Midterm Friday

closed book, no notes
(no bluebook needed; scratch paper may be handy; calculators unnecessary)
All assigned reading up through 6.1; slides through today; homework.

6.1 Weighted Interval Scheduling

Weighted interval scheduling problem.
- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

![Weighted Interval Scheduling Diagram]
**Unweighted Interval Scheduling Review**

Recall. Greedy algorithm works if all weights are 1.  
- Consider jobs in ascending order of finish time.  
- Add job to subset if it is compatible with previously chosen jobs.

**Observation.** Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

**Weighted Interval Scheduling**

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.  
**Def.** $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.  

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

**Dynamic Programming: Binary Choice**

**Notation.** $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$.

- **Case 1:** $OPT$ selects job $j$.  
  - can’t use incompatible jobs { $p(j) + 1, p(j) + 2, \ldots, j - 1$ }  
  - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$

- **Case 2:** $OPT$ does not select job $j$.  
  - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, j-1$

**Weighted Interval Scheduling: Brute Force**

**Brute force recursive algorithm.**

```
Input: n, s_1, s_2, \ldots, s_n , f_1, f_2, \ldots, f_n , v_1, v_2, \ldots, v_n
Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$
Compute-Opt(j) {
    if (j = 0)
        return 0
    else
        return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```
**Weighted Interval Scheduling: Brute Force**

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
p(1) = 0, p(j) = j-2
```

**Memoization.** Store sub-problem results in a cache; lookup as needed.

**Input:** $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

**Sort** jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$

**Compute** $p(1), p(2), \ldots, p(n)$

```
for j = 1 to n
    M[j] = empty  -- global array
    M[0] = 0
    M-Compute-Opt(j) {
        if (M[j] is empty)
            M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
        return M[j]
    }
Main() {
    ???
}
```

**Weighted Interval Scheduling: Running Time**

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.

- **Sort by finish time:** $O(n \log n)$
- **Computing $p()$:** $O(n)$ after sorting by start time.
- **M-Compute-Opt(j):** each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls
- **Progress measure $\Phi$:** the number of nonempty entries of $M[]$
  - initially $\Phi = 0$,
  - increases by most 2 at each stage
- **Overall running time of M-Compute-Opt(n) is $O(n)$.**

**Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.

**Weighted Interval Scheduling: Bottom-Up**

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

**Sort** jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$

**Compute** $p(1), p(2), \ldots, p(n)$

```
Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(v_j + M[p(j)], M[j-1])
    }
Output M[n]
```

**Claim:** $M[j]$ is value of optimal solution for jobs 1..j

(A bit subtler skipping details)
**Weighted Interval Scheduling**

**Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

**Def.** \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).

**Ex:** \( p(8) = 5 \), \( p(7) = 3 \), \( p(2) = 0 \).

<table>
<thead>
<tr>
<th>( j )</th>
<th>( v_j )</th>
<th>( p(j) )</th>
<th>( \text{opt}_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Sidebar: why does job ordering matter?**

It’s *Not* for the same reason as in the greedy algorithm for unweighted interval scheduling.

Instead, it’s because it allows us to consider only a small number of subproblems (\( O(n) \)), vs the exponential number that seem to be needed if the jobs aren’t ordered (seemingly, any of the \( 2^n \) possible subsets might be relevant).

Don’t believe me? Think about the analogous problem for weighted rectangles instead of intervals… (i.e., pick max weight non-overlapping subset of a set of axis-parallel rectangles.) Same problem for circles also appears difficult.

---

**Weighted Interval Scheduling: Finding a Solution**

**Q.** Dynamic programming algorithms computes optimal value. What if we want the solution itself?

**A.** Do some post-processing – “traceback”

```cpp
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- # of recursive calls \( \leq n \) \( \Rightarrow \) \( O(n) \).