Dynamic Programming

- Our 3rd major algorithm design technique

- Similar to divide & conquer
  - Build up the answer from smaller subproblems
  - More general than “simple” divide & conquer
  - Also more powerful

- Generally applies to algorithms where the brute force algorithm would be exponential.
Recall the interval scheduling problem we’ve seen several times: choose as many non-overlapping intervals as possible.

What if each interval had a value?

**Problem (Weighted Interval Scheduling)**

Given a set of $n$ intervals $(s_i, f_i)$, each with a value $v_i$, choose a subset $S$ of non-overlapping intervals with $\sum_{i \in S} v_i$ maximized.
Note that our simple greedy algorithm for the unweighted case doesn’t work.

This is because some interval can be made very important with a high weight.
Greedy Algorithm For Unweighted Case:

1. Sort by increasing finishing time

2. Repeat until no intervals left:
   
   1. Choose next interval
   
   2. Remove all intervals it overlaps with
Suppose for now we’re not interested in the actual set of intervals. Only interested in the value of a solution (aka it’s cost, score, objective value).

This is typical of DP algorithms:

- You want to find a solution that optimizes some value.
- You first focus on just computing what that optimal value would be. E.g. what’s the highest value of a set of compatible intervals?
- You then post-process your answer (and some tables you’ve created along the way) to get the actual solution.
Another View

Another way to look at Weighted Interval Scheduling:

Assume that the intervals are sorted by finishing time and represent each interval by its value.

Goal is to choose a subset of the values of maximum sum, so that none of the chosen (√) intervals overlap:

\[ V_1 \quad V_2 \quad V_3 \quad V_4 \quad \cdots \quad V_{n-1} \quad V_n \]

\[ X \quad \checkmark \quad X \quad \checkmark \quad \checkmark \quad X \]
**Notation**

**Definition**

\[ p(j) = \text{the largest } i < j \text{ such that interval } i \text{ doesn’t overlap with } j. \]

\[ p(1) = 0 \]
\[ p(2) = 0 \]
\[ p(3) = 1 \]
\[ p(4) = 0 \]
\[ p(5) = 3 \]
\[ p(6) = 3 \]

\[ p(j) \] is the interval farthest to the right that is compatible with \( j \).
What does an OPT solution look like?

Let OPT be an optimal solution.

Let $n$ be the last interval.

\begin{itemize}
  \item If \textbf{Yes} to \textit{Does OPT contain interval $n$?}, \textbf{OPT} = n + \text{Optimal solution on } \{1, \ldots, p(n)\}
  \item If \textbf{No}, \textbf{OPT} = \text{optimal solution on } \{1, \ldots, n-1\}
\end{itemize}
Generalize

**Definition**

\[ \text{OPT}(j) = \text{the optimal solution considering only intervals } 1, \ldots, j \]

\[
\text{OPT}(j) = \max \begin{cases} 
  v_j + \text{OPT}(p(j)) & \text{j in OPT solution} \\
  \text{OPT}(j-1) & \text{j not in solution} \\
  0 & \text{j = 0}
\end{cases}
\]

This kind of recurrence relation is very typical of dynamic programming.
Implementing the recurrence directly:

```python
WeightedIntSched(j):
    If j = 0:
        Return 0
    Else:
        Return max(
            v[j] + WeightedIntSched(p[j]),
            WeightedIntSched(j-1)
        )
```

Unfortunately, this is exponential time!
Why is this exponential time?

Consider this set of intervals:

\[ p(j) = j - 2 \text{ for all } j \geq 3 \]

- What’s the shortest path from the root to a leaf?
- Total # nodes is \( \geq 2^{n/2} \)
- Each node does constant work \( \implies \Omega(2^n) \)
Why is this exponential time?

Consider this set of intervals:

\[ p(j) = j - 2 \text{ for all } j \geq 3 \]

- What’s the shortest path from the root to a leaf? \( n/2 \)
- Total \# nodes is \( \geq 2^{n/2} \)
- Each node does constant work \( \implies \Omega(2^n) \)
Problem: Repeatedly solving the same subproblem.

Solution: Save the answer for each subproblem as you compute it.

When you compute $OPT(j)$, save the value in a global array $M$. 
MemoizedIntSched(j):
    If j = 0: Return 0
    Else If M[j] is not empty:
      Return M[j]
    Else
      M[j] = max(
        v[j] + MemoizedIntSched(p[j]),
        MemoizedIntSched(j-1)
      )
    Return M[j]

• Fill in 1 array entry for every two calls to MemoizedIntSched.  
  $\Rightarrow O(n)$
When we compute $M[j]$, we only need values for $M[k]$ for $k < j$:

```
ForwardIntSched(j):
    M[0] = 0
    for j = 1, ..., n:
        M[j] = max(v[j] + M[p(j)], M[j-1])
```

**Main Idea of Dynamic Programming:** solve the subproblems in an order that makes sure when you need an answer, it’s already been computed.
$v_j + M[p(j)]$

$M[j-1]$
Example

\[ v_j + M[p(j)] \]
\[ M[j-1] \]

10 1 10
20 2 20
5 3 5
20 4 20
15 5 15
Example

\[ v_j + M[p(j)] \]
\[ M[j-1] \]
Example

\[ v_j + M[p(j)] \]
\[ M[j-1] \]
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]
1. Optimal value of the original problem can be computed easily from some subproblems.

2. There are only a polynomial # of subproblems.

3. There is a “natural” ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.
1. Optimal value of the original problem can be computed easily from some subproblems. $\text{OPT}(j) = \max$ of two subproblems

2. There are only a polynomial number of subproblems. $\{1, \ldots, j\}$ for $j = 1, \ldots, n$.

3. There is a “natural” ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems. $\{1, 2, 3\}$ is smaller than $\{1, 2, 3, 4\}$
Getting the actual solution

We now have an algorithm to find the value of OPT. How do we get the actual choices of intervals?

Interval $j$ is in the optimal solution for the subproblem on intervals $\{1, \ldots, j\}$ only if

$$v_j + OPT(p(j)) \geq OPT(j - 1)$$

So, interval $n$ is in the optimal solution only if

$$v[n] + M[p[n]] \geq M[n - 1]$$

After deciding if $n$ is in the solution, we can look at the relevant subproblem: either $\{1, \ldots, p(n)\}$ or $\{1, \ldots, n - 1\}$.
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]
Example

```
v_j + M[p(j)]
M[j-1]
10  |  20  |  20  |  30  |  35
0   |  10   |  20   |  15   |  30   |  35
```
Example

\[ v_j + M[p(j)] \]

\[
\begin{array}{cccccc}
0 & 10 & 20 & 20 & 30 & 35 \\
1 & 2 & 3 & 4 & 5 & \\
\end{array}
\]
Example

\[ v_j + M[p(j)] \]
\[ M[j-1] \]
BacktrackForSolution(M, j):
    If j > 0:
        If v[j] + M[p[j]] ≥ M[j-1]: // find the winner
            Output j // j is in the soln
            BacktrackForSolution(M, p[j])
        Else:
            BacktrackForSolution(M, j-1)
    EndIf
EndIf
Running Time

Time to sort by finishing time: $O(n \log n)$

Time to compute $p(n)$: $O(n^2)$

Time to fill in the $M$ array: $O(n)$

Time to backtrack to find solution: $O(n)$