1. Suppose we want to write an efficient function \texttt{SHUFFLE}(n) that returns a permutation of the set \{1, 2, \ldots, n\} chosen uniformly at random.

(a) Prove that the following algorithm is not correct. [\textit{Hint: Consider the case } n = 3.]

\[
\texttt{SHUFFLE}(n): \begin{align*}
\text{for } i &\leftarrow 1 \text{ to } n \\
\pi[i] &\leftarrow i \\
\text{for } i &\leftarrow 1 \text{ to } n \\
\text{swap } \pi[i] &\leftarrow \pi[\text{RANDOM}(n)] \\
\text{return } \pi[1..n]
\end{align*}
\]

(b) Consider the following implementation of \texttt{SHUFFLE}.

\[
\texttt{SHUFFLE}(n): \begin{align*}
\text{for } i &\leftarrow 1 \text{ to } n \\
\pi[i] &\leftarrow \text{NULL} \\
\text{for } i &\leftarrow 1 \text{ to } n \\
j &\leftarrow \text{RANDOM}(n) \\
\text{while } (\pi[j] \neq \text{NULL}) \\
j &\leftarrow \text{RANDOM}(n) \\
\pi[j] &\leftarrow i \\
\text{return } \pi[1..n]
\end{align*}
\]

Prove that this algorithm is correct. What is its expected running time?

(c) Describe and analyze an implementation of \texttt{SHUFFLE} that runs in \(O(n)\) time. (An algorithm that runs in \(O(n)\) \textit{expected} time is fine, but \(O(n)\) \textit{worst-case} time is possible.)

2. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with \(4^n\) leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after \(2n\) moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

\[
\text{You can decide whether it's worth playing or not as follows. Imagine that the tree is a Boolean circuit whose inputs are specified at the leaves: white and black represent } \texttt{TRUE} \text{ and } \texttt{FALSE} \text{ inputs, respectively. Each internal node in the tree is a } \texttt{NAND} \text{ gate that gets its input from its children and passes its output to its parent. (Recall that a } \texttt{NAND} \text{ gate outputs } \texttt{FALSE} \text{ if and only if both its inputs are } \texttt{TRUE}. \text{ If the output at the top of the tree is } \texttt{TRUE}, \text{ then you can win and live forever! If the output at the top of the tree is } \texttt{FALSE}, \text{ you should challenge Death to a game of Twister instead. Or maybe Battleship.)}
\]
(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. 
[Hint: This is easy!]

(b) Unfortunately, Death won’t give you enough time to look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $O(3^n)$ expected time. [Hint: Consider the case $n = 1$.]

*(c) [Extra credit] Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. [Hint: You may not need to change your algorithm from part (b) at all!]

3. A meldable priority queue stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

• **MAKEQ**U**EUE**: Return a new priority queue containing the empty set.
• **FINDMIN**($Q$): Return the smallest element of $Q$ (if any).
• **DELETEMIN**($Q$): Remove the smallest element in $Q$ (if any).
• **INSERT**($Q$, $x$): Insert element $x$ into $Q$, if it is not already there.
• **DECREASEKEY**($Q$, $x$, $y$): Replace an element $x \in Q$ with a smaller key $y$. (If $y > x$, the operation fails.) The input is a pointer directly to the node in $Q$ containing $x$.
• **DELETE**($Q$, $x$): Delete the element $x \in Q$. The input is a pointer directly to the node in $Q$ containing $x$.
• **MELD**($Q_1$, $Q_2$): Return a new priority queue containing all the elements of $Q_1$ and $Q_2$; this operation destroys $Q_1$ and $Q_2$.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. **MELD** can be implemented using the following randomized algorithm:

```
MELD($Q_1$, $Q_2$):
    if $Q_1$ is empty return $Q_2$
    if $Q_2$ is empty return $Q_1$
    if key($Q_1$) > key($Q_2$)
        swap $Q_1 \leftrightarrow Q_2$
        with probability 1/2
        left($Q_1$) $\leftarrow$ MELD(left($Q_1$), $Q_2$)
    else
        right($Q_1$) $\leftarrow$ MELD(right($Q_1$), $Q_2$)
    return $Q_1$
```

(a) Prove that for any heap-ordered binary trees $Q_1$ and $Q_2$ (not just those constructed by the operations listed above), the expected running time of **MELD**($Q_1$, $Q_2$) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an $n$-node binary tree if each left/right choice is made with equal probability?]

(b) Show that each of the other meldable priority queue operations can be implemented with at most one call to **MELD** and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)