# Experiment-4

**Name:** Aayushi Sinha **UID:** 22BET10268

**Branch:** BE-IT **Section/Group:** 22BET\_IOT-702/B

**Semester:** 6th **Date of Performance:**14/02/25

**Subject Name:** AP LAB-II **Subject Code:** 22ITP-351

**Problem-1**

**1.Aim:**

Given a string s, return the longest **substring** of s that is **nice**. If there are multiple, return the substring of the **earliest** occurrence. If there are none, return an empty string.

**2.Objective:**

To implement a solution that finds the longest "nice" substring of s using an efficient approach, ensuring that the substring's length is maximized, and if there are multiple longest substrings, the earliest occurrence is returned.

## 3.Code:

class Solution {

public:

string longestNiceSubstring(string s) {

if (s.length() < 2) return "";

unordered\_set<char> charSet(s.begin(), s.end());

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

if (charSet.count(tolower(s[i])) && charSet.count(toupper(s[i]))) continue;

string left = longestNiceSubstring(s.substr(0, i));

string right = longestNiceSubstring(s.substr(i + 1));

return left.length() >= right.length() ? left : right;

}

return s;

}

};

## 4.Output:

## 

**Fig.1.Longest Nice Substring**

**Problem-2**

**1.Aim:**

Reverse bits of a given 32 bits unsigned integer.

**2.Objective:**

To implement a function that takes a 32-bit unsigned integer as input and returns the integer obtained by reversing its bits.

## 3.Code:

## class Solution {

## public:

## uint32\_t reverseBits(uint32\_t n) {

## uint32\_t result = 0;

## for (int i = 0; i < 32; i++) {

## result = (result << 1) | (n & 1); // Shift result left, add last bit of n

## n >>= 1; // Shift n right to process the next bit

## }

## return result;

## }

## };

## 4.Output:

## 

**Fig.2:Reverse Bits**

# Problem-3

# 1.Aim:

Given a positive integer n, write a function that returns the number of

set bits in its binary representation (also known as the [**Hamming weight**](http://en.wikipedia.org/wiki/Hamming_weight)).

**2.Objective:**

The objective of this program is to **count the number of set bits (1s)** in the binary representation of a given positive integer n, also known as the **Hamming weight**. A set bit is a bit that is 1 in the binary representation of the integer.

## 3.Code:

## class Solution {

## public:

## int hammingWeight(uint32\_t n) {

## int count = 0;

## while (n) {

## count += n & 1; // Add 1 if the last bit is set

## n >>= 1; // Shift right to process the next bit

## }

## return count;

## }

## };

## 4.Output:

## 

**Fig.3:Number of 1 bits**

# Problem-4

**1.Aim:**

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

**2.Objective:**

The objective of this program is to find the subarray with the largest sum in a given integer array nums and return its sum.

## 3.Code:

## class Solution {

## public:

## int maxSubArray(vector<int>& nums) {

## int maxSum = nums[0], currentSum = nums[0];

## for (int i = 1; i < nums.size(); i++) {

## currentSum = max(nums[i], currentSum + nums[i]);

## maxSum = max(maxSum, currentSum);

## }

## return maxSum;

## }

## };

## 4.Output:

## 

**Fig.4:Maximum Subarray**

# Problem-5

**1.Aim:**

To calculate ab mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

**2.Objective:**

a is a positive integer,b is a very large positive integer, represented as an array of digits (each element is a single digit of the exponent).The solution must efficiently handle the large exponent using modular exponentiation and mod properties to prevent overflow.

## 3.Code:

## class Solution {

## public:

## int modExp(int a, int b, int mod) {

## int result = 1;

## a = a % mod;

## while (b > 0) {

## if (b % 2 == 1) {

## result = (result \* a) % mod;

## }

## a = (a \* a) % mod;

## b /= 2;

## }

## return result;

## } int superPow(int a, vector<int>& b) {

## int mod = 1337;

## int result = 1;

## int base = a % mod;

## for (int i = 0; i < b.size(); i++) {

## result = modExp(result, 10, mod); // Raise previous result to the power of 10

## result = (result \* modExp(base, b[i], mod)) % mod; // Multiply with current digit of b

## }

## return result;

## }

## };

## 4.Output:

## 

**Fig.5:Super Pow**