Algorithms

Assignment Solutions: Sorting
1. 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

\hspace{1em} if $0 < A_{i+1} - A_i < m$

\hspace{2em} 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.
2. 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 $A_i > A_j$ 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 both $A_3$ 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 another chain $A_1 < A_2 < A_4 < A_5$ of length four.
- The fourth comparison 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>
<th>$&lt; A_2 : A_4 &gt;$= $&lt; 13, 3 &gt;$ $\implies$ 21, 3, 5, 13, 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$&lt; A_3 : A_5 &gt;$= $&lt; 5, 8 &gt;$ $\implies$ 21, 3, 5, 13, 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$&lt; A_4 : A_5 &gt;$= $&lt; 13, 8 &gt;$ $\implies$ 21, 5, 3, 8, 13 (<em>3 and 5 also swap places</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$&lt; A_1 : A_4 &gt;$= $&lt; 21, 8 &gt;$ $\implies$ 5, 8, 3, 21, 13 (<em>5 and 8 also swap places</em>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$&lt; A_4 : A_5 &gt;$= $&lt; 21, 13 &gt;$ $\implies$ 5, 8, 3, 13, 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$&lt; A_2 : A_3 &gt;$= $&lt; 8, 3 &gt;$ $\implies$ 5, 3, 8, 21, 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$&lt; A_1 : A_2 &gt;$= $&lt; 5, 3 &gt;$ $\implies$ 3, 5, 8, 21, 13</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:

$$\lceil \log_2(5!) \rceil = \lceil \log_2(120) \rceil = 7.$$
3. Let \( A = [A_1, A_2, \ldots, A_n] \) be an array of \( n \geq 1 \) distinct positive integers. An \textit{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 \textit{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 \textit{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.

\[ \text{Sort-And-Count}(A) \]

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

\[ \text{divide } A \text{ into two almost equal-size arrays } L \text{ and } R \]

\[ (L, c_L) = \text{Sort-And-Count}(L) \]

\[ (R, c_R) = \text{Sort-And-Count}(R) \]

\[ (A, c) = \text{Merge-And-Count}(L, R) \]

\[ \text{return } (c = c_L + c_R + c) \]

**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 \( R \).

\[ \text{Merge-And-Count}(L, R) \]

\[ \text{initialize a counter } c = 0 \]

\[ \text{initialize } i \text{ the index of the first integer in } L \]

\[ \text{initialize } j \text{ the index of the first integer in } R \]

\[ \text{initialize an empty array } A \]

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

\[ \text{if } L_i < R_j \]

\[ \text{then append } L_i \text{ to } A \]

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

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

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

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

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

**Correctness:** In the last line of the while loop, procedure \textit{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 \( j \)-th largest number and let \( d_i = |A_i - j| \). Then, it is not true that the number of inversions is always \( (1/2) \sum_{i=1}^{n} d_i \). In the above example: \( d_1 = 2, \)
\( d_2 = 3, d_3 = 1, d_4 = 0, \) and \( d_5 = 4 \) implying \( (1/2) \sum_{i=1}^{5} d_i = 5 \). However, the number of inversions in this array is 7.
4. 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 if \( A \) dominates \( B \) with different rearrangements of the arrays then the sorting versions of the arrays also show that \( A \) dominates \( B \). Equivalently, 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.
5. 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.

**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 $A$. 
6. **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.

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

\[
F_8([8, 7, 6, 5, 1, 2, 3, 4]) = [4, 3, 2, 1, 5, 6, 7, 8] \\
F_4([4, 3, 2, 1, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8]
\]

(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.

\[
F_8([8, 6, 4, 2, 1, 3, 5, 7]) = [7, 5, 3, 1, 2, 4, 6, 8] \\
F_7([7, 5, 3, 1, 2, 4, 6, 8]) = [6, 4, 2, 1, 3, 5, 7, 8] \\
F_6([6, 4, 2, 1, 3, 5, 7, 8]) = [5, 3, 1, 2, 4, 6, 7, 8] \\
F_5([5, 3, 1, 2, 4, 6, 7, 8]) = [4, 2, 1, 3, 5, 6, 7, 8] \\
F_4([4, 2, 1, 3, 5, 6, 7, 8]) = [3, 1, 2, 4, 5, 6, 7, 8] \\
F_3([3, 1, 2, 4, 5, 6, 7, 8]) = [2, 1, 3, 4, 5, 6, 7, 8] \\
F_2([2, 1, 3, 4, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8]
\]
(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:

\[
F_2([2, 4, 3, 1]) = [4, 2, 3, 1] \\
F_4([4, 2, 3, 1]) = [1, 3, 2, 4] \\
F_2([1, 3, 2, 4]) = [3, 1, 2, 4] \\
F_3([3, 1, 2, 4]) = [2, 1, 3, 4] \\
F_2([2, 1, 3, 4]) = [1, 2, 3, 4]
\]

While the following 4 flips also sort the array:

\[
F_4([2, 4, 3, 1]) = [1, 3, 4, 2] \\
F_3([1, 3, 4, 2]) = [4, 3, 1, 2] \\
F_4([4, 3, 1, 2]) = [2, 1, 3, 4] \\
F_2([2, 1, 3, 4]) = [1, 2, 3, 4]
\]