Seminar Preparation:

Question:

Recursion

One of the classic programming problems that is often solved by recursion is the towers of Hanoi problem. A good explanation and walkthrough are provided by Cormen & Balkcom (n.d.) - the link is in the reading list. (the code they used for their visual example is provided on their website as well).

- Read the explanation, study the code and then create your own version using Python (if you want to make it more interesting you can use asterisks to represent the disks). Create a version that asks for the number of disks and then executes the moves, and then finally displays the number of moves executed.
- What is the (theoretical) maximum number of disks that your program can move without generating an error?
- What limits the number of iterations? What is the implication for application and system security?

Regex

-

The second language concept we will look at is regular expressions (regex). We have already presented some studies on their use, and potential problems, above. The lecturecast also contains a useful link to a tutorial on creating regex. Re-read the provided links and tutorial (Jaiswal, 2020) and then attempt the problem presented below:

- The UK postcode system consists of a string that contains a number of characters and numbers – a typical example is ST7 9HV (this is not valid – see below for why). The rules for the pattern are available from idealpostcodes (2020).

- Create a python program that implements a regex that complies with the rules provided above – test it against the examples provided.
- Examples:
- M1 1AA
- M60 1NW
- CR2 6XH
- DN55 1PT
- W1A 1HQ
- EC1A 1BB
- How do you ensure your solution is not subject to an evil regex attack?

You can share your responses with tutor for formative feedback or discuss it in this week's seminar. There will also be an opportunity to review your team's progress during the seminar.

Remember to record your results, ideas and team discussions in your e-portfolio.

Response:

1. Towers of Hanoi using Recursion:

Here's a Python program that asks for the number of disks, calculates the moves, and displays them:

```
def hanoi(n, source, target, auxiliary, move_count):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        move_count[0] += 1
        return
    hanoi(n-1, source, auxiliary, target, move_count)
    print(f"Move disk {n} from {source} to {target}")
    move_count[0] += 1
    hanoi(n-1, auxiliary, target, source, move_count)


def towers_of_hanoi():
    n = int(input("Enter the number of disks: "))
    move_count = [0]  # Using a list to pass by reference
    hanoi(n, 'A', 'C', 'B', move_count)
    print(f"\nTotal moves: {move_count[0]}")


if __name__ == "__main__":
    towers_of_hanoi()
```

- This code moves the disks between three rods labeled 'A', 'B', and 'C'.
- It counts the number of moves and displays them at the end.

Theoretical Maximum Number of Disks: The recursion depth is limited by Python's recursion limit (default is 1000). For a problem like Towers of Hanoi, the recursive depth increases rapidly with the number of disks, since the number of moves required is

2

n

−

1

2^n - 1

2n−1.

This means that for disks beyond a certain number (around 20-25 depending on the system), recursion might hit the limit and cause a RecursionError.

You can adjust the recursion limit using sys.setrecursionlimit(), but increasing the recursion depth can pose a risk for application and system security, such as potential stack overflow attacks in recursive functions.

2. Regex for UK Postcode Validation:

Here's a Python program using regular expressions to validate UK postcodes:

```
import re
# UK Postcode Regex Pattern (simplified for standard cases)
postcode_regex = r"^[A-Z]{1,2}\d{1,2}\s?\d[A-Z]{2}$"


def validate_postcode(postcode):
    if re.match(postcode_regex, postcode):
        return True
    else:
        return False


def test_postcodes():
    postcodes = ["M1 1AA", "M60 1NW", "CR2 6XH", "DN55 1PT", "W1A 1HQ", "EC1A 1BB",
"ST7 9HV"]  # Last one is invalid
    for postcode in postcodes:
        result = validate_postcode(postcode)
        print(f"{postcode} is {'valid' if result else 'invalid'}")


if __name__ == "__main__":
    test_postcodes()
```

This code matches postcodes according to general rules, where:

- The first part consists of 1 or 2 letters followed by 1 or 2 digits.
- An optional space.
- A second part of 1 digit followed by 2 letters.

Preventing Evil Regex Attacks: Regular expressions can be vulnerable to ReDoS (Regular Expression Denial of Service) attacks, where specially crafted input leads to excessive backtracking and long processing times. To avoid such risks:

- Keep the regex pattern simple and efficient.
- Avoid unnecessary quantifiers or complex patterns that can lead to backtracking.
- Use timeouts or limits in environments where regex patterns might face untrusted input.

Let me know if you need more details or further help on either topic!