DAY-11 JAVA ASSIGNMENT

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.

KMP Algorithm Explanation

The KMP algorithm improves the search time for pattern matching by avoiding unnecessary comparisons. It does this through a preprocessing step where it builds a partial match (also known as "prefix") table. This table is used to skip characters in the text string that we know will match the pattern string, thus reducing the number of comparisons needed.

Key Concepts

1. **Partial Match Table (LPS Array)**: The LPS (Longest Prefix which is also Suffix) array is built for the pattern. For each position in the pattern, the LPS array contains the length of the longest proper prefix which is also a suffix.
2. **Search Process**: While searching through the text, if a mismatch occurs after a certain number of matches, the LPS array is used to determine the next positions to check in the pattern without rechecking characters that we know will match.

Preprocessing to Build LPS Array

The preprocessing step constructs the LPS array which helps to determine the next positions to check in case of a mismatch.

How Preprocessing Improves Search Time

In the naive approach, every mismatch after a match requires the pattern to be shifted one position to the right and start again from the beginning of the pattern. This results in a worst-case time complexity of O(M * N), where M is the length of the pattern and N is the length of the text.

The KMP algorithm, however, utilizes the LPS array to skip unnecessary comparisons. When a mismatch occurs after j matches, the algorithm uses the LPS array to avoid rechecking characters that are known to match. This optimization results in an improved worst-case time complexity of O(M + N).

- **Time Complexity**:
  - Preprocessing: O(M) to build the LPS array.
  - Searching: O(N) to search the pattern in the text using the LPS array.

Thus, the overall time complexity of the KMP algorithm is O(M + N), making it significantly more efficient than the naive approach, especially for larger texts and patterns.

KMPAlgorithm.java

```java
package com.wipro.patterns;

public class KMPAlgorithm {

    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        search(text, pattern);
    }

    public static void search(String text, String pattern) {
        int[] lps = computeLPSArray(pattern);
        int i = 0; // index for text
        int j = 0; // index for pattern
        int M = pattern.length();
        int N = text.length();

        while (i < N) {
            if (pattern.charAt(j) == text.charAt(i)) {
                i++;
                j++;
            }

            if (j == M) {
                System.out.println("Pattern found at index " + (i - j));
                j = lps[j - 1];
            } else if (i < N && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
    }

    private static int[] computeLPSArray(String pattern) {
        int M = pattern.length();
        int[] lps = new int[M];
        int length = 0; // length of the previous longest prefix suffix
        int i = 1;
        lps[0] = 0; // lps[0] is always 0

        while (i < M) {
            if (pattern.charAt(i) == pattern.charAt(length)) {
                length++;
                lps[i] = length;
                i++;
            } else {
                if (length != 0) {
                    length = lps[length - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }
}
```

Console

```
<terminated> KMPAlgorithm [Java A
Pattern found at index 10
```

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.

Rabin-Karp Algorithm Explanation

The Rabin-Karp algorithm uses hashing to find a pattern in a text. It computes a hash value for the pattern and a hash value for each substring of the text of the same length as the pattern. If the hash values match, it then checks the actual substring to confirm a match. This approach allows for efficient pattern searching.

Key Concepts

1. **Hash Function**: A good hash function distributes values uniformly and minimizes collisions.
2. **Rolling Hash**: To efficiently compute hash values for the sliding window of substrings in the text, the rolling hash technique is used. It allows the hash value of the next substring to be computed in constant time.

Handling Hash Collisions

- **Hash Collisions**: Two different strings may have the same hash value (a collision). To handle this, whenever a hash match is found, the actual substring is compared to the pattern to confirm the match.
- **Modulo Operation**: A prime number is used in the modulo operation to reduce the likelihood of collisions.

Impact of Hash Collisions

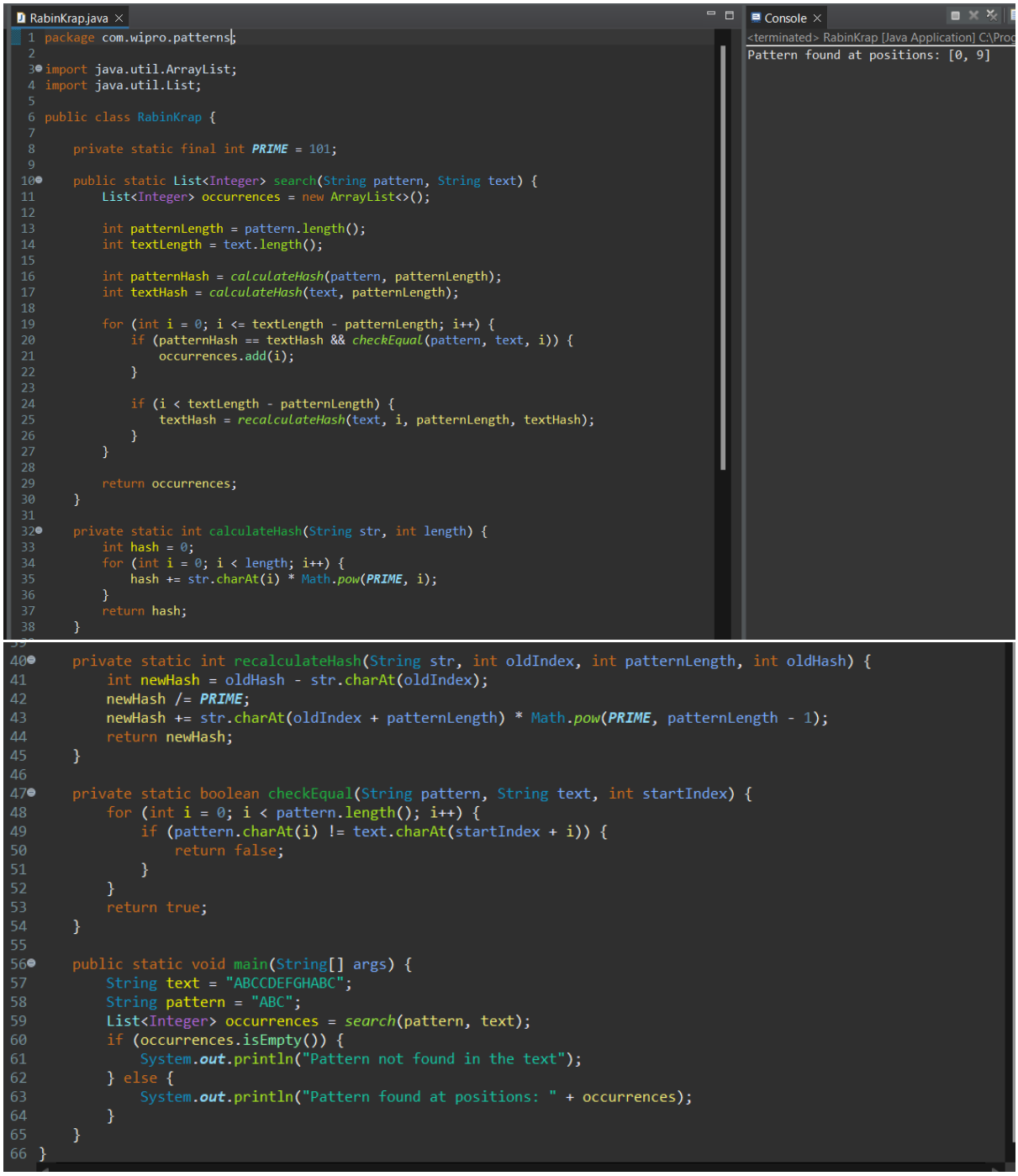
- **Hash Collisions**: When different substrings produce the same hash value, it leads to hash collisions. In the Rabin-Karp algorithm, hash collisions cause unnecessary substring comparisons, impacting performance.
- **Performance**: In the average case, the Rabin-Karp algorithm runs in O(N + M) time. However, if there are many hash collisions, the performance degrades to O(NM) in the worst case.

Handling Hash Collisions

- **Verification Step**: After finding a hash match, perform a direct string comparison to verify the match. This step ensures that false positives due to hash collisions are correctly handled.
- **Choice of q**: Use a large prime number for the modulo operation to minimize the probability of collisions.

Conclusion

The Rabin-Karp algorithm is efficient for multiple pattern searches and large texts due to its hashing and rolling hash mechanism. However, careful handling of hash collisions through verification and appropriate choice of hash function parameters is crucial for maintaining its efficiency.



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.

Boyer-Moore Algorithm Explanation

The Boyer-Moore algorithm is efficient because it uses information gained from mismatches to skip sections of the text, potentially moving more than one character ahead with each step. It uses two heuristics:

1. **Bad Character Heuristic**: When a mismatch occurs, the algorithm skips alignments that cannot possibly match based on the character that caused the mismatch.
2. **Good Suffix Heuristic**: When a mismatch occurs, the algorithm uses information about the matched suffix to skip sections of the text where the same suffix would appear.

Why Boyer-Moore Can Outperform Other Algorithms

1. **Large Alphabet Advantage**: Boyer-Moore is particularly efficient for large alphabets. The bad character heuristic can skip large sections of the text, especially when the alphabet size is large.
2. **Mismatches**: The algorithm skips sections of the text more aggressively on mismatches compared to algorithms like KMP. This is especially beneficial when the pattern length is long and mismatches are frequent.
3. **Performance in Practice**: While the worst-case time complexity is O(MN) (same as the naive algorithm), in practice, Boyer-Moore often performs much better than KMP and Rabin-Karp due to fewer comparisons. It is closer to O(N/M) in many practical scenarios.
4. **Flexibility with Different Patterns**: The algorithm performs well even with varied and complex patterns due to the dual heuristics that complement each other.

Conclusion

The Boyer-Moore algorithm's use of the bad character and good suffix heuristics allows it to skip large parts of the text, making it more efficient in practice compared to other algorithms. This efficiency is especially notable with large alphabets and longer patterns, making it a powerful tool for string searching tasks.

BoyerMooreAlgorithm.java

```java
package com.wipro.patterns;

public class BoyerMooreAlgorithm {

    // Function to find the last occurrence of a pattern in a text
    public static int lastOccurrence(String txt, String pat) {
        int[] badChar = badCharHeuristic(pat);
        int[] goodSuffix = goodSuffixHeuristic(pat);

        int m = pat.length();
        int n = txt.length();

        int s = 0; // s is the shift of the pattern with respect to text
        int lastOccurrence = -1;

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

            // Keep reducing index j of pattern while characters of pattern and text are matchin
            while (j >= 0 && pat.charAt(j) == txt.charAt(s + j)) {
                j--;
            }

            // If the pattern is present at the current shift, update lastOccurrence
            if (j < 0) {
                lastOccurrence = s;
                s += (s + m < n) ? m - badChar[txt.charAt(s + m)] : 1;
            } else {
                s += Math.max(1, j - badChar[txt.charAt(s + j)]);
            }
        }

        return lastOccurrence;
    }

    private static int[] badCharHeuristic(String pat) {
        int[] badChar = new int[256]; // Assuming ASCII character set
        int m = pat.length();

        for (int i = 0; i < 256; i++) {
            badChar[i] = -1; // Initialize all occurrences as -1
        }

        for (int i = 0; i < m; i++) {
            badChar[pat.charAt(i)] = i; // Fill the actual value of last occurrence of a charact
        }

        return badChar;
    }

    private static int[] goodSuffixHeuristic(String pat) {
        int m = pat.length();
        int[] suffix = new int[m];
        int[] shift = new int[m];

        for (int i = 0; i < m; i++) {
            suffix[i] = -1;
        }

        int lastPrefixPosition = m;
        for (int i = m - 1; i >= 0; i--) {
            if (isPrefix(pat, i + 1)) {
                lastPrefixPosition = i + 1;
            }
            shift[m - 1 - i] = lastPrefixPosition - i + m - 1;
        }

        for (int i = 0; i < m - 1; i++) {
            int slen = suffixLength(pat, i);
            shift[slen] = m - 1 - i + slen;
        }

        return shift;
    }
```

Console

```
<terminated> BoyerMooreAlgorithm [Java Application] C:\
Last occurrence of pattern is at index 10
```

```
    private static boolean isPrefix(String pat, int p) {
        int m = pat.length();
        for (int i = p, j = 0; i < m; i++, j++) {
            if (pat.charAt(i) != pat.charAt(j)) {
                return false;
            }
        }
        return true;
    }

    private static int suffixLength(String pat, int p) {
        int m = pat.length();
        int len = 0;
        for (int i = p, j = m - 1; i >= 0 && pat.charAt(i) == pat.charAt(j); i--, j--) {
            len++;
        }
        return len;
    }

    // Driver code
    public static void main(String[] args) {
        String txt = "abacaabadcabacabaabb";
        String pat = "abacab";
        int result = lastOccurrence(txt, pat);
        if (result == -1) {
            System.out.println("Pattern not found");
        } else {
            System.out.println("Last occurrence of pattern is at index " + result);
        }
    }
}
```