__/__/____

Pattern Matching by Kruth-Morris-Pratt

Given a string "string" and another string "pat", not longer than string, check whether pat is there in string.

1. It can be done by a library function.

2. It can be done interiteively where the computing time can go upto $O(nm)$ where n and m are lengths of pat and string respectively.

3. An improvement:

   a) Quit the search when length of pat is greater than the number of remaining characters in the string.

   b) Comparing the first and last characters of pat with the corresponding characters in the search space, that is, string.

```
int find(char *string, char *pat){
    int i,j, start = 0;
    int lasts = strlen(string)-1; int lastp = strlen(pat)-1;
    int endmatch = lastp;
    for(i=0; endmatch<=lasts; endmatch++, start++){
        if(string[endmatch]==pat[lastp])
            for(j=0, i=start; j<lastp && string[i]==pat[j]; i++,j++);
        if(j==lastp) return start;
    }
    return -1;
}
```

__/__/____

Try run the algorithm on pat = aab and
string = ababbaabaa..

4. The Knuth, Morris and Pratt (KMP) algorithm:

Using their example, let pat = abcabcacab and
let $s = s_0 s_1 \cdots s_{n-1}$ be the string and we are to determine
whether there is a match beginning at s_i. If $s_i \neq a$ then we may
proceed by comparing s_{i+1} and a. Similarly, if $s_i = a$ and
$s_{i+1} \neq b$ then we may proceed by comparing s_{i+1} and a.
If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation

s = ' - a b ? ?? . . . ?'
pat = 'a b c a b c a c a b'

The first ? in s represents s_{i+2} and $s_{i+2} \neq c$. At this point
we know that we may continue the search for a match by
comparing the first character in pat with s_{i+2}. There is no
need to compare this character of pat with s_{i+1}, as we already
know that s_{i+1} is the same as the second character of pat, b
and so $s_{i+1} \neq a$.
Let us try this again assuming a match of the first four
characters in pat followed by a non-match, that is,
$s_{i+4} \neq b$.

s = ' - a b c a ? ? . . . ?'
pat = 'a b c a b c a c a b'

We observe that the search for a match can proceed by comparing
s_{i+4} and the second character in pat, b. This is the first place a

__/__/____

partial match can occur by sliding the pattern pat towards the right. Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in s we can determine where in the pattern to continue the search for a match without moving backwards in s. This leads to the def'n of a failure function for a pattern:

If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its failure function, f, is defined as:

$$f(i) = \begin{cases} \text{largest } j < i \text{ such that } p_0 p_1 \cdots p_j = p_{i-j} p_{i-j+1} \cdots p_i \\ \qquad \text{if such an } j \geq 0 \text{ exists} \\ -1 \qquad \text{otherwise} \end{cases}$$

For the example pattern, pat = abcabcacab, we have

i	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

So, we arrive at the following rule for pattern matching:

If a partial match is found such that $s_{i-j} \cdots s_{i-1} = p_0 p_1 \cdots p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by comparing s_i and $p_{f(j-1)+1}$ if $j \neq 0$. If $j = 0$, then we may continue by comparing s_{i+1} and p_0.

The program follows:

3/6

__/__/____

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];
int match(char *string, char *pat){
    int i=0, j=0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while (i<lens && j<lenp){
        if (string[i]==pat[j]){i++; j++;};
        else if (j==0) i++;
            else j = failure[j-1]+1;
    };
    return((j==lenp)? (i-lenp) : -1);
}
```

We explain it below:

__/__/____

There is no pointer to the start of the pattern in the string; we instead, are using the return statement. This statement checks to see whether or not we found the pattern. If we didn't find the pattern, the pattern index j is not equal to the length of the pattern and we return −1. If we found the pattern, then the starting position is i − length of the pattern.

Analysis:

The while loop is iterated until the end of either the string or the pattern is reached. Since i is never decreased, the lines that increase i cannot be executed more than m = strlen(string) times. The resetting of j to failure[j−1]+1 decreases the value of j. So, this cannot be done more times than j is incremented by the statement j++ as otherwise, j falls off the pattern. Each time the statement j++ is executed, i is also incremented. So, j cannot be incremented more than m times. Therefore, no statement of the program is executed more than m times. Hence, match is $O(m) = O(\text{strlen(string)})$.

We can recode $f(i)$ as

$$f(i) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

See that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$.

__/__/____

This leads to the failure f^n computation as:

```
void fail (char *pat) {
    int n = strlen(pat);
    failure[0] = -1;
    for (j=1; j<n; j++) {
        i = failure[j-1];
        while ((pat[j] != pat[i+1]) && (i>=0))
            i = failure[i];
        if (pat[j] == pat[i+1]) failure[j] = i+1;
        else failure[j] = -1;
    }
}
```

Show that the computing time of fail is
$O(n) = O(\text{strlen}(pat))$.

So, when the failure f^n is not known in advance, the time to first compute this f^n and then perform a pattern match is
$O(\text{strlen}(pat)) + O(\text{strlen}(string))$.

6/6