David Sun
CMPS 102
Homework 1

Problem 1: In a binary tree all nodes are either internal or they are leaves. In our definition, internal nodes always have two children and leaves have zero children. Prove that for such trees, the number of leaves is always one more than the number of internal nodes.

Proof:
If T is a tree containing n internal nodes, then T contains $n + 1$ leaves

I. Base Case
If T has just one internal node, then T has two leaves, thus the number of leaves in T is one greater than the number of internal nodes in T.

II. Induction Step
Let $n \geq 1$. Assume that all trees containing n internal nodes contain $n + 1$ leaves. Generate another internal node on T by taking an arbitrary leaf node in T and attaching two leaf nodes to it. By generating another internal node, T now contains $n + 1$ internal nodes. In the process of generating a new internal node, one leaf node is eliminated from T and two leaf nodes are attached to T, so T now contains $((n + 1) - 1) + 2$ leaf nodes. By the induction hypothesis, a tree containing $n + 1$ internal nodes contains $(n + 1) + 1$ leaves.
Thus, $((n + 1) - 1) + 2 = (n + 1) + 1$ as required.

Problem 2: For $n \geq 0$ consider $2^n \times 2^n$ matrices of 1s and 0s in which all elements are 1, except one which is 0 (The 0 is at an arbitrary position). Operation: At each step, we can replace three 1s forming an "L" with three 0s (The L's can have an arbitrary orientation).

Proof:
If M is a $2^n \times 2^n$ matrix consisting of all 1s and one 0, there exists a sequence of "L" operations such that replacing three 1s in an "L" pattern in M will give us the 0 matrix.

I. Base Case
$n = 0$. A $2^0 \times 2^0$ matrix is a 1×1 matrix. The only entry can be a 0, and thus applying the "L" operation 0 times is sufficient to obtain the 0 matrix.

II. Induction Step
Let $n \geq 0$. Let M be a $2^n \times 2^n$ matrix where M contains exactly one 0 and the rest 1s. Assume there exists a sequence of "L" operations to obtain a 0 matrix for M. Let M' be a $2^{n+1} \times 2^{n+1}$ matrix containing exactly one 0 entry with 1s for the rest of the entries.
(continued on the next page.)

Since M' is a $2^{n+1} \times 2^{n+1}$ matrix, M' can be evenly divided into four quadrants with each quadrant containing $2^n \times 2^n$ matrix entries. This means that the initial 0 entry in M' must lie within one of the four quadrants.

$$M' = \begin{bmatrix} 0 & \dots & 1 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 1 & \dots & 1 \\ 1 & \dots & 1 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 1 & \dots & 1 \end{bmatrix}$$

In the center of M' all of the four quadrants are adjacent to each other. In the center of M' an "L" operation can be applied to eliminate three 1s in the three quadrants not containing the initial 0.

$$\begin{bmatrix} 0 & \dots & 1 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 1 & \dots & 1 \\ 1 & \dots & 1 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 1 & \dots & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & \dots & 1 & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 0 & \dots & 1 \\ 1 & \dots & 0 & 0 & \dots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & 1 & \dots & 1 \end{bmatrix}$$

By applying the "L" operation in the center of M', we obtain four quadrants each containing $2^n \times 2^n$ entries with one zero entry per quadrant. By the induction hypothesis, it is possible to transform each of the four $2^n \times 2^n$ quadrants to all 0s by a sequence of "L" operations, thus a $2^{n+1} \times 2^{n+1}$ containing one 0 entry and 1s for the other entries can be transformed into the 0 matrix.

Problem 3: Suppose you are given an array A of n distinct integers with the following property: There exists a unique index p such that the values of $A[1 \dots p]$ are decreasing and the values of $A[p \dots n]$ are increasing.

Algorithm: Our divide and conquer algorithm will be called findIndex(A, low, high). Suppose A has length N, then we will initially call the function in the manner findIndex(A, 1, N). If A has length 1, we return 1 as the index. Otherwise, we compute the midpoint of the low index and the high index such that $mid = \left\lfloor \frac{low + high}{2} \right\rfloor$.
If $mid > 1$ and $mid < N$, check if $A[mid - 1] > A[mid] < A[mid + 1]$. If such is the case, return mid since we have found the correct index. If that condition does not hold, check if A[mid - 1] > A[mid] > A[mid + 1]. If such is the case, then p lies within $A[mid + 1 \dots N]$ So we will recur to the right of the array by calling **findIndex**$(A, mid + 1, high)$. However, if instead A[mid - 1] < A[mid] < A[mid + 1] then p lies in the left half of A, so we will recur to the left of the array by calling **findIndex**$(A, low, mid - 1)$.

Edge cases in which $mid = 1$ or $mid = N$ must also be considered.
In the case that $mid = 1$, check if $A[mid] < A[mid + 1]$, if such is the case, then $p = 1$ so return mid. Otherwise recur to the right by calling **findIndex**$(A, mid + 1, high)$
In the case that $mid = N$, check if $A[mid] < A[mid - 1]$, if such is the case, then $p = N$ so return mid. Otherwise, recur to the left by calling **findIndex**$(A, low, mid - 1)$

Proof of Correctness: The invariant that all elements from $A[1 \dots p]$ are decreasing and all elements from $A[p \dots N]$ are increasing suggests that p must either lie within $A[1 \dots \lfloor \frac{N}{2} \rfloor]$ or $A[\lfloor \frac{N}{2} \rfloor + 1 \dots N]$.
If the middle element satisfies the condition for p, then we're done. If the middle element is within a decreasing pattern, p lies within the right subarray. If the middle element is within an increasing pattern *(5, 4, 3,...)*, p lies within the left subarray In any case, our algorithm converges to a p index by checking the conditions of the middle element and deciding whether to return the index of the middle element or to recur on the the left subarray or the right subarray.

Proof of Runtime: By recurring either to left subarray or to the right subarray, the algorithm essentially discards half the input size each time. There are also a few array element comparisons, but those are done in constant time. The recurrence can be written as $T(n) = T(n/2) + O(1)$. Using the master theorem, we can show that $T(n) = O(\log n)$.

Problem 4: Consider a complete binary tree T with n nodes ($n = 2^d - 1$ for some d). Each node v of T is labeled with a distinct real number x_v. Define a node v of T to be a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. You are given such a complete binary tree T, but the labeling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v. Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T.

Algorithm: Start at the root of T. We'll call that node v and its label x_v. If the left child of v is not NIL and x_v is greater than the label of $v's$ left child, recur on the left subtree with the left child of v as the root. Else if the right child of v is not NIL and x_v is greater than the label of $v's$ right child, recur on the right subtree with the right child of v as the root. Else If both of these comparisons do not hold, then v must be a local minimum and therefore we return x_v.

Proof of Correctness: By definition, a local minimum is a node in which its label is less than that of the node's connected to it by an edge. Our algorithm searches a node's left child and right child for a lower valued label. If such a node exists, then the current node is not a local minimum and we recur on the first node we find with a lower valued label.
However, if a node is a leaf, or its two children have label values greater than itself, then the node must be a minimum. In this case, we do not need to check the label of the node's parent because this node must have had a label value less than that of its parent's in order for the algorithm to recur on it.

Proof of Runtime: If N is the input size, recurring on the left or the right discards $\lceil \frac{N}{2} \rceil$ elements, meaning that $\lfloor \frac{N}{2} \rfloor$ elements remain. There are also steps in the algorithm to compare the labels of the nodes, which take constant $O(1)$ time. The recurrence can be summarized as $T(n) = T(\lfloor \frac{N}{2} \rfloor) + O(1)$. Using the master theorem, we can show that $T(n) = O(\log n)$.

Problem 5: Show that for any non-negative integers a, b, $(n+a)^b = \Theta(n^b)$.
We need to show that $(n+a)^b$ is both $O(n^b)$ as well as $\Omega(n^b)$
Proof
Case 1: $(n+a)^b = O(n^b)$.
If $f(n) = O(g(n))$, then $\exists c > 0\ \exists n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$. Then ...

$$\exists c > 0\ \exists n_0 > 0 \text{ such that } 0 \leq (n+a)^b \leq c \cdot n^b.$$

$$(n+a)^b \leq c \cdot n^b$$

$$n + a \leq c^{1/b} \cdot n$$

$$a \leq c^{1/b} \cdot n - n$$

$$a \leq n(c^{1/b} - 1)$$

$$\text{Let } c^{1/b} = 2, \text{ therefore } c = 2^b$$

$$a \leq n(2-1)$$

$$a \leq n$$

Thus if we choose $c = 2^b$ and $n_0 = a$, we can show that $0 \leq (n+a)^b \leq c \cdot n^b$
thus $(n+a)^b = O(n^b)$.

Case 2: $(n+a)^b = \Omega(n^b)$.
If $f(n) = \Omega(g(n))$, then $\exists c > 0\ \exists n_0 > 0$ such that $0 \leq c \cdot g(n) \leq f(n)$. Then ...

$$\exists c > 0\ \exists n_0 > 0 \text{ such that } 0 \leq c \cdot n^b \leq (n+a)^b.$$

$$c \cdot n^b \leq (n+a)^b$$

$$c^{1/b} \cdot n \leq n + a$$

$$-a \leq n - c^{1/b} \cdot n$$

$$-a \leq n(1 - c^{1/b})$$

$$\text{Let } c^{1/b} = 1, \text{ therefore } c = 1, \text{ thus}$$

$$-a \leq 0$$

In this case, $c = 1$, and we can pick any arbitrary $n_0 > 0$, so choose $n_0 = a$. Using these values for c and n_0, we can show that $0 \leq c \cdot n^b \leq (n+a)^b$, and therefore $(n+a)^b = \Omega(n^b)$. Since we have shown that $(n+a)^b$ is $O(n^b)$ and $\Omega(n^b)$, $(n+a)^b = \Theta(n^b)$