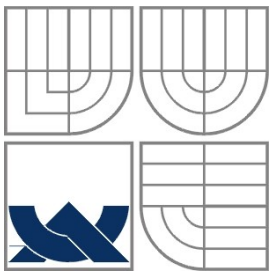
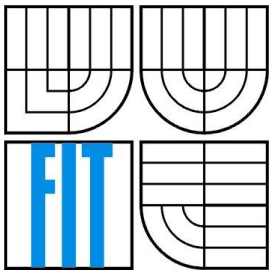


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KONSTRUKCE SUFIXOVÝCH POLÍ A JEJICH VYUŽITÍ V BIOINFORMATICE

SUFFIX ARRAYS CONSTRUCTION AND THEIR USE IN BIOINFORMATICS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ HLÁVKA

VEDOUCÍ PRÁCE
SUPERVISOR

ING. TOMÁŠ MARTÍNEK PH.D.

BRNO 2011

Abstrakt

Práce pojednává o perspektivní datové struktuře, která se nazývá sufixové pole. Tato datová struktura je zde podrobněji popsána a v práci je dále uvedeno rozdělení algoritmů pro konstrukci tohoto pole. Je zde popsáno několik konstrukčních algoritmů a nejpodrobněji se práce zaobírá algoritmem nazývaným qsufsort. Nakonec si ukážeme využití sufixového pole pro vyhledávání přesných (pomocí binárního vyhledávání) a přibližných (metoda QUASAR) vzorů v sekvencích DNA.

Abstract

This work describes perspective data structure called suffix array. This data structure is described in more detail and this paper also contains taxonomy of suffix array construction algorithms. A few algorithms are described more precisely and most space is devoted to algorithm called qsufsort. Finally we will show how can be suffix array used in practice. This work shows usage of suffix array in exact (binary search) and approximate (QUASAR) string matching in DNA sequences.

Klíčová slova

Sufixové pole, sufix, přesné vyhledávání, přibližné vyhledávání, QUASAR, qsufsort

Keywords

Suffix array, suffix, exact matching, approximate matching, QUASAR, qsufsort

Citace

Ondřej Hlávka: Konstrukce sufixových polí a jejich využití v bioinformatice, bakalářská práce, Brno, FIT VUT v Brně, 2011

Konstrukce sufixových polí a jejich využití v bioinformatice

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Tomáše Martínka Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ondřej Hlávka
Datum (18.5.2011)

Poděkování

Chtěl bych poděkovat svému vedoucímu, panu Ing. Tomáši Martínkovi Ph.D., za jeho vůli a ochotu diskutovat se mnou všechny problémy, na které jsem při vypracovávání narazil.

© Ondřej Hlávka, 2011
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

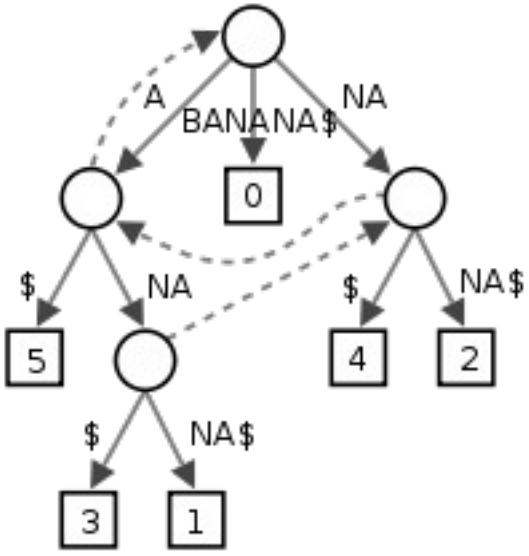
Obsah

1 Úvod

Nejběžnější informací, se kterou se v dnešním světě setkáváme, je prostý text. Nalezneme ho v knížkách, novinách a dalších tištěných médiích. S takovým textem zacházíme jako se sekvencí symbolů a nazýváme jí řetězec, sekvence, slovo nebo text. Takové řetězce hrají klíčovou roli ve spoustě softwarových aplikací počínaje textovými editory, e-mailovými klienty a mnoho dalších konče internetovými prohlížeči, které nám umožňují přijímat a následně číst zprávy z internetu. V molekulární biologii se setkáváme se sekvencemi DNA, RNA nebo sekvencí amino kyselin.

V sekvenční analýze nás zajímá vývoj efektivních datových struktur a algoritmů, které dokáží zpracovat různé typy sekvencí. Velice časté je vyhledávání určitých vzorů v textu, se kterým přichází na řadu následující otázka: Vyskytuje se vzor v dané sekvenci přesně nebo přibližně, a pokud ano na kterém místě se vyskytuje?

Fulltextové rejstříky jsou datovou strukturou, jenž umožňuje zpracovávat ruzné typy sekvencí pro dané aplikace. Tyto rejstříky umožňují rychlý přístup ke každému podřetězci daného řetězce. Sufixový strom je pravděpodobně nejznámější fulltextový rejstřík, který může být sestaven a uložen v *O(n)* čase a paměti pro vstupní řetězec **x** délky *n*. Sufixový strom byl spolu s prvním algoritmem pro jeho konstrukci v lineárním čase představen v roce 1973 (Weiner). V dalších letech byli prezentovány další algoritmy s vylepšenými vlastnostmi *např.* menší paměťová náročnost (McCreight 1976), lineární konstrukční čas pro libovolné abecedy (Farach 1997).



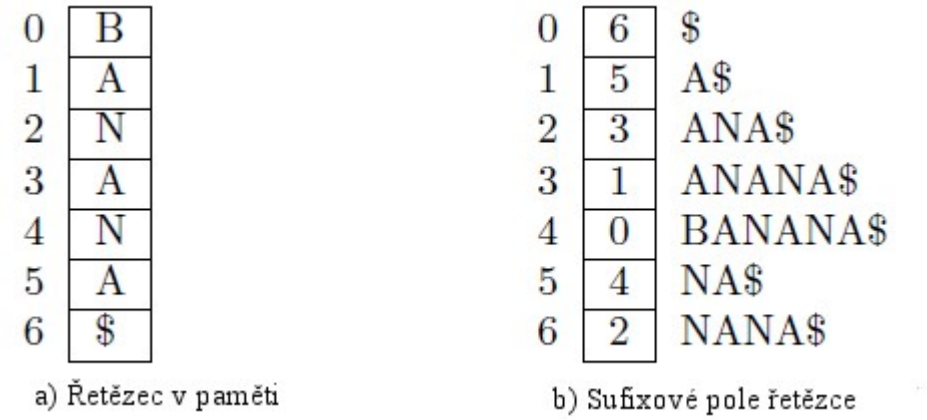
Obr 1.1: Sufixový strom řetězce banana

Sufixové stromy mají v praxi mnoho využití. Jako klasické můžeme brát vyhledávání přesných vzorů: Pro podřetězec *m*, použijeme sufixový strom řetězce a dokážeme rozhodnout v čase *O(m)*, zda se daný podřetězec v řetězci skutečně nachází. Pravá síla sufixových stromů se, ale nalézá v jejich použití pro komplexnější úlohy *(např.* opakované vyhledávání*).*

Hlavní nevýhodou sufixových stromů je jejich paměťová náročnost, jenž několikanásobně převyšuje paměťové nároky na uchovaní samotného řetězce. Do roku 1990 vyžadovala nejúspornější varianta McCreightova algoritmu 28*n* bytů pro řetězec délky *n* (v nejhorším případě a pro čtyř bytový integer). V této práci [1], implementovali Manber a Mayers sufixový strom tak, že jeho paměťová

náročnost byla 14.2*n* až 27.8*n*. Dnes nejúspornější varianta využívá mezi 8*n* a 14*n* byty. Tyto velké paměťové nároky jsou neúnosné v porovnání se zvyšujícími se objemy dat, která jsou potřeba indexovat. Typicky pro některé sekvence genomů. V devadesátých letech se objevili dva technologické projekty s enormními nároky: Google a Human Genome Project. Google se pokusil indexovat čitelné informace dostupné přes internet a Human Genome Project poskytuje genomické sekvence dat lidského druhu.

Jako paměťově úspornou alternativu k sufixovým stromům vymysleli v roce 1990 Manber a Mayers [1] sufixové pole. Toto pole bylo vybaveno ještě dalším pomocným polem LCP, které uchovává délku nejdelšího společného prefixu spolu sousedících sufixů. Bylo představeno i několik dalších alternativ, ale žádná z nich se nestala tak populární jako sufixové pole. Jejich neúspěch lze přičítat hlavně nízkým paměťovým nárokům sufixového pole. Toto pole zabírá v paměti 5*n* bytů včetně vstupního řetězce. V neposlední řadě se jeho jedno-rozměrná struktura dá jednoduše implementovat a dobře se s ní manipuluje.



Obr 1.2: Sufixové pole řetězce banana

Ve druhé kapitole této práce se podíváme na sufixové pole podrobněji. Ukážeme si rozdělení algoritmů pro jeho konstrukci a některé z těchto algoritmů si popíšeme. Ve třetí části se budeme zabývat optimalizací konstrukčního algoritmu qsufsort. V předposlední části této práce se zaměříme na praktické využití sufixového pole v oblasti bioinformatiky. Konkrétně v oblasti přesného a přibližného vyhledávání vzorů v řetězcích. V závěrečné části budeme diskutovat možná rozšíření této práce.

2 Sufixové pole

2.1 Co to je?

Sufixové pole je v porovnání se sufixovým stromem paměťově efektivnější datová struktura, která nám umožňuje rychlé vyhledávání podřetězců v textu. Jedná o seřazené pole *Pos* všech sufixů textu. Sufixové pole pro text délky *n* může být sestaveno v čase $O(n^2 \log n)$, $O(n \log n)$ nebo i s lineární časovou složitostí $O(n)$. Vyhledávání je poté realizováno pomocí rychlého binárního vyhledávání s časovou náročností $O(m \log n)$. Pokud je sufixové pole spárováno s informací o nejdelším společném prefixu (longest common prefix – lcp), může být vyhledávání zrychleno až na $O(m + \log n)$. Informace o *lcp* bývá většinou vypočítávána během sestavování pole, nicméně v některých případech nemusí být okamžitě k dispozici.

1 $
2 i$
3 ippi$
4 issippi$
5 ississippi$
6 mississippi$
7 pi$
8 ppi$
9 sippi$
10 sissippi$
11 ssippi$
12 ssissippi$

Obr 2.1: Sufixové pole řetězce mississippi$ (*$* značí konec řetězce)

Sufixové pole poprvé představili pánové Manber a Myers [1] v roce 1990 jako paměťově méně náročnou alternativu k sufixovým stromům. Společně s ním vytvořili algoritmy pro jejich konstrukci a praktické používání. Od té doby byly publikovány stovky vědeckých článků, které se zabývají používáním sufixových stromů a polí. Z nich vyplývá několik zajímavých poznatků:

- Praktické paměťově efektivní algoritmy pro konstrukci sufixových polí vyžadují v nejhorším případě lineární čas rovný *n*, kde *n* značí délku řetězce *(O(n))*.
- Existují algoritmy, které jsou na reálných datech rychlejší, ale v nejhorším případě mají nelineární složitost *(př. $O(n^2 \log n)$)*.
- Jakýkoli problém, který se dá řešit pomocí sufixových stromů je řešitelný se stejnou asymptotickou složitostí pomocí sufixových polí.

Díky tomu sufixové pole postupně začalo vytlačovat sufixové stromy a většina problémů, týkajících se zpracování řetězců, které se dají řešit pomocí sufixových stromů, se nyní řeší pomocí těchto polí.

Samotné sufixové pole nemá stejnou vyjadřovací schopnost jako sufixový strom. Pokud k sufixovému poli přidáme pole, které nese informaci o *lcp*, můžeme na těchto dvou polích simulovat každý průchod odspoda-nahoru sufixovým stromem. Nicméně informace o *lcp* nám umožňuje simulovat průchod sufixovým stromem pouze směrem od potomků k rodičovským uzlům. Abouelhoda [3] rozšířil sufixové pole o další pomocná pole díky kterým jsme schopni simulovat

průchody sufixovým stromem od rodičovských uzlů k synovským uzlům. Na základě tohoto rozšířeného sufixového pole (enchanced suffix array) sestavili koncept lcp-intervalových stromů. Tyto stromy není nutné v praxi konstruovat a přitom jsou ekvivalentní k sufixovým stromům. Navíc rozšířené pole umí průchod po sufixových linkách (suffix link traversal). Tento průchod můžeme použít k vypočítání statistik úspěšných porovnání. V této práci [3] bylo dokázáno, že obecně kterýkoli algoritmus pracující se sufixovými stromy může být převeden na ekvivalentní algoritmus využívající rozšířené sufixové pole s identickou časovou náročností. Sufixové pole tak má potenciál plně nahradit sufixové stromy v praktických aplikacích.

Rozšířené sufixové pole má oproti sufixovým stromům několik praktických výhod:

1. Je možné je uložit v sekundárních pamětech bez serializování datové struktury, což by bylo nutné u sufixových stromů.
2. Jsou na sobě pomocná pole nezávislá a pro určité aplikace jsou potřeba jen některé, což vede k menší paměťové zátěži.

Sufixové pole se už využívá v různých aplikacích v bionformatice *např.* pro vyhledávání obdobných DNA sekvencí, EST shlukování atd.

V další části se podíváme na základní pojmy a ukážeme si rozdělení konstrukčních algoritmů do různých tříd.

2.2 Rozšířené sufixové pole

Rozšířené sufixové pole bylo poprvé představeno v práci [3]. Jedná se vlastně o sufixové pole doplněné o několik dalších pomocných polí. Tyto pole ovšem vyžadují další paměť navíc. Rozšířené sufixové pole tedy vyžaduje v paměti 8n bytů. Oproti obyčejnému sufixovému poli je to dvojnásobná velikost, ale v porovnání se sufixovým stromy, jejichž paměťová náročnost je kolem 20n bytů, jde stále o přijatelné nároky. Celkově tato struktura nahlíží na data z trochu jiné perspektivy než obyčejné sufixové pole, která je velice podobná sufixovým stromům.

Prvním z těchto rozšiřujících polí je *lcp* pole. Jedná se o pole, jenž udává kolik společných znaků mají sousedící sufixy. Mějme sufixy *yrdenaard* a *yrdenquen*, jejich *lcp* má hodnotu 5 a jedná se o prefix *yrden*. Podrobněji je *lcp* popsáno v další kapitole. Hodnoty *lcp* mohou být využity k definování intervalů nazývaných *lcp intervaly*. Pod *lcp intervalem* si můžeme představit interval, který koresponduje určitému rozsahu sufixů (v sufixovém poli) se specifickým prefixem. Interval $[i...j]$, $0 \leq i < j \leq n$, kde n je délka sufixového pole, je *lcp intervalem* pro *lcp* hodnoty l pokud jsou splněny následující podmínky (lcptab označuje lcp pole):

1. lcptab[i] < l,
2. lcptab[k] ≥ l pro všechna k, pro která platí $i+1 \leq k \leq j$,
3. lcptab[k] = k alespoň pro jedno k, pro které platí $i+1 \leq k \leq j$,
4. lcptab[$j+1$] < l.

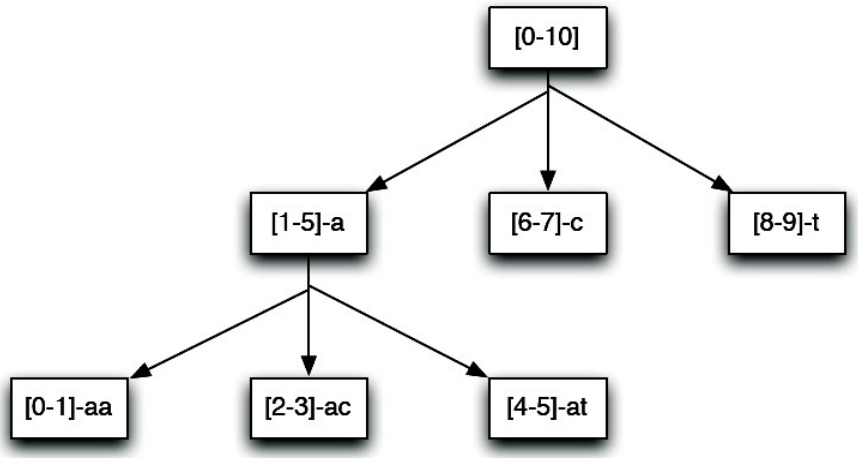
Lcp intervaly v sobě mohou mít začleněny další menší lcp intervaly. Mějme kupříkladu m-interval[$l..r$] a l-interval[$i..j$]. Interval m je začleněn v intervalu l pokud je podintervalem [$i..j$] (př. $i \leq l < r \leq j$) a zároveň $m > l$ (pozn. nemůžeme mít $i = l$ a zároveň $r = j$ protože $m > l$). l-interval[$i..j$] je pak nazýván intervalem uzavírajícím [$l..r$]. Pokud [$i..j$] uzavírá [$l..r$] a v [$i..j$] není začleněn žádný další interval, který by také uzavíral [$l..r$], pak [$l..r$] nazýváme jako synovský interval (child interval) [$i..j$].

Na tyto vnořené intervaly může být nahlíženo jako na stromovou strukturu nazývanou lcp intervalový strom (lcp interval tree). Tento intervalový strom je implicitní a navíc má stejnou strukturu jako sufixový strom, jenž by byl sestaven nad stejnou sekvencí znaků jako dané sufixové pole. K této implicitní stromové struktuře bychom rádi efektivně přistupovali. Proto si uchováme skoky skrz sufixové pole, potřebné pro průchod této stromové struktury shora-dolů, v pomocném poli jež se nazývá *child table*. S pomocí tohoto pole můžeme určit pro každý interval nejdelší společný prefix a jeho synovské intervaly (child intervals) jednoduchým vyhledáním v poli. Jako kořenový interval se vždy bere [0..*n*].

Konstrukce child table může být provedena stejně, tak jako lcp tabulka, pomocí jediného průchodu sufixovým polem. Bohužel sestavení této tabulky závisí na přítomnosti lcp pole, takže po sestavení sufixového pole musíme provést dva průchody pro naplnění pomocných struktur.

			childtab			
i	suftab[i]	lcptab[i]	1.	2.	3.	S[suffix]
0	2	0		2	6	*aaacatat$*
1	3	2				*aacatat$*
2	0	1	1	3	4	*acaaacatat$*
3	4	3				*acatat$*
4	6	1	3	5		*atat$*
5	8	2				*at$*
6	1	0	2	7	8	*caaacatat$*
7	5	2				*catat$*
8	7	0	7	9	10	*tat$*
9	9	1				*t$*
10	10	0	9			*$*

Tabulka 2.1: Rozšířené sufixové pole pro řetězec S = acaaaacatat$ [3]



Obr 2.2: Lcp intervalový strom nad řetězcem S = acaaaacatat$ [3]

2.3 Základní pojmy

Předpokládejme konečný, neprázdný řetězec **x** = **x** [1..*n*] délky *n* ≥ 1, který je definovaný nad indexovanou abecedou Σ pro kterou platí:

- znaky λ_j, *j* = 1, 2, …, σ z |Σ| jsou v pořadí: $\lambda_1 < \lambda_2 < \ldots < \lambda_\sigma$;
- může být definováno pole A[λ_1..λ_σ], ve kterém je pro každé j ∈ 1.. σ, A[λ_j] přístupné v konstantním čase;
- λ_σ - λ_1 ∈ *O(n);*

Předpokládáme, že s Σ můžeme zacházet jako se sekvencí celých čísel typu integer, jejichž rozsah není příliš velký. Tento předpoklad je univerzální pro zpracovávání řetězců v počítači a tudíž je vyžadován pro nejefektivnější algoritmy zpracovávající řetězce. Typicky může λ_j představovat znaky ASCII tabulky (anglická abeceda), binární integery 00..11 (DNA) nebo prostě bity tak jak jsou. Znak může být uchován v bytu a *n* může být uchováno v jednom počítačovém slově (computer word, 4 bajty).

Chceme vytvořit sufixové pole z řetězce **x**, které budeme zapisovat jako $\mathbf{SA_x}$ (suffix array) nebo **SA**. Jedná se o pole SA[1..*n*], ve kterém SA[*j*] = *i* odpovídá **x[***i*..n**]** a zároveň udává *j*-tý sufix řetězce **x** ve vzestupném lexikografickém pořadí. Vždy je vhodné na konec řetězce přidat ukončovací znak $, o kterém předpokládáme, že je menší než kterýkoli znak λ_j.

Příklad:

Mějme abecedu Σ = {$, a, b, c, d, e} a nad ní řetězec:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	*a*	*b*	*e*	*a*	*c*	*a*	*d*	*a*	*b*	*e*	*a*	$
SA =	12	11	8	1	4	6	9	2	5	7	10	3

SA nám udává, že **x**[12..12] = $ je poslední sufix řetězce **x**, **x**[11..12] = a$ je předposlední a tak dále (předpokládáme, že sufixy jsou abecedně seřazeny). Sufixové pole je vždy permutací 1..*n*.

Společně s SA se často používá pole nejdelších společných prefixů (dále jen lcp – longest common prefix) lcp = lcp[1..*n*]. Pro každé *j* ∈ 2..*n* značí lcp[j] délku nejdelšího společného prefixu sufixů SA[*j* - 1] a SA[*j*]. V našem příkladě to bude takto:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	*a*	*b*	*e*	*a*	*c*	*a*	*d*	*a*	*b*	*e*	*a*	$
SA =	12	11	8	1	4	6	9	2	5	7	10	3
lcp =	–	0	1	4	1	1	0	3	0	0	0	2

Nejdelší společný prefix sufixů 11 (a$) a 8 (abea$) je *a* délky 1, zatímco u sufixů 8 (abea$) a 1 (abeacadabea$) to je *abea* délky 4. Jelikož se dá *lcp* vypočítat s lineární časovou složitostí (k jeho sestrojení stačí jediný průchod sufixovým polem), bývá jeho sestavení součástí algoritmů pro sestavování sufixových polí.

Mnoho algoritmů používá i tzv. Inverzní sufixové pole (Inverse suffix array), které budeme zapisovat jako $\mathrm{ISA_x}$ nebo jako ISA, ve kterém platí: ISA[*i*] = *j* <=> SA[*j*] = *i*. ISA[*i*] = *j* nám tedy sděluje, že sufix *i* (**x**[*i*..*n*])má v lexikografickém pořadí pozici *j*.

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
ISA =	4	8	12	5	9	6	10	3	7	11	2	1

Z toho můžeme vyvodit, že ISA nám říká, na kterém místě je v sufixovém poli daný sufix. Vidíme, že sufix 1 je na 4. místě v SA, sufix 2 je na 8. místě atd. ISA je, stejně jako SA, permutací 1..*n*.

Některé algoritmy závisí na dílčím seřazení některých nebo všech sufixů **x**. Dílčí proto, že je to založeno na seřazení prefixů těchto sufixů, které jsou délky $h \geq 1$. Tomuto dílčímu řazení budeme říkat *h*-***řazení*** sufixů do *h*-***řádů*** a samotný proces bude nazývat *h*-***sort***. Pokud si jsou dva nebo více sufixů v rámci *h*-řádu rovny, můžeme o nich prohlásit, že mají stejnou *h*-***pozici***, a tudíž spadají do stejné *h*-***skupiny***. Ekvivalentně toto můžeme nazývat jako *h*-***rovno***. Obyčejně je *h*-sort stabilní, takže předchozí rozřazení sufixů do *h*-skupin zůstane zachováno. Výsledky *h*-sortu jsou často uchovány v přibližném sufixovém poli SA_h a přibližném inverzním sufixovém poli ISA_h. Jako příklad si ukážeme výsledek 1-sortu našeho řetězce:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$SA_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$ISA_1 =$	2	7	11	2	9	2	10	2	7	11	2	1
or	6	8	12	6	9	6	10	6	8	12	6	1
or	2	3	6	2	4	2	5	2	3	6	2	1

Závorky v SA_1 nám uzavírají 1-skupiny, které ještě nebyly redukovány na jedináčky, tedy ještě nebyly kompletně seřazeny. Na příkladě vidíme, že SA_h je stále permutací 1..*n*, kdežto ISA_h tuto vlastnost mít nemusí. Podle požadavků konkrétního algoritmu, může ISA_h vyjadřovat *h*-pozici každé *h*-skupiny různým způsobem. Možnosti máme 3:

- Nejlevější pozice *j* v SA_h prvku z *h*-skupiny, nazývaný **head** (vedoucí) *h*-skupiny.
- Nejpravější pozice *j* v SA_h prvku z *h*-skupiny, nazývaný **tail** (terminál) *h*-skupiny.
- Ordinální počítadlo *h*-skupiny v SA_h.

Porovnání s výsledkem 3-sort:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$SA_3 =$	12	11	(1	8)	4	6	(2	9)	5	7	10	3
$ISA_3 =$	3	7	12	5	9	6	10	3	7	11	2	1
or	4	8	12	5	9	6	10	4	8	11	2	1
or	3	6	10	4	7	5	8	3	6	9	2	1

Povšimněme si, že (*h*+1)-sort je zpřesnění *h*-sort: všechny prvky (*h*+1)-skupiny patří do jedné *h*-skupiny.

2.4 Rozdělení algoritmů

V posledních letech bylo vymyšleno spoustu algoritmů, které využívají různých technik pro sestavování sufixových polí. Puglisi [2] v roce 2007 roztřídil algoritmy pro konstrukci sufixových polí do tří rozdílných tříd: *prefix-doubling*, *recursive* a *induced copying*. Některé algoritmy ovšem jednoznačně nespadají pouze do jedné třídy a tak jsou označovány jako hybridní.

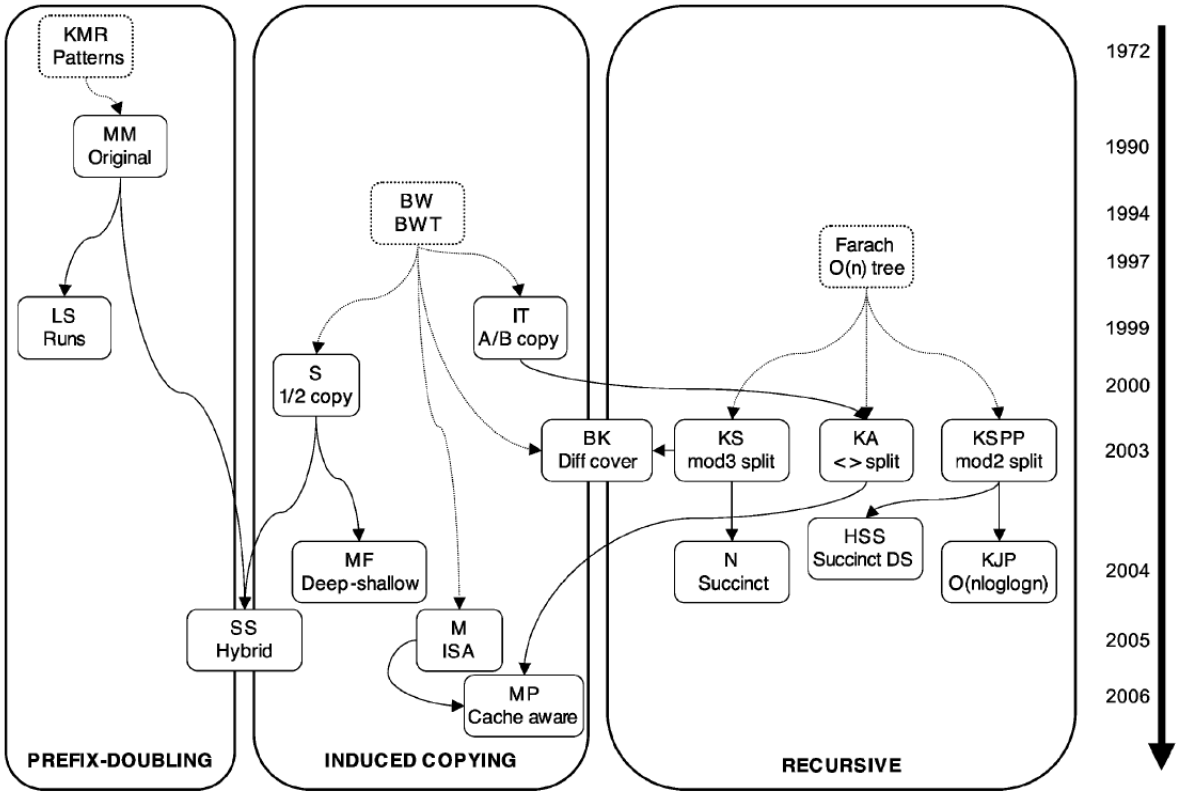
1. Prefix-doubling: Jako první se provede rychlý 1-sort (jelikož je Σ indexovaná, můžeme použít bucket sort). Z toho nám vznikne SA_1/ISA_1. Poté je pro každé h = 1, 2, …, SA_{2h}/ISA_{2h} vypočítáno v čase $\Theta(n)$ z Sa_h/ISA_h, dokud není každá $2h$-skupina jedináček. Protože je zde maximálně $\log_2 n$ iterací, je čas potřebný pro sestavení pole roven $O(n \log n)$. V této třídě jsou dva algoritmy.
2. Recursive: Utvoříme řetězce **x'** a **y** z řetězce **x**. Pokud umíme sestavit $SA_{x'}$, můžeme sestavit SA_y a konečně i SA_x v čase $O(n)$. Výpočet $SA_{x'}$ rekurzivně nahrazuje sestavení SA_x. Jelikož $|\mathbf{x'}|$ vždy volíme tak, aby bylo menší jak $2|\mathbf{x}|/3$, vychází celkový čas potřebný pro konstrukci SA_x $\Theta(n)$. V této třídě se nachází tři hlavní algoritmy
3. Induced copying: Klíčová myšlenka je zde stejná jako u rekurzivních algoritmů – seřazená podmnožina sufixů může být využita k odvození kompletního seřazení jiné podmnožiny sufixů. Zde se ovšem používá nerekurzivní přístup. První myšlenka se objevila v roce 1994 (Burrows a Wheeler) a byla implementována různými způsoby. Obecně jsou tyto metody velmi efektivní v praxi, ačkoli je jejich nejhorší asymptotická složitost $O(n^2 \log n)$. Nicméně toto platí pouze pro řetězce s malou průměrnou hodnotou *lcp*.

Cílem je navrhnout algoritmus takový, který:

- má minimální složitost $\Theta(n)$
- je rychlý na kolekci velkých reálných dat jako je např. lidský chromozom 22
- má malou paměťovou náročnost – zabírá co nejméně místa navíc k $5n$ bytům potřebným pro uložení **x** a SA_x.

Dosud se nepodařilo vytvořit algoritmus, který by splňoval všechny podmínky.

Toto třídění ovšem není jediné. Na základě disertační práce Klause-Bernda Schürmanna [4] se algoritmy dají ještě dělit do dvou ortogonálních tříd. U obou platí, že každý algoritmus může být jednoznačně zařazen pouze do jedné ze dvou možných tříd. První rozřazuje algoritmy s ohledem na postup v řadícím procesu sufixů a druhá rozřazuje algoritmy podle toho, jaké používají závislosti mezi sufixy.



Obr. 2.3: Rozdělení algoritmů dle Puglisi [2]

2.4.1 Prefix doubling

Zde uvažujeme algoritmy, které pro za daný h-řád SA_h sufixů **x**, $h \geq 1$, spočítají $2h$-řád v čase $O(n)$. Tudíž tyto algoritmy vyžadují maximálně $\log_2 n$ kroků pro dokončení řazení sufixů a jsou vykonány přinejhorším v čase $O(n \log n)$.

Poznatek 1: Předpokládejme, že SA_h a ISA_h bylo vypočítáno pro nějaké $h > 0$, kde $i = SA_h[j]$ je j-tý sufix v h-řádu a h-pozice$[i]$ $=ISA_h[i]$. Pak řazení používající pár integerů $(ISA_h[i], ISA_h[i+h])$ jako klíč $(i+h \leq n)$ vede k získání $2h$-řádu sufixů i. (Sufixy $i > n$-h jsou již plně seřazeny) [2]

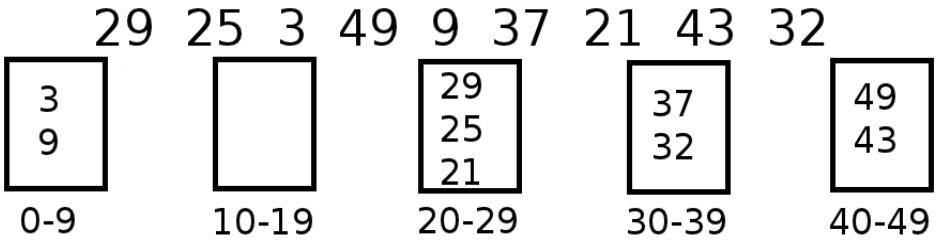
Prefix-doubling algoritmy ustavují pole SA_1 pro $h = 1$ s využitím bucket sortu. Hlavní dva algoritmy se liší v tomto bodě (a zároveň se líší tím, jak aplikují *poznatek 1*):

- originální algoritmus MM (tak jak ho Manber a Myers představili) dělá implicitně $2h$-sort tím způsobem, že prochází pole SA_h zleva-doprava, kterým vyvodí $2h$-pozici $SA_h[j] - h$, $j = 1, 2, \ldots, n$;
- algoritmus Larsson a Sadakane (většinou nazývaný jako *qsufsort*) explicitně řadí každou h-skupinu pomocí ternary-split quicksortu (TSQS, Bentley and McIlroy [1993]).

MM [Manber a Myers 1990]

Tento algoritmus aplikuje *poznatek 1* následovně:

Pokud je SA_h procházeno zleva-doprava (čili podle h-řádu sufixů), $j = 1, 2, \ldots, n$, pak jsou sufixy $i - h = SA_h[j] - h > 0$ procházeny v $2h$-řádu v jejich h-skupinách v SA_h.



Obr. 2.4: Ukázka bucket sortu pro obyčejný číselný seznam.

Algoritmus provede bucket sort sufixů podle jejich prvního znaku. Bucket sort je řadící algoritmus, který vstupní pole rozřadí do přihrádek. Každá přihrádka představuje jeden znak abecedy a jsou v ní sufixy začínající tímto znakem. Přihrádky jsou poté řazeny každá zvlášť. Tím nám vlastně vznikne 1-sort a SA_1. ISA_1 je dopočítána tak, že se specifikuje *h*-pozice pro každý sufix *i* v SA_1 jako nejlevější pozice ve svojí *h*-skupině (tzn. podle vedoucího (head) skupiny):

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$SA_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$ISA_1 =$	2	7	11	2	9	2	10	2	7	11	2	1

Na příkladě výše vidíme, že sufixy, které začínají písmenem *a*, *e* nebo *b*, mají v ISA_1 stejnou pozici odpovídající vedoucímu (head) jejich *h*-skupiny.

Pro vytvoření SA_2 budeme uvažovat kladné hodnoty (větší než 0) $i - h = SA_1[j] - h$ pro $j = 1, 2, \ldots, n$ (*i* zde udává sufix ***x****[i-h...n]):*

- pro j = 1, 7, 8, 9, 10, získáme v 2-řádu sufixy 11, (1, 8), 4, 6 začínající písmenem ***a*** (pro představu kdy $j = 1$ získáme $SA_1[1] = 12$, odečteme h (v tomto případě h = 1) a získáme 11);
- pro $j = 11, 12$, získáme v 2-řádu 2-rovno (na druhé pozici jsou si stále rovny) sufixy (2, 9) začínající písmenem ***b***;
- pro $j = 3, 6$, získáme v 2-řádu 2-rovno sufixy (3, 10) začínající písmenem ***e***;
- pro $j = 2$ získáme index 0, který přeskočíme (viz. popis algoritmu níže). Pro $j = 4, 5$ dostaneme indexy 5 a 7, které jsou již na správné pozici a začínají písmeny ***c*** a ***d***.

Samozřejmě, že skupiny které byly jednočlenné v SA_1 zůstávají jednočlenné i v SA_2. Po zpracování skupin dostaneme:

	1	2	3	4	5	6	7	8	9	10	11	12
$SA_2 =$	12	11	(1	8)	4	6	(2	9)	5	7	(3	10)
$ISA_2 =$	3	7	11	5	9	6	10	3	7	11	2	1

Pro zformování SA_4 budeme uvažovat kladné hodnoty $i - h = SA_2[j] - h$ pro $j = 1, 2, \ldots, n$:

- pro $j = 11, 12$, získáme ve 4-řádu 4-rovno sufixy (1, 8) začínající *ab*;
- pro $j = 2, 5$, získáme ve 4-řádu 4-odlišné sufixy 9, 2 začínající *be*;
- pro $j = 1, 9$, získáme ve 4-řádu 4-odlišné sufixy 10, 3 začínající *ea*;

Z toho vyplývá SA_4 a ISA_4:

	1	2	3	4	5	6	7	8	9	10	11	12
$SA_4 =$	12	11	(1	8)	4	6	9	2	5	7	10	3
$ISA_4 =$	3	8	12	5	9	6	10	3	7	11	2	1

Konečná podoba SA = SA_8 a ISA = ISA_8 se získá ještě jedním zdvojením, které rozdělí sufixy začínající *abea* (1, 8) do pořadí 8, 1.

Každá dvojící iterace je provedena v čase *O(n)* a maximálně je jich provedeno log *n*, což nám dohromady dává nejhůře *O(n* log *n).* Algoritmus obsahuje komplikaci v podobě nutnosti pamatování vedoucího každé *h*-skupiny, nicméně může být naimplementován způsobem, kdy vyžaduje celkem 8*n* bytů v paměti. Vstupní řetězec musí být v hlavní paměti po celou dobu konstrukce SA.

Algoritmus MM:

Algoritmus 1 [2]:

Vstupem této procedury je řetězec **x** a výstupem je jeho sufixové a inverzní sufixové pole

Inicializace SA_1, ISA_1
while nějaká *h*-skupina není jednočlenná
for $j \leftarrow 1$ **to** *n* **do**
$i \leftarrow SA_h[j] - h$
if $i > 0$ **then**
$q \leftarrow$ head [*h*-skupina[*i*]]
$SA_{2h}[q] \leftarrow i$
head[*h*-skupina[*i*]] $\leftarrow q + 1$
vypočti ISA_{2h} – aktualizace 2*h*-skupin
$h \leftarrow 2h$

LS [Larsson a Sadakane 1990] – qsufsort

První rozřazení probíhá stejně jako u algoritmu MM podle prvního znaku sufixu. Pro prvotní rozřazení je použit bucket sort. Pokud ovšem není možné bucket sort provést, použije se ternary-split quicksort. Změna je ve výpočtu ISA_1. Zde se bere nejpravější pozice z každé *h*-skupiny (čili terminál (tail)).

Na příkladu můžeme vidět rozdíl v ISA_1 oproti algoritmu MM, kde se do ISA dává pozice vedoucího prvku ve skupině. Další rozdíl oproti MM je ten, že q*sufsort* k identifikování *h*-skupin v SA_h, které mají víc jak jednoho člena, navíc přidává identifikaci po sobě jdoucích pozic, jenž mají pouze jednoho člena (jsou plně seřazeny), tzv. **run**. Pro tyto účely je vyhrazeno pole L = L[1..*n*], v němž L[*j*] = *l* (respektive -*l*), jestliže *j* je nejlevější pozice člena *h*-skupiny délky *l* (případně run jenž se značí -*l*) v Sa_h. Záporná hodnota označuje již seřazené skupiny. Tyto skupiny vznikají konkatenací již seřazených skupin a proto jsou též někdy nazývány jako kombinované seřazené skupiny (jsou zde skupiny s rozdílným *h*).

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$SA_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$ISA_1 =$	6	8	12	6	9	6	10	6	8	12	6	1

Pro zadaný 1=sort získáme takovéto pole L:

	1	2	3	4	5	6	7	8	9	10	11	12
L =	−1	5					2		−2		2	

Při zpracovávání zleva-doprava tak můžeme přeskočit již seřazené skupiny, které jsou identifikovány jako run. Díky tomu můžeme identifikovat *h*-skupiny s více členy v čase úměrném

počtu *h*-skupin a runs. Následně se provede další TSQS, pro seřazení sufixů *i*, nad každou identifikovanou *h*-skupinou podle klíčů $ISA_h[i + h]$. Vznikne nám kolekce podskupin a subruns v 2*h*-řádu. Přímá aktualizace L a ISA nám dá:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ SA_2 = & 12 & 11 & (1 & 8) & 4 & 6 & (2 & 9) & 5 & 7 & (3 & 10) \\ ISA_2 = & 4 & 8 & 12 & 5 & 9 & 6 & 10 & 4 & 8 & 12 & 2 & 1 \\ L = & -2 & & 2 & & -2 & & 2 & & -2 & & 2 & \end{array}$$

Dalším zdvojením získáme:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ SA_4 = & 12 & 11 & (1 & 8) & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3 \\ ISA_4 = & 4 & 8 & 12 & 5 & 9 & 6 & 10 & 4 & 7 & 11 & 2 & 1 \\ L = & -2 & & 2 & & -8 & & & & & & & \end{array}$$

Finálního výsledku SA_8 a ISA_8 je dosaženo stejně jako u MM s tím, že L[1] = -12.

Povšimněme si, že jako u MM, *qsufsort* udržuje $ISA2_h[i] = ISAh[i]$ pro každý sufix *i*, který je jedináček ve své *h*-skupině. Na rozdíl od MM se však *qsufsort* vyhýbá nutnosti procházet každou pozici v SA_h. Je to díky použití pole L. Jakmile je pro nějaké *h i* identifikováno jako jedináček, k $SA_h[i]$ již není přistupováno.

Algoritmus má stejnou časovou i paměťovou složitost jako MM, ale při praktickém testování se ukázal daleko rychlejší (10x i víc). Podrobněji bude popsán ve třetí kapitole.

2.4.2 Recursive

Všechny algoritmy v této třídě byli objeveny v roce 2003 nebo později. Využívají rekurzi a většinou je jejich nejhoší časová náročnost lineárně úměrná celkové délce řetězce **x**. Jsou založeny na myšlence, se kterou prišel v roce 1997 Farach, jenž se týká sestavování sufixových stromů v lineárním čase nad indexovanou abecedou. Závisí na počátečním přiřazení typu každému sufixu (pozici) v **x**. Tento typ rozděluje sufixy do dvou či více tříd. Rekurze je tedy ve všech případech založená na rozdělení (split) zadaného řetězce $\mathbf{x} = \mathbf{x}^{(0)}$ do disjunktních komponent (subsekvencí), které jsou transformovány na řetězce jež nazýváme $\mathbf{x}^{(1)}$ a $\mathbf{y}^{(1)}$. Vybrány jsou tak, že pokud je $SA_{x(1)}$ rekurzivně spočítáno, pak je to s lineární časovou složitostí.

- $SA_{x(1)}$ může být použito k odvození (induce) konstrukce $SA_{y(1)}$ atd.
- $SA_{x(0)}$ může být vypočítano sloučením (merge) $SA_{x(1)}$ a $SA_{y(1)}$

Díky tomu je výpočet $SA_{x(0)}$ redukován na výpočet $SA_{x(1)}$ (obecně tedy $SA_{x(i+1)}$). Aby byla tato strategie efektivní, musíme splnit dva požadavky:

1. V každém rekurzivním kroku musíme zajistit aby platilo:
   $$|\mathbf{x}^{(i+1)}|/|\mathbf{x}^{(i)}| \leq f < 1;$$
   Proto je součet délek řetězců zpracovaných všemi rekurzivními kroky:
   $$|\mathbf{x}| (1 + f + f^2 + \ldots) < |\mathbf{x}| / (1 - f).$$
   Všechny dosud vymyšlené algoritmy mají $f \leq 2/3$, takže součet jejich délek je zaručeně menší než $3|\mathbf{x}|$ – pro většinu z nich dokonce menší nebo rovno $2|\mathbf{x}|$.
2. Je třeba vymyslet proceduru řazení sufixů nazývanou ***semisort***, která pro dostatečně krátky řetězec $\mathbf{x}^{(i+1)}$ provede kompletní seřazení jeho sufixů a tím ukončí rekurzi. Jedná se tedy o ukončovací podmínku rekurze. Sufixy $\mathbf{x}^{(i)}, \mathbf{x}^{(i-1)}, \ldots, \mathbf{x}^{(0)}$ pak můžeme seřadit do správného pořadí. Nadto musíme zajistit, že čas potřebný pro *semisort* je lineárně úměrný délce zpracovávaného řetězce.

Algoritmus, který splňuje výše uvedené požadavky, sestaví SA_x (nebo ekvivalntně ISA_x) řetězce $\mathbf{x} = \mathbf{x}[1..n]$ v čase $\Theta(n)$. Obecná struktura rekurzivních algoritmů je popsána v algoritmu 2.

Všechny algoritmy z této rodiny vytvářejí $\mathbf{x}'$ (např. $\mathbf{x}^{(1)}$) a $\mathbf{y}$ (např. $\mathbf{y}^{(1)}$) z $\mathbf{x}$ (např. $\mathbf{x}^{(0)}$) podobným způsobem: abeceda rozdělených řetězců je vlastně množina sufixů (pozic) $1..n$ v $\mathbf{x}$, takže $\mathbf{x}'$ a $\mathbf{y}$ dohromady tvoří permutaci $1..n$.

Pozornost se poté soustředí na ohodnocení pozic i' v $\mathbf{x}'$, řetězce délky $n' \leq \lfloor n/2 \rfloor$. Toto hodnocení je založeno na výpočtu hodnot korespondujících sufixů z $\mathbf{x}$, které byli přiřazeny do $\mathbf{x}'$. Toto nazýváme jako řetězec hodnot $ISA_{x'} = ISA_{x'}[1..n']$

$$\boldsymbol{x}[\boldsymbol{x}'[i']..n] \qquad (1)$$

Jelikož nám může konstrukce $ISA_{x'}$ vyžadovat čas delší než $\Theta(n')$, voláme zde proceduru *semisort*, která v čase $\Theta(n')$ zkonstruuje aproximaci $ISA' = ISA'[1..n']$ k $ISA_{x'}$. Jedná se vlastně o částečné hodnocení (h-ranking) sufixů (1) a dokáže je správně rozřadit jen do určité délky společného prefixu, která je předem dána konstantní délkou h. V určité úrovni rekurze se aproximace ISA' stává přesnou (její záznamy jsou jedinečné), takže můžeme zapsat že $ISA_{x'} = ISA'$. Následně procedurou *invert* vytvoříme z $ISA_{x'}$ sufixové pole $SA_{x'}$.

Pokud však ISA' není přesné (obsahuje duplicitní hodnoty tzn. obsahuje řetězce, které mají shodný prefix délky h), je použito jako vstupní řetězec pro rekurzivní volání procedury *construct*. Tím získáme SA' z ISA'. Zde je společný klíčový poznatek rekurzivních algoritmů. Jakmile je SA' sufixové pole přibližných hodnot sufixů identifikovaných pomocí $\mathbf{x}'$, jedná se zároveň i o sufixové pole samotných sufixů. Proto můžeme zapsat $SA_{x'} = SA'$. Níže si popíšeme jeden rekurzivní algoritmus zvaný *smaller-larger*.

Rekurzivní algoritmy mají výhodu v tom, že mají lineární čas konstrukce, ale jejich nevýhodou je obvykle vyšší paměťová náročnost rekurze.

Obecný algoritmus rekurzivní konstrukce:

Algoritmus 2 [2]:

Vstupem je řetězec $\mathbf{x}$ a výstupem je jeho sufixové pole SA

procedure construct (**x;** SA)
  split (**x; x', y**)
  semisort **(x';** ISA'**)**
  if ISA' obsahuje duplicitní hodnoty **then**
    construct (ISA'; $SA_{x'} = SA'$)
  else
    invert ($ISA_{x'} = ISA'$; $SA_{x'}$)
  induce($SA_{x'}$, $ISA_{x'}$; SA_y)
  merge($SA_{x'}$, SA_y; Sa_x)

KA [Ko a Aluru 2003,2005] – smaller-larger

Dělící procedura tohoto algoritmu přiřazuje sufixy $i < n$ v pořadí zleva-doprava do sekvence S (respektive sekvence L) právě tehdy, když $\mathbf{x}[i..n] <$ (respektive $>$) $\boldsymbol{x}[i+1..n]$. Sufix n ($) je přiřazen jak

do *S*, tak do *L*. Jelikož **x**[*i*] = *x*[*i*+1] značí, že sufixy *i* a *i*+1 patří do stejné sekvence, potřebuje *smaller-larger* algoritmus pro rozdělení lineární čas úměrný délce řetězce **x**.

X' je poté formováno ze sekvence sufixů menší kardinality, kdežto **y** je formováno ze sekvence sufixů vyšší kardinality. Proto v algoritmu platí |**x'**| ≤ |**x**|/2. Příklad:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
type =	L	S	L	L	S	L	L	S	L	L	L	S/L

Získáme |*S*| = 4, |*L*| = *9*, ***x'*** = 2 5 8 12, **y** = 1 3 4 6 7 9 10 11 12.

Pro každé *j* ∈ 1..|**x'**| formuje KA *semisort* procedura *i* = **x'** [*j*], i_1 = **x'**[*j*+1] (i_1 = **x'**[*j*] pokud *j* = | **x'**|). Poté se provede radix sort, který nám dá podřetězce **x**[*i*..i_1], tato kalkulace vyžaduje *Θ(n)* času. Výsledek tohoto řazení jsou indexy v ISA' podřetězců **x**[*i*..i_1], čili se jedná o přibližnou pozici sufixů v konečném sufixovém poli. V našem případě po provedení semisortu dostaneme:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
$\boldsymbol{x'} =$	2				5			8				12
$\mathrm{ISA'} =$	3				3			2				1

Pokud po této proceduře jsou všechny záznamy v ISA' odlišné, získáváme kompletní seřazení sufixů **x'** a ISA' odpovídá $\mathrm{ISA}_{x'}$. Jestliže však ISA' obsahuje nějaké duplicitní hodnoty, provádí se tato procedura rekurzivně, dokud se duplicita neodstraní (můžeme vidět z Algoritmu 2). V našem případě stačí pouze jedno rekurzivní volání pro získání finálního seřazení (12, 8, 5, 2) sufixů **x'** a $\mathrm{ISA}_{x'}$ = 4321.

V tomto bodě se KA liší od obecného postupu vyjádřeného v algoritmu 2. KA kombinuje procedury *induce* a *merge* do jedné procedury obecně zvané KA-merge (algoritmus 3) a sestavuje přímo SA_{x} bez odkazu na $\mathrm{ISA}_{x'}$.

Při konstrukci SA_1 získáváme 1-skupiny pro které je nejlevější a nejpravější pozice specifikována v polích head[1..**α**] respektive tail[1..**α**]. Jelikož jsou v každé 1-skupině všechny *S*-sufixy lexikograficky větší než všechny *L*-sufixy a *S*-sufixy byly kompletně seřazeny může KA-merge všechny *S*-sufixy zařadit na jejich konečnou pozici v SA. Pokaždé když se tak děje je terminál (tail) každé skupiny snížen o jedničku. (V tomto popisu předpokládáme, že |*S*| ≤ |*L*|.)

Sufixové pole ve stávající fázi vypadá takto ('-' značí prázdné pozice):

	1	2	3	4	5	6	7	8	9	10	11	12
SA =	12	(–	8	5	2)	(–)	(–	–)	(–	–	–	–)
type =	S	L	S	S	S	L	L	L	L	L	L	L

Pro seřazení *L*-sufixů procházíme SA zleva-doprava. Pro každou sufixovou pozici *i* = SA[*j*], na kterou narazíme při procházení SA, kontrolujeme zda je *i*-1 *L*-sufix stále čekající na seřazení (nezařazen v SA). Pokud ano, zařadíme *i*-1 do jeho skupiny v SA a inkrementujeme index vedoucího (head) skupiny o jedničku. Sufix *i*-1 je nyní seřazen a znovu se s ním již nebude hýbat. Korektnost této metody závisí na faktu, že když při procházení SA dosáhneme pozice *j*, SA[*j*] je už na své finální pozici. V našem příkladě řazení začíná na pozici kdy *j* = 1, takže *i* = SA[1] = 12. Sufix *i*-1 = 11 a je typ *L* je proto zařazen do fronty *a* skupiny ve které je jediným členem (Zbytek *a* skupiny byl již seřazen v semisortu a proto se s ním nijak nemanipuluje).

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (- & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

Při dalším procházení narazíme na j = 2, i = SA[2] = 11 a zařadíme i-1 = 10 na začátek c skupiny v SA[7] a zvýšíme index vedoucího skupiny (vedoucí skupiny již není na 7. pozici, ale na 8.).

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (10 & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

Pokračujeme s procházením až dostaneme:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{array}$$

Algoritmus KA může být implementován tak, že zabírá pouze 4n bytů plus 1.25n bitů navíc k paměti potřebné pro uchování původního řetězce **x** a SA.

Procedura KA-merge:

Algoritmus 3 [2]:

Vstupem je $SA_{x'}$, získané z úvodní procedury *construct* (algoritmus 2) a výstupem je SA.

Inicializace SA ← SA_1, head[1..σ], tail[1..σ]
for i ← |**x'**| **downto** 1 **do**
  λ ← **x**[**x'**[i]]
  SA[tail[λ]] ← **x'**[i]
  tail[λ] ← tail[λ]-1
for j ← 1 **to** n **do**
  i ← SA[j]
  if type [i-1] = L **then**
    λ ← **x**[i-1]
    SA[head[λ]] ← i-1
    head[λ] ← head[λ]+1

2.4.3 Induced Copying

Algoritmy v této třídě jsou sjednoceny myšlenkou, že hotové seřazení vybrané podmožiny sufixů může být použito k odvození rychlého seřazení zbylých sufixů. Toto odvozené řazení je dosti podobné proceduře použité v rekurzivních algoritmech, zde se však místo rekurze používá iterace. Nahrazení rekurze iterací pravděpodobně vysvětluje proč je většina algoritmů založených na tomto principu v praxi rychlejší než rekurzivní algoritmy, přestože žádný z těchto algoritmů nemá nejhorší časovou složitost lineární. Nejhorší časová složitost těchto algoritmů bývá $O(n^2 \log n)$. Výhodou je ovšem jejich malá paměťová náročnost. Pro většinu z nich je třeba navíc použít méně než n bytů.

Naznačíme si několik možných přístupů:

- Seward [2000] řadí určité 1-skupiny a využívá výsledek k odvození korespondujících 2-skupin. Tento přístup tvoří také základ algoritmu zvaného *deep shallow* (Manzini a Ferragina 2004).

- Další přístup Burkhardt a Kärkkäinen [2003] používá malý integer *h* pro zformování vzorek *S* sufixů, které jsou následně *h*-seřazeny za použití technik připomínajích rekurzivní algoritmy. Výsledné *h*-hodnoty jsou použity k vyvození kompletního seřazení všech sufixů.

S [Seward 2000]

Algoritmus začíná tím, že v lineárním čase vytvoří 2-sort sufixů **x**. Vzniká nám sufixové pole SA_2 ve kterém jsou hranice každé 2-skupiny určeny vedoucím pole. Ten zároveň určuje hranice 1-skupin. Proto nám v tomto případě head = head [1..σ, 1.. σ] umožňuje přistoupit ke každému hraničnímu prvku head[λ, μ] pro každé λ, μ ∈ Σ. První krok tohoto algoritmu by mohl vypadat následovně:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$

$\mathrm{SA}_2 = 12\ (11\ 8\ [2\ 5])\ 1\ (10\ 9)\ ([4\ 7]\ [3\ 6])$

Kulaté závorky zde uzavírají 1-skupiny, které nejsou jedináčky (jejich první znak je stejný), a hranaté závorky uzavírají 2-skupiny (jejich první a druhý znak je stejný).

Nyní uvažujme 1-skupinu G_λ korespondující k jednopísmennému prefixu λ. Předpokládejme, že sufixy G_λ jsou plně seřazeny a dávají nám sekvenci G_λ* ve vzestupném lexikografickém pořadí. Představme si, že G_λ procházíme v lexikografickém pořadí: pro každý sufix $i > 1$ může být sufix i-1 zařazen na svojí konečnou pozici v SA_x jako vedoucí 2-skupiny pro **x**[i-1]λ. Poskytnutý index head[**x**[i-1],λ] je zvýšen o jedničku, jakmile je sufix umístěn na své místo, což nám umožňuje korektní umístění ostatních sufixů ze stejné 2-skupiny. Lexikografické pořadí G_λ* zajišťuje, že sufixy i-1 budou v lexikografickém pořadí i v každé 2-skupině.

Strategie algoritmu S je následující: Používá efektivní TSQS pro seřazení sufixů 1-skupiny, která obsahuje nejmíň neseřazených sufixů, následně použije seřazené sufixy i pro odvození seřazení sufixů i-1. Z toho vyplývá, že všechny sufixy mohou být seřazeny za cenu kompletního seřazení poloviny z nich.

Tento proces se dá ještě více zefektivnit díky poznatku, že když máme G_λ seřazeno, můžeme přeskočit sufixy s prefixem λ^2. 2-skupina λ^2 se tak vlastně stává poslední 2-skupinou G_λ, která musí být procházena. Předpokládejme existenci sufixu λ^k μ**v** v G_λ, $k \geq 2$, μ ≠ λ. Pak sufix λμ**v** bude zařazen do G_λ* a zpracován pro zařazení sufixu **x**[$i..n$] = λ^2μ**v** na index head[λ, λ]. Jakmile je zpracováváno samotné λ^2μ**v**, sufix **x**[i-1]λ^2μ**v** bude umístěn na pozici head[**x**[i-1], λ] – to bude opět (nyní inkrementován) head[λ, λ], pokud $k \geq 3$ (**x**[i-1] = λ).

Pro lepší představu aplikujeme tento algoritmus na náš ukázkový řetězec:

Iterace 1: 1-skupina koresponduje λ = \$ a obsahuje jediný neseřazený sufix $i = 12$. Seřazení je tedy triviální: 12 je na své finální pozici v SA (na první pozici) a sufix i-1 = 11 je pak zařazen na svojí pozici head[a, \$] = 2.

Iterace 2: Nejmenší 1-skupina (má nejmíň členů) odpovídá λ = b a obsahuje jediný sufix $i = 1$, který je proto na své konečné pozici (SA[6]). Protože i-1 = 0, nic dalšího se neděje (jedná se vlastně o celý řetězec, protože b je na začátku).

Iterace 3: Další nejmenší 1-skupina odpovídá λ = c. Opět má pouze jeden záznam k seřazení protože jedna z přítomných 2-skupin je cc. Sufix $i = 10$ má proto finální pozici head[c, a] = 7 a určuje finální pozici sufixu i-1 = 9 na pozici head[c, c] = 8. Nakonec je pro $i = 9$ finální pozice sufixu i-1 = 8 zafixována na pozici head[a, c] = 3 (z prvního kroku vidíme, že po poslední zařazení a inkrementování head skupiny a je head = 3).

Iterace 4: 1-skupina pro λ = *a* nyní obsahuje pouze dvě neseřazené sufixy 2 a 5, protože 11 a 8 byli na své pozice přesunuty při předchozích iteracích. Seřazením získáme SA[4] = 5, SA[5] = 2, takže kompletně seřazená 1-skupina je SA[2..5] = 11 8 5 2. Pro *i* = 11 je sufix *i*-1 = 10 na své pozici; pro *i* = 8 je sufix *i*-1 = 7 na své pozici head[*d, a*] = 9; a nakonec pro *i* = 5 (poté co je inkrementován head[*d, a*]) je sufix *i*-1 = 4 zařazen na svojí konečnou pozici head[*d, a*] = 10. Pro *i* = 2 je *i*-1 = 1 přítomno na své pozici.

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	*b*	*a*	*d*	*d*	*a*	*d*	*d*	*a*	*c*	*c*	*a*	$
$SA_2 =$	12	11	8	5	2	1	10	9	(7	4	[3	6])

Iterace 5: Poslední skupina se sestává z λ = *d*. Zatím se jedná o neseřazené sufixy 3 a 6 patřící do 2-skupiny *dd* a proto nepotřebují řadit. Jako výsledek Iterace 4 máme SA[9..10] = 7 4. A tak pro *i* = 7 máme sufix *i*-1 = 6, který je zařazen na index head[*d, d*] = 11 zatímco *i* = 4 je poslední sufix *i*-1 =3 a je umístěn na pozici head[*d, d*] = 12.

Nakonec dostáváme SA stejně jako u algoritmu KA:

	1	2	3	4	5	6	7	8	9	10	11	12
SA =	12	11	8	5	2	1	10	9	7	4	6	3

V tomto případě je nutné pouze jedno jednoduché seřazení (sufixy 2 a 5 ve 4. iteraci) pro zformování celého pole SA_z.

Tento algoritmus má nejhorší časovou složitost $O(n^2 \log n)$. Čili je stejná jako u ostatních algoritmů této třídy. Má však velkou nevýhodu. Při velké průměrné délce *lcp* dochází ke značné degeneraci algoritmu a stává se nepoužitelným. Tento problém odstraňuje následující algoritmus.

MF [Manzini a Ferragina 2004] – deep-shallow

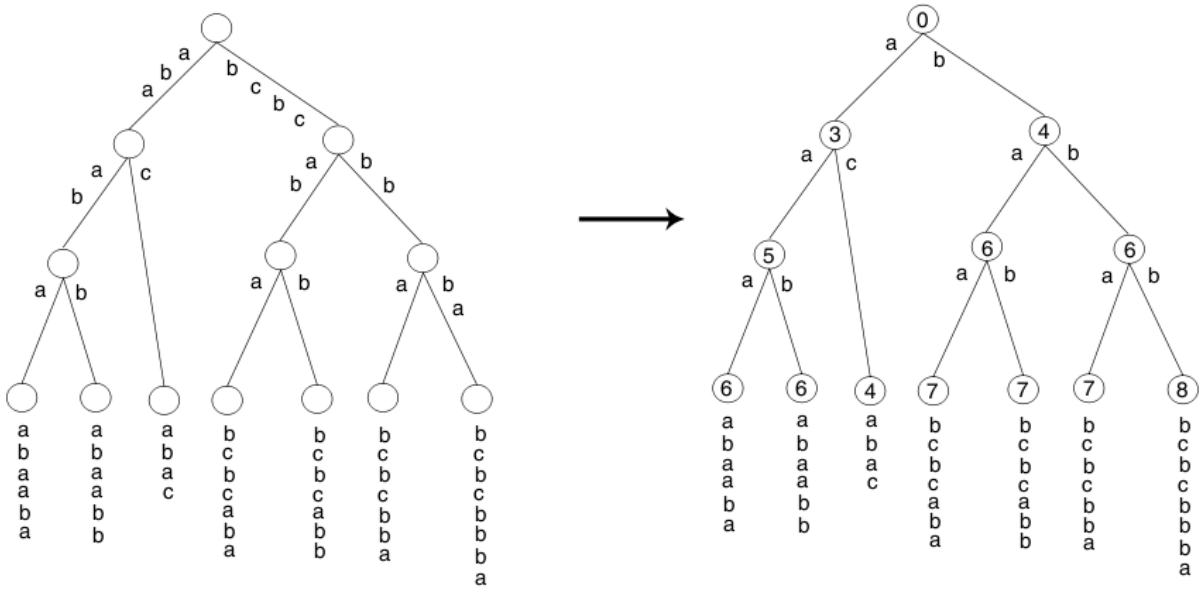
Algoritmus *deep-shallow* vychází ze Sewardova algoritmu zmíněného výše. TSQS, který řadí 2-skupiny ve vybraných 1-skupinách, je zde nahrazen komplikovanějším a sofistikovanějším přístupem k řazení sufixů. Tento přístup je dvou-úrovňový a závisí na uživatelem specifikované délce lcp*. To udává nejdelší lcp skupiny sufixů jež budou řazeny za pomocí standardní metody (typicky je pro velké soubory lcp* voleno v rozmezí 500-5000). Pokud má být tedy 2-skupina řazena, použije se MKQS (multi-key quicksort, místo TSQS) [Bentley a Sedgewick 1997] dokud jeho rekurze nedosáhne hloubky lcp*. Pokud není řazení dokončeno, máme definovanou množinu $I_m = \{i_1, i_2, .., i_m\}$, $m \geq 2$, sufixů pro něž platí: $lcp(i_1, i_2, .., i_m) \geq$ lcp*.

V tomto bodě je metodologie řazení těchto *m* sufixů určena podle toho, zda je *m "velké"* nebo *"malé"*.

Pokud je *m* malé, tak se použije metoda zvaná ***blind sort*** [Ferragina a Grossi 1999], která používá maximálně 36*m* bajtů paměti. Použijeme-li ho pouze pro $m \leq n/Q$, zabírané místo bude maximálně $(36/Q)n$ bajtů. Vhodným zvolením *Q*, řekněme $Q \geq 1000$, ošetříme případy kdy ne mnoho sufixů sdílí dlouhé lcp a tím dosáhneme, že pro malé *m* bude využité místo jen zlomek 5*n* bajtů potřebných pro **x** a Sa_x.

Blind sort I_m závisí na konstrukci speciální datové struktury ***blind trie*** [Ferragina a Grossi 1999]. V podstatě jsou řetězce $\mathbf{x}[i_j + \text{lcp}^* \, ..n]$, $j = 1, 2, \ldots, m$ vkládány do prázdné blind trie. Po průchodu této struktury zleva-doprava dostáváme tyto sufixy v požadovaném lexikografickém pořadí.

Pokud je m větší ($> n/Q$), algoritmus přepne na trochu modifikovaný TSQS (jako je použitý v algoritmu od Sewarda). Jestliže se však v nějaké úrovni rekurze narazí na novou množinu sufixů I'_m, jejíž $m \leq n/Q$, znovu se zavolá blind sort pro seřazení I'_m.



Obr 2.4.: Compacted trie (vlevo) a Blind trie (vpravo)

Po úvodním seřazení, pomocí MKQS do určité hloubky lcp*, je tato duální strategie (blind sort/TSQS) jednou ze dvou strategií, kterou *deep-shallow* využívá. Předtím, než se uchýlí k této strategii, se algoritmus pokusí použít ***generalized induced copying***.

Předpokládejme, že pro každé $i_1 \in I_m$ a zároveň $l \in 1..\text{lcp*-1}$ platí $\mathbf{x}[i_1 + l..i_1 + l + 1] = \lambda\mu$, kde [λ, μ] identifikuje 2-skupinu, která je následkem předchozího zpracování plně seřazena. Protože sufixy m v I_m sdílejí stejný společný prefix, můžeme odvodit, že každý sufix v I_m se vyskytuje v té samé 2-skupině [λ, μ]. Navíc sufixy m v I_m jsou identické do pozice l. Z toho vyplývá, že pořadí sufixů v I_m je určeno jejich pořadím v [λ, μ]. Pokud tedy taková 2-skupina existuje, můžeme ji využít k odvození korektního seřazení sufixů v I_m a to následujícím způsobem:

(1) Použijeme bucket-sort pro seřazení záznamů $i_j \in I_m$ ve vzestupném pořadí podle *pozice* (ne sufixu) tak, abychom členství v I_m mohli určit pomocí binárního vyhledávání (krok (3)).

(2) Projdeme 2-skupinu [λ, μ] abychom mohli najít shodu pro sufix $i_1 + l$, řekněme na pozici q.

(3) Prozkoumáme sufixy (pozice) vypsány doleva a doprava od q v 2-skupině [λ, μ]; pro každý sufix i použijeme binární vyhledávání pro určení, zda se i-l vyskytuje v právě seřazeném I_m. Pokud ano, označíme si sufix i v [λ, μ].

(4) Jakmile bylo označeno m sufixů, projdeme 2-skupinu [λ, μ] zleva-doprava a pro každý označený sufix i zkopírujeme i-l zleva-doprava do I_m.

Krok (2) této procedury může být časově náročný, protože může zahrnovat hledání shody dvou sufixů v čase $\Theta(n)$. Bylo dokázáno [5], že krok 2 může být efektivně implementován a využívá navíc jen minimum místa.

Jestliže však neexistuje žádné l, tedy žádná seřazená 2-skupina jejíž lcp je menší než lcp*, nelze tuto metodu použít. Místo toho musíme použít dříve popsanou duální strategii.

Algoritmus byl představen ve třech verzích [5]. První využívá pouze duální strategii byl označen jak ds0. Druhá přidává výše popsanou metodu generalized induce copying. Označuje se jako ds1 a ds0 překonává téměř ve všech směrech. Je ovšem trošku pomalejší pro řetězce s malou průměrnou délkou lcp, ale pro soubory s dlouhým průměrným lcp je znatelně rychlejší. Třetí modifikace pozměňuje krok (2). Za cenu vyšší paměťové náročnosti se této operaci dá vyhnout. Tato varianta označována jako ds2 je v průměru nejrychlejší. V porovnání se všemi zde zmíněnými algoritmy vítězí *deep-shallow* ve všech ohledech. Paměťová náročnost algoritmu je 5.03*n* bajtů. Nejhorší časová složitost je stejná jako u algoritmu S a to $O(n^2 \log n)$.

BK [Burkhardt a Kärkkäinen 2003] – difference cover

Algoritmus sestaví SA_x tím, že seřadí množinu sufixů *S*. Relativní hodnoty sufixů v *S* jsou použity pro urychlení základního algoritmu pro sezařezení řetězců (např. multi-key quicksort MKQS). Jádrem DC je matematický konstrukt zvaný ***difference cover,*** který určuje sufixy v *S*. Difference cover D_h je množina integerů v rozsahu 0..*h* – 1, ve které pro každé i ∈ 0..h – 1 existuje *j*, *k* ∈ D_h takové, že *i* = *k* - *j* (mod *h*). Pro zvolené $D_h S$ obsahuje sufixy z **x** začínající na pozici i, kde *i* mod h ∈ D_h.

Mějme například D_7 = D{1, 2, 4} difference cover modulo 7. Pokud budeme vzorkovat podle D_7 pak pro řetězec

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\boldsymbol{x} =$	*b*	*a*	*d*	*d*	*a*	*d*	*d*	*b*	*a*	*d*	*d*	*a*	*d*	*d*	*b*	*a*	*d*	*d*	$

získáme *S* = {1, 2, 4, 8, 9, 11, 15, 16, 18, 22, 23, 25}. Všimněme si, že pro každé *i* ∈ *S* je *i* mod 7 přítomno v D_7.

Algoritmus se skládá ze dvou hlavních částí. Cílem první fáze je vytvořit datovou strukturu $ISA_{x'}$, jenž nám umožňuje sestavit lexikografické hodnoty *i* ∈ *S* v konstatním čase, relativně k ostatním členům *S*. Za tímto účelem DC prvně provede *h*-sort *S* za pomocí MKQS (nebo jiné alternativy) a pak přiřadí každému sufixu jeho *h*-hodnotu. V našem případě jsou *h*-hodnoty:

$i \in S$	1	2	4	8	9	11	15	16	18
h-rank	3	6	4	3	6	4	2	5	1

Tyto hodnoty jsou použity pro konstrukci nového řetězce **x'**:

$i \in S$	1	8	15	2	9	16	4	11	18
$\boldsymbol{x'} =$	(3	3	2)	(6	6	5)	(4	4	1)

H-hodnoty *(r_i)* se objevují v $|D_h|$ skupinách v řetězci **x'** (jsou v závorkách) v souladu s *i* modulo *h*. Pak jsou v každé skupině skupině hodnoty r_i seřazeny vzestupně podle *i*. Díky této struktuře v **x'** může být $ISA_{x'}$ použita pro získání hodnoty jakéhokoli *i* ∈ *S* v konstatním čase. Pro sestavení $ISA_{x'}$ používá *difference cover* algoritmus *qsufsort* jako pomocnou rutinu (*qsufsort* sestavuje ISA i SA viz. Str. 10). Místo *qsufsort* můžeme použít jiný konstrukční algoritmus, který je vhodný pro omezené integer abecedy.

Jakmile máme $ISA_{x'}$, můžeme začít sestavovat SA_x. Všechny sufixy jsou *h*-řazeny za pomoci algoritmu pro řazení řetězců (př. MKQS). Jako výsledek dostáváme SA_h. Řazení zbylých nejednočlenných *h*-skupin je dokončeno za pomocí řadícího algoritmu na bázi porovnávání, jenž jako klíče, při porovnávání sufixů *i* a *j*, využívá $ISA_{x'}[\, i + \delta\,(i, j)]$ a $ISA_{x'}[\, j + \delta\,(i, j)]$.

V roce 2003 bylo dokázáno, že pokud zvolíme $h = \log_2 n$, dostaneme nejhorší časovou náročnost O *(n* log *n)*. další dobrou vlastností tohoto algoritmu je jeho malá paměťová náročnost.

Celkově vyžaduje v paměti méně než 6*n* bytů. Toto je možné díky *S*, které je v porovnání s **x** menší, a zároveň díky řazení in situ (inplace nebo-li na místě).

3 Qsufsort

V této sekci si podrobněji vysvětlíme tento algoritmus, neboť je implementován v přiložené aplikaci a obsahuje několik odlišností od základní vize popsané v postupu 1, které si popíšeme. Změny vychází z vědeckého článku o tomto algoritmu [6].

3.1 Základní algoritmus

1. Vložit sufixy reprezentovány čísly 0...*n* do SA. Seřadit sufixy za použití x_i jako klíče pro *i*. Nastavit $h = 1$.
2. Pro každé $i \in [0, n]$ nastavit ISA[*i*] na číslo skupiny sufixu *i*. Jedná se vlastně o poslední pozici v SA, která má sufix začínající stejným znakem jako sufix *i*.
3. Pro každou neseřazenou nebo seřazenou skupinu zabírající pole SA[*f*..*g*] nastavit L[*f*] na délku tohoto pole (v případě seřazené skupiny na zápornou hodnotu této délky).
4. Zpracovat každou neseřazenou skupinu v SA pomocí ternary-split quicksortu za použití ISA[*i*+*h*] jako klíče pro sufix *i*.
5. Označit pozice mezi nerovnými klíči v neseřazených skupinách.
6. Zdvojnásobit *h* (prefix doubling). Vytvořit nové skupiny rozdělením na označených pozicích z předchozího kroku a aktualizování SA a ISA.
7. Pokud se SA skládá z jedné plně seřazené skupiny, tak algoritmus končí. V opačném případě se jde na krok číslo 4.

Postup 1 – Algoritmus *qsufsort*

3.2 Úpravy algoritmu

Nyní se podíváme na úpravy v algoritmu, které snižují jeho časovou a paměťovou náročnost. Všechny jsou implementovány v přiložené aplikaci.

3.2.1 Eliminace pole L

První změna se týká pomocného pole L. Toto pole můžeme úplně eliminovat a tím ušetřit místo při konstrukci pole. Informace v tomto poli se používá pouze k nalezení pravých koncových bodů *h*-skupin v té fázi, kdy se procházejí a řadí *h*-skupiny. Díky tomu můžeme v konstantním čase přeskočit seřazené skupiny a pro neseřazené skupiny využíváme hodnotu tohoto pole jako parametr řadící procedury. Avšak koncový bod neseřazených skupin je znám přímo bez použití L, jelikož je roven číslu skupiny a může být přímo získáno z ISA. To dokazuje následující definice: číslo skupiny, která se nachází v podpoli SA[*f*...*g*], je rovno *g*.

Z toho nám vyplývá, že potřebujeme nalézt alternativní místo pro uložení délek seřazených v poli L. Mějme na paměti, že jakmile je sufix zařazen do seřazené skupiny, tak k jeho pozici v SA již nikdy nepřistoupíme. Proto můžeme použít podpole v SA, jež uchovává indexy seřazených sufixů použít k jiným účelům aniž bychom ohrozili správnost algoritmu.

Samozřejmě díky tomuto přepsání nemá při skončení algoritmu SA požadovaný výstup, jež je seřazené sufixové pole. Nicméně tento problém můžeme lehce obejít. Veškeré potřebné informace pro konstrukci sufixového pole nám obsahuje inverzní sufixové pole, které se sestavuje během řazení. Stačí pouze pro každé $i \in [0..n]$ přiřadit i do SA[ISA[i]].

3.2.2 Kombinace řazení a aktualizace

Po řazení se dle postupu 1 prochází zpracovávané části dvakrát (kvůli aktualizování informace v ISA a L). Tento průchod navíc se provádí jak při prvotním rozřazení, tak při každém řadícím průchodu přes body 4-7. Nyní si ukážeme jak lze toto procházení navíc eliminovat.

Prvně si povšimněme, že konkatenaci sousedních seřazených skupin, za účelem získání maximální seřazené kombinované skupiny, můžeme odložit a provést jako součást procházení a řazení (krok 4) následující iterace.

Zbylé aktualizování všech čísel skupin (jejich pozic v SA, které nám uchovává ISA) a délek ostatních skupin může být zahrnuto do řadící procedury TSQS. Tato změna vyžaduje hlubší zamyšlení, protože měnění čísel skupin některých sufixů může ovlivnit řadící klíče jiných sufixů. Proto musí být aktualizování čísla skupiny neseřazených skupin provedeno v takovém pořadí, že nikdy nenastane situace, kdy by byť dočasně bylo skupině přiřazeno nižší číslo skupiny než má nějaká skupina ve vyšší části SA. S využitím řadící procedury TSQS není problém tuto podmínku dodržet. Posloupnost akcí při řazení bude následující:

1. Rozdělíme podmnožinu na 3 části: menší než, rovno a větší než pivot.
2. Rekurzivně seřadíme menší část (myšleno podmnožinu prvků menších než pivot).
3. Aktualizujeme číslo číslo skupiny u prvků, které jsou rovny pivotu. Tyto prvky se nyní stávají samostatnou skupinou.
4. Rekurzivně seřadíme větší část (podmnožina prvků větší než pivot).

Čísla skupin uložené v ISA se nikdy nezvyšují (je zde nejpravější pozice skupiny viz. Kapitola 2.3.1) – rozdělování skupin vždy zahrnuje pouze snižování hodnot – díky tomu nám zůstávají řadící klíče konzistentní.

Tato změna může mít ale stále vliv na řadící proces, ovšem pouze pozitivní. Některé elementy teď mohou být přímo rozřazeny podle klíče, kterého by jinak dosáhly až po aktuálním cyklu. Tento efekt se může rozšířit do dalších skupin. Nicméně tato změna neovlivňuje nejhorší časovou složitost algoritmu, pouze se jedná o netriviální zlepšení časové složitosti pro určité vstupní data.

3.2.3 Transformace vstupu

Pokud předpokládáme, že vstupní abeceda je dostatečně malá tak, aby každý její symbol mohl být reprezentován jako nezáporný integer (což je nevalidní pouze pro pár strojových modelů), můžeme začít převádět prvky **x** do ISA a provést prvotní rozřazení v kroku 1 za použití ISA[i] jako klíče pro sufix i. To má určité potencionální výhody:

- Nastavením $h = 0$ můžeme použít přesně tu samou řadící proceduru jak pro prvotní rozřazení, tak i pro následné řadící kroky.
- Jelikož k **x** již nebude přistupováno, nemusíme ho mít v primární paměti během řazení. Pokud tedy nechceme **x** zachovat, můžeme pole řetězce **x** překrýt inverzním sufixovým polem ISA.
- Při přenosu symbolů z **x** do ISA může abeceda podstoupit jakoukoli transformaci pod podmínkou, že je zachováno pořadí mezi sufixy.

McIlroyova implementace algoritmu TSQS vyžaduje takovou transformaci abecedy, jenž reprezentuje unikátní terminální symbol $ nulou a originální symboly mapuje na integery v rozsahu <1, *k*), kde $k - 1$ je počet odlišných symbolů na vstupu. Tato transformace také usnadňuje úvodní bucket sort.

Nyní si ukážeme jak vytvořit abecedu ze které náš algoritmus může těžit i přesto, že nepoužijeme bucket sort. Po zbytek této sekce budeme uvažovat rozsah vstupních symbolů <*l, k*) pro nějaké *k* a *l*. Do tohoto rozsahu nebudeme započítávat terminální symbol $.

Možnost vnést explicitní reprezentaci terminálního symbolu ($) je malý, ale pozitivní efekt transformace abecedy. Nejjednodušší cesta jak tohoto dosáhnout je při přesouvání symbolů z **x** nastavit ISA[*i*] na $\mathbf{x}_i - l + 1$ pro všechna $i \in [0, n)$ a nastavit ISA[*n*] rovno nule. Od této chvíle se zbytek algoritmu nemusí starat o rozsah nebo limit abecedy.

a b d g x -> 1 2 3 4 5 0

97 98 100 103 120

Obr 3.1: Transformace vstupního řetězce (pod řetězcem jsou ASCII čísla znaků). 0 je vložený terminální symbol

Transformace může mít přímý vliv na časovou složitost. Toto je možné pokud je vstupní rozsah dostatečně malý a několik vstupních symbolů může být agregováno do jediného integeru. Nechť *K* označuje $k - l + 1$,velikost původní abecedy spolu s terminálním symbolem $, a nechť je *r* největší integer takový, že $K^r - 1$ může být uchováno v jednom počítačovém slově (4 byty). Nyní pro každé $i \in [0, n]$ nastavíme

$$ISA[i] := \sum_{j=1}^{r} x_{i+j-1} \cdot K^{r-j}$$

a kde definujeme $\mathbf{x}_i = 0$ pro $i \geq n$.

Díky tomu poté prvotní rozřazení, kde je jako klíč použito ISA[*i*], nebere v potaz pouze první symbol každého sufixu, ale bere v potaz prvních *r* symbolů. Proto následující řazení pomocí TSQS může začít s *h* rovno *r* místo toho, aby bylo *h* rovno jedné. Tím se vlastně sníží i počet potřebných průchodů při řazení.

Transformace může být provedena v lineárním čase nezávisle na *r* v této alternativní formě

$$V[i+1] := (V[i] \bmod K^{r-1}) \cdot K + x_{i+r}$$

pro $i > 0$. Pokud je K zaokrouhleno nahoru k nejbližší mocnině dvou, operace násobení a modula mohou být nahrazeny rychlejšími operacemi jako je logický posun a logický součin (shift a and).

Protože je *r* vysoce závislé na *K* a tím i tedy na *k* a *l* – limity rozsahu vstupní abecedy – je tedy velice výhodné před transformací co nejvíce zúžit tyto limity. Zkontrolovat minimální a maximální hodnotu symbolů, které se opravdu vyskytují na vstupu a nastavit podle nich *k* a *l* je jednoduchý úkol, jež může vést ke znatelnému zlepšení.

Další vylepšení lze zajistit v mnoha případech komprimováním abecedy ještě před transformací symbolů. Označme si symboly na vstupu $\Sigma = \{s_1, \ldots, s_{|\Sigma|}\}$ kde $s_i < s_j$ jedině pokud $i < j$. Nahrazením každého symbolu s_i jeho ordinálním číslem i nám dovoluje ustanovit $l = 0$ a $k = |\Sigma|$. Kdyby byla použita jen malá podmnožina povolené vstupní abecedy, výsledná hodnota r by mohla být v konečném důsledku větší než by byla jinak možná.

Nechť K_0 představuje povolený rozsah originální abecedy. Jestliže není K_0 příliš velké, může být přípravná komprimující transformace provedena efektivně s použitím pomocného pole o velikosti K_0. Toto pole může překrývat SA, protože to v této fázi zatím není potřeba. Pozice v poli odpovídající použitým symbolům jsou označeny a ordinální čísla se poté akumulují ve stejném poli. Časová náročnost je $O(n + K_0)$.

3.2.4 Bucket sort

První řadící krok je poněkud odlišný od zbytku řadící procesu algoritmu a není nutné pro něj použít tu samou proceduru jako v pozdějším řazení. Toto první rozřazení musí zpracovat všechny vstupní znaky v jediném řadícím cyklu. Proto zde můžeme dosáhnout znatelného zrychlení pokud použijeme bucket sort, který má lineární časovou složitost, místo TSQS jehož složitost je $O(n \log n)$.

V této fázi pole SA neobsahuje žádná data. A proto pokud je tedy vstupní abecedy v rozsahu maximálně $n + 1$, můžeme použít SA jako pomocné pole při bucket sortu bez použití speciálního místa navíc. Pokud je vstupní abeceda větší jak $n + 1$ a nemůže být snad přetransformována, nemůžeme tuto techniku použít. V praxi se s tímto ovšem setkáme velice zřídka a to v případě pro velmi malé n, kdy je ovšem poněkud zbytečné používat sofistikovaný řadící algoritmus.

Ještě znatelnějšího vylepšení dosáhneme když zkombinujeme bucket sort a transformace vstupní abecedy popsané v kapitole 3.1.3. V tomto případě při volbě r (počet originálních vstupních symbolů seskupených do jednoho) požadujeme aby v jednom počítačovém slově nebylo uchováno jen maximálně $K^r - 1$, ale aby v něm bylo i maximálně n. Výsledná transformovaná abeceda tak v konečném důsledku může být větší než původní, ale pořád umožňuje použít bucket sort bez využití přídavného místa. Nakonec tak můžeme provést první rozřazení nejen podle prvního znaku, ale rovnou podle prvních r symbolů každého sufixu. Tím obvykle značně ulevíme hlavní řadící proceduře.

3.3 Praktické výsledky a popis knihovny

Implementovaná knihovna qsufsort umožňuje uživateli využívat dvě funkce:

1. Sestavení sufixového pole. Vstupem této funkce je vstupní řetězec, délka tohoto řetězce, maximální a minimální vstupní symbol. Výstupem je sufixové a inverzní sufixové pole.
2. Přesné vyhledání vzoru v řetězci (ať už DNA či v klasickém textu). Vstupem je sufixové pole, řetězec a jeho délka, hledaný vzor a délka tohoto vzoru. Výstupem této funkce je interval v sufixovém poli, kde se tento vzor vyskytuje (dva indexy start a end).

Další knihovna qgram umožňuje uživateli přibližné vyhledání vzoru v DNA sekvenci, které je nutné předat dalšímu algoritmu (např. BLAST). Tato problematika je popsána v kapitole 4.2. Vstupem této funkce přibližného vyhledání je maximální dovolená editační vzdálenost, hledaný vzor a jeho délka, sufixové pole, řetězec a délka tohoto řetězce. Výstupem je seznam bloků s možným výskytem tohoto vzoru.

Následující tabulka ukazuje výsledky implementovaného řadícího algoritmu qsufsort:

Testovací data	Doba sestavení SA
DNA 33 MB (chromozom 22)	13.97s
HowTo 37,6 MB	22.33s
JDK13c 66,5 MB	53.67s

4 Využití v bioinformatice

Nechť *S* je text délky *n* a *P* je vyhledávaný vzor délky *m.* Oba řetězce jsou definovány nad konečnou abecedou znaků *A.* Problém vyhledávání řetězců spočívá v nalezení všech výskytů *P* v *S,* které splňují určité kritéria. V závislosti na kritériu rozlišujeme 3 rozdílné problémy.

1. Přesné vyhledání vzorů (exact string matching), jež vyhledává všechny přesné výskyty vzoru *P* v *S.*
2. *K*-mismatch problém spočívá ve vyhledání všech výskytů *P* v *S,* které mají Hammingovu vzdálenost (Hamming distance) od *P* rovnu maximálně *k.*
3. K-difference problém spočívá v nalezení všech výskytů P v S, které mají editační vzdálenost (edit distance) od P rovnu maximálně k.

Edit distance mezi dvěma řetězci je definována jako počet vložení, umazání a nahrazení znaků při převodu jednoho řetězce na druhý. Pokud je dovoleno znaky pouze nahrazovat dostáváme Hammingovu vzdálenost.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

Obr 4.1: Algoritmus dynamického programování pro výpočet edit distance mezi řetězci survey a surgery. Tučné položky označují cestu ke konečnému výsledku. [8]

Vyhledávání se dále dělí na offline a online. Online vyhledávání nemá řetězec *S* nijak předzpracovaný a tudíž není předem znám. Pro přesné vyhledávání se zde používají velmi známé algoritmy - Boyer-Mooreův a Knuth-Morris.Prattův. Problém přibližného vyhledávání je řešen za pomoci dynamického programování s časovou složitostí *O(mn).* Landau a Vishkin tuto časovou složitost ještě zlepšili na *O(kn).* Online vyhledávání ovšem není téměř možné použít pro zpracovávání enormně velkých dat jako kupříkladu sekvence DNA. Takových případech se používá offline vyhledávání. U offline varianty je text *T* znám předem a je provedeno jeho předzpracování v podobě

sestavení rejstříku. Vyhledávání je pak velice rychlé. Jako rejstříky se nejčastěji používají sufixové pole nebo sufixové stromy. V našem případě se zaměříme na vyhledávání sekvencí DNA nad sufixovým polem.

4.1 Vyhledání přesných vzorů

Jedním z mnoha možných použití sufixových polí je vyhledávání přesných vzorů. Jedná se vlastně o dotazy typu zda je podřetězec *P* součástí řetězce *S*. Jelikož je sufixové pole lexikograficky seřazené je zde možné použít velice rychlé binární vyhledávání. Nechť *m* je délka vyhledávaného podřetězce *P* a *n* je délka řetězce *S*, pak vyhledání informace o tom, zda se vzor *P* nachází v *S* trvá maximálně $O(m \log n)$. Pro každý znak vzoru musíme provést binární vyhledání, které má nejhorší časovou složitost $O(\log n)$ a proto tedy $m \log n$. Pokud máme k dispozici lcp pole můžeme tuto složitost redukovat na $O(m + \log n)$.

Algoritmus pro vyhledání všech *P* v *S*:

Algoritmus 4:

Vstupem algoritmu je hledaný podřetězec, sufixové pole a řetězec **x** (zde uvedený jako S). Výstupem je seznam indexů sufixového pole s výskytem daného vzoru.

```
find (Pattern P in SuffixArray SA):
    i = 0
    lo = 0, hi = length(SA)
    for i = 0; i < length(P); i++:
        Binary search for x, y
        where P[i] == S[SA[j] + i] for lo <= x <= j < y <= hi
        lo = x, hi = y
    return {SA[lo], SA[lo+1], …., SA[hi-1]}
```

Algoritmus 4 popisuje pseudokód algoritmu pro vyhledání všech vzorů *P* v řetězci *S*. Po znaku procházíme vzor *P* a pomocí binárního vyhledávání hledáme interval <x, y> ve kterém se vyskytuje část tohoto vzoru P[0..*i*].

Ukážeme si postup na příkladě. Mějme řetězec *S* = *mississippi* (tabulka 4.1) a vzor pro vyhledání *P* = *is*. V prvním průchodu vlastně hledáme v celém sufixovém poli interval sufixů jež začínají písmenem *i*. *X* určuje první sufix, který začíná písmenem *i*, a *y* určuje poslední sufix v SA, který začíná písmenem *i*. Po prvním průchodu tedy dostáváme $x = 0$ a $y = 3$. Poté snížíme hranice vyhledávání. Další znak už nebudeme hledat ve všech sufixech, ale pouze v intervalu, který získáme v prvním kroku. Nyní hledáme druhé písmeno vzoru což je *s* hledáme ho na druhém místě sufixů 0-3. Z tabulky níže vidíme, že *s* se vyskytuje na druhém místě u sufixů 2 a 3, jež jsou ve výsledku naším hledaným intervalem.

0	11	i$
1	8	ippi $
2	5	issippi $
3	2	ississippi $
4	1	mississippi $
5	10	pi$
6	9	ppi $
7	7	sippi $
8	4	sissippi $
9	6	ssippi $
10	3	ssissippi $
11	12	$

Tabulka 4.1: Sufixové pole pro řetězec S = mississippi$

4.2 Přibližné vyhledávání - Quasar

V molekulární biologii se poslední dobou stalo vyhledání podobných sekvencí DNA základní operací. Algoritmy jako je BLAST jsou sice velice rychlé, ale dosahují svých limitů pokud se pokoušejí o porovnání všech-proti-všech (all-versus-all) nad velkými databázemi. Nyní si ukážeme jak se tento problém dá s pomocí sufixového pole překonat. Algoritmus nazvaný QUSAR (Q-gram Alignment based on Suffix Array) byl vyvinut za účelem rychlého vyhledání sekvencí se silnou podobností. QUASAR používá q-tuple filtrování založené na sufixovém poli. To znamená, že QUASAR prvně vyfiltruje všechny možné pozice se silnou podobností a poté předá tyto možné pozice shody dalšímu algoritmu, který otázku podobnosti vyřeší přesně. Jako tento další algoritmus se velice často používá právě BLAST, který dokáže tyto jednotlivé podproblémy řešit velice rychle. Pokud by byl BLAST použit na celou úlohu bez použití filtrování, byl by o mnoho pomalejší [7]. Nicméně v této práci je implementována pouze první část a to vyhledání pravděpodobně shodných sekvencí.

4.2.1 Editační vzdálenost a podobnost

V této sekci se podíváme na termín silná podobnost a editační vzdálenost. Pod silnou podobností si můžeme představit, že dva řetězce jsou téměř identické např. „ABCDEF“ a „ABXDEF“. Silná podobnost závisí pouze na malém počtu rozdílných znaků. K určení zda jsou si dva řetězce silně podobné nebo ne potřebujeme měřit podobnost trochu hlouběji. Pro tento účel si zavedeme editační vzdálenost. Jak již bylo řečeno editační vzdálenost udává minimální počet vložení, umazání a změnění jednoho znaku, jež je potřeba provést v řetězci s_1 abychom z něj dostali řetězec s_2.

Příklad 4.1:

s_1 = ACTAT, s_2 = ATTAA

Budeme transformovat s_1 na s_2:

s_1	ACTAT
1. změna (nahrazení C za T)	A**T**TAT
2. změna (nahrazení T za A)	ATTA**A** = s_2

Dvě změny jsou zároveň nejmenší počet potřebných změn s_1 získání s_2. Proto je v tomto případě editační vzdálenost rovna 2.

Příklad 4.2:

s_1 = ATTGC, s_2 = TTGGCA

s_1	ATTGC
1. změna (smazání A)	TTGC
2. změna (přidání G)	TTG**G**C
3. změna (přidání A)	TTGGC**A**

Tři změny jsou zároveň nejmenší počet potřebných změn s_1 získání s_2. Proto je v tomto případě editační vzdálenost rovna 3.

4.2.2 q-gram filtrování

Vypočítávat editační vzdálenost mezi dvěma řetězci je příliš náročné. QUASAR toto dělá přibližně za pomoci tzv. q-gram filtrování. Q-gram v našem kontextu odpovídá subsekvenci s slova w s definovanou délkou $|s| = q$.

Příklad 4.3:

Definujme q = 2

w_1 = VESLO má q-gramy: VE ES SL LO

w_2 = HESLO má q-gramy: HE ES SL LO

Abychom zjistili podobnost mezi dvěma řetězci zajímají nás q-gramy, které mají řetězce společné. V příkladě 4.3 máme dvě slova w_1 = VESLO a w_2 = HESLO, jenž mají společné 3 q-gramy a to ES, SL a LO.

Pro získání přibližné editační vzdálenosti nás zajímá vztah mezi počtem společných q-gramů a editační vzdáleností mezi dvěma řetězci. K získání prahu t, který popisuje minimální počet společných q-gramů dvou řetězců P a S dané délky w pro zadanou editační vzdálenost k, můžeme použít následující tvrzení.

Tvrzení 4.1:

Nechť P a S jsou řetězce dané délky w a zadané editační vzdálenosti k. Pak P a S mají nejméně $t = w + 1 – (k + 1)q$ společných q-gramů. [9]

Příklad 4.4:

P = ACAGCTTA a S = ACACCTTA, $w = |P| = |S| = 8$

Jak můžeme vidět, editační vzdálenost je $k = 1$

definujme si $q = 3$

→ $t = w + 1 – (k + 1)q = 8 + 1 – (1 + 1)\ 3 = 9 – 6 = 3$ (společné q-gramy jsou ACA, CTT, TTA).

Je důležité si uvědomit, že t nám udává pouze minimální nutný počet společných q-gramů P a S. Podívejme se na příklad velice podobný předchozímu:

Příklad 4.5:

P = ACACTTAG a S = ACACTTAC, $w = |P| = |S| = 8$

editační vzdálenost je stále $k = 1$ a $q = 3$

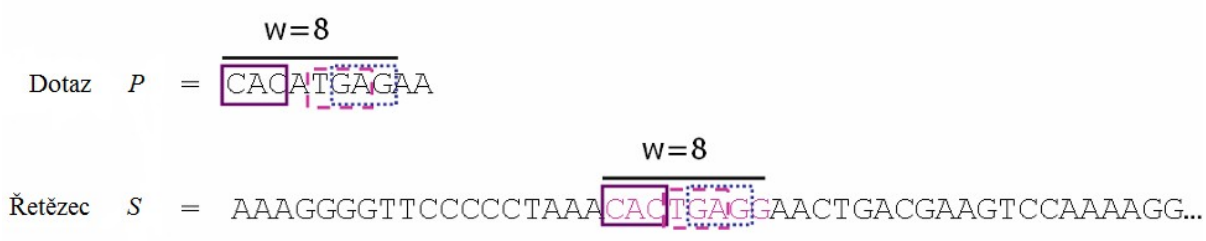
→ $t = w + 1 – (k + 1)q = 8 + 1 – (1 + 1)\ 3 = 9 – 6 = 3$ (opět stejné jako u příkladu 4.4)

Rozdíl zde je v počtu společných q-gramů. Zatímco výše byli 3, zde jich je 5: ACA, CAC, ACT, CTT, TTA.

Tím jsme problém nalezení všech přibližných shod v řetězci S na tuto úlohu: Pro zadané k maximálních rozdílů nad fixní velikostí okna w nalezneme přibližné shody nalezením subsekvencí v řetězci s minimálně t shodnými q-gramy. Pro upřesnění si ukážeme příklad.

Příklad 4.6:

mějme $k = 1$, $w = 8$, $j = 3$ → $t = w + 1 – (k + 1)q = 3$



Myšlenka je taková: projdeme všechny sekvence délky w v P a pro každé z těchto oken zkontrolujeme všechny sekvence délky w v řetězci S, abychom zjistili počet společných q-gramů. Pokud najdeme subsekvenci délky w, která má minimálně t společných q-gramů, nalezli jsme hledanou přibližnou shodu. Začneme prvním oknem o velikosti $w = 8$ v dotazu $P_{1,w}$ (jedná se o podřetězec v P začínající na pozici 1 a končící na pozici w). V našem příkladě nalezneme v S okno CACTGAGG se společnými q-gramy {CAC, TGA, GAG}. Počet společných q-gramů je $t = 3$. Tato subsekvence je tedy možná shoda pro edit distance $k = 1$, ovšem jak můžeme vidět v tomto případě tomu tak není. V dalším kroku bychom tuto subsekvenci předali algoritmu BLAST.

Nyní tedy potřebujeme vyhledat pro každé okno v P $(P_{1,w}, P_{2,w+1}, \ldots)$ všechny možné přibližné shody v řetězci S. Abychom toto dělali trochu chytřeji, uděláme další aproximaci.

4.2.3 Počítání a rozdělování

Naším cílem teď je identifikovat všechny podřetězce v *S*, které mají společných alespoň *t* q-gramů s $P_{1,w}$. Nejjednodušší přístup by byl přidat počítadlo ke každému podřetězci délky *w* v *S*. V tom případě bychom ale potřebovali přibližně |S| - *w* + 1 počítadel a museli bychom zvyšovat všechny počítadla těch podřetězců, které obsahují společné q-gramy s $P_{1,w}$. Všechny podřetězce s počítadlem vyšším nebo rovno *t* jsou přibližné shody.

Největší nevýhodou tohoto řešení je paměť potřebná pro počítadla. Paměťové nároky můžeme snížit tím, že zkombinujeme více podřetězců do jednoho bloku a tomuto bloku přiřadíme pouze jedno počítadlo. Tímto sice dosáhneme menších nároků na paměť, ale zároveň snížíme citlivost algoritmu.

Řetězec *S* si rozdělíme na bloky *b*, které se vzájemně nepřekrývají, o velikosti $b \geq 2w$.

Příklad 4.7:

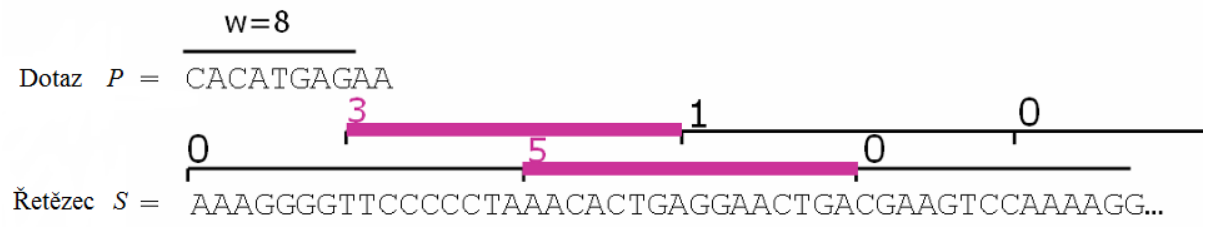
k = 1, *w* = 8, *q* = 3 → *t* = *w* + 1 – (k + 1)q = 3, *b* = 16 *(b* ≥ 2*w)*

V druhém bloku nalezneme 5 společných q-gramů {CAC, ACA, TGA, TGA, GAG} mezi dotazem *P* a řetězcem *S*.

Protože se tyto bloky vzájemně nepřekrývají, může se nám stát, že mineme q-gramy, které jsou přesně na hranicích bloků. Proto použijeme druhé pole bloků, jež je posunuto o *b*/2. Pokud bychom v příkladě výše hledali q-gram TAA, tak bychom ho přeskočili (na hranicích prvního a druhého bloku). Ze stejného důvodu musíme zvolit $b \geq 2w$. Ukážeme si další příklad pro úplnost.

Příklad 4.8:

k = 1, *w* = 8, *q* = 3 → *t* = *w* + 1 – (k + 1)q = 3, *b* = 16 *(b* ≥ 2*w)*

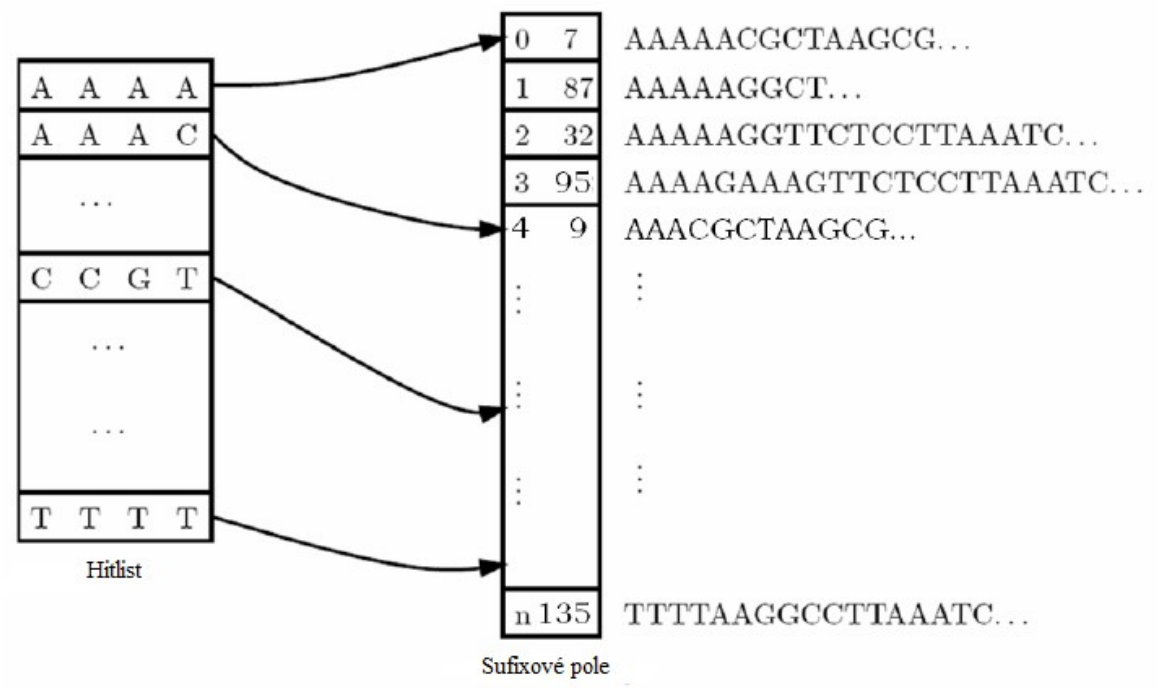


Nyní by se dále do algoritmu BLAST předávali bloky s počítadlem 3 a 5. Kvůli tvrzení 4.1 bloky s počítadlem menším *t* (v tomto případě *t* = 3) nemohou obsahovat podřetězec *w* délky s editační vzdáleností *k* = 1, oproti $P_{1,w}$, a tak je tedy nebudeme předávat do BLASTu.

4.2.4 Hitlist

Do této chvíle jsme neřešili vcelku zásadní problém efektivity a to jak najít q-gramy v řetězci *S*. Momentálně prakticky bereme každý q-gram z okna $P_{1,w}$ a pro každý q-gram procházíme *S* a zvyšujeme počítadla na blocích, které tento q-gram obsahují. Abychom předešli tomuto zdlouhavému procházení celého řetězce *S*, sestavíme si hitlist, jenž je vlastně indexem pro získání všech pozic v řetězci na nichž se daný q-gram vyskytuje.

Hitlist je seznam všech možných q-gram ukazující na jeho první výskyt v sufixovém poli řetězce *S*. Hitlist můžeme implementovat jako hashovací tabulku, kde klíč je daný q-gram a ukládaná hodnota klíče je index prvního výskytu v sufixovém poli.



Obr 4.2: Ukázka hitlistu pro q = 4

Ve snaze získat pozici q-gramu, např. AAAA, se podíváme do hitlistu na záznam AAAA a index u tohoto klíče slouží jako pozice s prvním výskytem q-gramu v sufixovém poli. Následné projdeme od této pozice sufixové pole směrem dolů, dokud nenarazíme na záznam v sufixovém poli kam ukazuje další záznam v hitlistu. Tato technika nám umožňuje nalézt pozici q-gramu v konstantním čase. Představme si ale kdyby se řetězec *S* skládal pouze z jednoho znaku. Pak by nám vyhledání jednoho q-gramu degradovalo na *O(n) (n* bereme jako délku řetězce *S*). Další důležitá poznámka se týká velikosti hitlistu. Jeho velikost je rovna počtu symbolů v abecedě umocněném *q* (délka q-gramů). V tomto případě se jedná o $4^4 = 256$ záznamů v hitlistu. Pokud bychom chtěli použít stejnou techniku pro vyhledávání v běžném ASCII textu bylo by zapotřebí 256^4 záznamů což dělá 4GB paměti pouze na hitlist.

4.2.5 Proces sčítání

Pro nalezení všech přibližných shod mezi hledaným oknem $P_{1,w}$ a S musíme najít všechny podřetězce S, který sdílejí alespoň t q-gramů s $P_{1,w}$. Poté posuneme okno v P a hledáme společné q-gramy pro $P_{2,w+1}$ a tak dále, dokud nedorazíme na konec hledaného řetězce P.

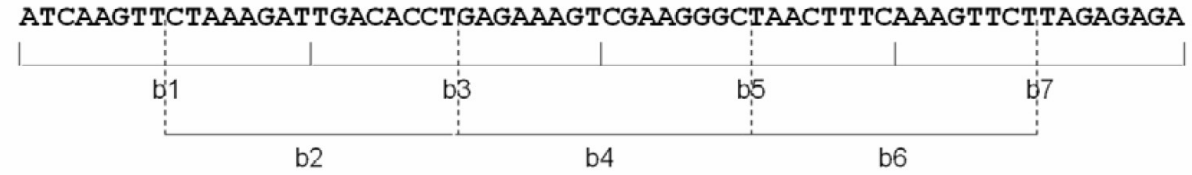
Abychom předešli celé vyhledávací proceduře všech q-gramů pro $P_{1,w,}$ $P_{2,w+1}$, … atd., můžeme pouze zpracovat rozdíl mezi aktuálním oknem a předcházejícím oknem. Např. Pokud chceme kontrolovat okno $P_{2,w+1}$, stačí nám vzít výsledek $P_{1,w}$ a zpracovat pouze rozdíly mezi nimi. Takže zahodíme výsledek $P_{1,q}$ (první q-gram, který nyní není v $P_{2,w+1}$) a zvýšíme čítače pro nový q-gram $P_{w-q+1,w+1}$. Celkově tedy musíme snížit čítače bloků, které zahrnovaly $P_{1,q}$ a nedosáhli prahu t a zvýšit čítače bloků, jež obsahují nový q-gram $P_{w-q+1,w+1}$.

Tímto způsobem projdeme celé P. Na konci získané bloky s čítačem vyšším nebo rovno t předáme do algoritmu BLAST a získáme přibližné shody.

Příklad 4.9:

P = CACATGAGAA

S =



Zvolíme-li w = 8, máme první část CACATGAG a q-gramy {CAC, ACA, CAT, ATG, TGA, GAG}

Bloky máme:

Blok	Vyskytující se q-gramy	Shodných q-gramů
b1		
b2	CAC, ACA, TGA	3 ≥ t
b3	CAC, ACA, TGA, GAG	4 ≥ t
b4	GAG	1
b5		
b6		
b7	GAG	1

Nyní okno posuneme jednu pozici doleva a dostáváme ACATGAGA. Blokům, které obsahují q-gram CAC snížíme jejich čítač o jedna a blokům, jenž obsahují q-gram AGA naopak čítač zvýšíme. Tím vlastně zpracujeme pouze rozdíl mezi prvním a druhým oknem. Z tabulky níže vidíme, že u bloků 2 a 3 by se čítač snižoval, kdyby v první části neměli počet shodných q-gramů větší nebo roven t.

Blok	Vyskytující se q-gramy	Shodných q-gramů
b1	AGA	1
b2	CAC, ACA, TGA, AGA	$3 - \cancel{1} + 1 \geq t$
b3	CAC, ACA, TGA, GAG, AGA	$4 - \cancel{1} + 1 \geq t$
b4	GAG, AGA	1 + 1
b5		
b6		
b7	GAG, AGA	1 + 1

Poslední okno je CATGAGAA. Nyní se snižuje čítač blokům obsahující q-gram ACA a zvyšuje se čítač blokům s q-gramem GAA.

Blok	Vyskytující se q-gramy	Shodných q-gramů
b1	AGA	1
b2	CAC, ACA, TGA, AGA	$4 - \cancel{1} \geq t$
b3	CAC, ACA, TGA, GAG, AGA, GAA	$5 - \cancel{1} + 1 \geq t$
b4	GAG, AGA, GAA	2 + 1
b5	GAA	1
b6		
b7	GAG, AGA	2

5 Závěr

Sufixové pole je tedy jednou z možných cest, jak analyzovat biologické sekvence a indexovat rozsáhlé databáze textu. Běžné vyhledávání pomocí algoritmů BM (Boyer-Mooreův) nebo KMP (Knuth-Morris-Prathův) by nad těmito velkými sekvencemi trvalo velmi dlouho. Alternativou k sufixovým polím je sufixový strom, ale má i několikanásobně vyšší paměťové požadavky než základní sufixové pole.

Implementovaný algoritmus konstrukce sufixového pole má časovou složitost $O(n \log n)$ a v paměti zabírá $9n$ místa. $4n$ sufixové pole, $4n$ pro inverzní sufixové pole a n pro samotný řetězec, v němž je prováděno následné vyhledávání.

Tato práce skýtá několik možností na rozšíření.

1. Vytvořit nejen obyčejné sufixové pole, ale přidat i možnost vytvoření rozšířeného sufixového pole a tak dosáhnout plné ekvivalentnosti vůči sufixovému stromu.
2. Implementovat a podrobněji popsat další řadící algoritmy, jež jsou rychlejší a lepší než zde použitý qsufsort.
3. Implementovat BLAST (Basic Local Alignment Search Tool) jako doplněk metody QUASAR a tím zajistit plnohodnotné přibližné vyhledávání.
4. Demonstrovat použití sufixového pole na dalších příkladech. Např. tandemové opakování, nejkratší unikátní podřetězec, statistiky vyhledávání atd.

Literatura

[1] Manber, U. and Myers, G.W. 1990. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 319–327.

[2] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. June 2007. A taxonomy of sux array construction algorithms. *ACM Computing Surveys, 39(2).*

[3] Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *J. Disc. Algor. 2*, 1, 53–86.

[4] Schürmann, Klaus-Bernd. Suffix Arrays in Theory and Practice. Bielefeld, 2007. 140 s. Diplomová práce. University Bielefeld, Faculty of Technology, Germany.

[5] Manzini, G. and Ferragina, P. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica* 40, 33–50.

[6] Larsson, J. N. and Sadakane, K. 1999. Faster suffix sorting. Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS- 3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden.

[7] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. 1999 . q-gram Based database Searching Using a Suffix Array (QUASAR). *RECOMB*, 77–83.

[8] G. Navarro. A guided tour to approximate string matching. 1999. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile. *ACM Computing Surveys*

[9] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. 1991. *Proc. of the 16th Symposium on Mathematical Foundations of Computer Science, volume 520 of Lecture Notes in Computer Science*, 240-248.

Seznam příloh

Příloha 1. CD s testovacími daty a zdrojovými kódy.