CSC263 Summer 2022 Problem Set 3

Due August 5, 2022 23:59 ET on MarkUs

- Your problem sets are graded on both correctness and clarity of communication. Solutions that are correct but poorly written will not receive full marks.
- Each problem set may be completed individually or in a group of two students.
- Solutions must be typeset electronically and submitted to MarkUs as a pdf with the filename `ps3.pdf`. Handwritten submissions will receive a grade of zero.
- The work you submit must be that of your group. You may not use or copy from the work of other groups or from external sources like websites or textbooks. All the information you need to answer these questions has been covered in lecture and/or in the textbook.
- If you consult any external sources (i.e. other than the lecture notes, CLRS, or documents posted to the course website), then you must clearly indicate these additional sources on the first page of your submission.

Question 1 [6 marks]

A *partially* sorted array stores an array of integers `arr` and two nonnegative integers `size` and `sortedsize`. It maintains the following invariant: the first `sortedsize` integers of the array are sorted in nondecreasing order. There are no guarantees about the order of the last `size - sortedsize` integers of the partially sorted array. Initially, the array `arr` is empty and `size` = `sortedsize` = 0. For the sake of this question, assume that the array `arr` has an unbounded amount of allocated space, so we do not need to worry about resizing the array.

The partially sorted array supports one operation, `Insert(`x`)`, which appends the given integer x to `arr`. After sufficiently many integers are appended to `arr` since the last time `arr` was sorted, we use insertion sort to sort the entire array. The precise implementation of insertion sort is included below. Assume that `sortedsize` and `size` are both global variables.

```
1 def insertionsort(arr):
2     i = sortedsize
3     while i < size:
4         j = i
5         while j > 0 and arr[j-1] > arr[j]:
6             swap arr[j-1] and arr[j]
7             j -= 1
8         i += 1
```

More precisely, to perform `Insert(`x`)`, we first set `arr[size]` $= x$ and then increment `size`. Following this, if the new value of `size` $\geq 2\cdot$`sortedsize`, then we sort the items `arr[0], ..., arr[size-1]` using the insertion sort algorithm above and then set `sortedsize = size`. For the sake of this question, only count the number of times we append a new element to `arr` or execute line 6 of `insertionsort(arr)`.

(a) Prove a tight bound on the worst-case runtime complexity of an individual `Insert` operation on a partially sorted array that contains n integers.

(b) Consider a sequence of k insert operations performed on an initially empty partially sorted array. Use aggregate analysis to prove a tight bound on the amortized complexity per operation in this sequence.

Question 2 [8 marks]

A group of students $s_1, \ldots, s_n$ with $n \geq 4$ are trying to play a game of dodgeball, but they need to form teams beforehand. Each student would like to be at least as friendly with some other student on their team than they are with any student on the other team.

More formally, define the integer $f(s_i, s_j)$ as the *friendship score* of students s_i and s_j. A high value of $f(s_i, s_j)$ indicates that s_i and s_j are close friends, whereas a low value of $f(s_i, s_j)$ indicates that s_i and s_j are enemies. Friendship scores are symmetric, meaning that $f(s_i, s_j) = f(s_j, s_i)$ for all students s_i and s_j. We wish to partition the students into two nonempty teams $T_0, T_1 \subseteq \{s_1, \ldots, s_n\}$ such that the following property is satisfied:

for each team T_j and each student $s_i \in T_j$, there exists a student $s_{i'} \in T_j - \{s_i\}$ such that, for all students s_k on the other team T_{1-j}, $f(s_i, s_{i'}) \geq f(s_i, s_k)$. $(\star)$

(a) Give an example of a set of students for which property $(\star)$ *cannot* be satisfied. As part of your answer, you should give the friendship score of each pair of students. Explain why the property cannot be satisfied for your given example.

(b) Suppose we replaced property $(\star)$ with the following: For each team T_j and each student $s_i \in T_j$, either

(i) there exists a student $s_{i'} \in T_j - \{s_i\}$ such that, for all students s_k on the other team T_{1-j}, $f(s_i, s_{i'}) \geq f(s_i, s_k)$, or

(ii) T_j contains only a single student (i.e. the student s_i).

Using a data structure that we have learned about in this course, design an algorithm that outputs two disjoint teams T_0, T_1 such that $T_0 \cup T_1 = \{s_1, \ldots, s_n\}$ and the property above is satisfied. The worst-case runtime of your algorithm should be $O(n^2 \log n)$.

(c) Explain why your algorithm is correct.

(d) Briefly explain why your algorithm's worst-case runtime is $O(n^2 \log n)$.

Question 3 [6 marks]

A group of n people $R_1, \ldots, R_n$ are running a marathon around the city. To avoid crowding the running trail, the starting times of the runners are staggered. That is, the runner R_1 starts running before R_2, the runner R_2 starts running before R_3, and so on. The runner R_n starts running last.

Unfortunately, none of the marathon organizers were paying attention to the finish line during the race, so we do not know who completed the marathon first! We interview a set of bystanders who each provide an *observation* of one of the following forms, for some $i, j \in \{1, \ldots, n\}$ such that $i < j$:

- "I saw R_i finish the marathon before R_j started" (abbreviated as $R_i \to R_j$), or
- "I saw R_i and R_j running at the same time" (abbreviated as $R_i \sim R_j$).

Of course, the bystanders are prone to mistakes, so the observations may not make sense together. For example, suppose that $n = 3$ and we obtained the observations $R_1 \to R_2$ and $R_1 \sim R_3$ from some bystanders. These observations do not make sense together. Recall that R_2 starts the marathon before R_3. Hence, if R_1 actually finished the marathon before R_2 started as claimed, then R_1 must have also finished the marathon before R_3 started. Therefore, R_1 and R_3 could not have been running at the same time.

More formally, a set of observations $\mathcal{B}$ is *internally consistent* if, for all observations $R_i \to R_j \in \mathcal{B}$ and all $k \geq j$, $R_i \sim R_k \notin \mathcal{B}$.

(a) Describe a data structure that could be used to model this scenario.

(b) Design an algorithm that uses your data structure from part (a) to determine if a given set of m observations $\mathcal{B}$ is internally consistent. Your algorithm should have worst-case runtime $O(n + m)$. Explain why your algorithm is correct.

(c) Given an internally consistent set $\mathcal{B}$ of m observations, describe an algorithm that determines the set of all runners who could have finished the race first according to $\mathcal{B}$. (Of course, this marathon isn't entirely *fair*, since R_i starts the race before R_{i+1} — but this doesn't concern us for this question.) Your algorithm should have worst-case runtime $O(n + m)$. Explain why your algorithm is correct.