Approaches to Problem Solving

- greedy algorithms
- dynamic programming
- backtracking
- divide-and-conquer
- reduction to another problem
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: not work!
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.

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.

works! we'll prove it
**Interval Scheduling**

INTERVAL-SCHEDULING( (s₀, f₀), ..., (sₙ₋₁, fₙ₋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( (s₀, f₀), ..., (sₙ₋₁, fₙ₋₁) )

1. Remain = {0,...,n-1}
2. Selected = {}
3. while ( |Remain| > 0 ) {
   4. k ∈ Remain is such that fₖ = minᵢ∈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

Running time: $O(n^2)$ but can be implemented in $O(n \log n)$ + single pass through the sorted list.
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} \text{ 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

Thm: Algorithm works.
Interval Scheduling

Thm: Algorithm works.

Proof: Let \( S \) be the solution returned by the algorithm.

Let \( S_{opt} \) be an optimum solution.

By contradiction, suppose \(|S| < |S_{opt}|\)

Let's compare \( S \) and \( S_{opt} \):

\[ X \]

\[ \text{Just} \]

Maybe it looks like this:

\[ \text{Observation 1:} \]

\[ \leftarrow \text{would not happen because our algo would have taken } S_{opt} \text{ instead of } I \text{ in step 4.} \]

\[ \text{Observation 2:} \]

\[ \text{Exchange of 4th intervals, agreement on first 3} \]

\[ \text{Exchange argument: replace 1st yellow interval by 1st blue, to get another optimum solution: no overlaps, and } |S_{opt}| = |S_{opt}|_2 \]

\[ \text{blue for yellow} \]

Which type of algorithm did we use?

next, do the same w. 2nd intervals (2nd blue finishes bef. 2nd yellow, exchange)

get \( S_{opt1} \), then exchange 3rd intervals, etc. until \( S_{opt} \) = \( S \)
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

Idea: run the previous algo, assign classes to 1st person, etc. remaining classes assign to remaining people.
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

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

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

\begin{verbatim}
SCHEDULE-ALL_INTERVALS ((s0,f0), ..., (sn-1,fn-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 \text{ \{never happens, see previous slide\}}
9. Label[j] = nothing
10. return Label[]
\end{verbatim}
Thm: Every interval gets a real label. ✓

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

Running time: The algo below is $O(n^3)$ can be implemented in $O(n \log n + n \cdot \text{depth})$ or $O(n \log n)$ if sorted.

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,...,\text{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 \textit{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 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_1,f_1,c_1),…,(s_n,f_n,c_n))
1. sort intervals by their finishing time
2. return WEIGHTED-SCHEDULING-RECURSIVE (n)

WEIGHTED-SCHEDULING-RECURSIVE (j)
1. if (j==0) then RETURN 0 // cost 0 w/ no intervals
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

Does the algorithm below work?

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

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

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

```
WEIGHTED-SCHED-ATTEMPT((s_1,f_1,c_1),...,(s_n,f_n,c_n))
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

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

WEIGHTED-SCHED-ATTEMPT((s_1,f_1,c_1),…,(s_n,f_n,c_n))
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 intervals

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

\[ S[j] = \max \{ c_j + S[k], S[j-1] \} \]

the same as before, i.e. k is the largest index s.t. k.k, j-k intervals do not overlap

Finally, what do we return?

\[ S[n] \]
Weighted Interval Scheduling

Heart of the solution:

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

```plaintext
WEIGHTED-SCHED ((s_1,f_1,c_1), ..., (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:

WEIGHTED-SCHED ((s<sub>1</sub>,f<sub>1</sub>,c<sub>1</sub>), ... , (s<sub>n</sub>,f<sub>n</sub>,c<sub>n</sub>))
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<sub>j</sub> + S[k] )
7. if S[j] = S[j-1]: pred[j] = j-1; else: pred[j] = k
8. \( \hat{j} = n \)
9. while (\( \hat{j} > 0 \)):
   if S[\( \hat{j} \)] ≠ S[\( \hat{j} - 1 \)]: output the j-th interval
   \( \hat{j} = \text{pred}[\hat{j}] \)
10. 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

Output: an increasing subsequence

Objective: maximize length of the subsequence

Heart of the solution:

1. \( S[j] = \) the max length of an increasing subsequence of the first \( j \) elements, ending with \( j \)th element \( a_j \)

2. \( S[j] = 1 + \max_{k: k < j, a_k < a_j} S[k] \)

3. return \( \max_j S[j] \)
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 $a_1, a_2, ..., a_n$
Output: an increasing subsequence
Objective: maximize length of the subsequence

Heart of the solution:

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

What to return?
LONGEST-INCR-SUBSEQ \((a_0, \ldots, a_{n-1})\)
1. for \(j = 0\) to \(n-1\) do
2. \(S[j] = 1\)
3. for \(k = 0\) to \(j-1\) do
4. \(\text{if } a_k < a_j \text{ and } S[j] < S[k] + 1 \text{ then}\)
5. \(S[j] = S[k] + 1\)
6. return \(\max_j S[j]\)