
Describe an efficient algorithm to find 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 complexity of your algorithm?

**Observation:** Let $Max$ and $Min$ be the maximum and minimum numbers in $A$. Then the answer is

$$M = \max_{1 \leq i, j \leq n} \{ A[i] - A[j] \} = Max - Min$$

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

**Remark:** The algorithm is optimal since there is no algorithm that can find both $Max$ and $Min$ with less than $\lceil \frac{3n^2}{2} \rceil - 2$ comparisons.
2. Let \( A = A[1], \ldots, A[n] \) be an array of \( n \geq 4 \) distinct keys.

Describe an efficient algorithm to find the three smallest keys in \( A \).

What is the worst case number of comparisons performed by your algorithm. Try to find an exact number. Ignore floors and ceilings.

**Answer:** The algorithm that finds the two smallest keys has two phases. In the first phase, it runs a tournament on all \( n \) keys to find the smallest key with exactly \( n - 1 \) comparisons in \( \lceil \log n \rceil \) rounds. In the second phase, it runs a second smaller tournament only on those \( \lceil \log n \rceil \) keys that were directly compared with the smallest key to find the second smallest key with \( \lceil \log n \rceil - 1 \) comparisons in \( \lceil \log \lceil \log n \rceil \rceil \) rounds. This is because in a tournament any key is directly compared with at most \( \lceil \log n \rceil \) other keys. The total number of comparisons performed by this algorithm is at most

\[
(n - 1) + (\lceil \log n \rceil - 1) = n + \lceil \log n \rceil - 2 .
\]

**Observation:** Let \( x \) be the smallest key and let \( y \) be the second smallest key. Let \( z \) be another key.

- **Case I:** \( z \) was directly compared with \( x \) but was not directly compared with \( y \): Then \( z \) must be larger than another key in the second tournament and therefore cannot be the third smallest key.

- **Case II:** \( z \) was directly compared neither with \( x \) nor with \( y \): Then \( z \) must be larger than another key in the first tournament and therefore cannot be the third smallest key.

- **Case III:** \( z \) was directly compared with \( y \) in any of the two tournaments: Then \( z \) can be the third smallest key.

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

**Correctness:** By the above observation.

**Size of the third tournament:** There are at most \( \lceil \log n \rceil - 1 \) such keys from the first tournament and at most \( \lceil \log \lceil \log n \rceil \rceil \) such keys from the second tournament. Therefore, the number of comparisons performed by the third tournament is at most \( \lfloor \log n \rfloor + \lceil \log \lceil \log n \rceil \rceil - 2 \).

**Total number of comparisons in all three tournaments:**

\[
(n - 1) + (\lceil \log n \rceil - 1) + (\lfloor \log n \rfloor + \lceil \log \lceil \log n \rceil \rceil - 2) .
\]

Since \( \lfloor x \rfloor \leq x + 1 \) for a positive \( x \), it follows that

\[
n + 2 \log(n) + \log \log(n) - 1 .
\]
3. Design an efficient algorithm to find the median of 5 distinct keys.

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 and it is greater than $A_2$ because $A_4$ is greater than $A_2$. Since the median can be greater than only 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 $< A_3 : A_4 >$.

At this stage, $A_4$ is greater than $A_3$ because it was directly compared with it and it is greater than $A_1$ and $A_2$ because it was directly compared with one of them while the other is smaller than $A_3$. Since the median can be greater than only two keys, it follows that $A_4$ cannot be the median.

There are two options:
- $A_3 > A_1$: $A_1$ cannot be the median since it is smaller than $A_3$, $A_4$, and $A_5$. Therefore, the last comparison is $< A_2 : A_3 >$.
- $A_3 > A_2$: $A_2$ cannot be the median since it is smaller than $A_3$, $A_4$, and $A_5$. Therefore, the last comparison is $< A_1 : A_3 >$.

At this stage $A_3$ is greater than $A_1$ and $A_2$ and is less than $A_4$ and $A_5$. Therefore, $A_3$ is the median after 6 comparisons.

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 *)
$< A_1 : A_3 > = < 21, 3 > \implies 3, 5, 21, 8, 13$
$< A_3 : A_4 > = < 21, 8 > \implies 3, 5, 8, 21, 13$ (*8 was directly compared with 5 *)
$< A_1 : A_3 > = < 3, 8 > \implies 3, 5, 8, 21, 13 \implies 8$ 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 *)
$< A_1 : A_3 > = < 13, 5 > \implies 5, 3, 13, 8, 21$
$< A_3 : A_4 > = < 13, 8 > \implies 5, 3, 8, 13, 21$ (*8 was directly compared with 3 *)
$< A_1 : A_3 > = < 5, 8 > \implies 5, 3, 8, 21, 13 \implies 8$ is the median
4. Let $A$ be an array containing $n$ very large positive integers not necessarily distinct. A *majority* is a number that appears at least $\left\lceil \frac{n+1}{2} \right\rceil$ times in the array (note that there can be at most one majority). Describe an $O(n)$-time algorithm that finds a majority in $A$ if exists.

**Observation I:**

- Even $n$: $\left\lceil \frac{n+1}{2} \right\rceil = \frac{n}{2} + 1$ and therefore $n - \left\lceil \frac{n+1}{2} \right\rceil = \frac{n}{2} - 1$.
- Odd $n$: $\left\lceil \frac{n+1}{2} \right\rceil = \frac{n+1}{2}$ and therefore $n - \left\lceil \frac{n+1}{2} \right\rceil = \frac{n-1}{2}$.

**Corollary 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.

**Observation II:** If $m$ is a majority then $m$ is the median of the array.

**Proof:** Sort $A$. Then $A[\lceil (n+1)/2 \rceil]$ must equal $m$ since both the number of indices smaller than the median $\left\lfloor (n+1)/2 \right\rfloor - 1$ and the number of indices greater than the median $n - \left\lfloor (n+1)/2 \right\rfloor$ are less than $\left\lceil (n+1)/2 \right\rceil$.

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

**Correctness:** By Observation II.

**Complexity:** The median can be found with a $\Theta(n)$ algorithm and the array scan can be implemented in $\Theta(n)$ time. The total complexity is $\Theta(n)$.

**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=iuG4SvDEFlk](https://www.youtube.com/watch?v=iuG4SvDEFlk)
5. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be a sorted array containing \( n \) distinct positive integers and let \( k \) be a positive integer. Describe an \( O(n) \)-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 \([\ell..r]\). Follow the recursive steps 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 \([\ell..(r - 1)]\). \( \text{(*) Observation I *)} \)
- If \( A_\ell + A_r < k \): continue recursively with the range \([(\ell + 1)..r]\). \( \text{(*) Observation II *)} \)

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

**Complexity:** Let \( \Delta = r - \ell \). Initially, \( \Delta = n - 1 \). In each recursive call the value of \( \Delta \) is decreased by 1. The maximum number of recursive calls happens when the answer is NO. At this stage, \( \Delta = -1 \). Therefore there are at most \( n \) recursive calls. The total complexity is \( \Theta(n) \) since the complexity of each recursive call is \( \Theta(1) \).
6. Let \( A = [A_1 < A_2 < \cdots < A_n] \) be a sorted array containing \( n \) distinct negative and positive integers. Describe an \( O(\log(n)) \)-time algorithm that finds if exists an index \( 1 \leq i \leq n \) such that \( A_i = i \).

**Observation I:** For any two indices \( 1 \leq i < j \leq n \):

\[
(j - i) \leq A_j - A_i
\]

**Proof I:** Since the integers in the sorted array are distinct, it follows that

\[
\begin{align*}
1 & \leq A_{i+1} - A_i \\
2 & \leq A_{i+2} - A_i \\
3 & \leq A_{i+3} - A_i \\
& \vdots \\
(j - i) & \leq A_{i+(j-i)} - A_i
\end{align*}
\]

The last inequality is equivalent to \( j - i \leq A_j - A_i \).

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

**Proof II:** By Observation I, \( j - A_j \leq i - A_i \). By the assumption, \( 0 < j - A_j \). Therefore, \( 0 < i - A_i \) which is equivalent to \( A_i < i \).

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

**Proof III:** By Observation I, \( A_j - j \geq A_i - i \). By the assumption, \( A_i - i > 0 \). Therefore, \( A_j - j > 0 \) which is equivalent to \( A_j > j \).

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

**Recursive step** for the range \([\ell..r]\) for which \( \ell \leq r \): Let \( m = \left\lfloor \frac{\ell+r}{2} \right\rfloor \) be the middle index of the range \([\ell..r]\). Compare \( A_m \) with \( m \).

- If \( A_m = m \): return \( m \).
- If \( A_m < m \): continue recursively with the range \([(m+1)..r]\). (* Observation II *)
- If \( A_m > m \): continue recursively with the range \([\ell..(m-1)]\). (* Observation III *)

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

**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 \( \lfloor \log(n/2) \rfloor \) recursive steps. The time complexity of each recursive step is \( \Theta(1) \) which implies that the complexity of the algorithm is \( \Theta(\log n) \).