Interval Scheduling: Greedy Algorithms and Dynamic Programming
Overview of Interval Scheduling

The Basic Interval Scheduling Problem
- Schedule as many non-overlapping tasks as possible in given timeframe
- (Representative problem #1 from day #1)

Total Interval Scheduling
- Must schedule all tasks
- Identify the fewest number of processors needed to schedule within given timeframe

Weighted Interval Scheduling
- Schedule non-overlapping tasks of maximum weight in given timeframe
- (Representative problem #2 from day #1)

We’ll look for greedy solutions when possible, and use dynamic programming when greedy algorithms don’t appear to work out.
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.

- [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 $c_j$. 
Interval Scheduling: Greedy Algorithms

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

- Breaks earliest start time
- Breaks shortest interval
- Breaks fewest conflicts
Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it’s compatible with the ones already taken.

```
INTERVAL-SCHEDULING( s₁, f₁, ..., sₙ, fₙ )
1. Remain = {1,...,n}
2. Selected = {}
3. while ( |Remain| > 0 ) {
4.     k = argmin  i ∈ Remain  fᵢ
5.     Selected = Selected ∪ {k}
6.     Remain = Remain – {k}
7.     for every i in Remain {
8.         if (sᵢ < fₖ) then Remain = Remain – {i}
9.     }
10. }
11. RETURN Selected
```

Implementation. $O(n^2)$.
- While loop is $O(n)$.
- Inside of loop is $O(n)$. (Argmin is $O(n)$. Updating Remain is $O(n)$.)
**Interval Scheduling: Greedy Algorithm**

**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 \( f_1 \leq f_2 \leq \ldots \leq f_n \).

```plaintext
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 \( s_j \geq f_{j^*} \).
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

```
Greedy:  j_1  j_2  \ldots  j_r  i_{r+1}
```

```
OPT:    j_1  j_2  \ldots  j_r  j_{r+1}  \ldots

job $i_{r+1}$ finishes before $j_{r+1}$

why not replace job $j_{r+1}$ with job $i_{r+1}$?
```

Slides based on Kevin Wayne / Pearson-Addison Wesley
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)
- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote the set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

Greedy: $i_1, i_1, i_r, i_{r+1}$

OPT: $j_1, j_2, \ldots, j_r, i_{r+1}$

Job $i_{r+1}$ finishes before $j_{r+1}$

Solution still feasible and optimal

Could be added to Greedy. Contradicts greedy construction.
Interval Scheduling: Analysis

Interval Scheduling by Dynamic Programming

Could this problem also be solved by dynamic programming?

- Yes. Sort by finish time.
- Let \( S[k] = \max(S[k-1], 1 + S[j]) \)
  - Where \( k \) is the items (intervals) ordered by finish time
  - Where \( j < k \) is the largest index such that the finish time of item \( j \) does not overlap the start time of item \( k \)
Interval Partitioning: Scheduling All
Interval Partitioning

Interval partitioning.
- Lecture j starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.
Interval Partitioning

Interval partitioning.
- Lecture $j$ starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.
Interval Partitioning: Lower Bound on Optimal Solution

**Def.** The depth of a set of open intervals is the maximum number that contain any given time.

**Key observation.** Number of classrooms needed \( \geq \) depth.

**Ex:** Depth of schedule below = 3 \( \Rightarrow \) schedule below is optimal.

\[ \uparrow \]

a, b, c all contain 9:30

**Q.** Does there always exist a schedule equal to depth of intervals?

---

Slides based on Kevin Wayne / Pearson-Addison Wesley
Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

Sort intervals by starting time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).

\[
d \leftarrow 0 \quad \text{number of allocated classrooms}
\]

for \( j = 1 \) to \( n \) {
    if (lecture \( j \) is compatible with some classroom \( k \))
        schedule lecture \( j \) in classroom \( k \)
    else
        allocate a new classroom \( d + 1 \)
        schedule lecture \( j \) in classroom \( d + 1 \)
        \( d \leftarrow d + 1 \)
}

Implementation. \( O(n \log n) \).
- For each classroom \( k \), maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.
Interval Partitioning: Greedy Analysis

**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**Theorem.** Greedy algorithm is optimal.

**Pf.**
- Let \( d \) = number of classrooms that the greedy algorithm allocates.
- Classroom \( d \) is opened because we needed to schedule a job, say \( j \), that is incompatible with all \( d-1 \) other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than \( s_j \).
- Thus, we have \( d \) lectures overlapping at time \( s_j + \varepsilon \).
- Key observation \( \Rightarrow \) all schedules use \( \geq d \) classrooms. ∙
Weighted Interval Scheduling
Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight/cost/value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
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.** \( S[j] = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- **Case 1:** j is selected.
  - 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:** j is not selected.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

Brute force algorithm.

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

**Compute-Opt**(\( j \)) {
    if (\( j = 0 \))
        return 0
    else
        return max\( (v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}
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.

\[ p(1) = 0, \quad p(j) = j-2 \]
**Improved Complexity**

**Top-down dynamic programming:** Memoization.

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

**Running Time.** $O(n \log n)$ to sort. $O(n^2)$ for straightforward computation of all $p(i)$. (Can be done in $O(n \log n)$ by also sorting jobs by start time.) $O(n)$ for iterative loop.

**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** {
    
    $S[0] = 0$
    
    for $j = 1$ to $n$
    
    $S[j] = \max(v_j + S[p(j)], S[j-1])$

}