School: .... Campus: ....

Academic Year: .... Subject Name: .... Subject Code: ....

Semester: .... Program: .... Branch: .... Specialization: ....

Date: ....

Applied and Action Learning

(Learning by Doing and Discovery)

Name of the Experiement : Gas Race – Optimizing Smart Contract Efficiency

* Coding Phase: Pseudo Code / Flow Chart / Algorithm

Algorithm

1.Start the Solidity smart contract in Remix IDE.
2.Write the initial version of the contract with basic logic (e.g., storing and updating data).
3.Compile and deploy the contract to observe the initial gas consumption.
4.Analyze gas usage using Remix's "Gas Analysis" tool after each function execution.
5.Apply optimization techniques, such as: ○ Using memory instead of storage when possible. ○ Declaring variables with the smallest suitable data type. ○ Combining operations and reducing function calls.
6.Recompile and redeploy the optimized contract.
7.Compare gas usage before and after optimization.
8.Stop after verifying reduced gas consumption and correct functionality.

* Softwares used

1.Remix Ide
2.MetaMask
3.Web3.js/Ether.js
4.Sepolia testnet

*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.

* Implementation Phase: Final Output (no error)

Gas optimization in Solidity focuses on writing efficient smart contracts that reduce transaction costs and improve execution speed.

- **Efficient Storage Use:** Use `uint256` over smaller types unless packing; combine smaller variables in one slot to save gas.

```
// Inefficient:
struct Bad {
    uint128 a;
    uint256 b;
    bool c;
}

// Better:
struct Good {
    uint128 a;
    bool c;
    uint256 b; // 128 + 8 = 136 bits, fits nicely
}
```

- **Minimize SSTORE Calls:** Reduce multiple storage writes by computing values first, then storing once.

```
// Inefficient:
counter++;
mapping[user] = counter;

// Better:
uint256 newValue = counter + 1;
mapping[user] = newValue;
counter = newValue;
```

- **Avoid Unbounded Loops:** Keep loops short or replace them with pull-based mechanisms to prevent out-of-gas errors.

```
// Bad: gas cost grows with user count
function distributeRewards() public {
    for (uint i = 0; i < users.length; i++) {
        rewards[users[i]] += 100;
    }
}
```

- **Use calldata:** Prefer `calldata` over `memory` for external function inputs to avoid unnecessary copying.

```
// Inefficient
function storeData(string memory _data) external { ... }

// Efficient
function storeData(string calldata _data) external { ... }
```

- **Constants & Immutables:** Use `constant` and `immutable` to save on storage reads and make execution cheaper.

```
uint256 public constant FEE = 100;
address public immutable owner;
```

- **Bitwise & Assembly:** For advanced optimization, bitwise operations can pack data efficiently (use carefully).

```
// Bit-pack a bool into a uint256
uint256 flags;
flags |= (1 << 0); // Set bit 0
```


*As applicable according to the experiment.
Two sheets per experiment (10-20) to be used.*

* Implementation Phase: Final Output (no error)

Tip	Benefit
Use `calldata` over `memory`	Lower call data cost
Pack variables	Reduce storage slots
Use `constant` and `immutable`	Save storage reads
Avoid unbounded loops	Prevent OOG errors
Cache external calls	Reduce redundancy
Batch operations	Fewer transactions

* Observations

1.Code structure optimization leads to reduced gas costs
2.Better storage usage and loop optimization improves transaction performance
3.Benchmarking tools are effective for identifying expensive operations and measuring improvements

ASSESSMENT

Rubrics	Full Mark	Marks Obtained	Remarks
Concept	10		
Planning and Execution/ Practical Simulation/ Programming	10		
Result and Interpretation	10		
Record of Applied and Action Learning	10		
Viva	10		
Total	**50**		

Signature of the Student:

Name :

Signature of the Faculty:

Regn. No. :

****As applicable according to the experiment.***
Two sheets per experiment (10-20) to be used.