Řetězce a vyhledávání v textu

Řetězec je v podstatě jakákoli posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole 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 jejich algoritmů) najdeme v biologii. DNA není o mnoho více, než posloupnost čtyř znaků/nukleových bazí – a 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 popíšeme dva *stavební kameny* textových algoritmů, což bude jedna datová struktura pro adresáře (trie) a jedno vyhledání v textu s předzpracováním hledaného slova. S jejich znalostí se pak mnohem snáze vymýšlí 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 `01` pro čísla v binárním zápisu, klasické `A-Za-z` pro malou anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že 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 samotná 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 kapitolách.

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-li ji značit dále n, tak časová složitost bude $\mathcal{O}(n)$.

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 `ret`, ε i `kabaret` jsou podřetězce slova (řetězce) `kabaret`.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova usekneme nějaký souvislý úsek na konci, vznikne podřetězec, které říkáme *prefix* (česky předpona), a pokud usekneme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `ret` je suffix slova `kabaret`, `kaba` je zase jeho prefixem.

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

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce A a B, tak 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 (kromě binárního `0` < `1` se často používá „telefonní“ `A` = `a` < `B` = `b` < ... < `Z` = `z`, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce jej rozšíříme následovně: nejkratší je prázdný řetězec a ostatní řetězce třídíme podle znaků od začátku do konce. Zvláštnost je v tom, že řetězec je větší než jeho každá vlastní předpona (neboli *prefix*). Řetězec `a` tedy bude menší než `auto`, které samo bude menší než `autobus`.

Adresář pomocí trie

Typický problém v oblasti textu je, že máme seznam nějakých řetězců (často třeba jmenný adresář), můžeme si jej nějak předzpracovat, a pak bychom rádi efektivně odpovídali na otázku: „Je řetězec S obsažen v adresáři?“ Můžeme také po předzpracování chtít přidávat nové položky i odebírat staré.

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

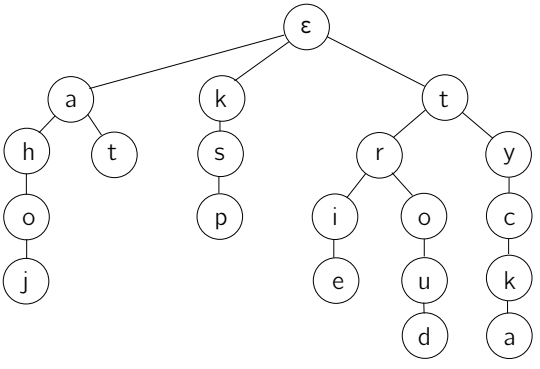
Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom, budeme jej stavět pro nějaký adresář A. Kořen bude odpovídat prázdnému slovu ε. Každá hrana, která z něj povede, odpovídá jednomu ze znaků, kterým slovo z adresáře A začíná, a to bez opakování (tedy jsou-li v A čtyři slova začínající na `a`, hranu vedeme jen jednu).

Na koncích těchto hran z kořene nám vznikly vrcholy, které odpovídají všem jednoznakovým prefixům slov z A, a už je celkem jasné, jak struktura dále pokračuje –

z každého vrcholu odpovídajícímu prefixu P vede hrana se znakem `c` právě tehdy, když slovo $P + \texttt{c}$ (za P přilepíme znak `c`) je také prefixem některého slova z A.

Obrázek vydá za tisíc definic, zde je postavená trie pro slova `ahoj, at, ksp, trie, troud, tyc, tycka`:



Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo z adresáře 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 z adresáře 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, anebo si rozšíříme abecedu o speciální znak (třeba `$`), který se v ní předtím nevyskytoval, a pak všem slovům z A přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo v adresáři, 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 v potomka přes hranu s písmenkem 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) na $\mathcal{O}(|\Sigma| \cdot D)$, kde D značí velikost vstupu, čili součet délek všech slov v adresáři. To je naprosto přijatelné pro malé abecedy, ale už pro `A-Za-z` je tento faktor roven 52 a pro Unicode je už taková alokace nemyslitelná.

Pokud tedy pracujeme s velkou abecedou, může se nám vyplatit oželet konstantní rychlost dotazu a použít v každém vrcholu vlastní binární vyhledávací strom pro znaky, kterými aktuální prefix může pokračovat. To zmírní časovou složitost konstrukce na $\mathcal{O}((\log |\Sigma|) \cdot D)$ a zhorší časovou složitost dotazu na slovo délky l na $\mathcal{O}(l \cdot \log |\Sigma|)$.

A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo v adresáři?“, 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 [Töpfer].
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova v adresáři. (Slovo *prefixové* je však v matematice hodně nadužívané (prefixová notace, prefixové kódy), a tak to může vést ke zmatení).
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v českém textu v lineárním čase. Můžeme přeci postavit adresář ze všech slov v daném textu, a pak procházet tu trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti. Druhak, pokud bychom použili jako oddělovač mezery, bychom mohli 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 [GrafAlg].

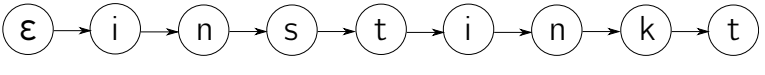
Cvičení

- Řekněme, že chceme adresář na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Můžeme použít nějaký klasický třídicí algoritmus, ale bohužel musíme počítat s tím, že porovnání dvou řetězců není konstantně rychlé. Vymyslete způsob, jak setřídit takový adresář 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.

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Chceme si slovo zpracovat, načež projdeme co nejrychleji text a zahlásíme jeden nebo všechny jeho výskyty. Často se hovoří o „hledání jehly v kupce sena“, a tedy se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme proměnnou j a délku textu n.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `instinkt`:



Mohli bychom text začít procházet písmenko po písmenku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly,

skočíme na další písmenko z textu a i na další písmenko v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další písmeno 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 $\mathcal{O}(nj)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(n + j)$. Ve skutečnosti nebylo vracení se to, co algoritmus zpomalovalo, 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 každé písmenko 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 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*.

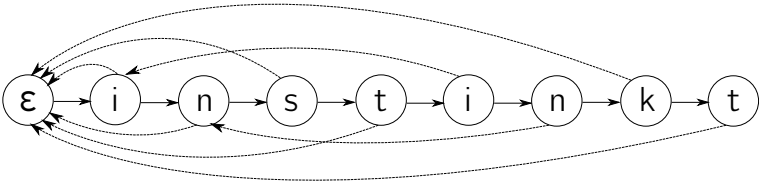
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `n` ve slově `instinkt`. Pracujeme teď s prefixem `instin`. Selsky řečeno, chceme najít „konec slova `instin` takový, že je stejný, jako začátek slova `instin`“.

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 `abababababc` 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 ještě jednou, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak 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 obohacený o zpětnou funkci (zakreslenou pomocí ukazatelů) takto:



Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? 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 lineární v délce textu.

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 prvních i znaků jehly bez prvního písmenka.

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 samotného slova j. 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 celou jehlu bez prvního písmena a sledovat, jakými stavy bude procházet, a 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 $\mathcal{O}(n + j)$. 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):

```
var
  Slovo: array[1..P] of char;   { jehla }
  Text: array[1..N] of char;    { seno }
```

```
  F: array[1..MaxS] of integer; { zpětná fce }

  function Krok(I: integer; C: char): integer;
  begin
    if (I < P) and (Slovo[I+1] = C) then
      Krok := I + 1
    else if I > 0 then
      Krok := Krok(F[I], C)
    else
      Krok := 0;
  end;

var
  I, R: integer;  { pomocné proměnné }
begin
  { konstrukce zpětné funkce }
  F[1]:= 0;
  for I:= 2 to P do
    F[I]:= Krok(F[I-1], Slovo[I]);

  { procházení textu }
  R:= 0;
  for I:= 1 to N do begin
    R:= Krok(R, Text[I]);
    if R = P then
      writeln(I);
  end;
end.
```

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* („okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaké písmeno na začátku a přidáme-li ho na konci.
- Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jako jsme řešili 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. Není to tak těžké vymyslet, pokud rozumíte tomu základnímu algoritmu.
- Algoritmus KMP je často zmiňován v souvislosti s *konečnými automaty*, protože náš postup skákání po spojovém seznamu se zpětnými hranami je vlastně jen

průchod konečným automatem. KSP ve 23. sérii vytvořilo seriál o regulárních výrazech, který teorii konečných automatů popisuje.

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.

Martin Böhm, Martin Mareš a Petr Škoda

Úloha 18-5-4: Detektýv

Slavný detektiv Šérlok Houmles je na stopě vážného zločinu. A to doslova a do písmene. S lupou až u země právě prohlíží stopy, které by ho měly dovést k pachateli. Pomůžete detektivovi s jeho případem?

Stopy jsou uspořádány do řady. Navíc každou stopu lze označit nějakým písmenem, nebo jiným znakem a těchto „typů“ stop není mnoho (desítky až stovky). Dále má Šérlok k dispozici Knihu Stopování Pachatelů, ve které jsou popsány všechny podezřelé výskyty stop.

Na vstupu dostanete všechny podezřelé sekvence stop a dále řetězec stop, které detektiv sleduje. Tento řetězec je velice dlouhý a nevejde se do operační paměti. Pro jednoduchost předpokládejte, že existuje funkce `GetFootprint`, která vrací právě přečtenou stopu (např. jako znak) a procedura `RewindFootprint`, která vrátí detektiva na začátek stop. Váš program by měl zjistit ke každé sekvenci podezřelých stop, kolikrát se vyskytla během stopování.

Zároveň si uvědomte, že času je málo, a tak by váš program měl pracovat ideálně v čase $\mathcal{O}(N+P)$ (N je délka stopovaného řetězce a P je součet délek všech podezřelých stop), bez ohledu na počet výskytů podezřelých sekvencí, přestože jejich počet může být až $\mathcal{O}(Nk)$, kde k je počet podezřelých sekvencí. Detektiv také nemůže stále běhat sem a tam, takže váš program by měl funkci `RewindFootprint` volat co nejméně (ideálně vůbec).

Nicméně i řešení v čase $\mathcal{O}(Nk+P)$ je daleko hodnotnější než řešení se složitostí $\mathcal{O}(NP)$.

Příklad: Podezřelé sekvence stop jsou `LPBBLP`, `BBBBO`, `OSSO`.
Prohledávané stopy buďtež `OSSOSSOLPBBLPBBLPBBBBO`.
Výstup programu by měl být

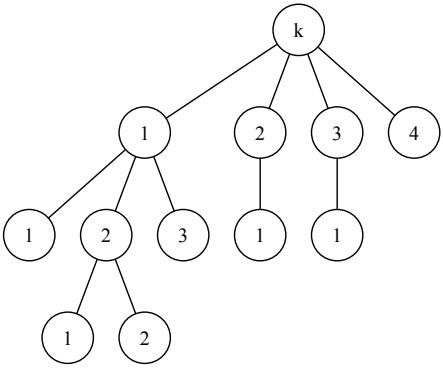
podezřelý vzorek	*počet jeho výskytů*
`LPBBLP`	2
`BBBBO`	1
`OSSO`	2

Úloha 22-4-4: Ořez stromu

V sadě jabloní se jako každé jaro ořezávají větve. Abychom si ověřili, že nám stromy nikdo neukradl a nevyměnil za nějaké atrapy, potřebujeme umět zjistit, jestli je-

den strom mohl vzniknout z druhého operací ořezávání, přičemž se nám pomíchaly informace a nevíme, který strom by měl vzniknout ořezáním z druhého.

V této úloze budeme za strom považovat souvislý graf bez kružnic s pevně daným kořenem. Navíc každý uzel má jasné uspořádání synů, takže podíváme-li se na některý vrchol, tak umíme vždy jasně říci, který syn je první, který je druhý, a tak dále.



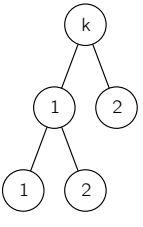
Jak probíhá takové ořezávání? Ze stromu se nejprve vybere vrchol (například první syn kořene z příkladu) a v tomto místě se ještě zvolí souvislý interval synů (např. druhý až třetí syn). Původně vybrané rozbočení se prohlásí za kořen, synové kořene budou jen ty vrcholy, které byly jeho syny ve vybraném intervalu, a uspořádání se zachová (druhý syn bude prvním synem nového kořene).

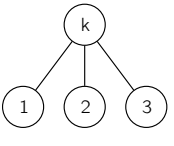
Spolu s tímto intervalem patří do ořezaného stromu také celé podstromy pod těmito syny. Vrcholy, které neležely v příslušném intervalu nebo byly jinde v původním stromě, v novém stromě prostě nebudou.

Na vstupu dostanete dva zakořeněné stromy a máte zjistit, jestli jeden mohl vzniknout ořezem druhého. Stromy mohou být zadány například takto: vrcholy si očíslujeme od 1 do N, kořenem bude vrchol s číslem 1 a na vstupu dostaneme pole spojových seznamů. i-tý prvek pole je spojový seznam, který obsahuje číselná označení synů i-tého vrcholu, uspořádaná zleva doprava. V této reprezentaci můžeme zadat „osekaný strom“ z obrázku níže například takto:

```
1: 2 3
2: 4 5
3:
4:
5:
```

osekaný strom





tento ale
nemohl vzniknout