Sufixové pole

V predchádzajúcej kapitole sme videli, že sufixové stromy dokážu riešiť ohromné množstvo úloh nad reťazcami aj množinami reťazcov – rýchlo, elegantne a často v lineárnom čase. Ich slabinou je však pamäťová náročnosť. Aj pri veľmi úspornej implementácii zaberajú typicky 10–20 bajtov na znak a pri textoch dlhých miliardy znakov je to jednoducho priveľa.

Pozrime sa na konkrétny príklad. Ľudský genóm má približne 3 miliardy (3×10^9) báz – znakov zo štvorpísmenovej abecedy `A`, `C`, `G`, `T`. Ak si ho uložíme po 1 bajte na znak, zaberá zhruba 3 GB. Pri zhustenej reprezentácii (2 bity na znak) dokonca iba okolo 750 MB.

Ak však nad týmto reťazcom vybudujeme sufixový strom, výsledná štruktúra bude zvyčajne potrebovať 30–60 GB pamäte, teda až približne 80× viac než samotný text. A teraz si predstavte, že chceme naraz analyzovať desiatky či stovky genómov... Pri podobných vstupoch nás veľmi rýchlo začne limitovať veľkosť RAM.

Samozrejme, vďaka virtuálnej pamäti vieme pracovať aj s oveľa väčšími dátami, no ak RAM zaplníme, systém začne stránky odkladať na disk (*page swapping*). Pri ďalšom použití ich bude musieť opäť načítať, zatiaľ čo iné stránky vysunie na disk, aby uvoľnil miesto. Toto neustále presúvanie je extrémne drahé.

Vysoká pamäťová náročnosť tak prirodzene viedla výskumníkov k hľadaniu úspornejších alternatív.

Vráťme sa k úvodným úlohám z kapitoly o sufixových stromoch. Tam sme videli, že úlohy na prefixoch vieme riešiť pomocou písmenkových stromov (trie), ale tesne na druhom mieste bolo jednoduché riešenie: *zotriediť reťazce lexikograficky*. A tak, podobne ako sufixový strom je „písmenkový strom zo všetkých sufixov“, myšlienka sufixového poľa je jednoduchá: vezmime všetky sufixy, zotrieďme ich podľa abecedy a uložme si výsledné poradie.

Napríklad, ak zotriedime všetky sufixy slova `MISSISSIPPI$`, dostaneme:

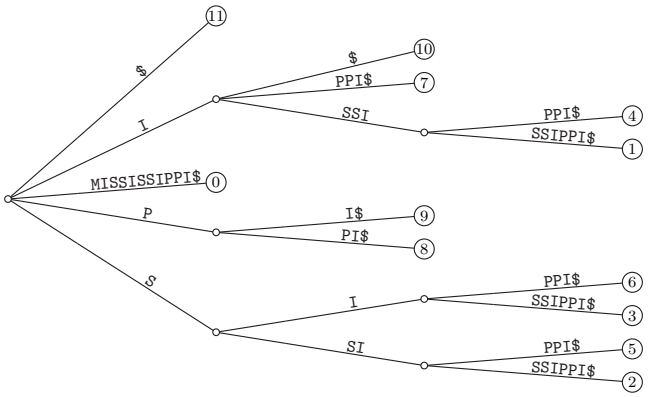
```
11  $
10  I$
 7  IPPI$
 4  ISSIPPI$
 1  ISSISSIPPI$
 0  MISSISSIPPI$
 9  PI$
 8  PPI$
 6  SIPPI$
 3  SISSIPPI$
 5  SSIPPI$
 2  SSISSIPPI$
```

Samozrejme, nebudeme si pamätať kópie všetkých n sufixov explicitne – to by zaberalo $\Theta(n^2)$ pamäte. Úplne stačí pamätať si pôvodný reťazec a každý sufix reprezentovať jedným číslom: indexom jeho začiatku. Sufixové pole je preto jednoducho pole n celých čísel – presne ľavý stĺpec v príklade vyššie.

Pamäťová zložitosť sufixového poľa je teda oveľa lepšia: potrebujeme len jedno celé číslo na každý znak textu. V našom príklade s 3 miliardovým DNA reťazcom nám stačia 32-bitové indexy, takže celé sufixové pole zaberie približne 12 GB (plus pôvodný reťazec, ktorý pri 2 bitoch na znak zaberie asi 0.75 GB). To je obrovský rozdiel oproti 60 GB sufixovému stromu.

Vzťah sufixových stromov a sufixových polí

Hoci sufixové pole (*suffix array*, SA) pôsobí oveľa jednoduchšie než plnohodnotný sufixový strom, v skutočnosti sú tieto dve štruktúry takmer ekvivalentné. Väčšinu aplikácií sufixových stromov vieme s pomocou sufixových polí vyriešiť tiež, ak si k nemu doplníme ešte jedno pole: *LCP pole* (Longest Common Prefix), ktoré ku každému dvojici *susedných* sufixov v sufixovom poli uchováva dĺžku ich najdlhšieho spoločného prefixu.



LCP	*SA*	*sufixy*
0	11	`$`
1	10	`I$`
1	7	`IPPI$`
4	4	`ISSIPPI$`
0	1	`ISSISSIPPI$`
0	0	`MISSISSIPPI$`
1	9	`PI$`
0	8	`PPI$`
2	6	`SIPPI$`
1	3	`SISSIPPI$`
3	5	`SSIPPI$`
–	2	`SSISSIPPI$`

Obr. 1: Vľavo sufixový strom, vpravo sufixové pole a LCP pole pre reťazec `MISSISSIPPI$`.

Ak si porovnáme sufixový strom a sufixové pole spolu s LCP poľom, ukáže sa, že:

- sufixové pole aj LCP vieme odvodiť zo sufixového stromu v lineárnom čase: všimnite si, že ak máme v každom vrchole stromu zotriedené znaky, tak prechodom listov zhora nadol dostaneme utriedenú postupnosť sufixov; a ak si pri prehľadávaní budeme pamätať textovú hĺbku, vieme zároveň vypisovať hodnoty LCP (ako vysoko sme sa pri prehľadávaní museli vrátiť, než sme sa zanorili do ďalšieho podstromu);
- naopak, aj sufixový strom sa dá zrekonštruovať z SA+LCP v lineárnom čase: stačí strom začať budovať zhora nadol; LCP pole nám prezradí, do akej hĺbky sa máme vrátiť, a kde máme odpojiť novú vetvu;

- LCP pole zodpovedá textovým hĺbkam vnútorných vrcholov; napríklad vrchol na ceste „`ISSI`“ sa nachádza v kompaktnom sufixovom strome, pretože je to najdlhší spoločný začiatok sufixov `ISSIPPI$` a `ISSISSIPPI$`; po štvrtom písmene sa sufixy oddelia;

- podstromy v sufixovom strome zodpovedajú intervalom v SA; napríklad ak prejdeme cestu „`I`“, všetky výskyty tejto vzorky sú listy daného podstromu: 10, 7, 4, 1; v sufixovom strome to zodpovedá intervalu $1 \ldots 4$ – vďaka triedeniu sa všetky sufixy začínajúce na „`I`“ dostanú vedľa seba.

Vyhľadávanie

Najjednoduchší spôsob, ako pomocou sufixového poľa nájsť, či sa vzorka P nachádza v texte T, je obyčajné binárne vyhľadávanie. V sufixovom poli sú všetky sufixy utriedené lexikograficky, takže všetky sufixy začínajúce na P tvoria jeden súvislý úsek. Stačí teda pomocou binárneho vyhľadávania nájsť ľavú a pravú hranicu tohto úseku.

Binárne vyhľadávanie trvá $O(\log n)$ krokov, avšak každé porovnanie dvoch reťazcov môže trvať až $O(m)$, preto je celková zložitosť

$$O(m \log n).$$

To je horšie než pri sufixových stromoch, kde sme vyhľadávali v čase $O(m)$.

Dá sa to zrýchliť? Ukáže sa, že áno – ak sme ochotní použiť trochu viac pamäte. Kľúčovou pomocnou štruktúrou je pole LCP, kde $\mathrm{lcp}(i, j)$ označuje dĺžku najdlhšieho spoločného prefixu i-teho a j-teho sufixu v sufixovom poli. LCP pole nám umožní pri porovnávaní preskakovať časti reťazcov, ktoré sme už raz porovnali.

Hľadajme vzorku P. Počas binárneho vyhľadávania si budeme udržiavať dva indexy: ℓ a r, pričom bude platiť, že hľadaná vzorka sa nachádza niekde medzi tým, presnejšie:

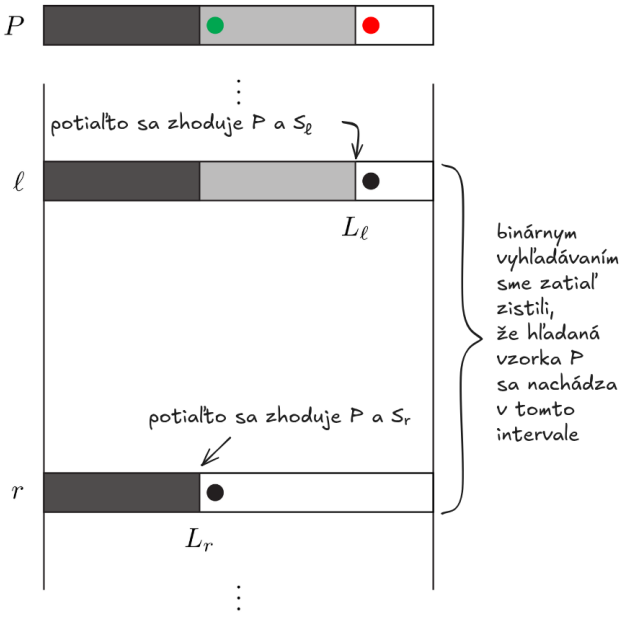
$$S[SA[\ell]] < P \leq S[SA[r]].$$

Keďže povedať $S[SA[i]]$ je celkom nálož, budeme hovoriť jednoducho i-ty sufix (v utriedenom poradí) a značiť ho S_i.

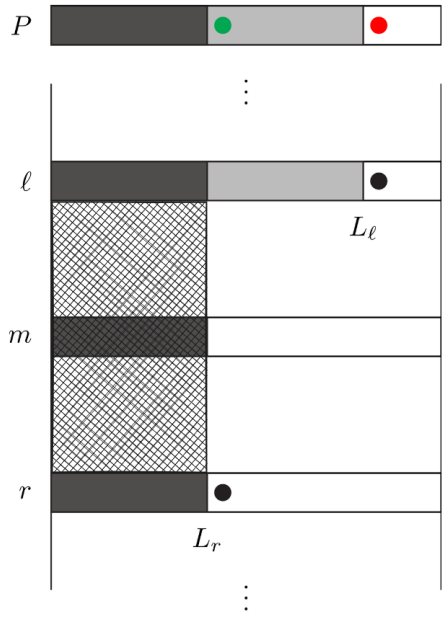
Zároveň si uchovávame dve hodnoty:

$$L_\ell = \mathrm{lcp}(P, S_\ell), \qquad L_r = \mathrm{lcp}(P, S_r),$$

teda dĺžku spoločného prefixu P a sufixu na ľavej a pravej hranici. (Tieto prefixy znázorňujú šedé úseky na obrázku. Za šedou zhodou nasleduje znak, v ktorom sa sufixy líšia – červený znak pri ℓ a zelený pri r.)



Pozrime sa teraz dostredu, na pozíciu m.



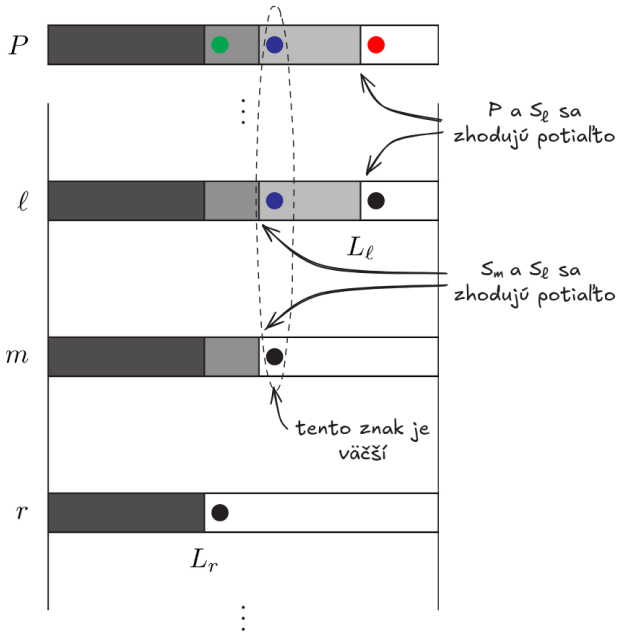
Keďže ℓ-tý a r-tý sa zhodujú s P v prvých $\min(L_\ell, L_r)$ znakoch, aj všetky reťazce medzi nimi (keďže sú zotriedené lexikograficky), sa musia začínať na tie

isté znaky (ako znázorňuje vyšrafovaná oblasť). Tieto znaky už teda nemusíme porovnávať.

Ba čo viac, pozrime sa teraz na $p = \text{lcp}(S_\ell, S_m)$ (bez újmy na všeobecnosti predpokladajme, že $L_\ell \geq L_r$, teda že S_l má dlší spoločný začiatok s hľadanou vzorkou ako S_r – v opačnom prípade budeme postupovať symetricky a pozrieme sa na $p = \text{lcp}(S_r, S_m)$).

Sú tri možné prípady:

Prípad 1: $p < L_\ell$**.** To znamená, že S_ℓ a P majú *dlhší spoločný začiatok* ako S_ℓ s S_m.

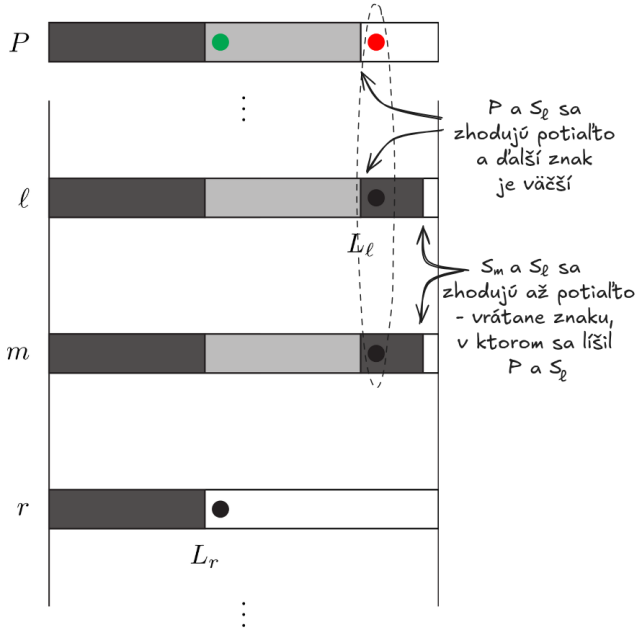


Čo z toho vyplýva? Nuž vyplýva z toho, že po p-ty znak sa P, ℓ-tý, aj m-tý sufix rovnajú, ale $(p+1)$-vý znak má S_m odlišný(!), konkrétne väčší, ako znak, ktorý zdieľajú S_ℓ a P.

Takže $P < S_m$ a treba hľadať v hornej polovici poľa. Zároveň platí, že $L_m = p$.

Všimnite si, čo sa stalo: nemuseli sme porovnať ani jeden znak a zistili sme, ktorou cestou sa v binárnom vyhľadávaní máme vydať.

Prípad 2: $p > L_\ell$**.** To znamená, že S_ℓ s S_m majú *dlhší spoločný začiatok* ako S_ℓ a P.



Čo z toho vyplýva? Nuž, zjavne S_m obsahuje ten istý znak, v ktorom sa líšili S_ℓ od P, takže výsledok porovnania bude ten istý.

Keďže $P > S_\ell$, tak $P > S_m$ a v tomto prípade treba hľadať v dolnej polovici poľa. Zároveň platí, že $L_m = L_\ell$.

Opäť sme zistili, ktorým smerom pokračovať v hľadaní bez toho, aby sme porovnali čo i len jediný znak!

Prípad 3: $p = L_\ell$**.** V tomto prípade nevieme okamžite povedať výsledok, takže začneme porovnávať S_m a P znak po znaku. Samozrejme, prvých p znakov preskočíme. Na koniec podľa výsledku porovnania zistíme, či máme pokračovať v hornej alebo dolnej polovici a zároveň spočítame L_m – aký dlhý je spoločný prefix L_m a P.

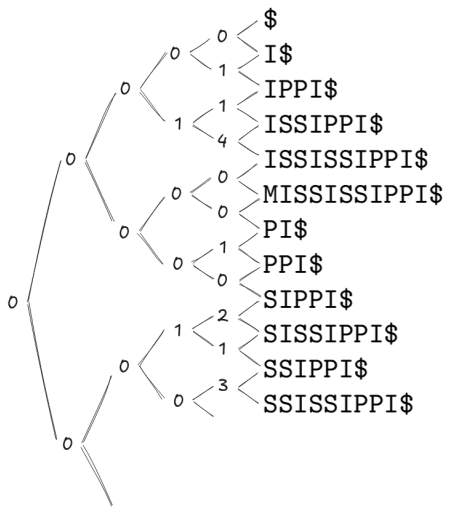
Zložitosť. Predpokladajme, že všetky hodnoty LCP si predpočítame dopredu. Potom prípad 1 a 2 trvá iba konštantný čas, takže všetky tieto kroky spolu trvajú iba $O(\log n)$ času. A koľko trvá prípad 3? Tvrdím, že dokopy za celé vyhľadávanie spravíme len $O(m+\log n)$ porovnaní, pretože sa nikdy nevraciame na pozície, ktoré sme už porovnali; keďže vzorka má dĺžku m, môžeme sa iba m-krát posunúť doprava. Okrem toho, ak nájdeme nezhodu, ostávame na rovnakej pozícii, avšak pri nezhode sa porovnávanie končí, takže týchto krokov je len $\log n$. Výsledná zložitosť je teda

$$O(m + \log n),$$

čo je skoro rovnako dobré ako $O(m)$ pri sufixových stromoch – keďže aj pre texty dlhé miliardy znakov je $\lg n \approx 30$, v praxi je často $m > \log n$ ten dominantný člen.

Na druhej strane, za toto rýchlejšie vyhľadávanie zaplatíme horšou pamäťou. Musíme si predpočítať nielen LCP susedných sufixov, ale aj LCP dvojíc, ktoré sú ďalej od seba.

Našťastie netreba LCP každej dvojice – stačí si prejsť rozhodovací strom pri binárnom vyhľadávaní a zapamätať si iba tie hodnoty, ktoré potrebujeme. Praktické riešenie je (aspoň pomyselne) „zaokrúhliť“ dĺžku sufixového poľa na najbližšiu väčšiu mocninu 2 a vybudovať nad LCP poľom úplný binárny strom. LCP dvoch vzdialenejších sufixov spočítame ako minimum LCP dvoch detí. Binárne vyhľadávanie potom prispôsobíme tak, že začneme s r mocninou dvojky:



Konštrukcia LCP poľa

Pole LCP (Longest Common Prefix) je pre sufixové pole rovnako dôležité, ako bola textová hĺbka pri sufixových stromoch. Definujme

$$\mathrm{lcp}[i] = \mathrm{lcp}(S[\mathrm{SA}[i]],\ S[\mathrm{SA}[i-1]]), \qquad i = 1, \ldots, n,$$

teda dĺžka najdlhšieho spoločného prefixu dvoch po sebe idúcich sufixov v utriedenom poradí. Hodnota lcp[0] sa zvykne definovať ako 0.

Použitím LCP poľa sa mnohé operácie nad sufixovým poľom dramaticky zrýchlia– napríklad zrýchlené binárne vyhľadávanie na $O(m + \log n)$, vyhľadávanie opakujúcich sa podreťazcov, najdlhší opakujúci sa substring a množstvo ďalších úloh.

Otázka znie: *Ako rýchlo LCP pole skonštruovať, ak už máme sufixové pole?*

Ukážeme si klasický algoritmus od Kasaiho, ktorý to zvládne v čase $O(n)$.

Kasaiho algoritmus

Kľúčová myšlienka algoritmu je jednoduchá:

> Ak poznáme $\mathrm{lcp}(i,j)$, potom $\mathrm{lcp}(i+1,j+1)$ je určite *aspoň* $\mathrm{lcp}(i,j)-1$.

Inými slovami: Dva susedné sufixy $T[i\,..]$ a $T[j\,..]$ zdieľajú určitý spoločný prefix dĺžky h. Keď ich „o znak posunieme doprava“ a porovnávame $T[i+1\,..]$ a $T[j+1\,..]$, tak spoločný prefix je dlhý aspoň $h-1$.

Takže ak pri výpočte ďalšieho lcp vždy začneme porovnávať nie od nuly, ale od $h-1$, dohromady vykonáme najviac n porovnaní znakov navyše.

Celý algoritmus funguje v troch krokoch:

1. *Vytvoríme inverzné sufixové pole.* Pre každý index i v texte chceme poznať jeho pozíciu v sufixovom poli:

   $$\mathrm{rank}[i] = \text{pozícia sufixu } T[i..] \text{ v sufixovom poli.}$$

   To vieme zostrojiť v čase $O(n)$.

2. *Prejdeme sufixy v poradí od najdlhšieho po najkratší (nie v utriedenom poradí!).* Teda v poradí $i = 0, 1, 2, \ldots, n-1$. Pomocou `rank[i]` zistíme, ktorý je predchádzajúci sufix v SA:

   $$j = \mathrm{SA}[\,\mathrm{rank}[i]-1\,],$$

   a chceme spočítať $\mathrm{lcp}(i,j)$.

3. *Porovnávame znaky.* Hodnotu h, ktorú si uchovávame medzi iteráciami, vždy najskôr znížime o 1 (ak $h>0$), a potom od tejto pozície ďalej porovnávame znaky, kým sú rovnaké:

   $$\text{while } T[i+h] = T[j+h], \text{ zväčši } h.$$

   Výsledkom je

   $$\mathrm{lcp}[\mathrm{rank}[i]] = h.$$

   Po skončení iterácie prechádzame na $i+1$.

Algoritmus je uvedený nižšie:

```
h = 0
for i = 0 .. n-1:
    if rank[i] = 0:
        LCP[0] = 0
        continue
    j = SA[rank[i] - 1]
    while T[i+h] == T[j+h]: h++
    LCP[rank[i]] = h
    if h > 0: h--
```

Prečo je časová zložitosť lineárna? V celom algoritme sa vykonávajú dva typy prác:

1. Pri každom kroku robíme konštantnú prácu (prístup do polí SA, rank).
2. V cykle `while` porovnávame znaky $T[\cdot]$.

Trik je v tom, že premenná h za celý beh nikdy nenarastie na viac než n. Zároveň len n-krát klesne o 1. Z toho vyplýva, že nemôže stúpnuť viac ako $2n$-krát. Pri každom porovnaní máme buď zhodu (vtedy h stúpne, takže týchto krokov je najviac $2n$), alebo nezhodu (vtedy porovnávanie končí a ideme na ďalšiu dvojicu, takže týchto krokov je najviac n). Celkový čas algoritmu je teda lineárny.

Konštrukcia sufixového poľa

Chceme zotriediť n sufixov. Ako na to?

Quicksort?

Najjednoduchší nápad je uložiť si všetkých n sufixov a zotriediť ich sortom zo štandardnej knižnice. To však znamená triediť reťazce priemernej dĺžky $n/2$, takže celkový čas bude $O(n^2 \log n)$ a pamäť $O(n^2)$. Pre dlhšie texty absolútne nepoužiteľné.

Lepšie riešenie je využiť, že v štandardných knižniciach často vieme na triedenie zadať *vlastnú* porovnávaciu funkciu. Potom stačí triediť indexy $0 \ldots n-1$, akurát pri porovnaní i a j porovnáme sufixy $T[i..]$ a $T[j..]$. Časová zložitosť bude rovnaká, ale pamäť bude lineárna.

Pre niektoré texty (napríklad v prirodzenom jazyku, ktoré nemajú príliš veľa dlhých opakovaní) to môže byť dokonca celkom praktický algoritmus. Ak totiž označíme L ako najdlhší spoločný prefix ľubovoľných dvoch sufixov, potom pri každom porovnaní stačí porovnať len L znakov a teda skutočná zložitosť je $O(L \times n \log n)$. Samozrejme, v najhoršom prípade je $L = n - 1$; zložitosť je stále $O(n^2 \log n)$ v najhoršom prípade, avšak ak je L malé, tento čas môže byť zvládnuteľný.

Dá sa to lepšie?

Radixsort?

Triedime predsa reťazce – na to je vhodnejší radixsort ako quicksort. V najhoršom prípade dostávame zložitosť $O(n^2)$ – pre texty dlhé miliardy znakov stále nepoužiteľné.

Všimnite si, že doteraz sme uvažovali iba všeobecné algoritmy, ktoré dokážu utriediť ľubovoľnú postupnosť reťazcov. Ak chceme lepší algoritmus, musíme využiť, že triedime veľmi špeciálne reťazce – všetky sufixy daného textu.

Manber–Myersov $n \log n$ algoritmus

Kľúčová myšlienka je prekvapivo jednoduchá:

Sufix sufixu je opäť sufix.

Ak máme sufixy utriedené podľa prvých K znakov, vieme ich veľmi jednoducho zotriediť podľa prvých $2K$ znakov. To nám umožní „zdvojnásobiť“ počet porovnaných znakov v každej fáze.

Algoritmus pracuje v $\log n$ fázach. V k-tej fáze budeme mať sufixy zotriedené podľa prvých 2^k znakov.

Označme:

$$r[i] = \text{poradové číslo sufixu } s_i \text{ v triedení podľa prvých } 2^k \text{ znakov.}$$

Ak majú dva rôzne sufixy rovnaký prefix dĺžky 2^k, ich ranku sa priradí rovnaká hodnota. Takto stačí v každej fáze zotriediť nie celé reťazce, ale len trojice čísel:

$$(r[i],\ r[i + 2^k],\ i),$$

kde:

- $r[i]$ určuje poradie podľa prvých 2^k znakov,
- $r[i + 2^k]$ určuje poradie „druhej polovice prefixu“,
- i je samotná pozícia sufixu.

Na začiatku (pre $k = 0$) sufixy triedime podľa prvého znaku – stačí jednoducho utriediť znaky abecedy.

V každej ďalšej fáze potom všetky sufixy zotriedime podľa dvojice $(r[i], r[i + 2^k])$.

Ak použijeme quicksort, jedna fáza bude trvať $O(n \log n)$, takže celkovo bude konštrukcia trvať maximálne $O(n \log^2 n)$ času.

Avšak keďže triedime celé čísla v rozsahu $0 \ldots n$, môžeme použiť radix sort a implementovať jednu fázu v lineárnom čase. Celkový čas potom bude $O(n \log n)$.

Zopár praktických vylepšení:

1. V prvej fáze nemusíme triediť podľa jediného písmena. Namiesto toho sufixy pred-triedime podľa prvých povedzme 64 alebo 128 písmen.
2. Nemusíme dokončiť všetkých $\lg n$ fáz – ak už sú všetky sufixy rozlíšené a jednoznačne zotriedené, môžeme skončiť.
3. Môžeme si pamätať úseky s rovnakým rankom a sústrediť sa len na ich triedenie.

Kärkkäinenov–Sandersov lineárny algoritmus

2003 bol dobrý rok. Dovoľte malú historickú vsuvku. Sufixové stromy vynašiel Peter Weiner už v roku 1973 a zároveň predstavil aj prvý lineárny algoritmus na ich konštrukciu. McCreight (1976) a neskôr Ukkonen (1995) objavili jednoduchšie a praktickejšie lineárne algoritmy. Tieto riešenia však ticho predpokladali, že veľkosť abecedy je konštantná. Až Farach (1997) publikoval prvý algoritmus, ktorý bol optimálny pre ľubovoľnú abecedu (kde „znaky“ môžu byť napríklad čísla v rozsahu 1 až $n^{O(1)}$).

Sufixové polia ako pamäťovo úspornejšiu alternatívu sufixových stromov predstavili Manber a Myers v roku 1990. Pomerne dlho však nikto nepoznal *priamu* lineárnu konštrukciu sufixového poľa – priamu v zmysle, že by nebolo potrebné najskôr zostrojiť celý sufixový strom a až z neho odvodiť pole.

A potom prišiel rok 2003.

V tom istom roku vyšli *tri* nezávislé články, ktoré po viac ako desaťročí priniesli *tri rôzne* lineárne algoritmy na konštrukciu sufixových polí.

V tejto kapitole si ukážeme algoritmus Kärkkäinena a Sandersa (často sa označuje ako *skew* algoritmus[1] alebo tiež DC3 (podľa použitého „difference cover“ modulo 3).

Začnime myšlienkou, ktorá pochádza z Farachovho algoritmu na konštrukciu sufixových stromov. Bez toho, aby sme zachádzali do detailov, Farach rozdelil všetky sufixy na dve skupiny: tie, ktoré začínajú na *párnych* pozíciách, a tie, ktoré začínajú na *nepárnych* pozíciách.

Najskôr si rekurzívne zostrojil sufixový strom pre sufixy na nepárnych indexoch. Z tohto stromu potom dokázal odvodiť poradie sufixov na párnych pozíciách. Nakoniec obe množiny zlúčil a získal kompletný sufixový strom pre celý text.

Skúsme podobný prístup použiť pri konštrukcii sufixového poľa. Vezmime si ako príklad reťazec `MISSISSIPPI$`. Predstavme si, že sa nám už podarilo zotriediť všetky sufixy začínajúce na *nepárnych* pozíciách:

```
11  $
 7  IPPI$
 1  ISSISSIPPI$
 9  PI$
 3  SISSIPPI$
 5  SSIPPI$
```

Ako teraz zotriediť sufixy na *párnych* pozíciách?

```
 0  MISSISSIPPI$
 2  SSISSIPPI$
 4  ISSIPPI$
 6  SIPPI$
 8  PPI$
10  I$
```

[1] “skew” by sme mohli preložiť ako „asymetrický“, keďže algoritmus pracuje s asymetrickým rozdelením indexov.

Kľúčové pozorovanie: každý párny sufix pozostáva z *prvého písmena* a za ním nasleduje *nepárny sufix* – a tieto nepárne sufixy už máme zotriedené! Napríklad:

- `SIPPI$` začína písmenom `S` a za ním nasleduje sufix 7, ktorý je v utriedenom poradí druhý;
- `SSISSIPPI$` začína písmenom `S`, ale za ním nasleduje sufix 3, ktorý je v poradí až piaty.

Takže každý párny sufix môžeme reprezentovať dvojicou:

(prvé písmeno, poradie nasledujúceho nepárneho sufixu).

Pre náš príklad to vyzerá takto:

0	(`M`, 3)
2	(`S`, 5)
4	(`I`, 6)
6	(`S`, 2)
8	(`P`, 4)
10	(`I`, 1)

Ak už teda poznáme poradie nepárnych sufixov, poradie párnych sufixov dokážeme získať pomocou jednoduchého dvojfázového radix sortu podľa tejto dvojice:

10	(`I`, 1)	`I$`
4	(`I`, 6)	`ISSIPPI$`
0	(`M`, 3)	`MISSISSIPPI$`
8	(`P`, 4)	`PPI$`
6	(`S`, 2)	`SIPPI$`
2	(`S`, 5)	`SSISSIPPI$`

Ako však dostaneme poradenie nepárnych sufixov?

Rekurzívne. Použijeme nasledovný trik: Zoberieme náš reťazec, odhodíme jeho prvé písmeno a zvyšok si rozdelíme na dvojice. Každú dvojicu budeme považovať za jeden nedeliteľný „super-znak“:

`IS`	`SI`	`SS`	`IP`	`PI`	`$$`

Na takto vytvorený reťazec sa teraz pozeráme ako na nový reťazec zo šiestich znakov. Všimnite si, že jeho sufixy zodpovedajú práve nepárnym sufixom pôvodného reťazca. Ak teda dokážeme zotriediť sufixy tohto „zbaleného“ reťazca, dostaneme poradie všetkých nepárnych sufixov pôvodného textu.

Nakoniec nám ostáva už „iba“ jeden krok: zobrať dve utriedené sufixové polia – jedno s nepárnymi sufixami a druhé s párnymi – a zlúčiť ich do jedného spoločného, úplne utriedeného sufixového poľa.

Ukazuje sa však, že toto zlučovanie vôbec nie je také jednoduché, ako by sa mohlo zdať. Hoci existuje riešenie v lineárnom čase (vďaka práci Kim et al.), technicky je pomerne komplikované.

A tu prichádza elegantný trik algoritmu DC3. Namiesto delenia sufixov na párne a nepárne spravíme rozdelenie *asymetricky*: na sufixy začínajúce na indexoch deliteľných tromi (približne tretina všetkých sufixov) a na sufixy začínajúce na indexoch nedeliteľných tromi (zvyšné dve tretiny).

Predstavme si, že sufixy na pozíciách $i \not\equiv 0 \pmod 3$ už máme zotriedené:

```
11  $
10  I$
 7  IPPI$
 4  ISSIPPI$
 1  ISSISSIPPI$
 8  PPI$
 5  SSIPPI$
 2  SSISSIPPI$
```

Ako zotriediť zostávajúce sufixy, teda tie na pozíciách $i \equiv 0 \pmod 3$? Rovnakým spôsobom ako v predošlom prístupe: pre každý taký sufix si vytvoríme dvojicu

$$(\text{prvé písmeno}, \text{pozícia o 1 kratšieho sufixu v už utriedenom poli})$$

a tieto dvojice zotriedime radix sortom:

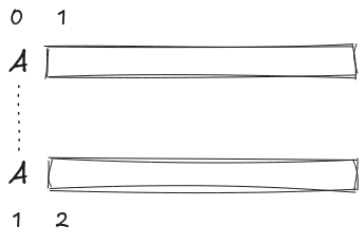
```
0  MISSISSIPPI$  (M, 5)
3  SISSIPPI$     (S, 4)
6  SIPPI$        (S, 3)
9  PI$           (P, 2)
```

Zotriedením týchto dvojíc dostaneme poradie sufixov 0, 9, 6, 3.

Teraz máme dve utriedené sufixové polia a potrebujeme ich zlúčiť do jedného kompletného poradia všetkých sufixov. Ako na to?

Utriedené polia zlučujeme podobne ako v mergesorte. Kľúčové je vedieť rýchlo porovnať sufix s indexom $i \equiv 0 \pmod 3$ voči sufixu s indexom $j \equiv 1$ alebo $2 \pmod 3$.

Prípad 0 vs. 1 modulo 3. Ak porovnávame sufixy začínajúce na indexoch $i \equiv 0 \pmod 3$ a $j \equiv 1 \pmod 3$, stačí porovnať ich prvé písmená. Ak sú rovnaké, posúvame sa o jeden znak ďalej.

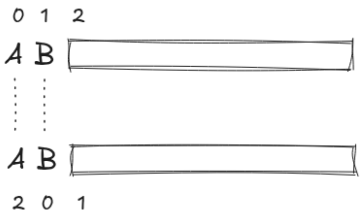


Zvyšok týchto dvoch sufixov začína na indexoch

$$i + 1 \equiv 1 \pmod 3, \qquad j + 1 \equiv 2 \pmod 3,$$

teda oba patria medzi indexy nedeliteľné tromi a ich vzájomné poradie už poznáme!

Prípad 0 vs. 2 modulo 3. Podobne, ak porovnávame sufixy na indexoch $i \equiv 0 \pmod 3$ a $j \equiv 2 \pmod 3$, najskôr porovnáme prvé písmeno. Ak sú rovnaké, porovnáme druhé písmeno.



Ak aj tie sú rovnaké, zvyšné časti sufixov začínajú na indexoch

$$i + 2 \equiv 2 \pmod 3, \qquad j + 2 \equiv 1 \pmod 3,$$

a teda opäť sme sa dostali na dva indexy nedeliteľné tromi, ktorých vzájomné poriadie už poznáme.

Tu vidíme, prečo sa oplatilo rozdeľovať úlohu *asymetricky*. V oboch prípadoch sa po jednom až dvoch krokoch presunieme na indexy *nedeliteľné tromi*. Tieto indexy patria do jednej spoločnej množiny, ktorú už máme zotriedenú. Vďaka tomu dokážeme rozhodnúť výsledok každého porovnania v konštantnom čase a zlučovanie oboch utriedených polí prebehne jednoduchým lineárnym prechodom.

Ako spočítame sufixové pole pre indexy nedeliteľné tromi?

Opäť použijeme podobný trik ako v predchádzajúcom prístupe. Zoberieme náš pôvodný reťazec od indexu 1 (teda $T[1 \ldots n]$), prípadne ho na konci doplníme znakmi `$`, aby mal dĺžku deliteľnú tromi. Zaň zapíšeme ešte jednu kópiu reťazca od indexu 2 (teda $T[2 \ldots n]$, opäť doplnenú `$` podľa potreby). Spolu tak dostaneme nový text dĺžky približne $2n$.

Tento nový reťazec si následne rozdelíme na postupné trojice znakov. Každú trojicu budeme považovať za jeden nedeliteľný „super-znak“:

0	1	2	3	4	5	6	7
`ISS`	`ISS`	`IPP`	`I$$`	`SSI`	`SSI`	`PPI`	`$$$`

Dĺžka tohto reťazca je teda $\approx 2n/3$.

Všimnite si, že sufixy z prvej polovice tohto nového reťazca presne zodpovedajú sufixom, ktoré začínajú na indexoch so zvyškom 1 (mod 3). Rovnako sufixy z druhej polovice zodpovedajú sufixom na indexoch so zvyškom 2 (mod 3).

Presnejšie, ak si reťazec porovnáme s tým pôvodným,

```
0 1 2 3 4 5 6 7 8 9 10 11
M I S S I S S I P P I  $
```

vidíme, že sufixy 0, 1, 2, 3 z prvej polovice zodpovedajú indexom 1, 4, 7, 10 a sufixy 4, 5, 6, 7 z druhej polovice zodpovedajú indexom 2, 5, 8, 11 v pôvodnom reťazci.

(Sufixy v prvej polovici síce obsahujú „čosi navyše“ na konci – druhú kópiu textu – ale z hľadiska poradia a porovnávania je všetko za znakmi $ irelevantné.)

Ak teda rekurzívne spočítame sufixové pole

7	$$$							
3	I$$	SSI	SSI	PPI	$$$			
2	IPP	I$$	SSI	SSI	PPI	$$$		
1	ISS	IPP	I$$	SSI	SSI	PPI	$$$	
0	ISS	ISS	IPP	I$$	SSI	SSI	PPI	$$$
6	PPI	$$$						
5	SSI	PPI	$$$					
4	SSI	SSI	PPI	$$$				

vieme z neho spätne zrekonštruovať pôvodné indexy a sufixové pole pre indexy nedeliteľné tromi:

11	$
10	I$
7	IPPI$
4	ISSIPPI$
1	ISSISSIPPI$
8	PPI$
5	SSIPPI$
2	SSISSIPPI$

No moment – a nie je to celé podvod?

To sa naozaj môžeme len tak rozhodnúť, že trojica znakov bude jeden „super-znak“ a tváriť sa, že nový reťazec má iba tretinovú dĺžku?

Pôvodne mal text abecedu veľkosti σ. Trojice znakov však pochádzajú z množiny všetkých trojíc Trojice znakov však pochádzajú z množiny všetkých trojíc $\Sigma^3 = \{(a, b, c) : a, b, c \in \Sigma\}$, ktorá má σ^3 prvkov. A keď sa v rekurzii zanoríme ešte o krok hlbšie, budeme pracovať s abecedou veľkosti $(\sigma^3)^3 = \sigma^9$, v ďalšom volaní dokonca $((\sigma^3)^3)^3 = \sigma^{27}$ „super-super-super znakov“.

Veľkosť abecedy nám teda rastie prudko exponenciálne a tieto „super-znaky“ sú čoraz zložitejšie objekty. Tradične pritom predpokladáme, že dva znaky vieme porovnať v konštantnom čase. Ale ako máme porovnávať tieto „super-$\cdots$-super“ znaky?

Našťastie má tento problém jednoduché riešenie:

> Každý reťazec dĺžky n môže obsahovať najviac len n rôznych znakov.

Inými slovami: aj keby bola pôvodná abeceda obrovská, v konkrétnom vstupe sa reálne vyskytne najviac n rôznych symbolov. A presne to isté platí aj pre naše „super-znaky“: hoci teoreticky pochádzajú z veľkej množiny Σ^3, v skutočnosti sa v celom zreťazenom reťazci objaví nanajvýš n rôznych trojíc.

Namiesto toho, aby sme pracovali s celou abecedou veľkosti σ^3, spravíme jednoduchú vec: všetky trojice, ktoré sa v texte objavia, *zotriedime* (ako?) a v tomto poradí im priradíme nové mená – čísla $0, 1, 2, \ldots$

Tým:

- zachováme správne poradie sufixov (triedenie trojíc je stabilné),
- veľkosť abecedy sa v každom kroku rekurzie zmenší na najviac n, takže pri rekurzii exponenciálne nevybuchne,
- dva „super-znaky“ budeme vedieť porovnať v konštantnom čase (iba porovnaním ich čísel),