Daniel Adamiak
Adam Kuszczyński
Grzegorz Łoszewski

TASK 2
REPORT

BRUTE FORCE

BRUTE FORCE

First, let's define a method *brute_force_search* that will take 2 arguments pattern and *text*.

```python
def brute_force_search(pattern, text):
```

Then, we define two variables m and n to store the length of the text and pattern respectively.

```python
    m, n = len(text), len(pattern)
```

We also define two variables *i* and *j* to keep track of our current positions in *text* and pattern respectively and set them both to 0 (index 0).

```python
    i, j = 0, 0
```

The following while loop will 'run' until we reach the end of *text* or *pattern*.

```python
    while i < m and j < n:
```

The first thing that we check inside the loop is wheter the current character we 'are on' in *pattern* is an escape character('\').

```python
        if pattern[j] == '\\':
```

If that's the case - we increment *j*(index of the character in *pattern*) by 1. We increment by 1 which means that we move to the next character in *pattern*.

```python
            j += 1
```

BRUTE FORCE

Then, we check if the character in *pattern* we just shifted to is equal to *n*(length of *pattern*).

```python
            if j == n:
```

If *j* is equal to *n*, it means that the backslash is the last character in the *pattern*, which means that the *pattern* cannot be matched because there is no character in the *text* to match with the backslash(basically it means there is nothing after backslash - there is no character to escape).
We return False because the pattern has not been found.

```python
                return False
```

However if *j* is not equal to *n*, we can check if the character after a backslash in the *pattern* (*pattern[j]*) is equal to the current character in the *text* (*text[i]*).
If it's not -> we return False because the pattern has not been found.

```python
            if text[i] != pattern[j]:
                return False
```

If the current character in *pattern* is a `?` wildcard we increment both *i* and *j* indexes by one in order to move by one position in both *text* and *pattern* respectively. Basically we skip a one character since `?` is considered as any single character.

```python
        elif pattern[j] == '?':
            i += 1
            j += 1
```

BRUTE FORCE

If the current character in *pattern* is a '*' wildcard.

```
        elif pattern[j] == '*':
```

If it is, then we increment the index *j* to move to the next character in the *pattern*.

```
            j += 1
```

We check if *j* is equal to *n*(length of *pattern*).

```
            if j == n:
```

If it is, it means that the end of the *pattern* has been reached. This means that we have a match so we return *True*.

```
                return True
```

However i *j* is not at the end of the *pattern*, then we have to search for a match in the *text*. The while loop checks if we have not reached the end of the *text*(i < m), and if the current character of the *text* does not match the current character in the *pattern* (*text*[i] != *pattern*[j]), and if the current character in the *pattern* is not a `?' wildcard (*pattern*[j] != '?').

```
            while i < m and text[i] != pattern[j] and pattern[j] != '?':
```

So, the loop keeps searching for a match by incrementing *i* until one of the three conditions is not met.

```
                i += 1
```

BRUTE FORCE

If *i* becomes equal to *m* it means that the end of the *text* has been reached, and there is no match. So we return *False*.

```
        if i == m:
            return False
```

If the current character in *pattern* is not an escaped character `\`, a `?` wildcard, or a `*` wildcard, we check whether the current character in *text* matches the current character in *pattern*. If it doesn't match, we return *False* because the pattern has not been found.

```
        elif text[i] != pattern[j]:
            return False
```

If the current character in *pattern* matches the current character in *text*, we increment both *i* and *j* to move forward in *text*.

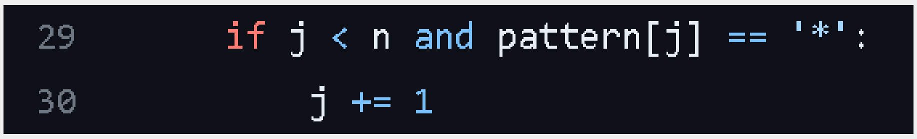
```
        else:
            i += 1
            j += 1
```

We check if *j* is equal to the length of the *pattern* *n* and if *i* is equal to the length of the *text* *m*. If they are both equal, it means that all the characters in the *pattern* have been compared with the corresponding characters in the *text* and there is a complete match. In this case, the function returns *True*.

```
    if j == n and i == m:
        return True
```

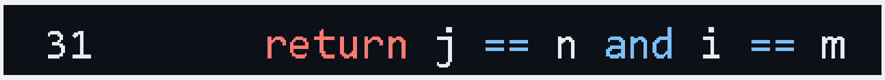
BRUTE FORCE

This line checks if there is a '*' wildcard at the end of the *pattern*.
If there is, it means that the star(`*`) can match any number of characters in the *text*, so we can ignore it and still have a complete match. So, we increment *j* to skip the star and check again if there is a complete match by returning *True*.



```
    if j < n and pattern[j] == '*':
        j += 1
```

This line returns *True* if both *j* is equal to the length of the *pattern n* and *i* is equal to the length of the *text m*. If they are both equal, it means that all the characters in the *pattern* have been compared with the corresponding characters in the *text* and there is a complete match. Otherwise, the function returns *False*.

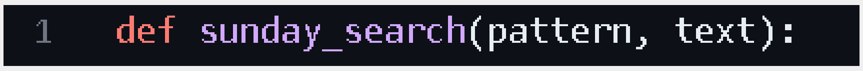


```
    return j == n and i == m
```

SUNDAY

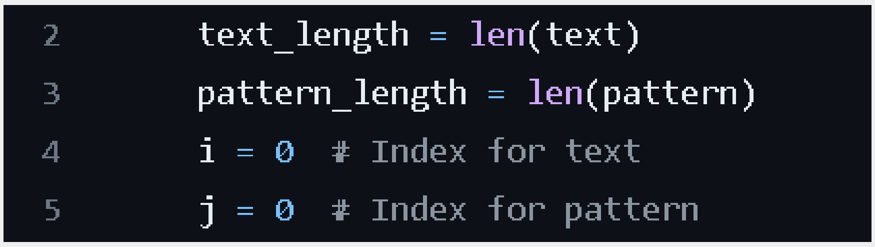
SUNDAY

This line defines the function *sunday_search* with two parameters *pattern* and *text*.



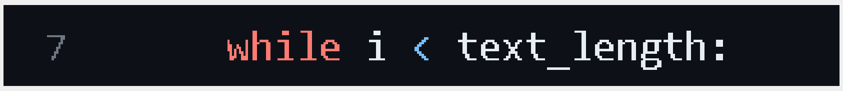
```python
def sunday_search(pattern, text):
```

These lines initialize variables *text_length*, *pattern_length*, *i*, and *j*. *text_length* and *pattern_length* are the lengths of the *text* and *pattern* strings respectively. *i* and *j* are the index variables used to traverse the *text* and *pattern* strings.



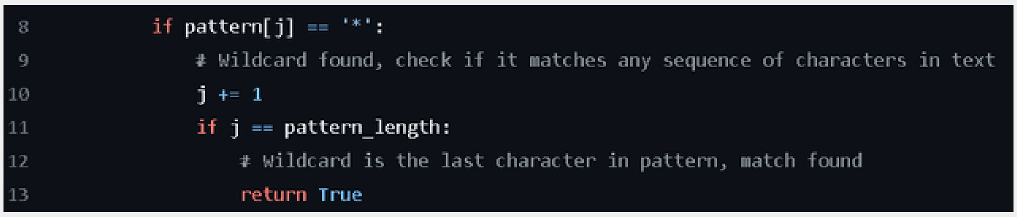
```python
    text_length = len(text)
    pattern_length = len(pattern)
    i = 0  # Index for text
    j = 0  # Index for pattern
```

This line starts a while loop that iterates while *i* is less than *text_length*, meaning there are still characters in *text* to search.



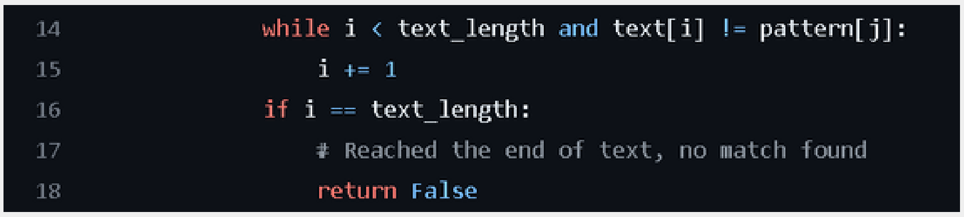
```python
    while i < text_length:
```

This if statement checks if the current character in *pattern* is a `*` wildcard. And if it is, we increment *j* and check if we've reached the end of the *pattern* string. If we have, we return *True* as the pattern matches the text.



```python
        if pattern[j] == '*':
            # Wildcard found, check if it matches any sequence of characters in text
            j += 1
            if j == pattern_length:
                # Wildcard is the last character in pattern, match found
                return True
```

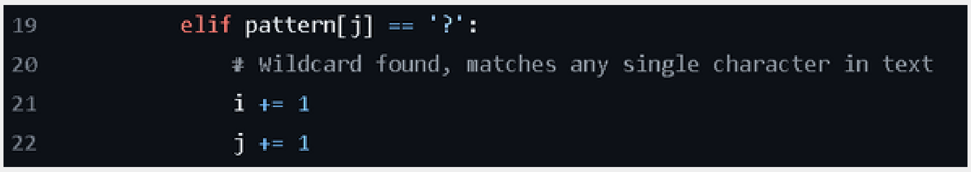
However if we haven't reached the end of *pattern* yet, we use a while loop to find the first character in *text* that matches the current character in *pattern* after the * wildcard. If we reach the end of *text* without finding a match, we return *False* as the *pattern* doesn't match the *text*.



```python
            while i < text_length and text[i] != pattern[j]:
                i += 1
            if i == text_length:
                # Reached the end of text, no match found
                return False
```

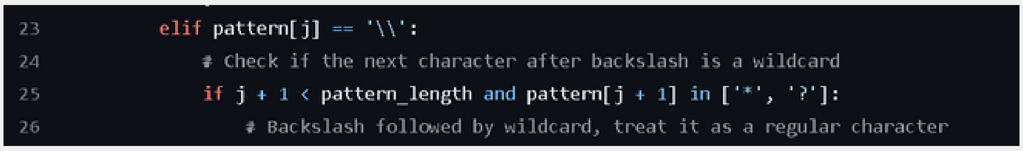
SUNDAY

If the current character in *pattern* is a `?` wildcard, we increment both i and *j* to move to the next character in *text* and *pattern*.



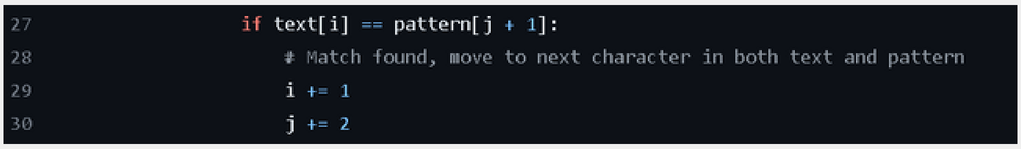
```python
        elif pattern[j] == '?':
            # Wildcard found, matches any single character in text
            i += 1
            j += 1
```

If the current character of *pattern* is an escape character `\` we check if the index of the next character in the *pattern*(*j* + 1) is within the bounds of the *pattern_length* and if the next character is either a sequence wildcard (*) or a single-character wildcard (?).



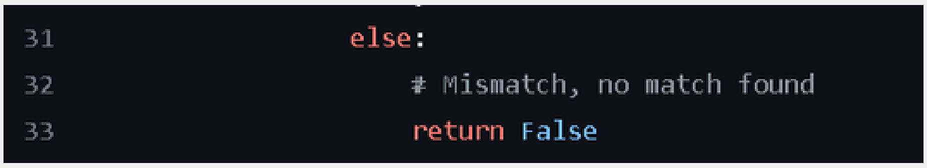
```python
        elif pattern[j] == '\\':
            # Check if the next character after backslash is a wildcard
            if j + 1 < pattern_length and pattern[j + 1] in ['*', '?']:
                # Backslash followed by wildcard, treat it as a regular character
```

If the condition is met, the next step is to check if the character in the *text* matches the character after the backslash in the *pattern*.
If there is a match, it means that the backslash is being used to escape the following character, so both the *text* and *pattern* indexes should be incremented by 1 and 2, respectively



```python
                if text[i] == pattern[j + 1]:
                    # Match found, move to next character in both text and pattern
                    i += 1
                    j += 2
```

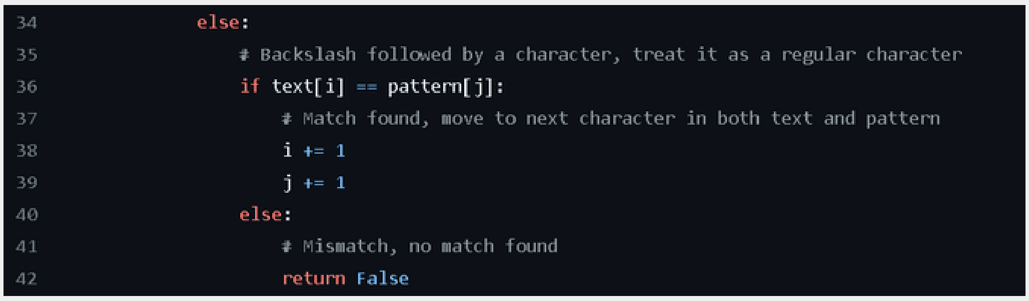
If there is no match between the current *text* character and the escaped character in the *pattern*, it means that the *pattern* does not match the *text*, so the function returns *False*.



```python
                else:
                    # Mismatch, no match found
                    return False
```

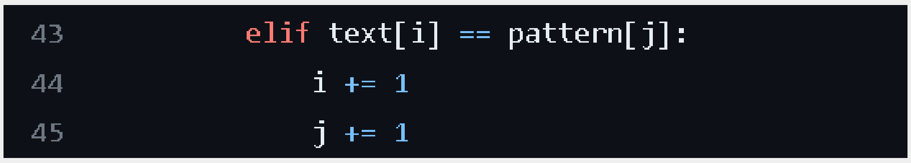
SUNDAY

This code block is executed when the current character in the *pattern* is a backslash. It checks if the next character in the *pattern* is a regular character and if it matches the current character in the *text*.
If there's a match, then it moves to the next character in both the *text* and the *pattern* by incrementing *i* and *j* by *1*.
If there's a mismatch, then the function returns *False* indicating that there is no match between the *pattern* and the *text*.



```
        else:
            # Backslash followed by a character, treat it as a regular character
            if text[i] == pattern[j]:
                # Match found, move to next character in both text and pattern
                i += 1
                j += 1
            else:
                # Mismatch, no match found
                return False
```

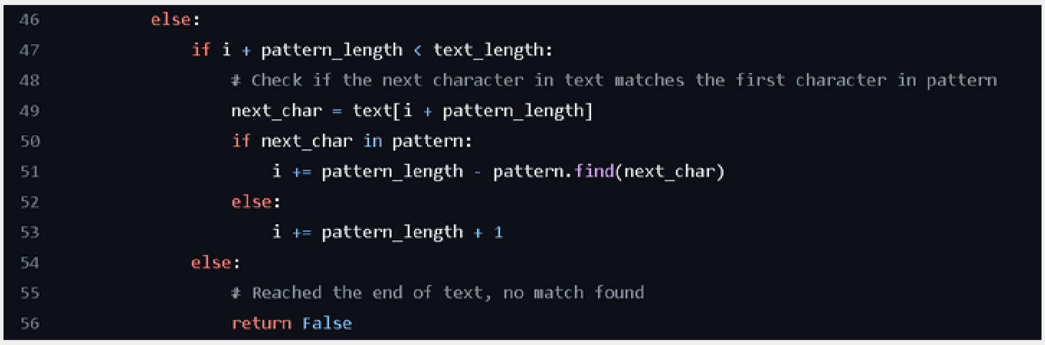
This code block is executed when the current character in the *pattern* is a regular character and it matches the current character in the *text*.
It increments *i* and *j* by *1*, moving to the next character in both the *text* and the *pattern*.



```
        elif text[i] == pattern[j]:
            i += 1
            j += 1
```

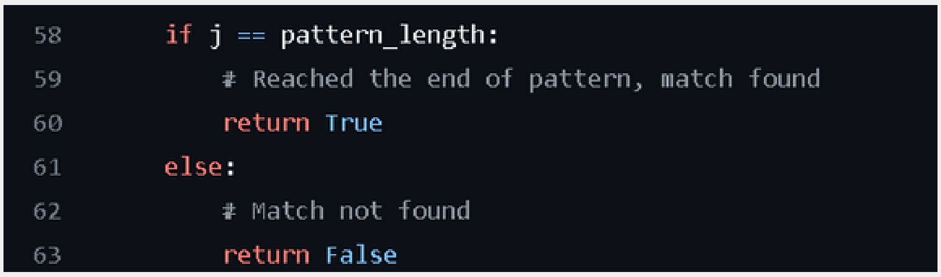
SUNDAY

This code block is executed when there is a mismatch between the current characters in the *text* and the *pattern*.
It checks if there are enough characters left in the *text* for a potential match with the *pattern*. If there are, it looks ahead to the next character in the *text* and checks if it matches the first character in the *pattern*.
If there is a match, it moves the index *i* to the location of the first matching character in the *text*. If there is no match, it increments i by the length of the pattern + *1*, moving to the next potential match in the *text*.
If there aren't enough characters left in the *text* for a potential match with the *pattern*, the function returns *False* indicating that there is no match between the *pattern* and the *text*.



```
        else:
            if i + pattern_length < text_length:
                # Check if the next character in text matches the first character in pattern
                next_char = text[i + pattern_length]
                if next_char in pattern:
                    i += pattern_length - pattern.find(next_char)
                else:
                    i += pattern_length + 1
            else:
                # Reached the end of text, no match found
                return False
```

This code block is executed after all the characters in the *pattern* have been compared to the characters in the *text*.
If the value of j is equal to the *pattern_length*, it means that all characters in the *pattern* have been matched to corresponding characters in the *text*. In this case, the function returns *True* indicating that the *pattern* matches the *text*.
If the value of j is not equal to the *pattern_length*, it means that there are unmatched characters in the *pattern*. In this case, the function returns *False* indicating that there is no match between the *pattern* and the *text*.



```
    if j == pattern_length:
        # Reached the end of pattern, match found
        return True
    else:
        # Match not found
        return False
```