1. Check if a given array contains duplicate elements within k distance from each other

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.

Examples:

```
Input: k = 3, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}
Output: false
All duplicates are more than k distance away.

Input: k = 3, arr[] = {1, 2, 3, 1, 4, 5}
Output: true
1 is repeated at distance 3.

Input: k = 3, arr[] = {1, 2, 3, 4, 5}
Output: false

Input: k = 3, arr[] = {1, 2, 3, 4, 4}
Output: true
```

A **Simple Solution** is to run two loops. The outer loop picks every element ‘arr[i]’ as a starting element, the inner loop compares all elements which are within k distance of ‘arr[i]’. The time complexity of this solution is O(kn).

We can solve this problem **in Θ(n) time using Hashing.** The idea is to one by add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

1) Create an empty hashtable.
2) Traverse all elements from left from right. Let the current element be ‘arr[i]’
….a) If current element ‘arr[i]’ is present in hashtable, then return true.
….b) Else add arr[i] to hash and remove arr[i-k] from hash if i is greater than or equal to k

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

/* C++ program to Check if a given array contains duplicate
   elements within k distance from each other */
bool checkDuplicatesWithinK(int arr[], int n, int k)
{
    // Creates an empty hashset
    unordered_set<int> myset;

    // Traverse the input array
    for (int i = 0; i < n; i++)
    {
        // If already present n hash, then we found
        // a duplicate within k distance
        if (myset.find(arr[i]) != myset.end())
            return true;
```

```
        // Add this item to hashset
        myset.insert(arr[i]);

        // Remove the k+1 distant item
        if (i >= k)
            myset.erase(arr[i-k]);
    }
    return false;
}

// Driver method to test above method
int main ()
{
    int arr[] = {10, 5, 3, 4, 3, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    if (checkDuplicatesWithinK(arr, n, 3))
        cout << "Yes";
    else
        cout << "No";
}

//This article is contributed by Chhavi
```

Output:

```
Yes
```

2. Remove duplicate elements in an Array using STL in C++

Given an array, the task is to remove the duplicate elements from the array using STL in C++

Examples:

```
Input: arr[] = {2, 2, 2, 2, 2}
Output: arr[] = {2}

Input: arr[] = {1, 2, 2, 3, 4, 4, 4, 5, 5}
Output: arr[] = {1, 2, 3, 4, 5}
```

Approach:

This can be done using set in standard template library. Set type variable in STL automatically removes duplicating element when we store the element in it.

Below is the implementation of the above approach:

```
// C++ program to remove the
// duplicate elements from the array
// using STL in C++

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

// Function to remove duplicate elements
void removeDuplicates(int arr[], int n)
{

    int i;

    // Initialise a set
    // to store the array values
    set<int> s;

    // Insert the array elements
    // into the set
    for (i = 0; i < n; i++) {

        // insert into set
        s.insert(arr[i]);
    }

    set<int>::iterator it;

    // Print the array with duplicates removed
    cout << "\nAfter removing duplicates:\n";
    for (it = s.begin(); it != s.end(); ++it)
        cout << *it << ", ";
    cout << '\n';
```

```
}

// Driver code
int main()
{
    int arr[] = { 4, 2, 3, 3, 2, 4 };

    int n = sizeof(arr) / sizeof(arr[0]);

    // Print array
    cout << "\nBefore removing duplicates:\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    // call removeDuplicates()
    removeDuplicates(arr, n);

    return 0;
}
```

Output:

```
Before removing duplicates:
4 2 3 3 2 4
After removing duplicates:
2, 3, 4,
```

How to check if two given sets are disjoint?

Given two sets represented by two arrays, how to check if the given two sets are disjoint or not? It may be assumed that the given arrays have no duplicates.

```
Input: set1[] = {12, 34, 11, 9, 3}
       set2[] = {2, 1, 3, 5}
Output: Not Disjoint
3 is common in two sets.

Input: set1[] = {12, 34, 11, 9, 3}
       set2[] = {7, 2, 1, 5}
Output: Yes, Disjoint
There is no common element in two sets.
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

There are plenty of methods to solve this problem, it's a good test to check how many solutions you can guess.

Method 1 (Simple)
Iterate through every element of first set and search it in other set, if any element is found, return false. If no element is found, return tree. Time complexity of this method is O(mn).

Following is implementation of above idea.

```
// A Simple C++ program to check if two sets are disjoint
#include<bits/stdc++.h>
using namespace std;

// Returns true if set1[] and set2[] are disjoint, else false
bool areDisjoint(int set1[], int set2[], int m, int n)
{
    // Take every element of set1[] and search it in set2
    for (int i=0; i<m; i++)
      for (int j=0; j<n; j++)
         if (set1[i] == set2[j])
            return false;

    // If no element of set1 is present in set2
    return true;
}

// Driver program to test above function
int main()
{
    int set1[] = {12, 34, 11, 9, 3};
    int set2[] = {7, 2, 1, 5};
    int m = sizeof(set1)/sizeof(set1[0]);
    int n = sizeof(set2)/sizeof(set2[0]);
```

```
    areDisjoint(set1, set2, m, n)? cout << "Yes" : cout << " No";
    return 0;
}
```

Output :

```
Yes
```

Method 2 (Use Sorting and Merging)

1) Sort first and second sets.

2) Use merge like process to compare elements.

```
// A Simple C++ program to check if two sets are disjoint
#include<bits/stdc++.h>
using namespace std;

// Returns true if set1[] and set2[] are disjoint, else false
bool areDisjoint(int set1[], int set2[], int m, int n)
{
    // Sort the given two sets
    sort(set1, set1+m);
    sort(set2, set2+n);

    // Check for same elements using merge like process
    int i = 0, j = 0;
    while (i < m && j < n)
    {
        if (set1[i] < set2[j])
            i++;
        else if (set2[j] < set1[i])
            j++;
        else /* if set1[i] == set2[j] */
            return false;
    }

    return true;
}

// Driver program to test above function
int main()
{
    int set1[] = {12, 34, 11, 9, 3};
    int set2[] = {7, 2, 1, 5};
    int m = sizeof(set1)/sizeof(set1[0]);
    int n = sizeof(set2)/sizeof(set2[0]);
    areDisjoint(set1, set2, m, n)? cout << "Yes" : cout << " No";
    return 0;
}
```

Output :

```
Yes
```

Time complexity of above solution is O(mLogm + nLogn).

The above solution first sorts both sets, then takes O(m+n) time to find intersection. If we are given that the input sets are sorted, then this method is best among all.

Method 3 (Use Sorting and Binary Search)
This is similar to method 1. Instead of linear search, we use Binary Search.
1) Sort first set.
2) Iterate through every element of second set, and use binary search to search every element in first set. If element is found return it.

Time complexity of this method is O(mLogm + nLogm)

Method 4 (Use Binary Search Tree)
1) Create a self balancing binary search tree (Red Black, AVL, Splay, etc) of all elements in first set.
2) Iterate through all elements of second set and search every element in the above constructed Binary Search Tree. If element is found, return false.
3) If all elements are absent, return true.

Time complexity of this method is O(mLogm + nLogm).

Method 5 (Use Hashing)
1) Create an empty hash table.
2) Iterate through the first set and store every element in hash table.
3) Iterate through second set and check if any element is present in hash table. If present, then return false, else ignore the element.
4) If all elements of second set are not present in hash table, return true.

Following are the implementation of this method.

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

/* C++ program to check if two sets are distinct or not */
// This function prints all distinct elements
bool areDisjoint(int set1[], int set2[], int n1, int n2)
{
    // Creates an empty hashset
    set<int> myset;

    // Traverse the first set and store its elements in hash
    for (int i = 0; i < n1; i++)
        myset.insert(set1[i]);

    // Traverse the second set and check if any element of it
    // is already in hash or not.
    for (int i = 0; i < n2; i++)
        if (myset.find(set2[i]) != myset.end())
            return false;

    return true;
```

```
}

// Driver method to test above method
int main()
{
    int set1[] = {10, 5, 3, 4, 6};
    int set2[] = {8, 7, 9, 3};

    int n1 = sizeof(set1) / sizeof(set1[0]);
    int n2 = sizeof(set2) / sizeof(set2[0]);
    if (areDisjoint(set1, set2, n1, n2))
        cout << "Yes";
    else
        cout << "No";
}
//This article is contributed by Chhavi
```

Output:

```
No
```

Time complexity of the above implementation is O(m+n) under the assumption that hash set operations like add() and contains() work in O(1) time.

4. Group multiple occurrence of array elements ordered by first occurrence

Given an unsorted array with repetitions, the task is to group multiple occurrence of individual elements. The grouping should happen in a way that the order of first occurrences of all elements is maintained.

Examples:

```
Input: arr[] = {5, 3, 5, 1, 3, 3}
Output:        {5, 5, 3, 3, 3, 1}

Input: arr[] = {4, 6, 9, 2, 3, 4, 9, 6, 10, 4}
Output:        {4, 4, 4, 6, 6, 9, 9, 2, 3, 10}
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

Simple Solution is to use nested loops. The outer loop traverses array elements one by one. The inner loop checks if this is first occurrence, if yes, then the inner loop prints it and all other occurrences.

```
// A simple C++ program to group multiple occurrences of
individual
// array elements
#include<bits/stdc++.h>
using namespace std;

// A simple method to group all occurrences of individual elements
void groupElements(int arr[], int n)
{
    // Initialize all elements as not visited
    bool *visited = new bool[n];
    for (int i=0; i<n; i++)
        visited[i] = false;

    // Traverse all elements
    for (int i=0; i<n; i++)
    {
        // Check if this is first occurrence
        if (!visited[i])
        {
            // If yes, print it and all subsequent occurrences
            cout << arr[i] << " ";
            for (int j=i+1; j<n; j++)
            {
                if (arr[i] == arr[j])
```

```
                {
                    cout << arr[i] << " ";
                    visited[j] = true;
                }
            }
        }
    }

    delete[] visited;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {4, 6, 9, 2, 3, 4, 9, 6, 10, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    groupElements(arr, n);
    return 0;
}
```

Output:

```
4 4 4 6 6 9 9 2 3 10
```

Time complexity of the above method is $O(n^2)$.

Binary Search Tree based Method: The time complexity can be improved to O(nLogn) using self-balancing binary search tree like Red-Black Tree or AVL tree. Following is complete algorithm.
1) Create an empty Binary Search Tree (BST). Every BST node is going to contain an array element and its count.
2) Traverse the input array and do following for every element.
……..a) If element is not present in BST, then insert it with count as 0.
……..b) If element is present, then increment count in corresponding BST node.
3) Traverse the array again and do following for every element.
…….. If element is present in BST, then do following
…….a) Get its count and print the element ‘count’ times.
…….b) Delete the element from BST.

Time Complexity of the above solution is O(nLogn).

Hashing based Method: We can also use hashing. The idea is to replace Binary Search Tree with a Hash Map in above algorithm.

Below is Implementation of hashing based solution.

```
/* Java program to group multiple occurrences of individual array
elements */
import java.util.HashMap;
```

```
class Main
{
    // A hashing based method to group all occurrences of
individual elements
    static void orderedGroup(int arr[])
    {
        // Creates an empty hashmap
        HashMap<Integer, Integer> hM = new HashMap<Integer,
Integer>();

        // Traverse the array elements, and store count for every
element
        // in HashMap
        for (int i=0; i<arr.length; i++)
        {
           // Check if element is already in HashMap
           Integer prevCount = hM.get(arr[i]);
           if (prevCount == null)
                prevCount = 0;

           // Increment count of element element in HashMap
           hM.put(arr[i], prevCount + 1);
        }

        // Traverse array again
        for (int i=0; i<arr.length; i++)
        {
            // Check if this is first occurrence
            Integer count =  hM.get(arr[i]);
            if (count != null)
            {
                // If yes, then print the element 'count' times
                for (int j=0; j<count; j++)
                   System.out.print(arr[i] + " ");

                // And remove the element from HashMap.
                hM.remove(arr[i]);
            }
        }
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 5, 3, 10, 10, 4, 1, 3};
        orderedGroup(arr);
    }
}
```

Output:

```
10 10 10 5 3 3 4 1
```

Time Complexity of the above hashing based solution is Θ(n) under the assumption that insert, search and delete operations on HashMap take O(1) time.

5. Group all occurrences of characters according to first appearance

Given a string of lowercase characters, the task is to print the string in a manner such that a character comes first in string displays first with all its occurrences in string.

Examples:

```
Input : str = "geeksforgeeks"
Output:  ggeeeekkssfor
Explanation: In the given string 'g' comes first
and occurs 2 times so it is printed first
Then 'e' comes in this string and 4 times so
it gets printed. Similarly remaining string is
printed.


Input :  str = "occurrence"
output : occcurreen


Input  : str = "cdab"
Output : cdab
```

This problem is a string version of following problem for array of integers.

Group multiple occurrence of array elements ordered by first occurrences

Since given strings have only 26 possible characters, it is easier to implement for strings.

Implementation:

1- Count the occurrence of all the characters in given string using an array of size 26.

2- Then start traversing the string. Print every character its count times.

```
// C++ program to print all occurrences of every character
```

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

// Since only lower case characters are there
const int MAX_CHAR = 26;

// Function to print the string
void printGrouped(string str)
{
    int n = str.length();

    // Initialize counts of all characters as 0
    int  count[MAX_CHAR] = {0};

    // Count occurrences of all characters in string
    for (int i = 0 ; i < n ; i++)
        count[str[i]-'a']++;

    // Starts traversing the string
    for (int i = 0; i < n ; i++)
    {
        // Print the character till its count in
        // hash array
        while (count[str[i]-'a']--)
            cout << str[i];

        // Make this character's count value as 0.
        count[str[i]-'a'] = 0;
    }
}

// Driver code
int main()
{
    string str = "geeksforgeeks";

    printGrouped(str);

    return 0;
}
```

Output:

```
ggeeeekkssfor
```

6. Count distinct elements in every window of size k

Given an array of size n and an integer k, return the of count of distinct numbers in all windows of size k.

Example:

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

k = 4

Output:

3

4

4

3

Explanation:

First window is {1, 2, 1, 3}, count of distinct numbers is 3

Second window is {2, 1, 3, 4} count of distinct numbers is 4

Third window is {1, 3, 4, 2} count of distinct numbers is 4

Fourth window is {3, 4, 2, 3} count of distinct numbers is 3

Source: Microsoft Interview Question

A Simple Solution is to traverse the given array, consider every window in it and count distinct elements in the window.

Below is the implementation of the simple solution.

```
// Simple C++ program to count distinct
// elements in every window of size k
#include <bits/stdc++.h>
using namespace std;

// Counts distinct elements in window of size k
int countWindowDistinct(int win[], int k)
{
    int dist_count = 0;

    // Traverse the window
    for (int i=0; i<k; i++)
    {
        // Check if element arr[i] exists in arr[0..i-1]
        int j;
```

```
        for (j=0; j<i; j++)
           if (win[i] == win[j])
              break;
        if (j==i)
            dist_count++;
    }
    return dist_count;
}

// Counts distinct elements in all windows of size k
void countDistinct(int arr[], int n, int k)
{
    // Traverse through every window
    for (int i=0; i<=n-k; i++)
       cout << countWindowDistinct(arr+i, k) << endl;
}

// Driver program
int main()
{
    int arr[] = {1, 2, 1, 3, 4, 2, 3},  k = 4;
    int n = sizeof(arr)/sizeof(arr[0]);
    countDistinct(arr, n, k);
    return 0;
}
```

Output:

```
3

4

4

3
```

Time complexity of the above solution is O(nk2). We can improve time complexity to O(nkLok) by modifying countWindowDistinct() to use sorting. The function can further be optimized to use hashing to find distinct elements in a window. With hashing the time complexity becomes O(nk). Below is a different approach that works in O(n) time.

An Efficient Solution is to use the count of the previous window while sliding the window. The idea is to create a hash map that stores elements of the current window. When we slide the window, we remove an element from the hash and

add an element. We also keep track of distinct elements. Below is the algorithm.

- Create an empty hash map. Let hash map be hM
- Initialize distinct element count ‘dist_count’ as 0.
- Traverse through the first window and insert elements of the first window to hM. The elements are used as key and their counts as the value in hM. Also, keep updating ‘dist_count’
- Print ‘dist_count’ for the first window.
- Traverse through the remaining array (or other windows).
- Remove the first element of the previous window.
    - If the removed element appeared only once, remove it from hM and do “dist_count–“
    - else (appeared multiple times in hM), then decrement its count in hM
- Add the current element (last element of the new window)
    - If the added element is not present in hM, add it to hM and do “dist_count++”
    - Else (the added element appeared multiple times), increment its count in hM

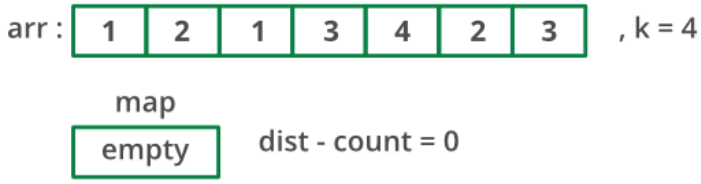
Below is a dry run of the above approach:

Below is the implementation of the above approach:

```
// An efficient C++ program to
// count distinct elements in
// every window of size k
#include <iostream>
#include <map>
using namespace std;

void countDistinct(int arr[], int k, int n)
{
    // Creates an empty hashmap hm
    map<int, int> hm;

    // initialize distinct element count for current window
    int dist_count = 0;

    // Traverse the first window and store count
```

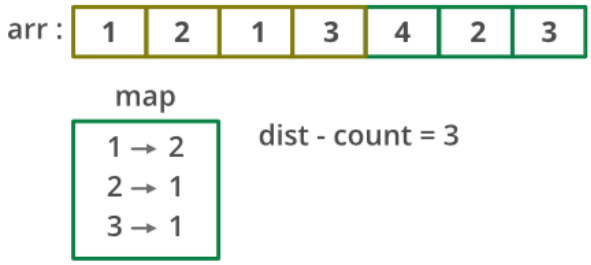
```
    // of every element in hash map
    for (int i = 0; i < k; i++)
    {
       if (hm[arr[i]] == 0)
       {
           dist_count++;
       }
    hm[arr[i]] += 1;
    }

   // Print count of first window
  cout << dist_count << endl;

   // Traverse through the remaining array
   for (int i = k; i < n; i++)
   {
     // Remove first element of previous window
     // If there was only one occurrence, then reduce distinct
count.
     if (hm[arr[i-k]] == 1)
     {
        dist_count--;
     }
   // reduce count of the removed element
     hm[arr[i-k]] -= 1;

   // Add new element of current window
   // If this element appears first time,
   // increment distinct element count

    if (hm[arr[i]] == 0)
    {
       dist_count++;
    }
    hm[arr[i]] += 1;

  // Print count of current window
    cout << dist_count << endl;
  }
}

int main()
{
   int arr[] = {1, 2, 1, 3, 4, 2, 3};
   int size = sizeof(arr)/sizeof(arr[0]);
   int k = 4;
   countDistinct(arr, k, size);

   return 0;
```

Initially :



Step 1:

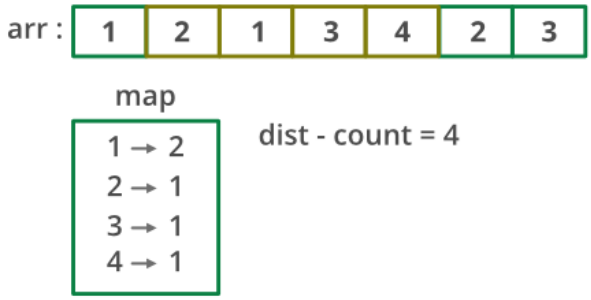
find number of distinct elements in first window and place them in map



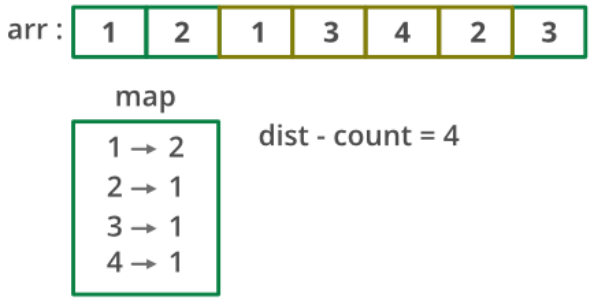
In all remaining steps :

Remove first element of previous window and add new element of current window

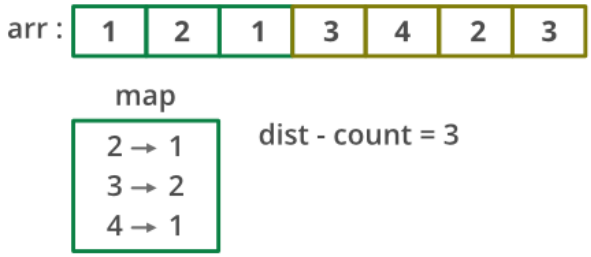
Step 2:



Step 3:



Step 3:




```
}
//This solution is contributed by Aditya Goel
```

Output:

```
3

4

4

3
```

Time complexity of the above solution is O(n).

7. Find missing elements of a range

Given an array arr[0..n-1] of distinct elements and a range [low, high], find all numbers that are in range, but not in array. The missing elements should be printed in sorted order.

Examples:

```
Input: arr[] = {10, 12, 11, 15},

      low = 10, hight = 15

Output: 13, 14



Input: arr[] = {1, 14, 11, 51, 15},

      low = 50, hight = 55

Output: 50, 52, 53, 54
```

There can be following two approaches to solve the problem.

1. Use Sorting : Sort the array, then do binary search for ‘low’. Once location of low is find, start traversing array from that location and keep printing all missing numbers.

```
// A sorting based C++ program to find missing
// elements from an array
#include <bits/stdc++.h>
using namespace std;

// Print all elements of range [low, high] that
// are not present in arr[0..n-1]
void printMissing(int arr[], int n, int low,
                  int high)
{
    // Sort the array
    sort(arr, arr + n);

    // Do binary search for 'low' in sorted
    // array and find index of first element
    // which either equal to or greater than
    // low.
    int* ptr = lower_bound(arr, arr + n, low);
```

```
    int index = ptr - arr;

    // Start from the found index and linearly
    // search every range element x after this
    // index in arr[]
    int i = index, x = low;
    while (i < n && x <= high) {
        // If x doesn't math with current element
        // print it
        if (arr[i] != x)
            cout << x << " ";

        // If x matches, move to next element in arr[]
        else
            i++;

        // Move to next element in range [low, high]
        x++;
    }

    // Print range elements thar are greater than the
    // last element of sorted array.
    while (x <= high)
        cout << x++ << " ";
}

// Driver program
int main()
{
    int arr[] = { 1, 3, 5, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int low = 1, high = 10;
    printMissing(arr, n, low, high);
    return 0;
}
```

Output:

```
2 6 7 8 9 10
```

2. Using Arrays : Create a boolean array, where each index will represent wether the (i+low)th element is present in array or not. Mark all those elements which are in the given range and are present in the array. Once all array items, present in given range have been marked true in the array we traverse through the boolean array and print all elements whose value is false,

```cpp
// An array based C++ program
// to find missing elements from
// an array
#include <bits/stdc++.h>
using namespace std;

// Print all elements of range
// [low, high] that are not present
// in arr[0..n-1]
void printMissing(
    int arr[], int n,
    int low, int high)
{
    // Create boolean array of size
    // high-low+1, each index i representing
    // wether (i+low)th element found or not.
    bool points_of_range[high - low + 1] = { false };

    for (int i = 0; i < n; i++) {
        // if ith element of arr is in
        // range low to high then mark
        // corresponding index as true in array
        if (low <= arr[i] && arr[i] <= high)
            points_of_range[arr[i] - low] = true;
    }

    // Traverse through the range and
    // print all elements  whose value
    // is false
    for (int x = 0; x <= high - low; x++) {
        if (points_of_range[x] == false)
            cout << low + x << " ";
    }
}

// Driver program
int main()
{
    int arr[] = { 1, 3, 5, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int low = 1, high = 10;
    printMissing(arr, n, low, high);
    return 0;
}

// This code is contributed by Shubh Bansal
```

3. Use Hashing : Create a hash table and insert all array items into the hash table. Once all items are in hash table, traverse through the range and print all missing elements.

```cpp
// A hashing based C++ program to find missing
// elements from an array
#include <bits/stdc++.h>
using namespace std;

// Print all elements of range [low, high] that
// are not present in arr[0..n-1]
void printMissing(int arr[], int n, int low,
                  int high)
{
    // Insert all elements of arr[] in set
    unordered_set<int> s;
    for (int i = 0; i < n; i++)
        s.insert(arr[i]);

    // Traverse throught the range an print all
    // missing elements
    for (int x = low; x <= high; x++)
        if (s.find(x) == s.end())
            cout << x << " ";
}

// Driver program
int main()
{
    int arr[] = { 1, 3, 5, 4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int low = 1, high = 10;
    printMissing(arr, n, low, high);
    return 0;
}
```

```java
// A hashing based Java program to find missing
// elements from an array

import java.util.Arrays;
import java.util.HashSet;

public class Print {
    // Print all elements of range [low, high] that
    // are not present in arr[0..n-1]
    static void printMissing(int ar[], int low, int high)
    {
        HashSet<Integer> hs = new HashSet<>();

        // Insert all elements of arr[] in set
        for (int i = 0; i < ar.length; i++)
            hs.add(ar[i]);
```

```
        // Traverse throught the range an print all
        // missing elements
        for (int i = low; i <= high; i++) {
            if (!hs.contains(i)) {
                System.out.print(i + " ");
            }
        }
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = { 1, 3, 5, 4 };
        int low = 1, high = 10;
        printMissing(arr, low, high);
    }
}

// This code is contributed by Rishabh Mahrsee
```

Python3

```
# A hashing based Python 3 program to
# find missing elements from an array

# Print all elements of range
# [low, high] that are not
# present in arr[0..n-1]
def printMissing(arr, n, low, high):

    # Insert all elements of
    # arr[] in set
    s = set(arr)

    # Traverse through the range
    # and print all missing elements
    for x in range(low, high + 1):
        if x not in s:
            print(x, end = ' ')

# Driver Code
arr = [1, 3, 5, 4]
n = len(arr)
low, high = 1, 10
printMissing(arr, n, low, high)

# This code is contributed
# by SamyuktaSHegde
```

```
// A hashing based C# program to
// find missing elements from an array
using System;
using System.Collections.Generic;
```

```
class GFG {

    // Print all elements of range
    // [low, high] that are not
    // present in arr[0..n-1]
    static void printMissing(int[] arr, int n,
                             int low, int high)
    {
        // Insert all elements of arr[] in set
        HashSet<int> s = new HashSet<int>();
        for (int i = 0; i < n; i++) {
            s.Add(arr[i]);
        }

        // Traverse throught the range
        // an print all missing elements
        for (int x = low; x <= high; x++)
            if (!s.Contains(x))
                Console.Write(x + " ");
    }

    // Driver Code
    public static void Main()
    {
        int[] arr = { 1, 3, 5, 4 };
        int n = arr.Length;
        int low = 1, high = 10;
        printMissing(arr, n, low, high);
    }
}

// This code is contributed by ihritik
```

Output:

```
2 6 7 8 9 10
```

Which approach is better?
Time complexity of first approach is O(nLogn + k) where k is number of missing elements (Note that k may be more than nLogn if array is small and range is big)
Time complexity of second and third solution is O(n + (high-low+1)).
If the given array has almost elements of range i.e., n is close to value of (high-low+1), then second and third approaches are definitely better as there is no Log n factor. But if n is much smaller than range, then first approach is better as it doesn't require extra space for hashing. We can also modify first approach to print adjacent missing elements as range to save time. For example if 50, 51, 52, 53, 54, 59 are missing, we can print them as 50-54, 59 in first method. And if printing this way is allowed, the first approach takes only O(n Log n) time.
Out of Second and Third Solution the second solution is better because worst case time complexity of second solution is better than third

8. Print all subarrays with 0 sum

Given an array, print all subarrays in the array which has sum 0.

Examples:

```
Input:  arr = [6, 3, -1, -3, 4, -2, 2,
           4, 6, -12, -7]

Output:
Subarray found from Index 2 to 4

Subarray found from Index 2 to 6

Subarray found from Index 5 to 6

Subarray found from Index 6 to 9

Subarray found from Index 0 to 10
```

Related posts: Find if there is a subarray with 0 sum

A simple solution is to consider all subarrays one by one and check if sum of every subarray is equal to 0 or not. The complexity of this solution would be O(n^2).

A better approach is to use Hashing.

Do following for each element in the array

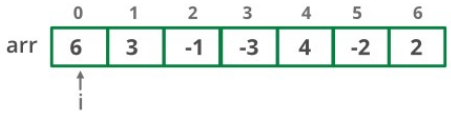
1. Maintain sum of elements encountered so far in a variable (say sum).
2. If current sum is 0, we found a subarray starting from index 0 and ending at index current index
3. Check if current sum exists in the hash table or not.
4. If current sum already exists in the hash table then it indicates that this sum was the sum of some sub-array elements arr[0]...arr[i] and now the same sum is obtained for the current sub-array arr[0]...arr[j] which means that the sum of the sub-array arr[i+1]...arr[j] must be 0.
5. Insert current sum into the hash table

Below is a dry run of the above approach:

Initially :

Sum = 0, map(empty), vector(empty)

Step 1:

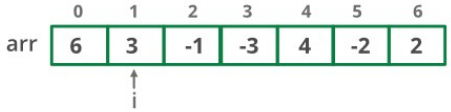


Sum = 6

map

6 → 0

Step 2:

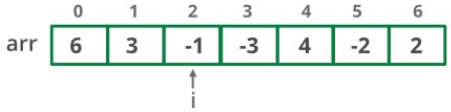


Sum = 9

map

6 → 0
9 → 1

Step 3:

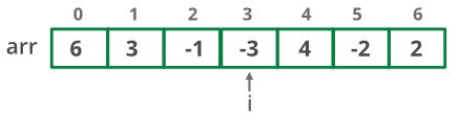


Sum = 8

map

6 → 0
9 → 1
8 → 2

Step 4:

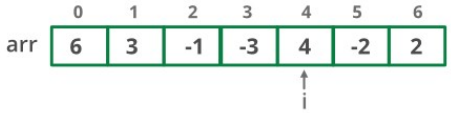


Sum = 5

map

6 → 0
9 → 1
8 → 2
5 → 3

Step 5:

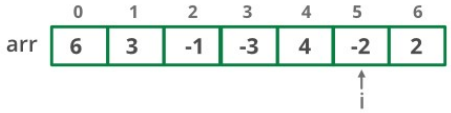


Sum = 9
Vector { (2,4) }

map

6 → 0
9 → 1, 4
8 → 2
5 → 3

Step 6:

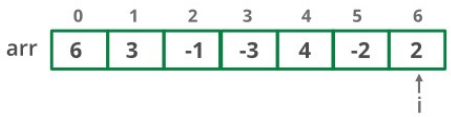


Sum = 7
Vector { (2,4) }

map

6 → 0
9 → 1, 4
8 → 2
5 → 3
7 → 5

Step 7:



Sum = 9
Vector { (2,4) (2,6) (5,6) }

map

6 → 0
9 → 1, 4, 6
8 → 2
5 → 3
7 → 5

Below is the implementation of the above approach

```cpp
// C++ program to print all subarrays
// in the array which has sum 0
#include <bits/stdc++.h>
using namespace std;

// Function to print all subarrays in the array which
// has sum 0
vector< pair<int, int> > findSubArrays(int arr[], int n)
{
    // create an empty map
    unordered_map<int, vector<int> > map;

    // create an empty vector of pairs to store
    // subarray starting and ending index
    vector <pair<int, int>> out;

    // Maintains sum of elements so far
    int sum = 0;

    for (int i = 0; i < n; i++)
    {
        // add current element to sum
        sum += arr[i];

        // if sum is 0, we found a subarray starting
        // from index 0 and ending at index i
        if (sum == 0)
            out.push_back(make_pair(0, i));

        // If sum already exists in the map there exists
        // at-least one subarray ending at index i with
        // 0 sum
        if (map.find(sum) != map.end())
        {
            // map[sum] stores starting index of all subarrays
            vector<int> vc = map[sum];
            for (auto it = vc.begin(); it != vc.end(); it++)
                out.push_back(make_pair(*it + 1, i));
        }

        // Important - no else
        map[sum].push_back(i);
    }

    // return output vector
    return out;
}
```

```
// Utility function to print all subarrays with sum 0
void print(vector<pair<int, int>> out)
{
    for (auto it = out.begin(); it != out.end(); it++)
        cout << "Subarray found from Index " <<
            it->first << " to " << it->second << endl;
}

// Driver code
int main()
{
    int arr[] = {6, 3, -1, -3, 4, -2, 2, 4, 6, -12, -7};
    int n = sizeof(arr)/sizeof(arr[0]);

    vector<pair<int, int> > out = findSubArrays(arr, n);

    // if we didn't find any subarray with 0 sum,
    // then subarray doesn't exists
    if (out.size() == 0)
        cout << "No subarray exists";
    else
        print(out);

    return 0;
}
```

Output:

```
Subarray found from Index 2 to 4

Subarray found from Index 2 to 6

Subarray found from Index 5 to 6

Subarray found from Index 6 to 9

Subarray found from Index 0 to 10
```

9. Find four elements a, b, c and d in an array such that a+b = c+d

Given an array of distinct integers, find if there are two pairs (a, b) and (c, d) such that a+b = c+d, and a, b, c and d are distinct elements. If there are multiple answers, then print any of them.

Example:

```
Input:   {3, 4, 7, 1, 2, 9, 8}

Output:  (3, 8) and (4, 7)

Explanation: 3+8 = 4+7


Input:   {3, 4, 7, 1, 12, 9};

Output:  (4, 12) and (7, 9)

Explanation: 4+12 = 7+9


Input:  {65, 30, 7, 90, 1, 9, 8};

Output:  No pairs found
```

Expected Time Complexity: $O(n^2)$

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

A Simple Solution is to run four loops to generate all possible quadruples of array element. For every quadruple (a, b, c, d), check if (a+b) = (c+d). Time complexity of this solution is O(n4).

An Efficient Solution can solve this problem in $O(n^2)$ time. The idea is to use hashing. We use sum as key and pair as value in hash table.

```
Loop i = 0 to n-1 :
```

```
   Loop j = i + 1 to n-1 :

       calculate sum

       If in hash table any index already exist

          Then print (i, j) and previous pair

          from hash table

       Else update hash table

   EndLoop;

EndLoop;
```

Below are implementations of above idea. In below implementation, map is used instead of hash. Time complexity of map insert and search is actually O(Log n) instead of O(1). So below implementation is $O(n^2 \text{ Log } n)$.

```
// Find four different elements a,b,c and d of array such that
// a+b = c+d
#include<bits/stdc++.h>
using namespace std;

bool findPairs(int arr[], int n)
{
    // Create an empty Hash to store mapping from sum to
    // pair indexes
    map<int, pair<int, int> > Hash;

    // Traverse through all possible pairs of arr[]
    for (int i = 0; i < n; ++i)
    {
        for (int j = i + 1; j < n; ++j)
        {
            // If sum of current pair is not in hash,
            // then store it and continue to next pair
            int sum = arr[i] + arr[j];
            if (Hash.find(sum) == Hash.end())
                Hash[sum] = make_pair(i, j);

            else // Else (Sum already present in hash)
            {
                // Find previous pair
                pair<int, int> pp = Hash[sum];// pp->previous pair

                // Since array elements are distinct, we don't
                // need to check if any element is common among
```

```
pairs
                cout << "(" << arr[pp.first] << ", " <<
arr[pp.second]
                     << ") and (" << arr[i] << ", " << arr[j] <<
")n";
                return true;
            }
        }
    }

    cout << "No pairs found";
    return false;
}

// Driver program
int main()
{
    int arr[] = {3, 4, 7, 1, 2, 9, 8};
    int n   =  sizeof arr / sizeof arr[0];
    findPairs(arr, n);
    return 0;
}
```

Output:

```
(3, 8) and (4, 7)
```

Thanks to Gaurav Ahirwar for suggesting above solutions.

Exercise:

1) Extend the above solution with duplicates allowed in array.

2) Further extend the solution to print all quadruples in output instead of just one. And all quadruples should be printed printed in lexicographical order (smaller values before greater ones). Assume we have two solutions S1 and S2.

```
S1 : a1 b1 c1 d1 ( these are values of indices int the array )

S2 : a2 b2 c2 d2



S1 is lexicographically smaller than S2 iff

  a1 < a2 OR

  a1 = a2 AND b1 < b2 OR

  a1 = a2 AND b1 = b2 AND c1 < c2 OR

  a1 = a2 AND b1 = b2 AND c1 = c2 AND d1 < d2
```