**String matching**

 searching a string within another string

<https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/>

S = stringul in care cautam patternul

P = pat = patternul / tiparul / stringul cautat

n = lungimea lui S

m = lungimea lui P

c in E = litera in alfabet

q = starea (intre 0 si n-1)

pi = lps = lungimea celui mai lung prefix care este si sufix propriu

A proper prefix is prefix with whole string **not** allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC” and “ABC”.

For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

1. Algoritm de căutare naivă (brute-force) -> O(nm)

( se compară literă cu literă )

1. Cautare exhaustiva -> O(m3)

Idei de optimizare:

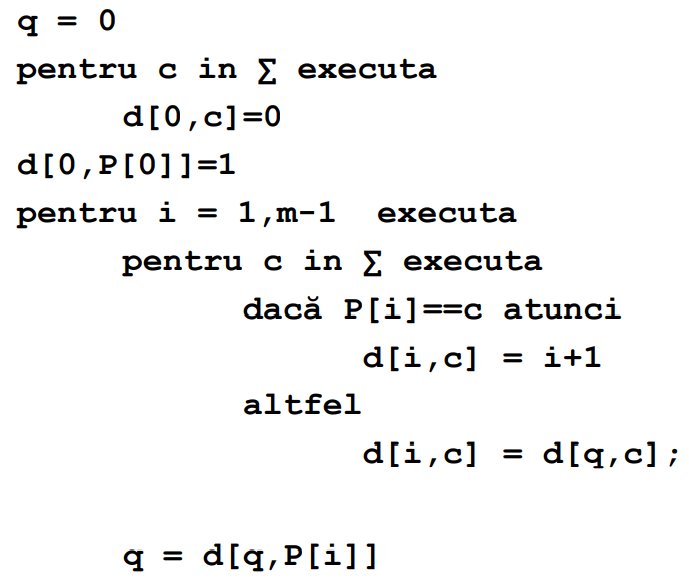
* la nepotrivire nu ne mai intoarcem; mutăm tiparul astfel încât un sufix cât mai mare al lui S[0:i] să fie prefix al tiparului P și continuăm cu i+1 și următorul caracter din tipar
* cum putem determina acel q maxim pentru care prefixul de lungime q al tiparului P[0..q-1] este sufix pentru S[0..i] ? Solutii:

1. Automat asociat tiparului P, cu stări corespunzătoare prefixelor lui P și tranziții corespunzătoare tuturor literelor din alfabet

2. Doar un vector prefix care pentru fiecare i memorează prefix maxim=sufix al lui P[0:i] este suficient pentru a face căutarea în O(n)

1. Cautarea tiparului cu automat de potrivire asociat tiparului -> O(m\*nr\_litere\_alfabet) timp & sp

Dezavantaj : memorie – reține câte o tranziție pentru fiecare literă



<https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/>

1. **Algoritm Knuth – Morris – Pratt (KMP)**  -> O(n)

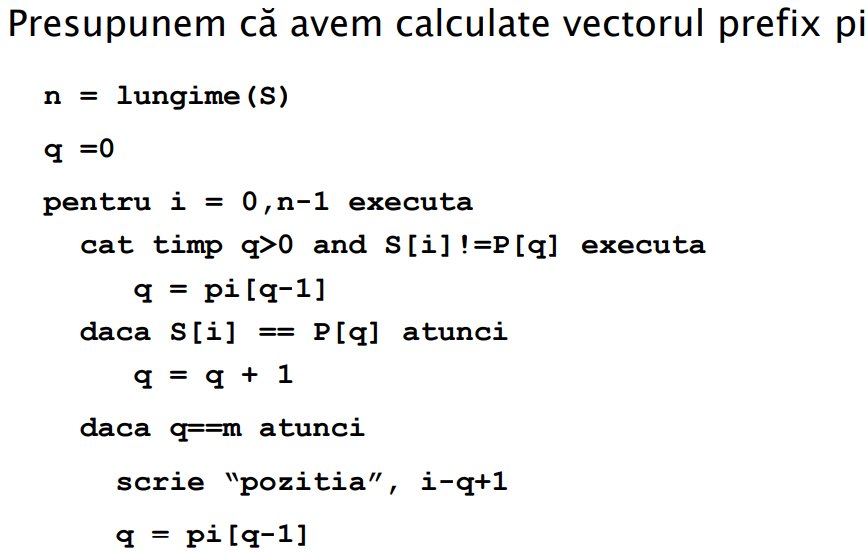
<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.

how to know how many characters to be skipped?

* pre-process pattern and prepare an integer array lps[] that tells the count of characters to be skipped.

în starea i nu reținem câte o tranziție pentru fiecare P[i], ci o singură tranziție de “eșec” către q =π[P[i]] (către q din construcția automatului) = cel mai lung prefix=sufix propriu determinat pana acum ignorând caracterul care nu se potrivește de la pasul curent => memorie O(m)



void computeLPSArray(char\* pat, int M, int\* lps) {

    int len = 0; // length of the previous longest prefix suffix

    lps[0] = 0; // lps[0] is always 0

    int i = 1; // the loop calculates lps[i] for i = 1 to M-1

    while (i < M) {

        if (pat[i] == pat[len]) {

            len++;

            lps[i] = len;

            i++;

        }

        else // (pat[i] != pat[len])

        {

            // This is tricky. Consider the example.

            // AAACAAAA and i = 7. The idea is similar to search step.

            if (len != 0) {

                len = lps[len - 1];

                // Also, note that we do not increment i here

            }

            else // if (len == 0)

            {

                lps[i] = 0;

                i++;

            }

        }

    }

}

// Prints occurrences of txt[] in pat[]

void KMPSearch(char\* pat, char\* txt)

{

    int M = strlen(pat);

    int N = strlen(txt);

    int lps[M]; // create lps[] that will hold the longest prefix suffix values for pattern

    computeLPSArray(pat, M, lps);    // Preprocess the pattern (calculate lps[] array)

    int i = 0; // index for txt[]

    int j = 0; // index for pat[]

    while (i < N) {

        if (pat[j] == txt[i]) {

            j++;

            i++;

        }

        if (j == M) {

            printf("Found pattern at index %d ", i - j);

            j = lps[j - 1];

        }

        // mismatch after j matches

        else if (i < N && pat[j] != txt[i]) {

            // Do not match lps[0..lps[j-1]] characters,

            // they will match anyway

            if (j != 0)

                j = lps[j - 1];

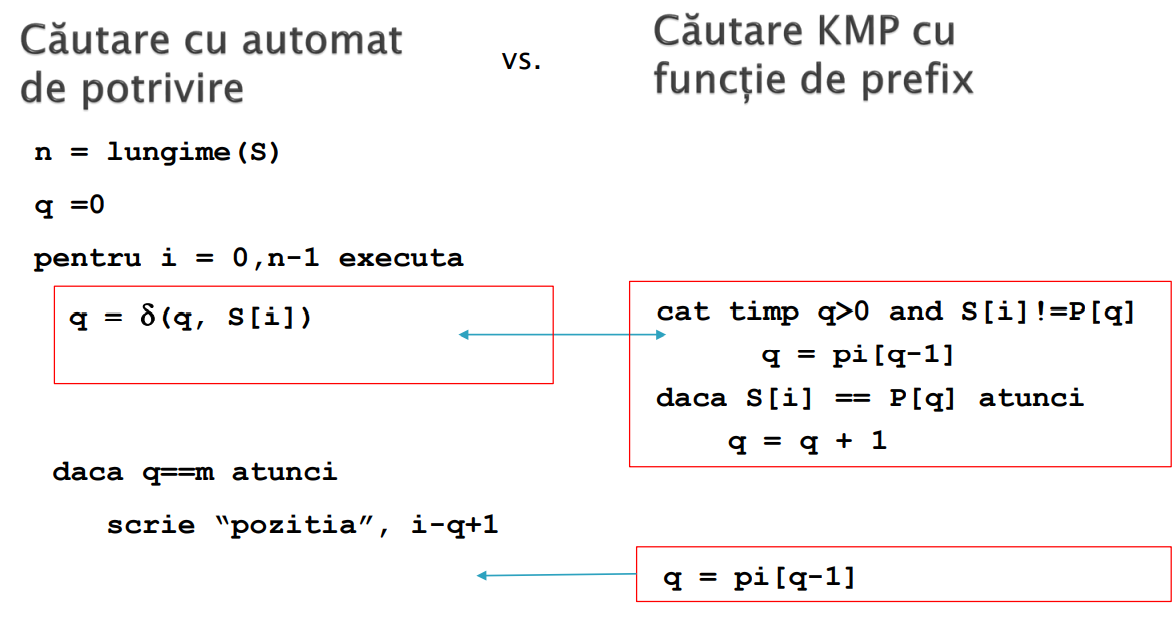
            else

                i = i + 1;

        }

    }

}



Cursul 12

Cel mai scurt superstring = Set Cover

<https://www.geeksforgeeks.org/set-cover-problem-set-1-greedy-approximate-algorithm/>

1. Greedy -> asemanator APCM

* alg aproximativ, cu grad 2 de aproximare <=> 2 stringuri coincid pe o lungime de maxim 2 litere

<https://gist.github.com/ovidiudeica/90f4f40a6045cce857dd344ea3c96aad>