Problem 1. (30 points)

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

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

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)

We have seen in class that the procedure MERGE in the Mergesort algorithm takes two sorted arrays of size \( n \) and produces one fully sorted array of size \( 2n \) in \( O(n) \) time. Use the decision tree method to prove a \( 2n - o(n) \) lower bound\(^1\) for the problem of merging two sorted arrays, each containing \( n \) items.

Answer:

Note that when we pick \( n \) elements for first sorted list, this determined a single, unique second list for the other \( n \) elements. Therefore the number of possible ways to divide \( 2n \) numbers into two sorted lists is the same as the number of ways to select \( n \) elements of out \( 2n \), which is \( \binom{2n}{n} \). We have

\[
\binom{2n}{n} = \frac{(2n)!}{n!(2n-n)!} = \frac{(2n)!}{(n!)^2} = \ldots = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n))
\]

\(^1\)The little-o notation is used here to denote an upper bound that is not asymptotically tight. Formally, we say that \( f(n) \in o(g(n)) \) if for any positive constant \( c \) we can find a constant \( n_0 \) such that \( o \leq f(n) < cg(n) \) for all \( n \geq n_0 \).
by using Stirling’s approximation for the numerator and the denominator. The height of the decision tree is therefore

\[
\log_2 \left( \frac{2^{2n}}{\sqrt{\pi n}} \left(1 + O(1/n)\right) \right) = \log_2 2^{2n} - \log_2 \sqrt{\pi n} + \log_2 (1 + O(1/n)) = 2n - o(n)
\]

**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.

**Answer:** We can implement a queue in the following way.

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

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