Homework 13

Brian Knotten, Brett Schreiber, Brian Falkenstein

September 26, 2018

10

a

Given an array of n intervals (sorted by lowest start point), the following is a recursive solution to determine the biggest collection of non-overlapping intervals.

```
# Assume the intervals are in order by start time.
intervals[n];

function A(start, end, i) {
    if i > n:
        return 0

    let interval = intervals[i]
    let length = interval.end - interval.start

    # Discard the potential interval if it falls out of bounds of the allowed space
    # and continue with the rest of the intervals.
    if interval.start < start || interval.end > end:
        return return A(start, end, i + 1)


    # Otherwise, take the maximum value of either taking the interval in the solution or not.
    # If it is in the solution, then the start point for the recursive subproblem must be the end of the in
    # the interval occupies the space between start and its end. If there was a smaller interval that could
    # it would have been considered earlier, since the intervals are sorted by start time.
    return max(
        A(interval.end, end, i + 1) + length,
        A(start, end, i + 1)
    )
}
```

This can be converted into a dynamic programming algorithm. The algorithm will operate on an $n \times n$ array, A. Each cell in the array will be a tuple of the form (l, e) where l is the length of the sequence of intervals chosen at that point, and e is the end point of the last interval.
Some definitions: I_i is interval i from the input (assuming the intervals are sorted in order by their start point). $.len$ is the length of a sequence of intervals. $.end$ is the endpoint of a sequence of intervals, and $.start$ is the start point of a sequence of intervals.
We can define $A[i, k]$ by:

1. If we can add I_i to $A[i-1, k]$ (if $I_i.start >= A[i-1, k].end$), then $A[i, k] = \max($
   $A[i, k]$.len,
   $A[i-1, k].len + I_i.len)$
   That is, take the max length of what was already in the cell, and the length you'd get by adding I_i. If the lengths are the same, take the set of intervals with the earlier end point.

2. If I_i cannot be added, then $A[i,k] = \max$
   $A[i,k].len$
   $A[i-1,k].len$
   Same as in rule 1, if the lengths are the same, take the one with the earlier end point.

The output of the algorithm will be the maximum length value found in the last row. This algorithm works by building up the table, which is of size $n \times n$, so it will run in polynomial time.

b

Define C to be the collection of intervals sorted by start time and L to be the total sum of the lengths of the intervals in C. The decision tree can be constructed as follows: for $0 \leq i \leq n$ at depth i have the right branch add interval v_i to C and add the length of v_i to L and have the left branch exclude v_i from C and L. This tree would enumerate all possible combinations of intervals and have 2^n leaves.
The tree can be pruned using the following rules:

1. If a node v contains two or more intervals that overlap, prune the tree rooted at v. It cannot possibly have the solution due to the restriction against overlaps.

2. If two nodes u and v rooted at the same level have equal sums of their respective intervals' lengths, and u's last interval ends later than v's last interval, then prune u. v has more free space and therefore can consider more future intervals, so it is the better choice. (v could have free space in between its intervals that u doesn't have, but since the intervals are considered in sorted order, any future intervals will start after the end of this free space.)

3. If two nodes u and v are rooted at the same level and end with the same interval (i.e. the last interval included in u is i and the last interval included in v is also i) and u has a shorter total length, prune u. Because u and v have the same endpoint (the endpoint of i) and we are considering the intervals in sorted order by starting point, this leaves a maximum of one node per interval considered thus far per level i.e. at level m there are a max of m intervals being considered.

The first rule works by the definition of the problem: no two intervals can overlap, so any combination of intervals in the subtree of that node is invalid. The second rule works because two nodes at the same depth have the same intervals left to consider and, because we are considering the intervals in sorted order by starting point, none of the remaining nodes can be inserted into earlier spaces. Therefore the length of nodes in the subtree of the remaining node will be at least as long as those in the subtree(s) of the pruned node(s). The third rule works because by the same logic as the second.
At any given level m, there are at most m nodes being considered and there are n levels (one for each interval), because in the worst case, all n intervals overlap each other. Moreover, these intervals can be ordered by their length. Therefore the array cell $A[i,k]$ is defined as: the length of the first i intervals considered with the kth largest interval sum. So the algorithm is the following:

```
for k = 0 to n do:
    A[0][k] = 0

for i = 1 to n do:
    for k = 1 to n do:
        if I[i] does not overlap:
            A[i][k] = max(A[i][k], A[i - 1][k] + I[i])

Output A[n][m]
```

This algorithm is $O(n^2)$ in the input size since it fills up an $n \times m$ table.