1. Suppose that each row of an $n \times n$ array $A$ consists of 1’s and 0’s such that, in any row of $A$, all the 1’s come before any 0’s in that row. Assuming $A$ is already in memory, describe a method running in $O(n \log n)$ time (not $O(n^2)$ time) for counting the number of 1’s in $A$.

Algorithm.

The algorithm is described in Algorithm 1. Essentially, for each row perform a modified binary search to find the index of the first 0. Then, sum up the result of each of the binary search calls to find the total number of 1’s.

Algorithm 1 Count 1’s

Input: Array $A$ of $n \times n$ 0’s and 1’s with specified structure
Output: Count of 1’s in the matrix

1: $c \leftarrow 0$
2: for $i \leftarrow 1 \ldots n$ do
3: \hspace{1em} $m \leftarrow \text{modifiedBinarySearchForFirst0}()$
4: \hspace{1em} $c \leftarrow c + m$
5: return $c$

Time Complexity.

Theorem 1. Algorithm 1 runs in $O(n \log n)$ time.

Proof. A single binary search runs in $O(\log n)$ time. Thus, $n$ of these searches totals to $O(n \log n)$ time.

Memory Complexity.

Theorem 2. Algorithm 1 uses $O(1)$ additional memory.

Proof. A constant number of temporary variables are used to (1) keep track of binary search indices and (2) the count for our algorithm. Thus, there is $O(1)$ additional memory is required.
2. Suppose we are given two sorted search tables $S$ and $T$, each with $n$ entries (with $S$ and $T$ being implemented with arrays). Describe an $O(\log^2 n)$-time algorithm for finding the $k$th smallest key in the union of the keys from $S$ and $T$ (assuming no duplicates).

**Algorithm.**

Algorithm 2 begins by examining the middle element of $S$. Then a binary search in $T$ is performed for the largest element less than that element in $S$. If the sum of their indices is $k$, then we are finished and return the correct element in $S$. If the sum is greater than $k$, then the binary search in $S$ is continued to the left, otherwise to the right. This process of repeated binary searches occurs until the element is found. If the element is not found, then the $k$th smallest element is not in $S$. So, we do a single level recursion by swapping the roles of $S$ and $T$ and repeat the search process.

<table>
<thead>
<tr>
<th>Algorithm 2 kth smallest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Sorted search tables $S$ and $T$, integer $k$</td>
</tr>
<tr>
<td><strong>Output:</strong> $k$th smallest element from $S \cup T$</td>
</tr>
<tr>
<td>1: $l \leftarrow 0; h \leftarrow</td>
</tr>
<tr>
<td>2: while $l \leq h$ do</td>
</tr>
<tr>
<td>3: $m \leftarrow (l + h)/2$</td>
</tr>
<tr>
<td>4: $i \leftarrow \text{BinarySearch}(T, S[m])$</td>
</tr>
<tr>
<td>5: if $m + i + 2 = k$ then</td>
</tr>
<tr>
<td>6: return $S[i]$</td>
</tr>
<tr>
<td>7: else if $m + i + 2 &lt; k$ then</td>
</tr>
<tr>
<td>8: $l \leftarrow m + 1$</td>
</tr>
<tr>
<td>9: else</td>
</tr>
<tr>
<td>10: $h \leftarrow m - 1$</td>
</tr>
<tr>
<td>11: return $\text{kthSmallest}(T, S, k)$ {Not found so reverse $S$ and $T$ and search again}</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 3.** Algorithm 2 runs in $O(\log^2 n)$ time.

**Proof.** A binary search of “probes” occurs in $S$. In each probe, a binary search in $T$ occurs. Each takes $O(\log n)$ time, and the phase of searching $S$ takes $O(\log^2 n)$-time. Since this is repeated at most twice ($S$ and $T$ reversed), the algorithm runs in $O(\log^2 n)$ time. \qed

**Memory Complexity.**

**Theorem 4.** Algorithm 2 uses $O(1)$ additional memory.

**Proof.** A constant number of temporary variables are used in the binary searches, and the recursive depth is 1. Thus, $O(1)$ additional memory is required. \qed
3. **Bonus.** Give an $O(\log n)$-time solution to the previous problem.

**Algorithm.**

Algorithm 3 is essentially a modified binary search over both arrays. The invariant $i + j = k - 1$ is maintained at all times. Because of this, the $k$th smallest is easy to identify and the pivots are also easy to create in a binary search fashion. The details are given in the algorithm, but essentially, if $S[i]$ is between $T[j - 1]$ and $T[j]$ then $S[i]$ is the $k$th smallest. A similar case exists for $T[j]$. Then, for narrowing the search, if $S[i] < T[j]$ we can exclude the bottom part of $S$ and the top part of $T$ and adjust $k$ appropriately. Similar for the opposite case.

**Algorithm 3 $k$th smallest**

**Input:** Sorted search tables $S$ and $T$, integer $k$

**Output:** $k$th smallest element from $S \cup T$

1: $l_S \leftarrow 0; h_S \leftarrow |S| - 1; l_T \leftarrow 0; h_T \leftarrow |T| - 1$
2: while $l_S \leq h_S \land l_T \leq h_T$ do
3: $m \leftarrow h_S - l_S + 1; n \leftarrow h_T - l_T + 1$
4: $i \leftarrow \lfloor (k - 1)m/(m + n) \rfloor$
5: $j \leftarrow k - 1 - i$
6: if $T[j - 1] < S[i] < T[j]$ then
7: return $S[i]$
8: else if $S[i - 1] < T[j] < S[i]$ then
9: return $T[j]$
10: else if $S[i] < T[j]$ then
11: $l_S \leftarrow i + 1$
12: $h_T \leftarrow j$
13: $k \leftarrow k - i - 1$
14: else
15: $l_T \leftarrow j + 1$
16: $h_S \leftarrow i$
17: $k \leftarrow k - j - 1$

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in $O(\log n)$ time.

*Proof.* Half of the elements are discard each step in the search for the answer. Thus, there are a logarithmic number of steps in the algorithm and it runs in $O(\log n)$ time.

**Memory Complexity.**

**Theorem 6.** Algorithm 3 uses $O(1)$ additional memory.

*Proof.* A constant number of temporary variables are used in the binary searches. Thus, $O(1)$ additional memory is required.