K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll 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
```

We have discussed a similar problem to print k largest elements.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple Solution)

A Simple Solution is to sort the given array using a O(nlogn) sorting algorithm like Merge Sort, Heap Sort, etc and return the element at index k-1 in the sorted array. Time Complexity of this solution is O(nLogn).

```
// Simple C++ program to find k'th smallest element
#include<iostream>
#include<algorithm>
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 program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 19};
    int n = sizeof(arr)/sizeof(arr[0]), k = 2;
    cout << "K'th smallest element is " <<  kthSmallest(arr, n, k);
    return 0;
}
```

Run on IDE

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

Method 2 (Using Min Heap – HeapSelect)

We can find k'th smallest element in time complexity better than O(nLogn). A simple optomization is to create a Min Heap of the given n elements and call extractMin() k times.

The following is C++ implementation of above method.

```
// A C++ program to find k'th smallest element using min heap
#include<iostream>
#include<climits>
```

```cpp
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void MinHeapify(int i);  //To minheapify subtree rooted with index i
    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }

    int extractMin();  // extracts root (minimum) element
    int getMin() { return harr[0]; } // Returns minimum
};

MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the minimum vakue.
    int root = harr[0];

    // If there are more than 1 items, move the last item to root
    // and call heapify.
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        MinHeapify(0);
    }
    heap_size--;

    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of n elements: O(n) time
    MinHeap mh(arr, n);

    // Do extract min (k-1) times
    for (int i=0; i<k-1; i++)
        mh.extractMin();

    // Return root
    return mh.getMin();
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 19};
    int n = sizeof(arr)/sizeof(arr[0]), k = 2;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
```

```
}
```

Run on IDE

Output:

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

Time complexity of this solution is O(n + kLogn).

Method 3 (Using Max-Heap)

We can also use Max Heap for finding the k'th smallest element. Following is algorithm.

1) Build a Max-Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the k'th element (arr[k] to arr[n-1]), compare it with root of MH.

……a) If the element is less than the root then make it root and call heapify for MH

……b) Else ignore it.

// The step 2 is O((n-k)*logk)

3) Finally, root of the MH is the kth smallest element.

Time complexity of this solution is O(k + (n-k)*Logk)

The following is C++ implementation of above algorithm

```
// A C++ program to find k'th smallest element using max heap
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Max Heap
class MaxHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of max heap
    int heap_size; // Current number of elements in max heap
public:
    MaxHeap(int a[], int size); // Constructor
    void maxHeapify(int i);  //To maxHeapify subtree rooted with index i
    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }

    int extractMax();  // extracts root (maximum) element
    int getMax() { return harr[0]; } // Returns maximum

    // to replace root with new node x and heapify() new root
    void replaceMax(int x) { harr[0] = x;  maxHeapify(0); }
};

MaxHeap::MaxHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        maxHeapify(i);
        i--;
    }
}

// Method to remove maximum element (or root) from max heap
int MaxHeap::extractMax()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the maximum vakue.
    int root = harr[0];

    // If there are more than 1 items, move the last item to root
    // and call heapify.
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        maxHeapify(0);
    }
    heap_size--;

    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MaxHeap::maxHeapify(int i)
```

```
{
    int l = left(i);
    int r = right(i);
    int largest = i;
    if (l < heap_size && harr[l] > harr[i])
        largest = l;
    if (r < heap_size && harr[r] > harr[largest])
        largest = r;
    if (largest != i)
    {
        swap(&harr[i], &harr[largest]);
        maxHeapify(largest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of first k elements: O(k) time
    MaxHeap mh(arr, k);

    // Process remaining n-k elements.  If current element is
    // smaller than root, replace root with current element
    for (int i=k; i<n; i++)
        if (arr[i] < mh.getMax())
           mh.replaceMax(arr[i]);

    // Return root
    return mh.getMax();
}

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

Run on IDE

Output:

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

Method 4 (QuickSelect)

This is an optimization over method 1 if QuickSort is used as a sorting algorithm in first step. In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the array around it. 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. The worst case time complexity of this method is $O(n^2)$, but it works in O(n) on average.

```
#include<iostream>
#include<climits>
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 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;
}
```

Run on IDE

Output:

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

There are two more solutions which are better than above discussed ones: One solution is to do randomized version of quickSelect() and other solution is worst case linear time algorithm (see the following posts).

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)
K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

References:

http://www.ics.uci.edu/~eppstein/161/960125.html
http://www.cs.rit.edu/~ib/Classes/CS515_Spring12-13/Slides/022-SelectMasterThm.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Heap Searching Order-Statistics

Recommended Posts:

k largest(or smallest) elements in an array | added Min Heap method
Print all nodes less than a value x in a Min Heap.
Adding elements of an array until every element becomes greater than or equal to k
std::make_heap() in C++ STL
Leftist Tree / Leftist Heap
Median of Stream of Running Integers using STL
Merge k sorted linked lists | Set 2 (Using Min Heap)
Largest triplet product in a stream
Find k numbers with most occurrences in the given array
Convert BST to Min Heap

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Share this post!

87 Comments GeeksforGeeks 1 Jae Hyun Lim

Recommend 5 Share Sort by Newest

Join the discussion…

geeknerdface • 6 days ago
Can someone explain how method 3 works exactly?
Reply • Share ›

Hemant Yadav • a month ago
Why to re-invent the wheel :)
http://ide.geeksforgeeks.or...
Reply • Share ›

Gaurav Miglani • 2 months ago
The codes need to be modify when there are repeated elements in array.
Reply • Share ›

scs@ • 2 months ago
What would be the time complexity if I use priority_queue STL for implementing max heap ?
Reply • Share ›

Vinay Kumar ➚ scs@ • 2 months ago
priority queue is implemented using the heap logic, so the complexities of PQ are same as heaps
Reply • Share ›

Shankar Daruga • 5 months ago
Never would have thought that Max Heap could be used for finding the mininum !!!
Reply • Share ›

Anonymous • 5 months ago
can any1 explain to me k-pos+l-1 in right subarray?
Reply • Share ›

Satyajit ➚ Anonymous • 5 months ago
While going right you need to subtract already found elements

so k should be k -(partitionindex+1)+start

e.g. 7, 9, 8, 4, 6, 2, 10, 1, 3, 5 and k = 7

now after one iteration with pivot = 9 (arr[9]=5)

4,2,1,3,5,7,9,8,6,10 with partitionindex= 4

this means we already have 5 smallest elements. Now we go to right and search.

So in right we need (k - (5) + start)th element = 7-5-0 = 2nd element

So in the right recursion we need k+start-(partitionindex+1)

2 Reply • Share ›

ggg • 6 months ago
how is the quickselect method O(n^2) in worst case?
Reply • Share ›

Vivek sharma ➚ ggg • 4 months ago
As quick select is just same as quick sort but it is saving some recursive calls. But if same worst case input came like sorted or reverse sorted order then parition will be called nearly n times and as partition function takes O(n) so T(n) would

be O(n^2).

1 ^ | v • Reply • Share ›

Ashish Jaiswal • 6 months ago

using method 3:
https://github.com/ashishni...

method 4: https://github.com/ashishni...

^ | v • Reply • Share ›

soumyadeep_roy • 6 months ago

I think the 1st method,aka the simple solution won't work if we've duplicate elements in a list as it simply returns the index. Correct me if I am incorrect?

1 ^ | v • Reply • Share ›

Shoaib Khan ➚ soumyadeep_roy • 5 months ago

You are correct

^ | v • Reply • Share ›

soumyadeep_roy ➚ Shoaib Khan • 2 months ago

then what would be the general approach to such problem

^ | v • Reply • Share ›

Cameron Ellis ➚ soumyadeep_roy • a month ago

sort the array. Keep going until you find the third distinct largest number in the array. Does that help?

^ | v • Reply • Share ›

Duplex • 7 months ago

I've also imp. kth largest element based on method 4: http://ideone.com/v3wOkg
Using a non-decreasing order of randomized parify() Feel free to check my code.

^ | v • Reply • Share ›

montu • 9 months ago

In 3rd approach, shouldn't it be k'th largest using max_heap or k'th smallest using min_heap?

^ | v • Reply • Share ›

Shankar Daruga ➚ montu • 5 months ago

It is kth smallest using max heap only. Better way is to think like this. Let us suppose you want kth minimum. That means, you have k elements that are less than the rest of array and you want max of those k elements. Now, he used max heap to store k elements. The trick is in how you add elements to the heap and in the end you get the k minimum elements in that heap. As it is a max heap, you have the root as max of those K minimum elements which is nothing but the kth minimum in the whole array.

2 ^ | v • Reply • Share ›

Learner_187 ➚ Shankar Daruga • 18 days ago

Good Explanation :)

^ | v • Reply • Share ›

Amrutha NK • 10 months ago

Very very simple approach using Selection Sort:

//n - Total array lenth
//k - Kth max element
int kth_ele(int *arr,int n,int k){
int i,j;
for(i=0;i<=k;i++){
for(j=i+1;j<n;j++){ if(arr[i]<arr[j])="" swap(&arr[i],&arr[j]);="" }="" }="" return="" arr[k];="" return="" kth="" max="" element="" }="" time="" complexity="" is="" o(n*k)="" simple="" is="" always="" best!!happy="" coding!!="" :p="">

^ | v • Reply • Share ›

soumyadeep_roy ➚ Amrutha NK • 6 months ago

Considering algorithmic complexity,selection sort would definitely the worst one as it requires O(n^2)

2 ^ | v • Reply • Share ›

Dhruv • 10 months ago

I came across a question at an interview, which was like this...

Find the 3 largest elements at any time in a stream of integers with complexity less than than that of the following code (using extra space is allowed)...

```
int m1 = Integer.MIN_VALUE, m2 = Integer.MIN_VALUE, m3 = Integer.MIN_VALUE;
Random random = new Random(System.currentTimeMillis());
while(true) {
int n = random.nextInt();
if(n > m1) {
m3 = m2;
m2 = m1;
m1 = n;
}
else if(n > m2) {
m3 = m2;
m2 = n;
}
else if(n > m3) {
```

see more

Reply • Share ›

Allen Zhao • a year ago

quick select is wrong, k must be between l+1 and r+1. other than k <= r - l + 1

Reply • Share ›

Prateek Agarwal • a year ago

http://code.geeksforgeeks.o...

Reply • Share ›

Dilip Kondaparthi • a year ago

Method 3 (Using Max-Heap)
An Easy Implementation using Priority_Queue STL, C++ : http://ideone.com/sJLGWV
Time complexity: O(k + (n-k)*Logk)

1 Reply • Share ›

Hrach • a year ago

Since when the 4th smallest of 12, 3, 5, 7, 19 is 5? It is the 2nd smallest and the 4th largest. Your method 3 finds the k-th largest element and not the k-th smallest.

Reply • Share ›

aishwat • a year ago

is this a correct approach http://stackoverflow.com/a/... ?

Reply • Share ›

Kartik Srivastava • a year ago

My solution (complexity: NK)

```
int k_largest(int a[], int n) {
int mx = 0; k_mx = 0;
for (int i = 0; i < k-1; i++)
for (j = 0; j < n-1; j++) {
if (a[j] > k_mx && a[j] < mx)
k_mx = a[j];
}
mx = k_mx; k_mx = 0;

return mx;
}
```

Reply • Share ›

Vivek Agrawal • a year ago

@GeeksforGeeks

in method 3 output should be 12 not 5

Reply • Share ›

kaushik Lele • a year ago

Considering amount of operations done to maintain the order in min/max heap, Is option of Min/Max heap really better than straightforward approach of sort and get kth element ?

straightforward approach of sort-and-get-kth-element ?
Can someone prove ?
1 · Reply · Share ›

Bala Murali Krishna ➚ kaushik Lele • a year ago
Sorting takes O(nlogn). Lets see how heap performs. Firstly building a heap from a array takes O(n) i.e if we are doing bottom-up heapification. Now extracting min (from min-heap) or max(from max-heap) is O(logn). So, for k elements its O(klogn) which beats O(nlogn).
Reply · Share ›

Deepak Dodeja • 2 years ago
I have made little changes in the code of Method 4 for better understanding in line no. 12,19,21,25
This is the url http://code.geeksforgeeks.o...
Correct me if i am wrong..
Reply · Share ›

Vishal Kumar ➚ Deepak Dodeja • 3 days ago
your code is wrong ac to me, since it doesn't handle all the cases while recursing. The index of 'l' will not always be zero while recursing.
Reply · Share ›

sandy • 2 years ago
fix a typo in method 2 : optomization to optimization
Reply · Share ›

Saurabh Singhal • 2 years ago
Method 2|3:
Shouldn't you call [min|max]Heapify() after decreasing the heap size?

Method 4:
Would this not be more intuitive? http://code.geeksforgeeks.o...
Reply · Share ›

Ayushrazz Choudhary • 2 years ago
A simple and small code in O(k) using Python

https://ideone.com/Usy4TT
1 · Reply · Share ›

Bala Murali Krishna ➚ Ayushrazz Choudhary • a year ago
I think finding min or max of a list in python takes O(n). So your solution is O(nk).
Reply · Share ›

Saurabh Singhal ➚ Ayushrazz Choudhary • 2 years ago
Your link will not work for others. It looks like you forgot to generate URL. If you did that, then you forgot to run your program.
Reply · Share ›

David Chen • 2 years ago
Heaps should be an array data structure!! You should not explicitly maintain an extra "heap". Instead the positions of elements in the array should be calculated.
Reply · Share ›

code down • 2 years ago
@GeeksforGeeks

Method 3 step 1 is O(k log k). we are calling Heapify (log k) for first k elements.
Reply · Share ›

code down ➚ code down • 2 years ago
ok i think there is already discussion below

http://www.geeksforgeeks.or...

that it is O(k)
Reply · Share ›

Ping Zhang • 2 years ago

Ping Zhang • 2 years ago

https://pingzhblog.wordpres... here is my solution, please take a look.

Reply • Share ›

Neha • 2 years ago

in method 4, can this be done?

```
if (l<=r)

{

// 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 == k-1)
```

see more

Reply • Share ›

Sowmya Vempati ➚ Neha • 5 months ago

I think this is correct.....Any one please comment.....

Reply • Share ›

Saurabh Singhal ➚ Neha • 2 years ago

Yes, this is more intuitive.

Reply • Share ›

Omi Pandey ➚ Neha • 2 years ago

i think u r correct .. anyone pls comment

Reply • Share ›

Sree Ranjini • 2 years ago

Kth smallest using QUICK SELECT and no recursion

```
int Partition_1(int A[], int start , int end)
{
int pivot = A[start], i = start, j = end;
do
{
do { i++; } while (A[i] <= pivot);
do { j--; } while (A[j] >= pivot);
if (i < j) swap(A[i], A[j]);
} while (i <= j);
swap(A[j], A[start]);
return j;
}

int findKthSmallestElement(int A[], int N, int k)
{
int low = 0, high = N - 1;
do
{
int j = Partition_1(A, low, high);
if (k == j) { return A[k]; }
else if (k < j) high = j; //try to Partition in left half
else low = j + 1; //repeat partition in right half
} while (true);
}
```

Reply • Share ›

Manish Chandel ↗ Sree Ranjini • 2 years ago

it will not work for inupt a[]={1,3,7,4,2,6,8};

updated version

#include<iostream>

#include<conio.h>

#include<algorithm>

using namespace std;

int Partition_1(int A[], int start , int end)

{

int pivot = A[start], i = start, j = end;

do

{

see more

Reply • Share ›

sasha • 2 years ago

return kthSmallest(arr, pos+1, r, k-pos+l-1);

plz explain why in method 4 ,k-pos+l-1 is wriiten instead of k?

Reply • Share ›

Saurabh Saluja ↗ sasha • a year ago

A better solution :

int kthSmallest(int arr[], int l, int r, int k)
{
int pos = partition(arr, l, r);

if (pos == k-1) //REPLACING L WITH 1
return arr[pos];
if (pos > k-1) //REPLACING L WITH 1
return kthSmallest(arr, l, pos-1, k);

return kthSmallest(arr, pos+1, r, k); //REPLACING K-POS+L-1 WITH K
}

1 Reply • Share ›

Load more comments