

Assignment Day-11

Core Java with DS and Algorithms

Name: Joshnitha Rangolu

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.

```
package pattern_searchingAlg;

import java.util.Scanner;

public class StringOperations {

    public static String extractMiddleSubstring(String str1, String str2, int
substringLength) {

        String concatenatedString = str1 + str2;

        String reversedString = new
        StringBuilder(concatenatedString).reverse().toString();

        // Handle edge cases: empty strings or substring length larger than
        concatenated string length

        if (reversedString.isEmpty() || substringLength > reversedString.length())
        {

            return "";

        }

        // Extract the middle substring considering zero-based indexing

        int middleStartIndex = (reversedString.length() - substringLength) / 2;

        return reversedString.substring(middleStartIndex, middleStartIndex +
        substringLength);

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter first string: ");

        String string1 = scanner.nextLine();

        System.out.print("Enter second string: ");
```

```

String string2 = scanner.nextLine();

System.out.print("Enter desired substring length: ");

int substringLen;

try {

substringLen = Integer.parseInt(scanner.nextLine());

} catch (NumberFormatException e) {

System.err.println("Invalid input: Please enter an integer for substring
length.");

return;

}

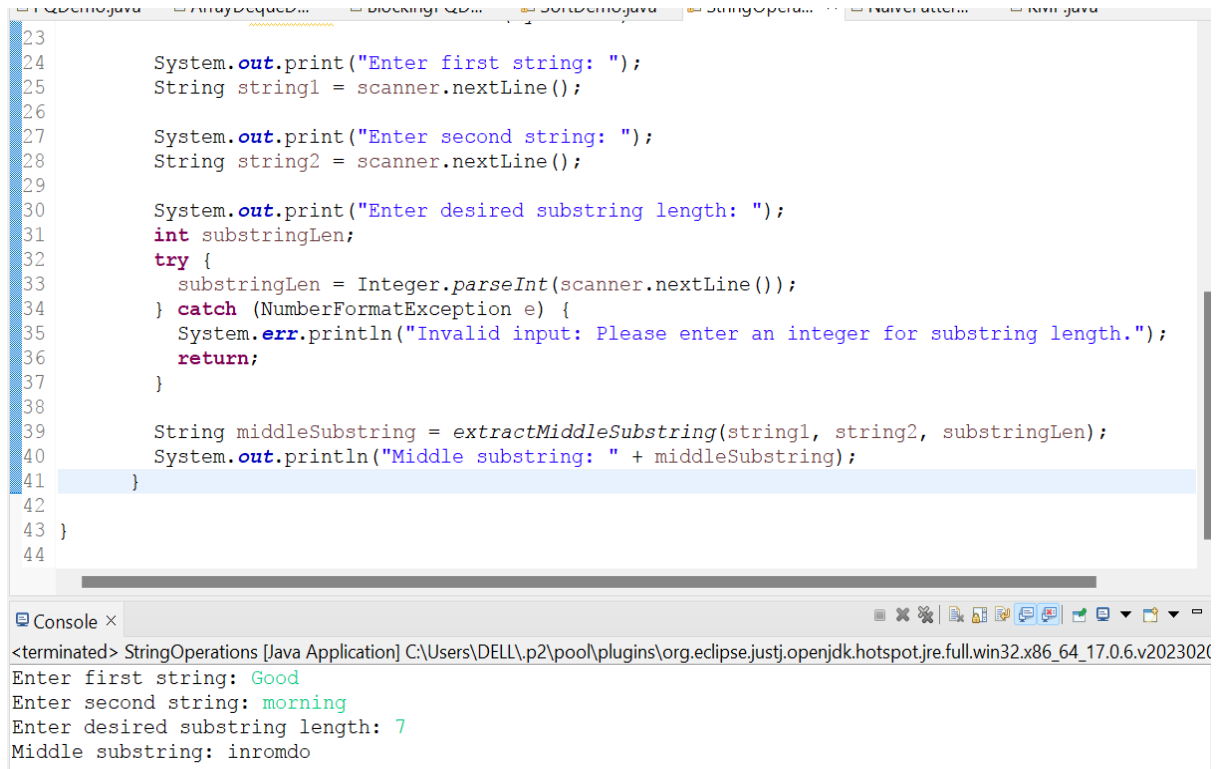
String middleSubstring = extractMiddleSubstring(string1, string2,
substringLen);

System.out.println("Middle substring: " + middleSubstring);

}

}

```



The screenshot shows the Eclipse IDE with a Java file named `StringOperations.java` open. The code in the editor is as follows:

```

23
24     System.out.print("Enter first string: ");
25     String string1 = scanner.nextLine();
26
27     System.out.print("Enter second string: ");
28     String string2 = scanner.nextLine();
29
30     System.out.print("Enter desired substring length: ");
31     int substringLen;
32     try {
33         substringLen = Integer.parseInt(scanner.nextLine());
34     } catch (NumberFormatException e) {
35         System.err.println("Invalid input: Please enter an integer for substring length.");
36         return;
37     }
38
39     String middleSubstring = extractMiddleSubstring(string1, string2, substringLen);
40     System.out.println("Middle substring: " + middleSubstring);
41 }
42
43 }
44

```

Below the editor, the **Console** window is visible, showing the output of the program:

```

<terminated> StringOperations [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230201
Enter first string: Good
Enter second string: morning
Enter desired substring length: 7
Middle substring: inromdo

```

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.

```
package pattern_searchingAlg;

public class NaivePatternSearch {

    public static void search(String text, String pattern) {

        int comparisons = 0;

        int textLength = text.length();

        int patternLength = pattern.length();

        for (int i = 0; i <= textLength - patternLength; i++) {

            int j = 0;

            while (j < patternLength && text.charAt(i + j) == pattern.charAt(j)) {

                comparisons++;

                j++;

            }

            if (j == patternLength) {

                System.out.println("Pattern found at index: " + i);

            }

        }

        System.out.println("Total comparisons: " + comparisons);

    }

    public static void main(String[] args) {

        String text = "AABAACAADAABAABA";

        String pattern = "AABAA";

        search(text, pattern);

    }

}
```

```

}
1 package pattern_searchingAlg;
2
3 public class NaivePatternSearch {
4
5     public static void search(String text, String pattern) {
6         int comparisons = 0;
7         int textLength = text.length();
8         int patternLength = pattern.length();
9
10        for (int i = 0; i <= textLength - patternLength; i++) {
11            int j = 0;
12            while (j < patternLength && text.charAt(i + j) == pattern.charAt(j)) {
13                comparisons++;
14                j++;
15            }
16
17            if (j == patternLength) {
18                System.out.println("Pattern found at index: " + i);
19            }
20        }
21        System.out.println("Total comparisons: " + comparisons);
22    }
23
24    public static void main(String[] args) {

```

Console ×

```

<terminated> NaivePatternSearch [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v
Pattern found at index: 0
Pattern found at index: 9
Total comparisons: 18

```

Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in C# 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.

```

package pattern_searchingAlg;

public class KMP {

    void KMPSearch(String pat, String txt)

    {

        int M = pat.length();

        int N = txt.length();

        int lps[] = new int[M];

        int j = 0;

        computeLPSArray(pat, M, lps);

        int i = 0;

```

```

while ((N - i) >= (M - j)) {

    if (pat.charAt(j) == txt.charAt(i)) {

        j++;

        i++;

    }

    if (j == M) {

        System.out.println("Found pattern " + "at index " + (i - j));

        j = lps[j - 1];

    }

    else if (i < N

        && pat.charAt(j) != txt.charAt(i)) {

        if (j != 0)

            j = lps[j - 1];

        else

            i = i + 1;

    }

}

void computeLPSArray(String pat, int M, int lps[])

{

    int len = 0;

    int i = 1;

    lps[0] = 0;

    while (i < M) {

        if (pat.charAt(i) == pat.charAt(len)) {

            len++;

            lps[i] = len;

            i++;

        }

    }

}

```

```
}

else

if (len != 0) {

len = lps[len - 1];

}

else

{

lps[i] = len;

i++;

}

}

}

public static void main(String args[])

{

String txt = "ABABDABACDABABCABAB";

String pat = "ABABCABAB";

new KMP().KMPSearch(pat, txt);

}

}
```

```
1 package pattern_searchingAlg;
2
3 public class KMP {
4
5     void KMPSearch(String pat, String txt)
6     {
7         int M = pat.length();
8         int N = txt.length();
9
10        int lps[] = new int[M];
11        int j = 0;
12        computeLPSArray(pat, M, lps);
13
14        int i = 0;
15        while ((N - i) >= (M - j)) {
16            if (pat.charAt(j) == txt.charAt(i)) {
17                j++;
18                i++;
19            }
20            if (j == M) {
21                System.out.println("Found pattern " + "at index " + (i - j));
22                j = lps[j - 1];
23            }
24            else if (i < N
```

```
<terminated> KMP [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1
Found pattern at index 10
```

How this pre-processing improves the search time compared to the naive approach.

The naive approach compares each character in the pattern with the corresponding character in the text.

The KMP algorithm pre-processes the pattern to create the LPP table. This table tells us, for a given mismatch, how many characters we can safely shift the pattern by without making unnecessary comparisons. This reduces redundant comparisons, especially when there are partial matches followed by mismatches.

In the worst case, both algorithms have the same time complexity of $O(n * m)$, where n is the text length and m is the pattern length. However, in many practical cases with repetitive patterns, the KMP algorithm can significantly reduce the number of comparisons compared to the naive approach.

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.

```
package pattern_searchingAlg;

public class RabinKrap {

    public final static int d = 256;

    static void search(String pat, String txt, int q)
```

```

{

int M = pat.length();

int N = txt.length();

int i, j;

int p = 0;

int t = 0;

int h = 1;

for (i = 0; i < M - 1; i++)

h = (h * d) % q;

for (i = 0; i < M; i++) {

p = (d * p + pat.charAt(i)) % q;

t = (d * t + txt.charAt(i)) % q;

}

for (i = 0; i <= N - M; i++) {

if (p == t) {

for (j = 0; j < M; j++) {

if (txt.charAt(i + j) != pat.charAt(j))

break;

}

if (j == M)

System.out.println(

"Pattern found at index " + i);

}

if (i < N - M) {

t = (d * (t - txt.charAt(i) * h)

+ txt.charAt(i + M))

% q;

if (t < 0)

```



```

t = (t + q);

}

}

}

public static void main(String[] args)

{

String txt = "AABAACAADAABAABA";

String pat = "AABAA";

int q = 101;

search(pat, txt, q);

}

}

```

```

27         System.out.println(
28             "Pattern found at index " + i);
29     }
30     if (i < N - M) {
31         t = (d * (t - txt.charAt(i) * h)
32             + txt.charAt(i + M))
33             % q;
34         if (t < 0)
35             t = (t + q);
36     }
37 }
38 }
39 public static void main(String[] args)
40 {
41     String txt = "AABAACAADAABAABA";
42     String pat = "AABAA";
43     int q = 101;
44     search(pat, txt, q);
45 }
46 }
47 }
48

```

Console ×

```

<terminated> RabinKrap [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230
Pattern found at index 0
Pattern found at index 9

```

Impact of Hash Collisions:

Hash collisions occur when different substrings have the same hash value. In the Rabin-Karp algorithm, a collision leads to a false positive - the hash values match, but the substrings might not be actually identical.

- **Performance:** Hash collisions can lead to unnecessary character-by-character comparisons, increasing the execution time.
- **Handling Collisions:** The Rabin-Karp algorithm relies on explicit character comparison to confirm a match after a hash value match. This helps to eliminate false positives due to collisions.

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.

```
package pattern_searchingAlg;

public class BoyerMooreAlgorithm {

    static int NO_OF_CHARS = 256;

    static int max(int a, int b) { return (a > b) ? a : b; }

    static void badCharHeuristic(char[] str, int size,
int badchar[])
    {

        for (int i = 0; i < NO_OF_CHARS; i++)

            badchar[i] = -1;

        for (int i = 0; i < size; i++)

            badchar[(int)str[i]] = i;

    }

    static void search(char txt[], char pat[])

    {

        int m = pat.length;

        int n = txt.length;

        int badchar[] = new int[NO_OF_CHARS];

        badCharHeuristic(pat, m, badchar);

        int s = 0;

        while (s <= (n - m)) {
```

```

int j = m - 1;

while (j >= 0 && pat[j] == txt[s + j])

j--;

if (j < 0) {

System.out.println(

"Patterns occur at shift = " + s);

s += (s + m < n) ? m - badchar[txt[s + m]]

: 1;

}

else

s += max(1, j - badchar[txt[s + j]]);

}

}

public static void main(String[] args)

{

char txt[] = "ABAAABCD".toCharArray();

char pat[] = "ABC".toCharArray();

search(txt, pat);

}

}

```

```

1 package pattern_searchingAlg;
2
3 public class BoyerMooreAlgorithm {
4
5     static int NO_OF_CHARS = 256;
6     static int max(int a, int b) { return (a > b) ? a : b; }
7     static void badCharHeuristic(char[] str, int size,
8                                 int badchar[])
9     {
10         for (int i = 0; i < NO_OF_CHARS; i++)
11             badchar[i] = -1;
12         for (int i = 0; i < size; i++)
13             badchar[(int)str[i]] = i;
14     }
15     static void search(char txt[], char pat[])
16     {
17         int m = pat.length;
18         int n = txt.length;
19
20         int badchar[] = new int[NO_OF_CHARS];
21         badCharHeuristic(pat, m, badchar);
22     }
23 }

```

Console ×

<terminated> BoyerMooreAlgorithm [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v

Patterns occur at shift = 4

Why Boyer-Moore can outperform others:

- **Reduced comparisons:** The bad character table helps the algorithm avoid redundant character comparisons, especially when the pattern contains characters that frequently mismatch in the text.
- **Right-to-left approach:** By aligning and comparing from the right end of the pattern, the algorithm leverages the information about mismatched characters to perform larger shifts, potentially skipping over a significant portion of the text that likely won't contain a match.

This makes the Boyer-Moore algorithm a good choice for searching text with patterns that may contain repetitive characters or where the expected outcome is finding the pattern towards the end of the text (e.g., searching for keywords in a document). However, the pre-processing step and the worst-case time complexity of $O(nm)$ can be drawbacks in some scenarios.