British Informatics Olympiad Final

11–13 April 1997

Sponsored by Data Connection

String matching

Give a long string we may be interested in whether a smaller string (henceforth called a pattern) can be found within it. In this question we will look at various algorithms for solving this problem.

Throughout the question we will treat both the string and pattern as arrays of characters, called S and P respectively. S[1] and P[1] will be treated as the first characters (*not* S[0] and P[0]). The length of the long string will be n, and the pattern m.

Note : You are only required to write code when you are explicitly asked to do so. You are allowed to write code for any question, if you feel it helps with your explanation.

1. Naive algorithms

Suppose the string and the pattern are of equal length. The following pseudo-code checks to see if they are equal; `match` will end up *true* if they agree, and *false* if they differ :

```
match := TRUE;
for i := 1 to m do
  if (P[i] <> S[i]) then match := FALSE;
```

This will always take m steps to complete.

Question 1.1

(a) Is it possible to write a matching program which takes fewer than m steps when the two strings are the same? Justify your answer.

(b) How about when the two strings are different? Justify your answer.

Question 1.2

What we really need is a procedure/function that checks whether the pattern matches the string between S[x] and S[$x+m-1$] inclusive. Rewrite the above code as a procedure/function called `equal` that does this, taking x as a parameter, and returning either *true* or *false*. [You can assume a boolean type exists.]

We can try and find the pattern in the string by looking for it at every different position in the string. The following pseudo-code implements this naive method, setting *found* to point to the location of the pattern if it is located, or `-1` if it is not.

```
found := -1;
j := 1;
repeat
  if (equal(j) = TRUE) then found := j;
  j := j+1;
until (j > n-m+1) or (found <> -1);
```

Question 1.3

(a) In terms of m and n, how many comparisons are made in the `equal` procedure, in the worst possible case?

(b) How many are made in the best case?

Question 1.4

Suppose we have the special case where all the characters in the pattern are different. Explain how you could solve the problem using at most only n comparisons.

2. Rabin-Karp algorithm

It is possible to represent strings by numbers. If we suppose strings can just contain upper case letters we can convert them as follows :

1. Covert each letter into a number representing its place in the alphabet. Since this is a computing question, we will count from 0. For example '`HELLO`' will become '`7 4 11 11 14`'.

2. We treat this as a base 26 number, and calculate its value. For our example we calculate $7 \times 26^4 + 4 \times 26^3 + 11 \times 26^2 + 11 \times 26 + 14$. This number (3,276,872) represents the string.

No other string of the same length will have the same number. We could restore the string by doing integer divisions (`DIV` in Pascal, and / in C) and looking at remainders (`MOD` in Pascal, and % in C).

Suppose we calculate the number for the pattern, and the number for each substring of length m in the large string. We can see if the pattern is in the string by seeing if its number matches one of the numbers we have generated for the substrings. Once the numbers have been calculated we only have make at most $n - m + 1$ comparisons.

If the array V is used to hold these values (V[i] being the number for the substring from S[i] to S[i+m-1]), and PN the number for the pattern, we can use the following pseudo-code.

```
found := -1;
for i := 1 to n-m+1
  if (V[i] = PN) then found := i;
```

Question 2.1

The value of V[i] will be S[i]$\times 26^{m-1}$+S[i+1]$\times 26^{m-2}$ + $\cdots$ +S[i+m-1]. This is potentially a lot of work to do for each V[i]. Show how V[i] can be used to calculate V[i+1].

Question 2.2

Using the relation you have written for question 2.1, write code to compute the array V.

Questino 2.3

(a) As it stands this method is impracticable. Why?
(b) How could you go about making this method usable?
(c) Does your new 'fixed' method still take at most $n - m + 1$ comparisons?

3. Knuth-Morris-Pratt algorithm

When using naive pattern matching, if we fail to match the pattern with the string at a given position, we start looking for the pattern at the next position. We do not use any other information, such as how much of the string we were able to match. If we take into consideration the amount of pattern we matched, and the pattern iteself, we can often prevent checks by deducing the results.

For example, suppose we are looking for the pattern `abababc` in the string `ababacbabababc`. When we are comparing the pattern to the first part of the string, the first five characters are matches, but the sixth one fails to match. We can deduce :

- There is no point in comparing the pattern to the string from the second position. Since we have just managed to match five characters we know the second character in the string is equal to the second character in the pattern ('`b`'), and from our knowledge of the pattern we know this is different to the first character in the pattern ('`a`').

- If we try to match the pattern to the string from the third position, we know the first three characters will be matches. This is because we know P[3..5] matches P[1..3].

Given a pattern we need to compute a look-up table that provides us with information about its structure. If we know the first q characters of the pattern were a match, but the next one failed to match, we need to know which position in the string we should now try and match to, and how many characters we already know will match. We can combine these two facts by just calculating where the last character we matched in the pattern will be in the next string — a zero meaning 'just to the left of it'.

For example :
The pattern `abababc` has the table containing the entries `0 0 1 2 3 4 0`.

- If we only matched the first character, we should start looking at the string one position along, with no known matches.
- If we matched the first two characters we should start looking at the string two positions along, with no known matches.
- If we matched the first three characters we should start looking at the string two positions along, knowing the first character is a match.

a	b	a	b	a	b	c
		a				

- If we matched the first four characters we should start looking at the string two positions along, knowing the first two characters are matches.

a	b	a	b	a	b	c
		a	b			

etc...

Question 3.1
What is the table for the pattern `aababaabc`?

Question 3.2
Suppose you have generated the necessary look-up table, and it is in the array L. Write code that uses this table to search for the pattern in the given string.

Question 3.3
Write code to generate the look-up table for a given pattern.