asked to solve Compute-Opt(-).
    if n = 0
      set M[n] = 0
    else
      compute \( p(n) = \max \{ i \mid f_i < s_n \} \)
      set \( M[n] = \max \{ \text{Compute-Opt}(n-1), \text{V}_n + \text{Compute-Opt}(p(n)) \} \)
    return M[n].

\[
\begin{array}{ccc}
5 & 4 & 3 \\
8 & 6 \\
\end{array}
\]
Pending issues:

1. Running time analysis
2. How to output the contents of the opt set?
3. What if...?

\[
\text{Compute-Opt}(n):
\]
\[
\text{if } M[n] \text{ is non-null, return } M[n]. \leftarrow O(1)
\]
\[
\text{else } \quad \text{// This is the first time we've been asked to solve Compute-Opt(n).}
\]
\[
\text{if } n = 0
\]
\[
\text{set } M[n] = 0 \leftarrow O(1)
\]
\[
\text{else } \quad \text{// Precomputed in } O(n \log n)
\]
\[
\text{compute } p(n) = \max \{ i \mid f_i < s_n \} \leftarrow O(1)
\]
\[
\text{set } M[n] = \max \left\{ \begin{array}{l}
\text{Compute-Opt}(n-1), \\
V_n + \text{Compute-Opt}(p(n))
\end{array} \right\} \leftarrow O(1)
\]
\[
\text{return } M[n].
\]

\text{Precompute Predecessors:}

\text{// Recall } [s_1, f_1], \ldots, [s_n, f_n] \text{ are sorted by increasing finish time } f_1 \leq f_2 \leq \ldots \leq f_n

\text{for } i = 1, 2, \ldots, n:

\text{use binary search to find}

\[
O(n) = \max \{ i \mid f_i < s_3 \}.
\]
Store the \( p(\cdot) \) table for later use.

Everything in Compute-Opt is \( O(1) \) except the two recursive calls to Compute-Opt. Those are also \( O(1) \) if they retrieve a value stored in the dynamic prog table.

The only unaccounted-for running time is from the first call to Compute-Opt \( (k) \) for each \( k = 0, 1, \ldots, n \).

That takes \( O(n) \) time excluding recursive first-calls to Compute-Opt \( (j) \) \( (j < k) \).

Total running time:

\[
O(n \log n) \text{ to PreCompute Predecessors.} \\
O(A) \text{ on each of Compute-Opt}(0) \\
Compute-Opt(1) \\
\vdots \\
Compute-Opt(n) \\
\text{Overall this sums to } O(n \log n).
\]
(2) How to modify to output the set of intervals?

Make the return value an ordered pair \((v, S)\) where \(v\) is the value and \(S\) is the set.

\[
\text{Compute}^*\theta(n): \\
\text{if } M[n] \text{ is non-null return } M[n] \\
\text{else if } n = 0: \\
\text{return } (0, 0) \\
\text{else:} \\
\text{let } (a_0, S_0) = \text{Compute}^*\theta(n-1) \\
\text{let } (a_1, S_1) = \text{Compute}^*\theta(\phi(n)) \\
\text{if } v_n + a_1 > a_0: \\
(vS) = (v_n + a_1, \{1 \cup S_1\}) \\
\text{else:} \\
(vS) = (a_0, S_0) \\
M[n] = (vS) \\
\text{return } (v, S)
\]

(3) What if we sorted intervals in a different order?
For example, what if $v_1 \leq v_2 \leq \ldots \leq v_n$?

It still holds that the maximum weight conflict-free set either contains the $n$th interval or it doesn’t.

\[
\text{Opt}(\{1, \ldots, n\}) = \max \left\{ \text{Opt}(\{1, \ldots, n-1\}), \text{Opt}(S_n) + v_n \right\}
\]

$S_n$ denotes \{	ext{intervals that don't conflict with the $n$th one}\}

\
\begin{align*}
1 & \quad \quad \quad \quad \quad 3 \\
4 & \quad \quad \quad \quad \quad 5 \\
& \quad \quad \quad \quad \quad \quad 2
\end{align*}

$S_5 = \{2, 4\}$ for example.

The Knapsack Problem:

$n$ items with weights $w_1, w_2, \ldots, w_n$

Assume integer in the range $\{1, 2, \ldots, W\}$

Values $v_1, v_2, \ldots, v_n$

Values could be arbitrary (e.g. floating point)
You have a "knapsack" with capacity \( W \). Select a subset with combined weight \( \leq W \). Maximize combined value.

\[
\begin{align*}
&\omega_1 = 4 & \omega_2 = 6 & \omega_3 = 1 & \omega_4 = 4 \\
&v_1 = 50 & v_2 = 60 & v_3 = 10 & v_4 = 50
\end{align*}
\]

\( W = 7 \)

Optimal set is items 2 and 3. Combined weight 7. Value 70.

Plan of attack: optimal set either contains item \( n \) or it doesn't.

\[
\begin{align*}
&\text{If you omit } n, \text{ you should choose the optimal subset of items } 1, \ldots, n-1. \\
&\text{If you choose item } n, \text{ you also choose the opt subset } 1, \ldots, n-1 \text{ but with a modified capacity!}
\end{align*}
\]
Capacity $W-w_n$ rather than $W$.

How to design a recursive algorithm to find the optimal set that fits the knapsack, when the knapsack's capacity may change in the recursive calls?

Pass in the knapsack capacity as a parameter to the recursive function.

$\text{Compute-Opt}(n, W)$:

// find optimal knapsack solution using items 1,...,n with total weight $\leq W$

if $M[n,W]$ is non-null:
    return $M[n,W]$
else:
    if $n=0$:
        return $(0, \emptyset)$
    else if $w_n > W$:
        return $\text{Compute-Opt}(n-1, W)$
    else:
\((a_0, S_0) = \text{Compute-Opt}(n-1, W)\)

\((a_1, S_1) = \text{Compute-Opt}(n-1, W-w_n)\)

if \(v_n + a_1 > a_0\):

\((v, S) = (v_n + a_1, \frac{n}{n+1} \cdot S_1)\)

else:

\((v, S) = (a_0, S_0)\)

\(M[n, W] = (v, S)\)

return \((v, S)\)

Running time \(O(nW)\).

"pseudopolynomial": could be exponentially bigger than \# bits in the problem input if \(W\) is written in binary.