Basic Maths

Problem -1 :
Count all digits of a number.
(return 1 if the number = 0).

Pseudocode -

example 5627 = given number.

10	5627
10	562
10	56
10	5
	0

↷ /10

10 x 0 = 0

```
function (num) {
    if (num == 0) return 1;
    count = 0
    while (num > 0) {
        num = num//10;
        count ++;
    }
}
```

Dry Run -

1st run	2nd run	3rd run	4th run
5627 > 0	562 > 0	56 > 0	5 > 0
count = 1	count = 2	count = 3	count = 4
num = 562	num = 56	num = 5	num = 0

Time Complexity - determined by the no. of times we are running the while loop.

TC → O(digits)

SC → O(1) constant in nature.

Alternative Approach -

no. of digits = $[\log_{10}(num) + 1]$.

→ example →

100 → 3 digits = $\log_{10} 10^2 + 1$

$= 2 + 1$

$= 3$

Hence, digits = $\log_{10}(num) + 1$.

So,

TC → $O(\log_{10}(num) + 1)$.

Since 1 is a constant, so we ignore it.

TC → $O(\log_{10}(num))$.

TC → $O(\log_{10}(N))$ - In every iterati. we are dividing by 10.

SC → O(1) - Using a couple of variables., i.e., constant space.

Problem - 2 :

Count the no. of odd digits in a number.

Pseudocode –

10	5627
10	562
10	56
10	5
	0

num = num / 10
↳ removes the last digit.

```
function (num) {  // no. of odd digits.
    if (num == 0) return 0 ; int count = 0 ;
    while (num > 0) {
        int digit = num % 10 ;
        if (digit % 2 != 0)  count++ ;
        num = num / 10 ;
    }
    return count ;
}
```

digit % 2 == 1.

Dry Run →

1st run :
5627 > 0
digit = 7
count = 1

2nd run :
562
digit = 2
count = 1

3rd run :
56
digit = 6
count = 1

4th run :
5
digit = 5
count = 2

5th run :

Stop

Time Complexity →

$O(digits) = O(\log_{10}(N))$ since in every iteration, we are dividing N by 10.

Space Complexity → $O(1)$.

Problem - 3 :
Reverse a Number

Pseudocode -

Example 5627 → returns 7265.

num = num /10 → 5627 → 562 → 56 → 5.

reverseNum. = 0
lastDigitOfNum = num % 10;
reverseNum = reverseNum x 10 + lastDigitOfNum;

→ function (num) {
int reverseNum = 0;
while (num > 0) {
lastDigitOfNum = num % 10;
reverseNum = reverseNum * 10 + lastDigitOfNum;
}
return reverseNum;
} Again, TC → $O(\log_{10}(N))$ SC → $O(1)$.

Problem-4 :
Palindrome Number.

Pseudocode – Example : 6336 – True.
101 – True.

```
bool function (num) {
        revNumber = reverse (num);
        return (num == revNumber);
}
```

If we write the logic of reversing inside this, we will end up destroying our num.

pass by copy so num will stay as it is and won't get destroyed.

Code –

```
class Solution {
public :
    int reverseNum (int num) {
        int revNum = 0;
        // num will get destroyed
        while (num > 0) {
            int lastDigitOfNum = num % 10;
            revNum = revNum * 10 +
                     lastDigitOfNum;
            num = num / 10;
        }
        return revNum;
    }
```

```
bool isPalindrome (int n) {
        int revNum = reverseNum(n);
// pass by value or say pass by copy.
        return (revNum == n);
}
```

$TC \rightarrow O(\log_{10}(N))$.
$SC \rightarrow O(1)$.

Problem - 5 :

Return the largest Digit in a Number.

Pseudo Code –

To iterate through all the Digits, we write `num = num / 10;`

```
function (num) {
    largest = 0;
    while (num > 0) {
        lastDigit = num % 10;
        if (lastDigit > largest) {
            largest = lastDigit;
        }
    }
    return largest;
}
```

$TC \rightarrow O(\log_{10}(num))$, $SC \rightarrow O(1)$

Problem 7 :
Factorial of a given Number.

PseudoCode –

Example – let n = 3 fact = n
n = n-1 = 2 fact = fact * n.

```
function factorial (int n) {
        int factorial = 1;
        while (n > 0) {
                factorial = factorial * n;
                n = n - 1;
        }
        return factorial;
}
```

Another Approach –

```
function (n) {    //data-type of ans can
int ans = 1;  // be long or long long
for (int i = 1 → N) {  // as well.
      ans = ans * i;
}
return ans;
}
```

↳ loop runs N times. ↳ 'ans' variable

TC → O(N) SC → O(1).

Problem - 8 : (cool question).

Check if the number is armstrong.

Armstrong Number is a number which is equal to the sum of - the digits raised to the no. of digits.

Pseudocode - n = 153

Count = $\log_{10}(num) + 1$.

```
function (n) {
    int sum = 0;   int original = n;
    int power =    countDigits (n);
    while (n > 0)  {
        int lastDigit Of N =  n % 10;
        sum = sum + pow (last Digit Of N,
                                power);

        n = n / 10;
    }
    return (sum == original);
}

function  countDigits (n) {
    int count = 0;
    if (n == 0) return 1;
    while (n > 0) {
        count ++;
        n = n / 10;
    }
}
```

pow(number, power).

Note – The 'pow()' function has the TC of $\log_2(power)$.

Here, in this case, TC is O(no. of times while loop runs × TC of pow function).
Thus, TC = O(digits × $\log_2$ digits).
where digits = $\log_{10}(num) + 1$.

Hence, TC = $O(\log_{10} N \times \log_2(\log_{10} N + 1))$.
and SC = O(1).

Problem-9 :
Check for Perfect Numbers.

A perfect number is a number whose proper divisors add up to the number itself.
Proper divisors are the divisors excluding the number itself.

Example : 6 → {1, 2, 3}
1 + 2 + 3 = 6.
28 → {1, 2, 4, 7, 14}
1 + 2 + 4 + 7 + 14 = 28.

Pseudocode –

```
function (num) {
    int sum = 0;
    for (int i = 1 → num - 1) {
        if (num % i == 0) {
            // is a divisor
            sum = sum + i;
        }
    return (sum == num);
}
```

TC → O(N-1) ≈ O(N).
SC → O(1).

Problem – 6

Divisors of a Number.
Return the vector of all the divisors of a number.
Example – num = 6 → [1, 2, 3, 6].
Pseudo Code –

```
vector<int> divisors (int num) {
    vector<int> divisors;
    for (int i = 1 → num) {
        if (num % i == 0) {
            divisors.push_back (i);
        }
    } return divisors; }
```

TC → O(N) – iterating N times. for loop.
SC → O(sqrt(N)) – a number N can have at max 2*sqrt(N) divisors which are stored in the array.

Problem-9 : (Optimized Approach).
Check for the Perfect Number.
(number whose proper divisors add up to the number itself).

Pseudocode –
Let num = 36. → divisors are :

$\sqrt{36} = 6$

i	num/i
1	36 → num/1
2	18 → num/2
3	12 → num/3
4	9 → num/4
6	6 → num/6.

Let num = 48. → divisors are :

$\sqrt{48} = 4\sqrt{3}$
$= 6.928.$

1	48
2	24
3	16
4	12
6	8

We can reduce the time-complexity of our code by not running the loop from i=1 all the way till the number itself. Hence it's better to utilise i and count the other divisor as num/i.

So, we need to go till sqrt(num) and not till num itself (till i * i <= num). And count the other divisor as num/i, if i and num/i are not equal.
Also, we need to include 1 in our sum but not the number itself as we are summing only the proper divisors.

```
function (num) {
    sum = 1
    for(i = 2 ⟶ sqrt (num)) {
        if (num % i == 0) {
            sum = sum + i
            if (i != (num/i)) sum = sum + n/i
        }
    }
    return (sum == num)
}
```

Note - sqrt() fn. will end up taking a bit of time, hence we can insted write i * i <= num (insted of i <= sqrt(num))

TC → O(sqrt(N)) - loop is running for sqrt(N) times.
SC → O(1).

#Concept - Hence, to get the divisors, we don't need to iterate till the number, we can do it till sqrt(number)

Problem - 10 :

Check for Prime Number. (no divisors except 1 and itself).

Prime Number is a number which has exactly 2 divisors - 1 and itself.

Note that 1 is not a prime number as it has only 1 divisor (1 and itself are same).

Pseudocode -

```
bool isPrime = true ; if (n == 1) return false;
for ( i = 2 → n-1) {
    if ( n % i == 0)   return false;
    }
    return isPrime ;
}
```

false; → breaks from the fn.

```
function (num) {
    if (num == 1)  return false ;
    for (i = 2 → num - 1) {
        if ( num % i == 0)  return false;
    }
    return true ;
}
```

TC → O (N-2) ≈ O(N)

SC → O(1).

Optimized Approach –

But we learnt in the previous problem that for finding divisors, we don't need to run the loop for N times.
Hence, to improve TC of our code,

```
function (num) {
    if (num == 1) return false;
    for (i = 2 ⟶ sqrt(n)) {
        if (num % i == 0) return false;
    }
    return true;
}
```

for (i = 2; i * i ≤ n; i++)

↳ if i is a divisor of num, num/i will also be a divisor.

TC → O(sqrt(N))
SC → O(1).

Problem – 11 :

Count of Prime Numbers till N.
Find out the no. of prime numbers in the range [1, n].

Pseudocode –

```
function (num) {
    count = 0
    for (i = 2 → N) {
        if (isPrime (i))  count ++;
    }
    return count;
}
```

isPrime (num) → func. from prev. problem.

TC → O(N x sqrt(N)).
SC → O(1).

Problem-12 :

GCD of two numbers.
(GCD of 2 numbers is the largest +ve integer which divides both the numbers).

Pseudocode –

```
function(n1, n2) {
    int largest = 1;
    for (int i = 1 → min (n1, n2)) {
        if (n1% i == 0 and n2% i == 0){
            largest = i ;
        }
    }
    return largest ;
}
```

"any no. bigger than N can't divide N."

TC → O(min(n1, n2)), SC → O(1).

Optimized Approach –

Since we are searching for the largest, it is beneficial to go in the backward direction, that is, from min(n1, n2) till 1, and return the first divisor we get (because that will be the largest).

```
function (n1, n2) {
    largest = 1;
    for (i = min(n1, n2) → 1) {
        if (n1 % i == 0 && n2 % i == 0) {
            return i;
        }
    }
}
```

[Note that 1 can also be the gcd for co-primes]

But min(n1, n2) also has some TC, to reduce / optimize it, we need to know about – Euclidean Algorithm.

Euclidean Algorithm states that –
GCD(n1, n2) = GCD(n1 - n2, n2)
given that (n1 > n2).

Example – n1 = 35, n2 = 10 gcd = 5

EA
gcd(25, 10) 35 > 10
gcd(15, 10) 25 > 10
gcd(5, 10) 15 > 10

since difference of n1 and n2 is always

divisible by gcd.

Euclidean Algorithm –

(n1 = 35, n2 = 10)

– 10 ↓ smallest = n2 = 10

(n1 = 25, n2 = 10)

– 10 ↓

(n1 = 15, n2 = 10)

– 10 ↓

(n1 = 5, n2 = 10)

– 5 ↓ smallest = n1 = 5

(n1 = 5, n2 = 5)

– 5 ↓

(n1 = 0, n2 = 5) → GCD.

If one number becomes 0, other is the GCD.

Pseudocode for Euclidean Algorithm –

```
function (n1, n2) {
    while (n1 != 0 && n2 != 0) {
        if (n1 >= n2)  n1 = n1 - n2
        else  n2 = n2 - n1
    }
    return  n1 == 0 ? n2 : n1;
}
```

But this while loop ends up taking a lot of Time Complexity → ex: n1 = 100 and n2 = 5.

In such a case, while loop runs till $n1 = 5$ and $n2 = 5$.
Hence, we need a still optimized soln.
Example — $n1 = 100$, $n2 = 15$.
What we are doing is —

$(100, 15) \xrightarrow{-15} (85, 15) \xrightarrow{-15} (70, 15)$
$\downarrow -15$
$(10, 15) \xleftarrow{-15} (25, 15) \xleftarrow{-15} (40, 15) \xleftarrow{-15} (55, 15)$

Basically, we are subtracting the largest multiple of 15 which is nearest to 100 — and that is 90.

Better Approach → $100 \% 15 = 10$.
$(10, 5) \leftarrow 15 \% 10 = 5 \leftarrow (10, 15)$.
$\hookrightarrow 10 \% 5 = 0 \rightarrow (0, 5) \rightarrow$ GCD = 5.

Pseudocode of Approach —

```
function (n1, n2) {
    while (n1 != 0 && n2 != 0) {
        if (n1 >= n2)  n1 = n1 % n2 ;
        else  n2 = n2 % n1 ;
    }
    return  n2 == 0 ? n1 : n2 .
}
```

TC → O(log (min(n1, n2)))
SC → O(1).

Problem - 13:

LCM of two numbers

Pseudocode -

```
function (n1, n2) {
    int lcm = n1 >= n2 : n1 : n2 ; initialLcm = lcm;
    for (int i = 1 → ∞) {
        if (lcm % n1 == 0 && lcm % n2 == 0){
            return lcm;
        }
        else lcm = lcm + initialLcm ;
    }
}
```

LCM can't be smaller than the bigger of 2 numbers.

```
function (n1, n2) {
    i = 1
    maxNum = max(n1, n2) ;
    do {
        multiple = i x maxNum;
        if (multiple % n1 == 0 && multiple % n2 == 0){
            return multiple ;
        }     break ;
        i = i + 1
    } while (1) .
    return
}
```

TC → O(n1 x n2).

SC → O(1).

Optimized Approach -

$$LCM = \frac{n1 \times n2}{GCD}$$

TC → O(log(min(n1, n2)).
SC → O(1).

```
function GCD (n1, n2) {
    while (n1 != 0 && n2 != 0) {
        if (n1 >= n2)  n1 = n1 % n2;
        else  n2 = n2 % n1;
    }
    return  n1 == 0 ? n2 : n1
}

function LCM (n1, n2) {
        return  n1 * n2 / GCD(n1, n2);
}
```

Basic Arrays

What are Arrays?

Data-structure that can store multiple elements of the same data-type. — int, char, double, pair.

Syntax:

dataType arrayName[size] = {_, _, _};

Note - Cannot alter the size of the array when it's first assigned.

How variables are stored?

Inside memory, there are memory blocks. inside which variables are stored.

int num = 6; ⟶ num = 6.

How Arrays are stored?

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

Numbers are stored in contiguous memory blocks.

To get the size of the array,

int size = sizeof(arr) / sizeof(arr[0]).

To access the elements of the array -

→ Arrays go by 0-based indexing.

If size = 4, index = [0, 3].
If size = N, index = [0, N-1].
0 is the first element and (N-1) is the last element.

Arrays can be stored inside another array, and we call that array a 2-D array, that is, we can store arrays inside memory blocks.

1-D Array - int arr[] = {5, 6, 10, 4}.

Traversing of an 1-D array -

```
for (int i = 0 → n-1) {
        print(arr[i]).
}
```

→ print all array elts.

Problem - 1

Sum of Array elements.

Sum of all the elements in the array.

Pseudocode –

```
function (arr, n) {
        int sum = 0;
        for (i = 0 ⟶ n-1) {
            sum = sum + arr [i];
        }
        return sum;
}
```

Problem-2 :

Count of odd numbers in the array.

Pseudocode –

```
function (arr, n) {
        int count = 0
        for (int i = 0 ⟶ n-1) {
            if (arr [i] % 2 == 1) {
                count ++
            }
        }
        return count;
}
```

TC → O(N) SC → O(1)

Problem - 3 :

Reverse an Array . (don't return anything)

Pseudocode -

- Since arrays are passed by reference, we need to make the changes within the array itself.

Let arr1[] = {1, 2, 3, 4, 5}. $n = 5$.
(0, 1, 2, 3, 4)

arr2[] = {1, 2, 3, 4}.
(0, 1, 2, 3)

reversedArr[] = {5, 4, 3, 2, 1}.

If n is odd, run the loop till $\frac{n-1}{2}$.

If n is even, run the loop till $\frac{n}{2}$.

won't work

Replace arr[i] with arr[n - i - 1].

→ turns [1, 2, 3, 4, 5] into [5, 4, 3, 4, 5].

→ **First Approach -**

arr[] = {1, 2, 3, 4, 5}
(0, 1, 2, 3, 4)

temp[n] = {_, _, _, _, _}

n = 5

(i)	(n - i - 1)
0 →	4
1 →	3
2 →	2
3 →	1
4 →	0

// Reverse traversal of the arr.

```
for (i = 0 → n-1) {
        temp [n-1-i] = arr [i];
}
```

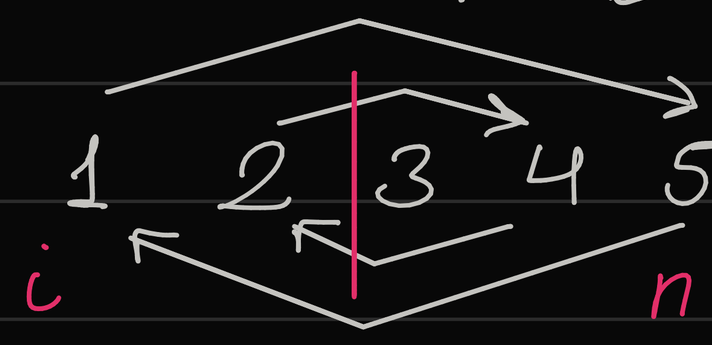
```
// Convert arr into temp.
for(i=0 ⟶ n-1) {
    arr[i] = temp[i];
}
```

Clean Pseudocode –

```
void reverseArray (arr, n) {
    temp[n]
    for(int i=0 ⟶ n-1) {
        temp[n-i-1] = arr[i]
    }
    for(int i=0 ⟶ n-1) {
        arr[i] = temp[i]
    }
}
```

TC → O(N) x 2 = O(2N).
SC → O(N).

Not the preferred way of doing it.



Optimized Approach – 2 Pointer Approach
Three variable concept.

{1, 2, 3, 4, 5}.

↓ ↓

left = 0 right = 4

temp.

Three variable Approach —

internal Swap function ←

```
temp = arr[left]   // temporarily store
arr[left] = arr[right]   the variable
arr[right] = temp.   to be replaced
                     by some other but later
                     to be used.
```

```
left = left + 1 ;
right = right - 1 ;
stop once left = right. (collide).
```

Pseudocode —

```
void reverseArray (arr, n) {
    left = 0 ;   right = n - 1 ;
    while (left < right) {
        swap (arr[left], arr[right]) ;
        left ++ ;
        right -- ;
    }
}
```

swap(arr[left], arr[right]) ;

```
int temp = arr[left]
arr[left] = arr[right]
arr[right] = temp.
```

Problem - 4 :

Check if the array is sorted (in ascending/ non - decreasing) order.

Pseudocode -

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

```
// sorted or not.
    i = 0 to n-1
    j = i+1 to n-1
        if (arr[j] < arr[i]) {
            return false;
        }
    }
    return true;
```

Naive Approach

```
bool function (arr, n) {
    for (int i=0 ⟶ n-1) {
        for (int j=i+1 ⟶ n-1) {
            if (arr[j] < arr[i])
                return false;
        }
    }
    return true;
}
```

TC ⟶ O(N²),
SC ⟶ O(1)

Optimized Approach –

$i-1$ i

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

$1 \leq 2 \leq 3 \leq 4 \leq 5$

check if $2 \geq 1$, then check if $3 \geq 2$ (no need to check if $3 \geq 1$), then check if $4 \geq 3$, and so on.

Pseudocode –

```
bool function(arr, n) {
    for (i = 1 ⟶ n-1) {
        if (arr[i-1] > arr[i]
            return false.
    }
    return true
}
```

TC ⟶ O(N)
SC ⟶ O(1).