CS161 Midterm
Date: May 3, 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to design an algorithm, please briefly describe your algorithm, analyze its running time, and prove its correctness. If you are asked to show your work, please include relevant calculations for deriving your answer, explaining steps as you go. If you are asked to prove a result, please write a complete proof at the level expected on your homework. When writing proofs, please strive for clarity and brevity (in that order).

You have **75 minutes** to complete the exam. The exam is closed-book, closed-note, closed-internet, etc. You may use 1 sheet (front and back) of notes as reference. The last page of the exam is a blank page (you may use it as scratch paper or write your answers).

By signing your name below, you acknowledge that you have abided by the Stanford Honor Code while taking this exam.

Name: _____

SUNetID: _____

Signature: _____

1: ____ /15

2: ____ /10

3: ____ /25

Total: ____ /50

1 (15 points) Short Answers on Trees

Recall that a BST is a binary search tree. Except for part (e), this problem concerns general BSTs, i.e. for the parts (a)-(c) that concern element insertions, you can assume the generic BST insertion algorithm from class that performs no rotations. Recall that the height of a BST is the number of edges of the longest root to leaf path. Let $n \geq 1$ be an arbitrary integer.

(Do NOT justify your answer; just give a short answer for each sub-question.)

(a) (3 pts) Give a permutation of the numbers $\{1, \ldots, n\}$, so that inserting them into a BST results in a BST of height $\Omega(n)$.

Solution: $\{1, \ldots, n\}$ or $\{n, \ldots, 1\}$.

Note that you must give a permutation explicitly, and for arbitrary n (not for some constant like $n = 10$).

(b) (3 pts) Give a permutation of the numbers $\{1, 2, 3, 4, 5, 6, 7\}$, so that inserting them into a BST results in a BST of height 2.

Solution: There are many possible answers (there are 80 correct permutations).

If ALL of the following four conditions are satisfied, the answer is correct.

- 4 must be inserted first; 2 or 6 should be inserted next (implied by the next two rules).
- 2 must be inserted before both 1 and 3; 6 must be inserted before both 5 and 7.

A few canonical examples: 4, $\{2, 6\}$, $\{1, 3, 5, 7\}$ 4, 2, $\{1, 3\}$, 6, $\{5, 7\}$ 4, 6, $\{5, 7\}$, 2, $\{1, 3\}$

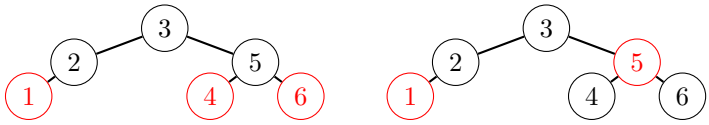
(c) (3 pts) Suppose we have a perfectly balanced binary tree of height h on n nodes ($n = 2^{h+1} - 1$). How many leaf nodes are there? Your answer can be in terms of n or h or both.

Solution: $\frac{n+1}{2}$ or 2^h. Either one is correct.

(d) (3 pts) Suppose we have a binary search tree with n nodes that may not be balanced ($n \geq 2$). What is the minimum number of leaf nodes it can have?

Solution: One.

(e) (3 pts) Consider the following Red-Black tree (RB-tree) with 6 (non-NIL) nodes whose keys are $\{1, 2, 3, 4, 5, 6\}$ (note that NILs are not drawn). Find two valid colorings of this RB-tree by explicitly stating which nodes should be colored red.



Solution: Color $\{1, 4, 6\}$ red or color $\{1, 5\}$ red.

Getting one correct valid coloring receives 1 point out of 3.

(f) (optional, extra credit of 3 pts) Concisely explain why there are only two valid colorings of the RB-tree shown above. (We may deduct points for excessively long answers.)

Solution: In every valid coloring of the given RB-Tree:

- 2 must be colored black. Otherwise, 1 must be black as 2 is red, resulting in different number of black nodes in 3-2-NIL and 3-2-1-NIL. Thus every root-to-NIL path must contain 3 black nodes.
- If 5 is colored red, 4 and 6 must be colored black by the rules (which we found in previous part). If 5 is colored black, then 4 and 6 must be colored red in order to have three black nodes in paths containing them (which we found in previous part).

This covers all possible cases.

2 (10 points) Recurrences

Consider the pseudocode given below for an algorithm called "Magic", and its helper function "Compute" whose code is not shown.

When Compute(X, Y) is called on two lists X and Y of integers, it computes and returns an integer in time $O(|X||Y|\log(|X||Y|))$.

Magic takes a list of integers L.

Algorithm 1 Magic(L)

```
1: n ← L.length                    ▷ Assume n is a power of 4. This line runs in time O(1).
2: if n = 1 then
3:     return 1
4: end if
5: x ← Select(L, n/4)              ▷ The Select algorithm from class that uses median of medians.
6: y ← Select(L, 3n/4)             ▷ The Select algorithm from class that uses median of medians.
7: X ← {i ∈ L : i ≤ x}             ▷ X is a list of integers in arbitrary order.
8: Y ← {i ∈ L : i > y}             ▷ Y is a list of integers in arbitrary order.
9: c ← Magic(X)
10: d ← Magic(Y)
11: e ← Compute(X, Y)
12: return c + d + e
```

(a) (4 pts) Let $T(n)$ be the runtime for calling Magic on a list of n distinct integers, which can be written as the following recurrence relation.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Fill in each of these constants below as they correspond to the recurrence relation for the magic function above. You may express $f(n)$ in terms of its big-O bound. (Do not justify your answer for this question.)

Solution:

(i) (1 point) $a = 2$

(ii) (1 point) $b = 4$

(iii) (2 points) $f(n) = O(n^2 \log n)$

Note that Lines 5-8 run in time $O(n)$, Lines 9-10 run in time $T(n/4)$ each, and Line 11 runs in time $O((n/4)^2 \log((n/4)^2)) = O(n^2 \log n)$. Thus $f(n) = O(n^2 \log n)$.

(No partial credit per item above.)

(b) (6 pts) Solve the recurrence and give the big-O running time of the Magic function. Show your work.

Solution: $T(n) = 2T(\frac{n}{4}) + O(n^2 \log n)$. We can apply the master theorem (case 3) with $\epsilon = 0.1$ and $c = 1/2$, for instance, and conclude that $T(n) = \Theta(n^2 \log n)$ (any $\epsilon \in (0, 1.5]$ and $c \in [1/8, 1)$ works). Note that $f(n) = \Omega(n^{0.5+\epsilon})$ and $2f(n/4) \leq c \cdot f(n)$ for our choice of ϵ and c. Therefore the Magic function runs in time $O(n^2 \log n)$.

Rubric:

- If you apply the master method correctly but do not specify a correct value (or range) for ϵ or c, you get 2 points off for each such error.
- If you solve it using the recursion tree or substitution method, and get a loose bound (such as $O(n^2 \log^2 n)$), you get a partial credit.
- If your answer to Part (a) is wrong, but answer to Part (b) is correct based on your answer to Part (a), you get up to 3 points as partial credit.

3 (25 points) Searching through a sorted matrix

Let A be an $m \times n$ matrix of distinct integers such that every row is sorted from left to right and every column is sorted from bottom to top. More precisely, for every $i \in \{1, \ldots, m\}$ and every $j \in \{1, \ldots, n\}$, $A[i, j] < A[i, j+1]$ (when $j < n$) and $A[i, j] > A[i+1, j]$ (when $i < m$). We say that A is *sorted*.

For example, the following matrix M is sorted, and $M[1, 3] = 9$:

$$M = \begin{bmatrix} 7 & 8 & 9 \\ 3 & 4 & 6 \\ 1 & 2 & 5 \end{bmatrix}.$$

In this question we will design algorithms that given a sorted $m \times n$ matrix A and an integer x determine whether x appears in A.

(a) (5 pts) Suppose that A is a $1 \times n$ sorted matrix (i.e. a row vector). Give an $O(\log n)$ time algorithm that determines whether x appears in A. Concisely argue why your algorithm is correct and show its running time is $O(\log n)$.

Solution:

Algorithm (2 points):

We can perform binary search on A because it is sorted. (You must describe what binary search is, to receive a full mark on this part.)

If the given array A contains no elements, return FALSE (this is to deal with degenerate cases). Pick the middle element of A (call it $A[c]$) and compare it with x. If they are equal return TRUE. If $A[c] < x$ then we recursively search for x in $A[c+1 : n]$. If $A[c] > x$ then we recursively search for x in $A[1 : c-1]$.

(If no description other than "binary search" is given, then one point deduction.)

Runtime (1 point):

Runtime of this algorithm is $O(\log n)$ because the recursion tree has height at most $O(\log n)$ and each node performs $O(1)$ works (by looking up an entry of A).

Correctness (2 points):

Correctness can be shown by induction on n. When $n = 1$, it is trivial that the algorithm correctly returns TRUE or FALSE depending on whether $A[1] = x$ or not. Suppose the recursive algorithm is correct when the given array has size k or less. Given an array of size $k + 1$, if the middle element is equal to x, then the algorithm correctly returns TRUE. Otherwise, if $A[c] < x$, then it is clear that x cannot exist in $A[1 : c]$ because all of them are less than x; and therefore x exists in A if and only if x exists in $A[c+1 : n]$. By inductive hypothesis, the recursive method will correctly return TRUE/FALSE depending on whether x exists in $A[c+1 : n]$ or not as the sub-array has a smaller size. Analogously we can show correctness for the case $A[c] > x$.

(Note that there are other ways to prove correctness. If your proof does not cover some of the cases, such as when $x \notin A$, deductions may apply.)

(b) (8 pts) Suppose that A is an $n \times n$ sorted matrix. Give an $O(n \log n)$ time algorithm that determines whether x appears in A. Concisely argue why your algorithm is correct and show its running time is $O(n \log n)$.

Solution: Algorithm (2 points):

We can perform binary search on each row of A (using the algorithm from Part (a)).

Runtime (2 points):

Each iteration runs in time $O(\log n)$, and there are n rows so the overall running time is $O(n \log n)$.

Correctness (4 points):

(2 points) If x appears in A, it must exist in exactly one row (call it r), and thus our algorithm will find it when it performs binary search on the rth row of A.

(2 points) If x does not appear in A, all n iterations of binary searches will not find x in A. This algorithm is thus correct.

(Note that there are other ways to prove correctness. If your proof does not cover some of the cases, such as when $x \notin A$, deductions may apply. You may use your result from Part (a), but because you are designing a new algorithm, you must prove correctness of your new algorithm (which is dependent on correctness of the algorithm from Part (a)).)

(c) (12 pts) Now we will try to obtain an $O(n)$ time algorithm for the problem. In the following, the recursive algorithm Exists takes in A, two integers r and c, and x, and searches for x in the submatrix of A consisting of the rows from 1 to r and the columns from 1 to c. Exists(A, r, c, x) returns TRUE if x is in the submatrix and FALSE otherwise.

Exists(A, r, c, x):
If $r < 1$ or $c < 1$: **return** FALSE
If $A\left[\boxed{}, \boxed{}\right] = x$: **return** $\boxed{}$
Else if $A\left[\boxed{}, \boxed{}\right] > x$: **return** $\boxed{}$
Else: **return** $\boxed{}$

Fill in the boxes above. Some of them will be integers related to r and c, some of them will be TRUE or FALSE, and some will be recursive calls to Exists. Analyze the running time of Exists(A, n, n, x) when A is $n \times n$, showing that it is $O(n)$. Concisely argue why your algorithm is correct.

Solution:

(3 points) Algorithm:

Exists(A, r, c, x):
If $r < 1$ or $c < 1$: **return** FALSE
If $A[r, c] = x$: **return** TRUE
Else if $A[r, c] > x$: **return** Exists$(A, r, c-1, x)$
Else: **return** Exists$(A, r-1, c, x)$

No need to further describe the algorithm after filling in the blanks above. If you get all boxes right for a line, you get 1 point (for a total of 3 points). Wrong algorithms receive 0 points on this part. You are not allowed to modify the code we provided. If your algorithm is incorrect, maximum points you get from the runtime and correctness are capped (depending on how wrong your algorithm is).

(3 points) Runtime:

During each call of Exists, there is $O(1)$ operations performed besides at most one recursive call. Let $T(r, c)$ be the running time of Exists if it is called on (A, r, c, x). Then $T(r, c) = O(1)$ when $r < 1$ or $c < 1$ and $T(r, c) = \max(T(r-1, c), T(r, c-1)) + O(1)$. Using recursion tree or substitution method, one can show that $T(r, c) = O(r + c)$; in particular, when $r = c = n$ we get $O(n)$.

(6 points) Correctness:

We shall prove that the algorithm returns TRUE if and only if x exists in A.

(1 point) If the algorithm returns TRUE, then we know that x exists in A as we have evidence ($A[r, c] = x$); this also implies that if x does not exist in A then our algorithm does not return TRUE. This is 'the base case' which you should mention (or lose a point).

(5 points) Now suppose that x exists in A (and thus assume $r, c \geq 1$) and we wish to prove that the algorithm returns TRUE. We prove this by induction on $r + c$. If $r + c < 1$, this correponds to some degenerate case where A contains no elements; the algorithm correctly returns FALSE. The base case is when $r = c = 1$, and correctness of the algorithm is trivial as it returns TRUE/FALSE correctly depending on whether $A[1, 1] = x$ or not (in fact by our assumption we know $A[1, 1] = x$ in this case). Now suppose that the algorithm returns TRUE when $r + c \leq k$ and x exists in $A[1 : r, 1 : c]$. We now prove for the case $r + c = k + 1$. If $A[r, c] = x$, then the algorithm correctly returns TRUE. If $A[r, c] > x$, then it is clear that $A[i, c] > x$ for all $i \leq r$ (because A is sorted), and therefore x must exist in the sub-array $A[1 : r, 1 : c - 1]$. In particular we call Exists$(A, r, c-1, x)$ and thus by inductive hypothesis the algorithm will correctly return TRUE. Analogously if $A[r, c] < x$ (the last Else statement), then it

is clear that $A[r, j] < x$ for all $j \leq c$ (because A is sorted); by inductive hypothesis the algorithm will correctly return TRUE.

Note that answers that justify why recursing on the submatrix with one less row/column is correct (assuming x is in A) can receive a full credit on this part, even if the answers do not explicitly mention induction as their method of proving correctness. Yet your arguments must show that discarding a column/row does not result in a wrong answer (because x cannot exist in the column/row being discraded).