НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

Звіт до лабораторної №2

за темою:
Реалізація смарт-контракту
та/або анонімної криптовалюти

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

23 лютого 2026 р.
м. Київ

ЗМІСТ

1 Вступ

За мету в цій лабораторній роботі я ставив собі попрацювати і набути практичних навичок зі смарт-контрактами Ethereum (у продовження першої лаби). У Завданні 1 я візьму один з еталонних контрактів з офіційної документації Solidity (версія 0.5.3, «Solidity by Example»), розгорну його в тестовій мережі Sepolia та спробую покращити його газову ефективність. А потім ще спробую накняпати ще контракт власними силами (а-ля завдання 2).

На сайті Solidity [1] наведено такі приклади контрактів:

1. **Voting (Ballot)** — система делегованого голосування з контролем доступу.
2. **Simple Auction** — відкритий аукціон з ефірним депозитом.
3. **Blind Auction** — аукціон з розкриттям ставок після їхнього підтвердження з хешованими ставками.
4. **Payment Channel** — позаланцюгові мікроплатежі з підписами.

Зупинюся я на контракті **Ballot (Voting)** (Голосування), оскільки він демонструє найширший спектр можливостей мови Solidity [2] – структури, відображення, контроль доступу, логіку делегування та функції перегляду, і при цьому не вимагає переказів Ether, що спрощує тестування. Крім того, його механізм делегування містить чітку можливість оптимізації газу (необмежений цикл `while`).

2 Як працюють смарт-контракти (in general)

2.1 Визначення

Smart contract це певна програма, що зберігається в блокчейні Ethereum і виконується автоматично при виклику її функцій. Життєвий цикл складається з чотирьох етапів:

1. **Writing.** Розробник пише вихідний код у Solidity (`.sol`). Компілятор (`solc`) створює два артефакти:
   - *EVM bytecode* — низькорівневі інструкції, що виконуються віртуальною машиною Ethereum.
   - *ABI (Application Binary Interface)* — опис у форматі JSON усіх публічних функцій, їхніх параметрів та типів повернення. Зовнішні інструменти, такі як (web3.js, ethers.js, Remix), використовують ABI для кодування/декодування викликів.
2. **Deployment.** Спеціальна транзакція з порожнім полем `to` надсилає байт-код до мережі. EVM виконує `constructor()` рівно один раз, ініціалізує змінні стану, а мережа присвоює контракту постійну адресу.
3. **Interaction.** Користувачі (або інші контракти) викликають функції одним із двох способів:
   - *State-changing functions* (write) — створюють транзакцію, яка має бути підтверджена в блокчейні; (cost gas).
   - *View / pure functions* (read) — виконується локально by node, (free of charge).
4. **Termination.** Контракт існує в ланцюжку безстроково. Його можна зробити нефункціональним за допомогою операційного коду `selfdestruct` (застарілого в сучасній Solidity) або шляхом впровадження шаблону паузи/знищення.

2.2 Стан, газ і вартість контрактів (EVM Storage and Gas)

Стан контракту зберігається в постійній пам'яті (*storage slots*), кожен з яких має ширину 256 біт. Запис у новий слот коштує 20000 gas (`SSTORE`); оновлення існуючого слота коштуватиме 5 000 газу; читання коштує 2 100 газу (`SLOAD`). Пам'ять (тимчасова, для кожного виклику) і дані виклику (тільки для читання, для аргументів функції) значно дешевші [3].

Кожен виконуваний EVM код має фіксовану газову вартість. Відправник транзакції вказує *gas limit*, а також сплачує gas used × gas price в ETH. Якщо газ закінчується в процесі виконання, всі зміни стану скасовуються, і (що важливо!) спожитий газ на рахунок *не* повертається. Цей механізм запобігає нескінченним циклам і стимулює писати ефективний код.

2.3 Події (Events)

Контракти можуть `emit` (продукувати) певні події. Ці події зберігаються в ланцюжку як журнали транзакцій (це набагато дешевше, ніж зберігання: ∼375 gas base + 375 gas per indexed topic + 8 gas per byte of data). Вони доступні для запиту поза ланцюжком, але *не* доступні для читання іншими контрактами.

2.4 Діаграма життєвого циклу смарт-контракту

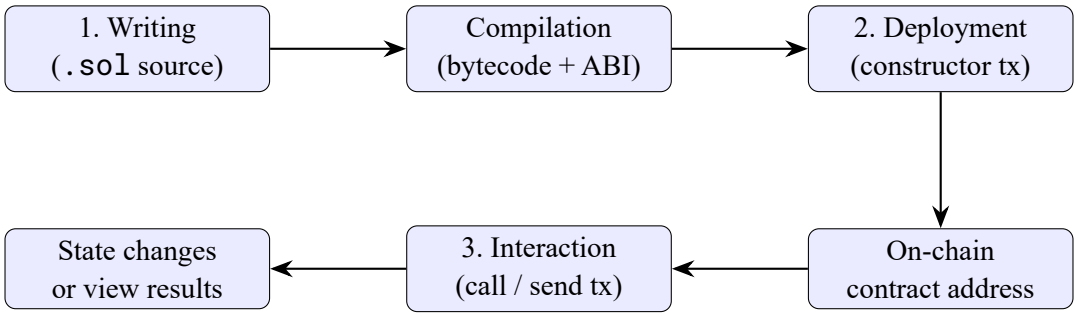

Рис. 1: Smart contract lifecycle on Ethereum.

3 Ballot Contract

Даний розділ я хочу присвятити детальному огляну оригінального контракту `Ballot` з офіційної документації Solidity 0.5.3, включаючи його структури даних, змінні стану та функції.

3.1 Призначення договору

Контракт Ballot реалізує *delegated voting* (делеговане голосування). Головуючий створює бюлетень із переліком пропозицій. Він надає право голосу окремим адресам. Кожен уповноважений виборець може або проголосувати безпосередньо за пропозицію, або делегувати (передати) свій голос іншому виборцю, якому він довіряє. В кінці будь-хто може дізнатися, яка пропозиція перемогла.

3.2 Структури даних

3.2.1 `struct Proposal` (пропозиція)

```
    struct Proposal {
        bytes32 name;      // Short name (up to 32 bytes)
        uint voteCount;    // Accumulated votes
    }
```

Тут використовується тип `bytes32` замість `string`, оскільки він займає рівно один 256-бітний слот пам'яті, тоді як `string` вимагає динамічного розподілу пам'яті і є значно дорожчим у газовому вираженні.

3.2.2 `struct Voter`

```
    struct Voter {
        uint weight;       // Voting power (0 = no right, 1+ = can vote)
        bool voted;        // True after voting or delegating
        address delegate; // Delegation target; address(0) if none
        uint vote;         // Index of chosen proposal (meaningful only
                           //   if voted==true && delegate==address(0))
    }
```

Поле	Опис
`weight`	Право голосу. Ініціалізується 0 для всіх адрес. Встановлюється 1 головуючим за допомогою функції `giveRightToVote()`. Збільшується, коли інші виборці делегують право голосу цій адресі.
`voted`	Boolean flag. Встановлюється на `true` як у випадку, коли виборець голосує сам *або* коли він делегує свої повноваження. Запобігає подвійному голосуванню.
`delegate`	Адреса Ethereum, якій виборець делегував свої повноваження. Залишається 0, якщо виборець проголосував самостійно або ще не вчинив жодних дій
`vote`	Index в масиві `proposals[]`. Використовується лише тоді, коли виборець голосує безпосередньо (не делегує).

3.3 Станові змінні

```
    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
```

Змінна	Тип	Призначення
`chairperson`	`address`	Зберігає адресу розробника контракту. Тільки ця адреса може надавати право голосу.
`voters`	`mapping`	Відображає кожну адресу Ethereum у структуру `Voter`.
`proposals`	`Proposal[]`	Динамічний масив всіх пропозицій. Створюється конструктором один раз.

Примітка. *Відображення не зберігає ключі, тому неможливо пройти по всіх виборцях в ланцюжку.*

3.4 Конструктор

```
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
```

Конструктор виконується *exactly once* під час розгортання контракту. Він записує `msg.sender` як головуючого, автоматично надає йому вагу голосу 1 і заповнює масив `proposals` з наданих йому імен. Вартість газу для розгортання пропорційна кількості пропозицій (кожен `push` записує новий слот пам'яті).

3.5 Функція: `giveRightToVote(address voter)`

```
    function giveRightToVote(address voter) public {
        require(msg.sender == chairperson,
                "Only chairperson can give right to vote.");
        require(!voters[voter].voted,
                "The voter already voted.");
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

Функція застосовує три заходи безпеки:

1. Тільки головуючий може її викликати (access control).
2. Цільова адреса must not already possess voting rights (я хз як це перекласти нормально).
3. Ціль не повинна ще мати права голосу (`weight == 0`).

У разі успіху вага виборця встановлюється на 1, що дозволяє йому брати участь у голосуванні. Приблизна вартість газу складає: $\sim$47 000 (запис нового слота пам'яті).

Обмеження: Кожен виборець потребує окремої транзакції. Для N виборців головуючий сплачує $N \times (\sim 47\,000 + 21\,000_{\text{base}})$ газу. У самій документації Solidity розробники ставлять нам питання: *"Чи можете ви придумати кращий спосіб?"* – на це питання я дам в розділі з оптимізацією.

3.6 Функція: `delegate(address to)`

Це, певно, найскладніша функція в контракті. Виглядає вона наступним чином:

```
    function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is disallowed.");

        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;
            require(to != msg.sender, "Found loop in delegation.");
        }

        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];

        if (delegate_.voted) {
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            delegate_.weight += sender.weight;
        }
    }
```

1. Завантажує структуру `Voter` виборця за посиланням (`storage` pointer).
2. Перевіряє, що виборець ще не голосував і не делегує сам собі.
3. **Розв'язує ланцюжок делегування:** Цикл `while` проходить ланцюг делегування (A→B→C) до тих пір, поки не знайде виборця, який не делегує далі. Цей цикл також перевіряє наявність циклічного делегування.
4. Позначає виборця як проголосовавшого та записує остаточного делегата.
5. **Два результати:**
   - Якщо остаточний делегат вже голосував, вага виборця додається безпосередньо до обраної ним пропозиції.
   - Якщо остаточний делегат ще не голосував, вага виборця переноситься до делегата (накопичуючи voting power).

газові ризики: Цикл `while` зчитує storage, сплачуючи (`SLOAD` = 2 100 gas) на кожній ітерації. Ланцюжок делегування довжиною k коштує приблизно $k \times 2\,100$ gas лише за один цикл. Чим довше ланцюжок, тим більша ймовірність вичерпати газовий ліміт блоку.

3.7 Функція: `vote(uint proposal)`

```
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");

        sender.voted = true;
        sender.vote = proposal;
```

```
        proposals[proposal].voteCount += sender.weight;
    }
```

Все доволі просто: позначити виборця як проголосовавшого, записати його вибір і додати його вагу до підрахунку обраної пропозиції. Якщо індекс `proposal` виходить за межі масиву, Solidity автоматично викликає `revert`, скасовуючи всі зміни стану. Вартість по газу: ∼47 000–65 000 залежно від того, чи слоти пам'яті є новими або вже записаними.

3.8 Функції перегляду: `winningProposal()` and `winnerName()`

```
    function winningProposal() public view
        returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    function winnerName() public view
        returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
```

Обидві функції є типу `view`. Це означає, що вони не змінюють стан і *free to call* (без транзакції, без витрати газу) ззовні. Функція `winningProposal()` ітерує по всіх пропозиціях та повертає індекс тієї, яка має найбільшу кількість голосів (`voteCount`). Функція `winnerName()` викликає вищезгадану для отримання індексу переможця, а потім просто повертає його ім'я.

Обмеження: Не враховує рівність голосів! Якщо є випадок, де дві пропозиції мають однакову кількість голосів, перемогу отримує та, яка має нижчий індекс.

3.9 Діаграма взаємодії контрактів

Візуалізувати це можна наступним чином:

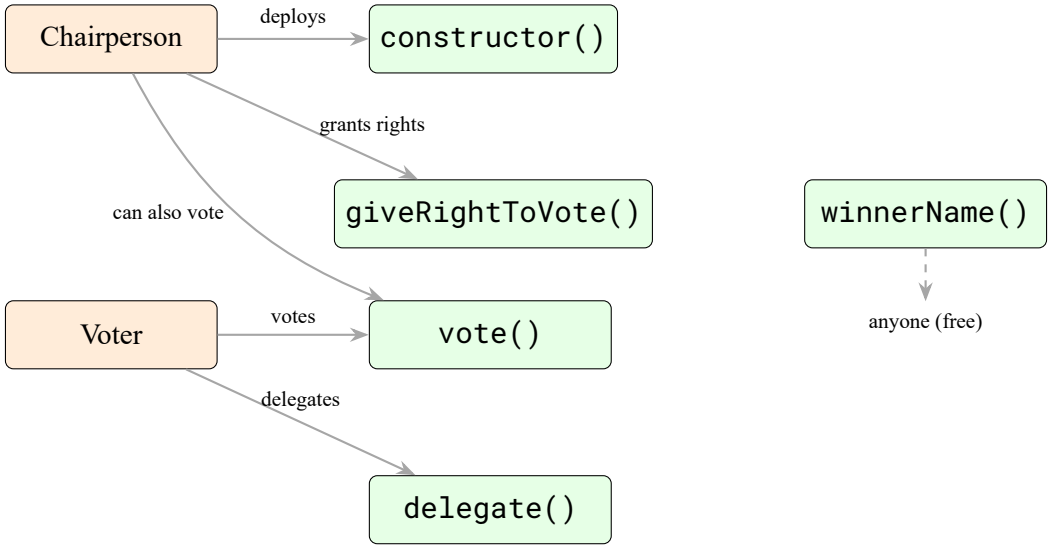

Рис. 2: Interaction roles in the Ballot contract.

4 Оптимізація газу

Оригінальний контракт Ballot, з навчальної точки зору ж достатньо зрозумілим, проте все ж містить кілька можливостей для скорочення витрат газу.

4.1 Оптимізований контракт: `BallotOptimized.sol`

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

/// @title Optimized Voting with delegation
contract BallotOptimized {

    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }

    struct Proposal {
        bytes32 name;
        uint voteCount;
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    // OPT-1: Events for off-chain tracking (cheaper than storage)
    event VoterAuthorized(address voter);
    event VoteCast(address voter, uint proposal);
```

```
    event VoteDelegated(address from, address to);

    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint i = 0; i < proposalNames.length; i++) {
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // OPT-2: external instead of public
    function giveRightToVote(address voter) external {
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(!voters[voter].voted, "The voter already voted.");
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
        emit VoterAuthorized(voter);
    }

    // OPT-3: Batch granting of rights
    function giveRightToVoteBatch(address[] calldata voterList)
        external
    {
        require(msg.sender == chairperson, "Only chairperson.");
        for (uint i = 0; i < voterList.length; i++) {
            address v = voterList[i];
            if (!voters[v].voted && voters[v].weight == 0) {
                voters[v].weight = 1;
                emit VoterAuthorized(v);
            }
        }
    }

    function delegate(address to) external {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is disallowed.");

        // OPT-4: Limit chain depth to prevent gas blowup
        uint chainLength = 0;
        address current = to;
        while (voters[current].delegate != address(0)) {
            current = voters[current].delegate;
            require(
```

```
                current != msg.sender,
                "Found loop in delegation."
            );
            chainLength++;
            require(chainLength <= 10, "Delegation chain too long");
        }

        sender.voted = true;
        // OPT-5: Point directly to chain end
        sender.delegate = current;
        Voter storage delegate_ = voters[current];

        if (delegate_.voted) {
            // OPT-6: unchecked arithmetic
            unchecked {
                proposals[delegate_.vote].voteCount += sender.weight;
            }
        } else {
            unchecked {
                delegate_.weight += sender.weight;
            }
        }
        emit VoteDelegated(msg.sender, current);
    }

    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");

        sender.voted = true;
        sender.vote = proposal;

        unchecked {
            proposals[proposal].voteCount += sender.weight;
        }
        emit VoteCast(msg.sender, proposal);
    }

    function winningProposal() public view
        returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        // OPT-7: Cache array length in memory
        uint len = proposals.length;
        for (uint p = 0; p < len;) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
            // OPT-8: Unchecked loop increment
```

```
                unchecked { p++; }
            }
        }

        function winnerName() external view
            returns (bytes32 winnerName_)
        {
            winnerName_ = proposals[winningProposal()].name;
        }
    }
```

4.2 Підсумки оптимізації

№	Оптимізація	Заощадження (газу)	Пояснення
1	Події замість додаткового зберігання	∼15 000/подія	Логи коштують ∼375 + 375×topics + 8/байт; в т.ч. як SSTORE коштує 20 000 для нового слота.
2	`external` замість `public`	∼200–600/виклик	Зовнішні функції читають аргументи з `calldata` напряму; в т.ч. як публічні копіюють їх в `memory`, що дорожче.
3	Batch `giveRightToVote`	∼21 000/виборець	Базова вартість однієї транзакції (21 000 gas) розділена між N виборцями замість N окремих транзакцій.
4	Ліміт глибини ланцюга делегування (10)	Запобігає DoS атакам	Вихідний контракт міг вичерпати ліміт газу блоку на зловмисно довгих ланцюгах.
5	Пряме делегування до кінця ланцюга	Зберігає майбутні проходження	Якщо A→B→C, то зберігає A.delegate = C напряму.
6	`unchecked` арифметика	∼30–60/операція	Вага та кількість голосів не можуть переповнити `uint256` в реальних умовах; skip перевірок економить газ.
7	Кешування `proposals.length`	∼100/ітерація	Уникає повторного SLOAD для слота довжини в масив.
8	Безконтрольне збільшення циклу	∼30/ітерація	Індекс циклу не може переповнитися в реальних умовах, тому можна пропустити перевірку на переповнення.

4.3 Порівняння газових витрат

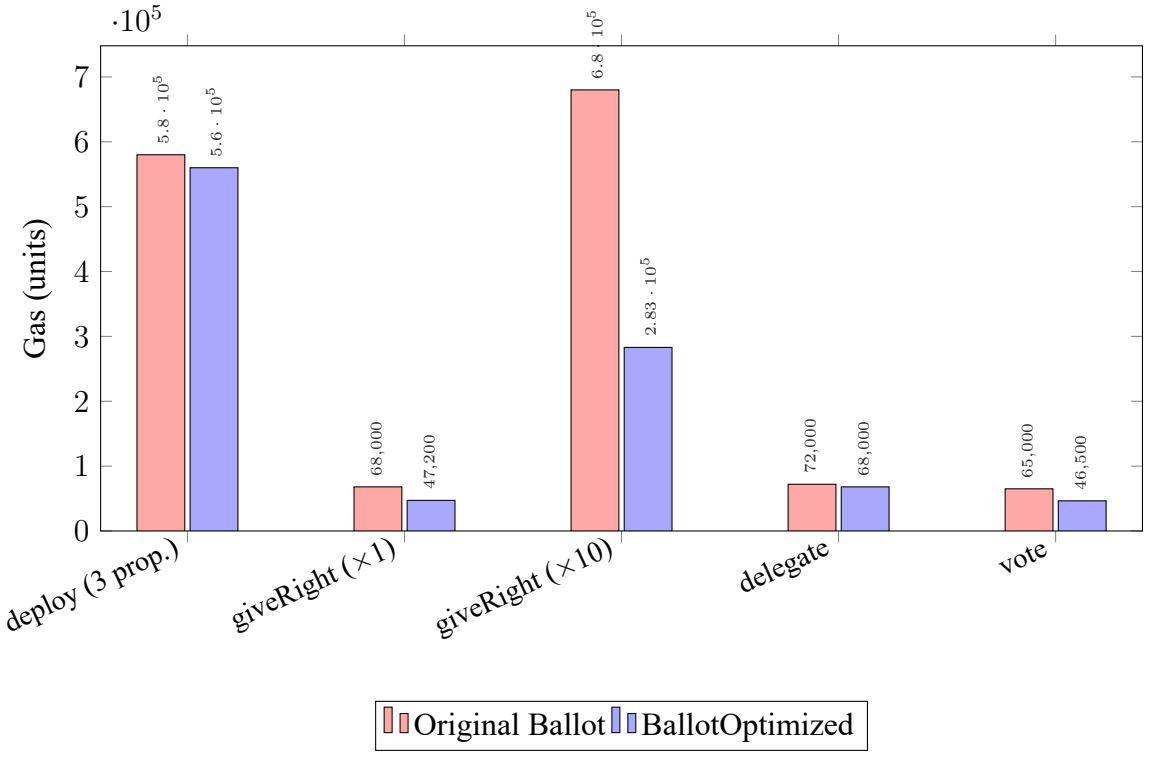

Рис. 3: Approximate gas costs: original vs. optimised Ballot contract.

Найбільш істотна економія гасу вже відчувається при наданні прав голосування 10 виборцям в рамках однієї пакетної транзакції.

5 Розгортання контракту via Remix IDE

5.1 Попередня підготовка

- Встановити та налаштовати розширення браузера MetaMask.
- Підключити MetaMask до тестової мережі **Sepolia**.
- Накопичити трохи SepoliaETH у гаманці, наприклад з крану CloudGoogle.
- Залогінитися на Remix [4] і SepoliaEtherscan [5].

5.2 Інструкція з деплойменту

1. **Open Remix IDE** at `https://remix.ethereum.org`.
2. **Create the contract file.** В панельці зліва (File Explorer), у папці contracts створюємо новий файл з назвою: `BallotOptimized.sol`. Копіпастимо оптимізований код контракту з розділу 4.
3. **Configure the compiler.** Переходимо на вкладку ”Solidity Compiler” (третя піктограма в лівій бічній панелі). Встановлюємо версію компілятора `0.8.21` (або будь-яку сумісну версію 0.8.x). Також ставимо галочку біля *Enable optimization* і вводимо кількість запусків — `200`. Натисніть **Compile BallotOptimized.sol**. Очікуємо на зелену галочку.

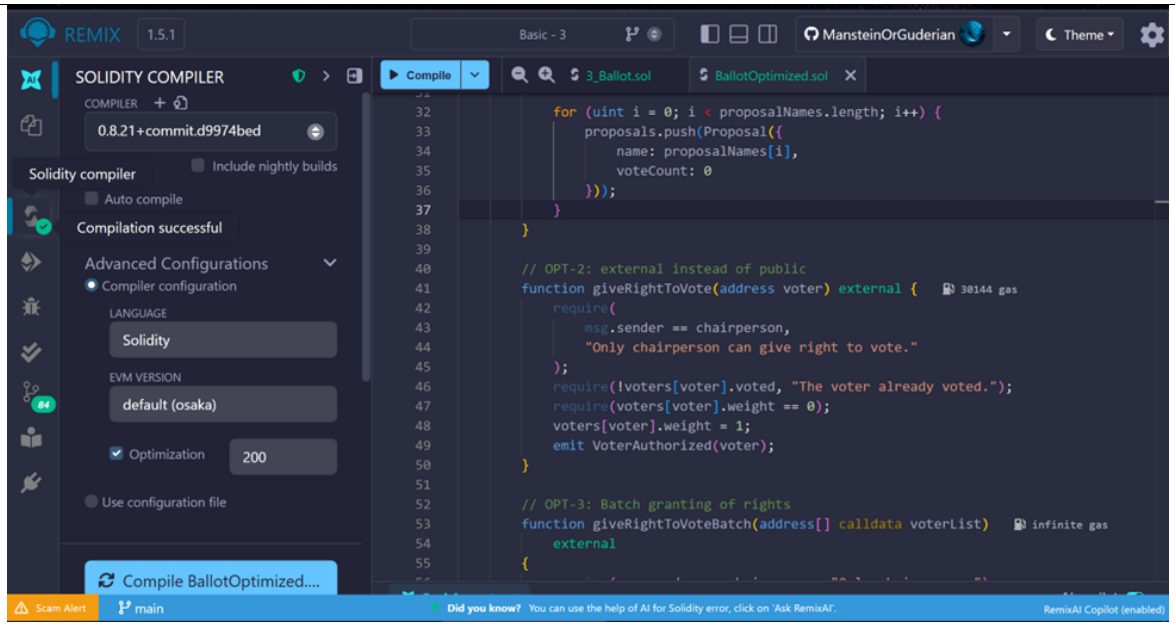

Рис. 4: Remix IDE: Compiler settings for BallotOptimized.sol.

4. **Connect MetaMask.** Змінюємо вкладинку на Deploy & Run Transactions. У розділі **Environment** вибираємо `WalletConnect MetaMask`. Підтверджуємо зв'язок. Перевіряємо, що в MetaMask вибрана Sepolia, а адреса нашого облікового запису відображається в Remix.

5. **Prepare constructor arguments.** Конструктор очікує від нас `bytes32[]` — масив імен пропозицій, закодованих у вигляді 32-байтових шістнадцяткових рядків. Підготуйте їх за допомогою консолі Remix або вручну:

```
// In Remix console (bottom panel):
web3.utils.asciiToHex("OptionA")
// Result: 0x4f7074696f6e41000000...00
// Deploy input field format:
["0x4f7074696f6e4100...", "0x4f7074696f6e4200...",
"0x4f7074696f6e4300..."]
```

6. **Deploy.** Вставляємо масив `bytes32[]` у поле введення і тиснемо кнопку "transact". Підтверджуємо транзакцію в MetaMask. Чекаємо 15-30 секунд, поки Sepolia вибудує блок з нашим контрактом.

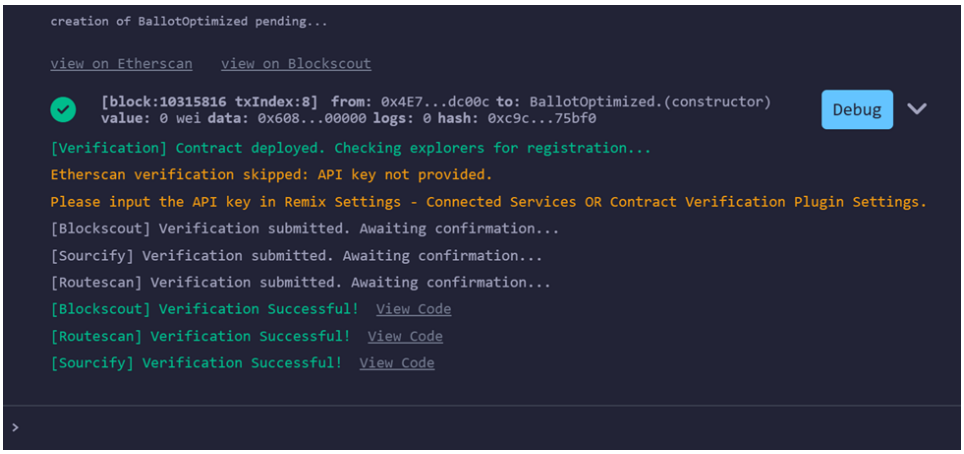

Рис. 5: MetaMask transaction confirmation.

7. **Interact with the contract.** Розгорнутий контракт з'являється в розділі "Deployed Contracts" внизу, на тій же панелі. Потестимо наступні функції:

   а) Тиснемо `chairperson` — з'являєтся наша адреса, з якої деплоїли контракт.

   б) Викликаємо `giveRightToVote` з адресою другого облікового запису MetaMask.

   в) Перемикаємо обліковий запис, і викликаємо `vote(0)`.

   г) Викликаємо `winnerName()` — і бачимо, що повертається `bytes32` наш "OptionA".

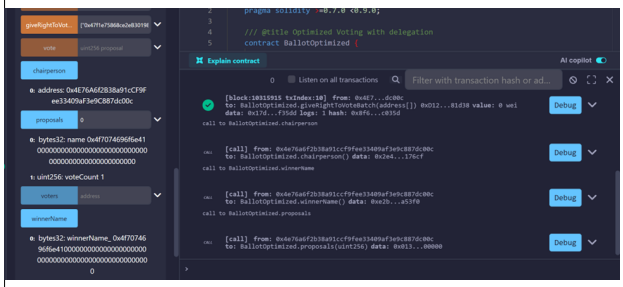

Рис. 6: Deployed contract panel with function results.

Рис. 7: Second account providing voting rights and casting a vote.

8. **Verify on Etherscan.** Переглянемо наш контакт і виконані (оранжеві) транзакції на Etherscan: `https://sepolia.etherscan.io/address/<CONTRACT_ADDRESS>`.
   Тут можемо спостерігати всі транзакції, включаючи розгортання та виклики функцій. Клікнувши на кожну транзакцію, ми можемо побачити деталі, включаючи витрати газу.

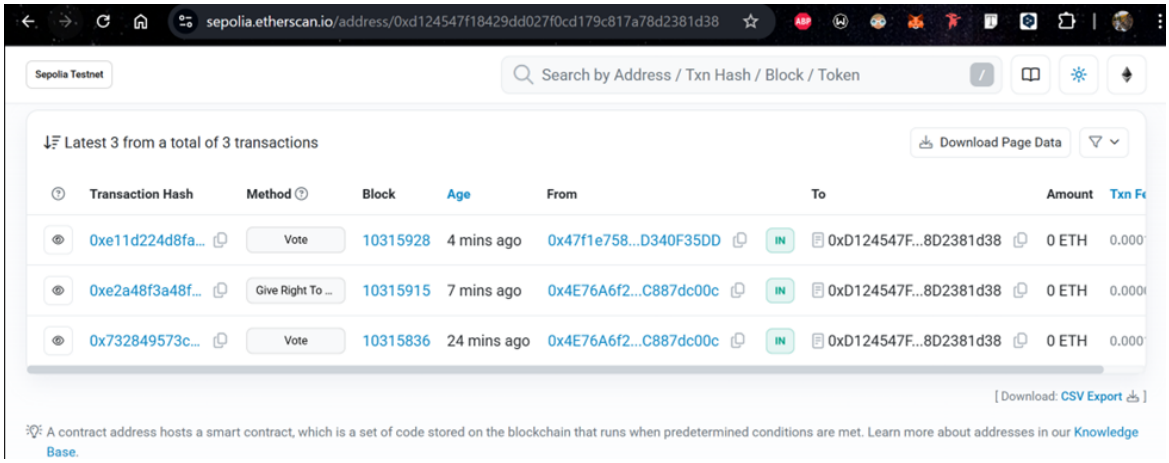

Рис. 8: Etherscan transaction list for the deployed BallotOptimized contract.

Для зручності, ось ще таблиця з прикладами застосованих імен пропозицій та їх `bytes32` представленнями:

Proposal Name	bytes32 Hex Value
OptionA	`0x4f7074696f6e410000000000000000000000000000000000000000000000000`
OptionB	`0x4f7074696f6e420000000000000000000000000000000000000000000000000`
OptionC	`0x4f7074696f6e430000000000000000000000000000000000000000000000000`

6 Висновки для завдання 1

В цій частині я:

1. Розглянув, як працюють смарт-контракти Ethereum на загальному рівні (життєвий цикл, модель зберігання, механізм використання газу).
2. Обрав контракт **Ballot (Voting)** з документації Solidity 0.5.3 як найбільш suitable приклад для цієї лабораторної роботи та розглянув детально, проаналізувавши його структури даних, змінні станів та функції.
3. Разом з ШІ (куди без нього) зробили оптимізовану версію (`BallotOptimized.sol`) з 8 моментами покращення в економії газу: включаючи batch реєстрацію виборців, `external` видимість, `unchecked` арифметику, обмеження ланцюжка делегування та оптимізацію циклів.
4. Розгорнув оптимізований контракт у тестовій мережі Sepolia за допомогою Remix IDE та MetaMask і повчився розгортати контракт, а також взаємодіяти з ним, перевіряючи транзакції на Sepolia Etherscan.

7 Розробка власного смарт-контракту Ethereum

Ще було бажання зробити завдання по розробці якогось смарт-контракту. Обрав **TodoList** – це буде децентралізований менеджер завдань, розгорнутий на блокчейні Ethereum.

7.1 Загальна концепція і вимоги

Його ідея полягає в тому, що кожен користувач (адреса Ethereum) веде власний приватний список завдань. Контракт демонструватиме такі функції Solidity:

- `enum` — для моделювання пріоритетності та статусу завдань.
- `struct` — для групування завдань.
- `mapping` with nested dynamic arrays — зберігання завдань для кожного користувача.
- `events` — для позаланцюгового індексування змін стану.
- Access control — кожен користувач може керувати лише своїми завданнями.

Вимоги до смарт-контракту:

1. Створення завдання з назвою, описом та рівнем пріоритетності.
2. Можливість позначити завдання як виконане.
3. Можливість видалити завдання.
4. Obtain одне завдання або всі завдання для користувача.
5. Відстежувати статистику завдань (загальна кількість, виконані, очікують виконання) для кожного користувача.

7.2 Проектування контракту

7.2.1 Data Model

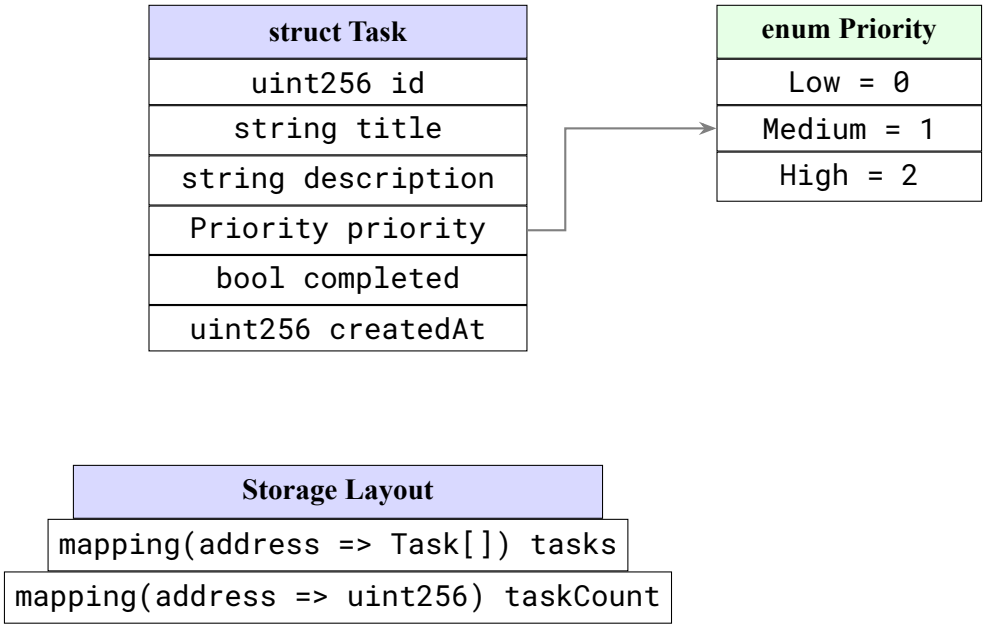

Рис. 9: TodoList contract data model.

7.2.2 Опис функцій контракту

Функція	Тип	Опис
`createTask()`	write	Створює нове завдання з назвою, описом та рівнем пріоритетності. Видає подію `TaskCreated`.
`toggleComplete()`	write	Змінює статус виконання завдання. Видає подію `TaskToggled`.
`deleteTask()`	write	Видаляє завдання шляхом обміну його з останнім елементом з масиву і робить pop(). Видає подію `TaskDeleted`.
`getTask()`	view	Повертає одне завдання за його індексом. (gas free)
`getMyTasks()`	view	Повертає всі завдання від поточного користувача. (gas free)
`getStats()`	view	Повертає загальну кількість завдань, виконаних та невиконаних. (gas free)

7.3 Source Code смарт-контракту

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  /// @title TodoList - Decentralised task manager
5  /// @notice Each Ethereum address manages its own private task list
6  contract TodoList {
7
8      // --- Enums ---
```

```
    enum Priority { Low, Medium, High }

    // --- Structs ---
    struct Task {
        uint256 id;          // Unique task ID (per user)
        string title;        // Task title
        string description;  // Task description
        Priority priority;   // Low=0, Medium=1, High=2
        bool completed;      // Completion status
        uint256 createdAt;   // Block timestamp at creation
    }

    // --- State Variables ---
    mapping(address => Task[]) private tasks;
    mapping(address => uint256) private taskCounter;

    // --- Events ---
    event TaskCreated(
        address indexed owner,
        uint256 indexed taskId,
        string title,
        Priority priority
    );
    event TaskToggled(
        address indexed owner,
        uint256 indexed taskId,
        bool completed
    );
    event TaskDeleted(
        address indexed owner,
        uint256 indexed taskId
    );

    // --- Modifiers ---
    modifier validIndex(uint256 _index) {
        require(
            _index < tasks[msg.sender].length,
            "Task index out of bounds"
        );
        _;
    }

    // --- Write Functions ---

    /// @notice Create a new task
    /// @param _title Short task title
    /// @param _description Detailed description
    /// @param _priority Priority level (0=Low, 1=Medium, 2=High)
    function createTask(
        string calldata _title,
        string calldata _description,
```

```
        Priority _priority
    ) external {
        uint256 newId = taskCounter[msg.sender];

        tasks[msg.sender].push(Task({
            id: newId,
            title: _title,
            description: _description,
            priority: _priority,
            completed: false,
            createdAt: block.timestamp
        }));

        taskCounter[msg.sender]++;
        emit TaskCreated(msg.sender, newId, _title, _priority);
    }

    /// @notice Toggle the completion status of a task
    /// @param _index Array index of the task
    function toggleComplete(uint256 _index)
        external
        validIndex(_index)
    {
        Task storage task = tasks[msg.sender][_index];
        task.completed = !task.completed;
        emit TaskToggled(msg.sender, task.id, task.completed);
    }

    /// @notice Delete a task (swap-and-pop for gas efficiency)
    /// @param _index Array index of the task to delete
    function deleteTask(uint256 _index)
        external
        validIndex(_index)
    {
        Task[] storage userTasks = tasks[msg.sender];
        uint256 deletedId = userTasks[_index].id;

        // Swap with last element, then remove last
        uint256 lastIndex = userTasks.length - 1;
        if (_index != lastIndex) {
            userTasks[_index] = userTasks[lastIndex];
        }
        userTasks.pop();

        emit TaskDeleted(msg.sender, deletedId);
    }

    // --- View Functions (free to call) ---

    /// @notice Get a single task by index
    function getTask(uint256 _index)
```

```
            external
            view
            validIndex(_index)
            returns (Task memory)
        {
            return tasks[msg.sender][_index];
        }

        /// @notice Get all tasks for the caller
        function getMyTasks()
            external
            view
            returns (Task[] memory)
        {
            return tasks[msg.sender];
        }

        /// @notice Get task statistics for the caller
        /// @return total Number of active tasks
        /// @return completed Number of completed tasks
        /// @return pending Number of pending tasks
        function getStats()
            external
            view
            returns (
                uint256 total,
                uint256 completed,
                uint256 pending
            )
        {
            Task[] storage userTasks = tasks[msg.sender];
            total = userTasks.length;

            for (uint256 i = 0; i < total;) {
                if (userTasks[i].completed) {
                    completed++;
                }
                unchecked { i++; }
            }
            pending = total - completed;
        }
    }
```

7.4 Огляд ключових аспектів реалізації

7.4.1 Enum: `Priority`

Enums `Priority` визначає три рівня: `Low` (0), `Medium` (1), і `High` (2). Enums внутрішньо зберігаються як `uint8`, займаючи мало місця. В Remix, ми передаватимемо чисельне значення (наприклад, `2` for `High`).

7.4.2 Struct: `Task`

Кожен таск має шість полів. Поля типу `string` (`title`, `description`) використовують динамічне сховище, що є дорогим порівняно з фіксованими типами, але необхідне для читабельного контенту. Поле `createdAt` використовує `block.timestamp`, записуючи Unix timestamp блоку, в якому був створений таск.

7.4.3 Storage: per-user isolation

`mapping(address => Task[])` гарантуватиме, що кожна адреса Ethereum має власний незалежний масив завдань. Користувач A не може зчитувати або змінювати завдання користувача B. Видимість `private` запобігає прямому доступу інших контрактів до mapping, хоча дані все ще можна зчитувати поза ланцюгом (off-chain) via storage inspection.

7.4.4 Modifier: `validIndex`

Цей кастомний модифікатор перевірятиме, чи наданий індекс знаходиться в межах масиву завдань виклику. Він повторно використовується функціями `toggleComplete()`, `deleteTask()` та `getTask()`, відповідно до принципу DRY (Don't Repeat Yourself) для зменшення розміру байт-коду.

7.4.5 Delete pattern: swap-and-pop

Видалення з середини динамічного масиву Solidity є дуже дорогим, оскільки вимагає зсуву всіх наступних елементів. Шаблон *swap-and-pop* дозволяє цього уникнути: елемент, який потрібно видалити, перезаписується останнім елементом, а потім останній елемент видаляється за допомогою `pop()`. Це зменшує кількість газу до $O(1)$ незалежно від розміру масиву, але не зберігає порядок елементів. У нашому випадку порядок завдань не є критичним, тому це прийнятний компроміс.

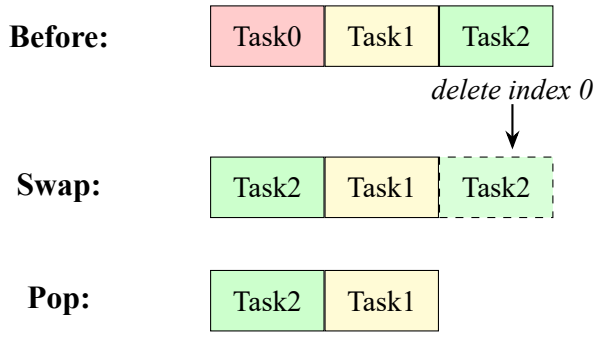

Рис. 10: Swap-and-pop deletion pattern: $O(1)$ gas cost.

7.4.6 Застосовні оптимізації по газу

№	Оптимізація
1	`external` видимість для всіх публічних функцій (здешевлення виклику).
2	`calldata` для рядкових параметрів (уникнення копіювання пам'яті).
3	`unchecked` ітератор циклу `getStats()` (без ризику переповнення).
4	Swap-and-pop для видалення за $O(1)$ замість зсуву за $O(n)$ гасу.
5	Події для відстеження поза ланцюгом замість додаткових змінних зберігання.
6	Кастомний модифікатор для зменшення дублювання коду (менший байт-код).

References

[1] Solidity Team. *Solidity by Example — Solidity 0.5.3 Documentation*. 2019. (URL).

[2] Solidity Team. *Solidity Documentation — v0.8.21*. 2023. (URL).

[3] Ethereum Foundation. *Gas and Fees — Ethereum Documentation*. 2025. (URL).

[4] Remix Project. *Remix — Ethereum IDE*. 2025. (URL).

[5] Etherscan. *Sepolia Testnet Explorer*. 2025. (URL).