Algorithms

Assignment Solutions: Sorting

1. Let $A[1], A[2], \ldots, A[n]$ be an array containing $n$ very large positive integers.

Describe an efficient algorithm to find the minimum positive difference between any two integers in the array.

What is the complexity of 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 $A[i] \neq A[j]$.

**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:


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 in the sorted array $A[j] - A[i]$ is the answer for some indices $1 \leq i < j \leq n$. If $j > i + 1$ then there exists an index $i < k < j$ such that both $A[k] - A[i]$ and $A[j] - A[k]$ are less or equal to $A[j] - A[i]$. Therefore, the answer exists for an index $1 \leq i < n$ and it is $A[i + 1] - A[i]$.

**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 or the element distinctness problem, is complexity-equivalent to sorting.\(^1\)

\(^1\)The proof is beyond the scope of this course
2. Design an efficient algorithm to sort 5 distinct keys.

**Solution:** It is possible to sort 5 keys with at most 7 comparisons.

**Algorithm:** Let the keys be \( A_1, A_2, A_3, A_4, \) and \( A_5 \). For \( 1 \leq i \neq j \leq 5 \), denote by \( < A_i : A_j > \) the operation that compares \( A_i \) with \( A_j \) and if \( A_i > A_j \) exchanges 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 exchange \( A_2 \) with \( A_3 \).
- At this stage, \( A_5 \) is greater than \( A_3 \) and \( A_4 \) because it was directly compared with them. It is 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 next two comparisons are \( < A_1 : A_4 > \) and then \( < A_4 : A_5 > \). If in the fourth comparison \( A_1 > A_4 \), then after \( A_1 \) is exchanged with \( A_4 \), also exchange the new \( A_1 \) with \( A_2 \) so the new \( A_2 \) is greater than the new \( A_1 \).
- At this stage \( A_1 < A_2 < A_4 < A_5 \) is a chain of length 4. 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 I:**

Initial order : \( 21, 13, 5, 3, 8 \)
\( < A_2 : A_4 >/= 13, 3 \) \( \Rightarrow \) \( 21, 3, 5, 13, 8 \)
\( < A_3 : A_5 >/= 5, 8 \) \( \Rightarrow \) \( 21, 3, 5, 13, 8 \)
\( < A_4 : A_5 >/= 13, 8 \) \( \Rightarrow \) \( 21, 5, 3, 8, 13 \) (*3 and 5 also exchange places *)
\( < A_1 : A_4 >/= 21, 8 \) \( \Rightarrow \) \( 5, 8, 3, 21, 13 \) (*5 and 8 also exchange places *)
\( < A_4 : A_5 >/= 21, 13 \) \( \Rightarrow \) \( 5, 8, 3, 21, 13 \)
\( < A_2 : A_3 >/= 8, 3 \) \( \Rightarrow \) \( 5, 3, 8, 21, 13 \)
\( < A_1 : A_2 >/= 5, 3 \) \( \Rightarrow \) \( 3, 5, 8, 21, 13 \)

**Example II:**

Initial order : \( 13, 21, 8, 5, 3 \)
\( < A_2 : A_4 >/= 21, 5 \) \( \Rightarrow \) \( 13, 5, 8, 21, 3 \)
\( < A_3 : A_5 >/= 8, 3 \) \( \Rightarrow \) \( 13, 5, 3, 21, 8 \)
\( < A_4 : A_5 >/= 21, 8 \) \( \Rightarrow \) \( 13, 3, 5, 8, 21 \) (*3 and 5 also exchange places *)
\( < A_1 : A_4 >/= 13, 8 \) \( \Rightarrow \) \( 3, 8, 5, 13, 21 \) (*3 and 8 also exchange places *)
\( < A_4 : A_5 >/= 13, 21 \) \( \Rightarrow \) \( 3, 8, 5, 13, 21 \)
\( < A_2 : A_3 >/= 8, 5 \) \( \Rightarrow \) \( 3, 5, 8, 21, 13 \)
\( < A_1 : A_2 >/= 3, 5 \) \( \Rightarrow \) \( 3, 5, 8, 21, 13 \)

**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:
\[ \log_2(5!) = \log_2(120) = 7. \]
3. Let $A = A[1], \ldots, A[n]$ be an array of $n$ distinct positive integers (the value of these integers could be very very large). An inversion is a pair of indices $i$ and $j$ such that $i < j$ but $A[i] > A[j]$.

For example in the array $[30000, 80000, 20000, 40000, 10000]$, the pair $i = 1$ and $j = 3$ is an inversion because $A[1] = 30000$ is greater than $A[3] = 20000$. On the other hand, the pair $i = 1$ and $j = 2$ is not an inversion because $A[1] = 30000$ is smaller than $A[2] = 80000$. In this array there are 7 inversions and 3 non-inversions.

Describe an efficient algorithm that counts the number of inversions in any array. What is the running time of your algorithm?

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

**Example:** The array $[30000, 80000, 20000, 40000, 10000]$ has 7 inversions. The left subarray $[30000, 80000, 20000]$ has 2 inversions, the right subarray $[40000, 10000]$ has 1 inversion, while there are 4 inversions between numbers from the left subarray and numbers from the right subarray.

**Sort and count:** This recursive procedure gets as an input an array with $n$ distinct positive integers. It sorts the array $A$ following the MergeSort algorithm while also counts the number of inversions in the array. The procedure returns as an output the number of inversions in the input array.

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

- if $A$ has one integer return (0)
- else divide $A$ into two almost equal-size arrays $L$ and $R$
  - $(L, c_L) = \text{Sort-And-Count}(L)$
  - $(R, c_R) = \text{Sort-And-Count}(R)$
  - $(A, c) = \text{Merge-And-Count}(L, R)$
  - return $(c = c_L + c_R + c)$

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

- initialize a counter $c = 0$
- initialize $i$ the index of the first integer in $L$
- initialize $j$ the index of the first integer in $R$
- initialize an empty array $A$
- while $L$ and $R$ are not empty
  - if $L[i] < R[j]$
    - then append $L[i]$ to $A$
    - increment $i$ by 1
  - else append $R[j]$ to $A$
    - increment $j$ by 1
    - increment $c$ by the number of remaining integers in $L$
- 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 $j$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.
4. 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 $1 \leq i \leq n$. Describe an efficient algorithm that decides if array $A$ dominates array $B$. What is the complexity of 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 $B$.

**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 exchanging $A[i]$ with $A[j]$ also dominates $B$ after exchanging $B[i]$ with $B[j]$.

**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. Let $1 \leq i < j \leq n$ be two indices for which $B[i] > B[j]$. Then $A$ dominates $B$ after exchanging $B[i]$ with $B[j]$.

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


Therefore exchanging $B[i]$ 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 increasingly sorted.

**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 $j \geq 2i$.

**Complexity:** $\Theta(2n \log(2n)) = \Theta(n \log n)$ for sorting and $\Theta(n)$ for testing the new locations of the integers from $A$ for an overall $\Theta(n \log n)$ complexity.

**Correctness:** If all the $n$ conditions hold, then for all $1 \leq j \leq n$ the $i$th largest integer in $A$ is greater than at least $i$ integers from $B$ and in particular 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 $B$. 

4
5. An array $A$ of $n \geq 1$ distinct positive integers is bitonic if there exists an index $1 \leq i \leq n$ such that:


Describe an algorithm that uses a linear number ($O(n)$) of comparisons to sort bitonic arrays with very large integers (you cannot use radix-sort like algorithms). Note that if the array is bitonic the index $i$ is not known.

**Definition:** Call the $A[i-1] < A[i] > A[i+1]$ index or the $A[i-1] > A[i] < A[i+1]$ index the bitonic index. Note that if the bitonic index is 1 than the array is sorted in a descending order and if the bitonic index is $n$ than the array is sorted in an ascending order.

**Algorithm:** First find the bitonic index $i$. Then merge the sorted subarray $A[1], \ldots, A[i]$ with the sorted subarray $A[i+1], \ldots, A[n]$.

**Complexity:** The complexity of the merge phase is $\Theta(n)$. The bitonic 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 bitonic index. In either case, the overall complexity is $\Theta(n)$.

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


Note that the array is guaranteed to be bitonic and therefore the last two cases would find the one bitonic index.

**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 $\lceil n/2 \rceil$ exchanges between pairs of integers. Otherwise, $n \geq 3$ and $1 < i < n$. Let $L[1..i] = A[1..i]$ contain the first $i$ integers from $A$ and let $R[1..(n-i)] = A[(i+1)..n]$ contain the last $n-i$ integers from $A$. If $L[1] > L[2]$ then reverse $L$ and if $R[1] > R[2]$ then reverse $R$. Again note that this can be done with $\lceil i/2 \rceil$ and $\lceil (n-i)/2 \rceil$ exchanges respectively. At this stage, $L$ of length $i$ and $R$ of length $n-i$ are two arrays sorted in an ascending order. Apply the following merge procedure.

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

1. **initialize** $\ell = 1$ the index of the first integer in $L$
2. **initialize** $r = 1$ the index of the first integer in $R$
3. **initialize** $j = 1$ the index of the first entry in the new sorted array $A$
4. while $\ell \leq i$ and $r \leq n-i$
   1. if $L[\ell] < R[r]$ then $A[j] = L[\ell]$ and increment $\ell$ by 1
   2. else $A[j] = R[r]$ and increment $r$ by 1
   3. increment $j$ by 1
5. Case ($\ell > i$ and $r \leq n-i$): $A[\ell..n] = R[r..(n-i)]$
6. Case ($\ell \leq i$ and $r > n-i$): $A[\ell..n] = L[\ell..i]$
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 purpose is to arrange pancakes according to their size with the biggest at the bottom.

**Model:** Let \( A \) be an array of size \( n \) 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 with as few as possible flips.

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

\[
F_8([8, 6, 4, 2, 1, 3, 5, 7]) = [7, 5, 3, 1, 2, 4, 6, 8] \\
F_7([7, 5, 3, 1, 2, 4, 6, 8]) = [6, 4, 2, 1, 3, 5, 7, 8] \\
F_6([6, 4, 2, 1, 3, 5, 7, 8]) = [5, 3, 1, 2, 4, 6, 7, 8] \\
F_5([5, 3, 1, 2, 4, 6, 7, 8]) = [4, 2, 1, 3, 5, 6, 7, 8] \\
F_4([4, 2, 1, 3, 5, 6, 7, 8]) = [3, 1, 2, 4, 5, 6, 7, 8] \\
F_3([3, 1, 2, 4, 5, 6, 7, 8]) = [2, 1, 3, 4, 5, 6, 7, 8] \\
F_2([2, 1, 3, 4, 5, 6, 7, 8]) = [1, 2, 3, 4, 5, 6, 7, 8]
\]
Describe an (efficient) algorithm to sort any array of size \( n \) with flips. How many flips, as a function of \( n \), your algorithm uses in the worst-case?

**Algorithm:** There are \( n - 1 \) phases each except the last is composed of at most two flips. In phase \( 1 \leq i < n - 1 \), let \( j = n + 1 - i \). The goal is to bring the \( j \)th pancake to its correct position.

**Phase** \( 1 \leq j \leq n - 1 \): 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.

**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: \( F_2, F_n, F_2, F_{n-1}, \ldots, F_2, F_3, F_2 \).

**Example:** The array \([2, 8, 6, 4, 3, 5, 7, 1]\) forces the algorithm to apply the following 13 flips: \( F_2, F_8, F_2, F_7, \ldots, F_2, F_3, F_2 \).

**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: \( F_2, F_n, F_2, F_{n-1}, \ldots, F_2, F_3, F_2 \).

**Example:** The array \([1, 7, 5, 3, 4, 6, 2]\) forces the algorithm to apply the following 11 flips: \( F_2, F_7, F_2, F_6, \ldots, F_2, F_3, F_2 \).

**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

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