Code Explanation

- ? - Matches zero or one occurrence of the preceding character
- $ - Matches the word end with given pattern
- ^ - Matches the word start with given pattern
- . - Matches any single character except ‘\n’ or ‘\0’
- \ - To escape the above symbols
- Plain text - Matches any word without special regex characters

This code contains the KMP algorithm. The reason for choosing it is because it is efficient.

1. Find lps

```
def find_lps(pattern):
    length = 0
    lps = [0] * len(pattern)
    i = 1
    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1

    return lps
```

The given code is an implementation of the Longest Prefix-Suffix (LPS) array construction algorithm, often used in string matching algorithms like KMP (Knuth-Morris-Pratt). The LPS array helps in

finding patterns within strings efficiently by providing information about the longest proper prefix that is also a suffix for each position in the pattern string.

2. kmp_search

```
def kmp_search(text, pattern):
    i = 0
    j = 0
    lineNum = 1
    charCount = 0
    lps = find_lps(pattern)
    outputList = []
    while i < len(text):
        if text[i] == '\n':
            lineNum += 1
            charCount = i + 1
        if pattern[j] == text[i]:
            i += 1
            j += 1
            if j == len(pattern):
                outputList.append([lineNum, i - (j + charCount)])
                j = lps[j - 1]
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return outputList
```

The code is an implementation of the Knuth-Morris-Pratt (KMP) string search algorithm. The KMP algorithm is used to efficiently find occurrences of a pattern within a given text.

3. kmp_search_backslash

```
def kmp_search_backslash(text, pattern):
    newPattern = ""
    for i in pattern:
        if i != '\\':
            newPattern += i

    return kmp_search(text, newPattern)
```

This code appears to be a modification of the KMP string search algorithm to handle the presence of backslashes (\) in the pattern. Backslashes are often used as escape characters in programming, so they can complicate pattern matching. This modified version aims to remove backslashes from the pattern before performing the KMP search.

4. kmp_search_dot

```
def kmp_search_dot(text, pattern):
    i = 0
    j = 0
    lineNum = 1
    charCount = 0
    lps = find_lps(pattern)
    outputList = []
    while i < len(text):
        if text[i] == '\n':
            lineNum += 1
            charCount = i + 1
        if pattern[j] == text[i] or pattern[j] == '.':
            i += 1
            j += 1
            if j == len(pattern):
                outputList.append([lineNum, i - (j + charCount)])
                j = lps[j - 1]
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return outputList
```

This code appears to be a further modification of the KMP string search algorithm, extending it to handle a special character '.' in the pattern. In many regex (regular expression) contexts, the '.' character is used as a wildcard that matches any character except a newline. This modified version aims to incorporate this wildcard behavior within the KMP search.

5. kmp_search_question

```python
def kmp_search_question(text, pattern):
    ptr1 = ""
    ptr2 = ""
    for i in pattern:
        if i != '?':
            ptr1 += i
            ptr2 += i
        else:
            ptr2 = ptr2[:-1]

    outputList = kmp_search(text, ptr1)
    for i in kmp_search(text, ptr2):
        if i not in outputList:
            outputList.append(i)

    sortedoutputList = sorted(outputList, key=lambda x: (x[0], x[1]))

    return sortedoutputList
```

This code appears to be another extension of the KMP string search algorithm, this time accommodating the special character '?' in the pattern. The '?' character often denotes a wildcard that can match any single character. The code generates two different patterns based on the positions of the '?' character and performs KMP searches using these patterns.

6. kmp_search_startwith

```
def kmp_search_startwith(text, pattern):
    pattern = pattern[1:]
    i = 0
    j = 0
    lineNum = 1
    charCount = 0
    lps = find_lps(pattern)
    outputList = []
    condition = True
    while i < len(text):
        if text[i] == '\n':
            lineNum += 1
            charCount = i + 1
        if i - charCount == 0 or text[i - 1] == ' ':
            condition = True
        if condition:
            if pattern[j] == text[i]:
                i += 1
                j += 1
                if j == len(pattern):
                    outputList.append([lineNum, i - (j + charCount)])
                    j = lps[j - 1]
                    condition = False
            else:
                condition = False
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1
        else:
            i += 1

    return outputList
```

This code is yet another modification of the KMP string search algorithm, designed to find occurrences of a given pattern within a larger text, but with the added constraint that the pattern should start with a specific character.

7.kmp_search_endwith

```python
def kmp_search_endwith(text, pattern):
    pattern = pattern[1:]
    text += '\0'
    i = 0
    j = 0
    k = 0
    lineNum = 1
    charCount = 0
    lps = find_lps(pattern)
    outputList = []
    sign = [' ', '\n', '\0']
    while i < len(text):
        if text[i] == '\n':
            lineNum += 1
            charCount = i + 1
        if text[i] in sign:
            k = i + 1 - (j + charCount)
        if j != len(pattern):
            if pattern[j] == text[i]:
                i += 1
                j += 1
                if j == len(pattern) and (text[i] in sign):
                    outputList.append([lineNum, k])
                    j = lps[j - 1]
            else:
                if j != 0:
                    j = lps[j - 1]
                else:
                    i += 1
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return outputList
```

This code appears to be yet another extension of the KMP string search algorithm, designed to find occurrences of a given pattern within a larger text, but with the additional constraint that the pattern should end with a specific character.

Main function

```
def main():
    n = int((input("Enter the number of file pairs : ")))

    try:

        for i in range(1, n + 1):
            patternFile = open(f'Inputs/Patterns/pattern{i}.txt', "r")
            textFile = open(f'Inputs/Texts/text{i}.txt', "r")
            outputFile = open(f'Outputs/patternmatch{i}.output', "w")

            pattern = patternFile.read().strip().lower()
            text = textFile.read().strip().lower()


            if '\\' in pattern:
                outputList = kmp_search_backslash(text, pattern)
            elif '.' in pattern:
                outputList = kmp_search_dot(text, pattern)
            elif '?' in pattern:
                outputList = kmp_search_question(text, pattern)
            elif '^' in pattern:
                outputList = kmp_search_startwith(text, pattern)
            elif '$' in pattern:
                outputList = kmp_search_endwith(text, pattern)
            else:
                outputList = kmp_search(text, pattern)


            if outputList:
                outputFile.write("Number of matches = "+str(len(outputList)) + "\n\n")
                for line, idx in outputList:
                    outputFile.write("Line number : " + str(line) + ", Index : " + str(idx) + "\n")
            else:
                outputFile.write("No Matches\n")



            outputFile.close()
            textFile.close()
            patternFile.close()



        print(f"Open the output folder to get report !!!")
    except:
        print("Error Occured!")
if __name__ == "__main__":
    main()
```

The given code appears to be a script that processes pairs of text and pattern files, performs various pattern matching operations using different modified versions of the KMP algorithm, and writes the results to output files.