Assignment 5

Sample Solution

CSCI 3110 — Fall 2018

I provide two solutions here.

The first one is the one that I intended you to come up with and which is very close to the algorithm for partitioning a sequence into monotonically increasing subsequences, discussed in one of the tutorials. This algorithm is quite obviously greedy because it starts with a single classroom, schedules classes in the available classroom (thereby greedily minimizing the number of classrooms needed for the classes processed so far) and only allocates a new classroom when it cannot schedule the next class in one of the classrooms already allocated.

The second algorithm ends up constructing the exact same allocation of classes to classrooms but is less obviously greedy. Its advantage is that it reuses (a variation of) the algorithm from class for scheduling as many classes as possible in a given classroom. Kudos to one of your classmates for coming up with this algorithm and discussing it with me during one of the office hours.

Solution 1

(a) Let $C_1, C_2, \ldots, C_n$ denote the n classes and let $I_1, I_2, \ldots, I_n$ be their time intervals. For each interval I_j, we use s_j and e_j to denote its starting and ending times, respectively. We sort the intervals by increasing starting times, that is, we arrange them so that $s_1 < s_2 < \cdots < s_n$. Now we allocate a classroom R_1 and schedule the first class C_1 in room R_1. We also record e_1 as the time l_1 the last class currently scheduled in R_1 ends. In general, if we have allocated h classrooms $R_1, R_2, \ldots, R_h$ so far, then we maintain h times $l_1, l_2, \ldots, l_h$, where l_i is the ending time of the last class scheduled in R_i. Now we process the classes $C_2, C_3, \ldots, C_n$ in order. For each class C_j, we check whether there exists a classroom R_i such that $l_i < s_j$. If so, we schedule C_j in room R_i and set $l_i = e_j$. (If there is more than one classroom R_i such that $l_i < s_j$, we choose one arbitrarily.) If $l_i > s_j$ for all classrooms we have allocated so far, we allocate a new classroom R_{h+1}, schedule C_j in room R_{h+1}, and set $l_{h+1} = e_j$.

(b) First we argue that the schedule produced by our algorithm is valid, that is, no two classes scheduled in the same classroom overlap. To this end, we prove that the algorithm maintains the following invariant:

> No two classes scheduled in the same classroom overlap and, for all i, all classes scheduled in room R_i end at or before time l_i.

This invariant clearly holds after we schedule C_1 because so far we have scheduled a single class, C_1, in a single room, R_1, and we set $l_1 = e_1$. So assume the invariant holds before scheduling the jth class C_j.

If we place C_j into a new room R_{h+1}, then the invariant is maintained: C_j is the only class scheduled in R_{h+1} and thus cannot overlap any classes scheduled in the same room, and it ends at time $l_{h+1} = e_j$. For the other classrooms, the sets of classes scheduled in them do not change, nor do the recorded maximum ending times $l_1, l_2, \ldots, l_h$, so the invariant is maintained for these classes as well.

If we place C_j into an existing room R_i, then $s_j > l_i$. Since all classes already scheduled in R_i end at or before time l_i, they do not overlap C_j. Also, since $l_i < s_j < e_j$ and e_j becomes the new value of l_i, all classes scheduled in room R_i continue to end at or before time l_i. As in the previous case, for all rooms other than R_i, the invariant cannot be violated because the set of classes we schedule in them and the maximum ending times we record for these rooms do not change.

Once we are done scheduling all n classes, the invariant states explicitly that there are no two overlapping classes scheduled in the same room. So the schedule we produce is valid.

Now assume the schedule produced by the above algorithm uses k classrooms $R_1, R_2, \ldots, R_k$. If $k = 1$, we clearly cannot do better because we need at least one classroom. So assume $k > 1$. Since we start with R_1 as the only room allocated initially, the other rooms are allocated one by one in response to our failure to schedule some classes in the rooms we have allocated so far. Let C_{j_k} be the first class we schedule in room R_k. Let $l_1, l_2, \ldots, l_{k-1}$ be the ending times recorded for rooms $R_1, R_2, \ldots, R_{k-1}$ at the time we schedule C_{j_k}. For all $1 \le i \le k-1$, l_i is the ending time e_{j_i} of a class C_{j_i} scheduled in room R_i. Since this class is scheduled before C_{j_k}, it satisfies $s_{j_i} < s_{j_k}$. Since we schedule C_{j_k} in a new room R_k, we also have $s_{j_k} < l_i = e_{j_i}$. This proves that the intervals $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ all contain the time s_{j_k}. Thus, the classes $C_{j_1}, C_{j_2}, \ldots, C_{j_k}$ must be scheduled in different classrooms, that is, any valid schedule needs to use at least k classrooms. Since our schedule uses k classrooms, it is optimal.

(c) Almost all of the above algorithm can be implemented in $O(n \lg n)$ time, assuming the list of classes is given as an array or linked list and the output we produce is a linked list of classrooms, each represented as a linked list of classes scheduled in it. Sorting the classes by increasing starting times then takes $O(n \lg n)$ time using an optimal sorting algorithm, such as Merge Sort. Then we inspect the classes one at a time. For each class C_j, once we have chosen the room R_i to schedule it in, updating l_i and appending C_j to the list of classes scheduled in R_i takes constant time. If R_i is a new room we allocate to accommodate C_j, appending R_i to the list of rooms also takes constant time. Thus, we spend constant time per class, $O(n)$ time in total, on this part of the algorithm. The expensive part is testing whether there exists a classroom R_i such that $l_i < s_j$. To find R_i, we simply scan the linked list of classrooms and test this condition for each inspected room. If we have allocated h classrooms by the time we schedule class C_j, then the cost of scanning the list of classrooms to find a room where to schedule class C_j is in $O(h)$. Since $h \le k$, the cost per class is in $O(k)$, $O(kn)$ for all n classes. By adding the costs of the different parts of the algorithm, we obtain the desired complexity of $O(n \lg n + kn)$.

(d) As just argued, the only part of the algorithm whose cost is not in $O(n \lg n)$ (if $k \in \omega(\lg n)$) is finding the classroom where to schedule each class. Here we reduce the $O(kn)$ cost of this part to $O(n \lg k)$, thereby reducing the overall cost of the algorithm to $O(n \lg n + n \lg k) = O(n \lg n)$ (since every room contains at least one class, we have $k \leq n$.)

We maintain the maximum ending times $l_1, l_2, \ldots, l_h$ of rooms $R_1, R_2, \ldots, R_h$ in a binary heap.[1] The binary heap supports three types of operations in $O(\lg h)$ time, where h is the number of elements it currently stores: Find the minimum ending time l_m (that's just the root of the heap). Delete the minimum ending time l_m (delete the root). Insert a new ending time l_i. Now, when trying to schedule a class C_j, there exists a room R_i such that $l_i < s_j$ if and only if $l_m < s_j$. Thus, finding the minimum ending time is sufficient to decide whether we should allocate a new room for C_j or schedule C_j in an existing room. If we can schedule C_j in an existing room, remember that the correctness of our algorithm did not depend on the choice of the room where we schedule C_j; any room R_i with $l_i < s_j$ is good enough. So we simply schedule C_j in room R_m if $l_m < s_j$. To reflect this in our data structure, we delete l_m from the heap and insert e_j as the new ending time of room R_m (which may no longer be the room with the minimum ending time, that is, e_j may not end up being the root of the heap). If we schedule C_j in a new room R_{h+1}, we only insert e_j into the heap as the ending time of room R_{h+1}. In summary, we perform at most three heap operations per class and each of these operations takes $O(\lg h) \subseteq O(\lg k)$ time. Thus, the total cost of maintaining the heap over the course of the algorithm is $O(n \lg k)$, as desired.

Solution 2

(a) An alternate strategy for constructing the same allocation of classes to classrooms as produced by the previous algorithm fills one classroom at a time. Once again, we sort the classes by their starting times $s_1, \ldots, s_n$. Let S_1 be this sorted list. This is the input we use to construct the set of classes to be scheduled in the first classroom R_1. In general, the ith iteration of the algorithm is given the list S_i of classes not already scheduled in classrooms $R_1, \ldots, R_{i-1}$ and selects from S_i the set of classes to be scheduled in room R_i. The classes in S_i not scheduled in room R_i are placed into the input list S_{i+1} for the next iteration. The ith iteration works as follows: If $S_i = \emptyset$, then there are no classes left to schedule and the algorithm terminates, having scheduled all classes in classrooms $R_1, \ldots, R_{i-1}$. Otherwise, it initialize the ending time ℓ_i of the last class scheduled in room R_i to be $\ell_i = -\infty$ and initializes $S_{i+1} = \emptyset$. Then it scans the classes in S_i in order. For each class C_j in S_i, if $s_j \geq \ell_i$, then C_j starts after the last class in R_i ends, so we can schedule C_j in room R_i. The algorithm does just that and reflects this decision by setting $\ell_i = e_j$ as the new

[1] A balanced binary search tree would also do just fine, but it is a more complicated data structure, so it is worthwhile to observe that the full power of binary search trees is not needed here.

ending time of the last class scheduled in room R_i. If $s_j < \ell_i$, then C_j starts before the last class in R_i ends and the algorithm appends C_j to S_{i+1}, to be scheduled in a subsequent classroom.

(b) Analogous to the correctness proof of the previous algorithm, observe that, if we place a class C_j into a classroom R_i, then $\ell_i \le s_j$ at the time we add C_j to class R_i. Thus, all classes scheduled in R_i so far end at or before time ℓ_i and thus do not overlap with class C_j. Thus, scheduling C_j in room R_i maintains the invariant that no two classes in R_i overlap. (A completely formal proof uses the same invariant as in Solution 1.) Since this argument applies to every classroom R_i, the classes scheduled in each room do not overlap, so the assignment of classes to classrooms is valid.

Now assume that we use k classrooms $R_1, \ldots, R_k$. Once again, let C_{j_k} be the first class we schedule in room R_k. Since $C_{j_k} \in S_k \subseteq S_{k-1} \subseteq \cdots \subseteq S_1$, that is, the class C_{j_k} is inspected by the scan of each of of the lists $S_1, \ldots, S_{k-1}$ and the algorithm decided not to schedule C_{j_k} in any of the rooms $R_1, \ldots, R_{k-1}$. Consider the scan of S_i. At the time we reach C_{j_k}, we would schedule C_{j_k} in room R_i if $\ell_i \le s_{j_k}$. Thus, $s_{j_k} < \ell_i$. Since ℓ_i is the ending time of some class C_{j_i} already scheduled in R_i, we have $e_{j_i} > s_{j_k}$ and, since this class was inspected before C_{j_k} during the scan of S_i, $s_{j_i} \le s_{j_k}$. Thus, I_{j_i} contains the time s_{j_k}. By applying this argument to each of the lists $S_1, \ldots, S_{k-1}$, we obtain $k-1$ classes $C_{j_1}, \ldots, C_{j_{k-1}}$ scheduled in rooms $R_1, \ldots, R_{k-1}$ whose intervals $I_{j_1}, \ldots, I_{j_{k-1}}$ all contain the time s_{j_k}. Since I_{j_k} also contains s_{j_k}, we once again have k intervals that contain s_{j_k}, that is, we need at least k classrooms to schedule all classes in S_1; the schedule computed by the algorithm is optimal.

(c) Sorting the classes by their starting times to produce S_1 takes $O(n \lg n)$ time using Merge Sort or any other optimal sorting algorithm. After that, each iteration of the algorithm scans S_i and splits it into the list of classes scheduled in room R_i and S_{i+1}. This takes $O(n)$ time per iteration. Since the algorithm terminates after k iterations, the total cost of the algorithm is $O(n \lg n + kn)$.

(d) The only disadvantage of this algorithm is that it is harder to reduce its running time to $O(n \lg n)$. Indeed, it seems impossible to make it run in $O(n \lg k)$ time excluding the initial sorting cost, which was possible for the algorithm in Solution 1. However, since the sorting cost is $O(n \lg n)$ anyway, it suffices to reduce the total running time of the k iterations of the algorithm to $O(n \lg n)$.

For $1 \le i \le k$, let n_i be the number of classes scheduled in room R_i. Then we show how to implement the ith iteration in $O(n_i \lg n)$ time. Since $\sum_{i=1}^{k} n_i = n$, this implies that the total cost of the k iterations of the algorithm is $O(\sum_{i=1}^{k} n_i \lg n) = O(n \lg n)$. To achieve this, each iteration expects the classes in S_i stored in a binary search tree with their starting times as keys. Instead of sorting the classes by their starting times to produce S_1, the preprocessing for the algorithm now consists of constructing the initial binary search tree representing S_1, which can be done using n insertions into the tree and thus still takes $O(n \lg n)$ time. The overall running time of the algorithm is thus $O(n \lg n)$, as for Solution 1.

Given a binary search tree T storing the classes in S_i, the first class we schedule in room R_i is the class with minimum starting time, which can be found in $O(\lg n)$ time by following the path to the leftmost leaf of T. The next class we schedule in room R_i is the first class $C_{j'}$ in S_i whose starting time is no less than the ending time e_j of the most recent class C_j scheduled in room R_i. We can locate $C_{j'}$ by searching T for the smallest starting time $s_{j'}$ that is no less than e_j. This is called a successor query in data structure parlance and can be performed on a binary search tree in $O(\lg n)$ time. Thus, each of the n_i classes scheduled in room R_i can be located in $O(\lg n)$ time using a successor query. The overall cost of these successor queries is $O(n_i \lg n)$, as required. In order to prepare T for the next iteration, we need to ensure that it stores the classes in S_{i+1} after the ith iteration. This is easily accomplished by deleting every class scheduled in R_i from T, which takes $O(\lg n)$ time per class, $O(n_i \lg n)$ time in total. Thus, the cost of the ith iteration is $O(n_i \lg n)$, as required.