Міністерство освіти та науки

Прикарпатський національний університет

Фізико-технічний факультет

Кафедра комп’ютерної інженерії та електроніки

Лабораторна робота № 9

З курсу “Захист інформації у комп’ютерних мережах та системах”

Виконав студент групи КІ-41

Воробій Віталій

Івано-Франківськ 2020

5 варіант

Програмна реалізація мовою *Java:*

Перед написанням алгоритму DES я реалізував усі необхідні алгоритми для його функціонування. Їх реалізацію було поміщено у окремий клас для їх гіпотетичного повторного використання.

import java.util.\*;  
  
public class Algorithms {  
  
 */\*\*  
 \* Функція для випадкової перестановки об'єктів  
 \* @param objects вхідний масив  
 \* @param <T> Тип даних  
 \*/* public static <T> List<T> shuffle(List<T> objects) {  
 final int N = objects.size();  
 final List<T> shuffledList = new ArrayList<>();  
 final Set<Integer> ownedIndexes = new HashSet<>();  
 final Random r = new Random();  
 for (int i = 0; i < N; i++) {  
 int newIndex = Math.*abs*(r.nextInt()) % N;  
 while (ownedIndexes.contains(newIndex)) {  
 newIndex = Math.*abs*(r.nextInt()) % N;  
 }  
 ownedIndexes.add(newIndex);  
 shuffledList.add(objects.get(newIndex));  
 }  
 return shuffledList;  
 }  
  
 */\*\*  
 \* @param num Вхідне 64 бітне число  
 \* @param position Індекс  
 \* @param value Значення (0 або 1)  
 \* @return Видозмінене значення  
 \*/* public static Long setBit(Long num, int position, long value) {  
 if (value == 0) {  
 return num & ~((long) 1 << position);  
 }  
 return num | ((long) 1 << position);  
 }

*/\*\*  
 \* Функція для отримання значення біту у позиції position  
 \* @param position Позиція (індекс)  
 \*/* public static Long getBit(Long val, int position) {  
 return ( ( val >> position) & 1);  
 }

*/\*\*  
 \* Генерація вектора, що містить послідовність значень від @from до @to  
 \* Решта елементів додаються довільним чином  
 \* @param length Довжина вихідного вектора  
 \* @param from Початкове значення  
 \* @param to Кінцеве значення  
 \*/* public static List<Integer> generateVectorAndFitToLengthByRandom(int length, int from, int to) {  
 final int m = to - from;  
 final List<Integer> result = new ArrayList<>();  
 for (int i = 0; i < length; i++) {  
 int v = from + (to - from + i - 1 ) % m ;  
 result.add(v);  
 }  
 return result;  
 }  
  
 */\*\*  
 \* Функція для перетворення вектора у масив  
 \* @param inputVector Вхідний вектор чисел  
 \*/* public static int[] convertVectorToArray(List<Integer> inputVector) {  
 final int N = inputVector.size();  
 final int[] res = new int[N];  
 for (int i = 0; i < N; i++) {  
 res[i] = inputVector.get(i);  
 }  
 return res;  
 }  
  
 */\*\*  
 \* Функція для генерування випадкового тензора  
 \*/* public static int[][][] generateRandomTensor(int r, int m, List<Integer> vector) {  
 final int N = vector.size();  
 final int[][][] resultTensor = new int[r][m][N];  
  
 for (int i = 0; i < r; i++) {  
 for (int j = 0; j < m; j++) {  
 final List<Integer> shuffledVector = *shuffle*(vector);  
 resultTensor[i][j] = *convertVectorToArray*(shuffledVector);  
 }  
 }  
  
 return resultTensor;  
 }  
  
 */\*\*  
 \* Функція для зменшення розміру вектора довільним чином  
 \* @param inputList Вхідний вектор  
 \* @param size Бажаний розмір  
 \*/* public static List<Integer> trimToSize(List<Integer> inputList, int size) {  
 final int listSize = inputList.size();  
 final Set<Integer> uniqueVals = new HashSet<>();  
 final Random r = new Random();  
 while (uniqueVals.size() != size) {  
 Integer v = inputList.get(Math.*abs*(r.nextInt()) % listSize);  
 uniqueVals.add(v);  
 }  
 return new ArrayList<>(uniqueVals);  
 }  
  
 */\*\*  
 \* Створення бітової маски to < from  
 \* @param from Початковий індекс  
 \* @param to Кінцевий індекс  
 \*/* public static Long createBitMask(int from, int to) {  
 long mask = 0L;  
 for (int i = from; i < to; i++) {  
 mask |= 1L << i;  
 }  
 return mask;  
 }  
  
 */\*\*  
 \* Зсув вліво без втрат  
 \* @param initialNum Початкове число  
 \* @param rounds Кількість зсувів вліво  
 \* @param mask Бітова маска  
 \*/* public static long safeLeftShift(final long initialNum, int rounds, long mask, int bitNum) {  
 final int neededRounds = rounds % bitNum;  
 long res = (initialNum << rounds) & mask;  
 for (int i = bitNum - neededRounds; i < bitNum; i++) {  
 long bitValue = *getBit*(initialNum, i);  
 res = *setBit*(res, i - bitNum + neededRounds , bitValue);  
 }  
 return res;  
 }  
  
  
 */\*\*  
 \* Функція для здійснення перестановки бітів у деякому n розрядному числі  
 \* @param initialNum Початкове значення  
 \* @param shuffleList Вектор перестановок  
 \* @param bitsNum Розрядність числа  
 \* @param readReverse Булеве значення, що вказує чи проводити читання з кінця (true) чи з початку  
 \* @param writeReverse Булеве значення, що вказує чи проводити запис з кінця (true) чи з початку  
 \*/* public static long permutate(final long initialNum, List<Integer> shuffleList, int bitsNum, boolean readReverse, boolean writeReverse) {  
 final int LIST\_SIZE = shuffleList.size();  
 long result = 0L;  
 for (int i = 0; i < LIST\_SIZE; i++) {  
 int v = shuffleList.get(i);  
 long bitValue = Algorithms.*getBit*(initialNum, readReverse ? bitsNum - v - 1 : v);  
 result = Algorithms.*setBit*(result, writeReverse ? LIST\_SIZE - i - 1 : i, bitValue);  
 }  
 return result;  
 }

*/\*\*  
 \* Функція для здійснення оберненої перестановки бітів у деякому n розрядному числі  
 \* @param initialNum Початкове значення  
 \* @param shuffleList Вектор перестановок  
 \* @param bitsNum Розрядність числа  
 \* @param readReverse Булеве значення, що вказує чи проводити читання з кінця (true) чи з початку  
 \*/* public static long permutateInverse(final long initialNum, List<Integer> shuffleList, int bitsNum, boolean readReverse) {  
 final int LIST\_SIZE = shuffleList.size();  
 long result = 0L;  
 for (int i = 0; i < LIST\_SIZE; i++) {  
 int v = shuffleList.get(i);  
 long bitValue = Algorithms.*getBit*(initialNum, readReverse ? bitsNum - i - 1 : i);  
 result = Algorithms.*setBit*(result, LIST\_SIZE - v - 1, bitValue);  
 }  
 return result;  
 }  
  
}

Написавши вищенаведені алгоритми я реалізував клас, що реалізує всю логіку роботи DES.

package solution;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
import java.util.function.LongFunction;  
import java.util.stream.Collectors;  
import java.util.stream.IntStream;  
  
public class DES {  
  
 */\*\*  
 \* Число бітів блоку  
 \*/* private static final Integer *NUM\_OF\_BITS* = 64;  
  
 */\*\*  
 \* Кількість раундів Фестеля  
 \*/* private static final Integer *FEISTEL\_ROUNDS* = 16;  
  
 */\*\*  
 \* Параметру вектора E  
 \*/* private static final Integer *E\_MIN* = 0, *E\_MAX* = 32, *E\_COUNT* = 48;  
  
 */\*\*  
 \* Параметри тензора S  
 \*/* private static final Integer *S\_BLOCKS* = 8, *S\_ROWS* = 4, *S\_VECTOR\_MIN* = 0, *S\_VECTOR\_MAX* = 16;  
  
 */\*\*  
 \* Параметри вектора P  
 \*/* private static final Integer *P\_MIN* = 0, *P\_MAX* = 32;

*/\*\*  
 \* Параметри вектора G  
 \*/* private static final Integer *G\_MIN* = 0, *G\_MAX* = 64;  
  
 */\*\*  
 \* Параметри вектора H  
 \*/* private static final Integer *H\_MIN* = 0, *H\_MAX* = 56, *H\_NEEDED* = 48;  
  
 */\*\*  
 \* IP  
 \*/* private List<Integer> IP;  
  
 */\*\*  
 \* Вектор E  
 \*/* private List<Integer> E;  
  
 */\*\*  
 \* Тензор S  
 \*/* private int[][][] S;  
  
 */\*\*  
 \* Вектор P  
 \*/* private List<Integer> P;  
  
 */\*\*  
 \* Вектор G  
 \*/* private List<Integer> G;  
  
 */\*\*  
 \* Вектор H  
 \*/* private List<Integer> H;  
  
 */\*\*  
 \* Вектор зсувів для формування ключів  
 \*/* private List<Integer> keyShifts;  
  
 */\*\*  
 \* Вектор ключів  
 \*/* private List<Long> keys;  
  
 */\*\*  
 \* Половина розміру IP і відповідна маска  
 \*/* final int HALF\_IP = *NUM\_OF\_BITS* / 2;  
 final long HALF\_IP\_MASK = Algorithms.*createBitMask*(0, HALF\_IP);

public DES() {  
 calcShuffleMaps();  
 initVectorE();  
 initTensorS();  
 initVectorP();  
 initVectorG();  
 initVectorH();  
 initKeysShifts();  
 initKeys();  
 }  
  
 */\*\*  
 \* Ініціалізація зсувів для ключів  
 \* Ініцілаізується вектор з 1 і 2 розмір якого рівний кількості раундів Фейстеля  
 \* Розподіл - нормальний (1 і 2 має однакову ймовірність бути обраною)  
 \*/* private void initKeysShifts() {  
 final Random r = new Random();  
 keyShifts = IntStream  
 .*range*(1, *FEISTEL\_ROUNDS* + 1)  
 .boxed()  
 .map(n -> r.nextInt(100) > 50 ? 1 : 2)  
 .collect(Collectors.*toList*());  
 }  
  
 */\*\*  
 \* Ініціалізація ключів  
 \*/* private void initKeys() {  
  
 *// Початкове значення ключа* final long initialKey = Math.*abs*(new Random().nextLong());  
  
 *// Кількість ключів рівна кількості раундів Фейштеля* keys = new ArrayList<>(*FEISTEL\_ROUNDS*);  
  
 final int HALF\_SIZE = G.size() / 2;  
  
 *// Бітова маска довжиною половини розміру вектора G (28 біт)* final long HALF\_SIZE\_MASK = Algorithms.*createBitMask*(0, HALF\_SIZE);  
  
 *// Перестановлене число відносно вектора G* final long shuffledNum = Algorithms.*permutate*(initialKey, G, *NUM\_OF\_BITS*, true, true);  
  
 *// Поточні 32 бітні 'половинки'* long C\_i = ((shuffledNum >> HALF\_SIZE) & HALF\_SIZE\_MASK);  
 long D\_i = (shuffledNum & HALF\_SIZE\_MASK);  
  
 for (int i = 0; i < *FEISTEL\_ROUNDS*; i++) {  
  
 final int currentShift = keyShifts.get(i);  
  
 *// Застосування попередньо розробленого алгоритму для зсуву вліво половинок на n позицій без втрати бітів* C\_i = Algorithms.*safeLeftShift*(C\_i, currentShift, HALF\_SIZE\_MASK, HALF\_SIZE);  
 D\_i = Algorithms.*safeLeftShift*(D\_i, currentShift, HALF\_SIZE\_MASK, HALF\_SIZE);  
  
 *// Об'єднання половинок у 64 бітне число* final long combined = (C\_i << HALF\_SIZE) | D\_i;  
  
 *// Застосування вектора H для перестановки бітів у сконкатенованому числі* final long K\_i\_h = Algorithms.*permutate*(combined, H, HALF\_SIZE \* 2, true, true);  
  
 *// Додаємо ключ у список* keys.add( K\_i\_h);  
 }  
  
 }

*/\*\*  
 \* Ініціалізація вектора H  
 \* З вектора H довільно вилучаються 8 чисел  
 \*/* private void initVectorH() {  
 final List<Integer> hVector = IntStream  
 .*range*(*H\_MIN*, *H\_MAX*)  
 .boxed()  
 .collect(Collectors.*toList*());  
 final List<Integer> trimmed = Algorithms.*trimToSize*(hVector, *H\_NEEDED*);  
 this.H = Algorithms.*shuffle*(trimmed);  
 }  
  
 */\*\*  
 \* Ініціалізація вектора G  
 \*/* private void initVectorG() {  
 final List<Integer> gVector = IntStream  
 .*range*(*G\_MIN* + 1, *G\_MAX* + 1)  
 .boxed()  
 .filter(num -> num < 8 || num % 8 != 0) *// Фільтр кратних чисел 8* .map(num -> num - 1)  
 .collect(Collectors.*toList*());  
 this.G = Algorithms.*shuffle*(gVector);  
 }  
  
 */\*\*  
 \* Ініціалізація P  
 \*/* private void initVectorP() {  
 final List<Integer> rangeVector = IntStream  
 .*range*(*P\_MIN*, *P\_MAX*)  
 .boxed()  
 .collect(Collectors.*toList*());  
 P = Algorithms.*shuffle*(rangeVector);  
 }  
  
 */\*\*  
 \* Ініціалізація S  
 \*/* private void initTensorS() {  
 S = Algorithms.*generateRandomTensor*(  
 *S\_BLOCKS*,  
 *S\_ROWS*,  
 IntStream  
 .*range*(*S\_VECTOR\_MIN*, *S\_VECTOR\_MAX*)  
 .boxed()  
 .collect(Collectors.*toList*())  
 );  
 }  
  
 */\*\*  
 \* Ініціалізація E  
 \*/* private void initVectorE() {  
 E = Algorithms.*generateVectorAndFitToLengthByRandom*(*E\_COUNT*, *E\_MIN*, *E\_MAX*);  
 }

*/\*\*  
 \* Розрахунок IP  
 \*/* private void calcShuffleMaps() {  
 final List<Integer> inputRange = IntStream  
 .*range*(0, *NUM\_OF\_BITS*)  
 .boxed()  
 .collect(Collectors.*toList*());  
 IP = Algorithms.*shuffle*(inputRange);  
 }  
  
 */\*\*  
 \* Твірна функція Фейстеля  
 \* @param K Поточний ключ  
 \*/* private long desFunction(long R, long K) {  
  
 *// Кількість блоків* final int NUM\_BLOCKS = 8;  
  
 *// Обчислене розширене Е XOR K* long extendedR = Algorithms.*permutate*(R, E, *NUM\_OF\_BITS* / 2, true, true);  
  
 extendedR = extendedR ^ K;  
  
 *// Розмір одного блоку* final int BLOCK\_SIZE = E.size() / NUM\_BLOCKS;  
  
 *// Маска розміром одного блоку (6 бітів)* final long BLOCK\_MASK = Algorithms.*createBitMask*(0, BLOCK\_SIZE);  
  
 *// Масив блоків* final byte[] blocks = new byte[NUM\_BLOCKS];  
  
 for (int i = 0; i < NUM\_BLOCKS; i++) {  
  
 *// Отримання кожних 6 бітів* final long b = ((extendedR >> (i \* BLOCK\_SIZE)) & BLOCK\_MASK);  
  
 *// Обчислення номеру рядку і стовпця* final long rowNum = (Algorithms.*getBit*(b, 5) << 1) | (Algorithms.*getBit*(b, 0));  
  
 final long columnNum = (Algorithms.*getBit*(b, 4) << 3) |  
 (Algorithms.*getBit*(b, 3) << 2) |  
 (Algorithms.*getBit*(b, 2) << 1) |  
 (Algorithms.*getBit*(b, 1));  
  
 *// Отримання значення з тензору S відносно порядкового номеру блока  
 // і розрахоаного рядка і стовпця вище* blocks[i] = (byte) (S[NUM\_BLOCKS - i - 1][(int) rowNum][(int) columnNum]);  
 }  
  
 *// Чотирьох бітова маска* final long FOUR\_BIT\_MASK = Algorithms.*createBitMask*(0, 4);  
  
 *// 'Конкатенація' блоків у одне 32 бітне значення* int res = 0;  
 for (int i = 0; i < NUM\_BLOCKS; i++) {  
 res |= (int)(blocks[i] & FOUR\_BIT\_MASK) << (i \* 4);  
 }  
 res &= Algorithms.*createBitMask*(0, *NUM\_OF\_BITS* / 2);  
  
 *// Перестановка бітів відносно вектора P* return Algorithms.*permutate*(res, P, 64 / 2, true, true);  
 }  
  
 */\*\*  
 \* Функція шифрування  
 \* @param inputValue 8 байтне значення (64 біт)  
 \*/* public long cipher(Long inputValue) {  
  
 *// Перестановка вхідного числа відносно вектора IP* long T\_0 = Algorithms.*permutate*(inputValue, IP, *NUM\_OF\_BITS*, true, true);  
  
 *// Розкладення 64 бітного числа на два 32 бітних* long L = ( (T\_0 >> HALF\_IP) & HALF\_IP\_MASK );  
 long R = ( T\_0 & HALF\_IP\_MASK );  
  
 *// Цикл шифрування заміною* for (int i = 0; i < *FEISTEL\_ROUNDS*; i++) {  
 final long T = L ^ desFunction(R, keys.get(i));  
 L = R;  
 R = T;  
 }  
  
 *// Об'єднання правої і лівої частини у 64 бітне число* long combined = ( R << HALF\_IP) | L;  
  
 *// Обернена перестановка вхідного числа відносно вектора IP* return Algorithms.*permutateInverse*(combined, IP, *NUM\_OF\_BITS*, true);  
 }  
  
 */\*\*  
 \* Функція дешифрування  
 \* @param inputValue 64 бітне значення  
 \*/* public long decipher(Long inputValue) {  
  
 *// Перестановка вхідного числа відносно вектора IP* long recovered = Algorithms.*permutate*(inputValue, IP, *NUM\_OF\_BITS*, true, true);  
  
 *// Розкладення 64 бітного числа на два 32 бітних* long R = ( (recovered >> HALF\_IP) & HALF\_IP\_MASK );  
 long L = ( recovered & HALF\_IP\_MASK );  
  
 *// Цикл шифрування заміною оберненим чином* for (int i = *FEISTEL\_ROUNDS* - 1; i >= 0; i--) {  
 final long T = R ^ desFunction(L, keys.get(i));  
 R = L;  
 L = T;  
 }  
  
 *// Об'єднання правої і лівої частини у 64 бітне число* long combined = ( L << HALF\_IP) | R;  
  
 *// Обернена перестановка вхідного числа відносно вектора IP* return Algorithms.*permutateInverse*(combined, IP, *NUM\_OF\_BITS*, true);  
 }  
  
 public List<Long> cipher(List<Long> inputValues) {  
 return apply(inputValues, this::cipher);  
 }  
  
 public List<Long> decipher(List<Long> cipheredValues) {  
 return apply(cipheredValues, this::decipher);  
 }  
  
 */\*\*  
 \* Функція що приймає список чисел і поелементно застосовує їх до фукції  
 \* Повертає список значень функції  
 \* @param values Список значень  
 \* @param lambda Функція  
 \*/* private List<Long> apply(List<Long> values, LongFunction<Long> lambda) {  
 return values  
 .stream()  
 .map(lambda::apply)  
 .collect(Collectors.*toList*());  
  
 }  
  
}

Реалізував клас для демонстрації роботи алгоритму.

package solution;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;  
  
public class DesDemo {  
  
 */\*\*  
 \* Функція для перетворення тексту у список 64 бітних чисел  
 \* @param text Вхідний текст  
 \*/* public static List<Long> stringToList(String text) {  
 final List<Long> list = new ArrayList<>();  
 for (char c : text.toCharArray()) {  
 list.add((long) c);  
 }  
 return list;  
 }

*/\*\*  
 \* Функція для перетворення списку чисел у текст  
 \*/* public static String listToString(List<Long> list) {  
 final int N = list.size();  
 final StringBuilder stringBuilder = new StringBuilder();  
 for (long aLong : list) {  
 stringBuilder.append((char) aLong);  
 }  
 return stringBuilder.toString();  
 }

public static void main(String[] args) {  
 final DES des = new DES();  
  
 final Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.println("Введіть текст: ");  
  
 while (scanner.hasNext()) {  
 final String inputText = scanner.nextLine();  
 if (inputText.equals("stop")) {  
 break;  
 }  
 final List<Long> inputList = *stringToList*(inputText);  
 final List<Long> cipheredList = des.cipher(inputList);  
 final List<Long> decipheredList = des.decipher(cipheredList);  
  
 System.*out*.println("Вхідний текст: " + inputText);  
 System.*out*.println("Зашифрований текст: " + *listToString*(cipheredList));  
 System.*out*.println("Розшифрований текст: " + *listToString*(decipheredList));  
 System.*out*.println("Розшифрування пройшло успішно? " + (inputList.equals(decipheredList) ? "Так" : "Ні"));  
 }  
  
 System.*out*.println("Програму завершено");  
 }  
  
}

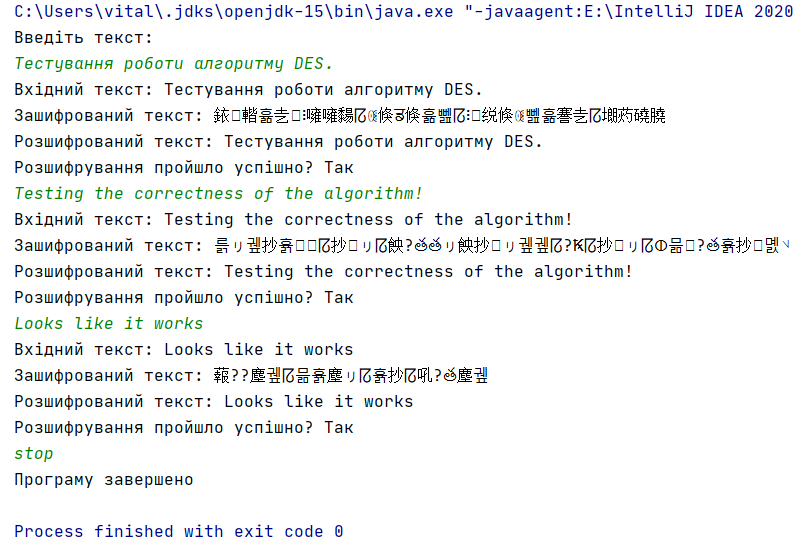


Рисунок 1. Результат виконання програми

Висновок: на цій лабораторній роботі я успішно реалізував алгоритм DES.