An Introduction to Greedy Algorithms: Interval Scheduling

Reading: KT 4.1, CLRS 16.1 — 16.3

CS231 Fundamental Algorithms
Lyn Turbak

Department of Computer Science
Wellesley College

Tue April 19, 2022 *(Revised Fri Apr 22)*

Greedy Algorithms

An algorithm is greedy if it makes a choice to optimize some criterion at every step.

Whether this leads to a globally optimal solution depends on the nature of the problem: sometimes it does, sometimes it doesn't!

Example: Hill Climbing

You are trying to climb to the top of a mountain in a dense fog. A greedy strategy is to take each step in the direction of the highest gradient.

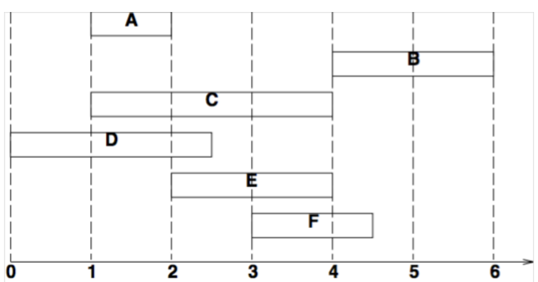
Whether or not the greedy strategy finds the top of the mountain depends on the terrain!

Interval Scheduling Problem

Given a set of intervals (a.k.a. events, activities) with start and finish times, return a subset of **compatible** (no two overlap in time) intervals with the **most** intervals.

E.g., intervals could be events that use a room, times to use a telescope, etc.

Example: What's the largest compatible subset of the following intervals?



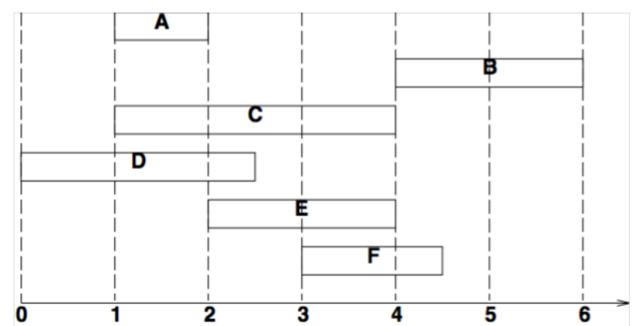
Important: Trying to maximize the **number** of nonoverlapping intervals, **not** the total time of these intervals (that's a different problem!)

Interval Representation

Each interval must have a start time and finish time. If v is an interval, use start(v), s(v) or s_v for its start time and finish(v), f(v) or f_v for its finish time.

It might also have a UID, a label, and extra information (e.g., variants of interval scheduling can associate a **value**/**weight** with each interval.)

```
(1, A, 1, 2), (2, B, 4, 6), (3, C, 1, 4),
(4, D, 0, 2), (5, E, 2, 4), (6, F, 3, 4.5)
```



Greedy Interval Scheduling Algorithm: Idea & Example

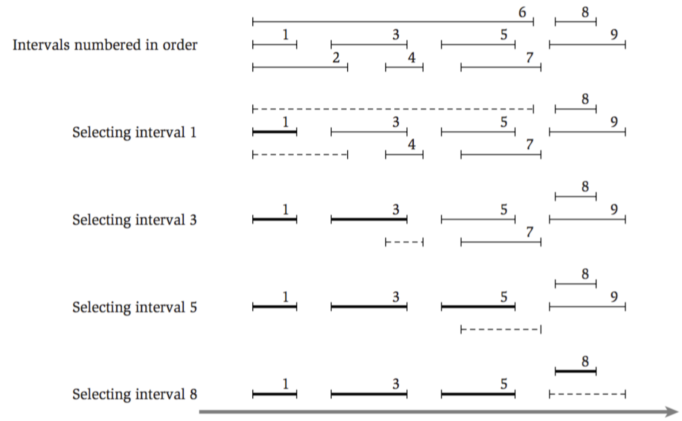
Idea: greedily choose the remaining interval with the earliest finish time, since this will maximize time available for other intervals.

Example (KT Fig 4.2):

Greedy Interval Scheduling Algorithm: Idea & Example

Idea: greedily choose the remaining interval with the earliest finish time, since this will maximize time available for other intervals.

Example (KT Fig 4.2):



Greedy Interval Scheduling Algorithm: Pseudocode

The following functions use linked lists, but could use arrays instead

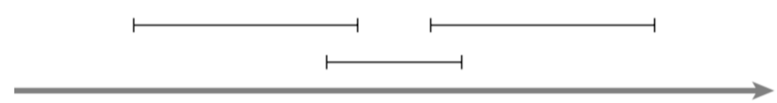
```
function ScheduleIntervals(intervals) # linked list of intervals
    sorted ← sortByFinish(intervals) # sort intervals by finish time (ascending)
    GreedilySchedule(sorted)

function GreedilySchedule(intervalsSortedByFinish) # linked list sorted by finish time
    if empty?(intervalsSortedByFinish) then
        return emptyList() # empty linked list
    else
        first ← head(intervalsSortedByFinish) # head returns first item of list
        rest ← tail(intervalsSortedByFinish) # tail returns all but first item of list
        while start(head(rest)) < finish(first) do
            rest ← Tail(rest) # remove intervals overlapping with first
        return prepend(first, Greedily-Schedule(rest))
            # prepend adds item to front of linked list
```

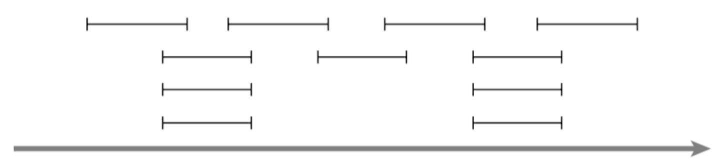

Not All Greedy Strategies Work

Earliest First (KT Fig 4.1a):

Shortest First (KT Fig 4.1b):



Fewest Overlaps First (KT Fig 4.1c):



Why Does GreedySchedule Work? (Part 1)

Claim: GreedySchedule returns a list containing maximal compatible subset of input intervals

Proof:

- Sorting intervals by finish time + **while** loop to remove intervals that overlap with first (interval with earliest finish time) guarantees result is compatible.
- There may be other compatible results with the same number of intervals, but not more.

  Assume GreedySchedule returns intervals result $R = [i_1, i_2, \ldots, i_k]$ and there is another optimal solution with intervals $O = [j_1, j_2, \ldots, j_m]$.

  Idea: *R* ``stays ahead" of *O*.

  KT 4.2: *For all indices* $r \leq k$ *we have* $f(i_r) \leq f(j_r)$

  Proof of 4.2 by induction:

  - *Base case:* true for r = 1, since i_1 chosen as interval with earliest finish time
  - *Inductive step:*

    $f(i_{r-1}) \leq f(j_{r-1})$ (by IH)

    $\leq s(j_r)$ (by fact that O is a solution with compatible intervals)

    So j_r is available as a candidate when GreedySchedule chooses next interval after i_{r-1}. Since it chooses i_r, either $i_r = j_r$ or $f(i_r) \leq f(j_r)$

Why Does GreedySchedule Work? (Part 2)

KT 4.3 (adapted) *GreedySchedule returns an optimal list R*

Proof by Contradiction

Assume there's an optimal result O with $m = len(O) > len(R) = k$

By 4.2, $f(i_k) \leq f(j_k)$

By $m > k$, O contains an interval j_{k+1} where $f(i_k) \leq f(j_k) \leq s(j_{k+1})$

So j_{k+1} is a compatible interval that's still avaiable after i_k is processed.
But GreedySchedule terminates only when the list of remaining compatible intervals is empty: a contradiction.

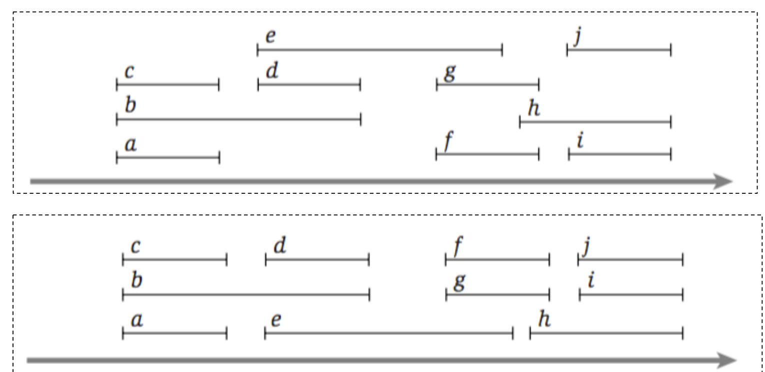
GreedySchedule Running Time

- For *n* intervals, sorting them by finish time takes θ(n log(n)) (could even be θ(n+k) if times are integers in restricited range)
- Processing sorted intervals and constructing result touches each interval once, taking time θ(n)
- Total time = θ(n log(n)) + θ(n) = θ(n log(n))

Interval Partitioning (Coloring) Problem

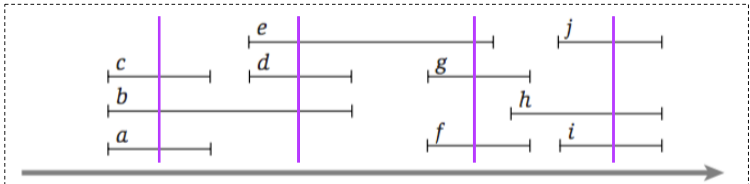
Schedule all requests using the fewest resources (e.g., rooms, telescopes) as possible.

Example (KT Fig 4.4)



Interval Partitioning: Depth

The depth of a set of intervals is the maximal number of intervals that overlap at a particular time.



Clearly the minimal number of partitions must be at least the depth.

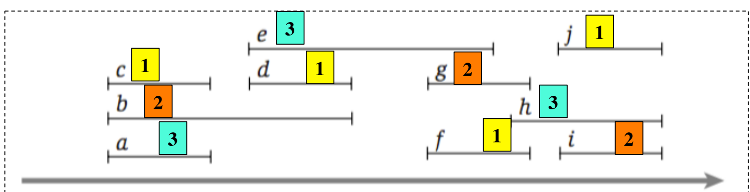
Interval Partitioning: Psuedocode

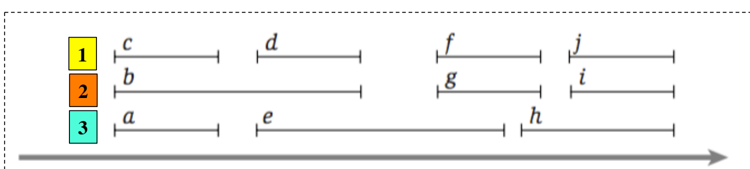
```
function LabelIntervalsByPartition(intervals)  # this time use arrays
    n ⟵ len(intervals)
    d ⟵ depth(intervals) # find depth of intervals
    labeling ⟵ new array of length n with each slot None, indexed by ID of interval
    sorted ⟵ sortByStart(intervals) # sort intervals by start time (ascending)
    for j in [1..n] do
        labels ⟵ {1 .. d} # possible labels (``colors'')  for j
        for i in [1..(j-1)] do # sort intervals by start time (ascending)
            if sorted[i] overlaps sorted[j] then
                labels ⟵ labels– labeling(ID(sorted[i])) # remove label of overlapping interval
        labeling (ID(sorted[j])) ← chooseOneOf(labels) # arbitrary choice of remaining labels
    return labeling
```


Interval Partitioning: Example

Example (KT Fig 4.4) revisited

Depth = 3: labels = 1 2 3





LabelIntervalsByPartition: Correctness

KT 4.5: In the greedy LabelIntervalsByPartition

1. Every interval is assigned a label (no **None** slots in labeling at end)

   Why? When interval sorted[j] is reached, at most d-1 intervals with earlier start times can overlap with it, so there is always at least one label left in {1..d} to assign to sorted[j] in labeling.

2. No two overlapping intervals receive the same label

   Why? Labels for all preceding overlapping intervals are removed from consideration from the labels set before one label is chosen.

Generic Greedy Algorithm

A greedy algorithm makes the locally optimal choice at every step:

```
function GreedyAlgorithm(problem)
    soln ← {}
    subproblem ← problem
    while not Solution?(soln, problem) do
        choice ← GreedyChoice(subproblem)
            # GreedyChoice makes locally optimal choice
            # for current subproblem.
        soln ← soln ∪ {choice}
            # Meaning of ∪ can depend on problem.
        subproblem ← Simplify(subproblem, choice)
            # Simplify gives remaining subproblem
            # after choice is made.
    return soln
```