Algorithms

Assignment Solutions: Array Problems
1. Let $A = A[1] < A[2] < \cdots < A[n]$ be a sorted array containing $n$ distinct negative and positive integers. Describe an efficient algorithm that finds, if it exists, an index $1 \leq i \leq n$ such that $A[i] = i$. What is the complexity of your algorithm?

**A trivial linear complexity algorithm:** For all indices $1 \leq i \leq n$, check if $A[i] = i$. If such an index is found, return it. Otherwise, after learning that $A[n] \neq n$, return a message that such an index does not exist.

Algorithm $\mathcal{X}(A)$:

- \textbf{for} $i := 1 \text{ \textbf{to} } n \text{ \textbf{do}}$
  - \textbf{if} $A[i] = i$ \textbf{then} \textbf{return}$(i)$
  - \textbf{return}(“$A[i] \neq i$ for all indices $1 \leq i \leq n$ in $A$”)

Algorithm $\mathcal{X}$ is correct because by inspecting all the $n$ indices in $A$, it cannot miss, if it exists, an index $i$ for which $A[i] = i$.

The complexity of algorithm $\mathcal{X}$ is $\Theta(n)$ because in the worst-case the algorithm needs to examine all the $n$ entries in the array with complexity $\Theta(1)$ for each entry and $n \cdot \Theta(1) = \Theta(n)$.

**Remark:** Algorithm $\mathcal{X}$ is correct with the same linear complexity even if the array is not sorted.

**Observation:** $A[i+1] - (i + 1) \geq A[i] - i$ for $1 \leq i < n$.


**A linear complexity algorithm with $\Theta(\log(n))$ comparisons:** Define an array $B$ such that $B[i] = A[i] - i$ for $1 \leq i \leq n$. It follows that if $B[i] = 0$ for some $1 \leq i \leq n$ then $A[i] = i$. The above observation implies that $B[1] \leq B[2] \leq \cdots \leq B[n]$. Use Binary-Search to find if $0$ appears in the array $B$. Return the index $i$ if there exists $1 \leq i \leq n$ such that $B[i] = 0$. Otherwise return the message $A[i] \neq i$ for all $1 \leq i \leq n$.

Algorithm $\mathcal{Y}(A)$:

- \textbf{for} $i := 1 \text{ \textbf{to} } n \text{ \textbf{do}}$
  - $i := \text{Binary-Search}(B, 0)$
  - \textbf{if} $A[i] = i$ \textbf{then} \textbf{return}$(i)$
  - \textbf{else} \textbf{return} (“$A[i] \neq i$ for all indices $1 \leq i \leq n$ in $A$”)

By definition of the array $B$, it follows that if $B[i] = 0$ for some $1 \leq i \leq n$ then $A[i] = i$. Since $B$ is sorted, the Binary-Search procedure finds the smallest index $i$ such that $B[i] = 0$. On the other hand, if $0$ is not in $B$ then the Binary-Search procedure returns an index $i$ for which $B[i] \neq 0$ and therefore $A[i] \neq i$. In this case, algorithm $\mathcal{Y}$ returns a negative message. Both arguments prove that algorithm $\mathcal{Y}$ is correct.

Algorithm $\mathcal{Y}$ is using $\Theta(\log(n))$ comparisons which is the complexity of the Binary-Search procedure. However, the overall complexity of the algorithm is $\Theta(n)$ since the for loop that defines the array $B$ has $n$ iterations.

**A $\Theta(\log(n))$-complexity algorithm:** In fact, there is no need for array $B$. The comparison $B[i] = 0$ is equivalent to the comparison $A[i] = i$. Therefore, the Binary-Search procedure can be modified to run directly on the array $A$.

Algorithm $\mathcal{Z}(A)$:

- $\ell := 1$ and $u := n$
- \textbf{while} $\ell < u$ \textbf{do}
  - $m := \left\lfloor \frac{u + \ell}{2} \right\rfloor$
  - \textbf{if} $A[m] > m$ \textbf{then} $u := m$
  - \textbf{else} $\ell := m + 1$
  - \textbf{if} $A[\ell] = \ell$ \textbf{then} \textbf{return}$(\ell)$
  - \textbf{else} \textbf{return} (“$A[i] \neq i$ for all indices $1 \leq i \leq n$ in $A$”)

Algorithm $\mathcal{Z}$ is correct because it is equivalent to algorithm $\mathcal{Y}$.

The while loop in algorithm $\mathcal{Z}$ has at most $\lceil \log_2(n) \rceil$ iterations the same number of iterations that the Binary-Search procedure has. The complexity of algorithm $\mathcal{Z}$ is $\Theta(\log(n))$ since each iteration has a $\Theta(1)$-complexity and $\Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n))$. 


2. 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, if it 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 \( m = \left\lfloor \frac{\ell + r}{2} \right\rfloor \) be the middle index of the range \([\ell..r]\). Compare \( A_m \) with \( B_m \).

- 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 \( \Theta(n) \).

**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 \( \lceil \log_2(n) \rceil \).
3. For \( n \geq 1 \), let \( A \) be an array of size \( n \) for which the first \( k \) entries contain positive integers and the rest of the array is all zeros. The value of \( n \) is known but the value of \( k \), which can be any number between 0 and \( n \), is unknown.

Examples:
- \([34, 13, 21, 0, 0, 0, 0, 0]\): \( k = 3 \) in this array of length 8.
- \([0, 0, 0, 0, 0, 0, 0, 0]\): \( k = 0 \) in this array of length 7.
- \([55, 8, 34, 13, 21, 89]\): \( k = 6 \) in this array of length 6.

Describe an efficient algorithm that determines the value of \( k \) which is the number of positive integers in \( A \). What is the complexity of your algorithm?

\textbf{A trivial linear complexity algorithm:} Scan the array starting with the first entry in the array until either finding a zero or reaching the end of the array.

Algorithm \( X(A) \):
\[
\begin{align*}
    k &:= 0 \\
    \text{while } (k < n) \text{ and } (A[k+1] > 0) \text{ do} \\
    &\quad k := k + 1 \\
\end{align*}
\]
return \( k \)

Algorithm \( X \) is correct because by inspecting all the \( n \) indices in \( A \), the algorithm identifies the last non-zero entry if it exists or returns 0 if \( A \) contains only zeros.

The complexity of algorithm \( X \) is \( \Theta(n) \) because in the worst-case the algorithm needs to examine all the \( n \) entries in the array with complexity \( \Theta(1) \) for each entry and \( n \cdot \Theta(1) = \Theta(n) \).

\textbf{A \( \Theta(\log n) \)-complexity algorithm:} Run the Binary-Search procedure to find the last zero in the array \( A \). The rules for the binary-search are that if \( A[i] = 0 \) then it must be the case that \( k < i \) while if \( A[i] > 0 \) it must be the case that \( k \geq i \).

Algorithm \( Y(A) \):
\[
\begin{align*}
    A[0] &:= 1 \\
    \ell &:= 0 \\
    r &:= n \\
    \text{while } (\ell < r) \text{ do} \\
    &\quad m := \left\lfloor \frac{\ell + r}{2} \right\rfloor \\
    &\quad \text{if } A[m] = 0 \text{ then} \\
    &\quad \quad r := m \\
    &\quad \text{else } \ell := m + 1 \\
\end{align*}
\]
return \( \ell \)

Algorithm \( Y \) is correct because the binary-search will find the last appearance of a positive integer in \( A \) which always exists after defining \( A[0] = 1 \).

The number of iterations in algorithm \( Y \) is \( \Theta(\log(n+1)) = \Theta(\log(n)) \) which is the complexity of the Binary-Search procedure on an array of length \( n + 1 \). Consequently, the complexity of algorithm \( Y \) is \( \Theta(\log(n)) \) since each iteration has a \( \Theta(1) \)-complexity and \( \Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n)) \).

Describe an efficient algorithm that finds the number of times $k$ appears in the array. What is the complexity of your algorithm?

**A trivial linear complexity algorithm:** Count the number of indices for which $A[i] = k$ by scanning the whole array.

Algorithm $\mathcal{X}(A)$:

- $Count := 0$
- for $i := 1$ to $n$ do
  - if $A[i] = k$
    - then $Count := Count + 1$
- return ("$k$ appears $Count$ times in $A$")

Algorithm $\mathcal{X}$ is correct because it examines all the integers in the array $A$.

There are $n$ iterations of the for loop in algorithm $\mathcal{X}$ and the complexity of each iteration is $\Theta(1)$. Therefore, the complexity of algorithm $\mathcal{X}$ is $\Theta(n)$ because $n \cdot \Theta(1) = \Theta(n)$.

**Remark:** Algorithm $\mathcal{X}$ is correct with the same linear complexity even if the array is not sorted.

**A $\Theta(\log n)$-complexity algorithm:** Run the Binary-Search procedure twice to search in $A$ for $k$ and $k + 1$ and then deduce the number of times $k$ appears in the array.

**Assumption:** When the Binary-Search procedure is looking to find a key in an array, it returns its first location if it appears at least once in the array. Otherwise it returns 0 if $k < A[1]$, returns $n$ if $A[n] < k$, and returns the index $i$ for which $A[i] < k < A[i + 1]$. The complexity of this version of Binary-Search is still $\Theta(\log n)$.

Algorithm $\mathcal{Y}(A)$:

- Run Binary-Search to find if $k$ appears in $A$.
- If $k$ does not appear in $A$:
  - return ("$k$ appears 0 times in $A$").
- Otherwise, assume $A[i] = k$ while $A[i - 1] < k$ or $i = 1$.
- Run Binary-Search to find if $k + 1$ appears in the sub array $A[i + 1] \leq \cdots \leq A[n]$.
  - return ("$k$ appears $(j - i + 1)$ times in $A$"").
- Otherwise, the Binary-Search returns $j$ such that $A[j] = k$ and $A[j + 1] > k + 1$. As in the previous case:
  - return ("$k$ appears $(j - i + 1)$ times in $A$"").

The high level description of algorithm $\mathcal{Y}$ explains why algorithm $\mathcal{Y}$ is correct.

The complexity of algorithm $\mathcal{Y}$ is $\Theta(\log n)$ which is the complexity of two executions of the Binary-Search procedure.

**Remark:** Consider the following variation of algorithm $\mathcal{Y}$ called algorithm $\mathcal{Z}$. After finding that the first appearance of $k$ is in $A[i]$, algorithm $\mathcal{Z}$ counts the number of appearances of $k$ in $A$ sequentially. Assume $A[i] = k$ for some $1 \leq i \leq n$. Then algorithm $\mathcal{Z}$ checks if $A[i + 1] = k$, $A[i + 2] = k$, … until it finds $j$ such that $A[j + 1] > k$ or until $j = n$. Then algorithm $\mathcal{Z}$ returns that $k$ appears $(j - i + 1)$ times in $A$.

While algorithm $\mathcal{Z}$ is more efficient than algorithm $\mathcal{Y}$ when $(j - i + 1)$ is small, in the worst case when $(j - i + 1) = \Theta(n)$, the complexity of algorithm $\mathcal{Z}$ is $\Theta(n)$. 

5
5. For \( n \geq 2 \), let \( A = A[1] < A[2] < \cdots < A[n] \) be a sorted array with \( n \) distinct positive integers from the range \( 1, 2, \ldots, n+1 \). That is, exactly one of the integers from this range is missing in the array \( A \).

**Examples:** The missing integer in the array \([1, 2, 3, 4, 5, 7, 8, 9]\) is 6, the missing integer in the array \([1, 2, 4, 5]\) is 3, the missing integer in the array \([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15]\) is 12, the missing integer in the array \([2, 3, 4, 5, 6, 7]\) is 1, and the missing integer in the array \([1, 2, 3, 4, 5, 6, 7, 8, 9]\) is 10.

Describe an efficient algorithm that finds the missing integer. The only questions about the integers in the arrays that your algorithm may ask are of the type “is \( A[i] = i? \)” for some integer \( 1 \leq i \leq n \). What is the worst-case complexity (number of questions asked) of the algorithm?

**A trivial linear complexity algorithm:** For all indices \( 1 \leq i \leq n \), check if \( A[i] = i \). The missing integer is \( i \) when \( A[i] \neq i \) which implies that \( A[i] = i + 1 \). If at the end \( A[n] = n \) then the missing integer is \( n + 1 \).

Algorithm \( X(A) \):
- for \( i := 1 \) to \( n \) do
  - if \( A[i] = i \) then continue
  - else return \( (i) \)
- return \((n + 1)\)

Algorithm \( X \) is correct because it inspects all the \( n \) indices in \( A \). If the missing number is \( i < n + 1 \), then it must be found during the scan because in this case \( A[i] = i + 1 \). If the missing number is \( n + 1 \), then the procedure finishes the scan and returns \( n + 1 \).

The complexity of algorithm \( X \) is \( \Theta(n) \) because in the worst-case the algorithm needs to examine all the \( n \) entries in the array with complexity \( \Theta(1) \) for each entry and \( n \cdot \Theta(1) = \Theta(n) \).

**A \( \Theta(\log n) \)-complexity algorithm:** Run the Binary-Search procedure to find the first index in the array \( A \) for which \( i < A[i] \).

Algorithm \( Y(A) \):
- \( A[n + 1] := n + 2 \)
- \( \ell := 1 \)
- \( r := n + 1 \)
- while \( (\ell < r) \) do
  - \( m := \left\lceil \frac{\ell + r}{2} \right\rceil \)
  - if \( A[m] = m \) then \( \ell := m + 1 \)
  - else \( r := m \)
- return \( (\ell) \)

Algorithm \( Y \) is correct because the binary-search will find the first index \( i \) in \( A \) such that \( A[i] = i + 1 \) which always exists after defining \( A[n + 1] = n + 2 \).

The number of iterations in algorithm \( Y \) is \( \Theta(\log(n + 1)) = \Theta(\log(n)) \) which is the complexity of the Binary-Search procedure on an array of length \( n + 1 \). Consequently, the complexity of algorithm \( Y \) is \( \Theta(\log(n)) \) since each iteration has a \( \Theta(1) \)-complexity and \( \Theta(\log(n)) \cdot \Theta(1) = \Theta(\log(n)) \).

**Remark:** There is a way to solve this problem without comparisons if the algorithm may apply addition operations involving entries of the array. First, find the sum \( S \) of all the integers in the array. If all the integers between 1 and \( n + 1 \) were in the array, then the sum would have been \( 1 + 2 + \cdots + (n + 1) = \frac{(n + 1)(n + 2)}{2} \).

As a result the missing number is

\[
\frac{(n + 1)(n + 2)}{2} - S
\]

For example, for the array \([1, 2, 3, 4, 5, 7, 8, 9]\) the sum is \( S = 39 \). The missing number is 6 because

\[
\frac{9 \cdot 10}{2} - 39 = 45 - 39 = 6
\]
6. 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 integers \( x, x + 1, \ldots, y \) 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 \textbf{YES}.

(c) If \( x \notin A \), 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 \textbf{NO}.

(e) Otherwise, compare \( A_{i+1} \) with \( y \).

(f) If \( A_{i+1} \leq y \), then return \textbf{YES}.

(g) If \( A_{i+1} > y \), then return \textbf{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 \textbf{YES}, since \( x \in [x..y] \).

(d) In this case, \( x \notin A \) and \( A_n < x \). The output is \textbf{NO}, since all the integers 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 \textbf{YES}.

(g) In this case, \( x \notin A \), \( A_i < x \), 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 integer from the range \( [x..y] \) appears in the array. Therefore the output is \textbf{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 \( y - x + 1 \) integers in the range \( [x..y] \). If none of these integers are in \( A \), then the complexity of this algorithm is \( (y-x+1) \log_2(n) \) comparisons. This integer 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.
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 integers \( x, x+1, \ldots, y \) 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 \( x \notin A \), 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, \( x \notin A \) and therefore the output is NO.

(d) in this case, \( x \in A \) and. If all the integers 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 \( y \notin A \).

(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 integers 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 \( y - x + 1 \) integers 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 integers 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 \( y - x + 1 \) integers in the range \([x..y]\). If all of these integers are in \( A \), then the complexity of this algorithm is \( (y - x + 1) \log_2(n) \) comparisons. This integer 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.
8. Let \( A = [A_1, A_2, \ldots, A_n] \) be an unsorted array of \( n \geq 1 \) positive integers. Design an efficient algorithm that finds the maximum difference between any two integers in the array. In other words, compute \( M = \max_{1 \leq i, j \leq n} \{A_i - A_j\} \).

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

Observation: Let \( \text{Max} \) and \( \text{Min} \) be the maximum and minimum integers in \( A \). It follows that the answer is

\[ M = \max_{1 \leq i, j \leq n} \{A_i - A_j\} = \text{Max} - \text{Min} \]

Algorithm: Find both \( \text{Max} \) and \( \text{Min} \) with the algorithm that performs exactly \( \lceil \frac{3n^2}{2} \rceil - 2 \) comparisons.

Optimality: The algorithm is optimal since there is no algorithm that can find both \( \text{Max} \) and \( \text{Min} \) with less than \( \lceil \frac{3n^2}{2} \rceil - 2 \) comparisons.

9. Let \( A = [A_1, A_2, \ldots, A_n] \) be an unsorted array of \( n \geq 1 \) positive integers. Design an efficient comparison-based algorithm that finds the minimum positive difference between any two integers in the array. In other words, compute \( m = \min_{1 \leq i, j \leq n} \{|A_i - A_j|\} \).

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

Trivial algorithm: Compute the minimum

\[ m = \min_{1 \leq i, j \leq n} \{A_i - A_j\} \]

only for indices \( 1 \leq i \neq j \leq n \) such that \( A[i] > A[j] \).

Correctness: By definition.

Complexity: Can be done with two for loops implying a \( \Theta(n^2) \) complexity.

Efficient Algorithm: Sort the array. If \( A_1 = A_n \) then all the integers are the same and there is no minimum positive difference. Otherwise, the answer can be found with the following scan of the array:

\[ m = A_n - A_1 \]

for \( i = 1 \) to \( n - 1 \) do

if \( 0 < A_{i+1} - A_i < m \)

then \( m = A_{i+1} - A_i \)

Correctness: Assume that for some indices \( 1 \leq i < j \leq n \), in the sorted array \( A_j - A_i \) is the answer. If \( j > i + 1 \), then necessarily \( A_{i+1} - A_i \leq A_j - A_i \). As a result, it is enough to examine only adjacent entries to compute \( m \).

Complexity: The complexity of the sorting part is \( \Theta(n \log n) \) and the complexity of the scanning part is \( \Theta(n) \). The overall complexity of the algorithm is therefore \( \Theta(n \log n) \).

Remark: There is no algorithm with better complexity since it can be shown that this problem, called the element uniqueness problem or the element distinctness problem, is complexity-equivalent to sorting.
10. Let \( A = [A_1, A_2, \ldots, A_n] \) be an unsorted array of \( n \geq 4 \) distinct integers. Design an efficient algorithm that finds the first, second, and third largest integers in \( A \). What is the worst case number of comparisons made by your algorithm?

**Background:** The algorithm that finds the two largest integers has two phases. In the first phase, it runs a tournament on all \( n \) integers to find the largest integer with \( n - 1 \) comparisons in \( \lceil \log_2(n) \rceil \) rounds. In the second phase, it runs a second smaller tournament only on those \( \lceil \log_2(n) \rceil \) integers that were directly compared with the largest integer to find the second largest integer with \( \lceil \log_2(\lceil \log_2(n) \rceil) \rceil \) rounds. This is because in a tournament on \( k \) integers, any integer is directly compared with at most \( \lceil \log_2(k) \rceil \) other integers. The number of comparisons made by this algorithm is at most
\[
(n - 1) + (\lceil \log_2(n) \rceil - 1) = n + \lceil \log_2(n) \rceil - 2.
\]

**Observation:** Let \( x \) be the largest integer in \( A \) and let \( y \) be the second largest integer in \( A \). Let \( z \notin \{x, y\} \) be another integer in \( A \).
- **Case I.** \( z \) was directly compared with \( x \) but was not directly compared with \( y \): Then \( z \) must be smaller than another integer in the second tournament and therefore cannot be the third largest key.
- **Case II.** \( z \) was not directly compared with \( x \) and was not directly compared with \( y \): Then \( z \) must be smaller than another integer in the first tournament and therefore cannot be the third largest integer.
- **Case III.** \( z \) was directly compared with \( y \) in any of the two tournaments: Then \( z \) can be the third largest integer.

**Algorithm:** Run the algorithm to find the two largest integers and then run a third tournament on all the integers that were directly compared with the second largest key in both tournaments to find the third largest integer.

**Correctness:** By the above observation.

**Size of the third tournament:** There are at most \( \lceil \log_2(n) \rceil - 1 \) such integers from the first tournament and at most \( \lceil \log(\lceil \log_2(n) \rceil) \rceil \) such integers from the second tournament. Therefore, the number of comparisons made by the third tournament is at most
\[
\lceil \log_2(n) \rceil + \lceil \log_2(\lceil \log_2(n) \rceil) \rceil - 2.
\]

**Total number of comparisons made by all three tournaments:**
\[
(n - 1) + (\lceil \log n \rceil - 1) + (\lceil \log n \rceil + \lceil \log(\lceil \log n \rceil) \rceil - 2).
\]

Since \( \lfloor x \rfloor \leq x + 1 \) for a positive \( x \), it follows that the total number of comparisons made in all three tournaments is less than \( n + 2 \log n + \log \log(n) \).

**A more efficient algorithm:** For \( 1 \leq k \leq n \), there exists an algorithm that finds the \( 1^{st}, 2^{nd}, \ldots, k^{th} \) largest integers in \( A \) with at most \( n + (k - 1) \lceil \log_2(n) \rceil \) comparisons. For \( k = 3 \), this algorithm finds the three largest integers in \( A \) with at most \( n + 2 \lceil \log_2(n) \rceil \) comparisons. It is therefore a more efficient than the above 3-tournament algorithm.

**Remark:** If the task is to find the \( k \) largest integers in an array without necessarily knowing the order among them, then it can be done with a linear-time algorithm for any \( 1 \leq k \leq n \). First run the linear-time \( k \)-selection algorithm and then with one scan find the \( k - 1 \) integers that are larger than the selected integer. For \( k = \omega(n/\log(n)) \), this algorithm is more efficient than the above algorithm. However, the above algorithm also finds the order among the \( k \) largest integers.
11. For an odd \( n \geq 1 \), let \( A = [A_1, A_2, \ldots, A_n] \) be an unsorted array of \( n \) positive integers that are not necessarily distinct. A \textit{majority} is an integer that appears at least \((n+1)/2\) times in the array. Design a linear-time algorithm that finds a majority in \( A \) if \textbf{exists}.

\textbf{Observation I:} There is at most one majority. Moreover, if a majority exists and appears \( f \) times in the array, then there are at most \( f - 1 \) non-majority integers in the array.

\textbf{Proof:} \[ n - \frac{n+1}{2} = \frac{n-1}{2} = \frac{n+1}{2} - 1. \]

\textbf{Observation II:} If \( m \) is a majority then \( m \) is the median of the array.

\textbf{Proof:} Sort \( A \). Then \( A_{(n+1)/2} \) must equal \( m \) since both the number of indices smaller than the median and the number of indices greater than the median are at most \((n-1)/2\) which is strictly less than \((n+1)/2\) by the proof of Observation I.

\textbf{Algorithm:} Find the median \( m \). Scan the array to count the number of times \( m \) appears in \( A \). Let this number be \( c \). If \( c \geq (n+1)/2 \) then \( m \) is a majority in \( A \). Otherwise the array does not have a majority.

\textbf{Correctness:} By Observation II.

\textbf{Complexity:} The median can be found with a \( \Theta(n) \)-time algorithm and the array scan can be implemented in \( \Theta(n) \)-time. The total complexity is therefore \( \Theta(n) \).

\textbf{BoyerMoore majority vote algorithm:}
- Algorithm and proof: https://gregable.com/2013/10/majority-vote-algorithm-find-majority.html
- Video without a proof: https://www.youtube.com/watch?v=mO6dHeF1xk
12. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be a sorted array of \( n \geq 1 \) distinct positive integers and let \( k \) be a positive integer. Design a linear time algorithm that finds, if exist, two indices \( 1 \leq i, j \leq n \) such that \( A_i + A_j = k \).

**Observation I:** Let \( 1 \leq \ell < r \leq n \) be two indices such that \( A_\ell + A_r > k \). Then \( A_h + A_r > k \) for all \( \ell \leq h \leq r \) because \( A_h + A_r \geq A_\ell + A_r > k \). Therefore, if there are two indices \( i \) and \( j \) in the range \([\ell..r]\) such that \( A_i + A_j = k \) then \( r \) is not one of them.

**Observation II:** Let \( 1 \leq \ell < r \leq n \) be two indices such that \( A_\ell + A_r < k \). Then \( A_\ell + A_h < k \) for all \( \ell \leq h \leq r \) because \( A_\ell + A_h \leq A_\ell + A_r < k \). Therefore, if there are two indices \( i \) and \( j \) in the range \([\ell..r]\) such that \( A_i + A_j = k \) then \( \ell \) is not one of them.

**Algorithm:** Initially, \( \ell = 1 \) and \( r = n \) forming the range \([\ell..r]\). Apply the following recursive step until \( \ell > r \). In this case, return a NO answer (that is, there are no two indices \( i \) and \( j \) such that \( A_i + A_j = k \) in the array).

**Recursive step** for the range \([\ell..r]\) for which \( \ell \leq r \):
- If \( A_\ell + A_r = k \): return a YES answer for the indices \( \ell \) and \( r \).
- If \( A_\ell + A_r > k \): continue recursively with the range \([\ell..(r-1)]\). (* Observation I *)
- If \( A_\ell + A_r < k \): continue recursively with the range \([(\ell+1)..r]\). (* Observation II *)

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

**Complexity:** Let \( \Delta = r - \ell \). Initially, \( \Delta = n - 1 \). After each recursive step the value of \( \Delta \) is decreased by 1. The maximum number of recursive steps happens when the answer is NO. At this stage, \( \Delta = -1 \). Therefore there are at most \( n \) recursive steps. The total complexity is \( \Theta(n) \) since the complexity of each recursive step is \( \Theta(1) \).
13. Let \( A = [A_1, A_2, \ldots, A_n] \) be an array of \( n \geq 1 \) distinct positive integers. An inversion is a pair of indices \( 1 \leq i, j \leq n \) such that \( i < j \) but \( A_i > A_j \).

**Example:** In the array \([30, 80, 20, 40, 10]\), the pair \( i = 1 \) and \( j = 3 \) is an inversion because \( A_1 = 30 \) is greater than \( A_3 = 20 \). On the other hand, the pair \( i = 1 \) and \( j = 2 \) is not an inversion because \( A_1 = 30 \) is smaller than \( A_2 = 80 \). In this array there are 7 inversions and 3 non-inversions.

Design an efficient algorithm that counts the number of inversions in \( A \). What is the worst-case number of comparisons made by your algorithm?

**Solution:** Modify the MergeSort sorting algorithm to count the number of inversions during the merge procedures for a \( \Theta(n \log n) \) complexity.

**Example:** The array \([30, 80, 20, 40, 10]\) has 7 inversions. The left subarray \([30, 80, 20]\) has 2 inversions, the right subarray \([40, 10]\) has 1 inversion, while there are 4 inversions between integers from the left subarray and integers from the right subarray.

**Sort and count:** This recursive procedure gets as an input an array \( A \) of \( n \geq 1 \) distinct positive integers. It sorts \( A \) following the MergeSort algorithm while also counting the number of inversions in the array. The procedure returns as an output the number of inversions in the input array.

**Sort-And-Count** \((A)\)

\[
\text{if } A \text{ has one integer } \text{return}(0) \text{ else}
\]

\[
(L, c_L) = \text{Sort-And-Count}(L) \quad (R, c_R) = \text{Sort-And-Count}(R) \quad (A, c) = \text{Merge-And-Count}(L, R) \quad \text{return}(c = c_L + c_R + c)
\]

**Merge-And-Count:** This procedure gets as an input two sorted arrays \( L \) and \( R \) and returns as an output a sorted array \( A \) that contains all the integers from \( L \) and \( R \). It also returns the number of inversions containing one integer from \( L \) and one integer from \( R \).

**Merge-And-Count** \((L, R)\)

\[
\text{initialize a counter } c = 0 \quad \text{initialize } i \text{ the index of the first integer in } L \quad \text{initialize } j \text{ the index of the first integer in } R \quad \text{initialize an empty array } A
\]

\[
\text{while } L \text{ and } R \text{ are not empty}
\]

\[
\text{if } L_i < R_j \quad \text{then append } L_i \text{ to } A
\]

\[
\text{increment } i \text{ by 1}
\]

\[
\text{else append } R_j \text{ to } A
\]

\[
\text{increment } j \text{ by 1}
\]

\[
\text{increment } c \text{ by the number of remaining integers in } L
\]

\[
\text{return}(A, c)
\]

**Correctness:** In the last line of the while loop, procedure Merge-And-Count updates the number of inversions whenever an integer is moved to the left. This is the only reason to update this counter.

**Remark:** For \( 1 \leq i \leq n \), assume that \( A_i \) is the \( j^{\text{th}} \) largest number and let \( d_i = |A_i - j| \). Then, it is not true that the number of inversions is always \( (1/2) \sum_{i=1}^{n} d_j \). In the above example: \( d_1 = 2, d_2 = 3, d_3 = 1, d_4 = 0, \) and \( d_5 = 4 \) implying \( (1/2) \sum_{i=1}^{5} d_j = 5 \). However, the number of inversions in this array is 7.
14. For \( n \geq 1 \), let \( A = [A_1, A_2, \ldots, A_n] \) and \( B = [B_1, B_2, \ldots, B_n] \) be arbitrary (not necessarily sorted) arrays containing \( 2n \) distinct positive integers. Array \( A \) dominates array \( B \) if it is possible to rearrange the arrays in a way such that \( A_i > B_i \) for all \( 1 \leq i \leq n \).

Design an efficient algorithm that decides if array \( A \) dominates array \( B \). What is the worst-case number of comparisons made by your algorithm?

**Algorithm:** Sort both arrays. Scan both of them together to check if \( A_i > B_i \) for all \( 1 \leq i \leq n \).

**Complexity:** \( \Theta(n \log n) \) for sorting and \( \Theta(n) \) for scanning for an overall \( \Theta(n \log n) \) complexity.

**Correctness:** If after sorting both arrays \( A_i > B_i \) for all \( 1 \leq i \leq n \) then the sorted arrays are the proof that \( A \) dominates \( B \). It remains to show that when \( A \) dominates \( B \) then necessarily the sorted version of \( A \) dominates the sorted version of \( B \).

**Observation I:** Assume \( A \) dominates \( B \) and let \( A \) and \( B \) be rearranged to show the domination. Then for any two indices \( 1 \leq i < j \leq n \), \( A \) after swapping \( A_i \) with \( A_j \) also dominates \( B \) after swapping \( B_i \) with \( B_j \).

**Proof:** This is because in the new versions of the arrays still \( A_i > B_i \) and \( A_j > B_j \).

**Corollary:** By applying Observation I until \( A \) is sorted, it follows that there is a rearrangement showing the domination in which \( A \) is sorted in an ascending order.

**Observation II:** Assume that \( A \) dominates \( B \) and let \( A \) and \( B \) be rearranged to show the domination in which \( A \) is sorted in an ascending order. Let \( 1 \leq i < j \leq n \) be two indices for which \( B_i > B_j \). Then \( A \) dominates \( B \) after swapping \( B_i \) with \( B_j \).

**Proof:** By the assumptions, it follows that

\[
B_j < B_i < A_i < A_j
\]

Therefore swapping \( B_i \) with \( B_j \) does not affect the domination.

**Corollary II:** By applying Observation II until \( B \) is sorted, it follows that there is a rearrangement showing the domination in which both \( A \) and \( B \) are sorted in an ascending order.

**Another algorithm:** Sort both arrays into a third array \( C \) that will contain all the \( 2n \) distinct integers. While sorting, record the original index of each integer from \( A \) and \( B \). Then \( A \) dominates \( B \) if and only if for any \( 1 \leq i \leq n \), if \( C_j = A_i \) then \( j \geq 2i \).

**Correctness:** If all the \( n \) conditions hold, then for all \( 1 \leq i \leq n \), the \( i \)-th largest integer in \( A \) is greater than at least \( i \) integers from \( B \) and in particular it is greater than the \( i \)-th largest integer in \( B \). Therefore, \( A \) dominates \( B \). On the other hand, if there exists \( i \) such that \( C_j = A_i \) and \( j < 2i \), then there are not enough integers in \( A \) to dominate the \( n - i \) largest integers in \( B \).

**Complexity:** \( \Theta(2n \log(2n)) = \Theta(n \log n) \) for sorting. Note that the \( \Theta(n) \) tests of the type \( j \geq 2i \) are not considered as comparisons.
15. An array $A$ of $n \geq 1$ distinct positive integers is a hill array if there exists an index $1 \leq i \leq n$, called the hilltop, such that: $A_1 < A_2 < \cdots < A_{i-1} < A_i > A_{i+1} > \cdots > A_{n-1} > A_n$. Note that when the hilltop is $i = 1$ then $A$ is sorted in a descending order and when the hilltop is $i = n$ then $A$ is sorted in an ascending order.

Design a comparison-based algorithm that sorts hill arrays in an ascending order with a linear number ($O(n)$) of comparisons.

Remark: The input is guaranteed to be a hill array but the hilltop index of the array is not known.

Algorithm: First find the hilltop index $i$. Then merge the ascending-ordered sorted subarray $A_1, \ldots, A_i$ with the descending-ordered sorted subarray $A_{i+1}, \ldots, A_n$.

Complexity: The complexity of the merge phase is $\Theta(n)$. The hilltop index can be found with a binary search whose complexity is $\Theta(\log n)$. However, since the complexity of the merge phase is already $\Theta(n)$, a simpler sequential search whose complexity is also $\Theta(n)$ can find the hilltop index. In either case, the overall complexity is $\Theta(n)$.

Finding the hilltop index: The array is trivially sorted when $n = 1$. Assume that $n \geq 2$. There are three cases:


The merge procedure: If $i = n$ return the array as is and if $i = 1$ return the reverse of the array. Note that this can be done with $\lfloor n/2 \rfloor$ swaps between pairs of integers. Otherwise, $1 < i < n$ for $n \geq 3$.

- Let $L[1..i] = A[1..i]$ contain the first $i$ integers from $A$. That is, $L_j = A_j$ for $1 \leq j \leq i$.
- Let $R[(i+1)..n] = A[(i+1)..n]$ contain the last $n-i$ integers from $A$. That is, $R_j = A_j$ for $i+1 \leq j \leq n$.

Note that array $L$ of length $i$ is sorted in an ascending order and the array $R$ of length $n-i$ is sorted in a descending order. Apply the following merge procedure.

$\text{Merge}(L, R)$

initialize $\ell = 1$ the index of the first (and smallest) integer in $L$
initialize $r = n$ the index of the last (and smallest) integer in $R$
initialize $j = 1$ the index of the first entry in the new sorted array $A$
while $\ell \leq i$ and $r \geq i + 1$
    if $L_\ell < R_r$
        then $A_j = L_\ell$ and increment $\ell$ by 1
        else $A_j = R_r$ and decrement $r$ by 1
        increment $j$ by 1
    Case ($\ell \leq i$ and $r < i + 1$): $A[j..n] = L[\ell..i]$
    Case ($\ell > i$ and $r \geq i + 1$): $A[j..n] = R[r..(i + 1)]$

Note that in the last line of the procedure, the remaining $r - i$ integers from the array $R$ are copied in a reversed order into the remaining $r - i$ entries of the array $A$. 
16. Design an algorithm that finds the median of 5 distinct keys with at most 6 comparisons.

**Algorithm:** Let the keys be $A_1$, $A_2$, $A_3$, $A_4$, and $A_5$. For $1 \leq i < j \leq 5$, denote by $< A_i : A_j >$ the operation that compares $A_i$ with $A_j$ and if $A_i > A_j$ swap them.

The first three comparisons are $< A_2 : A_4 >$, $< A_3 : A_5 >$, and $< A_4 : A_5 >$. If $A_4 > A_5$ then also swap $A_2$ with $A_3$.

At this stage, $A_5$ is greater than $A_3$ and $A_4$ because it was directly compared with them. $A_5$ is also greater than $A_2$ because $A_4$ is greater than $A_2$. Since the median is greater than exactly two keys, it follows that $A_5$ cannot be the median. Observe that $A_5$ is not necessarily the maximum key because it can be smaller than $A_1$.

The next two comparisons are $< A_1 : A_3 >$ and then $< A_3 : A_4 >$. If $A_3 > A_4$ then also swap $A_1$ with $A_2$.

At this stage, the following relationships are known:
- $A_4$ is greater than $A_3$ and $A_2$ because it was directly compared with them. $A_4$ is also greater than $A_1$ because $A_3$ is greater than $A_1$. Since the median is greater than exactly two keys, it follows that $A_4$ cannot be the median.
- $A_1$ is smaller than $A_3$, $A_4$, and $A_5$. Since the median is smaller than exactly two keys, it follows that $A_1$ cannot be the median.
- Both $A_2$ and $A_3$ are smaller than both $A_4$ and $A_5$ (but the order between $A_4$ and $A_2$ is not always known). Therefore, the median is either $A_2$ or $A_3$.

The 6th and last comparison is $< A_2 : A_3 >$. After this comparison, $A_3$ is the median.

**Example I:**

Initial order : 21, 13, 5, 3, 8

$< A_2 : A_4 > < 13, 3 > \implies 21, 3, 5, 13, 8$
$< A_3 : A_5 > < 5, 8 > \implies 21, 3, 5, 13, 8$
$< A_4 : A_5 > < 13, 8 > \implies 21, 5, 3, 8, 13$ (*3 and 5 also exchange places*)
$< A_1 : A_3 > < 21, 3 > \implies 3, 5, 21, 8, 13$
$< A_3 : A_4 > < 21, 8 > \implies 5, 3, 8, 21, 13$ (*3 and 5 also exchange places*)
$< A_2 : A_3 > < 3, 8 > \implies 5, 3, 8, 21, 13 \implies 8$ is the median

**Example II:**

Initial order : 13, 21, 8, 5, 3

$< A_2 : A_4 > < 21, 5 > \implies 13, 5, 8, 21, 3$
$< A_3 : A_5 > < 8, 3 > \implies 13, 5, 3, 21, 8$
$< A_4 : A_5 > < 21, 8 > \implies 13, 3, 5, 8, 21$ (*3 and 5 also exchange places*)
$< A_1 : A_3 > < 13, 5 > \implies 5, 3, 13, 8, 21$
$< A_3 : A_4 > < 13, 8 > \implies 3, 5, 8, 13, 21$ (*3 and 5 also exchange places*)
$< A_2 : A_3 > < 5, 8 > \implies 3, 5, 8, 21, 13 \implies 8$ is the median
Illustrating the algorithm: The evolution of the algorithm appears in the diagram below.

- In each cluster of five keys, if two keys are connected by a solid line then the top one is the larger key.
- The two keys that are connected by a dotted line are those that are compared for the next layer.
- The top layer is the knowledge status after the 3\textsuperscript{rd} comparisons.
- The knowledge illustrated in the second layer depends on the 4\textsuperscript{th} comparison between $A_1$ and $A_3$.
- The knowledge illustrated in the third layer depends on the 5\textsuperscript{th} comparison between $A_3$ and $A_4$.
- Finally, after the 6\textsuperscript{th} and last comparison between $A_2$ and $A_3$, the median is $A_3$ as illustrated in the bottom layer.
17. Design an algorithm that sorts 5 distinct keys with at most 7 comparisons.

**Algorithm:** Let the keys be \( A_1, A_2, A_3, A_4, \) and \( A_5 \). For \( 1 \leq i < j \leq 5 \), denote by \( A_i : A_j \) the operation that compares \( A_i \) with \( A_j \) and if \( A_i > A_j \) it swaps them.

- The first three comparisons are \( A_2 : A_4 \), \( A_3 : A_5 \), and \( A_4 : A_5 \). If \( A_4 > A_5 \) then also swap \( A_2 \) with \( A_3 \).
- At this stage, \( A_5 \) is greater than \( A_3 \) and \( A_4 \) because it was directly compared with them. \( A_5 \) is also greater than \( A_2 \) because \( A_4 \) is greater than \( A_2 \). As a result \( A_2 < A_4 < A_5 \) is a chain of length three. Note that \( A_5 \) is not necessarily the maximum key since it can be smaller than \( A_1 \).
- The goal of the next two comparisons is to insert \( A_1 \) into the chain \( A_2 < A_4 < A_5 \) of length three to create a the chain \( A_1 < A_2 < A_4 < A_5 \) of length four.
- The fourth comparisons is \( A_1 : A_4 \).
  - If \( A_1 < A_4 \), then the fifth comparison is \( A_1 : A_2 \).
  - If \( A_1 > A_4 \), then after \( A_1 \) is swapped with \( A_4 \), also swap the new \( A_1 \) with \( A_2 \) so the new \( A_2 \) is greater than the new \( A_1 \). Then the fifth comparison is \( A_4 : A_5 \).
- At this stage \( A_1 < A_2 < A_4 < A_5 \) is a chain of length four. Note that if in the fifth comparison \( A_4 < A_5 \), then the only information about \( A_3 \) is that it is smaller than \( A_5 \) and if \( A_5 < A_4 \) then \( A_3 \) is smaller than both \( A_4 \) and \( A_5 \).
- The sixth comparison is \( A_2 : A_3 \).
  - If \( A_3 < A_2 \) then the seventh and the last comparison is \( A_1 : A_2 \).
  - If \( A_3 > A_2 \) then the sorting is done if already \( A_3 < A_4 \). Otherwise the seventh and the last comparison is \( A_3 : A_4 \).
- The final order is \( A_1 < A_2 < A_3 < A_4 < A_5 \).

**Example:**

<table>
<thead>
<tr>
<th>Initial order</th>
<th>21, 13, 5, 3, 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_2 : A_4 )</td>
<td>( 13, 3 )</td>
</tr>
<tr>
<td>( A_3 : A_5 )</td>
<td>( 5, 8 )</td>
</tr>
<tr>
<td>( A_4 : A_5 )</td>
<td>( 13, 8 )</td>
</tr>
<tr>
<td>( A_1 : A_4 )</td>
<td>( 21, 8 )</td>
</tr>
<tr>
<td>( A_4 : A_5 )</td>
<td>( 21, 13 )</td>
</tr>
<tr>
<td>( A_2 : A_3 )</td>
<td>( 8, 3 )</td>
</tr>
<tr>
<td>( A_1 : A_2 )</td>
<td>( 5, 3 )</td>
</tr>
</tbody>
</table>

**BubbleSort:** The maximum is found 4 times. There are 4 comparisons in the first stage, 3 in the second, 2 in the third, and the final comparison compares the two smallest keys. The total number of comparisons is 10.

**MergeSort:** 3 comparisons are required to create a chain of length 3 and 1 comparison is required to create a chain of length 2. 4 more comparisons are required to merge the two chains. The total number of comparisons in the worst-case is 8.

**Optimality:** The algorithm is optimal due to the \( \lceil \log(n!) \rceil \) general lower bound for \( n \) keys:

\[
\lceil \log_2(5!) \rceil = \lceil \log_2(120) \rceil = 7.
\]
Illustrating the algorithm: The evolution of the algorithm appears in the diagram below.
- In each cluster of five keys, if two keys are connected by a solid line then the top one is the larger key.
- The two keys that are connected by a dotted line are those that are compared for the next layer.
- The top layer is the knowledge status after the 3\textsuperscript{rd} comparisons.
- The knowledge illustrated in the second layer depends on the 4\textsuperscript{th} comparison between $A_1$ and $A_4$.
- The knowledge illustrated in the third layer depends on the 5\textsuperscript{th} comparison either between $A_4$ and $A_5$ or between $A_1$ and $A_2$.
- The knowledge illustrated in the fourth layer depends on the 6\textsuperscript{th} comparison between $A_2$ and $A_3$.
- Finally, after the 7\textsuperscript{th} comparison between $A_1$ and $A_2$ or $A_3$ and $A_4$, the 5 keys are sorted as illustrated in the bottom layer.
18. **Story:** There are \( n \) pancakes all of different sizes that are stacked on top of each other. It is allowed to slip a flipper under one of the pancakes and flip over the whole sack above the flipper. The goal is to arrange pancakes according to their size with the biggest at the bottom.

**Model:** Let \( A \) be an array of size \( n \geq 1 \) containing the numbers \( 1, \ldots, n \) in any order (\( A \) represents an arbitrary permutation). For any \( 2 \leq i \leq n \), the \( F_i \) operation (flip) is to reverse the prefix of size \( i \) of the array.

**Example I:** \( F_5([1, 2, 3, 4, 5, 6, 7, 8]) = [5, 4, 3, 2, 1, 6, 7, 8] \).

**Example II:** \( F_3([3, 2, 1, 4, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8] \).

**Problem:** Sort the array in an ascending order with as few as possible flips.

**Remark:** In the comparison model, array operations were free and the optimization goal is to minimize the number of comparisons. In the pancake model, comparisons are free since the final location of each pancake is known. The optimization goal is to minimize the number of array (flip) operations.

(a) Sort the array \([8, 7, 6, 5, 1, 2, 3, 4]\) with as few as possible flips. Specify the list of flips that would sort the array.

(b) Sort the array \([8, 6, 4, 2, 1, 3, 5, 7]\) with as few as possible flips. Specify the list of flips that would sort the array.

(c) Describe an efficient algorithm that sorts any permutation-array of size \( n \geq 1 \) with flips. How many flips, as a function of \( n \), are made by your algorithm in the worst-case?

Use words to describe the algorithm and be as accurate as you can with your worst-case complexity. Justify the correctness and complexity of your algorithm.

---

**An online resource:** [https://en.wikipedia.org/wiki/Pancake_sorting](https://en.wikipedia.org/wiki/Pancake_sorting)

(a) Sort the array \([8, 7, 6, 5, 1, 2, 3, 4]\) with as few as possible flips.

**Solution:** The following 2 flips sort the array.

\[
F_4([8, 7, 6, 5, 1, 2, 3, 4]) = [4, 3, 2, 1, 5, 6, 7, 8] \\
F_4([4, 3, 2, 1, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8]
\]

(b) Sort the array \([8, 6, 4, 2, 1, 3, 5, 7]\) with as few as possible flips.

**Solution:** The following 7 flips sort the array.

\[
F_3([8, 6, 4, 2, 1, 3, 5, 7]) = [7, 5, 3, 1, 2, 4, 6, 8] \\
F_7([7, 5, 3, 1, 2, 4, 6, 8]) = [6, 4, 2, 1, 3, 5, 7, 8] \\
F_5([6, 4, 2, 1, 3, 5, 7, 8]) = [5, 3, 1, 2, 4, 6, 7, 8] \\
F_5([5, 3, 1, 2, 4, 6, 7, 8]) = [4, 2, 1, 3, 5, 6, 7, 8] \\
F_4([4, 2, 1, 3, 5, 6, 7, 8]) = [3, 1, 2, 4, 5, 6, 7, 8] \\
F_3([3, 1, 2, 4, 5, 6, 7, 8]) = [2, 1, 3, 4, 5, 6, 7, 8] \\
F_2([2, 1, 3, 4, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8]
\]
(c) Describe an efficient algorithm that sorts any permutation-array of size \( n \geq 1 \) with flips. How many flips, as a function of \( n \), are made by your algorithm in the worst-case?

**Algorithm:** There are \( n - 1 \) phases each except the last is composed of at most two flips. Starting with \( j = n \) down to \( j = 2 \), the goal in phase \( j \) is to bring the \( j \)th largest pancake to its correct position.

**Phase** \( 2 \leq j \leq n \): Assume pancake \( j \) is in position \( k \). If \( k = j \) there are no flips in this phase. Otherwise, if \( k > 1 \) then the flip \( F_k \) brings the \( j \)th pancake to the first position. Next, the flip \( F_j \) brings the \( j \)th pancake to its final correct position. Note that in the last phase (\( j = 2 \)), the flip \( F_2 \) is applied only if the first two pancakes are not in their correct position.

**Correctness:** It is possible to prove by induction that for \( 2 \leq j \leq n \), after phase \( j \), pancakes \( j, j+1, \ldots, n \) are in their correct positions. As a result, only pancakes \( 1, 2, \ldots, j - 1 \) might be in the wrong position. Therefore, after the last phase, \( j = 2 \), all the pancakes including the smallest one are in their correct position.

**Complexity:** At most \( 2n - 3 \) flips. Because each of the first \( n - 2 \) phases applies at most 2 flips while the last phase applies at most 1 flip.

**Worst Case for an even** \( n \): The array \([2, n, n-2, \ldots, 4, 3, 5, \ldots, n-1, 1]\) forces the algorithm to apply the following \( 2n - 3 \) flips:

\[ F_2, F_n, F_2, F_{n-1}, F_2, F_{n-2}, \ldots, F_2, F_4, F_2, F_3, F_2 \]

**Example:** The array \([2, 8, 6, 4, 3, 5, 7, 1]\) forces the algorithm to apply the following 13 flips:

\[ F_2, F_8, F_2, F_7, F_2, F_6, F_2, F_5, F_2, F_4, F_2, F_3, F_2 \]

**Worst Case for an odd** \( n \): The array \([1, n, n-2, \ldots, 3, 4, 6, \ldots, n-1, 2]\) forces the algorithm to apply the following \( 2n - 3 \) flips:

\[ F_2, F_n, F_2, F_{n-1}, F_2, F_{n-2}, \ldots, F_2, F_4, F_2, F_3, F_2 \]

**Example:** The array \([1, 7, 5, 3, 4, 6, 2]\) forces the algorithm to apply the following 11 flips:

\[ F_2, F_7, F_2, F_6, F_2, F_5, F_2, F_4, F_2, F_3, F_2 \]

**The algorithm is not always optimal:** Consider the array \([2, 4, 3, 1]\). The algorithm sorts it with the following 5 flips:

\[
\begin{align*}
F_2([2, 4, 3, 1]) &= [4, 2, 3, 1] \\
F_4([4, 2, 3, 1]) &= [1, 3, 2, 4] \\
F_2([1, 3, 2, 4]) &= [3, 1, 2, 4] \\
F_3([3, 1, 2, 4]) &= [2, 1, 3, 4] \\
F_2([2, 1, 3, 4]) &= [1, 2, 3, 4]
\end{align*}
\]

While the following 4 flips also sort the array

\[
\begin{align*}
F_4([2, 4, 3, 1]) &= [1, 3, 4, 2] \\
F_3([1, 3, 4, 2]) &= [4, 3, 1, 2] \\
F_4([4, 3, 1, 2]) &= [2, 1, 3, 4] \\
F_2([2, 1, 3, 4]) &= [1, 2, 3, 4]
\end{align*}
\]