

**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

<b>1</b>	<b>Вступ</b>	<b>1</b>
<b>2</b>	<b>Як працюють смарт-контракти (in general)</b>	<b>1</b>
2.1	Визначення	1
2.2	Стан, газ і вартість контрактів (EVM Storage and Gas)	2
2.3	Події (Events)	2
2.4	Діаграма життєвого циклу смарт-контракту	2
<b>3</b>	<b>Ballot Contract</b>	<b>2</b>
3.1	Призначення договору	2
3.2	Структури даних	2
3.2.1	struct Proposal (пропозиція)	3
3.2.2	struct Voter	3
3.3	Станові змінні	3
3.4	Конструктор	4
3.5	Функція: giveRightToVote(address voter)	4
3.6	Функція: delegate(address to)	4
3.7	Функція: vote(uint proposal)	5
3.8	Функції перегляду: winningProposal() and winnerName()	6
3.9	Діаграма взаємодії контрактів	6
<b>4</b>	<b>Оптимізація газу</b>	<b>7</b>
4.1	Оптимізований контракт: BallotOptimized.sol	7
4.2	Підсумки оптимізації	10
4.3	Порівняння газових витрат	11
<b>5</b>	<b>Розгортання контракту via Remix IDE</b>	<b>11</b>
5.1	Попередня підготовка	11
5.2	Інструкція з деплою	11
<b>6</b>	<b>Висновки для завдання 1</b>	<b>13</b>
<b>7</b>	<b>Розробка власного смарт-контракту Ethereum</b>	<b>14</b>
7.1	Загальна концепція і вимоги	14
7.2	Проектування контракту	15
7.2.1	Data Model	15
7.2.2	Опис функцій контракту	15
7.3	Source Code смарт-контракту	15
7.4	Огляд ключових аспектів реалізації	18
7.4.1	Enum: Priority	18
7.4.2	Struct: Task	19
7.4.3	Storage: per-user isolation	19
7.4.4	Modifier: validIndex	19
7.4.5	Delete pattern: swap-and-pop	19

7.4.6	Застосовні оптимізації по газу . . . . .	19
7.5	Деплоймент і тестування контракту . . . . .	20
<b>8</b>	<b>Висновки по тасці 2</b>	<b>22</b>

# 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 Serolia та спробую покращити його газову ефективність. А потім ще спробую накласти 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  $\text{gas used} \times \text{gas price}$  в ЕТН. 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` (продувати) певні події. Ці події зберігаються в ланцюжку як журнали транзакцій (це набагато дешевше, ніж зберігання:  $\sim 375 \text{ gas base} + 375 \text{ gas per indexed topic} + 8 \text{ gas per byte of data}$ ). Вони доступні для запиту поза ланцюжком, але *не* доступні для читання іншими контрактами.

## 2.4 Діаграма життєвого циклу смарт-контракту

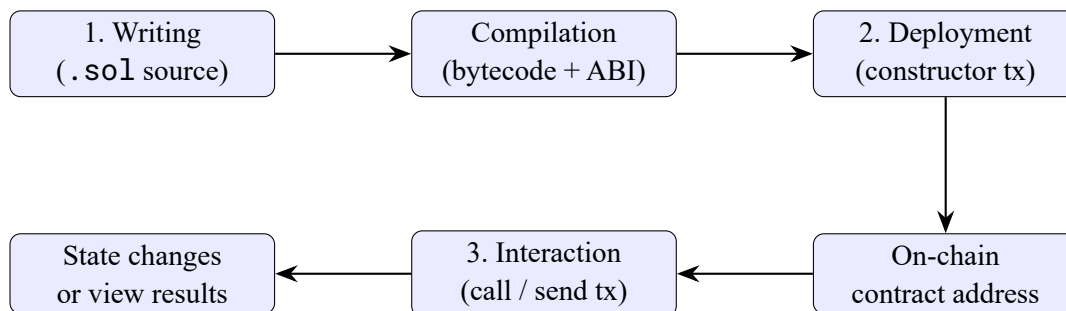


Рис. 1: Smart contract lifecycle on Ethereum.

## 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)

```
1 struct Proposal {
2     bytes32 name;      // Short name (up to 32 bytes)
3     uint voteCount;    // Accumulated votes
4 }
```

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

```
1 struct Voter {
2     uint weight;        // Voting power (0 = no right, 1+ = can vote)
3     bool voted;         // True after voting or delegating
4     address delegate;   // Delegation target; address(0) if none
5     uint vote;          // Index of chosen proposal (meaningful only
6                         // if voted==true && delegate==address(0))
7 }
```

Поле	Опис
weight	Право голосу. Ініціалізується 0 для всіх адрес. Встановлюється 1 головуючим за допомогою функції <code>giveRightToVote()</code> . Збільшується, коли інші виборці делегують право голосу цій адресі.
voted	Boolean flag. Встановлюється на <code>true</code> як у випадку, коли виборець голосує сам <i>або</i> 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 <code>proposals[]</code> . Використовується лише тоді, коли виборець голосує безпосередньо (не делегує).

## 3.3 Станові змінні

```
1 address public chairperson;
2 mapping(address => Voter) public voters;
3 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 <code>Voter</code> .
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

```
1  constructor(bytes32[] memory proposalNames) public {
2      chairperson = msg.sender;
3      voters[chairperson].weight = 1;
4
5      for (uint i = 0; i < proposalNames.length; i++) {
6          proposals.push(Proposal({
7              name: proposalNames[i],
8              voteCount: 0
9          }));
10     }
11 }
```

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)`

```
1  function giveRightToVote(address voter) public {
2      require(msg.sender == chairperson,
3          "Only chairperson can give right to vote.");
4      require(!voters[voter].voted,
5          "The voter already voted.");
6      require(voters[voter].weight == 0);
7      voters[voter].weight = 1;
8  }
```

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:

```

1  function delegate(address to) public {
2      Voter storage sender = voters[msg.sender];
3      require(!sender.voted, "You already voted.");
4      require(to != msg.sender, "Self-delegation is disallowed.");
5
6      while (voters[to].delegate != address(0)) {
7          to = voters[to].delegate;
8          require(to != msg.sender, "Found loop in delegation.");
9      }
10
11     sender.voted = true;
12     sender.delegate = to;
13     Voter storage delegate_ = voters[to];
14
15     if (delegate_.voted) {
16         proposals[delegate_.vote].voteCount += sender.weight;
17     } else {
18         delegate_.weight += sender.weight;
19     }
20 }

```

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 ( $A \rightarrow B \rightarrow C$ ) до тих пір, поки не знайде виборця, який не делегує далі. Цей цикл також перевіряє наявність циклічного делегування.
4. Позначає виборця як проголосованого 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)`

```

1  function vote(uint proposal) public {
2      Voter storage sender = voters[msg.sender];
3      require(sender.weight != 0, "Has no right to vote");
4      require(!sender.voted, "Already voted.");
5
6      sender.voted = true;
7      sender.vote = proposal;

```



```
8     proposals[proposal].voteCount += sender.weight;
9 }
```

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()`

```
1  function winningProposal() public view
2      returns (uint winningProposal_)
3  {
4      uint winningVoteCount = 0;
5      for (uint p = 0; p < proposals.length; p++) {
6          if (proposals[p].voteCount > winningVoteCount) {
7              winningVoteCount = proposals[p].voteCount;
8              winningProposal_ = p;
9          }
10     }
11 }
12
13 function winnerName() public view
14     returns (bytes32 winnerName_)
15 {
16     winnerName_ = proposals[winningProposal()].name;
17 }
```

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:

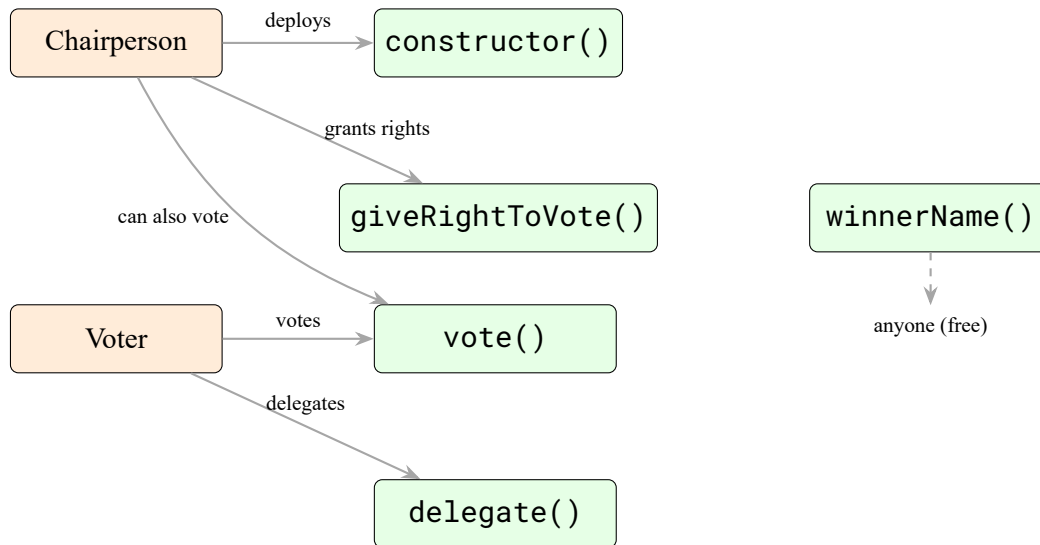


Рис. 2: Interaction roles in the Ballot contract.

## 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

```

1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  /// @title Optimized Voting with delegation
5  contract BallotOptimized {
6
7      struct Voter {
8          uint weight;
9          bool voted;
10         address delegate;
11         uint vote;
12     }
13
14     struct Proposal {
15         bytes32 name;
16         uint voteCount;
17     }
18
19     address public chairperson;
20     mapping(address => Voter) public voters;
21     Proposal[] public proposals;
22
23     // OPT-1: Events for off-chain tracking (cheaper than storage)
24     event VoterAuthorized(address voter);
25     event VoteCast(address voter, uint proposal);
  
```

```

26     event VoteDelegated(address from, address to);
27
28     constructor(bytes32[] memory proposalNames) {
29         chairperson = msg.sender;
30         voters[chairperson].weight = 1;
31
32         for (uint i = 0; i < proposalNames.length; i++) {
33             proposals.push(Proposal({
34                 name: proposalNames[i],
35                 voteCount: 0
36             }));
37         }
38     }
39
40     // OPT-2: external instead of public
41     function giveRightToVote(address voter) external {
42         require(
43             msg.sender == chairperson,
44             "Only chairperson can give right to vote."
45         );
46         require(!voters[voter].voted, "The voter already voted.");
47         require(voters[voter].weight == 0);
48         voters[voter].weight = 1;
49         emit VoterAuthorized(voter);
50     }
51
52     // OPT-3: Batch granting of rights
53     function giveRightToVoteBatch(address[] calldata voterList)
54         external
55     {
56         require(msg.sender == chairperson, "Only chairperson.");
57         for (uint i = 0; i < voterList.length; i++) {
58             address v = voterList[i];
59             if (!voters[v].voted && voters[v].weight == 0) {
60                 voters[v].weight = 1;
61                 emit VoterAuthorized(v);
62             }
63         }
64     }
65
66     function delegate(address to) external {
67         Voter storage sender = voters[msg.sender];
68         require(!sender.voted, "You already voted.");
69         require(to != msg.sender, "Self-delegation is disallowed.");
70
71         // OPT-4: Limit chain depth to prevent gas blowup
72         uint chainLength = 0;
73         address current = to;
74         while (voters[current].delegate != address(0)) {
75             current = voters[current].delegate;
76             require(

```

```

77         current != msg.sender,
78         "Found loop in delegation."
79     );
80     chainLength++;
81     require(chainLength <= 10, "Delegation chain too long");
82 }
83
84 sender.voted = true;
85 // OPT-5: Point directly to chain end
86 sender.delegate = current;
87 Voter storage delegate_ = voters[current];
88
89 if (delegate_.voted) {
90     // OPT-6: unchecked arithmetic
91     unchecked {
92         proposals[delegate_.vote].voteCount += sender.weight;
93     }
94 } else {
95     unchecked {
96         delegate_.weight += sender.weight;
97     }
98 }
99 emit VoteDelegated(msg.sender, current);
100 }
101
102 function vote(uint proposal) external {
103     Voter storage sender = voters[msg.sender];
104     require(sender.weight != 0, "Has no right to vote");
105     require(!sender.voted, "Already voted.");
106
107     sender.voted = true;
108     sender.vote = proposal;
109
110     unchecked {
111         proposals[proposal].voteCount += sender.weight;
112     }
113     emit VoteCast(msg.sender, proposal);
114 }
115
116 function winningProposal() public view
117     returns (uint winningProposal_)
118 {
119     uint winningVoteCount = 0;
120     // OPT-7: Cache array length in memory
121     uint len = proposals.length;
122     for (uint p = 0; p < len;) {
123         if (proposals[p].voteCount > winningVoteCount) {
124             winningVoteCount = proposals[p].voteCount;
125             winningProposal_ = p;
126         }
127         // OPT-8: Unchecked loop increment

```

```

128         unchecked { p++; }
129     }
130 }
131
132 function winnerName() external view
133     returns (bytes32 winnerName_)
134 {
135     winnerName_ = proposals[winningProposal()].name;
136 }
137 }

```

## 4.2 Підсумки оптимізації

№	Оптимізація	Заощадження (гасу)	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 напряду; 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 $A \rightarrow B \rightarrow C$ , то зберігає $A.delegate = C$ напряду.
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

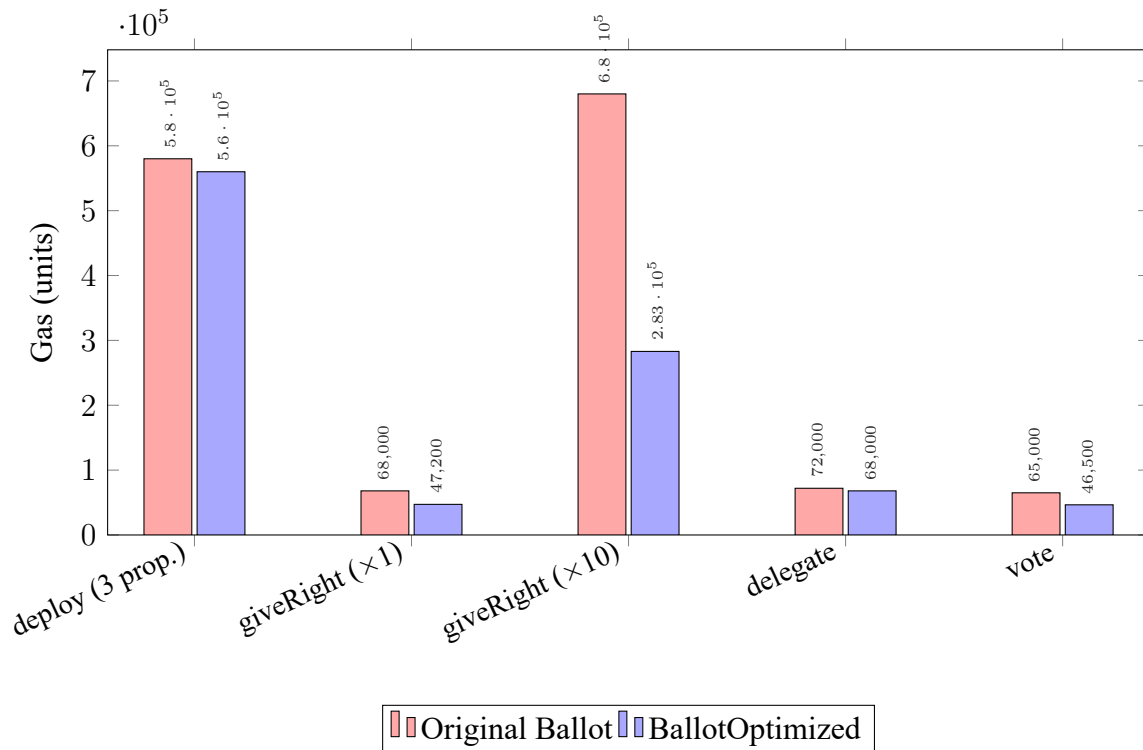


Рис. 3: Approximate gas costs: original vs. optimised Ballot contract.

Найбільш істотна економія газу 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 налаштувати розширення браузера MetaMask.
- Підключити MetaMask до тестової мережі **Sepolia**.
- Накопичити трохи SepoliaETH у гаманці, наприклад з крану [CloudGoogle](#).
- Залогінітися 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`. Копіюємо 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.

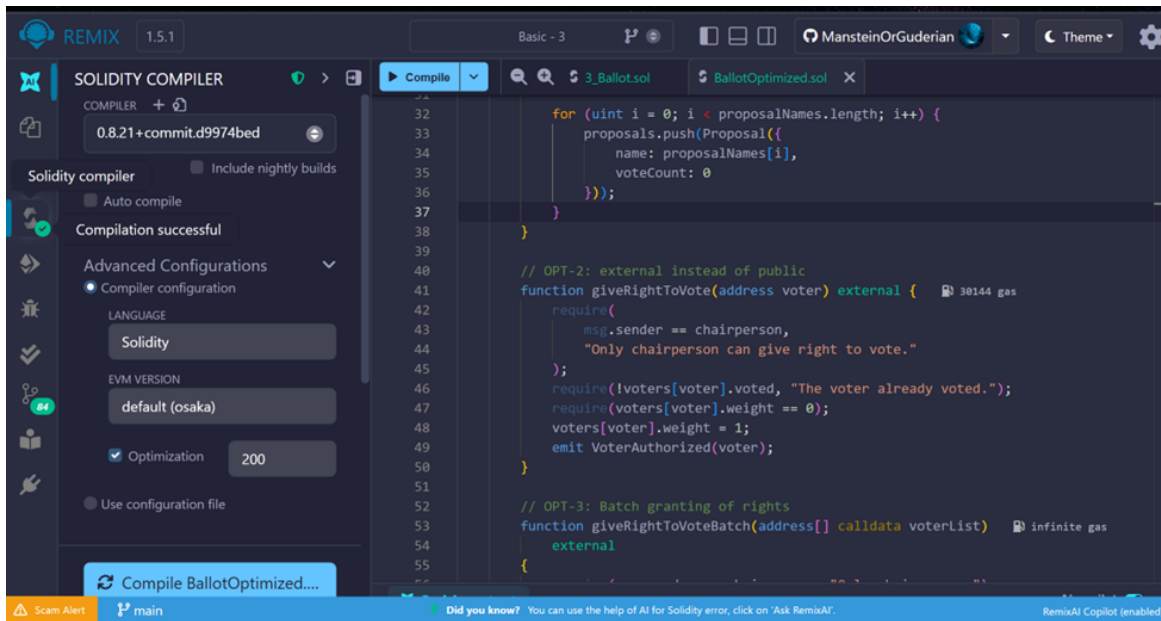


Рис. 4: Remix IDE: Compiler settings for BallotOptimized.sol.

4. **Connect MetaMask.** Змінюємо вкладку 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.

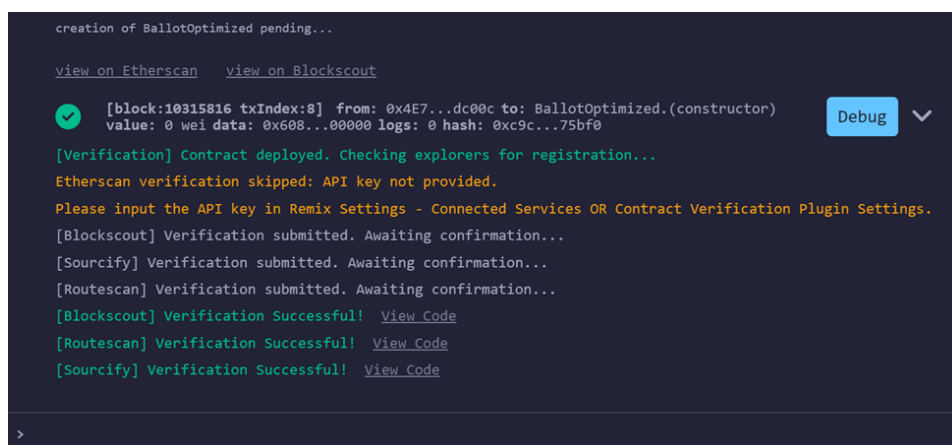


Рис. 5: MetaMask transaction confirmation.

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

- Тиснемо `chairperson` — з'являється наша адреса, з якої деплоїли контракт.
- Викликаємо `giveRightToVote` з адресою другого облікового запису MetaMask.
- Перемикаємо обліковий запис, і викликаємо `vote(0)`.
- Викликаємо `winnerName()` — і бачимо, що повертається `bytes32` наш "OptionA".

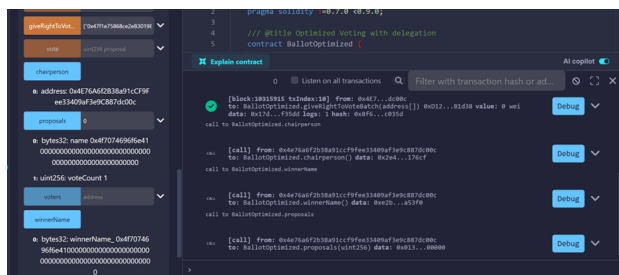


Рис. 6: Deployed contract panel with function results.

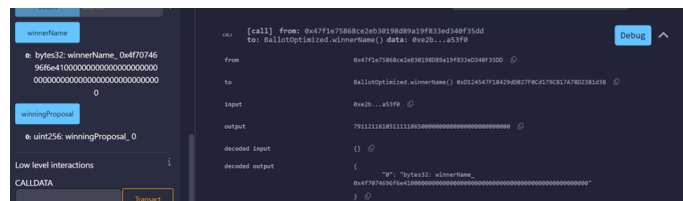


Рис. 7: Second account providing voting rights and casting a vote.

**8. Verify on Etherscan.** Переглянемо наш контакт і виконані (оранжеві) транзакції на Etherscan: [https://sepolia.etherscan.io/address/<CONTRACT\\_ADDRESS>](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.

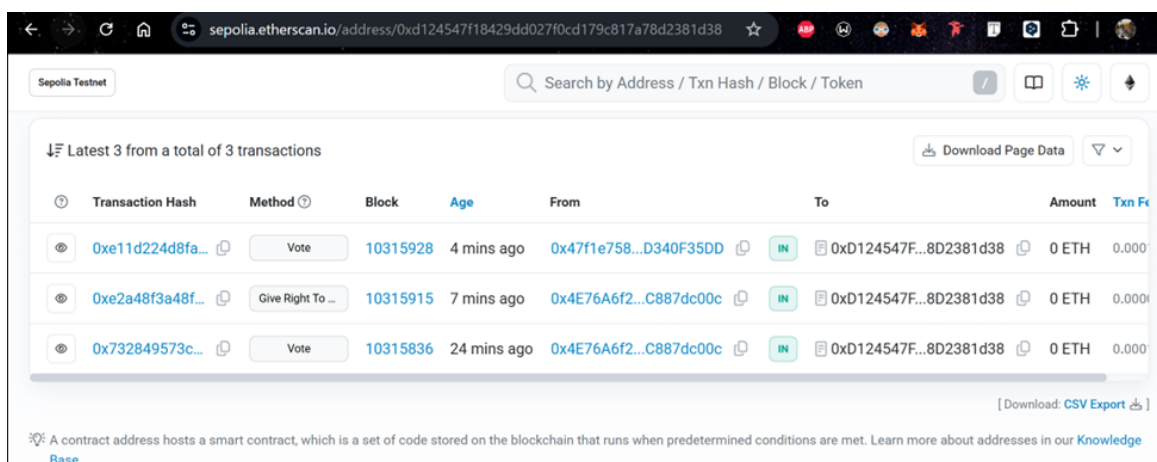


Рис. 8: Etherscan transaction list for the deployed BallotOptimized contract.

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

[illegible]

## 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

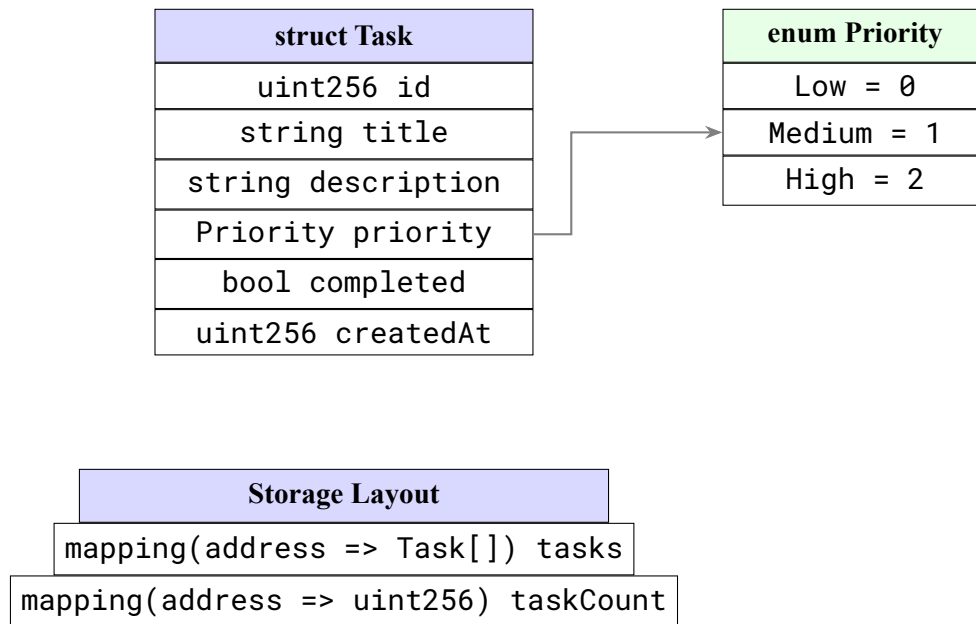


Рис. 9: TodoList contract data model.

### 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 ---
```

```

9      enum Priority { Low, Medium, High }
10
11     // --- Structs ---
12     struct Task {
13         uint256 id;           // Unique task ID (per user)
14         string title;         // Task title
15         string description;    // Task description
16         Priority priority;     // Low=0, Medium=1, High=2
17         bool completed;       // Completion status
18         uint256 createdAt;    // Block timestamp at creation
19     }
20
21     // --- State Variables ---
22     mapping(address => Task[]) private tasks;
23     mapping(address => uint256) private taskCounter;
24
25     // --- Events ---
26     event TaskCreated(
27         address indexed owner,
28         uint256 indexed taskId,
29         string title,
30         Priority priority
31     );
32     event TaskToggled(
33         address indexed owner,
34         uint256 indexed taskId,
35         bool completed
36     );
37     event TaskDeleted(
38         address indexed owner,
39         uint256 indexed taskId
40     );
41
42     // --- Modifiers ---
43     modifier validIndex(uint256 _index) {
44         require(
45             _index < tasks[msg.sender].length,
46             "Task index out of bounds"
47         );
48         _;
49     }
50
51     // --- Write Functions ---
52
53     /// @notice Create a new task
54     /// @param _title Short task title
55     /// @param _description Detailed description
56     /// @param _priority Priority level (0=Low, 1=Medium, 2=High)
57     function createTask(
58         string calldata _title,
59         string calldata _description,

```

```

60     Priority _priority
61 ) external {
62     uint256 newId = taskCounter[msg.sender];
63
64     tasks[msg.sender].push(Task({
65         id: newId,
66         title: _title,
67         description: _description,
68         priority: _priority,
69         completed: false,
70         createdAt: block.timestamp
71     }));
72
73     taskCounter[msg.sender]++;
74     emit TaskCreated(msg.sender, newId, _title, _priority);
75 }
76
77 /// @notice Toggle the completion status of a task
78 /// @param _index Array index of the task
79 function toggleComplete(uint256 _index)
80     external
81     validIndex(_index)
82 {
83     Task storage task = tasks[msg.sender][_index];
84     task.completed = !task.completed;
85     emit TaskToggled(msg.sender, task.id, task.completed);
86 }
87
88 /// @notice Delete a task (swap-and-pop for gas efficiency)
89 /// @param _index Array index of the task to delete
90 function deleteTask(uint256 _index)
91     external
92     validIndex(_index)
93 {
94     Task[] storage userTasks = tasks[msg.sender];
95     uint256 deletedId = userTasks[_index].id;
96
97     // Swap with last element, then remove last
98     uint256 lastIndex = userTasks.length - 1;
99     if (_index != lastIndex) {
100         userTasks[_index] = userTasks[lastIndex];
101     }
102     userTasks.pop();
103
104     emit TaskDeleted(msg.sender, deletedId);
105 }
106
107 // --- View Functions (free to call) ---
108
109 /// @notice Get a single task by index
110 function getTask(uint256 _index)

```

```

111     external
112     view
113     validIndex(_index)
114     returns (Task memory)
115 {
116     return tasks[msg.sender][_index];
117 }
118
119 /// @notice Get all tasks for the caller
120 function getMyTasks()
121     external
122     view
123     returns (Task[] memory)
124 {
125     return tasks[msg.sender];
126 }
127
128 /// @notice Get task statistics for the caller
129 /// @return total Number of active tasks
130 /// @return completed Number of completed tasks
131 /// @return pending Number of pending tasks
132 function getStats()
133     external
134     view
135     returns (
136         uint256 total,
137         uint256 completed,
138         uint256 pending
139     )
140 {
141     Task[] storage userTasks = tasks[msg.sender];
142     total = userTasks.length;
143
144     for (uint256 i = 0; i < total;) {
145         if (userTasks[i].completed) {
146             completed++;
147         }
148         unchecked { i++; }
149     }
150     pending = total - completed;
151 }
152 }

```

## 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 task 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 task.

## 7.4.3 Storage: per-user isolation

`mapping(address => Task[])` гарантуватиме, що кожна адреса Ethereum має власний незалежний масив завдань. Користувач А не може читувати або змінювати завдання користувача В. 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.

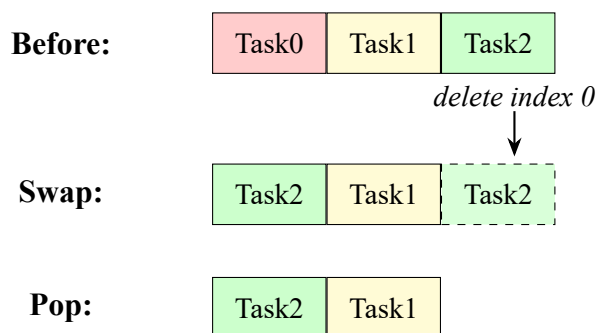


Рис. 10: Swap-and-pop deletion pattern:  $O(1)$  gas cost.

## 7.4.6 Застосовні оптимізації по газу

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

## 7.5 Деплоймент і тестування контракту

1. Створюємо новий файл в Remix у папці `contracts: TodoList.sol`. Додаємо в нього код з розділу 7.3.
2. Компілюємо, як і раніше, з Solidity 0.8.21, з увімкненою оптимізацією (200 runs).

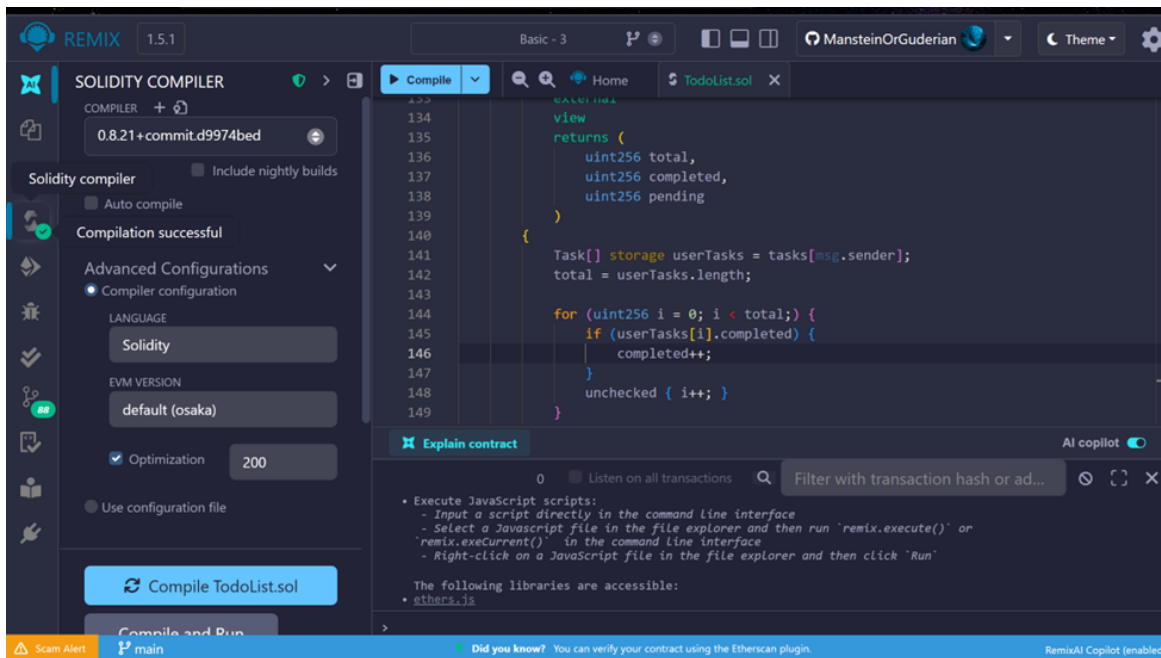


Рис. 11: Compilation settings in Remix IDE

3. **Deploy** via Injected Provider (MetaMask, Sepolia). Контруктор не потребує вхідних аргументів (порівняно з завданням 1) — просто натискаємо **Deploy** і підтверджуємо в MetaMask.

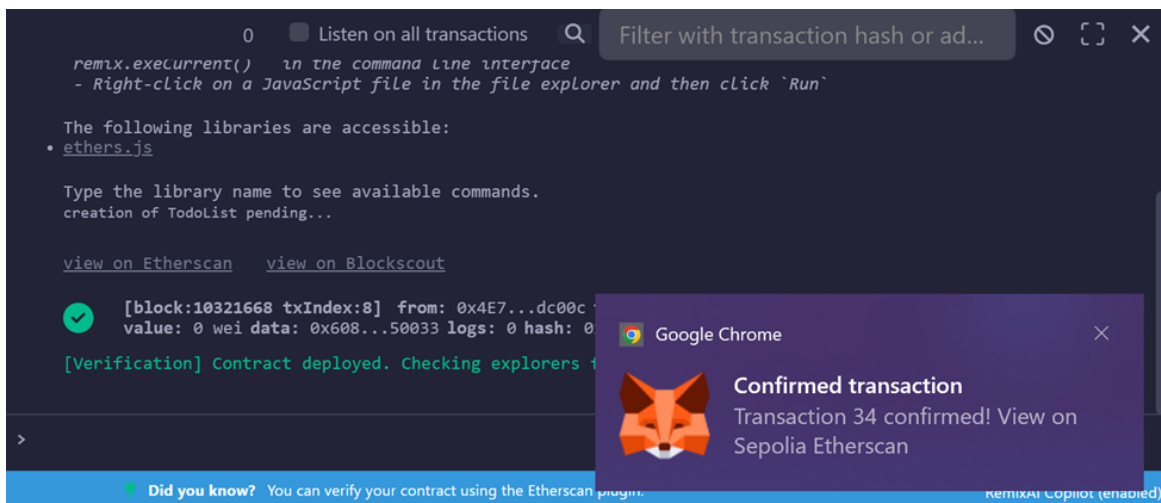


Рис. 12: Deployment result in Remix IDE

### 4. Тестимо роботу:

- а) Викликаємо `createTask` з довільними параметрами. Мені перше прийшло на думку – це список покупок. Тому "Buy groceries", "Milk, 10 Eggs, Bread, Sausages", 1 (Medium priority). І підтверджуємо наш запит-транзакцію в MetaMask.

- б) Викликаємо `createTask` знову. Нехай на цей раз багатостраждальна лаба: "Finish lab report", "Complete Task 2 analysis", 2 (High priority). Теж підтверджуємо.
- в) Викликаємо `getMyTasks` — повертає нам масив з двох, встановлених щойно, задач.
- г) Викликаємо `toggleComplete` з індексом 0 — відмічає першу задачу як виконану.
- д) Викликаємо `getStats` — повертає статус поточних: `total=2`, `completed=1`, `pending=1`.
- е) Викликаємо `deleteTask` with index 0 — видаляє виконану задачу.
- ж) Викликаємо `getMyTasks` — має показати лише одну задачу, яка залишилася.

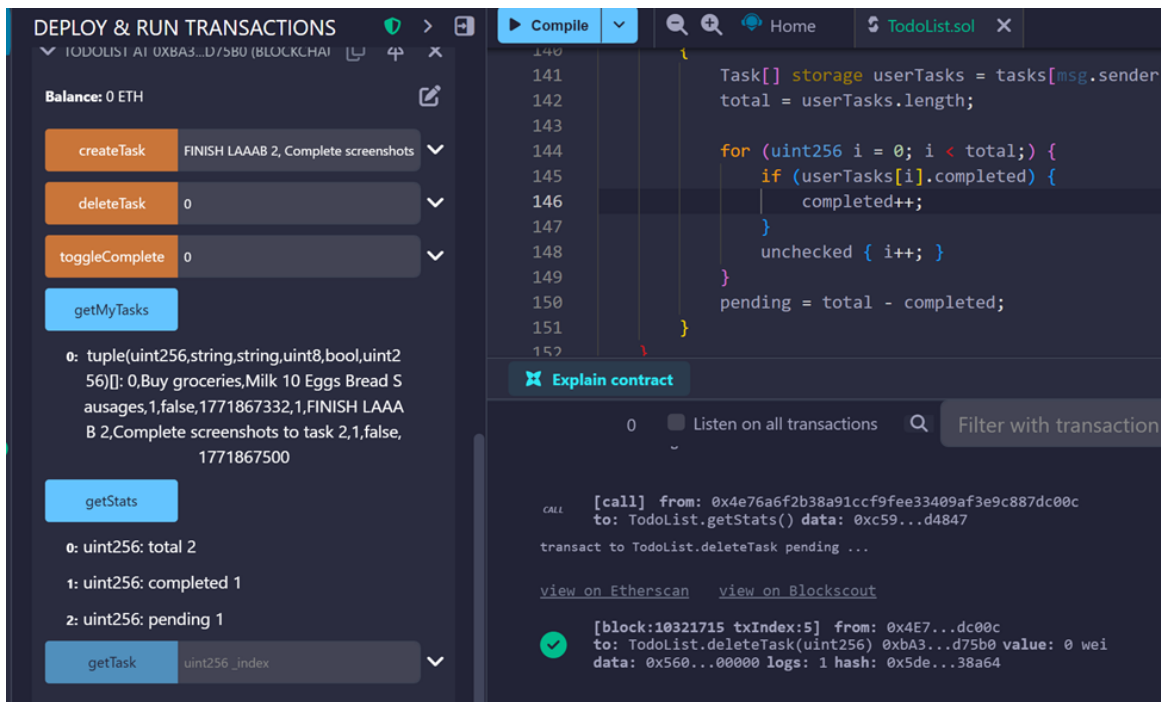


Рис. 13: Interaction results in Remix IDE

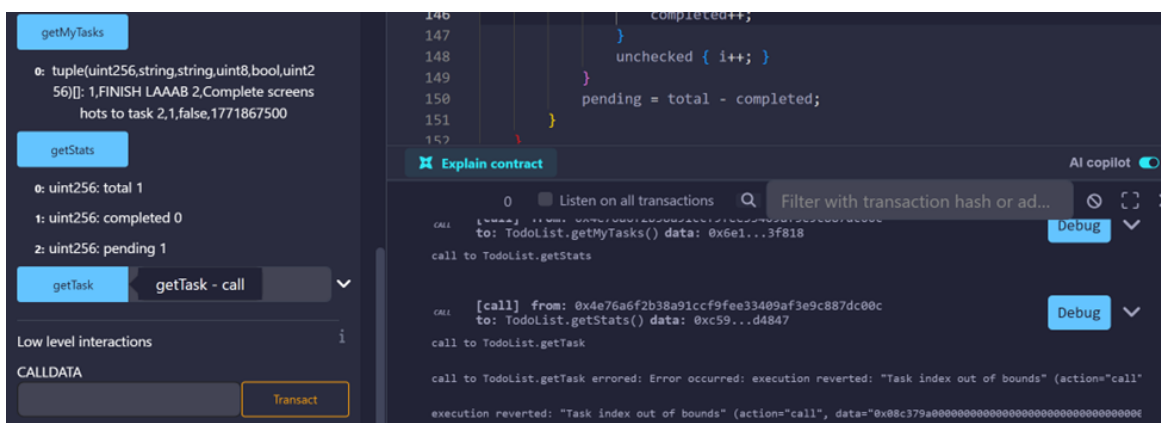


Рис. 14: Result after deleting a task in Remix IDE

## 5. Verify on Etherscan заходимо на

<https://sepolia.etherscan.io/address/<ADDRESS>>. Перевіряємо, що відбулися відповідні події (events) у логах транзакцій.



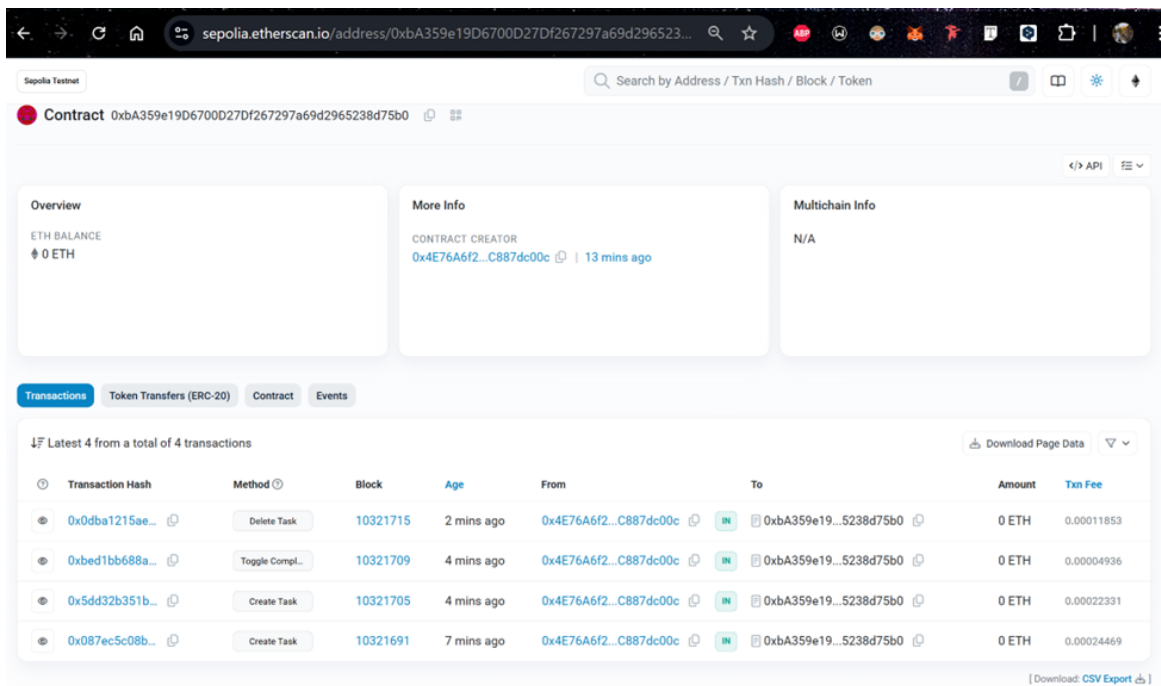


Рис. 15: Etherscan showing events

## 8 Висновки по тасці 2

В цьому завданні я створив смарт-контракт **ToDoList** з повною CRUD (Create, Read, Update, Delete) функціональністю. Контракт демонструє нам ключові можливості Solidity:

1. Designed and developed an original **ToDoList** smart contract implementing full CRUD functionality.
2. Demonstrated key Solidity features: `enum`, `struct`, nested mapping, custom modifier, events, and view functions.
3. Applied gas optimisation techniques including `external` visibility, `calldata` parameters, swap-and-pop deletion, and `unchecked` arithmetic.
4. Successfully deployed and tested the contract on the Sepolia testnet, verifying all operations via Remix IDE and Sepolia Etherscan.
5. Реалізували доволі оригінальний смарт-контракт **ToDoList**, що має повну функціональність CRUD.
6. Продемонстрував ключові функції Solidity: `enum`, `struct`, вкладений `mapping`, власний `modifier`, `events` та функції перегляду.
7. Трохи оптимізував витрати газу, використавши змінні `external`, параметри `calldata`, видалення `swap-and-pop` та арифметику `unchecked`.
8. Розгорнув цей контракт у тестовій мережі Sepolia, перевінив всі операції в середовищі Remix IDE та Sepolia Etherscan.

## 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](#)).