**What is an Array?**

*An****array****is a collection of items of the same variable type stored that are stored at contiguous memory locations. It’s one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0.*

The dream of every programmer is to become not just a good, but also a great programmer. We all want to achieve our goals and to achieve our goals, we must have a great plan with us. In this context, we have decided to provide a complete guide for Arrays interview preparation, which will help you to tackle the problems that are mostly asked in the interview, such as What is an Array, What is Array in C language, How do you initialize an Array in C, How to sort an Array, etc. We have also covered the topics such as Top **Theoretical interview questions**and**Top interview coding questions** in this complete guide for Array interview preparation.

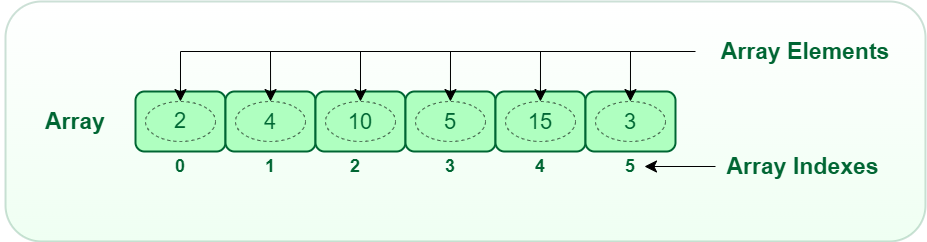
We can directly access an array element by using its index value.

**Basic terminologies of array**

* **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
* **Array element:**Elements are items stored in an array and can be accessed by their index.
* **Array Length:** The length of an array is determined by the number of elements it can contain.

**Representation of Array**

The representation of an array can be defined by its declaration. A declaration means allocating memory for an array of a given size.

*Array*

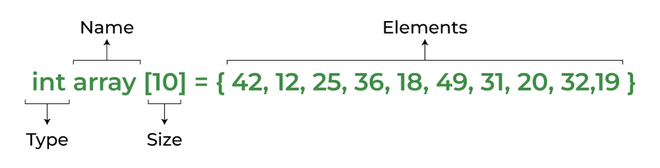
Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations.

C++Java

int arr[5]; // This array will store integer type element

char arr[10]; // This array will store char type element

float arr[20]; // This array will store float type element



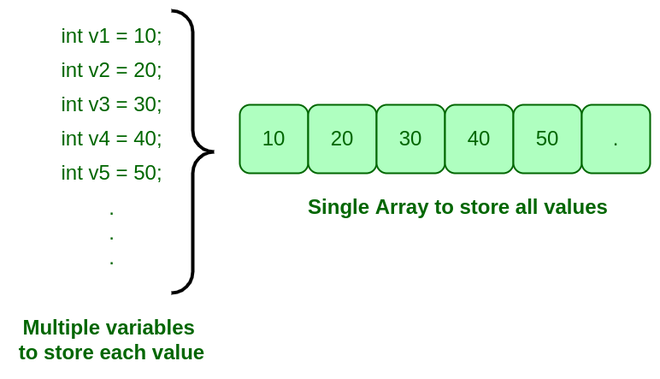
*Array  declaration*

However, the above declaration is **static** or **compile-time** memory allocation, which means that the array element’s memory is allocated when a program is compiled. Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage, but don’t you think it will not be the same situation as we know the size of the array every time, there might be a case where we don’t know the size of the array. If we declare a larger size and store a lesser number of elements will result in a wastage of memory or either be a case where we declare a lesser size then we won’t get enough memory to store the rest of the elements. In such cases, static memory allocation is not preferred.

**Why Array Data Structures is needed?**

Assume there is a class of five students and if we have to keep records of their marks in examination then, we can do this by declaring five variables individual and keeping track of records but what if the number of students becomes very large, it would be challenging to manipulate and maintain the data.

What it means is that, we can use normal variables (v1, v2, v3, ..) when we have a small number of objects. But if we want to store a large number of instances, it becomes difficult to manage them with normal variables. **The idea of an array is to represent many instances in one variable**..



*Need for Array*

Implementing Arrays in C++ using STL

We already have discussed the basic declaration of arrays. Arrays can also be implemented using some built-in classes available in the C++ Standard Template Library.   
  
Some of the most commonly used classes for implementing sequential lists or arrays are:

* Vector
* List

Let's look at each of these classes in details.

### Vector

Vector in C++ STL is a class that represents a dynamic array. The advantages of vector over normal arrays are,

* We do not need to pass size as an extra parameter when we pass vector.
* Vectors have many in-built functions for erasing an element, inserting an element etc.
* Vectors support dynamic sizes, we do not have to initially specify the size of a vector. We can also resize a vector.
* There are many other functionalities vector provide.

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.  
  
To use the Vector class, include the below header file in your program:

**#include< vector >**

**Declaring Vector**: 

vector< *Type\_of\_element* > *vector\_name*;

Here, *Type\_of\_element* can be any valid C++ data type,

or can be any other container also like Pair, List etc.

Some important and commonly used functions of Vector class are:

* **begin()** – Returns an iterator pointing to the first element in the vector.
* **end()** – Returns an iterator pointing to the theoretical element that follows the last element in the vector.
* **size()** – Returns the number of elements in the vector.
* **capacity()** – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
* **empty()** – Returns whether the container is empty.
* **push\_back()** – It push the elements into a vector from the back.
* **pop\_back()**– It is used to pop or remove elements from a vector from the back.
* **insert()** – It inserts new elements before the element at the specified position.
* **erase()** – It is used to remove elements from a container from the specified position or range.
* **swap()** – It is used to swap the contents of one vector with another vector of same type and size.
* **clear()** – It is used to remove all the elements of the vector container.
* **emplace()** – It extends the container by inserting new element at position.
* **emplace\_back()** – It is used to insert a new element into the vector container, the new element is added to the end of the vector.

Below program illustrate the above methods:

CPP

// C++ program to illustrate the above functions

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> v;

// Push elements

for (int i = 1; i <= 5; i++)

v.push\_back(i);

cout << "Size : " << v.size();

// checks if the vector is empty or not

if (v.empty() == false)

cout << "\nVector is not empty";

else

cout << "\nVector is empty";

cout << "\nOutput of begin and end: ";

for (auto i = v.begin(); i != v.end(); ++i)

cout << \*i << " ";

// inserts at the beginning

v.emplace(v.begin(), 5);

cout << "\nThe first element is: " << v[0];

// Inserts 20 at the end

v.emplace\_back(20);

int n = v.size();

cout << "\nThe last element is: " << v[n - 1];

// erases the vector

v.clear();

cout << "\nVector size after erase(): " << v.size();

return 0;

}

**Output**:

Size : 5

Vector is not empty

Output of begin and end: 1 2 3 4 5

The first element is: 5

The last element is: 20

Vector size after erase(): 0

### List

Lists are sequence containers that allow non-contiguous memory allocation. List in C++ STL implements a doubly linked list and not arrays. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked lists. For implementing a singly linked list, we can use **forward\_list** class in C++ STL.  
  
To use the List class, include the below header file in your program:

**#include< list >**

**Declaring List**: 

list< *Type\_of\_element* > *list\_name*;

Here, *Type\_of\_element* can be any valid C++ data type,

or can be any other container also like Pair, List etc.

Some important and commonly used functions of List are:

* **front()** – Returns the value of the first element in the list.
* **back()** – Returns the value of the last element in the list.
* **push\_front(g)** – Adds a new element ‘g’ at the beginning of the list.
* **push\_back(g)** – Adds a new element ‘g’ at the end of the list.
* **pop\_front()** – Removes the first element of the list, and reduces the size of the list by 1.
* **pop\_back()** – Removes the last element of the list, and reduces the size of the list by 1.
* **begin()** and **end()** – begin() function returns an iterator pointing to the first element of the list.
* **empty()** – Returns whether the list is empty(1) or not(0).
* **insert()** – Inserts new elements in the list before the element at a specified position.
* **reverse()** – Reverses the list.
* **size()** – Returns the number of elements in the list.
* **sort()** – Sorts the list in increasing order.

Below program illustrate the above functions:

CPP

#include <iostream>

#include <list>

#include <iterator>

using namespace std;

//function for printing the elements in a list

void showlist(list <int> g)

{

list <int> :: iterator it;

for(it = g.begin(); it != g.end(); ++it)

cout << '\t' << \*it;

cout << '\n';

}

int main()

{

list <int> gqlist1, gqlist2;

for (int i = 0; i < 10; ++i)

{

gqlist1.push\_back(i \* 2);

gqlist2.push\_front(i \* 3);

}

cout << "\nList 1 (gqlist1) is : ";

showlist(gqlist1);

cout << "\nList 2 (gqlist2) is : ";

showlist(gqlist2);

cout << "\ngqlist1.front() : " << gqlist1.front();

cout << "\ngqlist1.back() : " << gqlist1.back();

cout << "\ngqlist1.pop\_front() : ";

gqlist1.pop\_front();

showlist(gqlist1);

cout << "\ngqlist2.pop\_back() : ";

gqlist2.pop\_back();

showlist(gqlist2);

cout << "\ngqlist1.reverse() : ";

gqlist1.reverse();

showlist(gqlist1);

cout << "\ngqlist2.sort(): ";

gqlist2.sort();

showlist(gqlist2);

return 0;

}

**Output**:-

List 1 (gqlist1) is : 0 2 4 6 8 10 12 14 16 18

List 2 (gqlist2) is : 27 24 21 18 15 12 9 6 3 0

gqlist1.front() : 0

gqlist1.back() : 18

gqlist1.pop\_front() : 2 4 6 8 10 12 14 16 18

gqlist2.pop\_back() : 27 24 21 18 15 12 9 6 3

gqlist1.reverse() : 18 16 14 12 10 8 6 4 2

gqlist2.sort(): 3 6 9 12 15 18 21 24 27

Iterators in C++ STL

Iterators are used to point at the memory addresses of [STL](http://quiz.geeksforgeeks.org/the-c-standard-template-library-stl/) containers. They are primarily used in a sequence of numbers, characters etc. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.  
  
**Basic Operations of iterators** :-

* **begin()** :- This function is used to return the **beginning position** of the container.
* **end()** :- This function is used to return the***after* end position** of the container.CPP
* // C++ code to demonstrate the working of
* // iterator, begin() and end()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterator to a vector
* vector<int>::iterator ptr;
* // Displaying vector elements using begin() and end()
* cout << "The vector elements are : ";
* for (ptr = ar.begin(); ptr < ar.end(); ptr++)
* cout << \*ptr << " ";
* return 0;
* }

**Output:**

The vector elements are : 1 2 3 4 5

* **advance()**:- This function is used to **increment the iterator position**till the specified number mentioned in its arguments.CPP
* // C++ code to demonstrate the working of
* // advance()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterator to a vector
* vector<int>::iterator ptr = ar.begin();
* // Using advance() to increment iterator position
* // points to 4
* advance(ptr, 3);
* // Displaying iterator position
* cout << "The position of iterator after advancing is : ";
* cout << \*ptr << " ";
* return 0;
* }

**Output:**

The position of iterator after advancing is : 4

* **next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.
* **prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.CPP
* // C++ code to demonstrate the working of
* // next() and prev()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterators to a vector
* vector<int>::iterator ptr = ar.begin();
* vector<int>::iterator ftr = ar.end();

* // Using next() to return new iterator
* // points to 4
* auto it = next(ptr, 3);
* // Using prev() to return new iterator
* // points to 3
* auto it1 = prev(ftr, 3);
* // Displaying iterator position
* cout << "The position of new iterator using next() is : ";
* cout << \*it << " ";
* cout << endl;
* // Displaying iterator position
* cout << "The position of new iterator using prev() is : ";
* cout << \*it1 << " ";
* cout << endl;
* return 0;
* }

**Output:**

The position of new iterator using next() is : 4

The position of new iterator using prev() is : 3

* **inserter()** :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.CPP
* // C++ code to demonstrate the working of
* // inserter()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* vector<int> ar1 = {10, 20, 30};
* // Declaring iterator to a vector
* vector<int>::iterator ptr = ar.begin();
* // Using advance to set position
* advance(ptr, 3);
* // copying 1 vector elements in other using inserter()
* // inserts ar1 after 3rd position in ar
* copy(ar1.begin(), ar1.end(), inserter(ar,ptr));
* // Displaying new vector elements
* cout << "The new vector after inserting elements is : ";
* for (int &x : ar)
* cout << x << " ";
* return 0;
* }

**Output:**

The new vector after inserting elements is : 1 2 3 10 20 30 4 5

Iterators in C++ STL

Iterators are used to point at the memory addresses of [STL](http://quiz.geeksforgeeks.org/the-c-standard-template-library-stl/) containers. They are primarily used in a sequence of numbers, characters etc. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.  
  
**Basic Operations of iterators** :-

* **begin()** :- This function is used to return the **beginning position** of the container.
* **end()** :- This function is used to return the***after* end position** of the container.CPP
* // C++ code to demonstrate the working of
* // iterator, begin() and end()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterator to a vector
* vector<int>::iterator ptr;
* // Displaying vector elements using begin() and end()
* cout << "The vector elements are : ";
* for (ptr = ar.begin(); ptr < ar.end(); ptr++)
* cout << \*ptr << " ";
* return 0;
* }

**Output:**

The vector elements are : 1 2 3 4 5

* **advance()**:- This function is used to **increment the iterator position**till the specified number mentioned in its arguments.CPP
* // C++ code to demonstrate the working of
* // advance()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterator to a vector
* vector<int>::iterator ptr = ar.begin();
* // Using advance() to increment iterator position
* // points to 4
* advance(ptr, 3);
* // Displaying iterator position
* cout << "The position of iterator after advancing is : ";
* cout << \*ptr << " ";
* return 0;
* }

**Output:**

The position of iterator after advancing is : 4

* **next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.
* **prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.CPP
* // C++ code to demonstrate the working of
* // next() and prev()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* // Declaring iterators to a vector
* vector<int>::iterator ptr = ar.begin();
* vector<int>::iterator ftr = ar.end();

* // Using next() to return new iterator
* // points to 4
* auto it = next(ptr, 3);
* // Using prev() to return new iterator
* // points to 3
* auto it1 = prev(ftr, 3);
* // Displaying iterator position
* cout << "The position of new iterator using next() is : ";
* cout << \*it << " ";
* cout << endl;
* // Displaying iterator position
* cout << "The position of new iterator using prev() is : ";
* cout << \*it1 << " ";
* cout << endl;
* return 0;
* }

**Output:**

The position of new iterator using next() is : 4

The position of new iterator using prev() is : 3

* **inserter()** :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.CPP
* // C++ code to demonstrate the working of
* // inserter()
* #include<iostream>
* #include<iterator> // for iterators
* #include<vector> // for vectors
* using namespace std;
* int main()
* {
* vector<int> ar = { 1, 2, 3, 4, 5 };
* vector<int> ar1 = {10, 20, 30};
* // Declaring iterator to a vector
* vector<int>::iterator ptr = ar.begin();
* // Using advance to set position
* advance(ptr, 3);
* // copying 1 vector elements in other using inserter()
* // inserts ar1 after 3rd position in ar
* copy(ar1.begin(), ar1.end(), inserter(ar,ptr));
* // Displaying new vector elements
* cout << "The new vector after inserting elements is : ";
* for (int &x : ar)
* cout << x << " ";
* return 0;
* }

**Output:**

The new vector after inserting elements is : 1 2 3 10 20 30 4 5

Insertion and Deletion in Arrays

### Insertion in Arrays

Given an array of a given size. The task is to insert a new element in this array. There are two possible ways of inserting elements in an array:

1. Insert elements at the end of the array.
2. Insert element at any given index in the array.

**Special Case**: A special case is needed to be considered is that whether the array is already full or not. If the array is full, then the new element can not be inserted.  
  
  
  
Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array. 

* **Insert an element K at end in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

if(len < N)

// space left

else

// array is full

* If there is space left for the new element, insert it directly at the end at position **len + 1** and index **len**:

arr[len] = k;

* ***Time Complexity*** of this insert operation is constant, i.e. O(1) as we are directly inserting the element in a single operation.
* **Insert an element K at position, pos in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

if(len < N)

// space left

else

// array is full

* Now, if there is space left, the element can be inserted. The index of the new element will be **idx = pos - 1**. Now, before inserting the element at the index idx, shift all elements from the index idx till end of the array to the right by 1 place. This can be done as:

for(i = len-1; i >= idx; i--)

{

arr[i+1] = arr[i];

}

* After shifting the elements, place the element K at index idx.

arr[idx] = K;

* ***Time Complexity*** in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the right.

### Deletion in Arrays

To delete a given element from an array, we will have to first search the element in the array. If the element is present in the array then delete operation is performed for the element otherwise the user is notified that the array does not contains the given element.  
  
Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array.   
  
**Deleting an element K from the array arr[]**: Search the element K in the array arr[] to find the index at which it is present.

for(i = 0; i < N; i++)

{

if(arr[i] == K)

idx = i; return;

else

Element not Found;

}

Now, to delete the element present at index **idx**, left shift all of the elements present after *idx* by one place and finally reduce the length of the array by 1. 

for(i = idx+1; i < len; i++)

{

arr[i-1] = arr[i];

}

len = len-1;

***Time Complexity*** in worst case of this deletion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the left.

Sample Problems on Array

### **Problem #1 : Range Sum Queries using Prefix Sum**

**Description :** We are given an Array of **n** integers, We are given **q** queries having indices **l and r**. We have to find out sum between the given range of indices.

**Input**

[4, 5, 3, 2, 5]

3

0 3

2 4

1 3

**Output**

14 (4+5+3+2)

10 (3+2+5)

10 (5+3+2)

**Solution :** The numbers of queries are large. It will be very inefficient to iterate over the array and calculate the sum for each query separately. We have to devise the solution so that we can get the answer of the query in constant time. We will be storing the sum upto a particular index in prefix sum Array. We will be using the prefix sum array to calculate the sum for the given range.

prefix[] = Array stores the sum (A[0]+A[1]+....A[i]) at index i.

if l == 0 :

sum(l,r) = prefix[r]

else :

sum(l,r) = prefix[r] - prefix[l-1]

**Pseudo Code**

// n : size of array

// q : Number of queries

// l, r : Finding Sum of range between index l and r

// l and r (inclusive) and 0 based indexing

void range\_sum(arr, n)

{

prefix[n] = {0}

prefix[0] = arr[0]

for i = 1 to n-1 :

prefix[i] = a[i] + prefix[i-1]

for (i = 1 to q )

{

if (l == 0)

{

ans = prefix[r]

print(ans)

}

else

{

ans = prefix[r] - prefix[l-1]

print(ans)

}

}

}

**Time Complexity :** Max(O(n),O(q))  
**Auxiliary Space :** O(n)

### **Problem #2 : Equilibrium index of an array**

**Description -** Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. We are given an Array of integers, We have to find out the first index **i** from left such that -

A[0] + A[1] + ... A[i-1] = A[i+1] + A[i+2] ... A[n-1]

**Input**

[-7, 1, 5, 2, -4, 3, 0]

**Output**

3

A[0] + A[1] + A[2] = A[4] + A[5] + A[6]

**Naive Solution :** We can iterate for each index i and calculate the leftsum and rightsum and check whether they are equal.

for (i=0 to n-1)

{

leftsum = 0

for (j = 0 to i-1)

leftsum += arr[i]

rightsum = 0

for (j = i+1 to n-1)

rightsum += arr[i]

if leftsum == rightsum :

return i

}

**Time Complexity :** O(n^2)  
**Auxiliary Space :** O(1)

**Tricky Solution :** The idea is to first get the total sum of array. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get the right sum by subtracting the elements one by one. Then check whether the Leftsum and the Rightsum are equal.  
**Pseudo Code**

// n : size of array

int eqindex(arr, n)

{

sum = 0

leftsum = 0

for (i=0 to n-1)

sum += arr[i]

for (i=0 to n-1)

{

// now sum will be righsum for index i

sum -= a[i]

if (sum == leftsum )

return i

leftsum += a[i]

}

}

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

### **Problem #3 : Largest Sum Subarray**

**Description :** We are given an array of positive and negative integers. We have to find the subarray having maximum sum.

**Input**

[-3, 4, -1, -2, 1, 5]

**Output**

7

(4+(-1)+(-2)+1+5)

**Solution :**A simple idea is to look for all the positive contiguous segments of the array (max\_ending\_here is used for this), and keep the track of maximum sum contiguous segment among all the positive segments (max\_so\_far is used for this). Each time we get a positive sum compare it with max\_so\_far and if it is greater than max\_so\_far, update max\_so\_far.  
**Pseudo Code**

//n : size of array

int largestsum(arr, n)

{

max\_so\_far = INT\_MIN

max\_ending\_here = 0

for (i=0 to n-1)

{

max\_ending\_here += arr[i]

if max\_so\_far < max\_ending\_here :

max\_so\_far = max\_ending\_here

if max\_ending\_here < 0 :

max\_ending\_here = 0

}

return max\_so\_far

}

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

### **Problem #4 : Merge two sorted Arrays**

**Description :** We are given two sorted arrays **arr1[ ]** and **arr2[ ]**of size **m** and **n** respectively. We have to merge these arrays and store the numbers in arr3[ ] of size **m+n**.

**Input**

1 3 4 6

2 5 7 8

**Output**

1 2 3 4 5 6 7 8

**Solution :** The idea is to traverse both the arrays simultaneously and compare the current numbers from both the Arrays. Pick the smaller element and copy it to arr3[ ] and advance the current index of the array from where the smaller element is picked. When we reach at the end of one of the arrays, copy the remaining elements of another array to arr3[ ].  
**Pseudo Code**

// input arrays - arr1(size m), arr2(size n)

void merge\_sorted(arr1, arr2, m, n)

{

arr3[m+n] // merged array

i=0,j=0,k=0

while(i < m && j < n)

{

if arr1[i] < arr2[j] :

arr3[k++] = arr1[i++]

else :

arr3[k++] = arr2[j++]

}

while(i < m)

arr3[k++] = arr1[i++]

while(j < n)

arr3[k++] = arr2[j++]

}

**Time Complexity :** O(m+n)  
**Auxiliary Space :** O(m+n)

Largest Element in an Array

Given an array **arr** of size **N**, the task is to find the largest element in the given array.

**Example:**

***Input:****arr[] = {10, 20, 4}*  
***Output:****20*

***Input :****arr[] = {20, 10, 20, 4, 100}*  
***Output :****100*

**Approach 1 - Naive Method:**

C++Java

#include <bits/stdc++.h>

using namespace std;

using namespace std;

int getlargest(int arr[], int n)

{

for (int i = 0; i < n; ++i) {

bool flag = true;

for (int j = i; j < n; ++j) {

if (arr[j] > arr[i]) {

flag = false;

break;

}

}

if (flag == true) {

return arr[i];

}

}

return -1;

}

int main()

{

int arr[] = { 5, 8, 20, 15 };

cout << "Largest in given array is "

<< getlargest(arr, 4);

return 0;

}

**Output**

Largest in given array is 20

**Approach 2 – Linear Traversal:** One of the most simplest and basic approach to solve this problem is to simply traverse the whole list and find the maximum among them.

Follow the steps below to implement this idea:

* Create a local variable **max**to store the maximum among the list
* **Initialize max with the first element** initially, to start the comparison.
* Then **traverse the given array** from second element till end, and for each element:
  + **Compare the current element with max**
  + If the current element is greater than max, then **replace** the value of **max** with the current element.
* At the end, **return** and print the value of the largest element of array stored in **max**.

Below is the implementation of the above approach:

C++Java

// C++ program to find maximum

// in arr[] of size n

#include <bits/stdc++.h>

using namespace std;

int largest(int arr[], int n)

{

int i;

// Initialize maximum element

int max = arr[0];

// Traverse array elements

// from second and compare

// every element with current max

for (i = 1; i < n; i++)

if (arr[i] > max)

max = arr[i];

return max;

}

// Driver Code

int main()

{

int arr[] = {10, 324, 45, 90, 9808};

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

cout << "Largest in given array is "

<< largest(arr, n);

return 0;

}

**Output**

Largest in given array is 9808

Second Largest Element in Array

Given an array of integers, our task is to write a program that efficiently finds the second largest element present in the array.

**Example:**

**Input:** arr[] = {12, 35, 1, 10, 34, 1}

**Output:** The second largest element is 34.

**Explanation:** The largest element of the

array is 35 and the second

largest element is 34

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

**Output:** The second largest element is 5.

**Explanation:** The largest element of

the array is 10 and the second

largest element is 5

**Input:** arr[] = {10, 10, 10}

**Output:** The second largest does not exist.

**Explanation:** Largest element of the array

is 10 there is no second largest element

**Efficient Solution:**

**Approach:** Find the second largest element in a single traversal.   
Below is the complete algorithm for doing this:

1) Initialize the first as 0(i.e, index of arr[0] element

2) Start traversing the array from array[1],

a) If the current element in array say arr[i] is greater

than first. Then update first and second as,

second = first

first = arr[i]

b) If the current element is in between first and second,

then update second to store the value of current variable as

second = arr[i]

3) Return the value stored in second.

**Implementation:**

C++Java

// C++ program to find the second largest element

#include <iostream>

using namespace std;

// returns the index of second largest

// if second largest didn't exist return -1

int secondLargest(int arr[], int n) {

int first = 0, second = -1;

for (int i = 1; i < n; i++) {

if (arr[i] > arr[first]) {

second = first;

first = i;

}

else if (arr[i] < arr[first]) {

if (second == -1 || arr[second] < arr[i])

second = i;

}

}

return second;

}

int main() {

int arr[] = {10, 12, 20, 4};

int index = secondLargest(arr, sizeof(arr)/sizeof(arr[0]));

if (index == -1)

cout << "Second Largest didn't exist";

else

cout << "Second largest : " << arr[index];

}

**Output**

The second largest element is 12

**Complexity Analysis:**

* **Time Complexity: O(n).**  
  Only one traversal of the array is needed.
* **Auxiliary space:** **O(1).**  
  As no extra space is required.

Check if an Array is Sorted

Given an array of size **n**, write a program to check if it is sorted in ascending order or not. Equal values are allowed in an array and two consecutive equal values are considered sorted.

**Examples:**

Input : 20 21 45 89 89 90

Output : Yes

Input : 20 20 45 89 89 90

Output : Yes

Input : 20 20 78 98 99 97

Output : No

**Naive Approach:**

C++Java

#include <iostream>

#include <cmath>

using namespace std;

bool isSorted(int arr[], int n)

{

for(int i = 0; i < n; i++)

{

for(int j = i + 1; j < n; j++)

{

if(arr[j] < arr[i])

return false;

}

}

return true;

}

int main() {

int arr[] = {7, 2, 30, 10}, n = 4;

printf("%s", isSorted(arr, n)? "true": "false");

}

**Output:**

false

**Iterative approach:**

C++Java

// C++ program to check if an

// Array is sorted or not

#include <bits/stdc++.h>

using namespace std;

// Function that returns true if array is

// sorted in non-decreasing order.

bool arraySortedOrNot(int arr[], int n)

{

// Array has one or no element

if (n == 0 || n == 1)

return true;

for (int i = 1; i < n; i++)

// Unsorted pair found

if (arr[i - 1] > arr[i])

return false;

// No unsorted pair found

return true;

}

// Driver code

int main()

{

int arr[] = { 20, 23, 23, 45, 78, 88 };

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

if (arraySortedOrNot(arr, n))

cout << "Yes\n";

else

cout << "No\n";

}

**Output**

Yes

**Time Complexity: O(n)**  
**Auxiliary Space: O(1)**

Reverse an Array

Given an array (or string), the task is to reverse the array/string.  
**Examples :** 

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

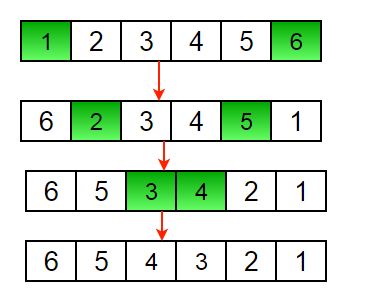
Output : arr[] = {3, 2, 1}

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

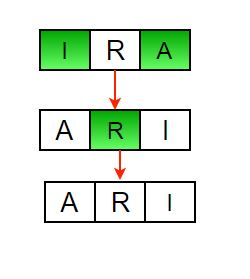
Output : arr[] = {2, 1, 5, 4}

**Iterative way :**

*1) Initialize start and end indexes as start = 0, end = n-1*  
*2) In a loop, swap arr[start] with arr[end] and change start and end as follows :*  
*start = start +1, end = end – 1*



Another example to reverse a string:



Below is the implementation of the above approach :

C++Java

// Iterative C++ program to reverse an array

#include <bits/stdc++.h>

using namespace std;

/\* Function to reverse arr[] from start to end\*/

void rvereseArray(int arr[], int start, int end)

{

while (start < end)

{

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

/\* Utility function to print an array \*/

void printArray(int arr[], int size)

{

for (int i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

/\* Driver function to test above functions \*/

int main()

{

int arr[] = {1, 2, 3, 4, 5, 6};

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

// To print original array

printArray(arr, n);

// Function calling

rvereseArray(arr, 0, n-1);

cout << "Reversed array is" << endl;

// To print the Reversed array

printArray(arr, n);

return 0;

}

**Output :**

1 2 3 4 5 6

Reversed array is

6 5 4 3 2 1

**Time Complexity :** O(n)

Remove duplicates from a sorted array

Given a sorted array, the task is to remove the duplicate elements from the array.

**Examples:**

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

Output : arr[] = {2}

new size = 1

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

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

new size = 5

**Method 1:** (Using extra space)

C++Java

#include <iostream>

#include <cmath>

using namespace std;

int remDups(int arr[], int n)

{

int temp[n];

temp[0] = arr[0];

int res = 1;

for(int i = 1; i < n; i++)

{

if(temp[res - 1] != arr[i])

{

temp[res] = arr[i];

res++;

}

}

for(int i = 0; i < res; i++)

{

arr[i] = temp[i];

}

return res;

}

int main() {

int arr[] = {10, 20, 20, 30, 30, 30}, n = 6;

cout<<"Before Removal"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

cout<<endl;

n = remDups(arr, n);

cout<<"After Removal"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

}

**Output**

**Before Removal**

**10 20 20 30 30 30**

**After Removal**

**10 20 30**

**Time Complexity : O(n)**  
**Auxiliary Space : O(n)**

**Method 2:** (Constant extra space)

C++Java

#include <iostream>

#include <cmath>

using namespace std;

int remDups(int arr[], int n)

{

int res = 1;

for(int i = 1; i < n; i++)

{

if(arr[res - 1] != arr[i])

{

arr[res] = arr[i];

res++;

}

}

return res;

}

int main() {

int arr[] = {10, 20, 20, 30, 30, 30}, n = 6;

cout<<"Before Removal"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

cout<<endl;

n = remDups(arr, n);

cout<<"After Removal"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

}

**Output**

**Before Removal**

**10 20 20 30 30 30**

**After Removal**

**10 20 30**

**Time Complexity : O(n)**  
**Auxiliary Space : O(1)**

Move Zeros to End

Given an array of **n** numbers. The problem is to move all the 0’s to the end of the array while maintaining the order of the other elements. Only single traversal of the array is required.  
Examples: 

Input : arr[] = {1, 2, 0, 0, 0, 3, 6}

Output : 1 2 3 6 0 0 0

Input: arr[] = {0, 1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9}

Output: 1 9 8 4 2 7 6 9 0 0 0 0 0

**Algorithm:** 

moveZerosToEnd(arr, n)

Initialize count = 0

for i = 0 to n-1

if (arr[i] != 0) then

arr[count++]=arr[i]

for i = count to n-1

arr[i] = 0

C++Java

// C++ implementation to move all zeroes at the end of array

#include <iostream>

using namespace std;

// function to move all zeroes at the end of array

void moveZerosToEnd(int arr[], int n)

{

// Count of non-zero elements

int count = 0;

// Traverse the array. If arr[i] is non-zero, then

// update the value of arr at index count to arr[i]

for (int i = 0; i < n; i++)

if (arr[i] != 0)

arr[count++] = arr[i];

// Update all elements at index >=count with value 0

for (int i = count; i < n; i++)

arr[i] = 0;

}

// function to print the array elements

void printArray(int arr[], int n)

{

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

}

// Driver program to test above

int main()

{

int arr[] = { 0, 1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9 };

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

cout << "Original array: ";

printArray(arr, n);

moveZerosToEnd(arr, n);

cout << "\nModified array: ";

printArray(arr, n);

return 0;

}

**Output**

Original array: 0 1 9 8 4 0 0 2 7 0 6 0 9

Modified array: 1 9 8 4 2 7 6 9 0 0 0 0 0

Time Complexity: O(n).   
Auxiliary Space: O(1).

Left Rotate an Array by D places

Given an array of integers **arr[]** of size **N** and an integer, the task is to rotate the array elements to the **left** by **d** positions.

**Examples:**

***Input:***  
*arr[] = {1, 2, 3, 4, 5, 6, 7}, d = 2*  
***Output:****3 4 5 6 7 1 2*

***Input:****arr[] = {3, 4, 5, 6, 7, 1, 2}, d=2*  
***Output:****5 6 7 1 2 3 4*

***Naive:***

C++Java

#include <iostream>

#include <cmath>

using namespace std;

void lRotateOne(int arr[], int n)

{

int temp = arr[0];

for(int i = 1; i < n; i++)

{

arr[i - 1] = arr[i];

}

arr[n - 1] = temp;

}

void leftRotate(int arr[], int d, int n)

{

for(int i = 0; i < d; i++)

{

lRotateOne(arr, n);

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5}, n = 5, d = 2;

cout<<"Before Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

cout<<endl;

leftRotate(arr, d, n);

cout<<"After Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

}

**Output:**

Before Rotation

1 2 3 4 5

After Rotation

3 4 5 1 2

**ReversaL Method:**

C++Java

#include <iostream>

#include <cmath>

using namespace std;

void reverse(int arr[], int low, int high)

{

while(low < high)

{

swap(arr[high], arr[low]);

low++;

high--;

}

}

void leftRotate(int arr[], int d, int n)

{

reverse(arr, 0, d - 1);

reverse(arr, d, n - 1);

reverse(arr, 0, n - 1);

}

int main() {

int arr[] = {1, 2, 3, 4, 5}, n = 5, d = 2;

cout<<"Before Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

cout<<endl;

leftRotate(arr, d, n);

cout<<"After Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

}

**Output:**

Before Rotation

1 2 3 4 5

After Rotation

3 4 5 1 2

**Efficient Approach:**

C++Java

#include <iostream>

#include <cmath>

using namespace std;

void leftRotate(int arr[], int d, int n)

{

int temp[d];

for(int i = 0; i < d; i++)

{

temp[i] = arr[i];

}

for(int i = d; i < n; i++)

{

arr[i - d] = arr[i];

}

for(int i = 0; i < d; i++)

{

arr[n - d + i] = temp[i];

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5}, n = 5, d = 2;

cout<<"Before Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

cout<<endl;

leftRotate(arr, d, n);

cout<<"After Rotation"<<endl;

for(int i = 0; i < n; i++)

{

cout<<arr[i]<<" ";

}

}

**Output:**

Before Rotation

1 2 3 4 5

After Rotation

3 4 5 1 2

Leaders in an Array problem

Write a program to print all the LEADERS in the array. An element is a leader if it is greater than all the elements to its right side. And the rightmost element is always a leader.

**For example:**

***Input:****arr[] = {16, 17, 4, 3, 5, 2},*  
***Output:****17, 5, 2*

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

**Naive Approach**: The problem can be solved based on the idea mentioned below:

*Use two loops. The outer loop runs from 0 to size – 1 and one by one pick all elements from left to right. The inner loop compares the picked element to all the elements on its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader.*

Follow the below steps to implement the idea:

* We run a loop from the first index to the 2nd last index.
  + And for each index, we run another loop from the next index to the last index.
  + If all the values to the right of that index are smaller than the index, we simply add the value in our answer data structure.

Below is the implementation of the above approach.

C++Java

#include<iostream>

using namespace std;

/\*C++ Function to print leaders in an array \*/

void printLeaders(int arr[], int size)

{

for (int i = 0; i < size; i++)

{

int j;

for (j = i+1; j < size; j++)

{

if (arr[i] <=arr[j])

break;

}

if (j == size) // the loop didn't break

cout << arr[i] << " ";

}

}

/\* Driver program to test above function \*/

int main()

{

int arr[] = {16, 17, 4, 3, 5, 2};

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

printLeaders(arr, n);

return 0;

}

**Output**

17 5 2

**Time Complexity:** O(N \* N)  
**Auxiliary Space:**O(1)

### **Find Leader by finding suffix maximum:**

*The idea is to scan all the elements from right to left in an array and keep track of the maximum till now. When the maximum changes its value, print it.*

Follow the below illustration for a better understanding

**Illustration:**

*Let the array be arr[] = {16, 17, 4, 3, 5, 2}*

* *arr[] = {16, 17, 4, 3, 5,****2****} , max\_from\_right = 2 , ans[] = { 2 }*
* *arr[] = {16, 17, 4, 3,****5****, 2} , max\_from\_right = 5 , ans[] = { 2, 5 }*
* *arr[] = {16, 17, 4,****3****, 5, 2} , max\_from\_right = 5 , ans[] = { 2, 5 }*
* *arr[] = {16, 17,****4****, 3,****5****, 2} , max\_from\_right = 5 , ans[] = { 2, 5 }*
* *arr[] = {16,****17****, 4, 3, 5, 2} , max\_from\_right = 17 , ans[] = { 2, 5, 17 }*
* *arr[] = {****16****, 17, 4, 3, 5, 2} , max\_from\_right = 17 , ans[] = { 2, 5, 17 }*

Follow the steps mentioned below to implement the idea:

* We start from the last index position. The last position is always a leader, as there are no elements towards its right.
* And then we iterate on the array till we reach index position = 0.
  + Each time we keep a check on the maximum value
  + Every time we encounter a maximum value than the previous maximum value encountered, we either print or store the value as it is the leader

Below is the implementation of the above approach.

C++Java

#include <iostream>

using namespace std;

/\* C++ Function to print leaders in an array \*/

void printLeaders(int arr[], int size)

{

int max\_from\_right = arr[size-1];

/\* Rightmost element is always leader \*/

cout << max\_from\_right << " ";

for (int i = size-2; i >= 0; i--)

{

if (max\_from\_right < arr[i])

{

max\_from\_right = arr[i];

cout << max\_from\_right << " ";

}

}

}

/\* Driver program to test above function\*/

int main()

{

int arr[] = {16, 17, 4, 3, 5, 2};

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

printLeaders(arr, n);

return 0;

}

**Output**

2 5 17

**Time Complexity:** O(n)  
**Auxiliary Space:**O(1)

Maximum Difference Problem with Order

Given an array arr[] of integers, find out the maximum difference between any two elements such that larger element appears after the smaller number.

**Examples :**

Input : arr = {2, 3, 10, 6, 4, 8, 1}

Output : 8

Explanation : The maximum difference is between 10 and 2.

Input : arr = {7, 9, 5, 6, 3, 2}

Output : 2

Explanation : The maximum difference is between 9 and 7.

**Method 1 (Simple)**   
Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far. Below is the implementation of the above approach :

C++Java

// C++ program to find Maximum difference

// between two elements such that larger

// element appears after the smaller number

#include <bits/stdc++.h>

using namespace std;

/\* The function assumes that there are

at least two elements in array. The

function returns a negative value if the

array is sorted in decreasing order and

returns 0 if elements are equal \*/

int maxDiff(int arr[], int arr\_size)

{

int max\_diff = arr[1] - arr[0];

for (int i = 0; i < arr\_size; i++)

{

for (int j = i+1; j < arr\_size; j++)

{

if (arr[j] - arr[i] > max\_diff)

max\_diff = arr[j] - arr[i];

}

}

return max\_diff;

}

/\* Driver program to test above function \*/

int main()

{

int arr[] = {1, 2, 90, 10, 110};

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

// Function calling

cout << "Maximum difference is " << maxDiff(arr, n);

return 0;

}

**Output :**

Maximum difference is 109

**Time Complexity :** O(n^2)   
**Auxiliary Space :** O(1)

**Method 2 (Tricky and Efficient)**   
In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:   
1) Maximum difference found so far (max\_diff).   
2) Minimum number visited so far (min\_element).

C++Java

// C++ program to find Maximum difference

// between two elements such that larger

// element appears after the smaller number

#include <bits/stdc++.h>

using namespace std;

/\* The function assumes that there are

at least two elements in array. The

function returns a negative value if the

array is sorted in decreasing order and

returns 0 if elements are equal \*/

int maxDiff(int arr[], int arr\_size)

{

// Maximum difference found so far

int max\_diff = arr[1] - arr[0];

// Minimum number visited so far

int min\_element = arr[0];

for(int i = 1; i < arr\_size; i++)

{

if (arr[i] - min\_element > max\_diff)

max\_diff = arr[i] - min\_element;

if (arr[i] < min\_element)

min\_element = arr[i];

}

return max\_diff;

}

/\* Driver program to test above function \*/

int main()

{

int arr[] = {1, 2, 90, 10, 110};

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

// Function calling

cout << "Maximum difference is " << maxDiff(arr, n);

return 0;

}

**Output:**

Maximum difference is 109

**Time Complexity :** O(n)   
**Auxiliary Space :** O(1)

Frequencies in a Sorted Array

Given a sorted [array](https://www.geeksforgeeks.org/array-data-structure/), **arr[]** consisting of **N** integers, the task is to find[the](https://www.geeksforgeeks.org/counting-frequencies-of-array-elements/)frequencies of [each array element](https://www.geeksforgeeks.org/counting-frequencies-of-array-elements/).

**Examples:**

***Input:****arr[] = {1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10}*  
***Output:****Frequency of 1 is: 3*  
*Frequency of 2 is: 1*  
*Frequency of 3 is: 2*  
*Frequency of 5 is: 2*  
*Frequency of 8 is: 3*  
*Frequency of 9 is: 2*  
*Frequency of 10 is: 1*

***Input:****arr[] = {2, 2, 6, 6, 7, 7, 7, 11}*  
***Output:****Frequency of 2 is: 2*  
*Frequency of 6 is: 2*  
*Frequency of 7 is: 3*  
*Frequency of 11 is: 1*

**Naive Approach:**The simplest approach is to[traverse the array](https://www.geeksforgeeks.org/c-program-to-traverse-an-array/)and keep the count of every element encountered in a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/) and then, in the end, print the frequencies of every element by[traversing the HashMap.](https://www.geeksforgeeks.org/traverse-through-a-hashmap-in-java/) This approach is already implemented [here](https://www.geeksforgeeks.org/find-frequency-of-each-element-in-a-limited-range-array-in-less-than-on-time/).

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

**Efficient Approach:**The above approach can be optimized in terms of space used based on the fact that, in a sorted array, the same elements occur consecutively, so the idea is to maintain a variable to keep track of the frequency of elements while traversing the array. Follow the steps below to solve the problem:

* Initialize a variable, say **freq** as **1** to store the frequency of elements.
* [Iterate in the range](https://www.geeksforgeeks.org/range-based-loop-c/)**[1, N-1]**using the variable **i**and perform the following steps:
  + If the value of **arr[i]** is equal to **arr[i-1]**, increment **freq** by **1**.
  + Else print value the frequency of **arr[i-1]** obtained in **freq** and then update **freq** to **1**.
* Finally, after the above step, print the frequency of the last distinct element of the array as **freq**.

Below is the implementation of the above approach:

C++Java

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to print the frequency

// of each element of the sorted array

void printFreq(vector<int>& arr, int N)

{

// Stores the frequency of an element

int freq = 1;

// Traverse the array arr[]

for (int i = 1; i < N; i++) {

// If the current element is equal

// to the previous element

if (arr[i] == arr[i - 1]) {

// Increment the freq by 1

freq++;

}

// Otherwise,

else {

cout << "Frequency of " << arr[i - 1]

<< " is: " << freq << endl;

// Update freq

freq = 1;

}

}

// Print the frequency of the last element

cout << "Frequency of " << arr[N - 1] << " is: " << freq

<< endl;

}

// Driver Code

int main()

{

// Given Input

vector<int> arr

= { 1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10 };

int N = arr.size();

// Function Call

printFreq(arr, N);

return 0;

}

**Output**

Frequency of 1 is: 3

Frequency of 2 is: 1

Frequency of 3 is: 2

Frequency of 5 is: 2

Frequency of 8 is: 3

Frequency of 9 is: 2

Frequency of 10 is: 1

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(1)*

Stock Buy and Sell Problem

The cost of a stock on each day is given in an array. Find the maximum profit that you can make by buying and selling on those days. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

**Examples:**

***Input:****arr[] = {100, 180, 260, 310, 40, 535, 695}*  
***Output:****865*  
***Explanation:****Buy the stock on day 0 and sell it on day 3 => 310 – 100 = 210*  
*Buy the stock on day 4 and sell it on day 6 => 695 – 40 = 655*  
*Maximum Profit  = 210 + 655 = 865*

***Input:****arr[] = {4, 2, 2, 2, 4}*  
***Output:****2*  
***Explanation:****Buy the stock on day 1 and sell it on day 4 => 4 – 2 = 2*  
*Maximum Profit  = 2*

A simple approach is to try buying the stocks and selling them every single day when profitable and keep updating the maximum profit so far.

Follow the steps below to solve the problem:

* Try to buy every stock from **start** to **end – 1**
* After that again call the maxProfit function to calculate answer
* curr\_profit = price[j] – price[i] + maxProfit(start, i – 1) + maxProfit(j + 1, end)
* profit = max(profit, curr\_profit)

Below is the implementation of the above approach:

C++Java

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Function to return the maximum profit

// that can be made after buying and

// selling the given stocks

int maxProfit(int price[], int start, int end)

{

// If the stocks can't be bought

if (end <= start)

return 0;

// Initialise the profit

int profit = 0;

// The day at which the stock

// must be bought

for (int i = start; i < end; i++) {

// The day at which the

// stock must be sold

for (int j = i + 1; j <= end; j++) {

// If buying the stock at ith day and

// selling it at jth day is profitable

if (price[j] > price[i]) {

// Update the current profit

int curr\_profit

= price[j] - price[i]

+ maxProfit(price, start, i - 1)

+ maxProfit(price, j + 1, end);

// Update the maximum profit so far

profit = max(profit, curr\_profit);

}

}

}

return profit;

}

// Driver code

int main()

{

int price[] = { 100, 180, 260, 310, 40, 535, 695 };

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

cout << maxProfit(price, 0, n - 1);

return 0;

}

**Output**

865

**Time Complexity:** O(N2), Trying to buy every stock and exploring all possibilities.  
**Auxiliary Space:** O(1)

**Stock Buy Sell to Maximize Profit using Valley Peak Approach:**

In this approach, we just need to find the next greater element and subtract it from the current element so that the difference keeps increasing until we reach a minimum. If the sequence is a decreasing sequence, so the maximum profit possible is 0.

Follow the steps below to solve the problem:

* maxProfit = 0
* if price[i] > price[i – 1]
  + maxProfit = maxProfit + price[i] – price[i – 1]

Below is the implementation of the above approach:

C++Java

#include <iostream>

#include <cmath>

using namespace std;

int maxProfit(int price[], int n)

{

int profit = 0;

for(int i = 1; i < n; i++)

{

if(price[i] > price[i - 1])

profit += price[i] - price[i -1];

}

return profit;

}

int main() {

int arr[] = {1, 5, 3, 8, 12}, n = 5;

cout<<maxProfit(arr, n);

}

**Output**

13

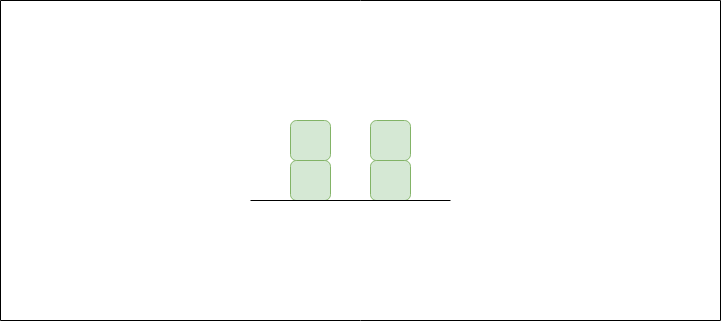
**Time Complexity**: O(N), Traversing over the array of size N.  
**Auxiliary** **Space:**O(1)

Trapping Rain Water

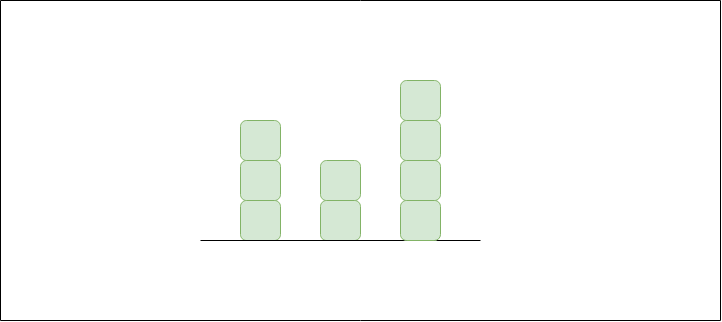
Given an array of **N** non-negative integers **arr[]** representing an elevation map where the width of each bar is **1**, compute how much water it is able to trap after raining.

**Examples**:

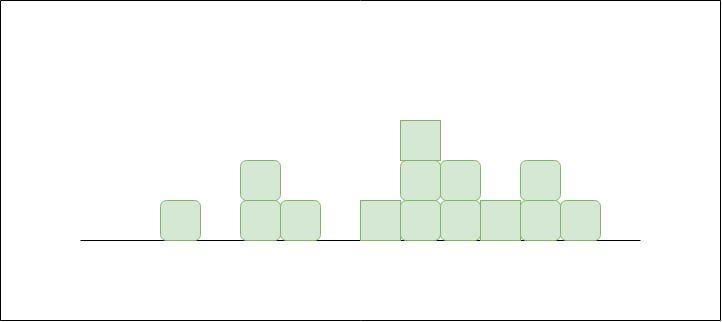
***Input:****arr[] = {2, 0, 2}*  
***Output:****2*  
***Explanation:****The structure is like below.*  
*We can trap 2 units of water in the middle gap.*

**

***Input:****arr[]   = {3, 0, 2, 0, 4}*  
***Output:****7*  
***Explanation:****Structure is like below.*  
*We can trap “3 units” of water between 3 and 2,*  
*“1 unit” on top of bar 2 and “3 units” between 2 and 4.*

**

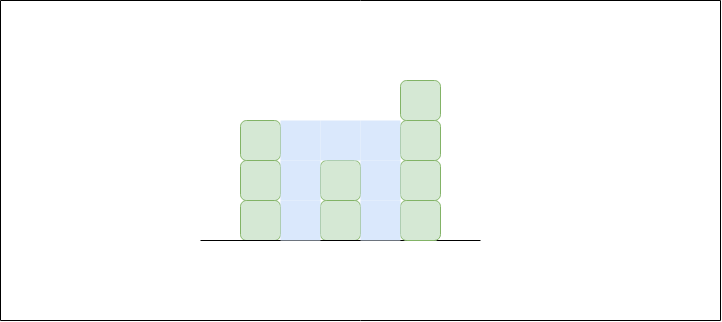
***Input:****arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}*  
***Output:****6*  
***Explanation:****The structure is like below.*  
*Trap “1 unit” between first 1 and 2, “4 units” between*  
*first 2 and 3 and “1 unit” between second last 1 and last 2*

**

**Intuition:** The basic intuition of the problem is as follows:

* *An element of the array can store water if there are higher bars on the left and the right.*
* *The amount of water to be stored in every position can be found by finding the heights of bars on the left and right sides.*
* *The total amount of water stored is the summation of the water stored in each index.*

***For example****– Consider the array****arr[] = {3, 0, 2, 0, 4}****.*  
*Three units of water can be stored in two indexes 1 and 3, and one unit of water at index 2.*  
*Water stored in each index = 0 + 3 + 1 + 3 + 0 =****7***

**

**Approach 1 (Brute Approach):** This approach is the **brute approach**. The idea is to:

*Traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and the height of the current element is the amount of water that can be stored in this array element.*

Follow the steps mentioned below to implement the idea:

* Traverse the array from start to end:
  + For every element:
    - Traverse the array from start to that index and find the maximum height *(a)* and
    - Traverse the array from the current index to the end, and find the maximum height *(b)*.
* The amount of water that will be stored in this column is min(a,b) – array[i], add this value to the total amount of water stored
* Print the total amount of water stored.

Below is the implementation of the above approach.

C++Java

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Function to return the maximum

// water that can be stored

int maxWater(int arr[], int n)

{

// To store the maximum water

// that can be stored

int res = 0;

// For every element of the array

for (int i = 1; i < n - 1; i++) {

// Find the maximum element on its left

int left = arr[i];

for (int j = 0; j < i; j++)

left = max(left, arr[j]);

// Find the maximum element on its right

int right = arr[i];

for (int j = i + 1; j < n; j++)

right = max(right, arr[j]);

// Update the maximum water

res = res + (min(left, right) - arr[i]);

}

return res;

}

// Driver code

int main()

{

int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

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

cout << maxWater(arr, n);

return 0;

}

**Output**

6

**Complexity Analysis:**

* **Time Complexity:** O(N2). There are two nested loops traversing the array.
* **Space Complexity:** O(1). No extra space is required.

**Approach 2 (Precalculation):** This is an efficient solution based on the precalculation concept:

*In previous approach, for every element we needed to calculate the highest element on the left and on the right.*

*So, to reduce the time complexity:*

* *For every element we can precalculate and store the highest bar on the left and on the right (say stored in arrays****left[]****and****right[]****).*
* *Then iterate the array and use the precalculated values to find the amount of water stored in this index,*  
  *which is the same as (****min(left[i], right[i]) – arr[i]****)*

Follow the below illustration for a better understanding:

**Illustration:**

*Consider****arr[] = {3, 0, 2, 0, 4}***

*Therefore,****left[] = {3, 3, 3, 3, 4}****and****right[] = {4, 4, 4, 4, 4}***  
*Now consider iterating using****i****from 0 to end*

***For i = 0:***  
*=> left[0] = 3, right[0] = 4 and arr[0] = 3*  
*=> Water stored = min(left[0], right[0]) – arr[0] = min(3, 4) – 3 = 3 – 3 =****0***  
*=> Total = 0 + 0 =****0***

***For i = 1:***  
*=> left[1] = 3, right[1] = 4 and arr[1] = 0*  
*=> Water stored = min(left[1], right[1]) – arr[1] = min(3, 4) – 0 = 3 – 0 =****3***  
*=> Total = 0 + 3 =****3***

***For i = 2:***  
*=> left[2] = 3, right[2] = 4 and arr[2] = 2*  
*=> Water stored = min(left[2], right[2]) – arr[2] = min(3, 4) – 2 = 3 – 2 =****1***  
*=> Total = 3 + 1 =****4***

***For i = 3:***  
*=> left[3] = 3, right[3] = 4 and arr[3] = 0*  
*=> Water stored = min(left[3], right[3]) – arr[3] = min(3, 4) – 0 = 3 – 0 =****3***  
*=> Total = 4 + 3 =****7***

***For i = 4:***  
*=> left[4] = 4, right[4] = 4 and arr[4] = 4*  
*=> Water stored = min(left[4], right[4]) – arr[4] = min(4, 4) – 4 = 4 – 4 =****0***  
*=> Total = 7 + 0 =****7***

*So total rain water trapped =****7***

Follow the steps mentioned below to implement the approach:

* Create two arrays **left[]** and **right[]** of size **N**. Create a variable (say **max**) to store the maximum found till a certain index during traversal.
* Run one loop from start to end:
  + In each iteration update max and also assign **left[i] = max**.
* Run another loop from end to start:
  + In each iteration update max found till now and also assign **right[i] = max**.
* Traverse the array from start to end.
  + The amount of water that will be stored in this column is **min(left[i], right[i]) – array[i]**
  + Add this value to the total amount of water stored
* Print the total amount of water stored.

Below is the implementation of the above approach.

C++Java

// C++ program to find maximum amount of water that can

// be trapped within given set of bars.

#include <bits/stdc++.h>

using namespace std;

int findWater(int arr[], int n)

{

// left[i] contains height of tallest bar to the

// left of i'th bar including itself

int left[n];

// Right [i] contains height of tallest bar to

// the right of ith bar including itself

int right[n];

// Initialize result

int water = 0;

// Fill left array

left[0] = arr[0];

for (int i = 1; i < n; i++)

left[i] = max(left[i - 1], arr[i]);

// Fill right array

right[n - 1] = arr[n - 1];

for (int i = n - 2; i >= 0; i--)

right[i] = max(right[i + 1], arr[i]);

// Calculate the accumulated water element by element

// consider the amount of water on i'th bar, the

// amount of water accumulated on this particular

// bar will be equal to min(left[i], right[i]) - arr[i]

// .

for (int i = 1; i < n - 1; i++) {

int var = min(left[i - 1], right[i + 1]);

if (var > arr[i]) {

water += var - arr[i];

}

}

return water;

}

// Driver program

int main()

{

int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

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

cout << findWater(arr, n);

return 0;

}

**Output**

6

**Complexity Analysis:**

* **Time Complexity:** O(N). Only one traversal of the array is needed, So time Complexity is O(N).
* **Space Complexity:** O(N). Two extra arrays are needed, each of size N.

Maximum Subarray Sum

Given an array **arr[]**, the task is to find the elements of a contiguous subarray of numbers that has the largest sum.

**Examples:**

***Input:****arr = [-2, -3, 4, -1, -2, 1, 5, -3]*  
***Output:****[4, -1, -2, 1, 5]*  
***Explanation:***  
*In the above input the maximum contiguous subarray sum is 7 and the elements of the subarray are [4, -1, -2, 1, 5]*

***Input:****arr = [-2, -5, 6, -2, -3, 1, 5, -6]*  
***Output:****[6, -2, -3, 1, 5]*  
***Explanation:***  
*In the above input the maximum contiguous subarray sum is 7 and the elements*  
*of the subarray are [6, -2, -3, 1, 5]*

**Naive Approach:** The naive approach is to generate all the possible subarray and print that subarray which has maximum sum.   
**Time complexity:** O(N2)   
**Auxiliary Space:** O(1)

**Efficient Approach:** The idea is to use the Kadane’s Algorithm to find the maximum subarray sum and store the starting and ending index of the subarray having maximum sum and print the subarray from starting index to ending index. Below are the steps:

1. Initialize 3 variables **endIndex** to 0, **currMax,** and **globalMax** to first value of the input array.
2. For each element in the array starting from index(say **i**) 1, update **currMax** to **max(nums[i], nums[i] + currMax)** and **globalMax** and **endIndex** to **i** only **if currMax > globalMax**.
3. To find the start index, iterate from endIndex in the left direction and keep decrementing the value of **globalMax** until it becomes 0. The point at which it becomes 0 is the start index.
4. Now print the subarray between **[start, end]**.

Below is the implementation of the above approach:

C++Java

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to print the elements

// of Subarray with maximum sum

void SubarrayWithMaxSum(vector<int>& nums)

{

// Initialize currMax and globalMax

// with first value of nums

int endIndex, currMax = nums[0];

int globalMax = nums[0];

// Iterate for all the elements

// of the array

for (int i = 1; i < nums.size(); ++i) {

// Update currMax

currMax = max(nums[i],

nums[i] + currMax);

// Check if currMax is greater

// than globalMax

if (currMax > globalMax) {

globalMax = currMax;

endIndex = i;

}

}

int startIndex = endIndex;

// Traverse in left direction to

// find start Index of subarray

while (startIndex >= 0) {

globalMax -= nums[startIndex];

if (globalMax == 0)

break;

// Decrement the start index

startIndex--;

}

// Printing the elements of

// subarray with max sum

for (int i = startIndex;

i <= endIndex; ++i) {

cout << nums[i] << " ";

}

}

// Driver Code

int main()

{

// Given array arr[]

vector<int> arr

= { -2, -5, 6, -2,

-3, 1, 5, -6 };

// Function call

SubarrayWithMaxSum(arr);

return 0;

}

**Output**

6 -2 -3 1 5

**Time complexity:** O(N)   
**Auxiliary Space:** O(1)

Longest Even Odd Subarray

Given an **array a[] of N** integers, the task is to find the length of the longest Alternating Even Odd [subarray](https://www.geeksforgeeks.org/array-subarray-subsequence-and-subset) present in the array.

**Examples:**

***Input:****a[] = {1, 2, 3, 4, 5, 7, 9}*  
***Output:****5*  
***Explanation:***  
*The subarray {1, 2, 3, 4, 5} has alternating even and odd elements.*

***Input:****a[] = {1, 3, 5}*  
***Output:****0*  
***Explanation:***  
*There is no such alternating sequence possible.*

**Naive approach:**

*The idea is to consider every subarray and find the length of even and odd subarrays.*

Follow the steps below to solve the problem:

* Iterate for every subarray from i = 0
* Make a nested loop, iterate from j = i + 1
* Now, check if a[j – 1] is even and a[j] is odd or a[j – 1] is odd and a[j] is even then increment count
* Maintain an answer variable which calculates max count so far

Below is the implementation of the above approach:

C++Java

#include <iostream>

using namespace std;

// Function to find the longest subarray

int longestEvenOddSubarray(int a[], int n)

{

// Length of longest

// alternating subarray

int ans = 1;

// Iterate in the array

for (int i = 0; i < n; i++) {

int cnt = 1;

// Iterate for every subarray

for (int j = i + 1; j < n; j++) {

if ((a[j - 1] % 2 == 0 && a[j] % 2 != 0)

|| (a[j - 1] % 2 != 0 && a[j] % 2 == 0))

cnt++;

else

break;

}

// store max count

ans = max(ans, cnt);

}

// Length of 'ans' can never be 1

// since even odd has to occur in pair or more

// so return 0 if ans = 1

if (ans == 1)

return 0;

return ans;

}

/\* Driver code\*/

int main()

{

int a[] = { 1, 2, 3, 4, 5, 7, 8 };

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

cout << longestEvenOddSubarray(a, n);

return 0;

}

**Output**

5

**Time Complexity:**O(N2), Iterating over every subarray therefore N2 are possible  
**Auxiliary Space:**O(1)

**Length of the longest alternating even odd subarray by Storing the previous element**

By simply storing the nature of the previous element we encounter( odd or even) and comparing it with the next element.

Follow the steps below to solve the problem:

* Initialize a variable maxLength to 0, to keep the track of maximum length of the alternating subarray obtained.
* Initialize a variable currLen to 1 considering first element as the part of alternating subarray.
* Starting with element at index 1, compare every element with it’s previous. If there nature are different, increment the currLen variable.
* Otherwise, reset the currLen to 1 again so that, this current element is considered in new alternating subarray.
* Keep storing the max length of subarray in maxLength before resetting the currLen.
* Return the found max length of subarray.

Below is the implementation of above approach:

C++Java

// C++ code to find longest subarray of alternating even and

// odds

#include <iostream>

using namespace std;

int maxEvenOdd(int arr[], int n)

{

if (n == 0)

return 0;

int maxLength = 0;

int currLen = 1;

for (int i = 1; i < n; i++) {

// everytime we check if previous

// element has opposite even/odd

// nature or not

if (arr[i] % 2 != arr[i-1] % 2)

currLen++;

else

{

// store max in maxLength

maxLength = max(maxLength, currLen);

// reset value when pattern is broken

currLen = 1;

}

}

// since, even-odd should occur in pair

if(maxLength == 1)

return 0;

return maxLength;

}

// Driver Code

int main()

{

int arr[] = { 1, 2, 3, 4, 5, 3, 7, 2, 9, 4 };

// longest subarray should be 1 2 3 4 5 , therefore

// length = 5

int n = sizeof(arr) / sizeof(int);

cout << "Length of longest subarray of even and odds "

"is : "

<< maxEvenOdd(arr, n);

return 0;

}

**Output**

**Length of longest subarray of even and odds is : 5**

**Time Complexity:** O(N), Since we need to iterate over the whole array once  
**Auxiliary Space:** O(1)

Maximum Circular Sum Subarray

Given a **circular array** of size n, find the **maximum subarray sum** of the non-empty subarray.

**Examples:**

***Input:****arr[] = {8, -8, 9, -9, 10, -11, 12}*  
***Output:****22*  
***Explanation:****Subarray 12, 8, -8, 9, -9, 10 gives the maximum sum, that is****22****.*

***Input:****arr[] = {10, -3, -4, 7, 6, 5, -4, -1}*  
***Output:****23*  
***Explanation:****Subarray 7, 6, 5, -4, -1, 10 gives the maximum sum, that is****23****.*

***Input:****arr[] = {-1, 40, -14, 7, 6, 5, -4, -1}*  
***Output:****52*  
***Explanation:****Subarray 7, 6, 5, -4, -1, -1, 40 gives the maximum sum, that is****52****.*

***Naive Approach:***

C++Java

#include <iostream>

#include <cmath>

using namespace std;

int maxCircularSum(int arr[], int n)

{

int res = arr[0];

for(int i = 0; i < n; i++)

{

int curr\_max = arr[i];

int curr\_sum = arr[i];

for(int j = 1; j < n; j++)

{

int index = (i + j) % n;

curr\_sum += arr[index];

curr\_max = max(curr\_max, curr\_sum);

}

res = max(res, curr\_max);

}

return res;

}

int main() {

int arr[] = {5, -2, 3, 4}, n = 4;

cout<<maxCircularSum(arr, n);

}

**Output**

12

**Efficient Approach:**

C++Java

#include <iostream>

#include <cmath>

using namespace std;

int normalMaxSum(int arr[], int n)

{

int res = arr[0];

int maxEnding = arr[0];

for(int i = 1; i < n; i++)

{

maxEnding = max(maxEnding + arr[i], arr[i]);

res = max(maxEnding, res);

}

return res;

}

int overallMaxSum(int arr[], int n)

{

int max\_normal = normalMaxSum(arr, n);

if(max\_normal < 0)

return max\_normal;

int arr\_sum = 0;

for(int i = 0; i < n; i++)

{

arr\_sum += arr[i];

arr[i] = -arr[i];

}

int max\_circular = arr\_sum + normalMaxSum(arr, n);

return max(max\_circular, max\_normal);

}

int main() {

int arr[] = {8, -4, 3, -5, 4}, n = 5;

cout<<overallMaxSum(arr, n);

}

**Output**

12

Majority Element

Find the majority element in the array. A ***majority element*** in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

**Examples :**

***Input :****{3, 3, 4, 2, 4, 4, 2, 4, 4}*  
***Output :****4*  
***Explanation:****The frequency of 4 is 5 which is greater than the half of the size of the array size.*

***Input :****{3, 3, 4, 2, 4, 4, 2, 4}*  
***Output :****No Majority Element*  
***Explanation:****There is no element whose frequency is greater than the half of the size of the array size.*

*The basic solution is to have two loops and keep track of the****maximum****count for all different elements. If the maximum count becomes greater than****n/2****then break the loops and return the element having the maximum count. If the maximum count doesn’t become more than n/2 then the majority element****doesn’t****exist.*

**Illustration:**

***arr[] = {3, 4, 3, 2, 4, 4, 4, 4}, n = 8***

*For i = 0:*

* *count = 0*
* *Loop over the array, whenever an element is equal to arr[i] (is****3****), increment count*
* *count of arr[i] is****2,****which is less than****n/2,****hence it can’t be****majority element.***

*For i = 1:*

* *count = 0*
* *Loop over the array, whenever an element is equal to arr[i] (is****4****), increment count*
* *count of arr[i] is****5,****which is****greater****than****n/2****(i.e 4)****,****hence it will be****majority element.***

*Hence,****4****is the****majority element****.*

Follow the steps below to solve the given problem:

* Create a variable to store the max count, *count = 0*
* Traverse through the array from start to end.
* For every element in the array run another loop to find the count of similar elements in the given array.
* If the count is greater than the max count update the max count and store the index in another variable.
* If the maximum count is greater than half the size of the array, print the element. Else print there is no majority element.

Below is the implementation of the above idea:

C++Java

// C++ program to find Majority

// element in an array

#include <bits/stdc++.h>

using namespace std;

// Function to find Majority element

// in an array

void findMajority(int arr[], int n)

{

int maxCount = 0;

int index = -1; // sentinels

for (int i = 0; i < n; i++) {

int count = 0;

for (int j = 0; j < n; j++) {

if (arr[i] == arr[j])

count++;

}

// update maxCount if count of

// current element is greater

if (count > maxCount) {

maxCount = count;

index = i;

}

}

// if maxCount is greater than n/2

// return the corresponding element

if (maxCount > n / 2)

cout << arr[index] << endl;

else

cout << "No Majority Element" << endl;

}

// Driver code

int main()

{

int arr[] = { 1, 1, 2, 1, 3, 5, 1 };

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

// Function calling

findMajority(arr, n);

return 0;

}

**Output**

1

**Time Complexity:** O(n\*n), A nested loop is needed where both the loops traverse the array from start to end.  
**Auxiliary Space:** O(1), No extra space is required.

**Majority Element Using Moore’s Voting Algorithm:**

*This is a two-step process:*

* *The first step gives the element that may be the majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise, it will return candidate for majority element.*
* *Check if the element obtained from the above step is the majority element. This step is necessary as there might be no majority element.*

**Illustration:**

***arr[] = {3, 4, 3, 2, 4, 4, 4, 4}, n = 8***

*maj\_index = 0, count = 1*

*At****i = 1****: arr[maj\_index] != arr[i]*

* *count = count – 1 = 1 – 1 = 0*
* *now count == 0 then:*
  + *maj\_index = i = 1*
  + *count = count + 1 = 0 + 1 = 1*

*At****i = 2****: arr[maj\_index] != arr[i]*

* *count = count – 1 = 1 – 1 = 0*
* *now count == 0 then:*
  + *maj\_index = i = 2*
  + *count = count + 1 = 0 + 1 = 1*

*At****i = 3****: arr[maj\_index] != arr[i]*

* *count = count – 1 = 1 – 1 = 0*
* *now count == 0 then:*
  + *maj\_index = i = 3*
  + *count = count + 1 = 0 + 1 = 1*

*At****i = 4****: arr[maj\_index] != arr[i]*

* *count = count – 1 = 1 – 1 = 0*
* *now count == 0 then:*
  + *maj\_index = i = 4*
  + *count = count + 1 = 0 + 1 = 1*

*At****i = 5****: arr[maj\_index] == arr[i]*

* *count = count + 1 = 1 + 1 = 2*

*At****i = 6****: arr[maj\_index] == arr[i]*

* *count = count + 1 = 2 + 1 = 3*

*At****i = 7****: arr[maj\_index] == arr[i]*

* *count = count + 1 = 3 + 1 = 4*

*Therefore, the****arr[maj\_index]****may be the possible candidate for majority element.*

*Now, Again traverse the array and check whether****arr[maj\_index]****is the majority element or not.*

***arr[maj\_index] is 4***

***4****occurs****5 times****in the array therefore 4 is our****majority element.***

Follow the steps below to solve the given problem:

* Loop through each element and maintains a count of the majority element, and a majority index, *maj\_index*
* If the next element is the same then increment the count if the next element is not the same then decrement the count.
* if the count reaches 0 then change the maj\_index to the current element and set the count again to 1.
* Now again traverse through the array and find the count of the majority element found.
* If the count is greater than half the size of the array, print the element
* Else print that there is no majority element

Below is the implementation of the above idea:

C++Java

// C++ Program for finding out

// majority element in an array

#include <bits/stdc++.h>

using namespace std;

/\* Function to find the candidate for Majority \*/

int findCandidate(int a[], int size)

{

int maj\_index = 0, count = 1;

for (int i = 1; i < size; i++) {

if (a[maj\_index] == a[i])

count++;

else

count--;

if (count == 0) {

maj\_index = i;

count = 1;

}

}

return a[maj\_index];

}

/\* Function to check if the candidate

occurs more than n/2 times \*/

bool isMajority(int a[], int size, int cand)

{

int count = 0;

for (int i = 0; i < size; i++)

if (a[i] == cand)

count++;

if (count > size / 2)

return 1;

else

return 0;

}

/\* Function to print Majority Element \*/

void printMajority(int a[], int size)

{

/\* Find the candidate for Majority\*/

int cand = findCandidate(a, size);

/\* Print the candidate if it is Majority\*/

if (isMajority(a, size, cand))

cout << " " << cand << " ";

else

cout << "No Majority Element";

}

/\* Driver code \*/

int main()

{

int a[] = { 1, 3, 3, 1, 2 };

int size = (sizeof(a)) / sizeof(a[0]);

// Function calling

printMajority(a, size);

return 0;

}

**Output**

No Majority Element

**Time Complexity:** O(n), As two traversal of the array, is needed, so the time complexity is linear.  
**Auxiliary Space:** O(1), As no extra space is required.

Minimum Consecutive Flips

Given a binary array, we need to convert this array into an array that either contains all 1s or all 0s.  We need to do it using the minimum number of group flips.

Examples :

***Input****: arr[] = {1, 1, 0, 0, 0, 1}*  
***Output****:  From 2 to 4*  
***Explanation****: We have two choices, we make all 0s or do all 1s.  We need to do two group flips to make all elements 0 and one group flip to make all elements 1.  Since making all elements 1 takes least group flips, we do this.*  
***Input****: arr[] = {1, 0, 0, 0, 1, 0, 0, 1, 0, 1}*  
***Output****:*  
*From 1 to 3*  
*From 5 to 6*  
*From 8 to 8*  
***Input****: arr[] = {0, 0, 0}*  
***Output****:*  
***Explanation****: Output is empty, we need not to make any change*  
***Input****: arr[] = {1, 1, 1}*  
***Output****:*  
***Explanation****: Output is empty, we need not to make any change*  
***Input****: arr[] = {0, 1}*  
***Output****:*  
*From 0 to 0*  
***OR***  
*From 1 to 1*  
***Explanation****:  Here number of flips are same either we make all elements as 1 or all elements as 0.*

A **Naive Solution**is to traverse do two traversals of the array. We first traverse to find the number of groups of 0s and the number of groups of 1.  We find the minimum of these two.  Then we traverse the array and flip the 1s if groups of 1s are less. Otherwise, we flip 0s.

**How to do it with one traversal of array?**

An **Efficient Solution**is based on the below facts :

* There are only two types of groups (groups of 0s and groups of 1s)
* Either the counts of both groups are same or the difference between counts is at most 1. For example, in {1, 1, 0, 1, 0, 0} there are two groups of 0s and two groups of 1s.  In example, {1, 1, 0, 0, 0, 1, 0, 0, 1, 1}, count of groups of 1 is one more than the counts of 0s.

Based on the above facts, we can conclude that if we always flip the second group and other groups that of the same type as the second group, we always get the correct answer.  In the first case, when group counts are the same, it does not matter which group type we flip as both will lead to the correct answer.  In the second case, when there is one extra, by ignoring the first group and starting from the second group, we convert this case to first case (for subarray beginning from the second group) and get the correct answer.

C++Java

// C++ program to find the minimum

// group flips in a binary array

#include <iostream>

using namespace std;

void printGroups(bool arr[], int n)

{

// Traverse through all array elements

// starting from the second element

for (int i = 1; i < n; i++) {

// If current element is not same

// as previous

if (arr[i] != arr[i - 1]) {

// If it is same as first element

// then it is starting of the interval

// to be flipped.

if (arr[i] != arr[0])

cout << "From " << i << " to ";

// If it is not same as previous

// and same as first element, then

// previous element is end of interval

else

cout << (i - 1) << endl;

}

}

// Explicitly handling the end of

// last interval

if (arr[n - 1] != arr[0])

cout << (n - 1) << endl;

}

int main()

{

bool arr[] = { 0, 1, 1, 0, 0, 0, 1, 1 };

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

printGroups(arr, n);

return 0;

}

**Output**

From 1 to 2

From 6 to 7

**Time Complexity:**O(n)  
**Auxiliary Space:** O(1)

Prefix Sum Technique

Given an array arr[] of size n, its prefix sum array is another array prefixSum[] of the same size, such that the value of prefixSum[i] is arr[0] + arr[1] + arr[2] … arr[i].

**Examples :**

***Input  :****arr[] = {10, 20, 10, 5, 15}*  
***Output :****prefixSum[] = {10, 30, 40, 45, 60}*  
***Explanation :****While traversing the array, update the element by adding it with its previous element.*  
*prefixSum[0] = 10,*  
*prefixSum[1] = prefixSum[0] + arr[1] = 30,*  
*prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.*

To fill the prefix sum array, we run through index 1 to last and keep on adding the present element with the previous value in the prefix sum array.  
Below is the implementation :

**Implementation:**

PythonC++Java

# Python Program for Implementing

# prefix sum array

# Fills prefix sum array

def fillPrefixSum(arr, n, prefixSum):

prefixSum[0] = arr[0]

# Adding present element

# with previous element

for i in range(1, n):

prefixSum[i] = prefixSum[i - 1] + arr[i]

# Driver code

arr =[10, 4, 16, 20 ]

n = len(arr)

prefixSum = [0 for i in range(n + 1)]

fillPrefixSum(arr, n, prefixSum)

for i in range(n):

print(prefixSum[i], " ", end ="")

**Output**

10 14 30 50

**Time Complexity:** O(n)  
**Auxiliary Space:** O(n)

Given an array arr[] of size n. Given Q queries and in each query given L and R, print sum of array elements from index L to R.

**Implementation:**

PythonC++Java

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

n = 6;

a = [ 3, 6, 2, 8, 9, 2 ];

pf = [0 for i in range(n+2)];

for i in range(n):

pf[i + 1] = pf[i] + a[i];

q =[ [ 2, 3 ],[ 4, 6 ],[ 1, 5 ],[ 3, 6 ]];

for i in range(4):

l = q[i][0];

r = q[i][1];

# Calculating sum from r to l.

print(pf[r] - pf[l - 1] );

**Output**

8

19

28

21

**Time Complexity:**O(n)  
**Auxiliary Space:**O(n)

Sliding Window Technique

This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.  
  
Let’s start with a problem for illustration where we can apply this technique:

Given an array of integers of size 'n'.

Our aim is to calculate the maximum sum of 'k'

consecutive elements in the array.

Input : arr[] = {100, 200, 300, 400}

k = 2

Output : 700

Input : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}

k = 4

Output : 39

We get maximum sum by adding subarray {4, 2, 10, 23}

of size 4.

Input : arr[] = {2, 3}

k = 3

Output : Invalid

There is no subarray of size 3 as size of whole

array is 2.

The **Naive Approach** to solve this problem is to calculate sum for each of the blocks of K consecutive elements and compare which block has the maximum sum possible. The time complexity of this approach will be O(n \* k).

**Window Sliding Technique**

The above problem can be solved in Linear Time Complexity by using Window Sliding Technique by avoiding the overhead of calculating sum repeatedly for each block of k elements.  
  
The technique can be best understood with the window pane in bus, consider a window of length **n** and the pane which is fixed in it of length **k**. Consider, initially the pane is at extreme left i.e., at 0 units from the left. Now, co-relate the window with array arr[] of size n and plane with current\_sum of size k elements. Now, if we apply force on the window such that it moves a unit distance ahead. The pane will cover next **k** consecutive elements.   
  
Consider an array **arr[]** = {5 , 2 , -1 , 0 , 3} and value of **k** = 3 and **n** = 5  
  
**Applying sliding window technique**:

1. We compute the sum of first k elements out of n terms using a linear loop and store the sum in variable window\_sum.
2. Then we will graze linearly over the array till it reaches the end and simultaneously keep track of maximum sum.
3. To get the current sum of block of k elements just subtract the first element from the previous block and add the last element of the current block .

The below representation will make it clear how the window slides over the array.

This is the initial phase where we have calculated the initial window sum starting from index 0 . At this stage the window sum is 6. Now, we set the maximum\_sum as current\_window i.e 6.

Now, we slide our window by a unit index. Therefore, now it discards 5 from the window and adds 0 to the window. Hence, we will get our new window sum by subtracting 5 and then adding 0 to it. So, our window sum now becomes 1. Now, we will compare this window sum with the maximum\_sum. As it is smaller we wont the change the maximum\_sum.

Similarly, now once again we slide our window by a unit index and obtain the new window sum to be 2. Again we check if this current window sum is greater than the maximum\_sum till now. Once, again it is smaller so we don't change the maximum\_sum.

Therefore, for the above array our maximum\_sum is 6.

**C++ code for sliding window Algorithm: -**

#include <bits/stdc++.h>

using namespace std;

// Returns maximum sum in a subarray of size k.

int maxSum(int arr[], int n, int k) {

// n must be greater

if (n < k) {

cout << "Invalid";

return -1;

}

//sum of first window of size k

int window\_sum = 0;

for (int i = 0; i < k; i++)

window\_sum += arr[i];

// Compute sums of remaining windows by

// removing first element of previous

// window and adding last element of

// current window.

int max\_sum = window\_sum;

for (int i = k; i < n; i++) {

window\_sum += (arr[i] - arr[i - k]);

max\_sum = max(max\_sum, window\_sum);

}

return max\_sum;

}

int main(){

int n = 5 , k = 3;;

int arr[] = { **16,12,9,19,11,8**};

cout << maxSum(arr, n, k);

return 0;

}

Maximum Appearing Element

Given two arrays **L[]** and **R[]** of size **N** where L[i] and R[i] (**0 ≤ L[i], R[i] < 106**)denotes a range of numbers, the task is to find the maximum occurred integer in all the ranges. If more than one such integer exists, print the smallest one.

**Examples:**

***Input:****L[] = {1, 4, 3, 1}, R[] = {15, 8, 5, 4}*  
***Output:****4*

***Input:****L[] = {1, 5, 9, 13, 21}, R[] = {15, 8, 12, 20, 30}*  
***Output:****5*  
***Explanation:****Numbers having maximum occurrence i.e 2 are*  
*5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. The smallest number among all are 5.*

**Naive Approach:**

*Traverse through all the ranges. Then for every range, count frequencies, make a hash table or hash map to store every item. Then traverse through other ranges and increment the frequency of every item. The item with the highest frequency is our answer.*

**Time Complexity:**O(N\*M). Here **N**is the number of ranges and **M**is themaximum number of elements in any of the ranges.  
**Auxiliary Space:**O(M). For Hash table.

**Maximum occurred integer in n ranges using Difference array technique.**

Below is the idea to solve the problem:

*The idea is to use the Difference array technique. Create a vector initialized with value zero. Iterate through****every range****and mark the presence of the beginning of every range by incrementing the****start****of the range with one i.e.****arr[L[i]]++****and mark the****end****of the range by decrementing at index one greater than the end of range by one i.e.****arr[R[i]+1]–****.*

*Now when computing the prefix sum, Since the beginning is marked with one, all the values after beginning will be incremented by one. Now as increment is only targeted only till the end of the range, the decrement on index****R[i]+1****prevents that for every range****i****.*

**Illustration:**

*L[] = {1, 2, 3} , R[] = {3, 5 , 7}*

*1. For beginning of range arr[L[i]]++ the array becomes {0,1,1,1,0,0,0,0,……}*

*2. For end of range arr[R[i]+1]– the array becomes  {0,1,1,1,-1, 0, -1, 0,-1,……}*

*3. After prefix sum the array becomes {0,1,2,3,2,2,1,1,0…}*

*Do prefix sum, the sum of elements after (1) is incremented by one because beginning was marked. Now elements after (3) must not be incremented so if there’s a range one, two, three, the values from one, two, three should only be incremented by one or their frequency should be incremented by one.*

*That is why decreasing the value of arr[R[i]+1] is needed so that elements after the end of this range have minus one subtracted to values. That is how to nullify the impact of incrementing the value when prefix will be taken.*

*So when taking the prefix, simply decrement every value after the range ends, since I want to increment elements only in the range. That’s the idea of this algorithm.*

Follow the below steps to Implement the idea:

* Initialize a Hash array **arr[]** to store the occurrence of every element in all the ranges combined.
* Iterate over all the **N**ranges and increment **L[i]**by one and decrement **R[i]**by one.
* Run a Loop from 1 to the maximum end value of all the ranges and take the Prefix sum.

Below is the Implementation of the above approach:

C++Java

// C++ program to find maximum occurred element in

// given N ranges.

#include <bits/stdc++.h>

#define MAX 1000000

using namespace std;

// Return the maximum occurred element in all ranges.

int maximumOccurredElement(int L[], int R[], int n)

{

// Initialising all element of array to 0.

int arr[MAX];

memset(arr, 0, sizeof arr);

// Adding +1 at Li index and subtracting 1

// at Ri index.

int maxi = -1;

for (int i = 0; i < n; i++) {

arr[L[i]] += 1;

arr[R[i] + 1] -= 1;

if (R[i] > maxi) {

maxi = R[i];

}

}

// Finding prefix sum and index having maximum

// prefix sum.

int msum = arr[0], ind;

for (int i = 1; i < maxi + 1; i++) {

arr[i] += arr[i - 1];

if (msum < arr[i]) {

msum = arr[i];

ind = i;

}

}

return ind;

}

// Driven code

int main()

{

int L[] = { 1, 4, 9, 13, 21 };

int R[] = { 15, 8, 12, 20, 30 };

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

cout << maximumOccurredElement(L, R, n) << endl;

return 0;

}

**Output**

4

**Time Complexity:** O(N + MAX)   
**Auxiliary space**: O(MAX)