**Approaches to Problem Solving**

- greedy algorithms
- dynamic programming
- backtracking
- divide-and-conquer
Interval Scheduling

Input: a set of time-intervals
Output: a subset of non-overlapping intervals
Objective: maximize \# of selected intervals
Interval Scheduling

Input: a set of time-intervals
Output: a subset of non-overlapping intervals
Objective: maximize # of selected intervals

Idea #1:
Select interval that starts earliest, remove overlapping intervals and recurse.
Interval Scheduling

Input: a set of time-intervals
Output: a subset of non-overlapping intervals
Objective: maximize # of selected intervals

Idea #2:
Select the shortest interval, remove overlapping intervals and recurse.
Interval Scheduling

Input: a set of time-intervals
Output: a subset of non-overlapping intervals
Objective: maximize # of selected intervals

Idea #3:
Select the interval with the fewest conflicts, remove overlapping intervals and recurse.
Interval Scheduling

Input: a set of time-intervals
Output: a subset of non-overlapping intervals
Objective: maximize # of selected intervals

Idea #4:
Select the earliest finishing interval, remove overlapping intervals and recurse.
Select the earliest finishing interval, remove overlapping intervals and recurse.
Interval Scheduling

INTERVAL-SCHEDULING( (s₁,f₁), …, (sₙ,fₙ) )
1. Remain = {1,…,n}
2. Selected = {}
3. while ( |Remain| > 0 ) {
   4. k ∈ Remain is such that fₖ = \min_{i \in \text{Remain}} f_i
   5. Selected = Selected ∪ {k}
   6. Remain = Remain - {k}
   7. for every i in Remain {
       8. if (sᵢ < fₖ) then Remain = Remain - {i}
   }
10. }
11. return Selected

Running time: 0(n²) – Think if can implement faster
Interval Scheduling

INTERVAL-SCHEDULING( (s₁,f₁), ..., (sₙ,fₙ) )

1. Remaining = {1,...,n}
2. Selected = {}
3. while ( |Remaining| > 0 ) {
   4. k ∈ Remaining is such that fₖ = minᵢ∈Remaining fᵢ
   5. Selected = Selected ∪ {k}
   6. Remaining = Remaining - {k}
   7. for every i in Remaining {
      8. if (sᵢ < fₖ) then Remaining = Remaining - {i}
   }
4. } }
10. return Selected

Thm: Algorithm works.
Interval Scheduling

Thm: Algorithm works.

Suppose our solution (the one returned by the algo) is \( \text{OUR} \)
Let OPT be an optimal solution \( \text{OPT} \)

By contradiction, assume \(|\text{OUR}| < |\text{OPT}|\).

We will gradually modify OPT into another optimal solution that will agree
more and more with OUR until we show that OUR solution is optimal, getting
a contradiction.

We will create \( \text{OPT}_1 \), an optimal solution that agrees with OUR on the 1st interval.

\[ \rightarrow \text{let } \text{OPT}_1 \text{ be like OPT, with 1st int. replaced by our 1st int.} \]
(Can do because our is earliest finishing)

We will create \( \text{OPT}_2 \), an optimal solution that agrees with OUR on the first 2 intervals.

\[ \rightarrow \text{let } \text{OPT}_2 \text{ be like } \text{OPT}_1 \text{, with 2nd int. replaced by our 2nd int.} \]
(can do, because our is earliest finishing among non-overlap. with our 1st
and intervals that do not overlap with OPT 1st do not overlap with our 1st,
hence our 2nd int. finishes before OPT 2nd)

Etc. until \( \text{OPT}_k = \text{OUR} \), hence OUR is optimal! \( \square \)

Which type of algorithm did we use? GREEDY
Schedule All Intervals

Input: a set of time-intervals

Output: a partition of the intervals, each part of the partition consists of non-overlapping intervals

Objective: minimize the number of parts in the partition
Schedule All Intervals

Input: a set of time-intervals

Output: a partition of the intervals, each part of the partition consists of non-overlapping intervals

Objective: minimize the number of parts in the partition
Schedule All Intervals

Input: a set of time-intervals

Output: a partition of the intervals, each part of the partition consists of non-overlapping intervals

Objective: minimize the number of parts in the partition

Def: $\text{depth} = \max \text{ (over time } t \text{) number of intervals that are “active” at time } t$
Schedule All Intervals

Input: a set of time-intervals
Output: a partition of the intervals, each part of the partition consists of non-overlapping intervals
Objective: minimize the number of parts in the partition

Def: depth = max (over time t) number of intervals that are “active” at time t
**Schedule All Intervals**

**Def:** depth = \( \text{max} \) (over time t) number of intervals that are "active" at time t

**Observation 1:** Need at least depth parts (labels).

SCHEDULE-ALL_INTERVALS \((s_1,f_1), \ldots, (s_n,f_n)\)

1. Sort intervals by their starting time
2. for \(j=1\) to \(n\) do
3. \(\text{Consider} = \{1,\ldots,\text{depth}\}\)
4. for every \(i<j\) that overlaps with \(j\) do
5. \(\text{Consider} = \text{Consider} - \{\text{Label}[i]\}\)
6. if \(|\text{Consider}| > 0\) then
7. \(\text{Label}[j] = \text{anything from Consider}\)
8. else
9. \(\text{Label}[j] = \text{nothing}\)
10. return \(\text{Label}[]\)
Thm: Every interval gets a real label.

Corollary: Algo returns an optimal solution (i.e. it works!).

Running time: the algo below is \(O(n^2)\)

```
SCHEDULE-ALL_INTERVALS ( (s_1,f_1), ... , (s_n,f_n) )
1. Sort intervals by their starting time
2. for j=1 to n do
3.    Consider = {1,...,depth}
4.    for every i<j that overlaps with j do
5.        Consider = Consider - { Label[i] }  
6.    if |Consider| > 0 then
7.        Label[j] = anything from Consider
8.    else
9.        Label[j] = nothing
10. return Label[]
```
Weighted Interval Scheduling

Input: a set of time-intervals, each interval has a cost
Output: a subset of non-overlapping intervals
Objective: maximize the sum of the costs in the subset

Greedy does not work.
Weighted Interval Scheduling

**Input:** a set of time-intervals, each interval has a *cost*

**Output:** a subset of non-overlapping intervals

**Objective:** maximize the sum of the costs in the subset

**Rough Outline:**

1. First attempt: (a correct algo but slow)
   - Use backtracking (a set of intervals \( \text{JOE} \))
   - If \( |\text{JOE}| = 1 \) then return the sum of the costs in \( \text{JOE} \)
   - Let \( I \) be an interval in \( \text{JOE} \)
   - \( I \) options: take \( I \) or not take \( I \)
   - if take \( I \):
     - \( \text{take}I = \text{cost}(I) + \text{backtracking}(\text{JOE-}\text{overlapping with } I) \)
   - if not take \( I \):
     - \( \text{notake}I = \text{backtracking}(\text{JOE-}I) \)
   - return max \[ \text{take}I, \text{notake}I \]

**Note:**
We will first compute the cost of an opt. sol.
Weighted Interval Scheduling

**Input:** a set of time-intervals, each interval has a cost

**Output:** a subset of non-overlapping intervals

**Objective:** maximize the sum of the costs in the subset

\[
\text{WEIGHTED-SCHED-ATTEMPT}((s_1, f_1, c_1), \ldots, (s_n, f_n, c_n))
\]

1. sort intervals by their finishing time
2. return \text{WEIGHTED-SCHEDULING-RECURSIVE} (n)

\[
\text{WEIGHTED-SCHEDULING-RECURSIVE} \ (j)
\]

1. if \ (j==0) \ then \ RETURN \ 0
2. \ k=j
3. while \ (\text{interval k and j overlap}) \ do \ k--
4. return

\[
\max(c_j + \text{WEIGHTED-SCHEDULING-RECURSIVE}(k), \text{WEIGHTED-SCHEDULING-RECURSIVE}(j-1))
\]
Weighted Interval Scheduling

Does the algorithm below work?

```
WEIGHTED-SCHED-ATTEMPT((s₁, f₁, c₁, ..., sₙ, fₙ, cₙ))
  1. sort intervals by their finishing time
  2. return WEIGHTED-SCHEDULING-RECURSIVE (n)

WEIGHTED-SCHEDULING-RECURSIVE (j)
  1. if (j==0) then RETURN 0
  2. k=j
  3. while (interval k and j overlap) do k--
  4. return max(c_j + WEIGHTED-SCHEDULING-RECURSIVE(k),
                 WEIGHTED-SCHEDULING-RECURSIVE(j-1))
```
Weighted Interval Scheduling

**Dynamic programming**! I.e. memorize the solution for $j$

WEIGHTED-SCHED-ATTEMPT((s₁,f₁,c₁),…,(sₙ,fₙ,cₙ))
1. sort intervals by their finishing time
2. return WEIGHTED-SCHEDULING-RECURSIVE (n)

WEIGHTED-SCHEDULING-RECURSIVE (j)
1. if (j==0) then RETURN 0
2. k=j
3. while (interval k and j overlap) do k--
4. return
   \[
   \max(c_j + \text{WEIGHTED-SCHEDULING-RECURSIVE}(k), \text{WEIGHTED-SCHEDULING-RECURSIVE}(j-1))
   \]
Weighted Interval Scheduling

Heart of the solution:

\[ S[j] = \max \text{ cost of a set of non-overlapping intervals selected from the first } j \text{ intervals} \]

Another part of the heart: how to compute \( S[j] \)?

\[
S[j] = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ c_j + S[k], S[j-1] \} & \text{otherwise}
\end{cases}
\]

where \( \text{where } k \text{ is the last non-overlapping interval with } c_j \text{ the } j\text{-th interval} \)

Finally, what do we return?

\( S[n] \)
Weighted Interval Scheduling

Heart of the solution:

\[ S[j] = \text{max cost of a set of non-overlapping intervals selected from the first } j \text{ intervals} \]

**WEIGHTED-SCHED** \(((s_1, f_1, c_1), \ldots, (s_n, f_n, c_n))\)
1. Sort intervals by their finishing time
2. Define \( S[0] = 0 \)
3. for \( j=1 \) to \( n \) do
4. \( k = j \)
5. while (intervals \( k \) and \( j \) overlap) do \( k-- \)
6. \( S[j] = \text{max}( S[j-1], c_j + S[k] ) \)
7. RETURN \( S[n] \)
Weighted Interval Scheduling

Reconstructing the solution:

idea: go through the array backwards, starting at \( j=n \), then if \( S[j] \) was computed from \( S[j-1] \), then do not take the \( j \)-th interval and move to \( j-1 \). Oh, \( S[j] \) was computed from \( c_j + S[k] \), i.e. we do take the \( j \)-th interval and move to \( j+k \).

WeighTed-SChed \((s_1, f_1, c_1), \ldots, (s_n, f_n, c_n)\)

1. Sort intervals by their finishing time
2. Define \( S[0] = 0 \)
3. for \( j=1 \) to \( n \) do
4. \( k = j \)
5. while (intervals \( k \) and \( j \) overlap) do \( k-- \)
6. \( S[j] = \max( S[j-1], c_j + S[k] ) \)
7. if \( S[j] = S[j-1] \) then \( \text{prev}[j] = j-1 \) else \( \text{prev}[j] = k \)
8. while \( (j>0) \) :
9. if \( \text{prev}[j] \neq j-1 \) then output the \( j \)-th interval
10. \( j = \text{prev}[j] \)
11. RETURN \( S[n] \)
Longest Increasing Subsequence

Input: a sequence of numbers
Output: an increasing subsequence
Objective: maximize length of the subsequence

Example: 2 3 1 7 4 6 9 5
Longest Increasing Subsequence

Input: a sequence of numbers \( a_1, a_2, \ldots, a_n \)

Output: an increasing subsequence

Objective: maximize length of the subsequence

Heart of the solution:

1. \( S[j] = \) the max length of an incr. subseq. of \( a_1, \ldots, a_j \) ending with the \( j \)-th elem.

2. \[ S[j] = \begin{cases} 0 & \text{if } j = 0 \\ 1 + \max_{k < j \text{ and } a_k < a_j} S[k] & \text{if } j > 0 \end{cases} \]

3. The answer is \( \max_j S[j] \)

For int. scheduling:

\( S[j] \): the opt. sol. when looking at the first \( j \) intervals (assuming sorted)
Longest Increasing Subsequence

Input: a sequence of numbers
Output: an increasing subsequence
Objective: maximize length of the subsequence

Heart of the solution:

\[ S[j] = \text{the maximum length of an increasing subsequence of the first } j \text{ numbers ending with the } j\text{-th number} \]
**Longest Increasing Subsequence**

**Input:** a sequence of numbers $a_1, a_2, \ldots, a_n$

**Output:** an increasing subsequence

**Objective:** maximize length of the subsequence

Heart of the solution:

$$S[j] = \text{the maximum length of an increasing subsequence of the first } j \text{ numbers ending with the } j\text{-th number}$$

$$S[j] = 1 + \text{maximum } S[k] \text{ where } k < j \text{ and } a_k < a_j$$

What to return?
Longest Increasing Subsequence

LONGEST-INCR-SUBSEQ \( (a_1, \ldots, a_n) \)

1. for \( j = 1 \) to \( n \) do
2. \( S[j] = 1 \)
3. for \( k = 1 \) to \( j - 1 \) do
4. if \( a_k < a_j \) and \( S[j] < S[k] + 1 \) then
5. \( S[j] = S[k] + 1 \)
6. return \( \max_j S[j] \)