Day 11

Task 1: String Operations.

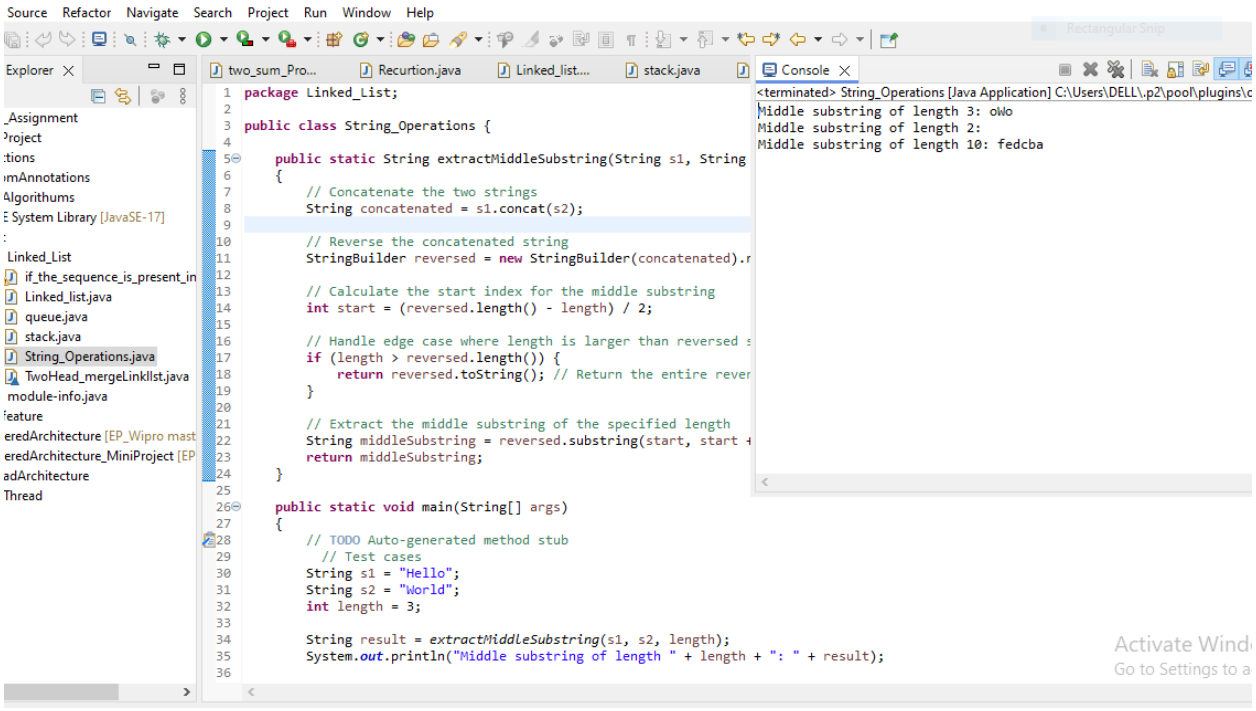
Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

 extractMiddleSubstring Method:

- Concatenates s1 and s2 using concat() method.
- Reverses the concatenated string using StringBuilder and reverse() method.
- Calculates the start index for the middle substring as (reversed.length() - length) / 2.
- Checks if length is larger than reversed.length() and handles it by returning the entire reversed string.
- Uses substring(start, start + length) to extract the middle substring of the specified length.
- Returns the extracted substring.

 main Method:

- Includes test cases to demonstrate the functionality:
    - Concatenates "Hello" and "World", and extracts the middle substring of length 3 ("roW").
    - Handles cases with empty strings and ensures the correct behavior when the substring length is larger than the concatenated string.



Task 2: Naive Pattern Search.

Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

The naive pattern searching algorithm is a straightforward method for finding all occurrences of a pattern in a given text. It works by checking for the pattern at every position in the text,

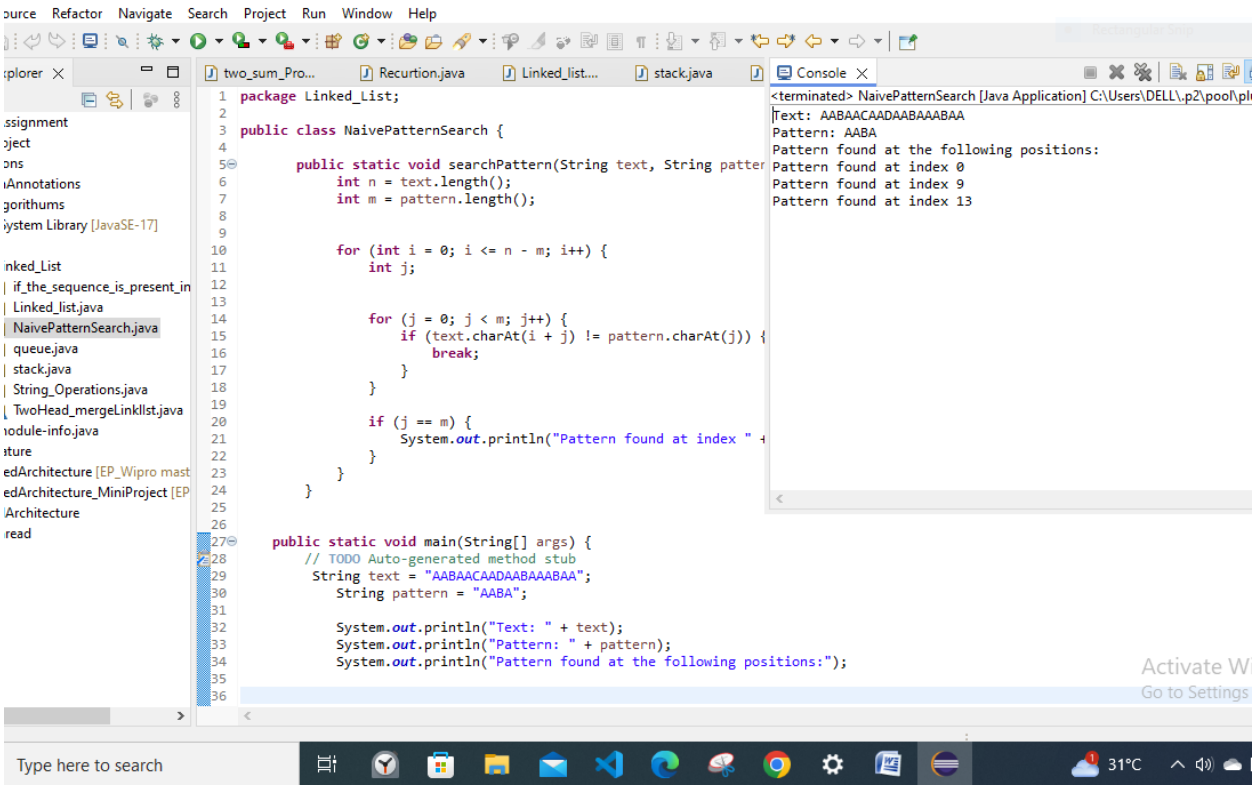
Steps:

1. **Iterate Through Text**: Loop through the text from the beginning to the position where the remaining substring is at least as long as the pattern.
2. **Check for Pattern Match**: For each position, check if the substring of the text starting at that position matches the pattern.
3. **Store or Print Match**: If a match is found, store or print the position.

searchPattern Method:

- Takes two parameters: text (the string to search within) and pattern (the substring to find).

- Iterates through the text from the start to the position where the remaining substring is at least as long as the pattern.
- For each position i, it checks if the substring of text starting at i matches the pattern.
- If a match is found (i.e., the inner loop completes), it prints the starting index i.



Task 3: Implementing the KMP Algorithm.

Code the Knuth-Morris-Pratt (KMP) algorithm in java for pattern searching which pre-processes the pattern to reduce the number of comparisons. Explain how this pre-processing improves the search time compared to the naive approach.

The Knuth-Morris-Pratt (KMP) algorithm is an efficient pattern searching algorithm that preprocesses the pattern to create a "longest prefix suffix" (LPS) array, which helps in reducing the number of comparisons during the search phase. The preprocessing step computes the LPS array, which is then used to avoid unnecessary comparisons in the text. This results in a time complexity of O(n+m)O(n + m)O(n+m), where nnn is the length of the text and mmm is the length of the pattern, making it significantly more efficient than the naive approach for large inputs.

Steps to Implement the KMP Algorithm:

1. **Preprocess the Pattern**:
    - Create the LPS array that stores the length of the longest proper prefix of the pattern which is also a suffix for every position in the pattern.
2. **Search the Pattern**:
    - Use the LPS array to skip unnecessary comparisons in the text.

LPS Array Construction:

- The LPS array helps in determining the next positions to match by storing the longest proper prefix which is also a suffix for substrings of the pattern.

 KMPSearch Method:

- Takes two parameters: pattern (the substring to find) and text (the string to search within).
- Calls computeLPSArray to preprocess the pattern and create the LPS array.
- Iterates through the text using indices i and j. If characters match, both indices are incremented.
- If j equals the length of the pattern, it means the pattern is found at index i - j.
- If a mismatch occurs after some matches, j is updated using the LPS array to avoid unnecessary comparisons.

Task 4: Rabin-Karp Substring Search.

Implement the Rabin-Karp algorithm for substring search using a rolling hash. Discuss the impact of hash collisions on the algorithm's performance and how to handle them.

The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to find any one of a set of pattern strings in a text. The key idea is to use a rolling hash function to efficiently compare the hash value of the pattern with the hash values of substrings of the text.

Steps to Implement Rabin-Karp Algorithm:

1. **Hash Function**: Compute the hash value for the pattern and the initial substring of the text.
2. **Sliding Window**: Slide the pattern over the text one character at a time, updating the hash value efficiently using the rolling hash technique.
3. **Compare Hashes**: If the hash values match, compare the actual substring with the pattern to check for a valid match.
4. **Handle Collisions**: Since hash collisions can occur (different strings having the same hash value), it's essential to verify the actual substring when a hash match is found.

Rolling Hash Function:

- The hash function typically used is a polynomial hash, where the hash value for a string sss of length mmm is computed as:
  hash(s)=(s[0]×basem−1+s[1]×basem−2+...+s[m−1]×base0)mod primehash(s) = (s[0] \times base^{m-1} + s[1] \times base^{m-2} + ... + s[m-1] \times base^0) \mod primehash(s)=(s[0]×basem−1+s[1]×basem−2+...+s[m−1]×base0)modprime
- The rolling hash function allows updating the hash value in constant time when sliding the window over the text.

Task 5: Boyer-Moore Algorithm Application.

Use the Boyer-Moore algorithm to write a function that finds the last occurrence of a substring in a given string and returns its index. Explain why this algorithm can outperform others in certain scenarios.

The Boyer-Moore algorithm is an efficient string searching algorithm that is particularly effective for large texts and patterns. It improves upon the naive approach by skipping sections of the text, thus reducing the number of comparisons. The algorithm preprocesses the pattern to create two arrays: the **bad character** and **good suffix** arrays, which help in deciding how many positions to skip when a mismatch occurs.

Steps to Implement Boyer-Moore Algorithm:

1. **Preprocess the Pattern**:
   - Create the **bad character** table that tells how far the search window can be shifted when a mismatch occurs.

- Optionally, create the **good suffix** table that tells how far to shift the window when a substring of the pattern matches but is followed by a mismatch.

2. **Search the Pattern**:
   - Use the precomputed tables to shift the pattern efficiently over the text.

Why Boyer-Moore Can Outperform Others:

1. **Skipping Sections of Text**: The Boyer-Moore algorithm can skip sections of the text entirely when a mismatch is found. This skipping is achieved by leveraging the bad character and good suffix rules, which can significantly reduce the number of comparisons.
2. **Efficient Mismatches Handling**: The preprocessing step (bad character table) allows the algorithm to quickly determine how far to shift the pattern in case of a mismatch.
3. **Optimized for Specific Cases**: It is particularly efficient when the pattern length is relatively long compared to the text. In the best case, the Boyer-Moore algorithm can achieve sublinear performance.

```
public class BoyerMoore {


  // Method to find the last occurrence of the pattern in the text
  public static int findLastOccurrence(String text, String pattern) {
    int n = text.length();
    int m = pattern.length();
    if (m == 0) return -1;


    // Preprocess the pattern to create the bad character table
    int[] badChar = new int[256];
    preprocessBadChar(pattern, m, badChar);


    int lastOccurrenceIndex = -1;
    int shift = 0;


    while (shift <= (n - m)) {
      int j = m - 1;
```

```
        // Move from right to left in the pattern
        while (j >= 0 && pattern.charAt(j) == text.charAt(shift + j)) {
            j--;
        }

        // If the pattern is found
        if (j < 0) {
            lastOccurrenceIndex = shift;
            shift += (shift + m < n) ? m - badChar[text.charAt(shift + m)] : 1;
        } else {
            shift += Math.max(1, j - badChar[text.charAt(shift + j)]);
        }
    }

    return lastOccurrenceIndex;
}

// Preprocess the pattern to create the bad character table
private static void preprocessBadChar(String pattern, int size, int[] badChar) {
    for (int i = 0; i < 256; i++) {
        badChar[i] = -1;
    }
    for (int i = 0; i < size; i++) {
        badChar[pattern.charAt(i)] = i;
    }
}

public static void main(String[] args) {
    String text = "ABAAABCD";
```

```
        String pattern = "ABC";


        int lastIndex = findLastOccurrence(text, pattern);
        System.out.println("Last occurrence of pattern is at index: " + lastIndex);
    }
}
```