1. Pick a random element of $B$ and check if this element is in $A$. Checking can be done using binary search in $O(\log n)$ time. Call these two steps a phase of the algorithm. Repeat this phase as many times as it takes to indentify a common element. The probability of success in any phase is $\geq \frac{1}{10}$ since we know that there are $\frac{n}{10}$ common elements between $A$ and $B$. Probability of failure in one phase is $\leq \frac{9}{10}$. Therefore, probability of failing in $k$ successive phases is $\leq \left(\frac{9}{10}\right)^k$. This probability will be $\leq n^{-\alpha}$ if $k \geq \frac{\alpha \log n}{\log(10/9)}$. In other words, the run time of the algorithm is $\tilde{O}(\log^2 n)$.

2. Pick a random sample $S$ of size $c\alpha \log n$ from $A$. If $S$ has only zeros, output Type I. If $S$ has only ones, output Type II. Otherwise output Type III.

   If the input is of type I or II, note that the above algorithm will never output an incorrect answer. However, if the input is of type III, the algorithm might give an incorrect answer. An incorrect answer will be given if $S$ has either only zeros or only ones.

   Assume that the input is of type III. Let $P_0$ be the probability that $S$ has only zeros and let $P_1$ be the probability that $S$ has all ones. Clearly, $P_0 \leq \left(\frac{2}{3}\right)^{c\alpha \log n}$. $P_0$ will be $\leq n^{-2\alpha}$ if $c \geq \frac{2}{\log(4/3)}$. Also, $P_1 \leq \left(\frac{1}{3}\right)^{c\alpha \log n}$. $P_1$ will be $\leq n^{-2\alpha}$ if $c \geq 1$.

   The probability of an incorrect answer is $\leq P_1 + P_2$. This will be $\leq 2n^{-2\alpha} \leq n^{-\alpha}$ if $c \geq \frac{2}{\log(4/3)}$.

3. Use a 2-3 tree and a counter. The value of the counter is set to zero when the 2-3 tree is initialized.

   $INC.ALL(y)$: Increment the counter by $y$. The elements in the 2-3 tree are left untouched. $INSERT(X), FIND.MIN(), DEL.MIN()$ are the standard 2-3 tree algorithms with the following changes. When an element $X$ is inserted, the value of the counter is subtracted from $X$ and the resulting value is inserted into the 2-3 tree. When an element is returned from the 2-3 tree, the value of the counter is added to the element and the resulting value is returned. All the operations take $O(\log n)$ time.

4. Assume w.l.o.g. that $n$ is an integral power of two. We make use of a full binary tree with $n$ leaves. Label the leaves as $1, 2, \ldots, n$ from left to right. We store $a[i]$ in leaf $i$, for $i = 1, 2, \ldots, n$. For each internal node, let its subtreesum refer to the sum of all the numbers in the leaves of the subtree rooted at this node. Let $N$ be any internal node with $L$ and $R$ as its left child and right child, respectively. Let $N_L$ and $N_R$ be the subtreesums of $L$ and $R$, respectively. Each internal node $N$ stores $N_L, N_R$, and $N_S = N_L + N_R$. At the beginning this preprocessing step takes $O(n)$ time and this is done only once.

   To process $Add(i, q)$ we add $q$ to the contents of leaf $i$. We also add $q$ to $N_S$ (and either $N_L$ or $N_R$) of all the internal nodes $N$ in the path starting from the parent of leaf $i$ to the root.
of the tree. The time taken is $O(\log n)$.

To process $\text{PrefixSum}(i)$ we use the following algorithm. We start from leaf $i$ with a value of $\text{Result} := a[i]$ and start ascending toward the root of the tree. If the current node is the right child of its parent node $N$, then when we reach the node $N$, we set: $\text{Result} := \text{Result} + N_L$. If the current node is the left child of its parent node $N$, then when we reach $N$, we don’t change the value of $\text{Result}$. In this manner we travel towards the root and when we reach the root, we update $\text{Result}$, if there is a need. After this update, we output $\text{Result}$. The time taken is $O(\log n)$.

5. Let $S$ represent the sum of all the elements in the array. We need to find an element $k$ such that the sum of all the elements less than or equal to $k$ (let’s call this sum as $LS$) is greater than or equal to $S/2$ and the sum of all the elements greater than or equal to $k$ (let’s call this sum as $US$) is greater than or equal to $S/2$.

Now, select the median (i.e., the $n/2$ th smallest element) $M$ in the array. Let $S'$ represent the sum of all the elements less than or equal to $M$ in the array. Let $S''$ ($= S - S' + M$) represent the sum of all the elements greater than or equal to $M$ in the array. If $S' \geq S/2$ and $S'' \geq S/2$ then $M$ itself is $k$ and the problem is solved. If $S' < S/2$ then do the following: repeat the problem only in the elements greater than $M$ with $LS$ as $S/2 - S'$ and $MS$ as $S/2$. If $S'' < S/2$ then repeat the problem only in the elements less than $M$ with $LS$ as $S/2$ and $US$ as $S/2 - S''$.

Analysis: Time for selecting the $n/2$ th element $= O(n)$ and time for partitioning the array based on $M = O(n)$.

Hence, $T(n) = T(n/2) + O(n)$ which implies $T(n) = O(n)$.

6. One way of solving this problem is to sort the following tuples using any in-place sorting algorithm: $(x_1, k_1), (x_2, k_2), \ldots, (x_n, k_n)$. For example, one could use heap sort. If the sorted sequence is $(1, l_1), (2, l_2), \ldots, (n, l_n)$, then we output $l_1, l_2, \ldots, l_n$. The run time is $O(n \log n)$.

This problem can also be solved in $O(n)$ time as follows.

```plaintext
i := 1;
while i < n do
    if π[i] = i
        then i := i + 1;
    else
        repeat
            swap $K[i]$ and $K[π[i]]$;
            swap $π[i]$ and $π[π[i]]$;
        until $π[i] = i$;
```
The run time of the above algorithm is $O(n)$ since whenever we do the two swaps in the \texttt{repeat} loop once, one element moves to its correct position. Once an element reaches its correct position, it won’t be displaced any more.