July 14th, 2020

# Exercises for Programming Practice 2

Note:
- ✓ Do not create “module-info.java”, when you create a Java Project.
- ✓ Do not set Package name in the window “New Java Class”.
- ✓ Do not use the title of exercise for “contents of the program”. Think about “contents of the program” yourself.

Pattern matching algorithms are used in Exercise 23 to 25.

The deadline for submitting the programs is 18:00 on July 16th, 2020.

The class “Exercise23.java” (although incomplete program) can be obtained from Week 11 of Resource page, “Programming Practice 2” course, manaba+R. The target string is assigned to the variable “str1” in “Exercise23.java”. The pattern of the string contained in “str1” is assigned to the variable “str2” in “Exercise23.java”.

## Exercise 23 (file name “Exercise23.java”)

Create a Java program “Exercise23.java” that implements the brute-force pattern-matching algorithm given in Code Fragment 13.1. Table 1 shows the comparison between Java program and Code Fragment 13.1.

```
1  def find_brute(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)              # introduce convenient notations
4    for i in range(n-m+1):             # try every potential starting index within T
5      k = 0                            # an index into pattern P
6      while k < m and T[i + k] == P[k]:        # kth character of P matches
7        k += 1
8      if k == m:                       # if we reached the end of pattern,
9        return i                       # substring T[i:i+m] matches P
10   return -1                          # failed to find a match starting with any i
```

**Code Fragment 13.1:** An implementation of brute-force pattern-matching algorithm.

Table 1 the comparison between Java program and Code Fragment 13.1

| Java program | Code Fragment 13.1 |
|---|---|
| String[] text | T |
| String[] pattern | P |

The outline of “Exercise23.java” is as follows:

```
/* comments */

public class Exercise23 {
    String str1 = "dabdabcabcba";
    String str2 = "abcb";

    public static void main(String[] args) {
        new Exercise23();
    }

    public Exercise23() {
        String[] text = str1.split("");
        String[] pattern = str2.split("");
        printList("text", text);
        printList("pattern", pattern);

        int result = find_brute(text, pattern);
        System.out.println(result);
    }

    public int find_brute(String[] text, String[] pattern) {
        int n = text.length;
        int m = pattern.length;
        int k;

        // Complete this part

    }

    public void printList(String str, String[] list) {
        System.out.print(str + ": ");
        for(int i = 0; i < list.length; i++) {
            System.out.print(list[i] + " ");
        }
        System.out.println();
    }
}
```

Put the following print statement immediately after the header statement of “while”.

```
System.out.println("k is " + k + ", text is " + text[i + k] + ", pattern is " + pattern[k]);
```

*Expected Output:*

```
text: d a b d a b c a b c b a
pattern: a b c b
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 2, text is c, pattern is c
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 2, text is c, pattern is c
k is 3, text is b, pattern is b
7
```

Exercise 24 (file name "Exercise24.java")

Create a Java program "Exercise24.java" that implements the Boyer-Moore algorithm given in Code Fragment 13.2. Use "printList" method of "Exercise23.java". Table 2 shows the comparison between Java program and Code Fragment 13.2.

```
1  def find_boyer_moore(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)                    # introduce convenient notations
4    if m == 0: return 0                      # trivial search for empty string
5    last = { }                               # build 'last' dictionary
6    for k in range(m):
7      last[ P[k] ] = k                       # later occurrence overwrites
8    # align end of pattern at index m-1 of text
9    i = m−1                                  # an index into T
10   k = m−1                                  # an index into P
11   while i < n:
12     if T[i] == P[k]:                       # a matching character
13       if k == 0:
14         return i                           # pattern begins at index i of text
15       else:
16         i −= 1                             # examine previous character
17         k −= 1                             # of both T and P
18     else:
19       j = last.get(T[i], −1)               # last(T[i]) is -1 if not found
20       i += m − min(k, j + 1)               # case analysis for jump step
21       k = m − 1                            # restart at end of pattern
22   return −1
```

**Code Fragment 13.2:** An implementation of the Boyer-Moore algorithm.

Table 2 the comparison between Java program and Code Fragment 13.2

| Java program | Code Fragment 13.2 |
|---|---|
| String[] text | T |
| String[] pattern | P |
| if(last.get(text[i]) == null) {<br>  j = -1;<br>} else {<br>  j = last.get(text[i]);<br>} | j = last.get(T[i], −1)<br># last(T[i]) is -1 if not found |

Put the following print statement immediately after the header statement of “while”.

```
System.out.println("k is " + k + ", text is " + text[i + k] + ", pattern is " + pattern[k]);
```

The outline of “Exercise24.java” is as follows:

```
/* comments */

import java.util.HashMap;

public class Exercise24 {
    String str1 = "dabdabcabcba";
    String str2 = "abcb";

    public static void main(String[] args) {
        new Exercise24();
    }

    public Exercise24() {
        String[] text = str1.split("");
        String[] pattern = str2.split("");
        printList("text", text);
        printList("pattern", pattern);

        int result = find_boyer_moore(text, pattern);
        System.out.println(result);
    }

    public int find_boyer_moore(String[] text, String[] pattern) {
        int n = text.length;
        int m = pattern.length;
        HashMap<String, Integer> last = new HashMap<String, Integer>();
        int j;

        // Complete this part

    }

    /* Use printList */

}
```

*Expected Output:*

```
text: d a b d a b c a b c b a
pattern: a b c b
k is 3, text is d, pattern is b
k is 3, text is a, pattern is b
k is 3, text is b, pattern is b
k is 2, text is c, pattern is c
k is 1, text is b, pattern is b
k is 0, text is a, pattern is a
7
```

Exercise 25 (file name “Exercise25.java”)

Create a Java program “Exercise25.java” that implements the Knuth-Morris-Pratt (KMP) algorithm given in Code Fragment 13.3 as method “find_kmp”, and implements the Knuth-Morris-Pratt (KMP) failure function given in Code Fragment 13.4 as method “compute_kmp_fail”. Use “printList” method of “Exercise23.java”.

```
1  def find_kmp(T, P):
2    """Return the lowest index of T at which substring P begins (or else -1)."""
3    n, m = len(T), len(P)                    # introduce convenient notations
4    if m == 0: return 0                      # trivial search for empty string
5    fail = compute_kmp_fail(P)               # rely on utility to precompute
6    j = 0                                    # index into text
7    k = 0                                    # index into pattern
8    while j < n:
9      if T[j] == P[k]:                       # P[0:1+k] matched thus far
10       if k == m - 1:                       # match is complete
11         return j - m + 1
12       j += 1                               # try to extend match
13       k += 1
14     elif k > 0:
15       k = fail[k-1]                        # reuse suffix of P[0:k]
16     else:
17       j += 1
18   return -1                                # reached end without match
```

**Code Fragment 13.3:** An implementation of the KMP pattern-matching algorithm. The compute_kmp_fail utility function is given in Code Fragment 13.4.

```
1  def compute_kmp_fail(P):
2    """Utility that computes and returns KMP 'fail' list."""
3    m = len(P)
4    fail = [0] * m                  # by default, presume overlap of 0 everywhere
5    j = 1
6    k = 0
7    while j < m:                    # compute f(j) during this pass, if nonzero
8      if P[j] == P[k]:              # k + 1 characters match thus far
9        fail[j] = k + 1
10       j += 1
11       k += 1
12     elif k > 0:                   # k follows a matching prefix
13       k = fail[k-1]
14     else:                         # no match found starting at j
15       j += 1
16   return fail
```

**Code Fragment 13.4:** An implementation of the compute_kmp_fail utility in support of the KMP pattern-matching algorithm. Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

Put the following print statement immediately after the header statement of “while” in “find_kmp” method.

```
System.out.println("k is " + k + ", text is " + text[i + k] + ", pattern is " + pattern[k]);
```

The outline of “Exercise25.java” is as follows:

```
/* comments */

public class Exercise25 {
    String str1 = "dabdabcabcba";
    String str2 = "abcb";

    public static void main(String[] args) {
        new Exercise25();
    }

    public Exercise25() {
        String[] text = str1.split("");
        String[] pattern = str2.split("");
        printList("text", text);
        printList("pattern", pattern);

        int result = find_kmp(text, pattern);
        System.out.println(result);
    }

    public int find_kmp(String[] text, String[] pattern) {
        int n = text.length;
        int m = pattern.length;

        /* Compete this part */

    }

    public int[] compute_kmp_fail(String[] pattern) {
        int m = pattern.length;
        int[] fail = new int[m];
        int j = 1;
        int k = 0;

        // Complete this part

    }

    /* Use printList */
}
```

*Expected Output:*

```
text: d a b d a b c a b c b a
pattern: a b c b
k is 0, text is d, pattern is a
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 2, text is d, pattern is c
k is 0, text is d, pattern is a
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 2, text is c, pattern is c
k is 3, text is a, pattern is b
k is 0, text is a, pattern is a
k is 1, text is b, pattern is b
k is 2, text is c, pattern is c
k is 3, text is b, pattern is b
7
```