# 215. Kth Largest Element in an Array

Medium Array Divide and Conquer **Quickselect** Sorting Heap (Priority Queue) Leetcode Link

## Problem Description

The given problem presents us with a scenario where we need to find the *k*(^{th}) largest element in a provided integer array `nums`. Unlike finding the maximum or minimum element, this task requires us to identify the value that would be placed at the *k*(^{th}) largest position if the array were sorted. However, the twist in this problem is that duplicates are allowed, and each instance counts for the position. For example, if `nums = [3,2,3,1,2,4,5,5,6]` and `k = 4`, the fourth largest element is `4`.

Moreover, the problem poses an additional challenge: it hints at the possibility of finding the solution without sorting the entire array, which might suggest that there are more optimal ways to solve this especially when we consider time complexity.

## Intuition

To solve this problem within the constraint of not fully sorting the array, we can draw on strategies from the concept of Quick Select, which is a selection algorithm based on the partitioning logic of Quick Sort.

Quick Sort operates by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The pivotal insight here is that once we position a pivot during partitioning, its final sorted position is fixed in the array.

We use this characteristic of Quick Sort to our advantage. By performing partitioning, if we position a pivot at index *p*, and *p* is the `n-k` position in a 0-indexed array (the position where our kth largest element would be if the array were sorted in ascending order), then we have our answer: the element at index *p*. If *p* is less than `n-k`, it implies the kth largest element is to the right, so we only need to partition the sub-array to the right of *p*. Conversely, if *p* is larger than `n-k`, we only need to partition to the left of *p*.

Using this approach, the algorithm short-circuits the sorting process, only partially sorting the elements around the `k`(^{th}) largest value until it is positioned correctly. In the best case, this results in linear time complexity, although in the worst case (which can be mitigated by good pivot selection), it can reach quadratic complexity—it's still generally more optimal than fully sorting the array.

## Solution Approach

To implement the solution based on the intuition described earlier, the Reference Solution Approach uses the Quick Select algorithm, which is inspired by the partition process of Quick Sort.

Here's a step-by-step explanation of the algorithm in the given solution:

1. Define a helper function `quick_sort` which will handle the partitioning and the recursive search.
2. Within `quick_sort`, calculate a pivot value `x` as the element in the middle index of `nums`—this is computed as `nums[(left + right) >> 1]` where `>> 1` is a bit-shift operation equivalent to integer division by 2. This pivot is arbitrary, and different selection methods can be used to optimize the algorithm.
3. Set two pointers, `i` starting from the left bound `left - 1` and `j` from the right bound `right + 1`. Then increment `i` and decrement `j` until an element greater than or equal to `x` is found from the left and an element less than or equal to `x` is found from the right.
4. If `i < j` which means elements are on the wrong side of the pivot, swap them. Repeat this process until `i` is not less than `j`.
5. After the loop, `j` will be at the partition point. If `j` is less than the index we're interested in (`k`), we know our kth largest element is on the right of `j` hence call `quick_sort` recursively for the right sub-array.
6. If `j` is greater or equal to `k`, the kth largest element is on the left side or at `j`.
7. Invoke this helper function with the entirety of the input array's bounds and `n - k` as the target position, starting the process.

The algorithm's success hinges on the partitioning step, which ensures that after each pass, at least one element (the pivot) is in its final sorted position. By targeting the `n - k` index, we aim to put the kth largest element (which is the `k`th from the last in sorted order) in its correct position.

To visualize this algorithm with an example, let's consider `nums = [3,2,1,5,6,4]` and `k = 2`. The steps would be:

- Pick a pivot (let's say it's `3`).
- After one partitioning step, the array could look like `[2, 1, 3, 5, 6, 4]`.
- `3` is now in its correct sorted position (index 2). We're looking for the index `4` (`n - k`), which is to the right of our pivot. Therefore, we'll now focus on the sub-array `[5, 6, 4]`.
- The algorithm will then repeat these steps on the sub-array until the element at the `n-k` place is positioned correctly, at which point it will return as the kth largest element.

By performing this targeted partitioning, we avoid the need to sort the entire array, which gives us a quicker path to the desired kth largest element.

### Example Walkthrough

Let's illustrate the solution approach with a small example. We will use the array `nums = [3, 2, 1, 5, 6, 4]` and `k = 2`, meaning we want to find the 2nd largest element.

1. We need to find the `n-k` **th** element in the sorted order, which translates to `6 - 2 = 4` in 1-indexed or `3` in 0-indexed notation in our array.
2. We pick a pivot, let's use the median value of `3` as an example. The `quick_sort` function is then called with the entire array along with indices of the bounds `0` and `5`.
3. Partition the array around the pivot `3`. One possible way the array could be partitioned is `[2, 1, 3, 5, 6, 4]`. After partitioning, the pivot `3` is positioned at its correct sorted index `2`.
4. Since our target index `3` is greater than the pivot's index `2`, we ignore the left part of the array including our pivot, and focus on the right part: `[5, 6, 4]`.
5. We repeat the process on this sub-array. We pick a new pivot, say `5`. Partitioning this sub-array might result in `[4, 5, 6]`.
6. The pivot `5` lands at index `1` of the sub-array (`4` in the original array, due to `0`-indexing). Now the position of `5` is the same as our target index `4` (`n-k`), which means it's the `k`th largest element.

The second-largest element in this array is `5`, and thus is the output of our algorithm using Quick Select. This process saved time by not having to sort the whole array and instead focused on finding the specific `k`th element directly.

## Python Solution

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # Helper method to perform the quickselect algorithm
        def quick_select(start, end, k_smallest):
            # If the list contains only one element,
            # return that element
            if start == end:
                return nums[start]
            pivot_index = (start + end) // 2  # Choose the middle element as the pivot
            pivot_value = nums[pivot_index]
            left, right = start - 1, end + 1

            # Partition the list such that all elements greater than
            # the pivot are to the left and all elements less than
            # are to the right
            while left < right:
                # Increment left index until finding an element less than the pivot
                while True:
                    left += 1
                    if nums[left] >= pivot_value:
                        break
                # Decrement right index until finding an element greater than the pivot
                while True:
                    right -= 1
                    if nums[right] <= pivot_value:
                        break
                # Swap elements from both sides if needed
                if left < right:
                    nums[left], nums[right] = nums[right], nums[left]

            # If the partitioning index is less than k_smallest, we know that
            # the kth largest element must be in the right partition.
            # If it's greater than or equal to k_smallest, the element will
            # be in the left partition.
            if right < k_smallest:
                return quick_select(right + 1, end, k_smallest)
            return quick_select(start, right, k_smallest)

        # Calculate the 'k_smallest' index based on the 'kth largest' requirement
        n = len(nums)
        k_smallest = n - k
        # Call the quick_select helper function to find the kth largest element
        return quick_select(0, n - 1, k_smallest)

# Note: The variable names 'start', 'end', 'k_smallest', 'pivot_index', 'pivot_value',
# 'left', and 'right' were chosen to improve clarity and adhere to standard naming conventions.
```

## Java Solution

```java
class Solution {
    // Function to find the kth largest element in the array
    public int findKthLargest(int[] nums, int k) {
        int n = nums.length;
        // Find the (n-k)th smallest element because the kth largest is also the (n-k)th smallest when sorted in ascending order
        return quickSelect(nums, 0, n - 1, n - k);
    }

    // Helper function to perform quick select
    private int quickSelect(int[] nums, int left, int right, int kSmallest) {
        // When the left and right pointers meet, we've found the kSmallest element
        if (left == right) {
            return nums[left];
        }

        // Initialize two pointers for the partitioning step
        int i = left - 1;
        int j = right + 1;
        // Choose pivot as the middle element
        int pivot = nums[(left + right) >>> 1];

        while (i < j) {
            // Move i right past any elements less than the pivot
            do {
                i++;
            } while (nums[i] < pivot);

            // Move j left past any elements greater than the pivot
            do {
                j--;
            } while (nums[j] > pivot);

            // Swap elements at i and j if they are out of order with respect to the pivot
            if (i < j) {
                swap(nums, i, j);
            }
        }

        // After partitioning, the pivot is now at index j
        // If we found the kSmallest element, return it
        if (j >= kSmallest) {
            return quickSelect(nums, left, j, kSmallest);
        }

        // Otherwise, continue the search in the right partition
        return quickSelect(nums, j + 1, right, kSmallest);
    }

    // Swap function to swap two elements in the array
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

## C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Public method to find the k-th largest element in the array.
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();
        // The k-th largest is the (n - k)-th smallest, adjust and use quick select.
        return quickSelect(nums, 0, n - 1, n - k);
    }

```

```
private:
    // Utility method to perform quick select.
    int quickSelect(vector<int>& nums, int left, int right, int k) {
        if (left == right) { // If the sub-array has only one element, return that element.
            return nums[left];
        }

        int pivotIndex = left + (right - left) / 2; // Choose the middle element as pivot.
        int pivotValue = nums[pivotIndex];
        int i = left, j = right;

        // Partition the array around the pivot.
        while (i <= j) {
            // Find leftmost element greater than or equal to the pivot.
            while (nums[i] < pivotValue) {
                i++;
            }
            // Find rightmost element less than or equal to the pivot.
            while (nums[j] > pivotValue) {
                j--;
            }
            // Swap elements to ensure all elements on left are less than pivot
            // and all on right are greater than pivot.
            if (i <= j) {
                swap(nums[i], nums[j]);
                i++;
                j--;
            }
        }

        // After partitioning, the pivot is at its final sorted position,
        // check if this position is the one we're looking for.
        if (k <= j) {
            // If k is in the left partition, recursively search in the left part.
            return quickSelect(nums, left, j, k);
        } else {
            // If k is in the right partition, recursively search in the right part.
            return quickSelect(nums, i, right, k);
        }
    }
};

```

## Typescript Solution

```
function findKthLargest(nums: number[], k: number): number {
    // Helper function to swap elements at indices i and j within nums array
    const swapElements = (i: number, j: number) => {
        [nums[i], nums[j]] = [nums[j], nums[i]];
    };

    // Helper function to implement Quick Select algorithm
    const quickSelect = (left: number, right: number) => {
        // Return early if the partition size is smaller than the kth element we are looking for
        if (left >= right || left + 1 > k) {
            return;
        }

        // Randomly select a pivot and move it to the start
        swapElements(left, left + Math.floor(Math.random() * (right - left)));
        const pivot = nums[left];
        let pivotIndex = left;

        // Partition the array around the pivot
        for (let i = left + 1; i < right; i++) {
            // If the current element is greater than the pivot, swap it with the element
            // at the pivotIndex and increment pivotIndex
            if (nums[i] > pivot) {
                pivotIndex++;
                swapElements(i, pivotIndex);
            }
        }

        // Put the pivot element at its correct position
        swapElements(left, pivotIndex);

        // Recursively apply the same logic to the left and right partitions
        quickSelect(left, pivotIndex);
        quickSelect(pivotIndex + 1, right);
    };

    // Start the quick select process on the entire array
    quickSelect(0, nums.length);

    // Return the kth largest element
    return nums[k - 1];
}

```

## Time and Space Complexity

The implemented function `findKthLargest` uses the QuickSelect algorithm, which is a variation of the QuickSort algorithm, to find the `k`th largest element in the list.

### Time Complexity

The average time complexity of the QuickSelect algorithm is `O(n)`, where `n` is the number of elements in the list. This is because it generally only needs to recursively partition one side of the pivot, effectively reducing the problem size by about half each time.

However, in the worst-case scenario, the time complexity degrades to `O(n^2)`. This happens when the pivot selected is always the smallest or largest element in each recursion, causing the algorithm to partition around each element one by one. In practice, with a good pivot selection strategy (such as using the median-of-medians algorithm), the worst-case complexity can be improved, but this code uses a simple middle element pivot, so it remains vulnerable to the worst-case behavior.

### Space Complexity

The space complexity of the QuickSelect algorithm used here is `O(log n)` on average due to the recursion stack. It requires additional space proportional to the depth of the recursion tree which, on average, has a depth of `log n`.

In the worst case, where the smallest or largest element is always chosen as the pivot, the recursive calls can go up to `n` levels deep resulting in a space complexity of `O(n)`.