Day 4 / 100 :

Topic - Array

1 Problem statement: **Factorial Trailing Zeroes** (medium)

Given an integer n, return the number of trailing zeroes in n!.

Note that n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1.

Example 1:
Input: n = 3
Output: 0
Explanation: 3! = 6, no trailing zero.

Example 2:
Input: n = 5
Output: 1
Explanation: 5! = 120, one trailing zero.

Solutions :

Approach 1 - Brute Force

- The brute force approach to finding the trailing zeros in a factorial is to calculate the factorial first and then count the number of trailing zeros.
- this code, the factorial function is recursively called to calculate the factorial of n. Then, the trailingZeroes function counts the number of trailing zeros in the calculated factorial using a similar approach as before.
- Calculating the factorial itself has a time complexity of O(n), and then checking the trailing zeros adds additional operations.

```
class Solution {
public:
    int trailingZeroes(int n) {
        long long factRes = factorial(n);
```

```
        int count = 0;
        while (factRes > 0 && factRes % 10 == 0) {
            count++;
            factRes /= 10;
        }
        return count;
    }

private:
    long long factorial(int n) {
        if (n < 0)
            return -1; // Invalid input
        if (n == 0)
            return 1; // Base case
        else
            return n * factorial(n - 1); // Recursive call
    }
};
```

Solutions :

Approach 2 - optimized Approach

Intution:

The reason we use 5 to count the trailing zeros in the factorial is because a trailing zero occurs whenever there is a factor of 10 in the factorial. And a factor of 10 can be formed by multiplying 2 and 5.

In a factorial, the number of factors of 2 is always more than the number of factors of 5. So, counting the factors of 5 will give us the number of trailing zeros.

```
class Solution {
public:
    int trailingZeroes(int n) {
        int count = 0;
        while (n >= 5) {
            n /= 5;
            count += n;
```

```
        }
        return count;
    }
};
```

Explanation:

1. Initialize a variable count to 0.
2. Start a loop while n is greater than or equal to 5:
    - Divide n by 5 and update the value of n with the quotient.
    - Add the quotient to the count variable.
    - Repeat the above steps until n is no longer greater than or equal to 5.
3. Return the value of count, which represents the number of trailing zeros in the factorial.

The optimized approach takes advantage of the fact that the number of trailing zeros in a factorial is determined by the number of factors of 5. By repeatedly dividing n by 5 and counting the quotient, we can efficiently calculate the number of trailing zeros without explicitly calculating the factorial.

2 Problem statement: Maximum Subarray Sum (medium)

Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
Example 2:

Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.

Solutions :

Approach 1 - Brute Force Approch

Intuition:

1. Iterate through each element as a potential starting point of a subarray.
2. For each starting point, iterate through the array again to consider all possible subarrays starting from that point.
3. Calculate the sum of each subarray and update the maximum sum if a larger sum is found.

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        int maxSum = INT_MIN;

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

            for (int j = i; j < n; j++) {
                currentSum += nums[j];
                maxSum = max(maxSum, currentSum);
            }
        }

        return maxSum;
    }
};
```

Explanation:

The code iterates through the array nums twice, resulting in a nested loop. The outer loop iterates through each element as a potential starting point of a subarray, and the inner loop iterates through the remaining elements to consider all possible subarrays starting from that point. The variable currentSum keeps track of the sum of the current subarray, and maxSum stores the maximum sum encountered so far.

Solutions :

Approach 2 - Kadane's algorithm

Kadane's algorithm is an efficient approach to find the maximum sum subarray in an array. It utilizes the concept of dynamic programming to keep track of the maximum sum subarray ending at each position. By updating this maximum sum dynamically, we can find the overall maximum sum subarray.

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        int currentSum = nums[0];
        int maxSum = nums[0];

        for (int i = 1; i < n; i++) {
            currentSum = max(nums[i], currentSum + nums[i]);
            maxSum = max(maxSum, currentSum);
        }

        return maxSum;
    }
};
```

Intuition:

1. Initialize two variables, currentSum and maxSum, to track the maximum sum subarray ending at the current position and the overall maximum sum subarray, respectively.

2. Iterate through the array from left to right.
3. For each element, update the currentSum by adding the current element or starting a new subarray if the current element is larger than the sum so far.
4. Update the maxSum if the currentSum becomes larger than the maxSum.
5. Finally, return the maxSum.