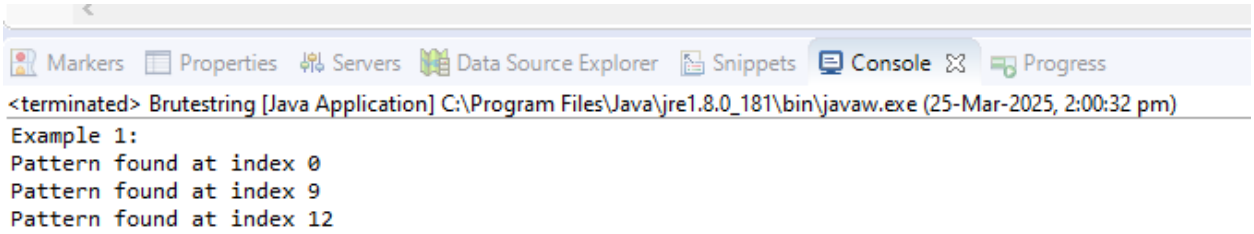
UNIT-6 Sting Matching

Program-20 Given a text txt [0...n-1] and a pattern pat [0...m-1], prints all occurrences of pat [ ] in txt [ ] by using the Brute force string matching approach. You may assume that n > m.

Program-23 Given a text txt [0...n-1] and a pattern pat [0...m-1], prints all occurrences of pat [ ] in txt [ ] by using the Naïve string matching approach. You may assume that n > m.

```
public class Brutestring {


    public static void search(String pat, String txt) {
        int lp = pat.length(); // Pattern length
        int ls = txt.length(); // Text length

        // A loop to slide pat[] one by one
        for (int i = 0; i <= ls - lp; i++) {
            int j;

            // For current index i, check for pattern match
            for (j = 0; j < lp; j++) {
                if (txt.charAt(i + j) != pat.charAt(j)) {
                    break;
                }
            }

            // If pattern matches at index i
            if (j == lp) {
                System.out.println("Pattern found at index " + i);
            }
        }
    }

    public static void main(String[] args) {

        String txt1 = "AABAACAADAABAABA";
        String pat1 = "AABA";
        System.out.println("Example 1:");
        search(pat1, txt1);
```

```
    }}
```

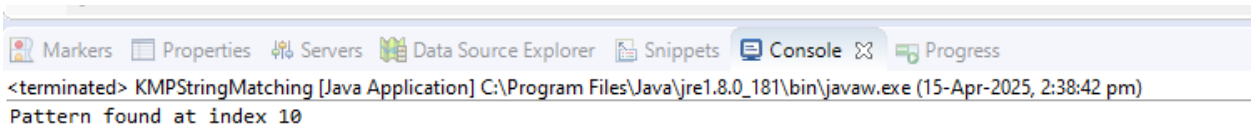


Program 21 - Given a text txt [0...n-1] and a pattern pat [0...m-1], prints all occurrences of pat [ ]in txt [ ] by using the KMP approach. You may assume that n > m.

```
public class KMPStringMatching {

   // Method to compute the Longest Prefix Suffix (LPS) array
   public static int[] computeLPSArray(String pattern) {
      int length = 0; // length of the previous longest prefix suffix
      int i = 1;
      int lps[] = new int[pattern.length()];
      lps[0] = 0; // lps[0] is always 0

      while (i < pattern.length()) {
         if (pattern.charAt(i) == pattern.charAt(length)) {
            length++;
            lps[i] = length;
            i++;
         } else {
            if (length != 0) {
               length = lps[length - 1];
               // Don't increment i here
            } else {
               lps[i] = 0;
               i++;
            }
         }
      }
      return lps;
   }
```

```
    // KMP search algorithm
    public static void KMPSearch(String pattern, String text) {
        int m = pattern.length();
        int n = text.length();

        int[] lps = computeLPSArray(pattern);

        int i = 0; // index for text[]
        int j = 0; // index for pattern[]

        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]; // Look for next match
            } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
                // Mismatch after j matches
                if (j != 0)
                    j = lps[j - 1];
                else
                    i++;
            }
        }
    }

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

        KMPSearch(pattern, text);
    }
}
```

OUTPUT:



- **Text**: ABABDABACDABABCABAB

- **Pattern**: ABABCABAB

Step 1: Build the LPS Array for the Pattern

The LPS array tells us the **length of the longest prefix which is also a suffix** for the substring pattern[0..i].
It helps to avoid redundant comparisons when a mismatch happens.

i	pattern[0..i]	LPS[i]	Reason
0	A	0	No proper prefix and suffix
1	AB	0	No match
2	ABA	1	A (prefix) == A (suffix)
3	ABAB	2	AB (prefix) == AB (suffix)
4	ABABC	0	No match
5	ABABCA	1	A (prefix) == A (suffix)
6	ABABCAB	2	AB (prefix) == AB (suffix)
7	ABABCABA	3	ABA (prefix) == ABA (suffix)
8	ABABCABAB	4	ABAB (prefix) == ABAB (suffix)

✅ Final LPS array: `[0, 0, 1, 2, 0, 1, 2, 3, 4]`

Step 2: Pattern Search Using KMP

We now use the LPS array to efficiently search the pattern inside the text.

Setup:

- i = index for **text**
- j = index for **pattern**

Start comparing text[i] with pattern[j]:

🔄 Steps (i → text index, j → pattern index):

Step	i (text)	j (pattern)	text[i] == pattern[j]?	Action
1	0	0	A == A	i++, j++
2	1	1	B == B	i++, j++
3	2	2	A == A	i++, j++
4	3	3	B == B	i++, j++
5	4	4	D != C	j = LPS[3] = 2 (backtrack pattern)
6	4	2	D != A	j = LPS[1] = 0
7	4	0	D != A	i++
8	5	0	A == A	i++, j++
9	6	1	B == B	i++, j++
10	7	2	A == A	i++, j++
11	8	3	C != B	j = LPS[2] = 1
12	8	1	C != B	j = LPS[0] = 0
13	8	0	C != A	i++
14	9	0	D != A	i++
15	10	0	A == A	i++, j++
16	11	1	B == B	i++, j++
17	12	2	A == A	i++, j++
18	13	3	B == B	i++, j++
19	14	4	C == C	i++, j++
20	15	5	A == A	i++, j++
21	16	6	B == B	i++, j++
22	17	7	A == A	i++, j++
23	18	8	B == B	i++, j++ → j = 9 = pattern.length

Pattern found at index (i - j) = 19 - 9 = 10

Program-22 Given a text txt [0...n-1] and a pattern pat [0...m-1], prints all occurrences of pat [ ] in txt [ ] by using the Rabin Krap approach. You may assume that n > m.

Step-by-Step Execution

Input:

```
Text:    "ABCCDDAEFG"
Pattern: "CDD"
```

Step 1: Compute Initial Hash Values

Using **mod 101**, we calculate:

- **Hash of pattern ("CDD")**: `p = Hash("CDD")`
- **Hash of first window in text ("ABC")**: `t = Hash("ABC")`

Step 2: Slide Over Text

- If `p == t`, check character by character.
- Otherwise, update the hash for the next window efficiently using:

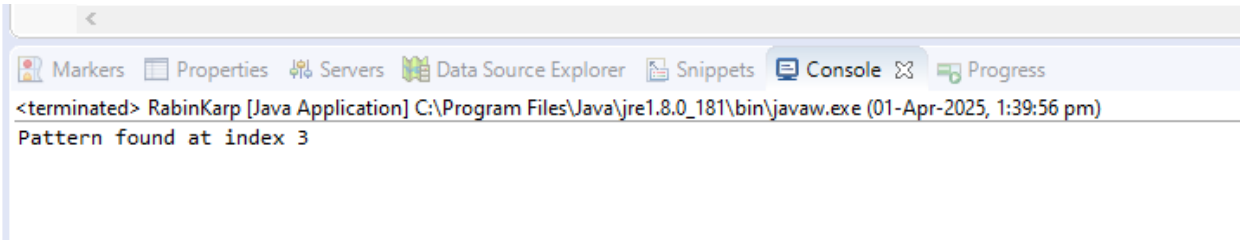
```
t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + m)) % q;
```

- Repeat until the pattern is found or the end of the text is reached.

Step 3: Output `Pattern found at index 2`

```
import java.util.*;

class RabinKarp {
    static final int d = 256; // Number of characters in input alphabet
    static final int q = 101; // A prime number for modulo operations (reduces hash
collisions)

    // Rabin-Karp search function
    static void search(String pat, String txt) {
        int m = pat.length();
        int n = txt.length();
        int p = 0; // Hash value for pattern
        int t = 0; // Hash value for text window
        int h = 1; // The value of d^(m-1) % q

        // Precompute h = d^(m-1) % q
```

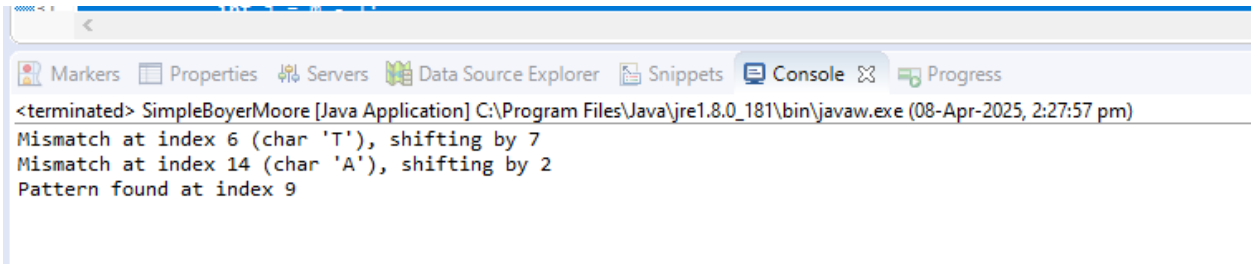
```
        for (int i = 0; i < m - 1; i++)
            h = (h * d) % q;

        // Calculate initial hash values for pattern and first text window
        for (int i = 0; i < m; i++) {
            p = (d * p + pat.charAt(i)) % q;
            t = (d * t + txt.charAt(i)) % q;
        }

        // Slide the pattern over the text one character at a time
        for (int i = 0; i <= n - m; i++) {
            // Check if hash values match
            if (p == t) {
                // If hash values match, do character check
                boolean match = true;
                for (int j = 0; j < m; j++) {
                    if (txt.charAt(i + j) != pat.charAt(j)) {
                        match = false;
                        break;
                    }
                }
                if (match) {
                    System.out.println("Pattern found at index " + i);
                }
            }

            // Compute hash for next window: Remove first char and add next char
            if (i < n - m) {
                t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + m)) % q;
                // Convert negative hash value to positive
                if (t < 0)
                    t += q;
            }
        }
    }

    // Main method
    public static void main(String[] args) {
        String txt = "ABCCDDAEFG";
        String pat = "CDD";
        search(pat, txt);
    }
}
```



Program 24 - Given a text txt [0...n-1] and a pattern pat [0...m-1], prints all occurrences of pat [ ] in txt [ ] by using the Boyer-Moore algorithm. You may assume that n > m.

```
public class SimpleBoyerMoore {

    // Build bad character SHIFT table (your formula)
    static int[] buildBadCharShiftTable(String pattern) {
        int[] badCharShift = new int[256]; // ASCII size
        int m = pattern.length();

        // Default shift for all characters not in pattern
        for (int i = 0; i < 256; i++) {
            badCharShift[i] = m; // Full shift if char not found
        }

        // Set shift = max(1, m - 1 - index)
        for (int i = 0; i < m - 1; i++) {
            char c = pattern.charAt(i);
            badCharShift[c] = Math.max(1, m - 1 - i);
        }

        return badCharShift;
    }

    // Search method using your bad character rule
    static void search(String text, String pattern) {
        int[] badCharShift = buildBadCharShiftTable(pattern);
        int n = text.length();
        int m = pattern.length();
        int shift = 0;

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

            // Compare from end of pattern
            while (j >= 0 && pattern.charAt(j) == text.charAt(shift + j)) {
                j--;
            }

            if (j < 0) {
                System.out.println("Pattern found at index " + shift);
                shift += m; // Full shift after match
            } else {
                char badChar = text.charAt(shift + m - 1);
```

```
            int shiftAmount = badCharShift[badChar];
            System.out.println("Mismatch at index " + (shift + j) + " (char '" + badChar + "'), shifting by " + shiftAmount);
            shift += shiftAmount;
        }
    }
  }

  public static void main(String[] args) {
    String text = "WELCOMETOTEAMMAST";
    String pattern = "TEAMMAST";

    search(text, pattern);
  }
}
```



Inputs:

- text = "WELCOMETOTEAMMAST"
- pattern = "TEAMMAST"

Output:

Prints index(es) where pattern is found.

Loop Flow:

Step 1: Initialize

- n = text.length() = 17
- m = pattern.length() = 8
- Start with shift = 0

While shift <= n - m (i.e., 0 to 9):

For each shift:

1. Compare pattern and text from right to left:

   ```
   while (j >= 0 && pattern.charAt(j) == text.charAt(shift + j))
     j--;
   ```

2. If j < 0: **Pattern matched completely** 🎯
   - Print match index
   - shift += m (jump ahead completely)
3. If mismatch:
   - Get the **bad character** at text[shift + m - 1]
   - Lookup shift value in badCharShift[char]
   - shift += shiftAmount

Example Flow:

First check: shift = 0

- **Compare "TEAMMAST" with "WELCOMET"**
- **Mismatch at 'S' vs 'E'**
- **Bad char = 'E' → shift = badCharShift['E'] = 6**
- **shift = 6**

Second check: shift = 6

- **Compare "TEAMMAST" with "ETOTEAMM"**
- **Mismatch at 'T' vs 'M'**
- **Bad char = 'M' → shift = 3**
- **shift = 9**

Third check: shift = 9

- **Compare "TEAMMAST" with "TEAMMAST" → full match**
- **Print: Pattern found at index 9**
- **shift += 8 → 17 (loop ends)**