Algorithms

Assignment Solutions: Order Statistics
1. 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 $A$. Then 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.
2. 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 $\lceil \log_2(n) \rceil - 1$ comparisons in $\lceil \log_2(\lceil \log_2(n) \rceil) \rceil$ 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 $z \not\in \{x, y\}$ 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

$$\lceil \log_2(n) \rceil + \lceil \log(\lceil \log_2(n) \rceil) \rceil - 2.$$

**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 $[x] \leq x + 1$ 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 $1^{\text{st}}, 2^{\text{nd}}, \ldots, k^{\text{th}}$ 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.
3. 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. Note that \( A_5 \) is not necessarily the maximum key since it can be smaller than \( A_1 \).

The next two comparisons are \(< A_1 : A_3 >\) 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 \( A_5 \) is not always known). Therefore, the median is either \( A_2 \) or \( A_3 \).

The 6\(^{th}\) and last comparison is \(< A_2 : A_3 >\). After this comparison, \( A_3 \) is the median.

**Example I:**

<table>
<thead>
<tr>
<th>Initial order</th>
<th>21,13,5,3,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt; A_2 : A_4 &gt;)</td>
<td>=&gt; 21,3,5,13,8</td>
</tr>
<tr>
<td>(&lt; A_3 : A_5 &gt;)</td>
<td>=&gt; 21,3,5,13,8</td>
</tr>
<tr>
<td>(&lt; A_4 : A_5 &gt;)</td>
<td>=&gt; 21,5,3,8,13 (* 3 and 5 also exchange places *)</td>
</tr>
<tr>
<td>(&lt; A_1 : A_3 &gt;)</td>
<td>=&gt; 3,5,21,8,13</td>
</tr>
<tr>
<td>(&lt; A_3 : A_4 &gt;)</td>
<td>=&gt; 5,3,8,21,13 (* 3 and 5 also exchange places *)</td>
</tr>
<tr>
<td>(&lt; A_2 : A_3 &gt;)</td>
<td>=&gt; 5,3,8,21,13 ➞ 8 is the median</td>
</tr>
</tbody>
</table>

**Example II:**

<table>
<thead>
<tr>
<th>Initial order</th>
<th>13,21,8,5,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt; A_2 : A_4 &gt;)</td>
<td>=&gt; 13,5,8,21,3</td>
</tr>
<tr>
<td>(&lt; A_3 : A_5 &gt;)</td>
<td>=&gt; 13,5,3,21,8</td>
</tr>
<tr>
<td>(&lt; A_4 : A_5 &gt;)</td>
<td>=&gt; 13,3,5,8,21 (* 3 and 5 also exchange places *)</td>
</tr>
<tr>
<td>(&lt; A_1 : A_3 &gt;)</td>
<td>=&gt; 5,3,13,8,21</td>
</tr>
<tr>
<td>(&lt; A_3 : A_4 &gt;)</td>
<td>=&gt; 3,5,8,13,21 (* 3 and 5 also exchange places *)</td>
</tr>
<tr>
<td>(&lt; A_2 : A_3 &gt;)</td>
<td>=&gt; 3,5,8,21,13 ➞ 8 is the median</td>
</tr>
</tbody>
</table>
4. 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 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 exists.

**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.

**Proof:**

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

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

**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.

**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.

**Correctness:** By Observation II.

**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 \( \Theta(n) \).

**BoyerMoore majority vote algorithm:**

- Algorithm and proof: [https://gregable.com/2013/10/majority-vote-algorithm-find-majority.html](https://gregable.com/2013/10/majority-vote-algorithm-find-majority.html)
- Video without a proof: [https://www.youtube.com/watch?v=x0yG0e95F1k](https://www.youtube.com/watch?v=x0yG0e95F1k)
5. 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 \( \Theta(1) \).