Searching Algorithms

**What is Searching Algorithm?**

*Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.*

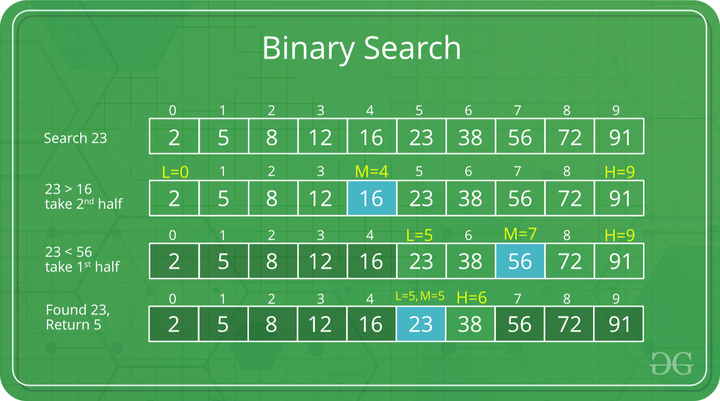
Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: [**Linear Search**](https://www.geeksforgeeks.org/linear-search/).  
   **Linear Search to find the element “20” in a given list of numbers**



*Linear-Search*

1. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: [**Binary Search**](https://www.geeksforgeeks.org/binary-search/).  
   **Binary Search to find the element “23” in a given list of numbers**

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/20221121132716/BinarySearch.png)

*Binary Search*

**Topic :**

* [Searching Algorithms](https://www.geeksforgeeks.org/searching-algorithms/#algo)
* [Comparisons](https://www.geeksforgeeks.org/searching-algorithms/#diff)
* [Library Implementations](https://www.geeksforgeeks.org/searching-algorithms/#implementation)
* [Some standard problems on Searching](https://www.geeksforgeeks.org/searching-algorithms/#standard)

**Searching Algorithm:**

1. [Linear Search](https://www.geeksforgeeks.org/linear-search/)
2. [Sentinel Linear Search](https://www.geeksforgeeks.org/sentinel-linear-search/)
3. [Binary Search](https://www.geeksforgeeks.org/binary-search/)
4. [Meta Binary Search | One-Sided Binary Search](https://www.geeksforgeeks.org/meta-binary-search-one-sided-binary-search/)
5. [Ternary Search](https://www.geeksforgeeks.org/ternary-search/)
6. [Jump Search](https://www.geeksforgeeks.org/jump-search/)
7. [Interpolation Search](https://www.geeksforgeeks.org/interpolation-search/)
8. [Exponential Search](https://www.geeksforgeeks.org/exponential-search/)
9. [Fibonacci Search](https://www.geeksforgeeks.org/fibonacci-search/)
10. [The Ubiquitous Binary Search](https://www.geeksforgeeks.org/the-ubiquitous-binary-search-set-1/)

**Comparisons:**

1. [Linear Search vs Binary Search](https://www.geeksforgeeks.org/linear-search-vs-binary-search/)
2. [Interpolation search vs Binary search](https://www.geeksforgeeks.org/g-fact-84/)
3. [Why is Binary Search preferred over Ternary Search?](https://www.geeksforgeeks.org/binary-search-preferred-ternary-search/)
4. [Is Sentinel Linear Search better than normal Linear Search?](https://www.geeksforgeeks.org/is-sentinel-linear-search-better-than-normal-linear-search/)

**Library Implementations of Searching Algorithms :**

1. [Binary Search functions in C++ STL (binary\_search, lower\_bound and upper\_bound)](https://www.geeksforgeeks.org/binary-search-functions-in-c-stl-binary_search-lower_bound-and-upper_bound/)
2. [Arrays.binarySearch() in Java with examples | Set 1](https://www.geeksforgeeks.org/arrays-binarysearch-java-examples-set-1/)
3. [Arrays.binarySearch() in Java with examples | Set 2 (Search in subarray)](https://www.geeksforgeeks.org/arrays-binarysearch-in-java-with-examples-set-2-search-in-subarray/)
4. [Collections.binarySearch() in Java with Examples](https://www.geeksforgeeks.org/collections-binarysearch-java-examples/)

**Some standard problems on Searching:**

* **Easy:**
  1. [Find the largest three elements in an array](https://www.geeksforgeeks.org/find-the-largest-three-elements-in-an-array/)
  2. [Find the Missing Number](https://www.geeksforgeeks.org/find-the-missing-number/)
  3. [Find the first repeating element in an array of integers](https://www.geeksforgeeks.org/find-first-repeating-element-array-integers/)
  4. [Find the missing and repeating number](https://www.geeksforgeeks.org/find-a-repeating-and-a-missing-number/)
  5. [Search, insert and delete in a sorted array](https://www.geeksforgeeks.org/search-insert-and-delete-in-a-sorted-array/)
  6. [Count 1’s in a sorted binary array](https://www.geeksforgeeks.org/count-1s-sorted-binary-array/)
  7. [Two elements whose sum is closest to zero](https://www.geeksforgeeks.org/two-elements-whose-sum-is-closest-to-zero/)
  8. [Find a pair with the given difference](https://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/)
  9. [k largest(or smallest) elements in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)
  10. [Kth smallest element in a row-wise and column-wise sorted 2D array](https://www.geeksforgeeks.org/kth-smallest-element-in-a-row-wise-and-column-wise-sorted-2d-array/)
  11. [Find common elements in three sorted arrays](https://www.geeksforgeeks.org/find-common-elements-three-sorted-arrays/)
  12. [Ceiling in a sorted array](https://www.geeksforgeeks.org/ceiling-in-a-sorted-array/)
  13. [Floor in a Sorted Array](https://www.geeksforgeeks.org/floor-in-a-sorted-array/)
  14. [Find the maximum element in an array which is first increasing and then decreasing](https://www.geeksforgeeks.org/find-the-maximum-element-in-an-array-which-is-first-increasing-and-then-decreasing/)
  15. [Given an array of of size n and a number k, find all elements that appear more than n/k times](https://www.geeksforgeeks.org/given-an-array-of-of-size-n-finds-all-the-elements-that-appear-more-than-nk-times/)
* **Medium:**
  1. [Find all triplets with zero sum](https://www.geeksforgeeks.org/find-triplets-array-whose-sum-equal-zero/)
  2. [Find the element before which all the elements are smaller than it, and after which all are greater](https://www.geeksforgeeks.org/find-the-element-before-which-all-the-elements-are-smaller-than-it-and-after-which-all-are-greater-than-it/)
  3. [Find the largest pair sum in an unsorted array](https://www.geeksforgeeks.org/find-the-largest-pair-sum-in-an-unsorted-array/)
  4. [K’th Smallest/Largest Element in Unsorted Array](https://www.geeksforgeeks.org/kth-smallest-largest-element-in-unsorted-array/)
  5. [Search an element in a sorted and rotated array](https://www.geeksforgeeks.org/search-an-element-in-a-sorted-and-pivoted-array/)
  6. [Find the minimum element in a sorted and rotated array](https://www.geeksforgeeks.org/find-minimum-element-in-a-sorted-and-rotated-array/)
  7. [Find a peak element](https://www.geeksforgeeks.org/find-a-peak-in-a-given-array/)
  8. [Maximum and minimum of an array using minimum number of comparisons](https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/)
  9. [Find a Fixed Point in a given array](https://www.geeksforgeeks.org/find-a-fixed-point-in-a-given-array/)
  10. [Find the k most frequent words from a file](https://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/)
  11. [Find k closest elements to a given value](https://www.geeksforgeeks.org/find-k-closest-elements-given-value/)
  12. [Given a sorted array and a number x, find the pair in array whose sum is closest to x](https://www.geeksforgeeks.org/given-sorted-array-number-x-find-pair-array-whose-sum-closest-x/)
  13. [Find the closest pair from two sorted arrays](https://www.geeksforgeeks.org/given-two-sorted-arrays-number-x-find-pair-whose-sum-closest-x/)
  14. [Find three closest elements from given three sorted arrays](https://www.geeksforgeeks.org/find-three-closest-elements-from-given-three-sorted-arrays/)
  15. [Binary Search for Rational Numbers without using floating point arithmetic](https://www.geeksforgeeks.org/binary-search-for-rational-numbers-without-using-floating-point-arithmetic/)
* **Hard:**
  1. [Median of two sorted arrays](https://www.geeksforgeeks.org/median-of-two-sorted-arrays/)
  2. [Median of two sorted arrays of different sizes](https://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/)
  3. [Search in an almost sorted array](https://www.geeksforgeeks.org/search-almost-sorted-array/)
  4. [Find position of an element in a sorted array of infinite numbers](https://www.geeksforgeeks.org/find-position-element-sorted-array-infinite-numbers/)
  5. [Given a sorted and rotated array, find if there is a pair with a given sum](https://www.geeksforgeeks.org/given-a-sorted-and-rotated-array-find-if-there-is-a-pair-with-a-given-sum/)
  6. [K’th Smallest/Largest Element in Unsorted Array | Worst case Linear Time](https://www.geeksforgeeks.org/kth-smallest-largest-element-in-unsorted-array-worst-case-linear-time/)
  7. [K’th largest element in a stream](https://www.geeksforgeeks.org/kth-largest-element-in-a-stream/)
  8. [Best First Search (Informed Search)](https://www.geeksforgeeks.org/best-first-search-informed-search/)

**Quick Links:**

1. [‘Practice Problems’ on Searching](https://practice.geeksforgeeks.org/topics/Searching/)
2. [‘Quizzes’ on Searching](https://www.geeksforgeeks.org/algorithms-gq/searching-gq/)

**Easy Questions:**

**Find the largest three distinct elements in an array**

Given an array with all distinct elements, find the largest three elements. Expected time complexity is O(n) and extra space is O(1).

**Examples :**

Input: arr[] = {10, 4, 3, 50, 23, 90}  
Output: 90, 50, 23

**Method 1:**

**Algorithm:**

1) Initialize the largest three elements as minus infinite.  
 first = second = third = -∞

2) Iterate through all elements of array.  
 a) Let current array element be x.  
 b) If (x > first)  
 {  
 // This order of assignment is important  
 third = second  
 second = first  
 first = x   
 }  
 c) Else if (x > second and x != first)  
 {  
 third = second  
 second = x   
 }  
 d) Else if (x > third and x != second)  
 {  
 third = x   
 }

3) Print first, second and third.

# Python3 code to find largest three

# elements in an array

**import** sys

# Function to print three largest

# elements

**def** print3largest(arr, arr\_size):

    # There should be atleast three

    # elements

**if** (arr\_size < 3):

**print**(" Invalid Input ")

**return**

    third **=** first **=** second **= -**sys.maxsize

**for** i **in** range(0, arr\_size):

        # If current element is greater

        # than first

**if** (arr[i] > first):

            third **=** second

            second **=** first

            first **=** arr[i]

        # If arr[i] is in between first

        # and second then update second

**elif** (arr[i] > second):

            third **=** second

            second **=** arr[i]

**elif** (arr[i] > third):

            third **=** arr[i]

**print**("Three largest elements are",

                  first, second, third)

# Driver program to test above function

arr **=** [12, 13, 1, 10, 34, 1]

n **=** len(arr)

print3largest(arr, n)

# This code is contributed by Smitha Dinesh Semwal

# and edited by Ayush Singla(@ayusin51).

**Find the Missing Number**

Given an array **arr[]** of size **N-1**with integers in the range of **[1, N]**, the task is to find the missing number from the first **N** integers.

**Note:** There are no duplicates in the list.

**Examples:**

***Input:****arr[] = {1, 2, 4, 6, 3, 7, 8}, N = 8*

***Output:****5*

***Explanation:****The missing number between 1 to 8 is 5*

***Input:****arr[] = {1, 2, 3, 5}, N = 5*

***Output:****4*

***Explanation:****The missing number between 1 to 5 is 4*

**Approach 1 (Using**[**Hashing**](https://www.geeksforgeeks.org/hashing-set-1-introduction/)**):**The idea behind the following approach is

*The numbers will be in the range****(1, N)****, an array of size****N****can be maintained to keep record of the elements present in the given array*

* Create a temp array **temp[]** of size**n + 1** with all initial values as 0.
* Traverse the input array **arr[],** and do following for each **arr[i]**
* if(temp[arr[i]] == 0) temp[arr[i]] = 1
* Traverse **temp[]** and output the array element having value as 0 (This is the missing element).

Below is the implementation of the above approach:

# Find Missing Element

**def** findMissing(arr, N):

    # create a list of zeroes

    temp **=** [0] **\*** (N**+**1)

**for** i **in** range(0, N):

        temp[arr[i] **-** 1] **=** 1

**for** i **in** range(0, N**+**1):

**if**(temp[i] **==** 0):

            ans **=** i **+** 1

**print**(ans)

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [1, 2, 3, 5]

    N **=** len(arr)

    # Function call

    findMissing(arr, N)

    # This code is contributed by nikhilm2302

**Find the first repeating element in an array of integers**

Given an array of integers **arr[]**, The task is to find the index of first repeating element in it i.e. the element that occurs more than once and whose index of the first occurrence is the smallest.

**Examples:**

***Input:****arr[] = {10, 5, 3, 4, 3, 5, 6}*

***Output:****5*

***Explanation:****5 is the first element that repeats*

***Input:****arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10}*

***Output:****6*

***Explanation:****6 is the first element that repeats*

**Naive Approach:**Below is the idea to solve the problem

*Run two nested loops, the outer loop picks an element one by one, and the inner loop checks whether the element is repeated or not. Once a repeating element is found, break the loops and print the element.*

**Time Complexity:**O(N2)

**Auxiliary Space:**O(1)

**Find the first repeating element in an array of integers using sorting:**

Below is the idea to solve the problem.

*Store the elements of****arr[]****in a duplicate array****temp[]****, sort temp[] and traverse arr[] from****0****to****N – 1****, Simultaneously check the count of this element in****temp[]****and if the current element****arr[i]****has more than one occurrence then return****arr[i]****.*

Follow the steps below to Implement the idea:

* Copy the given array to an auxiliary array **temp[]**and sort temp array.
* Traverse the input array arr[] from **0**to **N – 1**.
* For every element, [count its occurrences in temp[] using binary search](https://www.geeksforgeeks.org/count-number-of-occurrences-in-a-sorted-array/).
* If the count of occurrence of current element is more than one, then return the current element.
* If no repeating element is found print “No Repeating Number Found”.

**Time complexity:** O(NlogN).

**Auxiliary Space:**O(N)

**Find the first repeating element in an array of integers using**[Hashset](http://www.geeksforgeeks.org/hashset-in-java/)

Below is the idea to solve the problem

*The idea is to traverse the given array arr[] from****right to left****and update the minimum index whenever, an already visited element has been found. To check if the element was already visited [Hashset](http://www.geeksforgeeks.org/hashset-in-java/)can be used.*

Follow the steps below to implement the idea:

* Initialize an empty [Hashset](http://www.geeksforgeeks.org/hashset-in-java/)**myset**and a variable **min**with **-1**.
* Run a for loop for each index of array **arr[]**from **N – 1**to **0**.
* If the current element is present in **myset**then update min with **i**.
* Else insert arr[i] in myset.
* Return **min**.

Below is the implementation of the above approach.

# Python3 program to find first repeating

# element in arr[]

# This function prints the first repeating

# element in arr[]

**def** printFirstRepeating(arr, n):

    # Initialize index of first repeating element

    Min **= -**1

    # Creates an empty hashset

    myset **=** dict()

    # Traverse the input array from right to left

**for** i **in** range(n **-** 1, **-**1, **-**1):

        # If element is already in hash set,

        # update Min

**if** arr[i] **in** myset.keys():

            Min **=** i

**else**:  # Else add element to hash set

            myset[arr[i]] **=** 1

    # Print the result

**if** (Min !**= -**1):

        print("The first repeating element is",

              arr[Min])

**else**:

        print("There are no repeating elements")

# Driver Code

arr **=** [10, 5, 3, 4, 3, 5, 6]

n **=** len(arr)

printFirstRepeating(arr, n)

# This code is contributed by Mohit kumar 29

**Find the missing and repeating number**

Given an unsorted array of size n. Array elements are in the range of 1 to n. One number from set {1, 2, …n} is missing and one number occurs twice in the array. Find these two numbers.

**Examples:**

***Input:****arr[] = {3, 1, 3}*

***Output:****Missing = 2, Repeating = 3*

***Explanation:****In the array, 2 is missing and 3 occurs twice*

***Input:****arr[] = {4, 3, 6, 2, 1, 1}*

***Output:****Missing = 5, Repeating = 1*

[Recommended Practice Find Missing And RepeatingTry It!](https://practice.geeksforgeeks.org/problems/find-missing-and-repeating2512/1/)

**Below are various methods to solve the problems:**

**Method 1 (Use Sorting)**

**Approach:**

* Sort the input array.
* Traverse the array and check for missing and repeating.

**def** printTwoElements(arr,  n):

    # sorting the array

    arr.sort()

    print("The repeating element is",end**=**" ")

**for** i **in** range(0, n **-** 1):

**if**(arr[i] **==** arr[i **+** 1]):

**print**(arr[i])

**break**

    print("and the missing element is",end**=**" ")

**for** i **in** range(1, n **+** 1):

**if**(i !**=** arr[i **-** 1]):

            print(i)

**break**

arr **=** [7, 3, 4, 5, 5, 6, 2]

n **=** len(arr)

printTwoElements(arr, n)

# This code is contributed by akashish\_\_

# Python3 code to Find the repeating

# and the missing elements

**def** printTwoElements( arr, size):

**for** i **in** range(size):

**if** arr[abs(arr[i])**-**1] > 0:

            arr[abs(arr[i])**-**1] **= -**arr[abs(arr[i])**-**1]

**else**:

            print("The repeating element is ", abs(arr[i]))

**for** i **in** range(size):

**if** arr[i]>0:

**print**("and the missing element is ", i **+** 1)

# Driver program to test above function \*/

arr **=** [7, 3, 4, 5, 5, 6, 2]

n **=** len(arr)

printTwoElements(arr, n)

# This code is contributed by "Abhishek Sharma 44"

# Python3 code to Find the repeating

# and the missing elements

**def** printTwoElements( arr, size):

**for** i **in** range(size):

**if** arr[abs(arr[i])**-**1] > 0:

            arr[abs(arr[i])**-**1] **= -**arr[abs(arr[i])**-**1]

**else**:

            print("The repeating element is ", abs(arr[i]))

**for** i **in** range(size):

**if** arr[i]>0:

**print**("and the missing element is ", i **+** 1)

# Driver program to test above function \*/

arr **=** [7, 3, 4, 5, 5, 6, 2]

n **=** len(arr)

printTwoElements(arr, n)

# This code is contributed by "Abhishek Sharma 44"

**Time Complexity:** O(n)

**Auxiliary Space**: O(1) as it is using constant variables

Thanks to **Manish Mishra** for suggesting this method.

**Method 4 (Make two equations)**

**Approach:**

* Let x be the missing and y be the repeating element.
* Get the sum of all numbers using formula **S = n(n+1)/2 – x + y**
* Get product of all numbers using formula **P = 1\*2\*3\*…\*n \* y / x**
* The above two steps give us two equations, we can solve the equations and get the values of x and y.

**Time Complexity:** O(n)

Thanks to **disappearedng** for suggesting this solution.

**Note:** This method can cause arithmetic overflow as we calculate the product and sum of all array elements.

**Method 5 (Use XOR)**

**Approach:**

* Let x and y be the desired output elements.
* Calculate the XOR of all the array elements.

***xor1 = arr[0]^arr[1]^arr[2]…..arr[n-1]***

* XOR the result with all numbers from 1 to n

***xor1 = xor1^1^2^…..^n***

* In the result *xor1*, all elements would nullify each other except x and y. All the bits that are set in *xor1* will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of *xor1*and divide the elements of the array in two sets – one set of elements with the same bit set and another set with the same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in the first set, we will get x, and by doing the same in the other set we will get y.

Below is the implementation of the above approach:

# Python3 program to find the repeating

# and missing elements

# The output of this function is stored

# at x and y

**def** getTwoElements(arr, n):

**global** x, y

    x **=** 0

    y **=** 0

    # Will hold xor of all elements

    # and numbers from 1 to n

    xor1 **=** arr[0]

    # Get the xor of all array elements

**for** i **in** range(1, n):

        xor1 **=** xor1 ^ arr[i]

    # XOR the previous result with numbers

    # from 1 to n

**for** i **in** range(1, n **+** 1):

        xor1 **=** xor1 ^ i

    # Will have only single set bit of xor1

    set\_bit\_no **=** xor1 & ~(xor1 **-** 1)

    # Now divide elements into two

    # sets by comparing a rightmost set

    # bit of xor1 with the bit at the same

    # position in each element. Also,

    # get XORs of two sets. The two

    # XORs are the output elements.

    # The following two for loops

    # serve the purpose

**for** i **in** range(n):

**if** (arr[i] & set\_bit\_no) !**=** 0:

            # arr[i] belongs to first set

            x **=** x ^ arr[i]

**else**:

            # arr[i] belongs to second set

            y **=** y ^ arr[i]

**for** i **in** range(1, n **+** 1):

**if** (i & set\_bit\_no) !**=** 0:

            # i belongs to first set

            x **=** x ^ i

**else**:

            # i belongs to second set

            y **=** y ^ i

    # x and y hold the desired

    # output elements

# Driver code

arr **=** [ 1, 3, 4, 5, 5, 6, 2 ]

n **=** len(arr)

getTwoElements(arr, n)

print("The missing element is", x,

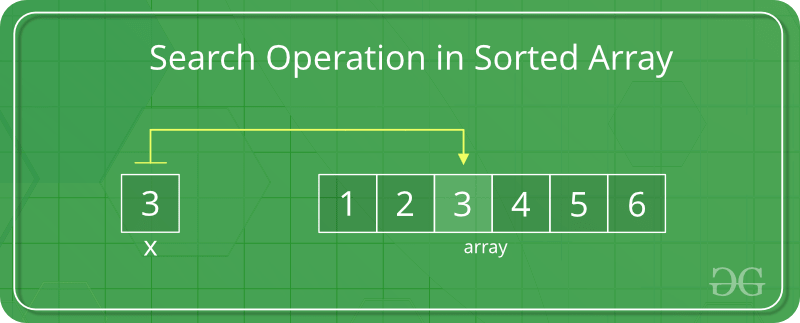
      "and the repeating number is", y)

# This code is contributed by stutipathak31jan

**Search, insert and delete in a sorted array**

**How to Search in a Sorted Array?**

In a sorted array, the search operation can be performed by using [binary search](https://www.geeksforgeeks.org/binary-search/).



Below is the implementation of the above approach:

# python 3  program to implement

# binary search in sorted array

**def** binarySearch(arr, low, high, key):

    mid **=** (low **+** high)**/**2

**if** (key **==** arr[int(mid)]):

**return** mid

**if** (key > arr[int(mid)]):

**return** binarySearch(arr,

                            (mid **+** 1), high, key)

**if** (key < arr[int(mid)]):

**return** binarySearch(arr, low, (mid**-**1), key)

**return** 0

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Let us search 3 in below array

    arr **=** [5, 6, 7, 8, 9, 10]

    n **=** len(arr)

    key **=** 10

    # Function call

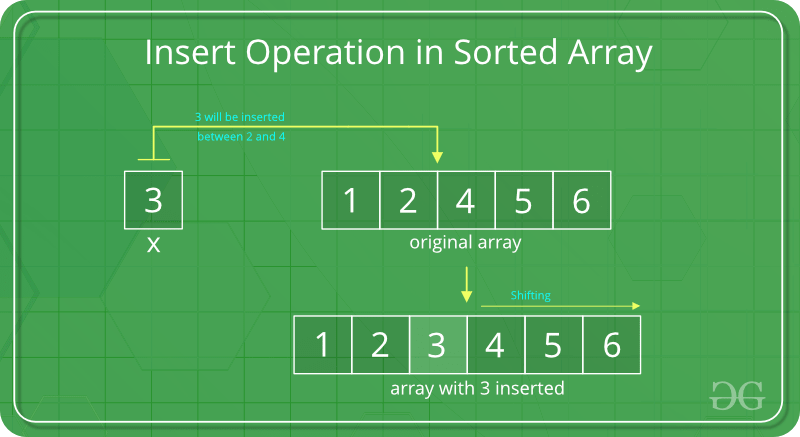
    print("Index:", int(binarySearch(arr, 0, n**-**1, key)))

# This code is contributed by

# Smitha Dinesh Semwal

**How to Insert in a Sorted Array?**

In a sorted array, a search operation is performed for the possible position of the given element by using [Binary search,](https://www.geeksforgeeks.org/binary-search/) and then an insert operation is performed followed by shifting the elements. And in an unsorted array, the insert operation is faster as compared to the sorted array because we don’t have to care about the position at which the element is placed.



# Python3 program to implement insert

# operation in an sorted array.

# Inserts a key in arr[] of given capacity.

# n is current size of arr[]. This function

# returns n+1 if insertion is successful, else n.

**def** insertSorted(arr, n, key, capacity):

    # Cannot insert more elements if n is

    # already more than or equal to capacity

**if** (n >**=** capacity):

**return** n

    i **=** n **-** 1

**while** i >**=** 0 **and** arr[i] > key:

        arr[i **+** 1] **=** arr[i]

        i **-=** 1

    arr[i **+** 1] **=** key

**return** (n **+** 1)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [12, 16, 20, 40, 50, 70]

**for** i **in** range(20):

        arr.append(0)

    capacity **=** len(arr)

    n **=** 6

    key **=** 26

**print**("Before Insertion: ", end**=**" ")

**for** i **in** range(n):

**print**(arr[i], end**=**" ")

    # Function call

    n **=** insertSorted(arr, n, key, capacity)

    print("\nAfter Insertion: ", end**=**"")

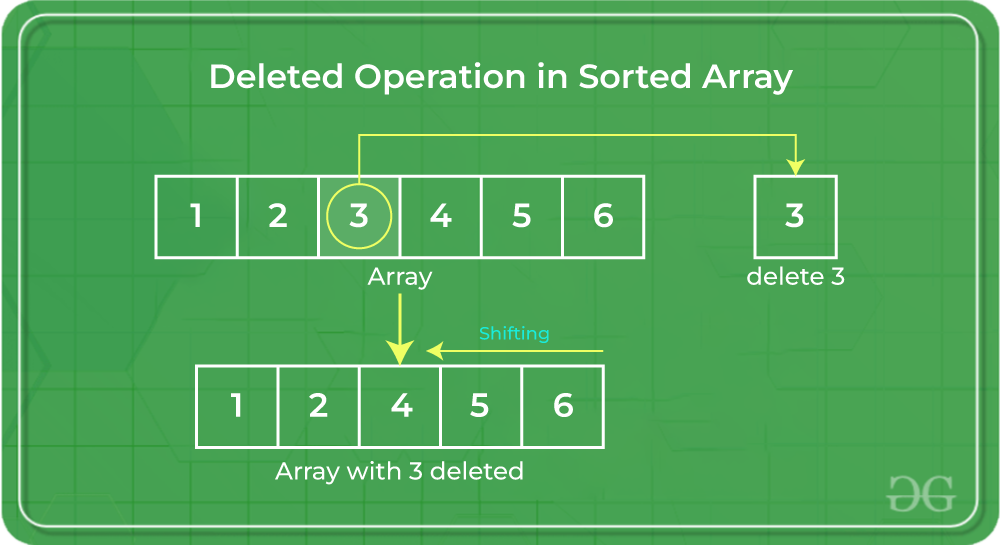
**for** i **in** range(n):

**print**(arr[i], end**=**" ")

# This code is contributed by Mohit Kumar

**How to Delete in a Sorted Array?**

In the delete operation, the element to be deleted is searched using binary search, and then the delete operation is performed followed by shifting the elements.



*Performing delete operation*

Below is the implementation of the above approach:

# Python program to implement delete operation in a

# sorted array

# /\* Function to delete an element \*/

**def** deleteElement(arr, n, key):

    # Find position of element to be deleted

    pos **=** binarySearch(arr, 0, n **-** 1, key)

**if** (pos **== -**1):

**print**("Element not found")

**return** n

    # Deleting element

**for** i **in** range(pos, n **-** 1):

        arr[i] **=** arr[i **+** 1]

**return** n **-** 1

# To search a key to be deleted

**def** binarySearch(arr, low, high, key):

**if** (high < low):

**return -**1

    mid **=** (low **+** high) **//** 2

**if** (key **==** arr[mid]):

**return** mid

**if** (key > arr[mid]):

**return** binarySearch(arr, (mid **+** 1), high, key)

**return** binarySearch(arr, low, (mid **-** 1), key)

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [10, 20, 30, 40, 50]

    n **=** len(arr)

    key **=** 30

    print("Array before deletion")

**for** i **in** range(n):

**print**(arr[i], end**=**" ")

    # Function call

    n **=** deleteElement(arr, n, key)

**print**("\n\nArray after deletion")

**for** i **in** range(n):

        print(arr[i], end**=**" ")

# This code is contributed by shubhamsingh10

**Count 1’s in a sorted binary array**

Given a binary array **arr[]**of size **N,** which is sorted in **non-increasing order**, count the number of **1’s** in it.

**Examples:**

***Input:****arr[] = {1, 1, 0, 0, 0, 0, 0}*

***Output:****2*

***Input:****arr[] = {1, 1, 1, 1, 1, 1, 1}*

***Output:****7*

***Input:****arr[] = {0, 0, 0, 0, 0, 0, 0}*

***Output:****0*

[Please try your approach on IDE first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

[Try It!](https://ide.geeksforgeeks.org/)

**Naive approach:**

A simple solution is to linearly traverse the array until we find the 1’s in the array and keep count of 1s. If the array element becomes 0 then return the count of 1’s.

**Time Complexity:** O(N).

**Auxiliary Space:** O(1)

**Count 1’s in a sorted binary array using Binary search recursively:**

*We can use*[*Binary Search*](https://www.geeksforgeeks.org/binary-search/)*to find count in O(Logn) time. The idea is to look for the last occurrence of 1 using Binary Search. Once we find the index’s last occurrence, we return index + 1 as count.*

**Follow the steps below to implement the above idea:**

* Do while low <= high:
* Calculate mid using low + (high – low) / 2.
* Check if the element at mid index is the last 1
* If the element is not last 1, move the low to right side recursively and return the result received from it.
* Otherwise, move the low to left recursively and return the result received from it.

The following is the implementation of the above idea.

# Python program to count one's in a boolean array

# Returns counts of 1's in arr[low..high].  The array is

# assumed to be sorted in non-increasing order

**def** countOnes(arr, low, high):

**if** high >**=** low:

        # get the middle index

        mid **=** low **+** (high**-**low)**//**2

        # check if the element at middle index is last 1

**if** ((mid **==** high **or** arr[mid**+**1] **==** 0) **and** (arr[mid] **==** 1)):

**return** mid**+**1

        # If element is not last 1, recur for right side

**if** arr[mid] **==** 1:

**return** countOnes(arr, (mid**+**1), high)

        # else recur for left side

**return** countOnes(arr, low, mid**-**1)

**return** 0

# Driver Code

arr **=** [1, 1, 1, 1, 0, 0, 0]

print("Count of 1's in given array is", countOnes(arr, 0, len(arr)**-**1))

# This code is contributed by \_\_Devesh Agrawal\_\_

**Count 1’s in a sorted binary array using binary search iteratively:**

Follow the steps below for the implementation:

* Do while low <= high
* Calculate the middle index say **mid**
* Check if arr[mid] is less than 1 then move the high to left side (i.e, high = mid – 1)
* If the element is not last 1 then move the low to the right side (i.e, low = mid + 1)
* Check if the element at the middle index is last 1 then return mid + 1
* Otherwise move to low to right (i.e, low = mid + 1)

**def** countOnes(arr, n):

    low **=** 0

    high **=** n **-** 1

**while** (low <**=** high):  # get the middle index

        mid **=** (low **+** high) **//** 2

        # else recur for left side

**if** (arr[mid] < 1):

            high **=** mid **-** 1

        # If element is not last 1, recur for right side

**elif**(arr[mid] > 1):

            low **=** mid **+** 1

**else**:

            # check if the element at middle index is last 1

**if** (mid **==** n **-** 1 **or** arr[mid **+** 1] !**=** 1):

**return** mid **+** 1

**else**:

                low **=** mid **+** 1

**return** 0

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [1, 1, 1, 1, 0, 0, 0]

    n **=** len(arr)

    print("Count of 1's in given array is ", countOnes(arr, n))

# This code is contributed by umadevi9616

**Two elements whose sum is closest to zero**

**Question:** An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

For the below array, program should print -80 and 85.

[Sum of two elements with sum nearest to zero](https://practice.geeksforgeeks.org/problems/two-numbers-with-sum-closest-to-zero1737/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/two-numbers-with-sum-closest-to-zero1737/1/)

**METHOD 1 (Simple)**

For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

**Implementation**:

# Python3 code to find Two elements

# whose sum is closest to zero

**def** minAbsSumPair(arr,arr\_size):

    inv\_count **=** 0

    # Array should have at least

    # two elements

**if** arr\_size < 2:

        print("Invalid Input")

**return**

    # Initialization of values

    min\_l **=** 0

    min\_r **=** 1

    min\_sum **=** arr[0] **+** arr[1]

**for** l **in** range (0, arr\_size **-** 1):

**for** r **in** range (l **+** 1, arr\_size):

            sum **=** arr[l] **+** arr[r]

**if** abs(min\_sum) > abs(sum):

                min\_sum **=** sum

                min\_l **=** l

                min\_r **=** r

**print**("The two elements whose sum is minimum are",

            arr[min\_l], "and ", arr[min\_r])

# Driver program to test above function

arr **=** [1, 60, **-**10, 70, **-**80, 85]

minAbsSumPair(arr, 6);

# This code is contributed by Smitha Dinesh Semwal

**STL implementation of Method-2**:

**Algorithm**

1) Sort all the elements of the input array using their absolute values.

2) Check absolute sum of arr[i-1] and arr[i] if their absolute sum is less than min update min with their absolute value.

3) Use two variables to store the index of the elements.

**Implementation:**

# Python3 implementation using STL

**import** sys

**def** findMinSum(arr, n):

**for** i **in** range(1, n):

        # Modified to sort by absolute values

**if** (**not** abs(arr[i **-** 1]) < abs(arr[i])):

            arr[i **-** 1], arr[i] **=** arr[i], arr[i **-** 1]

    Min **=** sys.maxsize

    x **=** 0

    y **=** 0

**for** i **in** range(1, n):

        # Absolute value shows how

        # close it is to zero

**if** (abs(arr[i **-** 1] **+** arr[i]) <**=** Min):

            # If found an even close value

            # update min and store the index

            Min **=** abs(arr[i **-** 1] **+** arr[i])

            x **=** i **-** 1

            y **=** i

**print**("The two elements whose sum is minimum are",

          arr[x], "and", arr[y])

# Driver code

arr **=** [ 1, 60, **-**10, 70, **-**80, 85 ]

n **=** len(arr)

findMinSum(arr, n)

# This code is contributed by avanitrachhadiya2155

**Find a pair with the given difference**

Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.

**Examples:**

***Input:****arr[] = {5, 20, 3, 2, 50, 80}, n = 78*

***Output:****Pair Found: (2, 80)*

***Input:****arr[] = {90, 70, 20, 80, 50}, n = 45*

***Output:****No Such Pair*

Recommended Practice

[Find Pair Given Difference](https://practice.geeksforgeeks.org/problems/find-pair-given-difference1559/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/find-pair-given-difference1559/1/)

**Method 1:**The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n. Time complexity of this method is O(n2).

**Method 2:**We can use sorting and Binary Search to improve time complexity to O(nLogn). The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element arr[i], binary search for arr[i] + n in arr[i+1..n-1]. If the element is found, return the pair. Both first and second steps take O(nLogn). So overall complexity is O(nLogn).

**Method 3:** The second step of the Method -2 can be improved to O(n). The first step remains the same. The idea for the second step is to take two index variables i and j, and initialize them as 0 and 1 respectively. Now run a linear loop. If arr[j] – arr[i] is smaller than n, we need to look for greater arr[j], so increment j. If arr[j] – arr[i] is greater than n, we need to look for greater arr[i], so increment i. Thanks to Aashish Barnwal for suggesting this approach.

**The following code is only for the second step of the algorithm, it assumes that the array is already sorted.**

# Python program to find a pair with the given difference

# The function assumes that the array is sorted

**def** findPair(arr,n):

    size **=** len(arr)

    # Initialize positions of two elements

    i,j **=** 0,1

    # Search for a pair

**while** i < size **and** j < size:

**if** i !**=** j **and** arr[j]**-**arr[i] **==** n:

**print** "Pair found (",arr[i],",",arr[j],")"

**return** True

**elif** arr[j] **-** arr[i] < n:

            j**+=**1

**else**:

            i**+=**1

    print "No pair found"

**return** False

# Driver function to test above function

arr **=** [1, 8, 30, 40, 100]

n **=** 60

findPair(arr, n)

# This code is contributed by Devesh Agrawal

# Python program to find a pair with the given difference

# The function assumes that the array is sorted

**def** findPair(arr, size, n):

    mpp **=** {}

**for** i **in** range(size):

**if** arr[i] **in** mpp.keys():

             mpp[arr[i]] **+=** 1

**if**(n **==** 0 **and** mpp[arr[i]] > 1):

**return** true;

**else**:

             mpp[arr[i]] **=** 1

**if**(n **==** 0):

**return** false;

**for** i **in** range(size):

**if** n **+** arr[i] **in** mpp.keys():

**print**("Pair Found: (" **+** str(arr[i]) **+** ", " **+** str(n **+** arr[i]) **+** ")")

**return** True

**print**("No Pair found")

**return** False

# Driver program to test above function

arr **=** [ 1, 8, 30, 40, 100 ]

size **=** len(arr)

n **= -**60

findPair(arr, size, n)

# This code is contributed by shinjanpatra

*From <*[*https://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/*](https://www.geeksforgeeks.org/find-a-pair-with-the-given-difference/)*>*

**k largest(or smallest) elements in an array**

Write an efficient program for printing K largest elements in an array. Elements in an array can be in any order

**Examples:**

***Input:****[1, 23, 12, 9, 30, 2, 50], K = 3*

***Output:****50, 30, 23*

***Input:****[11, 5, 12, 9, 44, 17, 2], K = 2*

***Output:****44, 17*

[k largest elements](https://practice.geeksforgeeks.org/problems/k-largest-elements4206/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/k-largest-elements4206/1/)

**Naive Approaches:** To solve the problem follow the below ideas:

**1. Using**[Bubble sort](https://www.geeksforgeeks.org/bubble-sort/)**:**

Follow the below steps to solve the problem:

* Modify Bubble Sort to run the outer loop at most K times.
* Print the last K elements of the array obtained in step 1

**Time Complexity:** O(N \* K)

Thanks to Shailendra for suggesting this approach.

**Note:**Like Bubble sort, other sorting algorithms like [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) can also be modified to get the K largest elements.

**2. Using temporary array:**

Follow the below steps to solve the problem:

* Store the first K elements in a temporary array temp[0..K-1]
* Find the smallest element in temp[], and let the smallest element be min
* For each element x in arr[K] to arr[N-1]. If x is greater than the min, remove min from temp[] and insert x
* Then, determine the new min from temp[]
* Print final K elements of temp[]

**Time Complexity:** O((N – K) \* K). If we want the output sorted then O((N – K) \* K + K \* log(K))

Thanks to nesamani1822 for suggesting this method.

**K largest(or smallest) elements in an array using sorting:**

To solve the problem follow the below idea:

*We can sort the input array in descending order so that the first K elements in the array are the K largest elements*

Follow the below steps to solve the problem:

* Sort the elements in descending order
* Print the first K numbers of the sorted array

Below is the implementation of the above approach

''' Python3 code for k largest elements in an array'''

**def** kLargest(arr, k):

    # Sort the given array arr in reverse

    # order.

    arr.sort(reverse**=**True)

    # Print the first kth largest elements

**for** i **in** range(k):

**print**(arr[i], end**=**" ")

# Driver code

arr **=** [1, 23, 12, 9, 30, 2, 50]

# n = len(arr)

k **=** 3

kLargest(arr, k)

# This code is contributed by shreyanshi\_arun.

**Efficient Approaches:** To solve the problem follow the below ideas:

**1. Using**[Max-Heap](https://www.geeksforgeeks.org/difference-between-min-heap-and-max-heap/)**:**

Follow the below steps to solve the problem:

* Build a Max Heap
* Use Extract Max K times to get K maximum elements from the Max Heap

**Time complexity:** O(N \* log(N) + K \* log(N))

**2. Using order Statistics:**

Follow the below steps to solve the problem:

* Use an order statistic algorithm to find the Kth largest element. Please [see the topic selection in worst-case linear time](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/)
* Use the [QuickSort](https://www.geeksforgeeks.org/quick-sort/)Partition algorithm to partition around the Kth largest number
* Sort the K-1 elements (elements greater than the Kth largest element)  
  **Note:**This step is needed only if the sorted output is required

**Time complexity:** O(N) if we don’t need the sorted output, otherwise O(N + K \* log(K))

Thanks to Shilpi for suggesting the first two approaches.

**K largest(or smallest) elements in an array using Min-Heap:**

To solve the problem follow the below idea:

*We can create a Min-Heap of size K and then compare the root of the Min-Heap with other elements and if it is greater than the root, then swap the value of the root and heapify the heap. This will help us to get the K largest elements in the end*

Follow the below steps to solve the problem:

* Build a Min Heap MH of the first K elements (arr[0] to arr[K-1]) of the given array
* For each element, after the Kth element (arr[K] to arr[N-1]), compare it with the root of MH
* If the element is greater than the root then make it root and call [heapify](https://www.geeksforgeeks.org/binary-heap/)for MH
* Else ignore it
* Finally, MH has the K largest elements, and the root of the MH is the Kth largest element

**Note:**All of the above methods can also be used to find the kth smallest elements

Below is the implementation of the above approach:

# Python3 program for the above approach

# importing heapq module

# to implement heap

**import** heapq as hq

**def** FirstKelements(arr, size, k):

    # Creating Min Heap for given

    # array with only k elements

    # Create min heap using heapq module

    minHeap **=** []

**for** i **in** range(k):

        minHeap.append(arr[i])

    hq.heapify(minHeap)

    # Loop For each element in array

    # after the kth element

**for** i **in** range(k, size):

        # If current element is smaller

        # than minimum ((top element of

        # the minHeap) element, do nothing

        # and continue to next element

**if** minHeap[0] > arr[i]:

**continue**

        # Otherwise Change minimum element

        # (top element of the minHeap) to

        # current element by polling out

        # the top element of the minHeap

**else**:

              # deleting top element of the min heap

            minHeap[0] **=** minHeap[**-**1]

            minHeap.pop()

            minHeap.append(arr[i])

            # maintaining heap again using

            # O(n) time operation....

            hq.heapify(minHeap)

    # Now min heap contains k maximum

    # elements, Iterate and print

**for** i **in** minHeap:

        print(i, end**=**" ")

# Driver code

arr **=** [11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45]

size **=** len(arr)

# Size of Min Heap

k **=** 3

FirstKelements(arr, size, k)

'''Code is written by Rajat Kumar.....'''

**K largest(or smallest) elements in an array using**[**Quick Sort partitioning algorithm**](https://www.geeksforgeeks.org/quick-sort/)**:**

To solve the problem follow the below idea:

*We will find the pivot in the array until pivot element index is equal to K, because in the quick sort partioning algorithm all the elements less than pivot are on the left side of the pivot and greater than or equal to that are on the right side. So we can print the array (low to pivot to get K-smallest elements and (N-pivot\_Index) to N for K-largest elements)*

Follow the below steps to solve the problem:

* Choose a pivot number
* if K is lesser than the pivot\_Index then repeat the step
* if K is equal to pivot\_Index: Print the array (low to pivot to get K-smallest elements and (n-pivot\_Index) to n for K-largest elements)
* if  K is greater than pivot\_Index: Repeat the steps for the right part

**Note:**We can improve on the standard quicksort algorithm by using the random() function. Instead of using the pivot element as the last element, we can randomly choose the pivot element randomly.

Below is the implementation of the above approach:

# Python3 program for the above approach

**import** random

**def** kthSmallest(arr, l, r, K, n):

    # If k is smaller than number of

    # elements in array

**if** (K > 0 **and** K <**=** r **-** l **+** 1):

        # Partition the array around last

        # element and get position of pivot

        # element in sorted array

        pos **=** partition(arr, l, r)

        # If position is same as k

**if** (pos **-** l **==** K **-** 1):

**return**

**if** (pos **-** l > K **-** 1):  # If position is more,

                              # recur for left subarray

**return** kthSmallest(arr, l, pos **-** 1, K, n)

        # Else recur for right subarray

**return** kthSmallest(arr, pos **+** 1, r,

                           K **-** pos **+** l **-** 1, n)

    # If k is more than number of

    # elements in array

    print("Invalid value of K")

**def** KthLargest(arr, l, r, K, N):

    # This function arranges k Largest elements in last k positions

    #   It means it arranges N-K-1 smallest elements from starting

    kthSmallest(arr, l, r, N **-** K **-** 1, N)

# 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

**def** partition(arr, l, r):

    x **=** arr[r]

    i **=** l

**for** j **in** range(l, r):

**if** (arr[j] <**=** x):

            arr[i], arr[j] **=** arr[j], arr[i]

            i **+=** 1

    arr[i], arr[r] **=** arr[r], arr[i]

**return** i

# Driver code

a **=** [11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45]

n **=** len(a)

low **=** 0

high **=** n **-** 1

# assume k is 3

k **=** 3

#  Function call

#  For Smallest

kthSmallest(a, 0, n **-** 1, k, n)

#  Print KSmallest no.

**if** (k >**=** 1 **and** k <**=** n):

    print(str(k) **+** " smallest elements are :", end**=**" ")

**for** i **in** range(k):

**print**(a[i], end**=**" ")

**print**()

#  For Largest

KthLargest(a, 0, n**-**1, k, n)

#  Print KLargest no.

**if** (k >**=** 1 **and** k <**=** n):

    print(str(k) **+** " largest elements are :", end**=**" ")

**for** i **in** range(n **-** 1, n**-**k**-**1, **-**1):

**print**(a[i], end**=**" ")

# This code is contributed by shubhamm050402

**Kth smallest element in a row-wise and column-wise sorted 2D array**

Given an n x n matrix, where every row and column is sorted in non-decreasing order. Find the kth smallest element in the given 2D array.

**Example,**

**Input:**k = 3 and array =  
 10, 20, 30, 40  
 15, 25, 35, 45  
 24, 29, 37, 48  
 32, 33, 39, 50   
**Output:** 20  
**Explanation:** The 3rd smallest element is 20

**Input:**k = 7 and array =  
 10, 20, 30, 40  
 15, 25, 35, 45  
 24, 29, 37, 48  
 32, 33, 39, 50   
**Output:** 30

**Explanation:** The 7th smallest element is 30

[Kth element in Matrix](https://practice.geeksforgeeks.org/problems/kth-element-in-matrix/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/kth-element-in-matrix/1/)

**Approach:** So the idea is to find the kth minimum element. Each row and each column is sorted. So it can be thought as [C sorted lists and the lists have to be merged into a single list](https://www.geeksforgeeks.org/merge-k-sorted-linked-lists-set-2-using-min-heap/), the kth element of the list has to be found out. So the approach is similar, the only difference is when the kth element is found the loop ends.

**Algorithm:**

1. The idea is to use min heap. Create a Min-Heap to store the elements
2. Traverse the first row from start to end and build a min heap of elements from first row. A heap entry also stores row number and column number.
3. Now Run a loop k times to extract min element from heap in each iteration
4. Get minimum element (or root) from Min-Heap.
5. Find row number and column number of the minimum element.
6. Replace root with the next element from same column and min-heapify the root.
7. Print the last extracted element, which is the kth minimum element

**Implementation:**

# Program for kth largest element in a 2d array

# sorted row-wise and column-wise

**from** sys **import** maxsize

# A structure to store an entry of heap.

# The entry contains a value from 2D array,

# row and column numbers of the value

**class** HeapNode:

**def** \_\_init\_\_(self, val, r, c):

        self.val **=** val # value to be stored

        self.r **=** r # Row number of value in 2D array

        self.c **=** c # Column number of value in 2D array

# A utility function to minheapify the node harr[i]

# of a heap stored in harr[]

**def** minHeapify(harr, i, heap\_size):

    l **=** i **\*** 2 **+** 1

    r **=** i **\*** 2 **+** 2

**if**(l < heap\_size **and** r<heap\_size **and** harr[l].val < harr[i].val **and** harr[r].val < harr[i].val):

      temp**=** HeapNode(0,0,0)

      temp**=**harr[r]

      harr[r]**=**harr[i]

      harr[i]**=**harr[l]

      harr[l]**=**temp

      minHeapify(harr ,l,heap\_size)

      minHeapify(harr ,r,heap\_size)

**if** (l < heap\_size **and** harr[l].val < harr[i].val):

      temp**=** HeapNode(0,0,0)

      temp**=**harr[i]

      harr[i]**=**harr[l]

      harr[l]**=**temp

      minHeapify(harr ,l,heap\_size)

# This function returns kth smallest element

# in a 2D array mat[][]

**def** kthSmallest(mat, n, k):

    # k must be greater than 0 and smaller than n\*n

**if** k < 0 **or** k > n **\*** n:

**return** maxsize

    # Create a min heap of elements from

    # first row of 2D array

    harr **=** [0] **\*** n

**for** i **in** range(n):

        harr[i] **=** HeapNode(mat[0][i], 0, i)

    hr **=** HeapNode(0, 0, 0)

**for** i **in** range(k):

        # Get current heap root

        hr **=** harr[0]

        # Get next value from column of root's value.

        # If the value stored at root was last value

        # in its column, then assign INFINITE as next value

        nextval **=** mat[hr.r **+** 1][hr.c] **if** (hr.r < n **-** 1) **else** maxsize

        # Update heap root with next value

        harr[0] **=** HeapNode(nextval, hr.r **+** 1, hr.c)

        # Heapify root

        minHeapify(harr, 0, n)

    # Return the value at last extracted root

**return** hr.val

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    mat **=** [[10, 20, 30, 40],

        [15, 25, 35, 45],

        [25, 29, 37, 48],

        [32, 33, 39, 50]]

    print("7th smallest element is",

            kthSmallest(mat, 4, 7))

# This returns count of elements in matrix

# less than of equal to num

**def** getElementsGreaterThanOrEqual(num,n,mat):

    ans **=** 0

**for** i **in** range(n):

        # if num is less than the first element

        # then no more element in matrix

        # further are less than or equal to num

**if** (mat[i][0] > num):

**return** ans

        # if num is greater than last element,

        # it is greater than all elements

        # in that row

**if** (mat[i][n **-** 1] <**=** num):

            ans **+=** n

**continue**

        # This contain the col index of last element

        # in matrix less than of equal

        # to num

        greaterThan **=** 0

        jump **=** n **//** 2

**while**(jump >**=** 1):

**while** (greaterThan **+** jump < n **and** mat[i][greaterThan **+** jump] <**=** num):

                    greaterThan **+=** jump

                jump **//=** 2

        ans **+=** greaterThan **+** 1

**return** ans

# returns kth smallest index in the matrix

**def** kthSmallest(mat, n, k):

    # We know the answer lies between

    # the first and the last element

    # So do a binary search on answer

    # based on the number of elements

    # our current element is greater than

    # the elements in the matrix

    l,r **=** mat[0][0],mat[n **-** 1][n **-** 1]

**while** (l <**=** r):

        mid **=** l **+** (r **-** l) **//** 2

        greaterThanOrEqualMid **=** getElementsGreaterThanOrEqual(mid, n, mat)

**if** (greaterThanOrEqualMid >**=** k):

            r **=** mid **-** 1

**else**:

            l **=** mid **+** 1

**return** l

# driver code

n **=** 4

mat **=** [[10, 20, 30, 40],[15, 25, 35, 45],[25, 29, 37, 48],[32, 33, 39, 50]]

**print**(f"7th smallest element is {kthSmallest(mat, 4, 7)}")

# This code is contributed by shinjanpatra

**Find common elements in three sorted arrays**

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

**Examples:**

***Input****:*

*ar1[] = {1, 5, 10, 20, 40, 80}*

*ar2[] = {6, 7, 20, 80, 100}*

*ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}*

***Output****: 20, 80*

***Input****:*

*ar1[] = {1, 5, 5}*

*ar2[] = {3, 4, 5, 5, 10}*

*ar3[] = {5, 5, 10, 20}*

***Output****: 5, 5*

Recommended Practice

[Common elements](https://practice.geeksforgeeks.org/problems/common-elements1132/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/common-elements1132/1/)

A simple solution is to first find [intersection of two arrays](https://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/)and store the intersection in a temporary array, then find the intersection of third array and temporary array.

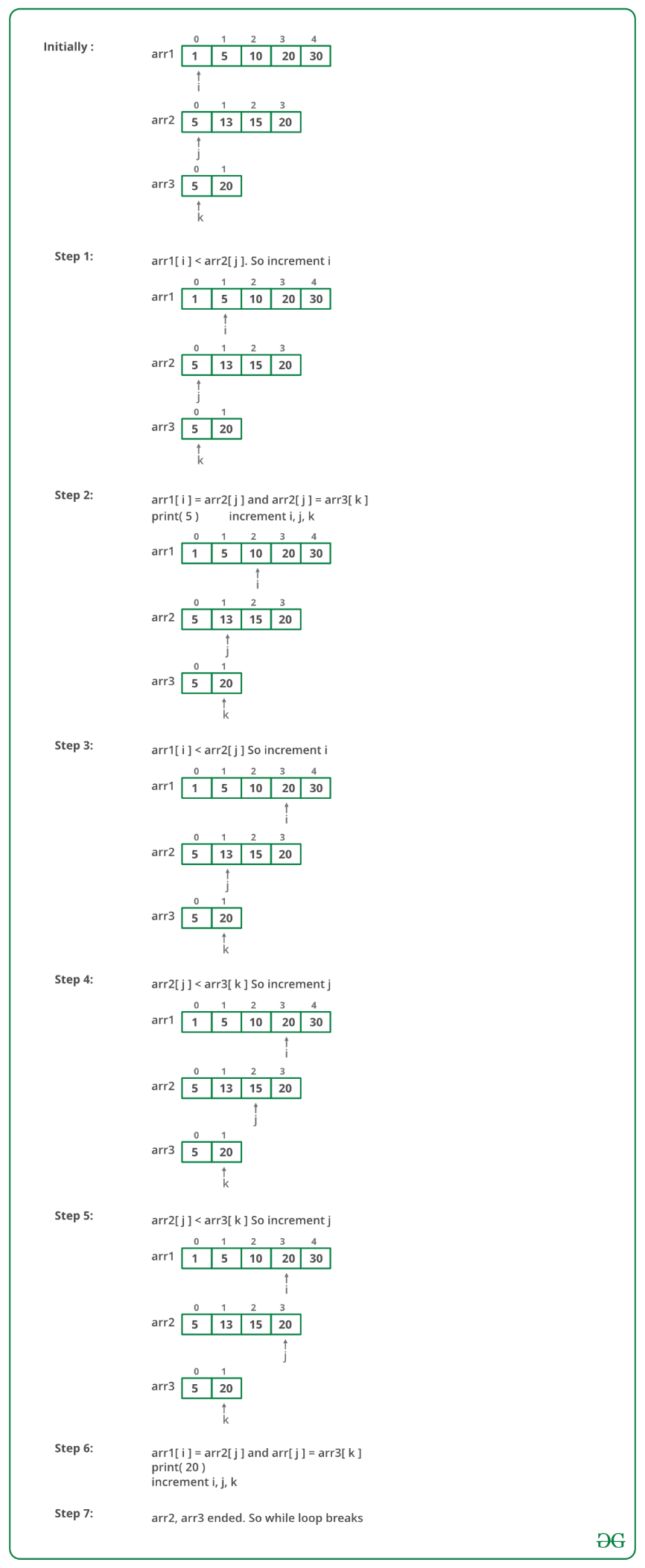
Time complexity of this solution is **O(n1 + n2 + n3)** where n1, n2 and n3 are sizes of ar1[], ar2[] and ar3[] respectively.

The above solution requires extra space and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to [intersection of two arrays](https://www.geeksforgeeks.org/union-and-intersection-of-two-sorted-arrays-2/). Like two arrays loop, we run a loop and traverse three arrays.

Let the current element traversed in ar1[] be x, in ar2[] be y and in ar3[] be z. We can have following cases inside the loop.

1. If x, y and z are same, we can simply print any of them as common element and move ahead in all three arrays.
2. Else If x < y, we can move ahead in ar1[] as x cannot be a common element.
3. Else If x > z and y > z), we can simply move ahead in ar3[] as z cannot be a common element.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

# Python function to print common elements in three sorted arrays

**def** findCommon(ar1, ar2, ar3, n1, n2, n3):

    # Initialize starting indexes for ar1[], ar2[] and ar3[]

    i, j, k **=** 0, 0, 0

    # Iterate through three arrays while all arrays have elements

**while** (i < n1 **and** j < n2 **and** k < n3):

        # If x = y and y = z, print any of them and move ahead

        # in all arrays

**if** (ar1[i] **==** ar2[j] **and** ar2[j] **==** ar3[k]):

            print ar1[i],

            i **+=** 1

            j **+=** 1

            k **+=** 1

        # x < y

**elif** ar1[i] < ar2[j]:

            i **+=** 1

        # y < z

**elif** ar2[j] < ar3[k]:

            j **+=** 1

        # We reach here when x > y and z < y, i.e., z is smallest

**else**:

            k **+=** 1

# Driver program to check above function

ar1 **=** [1, 5, 10, 20, 40, 80]

ar2 **=** [6, 7, 20, 80, 100]

ar3 **=** [3, 4, 15, 20, 30, 70, 80, 120]

n1 **=** len(ar1)

n2 **=** len(ar2)

n3 **=** len(ar3)

print "Common elements are",

findCommon(ar1, ar2, ar3, n1, n2, n3)

# This code is contributed by \_\_Devesh Agrawal\_\_

# Python 3 program for above approach

**import** sys

# This function prints

# common elements in ar1

**def** findCommon(ar1, ar2, ar3, n1,

               n2, n3):

    # Initialize starting indexes

    # for ar1[], ar2and

    # ar3[]

    i **=** 0

    j **=** 0

    k **=** 0

    # Declare three variables prev1,

    # prev2, prev3 to track

    # previous element

    # Initialize prev1, prev2,

    # prev3 with INT\_MIN

    prev1 **=** prev2 **=** prev3 **= -**sys.maxsize **-** 1

    # Iterate through three arrays

    # while all arrays have

    # elements

**while** (i < n1 **and** j < n2 **and** k < n3):

        # If ar1[i] = prev1 and i < n1,

        # keep incrementing i

**while** (ar1[i] **==** prev1 **and** i < n1**-**1):

            i **+=** 1

        # If ar2[j] = prev2 and j < n2,

        # keep incrementing j

**while** (ar2[j] **==** prev2 **and** j < n2):

            j **+=** 1

        # If ar3[k] = prev3 and k < n3,

        # keep incrementing k

**while** (ar3[k] **==** prev3 **and** k < n3):

            k **+=** 1

        # If x = y and y = z, pr

        # any of them, update

        # prev1 prev2, prev3 and move

        # ahead in each array

**if** (ar1[i] **==** ar2[j] **and** ar2[j] **==** ar3[k]):

            print(ar1[i], end**=**" ")

            prev1 **=** ar1[i]

            prev2 **=** ar2[j]

            prev3 **=** ar3[k]

            i **+=** 1

            j **+=** 1

            k **+=** 1

        # If x < y, update prev1

        # and increment i

**elif** (ar1[i] < ar2[j]):

            prev1 **=** ar1[i]

            i **+=** 1

        # If y < z, update prev2

        # and increment j

**elif** (ar2[j] < ar3[k]):

            prev2 **=** ar2[j]

            j **+=** 1

        # We reach here when x > y

        # and z < y, i.e., z is

        # smallest update prev3

        # and increment k

**else**:

            prev3 **=** ar3[k]

            k **+=** 1

# Driver code

ar1 **=** [1, 5, 10, 20, 40, 80, 80]

ar2 **=** [6, 7, 20, 80, 80, 100]

ar3 **=** [3, 4, 15, 20, 30, 70, 80, 80, 120]

n1 **=** len(ar1)

n2 **=** len(ar2)

n3 **=** len(ar3)

print("Common Elements are ")

findCommon(ar1, ar2, ar3, n1, n2, n3)

# This code is contributed by splevel62.

**Ceiling in a sorted array**

Given a sorted array and a value x, the ceiling of x is the smallest element in an array greater than or equal to x, and the floor is the greatest element smaller than or equal to x. Assume that the array is sorted in non-decreasing order. Write efficient functions to find the floor and ceiling of x.

**Examples :**

For example, let the input array be {1, 2, 8, 10, 10, 12, 19}  
For x = 0: floor doesn't exist in array, ceil = 1  
For x = 1: floor = 1, ceil = 1  
For x = 5: floor = 2, ceil = 8  
For x = 20: floor = 19, ceil doesn't exist in array

In the below methods, we have implemented only ceiling search functions. Floor search can be implemented in the same way.

**Method 1 (Linear Search)**

Algorithm to search ceiling of x:

1. If x is smaller than or equal to the first element in the array then return 0(index of the first element).
2. Else linearly search for an index i such that x lies between arr[i] and arr[i+1].
3. If we do not find an index i in step 2, then return -1.

Below is the implementation of the above approach:

# Function to get index of ceiling of x in arr[low..high] \*/

**def** ceilSearch(arr, low, high, x):

    # If x is smaller than or equal to first element,

    # then return the first element \*/

**if** x <**=** arr[low]:

**return** low

    # Otherwise, linearly search for ceil value \*/

    i **=** low

**for** i **in** range(high):

**if** arr[i] **==** x:

**return** i

        # if x lies between arr[i] and arr[i+1] including

        # arr[i+1], then return arr[i+1] \*/

**if** arr[i] < x **and** arr[i**+**1] >**=** x:

**return** i**+**1

    # If we reach here then x is greater than the last element

    # of the array,  return -1 in this case \*/

**return -**1

# Driver program to check above functions \*/

arr **=** [1, 2, 8, 10, 10, 12, 19]

n **=** len(arr)

x **=** 3

index **=** ceilSearch(arr, 0, n**-**1, x);

**if** index **== -**1:

**print** ("Ceiling of %d doesn't exist in array "**%** x)

**else**:

    print ("ceiling of %d is %d"**%**(x, arr[index]))

# This code is contributed by Shreyanshi Arun

# Function to get index of ceiling of x in arr[low..high]\*/

**def** ceilSearch(arr, low, high, x):

    # If x is smaller than or equal to the first element,

    # then return the first element \*/

**if** x <**=** arr[low]:

**return** low

    # If x is greater than the last element, then return -1 \*/

**if** x > arr[high]:

**return -**1

    # get the index of middle element of arr[low..high]\*/

    mid **=** (low **+** high)**/**2;  # low + (high - low)/2 \*/

    # If x is same as middle element, then return mid \*/

**if** arr[mid] **==** x:

**return** mid

    # If x is greater than arr[mid], then either arr[mid + 1]

    # is ceiling of x or ceiling lies in arr[mid+1...high] \*/

**elif** arr[mid] < x:

**if** mid **+** 1 <**=** high **and** x <**=** arr[mid**+**1]:

**return** mid **+** 1

**else**:

**return** ceilSearch(arr, mid**+**1, high, x)

    # If x is smaller than arr[mid], then either arr[mid]

    # is ceiling of x or ceiling lies in arr[low...mid-1] \*/

**else**:

**if** mid **-** 1 >**=** low **and** x > arr[mid**-**1]:

**return** mid

**else**:

**return** ceilSearch(arr, low, mid **-** 1, x)

# Driver program to check above functions

arr **=** [1, 2, 8, 10, 10, 12, 19]

n **=** len(arr)

x **=** 20

index **=** ceilSearch(arr, 0, n**-**1, x);

**if** index **== -**1:

**print** ("Ceiling of %d doesn't exist in array "**%** x)

**else**:

    print ("ceiling of %d is %d"**%**(x, arr[index]))

# This code is contributed by Shreyanshi Arun

**Floor in a Sorted Array**

Given a sorted array and a value **x**, the floor of x is the largest element in the array smaller than or equal to x. Write efficient functions to find the floor of x

**Examples:**

***Input:****arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 5*

***Output:****2*

***Explanation:****2 is the largest element in*

*arr[] smaller than 5*

***Input:****arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 20*

***Output:****19*

***Explanation:****19 is the largest element in*

*arr[] smaller than 20*

***Input :****arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 0*

***Output :****-1*

***Explanation:****Since floor doesn’t exist, output is -1.*

[Floor in a Sorted Array](https://practice.geeksforgeeks.org/problems/floor-in-a-sorted-array-1587115620/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/floor-in-a-sorted-array-1587115620/1/)

**Naive Approach:** To solve the problem follow the below idea:

*The idea is simple, traverse through the array and find the first element greater than x. The element just before the found element is the floor of x*

Follow the given steps to solve the problem:

* Traverse through the array from start to end.
* If the current element is greater than x print the previous number and break out of the loop
* If there is no number greater than x then print the last element
* If the first number is greater than x then print that the floor of x doesn’t exist

Below is the implementation of the above approach:

# Python3 program to find floor of a

# given number in a sorted array

# Function to get index of floor

# of x in arr[low..high]

**def** floorSearch(arr, n, x):

    # If last element is smaller than x

**if** (x >**=** arr[n **-** 1]):

**return** n **-** 1

    # If first element is greater than x

**if** (x < arr[0]):

**return -**1

    # Linearly search for the first element

    # greater than x

**for** i **in** range(1, n):

**if** (arr[i] > x):

**return** (i **-** 1)

**return -**1

# Driver Code

arr **=** [1, 2, 4, 6, 10, 12, 14]

n **=** len(arr)

x **=** 7

index **=** floorSearch(arr, n**-**1, x)

**if** (index **== -**1):

    print("Floor of", x, "doesn't exist \

**in** array ", end="")

**else**:

    print("Floor of", x, "is", arr[index])

# This code is contributed by Smitha Dinesh Semwal.

**Floor in a Sorted Array using binary search:**

To solve the problem follow the below idea:

*There is a catch in the problem, the given array is sorted. The idea is to use*[*Binary Search*](https://www.geeksforgeeks.org/binary-search/)*to find the floor of a number x in a sorted array by comparing it to the middle element and dividing the search space into half*

Follow the given steps to solve the problem:

* The algorithm can be implemented recursively or through iteration, but the basic idea remains the same.
* There are some base cases to handle
* If there is no number greater than x then print the last element
* If the first number is greater than x then print -1
* create three variables *low = 0*, mid and *high = n-1* and another variable to store the answer
* Run a loop or recurse until and unless low is less than or equal to high.
* check if the middle ((low + high) /2) element is less than x, if yes then update the low, i.e low = mid + 1, and update the answer with the middle element. In this step we are reducing the search space to half.
* Else update the high , i.e high = mid – 1
* Print the answer

Below is the implementation of the above approach:

# Python3 program to find floor of a

# given number in a sorted array

# Function to get index of floor

# of x in arr[low..high]

**def** floorSearch(arr, low, high, x):

    # If low and high cross each other

**if** (low > high):

**return -**1

    # If last element is smaller than x

**if** (x >**=** arr[high]):

**return** high

    # Find the middle point

    mid **=** int((low **+** high) **/** 2)

    # If middle point is floor.

**if** (arr[mid] **==** x):

**return** mid

    # If x lies between mid-1 and mid

**if** (mid > 0 **and** arr[mid**-**1] <**=** x

**and** x < arr[mid]):

**return** mid **-** 1

    # If x is smaller than mid,

    # floor must be in left half.

**if** (x < arr[mid]):

**return** floorSearch(arr, low, mid**-**1, x)

    # If mid-1 is not floor and x is greater than

    # arr[mid],

**return** floorSearch(arr, mid **+** 1, high, x)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1, 2, 4, 6, 10, 12, 14]

    n **=** len(arr)

    x **=** 7

    # Function call

    index **=** floorSearch(arr, 0, n**-**1, x)

**if** (index **== -**1):

        print("Floor of", x, "doesn't exist\

**in** array ", end="")

**else**:

**print**("Floor of", x, "is", arr[index])

# This code is contributed by Smitha Dinesh Semwal.

**Find the maximum element in an array which is first increasing and then decreasing**

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.

**Examples :**

Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}  
Output: 500

Input: arr[] = {1, 3, 50, 10, 9, 7, 6}  
Output: 50

Corner case (No decreasing part)  
Input: arr[] = {10, 20, 30, 40, 50}  
Output: 50

Corner case (No increasing part)  
Input: arr[] = {120, 100, 80, 20, 0}  
Output: 120

[Bitonic Point](https://practice.geeksforgeeks.org/problems/maximum-value-in-a-bitonic-array3001/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/maximum-value-in-a-bitonic-array3001/1/)

**Method 1 (Linear Search)**: We can traverse the array and keep track of maximum and element. And finally return the maximum element.

**Implementation:**

# Python3 program to find

# maximum element

**def** findMaximum(arr, low, high):

    max **=** arr[low]

    i **=** low

**for** i **in** range(high**+**1):

**if** arr[i] > max:

            max **=** arr[i]

**return** max

# Driver program to check above functions \*/

arr **=** [1, 30, 40, 50, 60, 70, 23, 20]

n **=** len(arr)

print ("The maximum element is %d"**%**

        findMaximum(arr, 0, n**-**1))

# This code is contributed by Shreyanshi Arun.

**def** findMaximum(arr, low, high):

    # Base Case: Only one element is present in arr[low..high]\*/

**if** low **==** high:

**return** arr[low]

    # If there are two elements and first is greater then

    # the first element is maximum \*/

**if** high **==** low **+** 1 **and** arr[low] >**=** arr[high]:

**return** arr[low];

    # If there are two elements and second is greater then

    # the second element is maximum \*/

**if** high **==** low **+** 1 **and** arr[low] < arr[high]:

**return** arr[high]

    mid **=** (low **+** high)**//**2   #low + (high - low)/2;\*/

    # If we reach a point where arr[mid] is greater than both of

    # its adjacent elements arr[mid-1] and arr[mid+1], then arr[mid]

    # is the maximum element\*/

**if** arr[mid] > arr[mid **+** 1] **and** arr[mid] > arr[mid **-** 1]:

**return** arr[mid]

    # If arr[mid] is greater than the next element and smaller than the previous

    # element then maximum lies on left side of mid \*/

**if** arr[mid] > arr[mid **+** 1] **and** arr[mid] < arr[mid **-** 1]:

**return** findMaximum(arr, low, mid**-**1)

**else**: # when arr[mid] is greater than arr[mid-1] and smaller than arr[mid+1]

**return** findMaximum(arr, mid **+** 1, high)

# Driver program to check above functions \*/

arr **=** [1, 3, 50, 10, 9, 7, 6]

n **=** len(arr)

**print** ("The maximum element is %d"**%** findMaximum(arr, 0, n**-**1))

# This code is contributed by Shreyanshi Arun.

**Given Array of size n and a number k, find all elements that appear more than n/k times**

Given an array of size n and an integer k, find all elements in the array that appear more than n/k times.

**Examples:**

***Input:****arr[] = {3, 1, 2, 2, 1, 2, 3, 3}, k = 4*

***Output:****{2, 3}*

***Explanation:****Here n/k is 8/4 = 2, therefore 2 appears 3 times in the array that is greater than 2 and 3 appears 3 times in the array that is greater than 2*

***Input:****arr[] = {9, 8, 7, 9, 2, 9, 7}, k = 3*

***Output:****{9}*

***Explanation:****Here n/k is 7/3 = 2, therefore 9 appears 3 times in the array that is greater than 2.*

Recommended Problem

[Please solve it on PRACTICE first, before moving on to the solution](https://practice.geeksforgeeks.org/problems/count-element-occurences/1)

[Solve Problem](https://practice.geeksforgeeks.org/problems/count-element-occurences/1)

**Find all elements that appear more than n/k times using**[**Hashing**](https://www.geeksforgeeks.org/introduction-to-hashing-data-structure-and-algorithm-tutorials/)**:**

*The idea is to pick all elements one by one. For every picked element, count its occurrences by traversing the array, if count becomes more than n/k, then print the element.*

Follow the steps below to solve the problem:

* First, make a frequency map of all the elements in the array
* Then traverse the map and check the frequency of every element
* If the frequency is greater than n/k then print the element.

Below is the implementation of the above approach:

# Python3 code to find elements whose

# frequency yis more than n/k

**def** morethanNbyK(arr, n, k):

    x **=** n **//** k

    # unordered\_map initialization

    freq **=** {}

**for** i **in** range(n):

**if** arr[i] **in** freq:

            freq[arr[i]] **+=** 1

**else**:

            freq[arr[i]] **=** 1

    # Traversing the map

**for** i **in** freq:

        # Checking if value of a key-value pair

        # is greater than x (where x=n/k)

**if** (freq[i] > x):

            # Print the key of whose value

            # is greater than x

**print**(i)

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [1, 1, 2, 2, 3, 5, 4, 2, 2, 3, 1, 1, 1]

    n **=** len(arr)

    k **=** 4

    morethanNbyK(arr, n, k)

# This code is contributed by mohit kumar 29

**Find all elements that appear more than n/k times using**[**Moore’s Voting Algorithm**](https://www.geeksforgeeks.org/boyer-moore-majority-voting-algorithm/)**:**

*The idea is to apply****Moore’s Voting algorithm,****as there can be at max****k – 1****elements present in the array which appears more than****n/k****times so their will be****k – 1****candidates. When we encounter an element which is one of our candidates then increment the count else decrement the count.*

**Illustration:**

*Consider k = 4, n = 9*

*Given array: 3 1 2 2 2 1 4 3 3*

***i = 0***

*temp[] has one element {3} with count 1*

*Advertisement*

***i = 1***

*temp[] has two elements {3, 1} with counts 1 and 1 respectively*

***i = 2***

*temp[] has three elements, {3, 1, 2} with counts as 1, 1 and 1 respectively.*

***i = 3***

*temp[] has three elements, {3, 1, 2} with counts as 1, 1 and 2 respectively.*

***i = 4***

*temp[] has three elements, {3, 1, 2} with counts as 1, 1 and 3 respectively.*

***i = 5***

*temp[] has three elements, {3, 1, 2 with counts as 1, 2 and 3 respectively.*

***i = 6***

*temp[] has two elements, {1, 2} with counts as 1 and 2 respectively.*

***i = 7***

*temp[] has three elements, {3, 1, 2} with counts as 1, 1 and 2 respectively.*

***i = 8***

*temp[] has three elements, {3, 1, 2} with counts as 2, 1 and 2 respectively.*

Follow the steps below to solve the problem:

* Create a temporary array of size (**k – 1**) to store elements and their counts (The output elements are going to be among these k-1 elements).
* Traverse through the input array and update temp[] (add/remove an element or increase/decrease count) for every traversed element. The array temp[] stores potential (k-1) candidates at every step.
* Iterate through final (k-1) potential candidates (stored in temp[]). or every element, check if it actually has counted of more than n/k.

 Below is the implementation of the above approach.

# A Python3 program to print elements with

# count more than n/k

# Prints elements with more than n/k

# occurrences in arrof size n. If

# there are no such elements, then

# it prints nothing.

**def** moreThanNdK(arr, n, k):

    # k must be greater than 1

    # to get some output

**if** (k < 2):

**return**

    # Step 1: Create a temporary array

    # (contains element and count) of

    # size k-1. Initialize count of all

    # elements as 0

    temp **=** [[0 **for** i **in** range(2)]

**for** i **in** range(k)]

**for** i **in** range(k **-** 1):

        temp[i][0] **=** 0

    # Step 2: Process all elements

    # of input array

**for** i **in** range(n):

        j **=** 0

        # If arr[i] is already present in

        # the element count array, then

        # increment its count

**while** j < k **-** 1:

**if** (temp[j][1] **==** arr[i]):

                temp[j][0] **+=** 1

**break**

            j **+=** 1

        # If arr[i] is not present in temp

**if** (j **==** k **-** 1):

            l **=** 0

            # If there is position available

            # in temp[], then place arr[i]

            # in the first available position

            # and set count as 1\*/

**while** l < k **-** 1:

**if** (temp[l][0] **==** 0):

                    temp[l][1] **=** arr[i]

                    temp[l][0] **=** 1

**break**

                l **+=** 1

            # If all the position in the

            # tempare filled, then decrease

            # count of every element by 1

**if** (l **==** k **-** 1):

**while** l < k:

                    temp[l][0] **-=** 1

                    l **+=** 1

    # Step 3: Check actual counts

    # of potential candidates in temp[]

**for** i **in** range(k **-** 1):

        # Calculate actual count of elements

        ac **=** 0  # Actual count

**for** j **in** range(n):

**if** (arr[j] **==** temp[i][1]):

                ac **+=** 1

        # If actual count is more

        # than n/k, then print

**if** (ac > n **//** k):

            print("Number:",

                  temp[i][1],

                  " Count:", ac)

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr1 **=** [4, 5, 6, 7, 8, 4, 4]

    size **=** len(arr1)

    k **=** 3

    moreThanNdK(arr1, size, k)

# This code is contributed by mohit kumar 29

**Find all elements that appear more than n/k times using Built-in Python functions:**

This approach is same the first approach but here in python their is a **counter()**that calculates the frequency array.

* Count the frequencies of every element using [**Counter()**](https://www.geeksforgeeks.org/python-counter-objects-elements/) function.
* Traverse the frequency array and print all the elements which occur at more than n/k times.

Below is the implementation of the above approach:

**Python3**

# Python3 implementation

**from** collections **import** Counter

# Function to find the number of array

# elements with frequency more than n/k times

**def** printElements(arr, n, k):

    # Calculating n/k

    x **=** n**//**k

    # Counting frequency of every

    # element using Counter

    mp **=** Counter(arr)

    # Traverse the map and print all

    # the elements with occurrence

    # more than n/k times

**for** it **in** mp:

**if** mp[it] > x:

            print(it)

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [1, 1, 2, 2, 3, 5, 4, 2, 2, 3, 1, 1, 1]

    n **=** len(arr)

    k **=** 4

    printElements(arr, n, k)