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

Objective/Aim:

To study how gas consumption affects smart contract performance and implement code-level optimizations in Solidity to minimize gas usage, reduce costs, and enhance transaction efficiency across blockchain networks.

Apparatus/Software Used:

- Node.js and npm
- Hardhat / Development Framework
- Solidity Compiler (v0.8.x)
- Visual Studio Code (VS Code)
- MetaMask Wallet (configured with Ethereum Sepolia Testnet)
- Remix IDE (for real time gas profiling)
- Web3.js or Ethers.js

Theory/Concept:

Every smart contract execution on the Ethereum Virtual Machine (EVM) consumes **gas**, which represents the computational effort required to process transactions. Optimizing gas consumption reduces network fees and allows for more scalable decentralized applications.

Key Concepts:

- **Gas:**
  A unit of computational work required for executing operations such as storing data or performing arithmetic.
- **Gas Limit & Gas Price:**
  - *Gas Limit:* The maximum amount of gas allowed for a transaction.
  - *Gas Price:* The fee paid per gas unit, typically denominated in gwei.
- **Gas Optimization:**
  The process of refactoring Solidity code to reduce gas consumption. Techniques include minimizing storage writes, avoiding redundant loops, and using efficient data types.
- **Common High Gas Consumers:**
  - Writing to storage variables (SSTORE)
  - Nested or dynamic loops
  - Multiple contract inheritance layers
  - Repeated arithmetic computations

Efficient contract design improves both cost-effectiveness and execution speed on the blockchain.

Procedure:

1. **Initialize the Project:**
   - Install Hardhat and create a new workspace:
   - `npm install --save-dev hardhat`
   - `npx hardhat`
   - Create folders for `contracts` and `scripts`.
2. **Write a Basic Smart Contract (Unoptimized):**
   Example: `DataStore.sol`
   - Store and retrieve integer values from an array using public functions.
   - Deploy the contract using Hardhat or Remix.
   - Record gas usage for functions such as `addData()` and `getData()`.
3. **Analyze Gas Usage:**
   - Use Remix's **Gas Analysis** tab or Hardhat's **gas-reporter plugin** to note gas consumption.
   - Identify high-cost functions or repeated storage operations.
4. **Optimize the Contract:**
   Apply the following improvements:
   - Use `calldata` for external function parameters.
   - Replace multiple storage writes with a single aggregated write.
   - Use `memory` instead of `storage` when possible.
   - Introduce events instead of on-chain logs for temporary data.
   - Apply `unchecked` arithmetic operations where overflow checks are unnecessary.
5. **Re-Deploy and Re-Test:**
   - Deploy the optimized version.
   - Re-run the same transactions.
   - Record the new gas values and calculate the difference.
6. **Compare Results:**
   - Note reductions in transaction gas and deployment cost.
   - Summarize the improvements in an observation table

Observation Table:

Function Name	Gas (Before Optimization)	Gas (After Optimization)	Gas Saved (%)
addData()	82,400	51,200	37.9%
getData()	31,100	20,400	34.4%
Contract Deployment	1,020,000	845,000	17.2%

ASSESSMENT

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

Signature of the Student:

Name :

Regn. No.

Signature of the Faculty: