**Arrays**

**What is Array?**

* An array is a collection of items stored at contiguous memory locations.
* In Arrays multiple items of same type can be stored together which makes it easier to calculate the position of each element by simply adding an offset to a base value.

**Defining An Array**

* The size of the array is fixed and the memory for an array needs to be allocated before use, the size of an array cannot be increased or decreased dynamically.

**General Declaration**

dataType arrayName[arraySize];

**Accessing array elements**

* Indexing of an array starts from 0 to n-1;
* Element present at ith index in the array arr[] can be accessed as arr[i].

**Operations:-**

1. ***Searching***

* We can use loops to perform the above operation of array traversal and access the elements, using indexes

Eg :-.For searching the element stored in key in array of size **n**

int a[n]; // Array Declaration

for(i=0 ;i<n;i++) { //Traversing An Array Through Loop

if(arr[i]==key) {

print "ELEMENT FOUND";

}

else{

print "ELEMENT NOT FOUND";

} }

**Time Complexity :** - **O(n)** --- as worst Case if required element is at last index

**O(1)**  --- as best case if required element is at first position

2. ***Insertion***

1. **Insert elements at the end of the array.**
   * Suppose there is an array ‘arr’ of size ‘N’ and length of an array is ‘len’.
   * To insert an element k at the end of the arr[]

The first step is to check if there is any space left in the array for new element.

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 :** - **O(1)** as we are directly inserting the element in a single operation.

1. Insert element at any given index in the array.
   * Suppose we have to insert element ‘k’ at index ‘idx’.
   * For this opearation we have to check  if there is any space left in the array for new element. To do this check following ,

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**.
  + Before inserting the elements at idx shift all elements from the index idx till end of the array to the right by 1 place.

Implementation

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

{

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

}

* + After Shifting the elements place k at index idx.

arr[idx] = K;

**Time Complexity : - O(N)** as in worst case we might have to shift all of the elements by one place to the right.

1. ***Deletion :-***
   * 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.
   * For 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++)

{

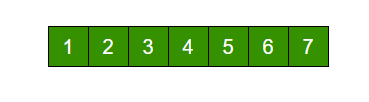
arr[i-1] = arr[i];

}

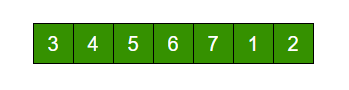
len = len-1;

**Time Complexity : - O(N)**  in worst case as we might have to shift all of the elements by one place to the left.

1. ***Array Rotation***
   * Consider the following array :-



* + The above array is rotated counter-clockwise(towards left) by 2 elements. After rotation, the array will be:



* + The process looks like following :-

1. Shift all elements after K-th element to the left by K positions.
2. Fill the K blank spaces at the end of the array by first K elements from the original array.
   * The similar approach can also be applied for clockwise array rotation.
3. Implementations
   1. Simple Method
4. Store the first K elements in a temporary array say temp[].
5. Shift all elements after K-th element to the left by K positions in the original array.
6. Fill the K blank spaces at the end of the original array by the K elements from the temp array.

//Algorithm

Say, arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2

1) Store first K elements in a temp array

temp[] = [1, 2]

2) Shift rest of the arr[]

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

3) Store back the K elements from temp

arr[] = [3, 4, 5, 6, 7, 1, 2]

**Time Complexity : - O(N)** as we have to shift all N elements

**Auxiliary Space : -** **O(K)** where k is the number of places by which elements will be rotated.

* 1. Another Method (Without Extra Space)
  + We can also rotate an array by avoiding the use of temporary array. The idea is to rotate the array one by one K times.

1. Store the first element in a temporary variable say temp.
2. Left shift all elements after the first element by 1 position. That is, move arr[1] to arr[0], arr[2] to arr[1] and so on.
3. Initialize arr[N-1] with temp.
   * **To rotate an array by K position to the left, repeat the above process K times.**

**Eg :-**

arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2

Rotate arr[] one by one 2 times.

After 1st rotation: [2, 3, 4, 5, 6, 7, 1]

After 2nd rotation: [ 3, 4, 5, 6, 7, 1, 2]

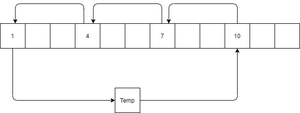
**Time Complexity :- O(N\*K)** where N is the number of elements in the array and K is the number of places by which elements will be rotated.

**Auxiliary Space : -** **O(1)** as one storage space is used.

* 1. **Juggling Algorithm : -**
  + Here , Instead of moving one by one , divide the array in different sets where number of sets is equal to GCD of N and K and move the elements within sets.
  + If GCD is 1 as is for the above example array (N = 7 and K = 2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.

**Eg :-**

Here is an example for N = 12 and K = 3. GCD of N and K is 3:

Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}  
  
a) Elements are first moved in first set – (See below   
 diagram for this movement)  
  
  
  
 arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}  
  
b) Then in second set.

arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}  
  
c) Finally in third set.

arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}

***Time Complexity:-***  **O(N),** as worst case because we have shift all elements of the array.

***Auxiliary Space:-***  **O(1) ,** here as one temporary variable is needed.

* **Reversing An Array : -**
  + Reversing an array means reversing the order of the elements in the given array.
  + Suppose you have to reverse the following array : -

Iterative Solution

* + - 1. **Using Temporary Array : -**
         * The idea is to first copy all of the elements of the given array in a temporary array. Then traverse the temporary array from end and replace elements in original array by elements of temp array.
         * **Auxiliary Space : -** **O(N)**
      2. **Efficient One :-**
         * The idea is to traverse the array from both ends and keep swapping elements from both ends until middle of the array is reached.
         * Algorithm

//

1) Initialize start and end indexes as

start = 0, end = N-1

2) In a loop, swap arr[start] with arr[end]

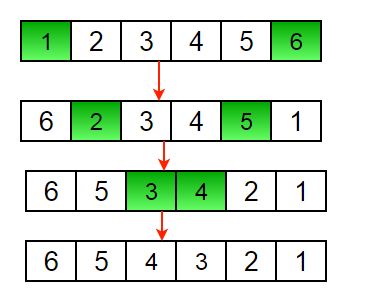
and change start and end as follows :

start = start + 1,

end = end – 1

//

For Reference : -



* + - * + **Time Complexity : -**  **O(N) ,** as we have to change all the elements.

**Recursive Solution**

Algorithm : -

1) Initialize start and end indexes as   
 start = 0, end = n-1  
2) Swap arr[start] with arr[end]  
3) Recursively call reverse for rest of the array.

Recursive Function : -

void reverseArray(arr[], start, end)

{

if (start >= end)

return;

// Swap elements at start and end

temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

// Recursive Function calling

reverseArray(arr, start + 1, end - 1);

}

**Time Complexity : - O(N) ,**  as we have to shift all elements .

* **Sliding Window Technique :-**

This technique shows how a nested ‘for loop’ in few problems can be converted to a single ‘for loop; and hence reducing **time complexity**.

For Eg : -

Q. **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

**Naïve Approach : -**

* + We have to find the sum of each k consecutive elements and compare which block has the maximum sum possible .

**Time Complexity : - O(n\*k)**

**Sliding Window Technique**

* + To understand this topic lets take an example of window pane in bus .
  + Consider of window of length n and the pane which is fixed in it of length k;
  + 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 .

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.  
  


#include <iostream>

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;

    }

    // Compute sum of first window of size k

    int max\_sum = 0;

    for (int i = 0; i < k; i++)

        max\_sum += arr[i];

    // Compute sums of remaining windows by

    // removing first element of previous

    // window and adding last element of

    // current window.

    int window\_sum = max\_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;

}

// Driver code

int main()

{

    int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };

    int k = 4;

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

    cout << maxSum(arr, n, k);

    return 0;

}

**Time Complexity : - O(n)**

* **Prefix Sum Array**
  + The prefix sum array of any array, arr[] is defined as an array of same size say, prefixSum[] such that the value at any index **i** in prefixSum[] is sum of all elements from indexes **0 to i** in arr[].

**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.

**Function To generate prefix sum array :-**

void fillPrefixSum(int arr[], int N, int prefixSum[])

{

prefixSum[0] = arr[0];

// Adding present element

// with previous element

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

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

}

Finding Sum in Range : -

Since the array prefixSum[i] stores the sum of all elements upto **i**. Therefore, **prefixSum[j] - prefixSum[i]** will give:

sum of elements upto j-th index - sum of elements upto i-th element

sumInRange = prefixSum[j] , if i = 0  
  
otherwise,  
  
sumInRange = prefixSum[j] - prefixSum[i-1] , if (i != 0)