Task 3

Debugging algorithms

Read and debug some of the client's code

Here is the background information on your task

The JavaScript web development part is going very well, but you notice something strange when integrating to the client's Python backend. You dig into the backend code base and notice that the client has implemented a lot of basic algorithms manually. Normally this would not be a problem, but some of the implementations are quite buggy and are causing you to have wrong results or even breaking your solution. Time to start debugging!

Here is your task

One of the algorithms is a binary search algorithm. Luckily, you are very familiar with it and know that binary search is a search algorithm that finds the position of a target value in a sorted table.

Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues for the remaining half, again taking the middle element to compare to the target value and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Here is the implementation you find in the client's codebase:

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

What happens if you run code containing this function?

Start your task

You may only credit this task to your CV if you make a genuine attempt at the work.
Practice or learning attempt:

Q 1/8: Refer to the image below. True or false – what happens if you run code containing this function?

It raises an index out of bounds error if searching for the last element of a table with an uneven number of elements, with a length greater than three.

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

accenture

Q 2/8: Refer to the image below. True or false - what happens if you run code containing this function?

It raises an index out of bounds error if searching for the first element of the table

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

Q 3/8: Refer to the image below. True or false - what happens if you run code containing this function?

It loops forever if searching for a number, in between the upper and lower boundaries of the table, that is not an element of the table

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

Q 4/8: Refer to the image below. True or false - what happens if you run code containing this function?

It loops forever if searching for the last element of the table

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

Q 5/8: Refer to the image below. True or false – what happens if you run code containing this function?

It raises a division by zero error if the table is empty

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

Q 6/8: Refer to the image below. True or false – what happens if you run code containing this function?

It raises an index out of bounds error if the table is empty

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

True

False

Q 7/8: Refer to the image below. True or false – what happens if you run code containing this function?

It loops forever if the length of the table is 1

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

- True
- False

Q 8/8: How would you implement a correct binary search function?

You might consider using recursion (a function that calls itself)

```
def binary_search(sorted_table, element):
    index = math.ceil(len(sorted_table) / 2)

    while element != sorted_table[index]:
        if sorted_table[index] > element:
            index = math.ceil(index / 2)
        else:
            index = math.ceil(index + index / 2)

    return index
```

- I would use recursion, since it's impossible to implement binary search without using recursion
- I would not use recursion, because with recursion has time complexity of O(n log n)
- Whatever I'm the most comfortable with: it is possible to implement both ways and the time complexity is always O(log n)