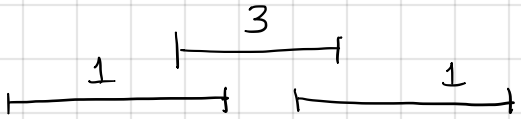
7 Feb 2018 Happy "e day", 2-7-18!

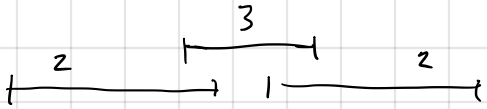
Dynamic Programming: The Weighted Interval Scheduling Problem (§6.1)

Input consists of n intervals numbered $i = 1, 2, \ldots, n$ each specified by
- start time s_i
- finish time f_i
- weight w_i

Goal: Select a non-conflicting subset of the intervals with maximum combined weight.



EFT doesn't work!



Highest weight first also doesn't work.

Algorithms that go through the intervals in sequence, making irrevocable decisions (to choose or omit) without look ahead seem ineffective.

DYNAMIC PROGRAMMING: First step is to reason exhaustively about the potential structures of optimal solutions.

[1] Assume intervals numbered so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

[2] Observation: the optimal solution either contains interval n, or it doesn't.

[3] An optimal solution that contains interval n consists of $\{\text{interval } n\} \cup \{\text{max-weight non-conflicting subset of the intervals that finish before } s_n\}$

[1] Assume intervals numbered so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

[2] Observation: the optimal solution either contains interval n, or it doesn't.

[3] An optimal solution that contains interval n consists of $\{\text{interval } n\} \cup \{\text{max-weight non-conflicting subset of the intervals that finish before } s_n\}$

Proof. Let S be a non-conflicting set of intervals that contains interval n. So $S = \{n\} \cup S'$

(1) Every interval in S' doesn't conflict with interval n. (Starts before s_n or ~~finishes after~~ f_n.)

(2) Every interval in S' starts before s_n and doesn't conflict with interval n, hence finishes before s_n.

(3) Let $S'' = \{\text{max-weight non-conflicting subset of the intervals that finish before } s_n\}$

By definition $\text{weight}(S'') \geq \text{weight}(S')$.

(4) So $\text{weight}(\{n\} \cup S'') \geq \text{weight}(S)$.

Since S was an arbitrary non-conflicting set containing interval n, we may conclude that $\{n\} \cup S''$ is the max-weight such set, as claimed.

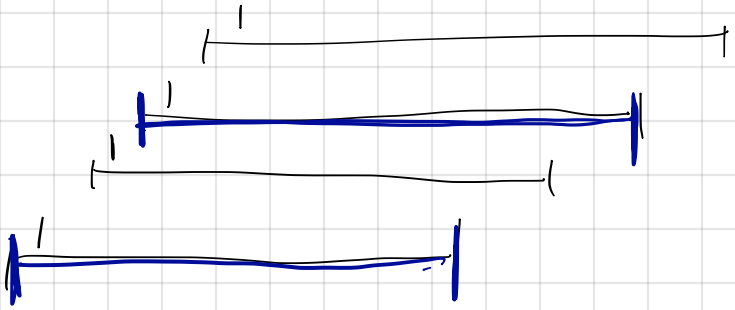
[4] An optimal solution that doesn't contain interval n is $\{\text{max-weight non-conflicting subset of intervals } 1, \ldots, n-1\}$.

A recursive algorithm suggested by this case analysis.

MWIS (intervals $1, \ldots, n$): Sort intervals so that $f_1 \leq \ldots \leq f_n$.

By calling MWIS recursively →

1. Find $S_0 := \{\text{max-weight non-conflicting subset of intervals } 1, \ldots, n-1\}$
2. Find $S_1 := \{\text{max-weight non-conflicting subset of intervals that finish before } s_n\}$
3. Compute $\text{weight}(S_0)$, $\text{weight}(S_1 \cup \{n\})$.
4. Output the one with higher weight.



Analysis of correctness. An exhaustive case analysis justifying correctness was already presented.

Running time. Uh oh.

MWIS (n intervals) calls MWIS twice, on inputs that each could potentially contain $n-1$ intervals.

Running time recurrence. If $T(n)$ denotes time to solve an instance with n intervals,

$$T(n) \leq 2 \cdot T(n-1) + O(n)$$

$$\therefore \quad T(n) \leq O(2^n + n^2) = O(2^n)$$

Eliminate exponential running time by "memoization":
for every initial segment of the sequence of intervals,
i.e. for every input to MWIS consisting of intervals $1, \ldots, k$,
cache the output of MWIS $(1, \ldots, k)$ and use the cached value instead of recomputing whenever the recursive alg. calls MWIS $(1, \ldots, k)$ in future.

This cache is called the "dynamic programming table."

$A[k]$ stores the output of MWIS $(1, \ldots, k)$.

The more careful implementation of MWIS is as follows.

Sort intervals so that $f_1 \leq f_2 \leq \dots \leq f_n$.
Initialize $A[k] = \text{NULL}$ for $k = 0, 1, \dots, n-1$.

MWIS (intervals $1, \dots, k$):

To verify that this works, these must be the first j intervals, for some $j < k$.

1. If $A[k] \neq \text{NULL}$ return $A[k]$. Else...
2. $S_0 = \text{MWIS}(1, \dots, k-1)$
3. $S_1 = \text{MWIS}$ (the set of intervals that finish before s_k)
4. Compute $w(S_0)$, $w(S_1 \cup \{k\})$.
5. Store the better of these two sets as $A[k]$.
6. Return $A[k]$.

Running time devoted to MWIS (k):
$O(k)$ first time the subroutine is called. (Line 4)
$O(1)$ every subsequent time.

$\therefore O(n)$ in total. $(= O(k) + O(n-k))$

That's $O(n)$ for each $k = 1, \dots, n$.
So the algorithm runs in $O(n^2)$ total.