Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мережних технологій

# Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**Розробка емулятора машини Тюрінга з визначенням асимптотичної складності алгоритму**»

Виконав: студент 4-го року навчання,
Освітньої програми «Інженерія програмного забезпечення», 121
Добровольський Іван Олександрович

Керівник Франчук О.В._____
доцент, к.н.
Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2024

Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій

проф., д.ф.-м.н.

_____ Г. І. Малашонок

(підпис)

“____” _____ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Добровольському Івану Олександровичу 4-го курсу факультету інформатики

ТЕМА Розробка емулятора машини Тюрінга з визначенням асимптотичної складності алгоритму

Зміст ТЧ до кваліфікаційної роботи:

Вступ

Розділ 1. Визначення машини Тюрінга

Розділ 2. Аналіз предметної області

Розділ 3. Реалізація емулятора машини Тюрінга

Розділ 4. Способи визначення асимптотичної складності алгоритму

Висновки

Список використаних джерел

Дата видачі “___” _____ 2024 р. Франчук О.В. _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання роботи

**Тема:** Розробка емулятора машини Тюрінга
з визначенням асимптотичної складності алгоритму

| № п/п | Назва етапу | Термін виконання | Примітка |
|---|---|---|---|
| 1. | Отримання завдання на кваліфікаційну роботу | листопад 2023 р. | |
| 2. | Огляд літератури за темою роботи | листопад-грудень 2023 р. | |
| 3. | Розробка алгоритму для вирахування асимптотики | січень 2024 р. | |
| 4. | Розробка інтерфейсу | лютий-квітень 2024 р. | |
| 5. | Написання текстової частини курсової | квітень 2024 р. | |
| 6. | Надання роботи науковому керівнику для перевірки | травень 2024 р. | |
| 7. | Коригування виконаної роботи | травень 2024 р. | |
| 8. | Створення слайдів для доповіді та написання доповіді | травень 2024 р. | |
| 9. | Подання роботи на кафедру для перевірки на плагіат | травень 2024 р. | |
| 10. | Захист кваліфікаційної роботи | травень 2024 р. | |

Студент Добровольський І.О.

Керівник Франчук О.В.

“_____”_____

ЗМІСТ

## Анотація

Дана робота присвячена розробці емулятора машини Тюрінга, який забезпечує автоматичне визначення асимптотичної складності алгоритмів.

У результаті було створено програмний продукт, який дозволяє створювати та аналізувати різні алгоритми, представляючи їх у вигляді програм для машин Тюрінга.

Ключові слова: машина Тюрінга, асимптотика, Kotlin, Compose

# Вступ

Сучасний розвиток комп'ютерних наук базується на фундаментальних теоріях та моделях обчислень, однією з яких є модель Тюрінга. Машина Тюрінга була запропонована британським математиком Аланом Тюрінгом у 1936 році. Вона дозволяє досліджувати принципи побудови та виконання алгоритмів, надаючи універсальний інструмент для моделювання будь-яких обчислювальних процесів.

Ця робота присвячена розробці емулятора машини Тюрінга, що дозволяє візуалізувати процес виконання алгоритмів на даній моделі. Головною метою є створення інструменту, який не лише відтворює функціонування машини Тюрінга, але й надає можливість визначення асимптотичної складності, оскільки це дозволяє оцінювати ефективність алгоритмів у контексті обробки різних обсягів даних.

Актуальність роботи полягає в необхідності надання студентам інструменту, що полегшує вивчення теоретичних основ програмування, зокрема теорії обчислень та алгоритмів. Крім того, такий емулятор може стати цінним допоміжним засобом для викладачів, науковців та програмістів, які прагнуть глибше зрозуміти принципи роботи алгоритмів та їх ефективність.

# Розділ 1. Машина Тюрінга

Машина Тюрінга — це абстрактний обчислювальний пристрій, який запропонував англійський математик Алан Тюрінг у 1936 році. Ця машина була створена для того, щоб формалізувати поняття алгоритму та механізмів обчислення, і це надало можливості розвиватись теорії обчислюваності та інформатики.

## 1.1. Формальне визначення машини Тюрінга

Однострічкова машина Тюрінга може бути формально визначена як

$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$, де

- $\Gamma$ – скінченна, непуста множина символів алфавіту стрічки
- $b \in \Gamma$ – символ пустоти, значення не заповненої клітинки стрічки
- $\Sigma \subseteq \Gamma \setminus \{b\}$ – множина вхідних символів
- $Q$ – скінченна, непуста множина станів машини
- $q_0 \in Q$ – початковий стан
- $F \subseteq Q$ – кінцеві стани, при потраплянні в які машина успішно завершила роботу
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ – функція переходу, де $L$ це зсув головки вліво, $R$ це зсув головки вправо, N – не переміщувати головку. Функція переходу визначає у який стан переходити з поточного, який символ записати на поточну клітинку стрічки, та як змістити головку.

Якщо, перефразувати, то машина Тюрінга складається з декількох основних компонентів:

- Стрічка: Це нескінченна послідовність клітинок, в кожній з яких може бути символ алфавіту. Стрічка це своєрідна пам'ять машини, де можуть бути записані та зчитані символи.
- Головка для зчитування-запису: Цей елемент, може переміщатися по стрічці вліво, вправо, або залишатися на своєму місці, а також зчитувати та записувати дані у клітинку, що знаходиться під головкою.
- Таблиця переходів: Це певний набір правил, що визначають дії машини в залежності від її поточного стану та символу, який зчитується головкою зі стрічки. На основі цих даних, з таблиці можна отримати новий стан, символ для запису на стрічку та куди рухати головку.

## 1.2. Принцип роботи машини Тюрінга

Машина Тюрінга циклічно виконує наступні дії:

- Головка зчитує символ на поточній клітинці стрічки.
- Згідно з таблицею переходів, машина визначає, в який стан їй потрібно перейти, який символ повинен бути записаний на поточну клітинку та в якому напрямку повинна рухатись головка.
- Після цього машина робить перехід у новий стан і переміщує головку або ж залишає її на тому самому місці.

Цей цикл повторюється, допоки машина не досягне кінцевого стану, який вказує на закінчення обчислення, або ж не зупиниться аварійно.

## 1.3. Можливості машини Тюрінга, та її вплив

Конструкція машини Тюрінга надає способи та можливості реалізовувати складні алгоритми використовуючи композиції простіших алгоритмів, наприклад, для певних алгоритмів X та Y виконати X, потім Y, або виконати X і, якщо результат задовольняє певну умову, виконати Y, інакше не виконувати Y. Довести реалізованість таких композицій можна незалежно від специфіки алгоритмів X та Y шляхом створення нових програм, що об'єднують їх. Такі принципи можна застосувати до будь яких композицій.

Тюрінг описав різноманітні алгоритми, використовуючи свою машину, та надав тезу, що будь-який алгоритм можна реалізувати за допомогою відповідної програми для машини Тюрінга. Використовуючи це твердження можна доводити існування або відсутність алгоритмів, створюючи відповідні програми або ж показуючи неможливість їх побудови, що дає загальний підхід до пошуку алгоритмічних рішень.

На жаль, теза Тюрінга не може бути повністю доведена через відсутність чіткого визначення будь-якого алгоритму, у його формулюванні. Її можна лише обґрунтовувати, представляючи відомі алгоритми у вигляді програм для машин Тюрінга.

Машина Тюрінга стала фундаментом для теоретичної інформатики. Вона допомогла формалізувати поняття алгоритму та обчислювальної функції, що дозволило створити основу для розвитку програмування та комп'ютерних наук. Також машина Тюрінга лягла в основу розробки сучасних комп'ютерів, незважаючи на те, що реальні комп'ютери мають набагато складнішу архітектуру та обмежену пам'ять.

# Розділ 2. Аналіз предметної області

Темою цієї роботи є створення програми для емуляції та аналізу програм для машин Тюрінга. Основною цільовою аудиторією для застосунку є студенти що вивчають обчислювальні машини та алгоритми пов’язані з ними, тобто студенти технічних спеціальностей.

Наразі, існує чимало програм та сайтів, що можуть бути корисними для даної цільової аудиторії, тому перед тим як створювати додаток потрібно проаналізувати вже існуючі рішення, спираючись на такі характеристики

- Зручний дизайн
- Наявність сполучень клавіш
- Автодоповнення тексту
- Вирахування асимптотики алгоритму
- Експорт та імпорт даних у читабельному форматі

Для порівняння вибирались сайти та програми враховуючи їх популярність та функціональні можливості.

Наявні рішення

- https://turingmachinesimulator.com/
- https://turingmachine.io/
- https://morphett.info/turing/turing.html
- http://turingmachine.vassar.edu/
- https://github.com/foggynight/turing-machine
- https://schaetzc.github.io/tursi/

| | Зручний дизайн | Наявність шорткатів | Автодоповнення | Вирахування асимптотики | Експорт/Імпотрт даних | Граф переходів |
|---|---|---|---|---|---|---|
| Мій застосунок | ☑ | ☑ | ☑ | ☑ | ☑ | ☐ |
| TuringMachineSimulator | ☑ | ☐ | ☐ | ☐ | ☑ | ☑ |
| TuringMachine.io | ☑ | ☑ | ☐ | ☐ | ☑ | ☐ |
| Morphett | ☐ | ☐ | ☐ | ☐ | ☑ | ☐ |
| TuringMachine.vassar | ☑ | ☐ | ☑ | ☐ | ☑ | ☐ |
| FoggyNight Turing Machine | ☐ | ☑ | ☐ | ☐ | ☑ | ☐ |
| Tursi | ☑ | ☑ | ☐ | ☐ | ☑ | ☐ |

*Рисунок 2.1*

- Turing Machine Simulator

  Цей сайт має базовий і функціональний дизайн, але відсутність гарячих клавіш і автодоповнення роблять його менш привабливим.
- TuringMachine.io

  TuringMachine.io має гарний та зрозумілий дизайн і наявні сполучення клавіш. Також має граф переходів і гарну візуалізацію, але немає автодоповнення.
- Morphett Turing Machine

  Morphett Turing Machine має доволі застарілий інтерфейс, і відсутні сполучення клавіш та автодоповнення.
- Vassar College Turing Machine

  Веб-додаток від Vassar College має базовий дизайн, немає гарячих клавіш, але на відміну від інших, тут є автодоповнення.

- FoggyNight Turing Machine

  Даний застосунок має не дуже зручний та немає автодоповнення, але має гарячі клавіші.
- Tursi

  Tursi має зручний, але застарілий інтерфейс, немає автодоповнення, але в цьому додатку можна використовувати сполучення клавіш.

Аналізуючи актуальні рішення, можна побачити, що автодоповнення та гарячі клавіші присутні лише в деяких сайтах, а розрахунку асимптотики, немає в жодному, тому саме на цьому потрібно сфокусуватись при розробці програми.

# Розділ 3. Реалізація емулятора машини Тюрінга

Дана програма розроблялась для десктопних платформ, з подальшою можливістю масштабуватись для браузерів та мобільних платформ.

## 3.1. Засоби розробки

IntelliJ IDEA — це інтегроване середовище розробки (IDE), від компанії JetBrains, що підтримує численні мови програмування, зокрема Java та Kotlin. Це найпопулярніше рішення для розробки на Java та інших JVM-базованих мовах. IntelliJ IDEA надає широкий набір інструментів та функцій для полегшення розробки, включаючи розширене автодоповнення коду, рефакторинг та інтеграцію з системами контролю версій, наприклад Git.

Kotlin — це сучасна, статично типізована мова програмування, яка була представлена у 2011 році. Її основні цілі полягають у підвищенні продуктивності розробників, забезпеченні надійності програмного коду та інтеграції з існуючими системами. Kotlin є повністю сумісною з Java, що дозволяє безперешкодно використовувати існуючі Java бібліотеки в проєктах написаних мовою Kotlin.

Особливості Kotlin

- Сумісність з Java
- Лаконічний код
- Безпека типів
- Наявність функціональних парадигм

Jetpack Compose — це сучасний інструментарій для розробки UI на різних платформах, створений компанією Google. Він пропонує декларативний підхід до побудови UI, що значно спрощує і прискорює процес розробки.

Ключові особливості Jetpack Compose

- Декларативний підхід
- Короткий, читабельний код
- Реактивність
- Широкі можливості кастомізації

Kotlin Serialization — бібліотека для серіалізації та десеріалізації даних, від компанії JetBrains. Ця бібліотека є частиною екосистеми Kotlin і забезпечує простий та ефективний спосіб роботи з даними.

Koin — це легка у використанні бібліотека для ін'єкції залежностей у мові програмування Kotlin. Вона була створена спеціально для цієї мови та надає декларативний стиль налаштування залежностей, що робить її ідеальним вибором для розробників, малих та середніх проєктів.

### 3.2. Ін’єкція залежностей

Основним патерном при розробці став Dependency Injection (Ін’єкція залежностей) що передбачає надання зовнішньої залежності програмному компоненту, використовуючи Inversion of control (Інверсія керування) для розв'язання (отримання) залежностей.

Додаток поділений на логічні модулі в яких прописується які залежності куди потрібно надати та спосіб їх створення (наприклад валідація, мапінг, таблиця переходів, емуляція алгоритму)

Також з допомогою DI додалось ще одне поняття – Scope (зона видимості), де певні залежності мають життєвий цикл, та можуть бути надані лише якщо отримувач знаходиться всередині цієї зони видимості.

Приклади зон видимості

- Емуляція машини
- Збереження програми
- Рядок таблиці
- Клітинка таблиці
- та інші

### 3.3. UI-Архітектура

Оскільки основою застосунку є фреймворк Compose Multiplatform, то для зв’язку візуального відображення на екрані та логікою програми використовується архітектурний патерн притаманний цьому фреймворку – Model-View-ViewModel (MVVM).

Для кожного візуального компоненту створюється відповідний клас ViewModel у якому відбувається отримання даних з певного джерела (UseCase, Repository чи Storage), а також обробка користувацьких дій – таких як натискання на кнопку, введення тексту, зміна фокусу компоненту чи виконання сполучення клавіш.

Для більш ефективної обробки користувацьких дій був використаний підвид патерну Observer(Наглядач), а саме EventChannel, який стає єдиним джерелом інформації для дій ззовні. Так, кожна дії повинна описуватись окремим класом, мати інформацію яка притаманна лише для цієї дії та наслідувати інтерфейс Event. Під час виконання наприклад кліку, потрібно викликати не ViewModel напряму, а відправляти дію в EventChannel. В той час, при ініціалізації, ViewModel буде підписуватись та отримувати події від EventChannel, після чого викликати відповідні до подій функції.

Ця концепція дозволяє більш гнучко та елегантно описувати користувацькі дії, а також з легкістю масштабуватись.

### 3.4. Валідація даних

Чималою частиною програми стала валідація вхідних даних, оскільки є багато полів для вводу специфічного тексту: стани програми, символи стрічки, символ руху каретки та інші. В додачу до цього, оскільки застосунок підтримує експорт та імпорт користувацьких програм - ці дані також потрібно валідувати.

Для покращення архітектури та перевикористання класів було додано інтерфейс Validator та абстрактний клас StringValidator, які мають єдину функцію validate() та повертають Result.

### 3.5. Обробка помилок

Для обробки помилок програма має два варіанти, локальні – обробка даних з поля введення, і показ помилки біля самого поля, та глобальні – всі інші помилки для яких показується повідомлення у правому кутку екрану.

Локальні помилки обробляються всередині ViewModel, та додаються як параметри стану компоненту.

Для того щоб показати помилку у вигляді повідомлення потрібно запровадити залежність ErrorChannel та викликати метод помилки у цього класу. В той же час в самому корені програми є компонент, що отримує дані від ErrorChannel про помилки які існують та показує їх на екрані.

За допомогою такого підходу не потрібно створювати спосіб обробки помилок для кожного екрану копіюючи/вставляючи код, а можна просто відсилати все у один глобальний обробник.

### 3.6. Навігація

Не менш важливою складовою є навігація. Для її імплементації використовується метод схожий до обробки помилок. Існує NavigationChannel, який потрібно додавати як залежність до класу, та використовувати його для того щоб змінити екран.

У корені програми так само існує компонент що буде отримувати запити на навігацію та перенаправляти користувача.

Ці підходи слідують першому правилу SOLID, Single Responsibility (принцип єдиного обов’язку), що дозволяє розділити обов’язки між класами і не перевантажувати ViewModel.

### 3.7. Автодоповнення

Однією з переваг та особливостей застосунку є наявність автодоповнення у певних полях вводу. Це дозволяє користувачу швидко та ефективно вводити інформацію.

Для того щоб ефективно обробляти запити автодоповнення для великої кількості даних, використовується структура даних – бор.

За допомогою цієї структури можна швидко шукати всі значення колекції рядків що мають спільний префікс.

Автодоповнення мають такі поля як, введення стану машини або ж введення символу який головка машини повинна зчитати або ж записати.

### 3.8. Серіалізація даних.

Для збереження та зчитування даних з файлу використовується формат JSON. Майже всі мови мають бібліотеки які вміють працювати з цим форматом, Kotlin не виключення.

Проблема цього формату що для великих структур, виходить дуже об’ємний файл який складно читати та редагувати.

Оскільки при розробці програми мета була зробити не лише зручний інтерфейс, але й зрозумілий та читабельний формат файлових даних то для більшості структур що зберігаються були написані власні серіалізатори да десеріалізатори.

Так замість виводу таблиці стану переходів у такому форматі


*Рисунок 3.1*

Став ось такий формат.

```
"stateTable": [
    {
      "readState": "q0, 0",
      "writeState": "q1, 1, <"
    },
    {
      "readState": "q1, 1",
      "writeState": "q0, 0, _"
    }
  ]
```

*Рисунок 3.2*

Найбільших змін зазнав формат алфавіту машини. Оскільки застосунок має певні стандартні формати алфавітів, такі як бінарний, десятковий, шістнадцятковий або ж латиниця, то для покращення читабельності файлу вивід перетворився з такого

```
"alphabet": {
  "alphabet": [
    "a","b","c","d","e","f","g","h","i","j","k","l","m","n",
    "o","p","q","r","s","t","u","v","w","x","y","z"
  ],
  "undefined": "_"
}
```

*Рисунок 3.3*

На ось такий.

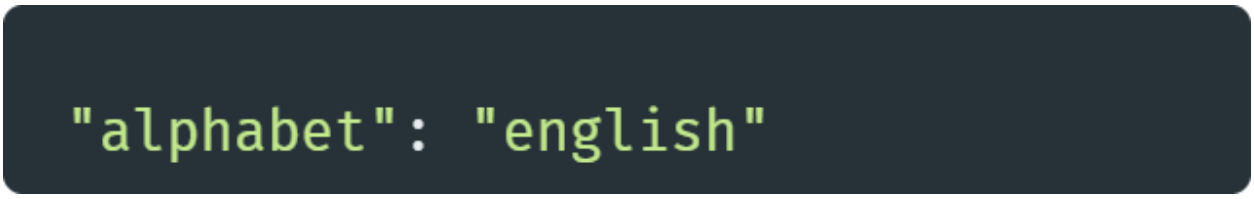


*Рисунок 3.4*


*Рисунок 3.5*

### 3.9. Збереження даних

Оскільки найбільші за обсягом дані - користувацькі програми зберігаються у файлах, то для всіх інших даних замість використання бази даних використовується Kotlin DataStore

Як приклад можна взяти збереження нещодавно відкритих програм на стартовому екрані. Дані тут зберігаються як просто рядок і записуються за допомогою Kotlin DataStore у файли програми на диску

Перевагою використання цієї бібліотеки є те, що всі зміни даних - реактивні, і тому не потрібно викликати безліч різних функцій щоб отримувати актуальну інформацію.

# Розділ 4. Обробка програм для розрахунку асимптотик

## 4.1. Генерація даних

Для того щоб приблизно розрахувати асимптотику потрібен чималий набір даних, оскільки користувачу буде незручно придумувати та вводити велику кількість вхідних стрічок, і цей об’єм все одно буде доволі малий, тому дані для розрахунку асимптотик генеруються автоматично.

Оскільки згенерувати дані під кожну програму неможливо, тому було зроблено два основні підвиди генерацій.

1. Повністю рандомізований – дані генеруються на основі заданої довжини стрічки, та кожен символ в цій стрічці вибирається випадково з алфавіту.
2. Edge-case – стрічки, що складаються лише з одного символу алфавіту, або мають чергування символів, або ж йдуть певними блоками.

## 4.2. Емуляція машин

Оскільки вхідних даних буде чимало (тисячі), то для того щоб швидше отримувати результаті всі обчислення проводяться паралельно. Для вирішення цієї задачі використовується бібліотека Kotlin Coroutines.

Для паралельного виконання програм створений окремий клас який має спеціальну suspend функцію що проводить емуляцію машини, особливістю цих функцій є те що вони не створюють окремий Java Thread, а виконуються на певному ThreadPool, зупиняючись на функціях очікування, цим самим даючи можливість іншим suspend функціям виконувати роботу.

### 4.3. Розрахунок асимптотики

Маючи набір пар даних, «довжина стрічки» – «максимальна кількість кроків», можемо представити їх у вигляді точок на координатній площині, та знайти функцію яка найточніше описує даний набір.

В застосунку використовується обмежений набір функцій, які можуть бути відповіддю. Наприклад: $O(1),\ O(n),\ O(n\ log\ n),\ O(n^2)$, тощо.

Алгоритм моєї програми, шукає константу $C$, для функції $f(x)$ та певного набору точок $(x_i, y_i)$ розміру $n$ для якої

$$\sum_{i=1}^{n} |C \cdot f(x_i) - y_i|$$

є мінімальною.

$$f(x) = \begin{cases} const \\ log\ x \\ \sqrt{x} \\ x \\ xlog\ x \\ x\sqrt{x} \\ x^2 \\ x^3 \\ 2^x \end{cases}$$

$$x_i\ -\ \text{довжина стрічки}$$

$$y_i - \text{максимальна кількість кроків}$$

Ця константа $C$ шукається для кожної $f(x)$, і з них вибирається та, для якої

$$\sum_{i=1}^{n} |C \cdot f(x_i) - y_i|$$

є мінімальною.

## 4.4. Приклади розрахунку асимптотики

Для кращого розуміння, як працює даний алгоритм, розглянемо елементарні та більш складні приклади.

Візьмемо елементарну програму, що зміщує головку перед початком стрічки.

Оскільки головка стоїть на першому символі стрічки, то дана програма завжди виконує 2 кроки – перейти вліво, завершити виконання.

Опис

$$Q = \{q_0, q_{finish}\}$$

$$\Gamma = \{0,1,\_\}$$

$$\mathrm{b} = \_$$

$$\Sigma = \{0,1\}$$

$$q_0 = q_0$$

$$\mathrm{F} = \{q_{finish}\}$$

Функцію переходу $\delta$ можна представити у вигляді таблиці

| Read State | Read Symbol | Write State | Write Symbol | Move |
|---|---|---|---|---|
| q0 | 0 | q0 | 0 | < |
| q0 | 1 | q0 | 1 | < |
| q0 | _ | qfinish | _ | _ |

*Рисунок 4.1*

## Приклад вхідних даних

| | Стрічка | x - Довжина стрічки | y - Кількість кроків | $\lvert C \cdot 1 - y_i \rvert$ | $\lvert C \cdot log\, x_i - y_i \rvert$ | $\lvert C \cdot x_i - y_i \rvert$ |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 0.00 | 2.00 | 1.78 |
| | 01 | 2 | 2 | 0.00 | 1.37 | 1.55 |
| | 001 | 3 | 2 | 0.00 | 1.00 | 1.33 |
| | 1010 | 4 | 2 | 0.00 | 0.74 | 1.11 |
| | 101000101 | 9 | 2 | 0.00 | 0.00 | 0.00 |
| | 10101000001 | 11 | 2 | 0.00 | 0.18 | 0.45 |
| $\sum_{i=1}^{n} \lvert C \cdot f(x_i) - y_i \rvert$ | | | | 0.00 | 5.29 | 6.23 |
| $C$ | | | | 2.0000 | 0.9111 | 0.2227 |

*Рисунок 4.2*

## Графік складності алгоритму - $O(1)$

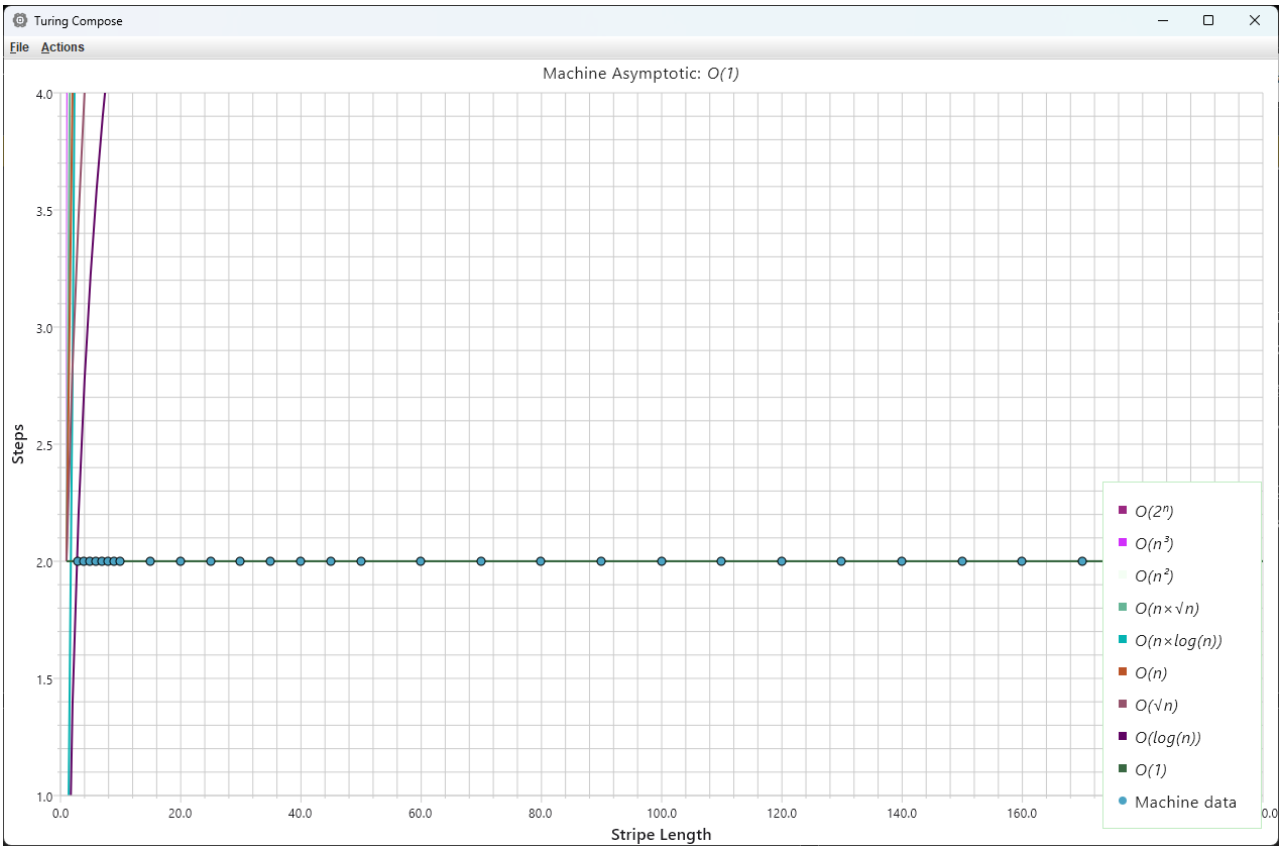


*Рисунок 4.3*

Якщо розглянути іншу елементарну програму, що зміщує каретку за кінець стрічки, то дана програма буде працювати за лінійну складність, оскільки потрібно буде змістити головку на кількість символів у стрічці та завершити програму.

Опис

$$Q = \{q_0, q_{finish}\}$$

$$\Gamma = \{0,1,\_\}$$

$$\mathrm{b} = \_$$

$$\Sigma = \{0,1\}$$

$$q_0 = q_0$$

$$\mathrm{F} = \{q_{finish}\}$$

Функцію переходу $\delta$ можна представити у вигляді таблиці

| Read State | Read Symbol | Write State | Write Symbol | Move |
|---|---|---|---|---|
| q0 | 0 | q0 | 0 | > |
| q0 | 1 | q0 | 1 | > |
| q0 | _ | qfinish | _ | _ |

*Рисунок 4.4*

## Приклад вхідних даних

| | Стрічка | x - Довжина стрічки | y - Кількість кроків | $\lvert C \cdot 1 - y_i \rvert$ | $\lvert C \cdot x_i - y_i \rvert$ | $\lvert C \cdot x_i \cdot \sqrt{x_i} - y_i \rvert$ |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3.00 | 0.89 | 1.63 |
| | 01 | 2 | 3 | 2.00 | 0.78 | 1.95 |
| | 001 | 3 | 4 | 1.00 | 0.67 | 2.07 |
| | 1010 | 4 | 5 | 0.00 | 0.55 | 2.04 |
| | 101000101 | 9 | 10 | 5.00 | 0.00 | 0.01 |
| | 10101000001 | 11 | 12 | 7.00 | 0.23 | 1.52 |
| $\sum_{i=1}^{n} \lvert C \cdot f(x_i) - y_i \rvert$ | | | | 18.00 | 3.12 | 9.22 |
| $C$ | | | | 5.0002 | 1.1116 | 0.3706 |

*Рисунок 4.5*

## Графік складності алгоритму - $O(n)$

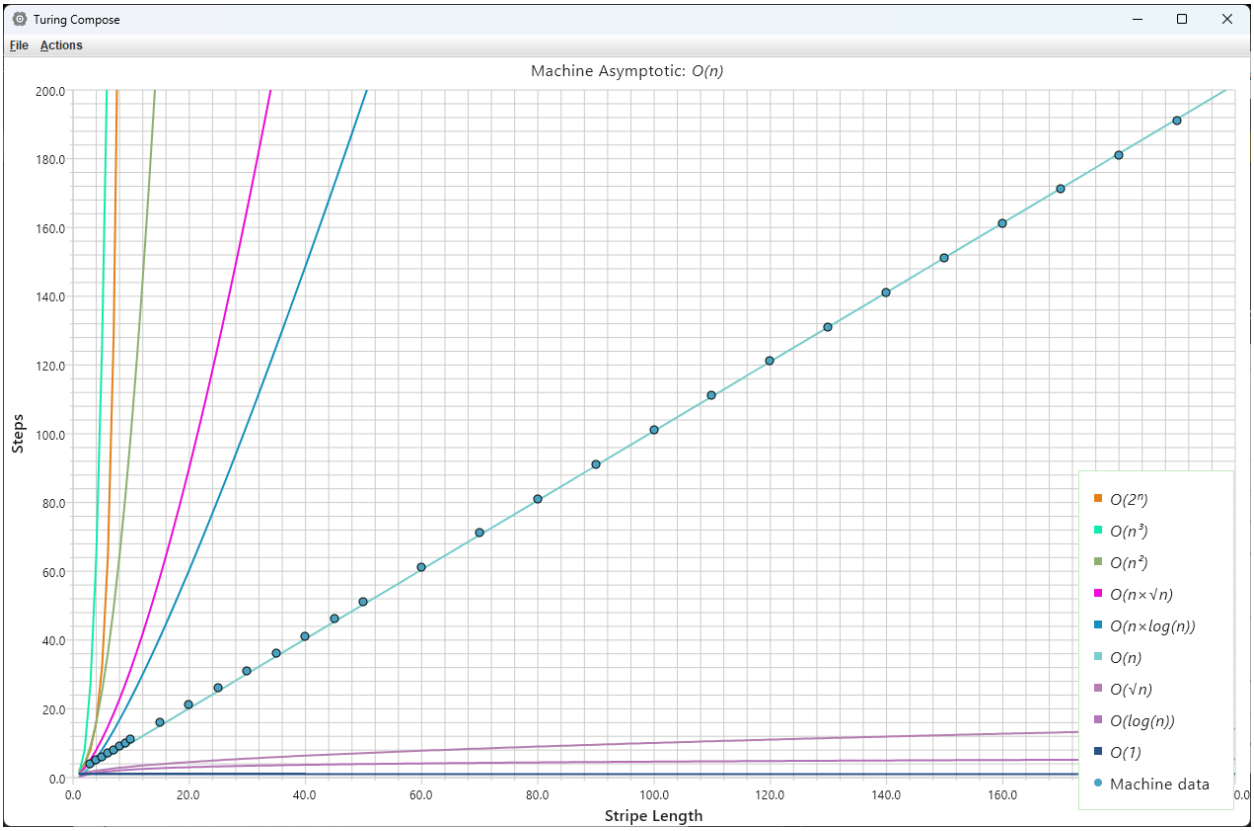


*Рисунок 4.6*

Розглянемо складнішу програму яка перевіряє чи належить вхідне слово $0^n1^n2^n$.

Опис

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$\Gamma = \{0,1,2, a, b, c, \_\}$$

$$\mathrm{b} = \_$$

$$\Sigma = \{0,1,2\}$$

$$q_0 = q_0$$

$$\mathrm{F} = \{q_5\}$$

| Read State | Read Symbol | Write State | Write Symbol | Move |
|---|---|---|---|---|
| q0 | 0 | q1 | a | > |
| q0 | b | q4 | b | > |
| q1 | 1 | q2 | b | > |
| q1 | 0 | q1 | 0 | > |
| q1 | b | q1 | b | > |
| q2 | 2 | q3 | c | < |
| q2 | 1 | q2 | 1 | > |
| q2 | c | q2 | c | > |
| q3 | 1 | q3 | 1 | < |
| q3 | 0 | q3 | 0 | < |
| q3 | b | q3 | b | < |
| q3 | c | q3 | c | < |
| q3 | a | q0 | a | > |
| q4 | b | q4 | b | > |
| q4 | c | q4 | c | > |
| q4 | _ | q5 | _ | _ |

*Рисунок 4.7*

Дана програма працює за квадратичну складність, оскільки для кожного з 0 потрібно знайти 1 справа, потім 2, і потім повернутись назад до наступного 0.

Приклад вхідних даних

| | Стрічка | x - Довжина стрічки | y - Кількість кроків | $\lvert C \cdot x_i \cdot \sqrt{x_i} - y_i \rvert$ | $\lvert C \cdot x_i^2 - y_i \rvert$ | $\lvert C \cdot x_i^3 - y_i \rvert$ |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 0.62 | 1.52 | 1.98 |
| | 012 | 3 | 8 | 5.63 | 3.68 | 7.55 |
| | 001122 | 6 | 23 | 15.55 | 5.74 | 19.41 |
| | 000111222 | 9 | 46 | 24.83 | 7.16 | 33.90 |
| | 000001111122222 | 15 | 116 | 36.39 | 8.12 | 59.97 |
| | 000000000011111111112222222222 | 30 | 431 | 0.03 | 0.54 | 17.28 |
| $\sum_{i=1}^{n} \lvert C \cdot f(x_i) - y_i \rvert$ | | | | 83.02 | 26.22 | 122.81 |
| $C$ | | | | 2.6232 | 0.4795 | 0.0166 |

*Рисунок 4.8*

Графік складності алгоритму - $O(n^2)$

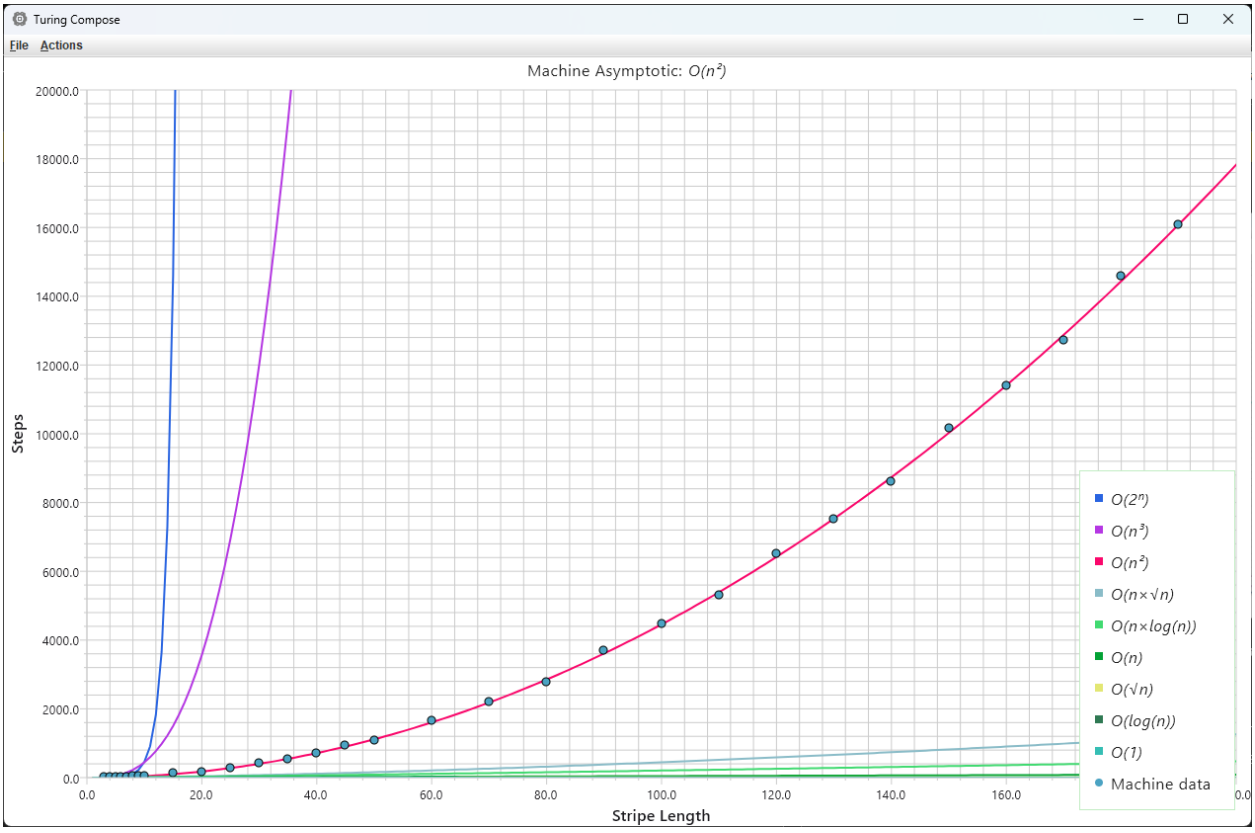


*Рисунок 4.9*

# Розділ 5. Екрани програми

## 5.1. Початковий екран

Початковий екран призначений для швидкого доступу до файлів.

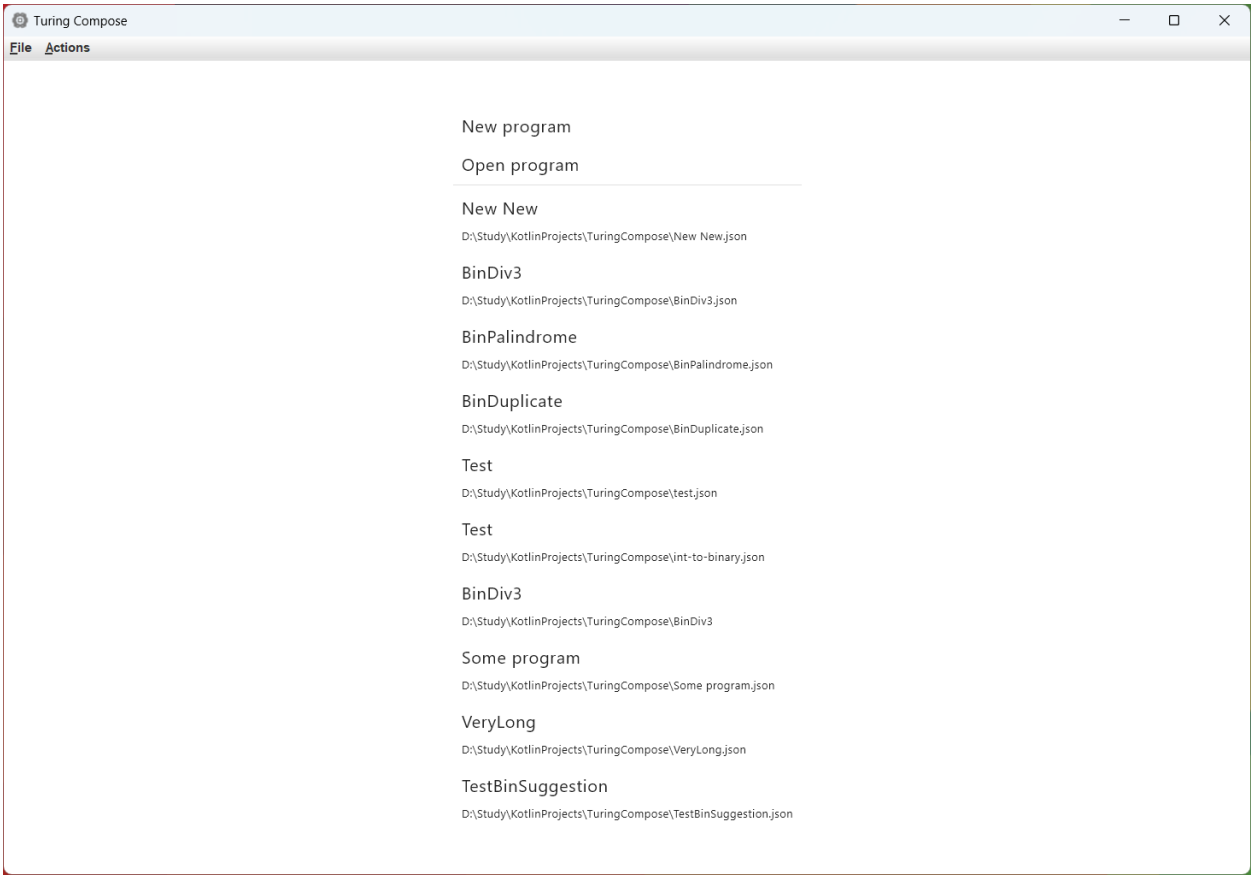


*Рисунок 5.1*

Першою функцією є створення нової програми для машини, що перенаправить користувача на пустий екран створення програми.

Другим є відкриття програми з файлу, що покаже діалог вибору файлів. При відкритті коректної програми користувач перейде на основний екран, де в таблиці переходів буде записана вибрана програма.

При відкритті некоректного файлу користувачу виводиться помилка у правому нижньому кутку, та залишить його на цьому ж екрані.

Основним на цьому екрані є список нещодавно відкритих програм, відсортованих у порядку користування (від нещодавньої, до тої якою користувались найдавніше). Поведінка аналогічна до відкриття програми з файлу. Кожен компонент цього списку містить ім’я програми та шлях до неї.

## 5.2. Екран створення та редагування програми

На цьому екрані відображається таблиця переходів, з можливістю додати опис до кожного з них.

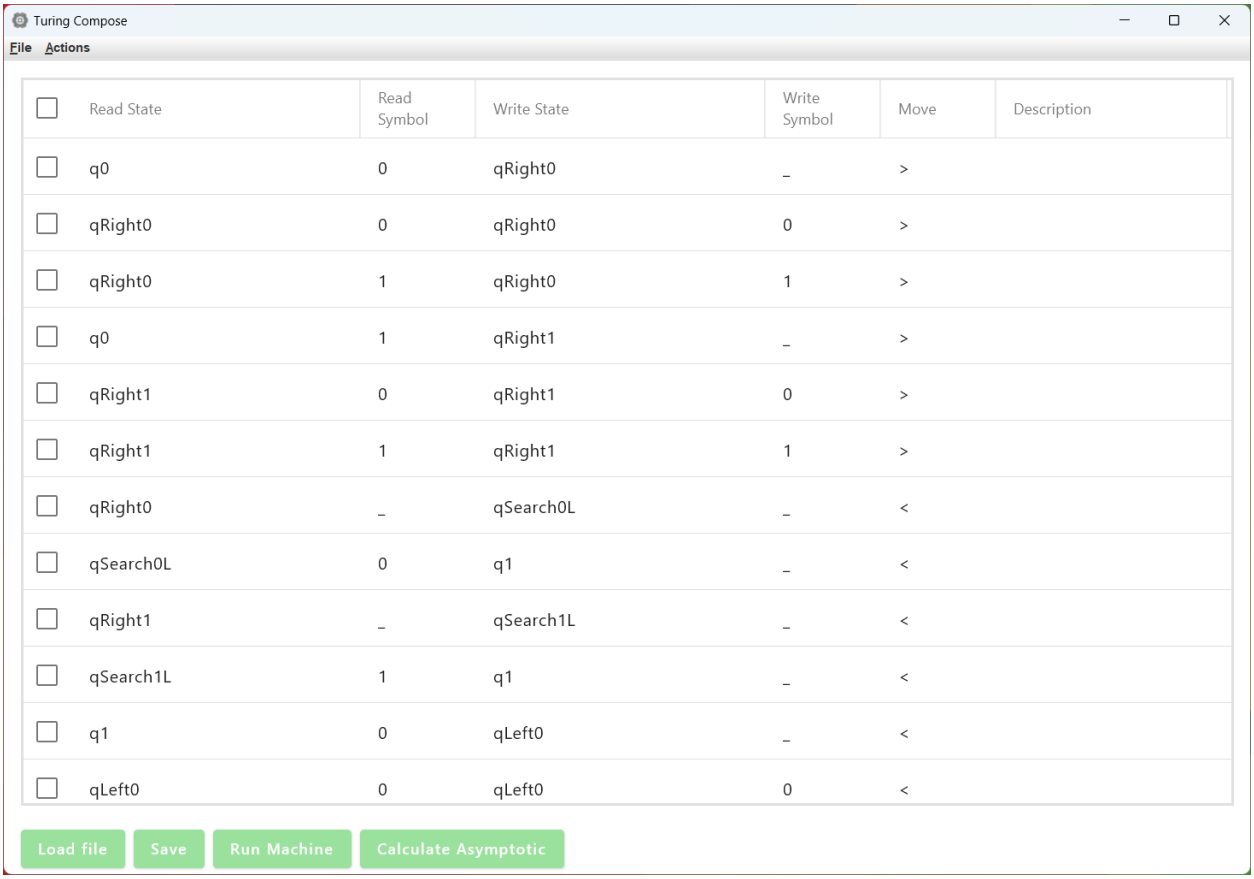


*Рисунок 5.2*

Кожна клітинка рядку має автодоповнення для введеного стану, символу, чи руху головки.

При введенні коректних даних у останній рядок, автоматично створюється новий рядок під ним, для зручної та ефективної роботи.

Таблиця підтримує створення та видалення рядків.

Також використовуючи чек бокси зліва можна вибирати декілька станів переходів та при натисканні правої кнопки миші з’являється контекстне меню з можливістю копіювати чи видалити вибрані рядки.

## 5.3. Екран збереження програми

Цей екран призначений для введення та підтвердження інформації про програму для машини перед збереженням її у файл.

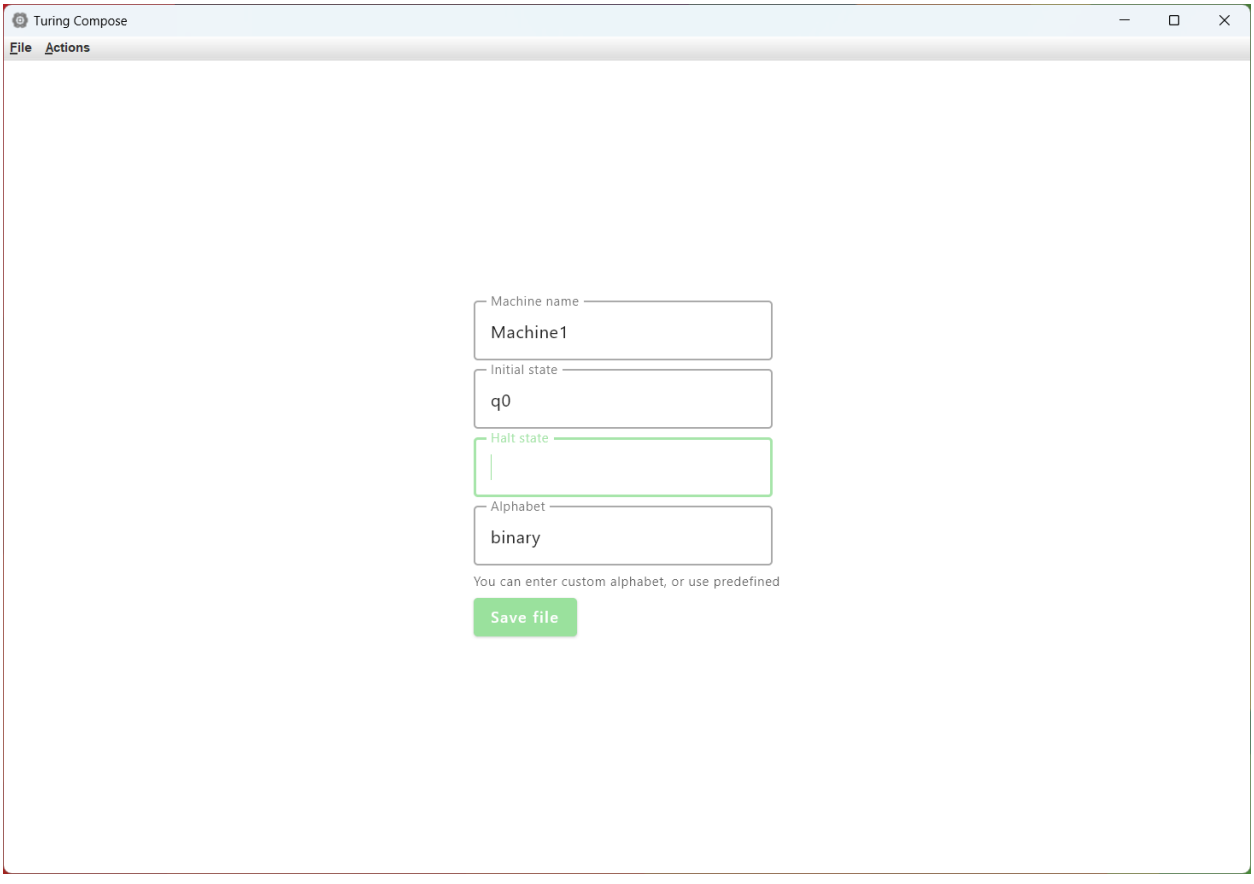


*Рисунок 5.3*

Першим є поле для введення назви програми, за яким слідують поля введення початкового та кінцевого стану. При початку вводу тексту в ці поля автопідбереться один із можливих станів.

Останнім є поле введення алфавіту, яке буде заповнене стандартним алфавітом у випадку якщо, всі символи таблиці переходу підходять під нього, або ж просто цими символами, якщо жоден стандартний алфавіт не підходить. При введені тексту підбір алфавіту буде базуватись не лише на назві стандартного алфавіту а й на символах з якого він складається

Так при введені 10 – будуть запропоновані такі варіанти

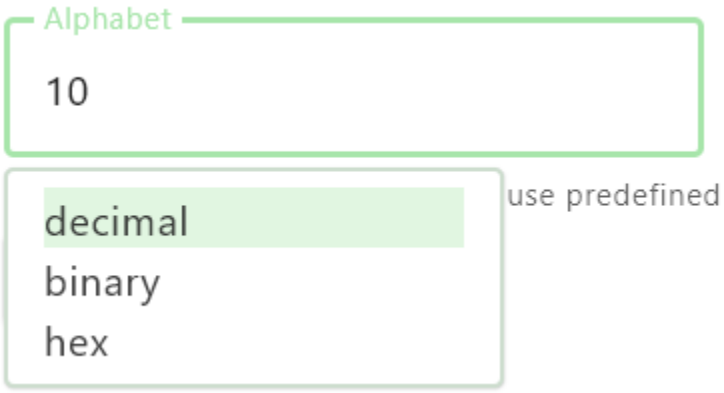


*Рисунок 5.4*

А при введенні 012345678 – будуть запропоновані ось такі

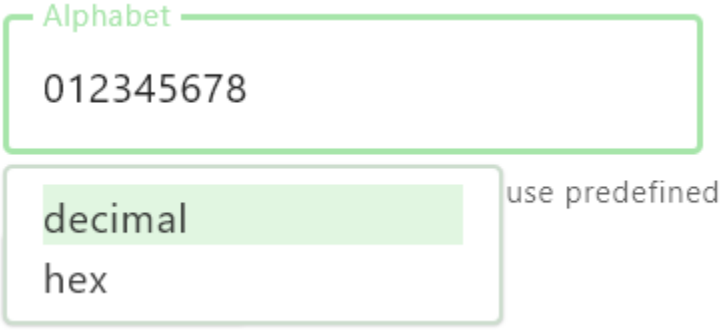


*Рисунок 5.5*

При натисканні збереження відкриється діалог з вибором файлу та програма запишеться у файл, а користувач повернеться на екран роботи з програмою

## 5.4. Екран підготовки до емуляції

На цьому екрані користувач повинен ввести вхідний стан стрічки, вибрати максимальну кількість кроків та опціонально додати вхідний та вихідний тип даних.

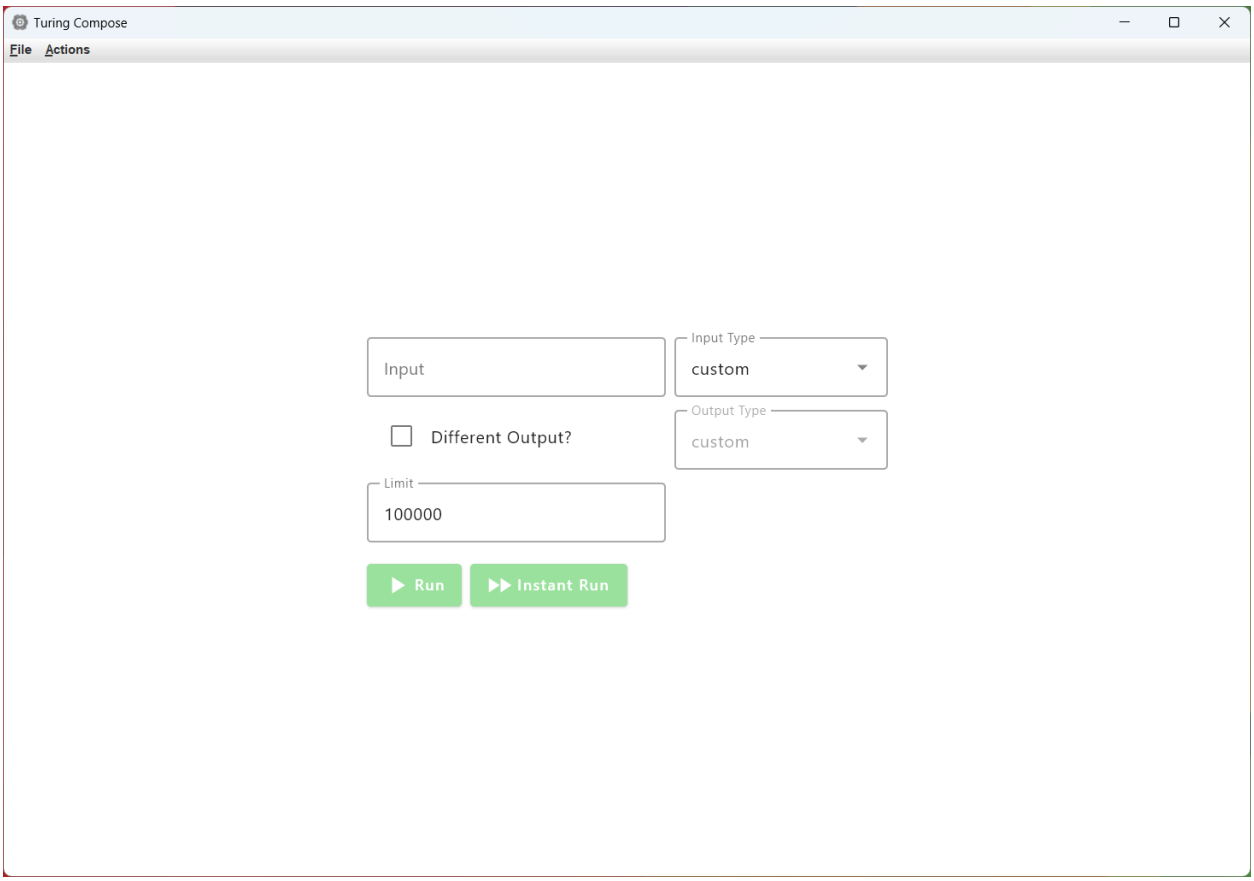


*Рисунок 5.6*

Перший елемент екрану це поле введення вхідного стану стрічки, користувач повинен заповнити її лише символами з алфавіту.

Якщо ж програма використовує числові типи (бінарний, десятковий, шістнадцятковий), то користувач може ввести їх у десятковій системі числення попередньо вибравши цю опцію у випадному списку.

Також на даному екрані є поле вводу максимальної кількості кроків які машина може виконати до аварійного завершення.

Останніми є кнопки запуску. Перша кнопка переведе користувача на екран емуляції, а друга одразу на екран результату.

## 5.5. Екран емуляції роботи машини

На цьому екрані користувач бачить стрічку, головку та таблицю переходів.

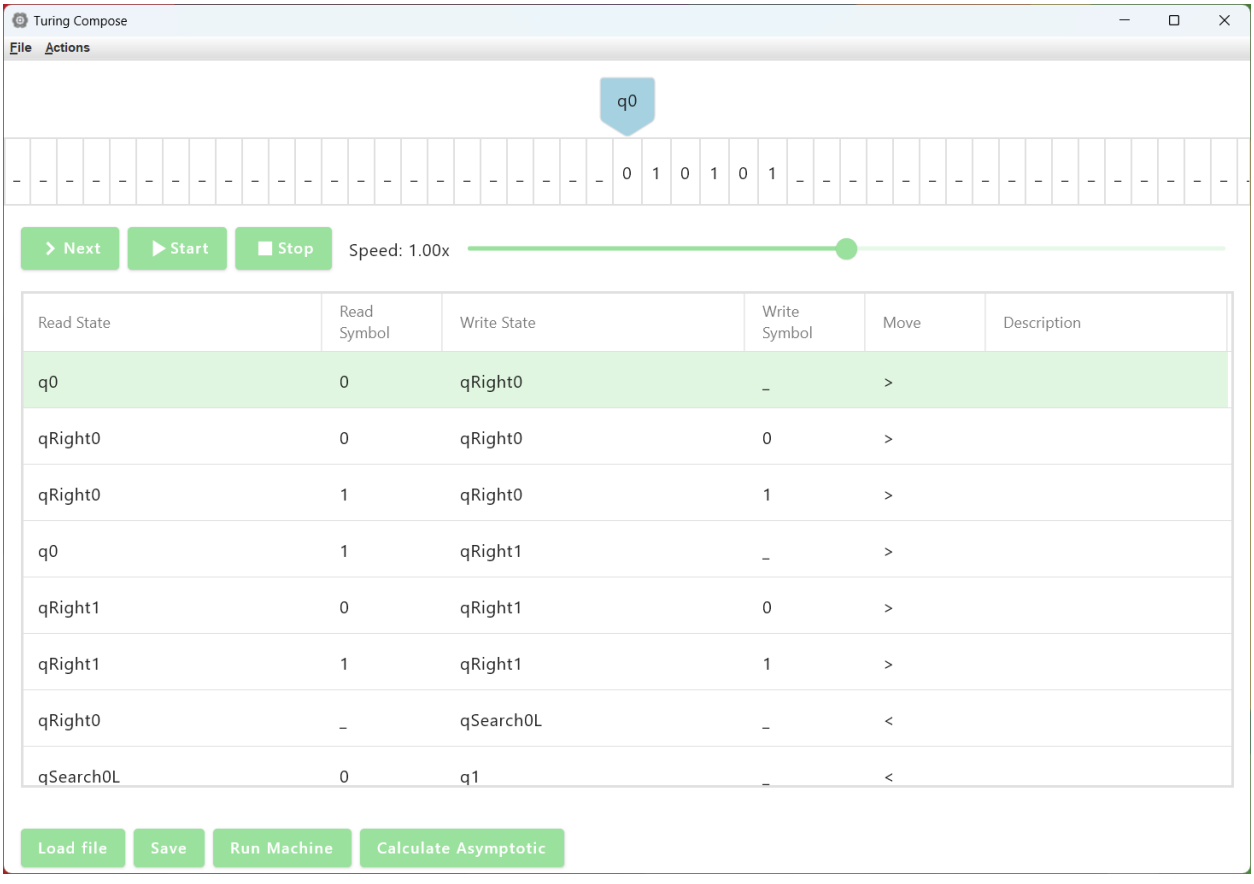


*Рисунок 5.7*

Компонент стрічки відображає актуальний стан машини, та анімується при його зміні, головка при цьому змінює текст стану.

Емуляцію можна контролювати за допомогою кнопок

- Крок – запустить обрахування наступного кроку, та змінить стан стрічки.
- Запуск – почне послідовно виконувати кроки один за одним.
- Стоп – зупинить виконання кроків.
- Повзунок швидкості – змінює швидкість анімації переходу з одного стану в інший.

Найбільшим елементом екрану є таблиця переходів, яка аналогічна тій що була на екрані редагування, але тут вона лише в форматі читання.

Під час емуляції поточний стан машини буде підсвічуватись в таблиці переходів, та при зміні станів ця таблиця буде прокручуватись до поточного.

Після завершення емуляції, користувача перенаправить на екран результату.

## 5.6. Екран результату

Цей екран показує результат роботи машини, кількість кроків, вихідний стан стрічки та кроки що були виконані.

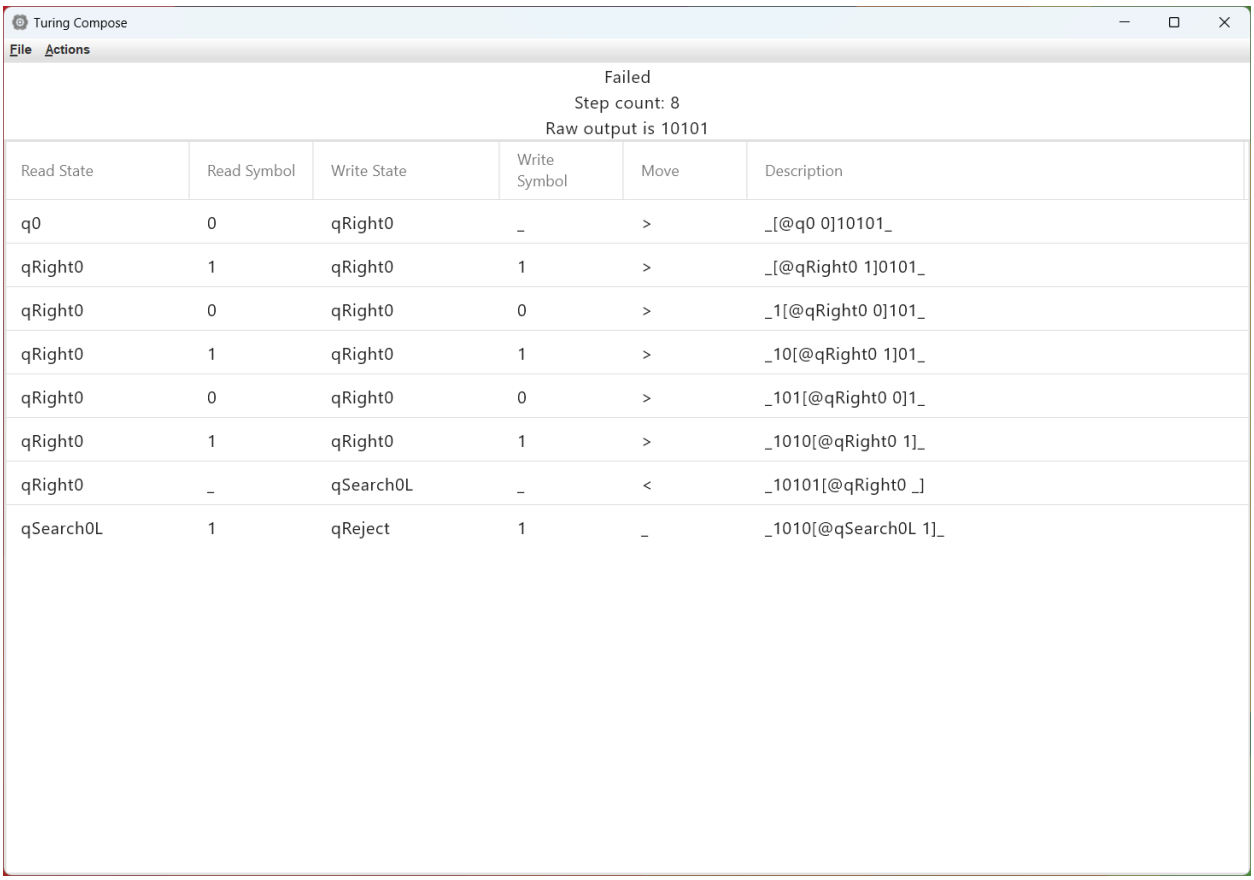


*Рисунок 5.8*

Першим рядком, є результат роботи машини

- “Success” якщо машина виконала роботу без помилок.
- “Failed” якщо машина завершила роботу аварійно.

У другому рядку показується кількість кроків, які машина виконала під час виконання програми.

Третій рядок містить у собі стан стрічки, у якому машина завершила виконання.

Четвертий рядок з’являється у тому випадку якщо користувач вказав вихідний тип даних на екрані підготовки до емуляції, то показує конвертований стан стрічки у читабельний формат.

Останнім елементом екрану є таблиця з історією кроків які зробив емулятор під час виконання програми.

## 5.7. Екран розрахунку асимптотики

Цей екран відображає асимптотику роботи алгоритму.

Екран має дві фази – підрахунок та відображення.

Під час першої фази на екрані можна бачити які дані зараз емулює машина та кількість кроків за який програма була виконана.


*Рисунок 5.9*

Під час другої фази на екрані можна побачити графік асимптотичної складності алгоритму.

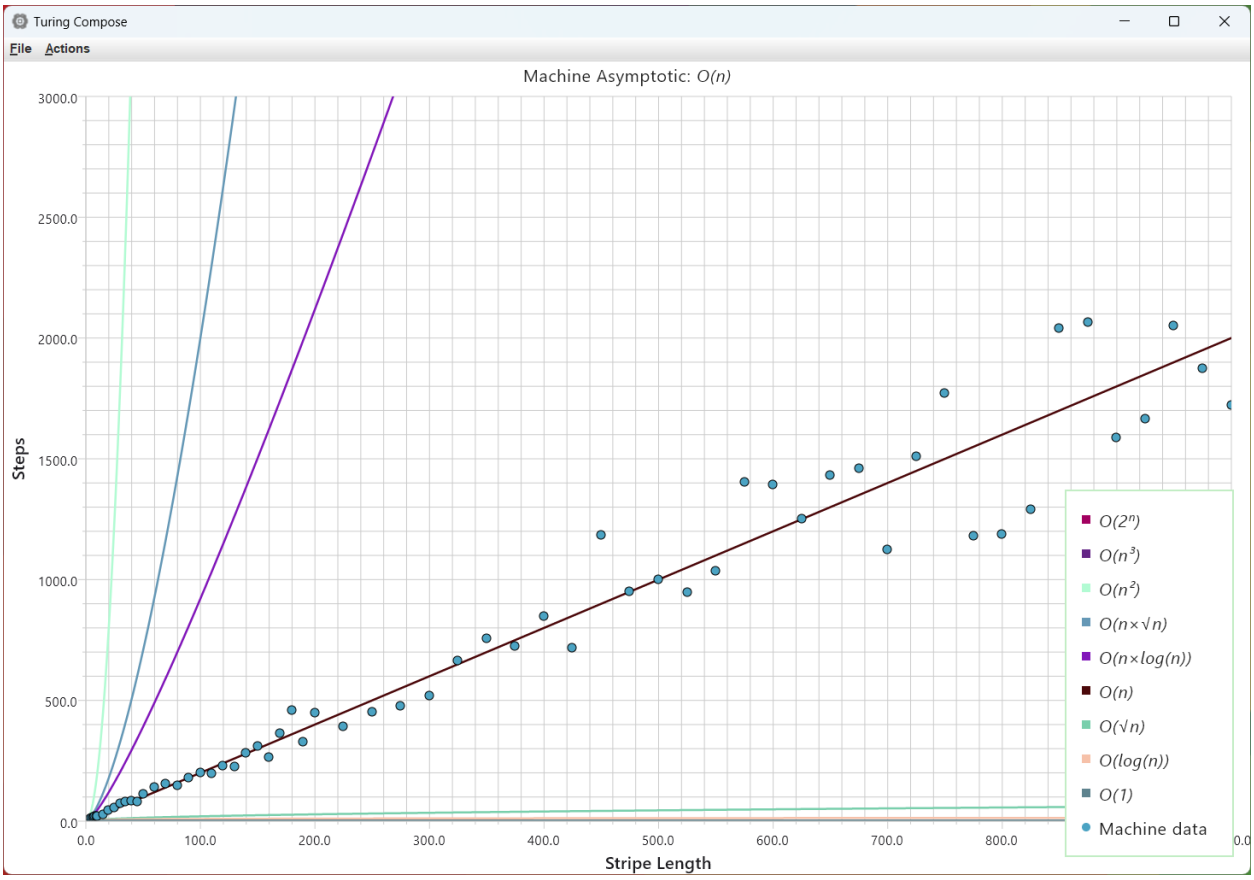


*Рисунок 5.10*

## 5.8. Рядок меню

На будь якому екрані можна побачити рядок меню, що знаходиться зверху екрана.

Цей рядок містить два підменю

- File – підменю що відповідає за роботу з файлами.
    - New file – кнопка що перенаправить користувача на пустий екран редагування програми.

- Load file – кнопка що відкриває діалог з вибором файлу, якщо це коректний файл машини то користувача перенаправить на екран редагування програми, а якщо файл не коректний, то в правому нижньому кутку буде показана помилка.
- Save file – кнопка що доступна лише на екрані редагування програми, при натисканні, користувача перенаправить на екран збереження.
- Recent files – кнопка що перенаправляє на стартовий екран де відображаються нещодавно відкриті файли.

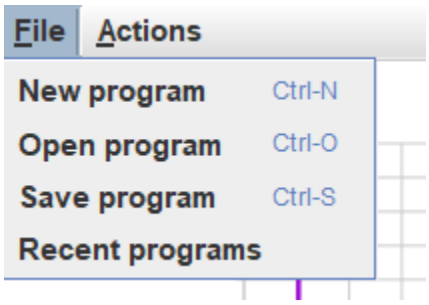


*Рисунок 5.11*

- Actions – підменю що відповідає за додаткові дії
  - Back – кнопка що перенаправляє користувача на попередній екран.
  - Exit – кнопка що закриває програму.

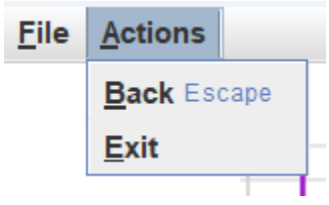


*Рисунок 5.12*

## 5.9. Компонент глобальної помилки

Цей компонент відображає помилки програми, та показується поверх всіх екранів.

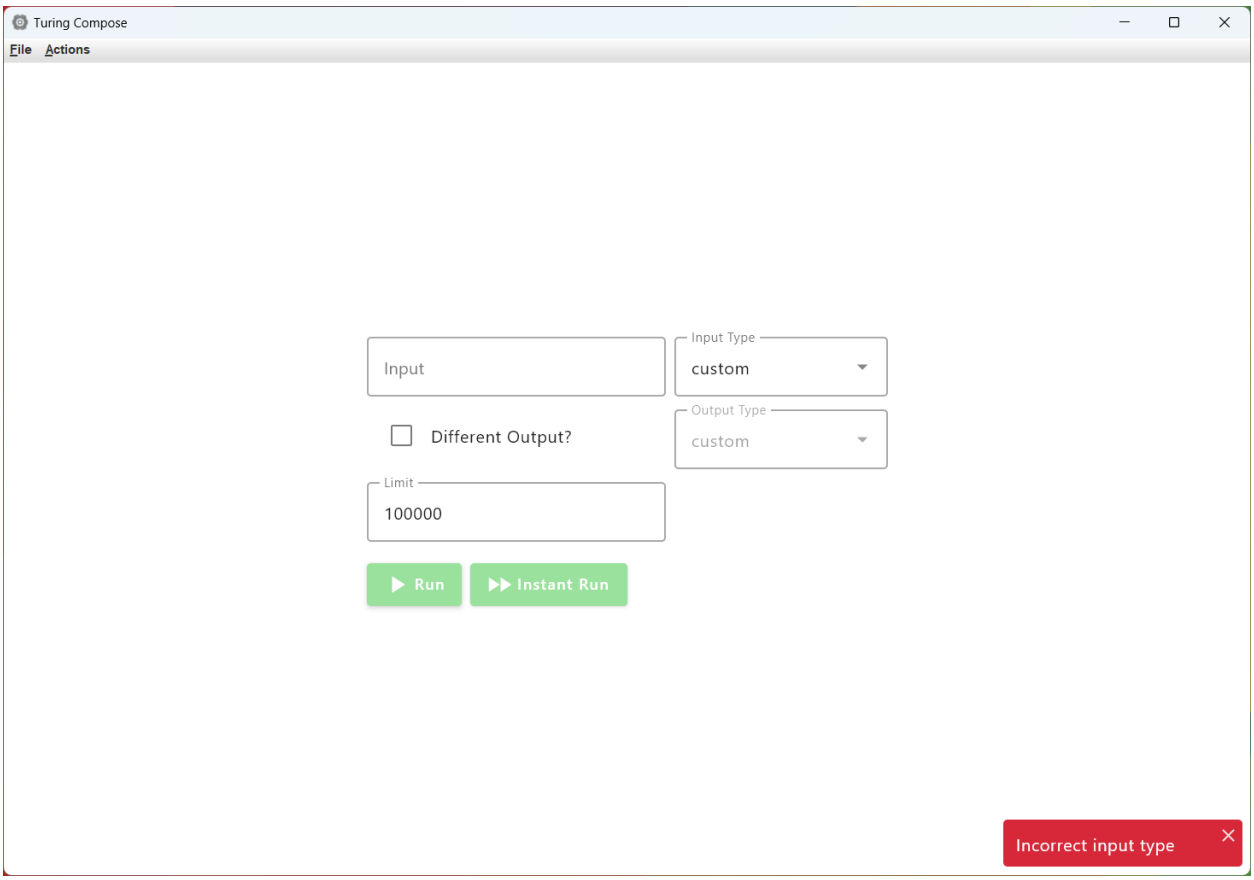


*Рисунок 5.13*

Компонент складається з тексту помилки та кнопки відхилення помилки.

Також помилка відхиляється автоматично, через 5 секунд після показу.

## 5.10. Компонент автодоповнення

На цьому компоненті відображаються значення які можуть бути автодоповненні до тексту що ввів користувач у рядок.

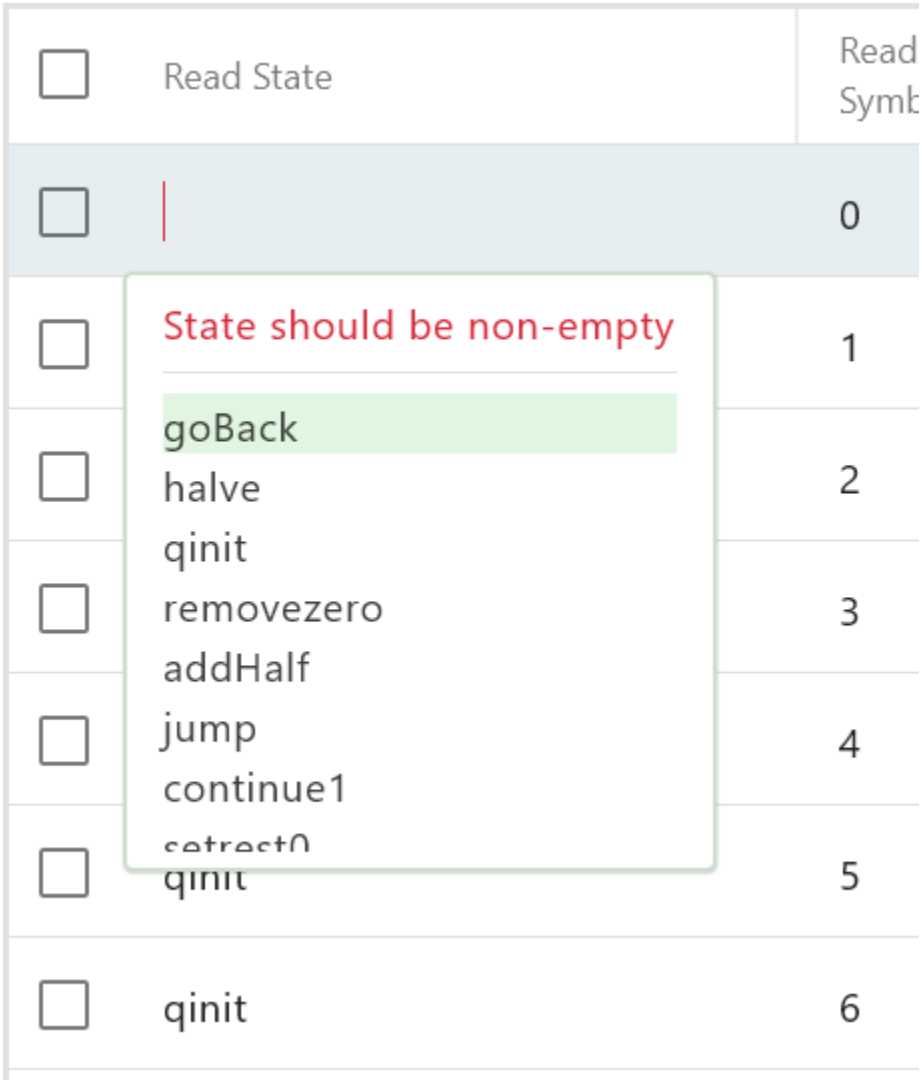


*Рисунок 5.14*

Особливістю цього компоненту є те що за замовчуванням при користуванні стрілками на клавіатурі втрачається фокус з поля вводу. Для того щоб обійти цю проблему було створено власний модифікатор компоненту, який забороняє стандартне фокусування на цьому компоненті, і натомість при натисканні

стрілок на сфокусованому полі вводу емулює події натискання стрілок на компоненті автодоповнення.

### 5.11. Компонент таблиці

Цей компонент відповідає за відображення рядків таблиці, можливості змінювати розміри стовпчиків та автопрокручування та заданого рядка.

| Read State | Read Symbol | Write State | Write Symbol | Move | Description |
|---|---|---|---|---|---|
| qinit | 0 | qinit | 0 | > | 1 |
| qinit | 1 | qinit | 1 | > | 2 |
| qinit | 2 | qinit | 2 | > | 3 |
| qinit | 3 | qinit | 3 | > | 4 |
| qinit | 4 | qinit | 4 | > | 5 |
| qinit | 5 | qinit | 5 | > | 6 |
| qinit | 6 | qinit | 6 | > | 7 |
| qinit | 7 | qinit | 7 | > | 8 |
| qinit | 8 | qinit | 8 | > | 9 |
| qinit | 9 | qinit | 9 | > | 0 |
| qinit | _ | halve | 0 | < | _ |
| halve | 0 | halve | 0 | < | |

*Рисунок 5.15*

Оскільки  Kotlin Multiplatform доволі новий фреймворк, то він ще не підтримує таблиці так як це не дуже затребуваний компонент, а на просторах інтернету немає такої бібліотеки яка виконує всі поставлені задачі, тому був написаний власний компонент таблиць та рядків таблиць.

Основною проблемою компоненту таблиці є проблема обробки великих даних під час зміни розмірів колонок. Проблема випливає з того що зміна розміру клітинки кожного рядку, викликає рекомпозицію кожного рядку, що в

свою чергу викликає рекомпозицію таблиці. Тому для вирішення цього компонент клітинки був переписаний вручну щоб підтримувати зміну розмірів не викликаючи рекомпозицію.

# Висновки

У даній роботі успішно реалізовано емулятор машини Тюрінга, який не лише відтворює функціонування цієї моделі, але й забезпечує визначення асимптотичної складності алгоритмів.

Результатами цієї роботи є власне сам емулятор, тобто програмний продукт, який моделює принципи роботи машини Тюрінга, дозволяючи користувачам виконувати алгоритми та спостерігати за їхнім виконанням у реальному часі, а також інтеграція функціоналу для автоматичного визначення асимптотичної складності алгоритмів, що дозволяє користувачам аналізувати ефективність алгоритмів та розуміти їхню поведінку при обробці даних.

Таким чином, розроблений емулятор машини Тюрінга з визначенням асимптотичної складності алгоритмів є корисним інструментом для поєднання теорії та практики при вивчені обчислювальних машин, сприяючи глибшому розумінню та ефективному навчанню ключових концепцій цієї галузі.

## Список використаних джерел


1. Kotlin | Kotlin. URL: https://kotlinlang.org/
2. Ternary search | GeeksForGeeks. URL: https://www.geeksforgeeks.org/ternary-search/
3. IntelliJ Idea | JetBrains. URL: https://www.jetbrains.com/idea/features/
4. IntelliJ Idea | Wikipedia. URL: https://uk.wikipedia.org/wiki/IntelliJ_IDEA
5. Koin | Koin. URL: https://insert-koin.io/docs/setup/why/
6. Поліноміальна інтерполяція | Wikipedia. URL: https://uk.wikipedia.org/wiki/Поліноміальна_інтерполяція
7. Машина Тюрінга | Wikipedia. URL: https://uk.wikipedia.org/wiki/Машина_Тюрінга
8. Turing machine | Wikipedia. URL: https://en.wikipedia.org/wiki/Turing_machine
9. Jetpack Compose | Android Developers. URL: https://developer.android.com/develop/ui/compose