Final Project

Name: Hardeep Kaur Gill
Student Number: 20705251
Course: CS210 Data Structures & Algorithms

1. Overview

The goal of this project was to find 2 sentences with the most similar SHA-256 hash values. We measured similarity by counting how many hexadecimal characters in the hash matched at the same positions. This task combined elements of cryptography, random sentence generation, and efficient data structure usage.

Inflation and Bitcoin:

Originally, coins were made of precious metals like gold, which gave them intrinsic value. Later, banks began issuing banknotes that could be redeemed for gold. However, governments eventually moved to flat currencies, which are not backed by real assets. Fiat currencies derive their value from government decree and are susceptible to inflation, where the value of money decreases over time.

Bitcoin, introduced in 2009 by an anonymous group/person called “Satoshi Nakamoto”, was designed as a decentralized digital currency beyond government control. Decentralized currency is a method of transferring wealth or ownership without the need for a third party. Using the blockchain and cryptographic principles like SHA-256, Bitcoin provides a secure way to own and exchange digital assets. Unlike fiat currencies, Bitcoin is resistant to inflation due to its fixed supply and decentralized nature.

The concept of hashing is critical in many areas of computer science and finance. For instance, Bitcoin relies heavily on SHA-256, a cryptographic hash function, to ensure security and immutability. Hash functions like SHA-256 have several key properties:

- They are deterministic, meaning the same input always produces the same hash.

- They are quick to compute.
- Even a tiny change to the input creates a drastically different output.
- It is nearly impossible to reverse-engineer the input from the hash.
- It is infeasible to find two different messages with the same hash value.

In this project, the focus was one of these properties: the randomness of hash outputs and the difficulty of finding inputs (sentences) that produce the same hash value.

How I Approached the Problem:

- The program generated random sentences in 2 different formats. One is subject + verb + object, such as “The cat eats the ball.” The second one is a word text file that contains a lot of words, and random sentences are made out of them in random lengths.
- To ensure the sentences were grammatically correct, we divided words into three categories: subjects, verbs, and objects, and randomly selected one from each.
- For each generated sentence, the program computed its SHA256 hash using Java’s MessageDigest library.
- The hashes were compared character-by-character to count matching hexadecimal characters at the same positions.
- The program tracked the pair the most matches.

2. Code Structure

Overview of Code

The program’s purpose is to find 2 sentences whose SHA-256 hashes have the most matching hexadecimal characters. It is organized into multiple methods, each designed to perform a specific task. This makes the code easier to understand and maintain. The main steps include generating sentences, hashing them, comparing their hashes, and finding the best matching pair.

Main Data Structures Used and Why:

1. HashSet:
   - Used to store sentences in both the random sentence generator and the grammatically correct sentence generator.
   - Ensures all sentences are unique, avoiding duplicates that could affect the results.

2. HashMap:
   - Stores the computed hashes and their corresponding sentences for efficient comparison during the closest pair search.

3. List:
    - Used for storing the final set of unique sentences or words loaded from the dictionary.java file.

Main Algorithms are their Functionality:

1. Random Sentence Generation (in the appendix this algorithm is called in a method the generareSentences):

- Generates random sentences using words loaded from the dictionary.java file.
- Sentences are between 5 to 10 words long.
- Ensures the first word is capitalized and ends with proper punctuation.
- Avoids duplicates using a HashSet.

2. Grammatically correct sentence generation (in the appendix this algorithm is called in a method called the generateGrammaticallyCorrectSentences):

- Follows the format subject + verb + object to produce sentences that are grammatically valid.
- Randomly selects words from predefined subject, verb, and object lists.
- Uses a Hashset to ensure all generates the hash in hexadecimal format.

3. SHA-256 Hash Computation (in the appendix this algorithm is called the computeSHA256):

- Computes the SHA-256 hash for each sentence.
- Uses Java's built-in MessageDigest library to generate the hash in hexadecimal format.

4. Hexadecimal Match Counting (in the appendix it is called in a method called the countHexMatches):

- Compared two SHA-256 hashes character by character to count matching hexadecimal characters.
- User to evaluate the similarity between two hashes.

5. Closest Pair Search (in the appendix this algorithm is in a method called the findClosestHashes)

- Iterates through all sentences hashes to find the pair with the highest number of matching hexadecimal characters.
- Stores sentence-hash pairs in a HashMap for efficient lookup.
- Keeps track of the maximum number of matches and updates the closest pair accordingly.

Program Flow:

1. User Input:

- The program first asks the user whether they want random longer sentences or shorter grammatically correct sentences.

- For grammatically correct sentences, the user specifies how many to generate (more than 2 required).

2. Sentence Generation:

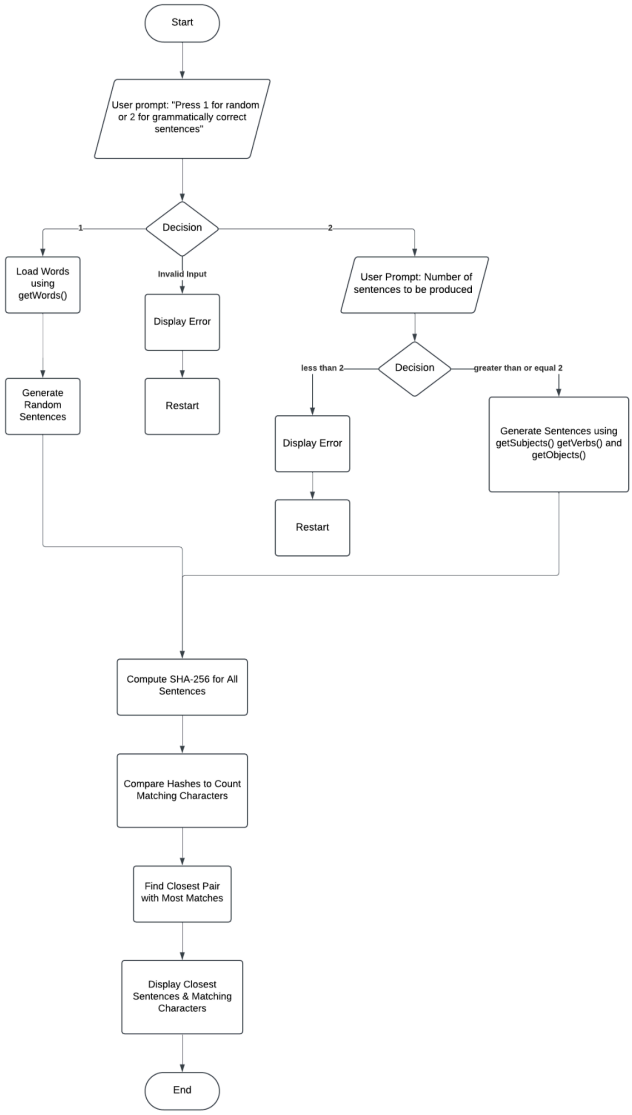
- Depending on the input, the program either:
  - Generates random sentences with varied lengths.
  - Creates grammatically correct sentences using subject, verb, and object lists.

3. Hashing and Comparison:

- Each generated sentence is converted into a SHA-256 hash.
- The program compares all hashes and counts how many characters match in the same positions.
- It identifies the two sentences with the highest number of matches.

4. Output:

- The program prints the 2 closest sentences and the number of matching characters.



3. Time and Space Complexity

Time and space complexity have to do with how long it takes to execute (time complexity) and how much memory it uses (space complexity). Below is an explanation in simple terms:

Time Complexity

The program consists of three main steps, each contributing to the total execution time:

Generating Sentences:

- The program creates sentences by randomly picking words from a dictionary. Each sentence consists of 5 to 10 words. This process is repeated for all sentences generated.
- The time it takes depends on the number of sentences (N) and the average number of words per sentence (M).
- **Time Complexity**: O(N×M).

Computing SHA-256 Hashes:

- Each sentence is converted into a SHA-256 hash. The time it takes depends on the length of the sentence (L), as the algorithm processes each character of the input.
- **Time Complexity**: O(N×L), where N is the number of sentences.

Finding the Closest Pair:

- The program compares every pair of hashes to find the two with the most matching characters. For N sentences, this involves approximately N×N comparisons.
- Each comparison takes a constant amount of time (H), as SHA-256 hashes are always 64 hexadecimal characters long.
- **Time Complexity**: O(N^2×H).

Overall, the slowest step is finding the closest pair, so the total time complexity is dominated by **O(N^2×H)**.

Space Complexity

The program requires memory for storing sentences, their hashes, and the dictionary:

Storing Sentences:

All sentences are stored in memory. If each sentence has an average length (W) and there are N sentences, this requires O(N×W) space.

Storing Hashes:

Each hash is 64 characters long. For N sentences, storing hashes requires $O(N \times 64)$ space.

Storing the Dictionary:

The dictionary of words is loaded into memory. If it contains D words, each with an average length (W_{word}), this requires $O(D \times W_{word})$ space.

Overall Space Complexity: $O(N \times W + N \times 64 + D \times W_{word})$.

Comparing all pairs of sentences to find the closest match is computationally expensive as the number of sentences increases. The memory usage is influenced by the number of sentences, their lengths, and the size of the dictionary. For small datasets, the program performs efficiently, but for larger datasets, the quadratic growth in comparison time makes it slower.

4. Learning Outcomes

During the course of this project, I encountered several challenges and gained valuable insights, which really contributed to my understanding of hashing algorithms, data structures and randomness in programming. Below is a summary of the key learning outcomes.

Challenges Faced and Solutions:

1. Duplicated Sentences in Sentence Generation:

- Initially, I noticed that the sentence generator was producing duplicate sentences, which affected the diversity of outputs and the accuracy of hash comparisons.

- **Solution:** I replaced the List used for storing sentences with a HashSet. This ensured that only unique sentences were generated, as HashSet inherently prevents duplicate entries. This change significantly improved the efficiency of the program.

2. Placement of the Hash Map Update:

- During the implementation of the findClosestHashes method, I faced an issue where the program was incorrectly identifying the best matching pair. The issue was, despite using hash sets, there would still be duplicate sentences. The output would be the same sentence twice with 64 matches, which is clearly the wrong result. After a lot of attempts, I realized that updating the hash map had to be done after the inner loop to avoid overwriting the sentence being compared.
- Adding the sentence-hash pair in the hashmap after the inner loop ensures that the current sentence is only added to the hashmap once all comparisons with existing hashes are complete. This avoids comparing the sentence with itself, which would artificially inflate the match count. (Refer to line 142 in the appendix for the exact code snippet).

Insights Gained

1. Random Sentence Generation:

- The use of random selection from word lists (subjects, verbs, and objects) ensures that each run of the program produces a different set of sentences. This randomness is achieved through Java's Random class, which provides a pseudo-random generator.
- This feature not only guarantees fresh outputs on every execution but also provides a wide variety of sentence structures, which is essential for finding the best hash match.

2. Understanding SHA-256 Hashing:

- Through this project, I gained a deeper appreciation for the properties of SHA-256:
    - Even a small change in input drastically changes the hash output.
    - The same input always produces the same hash, making comparisons reliable.
- I also learned how to efficiently compute and store hashes using Java's MessageDigest class.

3. Hexadecimal Character Matching:

- Comparing the SHA-256 hashes character by character to count matching hexadecimal characters provided a standard way to measure hash similarity. This approach emphasized how difficult it is to find meaningful similarities between 2 hashes due to their cryptographic design.

Achievements:

1. Finding 16 Matches:

- After debugging and refining the code, the program successfully identified two sentences matching hexadecimal characters in their SHA-256 hashes. This was a significant achievement given the inherent randomness of hash outputs and the cryptographic complexity of SHA-256.

2. Grammatically correct sentence generation:

- By designing a structured sentence generator (subject + verb + object), I ensured that all sentences were grammatically and semantically valid. This fulfilled the project requirement and improved the quality outputs.

3. Efficient Data Structures:

- The use of HashSet for uniqueness and HashMap for quick lookup during hash comparisons demonstrated the importance of selecting appropriate data structures for specific tasks.

5. Conclusion

This project was a great learning experience in applying data structures, hashing algorithms, and randomness to solve a real-world problem. The task of finding 2 sentences with the most similar SHA-256 hashes taught me how challenging it is to work with cryptographic functions. The randomness in sentence generation provided unique results each time (for the program

users choosing 1 when prompted to pick a type of sentence), while the use of grammatically correct sentences ensured that the outputs were meaningful.

I also learn the importance of selecting the right data structures such as HashSet and HashMap, to handle issues like duplicate sentences and efficient comparisons. Debugging the code and refining the logic to ensure accurate hash matches helped me improve my problem-solving skills.

Achieving 16 matching characters in the hashes was a satisfying result, demonstrating that the approach was effective. 17 matches were found using the second type of sentences (the ones semantically correct). This project has helped me better understand cryptography and hashing while giving me hands-on experience with Java programming and algorithm design.

6. Appendix

BitcoinHashingStrategy.java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.*;

//faced a bug with sentences being duplicates, in order to avoid that, I
decided to use hashsets for the sentences instead of lists.
//in the document need to also mention how the random sentences always
give a random output and why
//line 142 is really important to placed after the for loop and explain
why

public class BitcoinHashingStrategy{
    public static void main(String [] args){
        System.out.println("Press 1 for random longer sentences or 2 for
shorter more grammatically correct sentences.");
        System.out.println("Enter types of sentences: ");

        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();

        if(num == 1){
            String[] words = loadDictionary();
            List<String> sentences = generateSentences(words);
```

```
            //find the closest pair
            String[] closestPair = findClosestHashes(sentences);

            System.out.println("\nClosest Sentences: ");
            System.out.println(closestPair[0]);
            System.out.println(closestPair[1] + "\n\n");
        }
        else if(num == 2){
            System.out.println("Enter the number of sentences you want
(more than 2 ideally): ");
            int count = sc.nextInt();
            if(count > 1){
                List<String> sentences =
generateGrammaticallyCorrectSentences(count);

                String[] closestPair = findClosestHashes(sentences);

                System.out.println("\nNumber of generated sentences: " +
count);

                System.out.println("\nClosest Sentences: ");
                System.out.println(closestPair[0]);
                System.out.println(closestPair[1] + "\n\n");
            } else {
                System.out.println("Need more than 2 sentences please. Try
again by running the code from the start.");
            }
        } else {
            System.out.println("Please run this code again to retry.");
        }

        sc.close();


    }

    public static String[] loadDictionary(){
        List<String> wordList = Dictionary.getWords();
        return wordList.toArray(new String[0]);
    }
```

```
    public static List<String> generateSentences(String [] words){
        Set<String> sentences = new HashSet<>();
        Random random = new Random();


        for(int i = 0; i < 1000; i++){
            StringBuilder sentence = new StringBuilder();
            int sentenceLength = 5+random.nextInt(6);  //maximum of 10
words per sentence

            for(int j = 0; j <  sentenceLength; j++){
                String word = words[random.nextInt(words.length)];

                //first word to be capitalised
                if(j == 0){
                    word = word.substring(0,1).toUpperCase() +
word.substring(1).toLowerCase();
                } else {
                    word = word.toLowerCase();
                }

                //add word to sentence
                sentence.append(word).append(" ");
            }
            //punctuation in the end
            sentence.setCharAt(sentence.length() -1, '.');
            sentences.add(sentence.toString().trim());
        }
        return new ArrayList<>(sentences);
    }

    public static List<String> generateGrammaticallyCorrectSentences(int
count){
        Set<String> sentences = new HashSet<>();
        Random rn = new Random();
        List<String> subjects = Dictionary.getSubjects();
        List<String> verbs = Dictionary.getVerbs();
        List<String> objects = Dictionary.getObjects();

        for(int i = 0; i < count; i++){
```

```
            String subject = subjects.get(rn.nextInt(subjects.size()));
            String verb = verbs.get(rn.nextInt(verbs.size()));
            String object = objects.get(rn.nextInt(objects.size()));

            String sentence = subject + " " + verb + " " + object + ".";
            sentences.add(sentence);
        }
        return new ArrayList<>(sentences);
    }

    //given code
    public static String computeSHA256(String input){
        try{
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(input.getBytes());
            StringBuilder hexString = new StringBuilder();
            for(byte b : hash){
                String hex = Integer.toHexString(0xff & b);
                if(hex.length() == 1) hexString.append(0);
                hexString.append(hex);
            }
            return hexString.toString();
        } catch(NoSuchAlgorithmException e){
            throw new RuntimeException(e);
        }
    }

    public static String[] findClosestHashes(List<String> sentences){
        String[] closestPair = new String[2];
        int maxMatches = 0;

        //this hashmap stores the hashes with their corresponding
sentences
        Map<String, String> hashToSentence = new HashMap<>();

        for(String sentence : sentences) {
            //System.out.println(sentence);
            String hash = computeSHA256(sentence);

            for(String existingHash : hashToSentence.keySet()) {
```

```
                int matches = countHexMatches(hash, existingHash);

                //check for max matches
                if(matches > maxMatches) {
                    maxMatches = matches;
                    closestPair[0] = sentence;
                    closestPair[1] = hashToSentence.get(existingHash);

                }
            }
line142     hashToSentence.put(hash, sentence); //this is important to be
here
        }
        System.out.println("\n\n - Most matches:" + maxMatches);
        return closestPair;
    }


    public static int countHexMatches(String hash1, String hash2){
        int matches = 0;

        for(int i = 0; i < hash1.length(); i++) {
            if(hash1.charAt(i) == hash2.charAt(i)){
                matches++;
            }
        }
        return matches;
    }
}
```

Dictionary.java

```
import java.io.*;
import java.util.*;

public class Dictionary {
    public static List<String> getWords(){
        List<String> words = new ArrayList<>();
        String filePath = "C:\\Users\\gillh\\OneDrive\\Documents\\Java
Projects\\SHA-256 Problem\\words.txt";
```

```
        try(BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            while((line = reader.readLine()) != null) {
                words.add(line.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return words;
    }

    public static List<String> getSubjects() {
        return Arrays.asList(
            "The cat", "A dog", "The bird", "The girl", "The boy",
            "The chef", "My friend", "The teacher", "The scientist",
            "The athlete", "The artist", "The musician", "The pilot",
            "The astronaut", "The baby", "My neighbor", "The student",
            "The doctor", "The gardener", "The magician", "The engineer"
        );
    }

    public static List<String> getVerbs() {
        return Arrays.asList(
            "eats", "chases", "sees", "likes", "jumps over",
            "builds", "writes", "paints", "cooks", "drives",
            "teaches", "learns", "studies", "flies", "explores",
            "throws", "catches", "notices", "fixes", "admires"
        );
    }

    public static List<String> getObjects() {
        return Arrays.asList(
            "a ball", "a mouse", "a tree", "a car", "a book",
            "the sky", "the stars", "a painting", "a house",
            "a garden", "a piano", "a spaceship", "a computer",
            "a recipe", "using a map", "a telescope", "a robot",
            "a kite", "a lamp", "a masterpiece"
        );
```

```
        }
}
```

words.txt (provided as a file in the repository - contains 10 000 words)