**Array Data Structure**

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



**Sorting with Arrays:**

Sorting is a technique which helps us to sort the elements of an array in ascending or descending order**.**

**Sorting in Ascending order:**

Java's [util.Arrays.sort](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-) method provides us with a quick and simple way to sort an array of primitives or objects that implement the Comparable interface in ascending order. To sort elements in ascending order,we pass our array to sort method.

**Example:**

numbers = {1,5,2,8,10}

Arrays.sort(numbers);

**Program:**

importjava.util.Arrays;

publicclassSortExample

{

    publicstaticvoidmain(String[] args)

    {

        // Our arr contains 8 elements

        int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

        Arrays.sort(arr);

        System.out.printf("Modified arr[] : %s",

                          Arrays.toString(arr));

    }

}

**O/P:**

Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]

**Sorting in Descending Order:**

Sorting a primitive array in descending order is not quite as simple as sorting it in ascending order because Java doesn't support the use of Comparators on primitive types. To overcome this shortfall we have a few options.

First, we could sort our array in ascending order and then do an in-place reversal of the array.

Second, could convert our array to a list, use Guava's [Lists.reverse()](https://www.baeldung.com/guava-lists) method and then convert our list back into an array.

Finally, we could transform our array to a Stream and then map it back to an int array.

**Example**

importjava.util.Arrays;

importjava.util.Collections;

publicclassSortExample

{

    publicstaticvoidmain(String[] args)

    {

        // Note that we have Integer here instead of

        // int[] as Collections.reverseOrder doesn't

        // work for primitive types.

        Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sorts arr[] in descending order

        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : %s",

                          Arrays.toString(arr));

    }

}

**O/P:**

Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]

**There are different types of sorting algorithms.They are:**

* **Bubble Sort**
* **Insertion sort**
* **Selection Sort**
* **Merge Sort**
* **Heap Sort**
* **Quick sort**

**Bubble Sort:**

Bubble sort works by swapping adjacent elements if they're not in the desired order. This process repeats from the beginning of the array until all elements are in order.

We know that all elements are in order when we manage to do the whole iteration without swapping at all - then all elements we compared were in the desired order with their adjacent elements, and by extension, the whole array.The reason this algorithm is called Bubble Sort is because the numbers kind of "bubble up" to the "surface." All numbers move to their respective places bit by bit, left to right, like bubbles slowly rising from a body of water.

**Implementation:**

*public static void bubbleSort(int[] a) {  
boolean sorted = false;  
 int temp;  
 while(!sorted) {  
 sorted = true;  
 for (int i = 0; i<array.length - 1; i++) {  
 if (a[i] > a[i+1]) {  
 temp = a[i];  
 a[i] = a[i+1];  
 a[i+1] = temp;  
 sorted = false;  
 }  
 }  
 }  
}*

**Insertion Sort:**

The idea behind Insertion Sort is dividing the array into the sorted and unsorted subarrays.

The sorted part is of length 1 at the beginning and is corresponding to the first (left-most) element in the array. We iterate through the array and during each iteration, we expand the sorted portion of the array by one element.

Upon expanding, we place the new element into its proper place within the sorted subarray. We do this by shifting all of the elements to the right until we encounter the first element we don't have to shift.

**Implementation**

*public static void insertionSort(int[] array) {  
 for (int i = 1; i<array.length; i++) {  
 int current = array[i];  
 int j = i - 1;  
 while(j >= 0 && current < array[j]) {  
 array[j+1] = array[j];  
 j--;  
 }  
 // at this point we've exited, so j is either -1  
 // or it's at the first element where current >= a[j]  
 array[j+1] = current;  
 }  
}*

**Selection Sort:**

Selection Sort also divides the array into a sorted and unsorted subarray. Though, this time, the sorted subarray is formed by inserting the minimum element of the unsorted subarray at the end of the sorted array, by swapping.

**Implementation:**

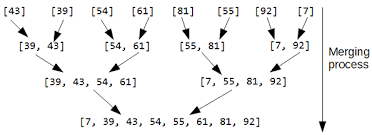
public static void selectionSort(int[] array) {  
 for (int i = 0; i<array.length; i++) {  
 int min = array[i];  
 int minId = i;  
 for (int j = i+1; j <array.length; j++) {  
 if (array[j] < min) {  
 min = array[j];  
minId = j;  
 }  
 }  
 // swapping  
 int temp = array[i];  
 array[i] = min;  
 array[minId] = temp;  
 }  
}

**Merge Sort:**

Merge Sort uses [recursion](https://stackabuse.com/understanding-recursive-functions-with-python/) to solve the problem of sorting more efficiently than algorithms previously presented, and in particular it uses a [divide and conquer](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) approach.

Using both of these concepts, we'll break the whole array down into two subarrays and then:

* Sort the left half of the array (recursively)
* Sort the right half of the array (recursively)
* Merge the solutions



**Implementation:**

The core function works pretty much as laid out in the explanation. We're just passing indexes left and right which are indexes of the left-most and right-most element of the subarray we want to sort. Initially, these should be 0 and array.length-1, depending on implementation.

The base of our recursion ensures we exit when we've finished, or when right and left meet each other. We find a midpoint mid, and sort subarrays left and right of it recursively, ultimately merging our solutions.

If you remember our tree graphic, you might wonder why don't we create two new smaller arrays and pass them on instead. This is because on really long arrays, this would cause huge memory consumption for something that's essentially unnecessary.

Merge Sort already doesn't work in-place because of the merge step, and this would only serve to worsen its memory efficiency. The logic of our tree of recursion otherwise stays the same

void merge(int[] array, int left, int mid, int right) {  
 // calculating lengths  
 int lengthLeft = mid - left + 1;  
 int lengthRight = right - mid;  
  
 // creating temporary subarrays  
 int leftArray[] = new int [lengthLeft];  
 int rightArray[] = new int [lengthRight];  
  
 // copying our sorted subarrays into temporaries  
 for (int i = 0; i<lengthLeft; i++)  
leftArray[i] = array[left+i];  
 for (int i = 0; i<lengthRight; i++)  
rightArray[i] = array[mid+i+1];  
  
 // iterators containing current index of temp subarrays  
 int leftIndex = 0;  
 int rightIndex = 0;  
  
 // copying from leftArray and rightArray back into array  
 for (int i = left; i< right + 1; i++) {  
 // if there are still uncopied elements in R and L, copy minimum of the two  
 if (leftIndex<lengthLeft&&rightIndex<lengthRight) {  
 if (leftArray[leftIndex] <rightArray[rightIndex]) {  
 array[i] = leftArray[leftIndex];  
leftIndex++;  
 }  
 else {  
 array[i] = rightArray[rightIndex];  
rightIndex++;  
 }  
 }  
 // if all the elements have been copied from rightArray, copy the rest of leftArray  
 else if (leftIndex<lengthLeft) {  
 array[i] = leftArray[leftIndex];  
leftIndex++;  
 }  
 // if all the elements have been copied from leftArray, copy the rest of rightArray  
 else if (rightIndex<lengthRight) {  
 array[i] = rightArray[rightIndex];  
rightIndex++;  
 }  
 }  
}

**Heap Sort:**

A heap is a tree that satisfies the heap property, which is that for each node, all of its children are in a given relation to it. Additionally, a heap must be almost complete. An almost complete binary tree of depth d has a subtree of depth d-1 with the same root that is complete, and in which each node with a left descendent has a complete left subtree. In other words, when adding a node, we always go for the leftmost position in the lowest incomplete level.

If the heap is a max-heap, then all of the children are smaller than the parent, and if it's a min-heap all of them are larger.

In other words, as you move down the tree, you get to smaller and smaller numbers (min-heap) or greater and greater numbers (max-heap).

**Implementation:**

static void heapify(int[] array, int length, int i) {  
 int leftChild = 2\*i+1;  
 int rightChild = 2\*i+2;  
 int largest = i;  
  
 // if the left child is larger than parent  
 if (leftChild< length && array[leftChild] > array[largest]) {  
 largest = leftChild;  
 }  
  
 // if the right child is larger than parent  
 if (rightChild< length && array[rightChild] > array[largest]) {  
 largest = rightChild;  
 }  
  
 // if a swap needs to occur  
 if (largest != i) {  
 int temp = array[i];  
 array[i] = array[largest];  
 array[largest] = temp;  
heapify(array, length, largest);  
 }  
}  
  
public static void heapSort(int[] array) {  
 if (array.length == 0) return;  
  
 // Building the heap  
 int length = array.length;  
 // we're going from the first non-leaf to the root  
 for (int i = length / 2-1; i>= 0; i--)  
heapify(array, length, i);  
  
 for (int i = length-1; i>= 0; i--) {  
 int temp = array[0];  
 array[0] = array[i];  
 array[i] = temp;  
  
heapify(array, i, 0);  
 }  
}

**Quick Sort:**

Quicksort is another Divide and Conquer algorithm. It picks one element of an array as the pivot and sorts all of the other elements around it, for example smaller elements to the left, and larger to the right.

This guarantees that the pivot is in its proper place after the process. Then the algorithm recursively does the same for the left and right portions of the array.

**Implementation:**

static int partition(int[] array, int begin, int end) {  
 int pivot = end;  
  
 int counter = begin;  
 for (int i = begin; i< end; i++) {  
 if (array[i] < array[pivot]) {  
 int temp = array[counter];  
 array[counter] = array[i];  
 array[i] = temp;  
 counter++;  
 }  
 }  
 int temp = array[pivot];  
 array[pivot] = array[counter];  
 array[counter] = temp;  
  
 return counter;  
}  
  
public static void quickSort(int[] array, int begin, int end) {  
 if (end <= begin) return;  
 int pivot = partition(array, begin, end);  
quickSort(array, begin, pivot-1);  
quickSort(array, pivot+1, end);  
}

**Quick Sort vs Merge Sort**

**Partition of elements in the array** :  
In the merge sort, the array is parted into just 2 halves (i.e. n/2).  
whereas  
In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.

**Worst case complexity** :  
The worst case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.  
whereas  
In merge sort, worst case and average case has same complexities O(n log n).

**Usage with datasets** :  
Merge sort can work well on any type of data sets irrespective of its size (either large or small).  
whereas  
The quick sort cannot work well with large datasets.

**Additional storage space requirement** :  
Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.  
whereas  
The quick sort is in place as it doesn’t require any additional storage.

**Efficiency** :  
Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.  
whereas  
Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.

**Sorting method** :  
The quick sort is internal sorting method where the data is sorted in main memory.  
whereas  
The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

**Stability** :  
Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.  
whereas  
Quick sort is unstable in this scenario. But it can be made stable using some changes in code.

**Preferred for** :  
Quick sort is preferred for arrays.  
whereas  
Merge sort is preferred for linked lists.

**Locality of reference** :  
Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

**Why is**[**Quick Sort**](http://geeksquiz.com/quick-sort/)**preferred for arrays?**

* Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm.
* Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.
* Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
* Quick Sort is also a cache friendly sorting algorithm as it has good [locality of reference](http://en.wikipedia.org/wiki/Locality_of_reference) when used for arrays.
* Quick Sort is also [tail recursive](https://www.geeksforgeeks.org/tail-recursion/), therefore tail call optimizations is done.

**Why is**[Merge Sort](http://geeksquiz.com/merge-sort/)**preferred for Linked Lists?**

* In case of [linked lists](http://geeksquiz.com/linked-list-set-1-introduction/) the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory.
* Unlike array, in [linked list](http://geeksquiz.com/linked-list-set-1-introduction/), we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
* In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list.
* Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

**Why quicksort is better than mergesort ?**

There are certain reasons due to which quicksort is better especially in case of arrays:

* **Auxiliary Space :** Mergesort uses extra space, quicksort requires little space and exhibits good cache locality. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.
* **Worst Cases :**The worst case of quicksort **O(n2)** can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.
* **Locality of reference :** Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment.
* **Merge sort is better for large data structures:**Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage.

**Searching in Arrays:**

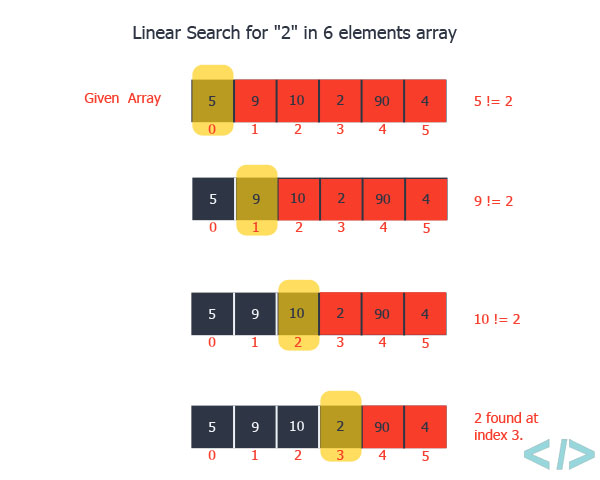
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

* **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: [Linear Search](https://www.geeksforgeeks.org/linear-search/).
* **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: [Binary Search](https://www.geeksforgeeks.org/binary-search/).

**Linear Search:**

A simple approach is to do **linear search**, i.e

* Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
* If x matches with an element, return the index.
* If x doesn’t match with any of elements, return -1.



The time complexity of above algorithm is O(n).

**Binary Search:**

We search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The time complexity of Binary Search can be written as

T(n) = T(n/2) + c

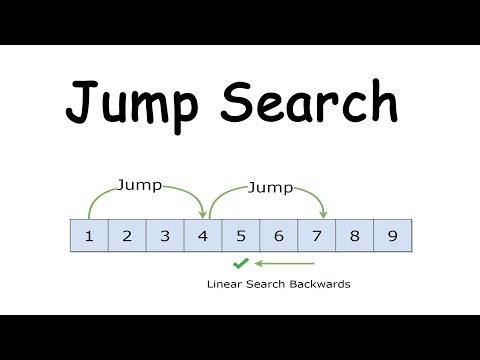
****

**Jump Search:**

Like [Binary Search](http://geeksquiz.com/binary-search/), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x <arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

In the worst case, we have to do n/m jumps and if the last checked value is greater than the element to be searched for, we perform m-1 comparisons more for linear search. Therefore the total number of comparisons in the worst case will be ((n/m) + m-1). The value of the function ((n/m) + m-1) will be minimum when m = √n. Therefore, the best step size is m = **√n.**

****

**Jump Search will**

* Works only sorted arrays.
* The optimal size of a block to be jumped is (√ n). This makes the time complexity of Jump Search O(√ n).
* The time complexity of Jump Search is between Linear Search ( ( O(n) ) and Binary Search ( O (Log n) ).
* Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be searched is the smallest element or smaller than the smallest). So in a system where binary search is costly, we use Jump Search.

**Interpolation Search**

Linear Search finds the element in O(n) time, [Jump Search](https://www.geeksforgeeks.org/jump-search/) takes O(√ n) time and [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) take O(Log n) time.The Interpolation Search is an improvement over [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side.

To find the position to be searched, it uses following formula.

// The idea of formula is to return higher value of **pos**

// when element to be searched is closer to **arr[hi]**. And

// smaller value when closer to **arr[lo]**

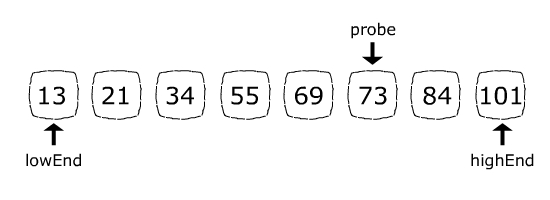
pos = lo + [ (x-arr[lo])\*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched

x ==> Element to be searched

lo ==> Starting index in arr[]

hi ==> Ending index in arr[]

****

**Exponential Search:**

**Exponential search involves two steps:**

* Find range where element is present
* Do Binary Search in above found range.

**How to find the range where element may be present?**

The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.  
Once we find an index i (after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration)

**Applications of Exponential Search:**

* Exponential Binary Search is particularly useful for unbounded searches, where size of array is infinite. Please refer [Unbounded Binary Search](https://www.geeksforgeeks.org/find-the-point-where-a-function-becomes-negative/) for an example.
* It works better than Binary Search for bounded arrays, and also when the element to be searched is closer to the first element.

**Sublist Search**

It will search a linked list in another list.

**Algorithm:**1- Take first node of second list.  
2- Start matching the first list from this first node.  
3- If whole lists match return true.  
4- Else break and take first list to the first node again.  
5- And take second list to its second node.  
6- Repeat these steps until any of linked lists becomes empty.  
7- If first list becomes empty then list found else not.

**Fibanocci Search :**

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

**Similarities with Binary Search:**

* Works for sorted arrays
* A Divide and Conquer Algorithm.
* Has Log n time complexity.

**Differences with Binary Search:**

* Fibonacci Search divides given array in unequal parts
* Binary Search uses division operator to divide range. Fibonacci Search doesn’t use /, but uses + and -. The division operator may be costly on some CPUs.
* Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

**Algorithm:**

Let the searched element be x.

The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of given array. Let the found Fibonacci number be fib (m’th Fibonacci number). We use (m-2)’th Fibonacci number as the index (If it is a valid index). Let (m-2)’th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

**Below is the complete algorithm**

Let arr[0..n-1] be the input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m’th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)’th Fibonacci Number] and fibMm2 [(m-2)’th Fibonacci Number].
2. While the array has elements to be inspected:
3. Compare x with the last element of the range covered by fibMm2
4. **If** x matches, return index
5. **Else If** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
6. **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
7. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

**Topics :**

* [Introduction](https://www.geeksforgeeks.org/array-data-structure/#introduction)
* [Array Rotations](https://www.geeksforgeeks.org/array-data-structure/#rotation)
* [Arrangement Rearrangement](https://www.geeksforgeeks.org/array-data-structure/#rearrange)
* [Order Statistics](https://www.geeksforgeeks.org/array-data-structure/#order)
* [Range Queries](https://www.geeksforgeeks.org/array-data-structure/#range)
* [Searching and Sorting](https://www.geeksforgeeks.org/array-data-structure/#searchSort)
* [Optimization Problems](https://www.geeksforgeeks.org/array-data-structure/#optimization)
* [Matrix](https://www.geeksforgeeks.org/array-data-structure/#Matrix)
* [Misc](https://www.geeksforgeeks.org/array-data-structure/#misc)

**Array Rotations :**

**Examples:**

**1) Program for array rotation**

class RotateArray

{

/\*Function to left rotate arr[] of size n by d\*/

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

{

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

leftRotatebyOne(arr, n);

}

void leftRotatebyOne(int arr[], int n)

{

int i, temp;

temp = arr[0];

for (i = 0; i < n - 1; i++)

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

arr[i] = temp;

}

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

void printArray(int arr[], int n)

{

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

System.out.print(arr[i] + " ");

}

// Driver program to test above functions

public static void main(String[] args)

{

RotateArray rotate = new RotateArray();

int arr[] = { 1, 2, 3, 4, 5, 6, 7 };

rotate.leftRotate(arr, 2, 7);

rotate.printArray(arr, 7);

}

}

Output:

3 4 5 6 7 1 2

**2) Reversal algorithm for array rotation**

import java.io.\*;

class LeftRotate {

/\* Function to left rotate arr[] of size n by d \*/

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

{

if (d == 0)

return;

int n = arr.length;

rvereseArray(arr, 0, d - 1);

rvereseArray(arr, d, n - 1);

rvereseArray(arr, 0, n - 1);

}

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

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

{

int temp;

while (start < end) {

temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

/\*UTILITY FUNCTIONS\*/

/\* function to print an array \*/

static void printArray(int arr[])

{

for (int i = 0; i < arr.length; i++)

System.out.print(arr[i] + " ");

}

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

public static void main(String[] args)

{

int arr[] = { 1, 2, 3, 4, 5, 6, 7 };

int n = arr.length;

int d = 2;

// in case the rotating factor is

// greater than array length

d = d % n;

leftRotate(arr, d); // Rotate array by d

printArray(arr);

}

}

Output:

3 4 5 6 7 1 2

**3) Block swap algorithm for array rotation**

import java.util.\*;

class GFG

{

public static void leftRotate(int arr[], int i,

int d, int n)

{

/\* Return If number of elements to be rotated

is zero or equal to array size \*/

if(d == 0 || d == n)

return;

/\*If number of elements to be rotated

is exactly half of array size \*/

if(n - d == d)

{

swap(arr, i, n - d + i, d);

return;

}

/\* If A is shorter\*/

if(d < n - d)

{

swap(arr, i, n - d + i, d);

leftRotate(arr, i, d, n - d);

}

else /\* If B is shorter\*/

{

swap(arr, i, d, n - d);

leftRotate(arr, n - d + i, 2 \* d - n, d); /\*This is tricky\*/

}

}

/\*UTILITY FUNCTIONS\*/

/\* function to print an array \*/

public static void printArray(int arr[], int size)

{

int i;

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

System.out.print(arr[i] + " ");

System.out.println();

}

/\*This function swaps d elements

starting at index fi with d elements

starting at index si \*/

public static void swap(int arr[], int fi,

int si, int d)

{

int i, temp;

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

{

temp = arr[fi + i];

arr[fi + i] = arr[si + i];

arr[si + i] = temp;

}

}

// Driver Code

public static void main (String[] args)

{

int arr[] = {1, 2, 3, 4, 5, 6, 7};

leftRotate(arr,0, 2, 7);

printArray(arr, 7);

}

}

Output:

3 4 5 6 7 1 2

**4) Find maximum value of Sum( i\*arr[i]) with only rotations on given array allowed**

|  |
| --- |
| import java.util.Arrays;    class Test  {      static int arr[] = new int[]{10, 1, 2, 3, 4, 5, 6, 7, 8, 9};        // Returns max possible value of i\*arr[i]      static int maxSum()      {          // Find array sum and i\*arr[i] with no rotation          int arrSum = 0;  // Stores sum of arr[i]          int currVal = 0;  // Stores sum of i\*arr[i]          for (int i=0; i<arr.length; i++)          {              arrSum = arrSum + arr[i];              currVal = currVal+(i\*arr[i]);          }            // Initialize result as 0 rotation sum          int maxVal = currVal;            // Try all rotations one by one and find          // the maximum rotation sum.          for (int j=1; j<arr.length; j++)          {              currVal = currVal + arrSum-arr.length\*arr[arr.length-j];              if (currVal > maxVal)                  maxVal = currVal;          }            // Return result          return maxVal;      }        // Driver method to test the above function      public static void main(String[] args)      {          System.out.println("Max sum is " + maxSum());      }  } |

**Output :**

Max sum is 330

**Time Complexity** : O(n)

**5) Maximum sum of i\*arr[i] among all rotations of a given array**

|  |
| --- |
| import java.util.\*;  import java.io.\*;    class GFG {    // Returns maximum value of i\*arr[i]  static int maxSum(int arr[], int n)  {  // Initialize result  int res = Integer.MIN\_VALUE;    // Consider rotation beginning with i  // for all possible values of i.  for (int i = 0; i < n; i++)  {        // Initialize sum of current rotation      int curr\_sum = 0;        // Compute sum of all values. We don't      // actually rotation the array, but compute      // sum by finding ndexes when arr[i] is      // first element      for (int j = 0; j < n; j++)      {          int index = (i + j) % n;          curr\_sum += j \* arr[index];      }        // Update result if required      res = Math.max(res, curr\_sum);  }    return res;  }    // Driver code  public static void main(String args[])  {          int arr[] = {8, 3, 1, 2};          int n = arr.length;          System.out.println(maxSum(arr, n));  }      } |

**Output :**

29

**Time Complexity :** O(n2)

**6) Find the Rotation Count in Rotated Sorted array**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  import java.io.\*;    class LinearSearch  {      // Returns count of rotations for an      // array which is first sorted in      // ascending order, then rotated      static int countRotations(int arr[], int n)      {          // We basically find index of minimum          // element          int min = arr[0], min\_index = -1;          for (int i = 0; i < n; i++)          {              if (min > arr[i])              {                  min = arr[i];                  min\_index = i;              }          }          return min\_index;      }        // Driver program to test above functions      public static void main (String[] args)      {          int arr[] = {15, 18, 2, 3, 6, 12};          int n = arr.length;            System.out.println(countRotations(arr, n));      }  } |

**Output:**

2

**Time Complexity :** O(n)

**7) Quickly find multiple left rotations of an arra**y

|  |
| --- |
| class LeftRotate  {      // Fills temp[] with two copies of arr[]      static void preprocess(int arr[], int n,                                     int temp[])      {          // Store arr[] elements at i and i + n          for (int i = 0; i<n; i++)               temp[i] = temp[i + n] = arr[i];      }        // Function to left rotate an array k time      static void leftRotate(int arr[], int n, int k,                                      int temp[])      {          // Starting position of array after k          // rotations in temp[] will be k % n          int start = k % n;            // Print array after k rotations          for (int i = start; i < start + n; i++)              System.out.print(temp[i] + " ");            System.out.print("\n");      }        // Driver program      public static void main (String[] args)      {          int arr[] = {1, 3, 5, 7, 9};          int n = arr.length;            int temp[] = new int[2\*n];          preprocess(arr, n, temp);            int k = 2;          leftRotate(arr, n, k, temp);            k = 3;          leftRotate(arr, n, k, temp);            k = 4;          leftRotate(arr, n, k, temp);      }  } |

**Output:**

5 7 9 1 3

7 9 1 3 5

9 1 3 5 7

**8) Find the minimum element in a sorted and rotated array**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  import java.io.\*;    class Minimum  {      static int findMin(int arr[], int low, int high)      {          // This condition is needed to handle the case when array          // is not rotated at all          if (high < low)  return arr[0];            // If there is only one element left          if (high == low) return arr[low];            // Find mid          int mid = low + (high - low)/2; /\*(low + high)/2;\*/            // Check if element (mid+1) is minimum element. Consider          // the cases like {3, 4, 5, 1, 2}          if (mid < high && arr[mid+1] < arr[mid])              return arr[mid+1];            // Check if mid itself is minimum element          if (mid > low && arr[mid] < arr[mid - 1])              return arr[mid];            // Decide whether we need to go to left half or right half          if (arr[high] > arr[mid])              return findMin(arr, low, mid-1);          return findMin(arr, mid+1, high);      }        // Driver Program      public static void main (String[] args)      {          int arr1[] =  {5, 6, 1, 2, 3, 4};          int n1 = arr1.length;          System.out.println("The minimum element is "+ findMin(arr1, 0, n1-1));            int arr2[] =  {1, 2, 3, 4};          int n2 = arr2.length;          System.out.println("The minimum element is "+ findMin(arr2, 0, n2-1));            int arr3[] =  {1};          int n3 = arr3.length;          System.out.println("The minimum element is "+ findMin(arr3, 0, n3-1));            int arr4[] =  {1, 2};          int n4 = arr4.length;          System.out.println("The minimum element is "+ findMin(arr4, 0, n4-1));            int arr5[] =  {2, 1};          int n5 = arr5.length;          System.out.println("The minimum element is "+ findMin(arr5, 0, n5-1));            int arr6[] =  {5, 6, 7, 1, 2, 3, 4};          int n6 = arr6.length;          System.out.println("The minimum element is "+ findMin(arr6, 0, n6-1));            int arr7[] =  {1, 2, 3, 4, 5, 6, 7};          int n7 = arr7.length;          System.out.println("The minimum element is "+ findMin(arr7, 0, n7-1));            int arr8[] =  {2, 3, 4, 5, 6, 7, 8, 1};          int n8 = arr8.length;          System.out.println("The minimum element is "+ findMin(arr8, 0, n8-1));            int arr9[] =  {3, 4, 5, 1, 2};          int n9 = arr9.length;          System.out.println("The minimum element is "+ findMin(arr9, 0, n9-1));      }  } |

**Output:**

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

The minimum element is 1

**9) Reversal algorithm for right rotation of an array**

|  |
| --- |
| import java.io.\*;    class GFG  {      // Function to reverse arr[]      // from index start to end      static void reverseArray(int arr[], int start,                               int end)      {          while (start < end)          {             int temp = arr[start];             arr[start] = arr[end];             arr[end] = temp;             start++;             end--;           }      }        // Function to right rotate      // arr[] of size n by d      static void rightRotate(int arr[], int d, int n)      {         reverseArray(arr, 0, n - 1);         reverseArray(arr, 0, d - 1);         reverseArray(arr, d, n - 1);      }        // Function to print an array      static void printArray(int arr[], int size)      {         for (int i = 0; i < size; i++)            System.out.print(arr[i] + " ");      }        public static void main (String[] args)      {          int arr[] = {1, 2, 3, 4, 5,                       6, 7, 8, 9, 10};        int n = arr.length;      int k = 3;        rightRotate(arr, k, n);      printArray(arr, n);        }  } |

**Output:**

8 9 10 1 2 3 4 5 6 7

**10) Find a rotation with maximum hamming distance**

|  |
| --- |
| class GFG  {  // Return the maximum hamming  // distance of a rotation  static int maxHamming(int arr[], int n)  {      // arr[] to brr[] two times so that      // we can traverse through all rotations.      int brr[]=new int[2 \*n + 1];      for (int i = 0; i < n; i++)          brr[i] = arr[i];      for (int i = 0; i < n; i++)          brr[n+i] = arr[i];        // We know hamming distance with      // 0 rotation would be 0.      int maxHam = 0;        // We try other rotations one by one      // and compute Hamming distance      // of every rotation      for (int i = 1; i < n; i++)      {          int currHam = 0;          for (int j = i, k=0; j < (i + n); j++,                                            k++)              if (brr[j] != arr[k])                  currHam++;            // We can never get more than n.          if (currHam == n)              return n;            maxHam = Math.max(maxHam, currHam);      }        return maxHam;  }    // driver code  public static void main (String[] args)  {      int arr[] = {2, 4, 6, 8};      int n = arr.length;      System.out.print(maxHamming(arr, n));  }  } |

**Output:**

4

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

**11) Print left rotation of array in O(n) time and O(1) space**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  import java.io.\*;    class arr\_rot  {      // Function to leftRotate array multiple      // times      static void leftRotate(int arr[], int n,                                       int k)      {          /\* To get the starting point of          rotated array \*/          int mod = k % n;            // Prints the rotated array from          // start position          for(int i = 0; i < n; ++i)          System.out.print(arr[(i + mod) % n]                            + " ");            System.out.println();      }        // Driver program      public static void main (String[] args)      {              int arr[] = { 1, 3, 5, 7, 9 };              int n = arr.length;                int k = 2;              leftRotate(arr, n, k);                k = 3;              leftRotate(arr, n, k);                k = 4;              leftRotate(arr, n, k);      }  } |

**Output :**

5 7 9 1 3

7 9 1 3 5

9 1 3 5 7

**Arrangement Rearrangement :**

**Examples:**

**1) Rearrange an array such that arr[i] = i**

import java.util.\*;

import java.lang.\*;

public class GfG {

// Function to rearrange an array

// such that arr[i] = i.

public static int[] fix(int[] A)

{

for (int i = 0; i < A.length; i++)

{

if (A[i] != -1 && A[i] != i)

{

int x = A[i];

// check if desired place

// is not vacate

while (A[x] != -1 && A[x] != x)

{

// store the value from

// desired place

int y = A[x];

// place the x to its correct

// position

A[x] = x;

// now y will become x, now

// search the place for x

x = y;

}

// place the x to its correct

// position

A[x] = x;

// check if while loop hasn't

// set the correct value at A[i]

if (A[i] != i)

{

// if not then put -1 at

// the vacated place

A[i] = -1;

}

}

}

return A;

}

// Driver function.

public static void main(String[] args)

{

int A[] = {-1, -1, 6, 1,

9, 3, 2, -1, 4,-1};

System.out.println(Arrays.toString(fix(A)));

}

}

Output:

[-1, 1, 2, 3, 4, -1, 6, -1, -1, 9]

**2) Write a program to reverse an array or string**

public class GFG {

/\* Function to reverse arr[] from

start to end\*/

static void rvereseArray(int arr[],

int start, int end)

{

int temp;

while (start < end)

{

temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

/\* Utility that prints out an

array on a line \*/

static void printArray(int arr[],

int size)

{

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

System.out.print(arr[i] + " ");

System.out.println();

}

// Driver code

public static void main(String args[]) {

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

printArray(arr, 6);

rvereseArray(arr, 0, 5);

System.out.print("Reversed array is \n");

printArray(arr, 6);

}

}

Output:

1 2 3 4 5 6

Reversed array is

6 5 4 3 2 1

**3) Rearrange array such that arr[i] >= arr[j] if i is even and arr[i]<=arr[j] if i is odd and j < i**

import java.util.\*;

import java.lang.\*;

public class GfG{

// function to rearrange the array

public static void rearrangeArr(int arr[],

int n)

{

// total even positions

int evenPos = n / 2;

// total odd positions

int oddPos = n - evenPos;

int[] tempArr = new int [n];

// copy original array in an

// auxiliary array

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

tempArr[i] = arr[i];

// sort the auxiliary array

Arrays.sort(tempArr);

int j = oddPos - 1;

// fill up odd position in

// original array

for (int i = 0; i < n; i += 2) {

arr[i] = tempArr[j];

j--;

}

j = oddPos;

// fill up even positions in

// original array

for (int i = 1; i < n; i += 2) {

arr[i] = tempArr[j];

j++;

}

// display array

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

System.out.print(arr[i] + " ");

}

// Driver function

public static void main(String argc[]){

int[] arr = new int []{ 1, 2, 3, 4, 5,

6, 7 };

int size = 7;

rearrangeArr(arr, size);

}

}

Output:

4 5 3 6 2 7 1

**4) Rearrange positive and negative numbers in O(n) time and O(1) extra space**

|  |
| --- |
| import java.io.\*;    class Alternate {        // The main function that rearranges elements of given      // array.  It puts positive elements at even indexes (0,      // 2, ..) and negative numbers at odd indexes (1, 3, ..).      static void rearrange(int arr[], int n)      {          // The following few lines are similar to partition          // process of QuickSort.  The idea is to consider 0          // as pivot and divide the array around it.          int i = -1, temp = 0;          for (int j = 0; j < n; j++)          {              if (arr[j] < 0)              {                  i++;                  temp = arr[i];                  arr[i] = arr[j];                  arr[j] = temp;              }          }            // Now all positive numbers are at end and negative numbers at          // the beginning of array. Initialize indexes for starting point          // of positive and negative numbers to be swapped          int pos = i+1, neg = 0;            // Increment the negative index by 2 and positive index by 1, i.e.,          // swap every alternate negative number with next positive number          while (pos < n && neg < pos && arr[neg] < 0)          {              temp = arr[neg];              arr[neg] = arr[pos];              arr[pos] = temp;              pos++;              neg += 2;          }      }        // A utility function to print an array      static void printArray(int arr[], int n)      {          for (int i = 0; i < n; i++)              System.out.print(arr[i] + "   ");      }        /\*Driver function to check for above functions\*/      public static void main (String[] args)      {          int arr[] = {-1, 2, -3, 4, 5, 6, -7, 8, 9};          int n = arr.length;          rearrange(arr,n);          System.out.println("Array after rearranging: ");          printArray(arr,n);      }  } |

**Output:**

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

**Time Complexity:** O(n) where n is number of elements in given array.

**5) Rearrange array in alternating positive & negative items with O(1) extra space**

|  |
| --- |
| class RearrangeArray  {      // Utility function to right rotate all elements      // between [outofplace, cur]      void rightrotate(int arr[], int n, int outofplace, int cur)      {          int tmp = arr[cur];          for (int i = cur; i > outofplace; i--)              arr[i] = arr[i - 1];          arr[outofplace] = tmp;      }        void rearrange(int arr[], int n)      {          int outofplace = -1;            for (int index = 0; index < n; index++)          {              if (outofplace >= 0)              {                  // find the item which must be moved into the out-of-place                  // entry if out-of-place entry is positive and current                  // entry is negative OR if out-of-place entry is negative                  // and current entry is negative then right rotate                  //                  // [...-3, -4, -5, 6...] -->   [...6, -3, -4, -5...]                  //      ^                          ^                  //      |                          |                  //     outofplace      -->      outofplace                  //                  if (((arr[index] >= 0) && (arr[outofplace] < 0))                          || ((arr[index] < 0) && (arr[outofplace] >= 0)))                  {                      rightrotate(arr, n, outofplace, index);                        // the new out-of-place entry is now 2 steps ahead                      if (index - outofplace > 2)                          outofplace = outofplace + 2;                      else                          outofplace = -1;                  }              }                // if no entry has been flagged out-of-place              if (outofplace == -1)              {                  // check if current entry is out-of-place                  if (((arr[index] >= 0) && ((index & 0x01)==0))                          || ((arr[index] < 0) && (index & 0x01)==1))                      outofplace = index;              }          }      }        // A utility function to print an array 'arr[]' of size 'n'      void printArray(int arr[], int n)      {          for (int i = 0; i < n; i++)              System.out.print(arr[i] + " ");          System.out.println("");      }        public static void main(String[] args)      {          RearrangeArray rearrange = new RearrangeArray();          //int arr[n] = {-5, 3, 4, 5, -6, -2, 8, 9, -1, -4};          //int arr[] = {-5, -3, -4, -5, -6, 2 , 8, 9, 1 , 4};          //int arr[] = {5, 3, 4, 2, 1, -2 , -8, -9, -1 , -4};          //int arr[] = {-5, 3, -4, -7, -1, -2 , -8, -9, 1 , -4};          int arr[] = {-5, -2, 5, 2, 4, 7, 1, 8, 0, -8};          int n = arr.length;            System.out.println("Given array is ");          rearrange.printArray(arr, n);            rearrange.rearrange(arr, n);            System.out.println("RearrangeD array is ");          rearrange.printArray(arr, n);      }  } |

**Output:**

Given array is

-5 -2 5 2 4 7 1 8 0 -8

Rearranged array is

-5 5 -2 2 -8 4 7 1 8 0

**6) Minimum swaps required to bring all elements less than or equal to k together**

|  |
| --- |
| import java.lang.\*;    class GFG {    // Utility function to find minimum swaps  // required to club all elements less than  // or equals to k together  static int minSwap(int arr[], int n, int k) {        // Find count of elements which are      // less than equals to k      int count = 0;      for (int i = 0; i < n; ++i)      if (arr[i] <= k)          ++count;        // Find unwanted elements in current      // window of size 'count'      int bad = 0;      for (int i = 0; i < count; ++i)      if (arr[i] > k)          ++bad;        // Initialize answer with 'bad' value of      // current window      int ans = bad;      for (int i = 0, j = count; j < n; ++i, ++j) {        // Decrement count of previous window      if (arr[i] > k)          --bad;        // Increment count of current window      if (arr[j] > k)          ++bad;        // Update ans if count of 'bad'      // is less in current window      ans = Math.min(ans, bad);      }      return ans;  }    // Driver code  public static void main(String[] args)  {      int arr[] = {2, 1, 5, 6, 3};      int n = arr.length;      int k = 3;      System.out.print(minSwap(arr, n, k) + "\n");        int arr1[] = {2, 7, 9, 5, 8, 7, 4};      n = arr1.length;      k = 5;      System.out.print(minSwap(arr1, n, k));  }  } |

**Output :**

1

2

**Time complexity:**O(n)

**7) Rearrange array such that even positioned are greater than odd**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG {    static void assign(int a[], int n) {        // Sort the array      Arrays.sort(a);        int ans[] = new int[n];      int p = 0, q = n - 1;      for (int i = 0; i < n; i++) {        // Assign even indexes with maximum elements      if ((i + 1) % 2 == 0)          ans[i] = a[q--];        // Assign odd indexes with remaining elements      else          ans[i] = a[p++];      }        // Print result      for (int i = 0; i < n; i++)      System.out.print(ans[i] + " ");  }    // Driver code  public static void main(String args[]) {      int A[] = {1, 3, 2, 2, 5};      int n = A.length;      assign(A, n);  }  } |

**Output:**

1 5 2 3 2

**8) Double the first element and move zero to end**

|  |
| --- |
| class GFG {        // function which pushes all      // zeros to end of an array.      static void pushZerosToEnd(int arr[], int n)      {          // Count of non-zero elements          int count = 0;            // Traverse the array. If element          // encountered is non-zero, then          // replace the element at index          // 'count' with this element          for (int i = 0; i < n; i++)              if (arr[i] != 0)                    // here count is incremented                  arr[count++] = arr[i];            // Now all non-zero elements          // have been shifted to front and          // 'count' is set as index of first 0.          // Make all elements 0 from count to end.          while (count < n)              arr[count++] = 0;      }        // function to rearrange the array      //  elements after modification      static void modifyAndRearrangeArr(int arr[], int n)      {          // if 'arr[]' contains a single element          // only          if (n == 1)              return;            // traverse the array          for (int i = 0; i < n - 1; i++) {                // if true, perform the required modification              if ((arr[i] != 0) && (arr[i] == arr[i + 1]))              {                    // double current index value                  arr[i] = 2 \* arr[i];                    // put 0 in the next index                  arr[i + 1] = 0;                    // increment by 1 so as to move two                  // indexes ahead during loop iteration                  i++;              }          }            // push all the zeros at          // the end of 'arr[]'          pushZerosToEnd(arr, n);      }        // function to print the array elements      static void printArray(int arr[], int n)      {          for (int i = 0; i < n; i++)              System.out.print(arr[i] + " ");          System.out.println();      }        // Driver program to test above      public static void main(String[] args)      {          int arr[] = { 0, 2, 2, 2, 0, 6, 6, 0, 0, 8 };          int n = arr.length;            System.out.print("Original array: ");          printArray(arr, n);            modifyAndRearrangeArr(arr, n);            System.out.print("Modified array: ");          printArray(arr, n);      }  } |

Output:

Original array: 0 2 2 2 0 6 6 0 0 8

Modified array: 4 2 12 8 0 0 0 0 0 0

Time Complexity: O(n).

**9) Reorder an array according to given indexes**

|  |
| --- |
| import java.util.Arrays;    class Test  {      static int arr[] = new int[]{50, 40, 70, 60, 90};      static int index[] = new int[]{3,  0,  4,  1,  2};        // Method to reorder elements of arr[] according      // to index[]      static void reorder()      {          int temp[] = new int[arr.length];            // arr[i] should be present at index[i] index          for (int i=0; i<arr.length; i++)              temp[index[i]] = arr[i];            // Copy temp[] to arr[]          for (int i=0; i<arr.length; i++)          {             arr[i]   = temp[i];             index[i] = i;          }      }        // Driver method to test the above function      public static void main(String[] args)      {            reorder();            System.out.println("Reordered array is: ");          System.out.println(Arrays.toString(arr));          System.out.println("Modified Index array is:");          System.out.println(Arrays.toString(index));        }  } |

**Output:**

Reordered array is:

40 60 90 50 70

Modified Index array is:

0 1 2 3 4

**10) Rearrange positive and negative numbers with constant extra space**

|  |
| --- |
| import java.io.\*;    class GFG {      // A utility function to print      // an array of size n      static void printArray(int arr[], int n)      {          for (int i = 0; i < n; i++)              System.out.print(arr[i] + " ");          System.out.println();      }        // Function to Rearrange positive and negative      // numbers in a array      static void RearrangePosNeg(int arr[], int n)      {          int key, j;          for (int i = 1; i < n; i++) {              key = arr[i];                // if current element is positive              // do nothing              if (key > 0)                  continue;                /\* if current element is negative,              shift positive elements of arr[0..i-1],              to one position to their right \*/              j = i - 1;              while (j >= 0 && arr[j] > 0) {                  arr[j + 1] = arr[j];                  j = j - 1;              }                // Put negative element at its right position              arr[j + 1] = key;          }      }        // Driver program      public static void main(String[] args)      {          int arr[] = { -12, 11, -13, -5, 6, -7, 5, -3, -6 };          int n = arr.length;          RearrangePosNeg(arr, n);          printArray(arr, n);      }  } |

**Output:**

-12 -13 -5 -7 -3 -6 11 6 5

Time complexity of above solution is O(n2)

**11) Arrange given numbers to form the biggest number**

|  |
| --- |
| import java.util.\*;    class GFG {        // The main function that prints the      // arrangement with the largest value.      // The function accepts a vector of strings      static void printLargest(Vector<String> arr){            Collections.sort(arr, new Comparator<String>(){            // A comparison function which is used by          // sort() in printLargest()          @Override          public int compare(String X, String Y) {            // first append Y at the end of X          String XY=X + Y;            // then append X at the end of Y          String YX=Y + X;            // Now see which of the two formed numbers          // is greater          return XY.compareTo(YX) > 0 ? -1:1;      }      });        Iterator it = arr.iterator();        while(it.hasNext())          System.out.print(it.next());        }        // driver program      public static void main (String[] args) {            Vector<String> arr;          arr = new Vector<>();            //output should be 6054854654          arr.add("54");          arr.add("546");          arr.add("548");          arr.add("60");          printLargest(arr);      }  } |

**Output:**

6054854654

**12) Rearrange an array such that ‘arr[j]’ becomes ‘i’ if ‘arr[i]’ is ‘j’**

|  |
| --- |
| class RearrangeArray {      // A simple method to rearrange 'arr[0..n-1]' so that 'arr[j]'      // becomes 'i' if 'arr[i]' is 'j'      void rearrangeNaive(int arr[], int n)      {          // Create an auxiliary array of same size          int temp[] = new int[n];          int i;            // Store result in temp[]          for (i = 0; i < n; i++)              temp[arr[i]] = i;            // Copy temp back to arr[]          for (i = 0; i < n; i++)              arr[i] = temp[i];      }        // A utility function to print contents of arr[0..n-1]      void printArray(int arr[], int n)      {          int i;          for (i = 0; i < n; i++) {              System.out.print(arr[i] + " ");          }          System.out.println("");      }        // Driver program to test above functions      public static void main(String[] args)      {          RearrangeArray arrange = new RearrangeArray();          int arr[] = { 1, 3, 0, 2 };          int n = arr.length;            System.out.println("Given array is ");          arrange.printArray(arr, n);            arrange.rearrangeNaive(arr, n);            System.out.println("Modified array is ");          arrange.printArray(arr, n);      }  } |

**Output:**

Given array is

1 3 0 2

Modified array is

2 0 3 1

Time complexity of the above solution is O(n)

**13) Rearrange an array in maximum minimum form**

|  |
| --- |
| import java.util.Arrays;    public class GFG  {      // Prints max at first position, min at second position      // second max at third position, second min at fourth      // position and so on.      static void rearrange(int[] arr, int n)      {          // Auxiliary array to hold modified array          int temp[] = new int[n];            // Indexes of smallest and largest elements          // from remaining array.          int small=0, large=n-1;            // To indicate whether we need to copy rmaining          // largest or remaining smallest at next position          boolean flag = true;            // Store result in temp[]          for (int i=0; i<n; i++)          {              if (flag)                  temp[i] = arr[large--];              else                  temp[i] = arr[small++];                flag = !flag;          }            // Copy temp[] to arr[]          arr = temp.clone();      }        // Driver method to test the above function      public static void main(String[] args)      {          int arr[] = new int[]{1, 2, 3, 4, 5, 6};            System.out.println("Original Array ");          System.out.println(Arrays.toString(arr));            rearrange(arr,arr.length);            System.out.println("Modified Array ");          System.out.println(Arrays.toString(arr));        }  } |

**Output:**

Original Array

1 2 3 4 5 6

Modified Array

6 1 5 2 4 3

**Time Complexity:** O(n)

**14) Rearrange array such that even index elements are smaller and odd index elements are greater**

|  |
| --- |
| class GFG {        static void rearrange(int arr[], int n)      {            int temp;          for (int i = 0; i < n - 1; i++) {              if (i % 2 == 0 && arr[i] > arr[i + 1]) {                  temp = arr[i];                  arr[i] = arr[i + 1];                  arr[i + 1] = temp;              }              if (i % 2 != 0 && arr[i] < arr[i + 1]) {                  temp = arr[i];                  arr[i] = arr[i + 1];                  arr[i + 1] = temp;              }          }      }        /\* Utility that prints out an array in      a line \*/      static void printArray(int arr[], int size)      {          for (int i = 0; i < size; i++)              System.out.print(arr[i] + " ");            System.out.println();      }        // Driver code      public static void main(String[] args)      {          int arr[] = { 6, 4, 2, 1, 8, 3 };          int n = arr.length;            System.out.print("Before rearranging: \n");          printArray(arr, n);            rearrange(arr, n);            System.out.print("After rearranging: \n");          printArray(arr, n);      }  } |

**Output:**

Before rearranging:

6 4 2 1 8 3

After rearranging:

4 6 1 8 2 3

**Time Complexity :**O(n)

**15) Positive elements at even and negative at odd positions (Relative order not maintained)**

|  |
| --- |
| import java.io.\*;    class GFG {    static void rearrange(int a[], int size)  {    int positive = 0, negative = 1, temp;        while (true) {        /\* Move forward the positive pointer till      negative number number not encountered \*/      while (positive < size && a[positive] >= 0)          positive += 2;        /\* Move forward the negative pointer till          positive number number not encountered \*/      while (negative < size && a[negative] <= 0)          negative += 2;        // Swap array elements to fix their position.      if (positive < size && negative < size) {          temp = a[positive];          a[positive] = a[negative];          a[negative] = temp;      }        /\* Break from the while loop when any index      exceeds the size of the array \*/      else          break;      }  }    // Driver code  public static void main(String args[]) {      int arr[] = {1, -3, 5, 6, -3, 6, 7, -4, 9, 10};      int n = arr.length;        rearrange(arr, n);      for (int i = 0; i < n; i++)      System.out.print(arr[i] + " ");  }  } |

**Output:**

1 -3 5 -3 6 6 7 -4 9 10

**16) Replace every array element by multiplication of previous and next**

|  |
| --- |
| import java.io.\*;  import java.util.\*;  import java.lang.Math;    class Multipy  {     static void modify(int arr[], int n)      {          // Nothing to do when array size is 1          if (n <= 1)              return;            // store current value of arr[0] and update it          int prev = arr[0];          arr[0] = arr[0] \* arr[1];            // Update rest of the array elements          for (int i=1; i<n-1; i++)          {              // Store current value of next interation              int curr = arr[i];                // Update current value using previos value              arr[i] = prev \* arr[i+1];                // Update previous value              prev = curr;          }            // Update last array element          arr[n-1] = prev \* arr[n-1];      }        // Driver program to test above function      public static void main(String[] args)      {          int arr[] = {2, 3, 4, 5, 6};          int n = arr.length;          modify(arr, n);          for (int i=0; i<n; i++)           System.out.print(arr[i]+" ");      }  } |

Output:

6 8 15 24 30

**17) Shuffle a given array using Fisher–Yates shuffle Algorithm**

|  |
| --- |
| import java.util.Random;  import java.util.Arrays;  public class ShuffleRand  {      // A Function to generate a random permutation of arr[]      static void randomize( int arr[], int n)      {          // Creating a object for Random class          Random r = new Random();            // Start from the last element and swap one by one. We don't          // need to run for the first element that's why i > 0          for (int i = n-1; i > 0; i--) {                // Pick a random index from 0 to i              int j = r.nextInt(i+1);                // Swap arr[i] with the element at random index              int temp = arr[i];              arr[i] = arr[j];              arr[j] = temp;          }          // Prints the random array          System.out.println(Arrays.toString(arr));      }        // Driver Program to test above function      public static void main(String[] args)      {             int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};           int n = arr.length;           randomize (arr, n);      }  } |

**Output :**

7 8 4 6 3 1 2 5

The above function assumes that rand() generates a random number.

**Time Complexity:** O(n)

**Order Statistics :**

**Examples:**

**1) K’th Smallest/Largest Element in Unsorted Array**

class GFG

{

// Function to return k'th smallest

// element in a given array

public static int kthSmallest(Integer [] arr,

int k)

{

// Sort the given array

Arrays.sort(arr);

// Return k'th element in

// the sorted array

return arr[k-1];

}

// driver program

public static void main(String[] args)

{

Integer arr[] = new Integer[]{12, 3, 5, 7, 19};

int k = 2;

System.out.print( "K'th smallest element is "+

kthSmallest(arr, k) );

}

}

Output:

K'th smallest element is 5

**2) Program for Mean and median of an unsorted array**

import java.util.\*;

class GFG

{

// Function for calculating mean

public static double findMean(int a[], int n)

{

int sum = 0;

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

sum += a[i];

return (double)sum / (double)n;

}

// Function for calculating median

public static double findMedian(int a[], int n)

{

// First we sort the array

Arrays.sort(a);

// check for even case

if (n % 2 != 0)

return (double)a[n / 2];

return (double)(a[(n - 1) / 2] + a[n / 2]) / 2.0;

}

// Driver program

public static void main(String args[])

{

int a[] = { 1, 3, 4, 2, 7, 5, 8, 6 };

int n = a.length;

System.out.println("Mean = " + findMean(a, n));

System.out.println("Median = " + findMedian(a, n));

}

}

Output:

Mean = 4.5

Median = 4.5

**3) Minimum product of k integers in an array of positive Integers**

import java.util.PriorityQueue;

class GFG

{

public static int minProduct(int[] arr, int n, int k)

{

PriorityQueue<Integer> pq = new PriorityQueue<>();

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

pq.add(arr[i]);

int count = 0, ans = 1;

// One by one extract items

while(pq.isEmpty() == false && count < k)

{

ans = ans \* pq.element();

pq.remove();

count++;

}

return ans;

}

// Driver Code

public static void main(String[] args)

{

int arr[] = {198, 76, 544, 123, 154, 675};

int k = 2;

int n = arr.length;

System.out.print("Minimum product is " +

minProduct(arr, n, k));

}

}

Output:

Minimum product is 9348

**4) k largest(or smallest) elements in an array | added Min Heap method**

|  |
| --- |
| import java.util.Arrays;  import java.util.Collections;    class GFG {      public static void kLargest(Integer[] arr, int k)      {          // Sort the given array arr in reverse order          // This method doesn't work with primitive data          // types. So, instead of int, Integer type          // array will be used          Arrays.sort(arr, Collections.reverseOrder());            // Print the first kth largest elements          for (int i = 0; i < k; i++)              System.out.print(arr[i] + " ");      }        public static void main(String[] args)      {          Integer arr[] = new Integer[] { 1, 23, 12, 9,                                          30, 2, 50 };          int k = 3;          kLargest(arr, k);      }  } |

**Output:**

50 30 23

**Time complexity:** O(nlogn)

**5) Find all elements in array which have at-least two greater elements**

|  |
| --- |
| import java.util.\*;  import java.io.\*;    class GFG  {    static void findElements(int arr[],                              int n)  {      // Pick elements one by one      // and count greater elements.      // If count is more than 2,      // print that element.      for (int i = 0; i < n; i++)      {          int count = 0;            for (int j = 0; j < n; j++)              if (arr[j] > arr[i])                  count++;            if (count >= 2)          System.out.print(arr[i] + " ");      }  }    // Driver code  public static void main(String args[])  {      int arr[] = { 2, -6 ,3 , 5, 1};      int n = arr.length;      findElements(arr, n);  }  } |

**Output :**

2 -6 1

**Time Complexity :** O(n2)

**6) Median of Stream of Running Integers using STL**

|  |
| --- |
| import java.util.Collections;  import java.util.PriorityQueue;    public class MedianMaintain  {        // method to calculate med of stream      public static void printMedian(int[] a)      {            double med = a[0];            // max heap to store the smaller half elements          PriorityQueue<Integer> smaller = new PriorityQueue<>          (Collections.reverseOrder());            // min heap to store the greater half elements          PriorityQueue<Integer> greater = new PriorityQueue<>();            smaller.add(a[0]);          System.out.println(med);            // reading elements of stream one by one          /\* At any time we try to make heaps balanced and              their sizes differ by at-most 1. If heaps are              balanced,then we declare median as average of              min\_heap\_right.top() and max\_heap\_left.top()              If heaps are unbalanced,then median is defined              as the top element of heap of larger size \*/          for(int i = 1; i < a.length; i++)          {                int x = a[i];                // case1(left side heap has more elements)              if(smaller.size() > greater.size())              {                  if(x < med)                  {                      greater.add(smaller.remove());                      smaller.add(x);                  }                  else                      greater.add(x);                  med = (double)(smaller.peek() + greater.peek())/2;              }                // case2(both heaps are balanced)              else if(smaller.size() == greater.size())              {                  if(x < med)                  {                      smaller.add(x);                      med = (double)smaller.peek();                  }                  else                  {                      greater.add(x);                      med = (double)greater.peek();                  }              }                // case3(right side heap has more elements)              else              {                  if(x > med)                  {                      smaller.add(greater.remove());                      greater.add(x);                  }                  else                      smaller.add(x);                  med = (double)(smaller.peek() + greater.peek())/2;                }              System.out.println(med);          }      }        // Driver code      public static void main(String []args)      {            // stream of integers          int[] arr = new int[]{5, 15, 10, 20, 3};          printMedian(arr);      }  } |

**Output:**

5

10

10

12.5

10

**Time Complexity:** O(n Log n)

**7) K-th Largest Sum Contiguous Subarray**

|  |
| --- |
| import java.util.\*;    class KthLargestSumSubArray  {      // function to calculate kth largest      // element in contiguous subarray sum      static int kthLargestSum(int arr[], int n, int k)      {          // array to store predix sums          int sum[] = new int[n + 1];          sum[0] = 0;          sum[1] = arr[0];          for (int i = 2; i <= n; i++)              sum[i] = sum[i - 1] + arr[i - 1];            // priority\_queue of min heap          PriorityQueue<Integer> Q = new PriorityQueue<Integer> ();            // loop to calculate the contigous subarray          // sum position-wise          for (int i = 1; i <= n; i++)          {                // loop to traverse all positions that              // form contiguous subarray              for (int j = i; j <= n; j++)              {                  // calculates the contiguous subarray                  // sum from j to i index                  int x = sum[j] - sum[i - 1];                    // if queue has less then k elements,                  // then simply push it                  if (Q.size() < k)                      Q.add(x);                    else                  {                      // it the min heap has equal to                      // k elements then just check                      // if the largest kth element is                      // smaller than x then insert                      // else its of no use                      if (Q.peek() < x)                      {                          Q.poll();                          Q.add(x);                      }                  }              }          }            // the top element will be then kth          // largest element          return Q.poll();      }        // Driver Code      public static void main(String[] args)      {          int a[] = new int[]{ 10, -10, 20, -40 };          int n = a.length;          int k = 6;            // calls the function to find out the          // k-th largest sum          System.out.println(kthLargestSum(a, n, k));      }  } |

**Output:**

-10

**Time complexity:** O(n^2 log (k))

**8) K maximum sum combinations from two arrays**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG {        // function to display first K      // maximum sum combinations      static void KMaxCombinations(int A[], int B[], int N,                                                     int K)      {          // max heap.          PriorityQueue<Integer> pq =                              new PriorityQueue<Integer>(Collections.reverseOrder());            // insert all the possible          // combinations in max heap.          for (int i = 0; i < N; i++)              for (int j = 0; j < N; j++)                  pq.add(A[i] + B[j]);            // pop first N elements          // from max heap and          // display them.          int count = 0;            while (count < K)          {              System.out.println(pq.peek());              pq.remove();              count++;          }      }        public static void main (String[] args)      {          int A[] = { 4, 2, 5, 1 };          int B[] = { 8, 0, 5, 3 };          int N = A.length;          int K = 3;            NMaxCombinations(A, B, N, K);        }  } |

Output:

13

12

10

**Time Complexity :**O(N^2)

**9) K maximum sums of non-overlapping contiguous sub-arrays**

|  |
| --- |
| class GFG {        // Method to compute k maximum      // sub-array sums.      static void kmax(int arr[], int k, int n) {            // In each iteration it will give          // the ith maximum subarray sum.          for(int c = 0; c < k; c++)          {              // Kadane's algorithm.              int max\_so\_far = Integer.MIN\_VALUE;              int max\_here = 0;                // compute starting and ending              // index of each of the sub-array.              int start = 0, end = 0, s = 0;              for(int i = 0; i < n; i++)              {                  max\_here += arr[i];                  if (max\_so\_far < max\_here)                  {                      max\_so\_far = max\_here;                      start = s;                      end = i;                  }                  if (max\_here < 0)                      {                      max\_here = 0;                      s = i + 1;                  }              }                // Print out the result.              System.out.println("Maximum non-overlapping sub-arraysum" +                                  (c + 1) + ": " +  max\_so\_far +                                  ", starting index: " + start +                                  ", ending index: " + end + ".");                // Replace all elements of the maximum subarray              // by -infinity. Hence these places cannot form              // maximum sum subarray again.              for (int l = start; l <= end; l++)                  arr[l] = Integer.MIN\_VALUE;          }          System.out.println();      }        // Driver Program      public static void main(String[] args)      {          // Test case 1          int arr1[] = {4, 1, 1, -1, -3, -5,                              6, 2, -6, -2};          int k1 = 3;          int n1 = arr1.length;            // Function calling          kmax(arr1, k1, n1);            // Test case 2          int arr2[] = {5, 1, 2, -6, 2, -1, 3, 1};          int k2 = 2;          int n2 = arr2.length;            // Function calling          kmax(arr2, k2, n2);      }  } |

Output:

Maximum non-overlapping sub-array sum1: 8, starting index: 6, ending index: 7.

Maximum non-overlapping sub-array sum2: 6, starting index: 0, ending index: 2.

Maximum non-overlapping sub-array sum3: -1, starting index: 3, ending index: 3.

Maximum non-overlapping sub-array sum1: 8, starting index: 0, ending index: 2.

Maximum non-overlapping sub-array sum2: 5, starting index: 4, ending index: 7.

**Time Complexity:** The outer loop runs for k times and kadane’s algorithm in each iteration runs in linear time O(n). Hence the overall time complexity is O(k\*n).

**10) k smallest elements in same order using O(1) extra space**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;    public class GfG {      // Function to print smallest k numbers      // in arr[0..n-1]      public static void printSmall(int arr[], int n, int k)      {          // For each arr[i] find whether          // it is a part of n-smallest          // with insertion sort concept          for (int i = k; i < n; ++i) {              // Find largest from top n-element              int max\_var = arr[k - 1];              int pos = k - 1;              for (int j = k - 2; j >= 0; j--) {                  if (arr[j] > max\_var) {                      max\_var = arr[j];                      pos = j;                  }              }                // If largest is greater than arr[i]              // shift all element one place left              if (max\_var > arr[i]) {                  int j = pos;                  while (j < k - 1) {                      arr[j] = arr[j + 1];                      j++;                  }                  // make arr[k-1] = arr[i]                  arr[k - 1] = arr[i];              }          }          // print result          for (int i = 0; i < k; i++)              System.out.print(arr[i] + " ");      }        // Driver function      public static void main(String argc[])      {          int[] arr = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };          int n = 10;          int k = 5;          printSmall(arr, n, k);      }    } |

Output :

1 3 4 2 0

**11) k smallest elements in same order using O(1) extra space**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;    public class GfG {      // Function to print smallest k numbers      // in arr[0..n-1]      public static void printSmall(int arr[], int n, int k)      {          // For each arr[i] find whether          // it is a part of n-smallest          // with insertion sort concept          for (int i = k; i < n; ++i) {              // Find largest from top n-element              int max\_var = arr[k - 1];              int pos = k - 1;              for (int j = k - 2; j >= 0; j--) {                  if (arr[j] > max\_var) {                      max\_var = arr[j];                      pos = j;                  }              }                // If largest is greater than arr[i]              // shift all element one place left              if (max\_var > arr[i]) {                  int j = pos;                  while (j < k - 1) {                      arr[j] = arr[j + 1];                      j++;                  }                  // make arr[k-1] = arr[i]                  arr[k - 1] = arr[i];              }          }          // print result          for (int i = 0; i < k; i++)              System.out.print(arr[i] + " ");      }        // Driver function      public static void main(String argc[])      {          int[] arr = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };          int n = 10;          int k = 5;          printSmall(arr, n, k);      }    } |

Output :

1 3 4 2 0

**12) Find k pairs with smallest sums in two arrays**

|  |
| --- |
| import java.io.\*;    class KSmallestPair  {      // Function to find k pairs with least sum such      // that one elemennt of a pair is from arr1[] and      // other element is from arr2[]      static void kSmallestPair(int arr1[], int n1, int arr2[],                                              int n2, int k)      {          if (k > n1\*n2)          {              System.out.print("k pairs don't exist");              return ;          }            // Stores current index in arr2[] for          // every element of arr1[]. Initially          // all values are considered 0.          // Here current index is the index before          // which all elements are considered as          // part of output.          int index2[] = new int[n1];            while (k > 0)          {              // Initialize current pair sum as infinite              int min\_sum = Integer.MAX\_VALUE;              int min\_index = 0;                // To pick next pair, traverse for all              // elements of arr1[], for every element, find              // corresponding current element in arr2[] and              // pick minimum of all formed pairs.              for (int i1 = 0; i1 < n1; i1++)              {                  // Check if current element of arr1[] plus                  // element of array2 to be used gives                  // minimum sum                  if (index2[i1] < n2 &&                      arr1[i1] + arr2[index2[i1]] < min\_sum)                  {                      // Update index that gives minimum                      min\_index = i1;                        // update minimum sum                      min\_sum = arr1[i1] + arr2[index2[i1]];                  }              }                System.out.print("(" + arr1[min\_index] + ", " +                              arr2[index2[min\_index]]+ ") ");                index2[min\_index]++;              k--;          }      }        // Driver code      public static void main (String[] args)      {          int arr1[] = {1, 3, 11};          int n1 = arr1.length;            int arr2[] = {2, 4, 8};          int n2 = arr2.length;            int k = 4;          kSmallestPair( arr1, n1, arr2, n2, k);      }  } |

**Output:**

(1, 2) (1, 4) (3, 2) (3, 4)

**Time Complexity :**O(k\*n1)

**13) Absolute Difference of even and odd indexed elements in an Array**

|  |
| --- |
| public class GFG{        // Function to calculate absolute difference      static void EvenOddAbsoluteDifference(int arr[], int n)      {          int even = 0;          int odd = 0;            for (int i = 0; i < n; i++) {                // Loop to find even, odd absolute difference              if (i % 2 == 0)                  even = Math.abs(even - arr[i]);              else                  odd = Math.abs(odd - arr[i]);          }            System.out.println("Even Index absolute difference : " + even);          System.out.println("Odd Index absolute difference : " + odd);      }           // Driver Code       public static void main(String []args){                  int arr[] = { 1, 2, 3, 4, 5, 6 };                int n = arr.length;                 EvenOddAbsoluteDifference(arr, n);       }       } |

**Output:**

Even Index absolute difference : 3

Odd Index absolute difference : 4

**Time complexity :** O(n)

**14) Find the smallest missing number**

|  |
| --- |
| class SmallestMissing  {      int findFirstMissing(int array[], int start, int end)      {          if (start > end)              return end + 1;            if (start != array[start])              return start;            int mid = (start + end) / 2;            // Left half has all elements from 0 to mid          if (array[mid] == mid)              return findFirstMissing(array, mid+1, end);            return findFirstMissing(array, start, mid);      }        // Driver program to test the above function      public static void main(String[] args)      {          SmallestMissing small = new SmallestMissing();          int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 10};          int n = arr.length;          System.out.println("First Missing element is : "                  + small.findFirstMissing(arr, 0, n - 1));      }  } |

**Output:**

Smallest missing element is 8

**Note:**This method doesn’t work if there are duplicate elements in the array.

**Time Complexity:** O(Logn)

**15) Maximum sum such that no two elements are adjacent**

|  |
| --- |
| class MaximumSum  {      /\*Function to return max sum such that no two elements        are adjacent \*/      int FindMaxSum(int arr[], int n)      {          int incl = arr[0];          int excl = 0;          int excl\_new;          int i;            for (i = 1; i < n; i++)          {              /\* current max excluding i \*/              excl\_new = (incl > excl) ? incl : excl;                /\* current max including i \*/              incl = excl + arr[i];              excl = excl\_new;          }            /\* return max of incl and excl \*/          return ((incl > excl) ? incl : excl);      }        // Driver program to test above functions      public static void main(String[] args)      {          MaximumSum sum = new MaximumSum();          int arr[] = new int[]{5, 5, 10, 100, 10, 5};          System.out.println(sum.FindMaxSum(arr, arr.length));      }  } |

Output:

110

**Time Complexity:**O(n)

**16) Maximum and minimum of an array using minimum number of comparisons**

|  |
| --- |
| public class GFG {  /\* Class Pair is used to return two values from getMinMax() \*/      static class Pair {            int min;          int max;      }        static Pair getMinMax(int arr[], int n) {          Pair minmax = new  Pair();          int i;            /\*If there is only one element then return it as min and max both\*/          if (n == 1) {              minmax.max = arr[0];              minmax.min = arr[0];              return minmax;          }            /\* If there are more than one elements, then initialize min      and max\*/          if (arr[0] > arr[1]) {              minmax.max = arr[0];              minmax.min = arr[1];          } else {              minmax.max = arr[1];              minmax.min = arr[0];          }            for (i = 2; i < n; i++) {              if (arr[i] > minmax.max) {                  minmax.max = arr[i];              } else if (arr[i] < minmax.min) {                  minmax.min = arr[i];              }          }            return minmax;      }        /\* Driver program to test above function \*/      public static void main(String args[]) {          int arr[] = {1000, 11, 445, 1, 330, 3000};          int arr\_size = 6;          Pair minmax = getMinMax(arr, arr\_size);          System.out.printf("\nMinimum element is %d", minmax.min);          System.out.printf("\nMaximum element is %d", minmax.max);        }    } |

**Output:**

Minimum element is 1

Maximum element is 3000

Time Complexity: O(n)

**Range Queries :**

**Examples:**

**1) MO’s Algorithm (Query Square Root Decomposition)**

import java.util.\*;

// Class to represent a query range

class Query{

int L;

int R;

Query(int L, int R){

this.L = L;

this.R = R;

}

}

class MO{

// Prints sum of all query ranges. m is number of queries

// n is size of array a[].

static void queryResults(int a[], int n, ArrayList<Query> q, int m){

// Find block size

int block = (int) Math.sqrt(n);

// Sort all queries so that queries of same blocks

// are arranged together.

Collections.sort(q, new Comparator<Query>(){

// Function used to sort all queries so that all queries

// of the same block are arranged together and within a block,

// queries are sorted in increasing order of R values.

public int compare(Query x, Query y){

// Different blocks, sort by block.

if (x.L/block != y.L/block)

return (x.L < y.L ? -1 : 1);

// Same block, sort by R value

return (x.R < y.R ? -1 : 1);

}

});

// Initialize current L, current R and current sum

int currL = 0, currR = 0;

int currSum = 0;

// Traverse through all queries

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

{

// L and R values of current range

int L = q.get(i).L, R = q.get(i).R;

// Remove extra elements of previous range. For

// example if previous range is [0, 3] and current

// range is [2, 5], then a[0] and a[1] are subtracted

while (currL < L)

{

currSum -= a[currL];

currL++;

}

// Add Elements of current Range

while (currL > L)

{

currSum += a[currL-1];

currL--;

}

while (currR <= R)

{

currSum += a[currR];

currR++;

}

// Remove elements of previous range.  For example

// when previous range is [0, 10] and current range

// is [3, 8], then a[9] and a[10] are subtracted

while (currR > R+1)

{

currSum -= a[currR-1];

currR--;

}

// Print sum of current range

System.out.println("Sum of [" + L +

", " + R + "] is "  + currSum);

}

}

// Driver program

public static void main(String argv[]){

ArrayList<Query> q = new ArrayList<Query>();

q.add(new Query(0,4));

q.add(new Query(1,3));

q.add(new Query(2,4));

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

queryResults(a, a.length, q, q.size());

}

}

Output:

Sum of [1, 3] is 4

Sum of [0, 4] is 8

Sum of [2, 4] is 6

**2) Sqrt (or Square Root) Decomposition Technique**

import java.util.\*;

class GFG

{

static int MAXN = 10000;

static int SQRSIZE = 100;

static int []arr = new int[MAXN];             // original array

static int []block = new int[SQRSIZE];         // decomposed array

static int blk\_sz;                             // block size

// Time Complexity : O(1)

static void update(int idx, int val)

{

int blockNumber = idx / blk\_sz;

block[blockNumber] += val - arr[idx];

arr[idx] = val;

}

// Time Complexity : O(sqrt(n))

static int query(int l, int r)

{

int sum = 0;

while (l < r && l % blk\_sz != 0 && l != 0)

{

// traversing first block in range

sum += arr[l];

l++;

}

while (l+blk\_sz <= r)

{

// traversing completely

// overlapped blocks in range

sum += block[l / blk\_sz];

l += blk\_sz;

}

while (l <= r)

{

// traversing last block in range

sum += arr[l];

l++;

}

return sum;

}

// Fills values in input[]

static void preprocess(int input[], int n)

{

// initiating block pointer

int blk\_idx = -1;

// calculating size of block

blk\_sz = (int) Math.sqrt(n);

// building the decomposed array

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

{

arr[i] = input[i];

if (i % blk\_sz == 0)

{

// entering next block

// incementing block pointer

blk\_idx++;

}

block[blk\_idx] += arr[i];

}

}

// Driver code

public static void main(String[] args)

{

// We have used separate array for input because

// the purpose of this code is to explain SQRT

// decomposition in competitive programming where

// we have multiple inputs.

int input[] = {1, 5, 2, 4, 6, 1, 3, 5, 7, 10};

int n = input.length;

preprocess(input, n);

System.out.println("query(3, 8) : " +

query(3, 8));

System.out.println("query(1, 6) : " +

query(1, 6));

update(8, 0);

System.out.println("query(8, 8) : " +

query(8, 8));

}

}

Output:

query(3,8) : 26

query(1,6) : 21

query(8,8) : 0

**3) Range LCM Queries**

class GFG

{

static final int MAX = 1000;

// allocate space for tree

static int tree[] = new int[4 \* MAX];

// declaring the array globally

static int arr[] = new int[MAX];

// Function to return gcd of a and b

static int gcd(int a, int b) {

if (a == 0) {

return b;

}

return gcd(b % a, a);

}

// utility function to find lcm

static int lcm(int a, int b)

{

return a \* b / gcd(a, b);

}

// Function to build the segment tree

// Node starts beginning index

// of current subtree. start and end

// are indexes in arr[] which is global

static void build(int node, int start, int end)

{

// If there is only one element

// in current subarray

if (start == end)

{

tree[node] = arr[start];

return;

}

int mid = (start + end) / 2;

// build left and right segments

build(2 \* node, start, mid);

build(2 \* node + 1, mid + 1, end);

// build the parent

int left\_lcm = tree[2 \* node];

int right\_lcm = tree[2 \* node + 1];

tree[node] = lcm(left\_lcm, right\_lcm);

}

// Function to make queries for

// array range )l, r). Node is index

// of root of current segment in segment

// tree (Note that indexes in segment

// tree begin with 1 for simplicity).

// start and end are indexes of subarray

// covered by root of current segment.

static int query(int node, int start,

int end, int l, int r)

{

// Completely outside the segment, returning

// 1 will not affect the lcm;

if (end < l || start > r)

{

return 1;

}

// completely inside the segment

if (l <= start && r >= end)

{

return tree[node];

}

// partially inside

int mid = (start + end) / 2;

int left\_lcm = query(2 \* node, start, mid, l, r);

int right\_lcm = query(2 \* node + 1, mid + 1, end, l, r);

return lcm(left\_lcm, right\_lcm);

}

// Driver code

public static void main(String[] args)

{

//initialize the array

arr[0] = 5;

arr[1] = 7;

arr[2] = 5;

arr[3] = 2;

arr[4] = 10;

arr[5] = 12;

arr[6] = 11;

arr[7] = 17;

arr[8] = 14;

arr[9] = 1;

arr[10] = 44;

// build the segment tree

build(1, 0, 10);

// Now we can answer each query efficiently

// Print LCM of (2, 5)

System.out.println(query(1, 0, 10, 2, 5));

// Print LCM of (5, 10)

System.out.println(query(1, 0, 10, 5, 10));

// Print LCM of (0, 10)

System.out.println(query(1, 0, 10, 0, 10));

}

}

Output:

60

15708

78540

**4) GCDs of given index ranges in an array**

import java.io.\*;

public class Main

{

private static int[] st; // Array to store segment tree

/\* Function to construct segment tree from given array.

This function allocates memory for segment tree and

calls constructSTUtil() to fill the allocated memory \*/

public static int[] constructSegmentTree(int[] arr)

{

int height = (int)Math.ceil(Math.log(arr.length)/Math.log(2));

int size = 2\*(int)Math.pow(2, height)-1;

st = new int[size];

constructST(arr, 0, arr.length-1, 0);

return st;

}

// A recursive function that constructs Segment

// Tree for array[ss..se]. si is index of current

// node in segment tree st

public static int constructST(int[] arr, int ss,

int se, int si)

{

if (ss==se)

{

st[si] = arr[ss];

return st[si];

}

int mid = ss+(se-ss)/2;

st[si] = gcd(constructST(arr, ss, mid, si\*2+1),

constructST(arr, mid+1, se, si\*2+2));

return st[si];

}

// Function to find gcd of 2 numbers.

private static int gcd(int a, int b)

{

if (a < b)

{

// If b greater than a swap a and b

int temp = b;

b = a;

a = temp;

}

if (b==0)

return a;

return gcd(b,a%b);

}

//Finding The gcd of given Range

public static int findRangeGcd(int ss, int se, int[] arr)

{

int n = arr.length;

if (ss<0 || se > n-1 || ss>se)

throw new IllegalArgumentException("Invalid arguments");

return findGcd(0, n-1, ss, se, 0);

}

/\*  A recursive function to get gcd of given

range of array indexes. The following are parameters for

this function.

st    --> Pointer to segment tree

si --> Index of current node in the segment tree. Initially

0 is passed as root is always at index 0

ss & se  --> Starting and ending indexes of the segment

represented by current node, i.e., st[si]

qs & qe  --> Starting and ending indexes of query range \*/

public static int findGcd(int ss, int se, int qs, int qe, int si)

{

if (ss>qe || se < qs)

return 0;

if (qs<=ss && qe>=se)

return st[si];

int mid = ss+(se-ss)/2;

return gcd(findGcd(ss, mid, qs, qe, si\*2+1),

findGcd(mid+1, se, qs, qe, si\*2+2));

}

// Driver Code

public static void main(String[] args)throws IOException

{

int[] a = {2, 3, 6, 9, 5};

constructSegmentTree(a);

int l = 1; // Starting index of range.

int r = 3; //Last index of range.

System.out.print("GCD of the given range is: ");

System.out.print(findRangeGcd(l, r, a));

}

}

Output:

GCD of the given range is: 3

**5) Sparse Table**

|  |
| --- |
| import java.io.\*;    class GFG {        static int MAX =500;        // lookup[i][j] is going to store minimum      // value in arr[i..j]. Ideally lookup table      // size should not be fixed and should be      // determined using n Log n. It is kept      // constant to keep code simple.      static int [][]lookup = new int[MAX][MAX];        // Fills lookup array lookup[][] in bottom up manner.      static void buildSparseTable(int arr[], int n)      {            // Initialize M for the intervals with length 1          for (int i = 0; i < n; i++)              lookup[i][0] = arr[i];            // Compute values from smaller to bigger intervals          for (int j = 1; (1 << j) <= n; j++) {                // Compute minimum value for all intervals with              // size 2^j              for (int i = 0; (i + (1 << j) - 1) < n; i++) {                    // For arr[2][10], we compare arr[lookup[0][7]]                  // and arr[lookup[3][10]]                  if (lookup[i][j - 1] <                              lookup[i + (1 << (j - 1))][j - 1])                      lookup[i][j] = lookup[i][j - 1];                  else                      lookup[i][j] =                              lookup[i + (1 << (j - 1))][j - 1];              }          }      }        // Returns minimum of arr[L..R]      static int query(int L, int R)      {            // Find highest power of 2 that is smaller          // than or equal to count of elements in given          // range. For [2, 10], j = 3          int j = (int)Math.log(R - L + 1);            // Compute minimum of last 2^j elements with first          // 2^j elements in range.          // For [2, 10], we compare arr[lookup[0][3]] and          // arr[lookup[3][3]],          if (lookup[L][j] <= lookup[R - (1 << j) + 1][j])              return lookup[L][j];            else              return lookup[R - (1 << j) + 1][j];      }        // Driver program      public static void main (String[] args)      {          int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };          int n = a.length;            buildSparseTable(a, n);            System.out.println(query(0, 4));          System.out.println(query(4, 7));          System.out.println(query(7, 8));        }  } |

Output:

Minimum of [0, 4] is 0

Minimum of [4, 7] is 3

Minimum of [7, 8] is 12

So sparse table method supports query operation in O(1) time with O(n Log n) preprocessing time and O(n Log n) space.

**6) Range sum query using Sparse Table**

|  |
| --- |
| class GFG  {    // Because 2^17 is larger than 10^5  static int k = 16;    // Maximum value of array  static int N = 100000;    // k + 1 because we need  // to access table[r][k]  static long table[][] = new long[N][k + 1];    // it builds sparse table.  static void buildSparseTable(int arr[],                               int n)  {      for (int i = 0; i < n; i++)          table[i][0] = arr[i];        for (int j = 1; j <= k; j++)          for (int i = 0; i <= n - (1 << j); i++)              table[i][j] = table[i][j - 1] +              table[i + (1 << (j - 1))][j - 1];  }    // Returns the sum of the  // elements in the range L and R.  static long query(int L, int R)  {      // boundaries of next query,      // 0-indexed      long answer = 0;      for (int j = k; j >= 0; j--)      {          if (L + (1 << j) - 1 <= R)          {              answer = answer + table[L][j];                // instead of having L', we              // increment L directly              L += 1 << j;          }      }      return answer;  }    // Driver Code  public static void main(String args[])  {      int arr[] = { 3, 7, 2, 5, 8, 9 };      int n = arr.length;        buildSparseTable(arr, n);        System.out.println(query(0, 5));      System.out.println(query(3, 5));      System.out.println(query(2, 4));  }  }    // This code is contributed  // by Kirti\_Mangal |

**Output:**

34

22

15

This algorithm for answering queries with Sparse Table works in O(k), which is O(log(n)), because we choose minimal k such that 2^k+1 > n.

**7) Range Minimum Query (Square Root Decomposition and Sparse Table)**

|  |
| --- |
| import java.util.\*;    class GFG  {  static int MAX = 500;    // lookup[i][j] is going to store index of  // minimum value in arr[i..j]  static int [][]lookup = new int[MAX][MAX];    // Structure to represent a query range  static class Query  {      int L, R;        public Query(int L, int R)      {          this.L = L;          this.R = R;      }  };    // Fills lookup array lookup[n][n] for  // all possible values of query ranges  static void preprocess(int arr[], int n)  {      // Initialize lookup[][] for      // the intervals with length 1      for (int i = 0; i < n; i++)          lookup[i][i] = i;        // Fill rest of the entries in bottom up manner      for (int i = 0; i < n; i++)      {          for (int j = i + 1; j < n; j++)            // To find minimum of [0,4],          // we compare minimum of          // arr[lookup[0][3]] with arr[4].          if (arr[lookup[i][j - 1]] < arr[j])              lookup[i][j] = lookup[i][j - 1];          else              lookup[i][j] = j;      }  }    // Prints minimum of given m query  // ranges in arr[0..n-1]  static void RMQ(int arr[], int n,                  Query q[], int m)  {      // Fill lookup table for      // all possible input queries      preprocess(arr, n);        // One by one compute sum of all queries      for (int i = 0; i < m; i++)      {          // Left and right boundaries          // of current range          int L = q[i].L, R = q[i].R;            // Print sum of current query range          System.out.println("Minimum of [" + L +                             ", " + R + "] is " +                              arr[lookup[L][R]]);      }  }    // Driver Code  public static void main(String[] args)  {      int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};      int n = a.length;      Query q[] = {new Query(0, 4),                   new Query(4, 7),                   new Query(7, 8)};      int m = q.length;      RMQ(a, n, q, m);  }  }    // This code is contributed by 29AjayKumar |

**Output:**

Minimum of [0, 4] is 0

Minimum of [4, 7] is 3

Minimum of [7, 8] is 12

This approach supports query in**O(1)**, but preprocessing takes **O(n2)**time. Also, this approach needs**O(n2)** extra space which may become huge for large input arrays.

**8) Range Queries for Frequencies of array elements**

|  |
| --- |
| class GFG {        // Returns count of element in arr[left-1..right-1]      public static int findFrequency(int arr[], int n,                                  int left, int right,                                        int element)      {          int count = 0;          for (int i = left - 1; i < right; ++i)              if (arr[i] == element)                  ++count;          return count;      }        /\* Driver program to test above function \*/      public static void main(String[] args)      {          int arr[] = {2, 8, 6, 9, 8, 6, 8, 2, 11};          int n = arr.length;            // Print frequency of 2 from position 1 to 6          System.out.println("Frequency of 2 from 1 to 6 = " +               findFrequency(arr, n, 1, 6, 2));            // Print frequency of 8 from position 4 to 9          System.out.println("Frequency of 8 from 4 to 9 = " +               findFrequency(arr, n, 4, 9, 8));        }    } |

**Output:**

Frequency of 2 from 1 to 6 = 1

Frequency of 8 from 4 to 9 = 2

**Time complexity** of this approach is O(right – left + 1) or O(n)

**9) Constant time range add operation on an array**

|  |
| --- |
| import java.io.\*;    class GFG  {      // Utility method to add value val,      // to range [lo, hi]      static void add(int arr[], int N, int lo,                             int hi, int val)      {          arr[lo] += val;          if (hi != N - 1)             arr[hi + 1] -= val;      }        // Utility method to get actual array from      // operation array      static void updateArray(int arr[], int N)      {          // convert array into prefix sum array          for (int i = 1; i < N; i++)              arr[i] += arr[i - 1];      }        // method to print final updated array      static void printArr(int arr[], int N)      {          updateArray(arr, N);          for (int i = 0; i < N; i++)              System.out.print(""+arr[i]+" ");          System.out.print("\n");      }        // Driver code to test above methods      public static void main (String[] args)      {          int N = 6;            int arr[] = new int[N];            // Range add Queries          add(arr, N, 0, 2, 100);          add(arr, N, 1, 5, 100);          add(arr, N, 2, 3, 100);            printArr(arr, N);      }  } |

**Output:**

100 200 300 200 100 100

**10) Queries for GCD of all numbers of an array except elements in a given range**

|  |
| --- |
| import java.util.\*;    class GFG {    // Calculating GCD using euclid algorithm  static int GCD(int a, int b)  {      if (b == 0)      return a;      return GCD(b, a % b);  }    // Filling the prefix and suffix array  static void FillPrefixSuffix(int prefix[],            int arr[], int suffix[], int n)  {      // Filling the prefix array following relation      // prefix(i) = GCD(prefix(i-1), arr(i))      prefix[0] = arr[0];      for (int i = 1; i < n; i++)      prefix[i] = GCD(prefix[i - 1], arr[i]);        // Filling the suffix array folowing the      // relation suffix(i) = GCD(suffix(i+1), arr(i))      suffix[n - 1] = arr[n - 1];        for (int i = n - 2; i >= 0; i--)      suffix[i] = GCD(suffix[i + 1], arr[i]);  }    // To calculate gcd of the numbers outside range  static int GCDoutsideRange(int l, int r,        int prefix[], int suffix[], int n) {        // If l=0, we need to tell GCD of numbers      // from r+1 to n      if (l == 0)      return suffix[r + 1];        // If r=n-1 we need to return the gcd of      // numbers from 1 to l      if (r == n - 1)      return prefix[l - 1];      return GCD(prefix[l - 1], suffix[r + 1]);  }    // Driver code  public static void main(String[] args) {      int arr[] = {2, 6, 9};      int n = arr.length;      int prefix[] = new int[n];      int suffix[] = new int[n];      FillPrefixSuffix(prefix, arr, suffix, n);        int l = 0, r = 0;      System.out.println(GCDoutsideRange               (l, r, prefix, suffix, n));        l = 1;      r = 1;      System.out.println(GCDoutsideRange               (l, r, prefix, suffix, n));        l = 1;      r = 2;      System.out.println(GCDoutsideRange               (l, r, prefix, suffix, n));  }  } |

**Output:**

3

1

2

**11) Count elements which divide all numbers in range L-R**

|  |
| --- |
| import java.io.\*;    class GFG  {    // function to count element  // Time complexity O(n^2) worst case  static int answerQuery(int a[], int n,                         int l, int r)  {      // answer for query      int count = 0;        // 0 based index      l = l - 1;        // iterate for all elements      for (int i = l; i < r; i++)      {          int element = a[i];          int divisors = 0;            // check if the element divides          // all numbers in range          for (int j = l; j < r; j++)          {              // no of elements              if (a[j] % a[i] == 0)                  divisors++;              else                  break;          }            // if all elements are divisible by a[i]          if (divisors == (r - l))              count++;      }        // answer for every query      return count;  }    // Driver Code  public static void main (String[] args)  {      int a[] = { 1, 2, 3, 5 };      int n = a.length;        int l = 1, r = 4;      System.out.println( answerQuery(a, n, l, r));        l = 2; r = 4;      System.out.println( answerQuery(a, n, l, r));  }  } |

**Output:**

1

0

**12) Number whose sum of XOR with given array range is maximum**

|  |
| --- |
| import java.lang.Math;    class GFG {        private static final int MAX = 2147483647;      static int[][] one = new int[100001][32];        // Function to make prefix array which counts      // 1's of each bit up to that number      static void make\_prefix(int A[], int n)      {          for (int j = 0; j < 32; j++)              one[0][j] = 0;            // Making a prefix array which sums          // number of 1's up to that position          for (int i = 1; i <= n; i++)          {              int a = A[i - 1];              for (int j = 0; j < 32; j++)              {                  int x = (int)Math.pow(2, j);                    // If j-th bit of a number is set then                  // add one to previously counted 1's                  if ((a & x) != 0)                      one[i][j] = 1 + one[i - 1][j];                  else                      one[i][j] = one[i - 1][j];              }          }      }        // Function to find X      static int Solve(int L, int R)      {          int l = L, r = R;          int tot\_bits = r - l + 1;            // Initially taking maximum          // value all bits 1          int X = MAX;            // Iterating over each bit          for (int i = 0; i < 31; i++)          {                // get 1's at ith bit between the range              // L-R by subtracting 1's till              // Rth number - 1's till L-1th number              int x = one[r][i] - one[l - 1][i];                // If 1's are more than or equal to 0's              // then unset the ith bit from answer              if (x >= tot\_bits - x)              {                  int ith\_bit = (int)Math.pow(2, i);                    // Set ith bit to 0 by                  // doing Xor with 1                  X = X ^ ith\_bit;              }          }          return X;      }        // Driver program      public static void main(String[] args)      {          // Taking inputs          int n = 5, q = 3;          int A[] = { 210, 11, 48, 22, 133 };          int L[] = { 1, 4, 2 }, R[] = { 3, 14, 4 };            make\_prefix(A, n);            for (int j = 0; j < q; j++)              System.out.println(Solve(L[j], R[j]));      }  } |

**Output :**

2147483629

2147483647

2147483629

**13) Array range queries for searching an element**

|  |
| --- |
| import java.util.\*;    class GFG  {    // Structure to represent a query range  static class Query  {      int L, R, X;        public Query(int L, int R, int X)      {          this.L = L;          this.R = R;          this.X = X;      }  };    static int maxn = 100;    static int []root = new int[maxn];    // Find the root of the group containing  // the element at index x  static int find(int x)  {      if(x == root[x])          return x;      else          return root[x] = find(root[x]);  }    // merge the two groups containing elements  // at indices x and y into one group  static void uni(int x, int y)  {      int p = find(x), q = find(y);      if (p != q)      {          root[p] = root[q];      }  }    static void initialize(int a[], int n,                         Query q[], int m)  {      // make n subsets with every      // element as its root      for (int i = 0; i < n; i++)          root[i] = i;        // consecutive elements equal in value are      // merged into one single group      for (int i = 1; i < n; i++)          if (a[i] == a[i - 1])              uni(i, i - 1);  }    // Driver code  public static void main(String args[])  {      int a[] = { 1, 1, 5, 4, 5 };      int n = a.length;      Query q[] = { new Query(0, 2, 2 ),                    new Query( 1, 4, 1 ),                    new Query( 2, 4, 5 ) };      int m = q.length;      initialize(a, n, q, m);        for (int i = 0; i < m; i++)      {          int flag = 0;          int l = q[i].L, r = q[i].R, x = q[i].X;          int p = r;            while (p >= l)          {                // check if the current element in              // consideration is equal to x or not              // if it is equal, then x exists in the range              if (a[p] == x)              {                  flag = 1;                  break;              }              p = find(p) - 1;          }            // Print if x exists or not          if (flag != 0)              System.out.println(x + " exists between [" +                                 l + ", " + r + "] ");          else              System.out.println(x + " does not exist between [" +                                 l + ", " + r + "] ");      }  }  } |

**Output:**

2 does not exist between [0, 2]

1 exists between [1, 4]

5 exists between [2, 4]

**14) Array range queries for elements with frequency same as value**

|  |
| --- |
| import java.util.HashMap;  import java.util.Map;    class GFG  {      /\* Returns the count of number x with  frequency x in the subarray from  start to end \*/  static int solveQuery(int start, int end, int arr[])  {      // map for frequency of elements      Map<Integer,Integer> mp = new HashMap<>();        // store frequency of each element      // in arr[start; end]      for (int i = start; i <= end; i++)          mp.put(arr[i],mp.get(arr[i]) == null?1:mp.get(arr[i])+1);        // Count elements with same frequency      // as value      int count = 0;      for (Map.Entry<Integer,Integer> entry : mp.entrySet())          if (entry.getKey() == entry.getValue())              count++;      return count;  }    // Driver code  public static void main(String[] args)  {      int A[] = { 1, 2, 2, 3, 3, 3 };      int n = A.length;        // 2D array of queries with 2 columns      int [][]queries = { { 0, 1 },                          { 1, 1 },                          { 0, 2 },                          { 1, 3 },                          { 3, 5 },                          { 0, 5 } };        // calculating number of queries      int q = queries.length;        for (int i = 0; i < q; i++)      {          int start = queries[i][0];          int end = queries[i][1];          System.out.println("Answer for Query " + (i + 1)              + " = " + solveQuery(start,              end, A));      }  }  } |

**Output:**

Answer for Query 1 = 1

Answer for Query 2 = 0

Answer for Query 3 = 2

Answer for Query 4 = 1

Answer for Query 5 = 1

Answer for Query 6 = 3

Time Complexity of this method is O(Q \* N)

**15) Number of indexes with equal elements in given range**

|  |
| --- |
| class GFG {        // function that answers every query      // in O(r-l)      static int answer\_query(int a[], int n,                                int l, int r)      {            // traverse from l to r and count          // the required indexes          int count = 0;          for (int i = l; i < r; i++)              if (a[i] == a[i + 1])                  count += 1;            return count;      }        // Driver Code      public static void main(String[] args)      {          int a[] = {1, 2, 2, 2, 3, 3, 4, 4, 4};          int n = a.length;            // 1-st query          int L, R;          L = 1;          R = 8;            System.out.println(                     answer\_query(a, n, L, R));            // 2nd query          L = 0;          R = 4;          System.out.println(                    answer\_query(a, n, L, R));      }  } |

**Output :**

5

2

**Time Complexity :** O(R – L) for every query

**16) Total numbers with no repeated digits in a range**

|  |
| --- |
| import java.util.LinkedHashSet;    class GFG  {  // Function to check if the given  // number has repeated digit or not  static int repeated\_digit(int n)  {      LinkedHashSet<Integer> s = new LinkedHashSet<>();        // Traversing through each digit      while (n != 0)      {          int d = n % 10;            // if the digit is present          // more than once in the          // number          if (s.contains(d))          {              // return 0 if the number              // has repeated digit              return 0;          }          s.add(d);          n = n / 10;      }        // return 1 if the number has      // no repeated digit      return 1;  }    // Function to find total number  // in the given range which has  // no repeated digit  static int calculate(int L, int R)  {      int answer = 0;        // Traversing through the range      for (int i = L; i < R + 1; ++i)      {            // Add 1 to the answer if i has          // no repeated digit else 0          answer = answer + repeated\_digit(i);      }        return answer;  }    // Driver Code  public static void main(String[] args)  {      int L = 1, R = 100;        // Calling the calculate      System.out.println(calculate(L, R));  }  } |

**Output:**

90

This method will answer each query in **O( N )** time.

**17) Difference Array | Range update query in O(1)**

|  |
| --- |
| class GFG {        // Creates a diff array D[] for A[] and returns      // it after filling initial values.      static void initializeDiffArray(int A[], int D[])      {            int n = A.length;            D[0] = A[0];          D[n] = 0;          for (int i = 1; i < n; i++)              D[i] = A[i] - A[i - 1];      }        // Does range update      static void update(int D[], int l, int r, int x)      {          D[l] += x;          D[r + 1] -= x;      }        // Prints updated Array      static int printArray(int A[], int D[])      {          for (int i = 0; i < A.length; i++) {                if (i == 0)                  A[i] = D[i];                // Note that A[0] or D[0] decides              // values of rest of the elements.              else                  A[i] = D[i] + A[i - 1];                System.out.print(A[i] + " ");          }            System.out.println();            return 0;      }        // Driver Code      public static void main(String[] args)      {          // Array to be updated          int A[] = { 10, 5, 20, 40 };          int n = A.length;          // Create and fill difference Array          // We use one extra space because          // update(l, r, x) updates D[r+1]          int D[] = new int[n + 1];          initializeDiffArray(A, D);            // After below update(l, r, x), the          // elements should become 20, 15, 20, 40          update(D, 0, 1, 10);          printArray(A, D);            // After below updates, the          // array should become 30, 35, 70, 60          update(D, 1, 3, 20);          update(D, 2, 2, 30);            printArray(A, D);      }  } |

Output:

20 15 20 40

20 35 70 60

# 

**Optimization Problems :**

**Examples:**

**1) Maximum profit by buying and selling a share at most twice**

class Profit

{

// Returns maximum profit with two transactions on a given

// list of stock prices, price[0..n-1]

static int maxProfit(int price[], int n)

{

// Create profit array and initialize it as 0

int profit[] = new int[n];

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

profit[i] = 0;

/\* Get the maximum profit with only one transaction

allowed. After this loop, profit[i] contains maximum

profit from price[i..n-1] using at most one trans. \*/

int max\_price = price[n-1];

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

{

// max\_price has maximum of price[i..n-1]

if (price[i] > max\_price)

max\_price = price[i];

// we can get profit[i] by taking maximum of:

// a) previous maximum, i.e., profit[i+1]

// b) profit by buying at price[i] and selling at

//    max\_price

profit[i] = Math.max(profit[i+1], max\_price-price[i]);

}

/\* Get the maximum profit with two transactions allowed

After this loop, profit[n-1] contains the result \*/

int min\_price = price[0];

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

{

// min\_price is minimum price in price[0..i]

if (price[i] < min\_price)

min\_price = price[i];

// Maximum profit is maximum of:

// a) previous maximum, i.e., profit[i-1]

// b) (Buy, Sell) at (min\_price, price[i]) and add

//    profit of other trans. stored in profit[i]

profit[i] = Math.max(profit[i-1], profit[i] +

(price[i]-min\_price) );

}

int result = profit[n-1];

return result;

}

public static void main(String args[])

{

int price[] = {2, 30, 15, 10, 8, 25, 80};

int n = price.length;

System.out.println("Maximum Profit = "+ maxProfit(price, n));

}

Output:

Maximum Profit = 100

**2) Find the minimum distance between two numbers**

class MinimumDistance

{

int minDist(int arr[], int n, int x, int y)

{

int i, j;

int min\_dist = Integer.MAX\_VALUE;

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

{

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

{

if ((x == arr[i] && y == arr[j]

|| y == arr[i] && x == arr[j])

&& min\_dist > Math.abs(i - j))

min\_dist = Math.abs(i - j);

}

}

return min\_dist;

}

public static void main(String[] args)

{

MinimumDistance min = new MinimumDistance();

int arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3};

int n = arr.length;

int x = 3;

int y = 6;

System.out.println("Minimum distance between " + x + " and " + y

+ " is " + min.minDist(arr, n, x, y));

}

}

Output:

Minimum distance between 3 and 6 is 4

**3) Minimum number of jumps to reach end**

import java.util.\*;

import java.io.\*;

class GFG {

// Returns minimum number of

// jumps to reach arr[h] from arr[l]

static int minJumps(int arr[], int l, int h)

{

// Base case: when source

// and destination are same

if (h == l)

return 0;

// When nothing is reachable

// from the given source

if (arr[l] == 0)

return Integer.MAX\_VALUE;

// Traverse through all the points

// reachable from arr[l]. Recursively

// get the minimum number of jumps

// needed to reach arr[h] from these

// reachable points.

int min = Integer.MAX\_VALUE;

for (int i = l + 1; i <= h && i <= l + arr[l]; i++) {

int jumps = minJumps(arr, i, h);

if (jumps != Integer.MAX\_VALUE && jumps + 1 < min)

min = jumps + 1;

}

return min;

}

// Driver code

public static void main(String args[])

{

int arr[] = { 1, 3, 6, 3, 2, 3, 6, 8, 9, 5 };

int n = arr.length;

System.out.print("Minimum number of jumps to reach end is "

+ minJumps(arr, 0, n - 1));

}

}

Output:

Minimum number of jumps to reach end is 4

**4) Largest Sum Contiguous Subarray**

|  |
| --- |
| import java.io.\*;  // Java program to print largest contiguous array sum  import java.util.\*;    class Kadane  {      public static void main (String[] args)      {          int [] a = {-2, -3, 4, -1, -2, 1, 5, -3};          System.out.println("Maximum contiguous sum is " +                                         maxSubArraySum(a));      }        static int maxSubArraySum(int a[])      {          int size = a.length;          int max\_so\_far = Integer.MIN\_VALUE, max\_ending\_here = 0;            for (int i = 0; i < size; i++)          {              max\_ending\_here = max\_ending\_here + a[i];              if (max\_so\_far < max\_ending\_here)                  max\_so\_far = max\_ending\_here;              if (max\_ending\_here < 0)                  max\_ending\_here = 0;          }          return max\_so\_far;      }  } |

**Output:**

Maximum contiguous sum is 7

**5) Find the subarray with least average**

|  |
| --- |
| class Test {        static int arr[] = new int[] { 3, 7, 90, 20, 10, 50, 40 };        // Prints beginning and ending indexes of subarray      // of size k with minimum average      static void findMinAvgSubarray(int n, int k)      {          // k must be smaller than or equal to n          if (n < k)              return;            // Initialize beginning index of result          int res\_index = 0;            // Compute sum of first subarray of size k          int curr\_sum = 0;          for (int i = 0; i < k; i++)              curr\_sum += arr[i];            // Initialize minimum sum as current sum          int min\_sum = curr\_sum;            // Traverse from (k+1)'th element to n'th element          for (int i = k; i < n; i++)          {              // Add current item and remove first              // item of previous subarray              curr\_sum += arr[i] - arr[i - k];                // Update result if needed              if (curr\_sum < min\_sum) {                  min\_sum = curr\_sum;                  res\_index = (i - k + 1);              }          }            System.out.println("Subarray between [" +                              res\_index + ", " + (res\_index + k - 1) +                              "] has minimum average");      }        // Driver method to test the above function      public static void main(String[] args)      {          int k = 3; // Subarray size          findMinAvgSubarray(arr.length, k);      }  } |

Output:

Subarray between [3, 5] has minimum average

Time Complexity: O(n)

**6) Minimize the maximum difference between the heights**

|  |
| --- |
| import java.util.\*;    class GFG {        // Modifies the array by subtracting/adding      // k to every element such that the difference      // between maximum and minimum is minimized      static int getMinDiff(int arr[], int n, int k)      {          if (n == 1)          return 0;            // Sort all elements          Arrays.sort(arr);            // Initialize result          int ans = arr[n-1] - arr[0];            // Handle corner elements          int small = arr[0] + k;          int big = arr[n-1] - k;          int temp = 0;            if (small > big)          {              temp = small;              small = big;              big = temp;          }            // Traverse middle elements          for (int i = 1; i < n-1; i ++)          {              int subtract = arr[i] - k;              int add = arr[i] + k;                // If both subtraction and addition              // do not change diff              if (subtract >= small || add <= big)                  continue;                // Either subtraction causes a smaller              // number or addition causes a greater              // number. Update small or big using              // greedy approach (If big - subtract              // causes smaller diff, update small              // Else update big)              if (big - subtract <= add - small)                  small = subtract;              else                  big = add;          }            return Math.min(ans, big - small);      }        // Driver function to test the above function      public static void main(String[] args)      {          int arr[] = {4, 6};          int n = arr.length;          int k = 10;          System.out.println("Maximum difference is "+                              getMinDiff(arr, n, k));      }  } |

**Output :**

Maximum difference is 2

**Time Complexity:** O(n Log n)

**7) Maximum Sum Increasing Subsequence**

|  |
| --- |
| class GFG  {      /\* maxSumIS() returns the      maximum sum of increasing      subsequence in arr[] of size n \*/      static int maxSumIS(int arr[], int n)      {          int i, j, max = 0;          int msis[] = new int[n];            /\* Initialize msis values             for all indexes \*/          for (i = 0; i < n; i++)              msis[i] = arr[i];            /\* Compute maximum sum values             in bottom up manner \*/          for (i = 1; i < n; i++)              for (j = 0; j < i; j++)                  if (arr[i] > arr[j] &&                      msis[i] < msis[j] + arr[i])                      msis[i] = msis[j] + arr[i];            /\* Pick maximum of all             msis values \*/          for (i = 0; i < n; i++)              if (max < msis[i])                  max = msis[i];            return max;      }        // Driver code      public static void main(String args[])      {          int arr[] = new int[]{1, 101, 2, 3, 100, 4, 5};          int n = arr.length;          System.out.println("Sum of maximum sum "+                              "increasing subsequence is "+                                maxSumIS(arr, n));      }  } |

**Output :**

Sum of maximum sum increasing subsequence is 106

Time Complexity: O(n^2)

**8)Smallest subarray with sum greater than a given value**

|  |
| --- |
| class SmallestSubArraySum  {      // Returns length of smallest subarray with sum greater than x.      // If there is no subarray with given sum, then returns n+1      static int smallestSubWithSum(int arr[], int n, int x)      {          //  Initilize length of smallest subarray as n+1          int min\_len = n + 1;            // Pick every element as starting point          for (int start = 0; start < n; start++)          {              // Initialize sum starting with current start              int curr\_sum = arr[start];                // If first element itself is greater              if (curr\_sum > x)                  return 1;                // Try different ending points for curremt start              for (int end = start + 1; end < n; end++)              {                  // add last element to current sum                  curr\_sum += arr[end];                    // If sum becomes more than x and length of                  // this subarray is smaller than current smallest                  // length, update the smallest length (or result)                  if (curr\_sum > x && (end - start + 1) < min\_len)                      min\_len = (end - start + 1);              }          }          return min\_len;      }        // Driver program to test above functions      public static void main(String[] args)      {          int arr1[] = {1, 4, 45, 6, 10, 19};          int x = 51;          int n1 = arr1.length;          int res1 = smallestSubWithSum(arr1, n1, x);          if (res1 == n1+1)             System.out.println("Not Possible");          else             System.out.println(res1);              int arr2[] = {1, 10, 5, 2, 7};          int n2 = arr2.length;          x = 9;          int res2 = smallestSubWithSum(arr2, n2, x);          if (res2 == n2+1)             System.out.println("Not Possible");          else             System.out.println(res2);            int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};          int n3 = arr3.length;          x = 280;          int res3 = smallestSubWithSum(arr3, n3, x);          if (res3 == n3+1)             System.out.println("Not Possible");          else             System.out.println(res3);      }  } |

**Output:**

3

1

4

**Time Complexity:**Time complexity of the above approach is clearly O(n2).

**9) Find maximum average subarray of k length**

|  |
| --- |
| import java .io.\*;    class GFG {        // Returns beginning index      // of maximum average      // subarray of length 'k'      static int findMaxAverage(int []arr,                             int n, int k)      {            // Check if 'k' is valid          if (k > n)              return -1;            // Create and fill array          // to store cumulative          // sum. csum[i] stores          // sum of arr[0] to arr[i]          int []csum = new int[n];            csum[0] = arr[0];          for (int i = 1; i < n; i++)          csum[i] = csum[i - 1] + arr[i];            // Initialize max\_sm as          // sum of first subarray          int max\_sum = csum[k - 1],                      max\_end = k - 1;            // Find sum of other          // subarrays and update          // max\_sum if required.          for (int i = k; i < n; i++)          {              int curr\_sum = csum[i] -                      csum[i - k];              if (curr\_sum > max\_sum)              {                  max\_sum = curr\_sum;                  max\_end = i;              }          }            // To avoid memory leak          //delete [] csum;            // Return starting index          return max\_end - k + 1;      }        // Driver Code      static public void main (String[] args)      {          int []arr = {1, 12, -5, -6, 50, 3};          int k = 4;          int n = arr.length;            System.out.println("The maximum "            + "average subarray of length "                  + k + " begins at index "              + findMaxAverage(arr, n, k));      }  } |

Output:

The maximum average subarray of length 4 begins at index 1

Time Complexity of above solution is O(n),

**10) Count minimum steps to get the given desired array**

|  |
| --- |
| class Test  {      static int arr[] = new int[]{16, 16, 16} ;        // Returns count of minimum operations to convert a      // zero array to arr array with increment and      // doubling operations.      // This function computes count by doing reverse      // steps, i.e., convert arr to zero array.      static int countMinOperations(int n)      {          // Initialize result (Count of minimum moves)          int result = 0;            // Keep looping while all elements of arr          // don't become 0.          while (true)          {              // To store count of zeroes in current              // arr array              int zero\_count = 0;                int i;  // To find first odd element              for (i=0; i<n; i++)              {                  // If odd number found                  if (arr[i] % 2 == 1)                      break;                    // If 0, then increment zero\_count                  else if (arr[i] == 0)                      zero\_count++;              }                // All numbers are 0              if (zero\_count == n)                return result;                // All numbers are even              if (i == n)              {                  // Divide the whole array by 2                  // and increment result                  for (int j=0; j<n; j++)                     arr[j] = arr[j]/2;                  result++;              }                // Make all odd numbers even by subtracting              // one and increment result.              for (int j=i; j<n; j++)              {                 if (arr[j] %2 == 1)                 {                    arr[j]--;                    result++;                 }              }          }      }        // Driver method to test the above function      public static void main(String[] args) {            System.out.println("Minimum number of steps required to \n" +                              "get the given target array is "+                                   countMinOperations(arr.length));        }  } |

**Output :**

Minimum number of steps required to

get the given target array is 7

**11) Find minimum number of merge operations to make an array palindrome**

|  |
| --- |
| class GFG  {      // Returns minimum number of count operations      // required to make arr[] palindrome      static int findMinOps(int[] arr, int n)      {          int ans = 0; // Initialize result            // Start from two corners          for (int i=0,j=n-1; i<=j;)          {              // If corner elements are same,              // problem reduces arr[i+1..j-1]              if (arr[i] == arr[j])              {                  i++;                  j--;              }                // If left element is greater, then              // we merge right two elements              else if (arr[i] > arr[j])              {                  // need to merge from tail.                  j--;                  arr[j] += arr[j+1] ;                  ans++;              }                // Else we merge left two elements              else              {                  i++;                  arr[i] += arr[i-1];                  ans++;              }          }            return ans;      }        // Driver method to test the above function      public static void main(String[] args)      {          int arr[] = new int[]{1, 4, 5, 9, 1} ;          System.out.println("Count of minimum operations is "+                                  findMinOps(arr, arr.length));        }  } |

**Output :**

Count of minimum operations is 1

Time complexity for the given program is : O(n)

**12) Find the smallest positive integer value that cannot be represented as sum of any subset of a given**

**array**

|  |
| --- |
| class FindSmallestInteger  {      // Returns the smallest number that cannot be represented as sum      // of subset of elements from set represented by sorted array arr[0..n-1]      int findSmallest(int arr[], int n)      {          int res = 1; // Initialize result            // Traverse the array and increment 'res' if arr[i] is          // smaller than or equal to 'res'.          for (int i = 0; i < n && arr[i] <= res; i++)              res = res + arr[i];            return res;      }        // Driver program to test above functions      public static void main(String[] args)      {          FindSmallestInteger small = new FindSmallestInteger();          int arr1[] = {1, 3, 4, 5};          int n1 = arr1.length;          System.out.println(small.findSmallest(arr1, n1));            int arr2[] = {1, 2, 6, 10, 11, 15};          int n2 = arr2.length;          System.out.println(small.findSmallest(arr2, n2));            int arr3[] = {1, 1, 1, 1};          int n3 = arr3.length;          System.out.println(small.findSmallest(arr3, n3));            int arr4[] = {1, 1, 3, 4};          int n4 = arr4.length;          System.out.println(small.findSmallest(arr4, n4));        }  } |

Output:

2

4

5

10

Time Complexity of above program is O(n).

**13) Size of The Subarray With Maximum Sum**

|  |
| --- |
| class GFG {        static int maxSubArraySum(int a[], int size)      {          int max\_so\_far = Integer.MIN\_VALUE,          max\_ending\_here = 0,start = 0,          end = 0, s = 0;            for (int i = 0; i < size; i++)          {              max\_ending\_here += a[i];                if (max\_so\_far < max\_ending\_here)              {                  max\_so\_far = max\_ending\_here;                  start = s;                  end = i;              }                if (max\_ending\_here < 0)              {                  max\_ending\_here = 0;                  s = i + 1;              }          }          return (end - start + 1);      }        // Driver code      public static void main(String[] args)      {          int a[] = { -2, -3, 4, -1, -2, 1, 5, -3 };          int n = a.length;          System.out.println(maxSubArraySum(a, n));      }  } |

**Output :**

5

**14) Find minimum difference between any two elements**

|  |
| --- |
| class GFG  {      // Returns minimum difference between any pair      static int findMinDiff(int[] arr, int n)      {          // Initialize difference as infinite          int diff = Integer.MAX\_VALUE;            // Find the min diff by comparing difference          // of all possible pairs in given array          for (int i=0; i<n-1; i++)              for (int j=i+1; j<n; j++)                  if (Math.abs((arr[i] - arr[j]) )< diff)                      diff = Math.abs((arr[i] - arr[j]));            // Return min diff          return diff;      }        // Driver method to test the above function      public static void main(String[] args)      {          int arr[] = new int[]{1, 5, 3, 19, 18, 25};          System.out.println("Minimum difference is "+                                findMinDiff(arr, arr.length));        }  } |

**Output :**

Minimum difference is 1

**15) Space optimization using bit manipulations**

|  |
| --- |
| import java.lang.\*;    class GFG {        // Driver code      public static void main(String[] args)      {          int a = 2, b = 10;          int size = Math.abs(b - a) + 1;          int array[] = new int[size];            // Iterate through a to b, If          // it is a multiple of 2 or 5          // Mark index in array as 1          for (int i = a; i <= b; i++)              if (i % 2 == 0 || i % 5 == 0)                  array[i - a] = 1;            System.out.println("MULTIPLES of 2"                                + " and 5:");          for (int i = a; i <= b; i++)              if (array[i - a] == 1)                  System.out.printf(i + " ");      }  } |

**Output :**

MULTIPLES of 2 and 5:

2 4 5 6 8 10

**16) Longest Span with same Sum in two Binary arrays**

|  |
| --- |
| class Test  {      static int arr1[] = new int[]{0, 1, 0, 1, 1, 1, 1};      static int arr2[] = new int[]{1, 1, 1, 1, 1, 0, 1};        // Returns length of the longest common sum in arr1[]      // and arr2[]. Both are of same size n.      static int longestCommonSum(int n)      {          // Initialize result          int maxLen = 0;            // One by one pick all possible starting points          // of subarrays          for (int i=0; i<n; i++)          {             // Initialize sums of current subarrays             int sum1 = 0, sum2 = 0;               // Conider all points for starting with arr[i]             for (int j=i; j<n; j++)             {                 // Update sums                 sum1 += arr1[j];                 sum2 += arr2[j];                   // If sums are same and current length is                 // more than maxLen, update maxLen                 if (sum1 == sum2)                 {                   int len = j-i+1;                   if (len > maxLen)                      maxLen = len;                 }             }          }          return maxLen;      }        // Driver method to test the above function      public static void main(String[] args)      {          System.out.print("Length of the longest common span with same sum is ");          System.out.println(longestCommonSum(arr1.length));      }  } |

**Output :**

Length of the longest common span with same sum is 6

**Time Complexity :** O(n2)

**Sorting :**

**Examples:**

**1) Alternative Sorting**

import java.io.\*;

import java.util.Arrays;

class AlternativeString

{

// Function to print alternate sorted values

static void alternateSort(int arr[], int n)

{

Arrays.sort(arr);

// Printing the last element of array

// first and then first element and then

// second last element and then second

// element and so on.

int i = 0, j = n-1;

while (i < j) {

System.out.print(arr[j--] + " ");

System.out.print(arr[i++] + " ");

}

// If the total element in array is odd

// then print the last middle element.

if (n % 2 != 0)

System.out.print(arr[i]);

}

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

public static void main (String[] args)

{

int arr[] = {1, 12, 4, 6, 7, 10};

int n = arr.length;

alternateSort(arr, n);

}

}

Output:

12 1 10 4 7 6

**2) Sort an array in wave form**

import java.util.\*;

class SortWave

{

// A utility method to swap two numbers.

void swap(int arr[], int a, int b)

{

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

// This function sorts arr[0..n-1] in wave form, i.e.,

// arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4]..

void sortInWave(int arr[], int n)

{

// Sort the input array

Arrays.sort(arr);

// Swap adjacent elements

for (int i=0; i<n-1; i += 2)

swap(arr, i, i+1);

}

// Driver method

public static void main(String args[])

{

SortWave ob = new SortWave();

int arr[] = {10, 90, 49, 2, 1, 5, 23};

int n = arr.length;

ob.sortInWave(arr, n);

for (int i : arr)

System.out.print(i + " ");

}

}

Output:

2 1 10 5 49 23 90

**3) Merge an array of size n into another array of size m+n**

class MergeArrays

{

/\* Function to move m elements at the end of array mPlusN[] \*/

void moveToEnd(int mPlusN[], int size)

{

int i, j = size - 1;

for (i = size - 1; i >= 0; i--)

{

if (mPlusN[i] != -1)

{

mPlusN[j] = mPlusN[i];

j--;

}

}

}

/\* Merges array N[] of size n into array mPlusN[]

of size m+n\*/

void merge(int mPlusN[], int N[], int m, int n)

{

int i = n;

/\* Current index of i/p part of mPlusN[]\*/

int j = 0;

/\* Current index of N[]\*/

int k = 0;

/\* Current index of output mPlusN[]\*/

while (k < (m + n))

{

/\* Take an element from mPlusN[] if

a) value of the picked element is smaller and we have

not reached end of it

b) We have reached end of N[] \*/

if ((i < (m + n) && mPlusN[i] <= N[j]) || (j == n))

{

mPlusN[k] = mPlusN[i];

k++;

i++;

}

else // Otherwise take element from N[]

{

mPlusN[k] = N[j];

k++;

j++;

}

}

}

/\* Utility that prints out an array on a line \*/

void printArray(int arr[], int size)

{

int i;

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

System.out.print(arr[i] + " ");

System.out.println("");

}

public static void main(String[] args)

{

MergeArrays mergearray = new MergeArrays();

/\* Initialize arrays \*/

int mPlusN[] = {2, 8, -1, -1, -1, 13, -1, 15, 20};

int N[] = {5, 7, 9, 25};

int n = N.length;

int m = mPlusN.length - n;

/\*Move the m elements at the end of mPlusN\*/

mergearray.moveToEnd(mPlusN, m + n);

/\*Merge N[] into mPlusN[] \*/

mergearray.merge(mPlusN, N, m, n);

/\* Print the resultant mPlusN \*/

mergearray.printArray(mPlusN, m + n);

}

}

Output:

2 5 7 8 9 13 15 20 25

**4) Sort a nearly sorted (or K sorted) array**

|  |
| --- |
| import java.util.Iterator;  import java.util.PriorityQueue;    class GFG  {      private static void kSort(int[] arr, int n, int k)      {            // min heap          PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();            // add first k + 1 items to the min heap          for(int i = 0; i < k + 1; i++)          {              priorityQueue.add(arr[i]);          }            int index = 0;          for(int i = k + 1; i < n; i++)          {              arr[index++] = priorityQueue.peek();              priorityQueue.poll();              priorityQueue.add(arr[i]);          }            Iterator<Integer> itr = priorityQueue.iterator();            while(itr.hasNext())          {              arr[index++] = priorityQueue.peek();              priorityQueue.poll();          }        }        // A utility function to print the array      private static void printArray(int[] arr, int n)      {          for(int i = 0; i < n; i++)              System.out.print(arr[i] + " ");      }        // Driver Code      public static void main(String[] args)      {          int k = 3;          int arr[] = { 2, 6, 3, 12, 56, 8 };          int n = arr.length;          kSort(arr, n, k);          System.out.println("Following is sorted array");          printArray(arr, n);      }  } |

**Output:**

Following is sorted array

2 3 6 8 12 56

The Min Heap based method takes O(nLogk) time

**5) Sort an array according to absolute difference with given value**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG  {        // Function to sort an array according absolute      // difference with x.      static void rearrange(int[] arr, int n, int x)      {              TreeMap<Integer, ArrayList<Integer>> m = new TreeMap<>();                // Store values in a map with the difference              // with X as key              for (int i = 0; i < n; i++)              {                  int diff = Math.abs(x - arr[i]);                  if (m.containsKey(diff))                  {                      ArrayList<Integer> al = m.get(diff);                      al.add(arr[i]);                      m.put(diff, al);                  }                  else                  {                          ArrayList<Integer> al = new ArrayList<>();                          al.add(arr[i]);                          m.put(diff,al);                  }              }                // Update the values of array              int index = 0;              for (Map.Entry entry : m.entrySet())              {                  ArrayList<Integer> al = m.get(entry.getKey());                  for (int i = 0; i < al.size(); i++)                          arr[index++] = al.get(i);              }      }        // Function to print the array      static void printArray(int[] arr, int n)      {              for (int i = 0; i < n; i++)                  System.out.print(arr[i] + " ");      }        // Driver code      public static void main(String args[])      {              int[] arr = {10, 5, 3, 9 ,2};              int n = arr.length;              int x = 7;              rearrange(arr, n, x);              printArray(arr, n);      }  } |

**Output:**

5 9 10 3 2

**Time Complexity :** O(n Log n)

**6) Sort 1 to N by swapping adjacent elements**

|  |
| --- |
| import java.util.Arrays;    // Java program to test whether an array  // can be sorted by swapping adjacent  // elements using boolean array    class GFG {        // Return true if array can be      // sorted otherwise false      static boolean sortedAfterSwap(int A[],                                     boolean B[], int n)      {          int i, j;            // Check bool array B and sorts          // elements for continuous sequence of 1          for (i = 0; i < n - 1; i++) {              if (B[i]) {                  j = i;                  while (B[j]) {                      j++;                  }                  // Sort array A from i to j                  Arrays.sort(A, i, 1 + j);                  i = j;              }          }            // Check if array is sorted or not          for (i = 0; i < n; i++) {              if (A[i] != i + 1) {                  return false;              }          }            return true;      }        // Driver program to test sortedAfterSwap()      public static void main(String[] args)      {          int A[] = { 1, 2, 5, 3, 4, 6 };          boolean B[] = { false, true, true, true, false };          int n = A.length;            if (sortedAfterSwap(A, B, n)) {              System.out.println("A can be sorted");          }          else {              System.out.println("A can not be sorted");          }      }  } |

**Output:**

A can be sorted

**7) Sort an array containing two types of elements**

|  |
| --- |
| static void segregate0and1(int arr[], int n) {          int type0 = 0;          int type1 = n - 1;            while (type0 < type1) {              if (arr[type0] == 1) {                    // swap type0 and type1                  arr[type0] = arr[type0] + arr[type1];                  arr[type1] = arr[type0]-arr[type1];                  arr[type0] = arr[type0]-arr[type1];                  type1--;              } else {                  type0++;              }          }      }        // Driver program      public static void main(String[] args) {            int arr[] = { 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1 };              segregate0and1(arr, arr.length);            for (int a : arr)                System.out.print(a+" ");        }  } |

Time Complexity : O(n)

**Output:**

0 0 0 1 1 1 1 1 1 1 1

**8) Count Inversions in an array**

|  |
| --- |
| class Test {      static int arr[] = new int[] { 1, 20, 6, 4, 5 };        static int getInvCount(int n)      {          int inv\_count = 0;          for (int i = 0; i < n - 1; i++)              for (int j = i + 1; j < n; j++)                  if (arr[i] > arr[j])                      inv\_count++;            return inv\_count;      }        // Driver method to test the above function      public static void main(String[] args)      {          System.out.println("Number of inversions are "                             + getInvCount(arr.length));      }  } |

**Output:**

Number of inversions are 5

**Complexity Analysis:**

**Time Complexity:** O(n^2),

**9) Two elements whose sum is closest to zero**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  class Main  {      static void minAbsSumPair(int arr[], int arr\_size)      {        int inv\_count = 0;        int l, r, min\_sum, sum, min\_l, min\_r;          /\* Array should have at least two elements\*/        if(arr\_size < 2)        {          System.out.println("Invalid Input");          return;        }          /\* Initialization of values \*/        min\_l = 0;        min\_r = 1;        min\_sum = arr[0] + arr[1];          for(l = 0; l < arr\_size - 1; l++)        {          for(r = l+1; r < arr\_size; r++)          {            sum = arr[l] + arr[r];            if(Math.abs(min\_sum) > Math.abs(sum))            {              min\_sum = sum;              min\_l = l;              min\_r = r;            }          }        }          System.out.println(" The two elements whose "+                                "sum is minimum are "+                          arr[min\_l]+ " and "+arr[min\_r]);      }        // main function      public static void main (String[] args)      {          int arr[] = {1, 60, -10, 70, -80, 85};          minAbsSumPair(arr, 6);      }    } |

**Output:**

The two elements whose sum is minimum are -80 and 85

**Time complexity:**O(n^2)

**10) Shortest Un-ordered Subarray**

|  |
| --- |
| import java.util.\*;  import java.io.\*;    class GFG {        // boolean function to check array elements      // are in increasing order or not      public static boolean increasing(int a[],int n)      {          for (int i = 0; i < n - 1; i++)              if (a[i] >= a[i + 1])                  return false;            return true;      }        // boolean function to check array elements      // are in decreasing order or not      public static boolean decreasing(int arr[],int n)      {          for (int i = 0; i < n - 1; i++)              if (arr[i] < arr[i + 1])                  return false;            return true;      }        public static int shortestUnsorted(int a[],int n)      {            // increasing and decreasing are two functions.          // if function return true value then print          // 0 otherwise 3.          if (increasing(a, n) == true ||                               decreasing(a, n) == true)              return 0;          else              return 3;      }        // driver program      public static void main (String[] args) {            int ar[] = new int[]{7, 9, 10, 8, 11};          int n = ar.length;            System.out.println(shortestUnsorted(ar,n));      }  } |

**Output :**

3

**11) Minimum number of swaps required to sort an array**

|  |
| --- |
| import javafx.util.Pair;  import java.util.ArrayList;  import java.util.\*;    class GfG  {      // Function returns the minimum number of swaps      // required to sort the array      public static int minSwaps(int[] arr)      {          int n = arr.length;            // Create two arrays and use as pairs where first          // array is element and second array          // is position of first element          ArrayList <Pair <Integer, Integer> > arrpos =                    new ArrayList <Pair <Integer, Integer> > ();          for (int i = 0; i < n; i++)               arrpos.add(new Pair <Integer, Integer> (arr[i], i));            // Sort the array by array element values to          // get right position of every element as the          // elements of second array.          arrpos.sort(new Comparator<Pair<Integer, Integer>>()          {              @Override              public int compare(Pair<Integer, Integer> o1,                                 Pair<Integer, Integer> o2)              {                  if (o1.getKey() > o2.getKey())                      return -1;                    // We can change this to make it then look at the                  // words alphabetical order                  else if (o1.getKey().equals(o2.getKey()))                      return 0;                    else                      return 1;              }          });            // To keep track of visited elements. Initialize          // all elements as not visited or false.          Boolean[] vis = new Boolean[n];          Arrays.fill(vis, false);            // Initialize result          int ans = 0;            // Traverse array elements          for (int i = 0; i < n; i++)          {              // already swapped and corrected or              // already present at correct pos              if (vis[i] || arrpos.get(i).getValue() == i)                  continue;                // find out the number of  node in              // this cycle and add in ans              int cycle\_size = 0;              int j = i;              while (!vis[j])              {                  vis[j] = true;                    // move to next node                  j = arrpos.get(j).getValue();                  cycle\_size++;              }                // Update answer by adding current cycle.              if(cycle\_size > 0)              {                  ans += (cycle\_size - 1);              }          }            // Return result          return ans;      }  }    // Driver class  class MinSwaps  {      // Driver program to test the above function      public static void main(String[] args)      {          int []a = {1, 5, 4, 3, 2};          GfG g = new GfG();          System.out.println(g.minSwaps(a));      }  } |

**Output:**

2

**Time Complexity:** O(n Log n)

**12) Sort an array of 0s, 1s and 2s**

|  |
| --- |
| import java.io.\*;    class countzot {        // Sort the input array, the array is assumed to      // have values in {0, 1, 2}      static void sort012(int a[], int arr\_size)      {          int lo = 0;          int hi = arr\_size - 1;          int mid = 0, temp = 0;          while (mid <= hi) {              switch (a[mid]) {              case 0: {                  temp = a[lo];                  a[lo] = a[mid];                  a[mid] = temp;                  lo++;                  mid++;                  break;              }              case 1:                  mid++;                  break;              case 2: {                  temp = a[mid];                  a[mid] = a[hi];                  a[hi] = temp;                  hi--;                  break;              }              }          }      }        /\* Utility function to print array arr[] \*/      static void printArray(int arr[], int arr\_size)      {          int i;          for (i = 0; i < arr\_size; i++)              System.out.print(arr[i] + " ");          System.out.println("");      }        /\*Driver function to check for above functions\*/      public static void main(String[] args)      {          int arr[] = { 0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1 };          int arr\_size = arr.length;          sort012(arr, arr\_size);          System.out.println("Array after seggregation ");          printArray(arr, arr\_size);      }  } |

**Output:**

array after segregation 0 0 0 0 0 1 1 1 1 1 2 2

**Time Complexity:** O(n)

**13) Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted**

|  |
| --- |
| class Main  {      static void printUnsorted(int arr[], int n)      {        int s = 0, e = n-1, i, max, min;          // step 1(a) of above algo        for (s = 0; s < n-1; s++)        {          if (arr[s] > arr[s+1])            break;        }        if (s == n-1)        {          System.out.println("The complete array is sorted");          return;        }          // step 1(b) of above algo        for(e = n - 1; e > 0; e--)        {          if(arr[e] < arr[e-1])            break;        }          // step 2(a) of above algo        max = arr[s]; min = arr[s];        for(i = s + 1; i <= e; i++)        {          if(arr[i] > max)            max = arr[i];          if(arr[i] < min)            min = arr[i];        }          // step 2(b) of above algo        for( i = 0; i < s; i++)        {          if(arr[i] > min)          {            s = i;            break;          }        }          // step 2(c) of above algo        for( i = n -1; i >= e+1; i--)        {          if(arr[i] < max)          {            e = i;            break;          }        }          // step 3 of above algo        System.out.println(" The unsorted subarray which"+                           " makes the given array sorted lies"+                         "  between the indices "+s+" and "+e);        return;      }        public static void main(String args[])      {        int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};        int arr\_size = arr.length;        printUnsorted(arr, arr\_size);      }  } |

**Output :**

The unsorted subarray which makes the given array

sorted lies between the indees 3 and 8

**Time Complexity:**O(n)

**14) Count the number of possible triangles**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class CountTriangles {      // Function to count all possible triangles with arr[]      // elements      static int findNumberOfTriangles(int arr[])      {          int n = arr.length;          // Sort the array elements in non-decreasing order          Arrays.sort(arr);            // Initialize count of triangles          int count = 0;            // Fix the first element. We need to run till n-3 as          // the other two elements are selected from arr[i+1...n-1]          for (int i = 0; i < n - 2; ++i) {              // Initialize index of the rightmost third element              int k = i + 2;                // Fix the second element              for (int j = i + 1; j < n; ++j) {                  /\* Find the rightmost element which is smaller                  than the sum of two fixed elements                  The important thing to note here is, we use                  the previous value of k. If value of arr[i] +                  arr[j-1] was greater than arr[k], then arr[i] +                  arr[j] must be greater than k, because the                  array is sorted. \*/                  while (k < n && arr[i] + arr[j] > arr[k])                      ++k;                    /\* Total number of possible triangles that can be                  formed with the two fixed elements is k - j - 1.                  The two fixed elements are arr[i] and arr[j]. All                  elements between arr[j+1] to arr[k-1] can form a                  triangle with arr[i] and arr[j]. One is subtracted                  from k because k is incremented one extra in above                  while loop. k will always be greater than j. If j                  becomes equal to k, then above loop will increment                  k, because arr[k] + arr[i] is always/ greater than                  arr[k] \*/                  if (k > j)                      count += k - j - 1;              }          }          return count;      }        public static void main(String[] args)      {          int arr[] = { 10, 21, 22, 100, 101, 200, 300 };          System.out.println("Total number of triangles is " + findNumberOfTriangles(arr));      }  } |

**Output:**

Total number of triangles possible is 6

**Complexity Analysis:**

**Time Complexity:** O(n^2).

**15) Find number of pairs (x, y) in an array such that x^y > y^x**

|  |
| --- |
| import java.util.Arrays;    class Test  {      // Function to return count of pairs with x as one element      // of the pair. It mainly looks for all values in Y[] where      // x ^ Y[i] > Y[i] ^ x      static int count(int x, int Y[], int n, int NoOfY[])      {          // If x is 0, then there cannot be any value in Y such that          // x^Y[i] > Y[i]^x          if (x == 0) return 0;            // If x is 1, then the number of pais is equal to number of          // zeroes in Y[]          if (x == 1) return NoOfY[0];            // Find number of elements in Y[] with values greater than x          // getting upperbound of x with binary search          int idx = Arrays.binarySearch(Y, x);          int ans;          if(idx < 0){              idx = Math.abs(idx+1);              ans = Y.length - idx;          }          else{              while (idx<n && Y[idx]==x) {                  idx++;              }              ans = Y.length - idx;          }            // If we have reached here, then x must be greater than 1,          // increase number of pairs for y=0 and y=1          ans += (NoOfY[0] + NoOfY[1]);            // Decrease number of pairs for x=2 and (y=4 or y=3)          if (x == 2)  ans -= (NoOfY[3] + NoOfY[4]);            // Increase number of pairs for x=3 and y=2          if (x == 3)  ans += NoOfY[2];            return ans;      }        // Function to returns count of pairs (x, y) such that      // x belongs to X[], y belongs to Y[] and x^y > y^x      static int countPairs(int X[], int Y[], int m, int n)      {          // To store counts of 0, 1, 2, 3 and 4 in array Y          int NoOfY[] = new int[5];          for (int i = 0; i < n; i++)              if (Y[i] < 5)                  NoOfY[Y[i]]++;            // Sort Y[] so that we can do binary search in it          Arrays.sort(Y);            int total\_pairs = 0; // Initialize result            // Take every element of X and count pairs with it          for (int i=0; i<m; i++)              total\_pairs += count(X[i], Y, n, NoOfY);            return total\_pairs;      }        // Driver method      public static void main(String args[])      {          int X[] = {2, 1, 6};          int Y[] = {1, 5};            System.out.println("Total pairs = " + countPairs(X, Y, X.length, Y.length));      }  } |

**Output:**

Total pairs = 3

**Time Complexity :** O(nLogn + mLogn), where m and n are the sizes of arrays X[] and Y[] respectively. The sort step takes O(nLogn) time. Then every element of X[] is searched in Y[] using binary search. This step takes O(mLogn) time.

**16) Count all distinct pairs with difference equal to k**

|  |
| --- |
| import java.util.\*;  import java.io.\*;    class GFG {        static int countPairsWithDiffK(int arr[],                                      int n, int k)      {          int count = 0;            // Pick all elements one by one          for (int i = 0; i < n; i++)          {              // See if there is a pair              // of this picked element              for (int j = i + 1; j < n; j++)                  if (arr[i] - arr[j] == k ||                      arr[j] - arr[i] == k)                      count++;          }          return count;      }        // Driver code      public static void main(String args[])      {          int arr[] = { 1, 5, 3, 4, 2 };          int n = arr.length;          int k = 3;          System.out.println("Count of pairs with given diff is "                          + countPairsWithDiffK(arr, n, k));      }  } |

Output :

Count of pairs with given diff is 2

Time Complexity of O(n2)

**17) Print All Distinct Elements of a given integer array**

|  |
| --- |
| import java.io.\*;    class GFG {        static void printDistinct(int arr[], int n)      {          // Pick all elements one by one          for (int i = 0; i < n; i++)          {              // Check if the picked element              // is already printed              int j;              for (j = 0; j < i; j++)              if (arr[i] == arr[j])                  break;                // If not printed earlier,              // then print it              if (i == j)              System.out.print( arr[i] + " ");          }      }        // Driver program      public static void main (String[] args)      {          int arr[] = {6, 10, 5, 4, 9, 120, 4, 6, 10};          int n = arr.length;          printDistinct(arr, n);        }  } |

Output:

6 10 5 4 9 120

Time Complexity of above solution is O(n2)

**18) Construct an array from its pair-sum array**

|  |
| --- |
| import java.io.\*;    class PairSum {        // Fills element in arr[] from its pair sum array pair[].      // n is size of arr[]      static void constructArr(int arr[], int pair[], int n)      {          arr[0] = (pair[0]+pair[1]-pair[n-1]) / 2;          for (int i=1; i<n; i++)              arr[i] = pair[i-1]-arr[0];      }        // Driver program to test above function      public static void main(String[] args)      {          int pair[] = {15, 13, 11, 10, 12, 10, 9, 8, 7, 5};          int n = 5;          int[] arr = new int[n];          constructArr(arr, pair, n);          for (int i = 0; i < n; i++)              System.out.print(arr[i]+" ");      }  } |

**Output :**

8 7 5 3 2

Time complexity of constructArr() is O(n) where n is number of elements in arr[].

**19) Product of maximum in first array and minimum in second**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;    class GfG  {        // Function to calculate the product      public static int minMaxProduct(int arr1[],                                      int arr2[],                                      int n1,                                      int n2)         {            // Initialize max of          // first array          int max = arr1[0];            // initialize min of          // second array          int min = arr2[0];            int i;          for (i = 1; i < n1 && i < n2; ++i)          {            // To find the maximum          // element in first array          if (arr1[i] > max)              max = arr1[i];            // To find the minimum element          // in second array          if (arr2[i] < min)              min = arr2[i];          }            // Process remaining elements          while (i < n1)          {              if (arr1[i] > max)              max = arr1[i];              i++;          }          while (i < n2)          {              if (arr2[i] < min)              min = arr2[i];              i++;          }            return max \* min;      }        // Driver Code      public static void main(String argc[])      {          int [] arr1= new int []{ 10, 2, 3,                                   6, 4, 1 };          int [] arr2 = new int []{ 5, 1, 4,                                    2, 6, 9 };          int n1 = 6;          int n2 = 6;          System.out.println(minMaxProduct(arr1, arr2,                                            n1, n2));      }  } |

**Output :**

10

**Time Complexity :** O(n)

**Searching :**

**Examples:**

**1) Search, insert and delete in a sorted array**

class Main {

// function to implement

// binary search

static int binarySearch(int arr[], int low, int high, int key)

{

if (high < low)

return -1;

/\*low + (high - low)/2;\*/

int mid = (low + high) / 2;

if (key == arr[mid])

return mid;

if (key > arr[mid])

return binarySearch(arr, (mid + 1), high, key);

return binarySearch(arr, low, (mid - 1), key);

}

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

public static void main(String[] args)

{

int arr[] = { 5, 6, 7, 8, 9, 10 };

int n, key;

n = arr.length;

key = 10;

System.out.println("Index: " + binarySearch(arr, 0, n, key));

}

}

Output:

Index: 5

**2) Find position of an element in a sorted array of infinite numbers**

class Test

{

// Simple binary search algorithm

static int binarySearch(int arr[], int l, int r, int x)

{

if (r>=l)

{

int mid = l + (r - l)/2;

if (arr[mid] == x)

return mid;

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

return binarySearch(arr, mid+1, r, x);

}

return -1;

}

// Method takes an infinite size array and a key to be

// searched and returns its position if found else -1.

// We don't know size of arr[] and we can assume size to be

// infinite in this function.

// NOTE THAT THIS FUNCTION ASSUMES arr[] TO BE OF INFINITE SIZE

// THEREFORE, THERE IS NO INDEX OUT OF BOUND CHECKING

static int findPos(int arr[],int key)

{

int l = 0, h = 1;

int val = arr[0];

// Find h to do binary search

while (val < key)

{

l = h;     // store previous high

//check that 2\*h doesn't exceeds array

//length to prevent ArrayOutOfBoundException

if(2\*h < arr.length-1)

h = 2\*h;

else

h = arr.length-1;

val = arr[h]; // update new val

}

// at this point we have updated low

// and high indices, thus use binary

// search between them

return binarySearch(arr, l, h, key);

}

// Driver method to test the above function

public static void main(String[] args)

{

int arr[] = new int[]{3, 5, 7, 9, 10, 90,

100, 130, 140, 160, 170};

int ans = findPos(arr,10);

if (ans==-1)

System.out.println("Element not found");

else

System.out.println("Element found at index " + ans);

}

}

Output:

Element found at index 4

**3) Maximum equlibrium sum in an array**

import java.io.\*;

class GFG {

// Function to find maximum

// equilibrium sum.

static int findMaxSum(int []arr, int n)

{

int res = Integer.MIN\_VALUE;

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

{

int prefix\_sum = arr[i];

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

prefix\_sum += arr[j];

int suffix\_sum = arr[i];

for (int j = n - 1; j > i; j--)

suffix\_sum += arr[j];

if (prefix\_sum == suffix\_sum)

res = Math.max(res, prefix\_sum);

}

return res;

}

// Driver Code

public static void main (String[] args)

{

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

int n = arr.length;

System.out.println(findMaxSum(arr, n));

}

}

Output:

7

**4) Search, insert and delete in an unsorted array**

|  |
| --- |
| class Main  {      // Function to implement      // search operation      static int findElement(int arr[], int n,                             int key)      {          for (int i = 0; i < n; i++)              if (arr[i] == key)                  return i;            return -1;      }        // Driver Code      public static void main(String args[])      {          int arr[] = {12, 34, 10, 6, 40};          int n = arr.length;            // Using a last element as search element          int key = 40;          int position = findElement(arr, n, key);            if (position == - 1)              System.out.println("Element not found");          else              System.out.println("Element Found at Position: "                                  + (position + 1));      }  } |

Output:

Element Found at Position: 5

**5) Given an array A[] and a number x, check for pair in A[] with sum as x**

|  |
| --- |
| import java.util.\*;    class GFG {      // Function to check if array has 2 elements      // whose sum is equal to the given value      static boolean hasArrayTwoCandidates(int A[],                                           int arr\_size, int sum)      {          int l, r;            /\* Sort the elements \*/          Arrays.sort(A);            /\* Now look for the two candidates          in the sorted array\*/          l = 0;          r = arr\_size - 1;          while (l < r) {              if (A[l] + A[r] == sum)                  return true;              else if (A[l] + A[r] < sum)                  l++;              else // A[i] + A[j] > sum                  r--;          }          return false;      }        // Driver code      public static void main(String args[])      {          int A[] = { 1, 4, 45, 6, 10, -8 };          int n = 16;          int arr\_size = A.length;            // Function calling          if (hasArrayTwoCandidates(A, arr\_size, n))              System.out.println("Array has two "                                 + "elements with given sum");          else              System.out.println("Array doesn't have "                                 + "two elements with given sum");      }  } |

**Output :**

Array has two elements with the given sum

**6) Searching in an array where adjacent differ by at most k**

|  |
| --- |
| import java.io.\*;    class GFG {        // x is the element to be searched      // in arr[0..n-1] such that all      // elements differ by at-most k.      static int search(int arr[], int n,                              int x, int k)      {            // Traverse the given array starting          // from leftmost element          int i = 0;            while (i < n) {                // If x is found at index i              if (arr[i] == x)                  return i;                // Jump the difference between              // current array element and x              // divided by k We use max here              // to make sure that i moves              // at-least one step ahead.              i = i + Math.max(1, Math.abs(arr[i]                                        - x) / k);          }            System.out.println("number is " +                                  "not present!");          return -1;      }        // Driver program to test above function      public static void main(String[] args)      {            int arr[] = { 2, 4, 5, 7, 7, 6 };          int x = 6;          int k = 2;          int n = arr.length;            System.out.println("Element " + x +                          " is present at index "                          + search(arr, n, x, k));      }  } |

**Output:**

Element 6 is present at index 5

**7) Find common elements in three sorted arrays**

|  |
| --- |
| class FindCommon  {      // This function prints common elements in ar1      void findCommon(int ar1[], int ar2[], int ar3[])      {          // Initialize starting indexes for ar1[], ar2[] and ar3[]          int i = 0, j = 0, k = 0;            // Iterate through three arrays while all arrays have elements          while (i < ar1.length && j < ar2.length && k < ar3.length)          {               // If x = y and y = z, print any of them and move ahead               // in all arrays               if (ar1[i] == ar2[j] && ar2[j] == ar3[k])               {   System.out.print(ar1[i]+" ");   i++; j++; k++; }                 // x < y               else if (ar1[i] < ar2[j])                   i++;                 // y < z               else if (ar2[j] < ar3[k])                   j++;                 // We reach here when x > y and z < y, i.e., z is smallest               else                   k++;          }      }        // Driver code to test above      public static void main(String args[])      {          FindCommon ob = new FindCommon();            int ar1[] = {1, 5, 10, 20, 40, 80};          int ar2[] = {6, 7, 20, 80, 100};          int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};            System.out.print("Common elements are ");          ob.findCommon(ar1, ar2, ar3);      }  } |

Output:

Common Elements are 20 80

Time complexity of the above solution is O(n1 + n2 + n3).

**8) Find the only repetitive element between 1 to n-1**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG  {        static int findRepeating(int[] arr, int n)      {          // Find array sum and subtract sum          // first n-1 natural numbers from it          // to find the result.            int sum = 0;          for (int i = 0; i < n; i++)              sum += arr[i];          return sum - (((n - 1) \* n) / 2);      }        // Driver code      public static void main(String args[])      {          int[] arr = { 9, 8, 2, 6, 1, 8, 5, 3, 4, 7 };          int n = arr.length;          System.out.println(findRepeating(arr, n));      }  } |

**Output :**

8

**9) Find the element that appears once in an array where every other element appears twice**

|  |
| --- |
| class MaxSum  {      // Return the maximum Sum of difference      // between consecutive elements.      static int findSingle(int ar[], int ar\_size)      {          // Do XOR of all elements and return          int res = ar[0];          for (int i = 1; i < ar\_size; i++)              res = res ^ ar[i];            return res;      }        // Driver code      public static void main (String[] args)      {          int ar[] = {2, 3, 5, 4, 5, 3, 4};          int n = ar.length;          System.out.println("Element occurring once is " +                              findSingle(ar, n) + " ");      }  } |

Output:

Element occurring once is 2

Time complexity of this solution is O(n)

**10) Maximum Subarray Sum Excluding Certain Elements**

|  |
| --- |
| import java.io.\*;    class GFG {        // Function to check the element      // present in array B      static boolean isPresent(int B[],                              int m,                              int x)      {          for (int i = 0; i < m; i++)              if (B[i] == x)                  return true;            return false;      }        // Utility function for findMaxSubarraySum()      // with the following parameters      // A => Array A,      // B => Array B,      // n => Number of elements in Array A,      // m => Number of elements in Array B      static int findMaxSubarraySumUtil(int A[], int B[],                                        int n, int m)      {            // set max\_so\_far to INT\_MIN          int max\_so\_far = -2147483648, curr\_max = 0;            for (int i = 0; i < n; i++) {                // if the element is present in B,              // set current max to 0 and move to              // the next element              if (isPresent(B, m, A[i])) {                  curr\_max = 0;                  continue;              }                // Proceed as in Kadane's Algorithm              curr\_max = Math.max(A[i], curr\_max + A[i]);              max\_so\_far = Math.max(max\_so\_far, curr\_max);          }          return max\_so\_far;      }        // Wrapper for findMaxSubarraySumUtil()      static void findMaxSubarraySum(int A[], int B[],                                     int n, int m)      {          int maxSubarraySum = findMaxSubarraySumUtil(A, B,                                                      n, m);            // This case will occour when all          // elements of A are present          // in B, thus no subarray can be formed          if (maxSubarraySum == -2147483648) {                          System.out.println("Maximum Subarray Sum"                                          + " " + "can't be found");            }          else {              System.out.println("The Maximum Subarray Sum = "                                  + maxSubarraySum);          }      }        // Driver code      public static void main(String[] args)      {            int A[] = { 3, 4, 5, -4, 6 };          int B[] = { 1, 8, 5 };            int n = A.length;          int m = B.length;            // Calling Function          findMaxSubarraySum(A, B, n, m);      }  } |

**Output :**

The Maximum Subarray Sum = 7

Time Complexity of this approach is O(n\*m)

**11) Equilibrium index of an array**

|  |
| --- |
| class EquilibriumIndex {      int equilibrium(int arr[], int n)      {          int i, j;          int leftsum, rightsum;            /\* Check for indexes one by one until             an equilibrium index is found \*/          for (i = 0; i < n; ++i) {                /\* get left sum \*/              leftsum = 0;              for (j = 0; j < i; j++)                  leftsum += arr[j];                /\* get right sum \*/              rightsum = 0;              for (j = i + 1; j < n; j++)                  rightsum += arr[j];                /\* if leftsum and rightsum are same,                 then we are done \*/              if (leftsum == rightsum)                  return i;          }            /\* return -1 if no equilibrium index is found \*/          return -1;      }      // Driver code      public static void main(String[] args)      {          EquilibriumIndex equi = new EquilibriumIndex();          int arr[] = { -7, 1, 5, 2, -4, 3, 0 };          int arr\_size = arr.length;          System.out.println(equi.equilibrium(arr, arr\_size));      }  }  **Output:**  3 |

**Time Complexity:** O(n^2)

**12) Ceiling in a sorted array**

|  |
| --- |
| class Main  {      /\* Function to get index of ceiling         of x in arr[low..high] \*/      static int ceilSearch(int arr[], int low, int high, int x)      {        int i;          /\* If x is smaller than or equal to first           element,then return the first element \*/        if(x <= arr[low])          return low;          /\* Otherwise, linearly search for ceil value \*/        for(i = low; i < high; i++)        {          if(arr[i] == x)            return i;            /\* if x lies between arr[i] and arr[i+1]          including arr[i+1], then return arr[i+1] \*/          if(arr[i] < x && arr[i+1] >= x)             return i+1;        }          /\* If we reach here then x is greater than the        last element of the array,  return -1 in this case \*/        return -1;      }          /\* Driver program to check above functions \*/      public static void main (String[] args)      {         int arr[] = {1, 2, 8, 10, 10, 12, 19};         int n = arr.length;         int x = 3;         int index = ceilSearch(arr, 0, n-1, x);         if(index == -1)           System.out.println("Ceiling of "+x+" doesn't exist in array");         else           System.out.println("ceiling of "+x+" is "+arr[index]);      }  } |

**Output :**

ceiling of 3 is 8

**Time Complexity :** O(n)

**13) Floor in a Sorted Array**

|  |
| --- |
| import java.io.\*;  import java.util.\*;  import java.lang.\*;    class GFG  {    /\* An inefficient function to get index of floor  of x in arr[0..n-1] \*/  static int floorSearch(int arr[], int n, int x)  {      // If last element is smaller than x      if (x >= arr[n-1])          return n-1;        // If first element is greater than x      if (x < arr[0])          return -1;        // Linearly search for the first element      // greater than x      for (int i=1; i<n; i++)      if (arr[i] > x)          return (i-1);        return -1;  }    // Driver Code  public static void main(String[] args)  {      int arr[] = {1, 2, 4, 6, 10, 12, 14};      int n = arr.length;      int x = 7;      int index = floorSearch(arr, n-1, x);      if (index == -1)          System.out.print("Floor of " + x + " doesn't exist in array ");      else          System.out.print("Floor of " + x + " is "+ arr[index]);  }  } |

**Output:**

Floor of 7 is 6.

**Complexity Analysis:**

**Time Complexity :** O(n).  
To traverse an array only one loop is needed so the time complexity is O(n).

**14) Majority Element**

|  |
| --- |
| import java.io.\*;    class GFG {    // Function to find Majority element  // in an array  static void findMajority(int arr[], int n)  {      int maxCount = 0;      int index = -1; // sentinels      for(int i = 0; i < n; i++)      {          int count = 0;          for(int j = 0; j < n; j++)          {              if(arr[i] == arr[j])              count++;          }            // update maxCount if count of          // current element is greater          if(count > maxCount)          {              maxCount = count;              index = i;          }      }        // if maxCount is greater than n/2      // return the corresponding element      if (maxCount > n/2)      System.out.println (arr[index]);        else      System.out.println ("No Majority Element");  }    // Driver code      public static void main (String[] args) {            int arr[] = {1, 1, 2, 1, 3, 5, 1};          int n = arr.length;        // Function calling      findMajority(arr, n);      }  } |

**Output:**

1

**Compelxity Analysis:**

**Time Complexity:** O(n\*n).  
A nested loop is needed where both the loops traverse the array from start to end, so the time complexity is O(n^2).

**15) Check for Majority Element in a sorted array**

|  |
| --- |
| import java.io.\*;    class Majority {        static boolean isMajority(int arr[], int n, int x)      {          int i, last\_index = 0;            /\* get last index according to n (even or odd) \*/          last\_index = (n%2==0)? n/2: n/2+1;            /\* search for first occurrence of x in arr[]\*/          for (i = 0; i < last\_index; i++)          {              /\* check if x is present and is present more                 than n/2 times \*/              if (arr[i] == x && arr[i+n/2] == x)                  return true;          }          return false;      }        /\* Driver function to check for above functions\*/      public static void main (String[] args) {          int arr[] = {1, 2, 3, 4, 4, 4, 4};          int n = arr.length;          int x = 4;          if (isMajority(arr, n, x)==true)             System.out.println(x+" appears more than "+                                n/2+" times in arr[]");          else             System.out.println(x+" does not appear more than "+                                n/2+" times in arr[]");      }  } |

Output :

4 appears more than 3 times in arr[]

**Time Complexity :**O(n)

**16) Check if an array has a majority element**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  import java.io.\*;    class Gfg  {      // Returns true if there is a      // majority element in a[]      static boolean isMajority(int a[], int n)      {          // Insert all elements          // in a hash table          HashMap <Integer,Integer> mp = new                              HashMap<Integer,Integer>();            for (int i = 0; i < n; i++)                if (mp.containsKey(a[i]))                  mp.put(a[i], mp.get(a[i]) + 1);                else mp.put(a[i] , 1);            // Check if frequency of any          // element is n/2 or more.          for (Map.Entry<Integer, Integer> x : mp.entrySet())                if (x.getValue() >= n/2)                  return true;          return false;      }        // Driver code      public static void main (String[] args)      {          int a[] = { 2, 3, 9, 2, 2 };          int n = a.length;            if (isMajority(a, n))              System.out.println("Yes");          else              System.out.println("No");      }  } |

**Output:**

Yes

**17) Find a peak element**

|  |
| --- |
| import java.util.\*;  import java.lang.\*;  import java.io.\*;    class PeakElement  {      // A binary search based function that returns index of a peak      // element      static int findPeakUtil(int arr[], int low, int high, int n)      {          // Find index of middle element          int mid = low + (high - low)/2;  /\* (low + high)/2 \*/            // Compare middle element with its neighbours (if neighbours          // exist)          if ((mid == 0 || arr[mid-1] <= arr[mid]) && (mid == n-1 ||               arr[mid+1] <= arr[mid]))              return mid;            // If middle element is not peak and its left neighbor is          // greater than it,then left half must have a peak element          else if (mid > 0 && arr[mid-1] > arr[mid])              return findPeakUtil(arr, low, (mid -1), n);            // If middle element is not peak and its right neighbor          // is greater than it, then right half must have a peak          // element          else return findPeakUtil(arr, (mid + 1), high, n);      }        // A wrapper over recursive function findPeakUtil()      static int findPeak(int arr[], int n)      {          return findPeakUtil(arr, 0, n-1, n);      }        // Driver method      public static void main (String[] args)      {          int arr[] = {1, 3, 20, 4, 1, 0};          int n = arr.length;          System.out.println("Index of a peak point is " +                              findPeak(arr, n));      }  } |

**Output:**

Index of a peak point is 2

**Compelxity Analysis:**

**Time Complexity:** O(Logn).  
Where n is the number of elements in the input array. In each step our search becomes half. So it can be compared to Binary search, So the time complexity is O(log n)

**18) Find the two repeating elements in a given array**

|  |
| --- |
| class RepeatElement  {      void printRepeating(int arr[], int size)      {          int i, j;          System.out.println("Repeated Elements are :");          for (i = 0; i < size; i++)          {              for (j = i + 1; j < size; j++)              {                  if (arr[i] == arr[j])                      System.out.print(arr[i] + " ");              }          }      }        public static void main(String[] args)      {          RepeatElement repeat = new RepeatElement();          int arr[] = {4, 2, 4, 5, 2, 3, 1};          int arr\_size = arr.length;          repeat.printRepeating(arr, arr\_size);      }  } |

**Output :**

Repeating elements are 4 2

Time Complexity: O(n\*n)

**19) Find a Fixed Point (Value equal to index) in a given array**

|  |
| --- |
| class Main  {      static int linearSearch(int arr[], int n)      {          int i;          for(i = 0; i < n; i++)          {              if(arr[i] == i)                  return i;          }            /\* If no fixed point present             then return -1 \*/          return -1;      }      //main function      public static void main(String args[])      {          int arr[] = {-10, -1, 0, 3, 10, 11, 30, 50, 100};          int n = arr.length;          System.out.println("Fixed Point is "                       + linearSearch(arr, n));      }  } |

**Output:**

Fixed Point is 3

Time Complexity: O(n)

**20) Find subarray with given sum | Set 1 (Nonnegative Numbers)**

|  |
| --- |
| class SubarraySum  {      /\* Returns true if the there is a subarray of arr[] with sum equal to         'sum' otherwise returns false.  Also, prints the result \*/      int subArraySum(int arr[], int n, int sum)      {          int curr\_sum = arr[0], start = 0, i;            // Pick a starting point          for (i = 1; i <= n; i++)          {              // If curr\_sum exceeds the sum, then remove the starting elements              while (curr\_sum > sum && start < i-1)              {                  curr\_sum = curr\_sum - arr[start];                  start++;              }                // If curr\_sum becomes equal to sum, then return true              if (curr\_sum == sum)              {                  int p = i-1;                  System.out.println("Sum found between indexes " + start                          + " and " + p);                  return 1;              }                // Add this element to curr\_sum              if (i < n)              curr\_sum = curr\_sum + arr[i];            }            System.out.println("No subarray found");          return 0;      }        public static void main(String[] args)      {          SubarraySum arraysum = new SubarraySum();          int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};          int n = arr.length;          int sum = 23;          arraysum.subArraySum(arr, n, sum);      }  } |

**Output :**

Sum found between indexes 1 and 4

**Complexity Analysis:**

**Time Complexity :** O(n). Only one traversal of the array is required. So the time complexity is O(n).

**21) Maximum triplet sum in array**

|  |
| --- |
| import java.io.\*;  import java.util.\*;      class GFG {        // This function assumes that there      // are at least  three elements in arr[].      static int maxTripletSum(int arr[], int n)      {          // Initialize Maximum, second maximum and third          // maximum element          int maxA = -100000000, maxB = -100000000;          int maxC = -100000000;            for (int i = 0; i < n; i++) {                // Update Maximum, second maximum              // and third maximum element              if (arr[i] > maxA)              {                  maxC = maxB;                  maxB = maxA;                  maxA = arr[i];              }                // Update second maximum and third maximum              // element              else if (arr[i] > maxB)              {                  maxC = maxB;                  maxB = arr[i];              }                // Update third maximum element              else if (arr[i] > maxC)                  maxC = arr[i];          }            return (maxA + maxB + maxC);      }        // Driven code      public static void main(String args[])      {          int arr[] = { 1, 0, 8, 6, 4, 2 };          int n = arr.length;          System.out.println(maxTripletSum(arr, n));      }  } |

**Output :**

18

**Time complexity :** O(n)

**22) Smallest Difference Triplet from Three arrays**

|  |
| --- |
| import java.util.Arrays;    class GFG {        // function to find maximum number      static int maximum(int a, int b, int c)      {          return Math.max(Math.max(a, b), c);      }        // function to find minimum number      static int minimum(int a, int b, int c)      {          return Math.min(Math.min(a, b), c);      }        // Finds and prints the smallest Difference      // Triplet      static void smallestDifferenceTriplet(int arr1[],                         int arr2[], int arr3[], int n)      {            // sorting all the three arrays          Arrays.sort(arr1);          Arrays.sort(arr2);          Arrays.sort(arr3);            // To store resultant three numbers          int res\_min=0, res\_max=0, res\_mid=0;            // pointers to arr1, arr2, arr3          // respectively          int i = 0, j = 0, k = 0;            // Loop until one array reaches to its end          // Find the smallest difference.          int diff = 2147483647;            while (i < n && j < n && k < n)          {              int sum = arr1[i] + arr2[j] + arr3[k];                // maximum number              int max = maximum(arr1[i], arr2[j], arr3[k]);                // Find minimum and increment its index.              int min = minimum(arr1[i], arr2[j], arr3[k]);              if (min == arr1[i])                  i++;              else if (min == arr2[j])                  j++;              else                  k++;                // comparing new difference with the              // previous one and updating accordingly              if (diff > (max - min))              {                  diff = max - min;                  res\_max = max;                  res\_mid = sum - (max + min);                  res\_min = min;              }          }            // Print result          System.out.print(res\_max + ", " + res\_mid                                   + ", " + res\_min);      }        //driver code      public static void main (String[] args)      {            int arr1[] = {5, 2, 8};          int arr2[] = {10, 7, 12};          int arr3[] = {9, 14, 6};            int n = arr1.length;            smallestDifferenceTriplet(arr1, arr2, arr3, n);      }  } |

**Output :**

7, 6, 5

**Time Complexity :** O(n log n)

**23) Find a triplet that sum to a given value**

|  |
| --- |
| import java.util.\*;    class GFG {        // returns true if there is triplet      // with sum equal to 'sum' present      // in A[]. Also, prints the triplet      static boolean find3Numbers(int A[],                                  int arr\_size, int sum)      {          // Fix the first element as A[i]          for (int i = 0; i < arr\_size - 2; i++) {                // Find pair in subarray A[i+1..n-1]              // with sum equal to sum - A[i]              HashSet<Integer> s = new HashSet<Integer>();              int curr\_sum = sum - A[i];              for (int j = i + 1; j < arr\_size; j++) {                  if (s.contains(curr\_sum - A[j]) && curr\_sum - A[j] != (int)s.toArray()[s.size() - 1]) {                      System.out.printf("Triplet is %d, %d, %d", A[i],                                        A[j], curr\_sum - A[j]);                      return true;                  }                  s.add(A[j]);              }          }            // If we reach here, then no triplet was found          return false;      }        /\* Driver code \*/      public static void main(String[] args)      {          int A[] = { 1, 4, 45, 6, 10, 8 };          int sum = 22;          int arr\_size = A.length;            find3Numbers(A, arr\_size, sum);      }  } |

**Output :**

Triplet is 4, 8, 10

**24) Find all triplets with zero sum**

|  |
| --- |
| import java.util.\*;    class GFG  {        // function to print triplets with 0 sum      static void findTriplets(int arr[], int n)      {          boolean found = false;            for (int i = 0; i < n - 1; i++)          {              // Find all pairs with sum equals to              // "-arr[i]"              HashSet<Integer> s = new HashSet<Integer>();              for (int j = i + 1; j < n; j++)              {                  int x = -(arr[i] + arr[j]);                  if (s.contains(x))                  {                      System.out.printf("%d %d %d\n", x, arr[i], arr[j]);                      found = true;                  }                  else                  {                      s.add(arr[j]);                  }              }          }            if (found == false)          {              System.out.printf(" No Triplet Found\n");          }      }        // Driver code      public static void main(String[] args)      {          int arr[] = {0, -1, 2, -3, 1};          int n = arr.length;          findTriplets(arr, n);      }  } |

**Output:**

-1 0 1

-3 2 1

**Complexity Analysis:**

**Time Complexity:** O(n2).

**Matrix:**

**1) Rotate Matrix Elements**

|  |
| --- |
| import java.lang.\*;  import java.util.\*;    class GFG  {      static int R = 4;      static int C = 4;        // A function to rotate a matrix      // mat[][] of size R x C.      // Initially, m = R and n = C      static void rotatematrix(int m,                      int n, int mat[][])      {          int row = 0, col = 0;          int prev, curr;            /\*          row - Staring row index          m - ending row index          col - starting column index          n - ending column index          i - iterator          \*/          while (row < m && col < n)          {                if (row + 1 == m || col + 1 == n)                  break;                // Store the first element of next              // row, this element will replace              // first element of current row              prev = mat[row + 1][col];                // Move elements of first row              // from the remaining rows              for (int i = col; i < n; i++)              {                  curr = mat[row][i];                  mat[row][i] = prev;                  prev = curr;              }              row++;                // Move elements of last column              // from the remaining columns              for (int i = row; i < m; i++)              {                  curr = mat[i][n-1];                  mat[i][n-1] = prev;                  prev = curr;              }              n--;                // Move elements of last row              // from the remaining rows              if (row < m)              {                  for (int i = n-1; i >= col; i--)                  {                      curr = mat[m-1][i];                      mat[m-1][i] = prev;                      prev = curr;                  }              }              m--;                // Move elements of first column              // from the remaining rows              if (col < n)              {                  for (int i = m-1; i >= row; i--)                  {                      curr = mat[i][col];                      mat[i][col] = prev;                      prev = curr;                  }              }              col++;          }                // Print rotated matrix              for (int i = 0; i < R; i++)              {                  for (int j = 0; j < C; j++)                  System.out.print( mat[i][j] + " ");                  System.out.print("\n");              }      }    /\* Driver program to test above functions \*/      public static void main(String[] args)      {      // Test Case 1      int a[][] = { {1, 2, 3, 4},                    {5, 6, 7, 8},                  {9, 10, 11, 12},                  {13, 14, 15, 16} };        // Tese Case 2      /\* int a[][] = new int {{1, 2, 3},                              {4, 5, 6},                              {7, 8, 9}                          };\*/      rotatematrix(R, C, a);        }  } |

**Output:**

5 1 2 3

9 10 6 4

13 11 7 8

14 15 16 12

**2) Inplace rotate square matrix by 90 degrees**

|  |
| --- |
| import java.io.\*;    class GFG {      // An Inplace function to rotate a N x N matrix      // by 90 degrees in anti-clockwise direction      static void rotateMatrix(int N, int mat[][])      {          // Consider all squares one by one          for (int x = 0; x < N / 2; x++) {              // Consider elements in group of 4 in              // current square              for (int y = x; y < N - x - 1; y++) {                  // store current cell in temp variable                  int temp = mat[x][y];                    // move values from right to top                  mat[x][y] = mat[y][N - 1 - x];                    // move values from bottom to right                  mat[y][N - 1 - x] = mat[N - 1 - x][N - 1 - y];                    // move values from left to bottom                  mat[N - 1 - x][N - 1 - y] = mat[N - 1 - y][x];                    // assign temp to left                  mat[N - 1 - y][x] = temp;              }          }      }        // Function to print the matrix      static void displayMatrix(int N, int mat[][])      {          for (int i = 0; i < N; i++) {              for (int j = 0; j < N; j++)                  System.out.print(" " + mat[i][j]);                System.out.print("\n");          }          System.out.print("\n");      }        /\* Driver program to test above functions \*/      public static void main(String[] args)      {          int N = 4;            // Test Case 1          int mat[][] = {              { 1, 2, 3, 4 },              { 5, 6, 7, 8 },              { 9, 10, 11, 12 },              { 13, 14, 15, 16 }          };            // Tese Case 2          /\* int mat[][] = {                              {1, 2, 3},                              {4, 5, 6},                              {7, 8, 9}                          };           \*/            // Tese Case 3          /\*int mat[][] = {                          {1, 2},                          {4, 5}                      };\*/            // displayMatrix(mat);            rotateMatrix(N, mat);            // Print rotated matrix          displayMatrix(N, mat);      }  } |

**Output :**

4 8 12 16

3 7 11 15

2 6 10 14

1 5 9 13

**Complexity Analysis:**

**Time Complexity :** O(n\*n), where n is side of array, A single traversal of the matrix is needed.

**3) Rotate a matrix by 90 degree without using any extra space**

|  |
| --- |
| import java.util.\*;    class GFG {        // After transpose we swap elements of      // column one by one for finding left      // rotation of matrix by 90 degree      static void reverseColumns(int arr[][])      {          for (int i = 0; i < arr[0].length; i++)              for (int j = 0, k = arr[0].length - 1;                   j < k; j++, k--) {                  int temp = arr[j][i];                  arr[j][i] = arr[k][i];                  arr[k][i] = temp;              }      }        // Function for do transpose of matrix      static void transpose(int arr[][])      {          for (int i = 0; i < arr.length; i++)              for (int j = i; j < arr[0].length; j++) {                  int temp = arr[j][i];                  arr[j][i] = arr[i][j];                  arr[i][j] = temp;              }      }        // Function for print matrix      static void printMatrix(int arr[][])      {          for (int i = 0; i < arr.length; i++) {              for (int j = 0; j < arr[0].length; j++)                  System.out.print(arr[i][j] + " ");              System.out.println("");          }      }        // Function to anticlockwise rotate      // matrix by 90 degree      static void rotate90(int arr[][])      {          transpose(arr);          reverseColumns(arr);      }        /\* Driver program to test above function \*/      public static void main(String[] args)      {          int arr[][] = { { 1, 2, 3, 4 },                          { 5, 6, 7, 8 },                          { 9, 10, 11, 12 },                          { 13, 14, 15, 16 } };            rotate90(arr);          printMatrix(arr);      }  } |

**Output:**

4 8 12 16

3 7 11 15

2 6 10 14

1 5 9 13

**Complexity Analysis:**

**Time complexity :**O(R\*C), The matrix is traversed twice, So the complexity is O(R\*C).

**4) Rotate a Matrix by 180 degree**

|  |
| --- |
| import java.util.\*;    class GFG {      static int N = 3;        // Function to Rotate the      // matrix by 180 degree      static void rotateMatrix(int mat[][])      {            // Simply print from last          // cell to first cell.          for (int i = N - 1; i >= 0; i--) {              for (int j = N - 1; j >= 0; j--)                  System.out.print(mat[i][j] + " ");                System.out.println();          }      }        // Driver Code      public static void main(String[] args)      {          int[][] mat = { { 1, 2, 3 },                          { 4, 5, 6 },                          { 7, 8, 9 } };            rotateMatrix(mat);      }  } |

**Output :**

9 8 7

6 5 4

3 2 1

**Time complexity :**O(N\*N)

**5) Check if all rows of a matrix are circular rotations of each other**

|  |
| --- |
| class GFG  {        static int MAX = 1000;        // Returns true if all rows of mat[0..n-1][0..n-1]      // are rotations of each other.      static boolean isPermutedMatrix(int mat[][], int n)      {          // Creating a string that contains          // elements of first row.          String str\_cat = "";          for (int i = 0; i < n; i++)          {              str\_cat = str\_cat + "-" + String.valueOf(mat[0][i]);          }            // Concatenating the string with itself          // so that substring search operations          // can be performed on this          str\_cat = str\_cat + str\_cat;            // Start traversing remaining rows          for (int i = 1; i < n; i++)          {              // Store the matrix into vector in the form              // of strings              String curr\_str = "";              for (int j = 0; j < n; j++)              {                  curr\_str = curr\_str + "-" + String.valueOf(mat[i][j]);              }                // Check if the current string is present in              // the concatenated string or not              if (str\_cat.contentEquals(curr\_str))              {                  return false;              }          }            return true;      }        // Drivers code      public static void main(String[] args)      {          int n = 4;          int mat[][] = {{1, 2, 3, 4},          {4, 1, 2, 3},          {3, 4, 1, 2},          {2, 3, 4, 1}          };          if (isPermutedMatrix(mat, n))          {              System.out.println("Yes");          }          else          {              System.out.println("No");          }      }  } |

**Output:**

Yes

**6) Sort the given matrix**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG {        static int SIZE = 10;        // function to sort the given matrix      static void sortMat(int mat[][], int n)      {          // temporary matrix of size n^2          int temp[] = new int[n \* n];          int k = 0;            // copy the elements of matrix          // one by one into temp[]          for (int i = 0; i < n; i++)              for (int j = 0; j < n; j++)                  temp[k++] = mat[i][j];            // sort temp[]          Arrays.sort(temp);            // copy the elements of temp[]          // one by one in mat[][]          k = 0;          for (int i = 0; i < n; i++)              for (int j = 0; j < n; j++)                  mat[i][j] = temp[k++];      }        // function to print the given matrix      static void printMat(int mat[][], int n)      {          for (int i = 0; i < n; i++) {              for (int j = 0; j < n; j++)                  System.out.print( mat[i][j] + " ");              System.out.println();          }      }        // Driver program to test above      public static void main(String args[])      {          int mat[][] = { { 5, 4, 7 },                          { 1, 3, 8 },                          { 2, 9, 6 } };          int n = 3;            System.out.println("Original Matrix:");          printMat(mat, n);            sortMat(mat, n);            System.out.println("Matrix After Sorting:");          printMat(mat, n);        }  } |

Output :

Original Matrix:

5 4 7

1 3 8

2 9 6

Matrix After Sorting:

1 2 3

4 5 6

7 8 9

Time Complexity: O(n2log2n).

**7) Find the row with maximum number of 1s**

|  |
| --- |
| import java.io.\*;    class GFG {      static int R = 4, C = 4;      // Function to find the index of first index      // of 1 in a boolean array arr[]      static int first(int arr[], int low, int high)      {          if (high >= low) {              // Get the middle index              int mid = low + (high - low) / 2;                // Check if the element at middle index is first 1              if ((mid == 0 || (arr[mid - 1] == 0)) && arr[mid] == 1)                  return mid;                // If the element is 0, recur for right side              else if (arr[mid] == 0)                  return first(arr, (mid + 1), high);                // If element is not first 1, recur for left side              else                  return first(arr, low, (mid - 1));          }          return -1;      }        // Function that returns index of row      // with maximum number of 1s.      static int rowWithMax1s(int mat[][])      {          // Initialize max values          int max\_row\_index = 0, max = -1;            // Traverse for each row and count number of          // 1s by finding the index of first 1          int i, index;          for (i = 0; i < R; i++) {              index = first(mat[i], 0, C - 1);              if (index != -1 && C - index > max) {                  max = C - index;                  max\_row\_index = i;              }          }            return max\_row\_index;      }      // Driver Code      public static void main(String[] args)      {          int mat[][] = { { 0, 0, 0, 1 },                          { 0, 1, 1, 1 },                          { 1, 1, 1, 1 },                          { 0, 0, 0, 0 } };          System.out.println("Index of row with maximum 1s is "                                              + rowWithMax1s(mat));      }  } |

**Output:**

Index of row with maximum 1s is 2

**Time Complexity:** O(mLogn) where m is number of rows and n is number of columns in matrix.

**8) Find median in row wise sorted matrix**

|  |
| --- |
| import java.util.Arrays;    public class MedianInRowSorted  {      // function to find median in the matrix      static int binaryMedian(int m[][],int r, int c)      {          int max = Integer.MIN\_VALUE;          int min = Integer.MAX\_VALUE;            for(int i=0; i<r ; i++)          {                // Finding the minimum element              if(m[i][0] < min)                  min = m[i][0];                // Finding the maximum element              if(m[i][c-1] > max)                  max = m[i][c-1];          }            int desired = (r \* c + 1) / 2;          while(min < max)          {              int mid = min + (max - min) / 2;              int place = 0;              int get = 0;                // Find count of elements smaller than mid              for(int i = 0; i < r; ++i)              {                    get = Arrays.binarySearch(m[i],mid);                    // If element is not found in the array the                  // binarySearch() method returns                  // (-(insertion\_point) - 1). So once we know                  // the insertion point we can find elements                  // Smaller than the searched element by the                  // following calculation                  if(get < 0)                      get = Math.abs(get) - 1;                    // If element is found in the array it returns                  // the index(any index in case of duplicate). So we go to last                  // index of element which will give  the number of                  // elements smaller than the number including                  // the searched element.                  else                  {                      while(get < m[i].length && m[i][get] == mid)                          get += 1;                  }                    place = place + get;              }                if (place < desired)                  min = mid + 1;              else                  max = mid;          }          return min;      }        // Driver Program to test above method.      public static void main(String[] args)      {          int r = 3, c = 3;          int m[][]= { {1,3,5}, {2,6,9}, {3,6,9} };            System.out.println("Median is " + binaryMedian(m, r, c));      }  } |

Output:

Median is 5

**9) Program for scalar multiplication of a matrix**

|  |
| --- |
| import java.io.\*;    class GFG {    static final int N = 3;  static void scalarProductMat(int mat[][],                                    int k)  {        // scalar element is multiplied      // by the matrix      for (int i = 0; i < N; i++)          for (int j = 0; j < N; j++)              mat[i][j] = mat[i][j] \* k;  }    // Driver code  public static void main (String[] args)  {      int mat[][] = { { 1, 2, 3 },                      { 4, 5, 6 },                      { 7, 8, 9 } };      int k = 4;        scalarProductMat(mat, k);        // to display the resultant matrix      System.out.println("Scalar Product Matrix is : ");        for (int i = 0; i < N; i++)      {          for (int j = 0; j < N; j++)              System.out.print(mat[i][j] + " ");          System.out.println();      }  }  } |

**Output:**

Scalar Product Matrix is :

4 8 12

16 20 24

28 32 36

**10) Find distinct elements common to all rows of a matrix**

|  |
| --- |
| import java.util.\*;    class GFG {        // function to individually sort      // each row in increasing order      public static void sortRows(int mat[][], int n)      {          for (int i=0; i<n; i++)              Arrays.sort(mat[i]);      }        // function to find all the common elements      public static void findAndPrintCommonElements(int mat[][],                                                       int n)      {          // sort rows individually          sortRows(mat, n);            // current column index of each row is stored          // from where the element is being searched in          // that row          int curr\_index[] = new int[n];            int f = 0;            for (; curr\_index[0]<n; curr\_index[0]++)          {              // value present at the current column index              // of 1st row              int value = mat[0][curr\_index[0]];                boolean present = true;                // 'value' is being searched in all the              // subsequent rows              for (int i=1; i<n; i++)              {                  // iterate through all the elements of                  // the row from its current column index                  // till an element greater than the 'value'                  // is found or the end of the row is                  // encountered                  while (curr\_index[i] < n &&                         mat[i][curr\_index[i]] <= value)                      curr\_index[i]++;                    // if the element was not present at the                  // column before to the 'curr\_index' of the                 // row                  if (mat[i][curr\_index[i]-1] != value)                      present = false;                    // if all elements of the row have                  // been traversed                  if (curr\_index[i] == n)                  {                      f = 1;                      break;                  }              }                // if the 'value' is common to all the rows              if (present)                 System.out.print(value+" ");                // if any row have been completely traversed              // then no more common elements can be found              if (f == 1)                  break;          }      }        /\* Driver program to test above function \*/      public static void main(String[] args)      {          int mat[][] = {  {12, 1, 14, 3, 16},                           {14, 2, 1, 3, 35},                           {14, 1, 14, 3, 11},                           {14, 25, 3, 2, 1},                           {1, 18, 3, 21, 14}                                              };                int n = 5;              findAndPrintCommonElements(mat, n);      }    } |

**Output:**

1 3 14

**Time Complexity:** O(n2log n), each row of size n requires O(nlogn) for sorting and there are total n rows.

**11) Print a given matrix in spiral form**

|  |
| --- |
| import java.io.\*;    class GFG {      // Function print matrix in spiral form      static void spiralPrint(int m, int n, int a[][])      {          int i, k = 0, l = 0;          /\*  k - starting row index          m - ending row index          l - starting column index          n - ending column index          i - iterator          \*/            while (k < m && l < n) {              // Print the first row from the remaining rows              for (i = l; i < n; ++i) {                  System.out.print(a[k][i] + " ");              }              k++;                // Print the last column from the remaining columns              for (i = k; i < m; ++i) {                  System.out.print(a[i][n - 1] + " ");              }              n--;                // Print the last row from the remaining rows \*/              if (k < m) {                  for (i = n - 1; i >= l; --i) {                      System.out.print(a[m - 1][i] + " ");                  }                  m--;              }                // Print the first column from the remaining columns \*/              if (l < n) {                  for (i = m - 1; i >= k; --i) {                      System.out.print(a[i][l] + " ");                  }                  l++;              }          }      }        // driver program      public static void main(String[] args)      {          int R = 3;          int C = 6;          int a[][] = { { 1, 2, 3, 4, 5, 6 },                        { 7, 8, 9, 10, 11, 12 },                        { 13, 14, 15, 16, 17, 18 } };          spiralPrint(R, C, a);      }  } |

**Output:**

1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11

**Time Complexity:** Time complexity of the above solution is O(mn).

**12) Find maximum element of each row in a matrix**

|  |
| --- |
| public class GFG{        // Function to get max element      public static void maxelement(int no\_of\_rows, int[][] arr) {          int i = 0;            // Initialize max to 0 at beginning          // of finding max element of each row          int max = 0;          int[] result = new int[no\_of\_rows];          while (i < no\_of\_rows) {              for (int j = 0; j < arr[i].length; j++) {                  if (arr[i][j] > max) {                      max = arr[i][j];                  }              }              result[i] = max;              max =0;              i++;            }          printArray(result);        }        // Print array element      private static void printArray(int[] result) {          for (int i =0; i<result.length;i++) {              System.out.println(result[i]);          }        }        // Driver code      public static void main(String[] args) {          int[][] arr = new int[][] { {3, 4, 1, 8},                                      {1, 4, 9, 11},                                      {76, 34, 21, 1},                                     {2, 1, 4, 5} };         // Calling the function          maxelement(4, arr);      }  } |

**Output :**

8

11

76

5

**13) Find unique elements in a matrix**

|  |
| --- |
| class GFG  {  static int R = 4, C = 4;    // function that calculate  // unique element  static void unique(int mat[][],                     int n, int m)  {      int maximum = 0, flag = 0;      for(int i = 0; i < n; i++)          for(int j = 0; j < m; j++)                // Find maximum element              // in a matrix              if(maximum < mat[i][j])                      maximum = mat[i][j];        // Take 1-D array of      // (maximum + 1) size      int b[] = new int [maximum + 1];      for(int i = 0 ; i < n; i++)          for(int j = 0; j < m; j++)              b[mat[i][j]]++;        //print unique element      for(int i = 1; i <= maximum; i++)          if(b[i] == 1)              System.out.print(i + " ");              flag = 1;        if(flag == 0)      {          System.out.println("No unique element " +                                  "in the matrix");      }  }    // Driver Code  public static void main(String args[])  {      int mat[][] = {{1, 2, 3, 20},                     {5, 6, 20, 25},                     {1, 3, 5, 6},                     {6, 7, 8, 15}};        // function that calculate      // unique element      unique(mat, R, C);  }  } |

**Output:**

2 7 8 25 15

**14) Print a given matrix in counter-clock wise spiral form**

|  |
| --- |
| import java.io.\*;    class GFG  {      static int R = 4;      static int C = 4;        // function to print the      // required traversal      static void counterClockspiralPrint(int m,                                          int n,                                          int arr[][])      {          int i, k = 0, l = 0;        /\* k - starting row index          m - ending row index          l - starting column index          n - ending column index          i - iterator \*/            // initialize the count          int cnt = 0;            // total number of          // elements in matrix          int total = m \* n;            while (k < m && l < n)          {              if (cnt == total)                  break;                // Print the first column              // from the remaining columns              for (i = k; i < m; ++i)              {                  System.out.print(arr[i][l] + " ");                  cnt++;              }              l++;                if (cnt == total)                  break;                // Print the last row from              // the remaining rows              for (i = l; i < n; ++i)              {                  System.out.print(arr[m - 1][i] + " ");                  cnt++;              }              m--;                if (cnt == total)                  break;                // Print the last column              // from the remaining columns              if (k < m)              {                  for (i = m - 1; i >= k; --i)                  {                      System.out.print(arr[i][n - 1] + " ");                      cnt++;                  }                  n--;              }                if (cnt == total)                  break;                // Print the first row              // from the remaining rows              if (l < n)              {                  for (i = n - 1; i >= l; --i)                  {                      System.out.print(arr[k][i] + " ");                      cnt++;                  }                  k++;              }          }      }    // Driver Code  public static void main(String[] args)  {      int arr[][] = { { 1, 2, 3, 4 },                      { 5, 6, 7, 8 },                      { 9, 10, 11, 12 },                      { 13, 14, 15, 16 } };        // Function calling      counterClockspiralPrint(R, C, arr);  }  } |

**Output :**

1 5 9 13 14 15 16 12 8 4 3 2 6 10 11 7

**Time Complexity :** O(mn).

**15) Maximum path sum in matrix**

|  |
| --- |
| import static java.lang.Math.max;    class GFG  {      public static int N = 4, M = 6;        // Function to calculate max path in matrix      static int findMaxPath(int mat[][])      {          // To find max val in first row          int res = -1;          for (int i = 0; i < M; i++)              res = max(res, mat[0][i]);            for (int i = 1; i < N; i++)          {              res = -1;              for (int j = 0; j < M; j++)              {                  // When all paths are possible                  if (j > 0 && j < M - 1)                      mat[i][j] += max(mat[i - 1][j],                                   max(mat[i - 1][j - 1],                                      mat[i - 1][j + 1]));                    // When diagonal right is not possible                  else if (j > 0)                      mat[i][j] += max(mat[i - 1][j],                                      mat[i - 1][j - 1]);                    // When diagonal left is not possible                  else if (j < M - 1)                      mat[i][j] += max(mat[i - 1][j],                                  mat[i - 1][j + 1]);                    // Store max path sum                  res = max(mat[i][j], res);              }          }          return res;      }        // driver program      public static void main (String[] args)      {          int mat[][] = { { 10, 10, 2, 0, 20, 4 },                          { 1, 0, 0, 30, 2, 5 },                          { 0, 10, 4, 0, 2, 0 },                          { 1, 0, 2, 20, 0, 4 }                      };            System.out.println(findMaxPath(mat));      }  } |

**Output:**

74

**Time Complexity:**O(N\*M)

**16) Squares of Matrix Diagonal Elements**

|  |
| --- |
| import java.io.\*;    class GFG  {      static int MAX =100;        // function of diagonal square      static void diagonalsquare(int mat[][], int row,                                           int column)      {          // This loop is for finding square of first          // diagonal elements          System.out.print( "Diagonal one : ");          for (int i = 0; i < row; i++)          {              for (int j = 0; j < column; j++)                    // if this condition will become true                  // then we will get diagonal element                  if (i == j)                        // printing square of diagonal element                      System.out.print ( mat[i][j] \* mat[i][j] +" ");          }          System.out.println();            // This loop is for finding square of second          // side of diagonal elements          System.out.print("Diagonal two : ");          for (int i = 0; i < row; i++)          {              for (int j = 0; j < column; j++)                    // if this condition will become true                  // then we will get second side diagonal                  // element                  if (i + j == column - 1)                        // printing square of diagonal element                      System.out.print(mat[i][j] \* mat[i][j] +" ");          }      }        // Driver code      public static void main (String[] args)      {          int mat[][] = { { 2, 5, 7 },                          { 3, 7, 2 },                          { 5, 6, 9 } };          diagonalsquare(mat, 3, 3);        }  } |

Output:

Diagonal one : 4 49 81

Diagonal two : 49 49 25

**Time Complexity**O(n \* n )

**17) Move matrix elements in given direction and add elements with same value**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class GFG {      // Function to shift the matrix      // in the given direction      static void moveMatrix(char d,                             int n,                             int a[][])      {            // For right shift move.          if (d == 'r') {                // for each row from              // top to bottom              for (int i = 0; i < n; i++) {                  ArrayList<Integer> v = new ArrayList<Integer>();                  ArrayList<Integer> w = new ArrayList<Integer>();                  int j;                    // for each element of                  // row from right to left                  for (j = n - 1; j >= 0; j--) {                      // if not 0                      if (a[i][j] != 0)                          v.add(a[i][j]);                  }                    // for each temporary array                  for (j = 0; j < v.size(); j++) {                      // if two element have                      // same value at                      // consecutive position.                      if (j < v.size() - 1 && v.get(j) == v.get(j + 1)) {                          // insert only one element                          // as sum of two same element.                          w.add(2 \* v.get(j));                          j++;                      }                      else                          w.add(v.get(j));                  }                    // filling the each                  // row element to 0.                  for (j = 0; j < n; j++)                      a[i][j] = 0;                    j = n - 1;                    // Copying the temporary                  // array to the current row.                  for (int it = 0; it < w.size(); it++)                      a[i][j--] = w.get(it);              }          }            // for left shift move          else if (d == 'l') {                // for each row              for (int i = 0; i < n; i++) {                  ArrayList<Integer> v = new ArrayList<Integer>();                  ArrayList<Integer> w = new ArrayList<Integer>();                  int j;                    // for each element of the                  // row from left to right                  for (j = 0; j < n; j++) {                      // if not 0                      if (a[i][j] != 0)                          v.add(a[i][j]);                  }                    // for each temporary array                  for (j = 0; j < v.size(); j++) {                      // if two element have                      // same value at                      // consecutive position.                      if (j < v.size() - 1 && v.get(j) == v.get(j + 1)) {                          // insert only one                          // element as sum                          // of two same element.                          w.add(2 \* v.get(j));                          j++;                      }                      else                          w.add(v.get(j));                  }                    // filling the each                  // row element to 0.                  for (j = 0; j < n; j++)                      a[i][j] = 0;                    j = 0;                    for (int it = 0; it < w.size(); it++)                      a[i][j++] = w.get(it);              }          }            // for down shift move.          else if (d == 'd') {              // for each column              for (int i = 0; i < n; i++) {                  ArrayList<Integer> v = new ArrayList<Integer>();                  ArrayList<Integer> w = new ArrayList<Integer>();                  int j;                    // for each element of                  // column from bottom to top                  for (j = n - 1; j >= 0; j--) {                      // if not 0                      if (a[j][i] != 0)                          v.add(a[j][i]);                  }                    // for each temporary array                  for (j = 0; j < v.size(); j++) {                        // if two element have                      // same value at consecutive                      // position.                      if (j < v.size() - 1 && v.get(j) == v.get(j + 1)) {                          // insert only one element                          // as sum of two same element.                          w.add(2 \* v.get(j));                          j++;                      }                      else                          w.add(v.get(j));                  }                    // filling the each                  // column element to 0.                  for (j = 0; j < n; j++)                      a[j][i] = 0;                    j = n - 1;                    // Copying the temporary array                  // to the current column                  for (int it = 0; it < w.size(); it++)                      a[j--][i] = w.get(it);              }          }            // for up shift move          else if (d == 'u') {              // for each column              for (int i = 0; i < n; i++) {                  ArrayList<Integer> v = new ArrayList<Integer>();                  ArrayList<Integer> w = new ArrayList<Integer>();                  int j;                    // for each element of column                  // from top to bottom                  for (j = 0; j < n; j++) {                      // if not 0                      if (a[j][i] != 0)                          v.add(a[j][i]);                  }                    // for each temporary array                  for (j = 0; j < v.size(); j++) {                      // if two element have                      // same value at                      // consecutive position.                      if (j < v.size() - 1 && v.get(j) == v.get(j + 1)) {                          // insert only one element                          // as sum of two same element.                          w.add(2 \* v.get(j));                          j++;                      }                      else                          w.add(v.get(j));                  }                    // filling the each                  // column element to 0.                  for (j = 0; j < n; j++)                      a[j][i] = 0;                    j = 0;                    // Copying the temporary                  // array to the current                  // column                  for (int it = 0; it < w.size(); it++)                      a[j++][i] = w.get(it);              }          }      }        // Driver Code      public static void main(String args[])      {          char d = 'l';          int n = 5;          int a[][] = { { 32, 3, 3, 3, 3 },                        { 0, 0, 1, 0, 0 },                        { 10, 10, 8, 1, 2 },                        { 0, 0, 0, 0, 1 },                        { 4, 5, 6, 7, 8 } };            moveMatrix(d, n, a);            // Printing the          // final array          for (int i = 0; i < n; i++) {              for (int j = 0; j < n; j++)                  System.out.print(a[i][j] + " ");                System.out.println();          }      }  } |

**Output:**

32 6 6 0 0

1 0 0 0 0

20 8 1 2 0

1 0 0 0 0

4 5 6 7 8

**Misc :**

**Examples:**

**1) A Product Array Puzzle**

class ProductArray {

/\* Function to print product array for a given array

arr[] of size n \*/

void productArray(int arr[], int n)

{

// Base case

if (n == 1) {

System.out.print(0);

return;

}

// Initialize memory to all arrays

int left[] = new int[n];

int right[] = new int[n];

int prod[] = new int[n];

int i, j;

/\* Left most element of left array is always 1 \*/

left[0] = 1;

/\* Rightmost most element of right array is always 1 \*/

right[n - 1] = 1;

/\* Construct the left array \*/

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

left[i] = arr[i - 1] \* left[i - 1];

/\* Construct the right array \*/

for (j = n - 2; j >= 0; j--)

right[j] = arr[j + 1] \* right[j + 1];

/\* Construct the product array using

left[] and right[] \*/

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

prod[i] = left[i] \* right[i];

/\* print the constructed prod array \*/

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

System.out.print(prod[i] + " ");

return;

}

/\* Driver program to test the aboe function \*/

public static void main(String[] args)

{

ProductArray pa = new ProductArray();

int arr[] = { 10, 3, 5, 6, 2 };

int n = arr.length;

System.out.println("The product array is : ");

pa.productArray(arr, n);

}

}

**Output:**

The product array is :

180 600 360 300 900

**2) Implement two stacks in an array**

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(5);

ts.push2(10);

ts.push2(15);

ts.push1(11);

ts.push2(7);

System.out.println("Popped element from" +

" stack1 is " + ts.pop1());

ts.push2(40);

System.out.println("Popped element from" +

" stack2 is " + ts.pop2());

}

}

Output:

Popped element from stack1 is 11

Popped element from stack2 is 40

**3) Minimum increment by k operations to make all elements equal**

import java.io.\*;

import java.util.Arrays;

class GFG {

// function for calculating min operations

static int minOps(int arr[], int n, int k)

{

// max elements of array

Arrays.sort(arr);

int max = arr[arr.length - 1];

int res = 0;

// iterate for all elements

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

// check if element can make equal to

// max or not if not then return -1

if ((max - arr[i]) % k != 0)

return -1;

// else update res for required operations

else

res += (max - arr[i]) / k;

}

// return result

return res;

}

// Driver program

public static void main(String[] args)

{

int arr[] = { 21, 33, 9, 45, 63 };

int n = arr.length;

int k = 6;

System.out.println(minOps(arr, n, k));

}

}

Output:

24