**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**

Кафедра прикладної математики та моделювання складних систем

Допущено до захисту

Завідувач кафедри ПМ та МСС

_________ доц. Коплик І.В.
(підпис)

«__» ______________ 20__ р.

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр» / «магістр»

спеціальність 113 «Прикладна математика»

освітньо-професійна програма «Прикладна математика»

тема роботи

«Дослідження можливостей підвищення ефективності програмних алгоритмів для моделювання генератора випадкових та псевдовипадкових чисел»

**Виконавець**

студент факультету ЕлІТ

Железняков Владислав Сергійович

**Науковий керівник**

к.ф. – м.н.

Козлова Ірина Іванівна

Суми – 2020

# ЗМІСТ

ВСТУП……………………………………………………………………………..3

**І.Теоретична частина**

1. Основні визначення……………………………………………………………4

2. Історія розвитку «генераторів» випадкових чисел……………………………...........6

3. Генератори випадкових чисел, їх класифікація, особливості……………………...9

4. Методи передбачення наступних чисел та злом генераторів псевдовипадкових чисел………………………………………………...................12

**ІІ. Практична частина**

5. Алгоритм програми генератора……………………………………….............20

6.Досліджувана система………………………………………………….............23

Висновок………………………………………………………………………….26

Перелік джерел посилання……………………………………………….............27

Додатки…………………………………………………………………............28

# ВСТУП

У сучасному світі приховати інформацію, часом, буває дуже складно. У зв’язку з дуже швидким розвитком технологій, кожній людині необхідно постійно слідкувати за тим, щоб її дані були надійно захищені від сторонніх осіб. На сьогоднішній день майже кожна людина має хоча б один аккаунт у якійсь соціальній мережі. І до кожного аккаунту потрібен свій надійний пароль, адже якщо використовувати один, то в випадку його дізнання сторонніми особами, вони отримають доступ до всіх інших мереж. Тому в таких випадках часто використовують спеціальні генератори випадкових чисел[1].

Випадкові числа використовуються давно і повсюдно в таких сферах:

- Соціологічні та наукові дослідження.
- Моделювання
- Криптографія та інформаційна безпека
- Прийняття рішень в автоматизованих експертних системах
- Оптимізація функціональних залежностей
- Розваги та ігри

У літературі числа, згенеровані арифметичним способом, називаються псевдовипадковими, а метод їх генерації називають генератором псевдовипадкових чисел.

Послідовності псевдовипадкових чисел мають ряд істотних недоліків. Один з головних недоліків - циклічність послідовності. Тобто генератор псевдовипадкових чисел може дати, наприклад, 6100 чисел, що відрізняються один від одного, але потім, починаючи з 6101-го числа, буде циклічно відтворювати ці 6100 попередніх чисел. Іншим недоліком, властивим всім алгоритмічним методам генерації чисел, є очевидна залежність наступних чисел від попередніх.

# 1.ОСНОВНІ ВИЗНАЧЕННЯ

**ГПВЧ (генератор псевдовипадкових чисел)** – алгоритм, створюючий послідовність чисел, елементи якої майже незалежні один від одного.

**Генератор випадкових чисел (ГВЧ)** – алгоритм, який працює так само як і ГПВЧ, але додатково використовує випадкову величину, представлену різними джерелами ентропії.

ГВЧ = ГПВЧ + джерело ентропії

**Ентропія** – це міра хаосу.

**Інформаційна ентропія** - міра невизначеності, або ж непередбачуваності інформації.

**Текст** - упорядкована послідовність із символів алфавіту.

Як приклади алфавітів, що використовуються в сучасних інформаційних системах, можна навести наступні:

- алфавіт $Z_{33}$ - 33 літери українського алфавіту і пробіл;
- алфавіт $Z_{256}$ - символи, що входять в стандартні кодування ASCII;
- бінарний алфавіт - $Z_2 = \{0,1\}$;
- вісімкова або шістнадцяткова система числення;

**Пароль** – набір чисел або символів, який потрібен для підтвердження особистості та її повноважень.

**Випадкове число** - число, що представляє собою реалізацію випадкової величини.

**Псевдовипадкове число** - число, отримане детермінує-ванним алгоритмом, що використовується в якості випадкового числа.

**Фізичне випадкове число (істинно випадкове)** - випадкове число, отримане на основі деякого фізичного явища. Як правило, генерація випадкового числа складається з двох етапів:

**1**.Генерація нормалізованого випадкового числа (тобто рівномірно розподіленого від 0 до 1);

**2**.Перетворення нормалізованих випадкових чисел $r_i$ в випадкові числа $x_i$ , які розподілені по заданому закону розподілу або в необхідному інтервалі.

**Кейлоггер** – програма або деякий апарат, що може відстежувати дії користувача. Наприклад можуть реєструватися  рухи та натискання мишки, натискання клавіш та навіть дату та час, коли це було зроблено.

**Антифрод-захист** – спосіб захисту даних користувача від зловмисників. Наприклад, після декількох невдалих спроб введення паролю система автоматично блокує на певний час доступ до даних.

**Детермінований алгоритм** - алгоритмічний процес, який видає унікальний і зумовлений результат для заданих вхідних даних.

# 2.ІСТОРІЯ РОЗВИТКУ ГЕНЕРАТОРІВ ВИПАДКОВИХ ЧИСЕЛ

Як ми можемо згенерувати рівномірну послідовність випадкових чисел? Випадковість, настільки прекрасна в своєму різноманітті, що часто зустрічається в живій природі, але її не завжди легко було витягти штучним шляхом. Найдавніші з гральних кісток були знайдені на Середньому Сході, і вони датуються приблизно 24 століттям до нашої ери. Іншим прикладом може бути Китай, де ще в 11 столітті до нашої ери застосовувалося розбивання ударом черепашачого панцира, з подальшою інтерпретацією розміру отриманих випадкових частин. Століттями пізніше люди розрубували кілька разів стебла рослин і порівнювали розміри отриманих частин.

«Коли я поставив собі за мету отримати дійсно випадкове число, то не знайшов для цього нічого кращого, ніж звичайні гральні кістки» - писав в 1890 році Френсіс Гальтон в журналі Nature. «Після того, як кістки струшують і кидають в кошик, вони вдаряються одна об одну і об стінки кошика настільки непередбачуваним чином, що навіть після легкого кидка стає абсолютно неможливим визначити його результат».

В середині 40-их років XX століття людству знадобилося значно більше випадкових чисел, ніж можна було отримати від кидків кісток або розрізання стебел. Американський аналітичний центр RAND створив машину, яка була здатна генерувати випадкові числа з допомогою використання випадкового генератора імпульсів (щось на зразок електронної рулетки). Вони запустили її і через якийсь час отримали достатню кількість випадкових чисел, які опублікували в вигляді книги "Один мільйон випадкових чисел і сто тисяч нормальних відхилень".

У 1951 році випадковість нарешті була формалізована і втілена в реальному комп'ютері Ferranti Mark 1, який поставлявся з вбудованим генератором випадкових даних на основі дробових шумів і інструкцією, що дозволяє отримати 20 випадкових біт. Цей функціонал був розроблений Аланом Тьюрингом. Його колега Крістофер Стречі застосував його для створення «генератора любовних листів».

Джон фон Нейман розробив ГПСЧ в 1946 році. Його ідеєю було почати з деякого числа, взяти його квадрат, відкинути цифри з середини результату, знову взяти квадрат і відкинути середину, і т.д. Отримана послідовність, на його думку, мала ті ж властивості, що й випадкові числа. Теорія фон Неймана не витримала випробування часом, оскільки яке б початкове число ви не вибрали, згенерована таким чином послідовність вироджувалася в короткий цикл повторюваних значень, на кшталт 8100, 6100, 4100 8100, 6100, 4100 ...

Більшість популярних процесорів в 90-их роках не мали спеціальної інструкції для генерації випадкових чисел, так що вибір нормального початкового значення для генератора псевдовипадкових чисел мав дуже важливе значення. Це вилилося в проблеми з безпекою, коли Phillip Hallam-Baker виявив, що в браузері Netscape (домінуючому в той час) реалізація SSL використовувала для ініціалізації ГПСЧ комбінацію поточного часу і ID свого процесу. Він показав, як хакер може легко підібрати це значення і розшифрувати SSL-трафік. Вгадування початкового значення алгоритму генерації псевдовипадкових чисел - до сих пір популярна атака, хоча з роками вона стала злегка складніше.

У 1997 році програмісти були обмежені способами отримання по-справжньому випадкових чисел, так що команда SGI створила LavaRand, який складався з веб-камери, спрямованої на пару лава-ламп, що стояли поруч на столі. Картинка з цієї камери була відмінним джерелом ентропії - Справжнім Генератором Випадкових Чисел, таким же, як у Тьюринга. Але в цей раз виходило генерувати 165 Кб випадкових чисел в секунду. Винахід було тут же запатентовано. Більшість експериментів в цій області не витримали випробування часом, але один ГПВЧ, названий Вихрем Мерсенна і розроблений в 1997 році Макото Мацумото і Такудзі Нісімура, зміг утримати пальму першості. У ньому поєднувалися продуктивність і якість результатів, що дозволило йому швидко поширитися на багато мов і фреймворки. Вихор Мерсенна є витковим регістром зсуву з узагальненою віддачою і генерує детерміновану послідовність з великим періодом циклу. У поточній реалізації період дорівнює $2^{19937}$- 1 і саме ця реалізація входить сьогодні в більшість мов програмування.

У 1999 році Intel додав апаратний генератор випадкових чисел в чіпсет i810. Це було добре, оскільки дана реалізація давала по-справжньому випадкові числа (на основі температурних шумів), але працювало не настільки ж швидко, як програмні ГПСЧ, так що більшість криптододатків все ще використовують ГПСЧ, які тепер, принаймні, можна ініціалізувати початковим значенням від апаратного генератора випадкових чисел.

Це призводить нас до поняття криптографічно-безпечного генератора псевдовипадкових чисел (КБГПСЧ). КБГПСЧ стали дуже популярними в епоху SSL. У 2012 році INTEL додав у свої чіпи інструкції RDRAND і RDSEED, що дозволяють отримати справжні випадкові числа на основі тих же коливань температури, але тепер уже на швидкості до 500 Мб / с

В останні роки почали з'являтися також апаратні генератори випадкових чисел від сторонніх виробників і навіть повністю відкриті (як в плані софту, так і апаратної частини). Сила цих продуктів саме в відкритості: можна вивчити їх самостійно і навіть побудувати вдома самому із загальнодоступних компонентів. Прикладами можуть бути REDOUBLER і Infinite Noise TRNG.

# 3. ГЕНЕРАТОРИ ВИПАДКОВИХ ЧИСЕЛ, ЇХ КЛАСИФІКАЦІЯ ТА ОСОБЛИВОСТІ

Генератор псевдовипадкових чисел (ГПВЧ, англ. Pseudorandom number generator, PRNG) - алгоритм, який породжує послідовність чисел, елементи якої майже незалежні один від одного і підкоряються заданому розподілу (зазвичай рівномірному).

ГПВЧ має певний алгоритм, який можна відтворити.

ГВЧ - це отримання чисел повністю з будь-якого шуму, тиску або іншої міри ентропії, можливість прорахування яких наближається до нуля. При цьому в ГВЧ є певні алгоритми для вирівнювання розподілу.

Генератори випадкових чисел за способом отримання чисел діляться на:

- апаратні;
- табличні;
- алгоритмічні.

Табличні генератори в якості джерела випадкових чисел використовують заздалегідь підготовлені таблиці, що містять перевірені не корельовані числа і не є генераторами в строгому розумінні цього поняття. Недоліки такого способу очевидні: використання зовнішнього ресурсу для зберігання чисел, обмеженість послідовності, зумовленість значень. Як приклад табличного методу можна навести книгу.

Апаратні генератори (істинно) випадкових послідовностей повинні володіти джерелом ентропії. Розробка генераторів, що використовують джерела ентропії, генерується на не корельованих і статистично незалежних числах - досить складне завдання. Крім того, для більшості криптографічних додатків такий ГПВЧ не повинен бути предметом вивчення і впливів іншої сторони.

Алгоритмічний генератор є комбінацією фізичного генератора і детермінованого алгоритму. Такий генератор використовує обмежений набір даних, отриманий з виходу фізичного генератора для створення довгої послідовності чисел перетвореннями вихідних чисел. Даний вид генераторів

становить найбільший інтерес в силу його очевидних переваг над генераторами випадкових чисел інших видів.

### 3.1. Алгоритми генерації псевдовипадкових чисел

Розробка алгоритмічних генераторів може бути ще більш складним завданням в порівнянні зі створенням апаратних генераторів випадкових чисел.
Через дороговизну апаратних генераторів випадкових чисел в більшості випадків, як джерело ентропії використовуються ресурси обчислювальної машини, на якій виконується програма генерації ПВЧ. При відсутності апаратного генератора випадкових чисел в якості джерела ентропії можуть використовуватися:

- стан системного годинника;
- час затримок між натисканням клавіш клавіатури або рухами мишки;
- вміст буферів введення / виведення;
- значення, одержувані при роботі системи (час завантаження системи, взаємодія і т. п.).

Перевагами алгоритмічних генераторів є швидкодія, компактність реалізації. Основний недолік - низька якість «випадковості», - такі послідовності, як правило, не проходять більшості поліноміальних тестів на випадковість.
Ми вже згадували вище деякі методи отримання послідовності псевдовипадкових чисел, такі як метод серединних квадратів, вигаданий Джоном фон Нейманом або вихор Мерсена. Але їх набагато більше.
Більшість з них не є криптографічно стійкими. Для генерації ключів, шифрування, генерації солі для зберігання паролів в незворотному вигляді використовують більш захищені алгоритми. Найбільш відомі з них це алгоритм RSA, ANSI та хеш-функція.

### 3.2.Недоліки ГПВЧ

Головними недоліками ГПВЧ вважають:

- Передбачувана залежність між числами.

- Передбачуване початкове значення генератора.
- Мала довжина періоду генеруємої послідовності випадкових чисел, після якої генератор зациклюється.

Також доволі часто проводяться атаки на ГПВЧ. Це атака, спрямована на розкриття параметрів генератора псевдовипадкових чисел з метою подальшого передбачення псевдовипадкових чисел.

В залежності від того, які дані ГПВЧ простіше відслідковувати (вихідні значення, вхідні значення або внутрішній стан), можуть бути реалізовані наступні типи атак.

**Пряма криптоаналітична атака**

1. **Атаки, засновані на вхідних даних. Серед них:**
    - Атаки з відомими вхідними даними
    - Атаки з відтворюваними вхідними даними
    - Атаки на обрані вхідні дані
2. **Атаки, засновані на розкритті внутрішнього стану:**
    - Атака із зворотною прокруткою
    - Перманентне компрометування стану
    - Атака ітеративним вгадуванням
    - Зустріч посередині

Щоб захиститися від атак на ГПВЧ часто роблять один з наступних варіантів:

1. Використовують Хеш-функції, задля приховання вихідних значень генератора.
2. Хешують джерело ентропії різними постійно змінними значеннями: значення лічильника, температури процесора .
3. Періодично змінюють внутрішній стан ГПВЧ.

# 4.МЕТОДИ ПЕРЕДБАЧЕННЯ НАСТУПНИХ ЧИСЕЛ ТА ЗЛОМ ГЕНЕРАТОРІВ ПСВДОВИПАДКОВИХ ЧИСЕЛ

Через те, що Python використовує алгоритм Вихор Мерсена, для початку потрібно зрозуміти в чому він полягає та які в нього переваги та недоліки.

Вихор Мерсенна є витковим регістром зсуву з узагальненою віддачею (TGFSR). «Вихор» - це перетворення, яке забезпечує рівномірний розподіл псевдовипадкових чисел, що генеруються в 623 вимірах (для лінійних конгруентних генераторів воно обмежене 5 вимірами). Тому функція кореляції між двома послідовностями вибірок в вихідній послідовності вихору Мерсенна надзвичайно мала.

Псевдовипадкова послідовність, що породжується вихором Мерсенна, має дуже великий період, рівний числу Мерсенна, що більш ніж достатньо для багатьох практичних застосувань.

Існують ефективні реалізації Вихора Мерсенна, що перевершують за швидкістю багато стандартних ГПВЧ (зокрема, в 2-3 рази швидше лінійних конгруентних генераторів). Алгоритм вихору Мерсенна реалізований в програмній бібліотеці Boost, Glib і стандартних бібліотеках для C ++, Ruby , Python, MATLAB, R , PHP та Autoit.

Видані вихором Мерсенна послідовності псевдовипадкових чисел успішно проходять статистичні тести DIEHARD, що підтверджує їх хороші статистичні властивості.

Генератор не призначений для отримання криптографічно стійких випадкових послідовностей чисел. Для забезпечення криптостійкості вихідну послідовність генератора необхідно обробити одним з криптографічних алгоритмів хешування.

Вихор Мерсенна алгоритмічно реалізується двома основними частинами: рекурсивною і загартуванням. Рекурсивна частина – це регістр зсуву з лінійним зворотним зв'язком, в якому всі біти в його слові визначаються рекурсивно; потік вихідних бітів визначаються також рекурсивно функцією бітів стану.

Регістр зсуву складається з 624 елементів. Кожен елемент має довжину 32 біта за винятком першого елемента, який має тільки 1 біт за рахунок відкидання біта.

Процес генерації починається з логічного множення на бітову маску, що відкидає 31 біт (крім найбільш значучих). Наступним кроком виконується ініціалізація ($x_0$, $x_1$, ..., $x_{623}$) будь-якими беззнаковими 32-розрядними цілими числами. Наступні кроки включають в себе об'єднання і перехідні стани.

Простір станів має 19937 біт (624*32 - 31). Наступний стан генерується зміщенням одного слова вертикально вгору і вставкою нового слова в кінець. Нове слово обчислюється гамуванням середньої частини з виключених. Вихідна послідовність починається з $x_{624}$, $x_{625}$, ...

Параметри МТ були ретельно підібрані для досягнення згаданих вище властивостей. Параметри *n* і *r* обрані так, що характеристичний многочлен примітивний або $nw - r$ дорівнює числу Мерсенна 19937. Зверніть увагу, що значення *w* еквівалентно розміру слова комп'ютера. У цьому випадку це 32 біта. У той час як значення *n, m, r* і *w* фіксуються, значення останнього рядка матриці А вибирається випадковим чином. Для чисел Мерсенна тест на «простоту цілих» набагато простіший. Так, відомо багато простих чисел Мерсенна (тобто простих виду $2^p$-1), до p = 43112609.

У зв'язку зі змінами в технології, зокрема, зростанням продуктивності процесорів, були винайдені 64-бітові та 128-бітові версії МТ. 64-розрядної версії була запропонована Такудзі Нісимура у 2000 році, 128-розрядна - в 2006 році теж Такудзі Нісимура. Обидві версії мають той же період, що і оригінальний МТ.

64-бітний МТ має дві версії. Перша версія використовує те ж рекурсивне співвідношення, в другу версію були додані ще два вектора, що збільшило кількість членів характеристичного многочлена. У порівнянні з 32-розрядної МТ, 64-розрядної версії працює швидше.[6]

## 4.1 Особливості ГПВЧ в Python.

В Python існує три модуля, призначених для генерації випадкових / псевдовипадкових чисел: random, urandom і _random:

- _random реалізує алгоритм Mersenne Twister (Вихор Мерсенна) з невеликими змінами на мові C;
- urandom використовує зовнішні джерела ентропії (криптопровайдер Windows в функції CryptGenRandom) на мові C;
- random є оболонкою для модуля _random на Python, що поєднує обидві бібліотеки і має дві основні функції для генерації псевдовипадкових чисел: random () і SystemRandom ().

Random() використовує алгоритм MT (модуль _random), однак перш за все намагається ініціювати його за допомогою SEED, взятого з urandom, що перетворює ГПВЧ в ГВЧ (генератор випадкових чисел). Якщо викликати urandom не вдається (наприклад, відсутня /dev/urandom або не вдається викликати потрібну функцію з бібліотеки advapi32.dll), то в якості SEED використовується int(time.time() * 256) .

Цей модуль реалізує генератори псевдовипадкових чисел для різних розподілів.

Для цілих чисел рівномірний вибір з діапазону. Для послідовностей, рівномірного вибору випадкового елемента, функції для генерації випадкової перестановки списку на місці та функції для випадкової вибірки без заміни.

На реальній лінії є функції для обчислення рівномірного, нормального (гауссового), лонормального, негативного експоненціального, гамма-та бета-розподілів. Для генерації розподілів кутів доступний розподіл фон Мізеса.

Практично всі функції модуля залежать від основної функції random (), яка генерує випадковий поплавок рівномірно у напіввідкритому діапазоні [0.0, 1.0). Python використовує Mersenne Twister як основний генератор. Він виробляє 53-бітні точні поплавці і має період 2 ** 19937-1. Основна реалізація в C є як швидкою, так і безпечною. Вихор Мерсена - один з найбільш широко перевірених генераторів

випадкових чисел, що існують. Однак, будучи повністю детермінованою, вона не підходить для всіх цілей і абсолютно непридатна для криптографічних цілей.

SystemRandom() викликає модуль urandom, який використовує зовнішні джерела для генерації випадкових даних.

Зміни в реалізації алгоритму MT полягає в тому, що замість одного числа, заснованого на одному з 624 чисел з поточного стану ГПВЧ, використовуються два числа:

```
random_random()
{
   unsigned long a=genrand_int32(self)>>5, b=genrand_int32(self)>>6;
   return PyFloat_FromDouble((a*67108864.0+b)*(1.0/9007199254740992.0));
}
```

Так само, на відміну від PHP, форматувати генератор можна не тільки за допомогою long-змінної, але і за допомогою будь-якої послідовності байтів (відбувається виклик init_by_array ()), що і відбувається при імпорті модуля random за допомогою зовнішнього джерела ентропії (береться 32 байта з urandom), а в разі, коли це не вдається, використовується time ():

```
if a is None: try:
        a = int.from_bytes(_urandom(32), 'big')
      except NotImplementedError:
        import time
a = int(time.time() * 256)
```

[1]

## 4.2 Можливість передбачення наступних чисел генератора

Майже на кожній мові можна знайти варіанти для передбачення наступних чисел того чи іншого генератора.

### 4.2.1 Можливість злому генератора в Python

Однією особливістю фреймворків Python є те, що на відміну від PHP, Python запускається один раз разом з веб-сервером, а значить, ініціалізація стану за замовчуванням відбувається один раз при виконанні команди *import random* або при примусовому виклику *random.seed ()* (це вкрай рідкісний випадок для веб-додатків), що дозволяє провести атаку на стану MT за наступним алгоритмом:

- знаходимо сценарій, який виводить значення *random.random ()* (наприклад, в Plone цим займається логер помилок (файл SiteErrorLog.py), він виводить на сторінку "помилка з номером *** зафіксована", де буде показано випадкове число);
- виконуємо послідовно серію запитів, де фіксуємо випадкові числа. Номери запитів: 1, 2, 199, 200, 511, 625;
- 313-м запитом ми здійснюємо передбачати дії (наприклад, генерацію посилання на скидання пароля);
- на основі запитів 1,199 ми визначаємо стану state_1 [1], state_1 [2], state_1 [397];
- на основі запитів 2,200 - стану state_1 [3], state_1 [398];
- на основі запиту 511 - state_2 [397];
- на основі запиту 625 - state_3 [1].

Точність станів залежить від індексу елемента стану (і): для $i(mod\ 2) = 0$ ентропія становить $2^6$, для $i(mod\ 2) = 1 – 2^5$.

Далі за допомогою запитів 1, 2, 199, 200 можна визначити стану state_1 [1], state_1 [2], state_1 [3], state_1 [397], state_1 [398], на основі яких генеруються стану state_2 [1] і state_2 [2], з яких і виходить випадкове число запиту №313. Однак, ентропія цього числа складає $2^{24}$. Ентропія скорочується за допомогою запитів 511 і 625. Ці запити допомагають обчислити стану state_2 [397], state_3 [1]. Це зменшує кількість варіантів станів до $2^8$, тобто існує всього 256 варіантів «випадкового» числа, використовуваного в запиті №313.

Необхідною умовою виконання атаки є те, що ніхто не «вклинився» в процес запитів, тим самим не вплинувши на зміну стану ГПВЧ (іншими словами, що індекси станів будуть визначені правильно). Також необхідно щоб запит №1 використовував елементи стану ГПВЧ з індексами не більш 224, інакше запит №200 буде використовувати інший стан генератора, що не дозволить виконати алгоритм. Імовірність цієї події складає 36%.

Тому додаткове завдання запиту №625 - визначити, що всі попередні запити дійсно проводилися в потрібних станах і ніхто не вклинився в процес запитів.[3]

### 4.2.2 Можливість передачення чисел в модулі Math.random на Java.Scripts

Math.random () повертає значення Числа з позитивним знаком, більшим або рівним 0, але менше 1, вибраним випадковим чином або псевдовипадковим шляхом з приблизно рівномірним розподілом у цьому діапазоні, використовуючи алгоритм чи стратегію, залежну від реалізації. Ця функція не бере аргументів.

Math.random () - найбільш відоме і часто використовуване джерело випадковості у Javascript. У V8 та більшості інших двигунів Javascript реалізовано за допомогою генератора псевдовипадкових чисел (PRNG). Як і у всіх PRNG, випадкове число походить від внутрішнього стану, який змінюється за допомогою фіксованого алгоритму для кожного нового випадкового числа. Отже, для заданого початкового стану послідовність випадкових чисел є детермінованою. Оскільки розмір біта n внутрішнього стану обмежений, числа, які генерує PRNG, з часом повторяться. Верхня межа тривалості цього циклу перестановки становить $2^n$.

Існує багато різних алгоритмів PRNG; серед найвідоміших - Mersenne-Twister та LCG. Кожен має свої особливості, переваги та недоліки. В ідеалі, алгоритм має використовувати якомога менше пам'яті для початкового стану, бути швидким у виконанні, мати велику тривалість періоду та пропонувати якісний випадковий розподіл. Хоча використання пам'яті, продуктивність та тривалість періоду можна легко виміряти або обчислити, якість - визначити складніше. Існує багато статистичних тестів, щоб перевірити якість випадкових чисел. Стандартний тестовий набір PRNG - TestU01, включає в себе більшість цих тестів.

До недавнього часу (до версії 4.9.40) PR8 вибору V8 був MWC1616. Він використовує 64 біти внутрішнього стану і виглядає приблизно так:

```
uint32_t state0 = 1;
uint32_t state1 = 2;
uint32_t mwc1616() {
state0 = 18030 * (state0 & 0xFFFF) + (state0 >> 16);
state1 = 30903 * (state1 & 0xFFFF) + (state1 >> 16);
return state0 << 16 + (state1 & 0xFFFF);
}
```

32-бітове значення потім перетворюється на число з плаваючою комою між 0 і 1 відповідно до специфікації.

MWC1616 використовує мало пам'яті і досить швидко обчислює, але, на жаль, пропонує нижчу якість:

- Кількість випадкових значень, які він може генерувати, обмежена $2^{32}$, на відміну від $2^{52}$ чисел між 0 і 1, які плаваюча точка з подвійною точністю може представляти.
- Більш значна верхня половина результату майже повністю залежить від значення state0. Тривалість періоду становила б не більше 232, але замість кількох великих циклів перестановки існує багато коротких. При погано обраному початковому стані тривалість циклу може бути менше 40 мільйонів.
- Він не дає багатьох статистичних тестів у наборі TestU01.

Зрозумівши проблему, можна повторно застосувати Math.random на основі алгоритму під назвою *xorshift128+*. Він використовує 128 біт внутрішнього стану, має тривалість періоду $2^{128}$-1 і проходить усі тести з набору TestU01.

```
uint64_t state0 = 1;
uint64_t state1 = 2;
uint64_t xorshift128plus() {
  uint64_t s1 = state0;
  uint64_t s0 = state1;
```

```
  state0 = s0;
  s1 ^= s1 << 23;
  s1 ^= s1 >> 17;
  s1 ^= s0;
  s1 ^= s0 >> 26;
  state1 = s1;
  return state0 + state1;
}
```

Однак потрібно розуміти: хоча *xorshift128+* є величезним покращенням порівняно з MWC1616, він все ще не є криптографічно захищеним. Для випадків використання, таких як хешування, створення підписів та шифрування / дешифрування, звичайні PRNG непридатні. API веб-криптографії вводить window.crypto.getRandomValues - метод, який повертає криптографічно захищені випадкові значення за ціною продуктивності.[5]

**4.3 Захист**

З усіх фреймворків та веб-додатків досі ще не вдалося знайти недоліки в системі та можливість передбачування чисел. Найбільш вірогідним претендентом є веб-додаток Plone, бо саме він має логер помилок(файл, в якому містяться всі помилки та баги). Під час появи нової помилки він генерує тип помилки з «випадковим» числом. На Python 2.7 перехід з типу float в str() обрізає останні 5 цифр і цим самим розширює кількість варіантів для підбору. Але на жаль Python 2.7 більше не підтримується. В Python третього покоління цифри не обрізаються при зміні типів, і саме через це атаки на Plone стають ще більш актуальними.

Але враховуючи всі підсумовуючі факти можна зробити висновок, що використання модулів random() не дає нам надійного захисту від можливостей вгадування наступних чисел.

# 5.АЛГОРИТМ ПРОГРАМИ ГЕНЕРАТОРА

Мною була реалізована програма для генерації надійних паролів, які можуть бути використані в будь-яких сферах, не використовуючи модуль random().

Але через те, що використання деяких методів для отримання псевдовипадкових чисел поодинці має дуже малий період, ми будемо використовувати об'єднання цих алгоритмів і часткову зміну їх для більшої захищеності. Використання двох ентропій зводить вірогідність того, що пароль повториться більше одного разу майже до нульової.

## 5.1 Алгоритм генерації паролів за допомогою генератора випадкових чисел

Через неможливість використання апаратного джерела генерації випадкових чисел ми будемо використовувати кількість секунд від початку комп'ютерної епохи та дуже малі або великі числа з оперативної пам'яті. В цьому нам допоможе бібліотека numpy. Щоб забезпечити додаткову захищеність ми будемо видаляти всі файли після генерації паролю.

За допомогою функції numpy.empty() ми заповнюємо масив частинками блоків пам'яті комп'ютера. Це абсолютно безпечні дані, які можуть бути згенеровані, наприклад, рухом мишки, натисканням клавіші, завантаженням системи або іншими незначними даними.

Наступним кроком буде перевірка наших чисел, щоб вони не повторювались. Через те що, числа надзвичайно малі, або надзвичайно великі і іноді їх степені можуть бути як і 314, так і -308. Ми маємо зробити їх приблизно в одному діапазоні. Обираємо для зручності діапазон від -10 до 10.

Перше число з масиву використовуємо для видозміненого методу серединних квадратів. Сутність методу полягає в тому, щоб взяти якесь чотирьохзначне випадкове число. Порахуємо його квадрат та запишемо число, що складається з чотирьох середніх цифр цього числа. Але при певних цифрах число може зациклитись. Саме тому ми будемо брати не середні, а перші дві, та дві останні цифри і складати з них число для наступної ітерації.

Наприклад, візьмемо наше число $X_0 = 1387$

$X_0^2 = 1\ 923\ 769$, тоді $X_1 = 1969 \Rightarrow X_1^2 = 3\ 876\ 961$, $X_2 = 3861$, і так далі.

Через те що час ми вимірюємо у секундах і значення дуже схожі, а наша програма має бути максимально швидкою і, при цьому, безпечною то ми будемо використовувати лише мілі- та мікросекунди.

Застосовуємо другий метод – лінійно конгруентний. Або більш відомий, як ділення за модулем. Знайдемо залишок від ділення нашого часу на число, отримане добутком видозміненого методу серединних квадратів помножене на друге число з оперативної пам'яті.

Формула знаходження нашого числа виглядає так:

де $X_n$ – це наш час, c = 0, a – друге число з пам'яті округлене до цілого, а m – наше число пораховане модернізованим методом серединних квадратів.

Так ми знайшли лише перше число. Перетворюємо його у символ за допомогою бібліотеки **string**. Користувач може обрати як буде виглядати його

$$X_{n+1} = (aX_n + c) \mod m,$$

пароль. Якщо він обере «Використовувати спеціальні символи», то ми знайдемо залишок від ділення знайденого числа на 100 і оберемо потрібний символ з бібліотеки. Наступні ітерації будуть відрізнятися тим, що ми будемо використовувати вже наступне число, яке ми взяли з оперативної пам'яті. А в видозміненому методі серединних квадратів ми будемо використовувати число, отримане з попередньої ітерації.

## 5.2 Приклад генерації паролю на основі реальних чисел

Розглянемо програму на конкретному прикладі для ще більшого розуміння. Першим кроком згенеруємо число, що відповідає кількості секунд з початку комп'ютерної епохи. Воно дорівнює ***1591533906.1936944.*** Далі дізнаємось деякі числа, з яких почнемо рахувати. Візьмемо їх з оперативної пам'яті. Наприклад, першими двома числами буде ***1.1520175616206e-311*** та ***5.089162945494e-312***. Щоб ці числа належали проміжку [-10,10] помножимо їх на 10 в степені 311 та 312

відповідно. Якщо ж наш степінь більше нуля, то нам потрібно поділити 10 помножене на степінь.

Перше число будемо використовувати для нашого видозміненого методу серединних квадратів. Зробивши його чотирьох значним воно буде дорівнювати $a = 1152$. Піднесемо його у квадрат.

$square^2 = 1152^2 = 1327104$.

Перші дві цифри 13, останні дві 04, отже отримаємо число 1304.

Через те що час ми вимірюємо у секундах, то значення дуже схожі, тому ми будемо використовувати лише мілі- та мікросекунди. Для цього домножимо число на $10^5$ і отримуємо $t = 159153390619369$.

Застосуємо лінійно-конгруентний метод для наших чисел.

Для цього помножимо наш час на друге число з оперативної пам'яті, та знайдемо модуль від ділення на число, отримане видозміненним методом серединних квадратів.

$$linear = (t*Second_Number) \% square =$$

$$= (159153390619369*5) \ \% 1304 = 649.$$

Щоб час постійно змінювався і це давало додатковий захист ми будемо додавати до нашого часу невеликий проміжок, який буде дорівнювати $t - (t\text{-}1)$. Але наше (t-1) ми округлюємо до цілого числа. І виходить, що час буде змінюватись менше ніж на секунду.

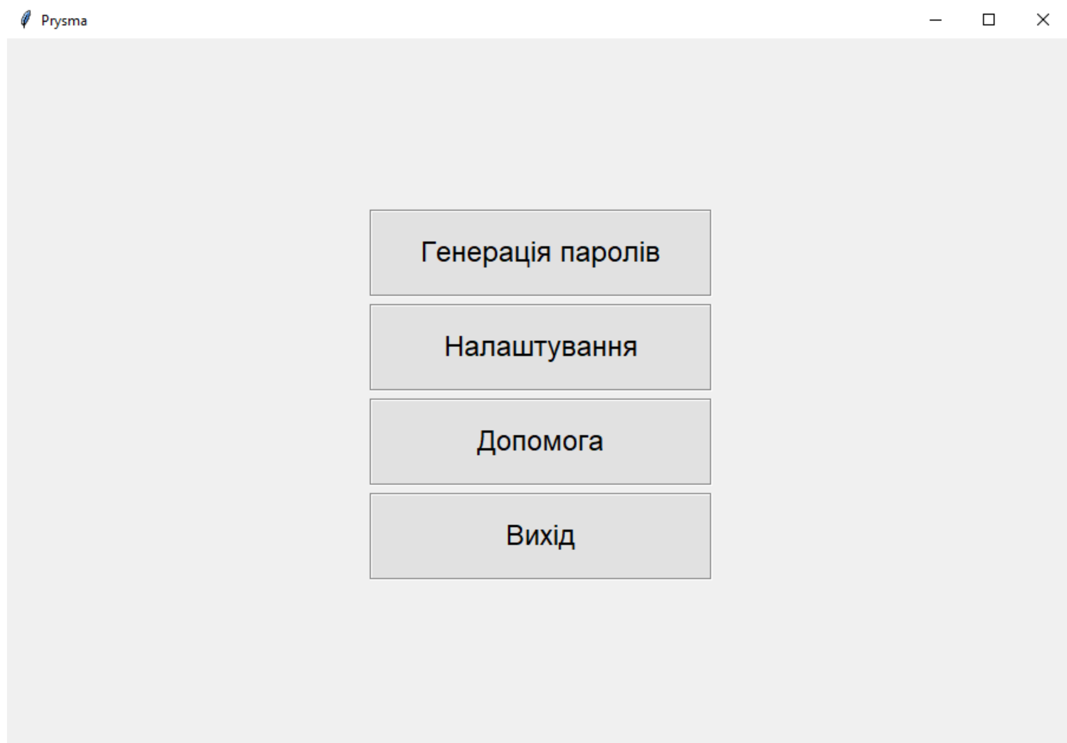
Останнім кроком буде отримати з нашого числа символ. Ми будемо використовувати функцію без спеціальних символів, отже будемо знаходити модуль від 63:

$linear \% 63 = 649 \% 63 = 29$.

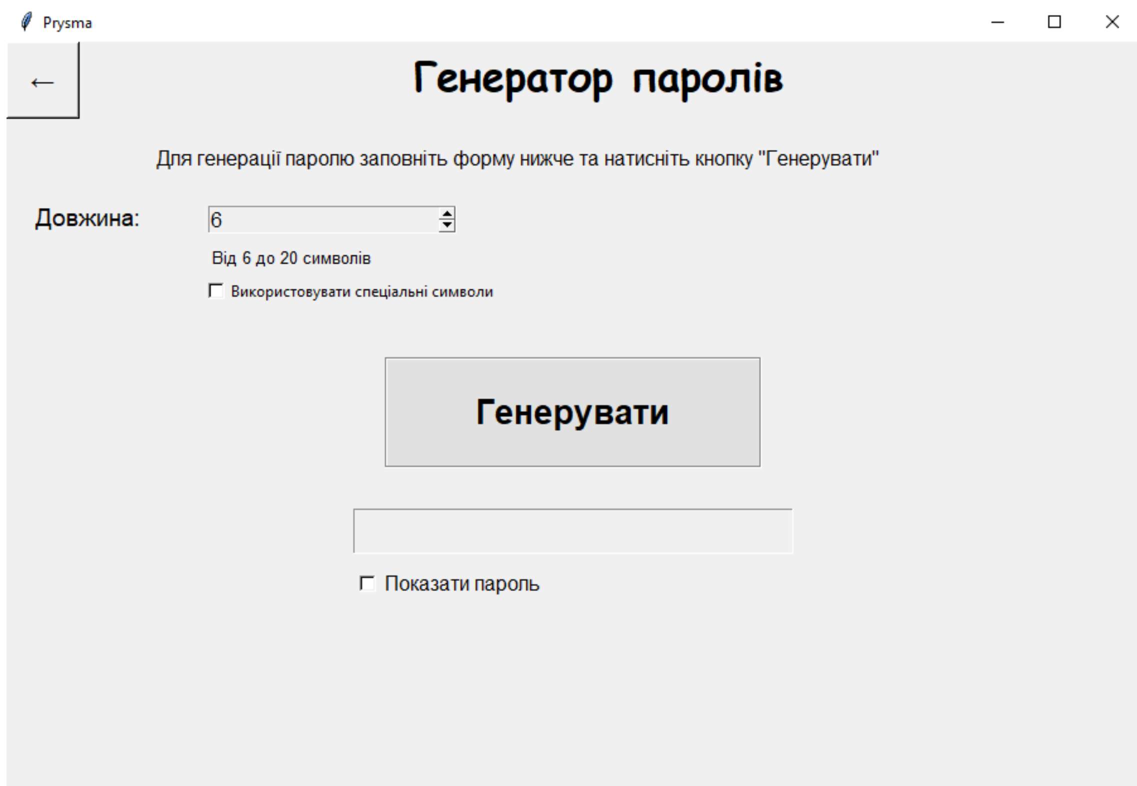
Через те, що в програмуванні рахунок зазвичай починається з 0, а не з 1, то нам потрібно знайти 30 елемент. Це літера t. Вона і буде нашим першим символом паролю. Повністю пароль буде виглядати в кінці так: **t4S0I2QO.** Можна отримати і набагато більші паролі(від 6 символів до 20 символів).

# 6. ДОСЛІДЖУВАНА СИСТЕМА

Для більшої зручності був написаний інтерфейс, який виглядає так:



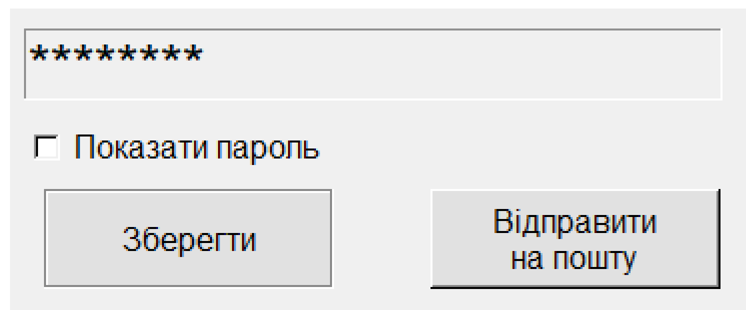

Ми можемо обрати іншу тему в налаштуваннях, та відкрити вікно для генерації паролю, яке виглядає так:



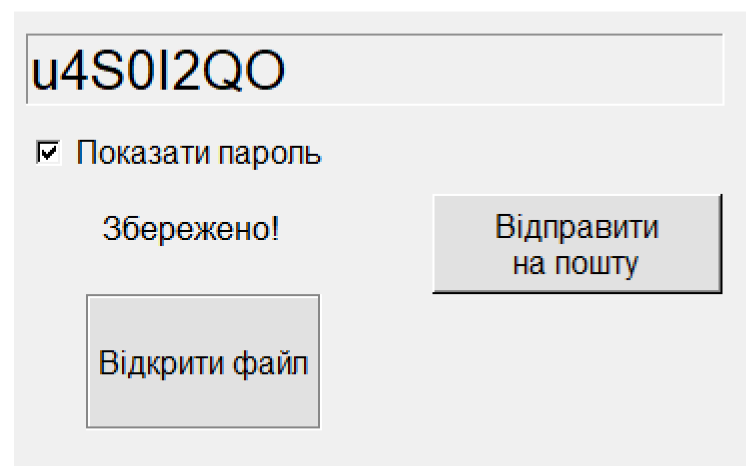
Користувач може обрати кількість символів у паролі (від 5 до 20), та обрати чи будуть використовуватися в паролі спеціальні символи, такі як:

!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~

Після генерації у нас з’явилась можливість переглянути пароль, записати його у текстовий документ, на випадок, якщо зараз він нам не потрібний, або ж нам потрібно його зберегти.

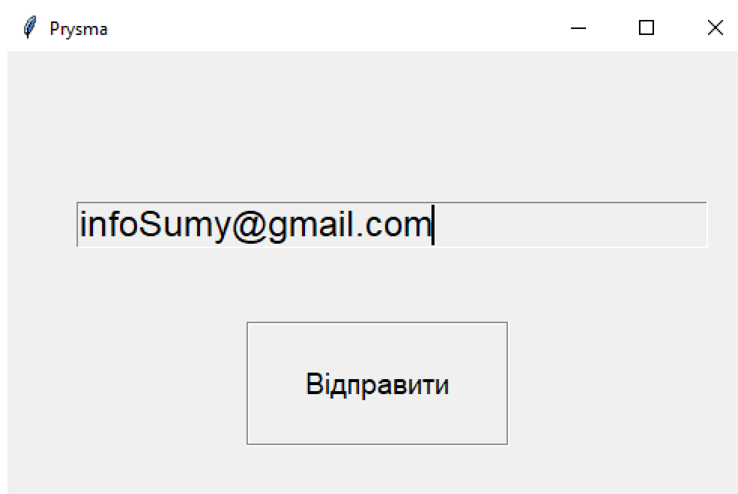



Після запису ми можемо одразу відкрити документ з програми, або в будь-який інший час на комп’ютері.




Також у нас є можливість відправити його на пошту, що є дуже зручним, якщо потрібно генерувати велику кількість паролів, наприклад, для співробітників компанії.

Для відправки на пошту відкриється вікно, де потрібно написати свою e-mail адресу на натиснути кнопку відправити.




Прибизно так буде виглядати лист, отриманий на пошту:

*Hello, thanks for contacting!*

*You have made a request in the program Prysma.*

*Your new password: t4S0I2QO*

*If it wasn`t you, ignore this message.*

Через невелику потужність обчислювальної системи ми не можемо дізнатися кількість ітерацій для зациклення системи. Адже кожний рух мишкою, кожне натискання клавіші або кожне повідомлення яке було отримано на комп’ютер не дає даним оперативної пам’яті залишатися однаковими. І навіть, якщо все-таки якимось чином дані в оперативній пам’яті в двох паролях будуть однаковими, то використання другої ентропії, а саме, кількості секунд не «дозволить» отримати однаковий пароль, через те що цей час не може повторюватись.

За даними сайту https://password.kaspersky.com/ru/ для злому пароля який складається з 8 символів потрібно приблизно 12 днів неперервної роботи програми брутфорсу. Якщо використовувати пароль для соціальних мереж, то його не зможуть підібрати, бо спрацює антифрод захист. А отже, пароль можна вважати достатньо надійним.

Також, мною був проведений експеримент, в якому було згенеровано 1,1млн паролів, а це трохи більше ніж $2^{20}$. І жоден з них не повторювався двічі.

# ВИСНОВОК

На підставі вивчення даної роботи можна виділити основні переваги та недоліки алгоритму об'єднання методів генерації випадкових чисел. Перевагами є більша надійність та непередбачуваність наступного числа. Використовуючи декілька методів ми збільшуємо кількість можливих варіантів до зациклення. А разом з ентропією вірогідність повторення ще менша. Через те що програма буде використовуватись на різних комп'ютерах, а отже дані в пам'яті теж будуть різні, вірогідність повторення майже нульова. Для порахування кількості ітерацій до зациклення паролів потрібна потужна обчислювальна машина. Хоч генерація чисел, а разом з ним і паролів, відбувається дуже швидко (приблизно 750-760 паролів на секунду), але їх перевірка потребує набагато більше часу. Для перевірки 1,1 млн паролів, згенерованих моєю програмою, знадобилося 30 годин (приблизно 610 паролів за хвилину), а на їх генерацію було витрачено лише 22 хвилини. Цей додаток може бути актуальним та корисним для різних компаній, коли потрібно згенерувати всім співробітникам пароль для певного сайту при реєстрації. Серед недоліків може бути те, що на різних версіях ОС програма може працювати не зовсім коректно.

## ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Сліповичев.І.І. «Генератор псевдовипадкових чисел» Навчальний посібник. 2017 р.
2. *Martin Aspeli.* Professional Plone 4 Development. — Packt Publishing Ltd., 2011. — 516 с.
3. Безпека випадкових чисел в Python [Електронний ресурс]. Режим доступу: https://habr.com/ru/company/pt/blog/156133/
4. Бібліотека Random в Python [Електронний ресурс]. Режим доступу: https://docs.python.org/release/2.6.8/library/random.html
5. Math.random in JavaScript [Electronic resource]. Mode of access: https://v8.dev/blog/math-random
6. *M. Matsumoto, T. Nishimura.* Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator (англ.) // ACM Trans. on Modeling and Computer Simulations : journal. — 2017.

# ДОДАТКИ

## Додаток А

```
import time, os, string, copy
import numpy as np
from decimal import Decimal
import tkinter as tk
from tkinter.filedialog import *
from tkinter import messagebox
import tkinter.ttk as ttk
import smtplib



"""Объявление функций для полноценной работы программы"""



class Password:
    """Main menu"""
    def main_menu(self):
        """Place widgets"""
        self.button_generate.place(x = 307, y = 145)
        self.button_settings.place(x = 307, y = 225)
        self.button_help.place(x = 307, y = 305)
        self.button_exit.place(x = 307, y = 385)

        """Hide Other Widgets"""
        """Hide generation"""
        self.label_generation.place_forget()
        self.check_special_symbols.place_forget()
```

```
        self.button_generation.place_forget()
        self.button_see.place_forget()
        self.button_open.place_forget()
        self.button_write.place_forget()
        self.text.place_forget()
        self.label_genHelp.place_forget()
        self.label_len.place_forget()
        self.number_column.place_forget()
        self.label_symbols.place_forget()
        self.label_save.place_forget()
        #self.f_open.place_forget()
        self.button_mail.place_forget()
        """Hide Back"""
        self.button_back.place_forget()
        """Hide Settings"""
        self.label_settings.place_forget()
        self.theme_dark_radio.place_forget()
        self.theme_white_radio.place_forget()
        self.theme_lightdark_radio.place_forget()
        self.label_dark_theme.place_forget()
        self.label_white_theme.place_forget()
        self.label_lightdark_theme.place_forget()

    def generation_for_one(self):
        """Show Widgets"""
        self.label_generation.place(x = 320, y = 0)
        self.check_special_symbols.place(x = 155, y = 185)
```

```
        self.button_generation.place(x = 300, y = 250)
        self.button_see.place(x = 275, y = 415)
        self.text.place(x = 275, y = 370)

        self.label_genHelp.place(x = 116, y = 80)
        self.label_len.place(x = 20, y = 125)
        self.number_column.place(x = 160, y = 130)
        self.label_symbols.place(x = 160, y = 160)

        if len(self.text.get())!= 0:
            self.button_write.place(x = 285, y = 450)
            self.button_mail.place(x = 478, y = 450)

        self.button_back.place(x = 0, y = 0)
        """Hide other Widgets"""
        self.button_generate.place_forget()
        self.button_settings.place_forget()
        self.button_help.place_forget()
        self.button_exit.place_forget()

    def settings(self):
        """Show Widgets"""
        self.label_settings.place(x = 375, y = 150)
        self.theme_dark_radio.place(x = 200, y = 250)
        self.theme_white_radio.place(x = 400, y = 250)
        self.theme_lightdark_radio.place(x = 600, y = 250)
        self.button_back.place(x = 0, y = 0)
        self.label_dark_theme.place(x = 190, y = 200)
        self.label_white_theme.place(x = 390, y = 200)
```

```
        self.label_lightdark_theme.place(x = 595, y = 200)
        """Hide other Widgets"""
        self.button_generate.place_forget()
        self.button_settings.place_forget()
        self.button_help.place_forget()
        self.button_exit.place_forget()

    def help(self):
        pass

    def white_theme(self):
        self.color = "#f0f0f0"
        self.color_text = "#000000"
        self.color_menu = "#e1e1e1"
        self.color_active_bg = "#e5f1fb"
        """Settings"""
        self.root["bg"] = self.color
        self.label_settings.configure(bg = self.color, foreground = self.color_text)
        self.button_back.configure(bg = self.color, activebackground = "#21252b", foreground = self.color_text, activeforeground = self.color_text)
        self.label_dark_theme.configure(bg = self.color, foreground = self.color_text)
        self.label_white_theme.configure(bg = self.color, foreground = self.color_text)
        self.label_lightdark_theme.configure(bg = self.color, foreground = self.color_text)
        """Menu"""
        self.button_generate.configure(bg = self.color_menu,  foreground = self.color_text, activeforeground = self.color_text, activebackground = self.color_active_bg)
        self.button_settings.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
```

```
        self.button_help.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.button_exit.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        """Generation"""
        self.label_generation.configure(bg = self.color, foreground = self.color_text)
        self.check_special_symbols.configure(bg = self.color, activebackground = self.color, foreground = self.color_text, activeforeground = self.color_text)
        self.text.configure(readonlybackground = self.color, bg = self.color, foreground = self.color_text)
        self.button_generation.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.button_see.configure(bg = self.color, activebackground = self.color, foreground = self.color_text, activeforeground = self.color_text)
        self.button_write.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.label_genHelp.configure(bg = self.color, foreground = self.color_text)
        self.label_len.configure(bg = self.color, foreground = self.color_text)
        self.number_column.configure(bg = self.color, foreground = self.color_text)
        self.label_symbols.configure(bg = self.color, foreground = self.color_text)
        self.label_save.configure(bg = self.color, foreground = self.color_text)
        self.button_open.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.button_mail.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)

    def lightdark_theme(self):
        self.color = '#282c34'
        self.color_text = "#edaf6b"
```

```
self.color_menu = "#21252b"
self.color_active_bg = "#21252b"
"""Settings"""
self.root["bg"] = self.color
self.label_settings.configure(bg = self.color, foreground = self.color_text)
self.button_back.configure(bg = self.color, activebackground = "#21252b", foreground = self.color_text, activeforeground = self.color_text)
self.label_dark_theme.configure(bg = self.color, foreground = self.color_text)
self.label_white_theme.configure(bg = self.color, foreground = self.color_text)
self.label_lightdark_theme.configure(bg = self.color, foreground = self.color_text)
"""Menu"""
self.button_generate.configure(bg = self.color_menu,  foreground = self.color_text, activeforeground = self.color_text, activebackground = self.color_active_bg)
self.button_settings.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
self.button_help.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
self.button_exit.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
"""Generation"""
self.label_generation.configure(bg = self.color, foreground = self.color_text)
self.check_special_symbols.configure(bg = self.color, activebackground = self.color, foreground = self.color_text, activeforeground = self.color_text)
self.text.configure(readonlybackground = self.color, bg = self.color, foreground = self.color_text)
self.button_generation.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
self.button_see.configure(bg = self.color, activebackground = self.color, foreground = self.color_text, activeforeground = self.color_text)
```

```
        self.button_write.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.label_genHelp.configure(bg = self.color, foreground = self.color_text)
        self.label_len.configure(bg = self.color, foreground = self.color_text)
        self.number_column.configure(bg = self.color, foreground = self.color_text)
        self.label_symbols.configure(bg = self.color, foreground = self.color_text)
        self.label_save.configure(bg = self.color, foreground = self.color_text)
        self.button_open.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)
        self.button_mail.configure(bg = self.color_menu, activebackground = self.color_active_bg, foreground = self.color_text, activeforeground = self.color_text)

    def dark_theme(self):
        self.root["bg"] ='#202020'

    def theme(self):
        if self.variable_theme.get() == 0:
            self.white_theme()
        elif self.variable_theme.get() == 1:
            self.dark_theme()
        elif self.variable_theme.get() == 2:
            self.lightdark_theme()

    def securityMemory (self,a, i, j,c):
        for h in range(i, len(c), 1):
            for g in range(j, len(c[i]), 1):
                if c[h][g] != 0 or c[h][g] != a:
                    return True
                else: return False
```

```
def method_square(self,a):
    b = a * a
    n = str(b)
    y = list(n)
    y = int(len(y))
    d = b // 10 ** (y - 2)
    f = b % 100
    a = d * 100 + f
    return a

def proverka(self, bit):
    bit = bit %62
    self.pas.append(string.printable[bit])

def proverkaIzSimvolami(self,bit):
    bit = bit % 94
    self.pas.append(string.printable[bit])

def send_mail(self):
    def send():
        mailget = self.write_mail.get()
        if len(mailget) != 0:
            server = smtplib.SMTP('smtp.gmail.com', 587)
            server.ehlo()
            server.starttls()
            server.ehlo()

            server.login('vladosik2108@gmail.com', 'fjkxelkareqyrjwh')
```

```
            subject = 'New password'
            body = f'Hello, thanks for contacting!\nYou have made a request in the program Prysma.\nYour new password: {self.abd}\n\nIf it wasn`t you, ignore this message.'
            message = f'Subject:{subject}\n\n{body}'

            server.sendmail(
                'vladosik2108@gmail.com',
                f'{mailget}',
                message
            )

            server.quit()
            a.after(1000, lambda: a.destroy())
        else:
            messagebox.showerror('Error', 'Write your e-mail')
            self.send_mail()

    a = Toplevel()
    a.geometry('495x300+760+390')
    a.configure(bg = self.color)
    self.write_mail = Entry(a,width = 32, font = ("roboto",18), background = self.color, foreground = self.color_text)
    self.button_send = Button(a,text = "Відправити", font = ("roboto",14), width = 15, height = 3,background = self.color, foreground = self.color_text, relief = GROOVE, command = send)
    self.write_mail.place(x = 47,y = 100)
    self.button_send.place(x = 160, y = 180)
```

```
def open_txt(self):
    os.startfile('password.txt')


def check_number(self, t1):
    try:
        t1 = int(t1)
    except:
        messagebox.showerror('Ошибка','Введите число!')
    return 5<= t1<=20, t1


def schet(self, *args):
    myTime = time.time()
    emptyMemory = np.empty((20, 2))
    c= copy.deepcopy(emptyMemory)
    c.tolist()
    if c[0,0] == c[1,0]:
        emptyMemory2 = np.empty((20, 2))
        c = copy.deepcopy(emptyMemory2)
        c.tolist()
    for i in range (0, len(c), 1):
        for j in range(len(c[i])):
            d = c[i][j]        #число которое нужно проверить
            indexTrue = self.securityMemory(d,i,j,c)
            if indexTrue == True:
                if c[i, j] >= 10000000000000000 or c[i, j] <= 0.00001:
                    d = Decimal(d)
                    b = str(c[i,j])
                    v1 = b.split('e')
                    poww = str(v1[1])
```

```
                if ord(poww[0]) == 43:
                    memori = (float(c[i,j])/10**int(poww[1:]))
                elif ord(poww[0]) == 45:
                    #print(type(c[i,0]))
                    memori = (float((d*10**int(poww[1:]))))
                self.arr.append(memori)

        square = self.arr[0]*10000
        square = round(square)

        for i in range(1,self.s+1,1):
            square = self.method_square(square)
            myTime = time.time()
            secund = myTime*10**5
            secund = round(secund)
            modul = (secund*round(self.arr[1])) % square
            maybe_time = myTime-round(myTime-1)
            self.text.insert(END, '*')
            if self.cvar.get() == 0:
                self.proverka(modul)
            else: self.proverkaIzSimvolami(modul)
            #time.sleep(maybe_time)
            myTime += maybe_time

        self.abd = self.p.join(self.pas)
        self.text.configure(state='readonly', readonlybackground = self.color, bg =
self.color, foreground = self.color_text)
        self.button_write.place(x = 285, y = 450)
        self.button_mail.place(x = 478, y = 450)
```

```
def clear(self):
    self.arr.clear()
    self.pas.clear()
    self.text.config(state='normal')
    self.text.delete(0, END)
    self.cvar1.set(0)
    self.s = self.number_column.get()
    Tit,self.s = self.check_number(self.s)
    if Tit == True:
        self.schet()
    else: messagebox.showerror('Ошибка','Ведите число от 6 до 20')

def see(self):
    if self.cvar1.get() == 1:
        self.text.config(state='normal')
        self.text.delete(0, END)
        self.text.insert(0, self.p.join(self.pas))
        self.text.config(state='readonly', readonlybackground = self.color, bg = self.color)
    else:
        self.text.config(state='normal', readonlybackground = self.color, bg = self.color)
        self.text.delete(0, END)
        self.text.insert(0, "*"*len(self.pas))
        self.text.config(state='readonly', readonlybackground = self.color, bg = self.color)

def writting_file(self):
    f = open('password.txt', 'w')
    for index in self.pas:
        f.write(str(index))
```

```
        f.close()
        self.button_write.place_forget()
        self.label_save.place(x = 310, y = 455)
        self.button_open.place(x = 305, y = 500)

    def __init__(self, root):
        self.x = 0
        self.arr = []
        self.pas = []
        self.p = ''
        self.root = root
        """Initialization variables for Radiobutton"""
        self.variable_theme = IntVar()
        self.variable_theme.set(0)
        """Initialization variables for Checkbutton"""
        self.cvar1 = BooleanVar()
        self.cvar1.set(0)
        self.cvar = BooleanVar()
        self.cvar.set(0)

        self.initUI()

    def initUI(self):
        but_style = ttk.Style()
        """Main Menu init"""

        but_style.configure("Menu.TButton", font = ("roboto", 12), background = "red")

        self.f_menu_gen = ttk.Frame(width = 180, height = 60)
```

self.f_menu_settings = ttk.Frame(width = 180, height = 60)

self.f_menu_help = ttk.Frame(width = 180, height = 60)

self.f_menu_exit = ttk.Frame(width = 180, height = 60)

self.f_menu_gen.pack_propagate(0)

self.f_menu_settings.pack_propagate(0)

self.f_menu_help.pack_propagate(0)

self.f_menu_exit.pack_propagate(0)

self.button_generate = Button(text = "Генерація паролів", font = ("roboto", 18),command = self.generation_for_one, width = 20, height = 2, relief = GROOVE, bd = 2)

self.button_settings = Button( text = "Налаштування", font = ("roboto", 18), command = self.settings, background = "#e1e1e1", width = 20, height = 2, activebackground = "#e5f1fb", relief = GROOVE, bd = 2)

self.button_help = Button( text = "Допомога", font = ("roboto", 18), command = self.help, background = "#e1e1e1", width = 20, height = 2, activebackground = "#e5f1fb", relief = GROOVE, bd = 2)

self.button_exit = Button( text = "Вихід", font = ("roboto", 18), command = exit, background = "#e1e1e1", width = 20, height = 2, activebackground = "#e5f1fb", relief = GROOVE, bd = 2)

"""Initialization Generator Window"""

self.button_generation = Button(text = 'Генерувати',width = 17, height = 2, font = ("SF UI",20, "bold"), relief = GROOVE, command = self.clear)

self.button_write = Button(text = 'Зберегти', width = 15, height = 2, command = self.writting_file,font = ("roboto",12), relief = GROOVE)

```
self.button_open = Button(text = 'Відкрити файл',width = 12, height = 3, command = self.open_txt,font = ("roboto",12), relief = GROOVE)
self.button_see = Checkbutton(text = 'Показати пароль',font = ("roboto", 12), command = self.see, variable=self.cvar1, onvalue=1, offvalue=0)
self.text = Entry(width = 23,  font = ("roboto",20), state = 'readonly')
self.number_column = Spinbox(width=20, from_=6, to=20, font = ("roboto", 12))
#self.number_column.set(6)
self.check_special_symbols = Checkbutton(text="Використовувати спеціальні символи", variable=self.cvar, onvalue=1, offvalue=0)
self.label_generation = Label(text = 'Генератор паролів', justify = CENTER, font=("Comic Sans MS", 24, "bold"))
self.label_genHelp = Label(text = 'Для генерації паролю заповніть форму нижче та натисніть кнопку "Генерувати"', font = ("roboto",))
self.label_len = Label(text = 'Довжина: ', font = ("roboto",14))
self.label_symbols = Label(text = 'Від 6 до 20 символів', font = ("roboto", 10))
self.label_save = Label(text = "Збережено!", font = "roboto 13")

"""Initialization Settings Window"""
self.label_settings = Label(text = "Оберіть тему:", justify = CENTER, font = ("roboto", 18))
self.theme_dark_radio = Radiobutton(indicatoron=0, height = 2, width = 12, background = '#202020', variable = self.variable_theme, value = 1, selectcolor ='#202020', command = self.theme)
self.theme_white_radio = Radiobutton(indicatoron=0, height = 2, width = 12, background = "#f0f0f0", variable = self.variable_theme, value = 0, selectcolor = "White", command = self.theme)
self.theme_lightdark_radio = Radiobutton(indicatoron=0, height = 2, width = 12, background = '#282c34', variable = self.variable_theme, value = 2, selectcolor ="#21252b", command = self.theme)
```

```
        self.label_dark_theme = Label(text = "Темна тема", font = ("roboto", 14))
        self.label_white_theme = Label(text = "Світла тема", font = ("roboto", 14))
        self.label_lightdark_theme = Label(text = "Синя тема", font = ("roboto", 14))


        """Initialization Button Back"""
        self.button_back = Button(text = '←', font = ("roboto", 14, "bold"), command = self.main_menu, height = 2, width = 4, bg = "#f0f0f0")
        #Label(text = 'Символи:', font = ("roboto", 14)).place(x = 10, y = 180)

        """Mail"""
        self.button_mail = Button( text = "Відправити\nна пошту", width = 15, height = 2, font = ("roboto",12), command = self.send_mail)

        """Start Programm menu"""
        self.white_theme()
        self.main_menu()

def main():
    root = Tk()
    root.geometry('900x600+510+240')
    #root.minsize(width = 800, height = 600)
    app = Password(root)
    #root.overrideredirect(1)
    root.title('Prysma')
    root.mainloop()

if __name__ == '__main__':
    main()
```