Announcement.

Problem Set 2, Question 2.
You can assume log entries have
distinct timestamps. However \( u_i \)'s, \( v_i \)'s
(or even \( u_i, v_i \) pairs) can occur
repeatedly.
Also don’t assume log entries are
in any particular order.

---

**DYNAMIC PROGRAMMING** (Chapter 6)

**Weighted Interval Scheduling**, Input consists of \( n \) requests or jobs.
Each has three properties:
- start time \( s_i \);
- finish time \( f_i \);
- value \( v_i \geq 0 \) if selected.

**Goal** Select a set of jobs that is
conflict-free (if \( s_j \) are selected then
\([s_i,f_i]\) is disjoint from \([s_j,f_j]\)) and
has max total value subject to conflict-free.
EFT no longer works: \[ \begin{array}{c}
& v_2 = 3 \\
v_1 = 1 & v_3 = 1
\end{array} \]
optimal set is job 2 only.
EFT picks jobs 1 and 3.

Greedy by value also doesn’t work: \[ \begin{array}{c}
& v_2 = 3 \\
v_1 = 2 & v_3 = 2
\end{array} \]
optimal set is jobs 1 and 3.
Greedy by value gets only job 2.

In fact nobody knows a greedy algorithm that solves every instance of WJS correctly.
There probably isn’t one, but it depends on how you define a greedy algorithm, which is still not standardized.

Plan: Think more carefully and systematically about the set of feasible solutions and how to optimize over it.

Structure Lemma for Conflict-Free Request Sets
Assume requests are sorted such that \( f_1 \leq f_2 \leq \ldots \leq f_n \). Let \( p(n) \) denote
the highest numbered request that finishes before \( s_n \).

Every conflict-free subset of \( \{1, 2, \ldots, n\} \) is either:

\( a \) a conflict-free subset of \( \{1, 2, \ldots, n-1\} \)

\( b \) \( s \cup \{n\} \) where \( s \) is a conflict-free subset of \( \{1, \ldots, \rho(n)\} \)

**Proof sketch.** The lemma statement encodes the obvious fact that set of requests either contains \( n \) or it doesn’t. And if it contains \( n \) and is conflict-free, then it can’t contain any interval whose finish time is in \( [s_n, f_n] \).

Recursive algorithm for computing the maximum value of a conflict-free set.

**(Not the contents of the set.)**

\[
\text{Compute-Opt}(n): \quad \text{// Compute max value of a conflict-free subset of \( \{1, \ldots, n\} \).}
\]

\[
\text{if } n = 0 \quad \text{return } 0
\]

\[
\text{else}
\]

\[
\text{Find } \rho(n).
\]

\[
\text{return } \max \{ \text{Compute-Opt}(n-1), \mu + \text{Compute-Opt}(\rho(n)) \}
\]
Correctness of Compute-Opt(n) follows by induction on n. Ind hyp is that the function correctly computes the max value of a conflict-free subset of \( \{1, 2, \ldots, n\} \).

Base case \( n=0 \): the only subset of the empty set is empty and has value 0.

Induction step: Structure lemma implies that max-value conflict-free subset is either

(a) the max value conf-free subset of \( \{1, \ldots, n-1\} \); or
(b) \( S \cup \{n\} \) where \( S \) is the max value conf-free subset of \( \{1, \ldots, p(n)\} \).

By ind hyp, max value achievable in case (a) is Compute-Opt(n-1). And max value achievable in case (b) is \( v_n + \text{Compute-Opt}(p(n)) \).

\[ \therefore \text{Compute-Opt}(n) \text{ outputs the right answer.} \]
Struct Lemma says: A conf-free subset either picks \([s_5, f_5]\) together with a subset of the first 3 requests, or it omits \([s_5, f_5]\).

If you pick \([s_5, f_5]\), you get 7 plus the value of the other jobs picked. Otherwise, you get the value of whatever subset of the first 4 jobs you pick.
Running the pseudo-code in blue above would generate a set of recursive calls to `compute-opt(...)` modeled by the tree above.

Watch out! The tree can get exponentially big. To make the algorithm efficient we'll use "memoization." Store results of previous `compute-opt` calls in a table, `M[ ]`.

```
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(n).
    if n = 0
        set M[n] = 0
    else
        compute p(n) = max { i | f_i < s_n }
        set M[n] = max { Compute-Opt(n-1), V_n + Compute-Opt(p(n)) }
    return M[n].
```
Analysis of running time:
Excluding time spent in recursive calls, Compute-Opt does $O(n)$ work.

Furthermore, Compute-Opt($i$) does this work at most once, for each $i = 0, 1, 2, \ldots, n$.

Running time, in total, is

\[
\sum_{i=0}^{n} \text{ (time spent on Compute-Opt($i$))} \\
\leq n \cdot O(n) \\
= O(n^2).
\]

Faster implementation: pre-compute $p(i)$ for each $i$. This preprocessing step happens before we ever call Compute-Opt.

The pre-processing to compute $p(1), p(2), \ldots, p(n)$ can be implemented to run in $O(n)$.

Then Compute-Opt does $O(1)$ work outside recursive calls, so the whole algorithm becomes $O(n)$. 