Symbol Puzzle

Advanced Techniques

These techniques are helper pieces of code to extend the Skeleton Program and functionality.

Note that the techniques demonstrated in this document are **BEYOND** the AQA 7517 specification. They **DO NOT** represent a mark scheme or an expected way of solving challenges presented for the pre-release material. The objective of this document is to extend students' knowledge and explore alternative techniques they could use as they experiment with the code. **Use of these techniques and ideas should be at the teacher's discretion**. The code shown below still demonstrates the 'wrap around' false positive matches identified by the standard skeleton code.

Extension Technique 1 – Regular expressions

The Skeleton Program provided by AQA does not include the regular expression (regex) library and, therefore, it is unlikely that AQA would include a Section D question which uses regex. *(A regex question in Section C is perfectly possible.)* Although the library has not been included, students can import the library themselves and use it if they are confident with regex and feel that its use would aid their solutions.

To use this code you will need to import the regex library: `import re`

An obvious place that could use regex is in the **MatchesPattern** method in the Pattern class, which could be rewritten as:

```
    def CheckforMatchWithPatternRegexAQAStandard(self, PatternStringToCheck):
        # These expressions comply with the techniques in the AQA specification
        # The hyphen must be last otherwise it is interpreted as a range
        t_check = re.compile(r'TTT[Q|X|T|@|-][Q|X|T|@|-]T[Q|X|T|@|-][Q|X|T|@|-]T')
        x_check = re.compile(r'X[Q|X|T|@|-]X[Q|X|T|@|-]X[Q|X|T|@|-]X[Q|X|T|@|-]X')
        q_check = re.compile(r'QQ[Q|X|T|@|-][Q|X|T|@|-]Q[Q|X|T|@|-][Q|X|T|@|-]QQ')

        return bool(t_check.search(PatternStringToCheck)) or bool(x_check.search(PatternStringToCheck)) or bool(q_check.search(PatternStringToCheck))
```

This expression can be extended further by defining duplicates of symbols:

```
    def CheckforMatchWithPatternRegexBeyondStandard(self, PatternStringToCheck):
        # These expressions use techniques which are beyond the AQA specification
        # The hyphen must be last otherwise it is interpreted as a range
        t_check = re.compile(r'T{3}[Q|X|T|@|-]{2}T[Q|X|T|@|-]{2}T')
        x_check = re.compile(r'(X[Q|X|T|@|-]){4}X')
        q_check = re.compile(r'Q{2}[Q|X|T|@|-]{2}Q[Q|X|T|@|-]{2}QQ')

        return bool(t_check.search(PatternStringToCheck)) or bool(x_check.search(PatternStringToCheck)) or bool(q_check.search(Patt
ernStringToCheck))
```

Extension Technique 2 – Regular expressions on the grid

The methods shown in technique 1 improve the time complexity of the **MatchesPattern** method from O(n) to O(1). The main data structure for the pre-release material uses a one-dimensional list; therefore, regex combined with a lambda-like expression could be used on this to make the same improvements by removing the need to generate the helix pattern string. The expression, however, needs further modification to take into account that the grid can be different sizes and, therefore, that the space between symbols within a pattern can change. A limitation of this code, however, is that it cannot detect overlapping patterns or some patterns right next to each other because the regex is not identifying overlapping matches.

```
    def CheckforMatchWithPatternRegexUsingGrid(self, SymbolToCheck):
        StringRepresentationOfWholeGrid = ''.join(cell.GetSymbol() for cell in self.__Grid)
        check = None

        # The hyphen must be last otherwise it is interpreted as a range
        # The user can only enter in these three chars per the "AllowedSymbols" list, therefore, we don't
        # need to have a default for the switch to fall through if no match case
        if (SymbolToCheck == "Q"):
            check = re.compile(f'Q{{2}}[Q|X|T|@|-]{{{self.__GridSize - 2}}}Q{{2}}[Q|X|T|@|-]{{{self.__GridSize}}}Q')
        elif (SymbolToCheck == "T"):
            check = re.compile(f'T{{3}}[Q|X|T|@|-]{{{self.__GridSize - 2}}}T[Q|X|T|@|-]{{{self.__GridSize - 1}}}T')
        elif (SymbolToCheck == "X"):
            check = re.compile(f'X[Q|X|T|@|-]X[Q|X|T|@|-]{{{self.__GridSize - 3}}}[Q|X|T|@|-]X[Q|X|T|@|-][Q|X|T|@|-
]{{{self.__GridSize - 3}}}X[Q|X|T|@|-]X')

        return bool(check.search(StringRepresentationOfWholeGrid))
```

Extension Technique 3 – Regular expressions on the grid setting a cell to be part of a pattern

Regex is designed to pattern match. On its own it will not change any of the attributes of the cells within the grid once a pattern has been matched. To achieve this, we need to match patterns and then calculate where they are in the grid so that we can call appropriate methods on those cells. It is possible to recognise overlapping matches using regex; however, this limits the ability to then easily identify where those matches are. A solution to this is to iterate through the string representation of the grid looking for matches, although this increases the time complexity of the method. Where matches are found, we need to record the position in the grid of those matches so that we can then call methods on the cells at those positions. This code improves the time complexity of the overall application from O(n^3) to O(n^2). You will need to add an accessor method and a mutator method to the Cell class to make this work. These should get and set a Boolean attribute called 'PartOfPattern'.

```python
    def CheckforMatchWithPatternRegexUsingGrid2(self, SymbolToCheck):
        StringRepresentationOfWholeGrid = ''.join(cell.GetSymbol() for cell in self.__Grid)
        check = None
        pattern_matches = []

        # The hyphen must be last otherwise it is interpreted as a range
        # The user can only enter in these three chars per the "AllowedSymbols" list, therefore, we don't
        # need to have a default for the switch to fall through if no match case
        if (SymbolToCheck == "Q"):
            check = re.compile(f'Q{{2}}[Q|X|T|@|-]{{{self.__GridSize - 2}}}Q{{2}}[Q|X|T|@|-]{{{self.__GridSize}}}Q')
        elif (SymbolToCheck == "T"):
            check = re.compile(f'T{{3}}[Q|X|T|@|-]{{{self.__GridSize - 2}}}T[Q|X|T|@|-]{{{self.__GridSize - 1}}}T')
        elif (SymbolToCheck == "X"):
            check = re.compile(f'X[Q|X|T|@|-]X[Q|X|T|@|-]{{{self.__GridSize - 3}}}[Q|X|T|@|-]X[Q|X|T|@|-][Q|X|T|@|-
]{{{self.__GridSize - 3}}}X[Q|X|T|@|-]X')

        # This will match all of the cells in the grid which contain the pattern, including the "spaces" in rows which I am not int
erested in.
        # Regex doesn't match overlapping patterns, so I need to iterate through the whole string version of the grid.
        for i in range(len(StringRepresentationOfWholeGrid) - 9):
            match = check.search(StringRepresentationOfWholeGrid, i)
            if (match):
                match_positions = self.GetJustMatchedSymbolPositionsInGrid(match)
                if (not pattern_matches or pattern_matches[-1] != match_positions):
                    pattern_matches.append(match_positions)
        if (pattern_matches):
            # Included purely for testing
            print("Symbol positions in each match in the grid:")
            for pattern_match in pattern_matches:
                # This may print multiple times if more than one pattern of the symbol being tested
                # currently exists in the grid.
                print(", ".join(map(str, pattern_match)))

            score_awarded = False
            for pattern_match in pattern_matches:
                for cell_position in pattern_match:
                    # If there is a match, add the symbol to the SymbolsNotAllowedList for those cells.
                    # But only if they are not already in a pattern.
                    if not self.__Grid[cell_position].IsPartOfPattern():
                        self.__Grid[cell_position].AddToNotAllowedSymbols(SymbolToCheck)
                        self.__Grid[cell_position].SetPartOfPattern()
                        score_awarded = True

            if (score_awarded):
                return 10
        else:
            print(f"No pattern matches for Symbol: {SymbolToCheck} found.")

        return 0

    def GetJustMatchedSymbolPositionsInGrid(self, match):
        symbol_positions = [0, 1, 2]

        for i in range(2, self.__GridSize * 3):
            if self.__GridSize - 1 < i <= self.__GridSize + 2:
                symbol_positions.append(i)
            if (self.__GridSize * 2) - 1 < i <= (self.__GridSize * 2) + 2:
                symbol_positions.append(i)

        return [i + match.start() for i in symbol_positions]
```

Extension Technique 4 – Reverse referencing the grid

The skeleton code includes a method called GetCell which takes the parameters of Row and Column and returns the Cell at the associated location in the grid data structure. This method reverses that process, allowing the user to pass the reference of a location in the grid data structure, and it returns a tuple containing the row and column for that cell.

```python
    def GetRowColumnFromIndex(self, Index):
        if (Index < 0 or Index >= len(self.__Grid)):
            return None
        Result = Index // self.__GridSize
        row = (self.__GridSize - 1) - Result
        column = Index % self.__GridSize
        return row + 1, column + 1
```