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: (greedy algo *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.

Claim: it works!
Interval Scheduling

INTERVAL-SCHEDULING( (s_0, f_0), ..., (s_{n-1}, f_{n-1}) )

1. Remain = {0,...,n-1}
2. Selected = {}
3. while ( |Remain| > 0 ) {
4.   \( k \in \text{Remain} \) is such that \( f_k = \min_{i \in \text{Remain}} f_i \)
5.   Selected = Selected \( \cup \) {k}
6.   Remain = Remain - {k}
7.   for every i in Remain {
8.     if \( s_i < f_k \) then Remain = Remain - {i}
9.   }
10. }
11. return Selected

Select the earliest finishing interval, remove overlapping intervals and recurse.
Interval Scheduling

INTERVAL-SCHEDULING( (s_0, f_0), ..., (s_{n-1}, f_{n-1}) )
1. Remain = {0,...,n-1}
2. Selected = {} 
3. while ( |Remain| > 0 ) { 
4. k \in Remain is such that f_k = \min_{i \in Remain} f_i 
5. Selected = Selected \cup \{k\} 
6. Remain = Remain - \{k\} 
7. for every i in Remain { 
8. if (s_i < f_k) then Remain = Remain - \{i\} 
9. } 
10. } 
11. return Selected

Running time: 0(n^2)

Note: can do in O(n log n) by sorting the intervals by their finishing times, then do O(n) pass through the sorted list to select the set of intervals.
**Interval Scheduling**

INTERVAL-SCHEDULING( \((s_0,f_0), \ldots, (s_{n-1},f_{n-1})\) )

1. \(\text{Remain} = \{0,\ldots,n-1\}\)
2. \(\text{Selected} = \{}\)
3. while (\(|\text{Remain}| > 0\) ) {
4. \(k \in \text{Remain} \) is such that \(f_k = \min_{i \in \text{Remain}} f_i\)
5. \(\text{Selected} = \text{Selected} \cup \{k\}\)
6. \(\text{Remain} = \text{Remain} - \{k\}\)
7. for every \(i\) in \(\text{Remain}\) {
8. \(\text{if } (s_i < f_k) \text{ then } \text{Remain} = \text{Remain} - \{i\}\)
9. \}
10. }
11. return Selected

Thm: Algorithm works.
Interval Scheduling

Thm: Algorithm works.

Let \text{OUR} be the set of intervals scheduled by the Algorithm.
Let \text{OPT} be an optimal set of intervals (i.e., maximum possible # intervals).

We will show that \text{OUR} contains the same # intervals as \text{OPT}.

Let's consider the first intervals where \text{OUR} and \text{OPT} differ.

**Observation 1:** interval \(x\) has to finish before (or at the same time) as interval \(y\),
beacuse otherwise we could choose \(y\) instead of \(x\) in our Algorithm
(to somebody earlier)

we can create \text{OPT}_2 with the same # intervals as \text{OPT} and \text{OPT}_2 will agree with OUR solution
on all intervals left of \(x\), including \(x\)

Which type of algorithm did we use? greedy

\text{OPT}_2 \rightarrow \text{OPT} with \(y\) replaced by \(x\)

Now OUR agrees with an optimum set on more intervals.
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: depth = max (over time t) number of intervals that are “active” at time t
Schedule All Intervals

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

Observation 1: Need at least \textit{depth} parts (labels).

\begin{verbatim}
SCHEDULE-ALL_INTERVALS ( (s_0,f_0), \ldots, (s_{n-1},f_{n-1}) )
1. Sort intervals by their starting time
2. for j=0 to n-1 do
3. \hspace{1em} Consider = \{1,\ldots,\textit{depth}\}
4. \hspace{1em} for every i<j that overlaps with j do
5. \hspace{2em} Consider = Consider - \{ Label[i] \}
6. \hspace{1em} if |Consider| > 0 then
7. \hspace{2em} Label[j] = anything from Consider
8. \hspace{1em} else
9. \hspace{2em} Label[j] = nothing \text{ } \text{\{never happen!\}}$\text{ because at the start of interval } j$
10. return Label[]
\end{verbatim}
Schedule All Intervals

Thm: Every interval gets a real label.

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

Running time: $O(n^2)$

- think about how to do it faster

SCHEDULE-ALL_INTERVALS ( (s_0, f_0), ..., (s_{n-1}, f_{n-1}) )

1. Sort intervals by their starting time
2. for j = 0 to n-1 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
# 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

---

**WEIGHTED-SCHED-ATTEMPT**\((s_0,f_0,c_0),\ldots,(s_{n-1},f_{n-1},c_{n-1})\)

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

**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

Does the algorithm below work? **Yes, but...**

```
WEIGHTED-SCHED-ATTEMPT((s_0,f_0,c_0),...,(s_{n-1},f_{n-1},c_{n-1}))
1. sort intervals by their finishing time
2. return WEIGHTED-SCHEDULING-RECURSIVE (n-1)

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 $j$

$$S[j] = \max \text{ cost of a set of non-overlapping intervals when considering only intervals up to } j$$

**WEIGHTED-SCHED-ATTEMPT**($((s_0,f_0,c_0),\ldots,(s_{n-1},f_{n-1},c_{n-1}))$)

1. sort intervals by their finishing time
2. return **WEIGHTED-SCHEDULING-RECURSIVE** (n-1)

**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 \( S[j] \)?

\[ S[j] = \max \{ S[j-1], c_j + S[k] \} \quad \text{where } k \text{ is the last interval not overlapping with } j \]

Finally, what do we return?

\( S[n-1] \)
Weighted Interval Scheduling

Heart of the solution:

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

Running time: \(O(n^2)\)

WEIGHTED-SCHED \(((s_0,f_0,c_0), \ldots, (s_{n-1},f_{n-1},c_{n-1}))\)
1. Sort intervals by their finishing time
2. Define \(S[-1] = 0\)
3. for \(j=0\) to \(n-1\) 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 \(S[n-1]\)
Weighted Interval Scheduling

Reconstructing the solution:

WEIGHTED-SCHED ((s_0, f_0, c_0), ..., (s_{n-1}, f_{n-1}, c_{n-1}))

1. Sort intervals by their finishing time
2. Define S[-1] = 0
3. for j=0 to n-1 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. i = n-1
8. while (i ≥ 0) do:
9.   if S[i] == S[i-1] then i = i-1
10. else print i, k = i, while (intervals k and i overlap) do k--
11. RETURN S[n-1]

\[ O(n) \text{ (this part, the above is } O(n^2) \text{) } \]
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

$S[j]$: the max length of an increasing subsequence taken from the first $j$ elements that ends in the $j$th elem.

$S[j] = 1 + \max_{k < j} S[k]$ if $a_k < a_j$

Correct:

```
1 2 1 3 3 4 5 4
```

Not correct:

```
1 2 2 3 3 4 5 5
```
Longest Increasing Subsequence

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

Heart of the solution:
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**

\[ \text{LONGEST-INCRR-SUBSEQ} \ (a_0, ..., a_{n-1}) \]

1. for \( j = 0 \) to \( n-1 \) do
2. \( S[j] = 1 \)
3. for \( k = 0 \) to \( j-1 \) do
4. \( \text{if } a_k < a_j \text{ and } S[j] < S[k] + 1 \) then
5. \( S[j] = S[k] + 1 \)
6. return \( \max_j S[j] \)