Arrays

> *1. WAP to `add all elements` of Array.*

```
public static int sumOfElement(int [] arr)
{
    int sum=0;
    for(int s:arr) {
        sum+=s;
    }
    return sum;
}
```

Method Tracing: `sumOfElement(int[] arr)`

Purpose

This method calculates the sum of all elements in an integer array.

Key Features

- Uses enhanced for-loop (for-each) to iterate through array
- Accumulates sum in `sum` variable
- Handles empty arrays (returns 0)
- Works with both positive and negative numbers

Tracing Steps

Example 1: `arr = {1, 2, 3, 4}`

Iteration	Current Element (s)	sum (before)	sum (after)
1	1	0	1
2	2	1	3
3	3	3	6
4	4	6	10

Final Result: `10`

Example 2: `arr = {-5, 10, 3}`

Iteration	Current Element (s)	sum (before)	sum (after)
1	-5	0	-5
2	10	-5	5
3	3	5	8

Final Result: `8`

Example 3: `arr = {}` **(Empty Array)**

- Loop doesn't execute (array length = 0)
- Returns initialized value

Final Result: `0`

Key Characteristics

1. **Iteration**:
    - Processes each element exactly once
    - Order of processing follows array order
2. **Summation**:
    - Initializes sum to 0
    - Adds each element to running total
3. **Edge Handling**:
    - Empty arrays return 0
    - Handles all integer values (-2^{31} to $2^{31}-1$)
    - Potential integer overflow with large numbers

Time Complexity

- O(n) where n is array length
- Each element processed exactly once

Space Complexity

- O(1) constant space (only sum variable used)

> *2. WAP to print* `Even Number` *from Array.*

```
public static void evenElement(int [] arr)
{
    for(int i:arr) {
        if(i%2==0) {
            System.out.println(i);
        }
    }
}
```

Method Tracing: `evenElement(int [] arr)`

Method Purpose

Prints all even numbers from an integer array.

Key Characteristics

1. **Input**: Accepts an integer array
2. **Processing**:
    - Uses enhanced for-loop to iterate through elements

- Checks if each element is even using `i % 2 == 0`
- Prints even numbers immediately

3. **Output**: Each even number on a new line

Execution Flow Example

For `arr = {3, 4, -2, 7, 0, 10}`:

Iteration	Current Value (i)	i%2	Print?	Output So Far
1	3	1	No	
2	4	0	Yes	4
3	-2	0	Yes	4 -2
4	7	1	No	4 -2
5	0	0	Yes	4 -2 0
6	10	0	Yes	4 -2 0 10

Time Complexity

- O(n) - Processes each element exactly once
- Each operation inside loop is O(1)

Space Complexity

- O(1) - Only uses constant extra space

> *3. WAP to `reverse each and every` Element of Array.*

```
//for reversing each digit
public static int reverseDigit(int n)
{
    int revNum=0;
    while(n!=0){
        revNum=revNum*10+n%10;
        n/=10;
    }
    return revNum;
}

public static void reverseArray(int[] arr)
{
    for(int i=0;i<arr.length;i++){
```

```
            arr[i]=reverseDigit(arr[i]);
        }
    }
```

Method Tracing: `reverseArray(int[] arr)`

reverseDigit(int n)

Purpose:

Reverses digits of an integer (e.g., 123 → 321)

Execution Flow (n = 1234):

Iteration	n	n%10	revNum Before	revNum After	n After /=10
1	1234	4	0	4	123
2	123	3	4	43	12
3	12	2	43	432	1
4	1	1	432	4321	0

Returns: 4321

Characteristics:

- Positive integers only
- Returns 0 for input 0
- Converts 100 → 1
- Time: O(d) (digits count)
- Space: O(1)

reverseArray(int[] arr)

Purpose:

Reverses each element's digits in array

Execution Flow (arr = [12, 34, 500]):

Iteration	i	arr[i] Before	reverseDigit()	arr After
1	0	12	21	[21, 34, 500]
2	1	34	43	[21, 43, 500]
3	2	500	5	[21, 43, 5]

Final Array: [21, 43, 5]

Characteristics:

- Modifies original array
- Handles each element independently
- Time: O(n*d) (n=array length, d=avg digits)

- Space: O(1) in-place

> *4.WAP to Store All the* `Palindromic Number into new Array` .

```
//Checking Palindrom Number
public static boolean isPalindrome(int n)
{
    int temp = n;
    int rev = 0;

    while (n != 0) {
        rev = rev * 10 + n % 10;
        n /= 10;
    }

    return temp == rev;
}
//Counting new Array Length
public static int countPalindromicNumbers(int[] arr)
{
    int count = 0;

    for (int num : arr) {
        if (isPalindrome(num)) {
            count++;
        }
    }
    return count;
}
public static int[] findPalindromicNumbers(int[] arr)
{
    int[] temp = new int[countPalindromicNumbers(arr)];
    if(temp.length==0)
        return temp;
    int i=0;
    for (int num : arr) {
        if (isPalindrome(num)) {
            temp[i++]=num;
        }
    }
    return temp;
}
```

> *5. WAP to print* `Prime Number` *from Array.*

```
//Checking Prime Number
public static boolean isPrime(int n)
{
    if (n <= 1)
        return false;
    if (n == 2)
```

```
            return true;
        if (n % 2 == 0)
            return false;
        for (int i = 3; i <= Math.sqrt(n); i+=2)
        {
            if (n % i == 0)
                return false;
        }
        return true;
    }

    public static void printPrimeNumber(int[] arr)
    {
        for(int i : arr){
            if(isPrime(i))
                System.out.print(i+" ");
        }
    }
```

Method Tracing: `printPrimeNumber(int[] arr)`

1. isPalindrome(int n)

Purpose:

Checks if a number reads the same backward as forward (e.g., 121)

Execution Flow (n = 121):

Iteration	n	n%10	rev Before	rev After	n After /=10
1	121	1	0	1	12
2	12	2	1	12	1
3	1	1	12	121	0

Returns: true (121 == 121)

Characteristics:

- Handles positive integers
- Time: O(d) where d = digit count
- Space: O(1)
- Preserves original number via temp variable

Edge Cases:

- 0 → true
- 10 → false
- -121 → false (negative numbers never palindromic)

2. countPalindromicNumbers(int[] arr)

Purpose:

Counts how many palindromic numbers exist in an array

Execution Flow (arr = [121, 123, 1331]):

Iteration	num	isPalindrome	count
1	121	true	1
2	123	false	1
3	1331	true	2

Returns: 2

Characteristics:

- Time: O(n*d) where n=array length
- Space: O(1)
- Doesn't modify input array

3. findPalindromicNumbers(int[] arr)

Purpose:

Creates new array containing only palindromic numbers

Execution Flow (arr = [121, 123, 1331]):

1. First calls countPalindromicNumbers → gets count=2
2. Creates temp array of size 2
3. Fills array:

Iteration	num	isPalindrome	temp Array State
1	121	true	[121, 0]
2	123	false	[121, 0]
3	1331	true	[121, 1331]

Returns: [121, 1331]

Characteristics:

- Time: O(2n*d) → counts then filters
- Space: O(k) where k=palindrome count

> *6. WAP to print `Largest Number` from Array.*

```
public static void largestNumber(int[] arr)
{
    int max = Integer.MIN_VALUE;
    for (int i : arr) {
        if (max < i)
            max = i;
    }
```

```
        System.out.println(max);
    }
```

Method Tracing: `largestNumber(int[] arr)`

Purpose:

Finds and prints the largest number in an integer array

Key Characteristics:

- Initializes `max` to minimum possible integer value
- Iterates through array elements
- Updates `max` when larger value found
- Prints final maximum value

Execution Flow (arr = [12, 45, 9, 32]):

Iteration	Current Element (i)	max Before	Condition (max < i)	max After
1	12	-2147483648	true	12
2	45	12	true	45
3	9	45	false	45
4	32	45	false	45

Output: 45

Time Complexity:

- O(n) where n is array length
- Each element examined exactly once

Space Complexity:

- O(1) - uses constant extra space

> *7. WAP to print `smallest number` from Array.*

```
    public static void smallestNumber(int[] arr)
    {
        int min = Integer.MAX_VALUE;
        for (int i : arr) {
            if (min > i)
                min = i;
        }
        System.out.println(min);
    }
```

Method Tracing: `smallestNumber(int[] arr)`

Execution Flow (arr = [12, 45, 9, 32]):

Iteration	Current Value (i)	min Before	Condition (min > i)	min After
1	12	2147483647	true	12
2	45	12	false	12
3	9	12	true	9
4	32	9	false	9

Final Output:
9

Characteristics:

- Initializes min to Integer.MAX_VALUE (∞)
- Single pass through array (O(n) time)
- Constant space usage (O(1))
- Handles all integer values (-2^{31} to $2^{31}-1$)

> *8. WAP to print `2nd Largest Number` from Array.*

```
public static void secondLargestNumber(int[] arr)

{

    if (arr == null || arr.length < 2) {
    System.out.println("Invalid input - need at least 2 distinct numbers");
    return;
}
    int max1 = Integer.MIN_VALUE;
    int max2 = Integer.MIN_VALUE;
    for (int i : arr) {
        if (i > max1)
            max1 = i;
    }
    for (int i : arr) {
        if (i > max2 && i != max1)
            max2 = i;
    }
     if (max2 == Integer.MIN_VALUE) {
    System.out.println("No distinct second largest");
    }
    else {
    System.out.println(max2);
}
}
```

Method Tracing: `secondLargestNumber(int[] arr)`

Execution Flow (arr = [12, 45, 9, 32, 45]):

First Pass (Find max1):

Iteration	Current Value (i)	max1 Before	Condition (i > max1)	max1 After
1	12	-2147483648	true	12
2	45	12	true	45
3	9	45	false	45
4	32	45	false	45
5	45	45	false	45

Second Pass (Find max2):

Iteration	Current Value (i)	max2 Before	Condition (i > max2 && i != max1)	max2 After
1	12	-2147483648	true	12
2	45	12	false (i == max1)	12
3	9	12	false	12
4	32	12	true	32
5	45	32	false (i == max1)	32

Final Output:
32

Characteristics:

- Uses two separate passes through the array
- First pass finds the absolute maximum (max1)
- Second pass finds the largest number excluding max1
- Time Complexity: O(2n) → O(n) (two linear passes)
- Space Complexity: O(1) (uses only two extra variables)

> *9. WAP to print `2nd smallest number` from Array.*

```
public static void secondSmallestNumber(int[] arr)
{

    if (arr == null || arr.length < 2) {
    System.out.println("Invalid input - need at least 2 distinct numbers");
    return;
    }
    int min1 = Integer.MAX_VALUE;
    int min2 = Integer.MAX_VALUE;
    for (int i : arr) {
        if (i < min1)
            min1 = i;
```

```
        }
        for (int i : arr) {
            if (i < min2 && i != min1)
                min2 = i;
        }
        if (min2 == Integer.MAX_VALUE) {
        System.out.println("No distinct second smallest number exists");
    } else {
        System.out.println("Second smallest number: " + min2);
    }

    }
```

Method Tracing: `secondSmallestNumber(int[] arr)`

Purpose:

Finds and prints the second smallest distinct number in an integer array.

Execution Flow (arr = [12, 45, 9, 1, 32, 1]):

First Pass (Find min1 - absolute minimum):

Iteration	Current Value (i)	min1 Before	Condition (i < min1)	min1 After
1	12	2147483647	true	12
2	45	12	false	12
3	9	12	true	9
4	1	9	true	1
5	32	1	false	1
6	1	1	false	1

Second Pass (Find min2 - second smallest):

Iteration	Current Value (i)	min2 Before	Condition (i < min2 && i != min1)	min2 After
1	12	2147483647	true	12
2	45	12	false	12
3	9	12	true	9
4	1	9	false (i == min1)	9
5	32	9	false	9
6	1	9	false (i == min1)	9

Final Output:
9

Characteristics:

- Uses two separate passes through the array
- First pass finds the absolute minimum (min1)
- Second pass finds the smallest number excluding min1
- Time Complexity: O(2n) → O(n) (two linear passes)
- Space Complexity: O(1) (uses only two extra variables)

> *10. WAP to move all the element `Zero(0)` to last in Array.*

```
public static void moveZero(int[] arr){
    for(int i=0,j=0;i<arr.length;i++){
        if(arr[i]==1){
            arr[i]=0;
            arr[j++]=1;
        }
    }
}
```

Method Tracing: `moveZero(int[] arr)`

Purpose:

Moves all 1s to the front of the array while replacing their original positions with 0s (effectively a "move ones forward" operation)

Key Characteristics:

- Uses two pointers technique (i for scanning, j for placement)
- Modifies the array in-place
- Preserves relative order of 1s
- Replaces original 1 positions with 0s

Execution Flow (arr = [0, 1, 0, 1, 1, 0, 1]):

Iteration	i	arr[i]	j	Condition (arr[i]==1)	Array State Before	Operation	Array State After
0	0	0	0	false	[0,1,0,1,1,0,1]	none	[0,1,0,1,1,0,1]
1	1	1	0	true	[0,1,0,1,1,0,1]	arr[0]=1, j=1	[1,0,0,1,1,0,1]
2	2	0	1	false	[1,0,0,1,1,0,1]	none	[1,0,0,1,1,0,1]
3	3	1	1	true	[1,0,0,1,1,0,1]	arr[1]=1, j=2	[1,1,0,0,1,0,1]
4	4	1	2	true	[1,1,0,0,1,0,1]	arr[2]=1, j=3	[1,1,1,0,0,0,1]
5	5	0	3	false	[1,1,1,0,0,0,1]	none	[1,1,1,0,0,0,1]

6	6	1	3	true	[1,1,1,0,0,0,1]	arr[3]=1, j=4	[1,1,1,1,0,0,0]

Final Array State: [1, 1, 1, 1, 0, 0, 0]

Time Complexity:

- O(n) - Single pass through the array
- Each element is examined exactly once

Space Complexity:

- O(1) - Operates in-place with constant extra space

> *11. WAP to Sort an Array in Accending Order.*

```
public static void sortAccendingOrder(int[] arr)
{
    for(int i=0;i<arr.length;i++){
        for(int j=i+1;j<arr.length;j++){
            if(arr[i]>arr[j])
                arr[j]=arr[j]+arr[i]-(arr[i]=arr[j]);
        }
    }
}
```

Method Tracing: `sortAccendingOrder(int[] arr)`

Purpose:

Sorts an integer array in ascending order using a nested loop swap technique

Key Characteristics:

- Implements bubble sort-like comparison
- Uses in-place swapping without temporary variable
- Modifies the original array
- Sorts in $O(n^2)$ time complexity

Execution Flow (arr = [4, 2, 5, 1]):

Initial Array: [4, 2, 5, 1]

Outer Loop (i):

i	j Range	Comparisons & Swaps	Array State After Iteration
0	1-3	4>2 → swap [2,4,5,1] 2>5 → no 2>1 → swap [1,4,5,2]	[1, 4, 5, 2]
1	2-3	4>5 → no 4>2 → swap [1,2,5,4]	[1, 2, 5, 4]

2	3	5>4 → swap [1,2,4,5]	[1, 2, 4, 5]
3	-	(no j iterations)	[1, 2, 4, 5]

Final Array: [1, 2, 4, 5]

Swap Mechanism Explained:

The line `arr[j]=arr[j]+arr[i]-(arr[i]=arr[j])` performs:

1. Stores sum in arr[j] temporarily
2. Assigns arr[i] to arr[j] (right side of subtraction)
3. Subtracts new arr[i] from sum to get original arr[j]

Time Complexity:

- Worst/Average Case: $O(n^2)$ - Nested loops
- Best Case: $O(n^2)$ - Even if sorted, still checks all pairs

Space Complexity:

- O(1) - In-place sorting, no additional storage

> *12. WAP to Sort an Array in Decending Order.*

```
public static void sortDecendingOrder(int[] arr)
{
    for(int i=0;i<arr.length;i++){
        for(int j=i+1;j<arr.length;j++){
            if(arr[i]<arr[j]){
                arr[j]=arr[j]+arr[i]-(arr[i]=arr[j]);
            }
        }
    }
}
```

Method Tracing: `sortDecendingOrder(int[] arr)`

Purpose:

Sorts an integer array in descending order using a nested loop swap technique

Key Characteristics:

- Implements selection sort-like comparison
- Uses in-place swapping without temporary variable
- Modifies the original array
- Sorts in $O(n^2)$ time complexity

Execution Flow (arr = [3, 1, 4, 2]):

Initial Array: [3, 1, 4, 2]

Outer Loop (i):

i	j Range	Comparisons & Swaps	Array State After Iteration
0	1-3	3<4 → swap [4,1,3,2] 4<1 → no 4<2 → no	[4, 1, 3, 2]
1	2-3	1<3 → swap [4,3,1,2] 3<2 → no	[4, 3, 1, 2]
2	3	1<2 → swap [4,3,2,1]	[4, 3, 2, 1]
3	-	(no j iterations)	[4, 3, 2, 1]

Final Array: [4, 3, 2, 1]

Swap Mechanism Explained:

The line `arr[j]=arr[j]+arr[i]-(arr[i]=arr[j])` performs:

1. Stores sum in arr[j] temporarily
2. Assigns arr[i] to arr[j] (right side of subtraction)
3. Subtracts new arr[i] from sum to get original arr[j]

Time Complexity:

- Worst/Average Case: $O(n^2)$ - Nested loops
- Best Case: $O(n^2)$ - Even if sorted, still checks all pairs

Space Complexity:

- O(1) - In-place sorting, no additional storage

> *13. WAP to Remove duplicate from an Array.*

```
public static void removeDuplicate(int[] arr){
       int count=0;
       for(int i=0;i<arr.length;i++){
           for(int j=i+1;j<arr.length;j++){
               if(arr[i]==-1)
                   break;
               else if(arr[i]!=-1 && arr[i]==arr[j]){
                   arr[j]=-1;
                   count++;
               }

           }
       }
       int[] newArray = new int[arr.length-count];
       count=0;
       for(int i=0;i<arr.length;i++){
           if(arr[i]!=-1){
               newArray[count++]=arr[i];
           }
```

```
        }
        System.out.println(Arrays.toString(newArray));
    }
```

Method Tracing: `removeDuplicate(int[] arr)`

Purpose:

Removes duplicate values from an array by marking duplicates with -1 and creating a new array without them

Key Characteristics:

- Uses nested loops to find duplicates
- Marks duplicates with -1
- Creates a new array without marked elements
- Preserves original order of unique elements
- Prints the result

Execution Flow (arr = [2, 3, 2, 4, 3, 5]):

Phase 1: Mark Duplicates

i	j Range	Comparisons	Array State	Count
0	1-5	2==3: no 2==2: yes (mark arr[2]=-1)	[2,3,-1,4,3,5]	1
1	2-5	3==-1: skip 3==4: no 3==3: yes (mark arr[4]=-1) 3==5: no	[2,3,-1,4,-1,5]	2
2	3-5	-1: skip	No changes	2
3	4-5	4==-1: skip 4==5: no	No changes	2
4	5	-1==5: skip	No changes	2
5	-	-	Final marked array: [2,3,-1,4,-1,5]	2

Phase 2: Create New Array

- New array length: 6 - 2 = 4
- Copy non -1 elements:
  - arr[0]=2 → newArray[0]=2
  - arr[1]=3 → newArray[1]=3
  - arr[3]=4 → newArray[2]=4
  - arr[5]=5 → newArray[3]=5

Final Output:
[2, 3, 4, 5]

Time Complexity:

- Phase 1: $O(n^2)$ - Nested loops
- Phase 2: O(n) - Single pass
- **Total:** $O(n^2)$

Space Complexity:

- O(n) - Additional array for results

> *14. WAP to Find EquilibriumPostion in a Array.*

```
public static int findEquilibrium(int[] arr) {
    if (arr == null || arr.length == 0) {
        return -1; // Indicate no equilibrium
    }

    int totalSum = 0;
    for (int num : arr) {
        totalSum += num;
    }

    int leftSum = 0;
    for (int i = 0; i < arr.length; i++) {
        totalSum -= arr[i]; // rightSum = totalSum - leftSum - arr[i]
        if (leftSum == totalSum) {
            return i;
        }
        leftSum += arr[i];
    }

    return -1; // No equilibrium found
}
```

Method Tracing: `equilibriumPostion(int[] arr)`

Purpose:

Finds an equilibrium position in an array where the sum of elements before the index equals the sum of elements after the index

Key Characteristics:

- Uses two-pointer approach (start and end)
- Accumulates sums from both ends
- Returns the equilibrium index when found
- Works in O(n) time with O(1) space

Execution Flow (arr = [1, 3, 5, 2, 2]):

Iteration	i	j	leftSum	rightSum	Condition (leftSum < rightSum)	Action	Array Visualization

0	0	4	0	0	false	rightSum+=arr[4]=2, j=3	[1,3,5,2,2]
1	0	3	0	2	true	leftSum+=arr[0]=1, i=1	[1,3,5,2,2]
2	1	3	1	2	true	leftSum+=arr[1]=4, i=2	[1,3,5,2,2]
3	2	3	4	2	false	rightSum+=arr[3]=4, j=2	[1,3,5,2,2]

Time Complexity:

- Original: O(n) single pass
- Improved: O(n) two passes (still linear)

Space Complexity:

- O(1) constant space