Problem 1. [Analysis][Lower bounds] (25 points)
Consider the following multi-search problem. Let $A[1, \ldots, n]$ be an array of distinct integers. Given an array $X[1, \ldots, k]$, we want to find the position 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 $I[i] = 0$ if $X[i]$ does not appear in $A$ or $I[i] > 0$ if $A[I[i]] = X[i]$. For instance, if $A = [2, 4, 5, 1]$ and $X = [4, 6, 1]$, the algorithm should produce $I = [2, 0, 4]$, because 4 is at position 2 in $A$, 6 does not exist in $A$, and 1 is at position 4 in $A$. Provide a lower bound on the multi-search problem, as a function of $n$ and $k$, assuming the decision-tree (comparison-based) model of computation.

**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 binary 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 2. [Analysis][Amortized Analysis] (25 points)
An ordered stack is a data structure that stores integers and supports the following operations.

- **ORDEREDPUSH($x$)** first removes integers from the top of the stack as long as they are smaller than $x$ (i.e., it stops popping elements as soon as the integer on top of the stack is greater than or equal to $x$), then pushes $x$ onto the stack

- **POP()** removes and returns the item on the top of the stack (or returns **null** if the stack is empty).

Suppose we implement an ordered stack with a simple linked list, using the obvious **ORDEREDPUSH** and **POP** algorithms. Use the accounting method to design a charging scheme to prove that an arbitrary sequence of $n$ these two operations starting from an empty stack will take $O(n)$ overall. Make sure you explain the credit invariant.

**Answer:** Assign $0$ to **POP($x$)**. Assign $2$ to **ORDEREDPUSH($x$)**: one pays for adding $x$ to the beginning of the sequence, and one is stored on the item. Credit invariant: at any time, each item on the data structure will have $\text{credit}$ of credit. When **ORDEREDPUSH($x$)** is executed, removing all items smaller than $x$ will be paid with the credit. When **POP** is executed, the cost of deleting $x$ will be paid by the credit stored on item $x$.

Problem 3. [Divide and Conquer][Design] (25 points)
The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows.

$$
H_k = \begin{cases} 
[1] & k = 0 \\
\begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} & k > 0
\end{cases}
$$

Note that $H_k$ is a $2^k \times 2^k$ matrix. Design a $O(n \log n)$ divide-and-conquer algorithm that given a column vector $v$ of length $n = 2^k$, computes the matrix-vector product $H_kv$. Analyze the time complexity of your algorithm.

**Answer:** We have to compute $H_kv$. Let’s split $v = [v_1v_2]$, where $v_1$ are the first $n/2 = 2^{k-1}$ components and $v_2$ the other $n/2 = 2^{k-1}$. We have

$$
H_kv = \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 & H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix}
$$
Therefore, in order to solve a problem of size $n$ we need to solve two problems of size $n/2$, namely $H_{k-1}v_1$ and $H_{k-1}v_2$. The recurrence relation for the time complexity is $T(n) = 2T(n/2) + O(n)$ because summing two vectors of size $n/2$ takes $O(n)$ time. The solution of the recurrence relation is $O(n \log n)$ (by Master Theorem case 2).

**Problem 4.** [Divide and Conquer][Knowledge] (25 points)

Let $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b_1, \ldots, b_{n-1}]$ two $n$-bits binary vectors. Explain how to compute the product $a \times b$ in $O(n \log n)$-time as we discussed in class. Make sure you explain the time complexity of each step of the algorithm.

**Answer:** Either

1. Form two $(n - 1)$-degree polynomials $A(x)$ and $B(x)$ with $a$ and $b$ as coefficients in $O(n)$
2. Multiply $A(x)B(x) = C(x)$ using the FFT in $O(n \log n)$
3. Return the $2n$ coefficients of $C(x)$

or

1. Compute the $A = \text{DFT}(a)$ using FFT in $O(n \log n)$
2. Compute the $B = \text{DFT}(b)$ using FFT in $O(n \log n)$
3. Multiply point-wise $A$ and $B$, obtain $C$ in $O(n)$
4. Compute the inverse $c = \text{DFT}^{-1}(C)$ using FFT in $O(n \log n)$
5. Return $c$ (bit vector)

Either cases the complexity is $O(n \log n)$