Arrays

An array is one of the simplest and most fundamental data structures in programming. It stores a fixed-size collection of elements of the same type (e.g., integers, strings) in contiguous memory locations.

Properties

- Same Type of Data: Every item in an array must be of the same type (like all numbers or all strings).
- Access by Position: You get values using their position number, starting from 0.
  Example: array[2] gives you the third item.
- Stored Together in Memory: The items are kept side-by-side in memory, which makes them quick to access.
- Harder to Insert or Delete in the middle: If you want to add or remove something in the middle, the other items have to shift around, which takes more time.

Array Operations

- Reading

  To read an array value, we use the array name with the index of the value we want to read in brackets, like this myFruits[0].

  We must also use a command to write myFruits[0] to the console/terminal, so that we can actually see the result, and that is done a little different depending on the programming language.

  - Python:

    ```
    myFruits = ['banana','apple','orange']

    print(myFruits[0])
    ```

  - C++:

    ```
    string myFruits[] = {"banana","apple","orange"};

    cout << myFruits[0];
    ```

  - Javascript:

    ```
    const myFruits = ['banana','apple','orange'];
    ```

```
console.log(myFruits[0]);
```

- Updating

To update a value in an array, we use the array name with the index position of the value we want to update, like this myFruits[0], and then we use the equal sign = to store a new value there.

- Python:

```
myFruits = ['banana','apple','orange']

myFruits[0] = 'kiwi'
```

- C++:

```
string myFruits[] = {"banana","apple","orange"};

myFruits[0] = "kiwi";
```

- Javascript:

```
const myFruits = ['banana','apple','orange'];

myFruits[0] = 'kiwi';
```

- Appending

To insert a value into an array, in addition to the existing values, we need:

-the array name

-a command to do the insert operation

-the value to be inserted

When we do insert this way, the new value is inserted at the end of the array.

- Python:

```
myFruits = ['banana','apple','orange']

myFruits.append('kiwi')
```

- C++:

```
vector<string> myFruits = {"banana", "apple", "orange"};

myFruits.push_back("kiwi");
```

- Javascript:

```
const myFruits = ['banana','apple','orange'];

myFruits.push('kiwi');
```

- Insertion

A value can also be added to a specific position in an array, using the index, like this:

- Python

```
myFruits = ['banana','apple','orange']

myFruits.insert(1,'kiwi')
```

- C++

```
vector<string> myFruits = {"banana", "apple", "orange"};

myFruits.insert(myFruits.begin() + 1, "kiwi");
```

- Javascript

```
const myFruits = ['banana','apple','orange'];

myFruits.splice(1,0,'kiwi');
```

- Deletion

An array value is removed by specifying the index where the value should be removed from.

- Python:

```
myFruits = ['banana','apple','orange']

myFruits.pop(1)
```

- C++:

```
vector<string> myFruits = {"banana", "apple", "orange"};

myFruits.erase(myFruits.begin() + 1);
```

- Javascript:

```
const myFruits = ['banana','apple','orange'];
```

```
myFruits.splice(1,1);
```

- Length
  - Python:

    ```
    myFruits = ['banana','apple','orange']
    print(len(myFruits))
    ```

  - C++:

    ```
    vector<string> myFruits = {"banana", "apple", "orange"};
    cout << myFruits.size();
    ```

  - Javascript:

    ```
    const myFruits = ['banana','apple','orange'];
    console.log(myFruits.length);
    ```

- Traversal

  This is how to look at every value in the array:

  - Python:

    ```
    myFruits = ['banana','apple','orange']
    for fruit in myFruits:
      print(fruit)
    ```

  - C++:

    ```
    string myFruits[] = {"banana", "apple", "orange"};
    for (auto fruit : myFruits) {
      cout << fruit + "\n";
    }
    ```

  - Javascript:

    ```
    const myFruits = ['banana','apple','orange'];
    for (let fruit of myFruits) {
      console.log(fruit);
    ```

Algorithms

Here are a few algorithms which are useful in solving questions related to arrays:

Linear Search

Linear Search is the simplest way to locate an element in an array. We iterate through all elements one by one, comparing each element with the target. If we find a match, we return its index. In the best case, the element is found at the very beginning, resulting in a time complexity of **O(1)**. In the worst case, we may need to check every element in the array, giving a time complexity of **O(n)**. Since it doesn't require the array to be sorted, it's a good choice for small or unsorted datasets.

It can be done in the following way:

```
int linearSearch(int arr[], int size, int target) {

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

        if (arr[i] == target) {

            return i;

        }

    }

    return -1;

}
```

Two pointer method

In the two pointers method, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently. Next we discuss a problem that can be solved using the two pointers method.

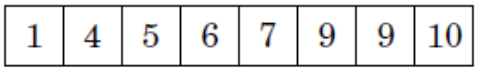
2sum problem:

Another problem that can be solved using the two pointers method is the following problem, also known as the 2SUM problem: given an array of n numbers and a target sum x, find two array values such that their sum is x, or report that no such values exist.

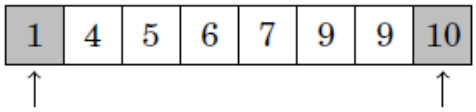
To solve the problem, we first sort the array values in increasing order. After that, we iterate through the array using two pointers. The left pointer starts at the first value and moves one step

to the right on each turn. The right pointer begins at the last value and always moves to the left until the sum of the left and right value is at most x. If the sum is exactly x, a solution has been found.

For example, consider the following array and a target sum x = 12:

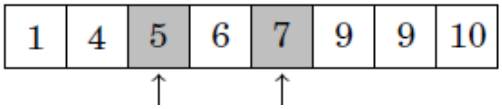


The initial positions of the pointers are as follows. The sum of the values is 1 + 10 = 11 that is smaller than x.



Then the left pointer moves one step to the right. The right pointer moves three steps to the left, and the sum becomes 4+7 = 11.

After this, the left pointer moves one step to the right again. The right pointer does not move, and a solution 5+7 = 12 has been found.



This can be implemented in C++ in the following way:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to check whether any pair exists
// whose sum is equal to the given target value
```

```
bool twoSum(vector<int> &arr, int target){

    // Sort the array
    sort(arr.begin(), arr.end());

    int left = 0, right = arr.size() - 1;

    // Iterate while left pointer is less than right
    while (left < right){
        int sum = arr[left] + arr[right];

        // Check if the sum matches the target
        if (sum == target)
            return true;
        else if (sum < target)
            left++; // Move left pointer to the right
        else
            right--; // Move right pointer to the left
    }
    // If no pair is found
    return false;
}

int main(){
    vector<int> arr = {0, -1, 2, -3, 1};
    int target = -2;

    // Call the twoSum function and print the result
    if (twoSum(arr, target))
        cout << "true";
    else
        cout << "false";
```

```
    return 0;
}
```

The running time of the algorithm is O(nlogn), because it first sorts the array in O(nlogn) time, and then both pointers move O(n) steps.

Note that it is possible to solve the problem in another way in O(nlogn) time using binary search. In such a solution, we iterate through the array and for each array value, we try to find another value that yields the sum x. This can be done by performing n binary searches, each of which takes O(logn) time.

Binary Search

We have seen in linear search that in the worst case, the code runs in O(n) time. However, if the array is sorted, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following binary search algorithm efficiently searches for an element in a sorted array in O(logn) time.

The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region. At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element.

The above idea can be implemented as follows:

```
int a = 0, b = n-1;
while (a <= b) {
int k = (a+b)/2;
if (array[k] == x) {
// x found at index k
}
if (array[k] > x) b = k-1;
else a = k+1;
}
```

In this implementation, the active region is a...b, and initially the region is 0...n¡1. The algorithm halves the size of the region at each step, so the time complexity is O(logn).

To learn more:

https://www.geeksforgeeks.org/array-data-structure-guide/

https://www.geeksforgeeks.org/quizzes/dsa-tutorial-array/

Two Pointers Technique | GeeksforGeeks

Binary Search Algorithm – Iterative and Recursive Implementation | GeeksforGeeks

Problems

Easy:
Pascal's Triangle - LeetCode | Solution

Best Time to Buy and Sell Stock - LeetCode | Solution

Single Number - LeetCode | Solution

Merge Sorted Array - LeetCode | Solution

Remove Duplicates from Sorted Array - LeetCode | Solution

Majority Element - LeetCode | Solution

Summary Ranges - LeetCode | Solution

Medium:

Majority Element II - LeetCode | Solution

4Sum - LeetCode | Solution

Subarray Sum Equals K - LeetCode | Solution

Search in Rotated Sorted Array - LeetCode | Solution

Sort Colors - LeetCode | Solution

Hard:
Median of Two Sorted Arrays - LeetCode | Solution

Split Array Largest Sum - LeetCode | Solution

Find K-th Smallest Pair Distance - LeetCode | Solution