K’th Smallest/Largest Element in Unsorted Array

Given an array **arr[]** of size **N** and a number **K,** where **K** is smaller than the size of the array. Find the **K’th** smallest element in the given array. Given that all array elements are distinct.

Examples:

Input: *arr[] = {7, 10, 4, 3, 20, 15}, K = 3*

Output: *7*

Input: *arr[] = {7, 10, 4, 3, 20, 15}, K = 4*

Output: *10*

K’th smallest element in an unsorted array using **Sorting**:

*Sort the given array and return the element at index **K-1** in the sorted array.*

- Sort the input array in the increasing order
- Return the element at the K-1 index (0 – Based indexing) in the sorted array

Below is the Implementation of the above approach:

C++

```
// C++ program to find K'th smallest element
#include <bits/stdc++.h>
using namespace std;

// Function to return K'th smallest element in a given array
int kthSmallest(int arr[], int N, int K)
{
    // Sort the given array
    sort(arr, arr + N);

    // Return k'th element in the sorted array
    return arr[K - 1];
}
```

```
// Driver's code
int main()
{
    int arr[] = { 12, 3, 5, 7, 19 };
    int N = sizeof(arr) / sizeof(arr[0]), K = 2;

    // Function call
    cout << "K'th smallest element is "
         << kthSmallest(arr, N, K);
    return 0;
}
```

Output

```
K'th smallest element is 5
```

Time Complexity: O(N log N)
Auxiliary Space: O(1)

K'th smallest element in an unsorted array using **Binary Search on Answer**:

*To find the kth smallest element using **binary search on the answer**, we start by defining a search range based on the minimum and maximum values in the input array. In each iteration of binary search, we count the elements smaller than or equal to the midpoint and update the search range accordingly. This process continues until the range collapses to a single element, which is the kth smallest element.*

Follow the given steps to solve the problem:

- Intialize **low** and **high** to **minimum** and **maximum** element of the array denoting the range within which the answer lies.
- Apply Binary Search on this range.

- If the selected element by calculating **mid** has less than K elements lesser to it then increase the number that is **low = mid + 1**.
- Otherwise, **Decrement** the high pointer, i.e **high = mid.**
- The Binary Search will end when only one element remains in the answer space that would be the answer.

Below is the implementation of above approach:
C++

```
// C++ code for the above approach


#include <bits/stdc++.h>
#include <iostream>


using namespace std;


int count(vector<int>& nums, int& mid)
{
    // function to calculate number of elements less than
    // equal to mid
    int cnt = 0;


    for (int i = 0; i < nums.size(); i++)
        if (nums[i] <= mid)
            cnt++;


    return cnt;
}


int kthSmallest(vector<int> nums, int& k)
```

```
{
    int low = INT_MAX;
    int high = INT_MIN;
    // calculate minimum and maximum the array.
    for (int i = 0; i < nums.size(); i++) {
        low = min(low, nums[i]);
        high = max(high, nums[i]);
    }
    // Our answer range lies between minimum and maximum
    // element of the array on which Binary Search is
    // Applied
    while (low < high) {
        int mid = low + (high - low) / 2;
        /*if the count of number of elements in the array
          less than equal to mid is less than k then
          increase the number. Otherwise decrement the
          number and try to find a better answer.
        */
        if (count(nums, mid) < k)
            low = mid + 1;

        else
            high = mid;
    }

    return low;
}
```

```
// Driver's code
int main()
{

    vector<int> nums{ 1, 4, 5, 3, 19, 3 };
    int k = 3;

    // Function call
    cout << "K'th smallest element is "
         << kthSmallest(nums, k);
    return 0;
}
```

Output

```
K'th smallest element is 3
```

Time complexity: O(n * log (mx-mn)), where mn be minimum and mx be maximum element of array.
Auxiliary Space: O(1)

K'th smallest element in an unsorted array using Priority Queue(Max-Heap):

*The intuition behind this approach is to maintain a **max heap** (priority queue) of size **K** while iterating through the array. Doing this ensures that the **max heap** always contains the **K** smallest elements encountered so far. If the size of the max heap exceeds **K**, remove the largest element this step ensures that the heap maintains the **K** smallest elements encountered so far. In the end, the max heap's top element will be the **Kth** smallest element.*

- Initialize a max heap (priority queue) **pq**.
- For each element in the array:
    - Push the element onto the max heap.
    - If the size of the max heap exceeds K, pop (remove) the largest element from the max heap. This step ensures that

the max heap maintains the K smallest elements encountered so far.

- After processing all elements, the max heap will contain the K smallest elements, with the largest of these K elements at the top.

Below is the Implementation of the above approach:

C++

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to find the kth smallest array element
int kthSmallest(int arr[], int N, int K)
{
    // Create a max heap (priority queue)
    priority_queue<int> pq;

    // Iterate through the array elements
    for (int i = 0; i < N; i++) {
        // Push the current element onto the max heap
        pq.push(arr[i]);

        // If the size of the max heap exceeds K, remove the largest element
        if (pq.size() > K)
            pq.pop();
    }

    // Return the Kth smallest element (top of the max heap)
    return pq.top();
}
```

```
// Driver's code:
int main()
{
    int N = 10;
    int arr[N] = { 10, 5, 4, 3, 48, 6, 2, 33, 53, 10 };
    int K = 4;

    // Function call
    cout << "Kth Smallest Element is: "
         << kthSmallest(arr, N, K);
}
```

Output

```
Kth Smallest Element is: 5
```

Time Complexity: O(N * log(K)), The approach efficiently maintains a container of the K smallest elements while iterating through the array, ensuring a time complexity of O(N * log(K)), where N is the number of elements in the array.

Auxiliary Space: O(K)

-

K'th smallest element in an unsorted array using QuickSelect:

*This is an optimization over method 1, if QuickSort is used as a sorting algorithm in first step. In QuickSort, pick a **pivot** element, then move the **pivot** element to its correct position and partition the surrounding array. The idea is, not to do complete quicksort, but stop at the point where **pivot** itself is k'th smallest element. Also, not to recur for both **left** and **right** sides of **pivot**, but recur for one of them according to the position of pivot.*

Follow the given steps to solve the problem:

- Run quick sort algorithm on the input array

- In this algorithm pick a **pivot** element and move it to it's correct position
- Now, if index of **pivot** is equal to **K** then return the value, else if the index of **pivot** is greater than K, then recur for the **left** subarray, else recur for the **right** subarray
- Repeat this process until the element at index **K** is not found

Below is the Implementation of the above approach:
C++

```
// C++ code for the above approach


#include <bits/stdc++.h>
using namespace std;


int partition(int arr[], int l, int r);


// This function returns K'th smallest element in arr[l..r]
// using QuickSort based method. ASSUMPTION: ALL ELEMENTS IN
// ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int K)
{
    // If k is smaller than number of elements in array
    if (K > 0 && K <= r - l + 1) {


        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);


        // If position is same as k
        if (pos - l == K - 1)
```

```
            return arr[pos];
        if (pos - l > K - 1) // If position is more, recur
                             // for left subarray
            return kthSmallest(arr, l, pos - 1, K);

        // Else recur for right subarray
        return kthSmallest(arr, pos + 1, r,
                           K - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers
// the last element as pivot and moves all smaller element
// to left of it and greater elements to right
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
```

```
    for (int j = l; j <= r - 1; j++) {
        if (arr[j] <= x) {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }

    swap(&arr[i], &arr[r]);
    return i;
}

// Driver's code
int main()
{
    int arr[] = { 12, 3, 5, 7, 4, 19, 26 };
    int N = sizeof(arr) / sizeof(arr[0]), K = 3;

    // Function call
    cout << "K'th smallest element is "
         << kthSmallest(arr, 0, N - 1, K);
    return 0;
}
```

Output

```
K'th smallest element is 5
```

Time Complexity: O(N_2) in worst case and O(N) on average. However if we randomly choose pivots, the probability of worst case could become very less.
Auxiliary Space: O(N)

K'th smallest element in an unsorted array using **Counting Sort**:

*Counting sort is a linear time sorting algorithm that counts the occurrences of each element in an array and uses this information to determine the sorted order. The intuition behind using counting sort to find the kth smallest element is to take advantage of its counting phase, which essentially calculates the cumulative frequencies of elements. By tracking these cumulative frequencies and finding the point where the count reaches or exceeds **K** can determine the kth smallest element efficiently.*

- Find the maximum element in the input array to determine the range of elements.
- Create an array **freq** of size **max_element + 1** to store the frequency of each element in the input array. Initialize all elements of **freq** to **0**.
- Iterate through the input array and update the frequencies of elements in the **freq** array.
- Initialize a **count** variable to keep track of the cumulative frequency of elements.
- Iterate through the **freq** array from **0** to **max_element**:
- If the frequency of the current element is non-zero, add it to the **count**.
- Check if **count** is greater than or equal to **k**. If it is, return the current element as the kth smallest element.

Below is the Implementation of the above approach:

C++

```
#include <iostream>
using namespace std;


// This function returns the kth smallest element in an array
int kthSmallest(int arr[], int n, int k) {
    // First, find the maximum element in the array
    int max_element = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max_element) {
```

```
            max_element = arr[i];
        }
    }

    // Create an array to store the frequency of each
   // element in the input array
    int freq[max_element + 1] = {0};
    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    // Keep track of the cumulative frequency of elements
   // in the input array
    int count = 0;
    for (int i = 0; i <= max_element; i++) {
        if (freq[i] != 0) {
            count += freq[i];
            if (count >= k) {
                // If we have seen k or more elements,
              // return the current element
                return i;
            }
        }
    }
    return -1;
}
```

```
// Driver Code
int main() {
    int arr[] = {12,3,5,7,19};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;
    cout << "The " << k << "th smallest element is " << kthSmallest(arr, n, k) << endl;

    return 0;
}
```

Output

```
The 2th smallest element is 5
```

Time Complexity:O(N + max_element), where max_element is the maximum element of the array.

Auxiliary Space: O(max_element)