# Day 17 Assignment

1. **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.**

```
public static String getConcatenatedMiddle(String string, int start, int end)
throws IndexOutOfBoundsException {

   String newstring = string.concat(string);
   String text = "";

   if (start < 0 || end >= newstring.length() || start > end ||
string.isEmpty()) {
       throw new IndexOutOfBoundsException();
   }

   for (int i = 0; i < newstring.length(); i++) {
       text = newstring.charAt(i) + text;
   }

   return text.substring(start, end);
}
```

```
package m5_core_java_programming.day_17;

import m5_core_java_programming.mycollection.PatternMatcher;

/*
   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.
*/

public class Assignment_1 {
   public static void main(String[] args) {
       System.out.println("Giving string sayan and asking for substring from
index 4 to 8");
       System.out.println(PatternMatcher.getConcatenatedMiddle("sayan",3,8));
   }
}
```

**Output**

```
C:\Users\coolr\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Files
Giving string sayan and asking for substring from index 4 to 8
asnay

Process finished with exit code 0
```

---

2. **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.**

In the example, “aaaaa” and “aaaab”. It's the worst case for this naive algorithm.
Reason it's the worst case and takes 9 steps for 5 length string is that :
It has go back the entire length of the pattern when it does not find the match which makes it run for in the worst case n*m time where n is the length of the string and m is the length of the pattern.

```java
public static int indexAt(String string, String pattern, String method) {
    if (method.equalsIgnoreCase("naive")) {
        return naive(string, pattern);
    } else if (method.equalsIgnoreCase("kmp")) {
        return kmp(string, pattern);
    } else if (method.equalsIgnoreCase("rabin")) {
        return rabin(string, pattern);
    } else if (method.equalsIgnoreCase("boyer")) {
        return boyer(string, pattern);
    } else {
        return -1;
    }
}
```

```java
private static int naive(String string, String pattern) {
    int i = 0;
    int j = 0;
    int steps = 0;
    if (string.length() < pattern.length()) {
        return -1;
    }
    while (i < string.length()) {
        if (string.toLowerCase().charAt(i) == pattern.toLowerCase().charAt(j)) {
            j++;
        } else {
            i = i - j;
```

```
            j = 0;
        }
        steps++;
        if (j == pattern.length()) {
            System.out.println("Number of steps involved : " + steps);
            return i - pattern.length() + 1;
        }
        i++;
    }
    System.out.println("Number of steps involved : " + steps);
    return -1;
}
```

```
package m5_core_java_programming.day_17;

import m5_core_java_programming.mycollection.PatternMatcher;

/*
   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.
*/
public class Assignment_2 {
    public static void main(String[] args) {
        String string = "aaaaa";
        String pattern = "aaaab";

        System.out.println("String to find from : " + string);
        System.out.println("String to find own : " + pattern);

        System.out.println(PatternMatcher.indexAt(string, pattern, "naive"));
    }
}
```

**Output**

```
C:\Users\coolr\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:
String to find from : aaaaa
String to find own : aaaab
Number of steps involved : 9
-1

Process finished with exit code 0
```

---

3. **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.**

With the same example “aaaaa” and “aaaab”. KMP seems to do better in performance because with each mismatch naive has to manually run the loop from the entire length of the pattern again. Whereas in the kmp due to the preprocessed lookup table we have the last prefix index of the current matching character which only shifts the pattern to the index directly. Also the pointer of the main string does not need to be reset as it makes sure there is a matching prefix available of length varies from 0 to m-1 where m is the length of the pattern. At worst case it will only run for n+m where n is the length of string and m is the length of the pattern.

```
private static void prefixTable(String pattern, int[] table) {
    int j = 0;
    for (int i = 1; i < pattern.length(); i++) {
        if (pattern.charAt(i) == pattern.charAt(j)) {
            table[i] = j;
            j++;
        } else {
            while (j != 0 && pattern.charAt(i) != pattern.charAt(j)) {
                j = table[j - 1];
            }
        }
    }
}
```

```
private static int kmp(String string, String pattern) {
    if (string.length() < pattern.length()) {
        return -1;
    }
    int[] table = new int[pattern.length()];
    int steps = 0;

    prefixTable(pattern, table);

    int j = 0;
    for (int i = 0; i < string.length(); ) {
        if (string.charAt(i) == pattern.charAt(j)) {
            j++;
            i++;
            steps++;
        } else {
            while (j != 0 && string.charAt(i) != pattern.charAt(j)) {
                j = table[j - 1];
                steps++;
            }
```

```
            if (j == 0) {

                if (pattern.charAt(j) == string.charAt(i)) {
                    j++;
                }
            }
            i++;
        }

        if (j == pattern.length()) {
            System.out.println("Number of steps involved : " + steps);
            return i - j;
        }
    }
    System.out.println("Number of steps involved : " + steps);
    return -1;
}
```

```
package m5_core_java_programming.day_17;

/*
   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.
*/

import m5_core_java_programming.mycollection.PatternMatcher;

public class Assignment_3 {
    public static void main(String[] args) {
        String string = "aaaaa";
        String pattern = "aaaab";

        System.out.println("String to find from : " + string);
        System.out.println("String to find own : " + pattern);

        System.out.println(PatternMatcher.indexAt(string, pattern, "kmp"));
    }
}
```

**Output**

```
C:\Users\coolr\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\P
String to find from : aaaaa
String to find own : aaaab
Number of steps involved : 5
-1

Process finished with exit code 0
```

---

4. **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.**

Hash Collision occurs when two unequal keys generate the same hash. It creates confusion and problem in the logic also it increases this algorithm n*m where n is the length of string and m is length of the pattern. To handle hash, first method is to check the substring and the pattern to confirm the equality but it will run for n*m. Second method is to choose a good hash function which has low collisions and more uniqueness. For example, multiplier and modulus can be the number of characters and prime number respectively for unique hash generation.

```
private static int rabin(String string, String pattern) {
    int multiplier = 127;
    int mod = 53;

    int pow = 1;

    for (int i = 0; i < pattern.length() - 1; i++)
        pow = (pow * multiplier) % mod;

    int hashPat = 0;
    int hashString = 0;

    for (int i = 0; i < pattern.length(); i++) {
        hashPat = (multiplier * hashPat + pattern.charAt(i)) % mod;
        hashString = (multiplier * hashString + string.charAt(i)) % mod;
    }

    for (int i = 0; i <= string.length() - pattern.length(); i++) {
        if (hashPat == hashString) {
            if (equal(string, pattern, i)) {
                return i;
            }
```

```
        }
        if (i >= string.length() - pattern.length()) {
            return -1;
        }
        hashString = (multiplier * (hashString - string.charAt(i) * pow) +
string.charAt(i + pattern.length())) % mod;

        if (hashString < 0) {
            hashString += mod;
        }
    }
    return -1;
}
```

```
private static boolean equal(String string, String pattern, int index) {
    int j = 0;
    for (int i = index; i < string.length() && j < pattern.length(); i++, j++) {
        if (string.charAt(i) != pattern.charAt(j)) {
            return false;
        }
    }
    return j == pattern.length();
}
```

```
package m5_core_java_programming.day_17;

import m5_core_java_programming.mycollection.PatternMatcher;

/*
   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.
*/
public class Assignment_4 {
    public static void main(String[] args) {
        String string = "aaaaaba";
        String pattern = "ab";

        System.out.println("String to find from : " + string);
        System.out.println("String to find own : " + pattern);

        System.out.println(PatternMatcher.indexAt(string, pattern, "rabin"));
    }
}
```

**Output**

```
C:\Users\coolr\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Files\JetB
String to find from : aaaaaba
String to find own : ab
4

Process finished with exit code 0
```

---

5. **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.**

It outperforms other string matching algorithms in the case where characters are unique and patterns might be long. Unique character helps create a good map thus making the search much faster. Long patterns and unmatched characters means the algorithm is absolutely sure there may or may not be any match available for next m distance which is the length of pattern in the string. Thus making skips easy.

```
private static int boyer(String string, String pattern) {
    if (string.length() < pattern.length()) {
        return -1;
    }

    HashMap<Character, Integer> mp = new HashMap<>();
    int lastOcc = -1;
    for (int i = 0; i < pattern.length(); i++) {
        mp.put(pattern.charAt(i), i);
    }

    int i = pattern.length() - 1;
    int j = pattern.length() - 1;

    while (i < string.length()) {
        if (string.charAt(i) == pattern.charAt(j)) {
            i++;
            j++;
        } else {
            if (mp.containsKey(string.charAt(i))) {
                j = mp.get(string.charAt(i));
            } else {
                i += pattern.length();
                j = pattern.length() - 1;
            }
        }
```

```
        if (j >= pattern.length()) {
            lastOcc = i - pattern.length();
            j = pattern.length()-1;
        }
    }
    return lastOcc;
}
```

```
package m5_core_java_programming.day_17;

import m5_core_java_programming.mycollection.PatternMatcher;

/*
   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.
*/
public class Assignment_5 {
    public static void main(String[] args) {
        String string = "aabaababab";
        String pattern = "aba";

        System.out.println("String to find from : " + string);
        System.out.println("String to find own : " + pattern);

        System.out.println(PatternMatcher.indexAt(string, pattern, "boyer"));
    }
}
```

**Output**

```
C:\Users\coolr\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:C:\Program Fil
String to find from : aabaababab
String to find own : aba
6

Process finished with exit code 0
```

---

**Tools Used :**

IntelliJ IDE
java version "1.8.0_411"
Java(TM) SE Runtime Environment (build 1.8.0_411-b09)
Java HotSpot(TM) Client VM (build 25.411-b09, mixed mode, sharing)