13 Dynamic Programming, Weighted Interval Scheduling

- Dynamic Programming
- Weighted Interval Scheduling
- Knapsack

13.1 Weighted Interval Scheduling

Recall the interval scheduling problem: given intervals $[s_i, f_i]$ for $i \in \{1, \cdots, n\}$, find a largest possible subset of nonoverlapping intervals.

New twist: Interval i has a value $v_i \in \mathbb{R}$. Our goal is to find a set of nonoverlapping intervals maximizing $\sum_{i \in S} v_i$.

We sort the intervals so that the finishing times are non decreasing: $f_1 \leq f_2 \leq \cdots \leq f_n$.

We ask if the last interval part of the optimal solution? We need to consider the possibility that it is, and that it isn't.

- If no, then the optimal solution of the whole problem is on intervals $\{1, \cdots, n-1\}$.
- If yes, then the optimal solution is v_n+ the optimal solution on $\{1, \cdots, p(n)\}$
  Where $P(j) = \max\{i < j : \text{intervals } i \text{ and } j \text{ are disjoint}\}$

Let $\text{opt}(j)$ be the optimal value on intervals $\{1, \cdots, j\}$.
$\text{opt}(j) = \max\{\text{opt}(j-1) + v_j + \text{opt}(p(j))\}$
$\text{opt}(0) = 0$.

Interval j is part of the solution if $v_j + \text{opt}(p(j)) \geq \text{opt}(j-1)$

13.1.1 Recursive Algorithm

We assume that the intervals are sorted by finishing time, and that values $p(j)$ have been precomputed in time $O(n \log n)$.

```
ComputeOpt(j):
    if j = 0 then
        return 0
    else
        return max{ComputeOpt(j-1), v_j + ComputeOpt(p(j))}
    endif
```

$\text{ComputeOpt}(n)$ will return the desired value.

Running time: let $T(j)$ denote the running time of $\text{ComputeOpt}(j)$.

$$T(j) = T(j-1) + T(p(j)) + O(1)$$

We know that $p(j) \leq j-1$, and $p(j) = j-1$ for every j in the worst case.

$$T(j) = 2 \cdot T(j-1) + O(1)$$

At every j we double the computation we do, so this grows exponentially with respect to j!

We can improve on this by storing the solutions of previously computed subproblems, called "Memoization".

Initially, let $M[j] = \varnothing$ for all $j \in \{1, \cdots, n\}$.

```
MComputeOpt(j):
    if j = 0 then
        return 0
    elseif M[j] != null, then
        return M[j]
```

```
    else
        let M[j] = max(MComputeOpt(j-1), v_j + MComputeOpt(p(j)))
        return M[j]
    endif
```

We can find the optimal solution by checking which terms achieve the max.

Lemma 13.1
The running time of MComputeOpt(n) is $O(n)$.

Proof. The running time of MComputeOpt(j) is $O(1)+$ cost of its recursive calls, so running time of MComputeOpt(n) is O(total number of recursive calls).

We make at most n recursive calls since there are only n values of $M[j]$, and once we've completed $M[j]$, we never call MComputeOpt(j) again. □

Alternatively, we can just compute the values $M[j]$ interatively:

```
ItComputeOpt:
    let M[0] = 0
    for j = 1 to n do
        let M[j] = max{M[j-1], v_j + M[p(j)]}
    endfor
```