Problem 1. (30 points)
Using the Master method, give an asymptotic tight bound for \( T(n) \) defined by the following recurrence relation

\[
T(n) = \begin{cases} 
2 & n = 2 \\
4T\left(\sqrt{n}\right) + \log^2 n & n > 2 
\end{cases}
\]

Answer: Let \( n = 2^k \) (that is, \( \log_2 n = k \)). Then

\[
T(n) = 4T\left(n^{1/2}\right) + \log^2 n \\
T(2^k) = 4T\left(2^{k/2}\right) + k^2
\]

Let \( S(k) = T(2^k) \). We have

\[
S(k) = \begin{cases} 
2 & k = 1 \\
4S(k/2) + k^2 & k > 1 
\end{cases}
\]

We can apply case 2 of the Master Theorem. In fact,

\[
k^2 \in \Theta\left(k^{\log_2 4 \log^t k}\right)
\]

for \( t = 0 \). Therefore \( S(k) \in \Theta(k^2 \log k) \).

Hence, \( T(2^k) \in \Theta(k^2 \log k) \), which implies that \( T(n) \in \Theta\left(\log^2 n \log \log n\right) \).

Problem 2. (30 points)
Consider the following multi-search problem. Let \( A[1, \ldots, n] \) be a fixed array of distinct integers. Given an array \( X[1, \ldots, k] \), we want to find the position (if any) of each integer \( X[i] \) in the array \( A \). In other words, we want to compute an array \( I[1, \ldots, k] \) where for each \( i \), either \( I[i] = 0 \) (so zero means ‘none’) or \( A[I[i]] = X[i] \). Determine the complexity of this problem, as a function of \( n \) and \( k \), in the binary decision tree model.

Answer: For each element \( X[i] \), we need to report its position in \( A \) or report that it does not exist. So the number of possible outputs is \( n + 1 \) for each element \( X[i] \). For \( k \) queries \( X[1, \ldots, k] \), the number of all possible output configurations is \((n + 1)^k\). The height of the decision tree (assuming a comparison-based computation model) is therefore \( \Omega(k \log_2 n) \), which is the lower bound on this problem. Note that we can achieve this lower bound when the array \( A \) is sorted by running \( k \) binary searches, for a total of \( O(k \log_2 n) \).

Problem 3. (40 points)
Show how to implement a queue using two stacks \( S_1 \) and \( S_2 \) so that the amortized cost of each operation on the queue is \( O(1) \). (1) Give the pseudocode for the \texttt{ENQUEUE}(x) operation
and the \texttt{Dequeue()} operation (you can omit error checking for underflow and overflow of the stacks). (2) Use the accounting method to charge each operation a constant amortized cost and prove that a sequence of \( n \) \texttt{Enqueue} and \texttt{Dequeue} cost \( O(n) \) time overall.

\textbf{Answer:} We can implement a queue in the following way.

\texttt{Enqueue}\((x)\)
1. \texttt{Push}\((S_1, x)\)

\texttt{Dequeue}()
1. if \( S_2 \neq \emptyset \)
2. then \texttt{return} \texttt{Pop}(S_2)
3. else
4. while \( S_1 \neq \emptyset \) do
5. \texttt{Push}(S_2, \texttt{Pop}(S_1))
6. \texttt{return} \texttt{Pop}(S_2)

Note that each element is first pushed in \( S_1 \), then is moved to \( S_2 \), and eventually gets popped. Since each \texttt{Pop} and \texttt{Push} in the stacks costs constant time, we count the overall number of \texttt{Pop} and \texttt{Push}.

The following is our charging scheme. We charge $4 for \texttt{Enqueue} and $0 for \texttt{Dequeue}. Out of $4, $1 pays for the \texttt{Push} in \texttt{Enqueue}(x) and $3 are left as credit. When \( x \) is popped from \( S_1 \) and pushed in \( S_2 \) we remove $2 from the credit. When \( x \) is finally popped from \( S_2 \) we use the remaining $1 to pay for the \texttt{Pop}.

A series of \( n \) \texttt{Enqueue} and \texttt{Dequeue} operations would take $4n in the worst case (\( O(n) \) overall) and therefore the amortized cost of each operation is \( O(1) \).