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.

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

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: