1. Vyhledávání v textu

Uvažujme následující úlohu: máme nějaký text σ délky S (budeme mu říkat *seno*) a chceme v něm najít všechny výskyty nějakého podřetězce ι délky J (*jehly*). Seno přitom bude řádově delší než jehla.

Kupříkladu v seně `bananas` se jehla `ana` vyskytuje hned dvakrát, přičemž výskyty se překrývají.

Triviální řešení přesně podle definice by vypadalo následovně: Zkusíme všechny možné pozice, kde by se v seně mohla jehla nacházet, a pro každou z nich otestujeme, zda tam opravdu je. Pozic je řádově S, každé porovnání stojí až J, celkově tedy algoritmus běží v čase $\mathcal{O}(SJ)$. (Rozmyslete si, jak by vypadaly vstupy, pro které skutečně spotřebujeme tolik času – viz cvičení 1.5.1.)

Zkusme jiný přístup: nalezneme v seně první znak jehly a od tohoto místa budeme porovnávat další znaky. Pokud se přestanou shodovat, přepneme opět na hledání prvního znaku. Jenže odkud? Pokud od místa, kde nastala neshoda, selže to třeba při hledání jehly `kokos` v seně `clanekokokosu` – neshoda nastane za `koko` a zbylý `kos` nás neuspokojí. Nebo se můžeme vrátit až k výskytu prvního znaku a pokračovat těsně za ním, jenže to je totéž, co dělal triviální algoritmus, takže je to také stejně pomalé.

V této kapitole si ukážeme algoritmus, který je o trochu složitější, ale nalezne všechny výskyty v čase $\mathcal{O}(S + J)$. Pak ho zobecníme, aby uměl hledat více různých jehel najednou.

1.1. Řetězce a abecedy

Aby se nám o řetězcových algoritmech lépe psalo, udělejme si nejprve pořádek v terminologii okolo řetězců.

Definice:

- *Abeceda* Σ je nějaká konečná množina, jejím prvkům budeme říkat *znaky* (někdy též *písmena*).
- Σ^* je množina všech *slov* neboli *řetězců* nad abecedou Σ, což jsou konečné posloupnosti znaků ze Σ.

Příklady: Abeceda může být tvořena třeba písmeny `a` až `z` nebo bity `0` a `1`. Potkáme ovšem i rozlehlejší abecedy: například dnes běžná znaková sada UniCode má $2^{16} = 65\,536$ znaků, v novějších verzích dokonce $2^{31} \approx 2 \cdot 10^9$ znaků. Ještě extrémnějším způsobem používají řetězce lingvisté: na český text se někdy dívají jako na řetězec nad abecedou, jejíž znaky jsou česká slova.

Pro naše účely budeme předpokládat, že abeceda je „rozumně malá“, čímž myslíme, že její velikost je konstantní a navíc dostatečně malá na to, abychom si mohli dovolit ukládat do paměti pole indexovaná znakem.

Značení:

- *Slova* budeme značit malými písmenky řecké abecedy α, β, ...
- *Znaky* abecedy označíme malými písmeny latinky `a`, `b`, ...
  Znak budeme používat i ve smyslu jednoznakového řetězce.
- *Délka slova* $|\alpha|$ udává, kolika znaky je slovo tvořeno.
- *Prázdné slovo* značíme písmenem ε, je to jediné slovo délky 0.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $|\alpha\beta| = |\alpha| + |\beta|$, $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- $\alpha[k]$ je k-tý znak slova α, indexujeme od 0 do $|\alpha| - 1$.
- $\alpha[k : \ell]$ je *podslovo* začínající k-tým znakem a končící těsně před ℓ-tým. Tedy $\alpha[k : \ell] = \alpha[k]\alpha[k+1]\ldots\alpha[\ell-1]$. Pokud $k \geq \ell$, je podslovo prázdné. Pokud některou z mezí vynecháme, míní se $k = 0$ nebo $\ell = |\alpha|$.
- $\alpha[: \ell]$ je *prefix* (předpona) tvořený prvními ℓ znaky řetězce.
- $\alpha[k :]$ je *suffix* (přípona) od k-tého znaku do konce řetězce.
- $\alpha[:] = \alpha$.

Dodejme ještě, že každé slovo je prefixem sebe sama a prázdné slovo je prefixem každého slova. Pokud budeme hovořit o *vlastním* suffixu, budeme tím myslet suffix různý od celého slova. Analogicky pro prefixy a podslova.

1.2. Inkrementální algoritmus

Vraťme se nyní zpět k původnímu problému hledání podřetězců. Nejprve si ujasněme, co má být výstupem algoritmu. Budeme chtít nalézt množinu všech indexů k takových, že $\sigma[k : k + |\iota|] = \iota$. To je dostatečně kompaktní výstup (nejvýše lineární s délkou sena), a přitom obsahuje informace o všech výskytech.

Na hledání podřetězce použijeme *inkrementální přístup.* Tím se obecně myslí, že chceme umět rozšířit vstup o další znak a přepočítat výstup. V našem případě budeme přidávat znak na konec sena a započítáme případný nový výskyt jehly, který končí tímto znakem.

Abychom toho dosáhli, budeme si průběžně udržovat informaci o tom, jakým nejdelším prefixem jehly končí zatím přečtená část sena. Tomu budeme říkat *stav algoritmu.* A jakmile bude tento prefix roven celé jehle, ohlásíme výskyt.

V našem „kokosovém“ příkladě se tedy po přečtení sena `clanekoko` nacházíme ve stavu `koko`, následují stavy `kok`, `koko` a `kokos`.

Představme si nyní obecně, že jsme přečetli řetězec σ, který končil stavem α. Teď vstup rozšíříme o znak x na σx. V jakém stavu se nyní máme nacházet? Pokud to nebude prázdný řetězec, musí končit na x, tedy ho můžeme napsat ve tvaru $\alpha' x$.

Všimneme si, že α' musí být suffixem slova α: Jelikož $\alpha' x$ je prefix jehly, je α' také prefix jehly. A protože $\alpha' x$ je suffixem σx, musí α' být suffixem σ. Tedy jak α,

tak α' jsou suffixy slova α, které jsou současně prefixy jehly. Ovšem stav α jsme vybrali jako nejdelší slovo s touto vlastností, takže α' musí být nejvýše tak dlouhé, a tudíž je prefixem α.

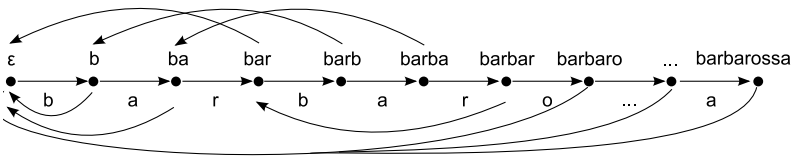
Stačilo by tedy probrat všechny suffixy slova α, které jsou prefixem jehly, a vybrat z nich nejdelší, který po rozšíření o znak x stále je prefixem jehly.

Abychom ale nemuseli suffixy procházet všechny, předpočítáme si *zpětnou funkci* z. Ta nám pro každý prefix jehly řekne, jaký je jeho nejdelší vlastní suffix, který je opět prefixem jehly. To nám umožní procházet rovnou kandidáty na nový stav: stačí probrat řetězce α, $z(\alpha)$, $z(z(\alpha))$, ... a použít první, který lze rozšířit o znak x. Pokud nepůjde rozšířit ani jeden z těchto kandidátů, novým stavem bude prázdný řetězec.

Na této myšlence je založen následující algoritmus.

1.3. Knuthův-Morrisův-Prattův algoritmus

Algoritmus se opírá o *vyhledávací automat.* To je orientovaný graf, jehož vrcholy (*stavy* automatu) odpovídají prefixům jehly. Vrcholy jsou spojeny hranami dvou druhů: *dopředné* popisují rozšíření prefixu přidáním jednoho písmene, *zpětné* vedou podle zpětné funkce, tedy z každého stavu do jeho nejdelšího vlastního suffixu, který je opět stavem.



Obr. 1.1: Vyhledávací automat pro slovo `barbarossa`

Reprezentace automatu bude přímočará: stavy očíslujeme od 0 do J, dopředná hrana povede vždy ze stavu s do $s+1$ a bude odpovídat rozšíření prefixu o příslušný znak jehly, tedy o $\iota[s]$. Zpětné hrany si budeme pamatovat v poli Z, tedy $Z[s]$ bude říkat číslo stavu, do nějž vede zpětná hrana ze stavu s, případně bude nedefinované, pokud taková hrana neexistuje.

Kdybychom takový automat měli, mohli bychom pomocí něj inkrementální algoritmus z předchozí sekce popsat následovně:

Procedura KmpKrok *(Jeden krok automatu)*
Vstup: Jsme ve stavu s, přečetli jsme znak x.

1. Dokud $\iota[s] \neq x$ & $s \neq 0 : s \leftarrow Z[s]$.
2. Pokud $\iota[s] = x$, pak $s \leftarrow s + 1$.

Výstup: Nový stav s.

Algoritmus KmpHledej *(Spuštění automatu na řetězec σ.)*
Vstup: Seno σ, zkonstruovaný automat.

1. $s \leftarrow 0$.
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow \text{KmpKrok}(s, x)$.
4. $\quad$ Pokud $s = J$, ohlásíme výskyt.

Invariant: Stav algoritmu s v každém okamžiku říká, jaký nejdelší prefix jehly je suffixem zatím přečtené části sena. (To už víme z úvah o inkrementálním algoritmu.)

Důsledek: Algoritmus ohlásí všechny výskyty. Pokud jsme totiž právě přečetli poslední znak nějakého výskytu, je celá jehla suffixem zatím přečtené části sena, takže se musíme nacházet v posledním stavu.

Jen musíme opravit drobnou chybu – těsně poté, co ohlásíme výskyt, se algoritmus zeptá na dopřednou hranu z posledního stavu. Ta ale neexistuje. Napravíme to jednoduše: přidáme fiktivní dopřednou hranu, na níž je napsán znak odlišný od všech skutečných znaků. Tím zajistíme, že se po této hraně nikdy nevydáme. Stačí tedy vhodně dodefinovat $\iota[J]$.[1]

Lemma: Funkce *Hledej* běží v čase $\mathcal{O}(S)$.

Důkaz: Výpočet funkce můžeme rozdělit na průchody dopřednými a zpětnými hranami. S dopřednými je to snadné – za každý z S znaků sena projdeme po nejvýše jedné dopředné hraně. To o zpětných hranách neplatí, ale pomůže nám, že každá dopředná hrana vede o právě 1 stav doprava a každá zpětná o aspoň 1 stav doleva. Proto je všech průchodů po zpětných hranách nejvýše tolik, kolik jsme prošli dopředných hran, takže také nejvýše S. □

Konstrukce automatu

Hledání tedy pracuje v lineárním čase, zbývá domyslet, jak v lineárním čase sestrojit automat. Stavy a dopředné hrany získáme triviálně, se zpětnými budeme mít trochu práce.

Podnikneme myšlenkový pokus: Představme si, že automat už máme hotový, ale nevidíme, jak vypadá uvnitř. Chtěli bychom zjistit, jak v něm vedou zpětné hrany, ovšem jediné, co umíme, je spustit automat na nějaký řetězec a zjistit, v jakém stavu skončil.

Tvrdíme, že pro zjištění zpětné hrany ze stavu α stačí automatu předložit řetězec $\alpha[1:]$. Definice zpětné funkce je totiž nápadně podobná invariantu, který jsme o funkci *Hledej* dokázali. Obojí hovoří o nejdelším suffixu daného slova, který je prefixem jehly. Jediný rozdíl je v tom, že v případě zpětné funkce uvažujeme pouze vlastní suffixy, zatímco invariant připouští i nevlastní. To ovšem snadno vyřešíme „ukousnutím“ prvního znaku jména stavu.

[1] V jazyce C můžeme zneužít toho, že každý řetězec je ukončen znakem s nulovým kódem.

Pokud chceme objevit všechny zpětné hrany, stačí automat spouštět postupně na řetězce $\iota[1:1]$, $\iota[1:2]$, $\iota[1:3]$, atd. Jelikož funkce *Hledej* je lineární, stálo by nás to dohromady $\mathcal{O}(J^2)$. Pokud si ale všimneme, že každý ze zmíněných řetězců je prefixem toho následujícího, je jasné, že stačí spustit automat jen jednou na řetězec $\iota[1:]$ a jen zaznamenávat, kterými stavy jsme prošli.

To je zajímavé pozorování, řeknete si, ale jak nám pomůže ke konstrukci automatu, když samo už hotový automat potřebuje? Pomůže pěkný trik: pokud hledáme zpětnou hranu z i-tého stavu, spouštíme automat na slovo délky $i-1$, takže se můžeme dostat pouze do prvních $i-1$ stavů a vůbec nám nevadí, že v tom i-tém ještě není zpětná hrana hotova.⟨2⟩

Při konstrukci automatu tedy nejdříve sestrojíme dopředné hrany, načež rozpracovaný automat spustíme na řetězec $\iota[1:]$ a podle toho, jakými stavy bude procházet, doplníme zpětné hrany. Jak už víme, vyhledávání má lineární složitost, takže celá konstrukce potrvá $\mathcal{O}(J)$.

Hotový algoritmus pro konstrukci automatu můžeme zapsat následovně:

Algoritmus KmpKonstrukce
Vstup: Jehla ι délky J.

1. $Z[0] \leftarrow ?$, $Z[1] \leftarrow 0$.
2. $s \leftarrow 0$.
3. Pro $i = 2, \ldots, J$:
4. $\quad s \leftarrow \text{KmpKrok}(s, \iota[i-1])$.
5. $\quad Z[i] \leftarrow s$.

Výstup: Pole zpětných hran Z.

Výsledky můžeme shrnout do následující věty:

Věta: Algoritmus KMP najde všechny výskyty v čase $O(J+S)$.

Důkaz: Lineární čas s délkou jehly potřebujeme na postavení automatu, lineární čas s délkou sena pak potřebujeme na samotné vyhledání. □

1.4. Více řetězců najednou: algoritmus Aho-Corasicková

Nyní si zahrajeme tutéž hru v trochu složitějších kulisách. Tentokrát bude jehel vícero: $\iota_1, \ldots, \iota_N$, jejich délky označíme $J_i = |\iota_i|$. Opět dostaneme nějaké seno σ délky S a chceme nalézt všechny výskyty jehel v seně.

⟨2⟩ Konstruovat nějaký objekt pomocí téhož objektu je osvědčený postup, který si už vysloužil i svůj vlastní název. V angličtině se mu říká *bootstrapping* a z tohoto názvu vzniklo bootování počítačů, protože při něm operační systém zavádí do paměti sám sebe. Kde se toto slovo vzalo? Bootstrap znamená česky *štruple* – to je takové to očko na patě boty, které usnadňuje nazouvání. A v jednom z příběhů o baronu Prášilovi slyšíme barona vyprávět, jak se uvíznuv v bažině zachránil tím, že se vytáhl za štruple. Krásný popis bootování, není-liž pravda?

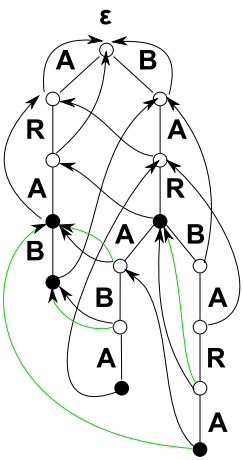
Předtím, než se pustíme do vlastního vyhledávacího algoritmu, měli bychom si opět ujasnit, co bude jeho výstupem. Dokud byla jehla jedna jediná, bylo to zřejmé – chtěli jsme nalézt množinu všech pozic v seně, na kterých začínaly výskyty jehly. Jak tomu bude zde? Chceme se dozvědět, která jehla se vyskytuje na které pozici. Jinými slovy vypsat všechny dvojice (k, i) takové, že $\sigma[k : k + J_i] = \iota_i$.

Těchto dvojic může být poměrně hodně. Pokud je totiž jedna jehla suffixem druhé, na jedné pozici v seně mohou končit výskyty obou. Celková velikost výstupu tak může být větší než lineární v délce vstupu (viz cvičení 1.5.2). Budeme proto hledat algoritmus, který bude lineární v délce vstupu plus délce výstupu, což je evidentně to nejlepší, čeho můžeme dosáhnout.

Algoritmus, který si nyní ukážeme, objevili v roce 1975 pan Aho a paní Corasicková. Je elegantním zobecněním Knuthova-Morrisova-Prattova algoritmu pro více řetězců.

Vyhledávací automat: Opět se budeme snažit sestrojit vyhledávací automat, jehož stavy budou odpovídat prefixům jehel a dopředné hrany budou popisovat rozšiřování prefixů o jeden znak. Hrany tedy budou tvořit strom orientovaný směrem od kořene (písmenkový strom neboli trii pro daný slovník).

Každý list stromu bude odpovídat některé z jehel, ale jak je vidět na obrázku, některé jehly se mohou vyskytovat i ve vnitřních vrcholech (pokud je jedna jehla prefixem jiné). Výskyty jehel ve stromu si tedy nějak označíme, příslušným stavům budeme říkat *koncové.*



Obr. 1.2: Vyhledávací automat pro slova `ara`, `bar`, `arab`, `baraba` a `barbara`.

Dále potřebujeme zpětné hrany (na obrázku černé šipky). Jejich definice bude úplně stejná jako u automatu KMP. Z každého stavu půjde zpětná hrana do jeho nejdelšího vlastního suffixu, který je také stavem. Čili se budeme snažit jméno

stavu zkracovat zleva tak dlouho, než dostaneme jméno dalšího stavu. Z kořene – prázdného stavu – pak evidentně žádná zpětná hrana nepovede.

Funkce pro hledání v seně bude vypadat stejně jako u KMP: začne v počátečním stavu (to je kořen stromu) a postupně bude rozšiřovat seno o další písmenka. Pokaždé zkusí jít dopřednou hranou a pokud to nepůjde, bude se vracet po zpětných hranách. Přitom se buďto dostane do vrcholu, kde vhodná dopředná hrana existuje, nebo se vrátí až do kořene stromu a tehdy nový znak zahodí.

Stejně jako u KMP nahlédneme, že procházení sena trvá $\mathcal{O}(S)$ a že platí analogický invariant, totiž že se v každém okamžiku nacházíme ve stavu, který odpovídá nejdelšímu suffixu zatím přečteného sena, který je prefixem některé jehly.

Hlášení výskytů: Jediné, co se bude od KMP lišit, je, kdy ohlásit výskyt. U KMP to bylo snadné: kdykoliv jsme dospěli do posledního stavu, znamenalo to nalezení jehly. Nabízí se hlásit výskyt, kdykoliv dojdeme do stavu označeného jako koncový. To ale nefunguje: pokud náš ukázkový automat přečte seno `bara`, skončí ve stavu `bara`, který není koncový, a přitom by zde měl ohlásit výskyt jehly `ara`. Stejně tak přečteme-li `barbara`, nevšimneme si, že na témže místě končí i `ara`.

Platí ale, že všechna slova, která bychom měli v daném stavu ohlásit, jsou suffixy jména tohoto stavu. Mohli bychom se tedy vydat po zpětných hranách až do kořene a kdykoliv projdeme přes koncový vrchol, ohlásit výskyt. To ovšem trvá příliš dlouho – jistě by se stávalo, že bychom podnikli dlouhou cestu do kořene a nenašli na ní nic.

Další, co se nabízí, je předpočítat si pro každý stav β množinu slov $M(\beta)$, jejichž výskyty máme v tomto stavu hlásit. To by fungovalo, ale existují množiny jehel, pro které bude celková velikost množin $M(\beta)$ superlineární (viz cvičení 1.5.4). Museli bychom se tedy vzdát lákavé možnosti stavby automatu v lineárním čase.

Jak to tedy vyřešíme? Zavedeme zkratky (na obrázku vyznačeny zeleně):

Definice: *Zkratková hrana* ze stavu α vede do nejbližšího koncového stavu $\zeta(\alpha)$ dosažitelného z α po zpětných hranách.

Jinými slovy, zkratka $\zeta(\alpha)$ nám řekne, jaký je nejdelší vlastní suffix slova α, který je jehlou. Pokud takový suffix neexistuje, žádná zkratková hrana ze stavu α nepovede. Pomocí zkratkových hran můžeme snadno vyjmenovat všechny výskyty. Budeme postupovat stejně, jako bychom procházeli po všech zpětných hranách, jen budeme dlouhé úseky zpětných hran, na nichž není nic k hlášení, přeskakovat v konstantním čase.

Reprezentace automatu: Vyhledávací automat se tedy sestává ze stromu dopředných hran, ze zpětných hran a ze zkratkových hran. Rozmysleme si, jak vše uložit do paměti. Pro každý stav si budeme pamatovat:

- s – pořadové číslo stavu (třeba podle toho, jak vrcholy vznikaly),
- *Zpět*(s) – kam z něj vede zpětná hrana (využíváme toho, ze může být nejvýše jedna, takže si zapamatujeme číslo stavu, do nějž vede),
- *Zkratka*(s) – kam z něj vede zkratková hrana (taktéž),

- *Slovo*(s) – zda tu končí nějaké slovo (a pokud ano, tak které),
- *Dopředu*(s, x) – kam vede dopředná hrana označená písmenem x (pro malé abecedy si to můžeme pamatovat v poli, pro velké viz cvičení 1.5.6).

Celý algoritmus pro zpracování sena automatem pak bude vypadat takto:

Procedura AcKrok *(Jeden krok automatu)*
Vstup: Jsme ve stavu s, přečetli jsme znak x.

1. Dokud *Dopředu*$(s, x) = \emptyset$ & $s \neq$ *kořen*: $s \leftarrow$ *Zpět*(s).
2. Pokud *Dopředu*$(s, x) \neq \emptyset$: $s \leftarrow$ *Dopředu*(s, x).

Výstup: Nový stav s.

Algoritmus AcHledej *(Spuštění automatu na daný řetězec)*
Vstup: Seno σ, zkonstruovaný automat.

1. $s \leftarrow$ *kořen*.
2. Pro znaky $x \in \sigma$ postupně provádíme:
3. $\quad s \leftarrow$ AcKrok(s, x).
4. $\quad j \leftarrow s$.
5. $\quad$ Dokud $j \neq \emptyset$:
6. $\quad\quad$ Je-li *Slovo*$(j) \neq \emptyset$:
7. $\quad\quad\quad$ Ohlásíme *Slovo*(j).
8. $\quad\quad j \leftarrow$ *Zkratka*(j).

Stejným argumentem jako u KMP zdůvodníme, že všechny kroky automatu dohromady trvají $\mathcal{O}(S)$. Mimo to ještě hlásíme výskyty, což trvá $\mathcal{O}(\textit{počet výskytů})$. Zbývá ukázat, jak automat sestrojit.

Konstrukce automatu: Opět se inspirujeme algoritmem KMP a nahlédneme, že zpětná hrana ze stavu β vede tam, kam by se automat dostal při hledání slova β bez prvního znaku. Chtěli bychom tedy začít sestrojením dopředných hran a pak spouštěním ještě nehotového automatu na jednotlivé jehly doplňovat zpětné hrany, doufajíce, že si vystačíme s už sestrojenou částí automatu.

Kdybychom však automat spouštěli na jednu jehlu po druhé (pokaždé bez prvního znaku), dostali bychom se do úzkých, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi stromu. Mohlo by se nám tedy stát, že při hledání nějakého slova potřebujeme zpětnou hranu, která vede do jiného slova, které jsme ještě nezkonstruovali. Proto tento postup selže.

Můžeme však využít toho, že každá zpětná hrana vede ve stromu alespoň o jednu hladinu výš, a strom konstruovat po hladinách. To si lze představit tak, že paralelně spustíme vyhledávání všech slov bez prvních písmenek a vždycky uděláme jeden krok každého z těchto hledání, což nám dá zpětné hrany v dalším patře stromu.

Kdykoliv vytvoříme zpětnou hranu, sestrojíme také zkratkovou hranu z téhož vrcholu. Pokud vede zpětná hrana ze stavu s do stavu z a *Slovo*(z) je definováno,

musí vést zkratka z s také do z. Pokud v z žádné slovo nekončí, musí zkratka z s vést do téhož vrcholu, kam vede zkratka ze z.

Algoritmus AcKonstrukce
Vstup: Slova $\iota_1, \ldots, \iota_n$.

1. Založíme strom, který obsahuje pouze kořen r.
2. Vložíme do stromu slova $\iota_1 \ldots \iota_n$, nastavíme *Slovo* ve všech stavech.
3. *Zpět*$(r) \leftarrow \emptyset$, *Zkratka*$(r) \leftarrow \emptyset$.
4. Založíme frontu F a vložíme do ní syny kořene.
5. Pro všechny syny s kořene: *Zpět*$(s) \leftarrow r$, *Zkratka*$(s) \leftarrow \emptyset$.
6. Dokud $F \neq \emptyset$:
7. Vybereme i z fronty F.
8. Pro všechny syny s vrcholu i:
9. $z \leftarrow$ AcKrok(*Zpět*(i), písmeno na hraně is).
10. *Zpět*$(s) \leftarrow z$.
11. Pokud *Slovo*$(z) \neq \emptyset$: *Zkratka*$(s) \leftarrow z$.
12. Jinak *Zkratka*$(s) \leftarrow$ *Zkratka*(z).
13. Vložíme s do fronty F.

Výstup: Strom, pole *Slovo*, *Zpět* a *Zkratka*.

Pro rozbor časové složitosti si uvědomíme, že konstrukce zpětných hran hledá všechny jehly, jen kroky jednotlivých hledání vhodným způsobem střídá (jakoby je prováděla paralelně). Její časovou složitost tedy můžeme shora omezit součtem složitostí hledání jehel, což, jak už víme, je lineární v délce jehel. Konstrukce dokonce může doběhnout i rychleji, jsou-li totiž dva z prováděných výpočtů v témže stavu, počítáme krok automatu pouze jednou. To je vidět třeba na společném začátku slov `araba` a `arbara`.

Chování celého algoritmu shrneme do následující věty:

Věta: Algoritmus Aho-Corasicková najde všechny výskyty v čase $\mathcal{O}\left(\sum_i J_i + S + V\right)$, kde $J_1, \ldots, J_n$ jsou délky jednotlivých jehel, S je délka sena a V počet ohlášených výskytů.

1.5. Rabinův-Karpův algoritmus

Nyní si ukážeme ještě jeden algoritmus na hledání jedné jehly, tentokrát založený na hešování. Ačkoliv jeho časová složitost v nejhorším případě bude srovnatelná s hledáním hrubou silou, v průměru bude lineární a v praxi často poběží rychleji než KMP.

Představme si, že máme seno délky S a jehlu délky J. Pořídíme si nějakou hešovací funkci H, které J-ticím znaků přiřazuje čísla z množiny $\{0, \ldots, N-1\}$ pro nějaké dost velké N. Budeme posouvat okénko délky J po seně, pro každou jeho polohu si spočteme heš znaků uvnitř okénka, porovnáme s hešem jehly a pokud se rovnají, porovnáme okénko s jehlou znak po znaku.

Pokud je hešovací funkce „kvalitní“, málokdy se stane, že by se heše rovnaly, takže místo času $\Theta(J)$ na porovnáváním řetězců si vystačíme s porovnáním hešů v konstantním čase. Jenže ouha, čas $\Theta(J)$ potřebujeme i na vypočtení heše pro každou polohu okénka. Jak z toho ven?

Pořídíme si hešovací funkci, kterou lze při posunutí okénka o jednu pozici doprava v konstantním čase přepočítat. Tyto požadavky splňuje třeba funkce

$$H(x_1, \ldots, x_J) = (x_1 P^{J-1} + x_2 P^{J-2} + \ldots + x_{J-1} P^1 + x_J P^0) \bmod N,$$

přičemž písmena považujeme za přirozená čísla a P je nějaká vhodná konstanta – potřebujeme, aby byla nesoudělná s N a aby P^J bylo řádově větší než N. Posuneme-li nyní okénko z $x_1, \ldots, x_J$ na $x_2, \ldots, x_{J+1}$, heš se změní takto:

$$\begin{aligned} H(x_2, \ldots, x_{J+1}) &= (x_2 P^{J-1} + x_3 P^{J-2} + \ldots + x_J P^1 + x_{J+1} P^0) \bmod N \\ &= (P \cdot H(x_1, \ldots, x_J) - x_1 P^J + x_{J+1}) \bmod N. \end{aligned}$$

Pokud si mocninu P^J předpočítáme, proběhne aktualizace heše v konstantním čase.

Celý algoritmus pak bude vypadat následovně:

Algoritmus RabinKarp
Vstup: Jehla ι délky J, seno σ délky S.

1. $j \leftarrow H(\iota)$. *(heš jehly)*
2. $h \leftarrow H(\sigma[:J])$. *(heš první pozice okénka)*
3. Zvolíme P a N a předpočítáme $P^J \bmod N$.
4. Pro i od 0 do $S - J$: *(možné pozice okénka)*
5. Je-li $h = j$:
6. Pokud $\sigma[i : i + J] = \iota$, ohlásíme výskyt na pozici i.
7. Pokud $i < S - J$: *(přepočítáme heš)*
8. $h \leftarrow (P \cdot h - \sigma[i] \cdot P^J + \sigma[i + J]) \bmod N$.

Analýza: V nejhorším případě pro každou pozici okénka porovnáváme řetězce a celý algoritmus spotřebuje čas $\Theta(JS)$. Abychom zjistili, jak tomu bude v průměrném případě, odhadněme pravděpodobnost porovnání.

Především pokud nastane výskyt, určitě porovnáváme. Nenastane-li, heš jehly se shoduje s hešem okénka s pravděpodobností $1/N$ (za předpokladu dokonale náhodného chování hešovací funkce, což té naší budeme věřit, ačkoliv jsme to pořádně nedokázali).

V průměru tedy spotřebujeme čas $\mathcal{O}(S + VJ + S/N \cdot J)$, kde V je počet nalezených výskytů. Pokud nám bude stačit najít první výskyt a zvolíme $N > SJ$, algoritmus poběží v průměrném čase $\mathcal{O}(S)$.

Cvičení:

1. Naivní algoritmus, který zkouší všechny možné začátky jehly v seně a vždy porovnává řetězce, má časovou složitost $\mathcal{O}(JS)$. Může být opravdu tak pomalý,

uvážíme-li, že porovnávání řetězců skončí, jakmile najde první neshodu? Sestrojte vstup, na kterém algoritmus poběží $\Theta(JS)$ kroků, přestože nic nenajde.

2. Nalezněte příklad jehel a sena, v němž se nachází superlineární počet výskytů vzhledem k celkové velikosti vstupu (součet velikostí jehel a délky sena).
3. Uvažujme zjednodušený algoritmus AC, který nepoužívá zkratkové hrany a vždy projde po zpětných hranách až do kořene. Ukažte příklad vstupu, na kterém je tento algoritmus asymptoticky pomalejší.
4. Označme $M(s)$ množinu řetězců, jejichž výskyty algoritmus AC ohlásí ve stavu s. Ukažte, že kdybychom si měli všechny množiny $M(s)$ pamatovat explicitně, budeme pro některé sady jehel potřebovat superlineární množství prostoru vzhledem k velikosti vstupu, takže automat nepůjde sestrojit v lineárním čase.
5. Rozmyslete si, že množiny $M(S)$ z předchozího příkladu by bylo možné reprezentovat jako srůstající spojové seznamy – tedy takové, kde si každý prvek pamatuje ukazatel na svého následníka, který ovšem může ležet v jiném seznamu. Ukažte, že námi zavedené zkratkové hrany lze interpretovat jako ukazatele v takových seznamech.
6. Rozmyslete si, jak algoritmy z této kapitoly upravit, aby si poradily s velkými abecedami.
7. Upravte algoritmus AC, aby pro každou pozici v seně vypsal nejdelší tam končící jehlu, a to efektivněji než vyjmenováním všech jehel. Jak se algoritmus změní, budeme-li chtít vypsat nejdelší jehlu, která na dané pozici začíná?
8. Mějme seno a jehly. Popište algoritmus, který v lineárním čase pro každou jehlu spočítá, kolikrát se v seně vyskytuje. Časová složitost by neměla záviset na počtu výskytů – ten, jak už víme, může být superlineární.
9. Cenzor dostane množinu zakázaných podřetězců a text. Vždy najde nejlevější výskyt zakázaného podřetězce v textu (s nejlevějším koncem; pokud jich je více, tak nejdelší takový), vystřihne ho a postup opakuje. Ukažte, jak text cenzurovat v lineárním čase.
10. Rotací řetězce α o K pozic nazýváme řetězec $\alpha[K:]\alpha[:K]$. Jak o dvou řetězcích zjistit, zda je jeden rotací druhého?
11. Jak v lineárním čase zrotovat řetězec, dostačuje-li paměť počítače jen na uložení jednoho řetězce a $\mathcal{O}(1)$ pomocných proměnných?
12.* Navrhněte algoritmus, který v lineárním čase nalezne tu z rotací zadaného řetězce, jež je lexikograficky minimální.
13. Navrhněte datovou strukturu pro básníky, která si bude pamatovat slovník a bude umět hledat rýmy. Tedy pro libovolné zadané slovo najde takové slovo ve slovníku, které má se zadaným nejdelší společný suffix.
14.* Upravte datovou strukturu z předchozího cvičení, aby v případě, že nejlepších rýmů je více, vypsala lexikograficky nejmenší z nich.
15. Jak reprezentovat slovník, abyste uměli rychle vyhledávat všechny přesmyčky zadaného slova?

16.* Je dán text a číslo K. Jak zjistit, který podřetězec délky K se v textu vyskytuje nejčastěji?

17.* Sestrojte co nejefektivnější algoritmus, který v daném řetězci najde nejdelší podřetězec, který se vyskytuje vícekrát.

18.* Ukažte, jak pro dané dva řetězce najít jejich nejdelší společný podřetězec.

19. Definujme Fibonacciho slova takto: $F_0 = \mathtt{a}$, $F_1 = \mathtt{b}$, $F_{n+2} = F_n F_{n+1}$. Jak v zadaném řetězci nad abecedou $\{\mathtt{a}, \mathtt{b}\}$ najít nejdelší Fibonacciho podslovo?

20.* Pokračujme v předchozím cvičení. Dostaneme řetězec nad nějakou obecnou abecedou, chceme nalézt jeho nejdelší podřetězec, který je isomorfní s nějakým Fibonacciho slovem (liší se pouze substitucí jiných znaků za `a` a `b`).

21. Je dáno slovo. Chceme nalézt jeho nejdelší prefix, který je současně suffixem.

22. Jak zjistit, zda je zadané slovo α periodické? Tím myslíme zda existuje slovo β a číslo $k > 1$ takové, že $\alpha = \beta^k$ (zřetězení k kopií řetězce β).