**Solidity Assignment**

**1)Hello World using solidity**

/ SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract fistcon{

    string public vari='hello world';

}

**2)Program to find Factorial of number via pure functions**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract fact{

    //uint f=1;

    function factorial(uint n) public pure returns(uint) {

        uint f=1;

        for(uint i=1;i<=n;i++)

        {

            f=f\*i;

        }

        return f;

    }

}

**3)Implementing decentralised voting system for 3 candidates, each voter can vote twice**

pragma solidity ^0.8.7;

contract Election {

struct Candidate{

    uint id;

    string candidateName;

    uint voteCount;

}

mapping (address=>uint) public voters;

//0 , 1 =

mapping (uint=> Candidate)public candidates;

uint public candidatesCount;

function addCandidate(string memory \_name) private {

    candidatesCount++;

    candidates[candidatesCount]=Candidate(candidatesCount,\_name,0);

}

constructor()  {

    addCandidate("Donald Trump");

    addCandidate("Joe Baiden");

    addCandidate("Narendra modi");

}

event consolePrint( string, address);

function vote(uint \_candidateId) public{

    require(voters[msg.sender] < 2); //msg.sender => person who has initiated smart contract

    require(\_candidateId>0 && \_candidateId<=candidatesCount); //correct set of candidates

    emit consolePrint("value of sender is ",msg.sender);

    voters[msg.sender] +=1;

    candidates[\_candidateId].voteCount++;

}

}

**4)Palindrome Program : to write a code to return palindrome of a string, if it is palindrome transfer 50 ETH from one account to manager account.**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract Palindrome{

    string name;

    function Reverse(string calldata \_name) external pure returns (string memory,bool){

        bytes memory \_baseBytes = bytes(\_name);

        //check if the string length is not zero, if it's zero then execution terminates

        assert(\_baseBytes.length > 0);

        string memory \_tempValue = new string(\_baseBytes.length);

        bytes memory \_newValue = bytes(\_tempValue);

        for(uint i = 0; i < \_baseBytes.length; i++){

            \_newValue[\_baseBytes.length - i -1] = \_baseBytes[i];

        }

        if( keccak256(abi.encodePacked(\_name)) == keccak256(abi.encodePacked(\_newValue)) ){

            return (string(\_newValue),true);

        }

        else{

            return (string(\_newValue),false);

        }

    }

}

**5)Write a contract 'Time' which implements a function named getTime.**

getTime() accepts a Unix timestamp (number of seconds since 1970-01-01 00:00:00 GMT). Method will check if this timestamp exists after the current time. If yes, it will return a timestamp after 1 hour, 20 minutes and 30 seconds from passed time else return 0.

Informational Hint

Time Units

Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

1 == 1 seconds

1 minutes == 60 seconds

1 hours == 60 minutes

1 days == 24 hours

1 weeks == 7 days

To get current time, now can be used being aware about the fact that it can be influenced by miners to some degree. It retuns uint

**6)Problem Statement**

Write a contract 'ThreeAndSeven' which implements a function named check.

check() accepts a number and return true if number is fully divided by 3 or 7 and greater than 10 else false. This function should not consume any gas.

Informational Hint

Boolean Value Type

bool: The possible values are constants true and false.

Operators:

! (logical negation)

&& (logical conjunction, “and”)

|| (logical disjunction, “or”)

== (equality)

!= (inequality)

The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract ThreeAndSeven{

    function check(uint n) public pure returns(bool){

        if(n>=10)

        {

            bool num = (n%3==0 || n%7==0) ? bool(true) : bool(false);

            return num;

        }else{

            bool num=false;

            return num;

        }

    }

}

**7)EvenOdd**

Problem Statement

Write a contract 'EvenOdd' to which implements a function named check.

check() accepts a number and return whether the passed number is odd or even without consuming gas. (Ignore various checks on passed parameters for now)

Informational Hint

Any integer that can be divided exactly (i.e. remainder = 0) by 2 is an even number.

Modulo: The modulo operation a % n yields the remainder r after the division of the operand a by the operand n

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract EvenOdd{

    function check(uint n) public pure returns (string memory){

        if(n%2==0)

        {

            return 'even';

        }else{

            return 'odd';

        }

    }

}

**8)Problem Statement**

Write a contract 'Calculator' to which returns addition, subtraction, multiplication and division of two passed integers without consuming gas. (Ignore various checks on passed parameters for now)

Informational Hint

To define a function which should not consume any gas and return the response immediatly, we should use view or pure function.

Operators

Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

Arithmetic operators: +, -, unary -, unary +, \*, /, % (remainder), \*\* (exponentiation)

Division by zero and modulus with zero throws an exception.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract calulator{

    function add(uint a, uint b) public pure returns(uint){

        uint x;

        x=a+b;

        return x;

    }

    function sub(uint a,uint b) public pure  returns(uint){

        uint y;

        y=a-b;

        return y;

    }

    function square(uint a,uint b) public pure returns (uint){

        return a\*\*b;

    }

    function returnAll(uint a, uint b) public pure returns (uint,uint,uint,uint){

        return (a\*\*b,a+b,a\*b,a/b);

    }

}

**9)Program to find Sum of Digits**

// SPDX-License-Identifier: MIT

pragma solidity 0.8.7;

contract digits{

    //uint[] a;

    //uint n;

    function sumDigit(uint num,uint n) public pure returns (uint) {

        //uint[] memory a = new uint[](n);

        //uint[n] memory a;

        uint sum=0;

        uint rem;

        for(uint i=0;i<=num;i++)

        {

            rem = n%10;

            sum=sum+rem;

            n=n/10;

        }

        return sum;

    }

}

**10)Problem Statement**

Write a contract named "AttendanceRegister" which will be deployed by teacher. There will be a function add which will take student name, class & joiningDate and will store it where:

name: String to store student name

class: Number from 1 to 12

joiningDate: Date timestamp of student joining day Each student will be stored corresponding to a unique serial number called roll number.

Only teacher can add the data of student. An event will be emitted on success.

Use struct, modifier & mapping wherever applicable.

Informational Hint

Structs

Solidity provides a way to define new types in the form of structs. Struct types can be used inside mappings and arrays and they can itself contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Modifiers

Modifiers can be used to easily change the behaviour of functions. For example, they can automatically check a condition prior to executing the function. Modifiers are inheritable properties of contracts and may be overridden by derived contracts.

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the “\_” in the preceding modifier.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

pragma solidity ^0.8.10;

struct student{

        string Name;

        uint Class;

        string JoiningDate;

    }

contract AttendanceRegister{

    address public Teacher = msg.sender;

    mapping (uint => student) public data;

    event Register(address Teacher, student Data);

    modifier onlyTeacher() {

        require(Teacher == msg.sender, "You are not a teacher");

        \_;

    }

    function add(uint check, string memory name, uint class, string memory date) public onlyTeacher {

        data[check] = student(name,class,date);

        emit Register(msg.sender, data[check]);

    }

}