

20232810008

Sedanur Gültekin

ECON381 Fall 2024

Homework Assignment 3

INTRODUCTION

In the current digital environment, security calls for solutions that both improve user experience and provide safeguards. In this regard, it has become imperative to provide secure and simply accessible passwords for devices with few input possibilities, including virtual keyboards. Based on user input and distance measurements, this research develops a straightforward yet safe technique for creating eight-character passwords. This algorithm is intended to satisfy security standards and be easy to use.

Question 1

Switch distances may be measured using the Manhattan Distance, often known as the grid-based distance. The total of the vertical and horizontal motions between two places is determined by this metric. For instance, the total number of vertical and horizontal steps needed to switch between characters.

On a virtual keyboard, the arrangement of the keys is two-dimensional. As a result, distances may be measured using the Manhattan Distance. The total of the vertical and horizontal lengths between two points is known as the Manhattan distance.

In terms of mathematics:

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

Since the arrow keys are used to describe motions, this metric is acceptable. We employ the Manhattan Distance metric to calculate the separations between keys. This measure determines how many vertical and horizontal strokes are needed to go between two keys on a keyboard layout that resembles a grid. Because it faithfully captures arrow key motions, this measure is ideal for the situation.

This measure is appropriate for the situation since it faithfully captures arrow key motions. For discrete grid-based motions, other metrics like Euclidean Distance or Chebyshev Distance are less significant, but they may be employed in other situations.

Question 2

We suggest two possible advanced data structures to represent the keyboard layout:

1. Tree Structure: Every key is viewed as a node in a network, with edges denoting legitimate movements that are two to three distances apart. This method guarantees that the search for legitimate movements is simple and effectively models relationships.

For instance:

'd' -> ['s', 'f', 'e']

's' -> ['d', 'w', 'x']

...

2. Hash Map with Nested Lists: This method provides $O(1)$ lookup time for identifying acceptable movements by directly mapping keys to a list of them.

For instance:

```
HashMap<Character, List<Character>> validMoves = new HashMap<>();
```

```
validMoves.put('d', Arrays.asList('s', 'f', 'e'));
```

This structure becomes permanently unneeded when the distances have been determined. We can employ a system that associates each character with a list for contemporary movements. A structure mapping each key to its valid motions can be used in place of the layout after the distances between keys have been precalculated.

The keyboard layout can be represented using a matrix data structure or a 2D array. Each character's location on the keyboard may be ascertained using this structure. After computing the distances, this structure may be swapped out with another one because the distances don't vary dynamically.

Question 3

For simplicity and speed, we suggest use a HashMap in conjunction with Lists to map each key to its valid movement. However, a Tree Map or Graph Structure could be better suitable in situations that call for more dynamic or hierarchical interactions.

This problem is especially well-suited for the HashMap because:

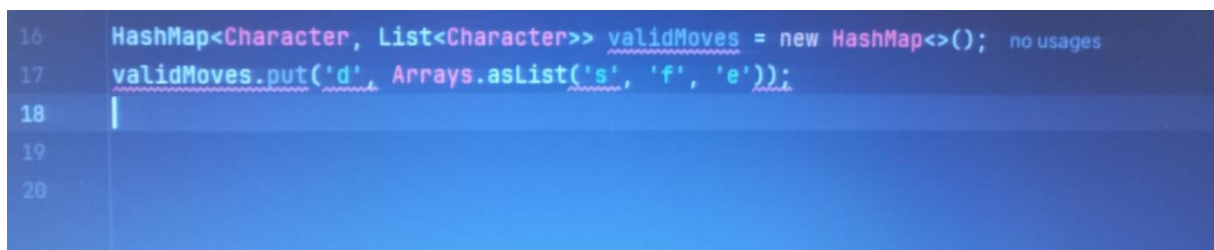
It permits obtaining valid movements for a key with an average time complexity of $O(1)$. Initializing and populating with precomputed valid movements is simple.

The current movements can be represented using a hash map. A list of keys that are two to three movements distant is mapped to each key. The current movements can be represented using a hash map. A list of keys that are two to three movements distant is mapped to each key.

An illustration of a hash map: **(Below is the ss taken from Java)**

```
HashMap<Character, List<Character>> validMoves = new HashMap<>();
```

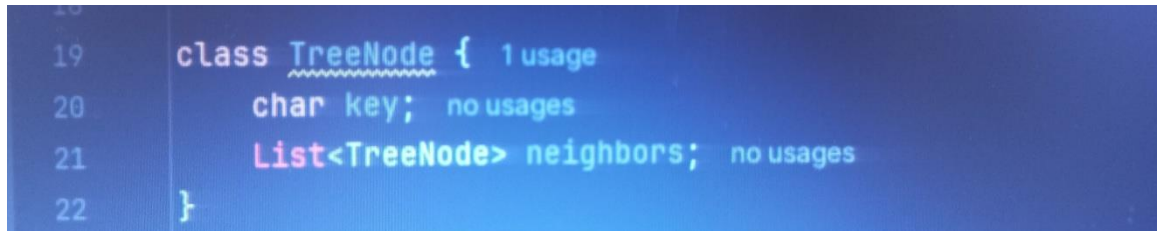
```
validMoves.put('d', Arrays.asList('s', 'f', 'e'));
```



```
16  HashMap<Character, List<Character>> validMoves = new HashMap<>(); no usages
17  validMoves.put('d', Arrays.asList('s', 'f', 'e'));
18
19
20
```

Regarding a tree-based strategy: **(Below is the ss taken from Java)**

```
class TreeNode {  
    char key;  
    List<TreeNode> neighbors;  
}
```

A screenshot of a code editor showing the definition of the TreeNode class. The code is as follows:

```
19 class TreeNode { 1 usage  
20     char key; no usages  
21     List<TreeNode> neighbors; no usages  
22 }
```

Each key's valid movements (two to three distances away) are stored and retrieved using the dictionary structure. Without recalculating distances, it is crucial for effectively creating passwords in real-time.

The main function of the dictionary (also known as a hash map) is to associate each key with its current transactions. This data structure, To effectively keep a list of movements that are valid for each key that has been computed When creating a password, it is essential to make sure that valid movements are recovered $O(1)$.

The technique is quicker and more dependable since it uses a dictionary instead of recalculating distances in real time.

For example, when the dictionary has been filled in:

```
validMoves.get('a'); // Returns the list ['s', 'd', 'w']
```

This feature guarantees the algorithm's effectiveness and usability on hardware with constrained processing power.

Question 4 (EXPLANATION: (I will add the Java file to Github. The code I created is in the MAIN section of the java file).(there are ss of the code below.) (I will also add the java code as a pdf to github.)

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

```
public class PasswordGenerator {
```

```
    private static final char[][] KEYBOARD = {  
        {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'},
```

```
        {'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p'},  
        {'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l'},  
        {'z', 'x', 'c', 'v', 'b', 'n', 'm'}  
    };
```

```
private static Map<Character, List<Character>> validMovesMap = new HashMap<>();
```

```
public static void main(String[] args) {
```

```
    Scanner scanner = new Scanner(System.in);
```

```
    System.out.print("Enter the first character: ");
```

```
    char firstChar = scanner.next().charAt(0);
```

```
    if (isValidCharacter(firstChar)) {
```

```
        initializeValidMovesMap();
```

```
        String password = generatePassword(firstChar);
```

```
        System.out.println("Generated password: " + password);
```

```
    } else {
```

```
        System.out.println("Invalid character. Please enter a digit or a lowercase English letter.");
```

```
    }
```

```
}
```

```
private static boolean isValidCharacter(char ch) {
```

```
    return (ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z');
```

```
}
```

```
private static String generatePassword(char startChar) {
```

```

StringBuilder password = new StringBuilder(String.valueOf(startChar));

Random random = new Random();

for (int i = 1; i < 8; i++) {

    char lastChar = password.charAt(i - 1);

    List<Character> validMoves = validMovesMap.get(lastChar);

    if (validMoves != null && !validMoves.isEmpty()) {

        int randomIndex = random.nextInt(validMoves.size());

        password.append(validMoves.get(randomIndex));

    } else {

        password.append(startChar);

    }

}

return password.toString();

}

private static void initializeValidMovesMap() {

    for (int i = 0; i < KEYBOARD.length; i++) {

        for (int j = 0; j < KEYBOARD[i].length; j++) {

            char key = KEYBOARD[i][j];

            List<Character> validMoves = new ArrayList<>();

            populateValidMoves(i, j, key, validMoves);

            validMovesMap.put(key, validMoves);

        }

    }

}

```

```
}
```

```
private static void populateValidMoves(int x, int y, char currentKey, List<Character> validMoves) {
```

```
    for (int i = 0; i < KEYBOARD.length; i++) {
```

```
        for (int j = 0; j < KEYBOARD[i].length; j++) {
```

```
            char otherKey = KEYBOARD[i][j];
```

```
            int distance = calculateDistance(x, y, i, j);
```

```
            if (distance >= 2 && distance <= 3) {
```

```
                validMoves.add(otherKey);
```

```
            }
```

```
        }
```

```
    }
```

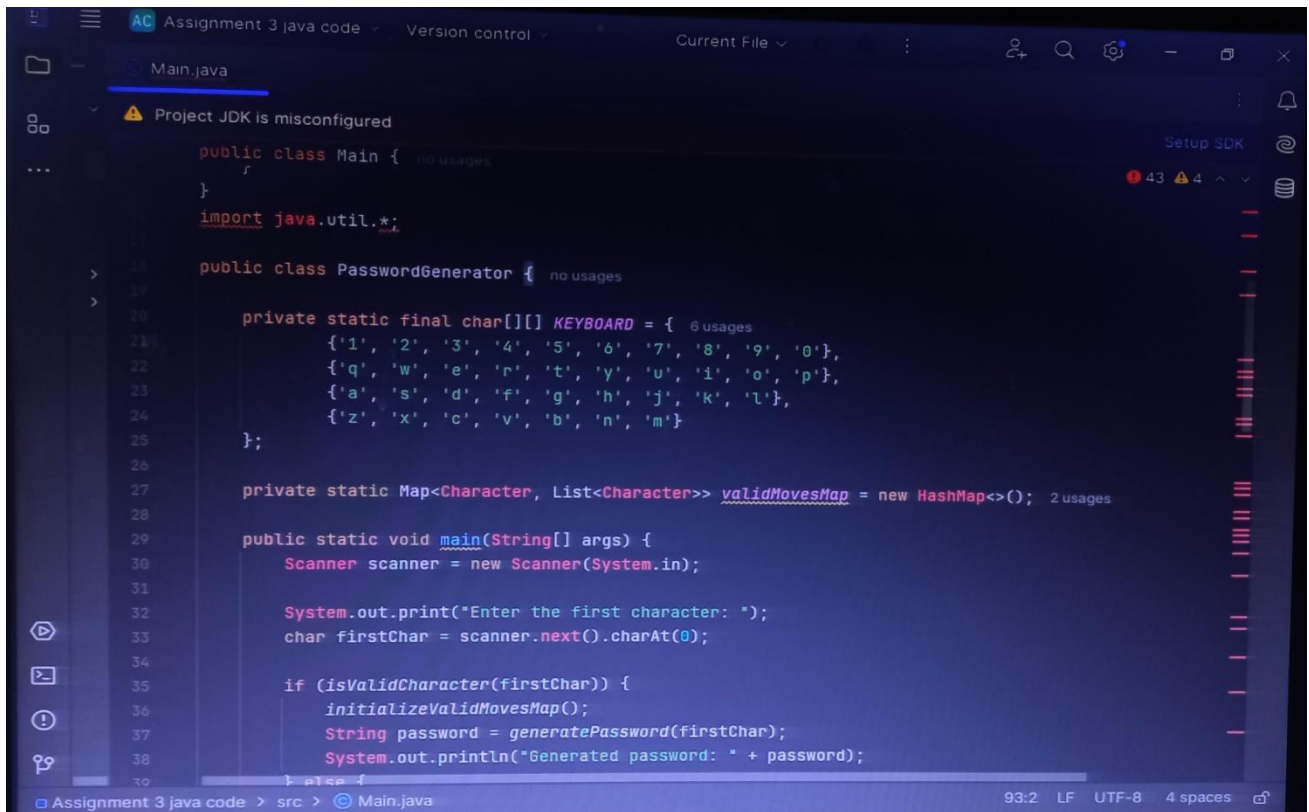
```
}
```

```
private static int calculateDistance(int x1, int y1, int x2, int y2) {
```

```
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
```

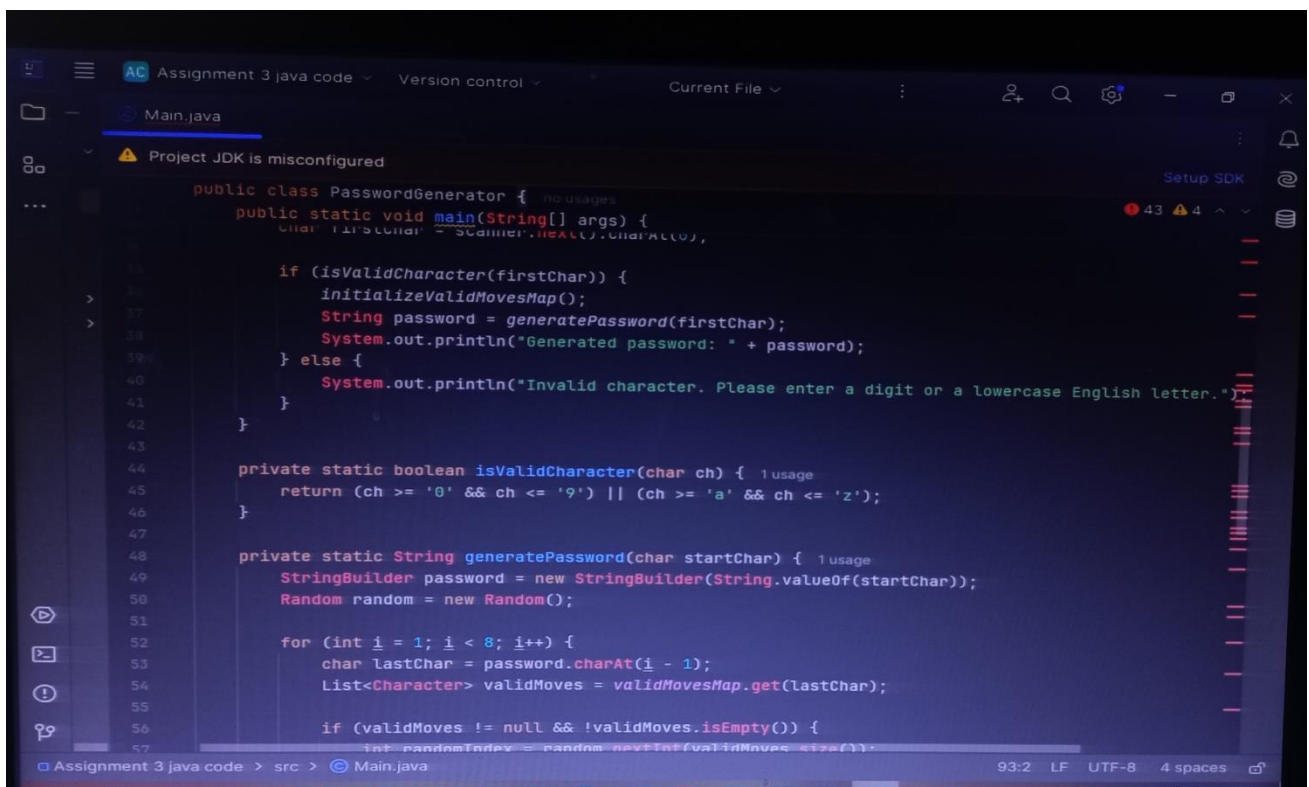
```
}
```

```
}
```



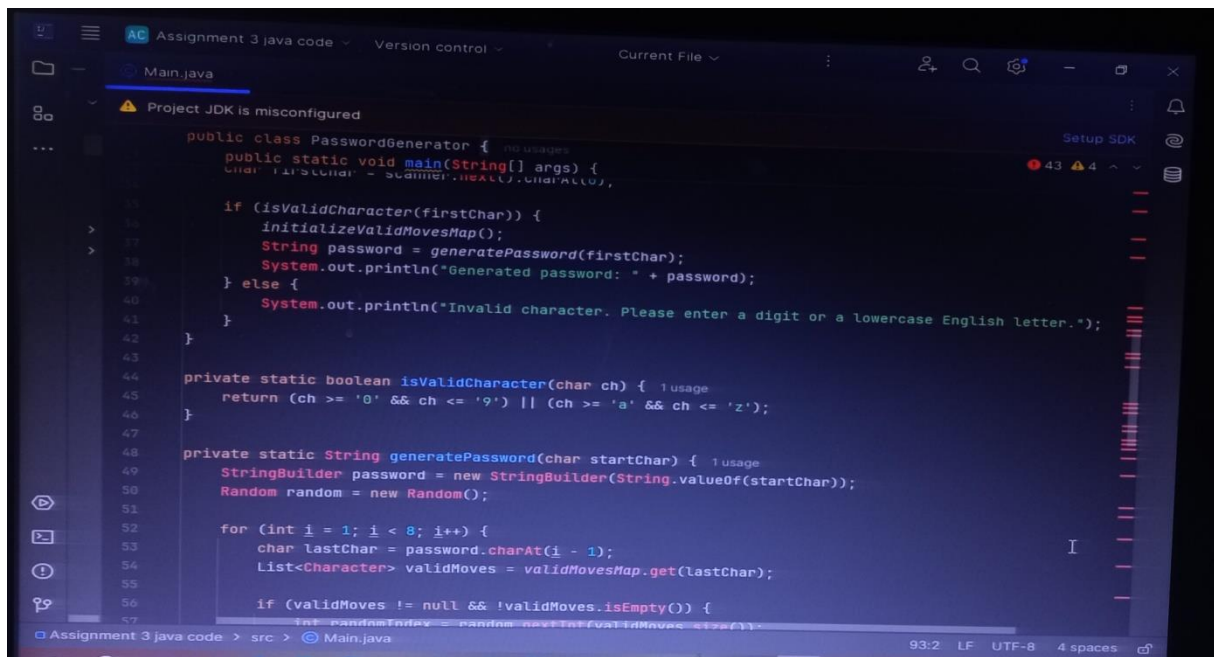
```
public class Main {  
    ...  
}  
  
import java.util.*;  
  
public class PasswordGenerator {  
    ...  
    private static final char[][] KEYBOARD = {  
        {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'},  
        {'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p'},  
        {'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l'},  
        {'z', 'x', 'c', 'v', 'b', 'n', 'm'}  
    };  
    ...  
    private static Map<Character, List<Character>> validMovesMap = new HashMap<>();  
    ...  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Enter the first character: ");  
        char firstChar = scanner.next().charAt(0);  
  
        if (isValidCharacter(firstChar)) {  
            initializeValidMovesMap();  
            String password = generatePassword(firstChar);  
            System.out.println("Generated password: " + password);  
        } else {  
            ...  
        }  
    }  
}
```

First SS



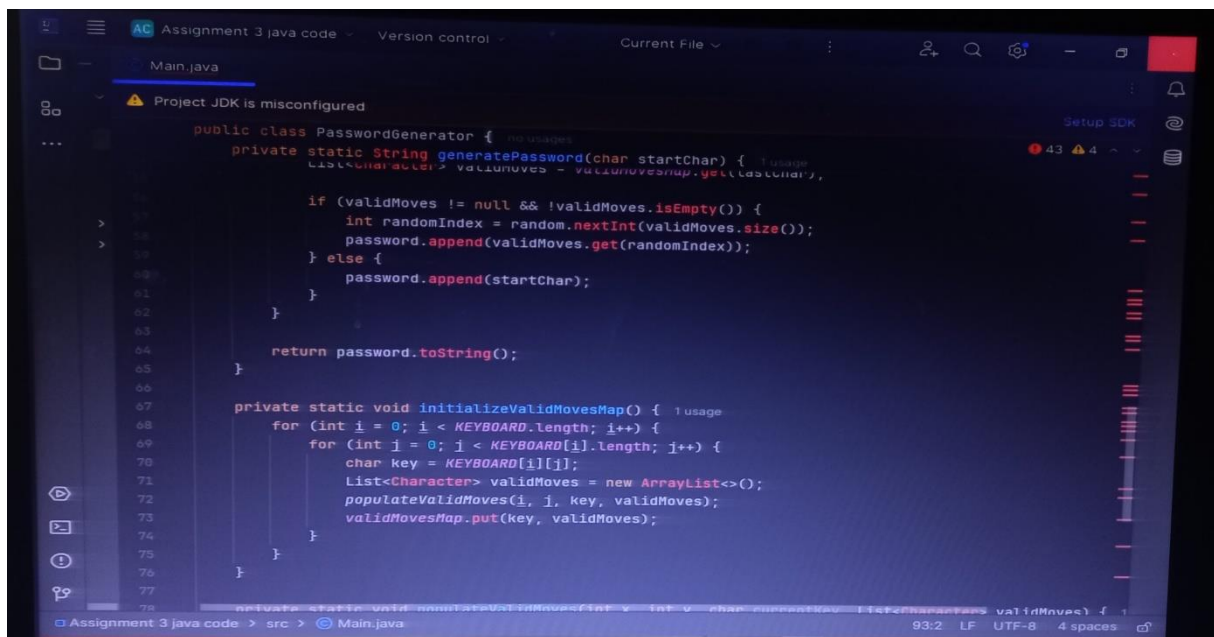
```
public class PasswordGenerator {  
    ...  
    public static void main(String[] args) {  
        char firstChar = scanner.next().charAt(0);  
  
        if (isValidCharacter(firstChar)) {  
            initializeValidMovesMap();  
            String password = generatePassword(firstChar);  
            System.out.println("Generated password: " + password);  
        } else {  
            System.out.println("Invalid character. Please enter a digit or a lowercase English letter.");  
        }  
    }  
  
    private static boolean isValidCharacter(char ch) {  
        return (ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z');  
    }  
  
    private static String generatePassword(char startChar) {  
        StringBuilder password = new StringBuilder(String.valueOf(startChar));  
        Random random = new Random();  
  
        for (int i = 1; i < 8; i++) {  
            char lastChar = password.charAt(i - 1);  
            List<Character> validMoves = validMovesMap.get(lastChar);  
  
            if (validMoves != null && !validMoves.isEmpty()) {  
                int randomIndex = random.nextInt(validMoves.size());  
                ...  
            }  
        }  
    }  
}
```

Second SS



```
public class PasswordGenerator {  
    public static void main(String[] args) {  
        char firstChar = Scanner.next().charAt(0);  
  
        if (isValidCharacter(firstChar)) {  
            initializeValidMovesMap();  
            String password = generatePassword(firstChar);  
            System.out.println("Generated password: " + password);  
        } else {  
            System.out.println("Invalid character. Please enter a digit or a lowercase English letter.");  
        }  
    }  
  
    private static boolean isValidCharacter(char ch) {  
        return (ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z');  
    }  
  
    private static String generatePassword(char startChar) {  
        StringBuilder password = new StringBuilder(String.valueOf(startChar));  
        Random random = new Random();  
  
        for (int i = 1; i < 8; i++) {  
            char lastChar = password.charAt(i - 1);  
            List<Character> validMoves = validMovesMap.get(lastChar);  
  
            if (validMoves != null && !validMoves.isEmpty()) {  
                int randomIndex = random.nextInt(validMoves.size());  
                password.append(validMoves.get(randomIndex));  
            } else {  
                password.append(startChar);  
            }  
        }  
  
        return password.toString();  
    }  
  
    private static void initializeValidMovesMap() {  
        for (int i = 0; i < KEYBOARD.length; i++) {  
            for (int j = 0; j < KEYBOARD[i].length; j++) {  
                char key = KEYBOARD[i][j];  
                List<Character> validMoves = new ArrayList<>();  
                populateValidMoves(i, j, key, validMoves);  
                validMovesMap.put(key, validMoves);  
            }  
        }  
    }  
  
    private static void populateValidMoves(int i, int j, char key, List<Character> validMoves) {  
        if (i > 0) validMoves.add(KEYBOARD[i-1][j]);  
        if (i < KEYBOARD.length-1) validMoves.add(KEYBOARD[i+1][j]);  
        if (j > 0) validMoves.add(KEYBOARD[i][j-1]);  
        if (j < KEYBOARD[i].length-1) validMoves.add(KEYBOARD[i][j+1]);  
    }  
}
```

Third SS



```
private static String generatePassword(char startChar) {  
    List<Character> validMoves = validMovesMap.get(startChar);  
  
    if (validMoves != null && !validMoves.isEmpty()) {  
        int randomIndex = random.nextInt(validMoves.size());  
        password.append(validMoves.get(randomIndex));  
    } else {  
        password.append(startChar);  
    }  
  
    return password.toString();  
}  
  
private static void initializeValidMovesMap() {  
    for (int i = 0; i < KEYBOARD.length; i++) {  
        for (int j = 0; j < KEYBOARD[i].length; j++) {  
            char key = KEYBOARD[i][j];  
            List<Character> validMoves = new ArrayList<>();  
            populateValidMoves(i, j, key, validMoves);  
            validMovesMap.put(key, validMoves);  
        }  
    }  
}
```

Fourth SS


```
public class PasswordGenerator {  
    private static void initializeValidMovesMap() {  
        populateValidMoves(1, 1, KEY, validMoves);  
        validMovesMap.put(key, validMoves);  
    }  
  
    private static void populateValidMoves(int x, int y, char currentKey, List<Character> validMoves) {  
        for (int i = 0; i < KEYBOARD.length; i++) {  
            for (int j = 0; j < KEYBOARD[i].length; j++) {  
                char otherKey = KEYBOARD[i][j];  
                int distance = calculateDistance(x, y, i, j);  
                if (distance >= 2 && distance <= 3) {  
                    validMoves.add(otherKey);  
                }  
            }  
        }  
    }  
  
    private static int calculateDistance(int x1, int y1, int x2, int y2) {  
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);  
    }  
}
```

Fifth SS

```
public class PasswordGenerator {  
    private static void initializeValidMovesMap() {  
        populateValidMoves(1, 1, KEY, validMoves);  
        validMovesMap.put(key, validMoves);  
    }  
  
    private static void populateValidMoves(int x, int y, char currentKey, List<Character> validMoves) {  
        for (int i = 0; i < KEYBOARD.length; i++) {  
            for (int j = 0; j < KEYBOARD[i].length; j++) {  
                char otherKey = KEYBOARD[i][j];  
                int distance = calculateDistance(x, y, i, j);  
                if (distance >= 2 && distance <= 3) {  
                    validMoves.add(otherKey);  
                }  
            }  
        }  
    }  
  
    private static int calculateDistance(int x1, int y1, int x2, int y2) {  
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);  
    }  
}
```

Sixth SS

Question 5

We determine acceptable movements (2–3 moves away) for the following keys using the Manhattan Distance:

Key	Valid Moves
a	S,d,w
f	d,g,v
h	g,j,n
8	5,7,9,i
0	g,o,p
p	o,L,0

-a Key

Available Movements:

Definition:

S: directly beneath the a key.

d: next to the a key on the right.

W: over the "a" key.

-f Key

Available Movements:

Definition:

d: the key to the left of the f.

g: to the f key's right.

v: directly beneath the f key.

-h Key

Available Movements:

Definition:

g: the key to the left of h.

j: the key to the right of h.

n: directly beneath the h key.

-8 Key

Available Movements:

Definition:

5: The diagonal of the 8 key at the bottom left.

7: To the left of the key for 8.

9: The key to the right of the 8.

I: A little bit above the 8 key.

-0 Key

Definition:

9: To the left of the zero key.

o: To the right of the zero key.

p: the 0 key's upper right diagonal.

-p Key

Definition:

o: Located to the left of the p key.

l: at the right side of the p key.

0: at the p key's lower left diagonal.

CONCLUSION

To sum up, our approach offers a practical way to enhance virtual keyboard security and user experience. The algorithm's efficiency and usability are supported by the use of mathematical measurements like Manhattan Distance and the use of suitable data formats. This study may be used as a foundation for further research and is a significant step in ensuring security in devices with constrained hardware resources.