Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,
Given `[3,2,1,5,6,4]` and k = 2, return 5.

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 1

This problem is well known and quite often can be found in various text books.

You can take a couple of approaches to actually solve it:

- O(N lg N) running time + O(1) memory

The simplest approach is to sort the entire input array and then access the element by it's index (which is O(1)) operation:

```java
public int findKthLargest(int[] nums, int k) {
        final int N = nums.length;
        Arrays.sort(nums);
        return nums[N - k];
}
```

- O(N lg K) running time + O(K) memory

Other possibility is to use a min oriented priority queue that will store the K-th largest values. The algorithm iterates over the whole input and maintains the size of priority queue.

```java
public int findKthLargest(int[] nums, int k) {

    final PriorityQueue<Integer> pq = new PriorityQueue<>();
    for(int val : nums) {
        pq.offer(val);

        if(pq.size() > k) {
            pq.poll();
        }
    }
    return pq.peek();
}
```

- O(N) best case / O(N^2) worst case running time + O(1) memory

The smart approach for this problem is to use the selection algorithm (based on the partion method - the same one as used in quicksort).

```java
public int findKthLargest(int[] nums, int k) {

        k = nums.length - k;
        int lo = 0;
        int hi = nums.length - 1;
        while (lo < hi) {
            final int j = partition(nums, lo, hi);
            if(j < k) {
                lo = j + 1;
            } else if (j > k) {
                hi = j - 1;
            } else {
                break;
            }
        }
        return nums[k];
    }

    private int partition(int[] a, int lo, int hi) {

        int i = lo;
        int j = hi + 1;
        while(true) {
            while(i < hi && less(a[++i], a[lo]));
            while(j > lo && less(a[lo], a[--j]));
            if(i >= j) {
                break;
            }
            exch(a, i, j);
        }
        exch(a, lo, j);
        return j;
    }

    private void exch(int[] a, int i, int j) {
        final int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }

    private boolean less(int v, int w) {
        return v < w;
    }
```

O(N) guaranteed running time + O(1) space

So how can we improve the above solution and make it O(N) guaranteed? The answer is quite simple, we can randomize the input, so that even when the worst case input would be provided the algorithm wouldn't be affected. So all what it is needed to be done is to shuffle the input.

```
public int findKthLargest(int[] nums, int k) {

        shuffle(nums);
        k = nums.length - k;
        int lo = 0;
        int hi = nums.length - 1;
        while (lo < hi) {
            final int j = partition(nums, lo, hi);
            if(j < k) {
                lo = j + 1;
            } else if (j > k) {
                hi = j - 1;
            } else {
                break;
            }
        }
        return nums[k];
    }

private void shuffle(int a[]) {

        final Random random = new Random();
        for(int ind = 1; ind < a.length; ind++) {
            final int r = random.nextInt(ind + 1);
            exch(a, ind, r);
        }
    }
```

There is also worth mentioning the Blum-Floyd-Pratt-Rivest-Tarjan algorithm that has a guaranteed O(N) running time.

written by jmnarloch original link here

Solution 2

Well, this problem has a naive solution, which is to sort the array in descending order and return the `k-1`-th element. However, sorting algorithm gives `O(nlogn)` complexity. Suppose `n = 10000` and `k = 2`, then we are doing a lot of unnecessary operations. In fact, this problem has at least two simple and faster solutions.

Well, the faster solution has no mystery. It is also closely related to sorting. I will give two algorithms for this problem below, one using quicksort(specifically, the partition subroutine) and the other using heapsort.

Quicksort

In quicksort, in each iteration, we need to select a pivot and then partition the array into three parts:

1. Elements smaller than the pivot;
2. Elements equal to the pivot;
3. Elements larger than the pivot.

Now, let's do an example with the array `[3, 2, 1, 5, 4, 6]` in the problem statement. Let's assume in each time we select the leftmost element to be the pivot, in this case, `3`. We then use it to partition the array into the above 3 parts, which results in `[1, 2, 3, 5, 4, 6]`. Now `3` is in the third position and we know that it is the third smallest element. Now, do you recognize that this subroutine can be used to solve this problem?

In fact, the above partition puts elements smaller than the pivot before the pivot and thus the pivot will then be the `k`-th smallest element if it is at the `k-1`-th position. Since the problem requires us to find the `k`-th largest element, we can simply modify the partition to put elements larger than the pivot before the pivot. That is, after partition, the array becomes `[5, 6, 4, 3, 1, 2]`. Now we know that `3` is the `4`-th largest element. If we are asked to find the `2`-th largest element, then we know it is left to `3`. If we are asked to find the `5`-th largest element, then we know it is right to `3`. So, in the **average** sense, the problem is reduced to approximately half of its original size, giving the recursion `T(n) = T(n/2) + O(n)` in which `O(n)` is the time for partition. This recursion, once solved, gives `T(n) = O(n)` and thus we have a linear time solution. Note that since we only need to consider one half of the array, the time complexity is `O(n)`. If we need to consider both the two halves of the array, like quicksort, then the recursion will be `T(n) = 2T(n/2) + O(n)` and the complexity will be `O(nlogn)`.

Of course, `O(n)` is the average time complexity. In the worst case, the recursion may become `T(n) = T(n - 1) + O(n)` and the complexity will be `O(n^2)`.

Now let's briefly write down the algorithm before writing our codes.

1. Initialize `left` to be 0 and `right` to be `nums.size() - 1`;
2. Partition the array, if the pivot is at the `k-1`-th position, return it (we are

done);
3. If the pivot is right to the `k-1`-th position, update `right` to be the left neighbor of the pivot;
4. Else update `left` to be the right neighbor of the pivot.
5. Repeat 2.

Now let's turn it into code.

```
class Solution {
public:
    int partition(vector<int>& nums, int left, int right) {
        int pivot = nums[left];
        int l = left + 1, r = right;
        while (l <= r) {
            if (nums[l] < pivot && nums[r] > pivot)
                swap(nums[l++], nums[r--]);
            if (nums[l] >= pivot) l++;
            if (nums[r] <= pivot) r--;
        }
        swap(nums[left], nums[r]);
        return r;
    }

    int findKthLargest(vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pos = partition(nums, left, right);
            if (pos == k - 1) return nums[pos];
            if (pos > k - 1) right = pos - 1;
            else left = pos + 1;
        }
    }
};
```

Heapsort

Well, this problem still has a tag "heap". If you are familiar with heapsort, you can solve this problem using the following idea:

1. Build a max-heap for `nums`, set `heap_size` to be `nums.size()`;
2. Swap `nums[0]` (after buding the max-heap, it will be the largest element) with `nums[heap_size - 1]` (currently the last element). Then decrease `heap_size` by 1 and max-heapify `nums` (recovering its max-heap property) at index `0`;
3. Repeat 2 for `k` times and the `k`-th largest element will be stored finally at `nums[heap_size]`.

Now I paste my code below. If you find it tricky, I suggest you to read the Heapsort chapter of Introduction to Algorithms, which has a nice explanation of the algorithm. My code simply translates the pseudo code in that book :-)

```cpp
class Solution {
public:
    inline int left(int idx) {
        return (idx << 1) + 1;
    }
    inline int right(int idx) {
        return (idx << 1) + 2;
    }
    void max_heapify(vector<int>& nums, int idx) {
        int largest = idx;
        int l = left(idx), r = right(idx);
        if (l < heap_size && nums[l] > nums[largest]) largest = l;
        if (r < heap_size && nums[r] > nums[largest]) largest = r;
        if (largest != idx) {
            swap(nums[idx], nums[largest]);
            max_heapify(nums, largest);
        }
    }
    void build_max_heap(vector<int>& nums) {
        heap_size = nums.size();
        for (int i = (heap_size >> 1) - 1; i >= 0; i--)
            max_heapify(nums, i);
    }
    int findKthLargest(vector<int>& nums, int k) {
        build_max_heap(nums);
        for (int i = 0; i < k; i++) {
            swap(nums[0], nums[heap_size - 1]);
            heap_size--;
            max_heapify(nums, 0);
        }
        return nums[heap_size];
    }
private:
    int heap_size;
}
```

If we are allowed to use the built-in `priority_queue`, the code will be much more shorter :-)

```cpp
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int> pq(nums.begin(), nums.end());
        for (int i = 0; i < k - 1; i++)
            pq.pop();
        return pq.top();
    }
};
```

Well, the `priority_queue` can also be replaced by `multiset` :-)

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        multiset<int> mset;
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            mset.insert(nums[i]);
            if (mset.size() > k)
                mset.erase(mset.begin());
        }
        return *mset.begin();
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

https://en.wikipedia.org/wiki/Quickselect

```
public class Solution {

  public int findKthLargest(int[] a, int k) {
    int n = a.length;
    int p = quickSelect(a, 0, n - 1, n - k + 1);
    return a[p];
  }

  // return the index of the kth smallest number
  int quickSelect(int[] a, int lo, int hi, int k) {
    // use quick sort's idea
    // put nums that are <= pivot to the left
    // put nums that are  > pivot to the right
    int i = lo, j = hi, pivot = a[hi];
    while (i < j) {
      if (a[i++] > pivot) swap(a, --i, --j);
    }
    swap(a, i, hi);

    // count the nums that are <= pivot from lo
    int m = i - lo + 1;

    // pivot is the one!
    if (m == k)     return i;
    // pivot is too big, so it must be on the left
    else if (m > k) return quickSelect(a, lo, i - 1, k);
    // pivot is too small, so it must be on the right
    else            return quickSelect(a, i + 1, hi, k - m);
  }

  void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }

}
```

written by jeantimex original link here