Consider the following 3×3 grid.

	col 0	col 1	col 2
row 0	0	1	2
row 1	3	4	5
row 2	6	7	8

Remark 0.1. Note that in this exercise, indexing always starts from 0.

Exercise 1

The row of an index i in an $m \times n$ is given by the floor division $i//n$, and the column of i is given by the modulus $i\%n$. For a 3×3 matrix, we have $n = 3$. So for an index i, the quotient $i//3$ gives the row of i, and $i\%3$ gives the column of i.

The functions row(i) and col(i) take as argument an integer i between 0 and 8 inclusive and return the quotient $i//3$ representing the row of i and the remainder $i\%3$ representing the column of i respectively.

For example, index 5 is in row $5//3 = 1$, which is the second row. It is also in col $5\%3 = 2$, which is the third column.

```
def row(i : int)->int:
    """
    Determine the row of index ‘i‘ in a 3x3 grid.

    Parameters
    ----------
    i : int
        An index between 0 and 8.

    Returns
    -------
    int
        The row number (0, 1, or 2) of the index ‘i‘.

    Example
    -------
    row(5) -> 1 (Second row)

    Raises
    ------
    TypeError
        If ‘i‘ is not an integer.
    IndexError
        If ‘i‘ is not between 0 and 8 inclusive.
    """

    if not isinstance(i, int):
        raise TypeError(f"Sorry. ’{i}’ must be an integer")

    if i not in range(0, 9):
        raise IndexError(f"Sorry. ’{i}’ must be a number between 0 and 8 inclusive.")


    return i//3

def col(i:int) -> int:
    """
    Determine the column of index ‘i‘ in a 3x3 grid.

    Parameters
    ----------
    i : int
        An index between 0 and 8.

    Returns
    --------
    int
        The column number (0,1, or 2) of the index ’i’.

    Example
    -----------------------
    col(5) -> 2 (Third column)

    Raises
    ------
    TypeError
        If ‘i‘ is not an integer.
    IndexError
        If ‘i‘ is not between 0 and 8 inclusive.
    """
    if not isinstance(i, int):
        raise TypeError(f"Sorry. ’{i}’ must be an integer")

    if i not in range(0, 9):
        raise IndexError(f"Sorry. ’{i}’ must be a number between 0 and 8 inclusive.")


    return i%3
```

Listing 1: Exercise 1

Exercise 2

The function $intermediate\_node(i : int,\ j : int)$ aims to determine whether there is an intermediate node between two given nodes i and j in a 3×3 grid, with nodes numbered 0 through 8, and if so, returns the intermediate node.

The function begins by initilizing two variables col_set and row_set. The varialbe col_set is assigned the set of column indices $\{col(i),\ col(j)\}$ of nodes the nodes i and j. The variable row_set is assigned the set $\{row(i),\ row(j)\}$ of row indices of the nodes i and j.

1. To ensure i and j are in different rows, we need $abs(row(i) - row(j)) == 2$. This implies the length of row_set is 2.

2. To ensure i and j are in different columns, we need $abs(col(i) - col(j)) == 2$. This implies the length of col_set is 2.

3. To ensure i and j are in the same rows, the length of row_set must be 1.

4. To ensure i and j are in the same columns, the length of col_set must be 1.

Now, if $abs(row(i) - row(j)) == 2$ and $len(col\_set) == 1$, then i and j are in the same column with an intermediate node $\frac{i+j}{2}$ between them.

Next, if $abs(col(i) - col(j)) == 2$ and $len(row\_set) == 1$, then i and j are in the same row with an intermediate node $\frac{i+j}{2}$ between them.

Finally, if $len(row_set) == 2$, $len(col_set) == 2$, $abs(row(i) - row(j)) == 2$ and $abs(col(i) - col(j)) == 2$ the i and j lie in a diagonal, with an intermediate node $\frac{i+j}{2}$ between them.

In summary, three sets of conditions are tested. If each set is satisfied, an intermediate node is returned by the function. Othersise, it returns -1.

```python

def intermediate_node(i: int, j: int) -> int:
    """
    Determine the intermediate node ‘k‘ lying between ‘i‘ and ‘j‘ in a 3x3 grid, if it exists.

    Parameters
    ----------
    i : int
        An index between 0 and 8.
    j : int
        An index between 0 and 8.

    Returns
    -------
    int
```

```
        The node ‘k‘ lying between ‘i‘ and ‘j‘ if it exists, otherwise -1.

    Raises
    ------
    TypeError
        If ‘i‘ or ‘j‘ is not an integer.
    IndexError
        If ‘i‘ or ‘j‘ is not between 0 and 8 inclusive.

    Examples
    --------
    >>> intermediate_node(0, 8)
    4
    >>> intermediate_node(1, 5)
    -1
    """
    if not all(isinstance(index, int) for index in (i, j)):
        raise TypeError(f"Sorry, both ’{i}’ and ’{j}’ must be integers")

    if not all(0 <= index <= 8 for index in (i, j)):
        raise IndexError(f"Sorry, both ’{i}’ and ’{j}’ must be numbers between 0 and 8 inclusive.")

    col_set = {col(i), col(j)}
    row_set = {row(i), row(j)}

    # Checking for an intermediate in a column.
    if abs(row(i) - row(j)) == 2 and len(col_set) == 1:
        return (i + j) // 2

    # Checking for an intermediate in a row.
    if abs(col(i) - col(j)) == 2 and len(row_set) == 1:
        return (i + j) // 2

    # Checking for an intermediate in a diagonal.
    if len(row_set) == 2 and len(col_set) == 2 \
        and abs(row(i) - row(j)) == 2 and abs(col(i) - col(j)) == 2:
        return (i + j) // 2

    return -1
```

Listing 2: Exercise 2

Exercise 3

In this exercise, the function $is\_admissible(pattern, i)$ is defined. It takes two arguments, $pattern$ and an index i, and determines if $pattern$ can be extended to $pattern + (i,)$.

The function starts by checking if i belongs to $pattern$; it returns false if this is the case. This ensures that a node is not visited more than once.

Next, $pattern$ is extended to a new pattern $pattern + (i,)$. The extension is assigned to a variable called $new\_pattern$. This new pattern in passed as an argument to a helper function $is\_valid\_pattern(new\_pattern)$ which returns true or false depending on the validity of $new\_pattern$. If true is returned, then $pattern$ can be extended to $pattern + (i,)$. Otherwise, it cannot be extended.

The helper function, $is\_valid\_pattern(pattern : tuple)$, takes as argument a tuple of integers ranging from 0 through 8 and returns true of false depending on the validity of pattern. A pattern p is deemed valid, if it satisfies the following:

1. p is valid if it is the empty tuple ().
2. If p in not empty then
    (a) loop through its elements to ensure a node is not repeated. If a node appears more that once, then p is not a valid pattern; the function returns false.
    (b) if an intermediate k exists between two consecutive nodes i and j in p and (i, k, j) is not a subset of p, then p is not a vlid pattern. Again the function returns false.

The function returns true otherwise.

```python
def is_admissible(pattern: tuple, i: int) -> bool:
    """
    Determines if the index i can be appended to the given pattern

    Parameters
    ----------
    pattern: tuple of int
        A tuple of digits from 0 to 8 representing a valid swipe pattern
    i: int
        An integer between 0 to 8 representing a potential extension node.

    Returns
    -------
    bool
        True if index i can be appended to pattern, False otherwise.
    """

    if i in pattern:
        return False

    new_pattern = pattern + (i,)
    return is_valid_pattern(new_pattern)
#######################################################
    # Helper function for Exercise 3: is_valid_pattern.
#######################################################

def is_valid_pattern(pattern):
    """
    Determines if the given swipe pattern is valid according to Android pattern rules.

    Parameters
    ----------
    pattern : tuple
        A tuple of digits from 0 to 8 representing a swipe pattern.

    Returns
    -------
    bool
        True if the pattern is valid, False otherwise.
    """

    # Empty pattern ()
    if len(pattern) == 0:
        return True

    # Non-empty pattern
    for i in range(1, len(pattern)):
        if pattern[i] in pattern[:i]:
            False

        intermediate = intermediate_node(pattern[i-1], pattern[i])
        if intermediate != -1 and intermediate not in pattern[i-1:i+1]:
            return False
    return True

###############################################
```

Listing 3: Exercise 3

Exercise 4

In this exercise, the function *extensions*(*pattern* : *tuple*) takes as argument a valid pattern represented as a Python tuple, and returns the set of all admissible extensions of this pattern by

1. Thus, $exten(pattern : tuple)$ will rely on the function $is\_admissible(pattern : tuple, i : int)$ to generate this set of admissible extensions.

Here, a for loop comes in handy. We loop through the nodes from 0 through 8 and ckeck if they are admissible to the pattern. An empty set, $set\_of\_tuples = set()$, is initialized. If i is one of the numbers from 0 through 8, and $is\_admissible(pattern, i)$ returns true, then $pattern + (i,)$ is added to the set set_of_tuples. This repeats until the for loop ends.

Finally, the function $extensions(pattern)$ returns the set set_of_tuples.

```python
    def extensions(pattern: tuple) -> set:
        """
        Extends a pattern by one more node.

        Parameters
        ----------
        pattern : tuple
            A tuple of digits from 0 to 8 representing a valid swipe
            pattern

        Returns
        -------
            The set of all possible extensions of pattern by 1 more node.
        """

        set_of_tuples = set()

        for i in range(9):
            if is_admissible(pattern, i):
                extended_pattern = pattern + (i,)
                set_of_tuples.add(extended_pattern)

        return set_of_tuples
```

Listing 4: Exercise 4

Question: What would go wrong here if we instead chose to represent our patterns as lists rather than tuples? How could you modify your approach to accommodate this?

If we choose to represent our patterns as lists, rather than tuples, then

- the expression $pattern + (i,)$ will throw a TypeError exception. This is because a list and a tuple cannot be concatenatd. If a list is chosen, then we nned to first convert the list to a tuple, concatenate.
- Also, the function $is\_admissible$ will throw a TypeError exception since it takse as argument a tuple of integers.

```python
    def extensions(pattern: list) -> set:
        '''
        Parameters
        ----------
        pattern : list
            A list of digits from 0 to 8 representing a valid swipe
            pattern

        Returns
        -------
            The set of all possible extensions of pattern by 1 more
            node.
        '''

        set_of_tuples = set()

        for i in range(9):
            if is_admissible(tuple(pattern), i):
                extend_pattern = tuple(pattern) + (i,)
                set_of_tuples.add(extend_pattern)

        return set_of_tuples
```

Listing 5: Modification for Exercise 4

Exercise 5

In this exercise, we write a Python function *generate_patterns*() which outputs a Python dictionary whose keys are integers 0 through 9 and whose value at integer i is the set of all swipe patterns of length i.

The function begins by first creating a dictionary whose keys are integers from 0 through 9 and whose initial values at each integer is an empty set. The first for loop is used to create a set of patterns of length one. So, the patterns of the form $(i,)$ are added to the set $pattern[1]$, where i is an integer from 0 through 8. The next for loop is used to create the other patterns of length 2 through 9.

Next, two nested for loops are used to create the other patterns of length 2 through 9. The outer for loop loops through the keys k of the dictionary starting from 2 through 9. The inner for loop loops through the set $patterns[k-1]$ and each pattern of length $k-1$ is passed to the function *extensions* from Exercise 5 to generate a set of admissible patterns of length k. The set $patterns[k]$ is then updated with the set of admissible patterns.

The function *generate_patterns*() then returns the dictionary *pattern*.

```
    def generate_patterns():
        '''
        Returns
        -------
        A dictionary whose keys are integers 0 through 8 and whose
        value at integer i is the set of all swipe patterns of length i
        '''

        patterns = {k : set() for k in range(10)}

        # patterns of length 1
        for i in range(9):
            patterns[1].add((i,))

        for k in range(2,10):
            for pattern in patterns[k-1]:
                new_pattern = extensions(pattern)
                patterns[k].update(new_pattern)
        return patterns
```

Listing 6: Sample Python Code

Exercise 6

How many possible swipe patterns are there of length 4 or more? How many patterns of length 9?

Generate all patterns with the function *generate_patterns* from Exercise 5. Use tuple comprehension – $sum(len(pattern_dict[i])foriinrange(4,10))$ – to sum all the patterns of length 4 or more. Here $len(pattern_dict[i])$ computes the length of patterns in the set $pattern_dict[i]$ of length i. The for loop loops over the number from 4 through 9.

Also, patterns of length 9 is $len(pattern_dict[9])$. The results are printed using f-string notation.

```
 pattern_dict = generate_patterns()
 pattern_length_4_or_more = sum(len(pattern_dict[i]) for i in range(4,10)
)
 pattern_length_9 = len(pattern_dict[9])
 print(pattern_length_9)
 print(f"Number of possible swipe patterns of length 4 or more: {
pattern_length_4_or_more}")
 print(f"Number of possible swipe patterns of length 9: {pattern_length_9
}")
```

Listing 7: Exercise 6

Output of code Snippet

- Number of possible swipe patterns of length 4 or more: 139880

- Number of possible swipe patterns of length 9: 32256

Exercise 7

I watched my friend type in her password over her shoulder. I saw that her pattern has the following properties:

- It starts by connecting node 1 to node 0
- It uses every node
- The final edge connected node 7 to node 8
- At some point, I saw her connect node 4 to node 6

How many possible patterns fit this description?

The code snippet below filters the patterns that fit the description above. Here, an empty list [] is assigned to the variable *filtered_patterns*. Since every node will be used, then all patterns that fit the description are of length 9. A for loop is used to loop through all the patterns of length 9 in the set *pattern_dict*. The if statement ensure that the pattern first connects node 1 to 0. It also ensures that the last ends with node 7 connected to 8. It also ensures that node 4 and 6 are in the pattern, and that node 4 comes before 6. The patterns that fit the condition are appended to *filtered_patterns* and its length computed.

```python
    filtered_patterns = []
    pattern_dict = generate_patterns()
    for pattern in pattern_dict[9]:
        if (pattern[0], pattern[1]) == (1,0) and pattern[-2:] == (7,8)\
            and 4 in pattern and 6 in pattern\
                and 4 in pattern[:pattern.index(6)+1]:
            filtered_patterns.append(pattern)
    number_of_valid_patterns = len(filtered_patterns)
```

Listing 8: Exercise 7

Output of code Snippet

- Number of possible swipe patterns fitting the description is: 18

Optional Task

In this section we loop at some metrics for measuring pattern complexity.

Metrics and Code snippets

The following are some metrics for measuring the complexity of a pattern.

- **Length of a pattern**
- **Change in direction:** Counting changes in direction between consecutive moves
- **Change in angle:** Counts the number of angle changes (e.g., straight, right angles, obtuse angles) in the pattern

The following code snippets implements the metrics given above

```python
import android_swipe_pattern as asp
from numpy import array, sqrt, dot, degrees, arccos

def compute_angle(x: tuple[int, int], y: tuple[int, int], z: tuple[int, int]) -> float:
    """
    Calculate the angle between three points.

    Parameters
    ----------
    x, y, z : tuple[int, int]
        Coordinates of the points.

    Returns
    -------
    float
        The angle in degrees between two vectors fromed by the three points.
    """

    def vector(a,b):
        return (b[0]-a[0], b[1]- a[1])
    # Angle formula: v.u = |u||v|cos(theta)

    u = vector(x,y)
    v = vector(z,y)

    dot_product = dot(array(u), array(v))
    norm_u = sqrt(u[0]**2 + u[1]**2)
    norm_v = sqrt(v[0]**2 + v[1]**2)
    cos_theta = dot_product/ (norm_u * norm_v)

    return degrees(arccos(cos_theta))
```

Listing 9: Code snippet to compute angle between two vectors

```python
def angle_change(pattern: tuple[int, ...]) -> int:
    """
    Compute the number of angle changes in the pattern.

    Parameters
    ----------
    pattern: tuple[int, ...]
        A tuple of digits from 0 to 8 representing a swipe pattern.

    Returns
    -------
    int
        The number of significan angle changes in the pattern.
    """
    coordinate_of_nodes = [(asp.row(node), asp.col(node)) for node in pattern]
    number_of_angle_changes = 0

    for i in range(1, len(coordinate_of_nodes)-1):
        angle = compute_angle(coordinate_of_nodes[i-1], coordinate_of_nodes[i], coordinate_of_nodes[i+1])
        if angle != 180:
            number_of_angle_changes += 1

    return number_of_angle_changes
```

Listing 10: Code snippet the metric Change in angle

```python
def calculate_direction(a: int, b: int) -> str:
    """
    Calculate the direction of the move from a to b.

    Parameters
    ----------
    a : int
        Starting index.
    b : int
        Ending index.

    Returns
    -------
    str
        Direction of the move: "horizontal", "vertical", "diagonal", or "same".
    """
    row_a, col_a = asp.row(a), asp.col(a)
    row_b, col_b = asp.row(b), asp.col(b)
    if row_a == row_b and col_a != col_b:
        return "horizontal"
    elif col_a == col_b and row_a != row_b:
        return "vertical"
    elif row_a != row_b and col_a != col_b:
        return "diagonal"
    else:
        return "same"
```

Listing 11: Code Snippet for the metric change in direction

```
def complexity_score(pattern: tuple[int, ...]) -> int:
    """
    Compute the complexity score of a pattern using the metrics:
        Length of pattern
        Changes in direction
        Changes in angle

    Parameters
    ----------
    pattern: tuple[int, ...]
        A tuple of digits from 0 to 8 representing a swipe pattern.

    Returns
    -------
    int
        The complexity score of the pattern
    """

    score = 0

    #Length of pattern
    score += len(pattern)

    # Changes in direction
    for i in range(2,len(pattern)):
        previous_direction = calculate_direction(pattern[i-2], pattern[i-1])
        current_direction = calculate_direction(pattern[i-1], pattern[i])
        if previous_direction != current_direction:
            score += 1

    # Changes in angle
    score += angle_change(pattern=pattern)


```

```
        return score
```

Listing 12: Code snippet to compute pattern complexity score.

Code snippet to visualize a complex pattern

```
    def random_pattern(length, pattern_dict):
        '''
        Parameters
        ----------
        length : int
            an integer length between 2 and 9
        pattern_dict : dict
            A dictionary mapping pattern lengths to sets of patterns
            (like the output of generate_patterns, for example)

        Returns
        -------
        A random choice of pattern from pattern_dict of the prescribed
        length
        '''
        out = random.choice(list(pattern_dict[length]))
        print(out)
        return out


    def draw_arrow(i, j):
        '''
        Parameters
        ----------
        i : int
            A node between 0 and 8.
        j : int
            A node between 0 and 8.

        Returns
        -------
        None. Plots an arrow connecting node i to node j
        '''
        x1 = asp.col(i)
        y1 = asp.row(i)
        x2 = asp.col(j)
        y2 = asp.row(j)
        dx, dy = x2 - x1, y2 - y1

        plt.arrow(x1, y1, dx, dy, head_width = 0.04, width = 0.01, ec ='green')

```

```python
def draw(path):
    '''
    Parameters
    ----------
    path : tuple
        A tuple of integers representing a swipe pattern

    Returns
    -------
    None. Plots a visualization of the input pattern
    '''

    #Clear any existing plots
    plt.clf()

    #Draw the 9 dots representing the grid
    for i in range(0,3):
        for j in range(0,3):
            plt.scatter(i, j, s=200, c='black',  edgecolors='black')

    #Invert the y-axis
    plt.ylim(2.1, -0.1)

    #Can't draw a path of length less than 2
    if len(path) < 2:
        return

    #Connect each pair of adjacent nodes with an arrow:
    for i in range(len(path)-1):
        draw_arrow(path[i], path[i+1])

    #Display the result
    plt.show()
```

Listing 13: Code snippet to visualize a complex pattern

```python
if __name__ == '__main__':
    from random import choice
    pattern_dict = generate_complex_patterns(threshold=5)
    random_length = choice(range(4,10))
    pattern = random_pattern(length=random_length, pattern_dict=
pattern_dict)
    draw(path=pattern)
```

Listing 14: Plot of complex pattern