Searching an Array Randomly (25 points)

Let $A$ be an unsorted array of length $n$, where each entry of $A$ is an integer. Suppose that we are looking for some integer $x$ in $A$, i.e., we want to find an index $i$ such that $A[i] = x$ if such an index exists. If no such index exists, we should return False. Consider the following randomized algorithm.

- Initially, all indices are unmarked.
- While not all indices are marked:
  - Pick an index $i \in [n]$ uniformly at random.
  - If $A[i] = x$ return $i$.
  - Else mark index $i$.
- Return false

Note that in each iteration we pick an index $i$ uniformly at random from $[n]$, not from the set of unmarked indices. So we might examine a given index more than once.

(a) (12 points) Suppose that $x$ appears in $k \geq 1$ places in $A$, i.e., $|\{i : A[i] = x\}| = k$. What is the expected running time of this algorithm, as a function of $n$ and $k$?

Solution: In each iteration, only constant time is used, so we only need to consider the number of iterations.

Let $X_i$ be a random variable which is 1 if the algorithm ends in iteration $i$ and is 0 otherwise (i.e., it is 0 if the number of iterations is either less than $i$ or greater than $i$). Then $E[X_i] = \Pr[X_i = 1] = \frac{k}{n} (1 - \frac{k}{n})^{i-1}$, since the algorithm ends in iteration $i$ precisely if one of the $k$ indices in which $x$ appears is chosen in iteration $i$ (probability $k/n$) and none of these indices were chosen earlier (probability $(1 - \frac{k}{n})^{i-1}$).
Let $X$ be a random variable denoting the number of iterations. Thus $X = \sum_{i=1}^{\infty} i X_i$. Therefore, by linearity of expectations the expected number of iterations is:

$$
\sum_{i=1}^{\infty} \frac{k}{n} \left(1 - \frac{k}{n}\right)^{i-1} \cdot i = \frac{k}{n} \sum_{i=0}^{\infty} (i+1) \cdot \left(1 - \frac{k}{n}\right)^i
$$

$$
= \frac{k}{n} \left(\sum_{i=0}^{\infty} (1 - \frac{k}{n})^i + \sum_{i=1}^{\infty} (1 - \frac{k}{n})^i + \sum_{i=2}^{\infty} (1 - \frac{k}{n})^i + \ldots\right)
$$

$$
= \frac{k}{n} \sum_{j=0}^{\infty} \sum_{i=j}^{\infty} (1 - \frac{k}{n})^j
$$

$$
= \frac{k}{n} \sum_{j=0}^{\infty} (1 - \frac{k}{n})^j \cdot \frac{1 - (1 - \frac{k}{n})^\infty}{1 - (1 - \frac{k}{n})}
$$

$$
= \sum_{j=0}^{\infty} (1 - \frac{k}{n})^j
$$

$$
= \frac{1 - (1 - \frac{k}{n})^\infty}{1 - (1 - \frac{k}{n})}
$$

$$
= \frac{n}{k}
$$

Thus the expected running time is $\frac{n}{k} \cdot \Theta(1) = \Theta\left(\frac{n}{k}\right)$.

(b) (13 points) Suppose that $x$ does not appear in $A$. What is the expected running time of the algorithm?

**Solution:** We also first calculate the expect number of iterations before all $n$ indices are marked.

Let $t_i$ be the number of iterations between when $i-1$ indices are marked and $i$ indices are marked. Let $T$ denote the total number of iterations. Then by the definition of $t_i$, we get that $T = \sum_{i=1}^{n} t_i$. Thus by linearity of expectations, the expected number of iterations is

$$
\mathbb{E}[T] = \mathbb{E}\left[\sum_{i=1}^{n} t_i\right] = \sum_{i=1}^{n} \mathbb{E}[t_i].
$$

In fact, after $i-1$ indices are marked, the expected number of extra iterations before $i$ indices are marked (i.e. $\mathbb{E}[t_i]$) is the same as the expected number of trials before the first success of Bernoulli($\frac{n+1-i}{n}$) distribution, because in each step the algorithm has probability $\frac{n+1-i}{n}$ to avoid all marked indices. In (a) we have shown that this expectation value is equal to $\frac{n}{n+1-i}$.

Therefore,

$$
\sum_{i=1}^{n} \mathbb{E}[t_i] = \sum_{i=1}^{n} \frac{n}{n+1-i} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = nH_n = n \cdot \Theta(\log n).
$$

Thus the expected running time is $n \cdot \Theta(\log n) \cdot \Theta(1) = \Theta(n \log n)$.  

2
2 Median of Sorted Arrays (25 points)

Let $A$ and $B$ be sorted arrays of $n$ elements each. We can easily find the median of $A$ or the median of $B$, since they are already sorted – it will be at index $\left\lceil \frac{n+1}{2} \right\rceil$. But what if we want to find the median element of $A \cup B$? We could just concatenate them and use the $O(n)$ time median algorithm, but is it possible to do better? In this problem you should give matching upper and lower bounds, i.e. you should find a function $f(n)$ and do the following:

(a) (13 points) Design a deterministic algorithm whose running time (measured in terms of the number of comparisons) is $O(f(n))$, and

Solution: The following algorithm finds the median of $A \cup B$ in $O(\log n)$ time (here we assume that all elements are distinct, and the arrays are indexed starting at 1):

```plaintext
findmedian(A,B,n):
    if n ≤ 2:
        return median(A[1],A[2],B[1],B[2])
    medA = \frac{A[\left\lceil \frac{n+1}{2} \right\rceil]+A[\left\lfloor \frac{n+2}{2} \right\rfloor]}{2}
    medB = \frac{B[\left\lceil \frac{n+1}{2} \right\rceil]+B[\left\lfloor \frac{n+2}{2} \right\rfloor]}{2}
    if (medA < medB):
        return findmedian(A[\left\lfloor \frac{n+1}{2} \right\rfloor+1:B, B[1:\left\lfloor \frac{n+2}{2} \right\rfloor])
    else:
        return findmedian(A[1:\left\lfloor \frac{n+2}{2} \right\rfloor], B[\left\lceil \frac{n+1}{2} \right\rceil:n], \left\lfloor \frac{n+2}{2} \right\rfloor)
```

We can prove that this algorithm is correct by induction on $n$. In the base case, where $n \leq 2$, the algorithm works correctly.

Assume this algorithm is correct for $n < k$. Consider the inductive case, where $n = k$. Without lose of generality, we can assume that $\text{medA} < \text{medB}$. Now, we know that $A[\left\lfloor \frac{n+1}{2} \right\rfloor] \leq \text{medA} < \text{medB} \leq B[\left\lfloor \frac{n+2}{2} \right\rfloor]$, thus $A[\left\lfloor \frac{n+1}{2} \right\rfloor]$ is at most the $\left\lfloor \frac{n+1}{2} \right\rfloor + \left\lfloor \frac{n+2}{2} \right\rfloor - 1 = n$-th smallest. Therefore any element in $A[1: \left\lceil \frac{n-1}{2} \right\rceil]$ will not be the median of $A \cup B$, and they are all less than the median. With similar discussion, we can prove that any element in $B[\left\lceil \frac{n+3}{2} \right\rceil:n]$ will not be the median of $A \cup B$, and they are all greater than the median.

Therefore, the median is in $A[\left\lfloor \frac{n+1}{2} \right\rfloor:n] \cup B[1: \left\lfloor \frac{n+2}{2} \right\rfloor]$. Furthermore, the median of $A \cup B$ is the median of $A[\left\lfloor \frac{n+1}{2} \right\rfloor:n] \cup B[1: \left\lfloor \frac{n+2}{2} \right\rfloor]$, because we removed $\left\lfloor \frac{n+1}{2} \right\rfloor$ elements from both the left and the right side of the median of $A \cup B$. From the inductive assumption, the algorithm correctly returns the median of $A \cup B$.

Now it just remains to compute the runtime. Each call to `findmedian` does a constant amount of work before recursing, so we can write the recurrence relation as:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

We can solve this recurrence using the master theorem, which tells us that $T(n) = \Theta(\log n)$.

(b) (12 points) Give a lower bound showing that any comparison-based algorithm must make $\Omega(f(n))$ comparisons in the worst case.
Solution: Any comparison-based algorithm must use $\Omega(\log n)$ comparisons in the worst case. Again, we can show this using the decision tree method. Each node in the decision tree will correspond to a comparison operation, and each leaf will correspond to a location of the median. The decision tree is binary, since each comparison will return either $<$ or $>$. Observe that the median could be appear at any location in either of the two arrays. (To see this, suppose we want to construct an example where the median appears at position $i$ of array $A$. Then we can design $B$ such that $(n - i)$ of its elements are less than $A[i]$, and $i$ of its elements are greater. Then, when we consider the array $A \cup B$, it will have exactly $(i - 1) + (n - i) = n - 1$ elements less than $A[i]$, and $(n - i) + i = n$ elements greater than $A[i]$.) This means there are $2n$ different locations where the median can appear. Thus, the decision tree needs to have at least $2n$ leaves. And if the decision tree has at least $2n$ leaves, then it needs to have at least $\log_2(2n)$ levels. Thus, any comparison-based algorithm for this problem will take at least $\log_2(2n)$ comparisons in the worst case, meaning the lower bound is $\Omega(\log n)$.

3 More Lower Bounds (25 points)

Consider the following two-dimensional sorting problem: we are given an arbitrary array of $n^2$ numbers (unsorted), and have to output an $n \times n$ matrix of the inputs in which all rows and columns are sorted.

As an example, suppose $n = 3$ so $n^2 = 9$. Suppose the 9 numbers are just the integers \{1, 2, \ldots, 9\}. Then possible outputs include (but are not limited to)

$$
\begin{array}{ccc}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 3 & 5 \\
2 & 4 & 6 \\
7 & 8 & 9 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 6 \\
3 & 4 & 8 \\
5 & 7 & 9 \\
\end{array}
$$

It is obvious that we can solve this in $O(n^2 \log n)$ time by sorting the numbers and then using the first $n$ as the first row, the next $n$ as the second row, etc. For this question, you should prove a matching lower bound of $\Omega(n^2 \log n)$ in the comparison-based model. For simplicity, you can (as always) assume that $n$ is a power of 2.

Hints: instead of reasoning directly about the decision tree, show that if we could solve this problem with $o(n^2 \log n)$ comparisons we could break the sorting lower bound. Useful facts to keep in mind are that $n! > (n/e)^n$ and that we can merge two sorted arrays of length $n$ using $2n - 1$ comparisons.

Solution: We first prove the following Claim:

Claim 1 Given an $2^k \times 2^k$ matrix which all rows and columns are sorted, we can sort all the items in the matrix within $2^{2k} \cdot k$ comparisons.

Proof: The algorithm is as follows: First partition the rows to $2^{k-1}$ groups, where each group has 2 rows. Merge the two rows in each group into a sorted array. Then partition the arrays to $2^{k-2}$ groups, merge the two rows in each group again. Keep doing this until only one array is left. This array must be sorted and it contains all the items in the matrix.
In the \(i\)-th step, there are \(2^{k-i}\) groups, and each array in the group has \(2^{k+i-1}\) items. Thus the number of comparisons used in each step is \(2^{k-i} \times (2 \cdot 2^{k+i-1} - 1) < 2^k\). There are \(k\) steps in total, so the total number of comparisons is at most \(2^k \cdot k\).

Let \(n = 2^k\) and assume that there is an algorithm to sort an \(n \times n\) matrix within \(o(n^2 \log n) = o(2^{k} \cdot k)\) comparisons. Then we can sort all items in an arbitrary \(n \times n\) matrix within \(2^{k} \cdot k + o(2^{k} \cdot k)\) comparisons using Claim 1. However, the input for sorting \(n^2\) items has \((n^2)!\) possibilities (all the permutations), so the number of comparisons to sort \(n^2\) items is at least

\[
\log((n^2)!) > \log \left( \left( \frac{n^2}{e} \right)^{n^2} \right) = 2n^2 \log n - n^2 \log e = 2^{2k+1} \cdot k - 2^k \cdot \log e = 2^{2k+1} \cdot k - O(2^k),
\]

and this is larger than \(2^{2k} \cdot k + o(2^{k} \cdot k)\) when \(k\) is large enough, contradicted.

Therefore, there is no algorithm that can sort an \(n \times n\) matrix within \(o(n^2 \log n)\) comparisons.

4 Linear Time Algorithms (25 points)

Let \(A\) be a list of \(n\) (not necessarily distinct) integers. Design a (deterministic) algorithm to test whether any item occurs more than \([n/2]\) times in \(A\) that runs in \(O(n)\) time (and prove correctness and running time).

**Solution:** Suppose that \(A\) is indexed from 1 to \(n\). The algorithm is simple: we call Select on \(A\) and index \(\left\lceil \frac{n+1}{2} \right\rceil\) (i.e. find the median). Let \(x\) be the integer returned. We then iterate through \(A\) and count the number of times that \(x\) appears. If it appears at least \([n/2]\) times, then we return TRUE. Otherwise we return FALSE.

We first prove the correctness of this algorithm. If there is no item which occurs more than \([n/2]\) times in \(A\), then the algorithm will return FALSE because the median will not occur at least \([n/2]\) times, which is correct. If there is an item which occurs more than \([n/2]\) times in \(A\), then this item must be the median of \(A\). This is because if this item is not the median of \(A\), without lose of generality, we assume it is less than the median, then there are \([n/2]\) items less than the median, which contradict with the definition of median. Therefore, the number of times that the median appears is at least \([n/2]\). Thus the algorithm will return TRUE, which is also correct.

The proof of running time is obvious. Select takes \(O(n)\) time if we use the linear-time select algorithm from class, and then iterating through \(A\) also takes \(O(n)\) time. Hence the total time is \(O(n)\).

**Alternative solution:** Initialize an integer \(v = A[1]\) and a counter \(c = 1\) in the memory. Scan the list starting from the second one. For each item \(A[i]\), if \(A[i] = v\), we increase the counter \(c\) by 1, and otherwise decrease \(c\) by 1. Whenever the counter \(c\) goes below 0, we set \(v\) to be the current item and reset \(c\) to be 1. When finished scanning the full list, scan again to check if \(v\) occurs more than \([n/2]\) times and return the result.

The running time is obviously \(O(n)\), because for each item the algorithm only has constant operations.

To prove the correctness, we first prove the following Claim:

Claim 2 If an integer \(v'\) occurs more than \([n/2]\) times in \(A\), then it must survive in the memory when the first pass of the algorithm ends.
Proof: Every time the algorithm visits a number, the counter $c$ either increase 1 or decrease 1. Let $i_{v'}$ represent the number of increments when visiting $v'$, and $d_{v'}$ represent the number of decrements. Also, let $i_{\text{other}}$ represent the number of increments when visiting other integers, and $d_{\text{other}}$ represent the number of decrements.

If $v'$ occurs more than $\lceil n/2 \rceil$ times in $A$, we know that it occurs more than the summation of all other integers, which means $i_{v'} + d_{v'} > i_{\text{other}} + d_{\text{other}}$, thus

$$i_{v'} - d_{\text{other}} > i_{\text{other}} - d_{v'}$$ (1)

Assume $v'$ is not the number survived in memory, then $i_{\text{other}} - d_{v'} \geq c \geq 0$ must hold because otherwise the counter for other integers will all be canceled out by $v'$. Thus $i_{v'} - d_{\text{other}} > 0$ because of (1), which means the counter with $v'$ cannot be canceled out by other integers, and this means $v'$ is finally survived, contradicted.

Therefore, $v'$ is the number survived in memory.

With this Claim, we can see that if an integer $v'$ occurs more than $\lceil n/2 \rceil$ times, the algorithm must output a TRUE, because it will check the occurrence of $v'$. If no integer occurs more than $\lceil n/2 \rceil$ times, the algorithm will always return FALSE, because no integer will pass the occurrence check.