How to find the kth largest element in an unsorted array of length n in O(n)?

I believe there's a way to find the kth largest element in an unsorted array of length n in O(n). Or perhaps it's "expected" O(n) or something. How can we do this?

performance algorithm big-o

edited Sep 15 '12 at 2:37
Bill the Lizard
255k ● 148 ● 470 ● 765

asked Oct 30 '08 at 21:06
MrDatabase
14k ● 33 ● 96 ● 146

36 By the way, pretty much every algorithm described here turns into O(n^2) or O(n log n) when k==n. That is, I don't think a single one of them is O(n) for all values of k. I got modded down for pointing this out but thought you should know anyway. – Kirk Strauser Nov 4 '08 at 22:09

17 Selection algorithms can be O(n) for any fixed value of k. That is, you can have a selection algorithm for k=25 that is O(n) for any value of n, and you can do this for any particular value of k that is unrelated to n. The case in which the algorithm is no longer O(n) is when the value of k has some dependency on the value of n, such as k=n or k=n/2. This doesn't, however, mean that if you happen to run the k=25 algorithm on a list of 25 items that it is suddenly no longer O(n) because the O-notation describes a property of the algorithm, not a particular run of it. – Tyler McHenry Jul 31 '09 at 16:58

1 I was asked this question in an amazon interview as a general case of finding the second greatest element. By the way the interviewer lead the interview I didn't ask if I could destroy the original array (i.e. sorting it), so I came up with a complicated solution. – Sambatyon May 9 '11 at 17:43

3 This is Question 9 in Column 11 (Sorting) of Programming Pearls by Jon Bentley. – Qiang Xu Sep 13 '12 at 20:08

2 @KirkStrauser : If k==n or k==n-1 then it becomes trivial. We can get max or 2nd max in single traversal. So algorithms provided here will be practically used for values of k which don't belong to {1,2, n-1, n} – Aditya Joshee May 5 '16 at 10:28

30 Answers

This is called finding the **k-th order statistic**. There's a very simple randomized algorithm (called *quickselect*) taking `O(n)` average time, `O(n^2)` worst case time, and a pretty complicated non-randomized algorithm (called *introselect*) taking `O(n)` worst case time. There's some info on Wikipedia, but it's not very good.

~~Everything you need is in these powerpoint slides~~. Just to extract the basic algorithm of the `O(n)` worst-case algorithm (introselect):

```
Select(A,n,i):
    Divide input into ⌈n/5⌉ groups of size 5.

    /* Partition on median-of-medians */
    medians = array of each group’s median.
    pivot = Select(medians, ⌈n/5⌉, ⌈n/10⌉)
    Left Array L and Right Array G = partition(A, pivot)

    /* Find ith element in L, pivot, or G */
    k = |L| + 1
    If i = k, return pivot
    If i < k, return Select(L, k-1, i)
    If i > k, return Select(G, n-k, i-k)
```

It's also very nicely detailed in the Introduction to Algorithms book by Cormen et al.

edited Aug 2 at 4:01
Ravi
12.1k ● 27 ● 74 ● 123

answered Oct 30 '08 at 21:48
eladv
1,641 ● 1 ● 10 ● 3

6 Thank you for the slides. – Kshitij Banerjee Jun 24 '14 at 7:33

2 Why does it have to work in size 5? Why can't it work with size 3? – Joffrey Baratheon Apr 20 '15 at 1:28

If you notices, this is almost quicksort, except that you stop when pivot is in the "k-th" position. – Sambatyon Jul 24 '15 at 14:49

11 @eladv The slides link is broken :(– Misha Moroshko May 17 '16 at 1:22

7 @eladv Plese fix the broken link. – maxx777 Jan 9 at 20:01

If you want a true `O(n)` algorithm, as opposed to `O(kn)` or something like that, then you should use quickselect (it's basically quicksort where you throw out the partition that you're not interested in). My prof has a great writeup, with the runtime analysis: (reference)

The QuickSelect algorithm quickly finds the k-th smallest element of an unsorted array of `n` elements. It is a RandomizedAlgorithm, so we compute the worst-case *expected* running time.

Here is the algorithm.

```
QuickSelect(A, k)
  let r be chosen uniformly at random in the range 1 to length(A)
  let pivot = A[r]
  let A1, A2 be new arrays
  # split into a pile A1 of small elements and A2 of big elements
  for i = 1 to n
    if A[i] < pivot then
      append A[i] to A1
    else if A[i] > pivot then
      append A[i] to A2
    else
      # do nothing
  end for
  if k <= length(A1):
    # it's in the pile of small elements
    return QuickSelect(A1, k)
  else if k > length(A) - length(A2)
    # it's in the pile of big elements
    return QuickSelect(A2, k - (length(A) - length(A2))
  else
    # it's equal to the pivot
    return pivot
```

What is the running time of this algorithm? If the adversary flips coins for us, we may find that the pivot is always the largest element and `k` is always 1, giving a running time of

```
T(n) = Theta(n) + T(n-1) = Theta($n^2$)
```

But if the choices are indeed random, the expected running time is given by

```
T(n) <= Theta(n) + (1/n) $\sum_{i=1 \text{ to } n}$T(max(i, n-i-1))
```

where we are making the not entirely reasonable assumption that the recursion always lands in the larger of `A1` or `A2`.

Let's guess that `T(n) <= an` for some `a`. Then we get

```
T(n)
 <= cn + (1/n) $\sum_{i=1 \text{ to } n}$T(max(i-1, n-i))
 = cn + (1/n) $\sum_{i=1 \text{ to floor}(n/2)}$ T(n-i) + (1/n) $\sum_{i=\text{floor}(n/2)+1 \text{ to } n}$ T(i)
 <= cn + 2 (1/n) $\sum_{i=\text{floor}(n/2) \text{ to } n}$ T(i)
 <= cn + 2 (1/n) $\sum_{i=\text{floor}(n/2) \text{ to } n}$ ai
```

and now somehow we have to get the horrendous sum on the right of the plus sign to absorb the `cn` on the left. If we just bound it as `2(1/n)` $\sum_{i=n/2 \text{ to } n}$ `an`, we get roughly `2(1/n)(n/2)an = an`. But this is too big - there's no room to squeeze in an extra `cn`. So let's expand the sum using the arithmetic series formula:

```
$\sum_{i=\text{floor}(n/2) \text{ to } n}$ i
 = $\sum_{i=1 \text{ to } n}$ i - $\sum_{i=1 \text{ to floor}(n/2)}$ i
 = n(n+1)/2 - floor(n/2)(floor(n/2)+1)/2
 <= $n^2$/2 - $(n/4)^2$/2
 = (15/32)$n^2$
```

where we take advantage of n being "sufficiently large" to replace the ugly `floor(n/2)` factors with the much cleaner (and smaller) `n/4`. Now we can continue with

```
cn + 2 (1/n) $\sum_{i=\text{floor}(n/2) \text{ to } n}$ ai,
 <= cn + (2a/n) (15/32) $n^2$
 = n (c + (15/16)a)
 <= an
```

provided `a > 16c`.

This gives `T(n) = O(n)`. It's clearly `Omega(n)`, so we get `T(n) = Theta(n)`.

edited Sep 1 '15 at 12:02
Dukeling
40.4k ●9 ●50 ●94

answered Oct 31 '08 at 22:23
Ying Xiao
1,459 ●1 ●8 ●5

10 Quickselect is only O(n) in the average case. The median-of-medians algorithm can be used to solve the problem in O(n) time in the worst case. – John Kurlak Jan 19 '13 at 5:01

What's the meaning of `k > length(A) - length(A2)` ? – MrROY Oct 24 '13 at 10:57

this isn't O(n), you are calling the function again as recursive, T(n). There is already an O(n) there inside the recursive function T(n), so obviously without thinking, overall complexity would be greater than O(n). – user1735921 Jul 30 '14 at 17:25

3 @MrROY Given that we splitted `A` into `A1` and `A2` around the pivot, we know that `length(A) == length(A1)+length(A2)+1`. So, `k > length(A)-length(A2)` is equivalent to `k > length(A1)+1`, which is true when `k` is somewhere in `A2`. – Filipe Gonçalves Aug 17 '14 at 19:28

@FilipeGonçalves, yes if there are no duplicate elements in pivot. len(A1) + len(A2) + K-duplicate = len(A) – d1val Nov 2 '14 at 7:21

A quick Google on that ('kth largest element array') returned this: http://discuss.joelonsoftware.com/default.asp?interview.11.509587.17

```
"Make one pass through tracking the three largest values so far."
```

(it was specifically for 3d largest)

and this answer:

```
Build a heap/priority queue.  O(n)
Pop top element.  O(log n)
Pop top element.  O(log n)
Pop top element.  O(log n)

Total = O(n) + 3 O(log n) = O(n)
```

edited Jul 25 '16 at 19:17
rogerdpack
27k ● 13 ● 107 ● 196

answered Oct 30 '08 at 21:12
warren
16.8k ● 12 ● 65 ● 103

11 well, its actually O(n)+ O(k log n) which doesn't reduce for significant values of K – Jimmy Oct 30 '08 at 21:19

1 right - Big-O is all about approximations :) – warren Oct 30 '08 at 21:21

2 But finding the insertion point in that doubly-linked list is O(k). – Kirk Strauser Oct 30 '08 at 23:25

1 And if k is fixed, O(k) = O(1) – Tyler McHenry Jul 31 '09 at 17:00

1 @warren: Big-O is approximating, but you always over-approximate. Quicksort is actually O(n^2), for example, since that is the worst case. this one is O(n + k log n). – Claudiu Apr 18 '10 at 18:26

You do like quicksort. Pick an element at random and shove everything either higher or lower. At this point you'll know which element you actually picked, and if it is the kth element you're done, otherwise you repeat with the bin (higher or lower), that the kth element would fall in. Statistically speaking, the time it takes to find the kth element grows with n, O(n).

answered Jun 23 '10 at 2:14
stinky
101 ● 1 ● 2

1 This is what quickselect is, FWIW. – rogerdpack Jul 25 '16 at 19:18

A Programmer's Companion to Algorithm Analysis gives a version that *is* O(n), although the author states that the constant factor is so high, you'd probably prefer the naive sort-the-list-then-select method.

I answered the letter of your question :)

answered Oct 30 '08 at 21:17
Jimmy
57.1k ● 12 ● 100 ● 131

2 Not really true in all cases. I have implemented median-of-medians and compared it to built-in Sort method in .NET and custom solution really ran faster by order of magnitude. Now the real question is: does that matter to you in given circumstances. Writing and debugging 100 lines of code compared to one liner pays off only if that code is going to be executed so many times that user starts noticing the difference in running time and feel discomfort waiting for the operation to complete. – Zoran Horvat Jul 19 '13 at 9:14

The C++ standard library has almost exactly that function call `nth_element`, although it does modify your data. It has expected linear run-time, O(N), and it also does a partial sort.

```
const int N = ...;
double a[N];
// ...
const int m = ...; // m < N
nth_element (a, a + m, a + N);
// a[m] contains the mth element in a
```

edited Jul 25 '16 at 19:20
rogerdpack
27k ● 13 ● 107 ● 196

answered Oct 30 '08 at 22:53
David Nehme
16.6k ● 5 ● 56 ● 98

1 No, it has an expected *average* O(n) runtime. For example, quicksort is O(nlogn) on average with a worst case of O(n^2). Wow, something straight up factually wrong! – Kirk Strauser Oct 30 '08 at 23:30

5 No, there's nothing factually wrong with this answer. It works and the C++ standard requires an expected linear run time. – David Nehme Oct 31 '08 at 0:21

I was asked in interview to assume space availability of O(k) and 'n' is very huge. I couldn't tell him O(n) solution as I thought nth_element would need space o(n). Am i wrong ?isn't underlying algorithm is quicksort based for nth_element ? – Manish Baphna Sep 13 '11 at 17:27

Although not very sure about O(n) complexity, but it will be sure to be between O(n) and nLog(n). Also sure to be closer to O(n) than nLog(n). Function is written in Java

```
public int quickSelect(ArrayList<Integer>list, int nthSmallest){
    //Choose random number in range of 0 to array length
    Random random =  new Random();
    //This will give random number which is not greater than length - 1
    int pivotIndex = random.nextInt(list.size() - 1);

    int pivot = list.get(pivotIndex);

    ArrayList<Integer> smallerNumberList = new ArrayList<Integer>();
    ArrayList<Integer> greaterNumberList = new ArrayList<Integer>();

    //Split list into two.
    //Value smaller than pivot should go to smallerNumberList
    //Value greater than pivot should go to greaterNumberList
    //Do nothing for value which is equal to pivot
    for(int i=0; i<list.size(); i++){
        if(list.get(i)<pivot){
            smallerNumberList.add(list.get(i));
        }
        else if(list.get(i)>pivot){
            greaterNumberList.add(list.get(i));
        }
        else{
            //Do nothing
        }
    }

    //If smallerNumberList size is greater than nthSmallest value, nthSmallest
number must be in this list
    if(nthSmallest < smallerNumberList.size()){
        return quickSelect(smallerNumberList, nthSmallest);
    }
    //If nthSmallest is greater than [ list.size() - greaterNumberList.size() ],
nthSmallest number must be in this list
    //The step is bit tricky. If confusing, please see the above loop once again
for clarification.
    else if(nthSmallest > (list.size() - greaterNumberList.size())){
        //nthSmallest will have to be changed here. [ list.size() -
greaterNumberList.size() ] elements are already in
        //smallerNumberList
        nthSmallest = nthSmallest - (list.size() - greaterNumberList.size());
        return quickSelect(greaterNumberList,nthSmallest);
    }
    else{
        return pivot;
    }
}
```

answered Aug 18 '11 at 7:25
prithvi zankat
58 ● 4

Nice coding, +1. But there is no need to use extra space. – Hengameh Jul 24 '15 at 12:47

I implemented finding kth minimimum in n unsorted elements using dynamic programming, specifically tournament method. The execution time is O(n + klog(n)). The mechanism used is listed as one of methods on Wikipedia page about Selection Algorithm (as indicated in one of the posting above). You can read about the algorithm and also find code (java) on my blog page Finding Kth Minimum. In addition the logic can do partial ordering of the list - return first K min (or max) in O(klog(n)) time.

Though the code provided result kth minimum, similar logic can be employed to find kth maximum in O(klog(n)), ignoring the pre-work done to create tournament tree.

edited Aug 14 at 22:58

answered Jan 4 '11 at 5:43
Malkit S. Bhasin
39 ● 3

You can do it in O(n + kn) = O(n) (for constant k) for time and O(k) for space, by keeping track of the k largest elements you've seen.

For each element in the array you can scan the list of k largest and replace the smallest element with the new one if it is bigger.

Warren's priority heap solution is neater though.

answered Oct 30 '08 at 21:17
Rob Walker
33.8k ● 10 ● 82 ● 125

3 This would have a worst case of O(n^2) where you're asked for the smallest item. – Elie Oct 30 '08 at 21:23

2 "Smallest item" means that k=n, so k is no longer constant. – Tyler McHenry Jul 31 '09 at 17:01

Or maybe keep a heap (or reversed heap, or balanced tree) of the largest k you've seen so far `O(n log k)` ...still degenerates to O(nlogn) in case of large k. I'd think it would work fine for small values of k however...possibly faster than some of the other algorithms mentioned here [???] – rogerdpack Jul 25 '16 at 20:24

Read Chapter 9, Medians and Other statistics from Cormen's "Introduction to Algorithms", 2nd Ed. It has an expected linear time algorithm for selection. It's not something that people would randomly come up with in a few minutes.. A heap sort, btw, won't work in O(n), it's O(nlgn).

answered Oct 30 '08 at 21:59
manche

If it uses median of medians my guess is it's the introselect algorithm,. – rogerdpack Jul 26 '16 at 7:18

Find the median of the array in linear time, then use partition procedure exactly as in quicksort to divide the array in two parts, values to the left of the median lesser(<) than than median and to the right greater than (>) median, that too can be done in lineat time, now, go to that part of the array where kth element lies, Now recurrence becomes: T(n) = T(n/2) + cn which gives me O (n) overal.

answered Apr 18 '10 at 18:21
pranjal
416 ● 4 ● 19

There is no need to find median. without median your approach is still fine. – Hengameh Jul 24 '15 at 12:27

And how do you find the median in linear time, dare I ask? ... :) – rogerdpack Jul 25 '16 at 19:25

Sexy quickselect in Python

```
def quickselect(arr, k):
    '''
     k = 1 returns first element in ascending order.
     can be easily modified to return first element in descending order
    '''

    r = random.randrange(0, len(arr))

    a1 = [i for i in arr if i < arr[r]] '''partition'''
    a2 = [i for i in arr if i > arr[r]]

    if k <= len(a1):
        return quickselect(a1, k)
    elif k > len(arr)-len(a2):
        return quickselect(a2, k - (len(arr) - len(a2)))
    else:
        return arr[r]
```

edited Apr 25 '15 at 13:56

answered Jul 22 '14 at 16:52
hoder
112 ● 1 ● 11

Nice solution, except that this returns the kth *smallest* element in an unsorted list. Reversing the comparison operators in the list comprehensions, `a1 = [i for i in arr if i > arr[r]]` and `a2 = [i for i in arr if i < arr[r]]`, will return the kth *largest* element. – gumption Apr 22 '15 at 22:03

From a small benchmark, even on large arrays, it is faster to sort (with `numpy.sort` for `numpy array` or `sorted` for lists) than to use this manual implementation. – Næreen Jun 29 at 11:43

Below is the link to full implementation with quite an extensive explanation how the algorithm for finding Kth element in an unsorted algorithm works. Basic idea is to partition the array like in QuickSort. But in order to avoid extreme cases (e.g. when smallest element is chosen as pivot in every step, so that algorithm degenerates into O(n^2) running time), special pivot selection is applied, called median-of-medians algorithm. The whole solution runs in O(n) time in worst and in average case.

Here is link to the full article (it is about finding Kth *smallest* element, but the principle is the same for finding Kth *largest*):

Finding Kth Smallest Element in an Unsorted Array

edited Jul 24 '15 at 13:53 answered Jul 19 '13 at 9:17

Zoran Horvat
5,528 ● 2 ● 14 ● 31

As per this paper Finding the Kth largest item in a list of n items the following algorithm will take `O(n)` time in worst case.

1. Divide the array in to n/5 lists of 5 elements each.
2. Find the median in each sub array of 5 elements.
3. Recursively find the median of all the medians, lets call it M
4. Partition the array in to two sub array 1st sub-array contains the elements larger than M , lets say this sub-array is a1 , while other sub-array contains the elements smaller then M., lets call this sub-array a2.
5. If k <= |a1|, return selection (a1,k).
6. If k− 1 = |a1|, return M.
7. If k> |a1| + 1, return selection(a2,k −a1 − 1).

Analysis: As suggested in the original paper:

> We use the median to partition the list into two halves(the first half, if `k <= n/2` , and the second half otherwise). This algorithm takes time `cn` at the first level of recursion for some constant `c` , `cn/2` at the next level (since we recurse in a list of size n/2), `cn/4` at the third level, and so on. The total time taken is `cn + cn/2 + cn/4 + .... = 2cn = o(n)` .

Why partition size is taken 5 and not 3?

As mentioned in original paper:

> Dividing the list by 5 assures a worst-case split of 70 − 30. Atleast half of the medians greater than the median-of-medians, hence atleast half of the n/5 blocks have atleast 3 elements and this gives a `3n/10` split, which means the other partition is 7n/10 in worst case. That gives `T(n) = T(n/5)+T(7n/10)+O(n). Since n/5+7n/10 < 1`, the worst-case running time is `O(n)` .

Now I have tried to implement the above algorithm as:

```
public static int findKthLargestUsingMedian(Integer[] array, int k) {
        // Step 1: Divide the list into n/5 lists of 5 element each.
        int noOfRequiredLists = (int) Math.ceil(array.length / 5.0);
        // Step 2: Find pivotal element aka median of medians.
        int medianOfMedian =  findMedianOfMedians(array, noOfRequiredLists);
        //Now we need two lists split using medianOfMedian as pivot. All elements
in list listOne will be grater than medianOfMedian and listTwo will have elements
lesser than medianOfMedian.
        List<Integer> listWithGreaterNumbers = new ArrayList<>(); // elements
greater than medianOfMedian
        List<Integer> listWithSmallerNumbers = new ArrayList<>(); // elements less
than medianOfMedian
        for (Integer element : array) {
            if (element < medianOfMedian) {
                listWithSmallerNumbers.add(element);
            } else if (element > medianOfMedian) {
                listWithGreaterNumbers.add(element);
            }
        }
        // Next step.
        if (k <= listWithGreaterNumbers.size()) return
findKthLargestUsingMedian((Integer[]) listWithGreaterNumbers.toArray(new
Integer[listWithGreaterNumbers.size()]), k);
        else if ((k - 1) == listWithGreaterNumbers.size()) return medianOfMedian;
        else if (k > (listWithGreaterNumbers.size() + 1)) return
findKthLargestUsingMedian((Integer[]) listWithSmallerNumbers.toArray(new
Integer[listWithSmallerNumbers.size()]), k-listWithGreaterNumbers.size()-1);
        return -1;
    }

    public static int findMedianOfMedians(Integer[] mainList, int
noOfRequiredLists) {
        int[] medians = new int[noOfRequiredLists];
        for (int count = 0; count < noOfRequiredLists; count++) {
            int startOfPartialArray = 5 * count;
            int endOfPartialArray = startOfPartialArray + 5;
            Integer[] partialArray = Arrays.copyOfRange((Integer[]) mainList,
startOfPartialArray, endOfPartialArray);
            // Step 2: Find median of each of these sublists.
            int medianIndex = partialArray.length/2;
            medians[count] = partialArray[medianIndex];
        }
        // Step 3: Find median of the medians.
        return medians[medians.length / 2];
    }
```

Just for sake of completion, another algorithm makes use of Priority Queue and takes time `O(nlogn)` .

```
public static int findKthLargestUsingPriorityQueue(Integer[] nums, int k) {
        int p = 0;
        int numElements = nums.length;
        // create priority queue where all the elements of nums will be stored
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

        // place all the elements of the array to this priority queue
        for (int n : nums) {
            pq.add(n);
        }

        // extract the kth largest element
        while (numElements - k + 1 > 0) {
            p = pq.poll();
            k++;
        }

        return p;
    }
```

Both of these algorithms can be tested as:

```
public static void main(String[] args) throws IOException {
        Integer[] numbers = new Integer[]{2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6,
10, 9, 17, 15, 19, 20, 18, 23, 21, 22, 25, 24, 14};
        System.out.println(findKthLargestUsingMedian(numbers, 8));
        System.out.println(findKthLargestUsingPriorityQueue(numbers, 8));
    }
```

As expected output is: `18`
`18`

answered Sep 4 '15 at 9:09
i_am_zero
7,477 ●2 ●34 ●41

Appears to be a description/exact match of introselect's median of medians... – rogerdpack Jul 25 '16 at 19:25

@rogerdpack I have provided the link I have followed. – i_am_zero Jul 26 '16 at 3:58

---

iterate through the list. if the current value is larger than the stored largest value, store it as the largest value and bump the 1-4 down and 5 drops off the list. If not,compare it to number 2 and do the same thing. Repeat, checking it against all 5 stored values. this should do it in O(n)

answered Oct 30 '08 at 21:10
Kevin
4,059 ●7 ●37 ●60

That "bump" is O(n) if you're using an array, or down to O(log n) (I think) if you use a better structure. – Kirk Strauser Oct 30 '08 at 21:11

It needn't be O(log k) - if the list is a linked list then adding the new element to the top and dropping the last element is more like O(2) – Alnitak Oct 30 '08 at 21:14

The bump would be O(k) for an array-backed list, O(1) for an appropriately-linked list. Either way, this sort of question generally assumes it to be of minimal impact compared to n and it introduces no more factors of n. – bobince Oct 30 '08 at 21:16

it would also be O(1) if the bump uses a ring-buffer – Alnitak Oct 30 '08 at 21:18

1 Anyhow, the comment's algorithm is incomplete, it fails to consider an element of n coming in which is the new (eg) second-largest. Worst case behaviour, where each element in n must be compared with each in the highscore table, is O(kn) - but that still probably means O(n) in terms of the question. – bobince Oct 30 '08 at 21:21

---

i would like to suggest one answer

if we take the first k elements and sort them into a linked list of k values

now for every other value even for the worst case if we do insertion sort for rest n-k values even in the worst case number of comparisons will be k*(n-k) and for prev k values to be sorted let it be k*(k-1) so it comes out to be (nk-k) which is o(n)

cheers

answered Jul 31 '09 at 16:51
Aashish bhandari

1 sorting takes nlogn time... the algorithm should run in linear time – MrDatabase Nov 13 '09 at 19:55

Explanation of the median - of - medians algorithm to find the k-th largest integer out of n can be found here: http://cs.indstate.edu/~spitla/presentation.pdf

Implementation in c++ is below:

```
#include <iostream>
#include <vector>
```

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

int findMedian(vector<int> vec){
//    Find median of a vector
    int median;
    size_t size = vec.size();
    median = vec[(size/2)];
    return median;
}

int findMedianOfMedians(vector<vector<int> > values){
    vector<int> medians;

    for (int i = 0; i < values.size(); i++) {
        int m = findMedian(values[i]);
        medians.push_back(m);
    }

    return findMedian(medians);
}

void selectionByMedianOfMedians(const vector<int> values, int k){
//    Divide the list into n/5 lists of 5 elements each
    vector<vector<int> > vec2D;

    int count = 0;
    while (count != values.size()) {
        int countRow = 0;
        vector<int> row;

        while ((countRow < 5) && (count < values.size())) {
            row.push_back(values[count]);
            count++;
            countRow++;
        }
        vec2D.push_back(row);
    }

    cout<<endl<<endl<<"Printing 2D vector : "<<endl;
    for (int i = 0; i < vec2D.size(); i++) {
        for (int j = 0; j < vec2D[i].size(); j++) {
            cout<<vec2D[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;

//    Calculating a new pivot for making splits
    int m = findMedianOfMedians(vec2D);
    cout<<"Median of medians is : "<<m<<endl;

//    Partition the list into unique elements larger than 'm' (call this sublist
L1) and
//    those smaller them 'm' (call this sublist L2)
    vector<int> L1, L2;

    for (int i = 0; i < vec2D.size(); i++) {
        for (int j = 0; j < vec2D[i].size(); j++) {
            if (vec2D[i][j] > m) {
                L1.push_back(vec2D[i][j]);
            }else if (vec2D[i][j] < m){
                L2.push_back(vec2D[i][j]);
            }
        }
    }

//    Checking the splits as per the new pivot 'm'
    cout<<endl<<"Printing L1 : "<<endl;
    for (int i = 0; i < L1.size(); i++) {
        cout<<L1[i]<<" ";
    }

    cout<<endl<<endl<<"Printing L2 : "<<endl;
    for (int i = 0; i < L2.size(); i++) {
        cout<<L2[i]<<" ";
    }

//    Recursive calls
    if ((k - 1) == L1.size()) {
        cout<<endl<<endl<<"Answer :"<<m;
    }else if (k <= L1.size()) {
        return selectionByMedianOfMedians(L1, k);
    }else if (k > (L1.size() + 1)){
        return selectionByMedianOfMedians(L2, k-((int)L1.size())-1);
    }

}

int main()
{
    int values[] = {2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6, 10, 9, 17, 15, 19, 20,
18, 23, 21, 22, 25, 24, 14};

    vector<int> vec(values, values + 25);

    cout<<"The given array is : "<<endl;
    for (int i = 0; i < vec.size(); i++) {
        cout<<vec[i]<<" ";
    }

    selectionByMedianOfMedians(vec, 8);

    return 0;
}
```

answered Feb 28 '13 at 19:50
thrill_seeker_ninja
183 ● 1 ● 7

This solution does not work. You need to sort the array before returning the median for the 5 element case. – Agnishom Chattopadhyay Mar 7 at 10:51

There is also Wirth's selection algorithm, which has a simpler implementation than QuickSelect. Wirth's selection algorithm is slower than QuickSelect, but with some improvements it becomes faster.

In more detail. Using Vladimir Zabrodsky's MODIFIND optimization and the median-of-3 pivot selection and paying some attention to the final steps of the partitioning part of the algorithm, i've came up with the following algorithm (imaginably named "LefSelect"):

```
#define F_SWAP(a,b) { float temp=(a);(a)=(b);(b)=temp; }

# Note: The code needs more than 2 elements to work
float lefselect(float a[], const int n, const int k) {
    int l=0, m = n-1, i=l, j=m;
    float x;

    while (l<m) {
        if( a[k] < a[i] ) F_SWAP(a[i],a[k]);
        if( a[j] < a[i] ) F_SWAP(a[i],a[j]);
        if( a[j] < a[k] ) F_SWAP(a[k],a[j]);

        x=a[k];
        while (j>k & i<k) {
            do i++; while (a[i]<x);
            do j--; while (a[j]>x);

            F_SWAP(a[i],a[j]);
        }
        i++; j--;

        if (j<k) {
            while (a[i]<x) i++;
            l=i; j=m;
        }
        if (k<i) {
            while (x<a[j]) j--;
            m=j; i=l;
        }
    }
    return a[k];
}
```

In benchmarks that i did here, LefSelect is 20-30% faster than QuickSelect.

edited Nov 5 '14 at 0:12

answered Nov 3 '14 at 23:15
estama
31 ● 4

Haskell Solution:

```
kthElem index list = sort list !! index

withShape ~[]     []     = []
withShape ~(x:xs) (y:ys) = x : withShape xs ys

sort []     = []
sort (x:xs) = (sort ls `withShape` ls) ++ [x] ++ (sort rs `withShape` rs)
  where
   ls = filter (<  x)
   rs = filter (>= x)
```

This implements the median of median solutions by using the withShape method to discover the size of a partition without actually computing it.

answered Jan 23 '15 at 23:42
user3585010
75 ● 4

Here is a C++ implementation of Randomized QuickSelect. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, rand() to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

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

int randomPartition(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 a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(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. This function is used by randomPartition()
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) //arr[i] is bigger than arr[j] so swap them
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]); // swap the pivot
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

The worst case time complexity of the above solution is still O(n2).In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is Θ(n)

answered Apr 19 '15 at 11:47
learner
68 ● 8

Nice coding. Thanks for sharing, +1 – Hengameh Jul 24 '15 at 12:53

This is an implementation in Javascript.

If you release the constraint that you cannot modify the array, you can prevent the use of extra memory using two indexes to identify the "current partition" (in classic quicksort style - http://www.nczonline.net/blog/2012/11/27/computer-science-in-javascript-quicksort/).

```
function kthMax(a, k){
    var size = a.length;

    var pivot = a[ parseInt(Math.random()*size) ]; //Another choice could have been
(size / 2)

    //Create an array with all element lower than the pivot and an array with all
element higher than the pivot
    var i, lowerArray = [], upperArray = [];
    for (i = 0; i  < size; i++){
        var current = a[i];

        if (current < pivot) {
            lowerArray.push(current);
        } else if (current > pivot) {
            upperArray.push(current);
        }
    }

    //Which one should I continue with?
    if(k <= upperArray.length) {
        //Upper
        return kthMax(upperArray, k);
    } else {
        var newK = k - (size - lowerArray.length);

        if (newK > 0) {
            ///Lower
            return kthMax(lowerArray, newK);
        } else {
            //None ... it's the current pivot!
            return pivot;
        }
    }
}
```

If you want to test how it perform, you can use this variation:

```
    function kthMax (a, k, logging) {
         var comparisonCount = 0; //Number of comparison that the algorithm uses
         var memoryCount = 0;     //Number of integers in memory that the algorithm
uses
         var _log = logging;

         if(k < 0 || k >= a.length) {
            if (_log) console.log ("k is out of range");
            return false;
         }

         function _kthmax(a, k){
             var size = a.length;
             var pivot = a[parseInt(Math.random()*size)];
             if(_log) console.log("Inputs:", a,  "size="+size, "k="+k,
"pivot="+pivot);

             // This should never happen. Just a nice check in this exercise
             // if you are playing with the code to avoid never ending recursion
             if(typeof pivot === "undefined") {
                 if (_log) console.log ("Ops...");
                 return false;
             }

             var i, lowerArray = [], upperArray = [];
             for (i = 0; i  < size; i++){
                 var current = a[i];
                 if (current < pivot) {
                     comparisonCount += 1;
                     memoryCount++;
                     lowerArray.push(current);
                 } else if (current > pivot) {
                     comparisonCount += 2;
                     memoryCount++;
                     upperArray.push(current);
                 }
             }
             if(_log) console.log("Pivoting:",lowerArray, "*"+pivot+"*",
upperArray);

             if(k <= upperArray.length) {
                 comparisonCount += 1;
                 return _kthmax(upperArray, k);
             } else if (k > size - lowerArray.length) {
                 comparisonCount += 2;
                 return _kthmax(lowerArray, k - (size - lowerArray.length));
             } else {
                 comparisonCount += 2;
                 return pivot;
             }
     /*
      * BTW, this is the logic for kthMin if we want to implement that... ;-)
      *

             if(k <= lowerArray.length) {
                 return kthMin(lowerArray, k);
             } else if (k > size - upperArray.length) {
                 return kthMin(upperArray, k - (size - upperArray.length));
             } else
                 return pivot;
     */
         }

         var result = _kthmax(a, k);
         return {result: result, iterations: comparisonCount, memory: memoryCount};
     }
```

The rest of the code is just to create some playground:

```
    function getRandomArray (n){
        var ar = [];
        for (var i = 0, l = n; i < l; i++) {
            ar.push(Math.round(Math.random() * l))
        }

        return ar;
    }

    //Create a random array of 50 numbers
    var ar = getRandomArray (50);
```

Now, run you tests a few time. Because of the Math.random() it will produce every time different results:

```
    kthMax(ar, 2, true);
    kthMax(ar, 2);
    kthMax(ar, 2);
    kthMax(ar, 2);
    kthMax(ar, 2);
    kthMax(ar, 2);
    kthMax(ar, 34, true);
    kthMax(ar, 34);
    kthMax(ar, 34);
    kthMax(ar, 34);
    kthMax(ar, 34);
    kthMax(ar, 34);
```

If you test it a few times you can see even empirically that the number of iterations is, on average, O(n) ~= constant * n and the value of k does not affect the algorithm.

answered Sep 30 '13 at 8:13
Chris Cinelli
2,280 ● 2 ● 18 ● 31

I came up with this algorithm and seems to be O(n):

Let's say k=3 and we want to find the 3rd largest item in the array. I would create three variables and compare each item of the array with the minimum of these three variables. If array item is greater than our minimum, we would replace the min variable with the item value. We continue the same thing until end of the array. The minimum of our three variables is the 3rd largest item in the array.

```
define variables a=0, b=0, c=0
iterate through the array items
    find minimum a,b,c
    if item > min then replace the min variable with item value
    continue until end of array
the minimum of a,b,c is our answer
```

And, to find Kth largest item we need K variables.

Example: (k=3)

```
[1,2,4,1,7,3,9,5,6,2,9,8]

Final variable values:

a=7 (answer)
b=8
c=9
```

Can someone please review this and let me know what I am missing?

edited Aug 6 '15 at 18:37

answered Aug 6 '15 at 17:23
advncd
1,064 ● 14 ● 19

Here is the implementation of the algorithm eladv suggested(I also put here the implementation with random pivot):

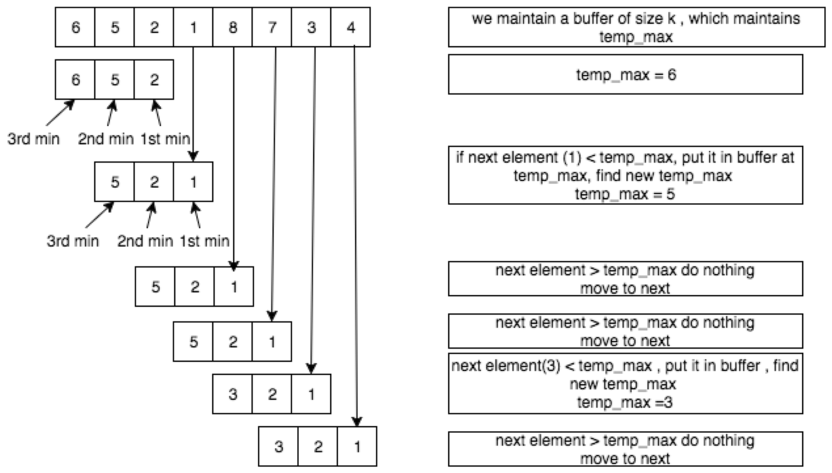
```
public class Median {

    public static void main(String[] s) {

        int[] test = {4,18,20,3,7,13,5,8,2,1,15,17,25,30,16};
        System.out.println(selectK(test,8));

        /*
        int n = 100000000;
        int[] test = new int[n];
        for(int i=0; i<test.length; i++)
            test[i] = (int)(Math.random()*test.length);

        long start = System.currentTimeMillis();
        random_selectK(test, test.length/2);
        long end = System.currentTimeMillis();
        System.out.println(end - start);
        */
    }

    public static int random_selectK(int[] a, int k) {
        if(a.length <= 1)
            return a[0];

        int r = (int)(Math.random() * a.length);
        int p = a[r];

        int small = 0, equal = 0, big = 0;
        for(int i=0; i<a.length; i++) {
            if(a[i] < p) small++;
            else if(a[i] == p) equal++;
            else if(a[i] > p) big++;
        }
        if(k <= small) {
            int[] temp = new int[small];
            for(int i=0, j=0; i<a.length; i++)
                if(a[i] < p)
                    temp[j++] = a[i];
            return random_selectK(temp, k);
        }

        else if (k <= small+equal)
            return p;

        else {
            int[] temp = new int[big];
            for(int i=0, j=0; i<a.length; i++)
                if(a[i] > p)
                    temp[j++] = a[i];
            return random_selectK(temp,k-small-equal);
        }
    }

    public static int selectK(int[] a, int k) {
        if(a.length <= 5) {
            Arrays.sort(a);
            return a[k-1];
        }

        int p = median_of_medians(a);

        int small = 0, equal = 0, big = 0;
        for(int i=0; i<a.length; i++) {
            if(a[i] < p) small++;
            else if(a[i] == p) equal++;
            else if(a[i] > p) big++;
        }

        if(k <= small) {
            int[] temp = new int[small];
            for(int i=0, j=0; i<a.length; i++)
                if(a[i] < p)
                    temp[j++] = a[i];
            return selectK(temp, k);
        }

        else if (k <= small+equal)
            return p;

        else {
            int[] temp = new int[big];
            for(int i=0, j=0; i<a.length; i++)
                if(a[i] > p)
                    temp[j++] = a[i];
            return selectK(temp,k-small-equal);
        }
    }

    private static int median_of_medians(int[] a) {
        int[] b = new int[a.length/5];
        int[] temp = new int[5];
        for(int i=0; i<b.length; i++) {
            for(int j=0; j<5; j++)
                temp[j] = a[5*i + j];
            Arrays.sort(temp);
            b[i] = temp[2];
        }

        return selectK(b, b.length/2 + 1);
    }
}
```

answered Jun 2 '16 at 9:05

TheLogicGuy
314 ● 2 ● 12

How about this kinda approach

Maintain a `buffer of length k` and a `tmp_max`, getting tmp_max is O(k) and is done n times so something like `O(kn)`



6 5 2 1 8 7 3 4
6 5 2
3rd min 2nd min 1st min
5 2 1
3rd min 2nd min 1st min
5 2 1
5 2 1
3 2 1
3 2 1
we maintain a buffer of size k , which maintains temp_max
temp_max = 6
if next element (1) < temp_max, put it in buffer at temp_max, find new temp_max temp_max = 5
next element > temp_max do nothing move to next
next element > temp_max do nothing move to next
next element(3) < temp_max , put it in buffer , find new temp_max temp_max =3
next element > temp_max do nothing move to next

Is it right or am i missing something ?

Although it doesn't beat average case of quickselect and worst case of median statistics method but its pretty easy to understand and implement.

answered Jun 16 '16 at 9:55

Aishwat Singh

2,077 ●7 ●26

---

it is similar to the quickSort strategy, where we pick an arbitrary pivot, and bring the smaller elements to its left, and the larger to the right

```
    public static int kthElInUnsortedList(List<int> list, int k)
    {
        if (list.Count == 1)
            return list[0];

        List<int> left = new List<int>();
        List<int> right = new List<int>();

        int pivotIndex = list.Count / 2;
        int pivot = list[pivotIndex]; //arbitrary

        for (int i = 0; i < list.Count && i != pivotIndex; i++)
        {
            int currentEl = list[i];
            if (currentEl < pivot)
                left.Add(currentEl);
            else
                right.Add(currentEl);
        }

        if (k == left.Count + 1)
            return pivot;

        if (left.Count < k)
            return kthElInUnsortedList(right, k - left.Count - 1);
        else
            return kthElInUnsortedList(left, k);
    }
```

answered Jul 1 '16 at 10:30

Lee.O.

13 ●5

Go to the End of this link : ....

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/

answered Jul 25 '16 at 17:03

Victor

321 ●2 ●3

---

1. Have Priority queue created.
2. Insert all the elements into heap.
3. Call poll() k times.

```
public static int getKthLargestElements(int[] arr)
{
    PriorityQueue<Integer> pq =  new PriorityQueue<>((x , y) -> (y-x));
    //insert all the elements into heap
    for(int ele : arr)
       pq.offer(ele);
    // call poll() k times
    int i=0;
    while(i&lt;k)
     {
        int result = pq.poll();
     }
   return result;
}
```

edited Oct 24 '16 at 4:57 slfan 6,313 ● 18 ● 45 ● 71

answered Oct 24 '16 at 4:17 Malav 411 ● 3 ● 19

What I would do is this:

```
initialize empty doubly linked list l
for each element e in array
    if e larger than head(l)
        make e the new head of l
        if size(l) > k
            remove last element from l

the last element of l should now be the kth largest element
```

You can simply store pointers to the first and last element in the linked list. They only change when updates to the list are made.

Update:

```
initialize empty sorted tree l
for each element e in array
    if e between head(l) and tail(l)
        insert e into l // O(log k)
        if size(l) > k
            remove last element from l

the last element of l should now be the kth largest element
```

edited Oct 30 '08 at 21:31

answered Oct 30 '08 at 21:19 Jasper Bekkers 5,932 ● 24 ● 44

What if e is smaller than head(l)? It could still be larger than the kth largest element, but would never get added to that list. You will need to sort the list of items in order for this to work, in ascending order. – Elie Oct 30 '08 at 21:22

You are right, guess I'll need to think this through some more. :-) – Jasper Bekkers Oct 30 '08 at 21:27

The solution would be to check if e is between head(l) and tail(l) and insert it at the correct position if it is. Making this O(kn). You could make it O(n log k) when using a binary tree that keeps track of the min and max elements. – Jasper Bekkers Oct 30 '08 at 21:30

First we can build a BST from unsorted array which takes O(n) time and from the BST we can find the kth smallest element in O(log(n)) which over all counts to an order of O(n).

answered Sep 17 '13 at 11:30 user2601131 1

For very small values of k (i.e. when k << n), we can get it done in ~O(n) time. Otherwise, if k is comparable to n, we get it in O(nlogn).

answered Oct 5 '12 at 5:21 Sandeep Mathias 37 ● 1

3 You aren't providing anything that hasn't been said already. – Austin Henley Oct 5 '12 at 21:00