Korespondenční Seminář z Programování

38. ročník | **KSP** | **Květen 2026**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám páté číslo hlavní kategorie 38. ročníku KSP.

Až si budete venku užívat teplých dnů, nezapomeňte si s sebou přibalit i toto zadání, ve kterém naleznete poslední sadu úloh tohoto ročníku. Získáte dostatečný počet bodů a diplom úspěšného řešitele? Pojedete na Podzimní soustředění? Ať už to vyjde nebo ne, doufáme, že jste si z řešení tohoto ročníku odnesli nové znalosti a zkušenosti a že nepovažujete čas strávený nad úlohami za promarněný.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **14. června 2026 ve 32:00 (tedy další ráno v 8:00)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/h/odevzdavatko/*

Značky úloh:
- Lehčí úloha (či její část) vhodná pro začátečníky
- Praktická open-data úloha
- Úloha, u které doporučujeme začíst se do kuchařky
- Seriálová úloha
- Experimentální (neobvyklá) úloha

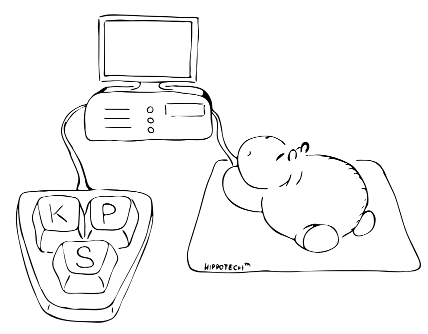
Pátá série třicátého osmého ročníku KSP

38-5-1 Rozmanitý mrak — 8 bodů

Jednodimenzionální Kevin jednoho jednodimenzionálního dne zkoumal jednodimenzionální oblohu. Zrovna na ní (směrem doleva) připlul krásný rozmanitý jednodimenzionální mrak. Kevin v něm s nadšením rozpoznával zvířátka. Třeba nalevo rozpoznal kočičku, když si ale odmyslel její hlavičku, viděl tam hada, když zase k tělíčku zvážil přidat i kousek napravo, co vypadal jako hroch, vypadalo to jako kočkopes.

Takových zvířátek viděl Kevin na mraku spoustu. Po chvíli ho ale napadlo, jestli jsou na mraku zvířátka úplně všude? Pomozte Kevinovi zjistit, zda každičký kousek mraku je součástí alespoň jednoho úseku, který vypadá jako zvířátko.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`



Dostanete kódovaní mraku M jako řetězec D znaků, a seznam kódování N zvířátek $Z_1, \ldots, Z_N$, každé z nich také jako řetězec znaků. Protože jednodimenzionální vědci zjistili, že existuje právě 26 typů kousku mraku, znaky jsou pro jednoduchost malá písmena anglické abecedy. Vaším úkolem je zjistit, zda je celý mrak pokrytý zvířátky, nebo se to Kevinovi jen zdá.

U řešení se vám jistě bude hodit přečíst si naši kuchařku o hledání v textu.[1] Doporučujeme si ověřit, že neděláte v řešení žádné zbytečné kroky kazící časovou složitost.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

38-5-2 Garážový výprodej 10 bodů

Kevinova kamaráda Prokopa zaujala výhodná nabídka v garážovém výprodeji: „Dva algoritmy za cenu jednoho!“

První nabízený algoritmus má za úkol najít komponenty silné souvislosti v orientovaném grafu.[2] Dostane na vstupu orientovaný graf G a funguje následovně:

1. Sestroj graf G^T s obrácenými hranami.
2. $Z \leftarrow$ prázdný zásobník
3. Pro všechny vrcholy $v \in G$ nastav komponenta$(v) \leftarrow$ nedefinováno.
4. Pro všechny vrcholy v:
5. Pokud DFS ještě nenavštívilo v:
6. Spusť DFS v G^T z vrcholu v. Při uzavření každého vrcholu jej vlož do Z.
7. Postupně odebíráme vrcholy ze zásobníku Z a pro každý takový vrchol v:
8. Pokud komponenta(v) = nedefinováno:
9. Spusť DFS v G z vrcholu v, přičemž DFS vstupuje jen do vrcholů u s nedefinovanou hodnotou komponenta(u) a tuto hodnotu přepisuje na v.
10. *Výstup:* komponenta(v) teď každému vrcholu v dává identifikátor jeho komponenty silné souvislosti.

Tento algoritmus skutečně funguje (a můžete si o něm přečíst více v Průvodci labyrintem algoritmů v oddílu 5.9). Má to ale jeden háček: algoritmus jako podproceduru využívá DFS, které prodejce nemá v nabídce. Druhý algoritmus v ceně je proto BFS se zásobníkem. Prodejce tvrdí, „Mám jenom BFS, ale to přece nevadí. Vyměním v něm frontu za zásobník a vznikne DFS, ne?“ Takhle vypadá „DFS“, které se použije v nabízeném algoritmu:

1. *Vstup:* Graf G a jeho vrchol u.
2. $S \leftarrow$ zásobník s jediným prvkem u
3. Označ vrchol u jako navštívený.
4. Dokud je zásobník S neprázdný:
5. Odeber ze S vrchol a označ ho v.
6. Pro všechny vrcholy w, do kterých z v vede hrana:
7. Pokud w ještě není označen jako navštívený:
8. Označ w jako navštívený.
9. Přidej w do S.
10. *Uzavření vrcholu v.*

Prokop je nadšený, že získá tyhle dva algoritmy v ceně jednoho, ale Kevinovi se to nezdá. Je přesvědčený, že s tímhle falešným „DFS“ algoritmus nemusí najít komponenty silné souvislosti. Chce Prokopovi ukázat, jak zrádný ten algoritmus je, a my mu s tím pomůžeme.

Od toho, v jakém pořadí algoritmus prochází vrcholy (ve čtvrtém řádku prvního algoritmu a v šestém řádku falešného DFS), se odvíjí, zda správně rozdělí graf na komponenty silné souvislosti. Na vstupu dostanete graf a vaším úkolem je najít dvě specifická pořadí vrcholů. První pořadí má tento zrádný algoritmus dovést ke správnému rozdělení grafu na komponenty, druhé pořadí ho má donutit selhat. Slibujeme, že obě pořadí půjde pro naše vstupy najít. Tím určitě Prokopa přesvědčíme, aby si algoritmy nekoupil. Není přece nic zrádnějšího, než chybný algoritmus, který ale za určitých okolností uspěje.

[2] http://ksp.mff.cuni.cz/viz/kucharky/grafy

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu:

Na prvním řádku dostanete dvě kladná čísla N a M. Zadaný graf má N vrcholů pojmenovaných 1 až N. Na dalších M řádcích je vyjmenovaných jeho M orientovaných hran.

Formát výstupu:

Na výstup vypište dva řádky. První by měl obsahovat pořadí vrcholů, tedy čísla od 1 do N, oddělená mezerou, na kterém popsaný algoritmus uspěje. Na druhém řádku bychom chtěli mít pořadí, na kterém neuspěje, tedy vrcholy rozdělí do skupin, které nejsou silně souvislé komponenty. Algoritmus přitom s pořadím pracuje ve čtvrtém řádku prvního algoritmu a v šestém řádku falešného „DFS“, kde slouží na seřazení následníků vrcholu v.

Ukázkový vstup:

```
3 3
1 2
2 3
3 2
```

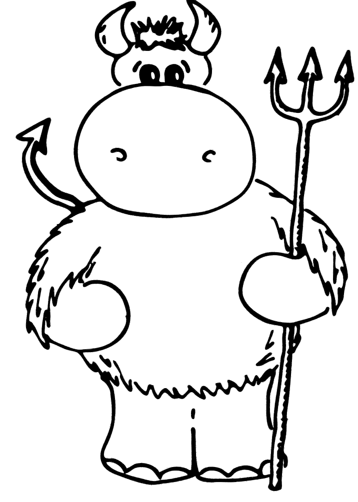
Ukázkový výstup:

```
1 2 3
2 1 3
```

Zadaný graf má dvě komponenty silné souvislosti – $\{1\}$ a $\{2, 3\}$. Když se algoritmus bude řídit prvním pořadím, nejprve „DFS“ v G^T spustí pro vrchol 1. Z toho v G^T nevede žádná hrana, takže bude rovnou uzavřený a vložený do Z. Potom se „DFS“ v G^T spustí pro vrchol 2, čímž se do Z vloží vrcholy 2 a 3.

Algoritmus pak postupně odebírá vrcholy ze Z a pouští na nich „DFS“ v G. První na řadu přijde vrchol 3, ze kterého je dostupný jen vrchol 2, tedy komponenta(3) = komponenta$(2) = 3$. Potom se ještě spustí „DFS“ pro vrchol 1, čímž se označí komponenta$(1) = 1$. „DFS“ už se nezavolá pro vrchol 2, protože ten už má označenou komponentu. Vrcholy jsou tak korektně rozdělené na komponentu „1“, která obsahuje vrchol 1, a komponentu „3“, která sestává z vrcholů 2 a 3.

Pro druhé pořadí algoritmus selže, protože vrcholy se do Z vloží v pořadí $2, 3, 1$, takže vrchol 1 bude ze Z odebraný jako první, a z něj už „DFS“ přejde i do vrcholů 2 a 3, čímž budou všechny vrcholy nesprávně přiřazeny stejné komponentě „1“.



38-5-3 Nespolehlivé autobusy 13 bodů

Kevin se vrací domů z noční procházky po svém oblíbeném evropském hlavním městě. Protože už ho bolí nohy, tak k cestě domů plánuje využít místní husté sítě autobusů. Pro zjednodušení orientace v dopravě nedávno dopravní podnik zavedl opatření, že každý spoj bude mít jen dvě zastávky, jednu nástupní a jednu výstupní. Kevin ale ze svých zkušeností ví, že takto pozdě večer existuje pravděpodobnost, že řidič autobusu během přestávky usne a autobus pak nepřijede. Poradíte mu, jak maximalizovat pravděpodobnost, že se Kevin do půlnoci dostane domů?

Kevin má k dispozici tabulku jízdních řádů, kde je pro každý spoj uvedeno místo a čas odjezdu a místo a čas příjezdu. Přirozeně je čas příjezdu vždy ostře větší než čas odjezdu a ze zastávky je v jeden čas naplánován vždy nejvýše jeden odjezd. Ke každému autobusu navíc na základě svých zkušeností Kevin dopsal, s jakou pravděpodobností pojede. To, jestli nějaký autobus pojede, nemá žádný vliv na další autobusy – formálně jsou tedy příjezdy autobusu nezávislé jevy. Autobusy s bdělými řidiči jezdí na vteřinu přesně, takže Kevin zjistí, zda autobus jede nebo ne, přesně v okamžiku plánovaného odjezdu.

Vaším úkolem je navrhnout interaktivní algoritmus, který Kevinovi bude radit, jak se má chovat. Nejprve do něj Kevin zadá jízdní řády s pravděpodobnostmi, název zastávky, na které začíná, a název zastávky na které kde chce skončit. Váš program by pak měl Kevinovi říci, kterým spojem má jet jako první. Kevin pak počká do jeho plánovaného příjedu a do programu zadá, že se autobusem svezl, nebo že autobus nepřijel. Program pak vypíše další spoj, kterým má Kevin jet, a tak dále. Jakmile Kevin dorazí domů, nebo začne být nemožné, aby dorazil do půlnoci, tak to má program zahlásit a skončit.

Popis jízdního řádu by mohl vypadat například takto:

```
Spoj 001: z A v 19:00 do B v 19:10 (60 %)
Spoj 002: z B v 19:15 do Z v 20:00 (10 %)
Spoj 003: z B v 19:20 do A v 19:30 (100 %)
Spoj 004: z A v 19:45 do Z v 20:10 (80 %)
Chci z A do Z, je 18:55.
```

V ukázkovém jízdním řádu existuje spoj 004 z `A` do `Z`, co jede s vysokou pravděpodobností 80 %. Vyplatí se ale nejprve zkusit jet spojem 001 a 002, což má sice pravděpodobnost úspěchu jen 6 %, ale pokud spoj 002 nepřijede, tak se můžeme jistě autobusem 003 vrátit zpátky do `A` a spoj 004 použít stejně. Tato strategie bude mít pravděpodobnost úspěchu 81,2 %.

Interakce Kevina s programem by pak tedy mohla vypadat například takto:

```
Jeď spojem 001.
> Jedu.
Jeď spojem 002.
> Nepřijel.
Jeď spojem 003.
> Jedu.
Jeď spojem 004.
> Nepřijel.
Máš smůlu, musíš pěšky.
```

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

Ve svém popisu algoritmu můžete předpokládat, že s reálnými čísly, například s pravděpodobnostmi, můžete provádět aritmetické operace s neomezenou přesností v konstantním čase. Můžete také předpokládat, že zastávky a časy dostanete zadané jako přirozená čísla. Časovou složitost určujte vzhledem k počtu spojů a počítejte ji za celý běh programu dohromady.

38-5-4 Linuxáci a Windowsáci 14 bodů

V hroší říši v nějakém paralelním vesmíru se právě koná soustředění hrochů v hrošení. Tato událost je veliká – každý rok se tam schází tisíce hrochů a poměřují se v této královské disciplíně. Nejlepších výsledků lze dosáhnout tak, že si hroši napíšou program, který jim s hrošením co nejvíce pomůže – proto každý hroch potřebuje svůj notebook. Přestože tento požadavek byl v pozvánce silně zdůrazněn, našli se tací, kteří si žádný notebook nevzali. Orgové však s tímto již dopředu počítali a mají velké množství notebooků v záloze.

Máme tu však jistý zádrhel. Víme, že je spousta účastníků, kteří mají velmi rádi svá okna nebo velmi rádi své tučňáky, a to tak moc, že pokud jejich kamarád nemá stejný operační systém jako oni, začnou na něj nadávat, že používá systém, který špehuje a využívá uživatele, nebo že má systém, na kterém nespustí software od Adobe. Orgové tedy chtějí rozdat notebooky tak, aby celkem bylo mezi kamarády co nejméně konfliktů. Každému hrochu, který si nevzal notebook, je jedno, co dostane za systém, hlavně že má notebook.

Zadání tedy zní: máme neorientovaný graf tvořený N hrochy a M kamarádskými vazbami mezi nimi. Každý hroch buď má notebook s Linuxem, s Windows, nebo si notebook zapomněl. Vaším úkolem je přiřadit všem hrochům bez notebooku Linux nebo Windows tak, abychom minimalizovali celkový počet hran mezi hrochy s Windows a Linuxem.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu:

Na první řádce dostanete číslo N. Na dalším řádku pak bude N znaků, kde i-tý znak buď značí, že i-tý hroch má buď Linux (znak `L`), Windows (znak `W`), nebo je bez notebooku (znak `B`). Na dalším řádku pak dostanete číslo M a pod ním M řádků. Na každém z těchto řádků pak máme čísla u_i a v_i, která značí, že hroši u_i a v_i jsou přátelé (hrochy počítáme od nuly).

Formát výstupu:

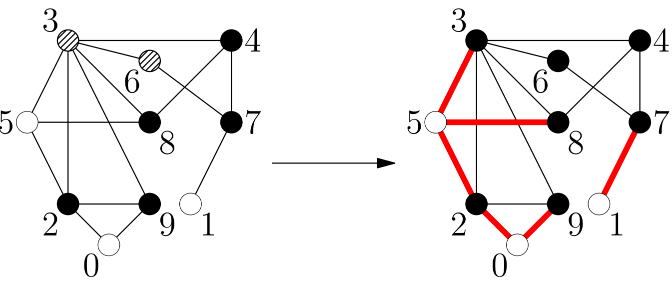
Na výstup vraťte N znaků, kde i-tý znak značí, zda i-tý hroch má notebook s Linuxem či Windows.

Ukázkový vstup:

```
10
WWLBLWBLLL
15
3 8
3 4
5 8
0 9
2 3
2 9
0 2
1 7
3 9
6 7
4 8
3 6
2 5
4 7
3 5
```

Ukázkový výstup:

```
WWLLLWLLLL
```



V ukázkovém vstupu jsou hroši 3 a 6 bez notebooku. Pokud oběma dáme notebook s Linuxem, budeme celkem mít 6 konfliktů, což je optimální.

38-5-S Hešování řetězců 15 bodů

V minulém dílu seriálu jsme studovali univerzální systémy hešovacích funkcí pro čísla. Teď si ukážeme, jak hešovat posloupnosti čísel a řetězce znaků a co se s tím dá vyvádět zajímavého.

Polynomiální hešování

Nejdřív budeme hešovat uspořádané k-tice $(x_1, \ldots, x_k)$. Jejich složky budou buďto čísla, nebo znaky, které si také převedeme na čísla. Každopádně je můžeme považovat za prvky nějakého konečného tělesa $\mathbb{Z}_p$ pro dost velké prvočíslo p.

Pořídíme si systém hešovacích funkcí $\{h_a \mid a \in \mathbb{Z}_p\}$, které vypadají následovně. Vše počítáme v $\mathbb{Z}_p$, tedy modulo p. Parametr a budeme jako obvykle volit náhodně ze $\mathbb{Z}_p$.

$$h_a(x_1, \ldots, x_k) = \sum_{i=1}^{k} x_i \cdot a^{i-1}.$$

Sumu můžeme rozepsat takto:

$$x_1 a^0 + x_2 a^1 + \ldots + x_{k-1} a^{k-2} + x_k a^{k-1}.$$

To je polynom v *proměnné* a, jehož *koeficienty* jsou složky k-tice. Proto se celému systému říká *polynomiální hešování.*

Uvažujme, jestli je polynomiální hešování c-univerzální pro nějaké c. Bude se nám hodit standardní věta z algebry (kterou nebudeme dokazovat):

Věta: Nenulový polynom stupně d nad libovolným tělesem má nejvýše d kořenů.

Co tato věta znamená? *Stupeň* polynomu je nejvyšší mocnina proměnné, která se v něm vyskytuje. *Nenulový* znamená, že aspoň jeden koeficient není nula. A *kořen* polynomu $p(t)$ je takové t, pro něž $p(t) = 0$.

Definice c-univerzality po nás chce, abychom pro každé dvě různé k-tice $(x_1, \ldots, x_k)$ a $(y_1, \ldots, y_k)$ omezili pravděpodobnost toho, že pro náhodné a nastane kolize

$$h_a(x_1, \ldots, x_k) = h_a(y_1, \ldots, y_k).$$

Do toho dosadíme definici h_a a získáme:

$$x_1 a^0 + x_2 a^1 + \ldots + x_k a^{k-1} = y_1 a^0 + y_2 a^1 + \ldots + y_k a^{k-1},$$

což po převedení pravé strany nalevo dává:

$$(x_1 - y_1) \cdot a^0 + (x_2 - y_2) \cdot a^1 + \ldots + (x_k - y_k) \cdot a^{k-1} = 0.$$

Levá strana je polynom v proměnné a, který má stupeň maximálně $k-1$ (může být nižší, pokud $x_k = y_k$). Jelikož aspoň pro jedno i je x_i různé od y_i, polynom není nulový. Kořeny polynomu jsou ta a, pro která rovnost platí, a podle věty jich je nejvýše $k-1$. Parametr a ovšem volíme náhodně z p-prvkového tělesa. Proto pravděpodobnost, že nastane kolize, je nejvýše $(k-1)/p$.

Zjistili jsme tedy, že *polynomiální hešování je c-univerzální pro $c = k-1$.* Většinou ho budeme prohlašovat za k-univerzální, což je slabší vlastnost, ale lépe se s ní pracuje.

To nám dává použitelné hešovací funkce (alespoň pokud k není moc velké), ale pozor, že časová složitost výpočtu funkce už není konstanta, nýbrž $\mathcal{O}(k)$.

Ještě dodejme, že praktické implementace často místo modulo prvočíslo počítají modulo nějaká mocnina dvojky (třeba 2^b, pokud pracujeme s b-bitovými čísly). Tehdy se nejedná o těleso, takže nemůžeme využívat věty o polynomech v tělese, a díky tomu zaručit univerzalitu. Nicméně experimenty ukazují, že se takové funkce stále chovají dobře a jsou rychlejší.

Hešování řetězců

Co kdybychom hešovali posloupnosti nebo řetězce, které nemají pevnou délku k? Pokud víme, že všechny délky budou shora omezené nějakým maximem M, můžeme všechny řetězce doplnit „mezerami" na délku M a hešovat polynomiálně. Mezerou myslíme hodnotu, která se v řetězcích nevyskytuje.

Tak získáme M-univerzální hešování, ale bohužel je dost pomalé: i kratičké řetězce hešujeme v čase $\mathcal{O}(M)$. Pomoc je snadná: pro mezeru si vybereme číslo 0, takže členy polynomů s mezerami můžeme vynechat. Pak už hešování k-prvkového řetězce potrvá $\mathcal{O}(k)$.

Úkol 1 [3b]: Představte si, že znáte $h_a(\alpha)$ pro nějaký řetězec α. Ukažte, jak v konstantním čase spočítat $h_a(\alpha')$ pro řetězce α', které vzniknou z α přidáním nebo ubráním jednoho znaku na začátku nebo na konci. Nezapomeňte, že h_a se počítá modulo p.

Rabinův-Karpův algoritmus

Na polynomiálních hešovacích funkcích je založený pěkný algoritmus na vyhledávání v textu. Mějme řetězec σ (*seno*), ve kterém chceme najít všechny výskyty řetězce ι (*jehla*) jako souvislý podřetězec. Označme J délku jehly a S délku sena.

Vezmeme hešovací funkci h_a v tělese $\mathbb{Z}_p$ pro dost velké p a náhodné $a \in \mathbb{Z}_p$. Spočítáme si heš jehly, pojedeme po seně „okénkem" délky J a budeme přepočítávat heš okénka. Z předchozího úkolu plyne, že se při posunutí okénka

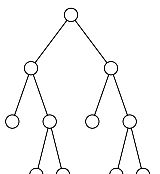
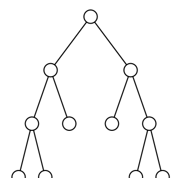
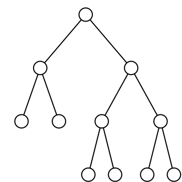
o znak doprava dá jeho heš přepočítat v konstantním čase. A kdykoliv se heš okénka rovná heši jehly, znamená to buďto nalezený výskyt, nebo kolizi hešovací funkce (těch snad nebude mnoho). Abychom tyto dva případy rozlišili, porovnáme obsah okénka s jehlou znak po znaku.

Úkol 2 [2b]: Spočítejte průměrnou časovou složitost algoritmu v závislosti na délce jehly J, délce sena S, počtu výskytů V a prvočíslu p. Jak byste nastavili p, aby čas vyplýtvaný na kolizích byl zanedbatelný?

Izomorfismus binárních stromů

Uvažujme *čistě binární stromy.* To jsou zakořeněné stromy, v nichž každý vnitřní vrchol (takový, co není list) má právě dvě děti. *Otočení* vnitřního vrcholu nazveme operaci, která prohodí jeho děti včetně jejich podstromů (podstromem myslíme dítě se všemi jeho potomky). Dva stromy jsou *izomorfní,* pokud jeden můžeme z druhého vyrobit nějakou posloupností otočení vrcholů.

Například na následujícím obrázku je první strom izomorfní s druhým, ale ne s třetím:

Hledejme algoritmus, který pro dva stromy rozhodne, zda jsou izomorfní. Postupně všem vrcholům v obou stromů přiřadíme *kódy.* To budou nějaká čísla $k(v)$. Bude platit, že dva vrcholy mají stejný kód právě tehdy, když jejich podstromy jsou izomorfní. Izomorfismus celých stromů tedy půjde rozhodnout porovnáním kódů kořenů.

Nejprve pro všechny vrcholy spočítáme výšky jejich podstromů. To zvládneme prohledáním obou stromů do hloubky. Kdykoliv budeme opouštět vrchol, přidělíme mu výšku. Je-li to list, výška bude nulová. Vnitřnímu vrcholu přidělíme o 1 víc, než je maximum z výšek dětí. Celé prohledání zvládneme v čase $\mathcal{O}(n)$, kde n je celkový počet vrcholů v obou stromech.

Vrcholy podle výšek rozdělíme na *hladiny* a v tomto pořadí jim budeme přidělovat kódy. Listům (vrcholům výšky 0) přidělíme všem stejný kód. Když už máme zakódované vrcholy výšek 0 až $h-1$, můžeme snadno zakódovat vrcholy výšky h.

Těm přidělíme nové kódy (vrcholy různých výšek nemohou nikdy mít izomorfní podstromy). Označme levé a pravé dítě vrcholu x jako $\ell(x)$ a $p(x)$ a příslušné podstromy $L(x)$ a $P(x)$ Pak dva vrcholy u a v mají dostat stejný kód, pokud buď $L(x)$ je izomorfní s $L(y)$ a $P(x)$ s $P(y)$, nebo „křížem“ $L(x)$ s $P(y)$ a $P(x)$ s $L(y)$. Jelikož děti vrcholu x mají menší výšku než x, už známe jejich kódy, tedy stačí porovnat neuspořádané dvojice $\{k(\ell(x)), k(p(x))\}$ a $\{k(\ell(y)), k(p(y))\}$.

Chceme tedy vzít všechny vrcholy výšky h, každému z nich přiřadit neuspořádanou dvojici kódů dětí, a každé skupině vrcholů se stejnou dvojicí přidělit jeden nový kód. To převedeme na porovnání uspořádaných dvojic tak, že vždy zapíšeme nejdřív menší z kódů dětí a pak ten větší. Jelikož nechceme porovnávat každou dvojici s každou, nabízí se následující možnosti:

První možnost: Kódy setřídíme lexikograficky. Tím se nám dostanou stejné dvojice k sobě, takže skupiny stejných dvojic nalezneme snadno. To potrvá $\mathcal{O}(n_h \log n_h)$, kde $n_h \leq n$ je počet vrcholů výšky h. V součtu přes všechny hladiny pak $\mathcal{O}(n \log n)$.

Druhá možnost: Pořídíme si datovou strukturu, jež si bude pamatovat dvojice, kterým už jsme přiřadili kód. A pokaždé, když potkáme nový vrchol, vyhledáme jeho dvojici v datové struktuře a buď použijeme známý kód, nebo přidělíme nový a zaznamenáme ho do datové struktury. Pokud datová struktura bude vyhledávací strom, budou operace s ní trvat $\mathcal{O}(\log n)$, a proto kódování všech vrcholů dohromady potrvá $\mathcal{O}(n \log n)$.

Třetí možnost: Jako datovou strukturu použijeme hešovací tabulku. Tehdy bude mít jedna operace průměrnou složitost $\mathcal{O}(1)$ a celý algoritmus průměrně $\mathcal{O}(n)$.

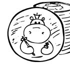
Čtvrtá možnost by byla použít šikovně přihrádkové třídění. Tím by se dala dosáhnout časová složitost celého algoritmu $\mathcal{O}(n)$ v nejhorším případě.

Raději prozkoumáme pátou možnost: obzvláště jednoduchý *pravděpodobnostní algoritmus.* Místo datové struktury budeme kódy přiřazovat hešovací funkcí $h(k_1, k_2)$, které dáme kódy dětí k_1 a k_2 a její výsledek prohlásíme za kód rodiče. Tehdy bude pořád platit, že izomorfní stromy dostanou stejný kód, ale kolize mohou způsobit, že stejný kód občas dostanou i neizomorfní stromy.

Úkol 3 [3b]: Vyberte pro pravděpodobnostní algoritmus vhodnou hešovací funkci, ideálně z nějakého univerzálního systému nad tělesem $\mathbb{Z}_p$ pro vhodné p. Jakou časovou složitost bude mít algoritmus? Jaká bude pravděpodobnost, že selže, tedy že neizomorfním stromům přiřadí stejný kód? Jak byste zvolili p tak, aby pravděpodobnost selhání klesla pod zadané ε?

Úkol 4 [2b]: Předpokládejme, že náš pravděpodobnostní algoritmus doběhl a oběma zadaným stromům vyšly stejné kódy kořenů. Vymyslete, jak v čase $\mathcal{O}(n)$ pomocí spočítaných hešů buď zjistit, že skutečně jsou izomorfní, nebo že došlo ke kolizi (aniž bychom používali celý komplikovaný deterministický algoritmus na izomorfismus). Co kdybychom v případě, že izomorfní nebudou, algoritmus znovu spustili? To už by nám dalo spolehlivý algoritmus. Jaká by byla jeho průměrná časová složitost?

Úkol 5 [2b]: Navrhněte, jak přidělování kódů hešováním rozšířit na stromy s libovolným počtem dětí. Tehdy operace otočení vrcholu bude moci libovolně přeházet pořadí dětí. Snažte se dosáhnout stejné průměrné složitosti a podobně malé pravděpodobnosti chyby. Tentokrát můžete předpokládat, že máte k dispozici dokonale náhodné hešovací funkce do množiny $\{0, \ldots, m-1\}$ pro jakékoliv m (nejvýše polynomiálně velké).



Mezigalaktická kandidátka

Na závěr seriálu si vyzkoušíme sílu pravděpodobnostních algoritmů na úloze 36-1-X1 (Mezigalaktická kandidátka). Přečtěte si zadání a prvních část vzorového řešení, zastavte se před nadpisem „Rozděl a panuj“.

Úkol 6 [3b]: Ukažte, jak k-složkové vektory kódovat tak, aby stejné kódy skoro vždy odpovídaly stejným vektorům a jedním kódem jste trávili čas jen $\mathcal{O}(1)$. Na tom založte pravděpodobnostní algoritmus, který poběží v průměrném čase $\mathcal{O}(n)$ a vydá vždy správný výsledek.

Závěrem

Tím končí náš pětidílný výlet do říše pravděpodobnostních algoritmů. Viděli jsme, že tyto algoritmy dokáží být v mnoha případech mnohem jednodušší než jejich determinističtí příbuzní. Ale jednoduchost bývá vykoupena tím, že algoritmy jsou rychlé pouze v průměru, nebo dokonce nemusí být úplně spolehlivé.

Dalo by se pokračovat mnoha dalšími tématy, třeba *treapy,* což jsou pravděpodobnostní vyhledávací stromy (viz kuchařka[3] nebo kapitola 11.6 v Průvodci). Ale to už je jiný příběh a ten si budeme vyprávět zase někdy jindy.

Martin „Medvěd“ Mareš

[3] http://ksp.mff.cuni.cz/viz/kucharky/treapy

Recepty z programátorské kuchařky: Hledání v textu

Řetězec je v podstatě jakákoliv posloupnost symbolů zapsaná za sebou a s nimi budeme v této kuchařce pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu,“ ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a algoritmů s nimi pracujících) najdeme v biologii. Například DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu do práce s řetězci popíšeme dva *stavební kameny* textových algoritmů, což bude datová struktura pro slovníky – *trie* a vyhledání v textu s předzpracováním hledaného slova a jeho rozšíření pro více slov. S jejich znalostí pak bude mnohem snazší vymýšlet řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen $\{\mathtt{0},\mathtt{1}\}$ pro čísla v binárním zápisu, klasické $\{\mathtt{A}\text{-}\mathtt{Z},\mathtt{a}\text{-}\mathtt{z}\}$ pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kuchařkách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme umět rozhodnout, který je menší a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné, a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního $\mathtt{0} < \mathtt{1}$ se často používá „telefonní“ $\mathtt{A} = \mathtt{a} < \mathtt{B} = \mathtt{b} < \ldots < \mathtt{Z} = \mathtt{z}$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon < \mathtt{A} < \mathtt{AUTO} < \mathtt{AUTOBUS} < \mathtt{AUTOGRAM} < \mathtt{AUTOR} < \mathtt{BAMBITKA} < \mathtt{BARNABAS} < \mathtt{Z}$.

Adresář pomocí trie

Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

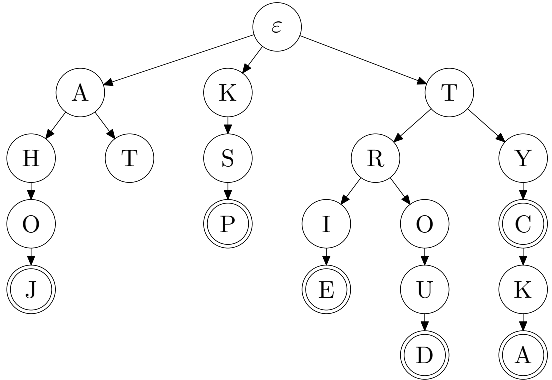
Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[4] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

[4] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom.“

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku, a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova, nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro $\{\texttt{A-Z}, \texttt{a-z}\}$ je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba $\{\texttt{0}, \texttt{1}\}$. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2 |\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log |\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log |\Sigma|)$.

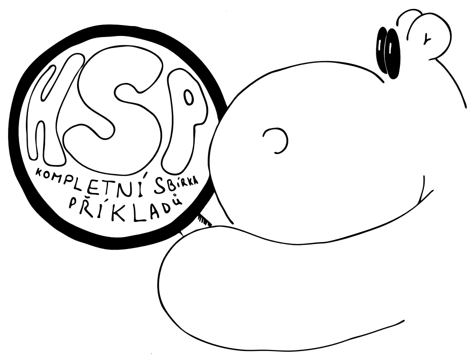
A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti; jednak pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů*.[5]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

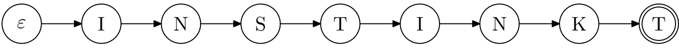


[5] http://mj.ucw.cz/vyuka/ga/

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu NANANA se slovo NANA vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo INSTINKT:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo INSTINSTINKT?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem INSTINSTINKT se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme INSTINS. Mohli jsme se vrátit jen na druhý znak, tedy do prvního N, a pak kontrolovat, jaký znak pokračuje dál. Když následuje S jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba INSTINB, vrátili bychom se po načtení B na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

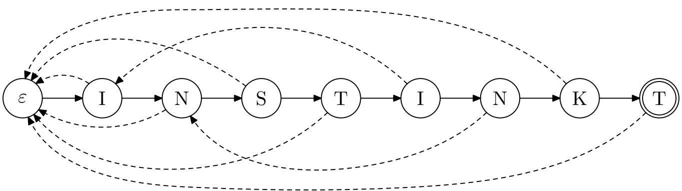
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé N ve slově INSTINKT. Pracujeme teď s prefixem INSTIN. Selsky řečeno, chceme najít „konec slova INSTIN takový, že je stejný, jako začátek slova INSTIN“.

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo ABABABC a my určovali zpětné políčko pro ABABAB? Kdybychom ukázali na první písmenko B, nebylo by to správně, protože pak bychom pro text ABABABABC nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na ABAB!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo INSTIN je samo sobě prefixem a suffixem, ale zpětná funkce pro N by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo INSTINKT vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:

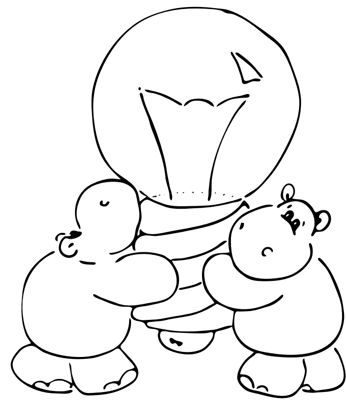


Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, tj. $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.



Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již

máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
jehla = "INSTINKT"
seno = "INSTINSTINKTINSTINKT"
J = len(jehla)
S = len(seno)
F = [None] * J  # Zpětná funkce

def krok(i, znak):
    if i < J and jehla[i] == znak:
        return(i + 1)
    elif i > 0:
        return krok(F[i - 1], znak)
    else:
        return 0

# Konstrukce zpětné funkce
F[0] = 0
for i in range(1, J):
    F[i] = krok(F[i - 1], jehla[i])

# Procházení textu
stav = 0
for i in range(S):
    stav = krok(stav, seno[i])
    if stav == J:
        print(i - J + 1, "až", i)
```

Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* (neboli „okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

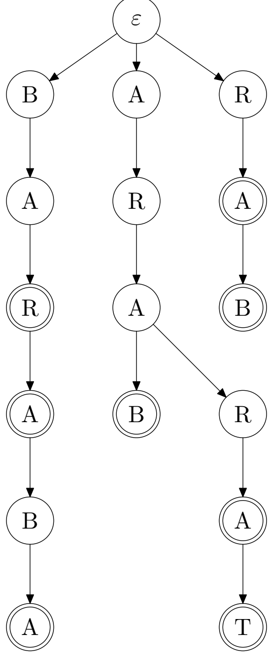
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. V trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

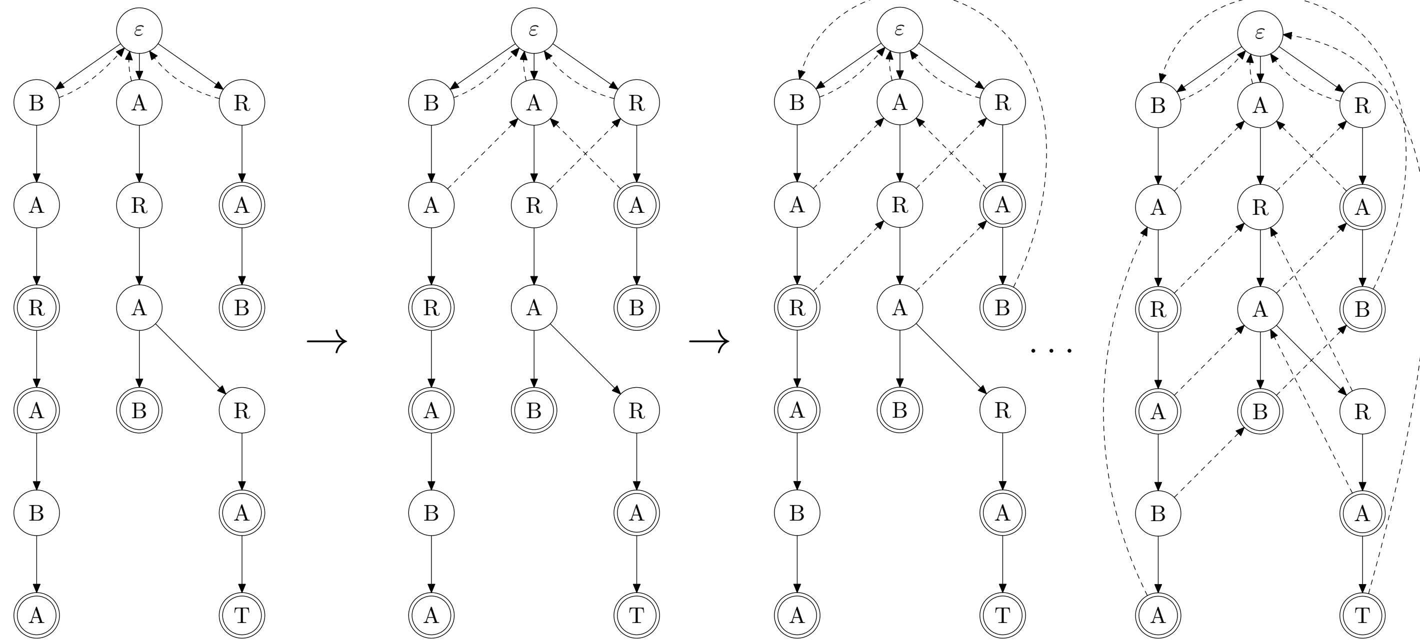
Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé ze slov. Ouha, to ale také nefunguje. Když začneme slovem `BARABA`, a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tak povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

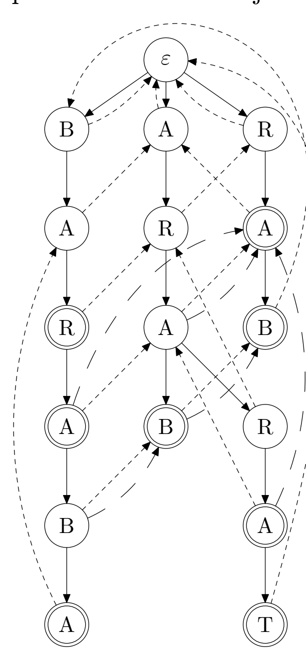


Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

1. $c =$ poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$.

Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.



Zkusme tedy automatem projít text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl algoritmus na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme. Na rozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít všechny suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

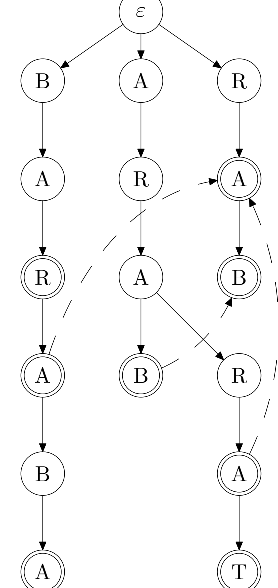
Představme si například slovník obsahující `A` a `AAAA...A` (délky $J - 1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J - 1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(S + O)$, resp. $\mathcal{O}(S \cdot \log |\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O + S + J \cdot |\Sigma|)$, resp. $\mathcal{O}(O + (S + J) \cdot \log |\Sigma|)$.



Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

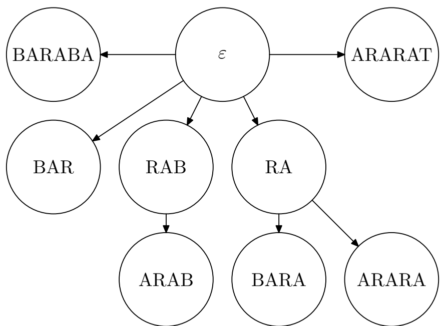
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale

maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si implementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*