**Data Structure Interview Question- Array**

1. **Program for array rotation?**

Ans-

**METHOD 1 (Using temp array)**

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

1) Store d elements in a temp array

temp[] = [1, 2]

2) Shift rest of the arr[]

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

3) Store back the d elements

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

**Time complexity :** O(n)  
**Auxiliary Space :**O(d)

**METHOD 2 (Rotate one by one)**

leftRotate(arr[], d, n)

start

For i = 0 to i < d

Left rotate all elements of arr[] by one

end

**Time complexity :** O(n \* d)  
**Auxiliary Space :** O(1)

  /\*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;

    }

**Method 3 (The Reversal Algorithm) :**

**Algorithm :**

rotate(arr[], d, n)

reverse(arr[], 1, d) ;

reverse(arr[], d + 1, n);

reverse(arr[], 1, n)

**Example :**  
Let the array be arr[] = [1, 2, 3, 4, 5, 6, 7], d =2 and n = 7  
A = [1, 2] and B = [3, 4, 5, 6, 7]

* Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7]
* Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]
* Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

 /\* 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--;

        }

    }

1. **Search for an element in sorted and rotated array?**

Ans- The idea is to find the pivot point, divide the array in two sub-arrays and call binary searc

Input arr[] = {3, 4, 5, 1, 2}

Element to Search = 1

1) Find out pivot point and divide the array in two

sub-arrays. (pivot = 2) /\*Index of 5\*/

2) Now call binary search for one of the two sub-arrays.

(a) **If** element is greater than 0th element then

search in left array

(b) **Else** Search in right array

(1 will go in else as 1 < 0th element(3))

3) **If** element is found in selected sub-array then return index

**Else** return -1.

/\* Standard Binary Search function\*/

int binarySearch(int arr[], int low,

                  int high, int key)

{

  if (high < low)

    return -1;

  int mid = (low + high)/2; /\*low + (high - low)/2;\*/

  if (key == arr[mid])

    return mid;

  if (key > arr[mid])

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

  // else

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

}

/\* Function to get pivot. For array 3, 4, 5, 6, 1, 2

   it returns 3 (index of 6) \*/

int findPivot(int arr[], int low, int high)

{

  // base cases

  if (high < low) return -1;

  if (high == low) return low;

   int mid = (low + high)/2; /\*low + (high - low)/2;\*/

   if (mid < high && arr[mid] > arr[mid + 1])

    return mid;

   if (mid > low && arr[mid] < arr[mid - 1])

    return (mid-1);

   if (arr[low] >= arr[mid])

    return findPivot(arr, low, mid-1);

   return findPivot(arr, mid + 1, high);

}

/\* Searches an element key in a pivoted

   sorted array arr[] of size n \*/

int pivotedBinarySearch(int arr[], int n, int key)

{

  int pivot = findPivot(arr, 0, n-1);

  // If we didn't find a pivot,

  // then array is not rotated at all

  if (pivot == -1)

    return binarySearch(arr, 0, n-1, key);

  // If we found a pivot, then first compare with pivot

  // and then search in two subarrays around pivot

  if (arr[pivot] == key)

    return pivot;

  if (arr[0] <= key)

    return binarySearch(arr, 0, pivot-1, key);

    return binarySearch(arr, pivot+1, n-1, key);

}

1. **Given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.**

Ans-

**METHOD 1 (Use Sorting)**

**Algorithm :**

hasArrayTwoCandidates (A[], ar\_size, sum)

1) Sort the array in non-decreasing order.

2) Initialize two index variables to find the candidate elements in the sorted array.

(a) Initialize first to the leftmost index: l = 0

(b) Initialize second the rightmost index: r = ar\_size-1

3) Loop while l < r.

(a) If (A[l] + A[r] == sum) then return 1

(b) Else if( A[l] + A[r] < sum ) then l++

(c) Else r--

4) No candidates in whole array - return 0

**Time Complexity:** Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then (-)(nlogn) in worst case. If we use Quick Sort then O(n^2) in worst case.  
**Auxiliary Space :** Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and O(1) for Heap Sort.

**METHOD 2 (Use Hashing)**  
This method works in O(n) time.

1) Initialize an empty hash table s.

2) Do following for each element A[i] in A[]

(a) If s[x - A[i]] is set then print the pair (A[i], x - A[i])

(b) Insert A[i] into s.

static void printpairs(int arr[], int sum)

    {

        HashSet<Integer> s = new HashSet<Integer>();

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

            int temp = sum - arr[i];

            // checking for condition

            if (s.contains(temp)) {

                System.out.println("Pair with given sum " + sum + " is (" + arr[i] + ", " + temp + ")");

            }

            s.add(arr[i]);

        }

    }

**Time Complexity:**O(n)  
**Auxiliary Space:** O(n) where n is size of array.

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

**Ans-** A **Simple Solution** is to find all rotations one by one, check sum of every rotation and return the maximum sum. Time complexity of this solution is O(n2).

**Example:**

Input: arr[] = {1, 20, 2, 10}

Output: 72

We can 72 by rotating array twice.

{2, 10, 1, 20}

20\*3 + 1\*2 + 10\*1 + 2\*0 = 72

Input: arr[] = {10, 1, 2, 3, 4, 5, 6, 7, 8, 9};

Output: 330

We can 330 by rotating array 9 times.

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

0\*1 + 1\*2 + 2\*3 ... 9\*10 = 330

We can solve this problem in O(n) time using an **Efficient Solution**.  
Let Rj be value of i\*arr[i] with j rotations. The idea is to calculate next rotation value from previous rotation, i.e., calculate Rj from Rj-1.

Let us calculate initial value of i\*arr[i] with no rotation

R0 = 0\*arr[0] + 1\*arr[1] +...+ (n-1)\*arr[n-1]

After 1 rotation arr[n-1], becomes first element of array,

arr[0] becomes second element, arr[1] becomes third element and so on.

R1 = 0\*arr[n-1] + 1\*arr[0] +...+ (n-1)\*arr[n-2]

R1 - R0 = arr[0] + arr[1] + ... + arr[n-2] - (n-1)\*arr[n-1]

After 2 rotations arr[n-2], becomes first element of array,

arr[n-1] becomes second element, arr[0] becomes third element and so on.

R2 = 0\*arr[n-2] + 1\*arr[n-1] +...+ (n-1)\*arr[n-3]

R2 - R1 = arr[0] + arr[1] + ... + arr[n-3] - (n-1)\*arr[n-2] + arr[n-1]

= arr[0] + arr[1] + ... + arr[n-3] - (n)\*arr[n-2] + arr[n-2]+arr[n-1]

If we take a closer look at above values, we can observe below pattern

Rj - Rj-1 = arrSum - n \* arr[n-j]

Where arrSum is sum of all array elements, i.e.

Below is complete algorithm:

1) Compute sum of all array elements. Let this sum be 'arrSum'.

2) Compute R0 by doing i\*arr[i] for given array. Let this value be currVal.

3) Initialize result: maxVal = currVal // maxVal is result.

// This loop computes Rj from Rj-1

4) Do following for j = 1 to n-1

......a) currVal = currVal + arrSum-n\*arr[n-j];

......b) If (currVal > maxVal)

maxVal = currVal

5) Return maxVal

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

1. **Find the rotation count in Rotated sorted array?**

Ans- Consider an array is sorted in the ascending order

Input: arr[] = {10, 1, 2, 3, 4, 5, 6, 7, 8, 9};

Output: 330

We can 330 by rotating array 9 times.

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

0\*1 + 1\*2 + 2\*3 ... 9\*10 = 330

If we take closer look at examples, we can notice that the number of rotations is equal to index of minimum element.

**Method 1 -Using linear search**

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;

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

**Method 2 (Efficient Using**[**Binary Search**](http://quiz.geeksforgeeks.org/binary-search/)**)**

* The minimum element is the only element whose previous is greater than it. If there is no previous element, then there is no rotation (first element is minimum). We check this condition for middle element by comparing it with (mid-1)’th and (mid+1)’th elements.
* If the minimum element is not at the middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.
  1. If middle element is smaller than last element, then the minimum element lies in left half
  2. Else minimum element lies in right half.

  static int countRotations(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 0;

        // If there is only one element left

        if (high == low)

            return low;

        // Find mid

        // /\*(low + high)/2;\*/

        int mid = low + (high - low)/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 (mid + 1);

        // Check if mid itself is minimum element

        if (mid > low && arr[mid] < arr[mid - 1])

            return mid;

        // Decide whether we need to go to left

        // half or right half

        if (arr[high] > arr[mid])

            return countRotations(arr, low, mid - 1);

        return countRotations(arr, mid + 1, high);

    }

**Time Complexity :**O(Log n)  
**Auxiliary Space :**O(1)

1. **Find minimum element in a sorted and rotated array?**

Ans-

A simple solution is to traverse the complete array and find minimum. This solution requires O(n) time.  
We can do it in O(Logn) using Binary Search. If we take a closer look at above examples, we can easily figure out following pattern:

* The minimum element is the only element whose previous is greater than it. If there is no previous element element, then there is no rotation (first element is minimum). We check this condition for middle element by comparing it with (mid-1)’th and (mid+1)’th elements.
* If minimum element is not at middle (neither mid nor mid + 1), then minimum element lies in either left half or right half.
  1. If middle element is smaller than last element, then the minimum element lies in left half
  2. Else minimum element lies in right half.

1. **Swap the array**

Ans-

Method 1- **Iterative way :**

*1) Initialize start and end indexes as start = 0, end = n-1  
2) In a loop, swap arr[start] with arr[end] and change start and end as follows :  
start = start +1, end = end – 1*

**Time Complexity :** O(n)

**Recursive Way :**

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

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

{

    if (start >= end)

    return;

    int temp = arr[start];

    arr[start] = arr[end];

    arr[end] = temp;

    // Recursive Function calling

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

}

1. **To reverse the digits in the no?**

Ans-

Input: num

(1) Initialize rev\_num = 0

(2) Loop while num > 0

(a) Multiply rev\_num by 10 and add remainder of num

divide by 10 to rev\_num

rev\_num = rev\_num\*10 + num%10;

(b) Divide num by 10

(3) Return rev\_num

**Time Complexity:** O(Log(n)) where n is the input number.

However, if the number is large such that the reverse overflows, the output is some garbage value. If we run the code above with input as any large number say **1000000045**, then the output is some garbage value like **1105032705** or any other garbage value. See [this](https://ide.geeksforgeeks.org/moL71L) for the output.

How to handle overflow?  
The idea is to store previous value of the sum can be stored in a variable which can be checked every time to see if the reverse overflowed or not.

/\* Iterative function to reverse digits of num\*/

    static int reversDigits(int num)

    {

        // Handling negative numbers

        boolean negativeFlag = false;

        if (num < 0)

        {

            negativeFlag = true;

            num = -num ;

        }

        int prev\_rev\_num = 0, rev\_num = 0;

        while (num != 0)

        {

            int curr\_digit = num%10;

            rev\_num = (rev\_num\*10) + curr\_digit;

            // checking if the reverse overflowed or not.

            // The values of (rev\_num - curr\_digit)/10 and

            // prev\_rev\_num must be same if there was no

            // problem.

            if ((rev\_num - curr\_digit)/10 != prev\_rev\_num)

            {

                System.out.println("WARNING OVERFLOWED!!!");

                return 0;

            }

            prev\_rev\_num = rev\_num;

            num = num/10;

        }

        return (negativeFlag == true)? -rev\_num : rev\_num;

    }