CS 341: ALGORITHMS (S18) — LECTURE 8
DYNAMIC PROGRAMMING I

ERIC BLAIS

We now turn our attention to another general technique for designing algorithms: *dynamic programming*. The main idea for this technique can be summarized as follows: We can solve a big problem by

(1) Breaking it up into smaller sub-problems;
(2) Solving the sub-problems from smallest to largest; and
(3) Storing solutions along the way to avoid repeating our work.

We have already seen one example of this algorithm design technique in the first lecture, when we computed Fibonacci numbers. Over the next few lectures, we will explore a range of other problems where dynamic programming is also useful.

1. Weighted Interval Scheduling

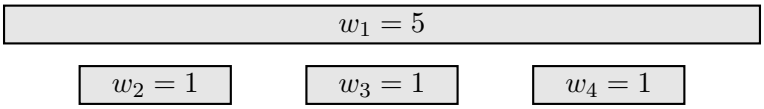
Let's revisit the interval scheduling problem, with a slight twist.

Definition 8.1 (Weighted interval scheduling)**.** In the *weighted interval scheduling problem*, an instance is a set of n pairs of intervals that have start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ and positive *weights* $w_1, \ldots, w_n$. A valid solution is a subset $I \subseteq \{1, 2, \ldots, n\}$ of non-overlapping intervals that maximizes $\sum_{i \in I} w_i$.

Without weights, we saw that the greedy algorithm that sorts the intervals by finish time and considers them in that order solves the problem. That's no longer the case when we have weights.

Proposition 8.2. *The* GreedyIntervalScheduler *algorithm does not solve the weighted interval scheduling problem.*

Proof. There are many instances where the algorithm does not return a valid solution, such as this one.



The valid solution is $I = \{1\}$ but the greedy algorithm outputs $I = \{2, 3, 4\}$ instead. □

A simple variation on the same counter-example shows that a modified greedy algorithm that considers the intervals by order of weights does not solve the problem either. Instead, we need to take a different approach.

As it turns out, the key to solving the weighted interval scheduler problem is to again sort the intervals by finish times and consider them in order. But instead of making greedy decisions, for a fixed instance of the problem let us define and for every $k \leq n$, we define

- $\text{OPT}(k) \subseteq \{1, 2, \ldots, k\}$ is a set of non-overlapping intervals with maximum total weight; and
- $w_{\text{OPT}}(k) = \sum_{j \in \text{OPT}(k)} w_j$ is its weight.

This immediately suggests how to solve the weighted interval scheduler problem using dynamic programming method: we compute $\text{OPT}(1), \text{OPT}(2), \ldots, \text{OPT}(n)$ in that order, and output $\text{OPT}(n)$.

To complete the algorithm, we only need to answer one more question: after we have computed $\text{OPT}(1), \ldots, \text{OPT}(k-1)$ and the weights of all those sets, how can we use those values to compute $\text{OPT}(k)$? If we examine the problem carefully, we realize there are exactly two possibilities to consider:

Case 1: $k \notin \text{OPT}(k)$: The first possibility is that the kth interval is not in the set $\text{OPT}(k)$. If that's the case, then $\text{OPT}(k) = \text{OPT}(k-1)$ and $w_{\text{OPT}}(k) = w_{\text{OPT}}(k-1)$.

Case 2: $k \in \text{OPT}(k)$: Otherwise, if $k \in \text{OPT}(k)$ and we let $j < k$ be the largest value such that $f_j < s_k$, then $\text{OPT}(k)$ can only be a set of non-overlapping intervals if $\text{OPT}(k) \subseteq \{1, 2, \ldots, j, k\}$ so $\text{OPT}(k) = \text{OPT}(j) \cup \{k\}$ and $w_{\text{OPT}}(k) = w_{\text{OPT}}(j) + w_k$.

We can compute the value of both candidates to determine which is correct. The resulting dynamic programming algorithm is as follows.

Algorithm 1: WEIGHTEDISP$((s_1, f_1), \ldots, (s_n, f_n), w_1, \ldots, w_n)$

```
(Sort intervals so that f_1 ≤ f_2 ≤ ··· ≤ f_n);
I[0] ← ∅;
W[0] ← 0;
f_0 ← 0;
for k = 1, ..., n do
    j ← max{0 ≤ i < k : f_i < s_k};
    if W[k−1] > W[j] + w_k then
        I[k] ← I[k−1];
        W[k] ← W[k−1];
    else
        I[k] ← I[j] ∪ {k};
        W[k] ← W[j] + w_k;
return I(n);
```

What is the running time of this algorithm? For the moment, let's assume that we can read, write, and manipulate the values $I[i]$ and $W[i]$ in constant time. Under this assumption, the bottleneck in the above implementation is the search for the value of j in every iteration of the for loop. With the naïve linear search for this value (and with the $\Theta(n \log n)$ time complexity of the sorting operation in the first step), the time complexity of the algorithm is

$$T(n) = \Theta(n \log n) + \sum_{k=1}^{n} \Theta(k) = \Theta(n^2).$$

But there's a better implementation: since the intervals are already sorted by finish times, we can also use binary search to find the value of j insteads. With this implementation, we

have the improved running time

$$T(n) = \Theta(n \log n) + \sum_{k=1}^{n} \Theta(\log k) = \Theta(n \log n).$$

But now, as a final step, let us revisit our assumption that we can read, write, and manipulate the values $I[i]$ and $W[i]$ in constant time. Is this assumption reasonable? You should be able to convince yourself that it is when we are considering the weights $W[i]$—but it is *not* reasonable for the sets $I[i]$; in general, storing these sets takes $\Theta(n)$ space and operations on those sets have time complexity $\Omega(n)$. So it would be more accurate to say that the algorithm has time complexity $\Theta(n^2)$. To get a "true" time complexity of $\Theta(n \log n)$, we should figure out if it's possible to modify the algorithm so that it computes the values $W[1], W[2], \ldots, W[n]$ *without* computing the sets $I[1], I[2], \ldots, I[n]$. It is indeed possible to do so; you should try to see how that can be done!