Algorithmic Paradigms

**Greed.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Interval Scheduling
Interval Scheduling

Interval scheduling.
- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum subset of mutually compatible jobs.
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- What order?
- Does that give best answer?
- Why or why not?
- Does it help to be greedy about order?
Interval Scheduling: Greedy Algorithms

**Greedy template.** Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- [**Earliest start time**] Consider jobs in ascending order of start time $s_j$.

- [**Earliest finish time**] Consider jobs in ascending order of finish time $f_j$.

- [**Shortest interval**] Consider jobs in ascending order of interval length $f_j - s_j$.

- [**Fewest conflicts**] For each job, count the number of conflicting jobs $c_j$. Schedule in ascending order of conflicts $c_j$. 
Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.

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

```
A ← φ
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
}
return A
```

Implementation. $O(n \log n)$.
- Remember job $j^*$ that was added last to $A$.
- Job $j$ is compatible with $A$ if $s_j \geq f_{j^*}$.
Interval Scheduling

The diagram illustrates an interval scheduling problem with tasks represented by colored bars on a timeline.

- Task A spans from time 0 to 5.
- Task B starts at time 1 and ends at time 3.
- Task C starts at time 2 and ends at time 4.
- Task D spans from time 3 to 7.
- Task E starts at time 4 and ends at time 6.
- Task F starts at time 5 and ends at time 7.
- Task G starts at time 6 and ends at time 9.
- Task H spans from time 8 to 11.

The timeline is marked from time 0 to 11, with each hour representing a unit of time.
Interval Scheduling

![Interval Scheduling Diagram]

- A: Task A starts at time 0 and ends at time 5.
- B: Task B starts at time 1 and ends at time 3.
- C: Task C starts at time 2 and ends at time 4.
- D: Task D starts at time 4 and ends at time 8.
- E: Task E starts at time 5 and ends at time 6.
- F: Task F starts at time 6 and ends at time 7.
- G: Task G starts at time 7 and ends at time 10.
- H: Task H starts at time 9 and ends at time 11.

Time: 0 1 2 3 4 5 6 7 8 9 10 11
Interval Scheduling

Time

0  1  2  3  4  5  6  7  8  9  10  11

B  C  A  E  D  F  G  H
Interval Scheduling
Interval Scheduling

The diagram illustrates an interval scheduling problem with the following tasks:

- Task A from time 0 to 6
- Task B from time 2 to 4
- Task C from time 1 to 4
- Task D from time 4 to 8
- Task E from time 5 to 7
- Task F from time 6 to 8
- Task G from time 8 to 10
- Task H from time 9 to 11
Interval Scheduling
Interval Scheduling
**Theorem.** Greedy algorithm is optimal.

**Pf.** ("greedy stays ahead")

Let $i_1, i_2, \ldots, i_k$ be jobs picked by greedy, $j_1, j_2, \ldots, j_m$ those in some optimal solution.

Show $f(i_r) \leq f(j_r)$ by induction on $r$.

- **Basis:** $i_1$ chosen to have min finish time, so $f(i_1) \leq f(j_1)$
- **Ind:** $f(i_r) \leq f(j_r) \leq s(j_{r+1})$, so $j_{r+1}$ is among the candidates considered by greedy when it picked $i_{r+1}$, & it picks min finish, so $f(i_{r+1}) \leq f(j_{r+1})$

Similarly, $k \geq m$, else $j_{k+1}$ is among (nonempty) set of candidates for $i_{k+1}$.
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.

```
<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Weighted Interval Scheduling
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:** \( p(8) = 5, p(7) = 3, p(2) = 0. \)

<table>
<thead>
<tr>
<th>j</th>
<th>p(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Dynamic Programming

One of the algorithmic sledgehammers

High level idea:

▪ Find a recurrence for the optimal solution in terms of optimal solutions to subproblems of the same type.

▪ Build up solutions to these subproblems in order of increasing size.
Dynamic Programming: Binary Choice

Notation. $OPT(j) = \text{value (weight) of optimal solution to the problem consisting of job requests 1, 2, ..., j.}$

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

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

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), \text{ OPT}(j - 1) \right\} & \text{otherwise}
\end{cases}
\]
Weighted Interval Scheduling: Brute Force

Leads to recursive algorithm.

**Input:** n, s₁,...,sₙ, f₁,...,fₙ, v₁,...,vₙ

Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

Compute p(1), p(2), ..., p(n)

Compute-Opt(j) {
  if (j = 0)
    return 0
  else
    return max(vⱼ + 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.

\[
p(1) = 0, \ p(j) = j-2
\]
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming: compute solutions in order of “smallest” to “largest”.

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..\( j \)

Timing: Easy. Main loop is \( O(n) \); sorting is \( O(n \log n) \)
**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:** \( p(8) = 5, p(7) = 3, p(2) = 0 \).
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming: compute solutions in order of “smallest” to “largest”.

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 \( 1..j \)

Timing: Easy. Main loop is \( O(n) \); sorting is \( O(n \log n) \)
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”

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 ≤ n ⇒ O(n).
Algorithmic Paradigms

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Key properties needed for it to work:

- only polynomially many subproblems

- solution to original problem can be easily computed from solutions to subproblems.

- there is a natural ordering on subproblems from “smallest” to “largest” together with easy-to-compute recurrence that allows us to compute optimal solution to a subproblem in terms of optimal solutions to smaller subproblems.