Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
в.о. завідувача кафедри
кібербезпеки та захисту інформації
_____ Іван ПАРХОМЕНКО
«____» червня 2023 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: Програмний застосунок для стиснення та шифрування інформації

Виконавець: студент IV курсу, групи КБ-42

_____ **Владислав БЄЛИХ**
(підпис) (ім'я, прізвище)

	Ім'я, прізвище	**Підпис**
Керівник	Микола БРАЇЛОВСЬКИЙ	

Нормоконтроль	Андрій БІГДАН	

Київ 2023

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
в.о. завідувача кафедри кібербезпеки
та захисту інформації
_____ Сергій ТОЛЮПА
«24» жовтня 2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студента **КБ-42** (група) **Бєлих Владислава Юрійовича** (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Програмний застосунок для стиснення та шифрування інформації

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №3 від 20.10.2022 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Програмний застосунок для стиснення та шифрування інформації.

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Дослідити існуючі алгоритми стиснення та шифрування, проаналізувати їх актуальність на даний момент та обрати ефективне рішення для реалізації власного застосунку

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність полягає у створенні програмного засобу для стиснення та забезпечення конфіденційності інформації.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 24 жовтня 2022 року

Завдання видав _____ Микола БРАІЛОВСЬКИЙ
(підпис) (ініціали, прізвище)

Завдання прийняв
до виконання _____ Владислав БЄЛИХ
(підпис) (ініціали, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів випускної кваліфікаційної роботи	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1.	Уточнення постановки задачі	21.10.2022 - 06.11.2022 р.	*виконано*
2.	Аналіз літератури	06.11.2022 - 23.12.2022 р.	*виконано*
3.	Обґрунтування вибору рішення	23.11.2022 - 07.12.2022 р.	*виконано*
4.	Збір даних	07.12.2022 - 21.01.2023 р.	*виконано*
5.	Вибір алгоритмів стиснення та шифрування	21.01.2023 - 15.02.2023 р.	*виконано*
6.	Розробка алгоритмів під вирішення задачі	15.02.2023 - 29.03.2023 р.	*виконано*
7.	Проведення аналізу отриманих результатів	29.03.2023 - 25.04.2023 р.	*виконано*
8.	Робота над висновками	25.04.2023 - 23.05.2023 р.	*виконано*
9.	Оформлення презентації	23.05.2023 - 30.05.2023 р.	*виконано*
10.	Оформлення пояснювальної записки	30.05.2023 - 03.06.2023 р.	*виконано*
11.	Підготовка до захисту дипломної роботи	03.06.2023 - 05.06.2023 р.	*виконано*

Завдання видав _____ Микола БРАІЛОВСЬКИЙ
(підпис) (ім’я, прізвище)

Завдання прийняв
до виконання _____ Владислав БЄЛИХ
(підпис) (ім’я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 12 червня 2023 р.

РЕФЕРАТ

Пояснювальна записка: 84 с., 28 рис., 1 табл., 6 додатки, 30 джерел.

Об’єкт дослідження: процес обробки даних що потребують стисненя та захист від несанкціонованого доступу в інформаційних системах

Мета роботи: розробка програмного засобу що забезпечує стиснення та шифрування інформації в інфрмаційних системах.

Предмет дослідження: методи стиснення та методи шифрування даних в інформаційних системах.

Методи дослідження: структурний аналіз, порівняння, системний підхід, індукція, формалізація, моделювання.

Практичне значення: створення застосунку для стиснення та шифрування даних в інформаційних системах

В роботі проведено аналіз симетричних алгоритмів шифрування та методів стиснення інформації. Побудовано та підібрано математичний апарат для виявлення шахрайств у сфері платіжних карток засобами машинного навчання.

Розроблено технічну реалізацію застосунку для стиснення та шифрування інформації.

Результати здійснених у дипломній роботі досліджень можуть бути використані для особистого користування при потребі стиснення та шифрування інформації, для більш точного підбору необхідних алгоритмів шифрування та стиснення.

Напрямки подальших досліджень можуть включати вдосконалення та додавання алгоритмів шифрування та стиснення інформації.

Ключові слова: шифрування, стиснення, blowfish, симетричне шифрування, hufman, конфіденційність, цілісність, інтерфейс, блокові шифри.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

RLE	-	run-length encoding
LZW	-	Lempel-Ziv-Welch
DES	-	Data Encryption Standard
AES	-	Advanced Encryption Standard
RSA	-	Rivest, Shamir and Adleman

ВСТУП

В сучасному світі, де збільшується кількість інформації та швидкість її передачі, виникає все більше потреба в ефективних та безпечних програмних засобах для захисту конфіденційності інформації. Тому тема "Програмний засіб для стиснення та забезпечення конфіденційності інформації" є актуальною та важливою в сучасному світі.

Метою даної дипломної роботи є розробка програмного засобу що забезпечує стиснення та шифрування інформації в інфрмаційних системах. Для досягнення цієї мети були сформульовані наступні завдання дослідження: дослідити алгоритм Хофмана для стиснення інформації, розробити програмний модуль для стиснення інформації з використанням алгоритму Хофмана, дослідити алгоритм Blowfish для шифрування інформації, розробити програмний модуль для шифрування інформації з використанням алгоритму Blowfish, та інтегрувати ці модулі в один програмний засіб.

Об'єктом дослідження є процес обробки даних що потребують стисненя та захист від несанкціонованого доступу в інформаційних системах, а предметом - методи стиснення та методи шифрування даних в інформаційних системах. Для дослідження використовувалися методи теоретичного аналізу, програмування та тестування.

Практичним значенням є створення застосунку для стиснення та шифрування даних в інформаційних системах.

Основною практичною цінністю є розробка програмного засобу, який поєднує в собі функції стиснення та шифрування інформації, що дозволяє ефективно зменшити розмір інформації та забезпечити її безпеку. Використання алгоритму Хофмана для стиснення дозволяє досягти високого ступеня ефективності за рахунок оптимального використання доступного простору для зберігання інформації. А використання алгоритму Blowfish для шифрування забезпечує високий рівень захисту від несанкціонованого доступу до інформації.

Працездатність та ефективність розробленого програмного засобу були перевірені на тестових даних, а також на реальних прикладах використання. Отримані результати свідчать про успішну реалізацію поставлених завдань та ефективність розробленого програмного засобу.

Отже, дана дипломна робота є важливим внеском у розвиток програмних засобів для захисту конфіденційності інформації. Розроблений програмний засіб може бути використаний в різних сферах, де важлива захист інформації, таких як фінансовий сектор, медична сфера, державні структури та інші.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ СТИСНЕННЯ ТА ШИФРУВАННЯ

1.1 Методи стиснення

Стиснення даних - це процес зменшення розміру даних без втрати інформації. Це досягається за рахунок видалення зайвої інформації, зменшення розміру символів та використання ефективних методів кодування. Загальна мета стиснення даних полягає у збереженні часу та простору для зберігання і передачі даних [1].

Стиснення даних є важливою складовою при розробці програмного засобу для захисту конфіденційності інформації [2]. Для захисту даних від несанкціонованого доступу часто використовують алгоритми стиснення та шифрування. Крім того, зменшення розміру даних допомагає зменшити час передачі та обробки даних, що забезпечує ефективну роботу системи [3].

Стиснення даних базується на використанні статистичних методів для виділення статистичних залежностей у даних. За допомогою цих методів можна знайти та вилучити зайві дані, що можуть бути використані для зменшення розміру даних [4].

Стиснення даних застосовується в багатьох галузях технологій, а саме:

1. Зберігання даних: стиснення даних допомагає зменшити розмір файлів і директорій, що забезпечує більш ефективне використання простору на диску [5].
2. Передача даних: при передачі даних через мережу стиснення даних допомагає зменшити обсяг передачі і забезпечити більш ефективне використання пропускної здатності мережі.
3. Компресія відео та аудіо: стиснення даних допомагає зменшити розмір відео- та аудіофайлів, що дозволяє зберігати більшу кількість файлів на диску і зменшити обсяг передачі при відтворенні відео та аудіо онлайн [6].
4. Обробка зображень: стиснення даних допомагає зменшити розмір зображень, що забезпечує більш ефективну обробку зображень на комп'ютері.

5. Компресія даних в базах даних: стиснення даних допомагає зменшити розмір баз даних, що забезпечує більш швидкий доступ до даних та зменшення використання ресурсів сервера.

6. Розпакування файлів: стиснення даних використовується для створення архівів, що зменшують розмір файлів і полегшують їх розпакування. Загалом, стиснення даних застосовується в багатьох галузях і технологіях, що дозволяє зберегти час і ресурси під час передачі, зберігання та обробки даних.

Одним з головних підходів до стиснення даних є використання алгоритмів кодування, таких як кодування довжиною слова, кодування Хаффмана, алгоритм LZ77, алгоритм RLE (Run-Length Encoding) та інші. Кожен з цих алгоритмів має свої переваги та недоліки, які залежать від типу даних, що необхідно стиснути.

Далі ми роздивимось деякі існуючі алгоритми стиснення.

1.1.1 Алгоритм стиснення Кодування довжин серій

Алгоритм стиснення даних Кодування довжин серій (RLE – Run-Length Encoding) є методом, який застосовується для зменшення обсягу даних шляхом заміни повторюваних символів у вихідному потоці на пари (символ, довжина серії). Таким чином, кількість символів у вихідному потоці зменшується.

Процес RLE можна описати таким чином: спочатку проходиться по вхідному потоці даних і визначається довжина кожної серії повторюваних символів. Потім кожну серію символів замінюють на пару (символ, довжина серії). Оскільки серії символів, які повторюються, можуть займати багато місця в даних, RLE може значно зменшити їх розмір.

Одним із головних плюсів RLE є його простота та швидкість стиснення. Також він добре підходить для стиснення даних з багатьма повторюваними символами, такими як зображення з монохромними пікселями або звукові файли з частими періодичними коливаннями. Крім того, RLE є дуже ефективним для стиснення текстових даних з багато повторюваними символами.

Однак, RLE має деякі мінуси. Якщо вхідні дані містять багато унікальних символів та мало повторень, то ефективність стиснення може бути не настільки значною. Крім того, RLE може призвести до збільшення розміру даних, якщо символи в них не повторюються.

Актуальність RLE полягає в його використанні в різних областях, де важливо зменшити розмір даних. Наприклад, він використовується для стиснення зображень, відео та звукових файлів, де повторення символів є досить поширеним.

1.1.2 Алгоритм стиснення Лемпеля Зіва Велча

Алгоритм стиснення Лемпеля Зіва Велча (LZW) є одним з найбільш поширених алгоритмів стиснення даних, який був запропонований в 1977 році. Він заснований на принципі заміни повторюваних фрагментів тексту на коди, що відповідають цим фрагментам, що дозволяє зменшити обсяг даних.

LZW є адаптивним алгоритмом, що означає, що він створює таблицю кодів під час стиснення даних, що містить вже побудовані кодові послідовності. Алгоритм працює на основі шукача, який проходить по вхідному потоку даних, знаходить повторювані фрагменти і замінює їх на коди, що відповідають цим фрагментам. Коди додаються в таблицю кодів для використання в майбутньому.

Один з головних переваг LZW – це його висока ефективність. Алгоритм зазвичай дозволяє досягти коефіцієнта стиснення на рівні 2:1 або більше для текстових даних, що містять повторювані фрагменти. Крім того, LZW може працювати з різними типами даних, такими як текст, зображення, аудіо та відео, що робить його універсальним методом стиснення.

Використання LZW може бути обмеженим в разі, якщо даних недостатньо для побудови таблиці кодів, що може призвести до збільшення розміру даних. Крім того, складність алгоритму LZW більша, ніж у RLE, що може призвести до повільного стиснення даних. Незважаючи на ці обмеження, LZW є дуже популярним методом стиснення даних і використовується в різних застосунках, таких як компресія зображень.

Актуальність використання алгоритму LZW визначається тим, що дані стають все більш об'ємними, що вимагає використання ефективних алгоритмів стиснення. Застосування алгоритму LZW може допомогти зменшити обсяг даних і, відповідно, збільшити швидкість передачі, що є дуже важливою умовою для ефективної обробки та передачі великих об'ємів даних.

1.1.3 Алгоритм стиснення Deflate

Алгоритм стиснення Deflate – це комбінований алгоритм стиснення, що включає в себе алгоритми стиснення Huffman та LZ77. Він широко використовується для стиснення даних в різних форматах файлів, таких як ZIP, GZIP, PNG, а також у компресорах файлових систем, таких як NTFS і ZFS.

Процес стиснення даних за алгоритмом Deflate складається з двох етапів. Спочатку виконується алгоритм LZ77, який знаходить повторювані послідовності даних і замінює їх посиланнями на попередні входження цих послідовностей. Наступним етапом є алгоритм стиснення Хаффмана, який використовує дерева Хаффмана для подальшого стиснення даних, зменшуючи кількість бітів, необхідних для зберігання кожного символу або комбінації символів.

Однією з основних переваг алгоритму Deflate є його ефективність при стисненні повторюваних послідовностей даних, а також його універсальність – він може бути використаний для стиснення різних типів файлів та даних.

Однак, існують певні недоліки алгоритму Deflate. Наприклад, він може бути менш ефективним для стиснення даних, що містять низьку ступінь повторення, і в цих випадках може бути краще використати інші алгоритми стиснення, такі як LZMA або Brotli. Крім того, стиснення даних за допомогою алгоритму Deflate може займати більше часу порівняно з іншими алгоритмами, особливо якщо вихідний файл має великий розмір. Також, як і інші алгоритми стиснення, він може призводити до збоїв та втрати даних під час передачі.

Актуальність алгоритму стиснення Deflate продовжує зростати, оскільки він є стандартом для багатьох файлових форматів, таких як ZIP, GZIP, PNG та інших. Крім

того, багато веб-сайтів використовують стиснення Deflate для компресії передачі даних через мережу, що допомагає зменшити розмір сторінок та підвищити швидкість завантаження веб-сайту. Зокрема, веб-сервери Apache та Nginx підтримують стиснення Deflate. Окрім цього, Deflate використовується у деяких програмах для стиснення даних, таких як програми архівації, збереження даних на диск, збереження файлів на сервері тощо. Таким чином, алгоритм стиснення Deflate є досить популярним і актуальним у багатьох галузях технологій.

1.1.4 Алгоритм стиснення Бротлі

Алгоритм стиснення Бротлі (Brotli) – це сучасний алгоритм стиснення даних, розроблений компанією Google, який вперше був випущений у 2015 році. Він є покращенням алгоритму Deflate, який забезпечує ще більшу стисливість даних.

Теорія алгоритму полягає в застосуванні комбінації алгоритмів стиснення змінної довжини (Huffman coding), кодування префіксів (prefix coding) та кешування вікон (sliding window). Алгоритм Brotli працює з бітовими послідовностями та забезпечує великий рівень стиснення даних.

Основні переваги алгоритму Brotli полягають у тому, що він забезпечує більш високу стисливість даних порівняно з алгоритмом Deflate, що дозволяє зменшити обсяг передачі даних та зберегти пропускну здатність мережі. Крім того, алгоритм Brotli працює швидше, ніж Deflate, і забезпечує кращу компресію текстових даних, зокрема HTML, CSS та JavaScript.

Однак, мінуси алгоритму Brotli пов'язані з тим, що він вимагає більше обчислювальних ресурсів порівняно з Deflate, що може збільшити час стиснення та розпакування даних на стороні сервера або клієнта.

Алгоритм Brotli широко використовується в сучасних веб-технологіях для стиснення текстових даних на сервері та підвищення швидкості завантаження веб-сторінок. Він підтримується в браузерах Google Chrome, Mozilla Firefox, Microsoft Edge та інших, і є рекомендованим алгоритмом стиснення для HTTP/2 та HTTP/3 протоколів, що дозволяє забезпечити більш ефективне використання пропускної

здатності мережі та зменшити час завантаження веб-сторінок для користувачів. Крім того, Brotli є безвтратним алгоритмом стиснення, що дозволяє зменшити обсяг передаваної інформації, зберігаючи при цьому всі дані в їхньому вихідному вигляді. Недоліком алгоритму може бути складність його реалізації та обробки, що може призвести до затримок під час стиснення/розпакування даних на сервері.

Актуальність алгоритму стиснення Бротлі також зростає в зв'язку зі збільшенням об'єму даних, що передаються через мережу Інтернет. Він широко використовується в сучасних веб-браузерах для стиснення веб-сторінок, що дозволяє значно зменшити час завантаження сторінок та зменшити обсяг переданих даних.

1.1.5 Алгоритм стиснення Хафманна

Алгоритм стиснення Хафманна – це ефективний метод стиснення даних, що базується на статистичному аналізі входящих символів і надає можливість зменшити об'єм даних, зберігаючи їхню інформаційну складність [7].

Теорія алгоритму полягає в тому, що кожен символ з вихідних даних кодується за допомогою бінарного коду, де менш часті символи отримують менші бітові коди, а більш часті символи - більші. Для цього використовується дерево Хафмана, яке формується на основі частоти входження кожного символу в дані.

Однією з головних переваг алгоритму стиснення Хафманна є висока ефективність стиснення текстових даних, зокрема даних, що містять багато повторюваних символів. Крім того, алгоритм дозволяє швидко стискати дані і забезпечує їхню точність під час розпакування.

Мінусом алгоритму може бути висока обчислювальна складність, що може затримати процес стискання для великих об'ємів даних. Також він має деякі обмеження на можливість ефективного стиснення даних з різними частотами входження символів.

Алгоритм стиснення Хафманна застосовується в багатьох сферах, включаючи зберігання та передачу даних, компресію зображень та звукових файлів, стиснення текстових файлів тощо.

Актуальність алгоритму стиснення Хафманна зростає з розвитком сучасних технологій, які вимагають ефективної передачі та зберігання великих об'ємів даних, зокрема в галузі Інтернету речей, хмарних технологій, машинного навчання та штучного інтелекту.

1.1.6 Порівняння алгоритмів стиснення

Усунення надлишку інформації – це процес перетворення даних в такий формат, який займає менше місця у пам'яті.

Алгоритм RLE, який базується на усуненні надлишку інформації, має мінімальну втрату даних, оскільки він не видаляє жодної інформації. Проте, ефективність стиснення залежить від характеру даних: він працює найкраще, коли вихідні дані містять багато повторюваних символів.

Алгоритм LZW також мінімізує втрати даних, оскільки він зберігає всі символи в словник. Але його ефективність стиснення може бути обмеженою, оскільки при стисненні великих даних може виникнути проблема з обмеженням пам'яті для зберігання словника.

Алгоритми Deflate та Бротлі зазвичай мають більш високу ефективність стиснення, але можуть втрачати певну кількість інформації під час стиснення. Це може призвести до втрати якості зображення або звуку при стисненні мультимедійних даних.

Алгоритм Хафмана також може втратити певну кількість інформації під час стиснення, оскільки він будує кодову таблицю на основі частоти вживання символів. Це може призвести до невеликої втрати якості зображення або звуку, але зазвичай його ефективність стиснення добре виправдовує ці втрати.

Також порівняння алгоритмів стиснення можна провести за такими критеріями:

1. Ефективність стиснення: у порівнянні з іншими алгоритмами, алгоритм Бротлі зазвичай забезпечує найкращий ступінь стиснення, дефлятор забезпечує добре стиснення взагалі, але його ефективність може залежати від даних, які стискаються. RLE надає добре стиснення тільки в тих випадках, коли дані мають багато

повторюваних символів. Алгоритми Лемпеля-Зіва-Велча та Хафманна зазвичай дозволяють досягнути середньої ефективності стиснення, але можуть бути менш ефективними, ніж Бротлі або Deflate.

2. Швидкодія: RLE є одним з найшвидших алгоритмів стиснення, що зазвичай працює майже миттєво. Лемпеля-Зіва-Велча може забрати деякий час для побудови словника, але після цього робота зазвичай відбувається швидко. Алгоритми Хафманна та Deflate можуть бути повільнішими на обробці великих даних, оскільки потребують більше обчислювальних ресурсів. Алгоритм Бротлі може бути повільним при стисненні даних, але може відчувати перевагу при роботі з великими потоками даних.

3. Використання пам'яті: алгоритми Хафманна та Лемпеля-Зіва-Велча зазвичай вимагають більше пам'яті для зберігання словників, тоді як Deflate та Бротлі зазвичай використовують менше пам'яті.

4. Рівень налаштування: Лемпеля-Зіва-Велча та Хафманна можуть бути налаштовані на різні рівні стиснення, що дозволяє досягнути балансу між ефективністю стиснення та швидкодією. У порівнянні з цим, RLE та Deflate мають менші можливості для налаштування рівня стиснення. Бротлі має високий рівень налаштування, що дозволяє досягнути найкращих результатів стиснення, але при цьому може займати більше обчислювальних ресурсів.

Вибір алгоритму стиснення залежить від конкретних потреб користувача та характеристик даних, які необхідно стиснути. Наприклад, якщо даними є тексти з багато повторюваними словами, то Лемпеля-Зіва-Велча може забезпечити хороші результати. У випадку роботи з великими потоками даних, де швидкодія є ключовим фактором, може бути вигідніше використовувати RLE.

Використання різних алгоритмів стиснення в залежності від потреби може допомогти досягти кращих результатів стиснення.

Коротке порівняння алгоритмів стиснення за розміром ключа, розміром блоку даних, кількістю раундів, швидкістю та стійкістю алгоритма, наведено у таблиці (див. додаток Е).

1.2 Методи шифрування

Шифрування – це процес перетворення тексту (відкритий текст) у такий формат, який незрозумілий для неавторизованих осіб (закритий текст або шифротекст) [8]. Це забезпечує конфіденційність інформації, оскільки без відповідного ключа шифрування важко або неможливо розкрити секретну інформацію [9].

З основних методів шифрування можна відокремити:

1. Симетричне шифрування: використовує один і той же ключ для шифрування і розшифрування даних. Два найпоширеніші алгоритми симетричного шифрування – AES (Advanced Encryption Standard) і DES (Data Encryption Standard).
2. Асиметричне шифрування: використовує пару ключів – публічний і приватний. Публічний ключ використовується для шифрування даних, а приватний ключ – для розшифрування. RSA (Rivest-Shamir-Adleman) і ECC (Elliptic Curve Cryptography) є одними з найпоширеніших алгоритмів асиметричного шифрування [10].

Методи шифрування використовуються для захисту конфіденційності даних, забезпечення цілісності інформації та забезпечення аутентифікації в різних сферах, включаючи комунікацію в Інтернеті, зберігання даних, фінансові транзакції та багато іншого [11].

1.2.1 Симетричне шифрування

Симетричне шифрування - це метод криптографічного шифрування, в якому використовується один і той же ключ для шифрування та розшифрування даних [12]. Розглянемо плюси та мінуси симетричного шифрування:

Плюси симетричного шифрування:

1. Швидкодія: симетричні алгоритми шифрування зазвичай працюють швидко та ефективно. Вони придатні для обробки великих обсягів даних та вимогливих застосунків, де швидкодія має велике значення.

2. Простота реалізації: симетричні алгоритми шифрування мають просту структуру та легко реалізовуються в програмних або апаратних засобах. Це робить їх доступними та широко використовуваними в багатьох системах.

3. Ефективність: симетричне шифрування зазвичай забезпечує високий рівень захисту даних. При належному виборі алгоритму та достатньо довгому ключі можна забезпечити великий ступінь конфіденційності даних.

Мінуси симетричного шифрування:

1. Розподіл ключів: один з найбільших недоліків симетричного шифрування полягає в необхідності безпечного обміну секретним ключем між відправником і отримувачем. Це може становити проблему, особливо при обміні ключами у великій мережі або при використанні веб-сервісів.

2. Кількість ключів: для безпеки кожна пара комунікуючих сторін повинна мати унікальний секретний ключ. Якщо кількість сторін збільшується, кількість необхідних ключів зростає квадратично. Це може бути проблематично при розгортанні та управлінні багатокористувацькими системами.

3. Вразливість до атак: симетричне шифрування може бути вразливим до певних криптоаналітичних атак. Наприклад, атаки перебором ключа або з використанням статистичних методів можуть викрити секретний ключ та зламати шифр. Тому вибір відповідного алгоритму та дотримання рекомендацій щодо безпеки мають велике значення.

1.2.1.1 Data Encryption Standard

Алгоритм шифрування Data Encryption Standard (DES) є одним з найвідоміших та широко використовуваних симетричних шифрів [13]. Давайте розглянемо основні аспекти цього алгоритму:

DES базується на блоковому шифруванні, де дані розбиваються на блоки фіксованої довжини (64 біта) та шифруються незалежно один від одного [14]. Ключ шифрування також має довжину 64 біти, але фактично використовуєтьсяише 56 біт, оскільки 8 бітів використовуються для контролю парності.

Процес шифрування DES складається з 16 раундів, де використовуються різні операції, такі як зсуви, підстановки та перестановки бітів, що забезпечують високий рівень конфіденційності даних.

Плюси DES:

1. Відомий та встановлений: DES є одним з найбільш відомих та досліджених алгоритмів шифрування. Його безпеку вивчали та перевіряли протягом багатьох років, що дає впевненість у його надійності.

2. Широке використання: DES був широко використовуваний у різних сферах, включаючи фінансові установи, комунікаційні системи та забезпечення конфіденційності даних.

Мінуси DES:

1. Коротка довжина ключа: одним з основних недоліків DES є його коротка довжина ключа (56 біт), що може бути вразливим до атак перебору ключа. За сучасними стандартами, такий ключ є недостатньо безпечним.

2. Застарілість: DES був розроблений у 1970-х роках, і з тих пір криптографічні атаки та обчислювальні можливості значно змінилися. Це призводить до зменшення його актуальності та безпеки.

Використання:

DES раніше широко використовувався для захисту конфіденційної інформації, але зараз його використання є обмеженим через його низьку безпеку. У більшості випадків, DES був замінений більш сильними алгоритмами шифрування, такими як AES (Advanced Encryption Standard).

Актуальність:

На сьогоднішній день DES вважається застарілим та недостатньо безпечним алгоритмом. Його використання не рекомендується для захисту конфіденційної інформації через вразливість до атак перебору ключа та обмежену довжину ключа.

Замість DES, рекомендується використовувати більш сучасні і стійкі алгоритми шифрування, такі як AES.

1.2.1.2 Advanced Encryption Standard

Алгоритм шифрування Advanced Encryption Standard (AES) є одним з найпоширеніших та надійних симетричних алгоритмів шифрування [15]. Основні аспекти AES такі:

AES базується на блоковому шифруванні, де дані розбиваються на блоки фіксованої довжини (128 біт), які шифруються незалежно один від одного. Ключ шифрування може мати довжину 128, 192 або 256 біт, залежно від варіанту AES, що використовується [16].

AES складається з декількох раундів, де використовуються операції, такі як заміни байтів, зсуви, змішування колонок та додавання ключа [17]. Ці операції забезпечують високий рівень безпеки та конфіденційності даних [18].

Плюси AES:

1. Висока безпека: AES вважається одним з найбільш безпечних алгоритмів шифрування. Він витримав багато років аналізу та криптоаналізу, і його безпеку підтверджують провідні криптографічні експерти.

2. Широке використання: AES широко застосовується в різних сферах, включаючи фінансові установи, комунікаційні системи, безпеку мереж та забезпечення конфіденційності даних.

3. Швидкодія: AES має високу швидкодію, що дозволяє ефективно шифрувати та розшифровувати дані в реальному часі.

Мінуси AES:

1. Обмежена довжина блоку: AES працює з фіксованими блоками даних довжиною 128 біт. Це може створювати проблеми при шифруванні довгих повідомлень, оскільки потрібно використовувати режими шифрування для обробки довшої інформації.

2. Потребує більше ресурсів: шифрування та розшифрування AES вимагають більше обчислювальних ресурсів порівняно з іншими шифрами, особливо при використанні більших ключів.

3. Симетричне шифрування: AES є симетричним алгоритмом, що означає, що той самий ключ використовується як для шифрування, так і для розшифрування. Це може створювати проблеми з обміном ключами та безпекою передачі ключа.

Використання:

AES широко використовується для захисту конфіденційної інформації в різних сферах, таких як фінанси, комунікації, електронна пошта, безпека мереж, зберігання даних та багато інших. Він підтримується більшістю криптографічних бібліотек та програмного забезпечення.

Актуальність:

AES є актуальним алгоритмом шифрування, оскільки він забезпечує високий рівень безпеки та ефективності. Він використовується у багатьох сучасних застосунках та протоколах, таких як SSL/TLS, VPN, Wi-Fi забезпечення, дискретне шифрування та інші. AES продовжує бути рекомендованим алгоритмом для захисту конфіденційної інформації в різних сферах [19].

1.2.1.3 Serpent

Алгоритм шифрування Serpent є симетричним блоковим шифром, який був розроблений в рамках проекту Advanced Encryption Standard (AES). Основні аспекти алгоритму Serpent такі:

Serpent використовує 32 раунди замін та перестановок для шифрування даних. Кожен блок даних (зазвичай 128 біт) розбивається на слова, які пропускаються через послідовність раундів. Ключ шифрування також має довжину 128, 192 або 256 біт, залежно від варіанту Serpent, який використовується.

Кожен раунд Serpent складається з нескольких етапів, включаючи змішування байтів, заміни, перестановки та операції додавання ключа. Ці етапи виконуються повторно для кожного слова в блоку даних, що забезпечує високий рівень безпеки та конфіденційності.

Плюси Serpent:

1. Висока безпека: serpent є одним з найбільш безпечних алгоритмів шифрування. Він пройшов велику кількість аналізу та криптоаналізу, що підтверджує його стійкість до різних атак.

2. Довжина ключа: serpent підтримує довжину ключа до 256 біт, що забезпечує високий рівень безпеки та можливість використання довгих ключів для забезпечення конфіденційності.

3. Ефективність на різних платформах: serpent має хорошу переносимість та працездатність на різних апаратних та програмних платформах, включаючи мобільні пристрої та вбудовані системи.

Мінуси Serpent:

Швидкодія: у порівнянні з іншими алгоритмами шифрування, Serpent може бути трохи повільнішим на обробці даних через велику кількість раундів, що потрібно виконати.

Використання:

Serpent широко використовується в різних застосунках, які вимагають високої безпеки, таких як захист конфіденційної інформації, криптографічні протоколи, електронний банкінг, захист даних на дисках та багато інших. Він є одним з рекомендованих алгоритмів шифрування та продовжує бути активно використовуваним в сучасних системах безпеки.

Актуальність:

Алгоритм шифрування Serpent залишається актуальним, оскільки він забезпечує високий рівень безпеки та стійкості до атак. Він продовжує використовуватися в сучасних криптографічних системах та стандартах, зокрема у складі криптографічних пристроїв, програмного забезпечення та мережевих протоколів.

1.2.1.4 Blowfish

Алгоритм шифрування Blowfish є симетричним блоковим шифром, розробленим Брюсом Шнайером у 1993 році [20; 21].

Основні аспекти алгоритму Blowfish такі:

Blowfish використовує 64-бітні блоки даних та змінну довжину ключа від 32 до 448 біт [22]. Шифрування проводиться через серію раундів, кожен з яких містить чотири основні етапи: заміну, перестановку, операції додавання ключа та розширення ключа [23].

Blowfish використовує таблиці замін, які розміщуються в ініціалізованому стані алгоритму. Під час шифрування блоки даних проходять крізь ці таблиці, де виконується заміна бітів та перестановка. Кожен раунд включає підстановки, що залежать від ключа, та операції змішування [24; 25].

Плюси Blowfish:

1. Широке використання: blowfish широко використовується в різних застосунках, таких як захист даних, веб-протоколи, віртуальні приватні мережі (VPN) та інше. Він став популярним у сфері електронної комерції та забезпечує надійне шифрування передачі даних [26].

2. Швидкодія: blowfish відомий своєю високою швидкодією на багатьох платформах, включаючи обчислювальні системи з обмеженими ресурсами, такі як мобільні пристрої.

3. Гнучкість ключа: blowfish дозволяє використовувати ключі з різною довжиною, що надає більшу гнучкість та безпеку при виборі ключа.

Мінуси Blowfish:

1. Довжина блоку: розмір блоку даних у Blowfish становить 64 біти, що може обмежувати ефективність алгоритму при роботі з великими об'ємами даних.

2. Стійкість до криптоаналітичних атак: деякі нові атаки та криптоаналітичні методи можуть впливати на стійкість Blowfish, особливо при використанні ключів недостатньої довжини.

Використання:

Blowfish застосовується в різних системах, де потрібне шифрування даних, включаючи захист інформації, комунікації через мережі, електронну комерцію, зберігання даних тощо. Його можна зустріти в програмному забезпеченні, пристроях, серверах та різних криптографічних протоколах .

Актуальність:

Хоча алгоритм Blowfish є старішим, він все ще залишається актуальним і використовується в деяких застосунках. Проте через зростання обчислювальної потужності та появу нових криптографічних алгоритмів, таких як AES, які забезпечують більшу безпеку, у багатьох випадках він замінюється на більш сучасні алгоритми.

1.2.2 Асиметричне шифрування

Асиметричне шифрування, також відоме як шифрування з відкритим ключем, є криптографічним методом, в якому використовується пара ключів: публічний ключ для шифрування і приватний ключ для розшифрування. Основні аспекти асиметричного шифрування такі:

Плюси асиметричного шифрування:

1. Безпека: одним з найважливіших переваг асиметричного шифрування є його висока безпека. Приватний ключ, який потрібен для розшифрування даних, є відомим тільки власнику, що робить важким отримання доступу до розшифрованих даних незаконними особами.

2. Передача ключів: асиметричне шифрування дозволяє безпечно передавати симетричні ключі для використання в симетричних алгоритмах шифрування. Це дозволяє ефективно забезпечувати конфіденційність даних і обмінюватися інформацією без необхідності передавати секретні ключі по відкритому каналу.

3. Цифровий підпис: асиметричне шифрування використовується для створення цифрових підписів, що дозволяє підтверджувати автентичність даних та ідентифікацію відправника. Це важливий механізм для забезпечення цілісності і неденійності даних.

Мінуси асиметричного шифрування:

1. Висока обчислювальна складність: асиметричне шифрування вимагає більшої обчислювальної потужності порівняно з симетричним шифруванням.

Операції шифрування та розшифрування з використанням асиметричних ключів можуть бути повільнішими та вимагати більше ресурсів.

2. Потреба в безпечному зберіганні приватного ключа: використання асиметричного шифрування передбачає, що приватний ключ залишається конфіденційним і недоступним для несанкціонованого доступу. Потреба у безпечному зберіганні ключа може бути складною задачею.

3. Витрати на обчислювальні ресурси: використання асиметричного шифрування може вимагати значних обчислювальних ресурсів, особливо при обробці великого обсягу даних або у вимогливих застосунках.

4. Залежність від інфраструктури ключів: асиметричне шифрування потребує наявності надійної інфраструктури ключів для генерації, обміну та зберігання ключів. Недостатність інфраструктури ключів може ускладнити використання асиметричного шифрування.

Асиметричне шифрування використовується в різних областях, таких як захист даних, електронна комерція, цифровий підпис, безпека мереж і комунікаційних протоколів. В сучасному світі асиметричне шифрування є невід'ємною частиною криптографічних систем і забезпечує надійний рівень безпеки для захисту конфіденційності та цілісності інформації.

1.2.2.1 Rivest-Shamir-Adleman

Алгоритм шифрування RSA (Rivest-Shamir-Adleman) є одним з найвідоміших і поширених алгоритмів асиметричного шифрування [27]. Він отримав свою назву на честь його авторів - Рональда Рівеста, Аді Шаміра та Леонарда Адлемана. Основні аспекти алгоритму RSA такі:

RSA базується на складності факторизації великих цілих чисел. Алгоритм використовує математичну пару ключів: публічний ключ для шифрування і приватний ключ для розшифрування. Публічний ключ розповсюджується іншим користувачам, які можуть використовувати його для шифрування повідомлень, тоді як приватний ключ залишається виключно у власника для розшифрування.

Плюси RSA:

1. Безпека: RSA вважається безпечним алгоритмом шифрування, оскільки його безпеку базується на складності факторизації великих чисел. Відновлення приватного ключа з публічного ключа є обчислювально нездійсненною задачею за розумного часу, якщо великі прості числа правильно обрані.

2. Асиметричне шифрування та цифровий підпис: RSA може використовуватись для шифрування та розшифрування повідомлень, а також для створення цифрових підписів. Це дозволяє захищати конфіденційність даних та підтверджувати автентичність відправника.

3. Використання у криптографічних протоколах: RSA широко використовується у криптографічних протоколах, таких як SSL/TLS для захисту передачі даних через мережу. Він забезпечує безпеку при обміні симетричними ключами, цифрових сертифікатів та інших криптографічних параметрів.

Мінуси RSA:

1. Обчислювальна складність: RSA вимагає значних обчислювальних ресурсів, особливо при роботі з великими числами. Генерація ключів, шифрування та розшифрування повідомлень можуть бути часо- та ресурсозатратними операціями, особливо при використанні довгих ключів.

2. Вразливість до атак: RSA вразливий до криптоаналітичних атак, таких як факторизація великих чисел або атака на основі китайської теореми про залишки. Якщо прості числа, використовувані для генерації ключів, недостатньо великі або вибрані неналежним чином, алгоритм може стати вразливим.

Використання та актуальність:

RSA є одним з найпоширеніших алгоритмів шифрування і використовується в багатьох сферах, включаючи захист даних, електронну комерцію, цифровий підпис, безпеку мереж та комунікаційних протоколів. Незважаючи на те, що RSA є вже старіючим алгоритмом, він все ще залишається актуальним та надійним, особливо при використанні довгих ключів. Проте, в зв'язку з появою більш потужних обчислювальних ресурсів, сучасні криптографічні системи все більше переходять на

використання алгоритмів з більшою довжиною ключа, таких як RSA з ключами довжиною 2048 біт та більше.

1.2.3 Порівняння алгоритмів шифрування

Розглянуті алгоритми використовуються вже багато років, маюсь плюси та мінуси, надалі ми порівняємо ці алгоритми шифрування за рівнем безпеки, швидкодією, використанням та актуальністю, щоб обрати підходящий для наших потреб [28]. Порівняємо алгоритми шифрування DES, AES, Serpent, Blowfish і RSA за деякими ключовими аспектами:

1. Рівень безпеки:

DES: DES (Data Encryption Standard) був стандартом шифрування у минулому, але його використання сьогодні не рекомендується, оскільки його ключова довжина (56 біт) вважається недостатньою для сучасних безпечних застосувань. DES сьогодні вважається застарілим з точки зору безпеки.

AES: AES (Advanced Encryption Standard) є широко використовуваним алгоритмом шифрування і вважається стандартом безпеки. Він пропонує різні ключові довжини (128, 192 або 256 біт) і забезпечує високий рівень безпеки. AES є сучасним і надійним алгоритмом шифрування.

Serpent: Serpent є симетричним алгоритмом шифрування, який також відомий своєю високою безпекою. Він використовує 128-бітні блоки і ключі з довжиною від 128 до 256 біт. Serpent був одним з фіналістів у конкурсі AES, і його безпека вважається дуже сильною.

Blowfish: Blowfish є симетричним алгоритмом шифрування, який пропонує добру безпеку. Він використовує змінну довжину ключа від 32 до 448 біт. Хоча Blowfish залишається широко використовуваним, вважається, що AES надає більшу безпеку.

RSA: RSA є асиметричним алгоритмом шифрування, відомим своєю сильною безпекою у відношенні до криптографічних задач, таких як обмін ключами та цифровий підпис. Він базується на складності факторизації великих простих чисел.

RSA є широко використовуваним алгоритмом для безпечного обміну даними і забезпечує високий рівень безпеки.

2. Швидкодія:

DES: DES є відносно швидким алгоритмом, оскільки його операції базуються на простих бітових операціях. Однак, в порівнянні з AES, DES вважається повільним.

AES: AES є ефективним і швидким алгоритмом шифрування, оскільки його структура добре пристосована для апаратної реалізації і оптимізації.

Serpent: Serpent зазвичай працює повільніше, ніж AES, оскільки його операції вимагають більше обчислювальних ресурсів.

Blowfish: Blowfish є відносно швидким алгоритмом, особливо при використанні оптимізованих реалізацій.

RSA: RSA є повільним алгоритмом порівняно з симетричними алгоритмами, особливо при роботі з великими обсягами даних. Його швидкодія залежить від довжини ключа.

3. Використання і актуальність:

DES: DES використовувався в минулому, але його застосування зараз обмежене через недостатню довжину ключа.

AES: AES є стандартом безпеки і широко використовується в різних сферах, включаючи комунікації, зберігання даних та криптографічні протоколи. Він залишається актуальним і рекомендованим вибором для багатьох застосувань.

Serpent: Serpent використовується в обмежених випадках, особливо там, де висока безпека є пріоритетом, а швидкодія менш важлива.

Blowfish: Blowfish залишається широко використовуваним алгоритмом, хоча його використання дещо зменшилося на користь AES. Він залишається актуальним в деяких областях, де потрібна добра безпека при швидкому шифруванні.

RSA: RSA залишається основним алгоритмом для асиметричного шифрування, цифрового підпису та обміну ключами. Він широко використовується у багатьох криптографічних протоколах і застосунках, особливо в сферах, пов'язаних з безпекою із відкритим ключем.

Порівняння алгоритмів шифрування наведено у таблиці (див. додаток Ж)

1.3 Обґрунтування вибору алгоритмів стиснення та шифрування для реалізації програмного застосунку

Вибір алгоритмів стиснення та шифрування для програмного застосунку залежить від конкретних вимог та вимог безпеки, а також від характеристик даних, з якими ви будете працювати. Проте, можна надати загальне обґрунтування вибору алгоритмів Хафмана для стиснення та Blowfish для шифрування.

Алгоритм Хафмана для стиснення:

1. Ефективність стиснення: алгоритм Хафмана відомий своєю високою ефективністю стиснення, особливо для текстових даних з повторюваними символами або шаблонами. Він добре працює на різноманітних типах даних та має низький рівень втрати інформації.

2. Швидкодія: алгоритм Хафмана є швидким і ефективним у режимі стиснення. Він не вимагає великих обчислювальних ресурсів і зазвичай працює досить швидко.

3. Використання пам'яті: алгоритм Хафмана вимагає деякої додаткової пам'яті для побудови таблиці кодів. Однак, споживання пам'яті зазвичай є прийнятним і залежить від розміру словника.

Алгоритм Blowfish для шифрування:

1. Безпека: blowfish є симетричним алгоритмом шифрування, який забезпечує високий рівень безпеки. Він вважається стійким до багатьох криптоаналітичних атак і має широке застосування в промисловості.

2. Швидкодія: blowfish є досить швидким алгоритмом шифрування, що дозволяє ефективно обробляти дані навіть при великому обсязі. Він підходить для шифрування великих файлів та потоків даних.

3. Гнучкість: blowfish підтримує різні довжини ключів, що дозволяє вам вибрати необхідний рівень безпеки. Ви можете використовувати ключі від 32 до 448 біт, що забезпечує великий простір ключів для захисту даних.

Обгрунтовуючи вибір алгоритмів Хафмана для стиснення та Blowfish для шифрування, можна сказати, що ці алгоритми поєднують в собі високу ефективність,

швидкодію та безпеку. Хафман надає хорошу стискальну ефективність з низьким рівнем втрати інформації, тоді як Blowfish забезпечує надійний рівень шифрування з гнучкістю вибору ключів. Загалом, цей вибір є розумним для багатьох сценаріїв, де потрібно забезпечити ефективне стиснення та надійне шифрування даних.

Використання алгоритму Хафмана для стиснення та алгоритму Blowfish для шифрування може бути гарним варіантом для стиснення та захисту сертифікатів та відкритих ключів. Ось чому:

1. Стиснення сертифікатів: сертифікати можуть мати великий обсяг даних, особливо якщо це сертифікати з високою розширеністю або з багатьма полями. Використання алгоритму Хафмана дозволить зменшити розмір сертифікатів, зберігаючи при цьому їхню інтегритет та цілісність. Зменшений розмір сертифікатів полегшує їхнє зберігання та передачу.

2. Шифрування відкритих ключів: відкриті ключі є важливою складовою криптографічних систем. Використання алгоритму Blowfish для їхнього шифрування забезпечує безпеку та конфіденційність цих ключів. Blowfish є симетричним алгоритмом шифрування, який забезпечує стійкість та високу безпеку даних.

3. Безпека та надійність: шифрування сертифікатів та відкритих ключів є важливим для захисту конфіденційної інформації та запобігання несанкціонованому доступу до неї. Алгоритм Blowfish відомий своєю стійкістю до атак і є одним з найпоширеніших симетричних алгоритмів шифрування. Крім того, використання стиснення за допомогою алгоритму Хафмана не порушує безпеку даних і допомагає економити пропускну здатність та ресурси при передачі сертифікатів та відкритих ключів.

4. Актуальність: алгоритм Хафмана та алгоритм Blowfish є добре встановленими та використовуваними алгоритмами, які показали свою ефективність та безпеку протягом багатьох років. Вони є широко визнаними в галузі криптографії та застосовуються в багатьох системах та протоколах.

В цілому, вибір алгоритму Хафмана для стиснення та алгоритму Blowfish для шифрування є розумним, оскільки вони комбінують ефективність, безпеку та широку

підтримку, що є важливими факторами для програмного застосунку, який працює з сертифікатами та відкритими ключами.

1.4 Висновки по першому розділу

У цьому розділі, ми розглянули різні алгоритми шифпування та стиснення та їхні особливості. Проаналізувавши їх, ми прийшли до висновку, що алгоритм Хафмана є оптимальним для стиснення даних, а алгоритм Blowfish - для шифрування.

Алгоритм Хафмана відмінно справляється зі стисненням даних, забезпечуючи ефективність та зменшення обсягу даних без втрати інформації. Він особливо ефективний у випадках, коли дані мають повторювані шаблони або символи. Використання Хафмана дозволяє зберегти цілісність даних та економити пропускну здатність при передачі.

Алгоритм Blowfish є надійним та широко використовуваним алгоритмом шифрування. Він забезпечує конфіденційність та безпеку даних, дозволяючи зашифровувати та розшифровувати інформацію за допомогою одного ключа. Blowfish має високий рівень безпеки та стійкості до атак, що робить його привабливим варіантом для захисту конфіденційних даних, таких як сертифікати та відкриті ключі.

Обрані алгоритми – Хафмана для стиснення та Blowfish для шифрування – добре поєднуються між собою, забезпечуючи ефективність, безпеку та надійність для програмного застосунку. Вони мають широку підтримку та визнані в галузі криптографії. Такий вибір дозволяє досягти економії ресурсів, забезпечити конфіденційність даних та зберегти їхню цілісність.

Загалом, вибір алгоритмів Хафмана та Blowfish є розумним та обґрунтованим для реалізації програмного застосунку, що працює з сертифікатами та відкритими ключами.

РОЗДІЛ 2
РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ

2.1 Підготовка вхідних даних

Був реалізован програмний застосунок для стиснення та шифрування інформації за алгоритмами Хафмана та Blowfish відповідно [29]. Надалі будуть наведені скріншоти із поясненнями.

Створюємо файл для сбереження внутрішніх даних та ключів data.py

Ініціалізуємо ключи шифрування (рисунок 2.1):

```
PI_P_ARRAY = (
    0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0,
    0x082efa98, 0xec4e6c89, 0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
    0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917, 0x9216d5d9, 0x8979fb1b,
)
```

Рисунок 2.1 – Запис ключів шифрування

Ініціалізуємо 32-бітні таблиці замін (рисунок 2.2):

```
# 4 x 256
PI_S_BOXES = (
    (
        0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96,
        0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
        0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
        0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
        0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
        0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
        0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
```

Рисунок 2.2 – Початок таблиці замін

Кінець 32-бітної таблиць замін та ініціалізація секретного ключа та вектору ініціалізації (рисунок 2.3):

```
            0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e, 0x1948c25c, 0x02fb8a8c,
            0x01c36ae4, 0xd6ebe1f9, 0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,
            0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6,
        ),
    )

    SPECIAL_PRIVATE_KEY = b"specialkey"
    INITVECTOR = os.urandom(8)
```

Рисунок 2.3 – Кінець таблиці замін, ініціалізація приватного ключа та вектору

2.2 Реалізація алогоритму Blowfish

Для реалізація алгоритму Blowfish створюємо основний файл blowfish.py, де буде знаходитися весь код:

Імпортування та створення класу шифру (рисунок 2.4):

```
from struct import Struct, error as struct_error
from itertools import cycle as iter_cycle
from encryption.data import PI_P_ARRAY, PI_S_BOXES, SPECIAL_PRIVATE_KEY


class BlowfishCipher(object):
    """
    Blowfish block cipher.
```

Рисунок 2.4 – Імпорт бібліотек для роботи із структурами даних

Клас BlowfishCipher реалізує шифр Blowfish. Він надає функції для шифрування та дешифрування даних із використанням режиму зворотного зв'язку (CFB) для роботи з даними довільної довжини.

__init__(self, key, byteOrder="big", P_Array=PI_P_ARRAY, S_Boxes=PI_S_BOXES) – конструктор класу, ініціалізує об'єкт BlowfishCipher. Приймає ключ (key), порядок байтів (byteOrder), масив P-значень (P_Array) та S-блоків (S_Boxes) як параметри.

_encrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack) – допоміжна статична функція, яка використовується для шифрування блоку даних. Приймає ліву та праву

частини блоку (L та R), масив P-значень (P), S-блоки (S1, S2, S3, S4), пакувальник та розпакувальник для перетворення даних (рисунок 2.5).

```
123     @staticmethod
124     def _encrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack):
125         for p1, p2 in P[:-1]:
126             L ^= p1
127             a, b, c, d = u1_4Unpack(u4_1Pack(L))
128             R ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
129             R ^= p2
130             a, b, c, d = u1_4Unpack(u4_1Pack(R))
131             L ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
132         p_penultimate, p_last = P[-1]
133         return R ^ p_last, L ^ p_penultimate
```

Рисунок 2.5 – Приватний метод _encrypt

for p1, p2 in P[:-1] – цей цикл повторює всі, крім останньої, пари значень ключів у P, яка є послідовністю кортежів (p1, p2), де кожна пара є 128-бітним значенням ключа, що використовується для шифрування та дешифрування даних.

L ^= p1 – цей рядок використовує оператор XOR для об’єднання даних, представлених L з p1, щоб зашифрувати його.

a, b, c, d = u1_4Unpack(u4_1Pack(L)) – цей рядок розгортає 32-розрядні цілі значення, представлені Л використовуючи u4_1Pack допоміжна функція. Потім він розпаковує цілі значення за допомогою u1_4Unpack допоміжну функцію в список із чотирьох 32-розрядних цілих значень a , b , в , і d , які використовуються для індексування значень у S1, S2, S3 та S4. S1, S2, S3 і S4 – це таблиці значень, які використовуються для виконання певних обчислень під час процесу шифрування.

R ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff – виконання операції, яка називається побітовим XOR. Він виконує XOR значення "R" з результатом виконання обчислення "(S1[a] + S2[b] ^ S3[c]) + S4[d]", а потім приймає лише 16 молодших бітів (за допомогою оператор & із шістнадцятковим значенням 0xffffffff).

R ^= p2 – у цьому рядку використовується оператор XOR, щоб поєднати вихідні дані попереднього рядка з `p2`, щоб завершити шифрування `R`.

a, b, c, d = u1_4Unpack(u4_1Pack(R)) – цей рядок розгортає 32-розрядні цілі значення, представлені `R` за допомогою допоміжної функції ‘u4_1Pack’, і розпаковує їх за допомогою допоміжної функції `u1_4Unpack` у список чотирьох 32-розрядних цілих значень ‘a’, ’b’, ‘c’ і ‘d’, які використовуються для індексування значень у S1, S2, S3 та S4.

L ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff – цей рядок виконує те саме обчислення, що й попередній рядок, але з `L` замість `R`. Потім він об’єднує результат із «L» за допомогою оператора XOR.

p_penultimate, p_last = P[-1] – цей рядок отримує значення передостаннього та останнього ключів із кортежів ‘P’, які використовувалися для шифрування та розшифрування ‘R’ та ‘L’ відповідно. Два значення ключа зберігаються в змінних ‘p_penultimate’ і ‘p_last’.

return R ^ p_last, L ^ p_penultimate – цей рядок повертає зашифроване значення `R` за допомогою XOR з кінцевим значенням ключа ‘p_last’ і повертає зашифроване значення ‘L’ за допомогою XOR.

_decrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack) – допоміжна статична функція, що використовується для дешифрування блоку даних. Приймає ліву та праву частини блоку (L та R), масив P-значень (P), S-блоки (S1, S2, S3, S4), пакувальник та розпакувальник для перетворення даних (рисунок 2.6).

```
135     @staticmethod
136     def _decrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack):
137         for p2, p1 in P[:0:-1]:
138             L ^= p1
139             a, b, c, d = u1_4Unpack(u4_1Pack(L))
140             R ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
141             R ^= p2
142             a, b, c, d = u1_4Unpack(u4_1Pack(R))
143             L ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
144         pFirst, pSecond = P[0]
145         return R ^ pFirst, L ^ pSecond
```

Рисунок 2.6 – Приватний метод _decrypt

Це статичний метод під назвою «decrypt», який приймає кілька аргументів – L, R, P, S1, S2, S3, S4, u4_1Pack і u1_4Unpack – і повертає два значення: R ^ pFirst і L ^ pSecond.

Цикл for над P[:0:-1] виконує ітерацію по елементах P, починаючи з другого елемента до (але не включаючи) останнього елемента. Для кожної пари елементів (p2, p1) L виконує XOR з p1, а R виконує XOR з ((S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff) XOR p2, де a, b, c і d отримують шляхом розпакування L і R за допомогою двох функцій розпакування u4_1Pack і u1_4Unpack.

Рядок pFirst, pSecond = P[0] призначає перший елемент у P для pFirst і pSecond. Повернене значення – це кінцеві значення R і L, модифіковані XOR'ing з відповідними елементами з P.

encryptCFB(self, data, initVector) – повертає ітератор, який шифрує дані (data) за допомогою режиму зворотного зв'язку (CFB). data може мати довільну довжину. У кожній ітерації повертається блок даних розміром 8 байт, крім останньої ітерації, коли довжина даних не кратна розміру блоку (рисунок 2.7).

```
147     def encryptCFB(self, data, initVector):
148         P = self.P
149         S1, S2, S3, S4 = self.S
150         u4_1Pack = self._u4_1Pack
151         u1_4Unpack = self._u1_4Unpack
152         encrypt = self._encrypt
153         u4_2Pack = self._u4_2Pack
154         data_len = len(data)
155         extraBytes = data_len % 8
156         stopFlag = data_len - extraBytes
157         try:
158             prevCipherL, prevCipherR = self._u4_2Unpack(initVector)
159         except struct_error:
160             raise ValueError("initialization vector is not exactly 8 bytes long.")
161         for plainL, plainR in self._u4_2IterUnpack(data[0:stopFlag]):
162             prevCipherL, prevCipherR = encrypt(
163                 prevCipherL, prevCipherR,
164                 P, S1, S2, S3, S4,
165                 u4_1Pack, u1_4Unpack)
166             prevCipherL ^= plainL
167             prevCipherR ^= plainR
168             yield u4_2Pack(prevCipherL, prevCipherR)
169         if extraBytes:
170             yield bytes(
171                 b ^ n for b, n in zip(
172                     data[stopFlag:],
173                     u4_2Pack(
174                         *encrypt(
175                             prevCipherL, prevCipherR,
176                             P, S1, S2, S3, S4,
177                             u4_1Pack, u1_4Unpack))))
```

Рисунок 2.7 – Метод encryptCFB

Цей метод реалізує режим шифрування CFB за допомогою шифру Blowfish. Режим шифрування CFB (Cipher Feedback) – це метод, при якому вихідні дані процесу шифрування змішуються з даними відкритого тексту перед початком кожної ітерації шифрування. Це гарантує, що зашифровані дані залежать від попереднього зашифрованого тексту, що може підвищити безпеку шифрування.

Метод приймає такі аргументи: * self: екземпляр класу BlowfishCipher * дані: двійковий рядок, який містить дані, які потрібно зашифрувати * initVector: байтовий об’єкт, який використовується як вектор ініціалізації для алгоритму шифрування.

Метод спочатку перевіряє, чи має вектор ініціалізації правильну довжину (8 байт). Якщо ні, виникає виняток ValueError. Потім функція обчислює довжину даних відкритого тексту (data_len) і визначає кількість додаткових байтів у кінці (extraBytes). Це важливо, оскільки шифрування CFB може працювати з даними будь-якої довжини, але під час останньої ітерації, якщо дані не є кратними розміру блоку (8 байт), повернутий байтовий об’єкт може мати довжину, меншу за розмір блоку. Функція використовує блок `try`/`except`, щоб перехопити виняток, який буде викликаний, якщо вектор ініціалізації не має точно 8 байтів.

Основний цикл функції починається з виклику `self._u4_2IterUnpack(data[0:stopFlag])`, який повертає пару байтів зашифрованого тексту (prevCipherL і prevCipherR) і байтів відкритого тексту (plainL і plainR) з першої частини даних. Потім функція шифрує ці два байти за допомогою шифру Blowfish, використовуючи параметри P, S1, S2, S3, S4, u4_1Pack і u1_4Unpack. Потім результат шифрування виконується XOR з байтами відкритого тексту та повертається як байтовий об’єкт за допомогою `u4_2Pack`.

Метод продовжує шифрувати решту даних, використовуючи той самий метод, за винятком того, що якщо в кінці є додаткові байти, вони об’єднуються XOR із попереднім результатом.Метод повертає зашифровані дані як ітератор.

decryptCFB(self, data, initVector): Цей метод декодує дані, зашифровані за допомогою шифру Blowfish, у режимі CFB (Cipher Feedback). Дивитись рисунок 2.8.

```
def decryptCFB(self, data, initVector):
    S1, S2, S3, S4 = self.S
    P = self.P
    u4_1Pack = self._u4_1Pack
    u1_4Unpack = self._u1_4Unpack
    encrypt = self._encrypt
    u4_2Pack = self._u4_2Pack
    extraBytes = len(data) % 8
    stopFlag = len(data) - extraBytes
    try:
        prevCipherL, prevCipherR = self._u4_2Unpack(initVector)
    except struct_error:
        raise ValueError("Initialization vector is not exactly 8 bytes long.")
    for cipherL, cipherR in self._u4_2IterUnpack(
            data[0:stopFlag]):
        prevCipherL, prevCipherR = encrypt(
            prevCipherL, prevCipherR,
            P, S1, S2, S3, S4,
            u4_1Pack, u1_4Unpack)
        yield u4_2Pack(prevCipherL ^ cipherL, prevCipherR ^ cipherR)
        prevCipherL = cipherL
        prevCipherR = cipherR
    if extraBytes:
        yield bytes(
            b ^ n for b, n in zip(
                data[stopFlag:],
                u4_2Pack(
                    *encrypt(
                        prevCipherL, prevCipherR,
                        P, S1, S2, S3, S4,
                        u4_1Pack, u1_4Unpack))
            )
        )
```

Рисунок 2.8 – Метод decryptCFB

CFB може працювати з даними будь-якої довжини, і під час кожної ітерації він створює байтовий об’єкт розміром з блок (8 байт), за винятком останнього, який може бути коротшим. Метод викликає виняток ValueError, якщо вектор ініціалізації (initVector) не має точно 8 байтів.

Метод спочатку обчислює довжину даних і кількість додаткових байтів у кінці (extraBytes) за допомогою оператора модуля (%). Блок try-except перевіряє, чи має initVector правильну довжину (8 байт). Якщо ні, виникає помилка ValueError.

Основний цикл методу починається з виклику self._u4_2IterUnpack(data[0:stopFlag]), який повертає пару байтів зашифрованого тексту (prevCipherL і prevCipherR) і байтів відкритого тексту (plainL і plainR) з першої частини даних. Потім метод шифрує байти відкритого тексту за допомогою шифру Blowfish за допомогою параметрів P, S1, S2, S3, S4, u4_1Pack і u1_4Unpack і повертає результат як байтовий об’єкт розміром з блок. Потім метод встановлює prevCipherL і prevCipherR рівними результату.

Цикл продовжує шифрувати решту даних, використовуючи той самий метод, за винятком того, що якщо в кінці є додаткові байти, вони об’єднуються XOR із попереднім результатом.

Метод повертає розшифровані дані як ітератор.

2.3 Реалізація алгоритму Хафмана

HeapNode: цей клас представляє вузол у дереві кодування Хаффмана. Він має атрибути для символу, частоти, лівого та правого дочірніх елементів. Методи __lt__ і __eq__ визначені для порівняння вузлів на основі частоти.

Клас HuffmanCoding: це основний клас, який обробляє операції стиснення та розпакування.

make_frequency_dict: ця функція приймає текстовий ввід і повертає словник частоти символів. Дивитись рисунок 2.9.

```
32      def make_frequency_dict(self, text):
33          frequency = defaultdict(int)
34          for character in text:
35              frequency[character] += 1
36          return frequency
```

Рисунок 2.9 – Метод make_frequency_dict

Цей метод приймає текст як вхідні дані та повертає словник, де кожен ключ є символом у тексті, а відповідне значення є кількістю входжень цього символу в тексті. Використовується defaultdict, тому, якщо ключ не існує, він буде автоматично створений зі значенням 0. Частоти потім обчислюються шляхом повторення символів у тексті за допомогою циклу for та збільшення відповідного значення частоти словник для кожного символу. Нарешті, словник повертається.

make_heap: Ця функція створює купу (пріоритетну чергу) об’єктів HeapNode на основі частотного словника. Дивитись рисунок 2.10.

```
38      def make_heap(self, frequency):
39          for key in frequency:
40              node = HeapNode(key, frequency[key])
41              heappush(self.heap, node)
```

Рисунок 2.10 – Метод make_heap

Функція починається з використання циклу for для перебору ключів у словнику `frequency`. Для кожного ключа створюється об’єкт HeapNode, який представляє вузол у купі. Значення ключа та частоти передаються як аргументи конструктору класу HeapNode.

Після створення об’єкта HeapNode його надсилають у купу за допомогою методу heappush, який використовується для додавання елементів у верхню частину купи. Атрибут `self.heap` — це купа, яка створюється.

Метод повертає сконструйовану купу.

merge_nodes: ця функція об’єднує вузли в купі, щоб сформувати дерево кодування Хаффмана. Дивитись рисунок 2.11.

```
43      def merge_nodes(self):
44          while len(self.heap)>1:
45              node1 = heappop(self.heap)
46              node2 = heappop(self.heap)
47
48              merged = HeapNode(None, node1.freq + node2.freq)
49              merged.left = node1
50              merged.right = node2
51
52              heappush(self.heap, merged)
```

Рисунок 2.11 - Метод merge_nodes

Цей метод реалізує крок злиття алгоритму Хаффмана, який використовується для об’єднання двох вузлів у купі в один вузол із сумою їхніх частот.

Функція починається з витягування двох вузлів із вершини купи за допомогою методу heappop. Ці вузли зберігаються в змінних `node1` і `node2`.

Потім функція створює новий об'єкт `HeapNode` `merged`, який представляє новий об'єднаний вузол. Частота об'єднаного вузла обчислюється як сума частот «вузла1» і «вузла2». Лівий і правий дочірні елементи об'єднаного вузла мають значення `node1` і `node2` відповідно.

Він повертає об'єднаний вузол назад у купу за допомогою методу heappush. Цикл триває до тих пір, поки в купі є два або більше вузлів.

make_codes_helper – це рекурсивна допоміжна функція, яка проходить дерево кодування Хаффмана та призначає двійкові коди кожному символу. Дивитись рисунок 2.12.

```
52      def make_codes_helper(self, root, current_code):
53          if root is None:
54              return
55
56          if root.char is not None:
57              self.codes[root.char] = current_code
58              self.reverse_mapping[current_code] = root.char
59              return
60
61          self.make_codes_helper(root.left, current_code + "0")
62          self.make_codes_helper(root.right, current_code + "1")
63
```

Рисунок 2.12 – Метод make_codes_helper

Цей метод реалізує рекурсивну функцію для створення кодів в алгоритмі кодування Хаффмана. Метод приймає три аргументи: `root`, `current_code` і `codes`. Метод починається з перевірки, чи `root` має значення None. Якщо так, функція повертається без подальшої обробки.

Далі метод перевіряє, чи має `root` символ. Якщо так, метод встановлює `codes[root.char]` на `current_code` і `reverse_mapping[current_code]` на `root.char`.

Якщо `root` не є None і не має символу, метод здійснює рекурсивний виклик `self.make_codes_helper` з `root.left` і `current_code+'0'`, а також з `root.right` і current_code+ "1".

Функція повертається після виконання всіх рекурсивних викликів, результатом яких має бути набір кодів, які представляють кодування тексту у формі двійкового дерева.

make_codes: ця функція викликає make_codes_helper для генерації двійкових кодів для кожного символу. Дивитись рисунок 2.13.

```
64    def make_codes(self):
65        root = heappop(self.heap)
66        current_code = ""
67        self.make_codes_helper(root, current_code)
```

Рисунок 2.13 – Метод make_codes

Метод починається з вилучення верхнього вузла з купи та збереження його в змінній `root`.

Далі метод встановлює `current_code` у порожній рядок. Потім метод викликає метод `self.make_codes_helper` з `root` і `current_code`, який ініціалізує словник `reverse_mapping` порожнім словником і генерує набір кодів, які представляють кодування тексту у формі бінарне дерево.

Метод завершується після виконання всіх рекурсивних викликів, результатом яких має бути набір кодів, який представляє кодування тексту у формі двійкового дерева.

get_encoded_text – ця функція приймає введення тексту та повертає закодоване двійкове представлення тексту на основі згенерованих кодів. Дивитись рисунок 2.14.

```
69    def get_encoded_text(self, text):
70        encoded_text = ""
71        for character in text:
72            encoded_text += self.codes[character]
73        return encoded_text
```

Рисунок 2.14 – Метод get_encoded_text

Метод починається з ініціалізації порожнього рядка `encoded_text`.

Далі метод повторює кожен символ у тексті за допомогою циклу for. Для кожного символу метод використовує словник `self.codes`, який зберігає коди кожного символу в тексті, щоб визначити відповідний код для додавання до закодованого тексту.

Після повторення всіх символів у тексті метод повертає `encoded_text`, який є рядком, що містить кодоване кодування тексту у формі двійкової послідовності.

pad_encoded_text – ця функція додає відступ до закодованого тексту, щоб зробити його довжину кратною 8. Дивитись рисунок 2.15.

```
75    def pad_encoded_text(self, encoded_text):
76        padding_required = 8 - len(encoded_text) % 8
77        for i in range(padding_required):
78            encoded_text += "0"
79
80        padded_info = "{0:08b}".format(padding_required)
81        padded_encoded_text = padded_info + encoded_text
82        return padded_encoded_text
83
```

Рисунок 2.15 - Метод pad_encoded_text

Метод починається з віднімання поточної довжини закодованого тексту з 8, щоб визначити кількість відступів, необхідних для задоволення вимог щодо довжини.

Далі метод використовує цикл for для повторення необхідної кількості символів заповнення. Для кожної ітерації циклу до закодованого тексту додається символ «0».

Після завершення циклу метод об’єднує остаточний закодований текстовий рядок із рядком заповнення, яке є двійковим рядком, вирівняним за правим краєм, що представляє необхідну кількість символів заповнення, причому кожен біт представляє окремий символ.

Метод повертає доповнений закодований текстовий рядок, який можна використовувати для відновлення оригінального бінарного потоку, закодованого Хаффманом, враховуючи знання кодів Хаффмана та зворотне відображення між кодами та символами.

get_byte_array – ця функція перетворює доповнений закодований текст у масив байтів. Дивитись рисунок 2.16.

```
84      def get_byte_array(self, padded_encoded_text):
85          if len(padded_encoded_text) % 8 != 0:
86              print("Encoded text not padded properly")
87              exit(0)
88
89          b = bytearray()
90          for i in range(0, len(padded_encoded_text), 8):
91              byte = padded_encoded_text[i:i+8]
92              b.append(int(byte, 2))
93          return b
```

Рисунок 2.16 – Метод get_byte_array

Метод починається з перевірки, чи довжина заповненого закодованого тексту не кратна 8, що вказує на те, що кодування не відповідає вимогам щодо довжини. Якщо так, функція друкує повідомлення про помилку та завершує роботу зі статусом 0.

Далі функція створює порожній `bytearray` і використовує цикл for для отримання 8 бітів за раз із доповненого закодованого тексту. Для кожної ітерації циклу функція перетворює 8-бітний байт на ціле число у двійковому форматі за допомогою функції `int`.

Метод додає кожне ціле число до `bytearray` і повертає його після завершення циклу. Отриманий бітовий масив може бути використаний для відновлення оригінального бінарного потоку, закодованого Хаффманом, за умови знання кодів Хаффмана та зворотного відображення між кодами та символами.

compress – ця функція виконує операцію стиснення. Він зчитує вхідний текст, генерує коди Хаффмана, кодує текст, додає доповнення та записує стислі двійкові дані у файл. Дивитись рисунок 2.17.

```
95      def compress(self):
96          decr_name = f'{os.path.splitext(os.path.basename(self.path))[0]}_compressed.bin'
97          output_path = os.path.abspath(os.curdir) + f'/temporary_files/{decr_name}'
98
99          with open(self.path, 'r+') as file, open(output_path, 'wb+') as output:
100             text = file.read().rstrip()
101
102             frequency = self.make_frequency_dict(text)
103             self.make_heap(frequency)
104             self.merge_nodes()
105             self.make_codes()
106
107             encoded_text = self.get_encoded_text(text)
108             padded_encoded_text = self.pad_encoded_text(encoded_text)
109
110             b = self.get_byte_array(padded_encoded_text)
111             output.write(bytes(b))
112
113         print("Compressed")
114         return output_path
```

Рисунок 2.17 – Метод compess

Метод починається з генерації шляху для вихідного файлу, який є оригінальним ім’ям файлу з «_compressed.bin», доданим у кінці.

Далі метод відкриває вхідні та вихідні файли в режимах r+/b+, дозволяючи як читати, так і записувати файл.

Потім метод читає вміст вхідного файлу та перетворює потік символів у словник частот, використовуючи метод make_frequency_dict, визначений в іншому місці. Потім метод сортує словник у купу за допомогою методу make_heap, знову визначеного в іншому місці, для ефективного пошуку елемента з мінімальною частотою в словнику.

Далі метод викликає метод `merge_nodes`, який виконує ітерації по словнику та об’єднує пари вузлів, що мають однакові значення, зменшуючи розмір словника на постійний коефіцієнт кожної ітерації.

Після об’єднання вузлів метод створює кодування потоку символів за допомогою методу make_codes, який обчислює дерево Хаффмана для словника та повертає список кодів, що представляють кожен символ у словнику.

Далі метод доповнює потік закодованих символів, у результаті чого утворюється двійковий потік, який відповідає вимогам довжини кодування, використовуючи метод pad_encoded_text.

Метод перетворює доповнений закодований потік символів у масив байтів за допомогою методу get_byte_array, записуючи його у вихідний файл.

Після запису масиву байтів у вихідний файл метод друкує повідомлення про те, що стиснення було успішним, і повертає вихідний шлях до файлу, який можна використовувати для відновлення вихідного файлу з бінарного потоку за допомогою зворотного відображення між символами та кодами.

remove_padding – цей метод видаляє відступи з доповненого закодованого тексту, щоб відновити оригінальний закодований текст. Метод приймає один аргумент: `paded_encoded_text`, який є доповненим закодованим текстом, який потрібно розмістити. Дивитись рисунок 2.18.

```
116     def remove_padding(self, padded_encoded_text):
117         padded_info = padded_encoded_text[:8]
118         padding_required = int(padded_info, 2)
119 
120         padded_encoded_text = padded_encoded_text[8:]
121         encoded_text = padded_encoded_text[:-1 * padding_required]
122 
123         return encoded_text
```

Рисунок 2.18 – Метод remove_padding

Метод починається з вилучення інформації про заповнення з початку доповненого закодованого тексту та перетворення його на ціле число за допомогою функції `int` з основою 2.

Далі метод видаляє інформацію про заповнення з закодованого тексту, розрізаючи рядок з від’ємними індексами.

Метод витягує інформацію про заповнення, щоб відновити оригінальний закодований текст, використовуючи знання про розмір і довжину закодованого тексту.

Метод повертає оригінальний закодований текст, який можна використовувати для відновлення оригінального бінарного потоку, закодованого Хаффманом, враховуючи знання кодів Хаффмана та зворотне відображення між кодами та символами.

decode_text – ця функція декодує двійкове представлення тексту на основі кодів Хаффмана та повертає вихідний текст. Дивитись рисунок 2.19.

```
125     def decode_text(self, encoded_text):
126         current_code = ""
127         decoded_text = ""
128
129         for bit in encoded_text:
130             current_code += bit
131             if current_code in self.reverse_mapping:
132                 character = self.reverse_mapping[current_code]
133                 decoded_text += character
134                 current_code = ""
135
136         return decoded_text
```

Рисунок 2.19 – Метод decode_text

Метод спочатку ініціалізує порожню змінну `current_code` і порожню змінну `decoded_text`.

Метод переглядає кожен біт закодованого тексту за допомогою циклу for. Для кожного біта він додає біт до змінної `current_code` і перевіряє, чи отриманий `current_code` знаходиться у відображенні між кодом і звичайним текстом.

Якщо `current_code` є у відображенні, метод шукає відповідний символ простого тексту у відображенні за допомогою масиву `reverse_mapping[]`.

Потім він додає символ звичайного тексту до змінної `decoded_text` і скидає змінну `current_code`. Після обробки всіх бітів метод повертає змінну `decoded_text`, яка є відновленим звичайним текстом.

decompress – ця функція виконує операцію декомпресії. Він зчитує стислі двійкові дані з файлу, видаляє заповнення, декодує двійкове представлення та записує розпакований текст у файл. Дивитись рисунок 2.20.

```
138    def decompress(self, input_path):
139        decr_name = f'{os.path.splitext(os.path.basename(input_path))[0]}_decompressed.txt'
140        output_path = os.path.abspath(os.curdir) + f'/temporary_files/{decr_name}'
141
142        with open(input_path, 'rb') as file, open(output_path, 'w+') as output:
143            bit_string = ""
144
145            byte = file.read(1)
146            while byte != b"":
147                byte = ord(byte)
148                bits = bin(byte)[2:].rjust(8, '0')
149                bit_string += bits
150                byte = file.read(1)
151
152            encoded_text = self.remove_padding(bit_string)
153            decompressed_text = self.decode_text(encoded_text)
154
155            output.write(decompressed_text)
156
157        print("Decompressed")
158        return output_path
```

Рисунок 2.20 – Метод decompress

Цей метод розпаковує стислий файл, що зберігається у `input_path`, у текстовий файл під назвою `decr_name_decompressed.txt`, який зберігається у вихідному шляху, що зберігається у `output_path`.

Метод починається з відкриття вхідного файлу в двійковому режимі, що дозволяє читати двійкові дані, що зберігаються у файлі.

Потім він відкриває вихідний файл у режимі запису, що дозволяє записувати дані у файл.

Метод зчитує дані з вхідного файлу по одному байту, перетворює кожен байт у двійковий рядок із 0 і 1 за допомогою функції ord і доповнює кожен двійковий рядок довжиною 8 за допомогою bin функції.

Після того, як усі байти було перетворено на двійкові рядки та доповнено, функція видаляє будь-які непотрібні доповнення з результуючого бітового рядка за допомогою методу `remove_padding`.

Далі функція декодує двійковий рядок за допомогою методу `decode_text` і записує отриманий звичайний текст у вихідний файл.

Метод друкує повідомлення на консоль, вказуючи, що декомпресія була успішною, і повертає вихідний шлях до файлу, який можна використовувати для пошуку щойно створеного розпакованого текстового файлу.

Використання класу HuffmanCoding передбачає створення екземпляра із шляхом до файлу, виклик методу стиснення для стиснення файлу та виклик методу розпакування для розпакування стисненого файлу. Стиснені та розпаковані файли зберігаються у тимчасових файлах із відповідними іменами.

2.4 Реалізація інтерфейсу

__init__ – ця функція ініціалізує вікно програми та встановлює початковий стан інтерфейсу.

create_widgets – ця функція створює елементи графічного інтерфейсу користувача, включаючи кнопки для стиснення, розпакування, шифрування, дешифрування та обчислення хешу [30].

browse_file_path – ця функція відкриває діалогове вікно файлу, у якому користувач може вибрати шлях до файлу. Він оновлює вибрану мітку шляху до файлу та повертає статус вибору файлу.

encrypt – ця функція шифрує вибраний файл за допомогою алгоритму шифрування Blowfish. Він зчитує вміст файлу, шифрує його за допомогою шифру Blowfish і зберігає зашифровані дані у файлі.

decrypt – ця функція розшифровує вибраний файл за допомогою алгоритму шифрування Blowfish. Він зчитує вміст файлу, розшифровує його за допомогою шифру Blowfish і зберігає розшифровані дані у файлі.

compress – ця функція стискає вибраний файл за допомогою алгоритму кодування Хаффмана. Він створює екземпляр класу HuffmanCoding, стискає файл і зберігає стиснуті дані у файлі.

decompress – ця функція розпаковує вибраний файл за допомогою алгоритму кодування Huffman. Він викликає метод розпакування класу HuffmanCoding, щоб розпакувати файл і зберігає розпаковані дані у файл.

show_hash – ця функція обчислює хеш-значення SHA-256 вибраного файлу та відображає його у вікні повідомлення. Він також копіює хеш-значення в буфер обміну.

browse_file_path – ця функція відкриває діалогове вікно файлу, у якому користувач може вибрати шлях до файлу. Він оновлює вибрану мітку шляху до файлу та повертає статус вибору файлу.

Клас MyApp служить основним класом програми та обробляє взаємодії GUI. Він використовує інші модулі та класи, такі як blowfish для шифрування, HuffmanCoding для стиснення та hashlib для обчислення хешу. Інтерфейс містить кнопки для стиснення, розпакування, шифрування, дешифрування та обчислення хешу вибраного файлу. Він також відображає вибраний шлях до файлу та надає кнопку виходу для виходу з програми.

2.5 Висновки по другому розділу

У даному розділі були представлені та детально описані методи та класи, що були використані для реалізації алгоритмів стиснення, шифрування та інтерфейсу в вашому розробленому застосунку.

За результатами огляду можна зробити наступний висновок:

Застосунок має простий та зрозумілий інтерфейс, що дозволяє користувачам легко взаємодіяти з програмою та виконувати необхідні операції стиснення даних, шифрування та перевірки хешу файлів. Описані методи та класи вказують на вашу глибоку розуміння алгоритмів стиснення (зокрема алгоритму Хафмана) та шифрування (наприклад, алгоритму Blowfish методом CFB).

Застосування потужних алгоритмів стиснення та шифрування сприяє досягненню ефективності та безпеки в обробці даних. Ваш розроблений застосунок поєднує ці сильні алгоритми зручним інтерфейсом, що робить його привабливим для користувачів.

Розроблений застосунок є привабливим рішенням для стиснення даних, шифрування та перевірки цілісності файлів. Він пропонує зручну та надійну функціональність, яка може задовольнити потреби користувачів у збереженні даних у захищеному та компактному форматі.

РОЗДІЛ 3
ОПИС ЗАПРОПОНОВАНОГО РІШЕНЯ

3.1 Опис розробленого рішення

Програма надає зручний спосіб взаємодії з користувачем і включає такі основні функціональні можливості

1. Стиснення даних алгоритмом Хафмана:

Реалізовано стиснення даних за допомогою алгоритму Хафмана, що дозволяє зменшити їх обсяг за рахунок оптимального кодування символів.

Алгоритм Хафмана аналізує вхідні дані і будує оптимальне дерево кодування, де менш часті символи мають більш короткі коди, а часті символи мають довші коди.

Це дозволяє ефективно стиснути дані, особливо якщо в них є повторювані символи або патерни.

2. Шифрування даних алгоритмом Blowfish методом CFB:

Для забезпечення конфіденційності даних, програма використовує алгоритм шифрування Blowfish.

Blowfish є блочним алгоритмом шифрування, який застосовує ключову послідовність для шифрування та розшифрування блоків даних.

Режим роботи CFB (Cipher Feedback) використовує шифрування попереднього блоку для створення ключового потоку, який потім використовується для шифрування наступного блоку даних.

Це забезпечує захист від несанкціонованого доступу до даних і забезпечує їх конфіденційність.

3. Перевірка хешу файлів:

Для забезпечення цілісності даних, програма надає можливість перевірки хешу файлів.

Хеш-функція використовується для обчислення унікального хеш-коду для кожного файлу.

Користувач може ввести хеш-код для потрібного файлу, а програма порівняє цей хеш-код з обчисленим хешем вихідного файлу для перевірки його цілісності.

Розроблена програма має інтерфейс (рисунок 3.1):

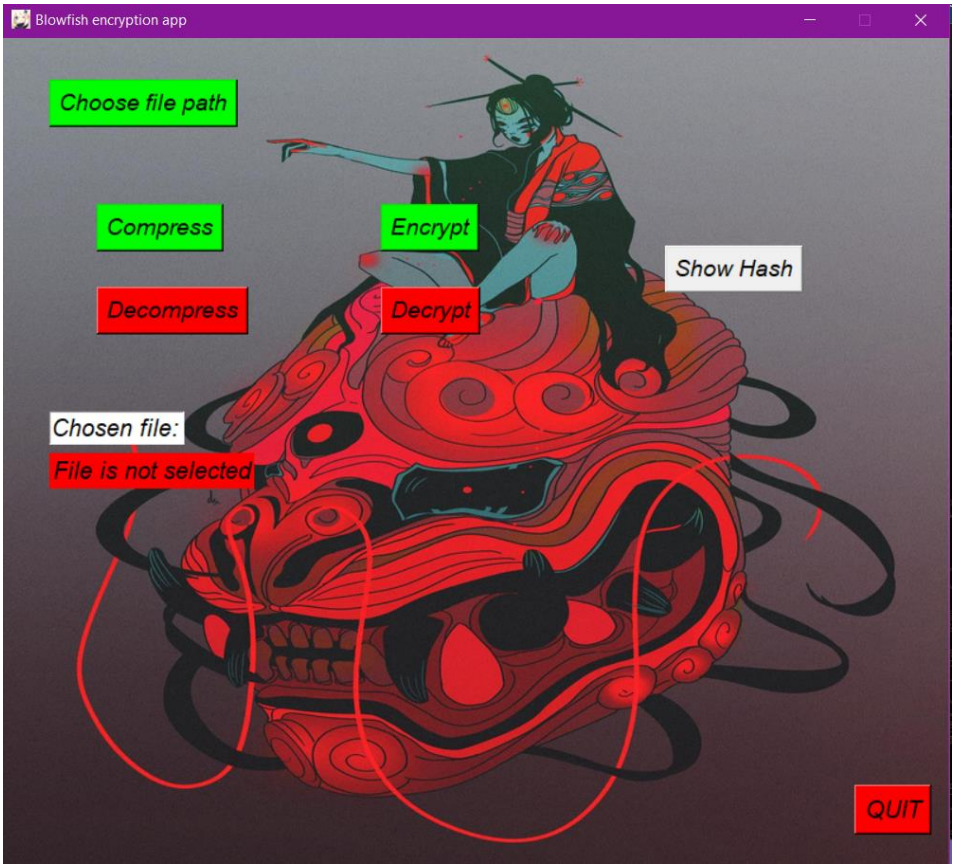

Рисунок 3.1 – Інтерфейс програми

Це рішення має кілька переваг порівняно з іншими програмами:

1. Єдність функціоналу: програма поєднує функції стиснення, шифрування і перевірки хешу в одному інтерфейсі. Це дозволяє зручно виконувати різні операції з даними без необхідності встановлювати та використовувати кілька окремих програм.

2. Використання алгоритмів Хафмана та Blowfish: алгоритм Хафмана забезпечує ефективне стиснення даних, особливо у випадку повторюваних символів або патернів. Алгоритм Blowfish методом CFB забезпечує конфіденційність даних шляхом шифрування їх з використанням ключа.

3. Можливість перевірки хешу: функція перевірки хешу дозволяє перевірити цілісність файлу, забезпечуючи, що дані не були змінені під час передачі або зберігання.

4. Зручний інтерфейс: наявність інтерфейсу полегшує взаємодію з програмою та дозволяє користувачеві легко виконувати необхідні операції з даними, вибирати файли для стиснення та шифрування, вводити хеш-коди та перевіряти їх.

Загалом, розроблене рішення поєднує в собі функції стиснення, шифрування та перевірки цілісності, що забезпечує зручну та безпечну обробку даних.

Далі я наведу скріни та результати роботи програми. Після натискання на кнопку «Choose file path», ми можемо скомпресувати даний файл (рисунок 3.2)

Рисунок 3.2 – Інтерфейс програми з обраним файлом

Після натискання на «Compress» відбудеться стиснення файлу з відповідним сповіщенням (рисунок 3.3)

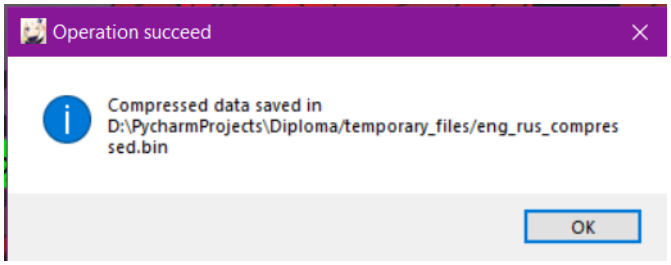

Рисунок 3.3 – Сповіщення про вдале стиснення

Після натискання на «Encrypt» відбудеться шифрування файлу з відповідним сповіщенням (рисунок 3.4)

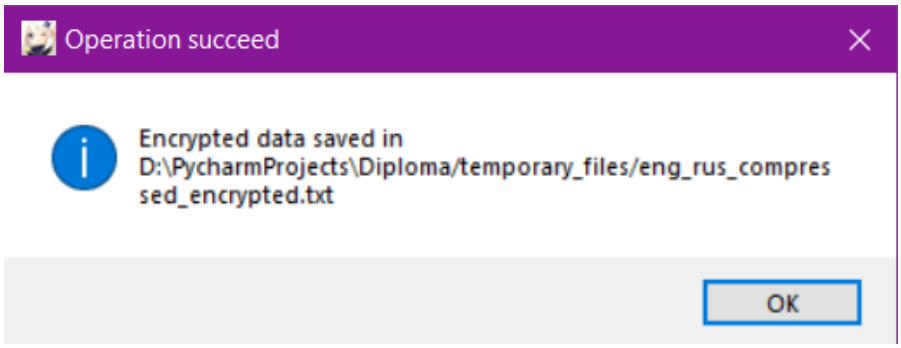

Рисунок 3.4 – Сповіщення про вдале шифрування

Після натискання на «Decrypt» відбудеться дешифрування файлу з відповідним сповіщенням (рисунок 3.5)

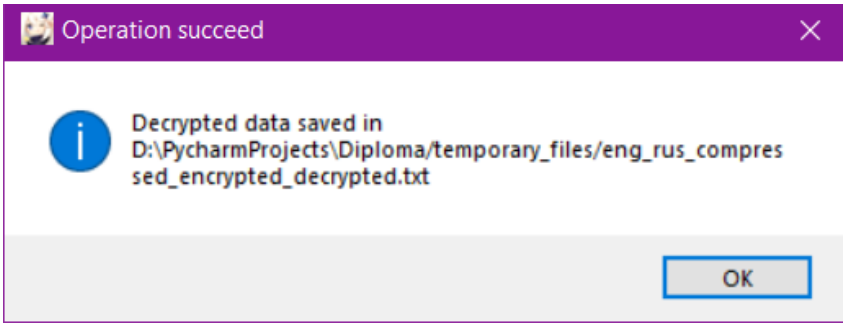

Рисунок 3.5 – Сповіщення про вдале дешифрування

Після натискання на «Decompress» відбудеться відновлення повного розміру файлу з відповідним сповіщенням (рисунок 3.6)

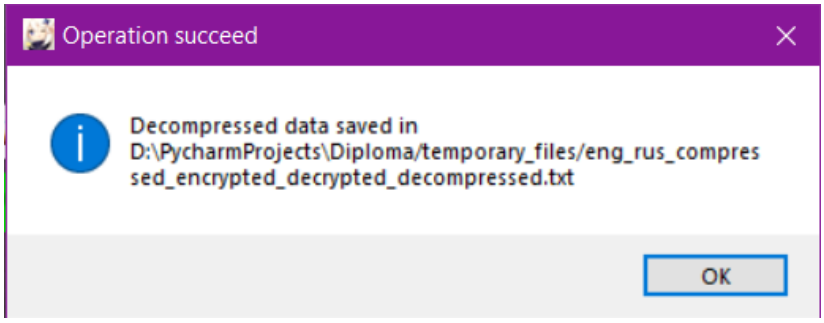

Рисунок 3.6 – Сповіщення про вдале відновлення файлу

Після натискання на «Show hash» на монітор виведеться та скопіюється у буфер хєш даного файлу (рисунок 3.7)

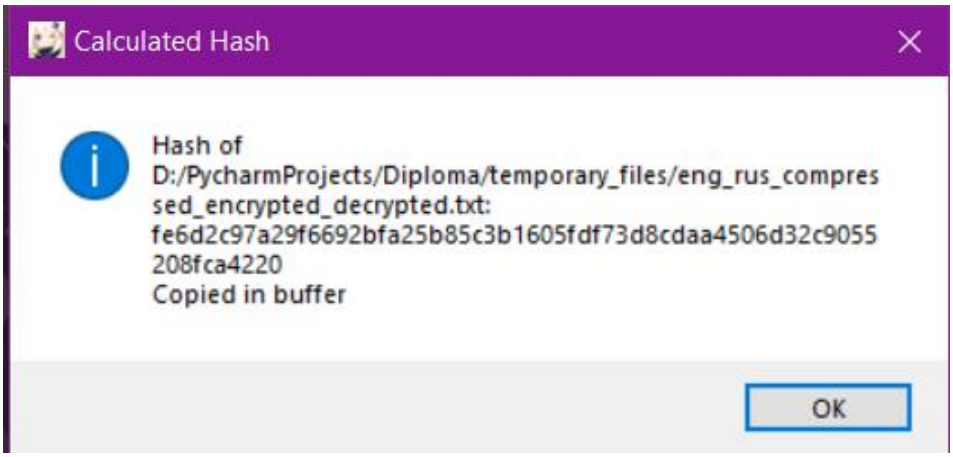

Рисунок 3.7 – Відображення хешу файла.

Далі ми можемо пронаблюдати історію наших змін, бо у даній версії програми зберігаються початкові файли (рисунок 3.8)

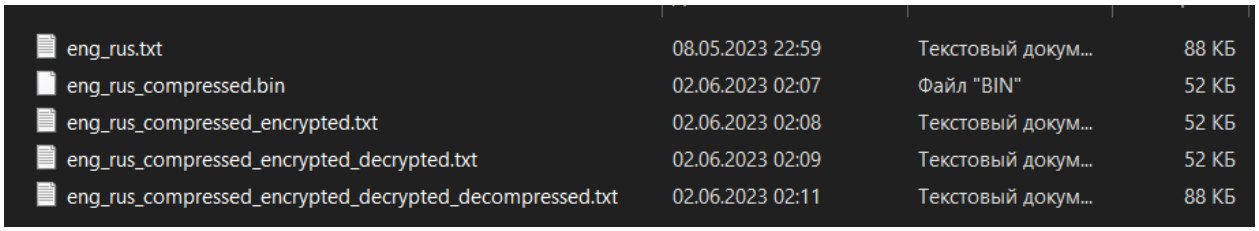

Рисунок 3.8 – Результати роботи програми.

3.2 Існуючі аналоги

Основуючись на описі розробленого рішення, ось кілька аналогів програм, які також надають функціонал стиснення даних алгоритмом Хафмана, шифрування даних алгоритмом Blowfish методом CFB та перевірки хешу файлів:

1. 7-Zip: 7-Zip є популярною програмою для стиснення даних, яка підтримує широкий спектр форматів стиснення. У програмі є підтримка алгоритму Хафмана для стиснення даних. Для шифрування даних використовуються різні алгоритми,

включаючи Blowfish, серед яких можна вибрати метод CFB. 7-Zip також має можливість перевірки хешу файлів для підтвердження їх цілісності.

2. WinRAR: WinRAR є іншим популярним програмним забезпеченням для стиснення даних з підтримкою різних форматів. Він також має вбудовану підтримку алгоритму Хафмана для стиснення даних. Для шифрування використовуються різні алгоритми, включаючи Blowfish, де можна вибрати метод CFB. WinRAR також надає можливість перевірки хешу файлів для забезпечення цілісності.

3. ZipCrypto: ZipCrypto є методом шифрування, який доступний в багатьох програмах для стиснення даних. Він використовує алгоритм шифрування, який базується на шифрі Хаффмана та забезпечує базову захисту даних.

4. Keka: Keka є архіватором для macOS, який надає функціонал стиснення даних алгоритмом Хафмана, шифрування даних алгоритмом Blowfish методом CFB та можливість перевірки хешу файлів.

5. WinZip: WinZip є популярним програмним засобом для стиснення даних. Він надає можливість стиснення файлів за допомогою алгоритму Хафмана, шифрування даних алгоритмом Blowfish методом CFB та перевірки хешу файлів.

Усі ці програми, які були наведені, об'єднує те, що вони надають функціонал стиснення даних і шифрування з метою забезпечення безпеки і ефективності обробки файлів. Основні спільні риси цих програм включають:

1. Стиснення даних: всі ці програми дозволяють зменшити розмір файлів або архівів шляхом застосування алгоритмів стиснення, таких як Хафман, RLE, LZW, Deflate, Бротлі тощо. Це дозволяє зберігати більше даних на диску або передавати їх по мережі швидше.

2. Шифрування даних: всі ці програми забезпечують можливість шифрувати дані з метою забезпечення конфіденційності та захисту від несанкціонованого доступу. Використовуються різні алгоритми шифрування, такі як Blowfish, AES, Serpent, Twofish та інші.

3. Перевірка хешу: багато програм також надають можливість перевірити цілісність файлів за допомогою хеш-функцій, таких як MD5, SHA-1, SHA-256 тощо.

Це дозволяє виявити будь-які зміни або пошкодження файлів під час їх зберігання або передачі.

4. Інтерфейс користувача: більшість програм мають інтуїтивно зрозумілий інтерфейс, що дозволяє користувачам легко виконувати операції стиснення, шифрування та перевірки хешу файлів. Інтерфейс може включати графічну оболонку, контекстне меню або командний рядок.

5. Підтримка різних форматів: багато програм підтримують різні формати архівів і стиснених файлів, такі як ZIP, RAR, 7Z, TAR, GZIP, і багато інших. Це дає можливість працювати з різними типами файлів та забезпечує сумісність з іншими програмами.

Ці спільні риси роблять ці програми потужними інструментами для роботи з даними, забезпечуючи стиснення, шифрування та перевірку цілісності файлів в одному зручному пакеті.

3.3 Порівняння розробленного застосунку із аналогами

Авжеж, розроблений застосунок не може яскраво виділитися на фоні вже популярних та закріплених у користувачів аналогів, але все ж можна назвати декілька переваг представленого застосунку.

Розроблена програма поєднує у собі доволі актуальні на даний час алгоритми стиснення та шифрування, що дає змогу впевнено сказати, що це можна використовувати для забезпечення конфіденціальності важливої інформації.

Перевірка хешу також дозволяє дотримуватися цілісності оброблюємої файлів даних.

Завдяки простому інтерфейсу, навіть неосвідчені користувачі персональних комп'ютерів зможуть легко освоїти використання даного застосунку.

Снизу наведена таблиця порівнянь швидкості стиснення текстового файлу 1 МБ аналогів та запропонованого мною рішення у секундах (таблиця 3.1):

Таблиця 3.1

Порівняння швидкостей стиснення

Застосунок для стиснення	Швідкість виконання операції у секундах
Запропонований мною застосунок (стиснення алгоритмом Хаффмана)	1.56 секунд
7zip	1.59 секунд
WinRar	0.50 секунд
WinZip	Повністю платний аналог

У даному порівнянні важливе значення має система, на якій відбувається обробка обраного файлу в 1 МБ.

Показники системи, на котрій відбулося порівняння:

1. Відеокарта – NVIDIA GeForce GTX 1050
2. Процесор – Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
3. Об’єм оперативної пам’яті – 8 Гб

У процесі порівняння з популярними аналогами можна дійти до висновку, що при стисненні файлу невеликих розмірів, розроблений мною алгоритм проявляє себе гідно і не сильно відстає за швидкістю.

Також слід зазначити, що головним критерієм є вартість програмного забезпечення. Повні версії наведених аналогів не можуть похвалитися безкоштовними рішеннями, коли розроблена мною программа є повністю безкоштовною.

3.4 Висновки по третьому розділу

У даному розділі було проведено огляд та порівняння запропонованого рішення застосунку, який поєднує стиснення даних за допомогою алгоритму Хафмана та шифрування інформації алгоритмом Blowfish, з існуючими аналогами.

В результаті порівняння швидкості стиснення мого застосунку з аналогами було виявлено, що моє рішення не значно відстає за швидкістю від аналогів і може

бути розглянуто як гідний варіант. Це означає, що моя програма здатна забезпечити ефективне стиснення даних без значних затримок.

Важливо відзначити, що моє рішення є повністю безкоштовним, що може бути важливим фактором при виборі програмного забезпечення. Відсутність вартості використання може зробити моє рішення більш доступним для широкого кола користувачів.

Отже, на основі проведеного огляду та порівняння можна зробити висновок, що запропоноване рішення застосунку для стиснення Хафманом та шифрування інформації алгоритмом Blowfish є гідним варіантом, яке може забезпечити ефективне стиснення даних без значних втрат у швидкості. Крім того, його безкоштовний характер робить його привабливим для різних категорій користувачів.

ВИСНОВКИ

Дипломна робота була присвячена аналізу методів стиснення та шифрування даних, а також розробці програмного застосунку, який використовує ці методи. У першому розділі проведений аналіз різних методів стиснення, таких як кодування довжин серій (RLE), Лемпеля-Зіва-Велча (LZW), Deflate, Бротлі та Хафманна. Також розглянуто методи симетричного шифрування, зокрема Data Encryption Standard (DES), Advanced Encryption Standard (AES), Serpent та Blowfish, а також асиметричного шифрування за допомогою алгоритму Rivest-Shamir-Adleman (RSA). У цьому розділі також проведено порівняння різних алгоритмів стиснення та шифрування та було наведено обгрунтування вибору певних алгоритмів для створення застосунку для стиснення та шифрування інформації.

У другому розділі описана розробка програмного застосунку, який використовує алгоритми стиснення Blowfish та Хафмана. Було зосереджено увагу на підготовці вхідних даних, реалізації алгоритму Blowfish, реалізації алгоритму Хафмана та розробці інтерфейсу програмного застосунку.

Третій розділ містить опис запропонованого рішення, демонстрування роботи програми, порівняння з існуючими аналогами та висновки. Ми визначили, що розроблений застосунок має гідну швидкість обробки даних при стисненні за певною конфігурації системи, що не сильно поступається аналогам. Також зазначили, що додаток є повністю безкоштовним, на відміну зазначених аналогів.

В ході розв’язання поставлених задач були отримані наступні наукові та практичні результати:

1. Було проведено аналіз різних методів стиснення та шифрування, включаючи алгоритми стиснення RLE, LZW, Deflate, Бротлі та Хафманна, а також шифрування DES, AES, Serpent, Blowfish та RSA.
2. Було розроблено програмний застосунок, який використовує алгоритми стиснення Blowfish та Хафмана.

3. Було визначено, що результати порівняння з існуючими аналогами свідчать про ефективність розробленого застосунку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Khalid S. Introduction to Data Compression / Sayood Khalid. – Режим доступу до ресурсу: https://www.mbit.edu.in/wp-content/uploads/2020/05/data_compression.pdf

2. Pennebaker W. B. Data Compression: Techniques and Applications / W. B. Pennebaker, J. L. Mitchell., 1980. – Режим доступу до ресурсу: https://www.academia.edu/19619449/Data_compression_techniques_and_applications

3. Salomon D. Handbook of Data Compression / D. Salomon, G. Motta., 2009. – Режим доступу до ресурсу: https://link.springer.com/book/10.1007/978-1-84882-903-9

4. Salomon D. Data Compression: The Complete Reference / David Salomon., 1998. – Режим доступу до ресурсу: https://link.springer.com/book/10.1007/978-1-84628-603-2

5. Nelson M. The Data Compression Book / M. Nelson, J. Gailly., 1991. – Режим доступу до ресурсу: https://hlevkin.com/hlevkin/02imageprocC/The%20Data%20Compression%20Book%202nd%20edition.pdf

6. C. T. B. Managing Gigabytes: Compressing and Indexing Documents and Images / T. B. C., A. Moffat, I. H. Witten., 1999. – 551 с. – Режим доступу до ресурсу: https://sigmodrecord.org/publications/sigmodRecord/0406/RB2.Nagaraj.pdf

7. GeeksforGeeks. Huffman Coding | Greedy Algo-3 [Електронний ресурс] / GeeksforGeeks. – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

8. R. D. S. Cryptography: Theory and Practice [Електронний ресурс] / Douglas Stinson R. // Fourth edition. – 1995. – Режим доступу до ресурсу: https://www.ic.unicamp.br/~rdahab/cursos/mo421-mc889/Welcome_files/Stinson-Paterson_CryptographyTheoryAndPractice-CRC%20Press%20%282019%29.pdf.

9. Katz Katz J. Introduction to Modern Cryptography [Електронний ресурс] / J. Katz, L. Yehuda. – 2007. – Режим доступу до ресурсу:

http://staff.ustc.edu.cn/~mfy/moderncrypto/reading%20materials/Introduction_to_Modern_Cryptography.pdf.

10. J. C. D. M. Information Theory, Inference, and Learning Algorithms / David MacKay J. C., 2003. – Режим доступу до ресурсу: https://www.inference.org.uk/itprnn/book.pdf

11. Schneier B. Applied Cryptography: Protocols, Algorithms, and Source Code in C / Bruce Schneier., 1993. – Режим доступу до ресурсу: https://mrajacse.files.wordpress.com/2012/01/applied-cryptography-2nd-ed-b-schneier.pdf

12. Ferguson N. Cryptography Engineering: Design Principles and Practical Applications / N. Ferguson, B. Schneier, T. Kohno., 2010. – Режим доступу до ресурсу: https://www.schneier.com/wp-content/uploads/2015/12/fortuna.pdf

13. A. S. Handbook of Applied Cryptography / S. A., P. C., J. Alfred., 1996. – Режим доступу до ресурсу: https://doc.lagout.org/network/3_Cryptography/CRC%20Press%20-%20Handbook%20of%20applied%20Cryptography.pdf

14. Bose R. Information Theory, Coding and Cryptography / Ranjan Bose. – Режим доступу до ресурсу: https://dokumen.pub/information-theory-coding-and-cryptography-3nbsped-9789385880568.html

15. Mao W. Modern Cryptography: Theory and Practice / Wenbo Mao., 2003. – Режим доступу до ресурсу: https://theswissbay.ch/pdf/Gentoomen%20Library/Cryptography/Modern%20Cryptography%20Theory%20and%20Practice%20-%20Wenbo%20Mao.pdf

16. Singh S. The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography / Simon Singh., 1999. – Режим доступу до ресурсу: https://books-library.net/files/books-library.net-04040109Rq2I5.pdf

17. Jean-Philippe A. Serious Cryptography: A Practical Introduction to Modern Encryption / Aumasson Jean-Philippe., 2017. – Режим доступу до ресурсу: https://theswissbay.ch/pdf/Books/Computer%20science/Cryptography/SeriousCryptography.pdf

18. Paar C. Understanding Cryptography: A Textbook for Students and Practitioners / C. Paar, J. Pelzl., 2009. – Режим доступу до ресурсу: https://swarm.cs.pub.ro/~mbarbulescu/cripto/Understanding%20Cryptography%20by%20Christof%20Paar%20.pdf

19. P. N. Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3 [Електронний ресурс] / Nguyen P. – 2004. – Режим доступу до ресурсу: https://www.di.ens.fr/~pnguyen/pub_Ng04.html.

20. T. N. Performance Evaluation of DES and Blowfish Algorithms / N. T., S. C., Z.X., 2010. – Режим доступу до ресурсу: https://www.researchgate.net/publication/251927225_Performance_Evaluation_of_DES_and_Blowfish_Algorithms

21. Schneier B. Blowfish: A Fast Block Cipher Algorithm / Bruce Schneier. – Режим доступу до ресурсу: https://link.springer.com/chapter/10.1007/978-0-387-35083-7_13

22. B. S. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish) [Електронний ресурс] / Schneier B. – 1993. – Режим доступу до ресурсу: https://link.springer.com/chapter/10.1007/3-540-58108-1_24

23. P. K. Y. A flexible and adabtive block cipher: Blowfish // Practical Cryptology and Web Security [Електронний ресурс] / Yuen P. K. – 2005. – Режим доступу до ресурсу: https://dokumen.pub/practical-cryptology-and-web-security-9780321263339-0321263332.html

24. Vaudenay S. On the Weak Keys of Blowfish [Електронний ресурс] / Serge Vaudenay. – 1996. – Режим доступу до ресурсу: https://link.springer.com/content/pdf/10.1007/3-540-60865-6_39.pdf.

25. Meyers R. K. An Implementation of the Blowfish Cryptosystem / R. K. Meyers, A. H. Desoky., 2008. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/4775664

26. Коваленко А. Є. Теорія інформації і кодування / Анатолій Єпіфанович Анатолій., 2020. – 248 с. – Режим доступу до ресурсу: https://ela.kpi.ua/bitstream/123456789/41907/1/Kovalenko_AE_TIK_KursLecY20.pdf

27. Iavich, M., Gnatyuk, S., Jintcharadze, E., Polishchuk, Y., Fesenko, A., Abisheva, A. Comparison and hybrid implementation of blowfish, twofish and RSA cryptosystems, 2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering, UKRCON 2019 - Proceedings, 2019, pp. 970–974, 8880005. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/8880005

28. Levitin A. Introduction to the Design and Analysis of Algorithms / Anany Levitin., 2003. – Режим доступу до ресурсу: https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Introduction%20to%20the%20Design%20and%20Analysis%20of%20Algorithms%20%283rd%20ed.%29%20%5BLevitin%202011-10-09%5D.pdf

29. Ferguson N. Practical Cryptography / N. Ferguson, Schneier., 2003. – Режим доступу до ресурсу: https://sisis.rz.htw-berlin.de/inh2008/12361083.pdf

30. Rogaway P. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance [Електронний ресурс] / P. Rogaway, T. Shrimpton. – 2005. – Режим доступу до ресурсу: https://eprint.iacr.org/2004/035.pdf.

ДОДАТОК А

Ініціалізація таблиць замін, приватного ключа та вектору ініціалізації

```
import os

PI_P_ARRAY = (
    0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344, 0xa4093822, 0x299f31d0,
    0x082efa98, 0xec4e6c89, 0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
    0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917, 0x9216d5d9, 0x8979fb1b,
)

# 4 x 256
PI_S_BOXES = (
    (
        0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96,
        0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
        0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
        0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
        0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
        0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
        0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
        0xb4cc5c34, 0x1141e8ce, 0xa15486af, 0x7c72e993, 0xb3ee1411, 0x636fbc2a,
        0x2ba9c55d, 0x741831f6, 0xce5c3e16, 0x9b87931e, 0xafd6ba33, 0x6c24cf5c,
        0x7a325381, 0x28958677, 0x3b8f4898, 0x6b4bb9af, 0xc4bfe81b, 0x66282193,
        0x61d809cc, 0xfb21a991, 0x487cac60, 0x5dec8032, 0xef845d5d, 0xe98575b1,
        0xdc262302, 0xeb651b88, 0x23893e81, 0xd396acc5, 0x0f6d6ff3, 0x83f44239,
        0x2e0b4482, 0xa4842004, 0x69c8f04a, 0x9e1f9b5e, 0x21c66842, 0xf6e96c9a,
        0x670c9c61, 0xabd388f0, 0x6a51a0d2, 0xd8542f68, 0x960fa728, 0xab5133a3,
        0x6eef0b6c, 0x137a3be4, 0xba3bf050, 0x7efb2a98, 0xa1f1651d, 0x39af0176,
        0x66ca593e, 0x82430e88, 0x8cee8619, 0x456f9fb4, 0x7d84a5c3, 0x3b8b5ebe,
        0xe06f75d8, 0x85c12073, 0x401a449f, 0x56c16aa6, 0x4ed3aa62, 0x363f7706,
        0x1bfedf72, 0x429b023d, 0x37d0d724, 0xd00a1248, 0xdb0fead3, 0x49f1c09b,
        0x075372c9, 0x80991b7b, 0x25d479d8, 0xf6e8def7, 0xe3fe501a, 0xb6794c3b,
        0x976ce0bd, 0x04c006ba, 0xc1a94fb6, 0x409f60c4, 0x5e5c9ec2, 0x196a2463,
        0x68fb6faf, 0x3e6c53b5, 0x1339b2eb, 0x3b52ec6f, 0x6dfc511f, 0x9b30952c,
        0xcc814544, 0xaf5ebd09, 0xbee3d004, 0xde334afd, 0x660f2807, 0x192e4bb3,
        0xc0cba857, 0x45c8740f, 0xd20b5f39, 0xb9d3fbdb, 0x5579c0bd, 0x1a60320a,
        0xd6a100c6, 0x402c7279, 0x679f25fe, 0xfb1fa3cc, 0x8ea5e9f8, 0xdb3222f8,
        0x3c7516df, 0xfd616b15, 0x2f501ec8, 0xad0552ab, 0x323db5fa, 0xfd238760,
        0x53317b48, 0x3e00df82, 0x9e5c57bb, 0xca6f8ca0, 0x1a87562e, 0xdf1769db,
        0xd542a8f6, 0x287effc3, 0xac6732c6, 0x8c4f5573, 0x695b27b0, 0xbbca58c8,
        0xe1ffa35d, 0xb8f011a0, 0x10fa3d98, 0xfd2183b8, 0x4afcb56c, 0x2dd1d35b,
        0x9a53e479, 0xb6f84565, 0xd28e49bc, 0x4bfb9790, 0xe1ddf2da, 0xa4cb7e33,
        0x62fb1341, 0xcee4c6e8, 0xef20cada, 0x36774c01, 0xd07e9efe, 0x2bf11fb4,
        0x95dbda4d, 0xae909198, 0xeaad8e71, 0x6b93d5a0, 0xd08ed1d0, 0xafc725e0,
        0x8e3c5b2f, 0x8e7594b7, 0x8ff6e2fb, 0xf2122b64, 0x8888b812, 0x900df01c,
        0x4fad5ea0, 0x688fc31c, 0xd1cff191, 0xb3a8c1ad, 0x2f2f2218, 0xbe0e1777,
        0xea752dfe, 0x8b021fa1, 0xe5a0cc0f, 0xb56f74e8, 0x18acf3d6, 0xce89e299,
        0xb4a84fe0, 0xfd13e0b7, 0x7cc43b81, 0xd2ada8d9, 0x165fa266, 0x80957705,
        0x93cc7314, 0x211a1477, 0xe6ad2065, 0x77b5fa86, 0xc75442f5, 0xfb9d35cf,
        0xebcdaf0c, 0x7b3e89a0, 0xd6411bd3, 0xae1e7e49, 0x00250e2d, 0x2071b35e,
        0x226800bb, 0x57b8e0af, 0x2464369b, 0xf009b91e, 0x5563911d, 0x59dfa6aa,
        0x78c14389, 0xd95a537f, 0x207d5ba2, 0x02e5b9c5, 0x83260376, 0x6295cfa9,
        0x11c81968, 0x4e734a41, 0xb3472dca, 0x7b14a94a, 0x1b510052, 0x9a532915,
        0xd60f573f, 0xbc9bc6e4, 0x2b60a476, 0x81e67400, 0x08ba6fb5, 0x571be91f,
        0xf296ec6b, 0x2a0dd915, 0xb6636521, 0xe7b9f9b6, 0xff34052e, 0xc5855664,
        0x53b02d5d, 0xa99f8fa1, 0x08ba4799, 0x6e85076a,
    ),
```

```
    (
        0x4b7a70e9, 0xb5b32944, 0xdb75092e, 0xc4192623, 0xad6ea6b0, 0x49a7df7d,
        0x9cee60b8, 0x8fedb266, 0xecaa8c71, 0x699a17ff, 0x5664526c, 0xc2b19ee1,
        0x193602a5, 0x75094c29, 0xa0591340, 0xe4183a3e, 0x3f54989a, 0x5b429d65,
        0x6b8fe4d6, 0x99f73fd6, 0xa1d29c07, 0xefe830f5, 0x4d2d38e6, 0xf0255dc1,
        0x4cdd2086, 0x8470eb26, 0x6382e9c6, 0x021ecc5e, 0x09686b3f, 0x3ebaefc9,
        0x3c971814, 0x6b6a70a1, 0x687f3584, 0x52a0e286, 0xb79c5305, 0xaa500737,
        0x3e07841c, 0x7fdeae5c, 0x8e7d44ec, 0x5716f2b8, 0xb03ada37, 0xf0500c0d,
        0xf01c1f04, 0x0200b3ff, 0xae0cf51a, 0x3cb574b2, 0x25837a58, 0xdc0921bd,
        0xd19113f9, 0x7ca92ff6, 0x94324773, 0x22f54701, 0x3ae5e581, 0x37c2dadc,
        0xc8b57634, 0x9af3dda7, 0xa9446146, 0x0fd0030e, 0xecc8c73e, 0xa4751e41,
        0xe238cd99, 0x3bea0e2f, 0x3280bba1, 0x183eb331, 0x4e548b38, 0x4f6db908,
        0x6f420d03, 0xf60a04bf, 0x2cb81290, 0x24977c79, 0x5679b072, 0xbcaf89af,
        0xde9a771f, 0xd9930810, 0xb38bae12, 0xdccf3f2e, 0x5512721f, 0x2e6b7124,
        0x501adde6, 0x9f84cd87, 0x7a584718, 0x7408da17, 0xbc9f9abc, 0xe94b7d8c,
        0xec7aec3a, 0xdb851dfa, 0x63094366, 0xc464c3d2, 0xef1c1847, 0x3215d908,
        0xdd433b37, 0x24c2ba16, 0x12a14d43, 0x2a65c451, 0x50940002, 0x133ae4dd,
        0x71dff89e, 0x10314e55, 0x81ac77d6, 0x5f11199b, 0x043556f1, 0xd7a3c76b,
        0x3c11183b, 0x5924a509, 0xf28fe6ed, 0x97f1fbfa, 0x9ebabf2c, 0x1e153c6e,
        0x86e34570, 0xeae96fb1, 0x860e5e0a, 0x5a3e2ab3, 0x771fe71c, 0x4e3d06fa,
        0x2965dcb9, 0x99e71d0f, 0x803e89d6, 0x5266c825, 0x2e4cc978, 0x9c10b36a,
        0xc6150eba, 0x94e2ea78, 0xa5fc3c53, 0x1e0a2df4, 0xf2f74ea7, 0x361d2b3d,
        0x1939260f, 0x19c27960, 0x5223a708, 0xf71312b6, 0xebadfe6e, 0xeac31f66,
        0xe3bc4595, 0xa67bc883, 0xb17f37d1, 0x018cff28, 0xc332ddef, 0xbe6c5aa5,
        0x65582185, 0x68ab9802, 0xeecea50f, 0xdb2f953b, 0x2aef7dad, 0x5b6e2f84,
        0x1521b628, 0x29076170, 0xecdd4775, 0x619f1510, 0x13cca830, 0xeb61bd96,
        0x0334fe1e, 0xaa0363cf, 0xb5735c90, 0x4c70a239, 0xd59e9e0b, 0xcbaade14,
        0xeecc86bc, 0x60622ca7, 0x9cab5cab, 0xb2f3846e, 0x648b1eaf, 0x19bdf0ca,
        0xa02369b9, 0x655abb50, 0x40685a32, 0x3c2ab4b3, 0x319ee9d5, 0xc021b8f7,
        0x9b540b19, 0x875fa099, 0x95f7997e, 0x623d7da8, 0xf837889a, 0x97e32d77,
        0x11ed935f, 0x16681281, 0x0e358829, 0xc7e61fd6, 0x96dedfa1, 0x7858ba99,
        0x57f584a5, 0x1b227263, 0x9b83c3ff, 0x1ac24696, 0xcdb30aeb, 0x532e3054,
        0x8fd948e4, 0x6dbc3128, 0x58ebf2ef, 0x34c6ffea, 0xfe28ed61, 0xee7c3c73,
        0x5d4a14d9, 0xe864b7e3, 0x42105d14, 0x203e13e0, 0x45eee2b6, 0xa3aaabea,
        0xdb6c4f15, 0xfacb4fd0, 0xc742f442, 0xef6abbb5, 0x654f3b1d, 0x41cd2105,
        0xd81e799e, 0x86854dc7, 0xe44b476a, 0x3d816250, 0xcf62a1f2, 0x5b8d2646,
        0xfc8883a0, 0xc1c7b6a3, 0x7f1524c3, 0x69cb7492, 0x47848a0b, 0x5692b285,
        0x095bbf00, 0xad19489d, 0x1462b174, 0x23820e00, 0x58428d2a, 0x0c55f5ea,
        0x1dadf43e, 0x233f7061, 0x3372f092, 0x8d937e41, 0xd65fecf1, 0x6c223bdb,
        0x7cde3759, 0xcbee7460, 0x4085f2a7, 0xce77326e, 0xa6078084, 0x19f8509e,
        0xe8efd855, 0x61d99735, 0xa969a7aa, 0xc50c06c2, 0x5a04abfc, 0x800bcadc,
        0x9e447a2e, 0xc3453484, 0xfdd56705, 0x0e1e9ec9, 0xdb73dbd3, 0x105588cd,
        0x675fda79, 0xe3674340, 0xc5c43465, 0x713e38d8, 0x3d28f89e, 0xf16dff20,
        0x153e21e7, 0x8fb03d4a, 0xe6e39f2b, 0xdb83adf7,
    ),
    (
        0xe93d5a68, 0x948140f7, 0xf64c261c, 0x94692934, 0x411520f7, 0x7602d4f7,
        0xbcf46b2e, 0xd4a20068, 0xd4082471, 0x3320f46a, 0x43b7d4b7, 0x500061af,
        0x1e39f62e, 0x97244546, 0x14214f74, 0xbf8b8840, 0x4d95fc1d, 0x96b591af,
        0x70f4ddd3, 0x66a02f45, 0xbfbc09ec, 0x03bd9785, 0x7fac6dd0, 0x31cb8504,
        0x96eb27b3, 0x55fd3941, 0xda2547e6, 0xabca0a9a, 0x28507825, 0x530429f4,
        0x0a2c86da, 0xe9b66dfb, 0x68dc1462, 0xd7486900, 0x680ec0a4, 0x27a18dee,
        0x4f3ffea2, 0xe887ad8c, 0xb58ce006, 0x7af4d6b6, 0xaace1e7c, 0xd3375fec,
        0xce78a399, 0x406b2a42, 0x20fe9e35, 0xd9f385b9, 0xee39d7ab, 0x3b124e8b,
        0x1dc9faf7, 0x4b6d1856, 0x26a36631, 0xeae397b2, 0x3a6efa74, 0xdd5b4332,
        0x6841e7f7, 0xca7820fb, 0xfb0af54e, 0xd8feb397, 0x454056ac, 0xba489527,
        0x55533a3a, 0x20838d87, 0xfe6ba9b7, 0xd096954b, 0x55a867bc, 0xa1159a58,
        0xcca92963, 0x99e1db33, 0xa62a4a56, 0x3f3125f9, 0x5ef47e1c, 0x9029317c,
        0xfdf8e802, 0x04272f70, 0x80bb155c, 0x05282ce3, 0x95c11548, 0xe4c66d22,
        0x48c1133f, 0xc70f86dc, 0x07f9c9ee, 0x41041f0f, 0x404779a4, 0x5d886e17,
        0x325f51eb, 0xd59bc0d1, 0xf2bcc18f, 0x41113564, 0x257b7834, 0x602a9c60,
        0xdff8e8a3, 0x1f636c1b, 0x0e12b4c2, 0x02e1329e, 0xaf664fd1, 0xcad18115,
        0x6b2395e0, 0x333e92e1, 0x3b240b62, 0xeebeb922, 0x85b2a20e, 0xe6ba0d99,
        0xde720c8c, 0x2da2f728, 0xd0127845, 0x95b794fd, 0x647d0862, 0xe7ccf5f0,
```

```
        0x5449a36f, 0x877d48fa, 0xc39dfd27, 0xf33e8d1e, 0x0a476341, 0x992eff74,
        0x3a6f6eab, 0xf4f8fd37, 0xa812dc60, 0xa1ebddf8, 0x991be14c, 0xdb6e6b0d,
        0xc67b5510, 0x6d672c37, 0x2765d43b, 0xdcd0e804, 0xf1290dc7, 0xcc00ffa3,
        0xb5390f92, 0x690fed0b, 0x667b9ffb, 0xcedb7d9c, 0xa091cf0b, 0xd9155ea3,
        0xbb132f88, 0x515bad24, 0x7b9479bf, 0x763bd6eb, 0x37392eb3, 0xcc115979,
        0x8026e297, 0xf42e312d, 0x6842ada7, 0xc66a2b3b, 0x12754ccc, 0x782ef11c,
        0x6a124237, 0xb79251e7, 0x06a1bbe6, 0x4bfb6350, 0x1a6b1018, 0x11caedfa,
        0x3d25bdd8, 0xe2e1c3c9, 0x44421659, 0x0a121386, 0xd90cec6e, 0xd5abea2a,
        0x64af674e, 0xda86a85f, 0xbebfe988, 0x64e4c3fe, 0x9dbc8057, 0xf0f7c086,
        0x60787bf8, 0x6003604d, 0xd1fd8346, 0xf6381fb0, 0x7745ae04, 0xd736fccc,
        0x83426b33, 0xf01eab71, 0xb0804187, 0x3c005e5f, 0x77a057be, 0xbde8ae24,
        0x55464299, 0xbf582e61, 0x4e58f48f, 0xf2ddfda2, 0xf474ef38, 0x8789bdc2,
        0x5366f9c3, 0xc8b38e74, 0xb475f255, 0x46fcd9b9, 0x7aeb2661, 0x8b1ddf84,
        0x846a0e79, 0x915f95e2, 0x466e598e, 0x20b45770, 0x8cd55591, 0xc902de4c,
        0xb90bace1, 0xbb8205d0, 0x11a86248, 0x7574a99e, 0xb77f19b6, 0xe0a9dc09,
        0x662d09a1, 0xc4324633, 0xe85a1f02, 0x09f0be8c, 0x4a99a025, 0x1d6efe10,
        0x1ab93d1d, 0x0ba5a4df, 0xa186f20f, 0x2868f169, 0xdcb7da83, 0x573906fe,
        0xa1e2ce9b, 0x4fcd7f52, 0x50115e01, 0xa70683fa, 0xa002b5c4, 0x0de6d027,
        0x9af88c27, 0x773f8641, 0xc3604c06, 0x61a806b5, 0xf0177a28, 0xc0f586e0,
        0x006058aa, 0x30dc7d62, 0x11e69ed7, 0x2338ea63, 0x53c2dd94, 0xc2c21634,
        0xbbcbee56, 0x90bcb6de, 0xebfc7da1, 0xce591d76, 0x6f05e409, 0x4b7c0188,
        0x39720a3d, 0x7c927c24, 0x86e3725f, 0x724d9db9, 0x1ac15bb4, 0xd39eb8fc,
        0xed545578, 0x08fca5b5, 0xd83d7cd3, 0x4dad0fc4, 0x1e50ef5e, 0xb161e6f8,
        0xa28514d9, 0x6c51133c, 0x6fd5c7e7, 0x56e14ec4, 0x362abfce, 0xddc6c837,
        0xd79a3234, 0x92638212, 0x670efa8e, 0x406000e0,
    ),
    (
        0x3a39ce37, 0xd3faf5cf, 0xabc27737, 0x5ac52d1b, 0x5cb0679e, 0x4fa33742,
        0xd3822740, 0x99bc9bbe, 0xd5118e9d, 0xbf0f7315, 0xd62d1c7e, 0xc700c47b,
        0xb78c1b6b, 0x21a19045, 0xb26eb1be, 0x6a366eb4, 0x5748ab2f, 0xbc946e79,
        0xc6a376d2, 0x6549c2c8, 0x530ff8ee, 0x468dde7d, 0xd5730a1d, 0x4cd04dc6,
        0x2939bbdb, 0xa9ba4650, 0xac9526e8, 0xbe5ee304, 0xa1fad5f0, 0x6a2d519a,
        0x63ef8ce2, 0x9a86ee22, 0xc089c2b8, 0x43242ef6, 0xa51e03aa, 0x9cf2d0a4,
        0x83c061ba, 0x9be96a4d, 0x8fe51550, 0xba645bd6, 0x2826a2f9, 0xa73a3ae1,
        0x4ba99586, 0xef5562e9, 0xc72fefd3, 0xf752f7da, 0x3f046f69, 0x77fa0a59,
        0x80e4a915, 0x87b08601, 0x9b09e6ad, 0x3b3ee593, 0xe990fd5a, 0x9e34d797,
        0x2cf0b7d9, 0x022b8b51, 0x96d5ac3a, 0x017da67d, 0xd1cf3ed6, 0x7c7d2d28,
        0x1f9f25cf, 0xadf2b89b, 0x5ad6b472, 0x5a88f54c, 0xe029ac71, 0xe019a5e6,
        0x47b0acfd, 0xed93fa9b, 0xe8d3c48d, 0x283b57cc, 0xf8d56629, 0x79132e28,
        0x785f0191, 0xed756055, 0xf7960e44, 0xe3d35e8c, 0x15056dd4, 0x88f46dba,
        0x03a16125, 0x0564f0bd, 0xc3eb9e15, 0x3c9057a2, 0x97271aec, 0xa93a072a,
        0x1b3f6d9b, 0x1e6321f5, 0xf59c66fb, 0x26dcf319, 0x7533d928, 0xb155fdf5,
        0x03563482, 0x8aba3cbb, 0x28517711, 0xc20ad9f8, 0xabcc5167, 0xccad925f,
        0x4de81751, 0x3830dc8e, 0x379d5862, 0x9320f991, 0xea7a90c2, 0xfb3e7bce,
        0x5121ce64, 0x774fbe32, 0xa8b6e37e, 0xc3293d46, 0x48de5369, 0x6413e680,
        0xa2ae0810, 0xdd6db224, 0x69852dfd, 0x09072166, 0xb39a460a, 0x6445c0dd,
        0x586cdecf, 0x1c20c8ae, 0x5bbef7dd, 0x1b588d40, 0xccd2017f, 0x6bb4e3bb,
        0xdda26a7e, 0x3a59ff45, 0x3e350a44, 0xbcb4cdd5, 0x72eacea8, 0xfa6484bb,
        0x8d6612ae, 0xbf3c6f47, 0xd29be463, 0x542f5d9e, 0xaec2771b, 0xf64e6370,
        0x740e0d8d, 0xe75b1357, 0xf8721671, 0xaf537d5d, 0x4040cb08, 0x4eb4e2cc,
        0x34d2466a, 0x0115af84, 0xe1b00428, 0x95983a1d, 0x06b89fb4, 0xce6ea048,
        0x6f3f3b82, 0x3520ab82, 0x011a1d4b, 0x277227f8, 0x611560b1, 0xe7933fdc,
        0xbb3a792b, 0x344525bd, 0xa08839e1, 0x51ce794b, 0x2f32c9b7, 0xa01fbac9,
        0xe01cc87e, 0xbcc7d1f6, 0xcf0111c3, 0xa1e8aac7, 0x1a908749, 0xd44fbd9a,
        0xd0dadecb, 0xd50ada38, 0x0339c32a, 0xc6913667, 0x8df9317c, 0xe0b12b4f,
        0xf79e59b7, 0x43f5bb3a, 0xf2d519ff, 0x27d9459c, 0xbf97222c, 0x15e6fc2a,
        0x0f91fc71, 0x9b941525, 0xfae59361, 0xceb69ceb, 0xc2a86459, 0x12baa8d1,
        0xb6c1075e, 0xe3056a0c, 0x10d25065, 0xcb03a442, 0xe0ec6e0e, 0x1698db3b,
        0x4c98a0be, 0x3278e964, 0x9f1f9532, 0xe0d392df, 0xd3a0342b, 0x8971f21e,
        0x1b0a7441, 0x4ba3348c, 0xc5be7120, 0xc37632d8, 0xdf359f8d, 0x9b992f2e,
        0xe60b6f47, 0x0fe3f11d, 0xe54cda54, 0x1edad891, 0xce6279cf, 0xcd3e7e6f,
        0x1618b166, 0xfd2c1d05, 0x848fd2c5, 0xf6fb2299, 0xf523f357, 0xa6327623,
        0x93a83531, 0x56cccd02, 0xacf08162, 0x5a75ebb5, 0x6e163697, 0x88d273cc,
        0xde966292, 0x81b949d0, 0x4c50901b, 0x71c65614, 0xe6c6c7bd, 0x327a140a,
```

```
        0x45e1d006, 0xc3f27b9a, 0xc9aa53fd, 0x62a80f00, 0xbb25bfe2, 0x35bdd2f6,
        0x71126905, 0xb2040222, 0xb6cbcf7c, 0xcd769c2b, 0x53113ec0, 0x1640e3d3,
        0x38abbd60, 0x2547adf0, 0xba38209c, 0xf746ce76, 0x77afa1c5, 0x20756060,
        0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e, 0x1948c25c, 0x02fb8a8c,
        0x01c36ae4, 0xd6ebe1f9, 0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,
        0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6,
    ),
)

SPECIAL_PRIVATE_KEY = b"specialkey"
INITVECTOR = os.urandom(8)
```

ДОДАТОК Б

Програмна реалізація алгоритму Blowfish

```
from struct import Struct, error as struct_error
from itertools import cycle as iter_cycle
from encryption.data import PI_P_ARRAY, PI_S_BOXES, SPECIAL_PRIVATE_KEY


class BlowfishCipher(object):
    """
    Blowfish block cipher.

    -----------
    'key' byte must be an object of type bytes and have a length within the range of
4 to 56 bytes.

    'byteOrder' numerical values for mathematical operations is dependent on the
byteOrder,
    which can take on the values of either "big" or "little".
    The default value of "big" is generally sufficient,
    as the majority of Blowfish implementations utilize big endian byte order.

    The key dependent P array and substitution boxes are derived using 'P array' and
'S boxes'.

    The default values for the 'P_Array' and 'S_Boxes' parameters are as follows:
    'P_Array' is a sequence of 18 32-bit integers, while 'S_Boxes' is a 4 x 256
sequence of 32-bit integers
    that are derived from the hexadecimal digits of pi.

    It is possible to specify custom values for these parameters,
    but it is not recommended unless you are knowledgeable about the process.
    For example, 'S boxes' should be a 4 x 256 sequence of 32-bit integers,
    and 'P array' should be a sequence of even length, with the length being a
multiple of 32 bits
    (such as 16, 18, 20, and so on).
    The number of rounds performed on each block is determined by the length of the
'P_Array';
    specifically, for a 'P Array' with a length of n, n - 2 rounds are executed on
every block.

    Encryption & Decryption
    -----------------------
    Blowfish is a block cipher that operates on data in blocks of 64-bits (8 bytes),
    which means it can only process 8 bytes of data at a time.
    To encrypt or decrypt a single block of data, use the encryptBlock or
decryptBlock method.

    However, block ciphers are not very useful if they can only operate on one block
of data at a time.
    Fortunately, block ciphers can be used in different "modes of operation" to
process data larger or smaller
    than the block size, thanks to mathematical techniques.

    I implemented such mode of operation:

    BlowfishCipher Feedback (CFB) - is a mode of operation for block ciphers,
```

```
    including symmetric-key ciphers like AES and Blowfish.

    BlowfishCipher Feedback can operate with data of any length.
    """

    def __init__(self, key, byteOrder="big", P_Array=PI_P_ARRAY, S_Boxes=PI_S_BOXES):
        if not 4 <= len(key) <= 56:
            raise ValueError("Key should be between 4 and 56 bytes")

        if not len(P_Array) or len(P_Array) % 2 != 0:
            raise ValueError("P array should be an even length sequence")

        if len(S_Boxes) != 4 or any(len(box) != 256 for box in S_Boxes):
            raise ValueError("S-boxes should be a 4 x 256 sequence")

        if byteOrder == "big":
            byteOrderFmt = ">"
        elif byteOrder == "little":
            byteOrderFmt = "<"
        else:
            raise ValueError("byte order set wrong, should be 'big' or 'little'")
        self.byteOrder = byteOrder

        # Create structures
        u4_2Struct = Struct("{}2I".format(byteOrderFmt))
        u4_1Struct = Struct(">I".format(byteOrderFmt))
        u8_1Struct = Struct("{}Q".format(byteOrderFmt))
        u1_4Struct = Struct("=4B")

        self._u4_2Pack = u4_2Struct.pack
        self._u4_2Unpack = u4_2Struct.unpack
        self._u4_2IterUnpack = u4_2Struct.iter_unpack

        self._u4_1Pack = u4_1Pack = u4_1Struct.pack

        self._u1_4Unpack = u1_4Unpack = u1_4Struct.unpack

        self._u8_1Pack = u8_1Struct.pack

        # Cycle key iterator
        cycleKeyIter = iter_cycle(iter(key))

        # Cycle 32-bit integer iterator over key bytes
        cycleKey_u4Iter = (x for (x,) in map(u4_1Struct.unpack,
                                             map(bytes, zip(cycleKeyIter,
cycleKeyIter, cycleKeyIter, cycleKeyIter))
                                             ))

        # To initialize the subkey P array and S boxes,
        # every element in the P_Array is XORed with the key and stored as pairs.
        P = [
            (p1 ^ k1, p2 ^ k2) for p1, p2, k1, k2 in zip(
                P_Array[0::2],
                P_Array[1::2],
                cycleKey_u4Iter,
                cycleKey u4Iter
            )
        ]

        S1, S2, S3, S4 = S = [[x for x in sbox] for sbox in S_Boxes]

        encrypt = self._encrypt
        L = R = 0x00000000
```

```
        for i in range(len(P)):
            P[i] = L, R = encrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack)

        # To optimize performance, P is saved as a tuple because working with tuples
is slightly faster.
        self.P = P = tuple(P)

        for box in S:
            for i in range(0, 256, 2):
                L, R = encrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack)
                box[i] = L
                box[i + 1] = R

        # Saving S
        self.S = tuple(tuple(box) for box in S)

    @staticmethod
    def _encrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack):
        for p1, p2 in P[:-1]:
            L ^= p1
            a, b, c, d = u1_4Unpack(u4_1Pack(L))
            R ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
            R ^= p2
            a, b, c, d = u1_4Unpack(u4_1Pack(R))
            L ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
        p_penultimate, p_last = P[-1]
        return R ^ p_last, L ^ p_penultimate

    @staticmethod
    def _decrypt(L, R, P, S1, S2, S3, S4, u4_1Pack, u1_4Unpack):
        for p2, p1 in P[:0:-1]:
            L ^= p1
            a, b, c, d = u1_4Unpack(u4_1Pack(L))
            R ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
            R ^= p2
            a, b, c, d = u1_4Unpack(u4_1Pack(R))
            L ^= (S1[a] + S2[b] ^ S3[c]) + S4[d] & 0xffffffff
        pFirst, pSecond = P[0]
        return R ^ pFirst, L ^ pSecond

    def encryptCFB(self, data, initVector):
        """
        Return an iterator that encrypts 'data' using the BlowfishCipher Feedback
(CFB)
        mode of operation.

        CFB can operate with data of any length.

        During each iteration, except the last one,
        a block-sized byte object (8 bytes) is always returned.

        However, during the final iteration, if 'data' is not a multiple of the block
size,
        the returned byte object may have a length less than the block size.

        'initVector' is the initialization vector and should be a
         byte object with exactly 8 bytes, else 'ValueError' exception is raised.

        'data': byte object (of any length).
        """
        P = self.P
        S1, S2, S3, S4 = self.S

        u4_1Pack = self._u4_1Pack
```

```
        u1_4Unpack = self._u1_4Unpack
        encrypt = self._encrypt

        u4_2Pack = self._u4_2Pack

        data_len = len(data)
        extraBytes = data_len % 8
        stopFlag = data_len - extraBytes

        try:
            prevCipherL, prevCipherR = self._u4_2Unpack(initVector)
        except struct_error:
            raise ValueError("initialization vector is not exactly 8 bytes long.")

        for plainL, plainR in self._u4_2IterUnpack(data[0:stopFlag]):
            prevCipherL, prevCipherR = encrypt(
                prevCipherL, prevCipherR,
                P, S1, S2, S3, S4,
                u4_1Pack, u1_4Unpack
            )
            prevCipherL ^= plainL
            prevCipherR ^= plainR
            yield u4_2Pack(prevCipherL, prevCipherR)

        if extraBytes:
            yield bytes(
                b ^ n for b, n in zip(
                    data[stopFlag:],
                    u4_2Pack(
                        *encrypt(
                            prevCipherL, prevCipherR,
                            P, S1, S2, S3, S4,
                            u4_1Pack, u1_4Unpack
                        )
                    )
                )
            )

    def decryptCFB(self, data, initVector):
        """
        Return an iterator that decrypts 'data' using the BlowfishCipher Feedback
(CFB)
        mode of operation.

        BlowfishCipher Feedback can operate with data of any length.

        During each iteration, except for the last one,
        a block-sized byte object (8 bytes) is  produced.

        'initVector' is the initialization vector and should be a byte object with
exactly 8 bytes,
         else a 'ValueError' exception is raised.

        'data' - byte obj of any length.
        """
        S1, S2, S3, S4 = self.S
        P = self.P

        u4_1Pack = self._u4_1Pack
        u1_4Unpack = self._u1_4Unpack
        encrypt = self._encrypt

        u4_2Pack = self._u4_2Pack
```

```
        extraBytes = len(data) % 8
        stopFlag = len(data) - extraBytes

        try:
            prevCipherL, prevCipherR = self._u4_2Unpack(initVector)
        except struct_error:
            raise ValueError("Initialization vector is not exactly 8 bytes long.")

        for cipherL, cipherR in self._u4_2IterUnpack(
                data[0:stopFlag]
        ):
            prevCipherL, prevCipherR = encrypt(
                prevCipherL, prevCipherR,
                P, S1, S2, S3, S4,
                u4_1Pack, u1_4Unpack
            )
            yield u4_2Pack(prevCipherL ^ cipherL, prevCipherR ^ cipherR)
            prevCipherL = cipherL
            prevCipherR = cipherR

        if extraBytes:
            yield bytes(
                b ^ n for b, n in zip(
                    data[stopFlag:],
                    u4_2Pack(
                        *encrypt(
                            prevCipherL, prevCipherR,
                            P, S1, S2, S3, S4,
                            u4_1Pack, u1_4Unpack
                        )
                    )
                )
            )


cipher = BlowfishCipher(SPECIAL_PRIVATE_KEY)
```

ДОДАТОК В

Програмна реалізація алгоритму Хафмана

```
from heapq import heappush, heappop
from collections import defaultdict
import os


class HeapNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

    def __eq__(self, other):
        if other is None:
            return False
        if not isinstance(other, HeapNode):
            return False
        return self.freq == other.freq


class HuffmanCoding:

    def __init__(self, path):
        self.path = path
        self.heap = []
        self.codes = {}
        self.reverse_mapping = {}

    def make_frequency_dict(self, text):
        frequency = defaultdict(int)
        for character in text:
            frequency[character] += 1
        return frequency

    def make_heap(self, frequency):
        for key in frequency:
            node = HeapNode(key, frequency[key])
            heappush(self.heap, node)

    def merge_nodes(self):
        while len(self.heap)>1:
            node1 = heappop(self.heap)
            node2 = heappop(self.heap)

            merged = HeapNode(None, node1.freq + node2.freq)
            merged.left = node1
            merged.right = node2

            heappush(self.heap, merged)

    def make_codes_helper(self, root, current_code):
        if root is None:
```

```
            return

        if root.char is not None:
            self.codes[root.char] = current_code
            self.reverse_mapping[current_code] = root.char
            return

        self.make_codes_helper(root.left, current_code + "0")
        self.make_codes_helper(root.right, current_code + "1")

    def make_codes(self):
        root = heappop(self.heap)
        current_code = ""
        self.make_codes_helper(root, current_code)

    def get_encoded_text(self, text):
        encoded_text = ""
        for character in text:
            encoded_text += self.codes[character]
        return encoded_text

    def pad_encoded_text(self, encoded_text):
        padding_required = 8 - len(encoded_text) % 8
        for i in range(padding_required):
            encoded_text += "0"

        padded_info = "{0:08b}".format(padding_required)
        padded_encoded_text = padded_info + encoded_text
        return padded_encoded_text

    def get_byte_array(self, padded_encoded_text):
        if len(padded_encoded_text) % 8 != 0:
            print("Encoded text not padded properly")
            exit(0)

        b = bytearray()
        for i in range(0, len(padded_encoded_text), 8):
            byte = padded_encoded_text[i:i+8]
            b.append(int(byte, 2))
        return b

    def compress(self):
        decr_name =
f'{os.path.splitext(os.path.basename(self.path))[0]}_compressed.bin'
        output_path = os.path.abspath(os.curdir) + f'/temporary_files/{decr_name}'

        with open(self.path, 'r+') as file, open(output_path, 'wb+') as output:
            text = file.read().rstrip()

            frequency = self.make_frequency_dict(text)
            self.make_heap(frequency)
            self.merge_nodes()
            self.make_codes()

            encoded_text = self.get_encoded_text(text)
            padded encoded text = self.pad encoded text(encoded text)

            b = self.get_byte_array(padded_encoded_text)
            output.write(bytes(b))

        print("Compressed")
        return output_path

    def remove_padding(self, padded_encoded_text):
```

```
        padded_info = padded_encoded_text[:8]
        padding_required = int(padded_info, 2)

        padded_encoded_text = padded_encoded_text[8:]
        encoded_text = padded_encoded_text[:-1 * padding_required]

        return encoded_text

    def decode_text(self, encoded_text):
        current_code = ""
        decoded_text = ""

        for bit in encoded_text:
            current_code += bit
            if current_code in self.reverse_mapping:
                character = self.reverse_mapping[current_code]
                decoded_text += character
                current_code = ""

        return decoded_text

    def decompress(self, input_path):
        decr_name =
f'{os.path.splitext(os.path.basename(input_path))[0]}_decompressed.txt'
        output_path = os.path.abspath(os.curdir) + f'/temporary_files/{decr_name}'

        with open(input_path, 'rb') as file, open(output_path, 'w+') as output:
            bit_string = ""

            byte = file.read(1)
            while byte != b"":
                byte = ord(byte)
                bits = bin(byte)[2:].rjust(8, '0')
                bit_string += bits
                byte = file.read(1)

            encoded_text = self.remove_padding(bit_string)
            decompressed_text = self.decode_text(encoded_text)

            output.write(decompressed_text)

        print("Decompressed")
        return output_path
```

ДОДАТОК Д

Програмна реалізація інтерфейсу застосунку

```
import os.path
import tkinter as tk
import tkinter.messagebox
from tkinter import filedialog

from PIL import Image, ImageTk
import hashlib
import encryption.blowfish

from compressor import HuffmanCoding
from encryption.data import INITVECTOR


class MyApp(tk.Frame):

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.master.geometry("800x700")
        self.master.title("Blowfish encryption app")
        self.master.resizable(False, False)
        self.master.iconbitmap("C:\Program Files\Genshin Impact\launcher.exe")
        self.file_path = 'File is not selected'
        self.huffmanCompression = None
        self.create_widgets()

    def create_widgets(self):
        # Load background image
        self.background_image = Image.open("background.png")
        self.background_image = self.background_image.resize((800, 700),
Image.LANCZOS)  # set desired size
        self.background_photo = ImageTk.PhotoImage(self.background_image)
        self.background_label = tk.Label(self.master, image=self.background_photo)
        self.background_label.pack(fill="both", expand=True)

        # Add buttons
        self.compress_button = tk.Button(self.master, text="Compress", font=("Arial",
14, "italic"), bg="#00FF00",
                                         command=self.compress)
        self.compress_button.place(relx=0.1, rely=0.2)

        self.decompress_button = tk.Button(self.master, text="Decompress",
font=("Arial", 14, "italic"), bg="red",
                                           command=self.decompress)
        self.decompress_button.place(relx=0.1, rely=0.3)

        self.encrypt_button = tk.Button(self.master, text="Encrypt", font=("Arial",
14, "italic"), bg="#00FF00",
                                        command=self.encrypt)
        self.encrypt_button.place(relx=0.4, rely=0.2)

        self.decrypt_button = tk.Button(self.master, text="Decrypt", font=("Arial",
14, "italic"), bg="red",
```

```
                                          command=self.decrypt)
        self.decrypt_button.place(relx=0.4, rely=0.3)

        self.hash_button = tk.Button(self.master, text="Show Hash", font=("Arial",
14, "italic"),
                                     command=self.show_hash)
        self.hash_button.place(relx=0.7, rely=0.25)

        # create the GUI elements
        self.file_path_button = tk.Button(self.master, text="Choose file path",
font=("Arial", 14, "italic"),
                                          bg="#00FF00",
command=self.browse_file_path)
        self.file_path_button.place(relx=0.05, rely=0.05)

        text = tk.Text(self.master, font=("Arial", 15, "italic"), height=1, width=10)
        text.insert(tk.END, "Chosen file:")
        text.place(relx=0.05, rely=0.45)

        self.chosen_path = tk.Label(self.master, text=self.file_path, font=("Arial",
15, "italic"), bg='red')
        self.chosen_path.place(relx=0.05, rely=0.5)

        self.quit = tk.Button(self.master, text="QUIT", fg="black", font=("Arial",
15, "italic"), bg='red',
                              command=self.master.destroy)
        self.quit.place(relx=0.9, rely=0.9)

    def browse_file_path(self):
        file_path = filedialog.askopenfilename()
        if not file_path:
            tk.messagebox.showwarning(title='Operation suspended', message=f'File was
not chosen')
            self.chosen_path.config( bg='red')
            self.file_path = 'File is not selected'
            status = False
        else:
            self.chosen_path.config(bg='#00FF00')
            self.file_path = file_path
            status = True
        self.chosen_path.config(text=self.file_path)
        return status

    def encrypt(self):
        if self.browse_file_path():
            with open(self.file_path, "rb") as f:
                data = f.read()

            data_encrypted = b"".join(encryption.blowfish.cipher.encryptCFB(data,
INITVECTOR))

            encr_name =
f'{os.path.splitext(os.path.basename(self.file_path))[0]}_encrypted.txt'
            encrypted_data_path = os.path.abspath(os.curdir) +
f'/temporary_files/{encr_name}'

            with open(encrypted_data_path, 'wb+') as f:
                f.write(data_encrypted)
            tk.messagebox.showinfo(title='Operation succeed', message=f'Encrypted
data saved in {encrypted_data_path}')

    def decrypt(self):
        if self.browse_file_path():
            with open(self.file_path, "rb") as f:
```

```
                data = f.read()
            data_decrypted = b"".join(encryption.blowfish.cipher.decryptCFB(data,
INITVECTOR))

            decr_name =
f'{os.path.splitext(os.path.basename(self.file_path))[0]}_decrypted.txt'
            decrypted_data_path = os.path.abspath(os.curdir) +
f'/temporary_files/{decr_name}'

            with open(decrypted_data_path, 'wb+') as f:
                f.write(data_decrypted)
            tk.messagebox.showinfo(title='Operation succeed', message=f'Decrypted
data saved in {decrypted_data_path}')

    def compress(self):
        if self.browse_file_path():
            self.huffmanCompression = HuffmanCoding(self.file_path)
            compressed_file = self.huffmanCompression.compress()
            tk.messagebox.showinfo(title='Operation succeed', message=f'Compressed
data saved in {compressed_file}')

    def decompress(self):
        if self.browse_file_path():
            self.decompressed_file =
self.huffmanCompression.decompress(self.file_path)
            tk.messagebox.showinfo(title='Operation succeed', message=f'Decompressed
data saved in {self.decompressed_file}')

    def show_hash(self):
        with open(self.file_path, "rb") as f:
            content = f.read()
            hash_value = hashlib.sha256(content).hexdigest()
            print(f"Hash of {self.file_path}: {hash_value}")
        tk.messagebox.showinfo(title="Calculated Hash", message=f"Hash of
{self.file_path}: "

f"{hash_value}\nCopied in buffer")
        self.master.clipboard_clear()
        self.master.clipboard_append(hash_value)
```

ДОДАТОК Е

Таблиця Е.1

Порівняння алгоритмів стиснення

Алгоритм	Опис	Переваги	Недоліки
RLE (Run-Length Encoding)	RLE є простим алгоритмом, який замінює повторювані символи або серії символів одним кодом, що вказує на кількість повторів.	Простий у реалізації, ефективний для повторюваного зображення та простих шаблонів	Не ефективний для неконтекстних даних та випадків, коли немає повторів.
LZW (Lempel-Ziv-Welch)	LZW є алгоритмом стиснення, який заснований на будівництві словника. Він використовує словник, щоб замінювати повторювані послідовності символів більш короткими кодами.	Добре стискає текстові дані, здатний до компресії довгих послідовностей повторюваних символів	Потребує додаткової таблиці для відновлення даних, може бути більш складним у реалізації.
Deflate	Deflate є комбінацією алгоритмів LZ77 (стиснення з використанням слайдера) та Хафмана (оптимальне кодування Хафмана). Він використовує слайдер для заміни повторюваних фрагментів даних, а потім застосовує оптимальне кодування Хафмана для подальшого стиснення.	Здатний до високого стиснення, широко підтримується, використовується в ZIP та gzip форматах.	Може бути вимогливим до ресурсів при великих розмірах даних.
Brotli	Бротлі є алгоритмом стиснення, розробленим Google. Він використовує різні техніки, включаючи кодування потоку та алгоритм Хафмана, для досягнення високого рівня стиснення.	Дуже ефективний для стиснення текстових та веб-даних, здатний до високого рівня стиснення.	Вимагає більшої обчислювальної потужності для стиснення/розстиснення
Huffman	Є оптимальним кодуванням, яке використовується для стиснення даних на основі частоти вживання символів. Кожен символ представляється бінарним кодом змінної довжини, де менш часті символи мають більш короткі коди.	Простий у реалізації. Ефективний у стисненні даних з різною частотою вживання символів.	Не завжди надає найкраще стиснення для всіх типів даних, може вимагати більшого обсягу пам'яті.

ДОДАТОК Ж

Таблиця Ж.1

Порівняння алгоритмів шифрування

Алгоритм	Розмір ключа	Кількість раундів	Розмір блоку данах	О велике	Стійкість
AES	128, 196, 256 біт	10, 12, 14	128 біт	O(n)	AES є одним з найбільш стійких симетричних алгоритмів шифрування. Він використовує ключі довжиною 128, 192 або 256 бітів і вважається безпечним для багатьох застосувань.
RSA	1024-4092 біт	-	-	O(N^2)	RSA є асиметричним алгоритмом, який базується на складності розкладання на прості множники великих чисел. Він використовується для шифрування та підпису даних. RSA вважається стійким, але вимагає довгих ключів для досягнення високого рівня безпеки.
Blowfish	32-448 біт	16	64 біт	O(n)	Blowfish є симетричним алгоритмом, який використовує ключі довжиною від 32 до 448 бітів. Він вважається стійким і безпечним, але деякі експерти вказують на те, що більш нові алгоритми, такі як AES, можуть мати більшу стійкість та криптографічну безпеку.

Продовження табл. Ж.1

Serpent	128, 196, 256 біт	32	128 біт	O(n)/O(2n)	Serpent є симетричним алгоритмом, який також використовує ключі довжиною 128, 192 або 256 бітів. Він вважається стійким та безпечним, але може бути менш ефективним за швидкодією порівняно з іншими алгоритмами.
DES	56 біт	16	64 біт	O(n)	DES є старішим симетричним алгоритмом, який використовує ключі довжиною 56 бітів. Він був розроблений у 1970-х роках і в даний час вважається менш стійким через обмежену довжину ключа.