KMP
Knuth Morris Pratt Algorithm

Mostafa Saad Ibrahim

Teaching Assistant @ Cairo University

Software Engineer @ NTPSoftware

Pattern Matching

- Problem: Given a string S, pattern P, find all P in S

- S = abababcabaddd, P = aba
  - 2 matches: abababcabaddd

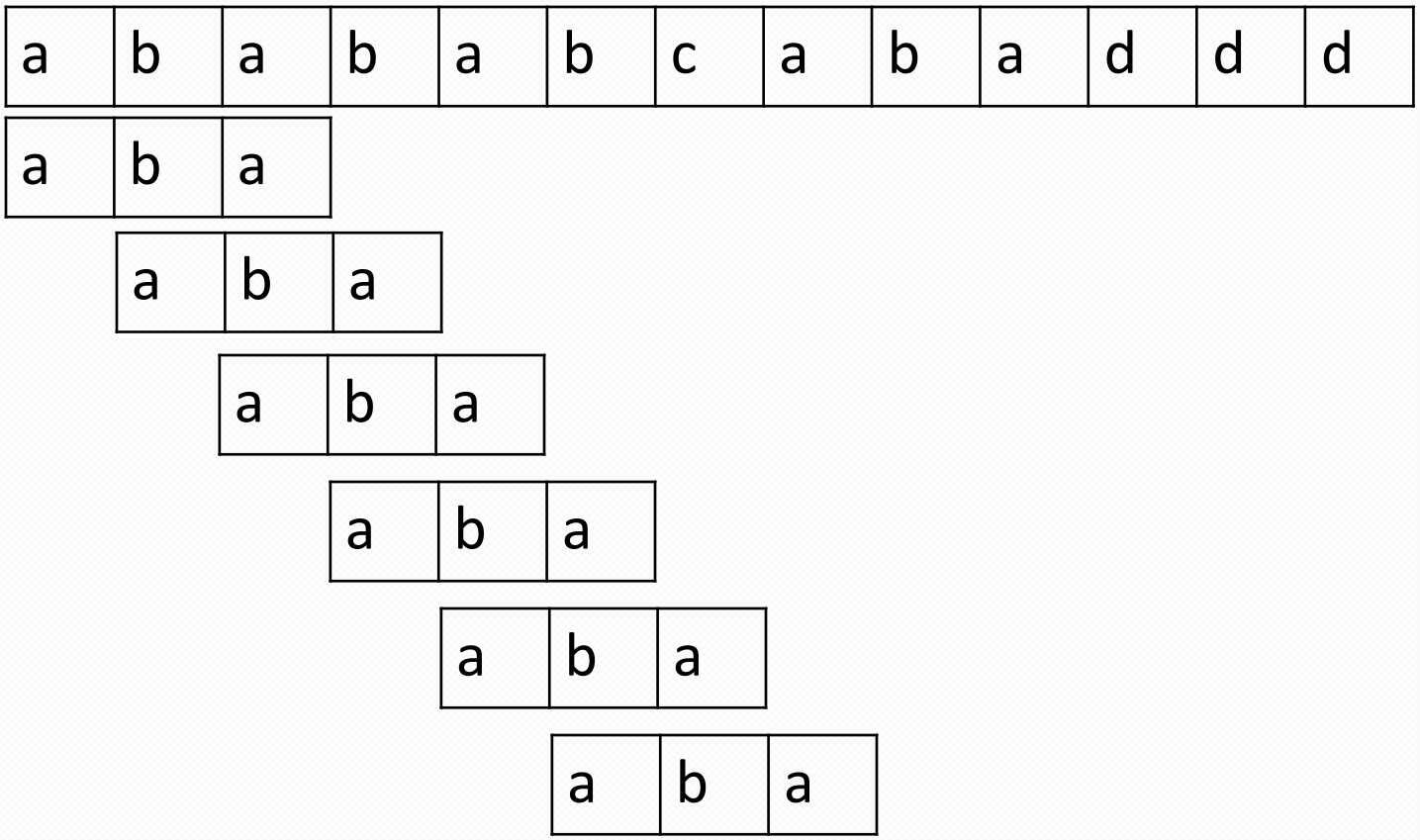
Brute Force

- Let: n = length(S) and m = length(P)
- Try every location in S, and check of it matches P

- For i = 0 to n-m
  - Let t = S[i, …i+m-1] => Substring from S
  - If equal(P, t)
    - Display, we found a match

- Order (nm)

Brute Force Example



and so on

Hmm

- Simple
- Big order
- Much redundancy
  - E.g. if m = 5, We compare positions: 3 4 5 6 7
  - and then compare: 4 5 6 7 8

- A clever solution should avoid redundancy
  - KMP does so

Helpful terminology

- Prefix of S
  - Any string that start from first character
- Proper Prefix of S
  - Any prefix of S, except S itself
  - Then a string of 1 letter, has **no** proper prefixes
- Suffix of S
  - Any string that ends at last character
- Proper Suffix of S
  - Any suffix of S, except S itself

Example: S = abcdef

- Prefix of S: n prefixes
  - PL = {a, ab, abc, abcd, abcde, abcdef}
- **Proper Prefix** of S: n-1 prefixes
  - PPL = {a, ab, abc, abcd, abcde}
- **Suffix** of S: n suffixes
  - SL = {abcdef, bcdef, cdef, def, ef, f}
- Proper Suffix of S : n-1 suffixes
  - PSL = {bcdef, cdef, def, ef, f}
- I did not consider the empty string case for some reasons

Prefix function

- Given String P of length m, define an array F[m]:
  - Let t = P[0..i]
  - F[i] = length of longest proper prefix of t that is suffix of t
- Other popular name: Failure Function
- Some references exclude ith position in calculation, hence F has different numbers

Prefix function: ababcaba

- i = 0, F[0] = 0 => by definition
- i = 1 => t = ab => F[1] = 0
  - PPL = {a}
  - SL = {ab, b}
- i = 2 => t = aba => F[2] = 1
  - PPL = {**a**, ab}
  - SL = {aba, ba, **a**}
- i = 3 => t = abab => F[3] = 2
  - PPL = {a, **ab**, aba}
  - SL = {abab, bab, **ab**, b}

Prefix function: ababcaba

- i = 4 => t = ababc => F[4] = 0
  - PPL = {a, ab, aba, abab}
  - SL = {ababc, babc, abc, bc, c}
- i = 5 => t = ababca => F[5] = 1
  - PPL = {**a**, ab, aba, abab, ababc}
  - SL = {ababca, babca, abca, bca, ca, **a**}
- i = 6 => t = ababcab => F[6] = 2
  - PPL = {a, **ab**, aba, abab, ababc, ababca}
  - SL = {ababcab, babcab, abcab, bcab, cab, **ab**, b}

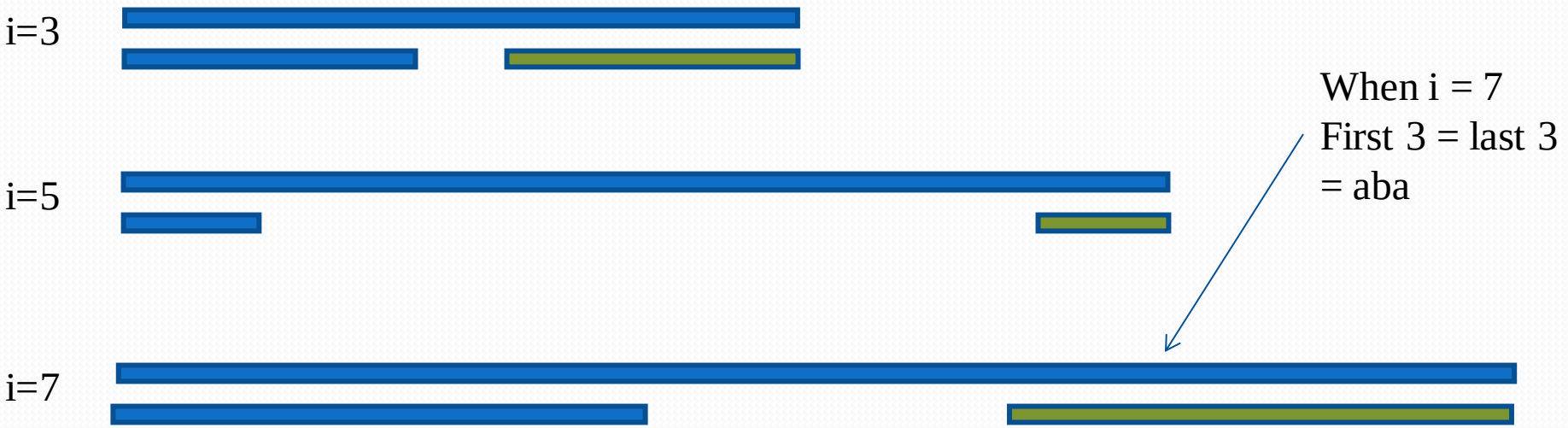
Prefix function: ababcaba

- i = 7 => t = ababcaba => F[7] = 3
  - PPL = {a, ab, **aba**, abab, ababc, ababca, ababcab}
  - SL = {ababcaba, babcaba, abcaba, bcaba, caba, **aba**, ba, a}

- Then F[] = {0, 0, 1, 2, 0, 1, 2, 3}

Prefix function: ababcaba

a	b	a	b	c	a	b	a

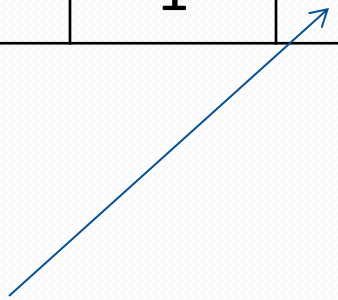


0	0	1	2	0	1	2	3

Prefix function: aaaaaaaa

a	a	a	a	a	a	a	a

0	1	2	3	4	5	6	7



Remember:
longest **proper** prefix = suffix
NOT
longest prefix = suffix

Prefix function

- Later we will now why named also Failure function.
- For now, assume we calculate this table using Brute Force

- Later, an O(m) implementation

KMP Observation?

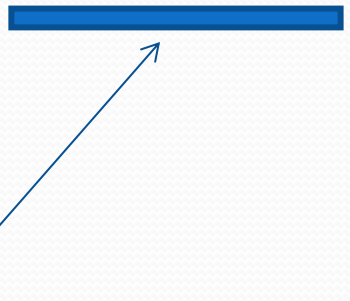
- Let's back to pattern matching
- Say we have pattern P = **ababcaba** and we want to search inside string **abababcabaddd**
- BF will try matching from position 0, and will match 4 characters then fail with c letter.
- Now, it will try to match from 1, however, we already know the 3 matched characters from 1(bab).
- Could we make use of that?

KMP Observation?

a	b	a	b	a	b	c	a	b	a	d	d	d

i = 0

i = 1



We already know that we match these 3 letters…
How to make use of that?

KMP Target

- After matching t letters and failing, instead of trying **next** position, could we jump to next **promising** position?
- Next vs Promising?
- In other words, the t letters of pattern P, are prefix of P. Could we do minimum left shift of k letters to have next matching?

KMP Question?

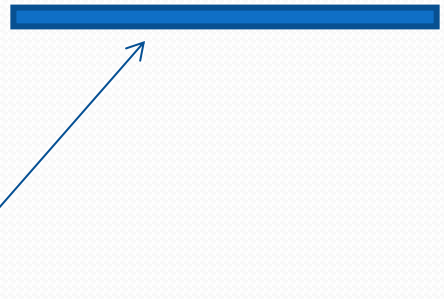
- Say we have pattern P = **aaaaabc** and we want to search inside string **a<u>aaaaab</u>cxy**
- Say starting from 0, we matched 5 characters, then failed
- Let's start to try from 1:
- **Q**: When could we match the 5-1 = 4 characters?
- **A**: IFF these 4 characters (**suffix** of matched so far) = to first 4 characters (**prefix** of matched so far)
  - Matched so far: aaaaa: suffix = aaaa and prefix = aaaa
- What if it matches less than 4? sure position 1 is useless

Match **aaaaabc**

a	a	a	a	a	a	b	c	x	y

i = 0

i = 1



When could these letters do match?

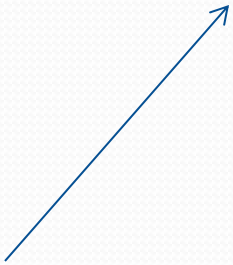
To match these 4 letters in the pattern
Prefix of 4 letters = suffix of 4 letters

Match ababcaba

a	b	a	b	a	b	c	a	b	a	d	d	d

i = 0

i = 1



If we can't match from 1,
Could we match from 2?

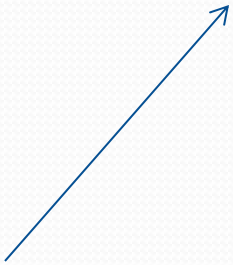
To match these 3 letters in the pattern
Prefix of 3 letters = suffix of 3 letters
aba != bab

Match ababcaba

a	b	a	b	a	b	c	a	b	a	d	d	d

i = 0

i = 2



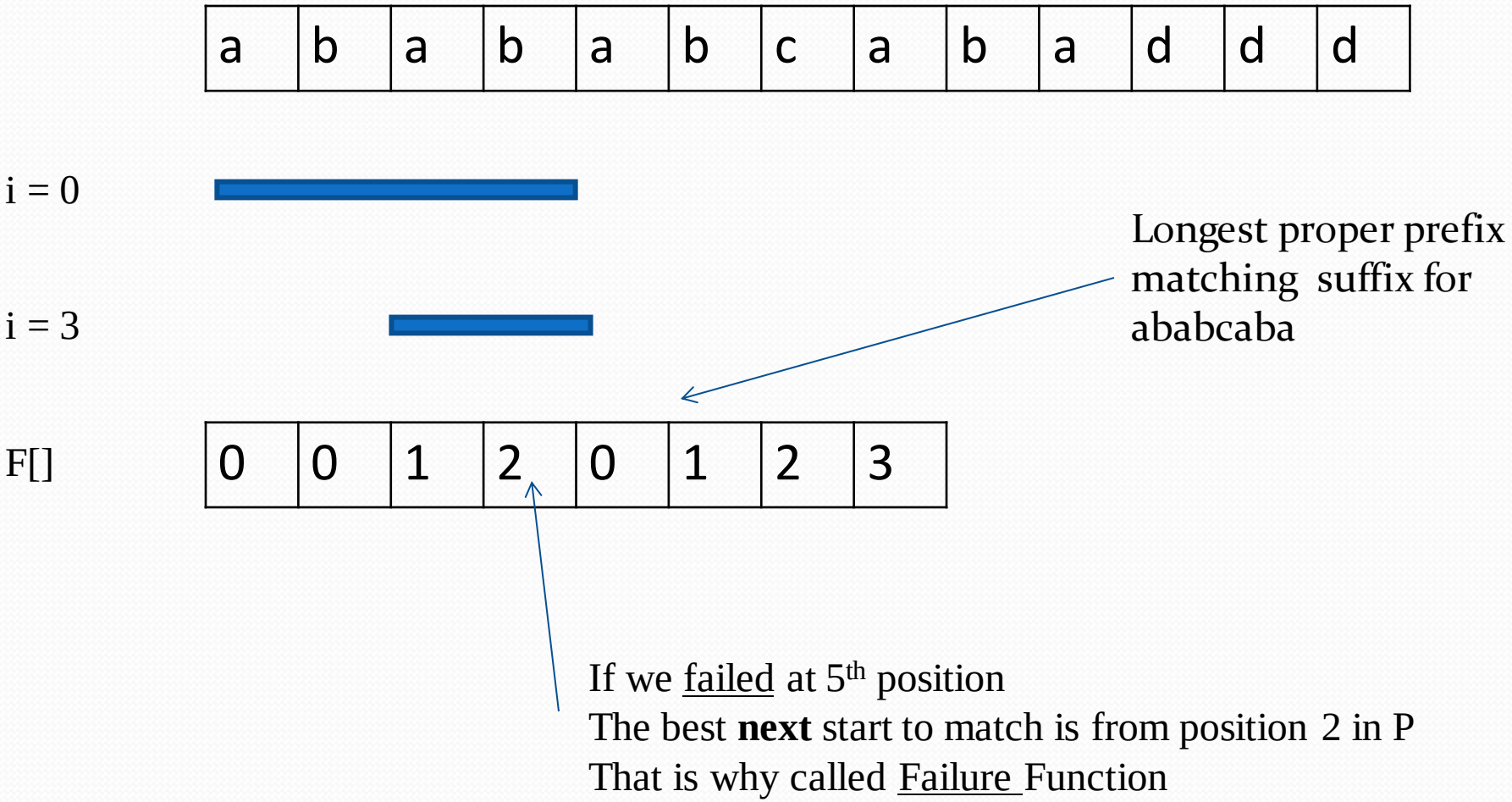
To match these 2 letters in the pattern
Prefix of 2 letters = suffix of 2 letters
ab = ab

Then, we know we could start from i = 2, and see if we could match after these first 4 letters

KMP Shift Rule

- If we matched N letters from i, but can't match more
  - The pattern could match the last c letters IFF last C letters are same as first C letters
  - In other words prefix of C letters = suffix of C letters
  - Make sense C != N, as N failed
  - So actually prefix is a proper prefix
- If we matched N letters from i, but can't match more
  - The next nearest match need minimum shift
  - Greedily we need the longest proper prefix = suffix

Match ababcaba



Match aabbaabb

F[]

0	1	0	0	1	2	3	4

a	a	b	b	a	a	b	c	a	a	b	a	x

i = 0

We matched 7 chars and failed

i = 4

F[7] = 3. Start after suffix aab
However we can't match with c

i = 8

F[3] = 0. Start after suffix ""
Which means we can't match at c, even with pattern start

Start to match from pattern start with position 9

KMP Algorithm

```
void KMP(string str, string pat)
{
    int n = sz(str);
    int m = sz(pat);
    vector<int> longestPrefix = computePrefix(pat);

    for(int i = 0, k = 0; i < n ; i++) {
        // as long as we can't add one more character in k, get best next prefix
        while (k > 0 && pat[k] != str[i])
            k = longestPrefix[k - 1];

        // if we match character in the pattern, move in pattern
        if (pat[k] == str[i])
            k++;

        // if we matched, print it and let's find one more matching
        if (k == m) {
            cout<<i - m + 1<<"\n";
            k = longestPrefix[k - 1];   // fail to next best suffix
        }
    }
}
```

Run Algorithm

- KMP(aaaa, aa)= 0, 1, 2
- KMP(ababacdab, aba) = 0, 2
- KMP(abc, abc) = 0
- KMP(abcabca, abca) = 0, 3

Failure Function Algorithm

- How to do it efficiently?
- Match the pattern against itself!
- In other words, use similar code to KMP

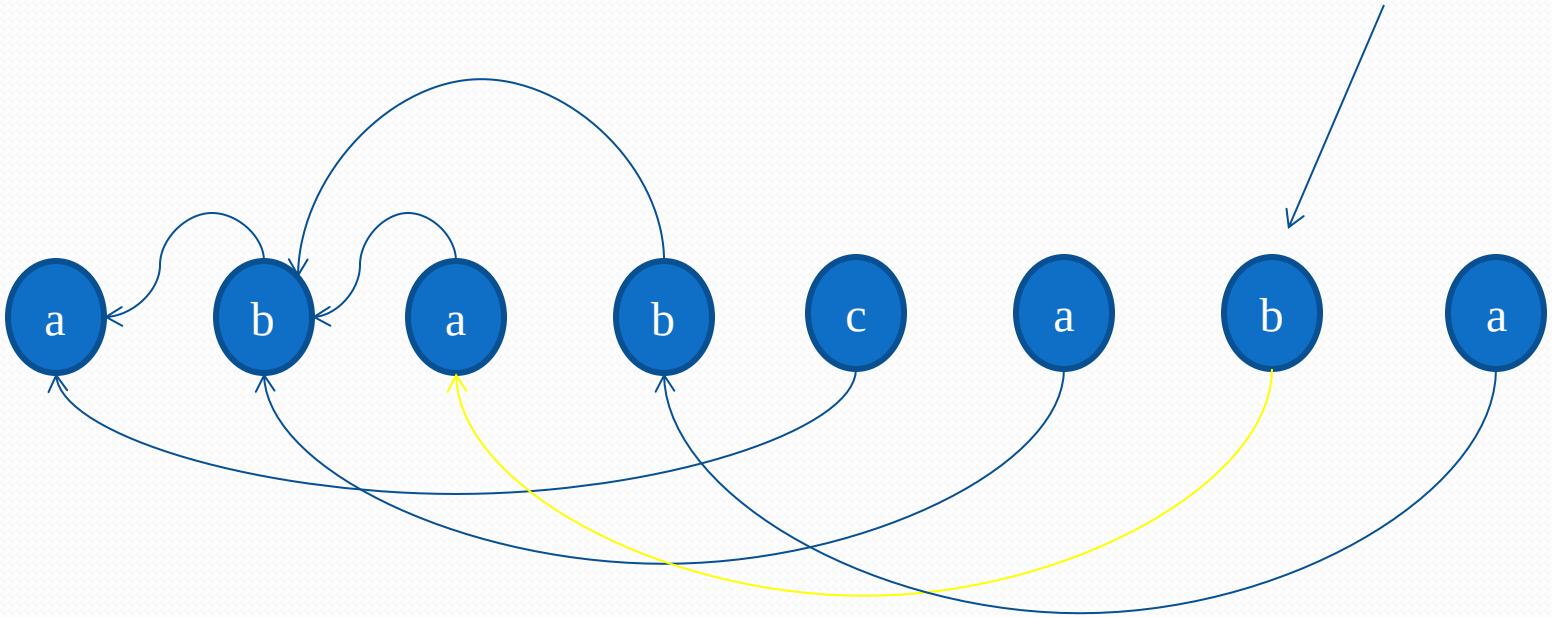
Failure Function Algorithm

```
vector<int> computePrefix(string pat) {
    int m = sz(pat);
    vector<int> longestPrefix(m);

    for (int i = 1, k = 0; i < m; ++i) {
        // as long as we can't add one more character in k, get best next prefix in P
        while (k > 0 && pat[k] != pat[i])
            k = longestPrefix[k - 1];

        if(pat[k] == pat[i])
            longestPrefix[i] = ++k;
        else
            longestPrefix[i] = k;
    }

    return longestPrefix;
}
```

“Failure” of ababcaba

0	0	1	2	0	1	2	3

If failed to match after 6th position,
Fail and try to match from 2nd position



Order Analysis

- O(m) for failure function
- O(n) for KMP matching

- Overall: O(n+m) instead of O(nm)

- Exercise1 : Simulate by hand examples to fully get it
- Exercise2 : prove order