KMP

Problem: given a string **text** find the number of occurrences (substrings) of the string **pat**.

Let's investigate the naive brute-force solution for the problem.

```
int brute_force(const string &text, const string &pat) {
    int n = text.size();
    int m = pat.size();
    int occ = 0; // number ocurrences of pat in text
    if(m > n) {
        return occ;
    }

    int i = 0;
    while(i <= (n - m)) {
        int j = 0;
        for(j = 0; j < m; ++j) {
            if(text[i + j] != pat[j]) {
                break;
            }
        }
        if(j == m) {
            ++occ;
        }
    }
    return occ;
}
```

Complexity = O(nm)

Here's the worst-case input for the bruteforce:

text = aaa...a
pat = aaa...a, **|pat| < |text|**

KMP
The key idea to the KMP solution is that we don't compare the same thing twice. For example If **text =** aaaaaa and **pat =** aaa, the naive solution knows that **text[1, 3] = pat[1, 3]** and he also knows that **text[2, 3] = pat[1, 2]** but he doesn't take advantage of that, and compare the same thing.

Considere

Tentativa de match:
text = "abcab**d**abc"
pat = "abcab**c**"

Depois disso nó sabemos que um parte do início de **pat** aparece no final de **text**, anterior ao mismatch.

text = "abc<u>ab</u>**d**abc"
   pat = "<u>ab</u>cab**c**"
Então nós podemos pular essa parte que já sabe que irá dar match.

Observe que é sempre garantido que podemos avançar sempre o tamanho da borda da substring de **pat** terminando em **j - 1** onde **pat[j]** é o caractere que deu mismatch com **text[i + j]**.
Isso é verdade por conta da seguinte observação

Prefixo começando em i em text = (a….b) x
Prefixo de pat = (a….b) y
Onde x = text[i + j], y = pat[j], x != y
Isso quer dizer que as duas substrings dentro do parêntesis são iguais, e o objetivo agora é "pular o maior prefixo (a …. b) (em pat) que também é sufixo (a … b) (em text)", porque sabemos que eles vã dar match, e como estas substrings são iguais, isto é justamente a definição de uma borda.

Portanto precisamos saber de **border[j - 1]** = tamanho da borda do prefixo de **pat** até **(j - 1)**.

Calculando as bordas de uma string eficientemente

Lembrando que border(s) é a maior borda da string **s** que seja menor que o tamanho de **s** (uma borda não-trivial)

Considere que estamos procurando uma borda para o sufixo de pat de tamanho **j**
Considere também que **t = j - 1** (borda do prefixo de **pat** de tamanho **(j - 1)**)
Nossa intenção é aumentar essa borda, aumentando o tamanho de **t** em 1.
Isso só vai ser possível se

pat[t] = pat[j - 1]

Ex:
Pat = "abxyzab" = "x"
T = border("abxyzab") = |"ab"| = 2
O próximo caractere depois de "ab" é 'x', que é igual ao caracterece que queremos colocar na ultima borda.

Mas se não conseguirmos aumentar o tamanho da borda atual, então vamos tentar anexar o 'x' em outras bordas (na maior que der) **pat** para calcular border**[j]**...

Lemma: Uma borda de uma borda é uma borda da string original
Prova:

Se **s'** é uma borda de **s**, então

s = s' …. S'

Se **s''** é uma borda de **s'** então
s ' = s'' … s''
Substituindo **s''** em **s**
s = s' …. S'
s = (s'' … s'') …. (s'' … s'')
Portanto **s''** é borda de **s.**

Então se não conseguirmos aumentar o tamanho da borda atual, iremos tentar aumentar uma borda de uma borda.

Referencias

https://github.com/edsomjr/TEP/blob/master/Strings/text/Algoritmos_de_Busca.md
http://algorithmsforcontests.blogspot.com.br/2012/08/borders-of-string.html