Pattern matching - Introduction

0. Defining the pattern matching problem

```
Input: T[0..n-1] (a text, with n symbols)
       P[0..m-1] (a pattern, with m symbols)
Output: i, such that T[i..i+m-1] == P
        or -1, if such i does not exist
```

1. Naive Search algorithm

i = the starting position of the current possible match in the text T
k = the position of the currently compared character in the pattern (and also = the number of characters that have been matched)

```
int naive(char *T, int n, char *P, int m)
{
  for (int i = 0; i < n - m + 1; ++i) {
    bool found = true;
    for (int k = 0; k < m; ++k) {
      if (P[k] != T[i + k]) {
        found = false;
        break;
      }
    }
    if (found) {
      return i;
    }
  }
  return -1;
}
```

Worst case complexity: O((n+m)^2)

Pattern matching - KMP part 1

2. KMP (Knuth-Morris-Pratt) algorithm

KMP: optimizes the Naive Search Algorithm.

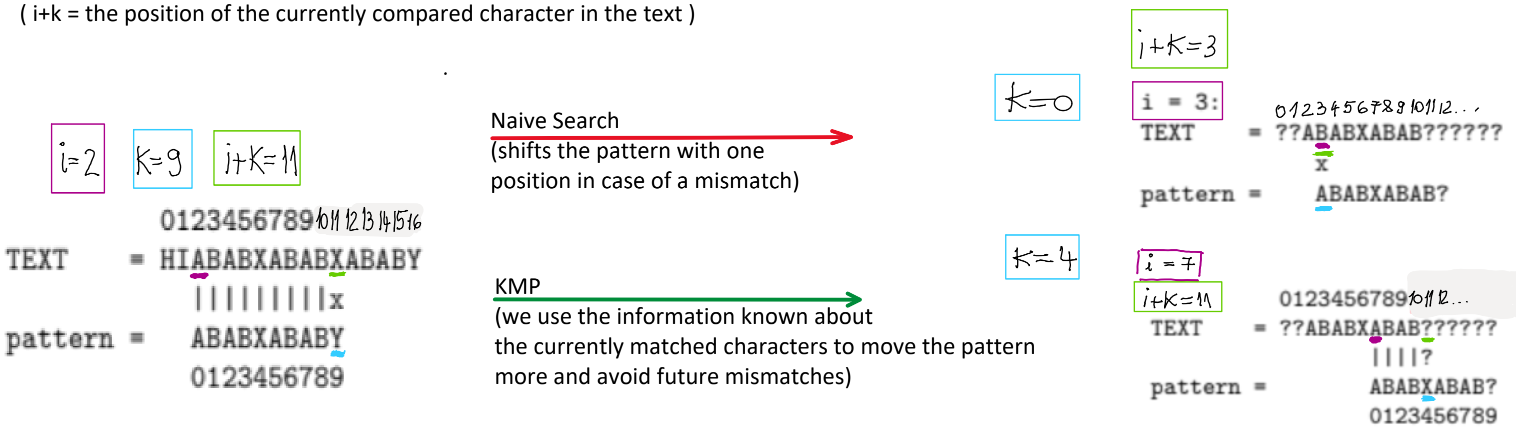
Worst case complexity: O(n+m)

2.1 General idea: use the information of the current matching of the first characters in the pattern with the text to:

1. move the pattern further (usually with more than one position) in case of a mismatch (= Update i smartly)
2 start the comparison by skipping the characters that we already know that are equal (= Update k smartly)

i = the starting position of the current possible match in the text T
k = the position of the currently compared character in the pattern (and also = the number of characters that have been matched)

(i+k = the position of the currently compared character in the text)



2.2 How to update indices k and i in KMP

We will call the portion of the currently matched string 'overlap' (in our case: ABABXABAB).
Only using the information given by the overlap, we can find the smallest shift that we can make to ensure that there is a chance of match.

```
i = 3:
TEXT    = ??ABABXABAB??????
            x
pattern =   ABABXABAB?

i = 4:
TEXT    = ??ABABXABAB??????
              ||x
pattern =     ABABXABAB?

i = 5:
TEXT    = ??ABABXABAB??????
             x
pattern =    ABABXABAB?

i = 6:
TEXT    = ??ABABXABAB??????
              x
pattern =     ABABXABAB?
```

i=7

suffix of the overlap (seen as part of the text)

```
TEXT    = ??ABABXABAB??????
                 ||||
pattern =        ABABXABAB?
```

prefix of the overlap (seen as part of the pattern)

Border of a string = a string of characters that is both a suffix and a prefix of the string

A string can have multiple borders (for e.g. the overlap ABABXABAB has borders AB and ABAB)

condition necessary to move the pattern

In case of a mismatch, we can shift the pattern (without skipping any matchings) such that we have a match for the largest border of the overlap (different from the entire string).

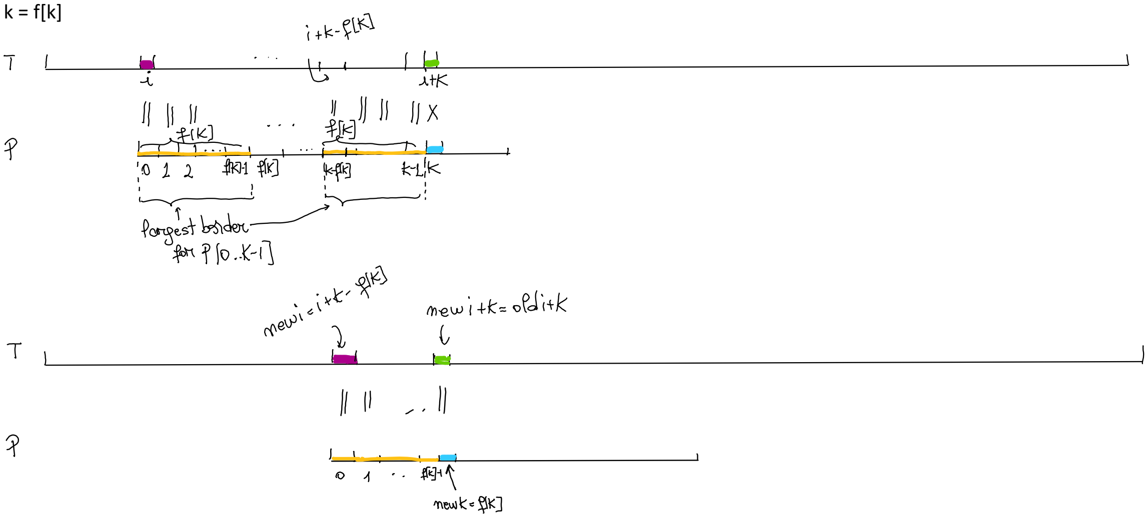
a smaller border could make us skip some matches

ABABXABAB
       ABABXABAB

Let f[k] = the length of the biggest border of the prefix of length k of the pattern (P[0...k-1])
(f = prefix/failure function)

In case of a mismatch (T[i+k]!=P[k]), we make the updates:

i = i + k - f[k]
k = f[k]



2.3 Code for KMP

```
int i = 0;
int k = 0;
while (i <= n - m && k < m) {
  if (T[i + k] == P[k]) {
    k++;
  } else if (k == 0) {
    i = i + 1;
  } else {
    i = i + k - f[k];
    k = f[k];
  }
}
if (k == m) {
  return i;
} else {
  return -1;
}
```

// As long as there are still possible matches (i <= n - m) and we haven't found a match (k < m)

// If the current ch. in text & pattern are equal, go forward 1 position (in pattern & in text)

// otherwise, if the first ch is not equal, move the pattern forward 1 position

// in the other cases of mismatch, update smartly i and k indices

// if we have a match of the entire pattern, return the start position of the match

// otherwise, no match was found

2.4 Computing the failure function f

f[k] = the length of the biggest border of P[0…k-1]

The failure function f can be computed using previous values as follows:

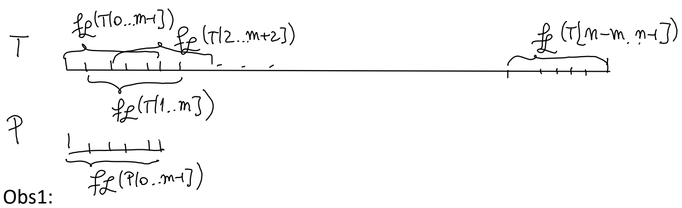
```
f[0]  =  -1;

for(i = 1; i < m; i++){
      k = f[i-1];
     while(k >= 0 && P[k] != P[i-1]){
           k = f[k];
    }
    if (k== -1){
      f[i] = 0;
    }  else {
      f[i] = k + 1;
   }

}
```

3. Rabin-Karp algorithm

Worst case complexity: O((n+m)^2)
Average case complexity: almost linear

3.1. General idea

We compute a hash for the pattern and hashes for all substrings (possible matches) from the text of the same size as the pattern.
The hash function fq receives as input a string and outputs a hash (a number in range {0,1,..,q-1}).
If the pattern has the same hash as a substring in the text, it is likely that they are equal (but we have to check this explicitly to make sure).

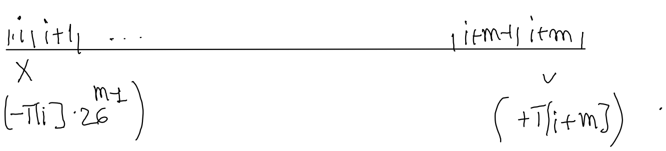


Obs1:

To calculate the hash function for a string S:
- we first need to assign each character S[i] in the string a number (for e.g., we can use consecutive numbers starting from 0 or the ASCII code)
- we encode the number formed by the string in a certain base (for e.g., in base 26)
- we take modulo q out of the finally resulted number (to ensure that we avoid operations on very large numbers and attain a good complexity)

An example: $fq(BABX) = (1 \times 26^3 + 0 \times 26^2 + 1 \times 26^1 + 23 \times 26^0) \% q = 17625 \% q$ (convention A → 0, B → 1, C → 2, ..., X → 23)

Obs2:
We can calculate a hash of a text substring T[i+1 ..i+m] using the hash of the previous text substring T[i..i+m-1] in O(1).




```
// Algoritmul Rabin-Karp
q = 23; // exemplu, poate fi orice număr prim

fq(S)
{
  result = 0;
  for i = 0,m-1:
    result = (result * 26 + S[i]) % q
  return result;
}
```
} the hash function

```
x = fq(P[0..m-1]) // hash value for pattern
y = fq(T[0..m-1]) // hash value for the first text substring
for i = 0,n-m:
  if y == x
    if P[0..m-1] == T[i..i+m-1] // (*) we need to check explicitly if the substrings are truly identical
      return i
  y = ((y + q - (T[i] * 26^(m-1)) % q) * 26 + T[i+m]) % q   // rule to calculate fq(T[i+1..i+m]) using fq(T[i..i+m-1])
return -1
```

4. References & other resources

2023 lecture for Pattern Matching (https://sites.google.com/view/fii-pa/2023/lectures) - for more examples & explanations regarding the algorithms and their time complexity
TrulyUnderstandingAlgorithms YouTube channel (https://www.youtube.com/@TrulyUnderstandingAlgorithms/videos) - additional explanations for KMP