Solutions to Exam 1. April 25, 2018

CS180: Algorithms and Complexity
Spring 2018

Guidelines:

- The exam is closed book and closed notes. Do not open the exam until instructed to do so.
- Write your solutions clearly and when asked to do so, provide complete proofs. You may use results and algorithms from class without proofs or details **except for Problem 4** as long as you state what you are using.
- I recommend taking a quick look at all the questions first and then deciding what order to tackle to them in. Even if you don't solve the problems fully, attempts that show some understanding of the questions and relevant topics will get reasonable partial credit.
- You can use extra sheets for scratch work, but you can only use the white space (it should be more than enough) on the exam sheets for your final solutions.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly and any cheating reported with the score automatically becoming zero.
- Write clearly and legibly. All the best!

Problem	Points	Maximum
1		8
2		4
3		4
4		4
5		2
Total		22

Name	
UID	
Section	

1 Problem

The answers to the following should fit in the white space below the question.

1. For each pair (f, g) below indicate the relation between them in terms of O, Ω, Θ. For each missing entry, write-down Y (for YES) or N (for NO) to indicate whether the relation holds (no need to justify your answers here). For example, if $f = O(g)$ but not $\Omega(g)$, then you should enter Y in the first box and N in the other two boxes. Similarly, if $f = \Theta(g)$, then you should enter Y in all the boxes. [1 point]

f	g	O	Ω	Θ
n^2	$n^2 - 2n + 2$	Y	Y	Y
$\log_2 n$	$(\log_{100} n)^2$	Y	N	N

2. Is the following True or False: Consider a divide and conquer algorithm which solves a problem on an instance of length n by making six recursive calls to instances of length $\lfloor n/3 \rfloor$ each, and combines the answers in $O(n^2)$ time. Then, the time-complexity of the algorithm is $O(n^2)$. [1 point]

   Yes. You need to apply master theorem case (1).

3. State the principles behind the divide and conquer technique for designing algorithms. [1 point]

   Check slides.

4. What is the solution to the recurrence $T(1) = 1$, $T(n) = 2T(n/2) + 10n$? [1 point]

   O(n log n). This is the same recurrence we solved for merge sort.

5. Let a_0, a_1, b_0, b_1 be four integers that are k bits long. Write down Karatsuba's trick (that we used in class for fast integer multiplication) to compute the four products $a_1 \cdot b_1, a_1 \cdot b_0, a_0 \cdot b_1, a_0 \cdot b_0$ using only three multiplications and some additions and subtractions.

   The main idea was to compute a_1b_1, a_0b_0, $(a_1 + a_0)(b_1 + b_0)$ and use $a_1b_0 + a_0b_1 = (a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0$.

6. Write down some pros and cons of the adjacency-list and adjacency-matrix representations of graphs. [1 point]

   Check slides.

7. Write down the definition of a path in a graph $G = (V, E)$. [1 point]

   Check slides.

8. How can we efficiently check if a graph given in adjacency-list representation is connected? (You can refer to algorithms done in class without writing them out fully.) [1 point]

   You can run BFS from a vertex and check if all vertices are marked discovered when the algorithm finishes.

2 Problem

You are given k sorted arrays, each with n numbers in them. Give an algorithm for merging these arrays into a single sorted array of numbers that runs in time $O(nk \log k)$. You don't have to analyze the running time or prove correctness. [4 points]

(You can assume that the solution to the following recurrence is $O(nk \log k)$: $T(1) = O(1)$, $T(k) \leq 2T(k/2) + O(n \cdot k)$.)

Solution CombineSortedArrays$(A_1, \ldots, A_k)$

1. If $k = 1$, return the array.
2. Let $L =$ CombineSortedArrays$(A_1, \ldots, A_{\lfloor k/2 \rfloor})$.
3. Let $R =$ CombineSortedArrays$(A_{\lfloor k/2 \rfloor + 1}, \ldots, A_k)$.
4. Merge the two arrays L, R using the same routine that we used to merge two sorted arrays in describing MergeSort in class and return the answer.

3 Problem

Given an array $A[0, 1, \ldots, n-1]$, an element $A[i]$ is said to be a *pluraility element* if more than $\lfloor n/3 \rfloor$ of its elements equal elements of A. For example, the array $A = [1, 11, 2, 4, 2, 2, 1, 2, 4]$ has one plurality element 2; the array $A = [1, 1, 2, 4, 2, 2, 1, 2, 1]$ has two plurality elements $1, 2$; the array $A = [1, 11, 2, 1, 2, 1, 11, 2, 11]$ has no plurality elements.

Given an array as input, the task is to design an efficient algorithm to tell whether the array has any plurality elements and, if so, to find **all** the plurality elements. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as mp3 files, say; so in particular, you cannot sort the elements.) However you can answer questions of the form: "is $A[i] = A[j]$" in constant time.

Give an algorithm to solve the problem. For full-credit, your algorithm should be correct and run in time $O(n \log n)$ and you should bound the run-time of the algorithm. (You don't have to prove correctness.). [4 points]

Solution This can be solved in an way similar to Homework 2 Problem 2.
FindPlurarity(A)

1. If the array is size of less than 4, check by brute-force if there is a pluarity element.
2. Split the array into three pieces: A_1 containing the first $\lfloor n/3 \rfloor$ elements, A_2 containing the next $\lfloor n/3 \rfloor$ elements and A_3 containing the remaining.
3. Let $L_1 =$ FindPlurarity(A_1), $L_2 =$ FindPlurarity(A_2), $L_3 =$ FindPlurarity(A_3).
4. Set $L = NULL$.
5. If L_1 is not empty, for each element a of L_1 check if it is a plurarity element in A by counting the number of times a occurs in A. (This takes $O(n)$ time.) If yes, add a to L.
6. If L_2 is not empty, for each element a of L_2 check if it is a plurarity element in A by counting the number of times a occurs in A. (This takes $O(n)$ time.) If yes, add a to L.
7. If L_3 is not empty, for each element a of L_3 check if it is a plurarity element in A by counting the number of times a occurs in A. (This takes $O(n)$ time.) If yes, add a to L.
8. Return L.

The algorithm makes three recursive calls and the additional steps cost $O(n)$ in total. So the run-time satisfies the recurrence $T(n) \leq 3T(n/3) + O(n)$. Therefore, by master theorem case (2) applied with $a = 3, b = 3$, we get $T(n) = O(n \log n)$.

4 Problem

Given a set of points $P = \{p_1, \ldots, p_n\}$ in the plane, give an algorithm for finding a pair of points with the smallest possible L4-distance among the points where L4-distance between two points is defined by $d_4((x, y), (x', y')) = (|x - x'|^4 + |y - y'|^4)^{1/4}$.

For full-credit your algorithm should be correct and run in time $O(n \log n)$. You don't have to prove correctness or analyze the run-time of the algorithm. You should describe all the steps in the algorithm at a level of detail similar to what was done in class (however, you don't have to describe how to manipulate the sorted lists). [4 points]

Solution The same algorithm that we had in class works for this problem as well. Nothing needs to change in the analysis as well.

5 Problem

Let $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{\{1, 2\}, \{1, 6\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 5\}, \{3, 6\}, \{4, 6\}, \{5, 6\}\}$. Suppose that G was given to you in adjacency list representation where the elements in the adjacency list are ordered in increasing order. For example, the adjacency list of vertex 2 would be $[1, 5, 6]$. Run the BFS algorithm on G starting from the vertex 1. It suffices to show the step-by-step evolution of the lists $L[0], L[1], ...$ as we described in class. [2 points]

Solution

- $L[0] = [1]$.
- $L[1] = [2, 6]$.
- $L[2] = [5, 3, 4]$.
- $L[3] = \emptyset$.