Problem 1. (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 $H_k v$. Analyze the time complexity of your algorithm.

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

$$ H_k v = \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 & H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} $$

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 2. (25 points)

You are given two sorted list of size $m$ and $n$. Give a $O(\log k)$ time algorithm for computing the $k$-th smallest element in the union of the two lists. **Note:** Observe that the $k$-th smallest element in the union of the arrays $a[1 \ldots m]$ and $b[1 \ldots n]$ has to be contained in $a[1 \ldots k]$ or $b[1 \ldots k]$.

**Answer:**

Our algorithm starts off by comparing elements $a[\lfloor k/2 \rfloor]$ and $b[\lfloor k/2 \rfloor]$. Suppose $a[\lfloor k/2 \rfloor] > b[\lfloor k/2 \rfloor]$. What does this tell us about where the $k$-th smallest element $s_k$ must lie? Consider how many elements can be less than $b[\lfloor k/2 \rfloor]$. In $b$, there are $\lfloor k/2 \rfloor - 1$ elements less than $b[\lfloor k/2 \rfloor]$. In $a$, since any element smaller than $b[\lfloor k/2 \rfloor]$ must also be smaller than $a[\lfloor k/2 \rfloor]$, there are at most $\lceil k/2 \rceil - 1$ elements less than $b[\lfloor k/2 \rfloor]$. Combined, there are at most $\lfloor k/2 \rfloor - 1 + \lceil k/2 \rceil - 1 = k - 2$ elements less than $b[\lfloor k/2 \rfloor]$. Thus, $b[\lfloor k/2 \rfloor]$ must be less than $s_k$, and $s_k$ cannot lie in $b[1 \ldots \lceil k/2 \rceil]$. In this case, we recurse on $a[1 \ldots m]$ and $b[\lceil k/2 \rceil + 1 \ldots n]$, lowering $k$ by the number of discarded elements, $\lfloor k/2 \rfloor$ (reducing $k$ by a factor of two).

(We can improve the case above by also considering the number of elements less than $a[\lfloor k/2 \rfloor] + 1$. Since this element is greater than $a[\lfloor k/2 \rfloor] > b[\lfloor k/2 \rfloor]$, there are at least $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$ elements smaller than $a[\lfloor k/2 \rfloor] + 1$. This means that the $k$th largest must be less than $a[\lfloor k/2 \rfloor] + 1$, so we can improve the recursive step: we can recurse on $a[1 \ldots \lceil k/2 \rceil]$ and $b[\lceil k/2 \rceil + 1 \ldots n]$ lowering $k$ by the number of elements discarded from $b$, that is, by $\lceil k/2 \rceil$. In case we discard at least half the elements.)

Or, it could be that $a[\lfloor k/2 \rfloor] < b[\lfloor k/2 \rfloor]$. In this case, consider the number of elements less than $a[\lfloor k/2 \rfloor]$. By the same reasoning, there are at most $\lceil k/2 \rceil - 1 + \lfloor k/2 \rfloor - 1 = k - 2$. So, recurse on $a[\lfloor k/2 \rfloor + 1 \ldots m]$ and $b[1 \ldots n]$, lowering $k$ by $\lfloor k/2 \rfloor$.

(We can improve this case too, by considering the number of elements less than $b[\lceil k/2 \rceil] + 1$. There must be at least $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$ of these, so $s_k < b[\lceil k/2 \rceil] + 1$. We can recurse on $a[\lceil k/2 \rceil + 1 \ldots m]$ and $b[1 \ldots \lceil k/2 \rceil + 1]$, lowering $k$ by $\lfloor k/2 \rfloor$.)
Finally, the third possible case is if $a\lfloor k/2 \rfloor = b\lfloor k/2 \rfloor$. In this case, we know that there are exactly $\lfloor k/2 \rfloor - 1 + \lfloor k/2 \rfloor = k - 2$ elements smaller than $a\lfloor k/2 \rfloor$ and $b\lfloor k/2 \rfloor$, so the $k$-th largest element must be one of these two elements. So, return $a\lfloor k/2 \rfloor$.

Even without the improvements described above, the algorithm cuts $k$ in half with each recursive call, so the algorithm will terminate in $O(\log k)$ recursive calls, where each recursive call costs constant time. The total complexity is therefore $O(\log k)$.

**Problem 3.** (25 points)

Describe and analyze an algorithm that takes an unsorted array $A$ of $n$ integers (in an unbounded range) and an integer $k$, and divides $A$ into $k$ equal-sized groups, such that the integers in the first group are lower than the integers in the second group, and the integers in the second group are lower than the integers in the third group, and so on (however, the integers inside each group do not need to be sorted). For instance if $A = \{4, 12, 3, 8, 7, 9, 10, 20, 5\}$ and $k = 3$, one possible solution would be $A_1 = \{4, 3, 5\}$, $A_2 = \{8, 7, 9\}$, $A_3 = \{12, 10, 20\}$. Sorting $A$ in $O(n \log n)$-time would solve the problem, but we want a faster solution. The running time of your solution should be bounded by $O(nk)$. For simplicity, you can assume that $n$ is a multiple of $k$, and that all the elements are distinct. **Note:** $k$ is an input to the algorithm, not a fixed constant.

**Answer:** Our algorithm first uses linear-time SELECT to find the $n/k$-th smallest element $s_{n/k}$, then scan the original array and PARTITION it into two groups: (1) the elements smaller or equal to $s_{n/k}$ and (2) the elements bigger that $s_{n/k}$. At this point, the first group contains the smallest $n/k$ elements of the array, which is the bottom group $A_1$. Then we use SELECT to find the $n/k$-th order statistic of the remainder of the array, and partitions around it, etc., until all the groups have been separated. Each SELECT and PARTITION requires linear time in the total number of remaining elements, which is at most $n$, so the total running time is $O(nk)$.

**Problem 4.** (25 points)

Given an array of numbers $X = \{x_1, x_2, \ldots, x_n\}$, an exchanged pair in $X$ is a pair $(x_i, x_j)$ such that $i < j$ and $x_i > x_j$. Note that an element $x_i$ can be part of up to $n - 1$ exchanged pairs, and that the maximal possible number of exchanged pairs in $X$ is $n(n - 1)/2$, which is achieved if the array is sorted in descending order. Develop a divide-and-conquer algorithm that counts the number of exchanged pairs in $X$ in $O(n \log n)$ time. Argue why your algorithm is correct, and why your algorithm takes $O(n \log n)$ time. You can assume that $n$ is a power of two.

**Answer:** The main idea is that if we divide the array $X$ into two halves, say $L$ and $R$, the total number of exchanged pairs in $X$ is the sum of the number of exchanged pairs completely contained in $L$ plus the number of exchanged pairs completely contained in $R$ plus the exchanges $(x_i, x_j)$, $i < j$, $x_i > x_j$ where $x_i \in L$ and $x_j \in R$ (let’s call these LR-exchanges). Clearly we can get the number of exchanges completely contained in $L$ and $R$ by calling recursively our algorithm, but in order to meet the $O(n \log n)$ time bound we need to find a way to compute the number of LR-exchanges in linear time. The key observation is that if $L$ and $R$ are sorted, then a linear scan is enough to count them.

So our algorithm is a variant of MERGESORT, where before we perform the actual merging we count in linear time the number of LR-pairs. The complexity is $O(n \log n)$ because the recurrence relation associated with our algorithm has the form $T(n) = 2T(n/2) + O(n)$. 
