1. Assume \( n \geq 1 \) is a power of 2. Let \( A = [A_1 \leq A_2 \leq \cdots \leq A_n] \) be a sorted array of \( n \) integers. Let \( x \) be an integer. Design an efficient algorithm that finds the number of times \( x \) appears in the array.

What is the worst-case number of comparisons made by your algorithm?

**Non-efficient Algorithm:** Let \( c \) be the number of times \( x \) appears in the array.

- Using the binary search procedure, check if \( x \) appears in the array.
- If \( x \notin A \), return \( c = 0 \).
- Otherwise, the binary search procedure returns the first index \( i \) such that \( A_i = x \).
- Scan the array starting with \( j = i + 1 \) until \( j = n \) or \( A_j > x \).
- If \( A_n = x \), return \( c = n - i + 1 \).
- If \( j \) is the smallest index such that \( A_j > x \), return \( c = j - i \).

**Correctness:** Because the array is sorted, all the appearances of \( x \) in \( A \) must be consecutive. Scanning the array from the position of the first appearance of \( x \) counts the number of appearances of \( x \) in \( A \).

**Complexity:** When \( n \) is a power of 2, the binary search procedure always makes \( \log_2(n) \) comparisons. In the worst case, all the entries in the array are equal to \( x \). In this case, the scanning part of the algorithm requires \( n - 1 \) additional comparisons. The total worst-case complexity is therefore \( n + \log_2(n) - 1 \) comparisons.

**Efficient Algorithm:** Let \( c \) be the number of times \( x \) appears in the array.

- Using the binary search procedure, check if \( x \) appears in the array.
- If \( x \notin A \), return \( c = 0 \).
- Otherwise, the binary search procedure returns the first index \( i \) such that \( A_i = x \).
- Using the binary search procedure on the range \((i + 1)\ldots n\), check if \( x + 1 \) appears in the array.
- If \( (x + 1) \in A \), let \( j \) be the first index \( j \) such that \( A_j = x + 1 \). Return \( j - i \).
- If \( (x + 1) \notin A \), let \( j \) be the largest index \( j \) such that \( A_j < x + 1 \). Return \( j - i + 1 \).

**Correctness:** Assume first that \((x + 1) \in A \). Since the array contains only integers, \( j \) the location of the first appearance of \( x + 1 \) in the array and \( i \) the location of the first appearance of \( x \) in the array determine the number of times \( x \) appears in the array to be \( j - i \) since \( A_i = A_{i+1} = \cdots = A_{j-1} = x \).

Now, if \( x + 1 \notin A \), then \( A_j < x + 1 \). But since \( j \geq i \), it follows that also \( A_j \geq A_i = x \) and therefore necessarily \( A_i = A_{i+1} = \cdots = A_j = x \). As a result, in this case \( x \) appears \( j - i + 1 \) times in the array.

**Complexity:** \( 2 \log_2(n) \) comparisons are made by running the binary search procedure twice for \( x \) and \( x + 1 \).

**Remark:** In the worst case, \( A_1 = x \) and the search for \( x + 1 \) is in the range \([2..n]\) which requires \( \lfloor \log_2(n - 1) \rfloor = \log_2(n) \) additional comparisons to the \( \log_2(n) \) comparisons required for the search for \( x \).
2. Assume \( n \geq 1 \) is a power of 2. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be a sorted array of \( n \) distinct positive integers. Let \( x \leq y \) be two positive integers. Describe an efficient algorithm that determines if one of the numbers \( x, x+1, \ldots, y \) appears in the array. What is the worst-case number of comparisons made by your algorithm?

Efficient Algorithm:

(a) Using the binary search procedure, check if \( x \) appears in the array.
(b) If \( x \in A \), then return YES.
(c) If \( x \not\in A \), let \( i \) be the largest index \( i \) such that \( A_i < x \) and let \( i = 0 \) if \( x < A_1 \).
(d) If \( i = n \), then return NO.
(e) Otherwise, compare \( A_{i+1} \) with \( y \).
(f) If \( A_{i+1} \leq y \), then return YES.
(g) If \( A_{i+1} > y \), then return NO.

Correctness: The following arguments justify the correctness in parts (b), (d), (f), and (g) of the algorithm.

(b) In this case, \( x \in A \). The output is YES, since \( x \in [x..y] \).
(d) In this case, \( x \not\in A \) and \( A_n < x \). The output is NO, since all the numbers in the array are smaller than \( x \).
(f) In this case, \( A_{i+1} \leq y \). Since \( i \) is the largest index such that \( A_i < x \), it follows that \( x < A_{i+1} \). Thus, \( x < A_{i+1} \leq y \) which implies that \( A_{i+1} \in [x..y] \). Therefore the output is YES.
(g) In this case, \( x \not\in A \), \( A_i < x \), and \( A_{i+1} > y \). Since the array is sorted, it follows that \( A_j < x \) for \( 1 \leq j \leq i \) and \( y < A_j \) for \( i+1 \leq j \leq n \). This implies that no number from the range \([x..y]\) appears in the array. Therefore the output is NO.

Complexity: The binary search procedure made exactly \( \log_2(n) \) comparisons since \( n \) is a power of 2. The algorithm might make one more comparison in part (e). The total complexity is therefore \( \log_2(n) + 1 \) comparisons in the worst-case.

Other algorithms: A non-efficient algorithm would run the binary search procedure for each of the \( y - x + 1 \) numbers in the range \([x..y]\). If none of these numbers are in \( A \), then the complexity of this algorithm is \( (y-x+1) \log_2(n) \) comparisons. This number could be very large (e.g., exponential in \( n \)) if \( y \) is much larger than \( x \). A more efficient algorithm would run the binary search procedure for both \( x \) and \( y \) and based on the “potential” locations of \( x \) and \( y \) in the array, if they are not in the array, would determine the correct answer. The complexity of this algorithm is \( 2 \log_2(n) \) which is inferior to the complexity of the efficient algorithm.
3. Assume $n \geq 1$ is a power of 2. Let $A = [A_1 < A_2 < \cdots < A_n]$ be a sorted array of $n$ distinct positive integers. Let $x \leq y$ be two positive integers. Describe an efficient algorithm that determines if all the numbers $x, x+1, \ldots, y$ appear in the array. What is the worst-case number of comparisons made by your algorithm?

**Efficient Algorithm:**

(a) Using the binary search procedure, check if $x$ appears in the array.
(b) If $x \not\in A$, then return NO.
(c) If $x \in A$, let $i$ be the index such that $A_i = x$.
(d) If $i + y - x > n$, then the answer is NO.
(e) Otherwise, compare $A_{i+y-x}$ with $y$.
(f) If $A_{i+y-x} = y$, then return YES.
(g) If $A_{i+y-x} > y$, then return NO.

**Correctness:** The following arguments justify the correctness in parts (b), (d), (f), and (g) of the algorithm.

(b) In this case, $x \not\in A$ and therefore the output is NO.
(d) In this case, $x \in A$ and. If all the numbers from the range $[x..y]$ are in $A$ then

$$A_{i+1} = x + 1, A_{i+2} = x + 2, \ldots, A_n = x + n - i$$

The output is NO, since in this case $x + n - i < y$ and therefore $y \not\in A$.
(f) In this case, $A_i = x$ and $A_{i+y-x} = y$. This could happens only if

$$A_{i+1} = x + 1, A_{i+2} = x + 2, \ldots, A_{i+y-x} = y$$

This implies that all the numbers from the range $[x..y]$ are in $A$ and therefore the output is YES.
(g) In this case, $A_i = x$ and $A_{i+y-x} > y$. This implies that the $y - x + 1$ numbers from the range $[x..y]$ if they all appear in $A$ they are all appear between $A[i]$ and $A[i+y-x-1]$. By the Pigeonhole Principle, at least one of the numbers from the range $[x..y]$ is not in $a$ and therefore the output is NO.

**Complexity:** The binary search procedure makes exactly $\log_2(n)$ comparisons since $n$ is a power of 2. The algorithm might make one more comparison in part (e). The total complexity is therefore $\log_2(n) + 1$ comparisons in the worst-case.

**Other algorithms:** A non-efficient algorithm would run the binary search procedure for each of the $y - x + 1$ numbers in the range $[x..y]$. If all of these numbers are in $A$, then the complexity of this algorithm is $(y - x + 1) \log_2(n)$ comparisons. This number could be very large (e.g., exponential in $n$) if $y$ is much larger than $x$. A more efficient algorithm would run the binary search procedure for both $x$ and $y$ and based on the locations of $x$ and $y$ in the array, if they are in the array, would determine the correct answer. The complexity of this algorithm is $2 \log_2(n)$ which is inferior to the complexity of the efficient algorithm.
4. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be a sorted array of \( n \) distinct integers (could be positive and/or negative). Describe an efficient algorithm that finds, if exists, an index \( 1 \leq i \leq n \) such that \( A_i = i \). What is the worst-case number of comparisons made by your algorithm?

**Lemma:** \( j - i \leq A_j - A_i \) for any two indices \( 1 \leq i < j \leq n \).

**Proof:** Since the integers in the sorted array are distinct, it follows that
\[
1 \leq A_{i+1} - A_i \\
2 \leq A_{i+2} - A_i \\
3 \leq A_{i+3} - A_i \\
\vdots \\
(j - i) \leq A_{i+(j-i)} - A_i
\]
The last inequality is equivalent to \( j - i \leq A_j - A_i \).

**Corollary I:** \( j - A_j \leq i - A_i \) for any two indices \( 1 \leq i < j \leq n \).

**Corollary II:** \( A_j - j \geq A_i - i \) for any two indices \( 1 \leq i < j \leq n \).

**Proposition I:** For \( 1 < j \leq n \), if \( A_j < j \) then \( A_i < i \) for all \( 1 \leq i < j \).

**Proof:** By Corollary I, \( j - A_j \leq i - A_i \). The assumption \( A_j < j \) is equivalent to \( 0 < j - A_j \). Therefore by transitivity, \( 0 < i - A_i \) which is equivalent to \( A_i < i \).

**Proposition II:** For \( 1 \leq i < n \), if \( A_i > i \) then \( A_j > j \) for all \( i < j \leq n \).

**Proof:** By Corollary II, \( A_j - j \geq A_i - i \). The assumption \( A_i > i \) is equivalent to \( A_i - i > 0 \). Therefore by transitivity, \( A_j - j > 0 \) which is equivalent to \( A_j > j \).

**Algorithm:** Apply a Binary Search like procedure. As long as an index \( i \) for which \( A_i = i \) has not been found, the search continues in a range \([\ell..r]\) of the array for some \( 1 \leq \ell \leq r \leq n \). Initially, \( \ell = 1 \) and \( r = n \). The search returns a negative answer if \( \ell > r \).

**Recursive step** for the range \([\ell..r]\) for which \( \ell \leq r \): Let \( m = \left\lfloor \frac{\ell + r}{2} \right\rfloor \) be the middle index of the range \([\ell..r]\). Compare \( A_m \) with \( m \).
- If \( A_m = m \): return \( m \).
- If \( A_m < m \): continue recursively with the range \([(m+1)..r]\). (*Proposition I*)
- If \( A_m > m \): continue recursively with the range \([\ell..(m-1)]\). (*Proposition II*)

**Correctness:** Implied by Propositions I and II.

**Complexity:** The size of the range of the next recursive step is at most half of the size of the current range. Therefore, there are at most \( \lceil \log(n) \rceil \) recursive steps. The time complexity of each recursive step is \( \Theta(1) \) which implies that the complexity of the algorithm is \( \Theta(\log n) \).

**Remark:** If distinguishing between the three cases \( A_m = m, A_m < m, \) and \( A_m > m \) can be done with one comparison, then the exact number of comparisons is \( \lceil \log_2(n) \rceil \).
5. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be an array of \( n \) distinct integers sorted in an ascending order and let \( B = [B_1 > B_2 > \cdots > B_n] \) be an array of \( n \) distinct integers sorted in a descending order. Describe an efficient algorithm that finds, exists, an index \( 1 \leq i \leq n \) such that \( A_i = B_i \). What is the worst-case number of comparisons made by your algorithm?

Proposition I: For \( 1 < j \leq n \), if \( A_j < B_j \) then \( A_i < B_i \) for all \( 1 \leq i < j \).

Proof: Since \( i < j \), the ways both arrays are sorted imply that \( A_i < A_j \) and \( B_j < B_i \). Combining these inequalities with the assumption’s inequality \( A_j < B_j \) implies that 
\[
A_i < A_j < B_j < B_i
\]

Proposition II: For \( 1 \leq i < n \), if \( B_i < A_i \) then \( B_j < A_j \) for all \( i < j \leq n \).

Proof: Since \( i < j \), the ways both arrays are sorted imply that \( A_i < A_j \) and \( B_j < B_i \). Combining these inequalities with the assumption’s inequality \( B_i < A_i \) implies that 
\[
B_j < B_i < A_i < A_j
\]

Algorithm: Apply a Binary Search like procedure. As long as an index \( i \) for which \( A_i = B_i \) has not been found, the search continues in a range \([\ell..r]\) of the arrays for some \( 1 \leq \ell \leq r \leq n \). Initially, \( \ell = 1 \) and \( r = n \). The search returns a negative answer if \( \ell > r \).

Recursive step for the range \([\ell..r]\) for which \( \ell \leq r \): Let \( m = \lfloor \ell + r/2 \rfloor \) be the middle index of the range \([\ell..r]\). Compare \( A_m \) with \( m \).
- If \( A_m = B_m \): return \( m \).
- If \( A_m < B_m \): continue recursively with the range \([(m+1)..<r]\). (* Proposition I *)
- If \( B_m < A_m \): continue recursively with the range \([\ell..<(m-1)]\). (* Proposition II *)

Correctness: If \( A_m < B_m \), Proposition I implies that \( A_i < B_i \) for all \( \ell \leq i \leq m \) and therefore the search should continue only in the range \([(m+1)..<r]\). If \( B_m < A_m \), Proposition II implies that \( B_j < A_j \) for all \( m \leq j \leq r \) and therefore the search should continue only in the range \([\ell..<(m-1)]\).

Complexity: The size of the range of the next recursive step is at most half of the size of the current range. Therefore, there are at most \( \lceil \log(n) \rceil \) recursive steps. The time complexity of each recursive step is \( \Theta(1) \) which implies that the complexity of the algorithm is \( \Theta(\log n) \).

Remark: If distinguishing between the three cases \( A_m = B_m, A_m < B_m, \) and \( B_m < A_m \) can be done with one comparison, then the exact number of comparisons is \( \lceil \log_2(n) \rceil \).