TOPIC - ARRAYS

Introduction

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

It falls under the category of linear data structures. The elements are stored at contiguous memory locations in an array. Most of the algorithms and problem-solving approaches use arrays in their implementation.

Arrays have a fixed size where the size of the array is defined when the array is initialized.

- Element—Each item stored in an array is called an element.
- Index—Each memory location of an element in an array is identified by a numerical index.

In the above image, [1, 2, 5, 3, 4, 8] are the elements of the array and 0, 1, 2, 3, 4, and 5 are the indexes of the elements of the array.

As arrays store elements in contiguous memory locations. This means that any element can be accessed by adding an offset to the base value of the array or the location of the first element. We can access elements as A[i] where "i" is the index of that element.

Array operations

- **Insertion:** To insert an element item at any position pos, we will shift the array elements from this position to one position forward and will do this for all the elements next to it.

```
// A is an array
// n is the number of elements in the array
// maxSize is the size of the array
// item is the element to insert
// pos is position at which element will be inserted
void insert(int A[], int n, int maxSize, int item, int pos)
{
   n += 1 // increasing the number of elements
   for(i = n; i >= pos; i--)
   {
      A[i] = A[i-1] // shift elements forward
   }
   A[pos-1] = item // insert item at pos
}
```

❖ **Deletion:** To delete an element at any position pos, we will shift the array elements from this position to one position forward and will do this for all elements next to it.

```
void delete(int A[], int n, int pos)
{
   for( i = pos-1 to n-2 )
   {
      A[i] = A[i+1] // shift element backwards
   }
   n -= 1 // decreasing size by 1
}
```

❖ **Search:** To search an element in the array, just traverse through the array and check at each position.

```
bool search(int A[], int n, int item)
{
   for( i = 0 to n-1 )
   {
```

```
        if( A[i] == item )
            return true
    }
    return false
}
```

- **Updation:** To update the value at any index just set the value

```
void update(int A[], int n, int idx, int val)
{
    A[idx] = val // update the value at that index
}
```

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. The most commonly used class for implementing arrays is the **vector** class.

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 vectors provide.

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 as the number of elements.
- **empty()**: Returns whether the container is empty.
- **push_back()**: It pushes 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 the specified position or range.
- **clear()** – It is used to remove all the elements of the vector container.

Look at this example to understand the array implementation using vector class and functions described above.

Array Rotation

https://www.geeksforgeeks.org/array-rotation/

Practice Problem:
https://leetcode.com/problems/rotate-array/

Reversing an Array

Arrays can be reversed in 3 ways.

1. Just print the elements in reverse order. Time complexity of this approach is O(n) and space complexity is O(1).

   The next two approaches are useful when it is required to store the elements in reverse order:-

2. Store the elements of the array in another array in reverse order. The time complexity of this approach is O(n) and space complexity is O(n) for the extra array.

3. Without using extra memory, an array can be reversed in O(n) runtime complexity.This is done by swapping the elements at the same position relative to the start and end of the array as it is traversed.

The programs to implement this can be found here:-
https://www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/

Prefix Sum Array

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

prefixSum[i] = arr[0] + arr[1] + arr[2] + ... + arr[i], for all **0 <= i <= N**

Example:
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.

Below function generates a prefix sum array for a given array arr[] of size N:

```
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];
}
```

Applications:

- Range sum queries without updates
- Equilibrium index of an array
- Find if there is a subarray with 0 sum
- Maximum subarray size, such that all subarrays of that size have sum less than k

Window Sliding Technique

This technique shows how a nested for loop in some problems can be converted to a single for loop to reduce the time complexity.

Example:
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[] = {5,2,-1,0, 3}, k = 2, n=5
Output : 6

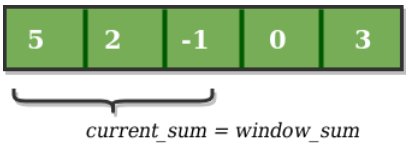
Naive Approach: Calculate sum for each of the blocks of k consecutive elements and compare which block has the maximum sum possible.The time complexity of this approach will be O(n*k).

Window Sliding Technique:

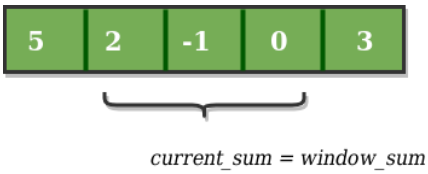
1. We compute the sum of first k elements out of n terms using a linear loop and store the sum in variable window_sum.
2. Then we will graze linearly over the array till it reaches the end and simultaneously keep track of maximum sum.
3. To get the current sum of blocks of k elements just subtract the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

Time Complexity of this approach will be O(n).

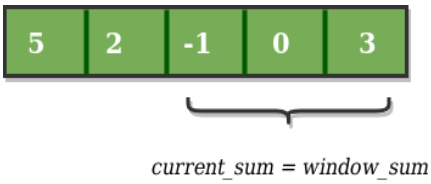
Initially, start from index 0, at this point window_sum = 6 and we set it as the maximum_sum=6.



Now, we slide our window by a unit index. So, our window_sum is 1 now and maximum_sum will remain 6 as current window_sum is smaller.



Similarly, now once again we slide our window by a unit index. We obtained a window sum as 2 which is smaller so we don't change the maximum_sum. Therefore, for the above array our maximum_sum is 6.



Applications:

- Count number of substrings having at least K distinct characters
- Check if a string contains an anagram of another string as its substring

Maximum Subarray Sum

The **maximum subarray problem** is the task of finding the largest possible sum of a contiguous subarray, within a given one-dimensional array A[1...n] of numbers.

-2	1	-3	4	-1	2	1	-5	4

The subarray marked yellow is the max sum subarray in the above example.

Brute Force Approach:

Every single subarray is checked. This takes $O(n^2)$ time as a total of n*(n-1)/2 subarrays exist if n is the size of the array which is too slow.

Kadane's Algorithm:

This algorithm uses optimal substructures i.e. the maximum subarray ending at each position is calculated in a simple way from a related but smaller and overlapping subproblem: the maximum subarray ending at the previous position.

Below is a very much self-explanatory implementation (in C++) of a function which takes an array as an argument and returns the sum of the maximum subarray.

The simple idea of Kadane's algorithm is to look for all positive contiguous segments of the array (local_max is used for this). And keep track of the maximum sum contiguous segment among all positive segments (global_max is used for this). Each time we get a positive-sum, compare it with global_max and update global_max if it is greater than global_max.

```
int maxSumSubArray(vector<int> &A)
{
    int n = A.size(); // Size of the array
    int local_max = 0;
    int global_max = INT_MIN; // -Infinity

    for (int i = 0; i < n; i++)
    {
        local_max = max(A[i], A[i] + local_max);
        if (local_max > global_max)
        {
           global_max = local_max;
        }
    }

    return global_max;
}
```

Since we traverse the array only once and perform a constant number of operations in each iteration, the runtime complexity of Kadane's algorithm is O(n) which is a huge improvement over the brute-force approach.

Additional Resources:

https://www.geeksforgeeks.org/array-data-structure/
https://www.faceprep.in/c/array-programs/
https://www.guru99.com/array-data-structure.html