



11. prednáška (07. 05. 2018)

As we already know, at any moment the rightmost lemming carries all the information we have about the current state of the gadget. Also, the rightmost lemming is the closest one to reaching the goal, i.e., completing the traversal of the gadget. If we keep track of it, we never risk missing a match.

In order to avoid off-by-one errors in the following paragraphs, we need to clearly define a way of numbering the positions in the gadget: The *position* of a lemming will be the *number of pits the lemming has already crossed*. (Or, equivalently, the number of seconds the lemming has been alive.) We will use the same number to denote the location where the lemming is standing at that moment. Hence, possible positions of a lemming range from 0 (the beginning) to n (the end of the gadget), inclusive.

In the new, faster simulation we will be keeping track of the position of the currently rightmost lemming in the gadget. Therefore, the *state* of the simulation will be a single integer: *the number of pits the rightmost lemming already crossed*.

Some steps of the new simulation will be faster: if the rightmost lemming gets another bridge, we just move it one step ahead and we are done with processing a letter of H . On the other hand, as soon as the rightmost lemming falls into a pit, we are out of luck. Whenever that happens, we have to check whether and which of the other lemmings in the gadget survive the next step. More precisely, we just need to find the location of the new rightmost lemming.

As we already know, at any moment the rightmost lemming carries all the information we have about the current state of the gadget. Also, the rightmost lemming is the closest one to reaching the goal, i.e., completing the traversal of the gadget. If we keep track of it, we never risk missing a match.

In order to avoid off-by-one errors in the following paragraphs, we need to clearly define a way of numbering the positions in the gadget: The *position* of a lemming will be the *number of pits the lemming has already crossed*. (Or, equivalently, the number of seconds the lemming has been alive.) We will use the same number to denote the location where the lemming is standing at that moment. Hence, possible positions of a lemming range from 0 (the beginning) to n (the end of the gadget), inclusive.

In the new, faster simulation we will be keeping track of the position of the currently rightmost lemming in the gadget. Therefore, the *state* of the simulation will be a single integer: *the number of pits the rightmost lemming already crossed*.

Some steps of the new simulation will be faster: if the rightmost lemming gets another bridge, we just move it one step ahead and we are done with processing a letter of H . On the other hand, as soon as the rightmost lemming falls into a pit, we are out of luck. Whenever that happens, we have to check whether and which of the other lemmings in the gadget survive the next step. More precisely, we just need to find the location of the new rightmost lemming.

Vyhľadávanie v textoch (podobnosť textov)





Motivácia

- Bakalárske štúdium – záverečná bakalárska práca
- Kontrola originality – centrálny register prác
- Odpoveď – percentuálne vyjadrenie podobnosti s nejakými prácami, ktoré už sú v registri a ukážky podobnosti
- **Záujmy študentov:**
 1. Napísať prácu, ktorá bude z hľadiska originality hodnotená dobre.
 2. Vedieť ako pracujú algoritmy (programy), ktoré toto robia.



Plagiátorské práce

- **Cudzí (extrensic) plagiát** – práca, v ktorej niekto opísal časti práce (CTRL-C/CTRL-V alebo aj málo pozmenil vety)
- **Vlastný (intrinsic) plagiát** – práca, v ktorej sa niekto pokúša napísať text za daného autora (musí poznať veľa vecí o jeho štýle, slovníku, atď.)
- Najjednoduchšie vyhodnotenie – zistiť či bolo urobené CTRL-C/CTRL-V

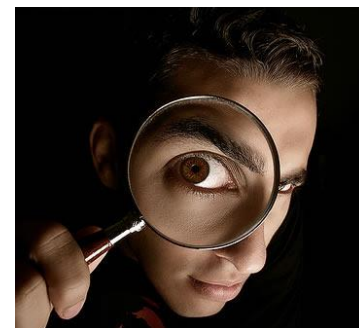


Vyhľadávanie v textoch

1. Jednoduchá úloha **nájsť v texte všetky výskyty** **zadaných textových vzoriek** (kľúčových slov, alebo ich častí)

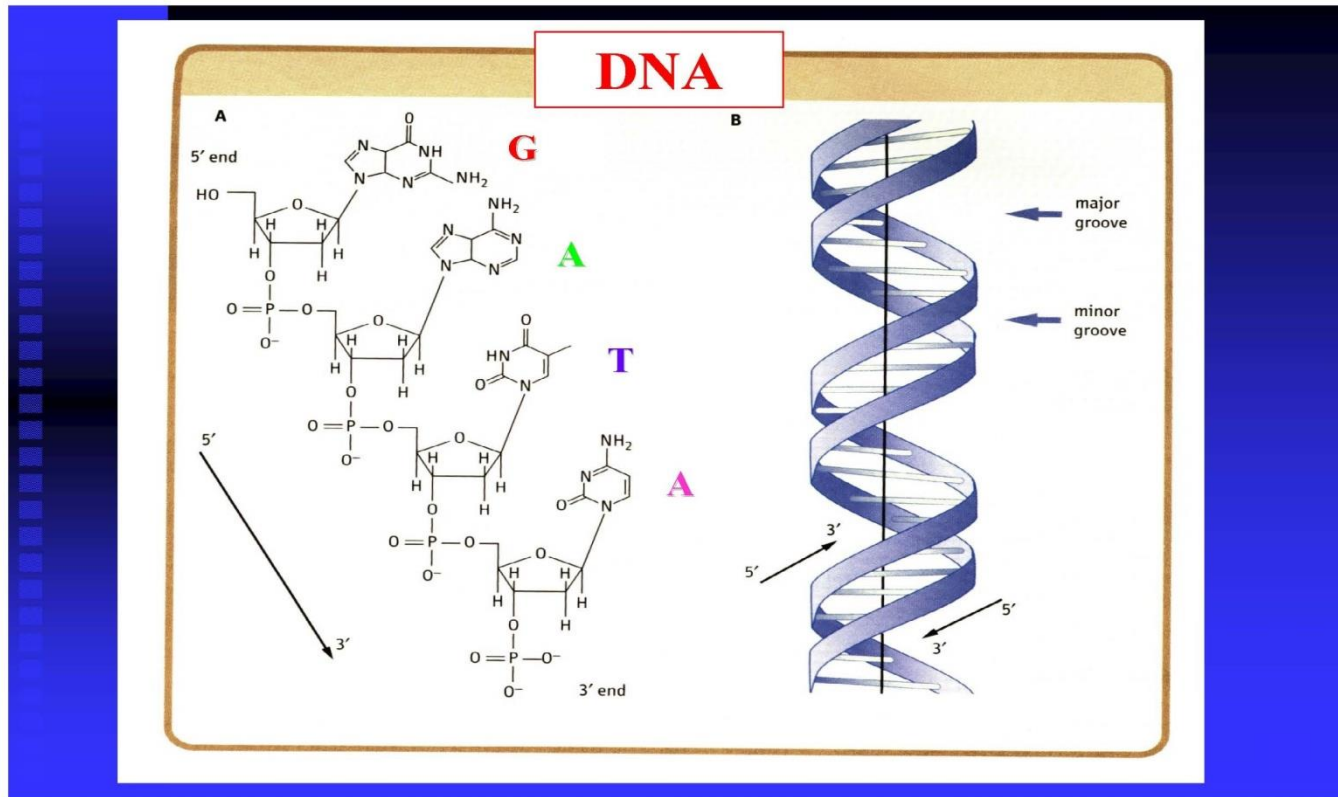
Kde všade sa stretneme s týmto vyhľadávaním:

- textové editory,
- rešeršné systémy,
- ale i taký internet (google, zoznam, ...),
- utility typu grep v Unixe,
- kontrola plagiátorstva





DNA reťazce



GATUTUUAGTGGUGGAUTATUTGGUTGUGTGUAU...

Hľadáme úsek: AUTATU



Vyhľadávanie v textoch

- Vyhľadávanie v bežnom textovom dokumente - „**naivný algoritmus**“, je časovo náročný, vyžaduje viacnásobné čítanie textu
- V prípade veľkých rešeršných systémov by naivný algoritmus pracoval veľmi pomaly, preto je dôležité hľadať efektívnejšie riešenia
- Existuje viacero zaujímavých algoritmov, ktoré pracujú v lineárnom čase vzhľadom na dĺžku textu



Vyhľadávanie v textoch

Obsah

- Naivný algoritmus
- Knuth-Morris-Prattov (KMP) algoritmus
- Boyer-Mooreov (BM) algoritmus
- Rabin-Karpov (RK) algoritmus



Vyhľadávanie v textoch

- Považujeme text za dlhý reťazec znakov.
- Hľadané slovo je krátky reťazec znakov.
- Základný problém – nájsť slovo v texte.
- Je možné hľadať niektorý výskyt, ale tiež všetky výskyty.
- Napríklad

Do lesa vedie cesta cez prales Koleso

les



História algoritmov

- Brute force (naivný) algoritmus
 - známy veľmi dávno
 - preskúmaný viackrát, stále zostáva východiskom
- Knuth & Pratt, 1970
 - nezávisle od nich toto vylepšenie našiel Morris
 - spoločne publikovali 1976 ako “Knuth-Morris-Pratt (KMP)”
- Boyer & Moore našli lepší algoritmus ešte pred 1976
 - nezávisle od nich toto vylepšenie našiel aj Gosper
- Karp & Rabin našli ďalšie vylepšenie v 1980



Na začiatku ...

Otázka:

Daný je textový reťazec “**abcdefghijkl**” a vzorový reťazec P, zistiť, či sa P vyskytuje v textovom reťazci.

Riešenie: Ako to je vlastne urobené?

Java:

```
String s="abcdefghijkl";  
If( s.indexOf(P)>0 )  
    return true;  
else  
    return false;
```

C++:

```
String s="abcdefghijkl";  
If( s.find(P,0)!=string::npos )  
    return true;  
else  
    return false;
```



Brute force

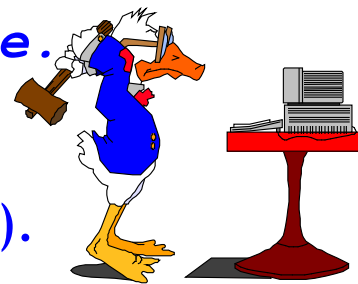
- Algoritmus je založený na myšlienke: pokúšať sa nájsť slovo od každej pozície v texte a porovnávať všetky znaky slova so znakmi v texte :

```

for  (i = 0; i<=n-m; i++)    { //text t dĺžky n
    for  (j = 0; j<= m-1; j++) { //slovo p dĺžky m
        porovnaj p[j] s t[i+j]
        if sa nerovná, exit vnútorného cyklu
    }
    //V prípade, že prebehne celý vnútorný cyklus,
    //vzorový reťazec sa v cieľovom vyskytuje.
}

```

- Zložitosť je v najhoršom prípade $O(m*n)$ a v najlepšom $O(n)$.





Ako dlho to trvá?

- Nech textový reťazec má 100GB a vzorový reťazec má 100MB? $n=100\text{GB}$, $m=100\text{MB}$
- Čo v prípade, že **indexOf()** a **find()** metódy použijú brute force vyhľadávanie?

Celkový čas je: $O(n*m)$

pre 100GB textové data a 100MB vzorový reťazec,

Čas je **277777 hodín (32 rokov)** na počítači s 10GHz CPU – za predpokladu, že jedno porovnanie zaberie 1 clock



Vylepšenie vyhľadávania v texte

- Existuje rýchlejší algoritmus, v ktorom sa porovnávajú posledné znaky najprv:

Do the first then do the other one

the

porovnaj 'e' s ' ', nerovný, tak sa posuň o 3 miesta

Do the first then do the other one

the

je možný posun len 2 miesta



Vylepšenie vyhľadávania v texte (2)

Pozorovanie:

- V každom prípade, keď **daný znak v dokumente nie je ani jedným zo znakov hľadaného slova**, je možný posun o n znakov. V niektorých prípadoch je to menej.



Definícia problému, terminológia

- Nech p je vzorový reťazec (pattern string)
- Nech t je cieľový reťazec (target string)
- Nech k je index znaku v cieľovom reťazci, ktorý “leží nad” prvým znakom vzorového reťazca
- Úloha:
 - Pre dané dva reťazce, p a t , v abecede Σ , určiť, či sa p vyskytuje ako podreťazec v t
 - Teda, určiť či existuje index k taký, že platí $p = \text{Substring}(t, k, |p|)$.



Priame vyhľadávanie v reťazci

Pseudokód:

```
function SimpleStringSearch(string p,t): integer;  
// Nájde prvý výskyt  $p$  v  $t$  (ak existuje);  
// vráti jeho pozíciu alebo -1, ak  $p$  nie je podreťazcom  $t$   
  for  $k = 0$  to  $Length(t) - Length(p)$  do {  
     $i = 0$ ;  
    while  $i < Length(p)$  a zároveň  $p[i] = t[k+i]$  do  
       $i = i+1$ ;  
    if  $i == Length(p)$  then return  $k$   
  }  
return -1
```



Aplikácia SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

Y Y Y N



Aplikácia SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

N



Aplikácia SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

N



Aplikácia SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

N



Aplikácia SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N



Aplikácia SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N



Aplikácia SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N



Aplikácia SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

Y Y Y Y



Zložitosť SimpleStringSearch

- Najhorší prípad:
 - Vzorový reťazec sa vždy zhodne s textom okrem posledného znaku
 - Príklad: hľadať **XXXXY** v cieľovom reťazci
XXXXXXXXXXXXXXXXXXXXXXX
 - Vonkajší cyklus je vykonávaný pre každý znak v cieľovom reťazci
 - Vnútorňý cyklus sa vykoná jedenkrát pre každý znak vo vzorovom reťazci
 - $\Theta(|p| * |t|)$
- Je to v poriadku, keď sú reťazce krátke, ale existuje lepší algoritmus



Knuth-Morris-Pratt

- Kľúčová idea:
 - ak vzor sa v znaku nezhodne s cieľom,
tak posunúť vzor napravo o toľko znakov, koľko je
najviac možné
so zachovaním už zistenej zhody



Knuth-Morris-Pratt

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

X	Y	X	Y	X	Y	c				
---	---	---	---	---	---	---	--	--	--	--

$p[0]$ $p[1]$ $p[2]$ $p[3]$ $p[4]$

X	Y	X	Y	Z
---	---	---	---	---

Y Y Y Y N

X	Y	X	Y	Z
---	---	---	---	---

Y Y Y Y ?



Knuth-Morris-Pratt

- Správne posunutie vzoru závisí od oboch indexov nezhody a od znakov nezhody
- Ak $c == X$: posun o 2 znaky napravo
- Ak $c == E$: posun o 5 znakov napravo
- Ak $c == Z$: zhoda nájdená; algoritmus končí
- ***Ako zistiť, o koľko znakov je možný posun?***



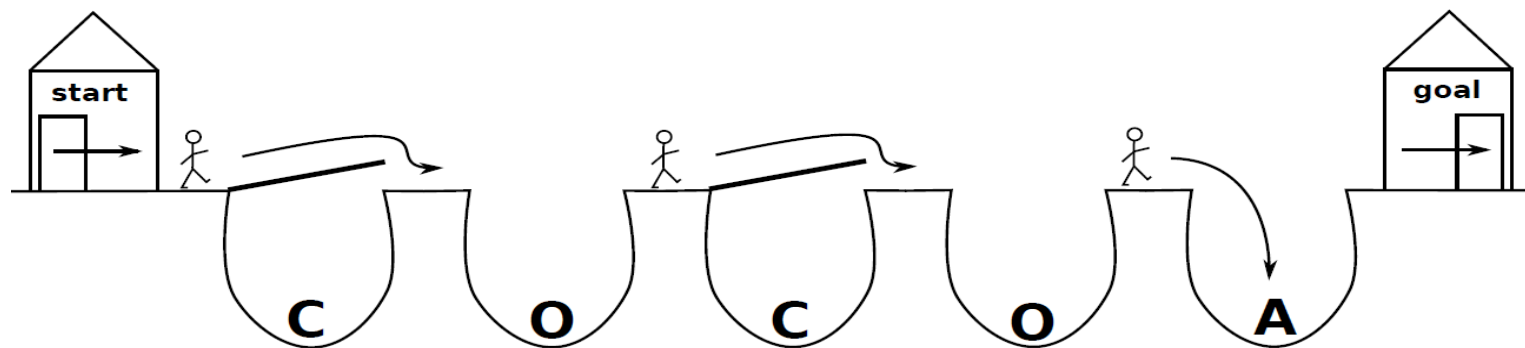
Metafóra pre KMP algoritmus

- *Na vysvetlenie KMP algoritmu použijeme metaforu „**lemingov**“ a **dômyselnú mašinku**.*
- ***Leming** je humanoidný robot, ktorý žije podľa určitých pravidiel.*
- ***Náš leming** v pravidelných časových okamihoch vykročí doprava.*
- *Ak leming spadne do jamy, rozbije sa.*



Lemingovia a mašinka

● *Dômyselná mašinka:*



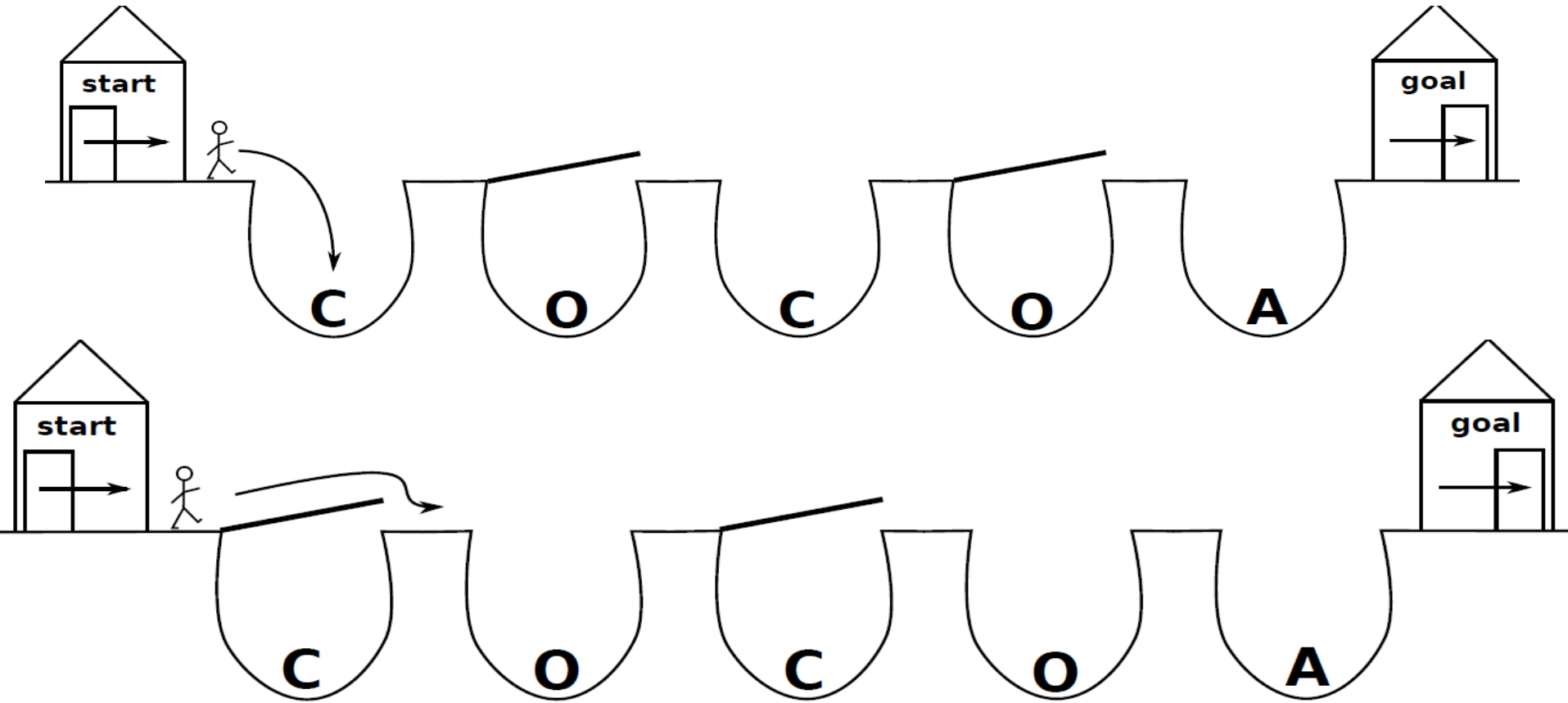
Pozostáva z generátora lemingov, jamôk na písmenká, mostíkov a cieľa.

Písmenká sú vložené do jamôk na začiatku (hľadané slovo).

Lemingovia sa generujú na štarte v každom časovom okamihu a vykročia po našom prečítaní písmenká z nejakého textu.



Lemingovia a mašinka



Metafóra je spracovaná podľa knihy M. Foríška a M. Steinovej

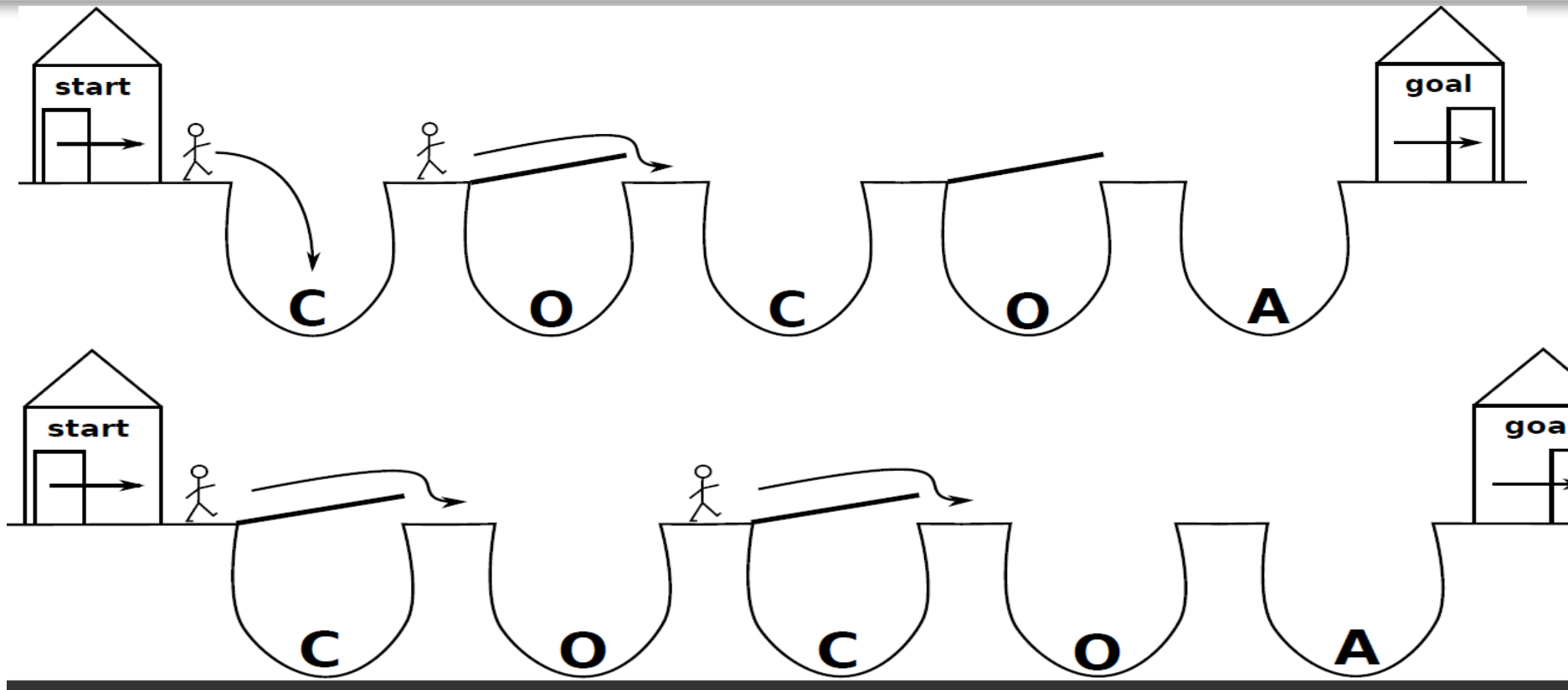


Lemingovia a mašinka

- Hovoríme (čítame) písmenká z textu
- Keď povieme písmenko, vtedy nad jamou, v ktorej sa nachádza také isté (prečítané) písmenko sa vysunie most a leming po ňom môže prejsť a nespadne do jamy
- Šťastní sú lemingovia, pre ktorých sa vygenerujú mosty, prežijú a dostanú sa do cieľa
- Mnoho lemingov sa rozbije v jamách
- Budeme hovoriť písmenká v poradí: **OCOCOC**.



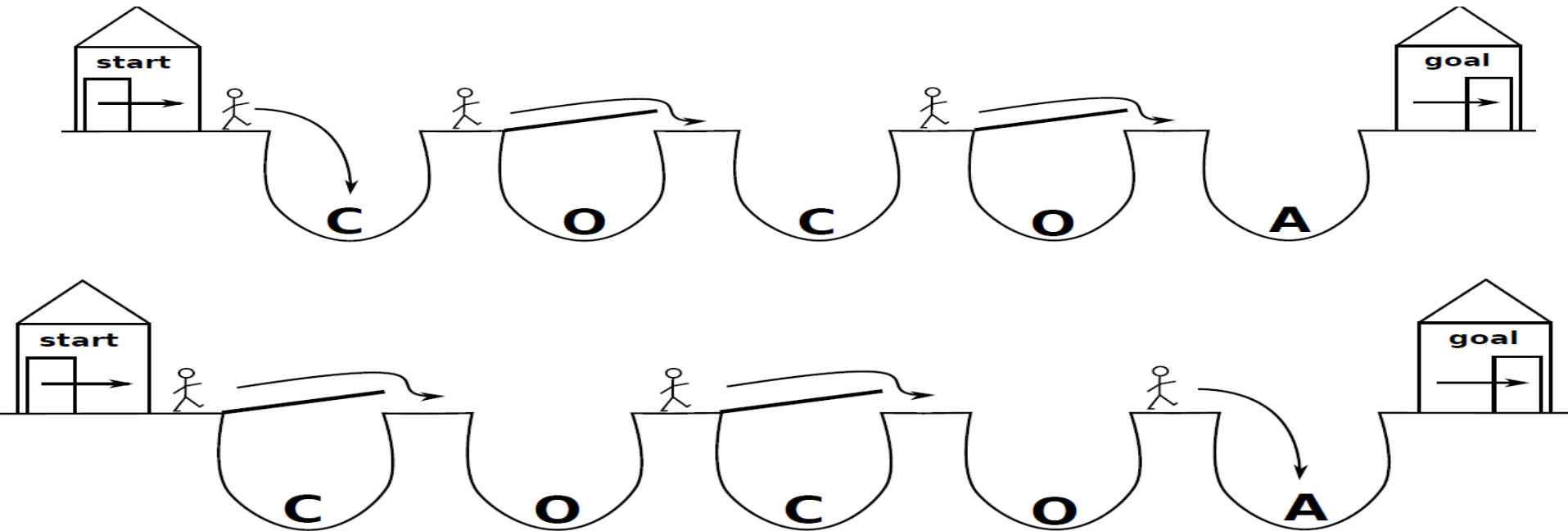
Metafóra pre KMP



Budeme hovoriť písmenká v poradí: **O**COCOC.



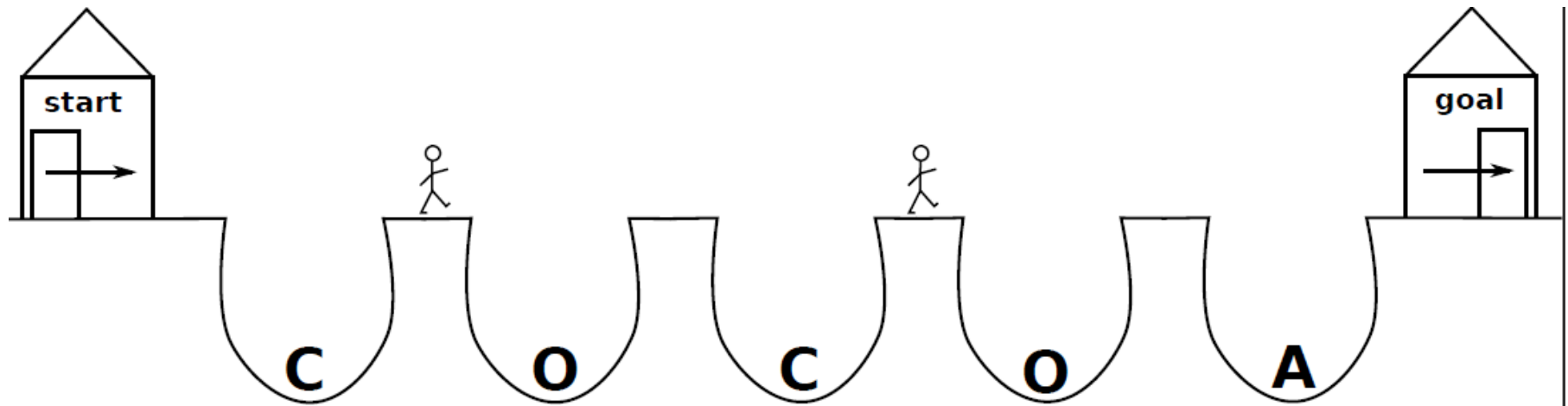
Metafóra pre KMP



Budeme hovoriť písmenká v poradí: **OCOCOC**.



Metafóra pre KMP



Situácia po skončení OCOCOC.



Lemingovia a mašinka

- 1. Nový leming sa objaví na začiatku mašinky.
- 2. Prečítame nahlas písmenko.
- 3. Niektoré mosty sa objavia (nad jamami s prečítaným písmenom).
- 4. Počkáme sekundu. Počas druhej sekundy lemingovia, ktorí majú most, prejdú a ktorí ho nemajú padnú do jamy a rozbijú sa.
- 5. Keď leming dôjde na koniec, práve sa našiel **výskyt podreťazca v jamkách v nami čítanom texte.**

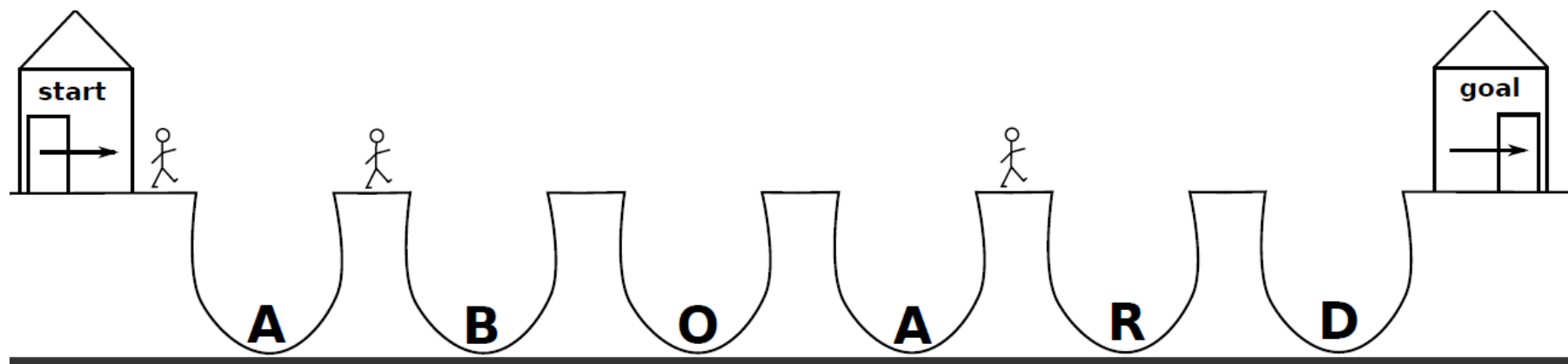


Metafóra pre KMP

- Pomocou tejto mašinky a lemingov môžeme korektne vyriešiť problém vyhľadávania podreťazca v texte.
- Sekvenčná simulácia má v najhoršom prípade časovú zložitosť $O(m \cdot n)$.



Metafóra pre KMP



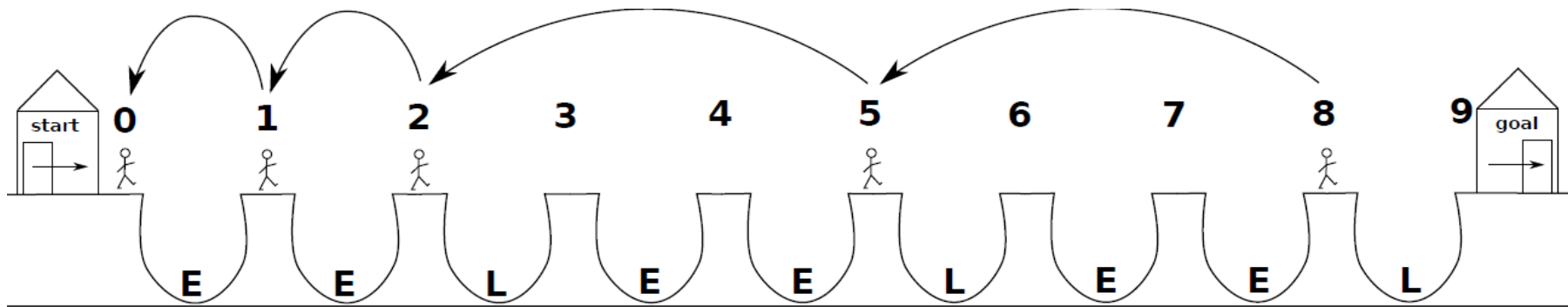
Analyzujte, čo sa stane pri prečítaní rôznych písmen.

Zaved'me si odkazové pole na predchádzajúcich lemingov – S.

$$S[0]=0, S[1]=0, S[4]=1, S[3]=N, S[2]=N$$



Metafóra pre KMP



Najpravejší leming je v pozícii 8, a implikuje, že druhý leming sprava je v pozícii 5, atď'.

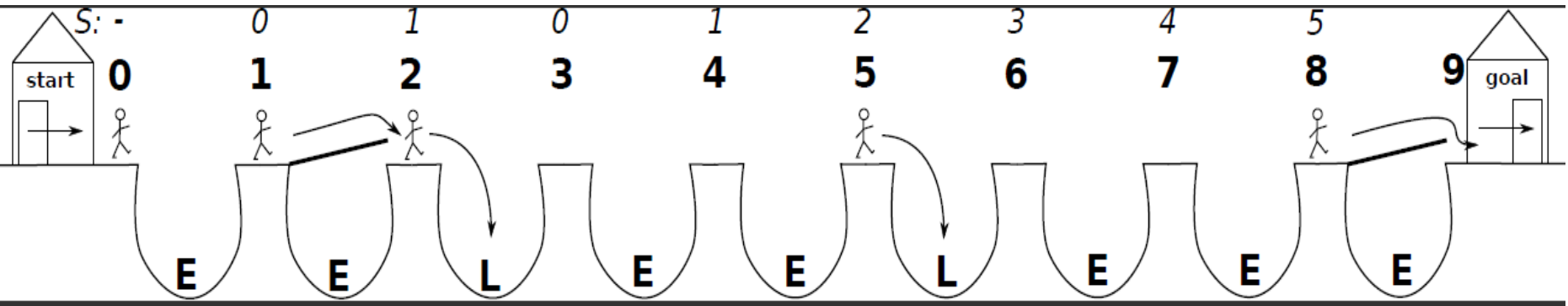
Nech hodnota $S[i]$ je index nasledujúceho leminga zľava od leminga, ktorý je v pozícii i

Predpokladajme, že už máme vypočítané posuny S

$$S[8]=5, S[S[8]] = S[5]=2, S[2]=1, S[1]=0$$



Metafóra pre KMP





Metafóra pre KMP

- Keď máme vypočítané S , tak simulácia mašinky je jednoduchá.
- 1. Nech f je pozícia leminga, ktorého práve sledujeme. Na začiatku je len jeden aktívny leming v pozícii 0, teda inicializujeme f na 0.
- 2. Pre každé písmeno x z reťazca t (textový reťazec):
 - a. Nájsť najpravejšieho leminga, ktorý prežíva, keď nasleduje písmenko x . Toto je možné urobiť v nasledujúcom cykle:

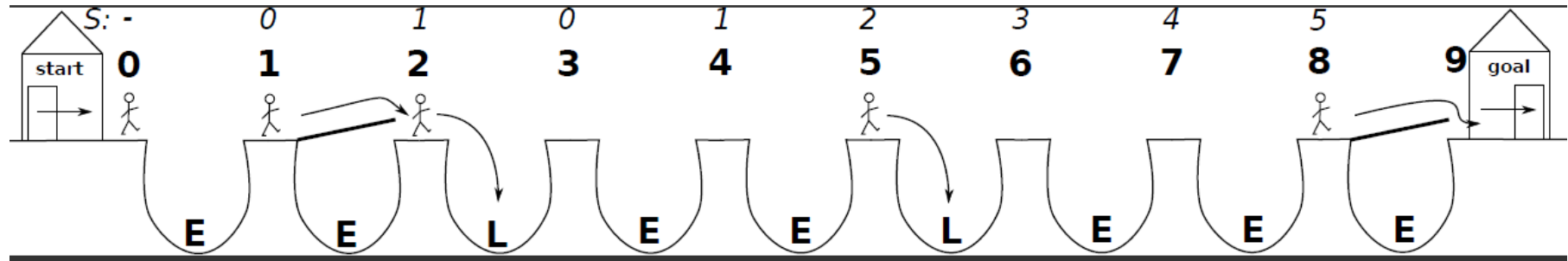


Metafóra pre KMP

- A) **POKIAL'** je f definované:
 - x. Overiť, či leming v pozícii f prežije ďalší krok.
 - y. Ak áno, prerušiť cyklus, nasleduje krok C (a potom v kroku C zvýšiť f o 1).
 - z. Ak nie, posunúť sa ďalšieho aktívneho leminga nastavením f na $S[f]$.
- B) Ak f je nedefinované (t. j. žiadny leming neprežil), nastaviť f na 0: po spracovaní x jediný leming, ktorý bude v mašinke žiť, bude leming na začiatku.
- C) Inak, zvýšiť f o 1: najpravejší leming, ktorý prežil, sa posúva cez most na nasledujúcu pozíciu.



Efektívny predvýpočet



Ako vypočítame $S[9]$?

Budeme počítat' $S[i]$ sekvenčne s rastúcim i .

Aby leming prešiel, musíme povedať E, ..., preto $S[9]=2$



Efektívny predvýpočet

- Algoritmus môžeme zovšeobecniť aj pre predvýpočet.
- Jeho implementácia je presne tá istá ako u predchádzajúceho algoritmu pre text s nasledujúcimi dvoma rozdielmi:
- Namiesto písmen z **t** budeme používať písmená z reťazca **p**, a
- Pozícia **f** bude vždy ukazovať na druhého aktívneho leminga sprava.



Efektívny predvýpočet

- Ten istý argument ako predtým nám pomôže ukázať, že časová zložitosť predvýpočtu je $\Theta(n)$,
- teda celková časová zložitosť je $\Theta(n+m)$.



Knuth-Morris-Pratt – iný pohľad

- **Cieľ:** určiť d , počet znakov, o ktoré je možný posun napravo; nájsť najmenšie d také, že:
 - $p[0] = t[k+d]$
 - $p[1] = t[k+d+1]$
 - $p[2] = t[k+d+2]$
 - ...
 - $p[i-d] = t[k+i]$



Knuth-Morris-Pratt

- **Poznámka:** pomocou vzoru samotného a znaku, ktorý spôsobil nezhodu
- Hodnota d závisí len od:
 - Vzorového reťazca
 - Hodnoty i
 - Od znaku v textovom reťazci, ktorý sa nezhoduje - znak c (v položke $t[k+i]$)



Knuth-Morris-Pratt

- Môžeme definovať funkciu *KMPskip(p, i, c)*, ktorá bude počítat správne d
 - Vráti najmenší index d , $0 \leq d \leq |p|$, taký že $p[i-d] == c$ a $p[j] == p[j+d]$ pre každé $0 \leq j \leq i-d-1$
 - Vráti $i+1$, ak žiadne také d neexistuje
- Vypočítat všetky hodnoty *KMPskip* pre vzor p a uložiť ich v dvojrozmernom poli *KMPskiparray*
- Urobiť cyklus pre každú nezhodu



Knuth-Morris-Pratt

- Pre vzorový reťazec ABCD :

c =

p=	A	B	C	D
A	0	1	2	3
B	1	0	3	4
C	1	2	0	4
D	1	2	3	0
iné	1	2	3	4

Vráti najmenší index d , $0 \leq d \leq |p|$,
taký že

$p[i-d] == c$ a

$p[j] == p[j+d]$ pre každé
 $0 \leq j \leq i-d-1$

$c == A$, $i = 0$, $p[i-0] == c == A$

Teda $d = 0$

$p[j] == p[j+d]$ pre všetky j ,
 j vlastne neexistuje



Knuth-Morris-Pratt

- Pre vzorový reťazec ABCD :

c =

p=	A	B	C	D
A	0	1	2	3
B	1	0	3	4
C	1	2	0	4
D	1	2	3	0
iné	1	2	3	4

Vráti najmenší index d , $0 \leq d \leq |p|$,
taký že

$p[i-d] == c$ a

$p[j] == p[j+d]$ pre každé

$0 \leq j \leq i-d-1$

$c == A$, $i = 1$, $p[i-1] == c == A$

Teda $d = 1$

$p[j] == p[j+d]$ pre všetky j ,
 j vlastne neexistuje



Knuth-Morris-Pratt

- Pre vzorový reťazec ABCD :

c =

p=	A	B	C	D
A	0	1	2	3
B	1	0	3	4
C	1	2	0	4
D	1	2	3	0
iné	1	2	3	4

Vráti najmenší index d , $0 \leq d \leq |p|$,
taký že

$p[i-d] == c$ a

$p[j] == p[j+d]$ pre každé

$0 \leq j \leq i-d-1$

$c == A$, $i = 2$, $p[i-2] == c == A$

Teda $d = 2$

$p[j] == p[j+d]$ pre všetky j ,
 j vlastne neexistuje



Knuth-Morris-Pratt

- Pre vzorový reťazec ABCD :

c =

p=	A	B	C	D
A	0	1	2	3
B	1	0	3	4
C	1	2	0	4
D	1	2	3	0
iné	1	2	3	4

Vráti najmenší index d , $0 \leq d < |p|$,
taký že

$p[i-d] == c$ a

$p[j] == p[j+d]$ pre každé

$0 \leq j \leq i-d-1$

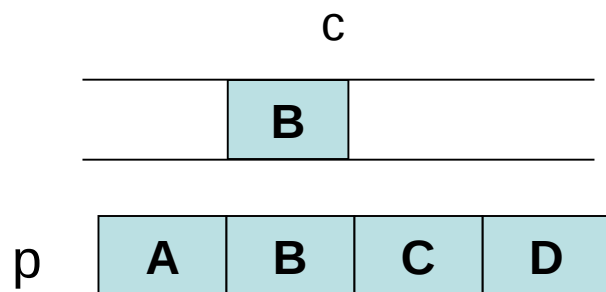
$c == A$, $i = 3$, $p[i-3] == c == A$

Teda $d = 3$

$p[j] == p[j+d]$ pre všetky j ,
 j vlastne neexistuje



Knuth-Morris-Pratt



Ak je porovnávaný prvý prvok vzorového reťazca s prvkom v c, je potrebný posun o 1 znak.

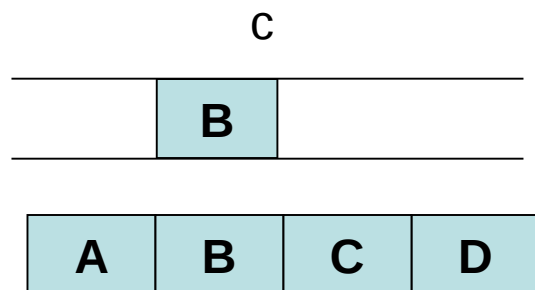
$$P[0] = A \neq B, \text{ teda } d=1$$

Ak je porovnávaný druhý prvok vzorového reťazca s prvkom v c, je potrebný posun o 0 znakov.

$$P[1]=B, \text{ teda } d=0$$



Knuth-Morris-Pratt



Ak je porovnávaný tretí prvok vzorového reťazca s prvkom v c, je potrebný posun o 3 znaky.

$P[2]=C \neq B$, je možný posun o $d=3$

Ak je porovnávaný štvrtý prvok vzorového reťazca s prvkom v c, je potrebný posun o 4 znaky.

$P[3]=D \neq B$, je možný posun o $d=4$

Toto je zaznamenané v druhom riadku matice.



Knuth-Morris-Pratt

- Pre vzorový reťazec XYXYZ:

	X	Y	X	Y	Z
X	0	1	0	3	2
Y	1	0	3	0	5
Z	1	2	3	4	0
iné	1	2	3	4	5



Knuth-Morris-Pratt

Function $KMPSearch(string\ p, t)$: integer

{Nájst' p v t ; vráti jeho pozíciu alebo -1 ak p nie je podreťazcom v t }

$KMPskiparray = \text{Vypočítat' } KMPskiparray(p)$

$K = 0$;

$i = 0$;

While $k < Length(t) - Length(p)$ do {

 if $i == Length(p)$ then return k

$d = KMPskiparray[l, t[k+i]]$;

$k = k + d$;

$i = i + 1 - d$;

}

Return -1;



Knuth-Morris-Pratt

$KMPskiparray = \text{Vypočítať } KMPskiparray(p)$

- Tento príkaz nám nevysvetľuje, ako možno pole $KMPskiparray$ vypočítať.



Boyer-Moore Algoritmus

- Podobný KMP v tomto:
 - Vzorový reťazec je porovnávaný s cieľovým
 - Pri nezhode je posun doprava čo najďalej
- Rozdielny od KMP v tomto:
 - Porovnáva vzory sprava doľava namiesto zľava doprava
- Spôsobí to rozdiel?
 - Áno!! - oveľa rýchlejší na veľkých textových reťazcoch; mnoho znakov v cieľovom reťazci nemusí byť odskúšaných



Boyer-Moore príklad

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

N

Neexistuje žiadne E vo vzore : teda vzor sa nemôže zhodovať so žiadnym znakom ležiacim pod $t[3]$. Takže je možný posun o 4 znaky napravo.



Boyer-Moore príklad

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N

Znovu, žiadna zhoda. Ale B je vo vzore. Takže je možný posun o 2 znaky napravo.



Boyer-Moore príklad

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

Y Y Y Y



Boyer-Moore príklad

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

1
a b a c a b

4 3 2
a b a c a b

13 12 11 10 9 8
a b a c a b

5
a b a c a b

7
a b a c a b

6
a b a c a b

x	a	b	c	d
	4	5	3	-1



Boyer-Moore: iný príklad

$t[k]$ $t[k+1]$... $t[k+i]$... $t[k+m-1]$

			...		C	E	...	R	G	
--	--	--	-----	--	---	---	-----	---	---	--

$p[0]$ $p[1]$... $p[i-1]$ $p[i]$ $p[i+1]$... $p[m-1]$

L	E	...	S	D	E	...	R	G
---	---	-----	---	---	---	-----	---	---

N Y Y Y Y

Problém: určiť d , počet znakov, o ktoré môže byť vzor posunutý napravo

d by malo byť najmenšie také, že $t[k+m-1] = p[m-1-d]$,
 $t[k+m-2] = p[m-2-d]$, ... $t[k+i] = p[i-d]$



Boyer-Moore Algoritmus

Vieme:

- **d by malo byť najmenšie také, že:**
 - $t[k+m-1] = p[m-1-d]$
 - $t[k+m-2] = p[m-2-d]$
 - $t[k+i] = p[i-d]$
- **Pripomíname:**
 - k = štartovací index v cieľovom reťazci
 - m = dĺžka vzorového reťazca
 - i = index nezhody
- **Problém: príkaz je vyhovujúci pre $d \leq i$**
 - Potrebujeme zabezpečiť, aby sme neprekročili ľavú stranu vzoru.



Boyer-Moore : iný príklad

 $t[k]$
 $t[k+5]$
 $t[k+8]$

						c	X	Y	Z	
--	--	--	--	--	--	---	---	---	---	--

 $p[0]$
 $p[1]$
 $p[2]$
 $p[3]$
 $p[4]$
 $p[5]$
 $p[6]$
 $p[7]$
 $p[8]$

Y	Z	W	X	Y	Z	X	Y	Z
---	---	---	---	---	---	---	---	---

N

Y

Y

Y

Ak $c == W$, tak d by mohlo byť 3

Ak $c == R$, tak d by mohlo byť 7



BMPSkip

- Nech $m = |p|$
- Pre ľubovoľný znak c a ľubovoľné i také, že $0 \leq i < m$, definujeme **BMPSkip**(p, i, c) takto:
 - Časť vzoru je možné posunúť napravo, keď znaky $i+1$ až $m-1$ vzoru sa zhodujú s odpovedajúcimi znakmi v cieľovom reťazci, ale $p[i]$ sa nezhoduje so znakom c .
- Potom BMPSkip(p, i, c) by mala vrátiť najmenšie d také, že:
 - $p[j] = p[j-d]$ pre všetky j také, že $\max(i+1, d) \leq j \leq m-1$, a
 - $p[i-d] = c$ ak $d \leq i$



Boyer-Moore

- Pre vzor ABCD:

A
nezhodný
znak v
cieľovom
reťazci je
tento →

A

B

C

D

iné

	A	B	C	D
A	-	4	4	3
B	4	-	4	2
C	4	4	-	1
D	4	4	4	-
iné	4	4	4	4

<- ak pozícia vo vzore je tento znak

Potom je možné vynechať toľko ...



Boyer-Moore

- Pre vzor XYXYZ:

		X	Y	X	Y	Z
A nezhodný znak v cieľovom reťazci je tento →	X	-	5	-	5	2
	Y	5	-	5	-	1
	Z	5	5	5	5	-
	iné	5	5	5	5	5

<- ak pozícia vo vzore je tento znak

Potom je možné vynechať toľko ...



Poznámka:

- Hodnoty v Boyer-Moore poliach sú vo všeobecnosti väčšie než pri KMP; teda vzor sa môže posúvať rýchlejšie



BMSearch

Function *BMSearch*(string *p*, *t*): int

{Nájde *p* v *t*; vráti jeho pozíciu alebo -1 ak *p* sa v *t* nenachádza}

BMSkiparray = **Vypočítať *BMSkipArray*(*p*)**

k = 0

while *k* ≤ *Length*(*t*) − *Length*(*p*) do {

i = *Length*(*p*) − 1

 while *i* ≥ 0 and *p*[*i*] = *t*[*k*+*i*] do

i = *i* − 1;

 if *i* = -1 then return *k*

k = *k* + *BMSkiparray*[*i*, *t*[*k*+*i*]]

}

return -1



BMSearch

BMSkiparray = Vypočítať BMSkipArray(p)

- Tento príkaz nevysvetľuje, ako vypočítať dvojrozmerné pole BMSkiparray
- Pokúste sa hľadať nejaké súvislosti, ktoré by viedli k tomu, aby nebolo nutné počítať toto pole.

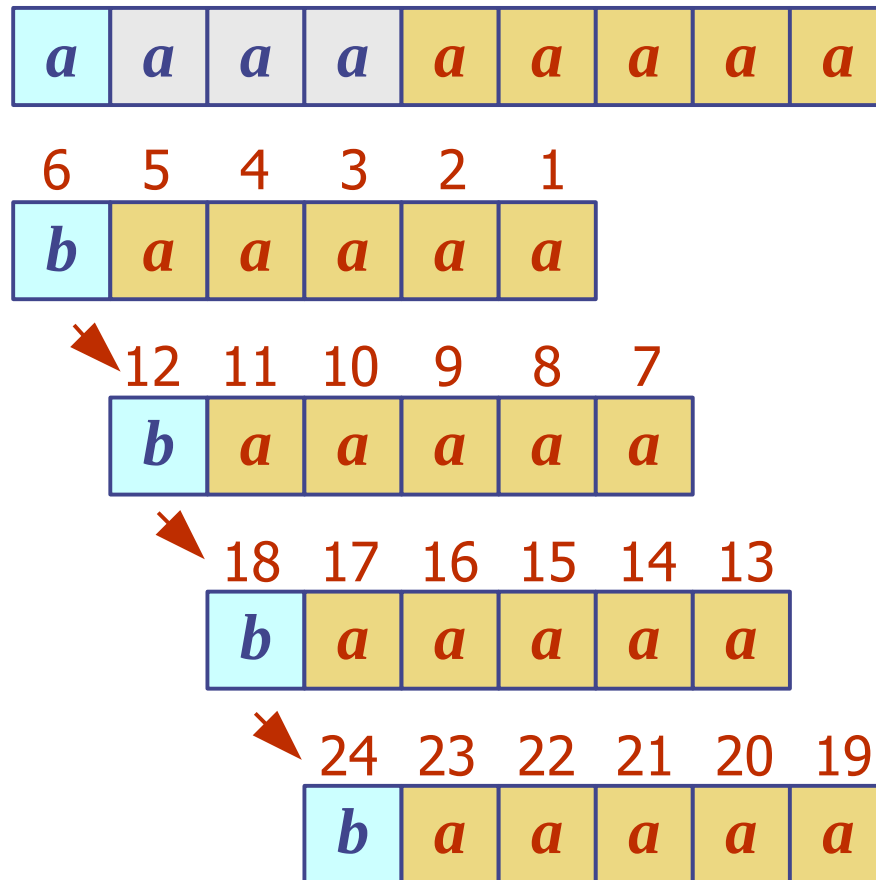


Zložitosť BM algoritmu

- Boyer-Moore algoritmus pracuje v najhoršom prípade v čase $O(nm + s)$
- Príklad najhoršieho prípadu:
 - $T = aaa \dots a$
 - $P = baaa$
- Najhorší prípad sa môže vyskytnúť v obrázkoch alebo v DNA sekvenciách. V obyčajných textoch je málo pravdepodobný.
- Boyer-Moore algoritmus je významne rýchlejší ako brute force algoritmus na anglických textoch.



BM algoritmus – najhorší prípad





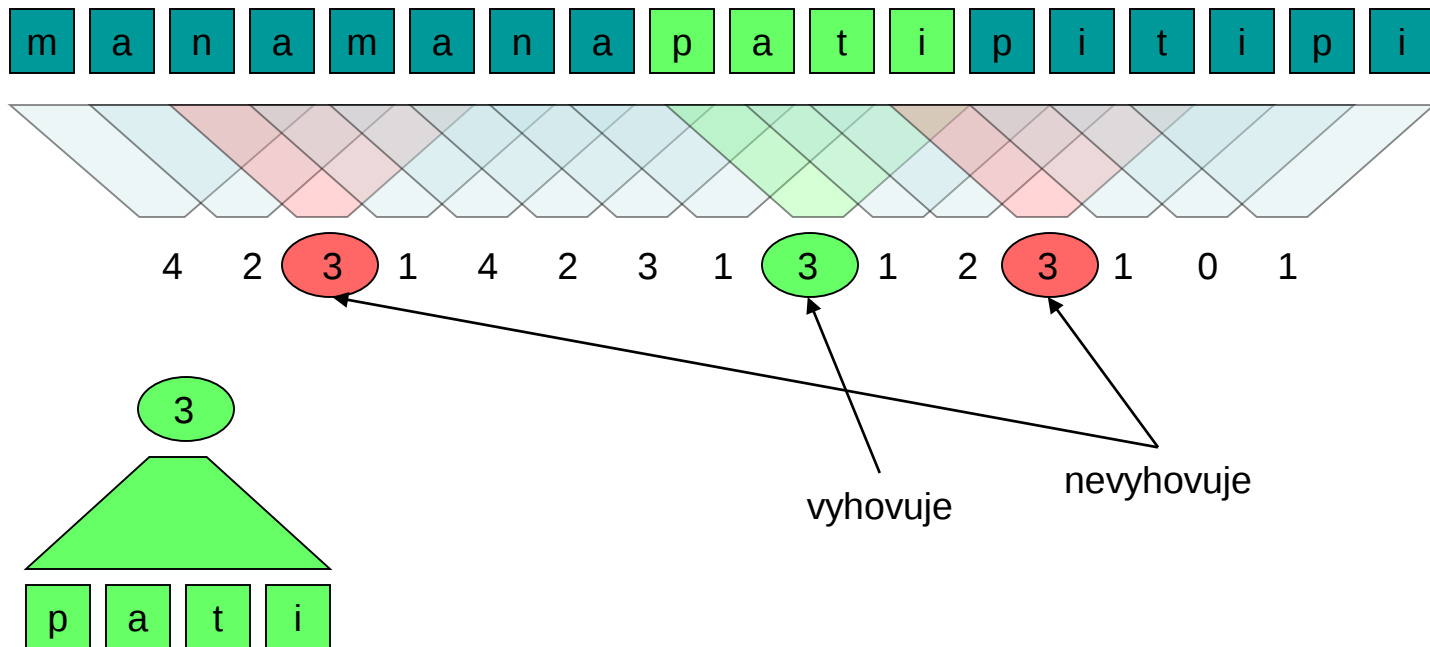
Idea Karp-Rabinovho algoritmu

- Karp & Rabin našli algoritmus, ktorý je:
 - Skoro taký rýchly ako Boyer-Moore
 - Dost' jednoduchý na pochopenie
 - Môže byť použitý na hľadanie v 2-dimenzionálnych cieľových obrázkoch
- Podme naspäť k brute force idee, ale teraz použijeme na jediné číslo reprezentáciu slova, ktoré hľadáme, a jediné číslo pre bežnú časť dokumentu, v ktorom hľadáme, s ktorou práve porovnávame.



Rabin-Karp algoritmus

- Idea: Vypočítat'
 - Kontrolný súčet pre vzor P a
 - Kontrolný súčet pre každý podreťazec T dĺžky m





Rabin - Karp algoritmus

- Výpočet kontrolných súčtov:
 - Vyberieme prvočíslo q , $B=10$
 - Nech $d = |\Sigma|$
 - Example:
 - Nech $d = 10$, $q = 13$, $\Sigma = \{ a, b, c, d, e, f, g, h, i, j \}$
 - Nech $P = \text{'albf'} = \text{'0815'}$
- $$S_4(0815) = (0 \cdot 1000 + 8 \cdot 100 + 1 \cdot 10 + 5 \cdot 1) \bmod 13 =$$

$$815 \bmod 13 = 9$$
- $$T = 1708650854108151$$



Rabin- Karp algoritmus

Rabin-Karp-Matcher(T,P,d,q)

1. $n \leftarrow \text{length}(T)$
2. $m \leftarrow \text{length}(P)$
3. $h \leftarrow d^{m-1} \bmod q$
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. for $i \leftarrow 1$ to m do
7. $p \leftarrow (d \cdot p + P[i]) \bmod q$
8. $t_0 \leftarrow (d \cdot t_0 + T[i]) \bmod q$
- od
9. for $s \leftarrow 0$ to $n-m$ do
10. if $p = t_s$ then
11. if $P[1..m] = T[s+1..s+m]$ then return "Pattern occurs with shift" s
- fi
12. if $s < n-m$ then
13. $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$
- fi
- od



Karp-Rabin Algoritmus

- Predpokladajme, že hľadáme 4-znakové slovo. Potom celé slovo (zakódované ako číslo) sa musí zhodovať so slovom **w** dĺžky 4 byty.
- Ak priebežné 4 byty dokumentu sú považované za jedno slovo **d**, jediné porovnanie môže vyjadriť, či sa rovnajú alebo nie.
- Posunúť pozdĺž dokumentu, v **d** pridať ďalší znak a odstrániť prvý znak.



Karp-Rabin Algoritmus

- Pre dlhšie slová použiť hashing. Znaky slova a dokumentu sú kombinované do jednoduchých hashových čísel **wh** a **dh**. Hašové číslo **dh** môže byť upravené pomocou vhodných matematických operácií v kóde pri pridaní ďalšieho znaku.



Rabin-Karp príklad

- Text „ahojrabinakarpakosamate“
- Hľadáme výraz „karp“
- Pre našu hashovaciu funkciu zvolíme $B=64$ a $P=113$
- P – Prvočíslo, B – Mocnina dvojky
- karp = 107, 97, 114, 112 (hodnoty znakov v ASCII)
- Text v ASCII :
- 97, 104, 111, 106, 114, 97, 98, 105, 110, 97, 107, 97, 114, 112, ...
- $\text{Kod}(\text{karp}) = (107 \cdot 64^3 + 97 \cdot 64^2 + 114 \cdot 64^1 + 112 \cdot 64^0)$
- $= 28\,454\,128$



Rabin-Karp príklad

- $H(\text{karp}) = \text{kod}(\text{karp}) \bmod 113 = 50$
- $T = \text{ahojrabinakarpakosamate}$
- $\text{kod}(\text{ahoj}) = 97 * 64^3 + 104 * 64^2 + 111 * 64 + 106 =$
- $25427968 + 425984 + 7104 + 106 = 25861162$
- $H(\text{ahoj}) =$
- $\text{Kod}(\text{hojr}) = (\text{kod}(\text{ahoj}) - 97 * 64^3) * 64 + 106 = \dots$



Odkazy na texty

- <http://www-igm.univ-mlv.fr/~lecroq/string/>
- <http://www.dreamincode.net/forums/topic/273377-knuth-morris-prattalgorithm%3B-reposted>
- http://pixelstech.net/article/1330941936_Overlap_Detection
- <https://www.youtube.com/watch?v=2ogqPWJSftE>
- <https://www.ics.uci.edu/~eppstein/161/960227.html>
- <http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm>



Pod'akovanie

Ďakujem za pozornosť

