Milí řešitelé a řešitelky!

Po malé přestávce zde konečně máme další sérii. Čtvrtá sada úloh 22. ročníku Korespondenčního semináře z programování je připravena. Termín jejího odevzdání budiž pondělí 26. dubna v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 22. dubna.

Elektronická řešení přijímáme na obvyklé stránce *https://ksp.mff.cuni.cz/submit/*. Náš certifikát sice není podepsán žádnou komerční autoritou, nicméně pro jeho ověření zde uvádíme jeho SHA1 hash:

`10:3B:C1:E2:4F:9A:01:98:DA:DB:5D:A0:D5:5C:80:57:DE:AD:28:C3`

Papírová řešení nám pak můžete zasílat klasickou poštou na adresu

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

Čtvrtá série dvaadvacátého ročníku KSP

Tentokrát se pro změnu přesouváme do daleké budoucnosti. Lidstvo dávno vymřelo, poslední člověk zmizel roku 4291. Země je osídlena roboty.

Dobrý den,
jmenuju se `CPU7DR3X` *a jsem výzkumný robot. Mám docela štěstí, že jsem výzkumný robot. Víte, roboti se dělí do několika skupin: zásobovací roboti, výrobní roboti, výzkumní roboti, ... Zásobovací roboti se starají, abychom měli vždy dostatek energie. Výrobní roboti vyrábějí různé spotřebiče (a samozřejmě i další roboty). My výzkumní roboti vymýšlíme nové věci, zkoumáme planetu a její historii, zkrátka zajišťujeme přísun nových informací.*

Nedávno nám opět centrála rozdělovala práci. Obvykle každému přiřazuje, co má zkoumat (samozřejmě i s termínem, do kdy se očekává slušný výsledek), ale tentokrát na mě nevyšla ŽÁDNÁ práce!! Nevím, jak se to mohlo stát, možná byla chyba v systému, nicméně měl jsem volno a chystal se toho využít. Hodlal jsem udělat nějaký velký objev, tak velký, že ani centrála by takový úkol nepřidělila, neboť by ho považovala za velmi složitý.

Zajel jsem k encyklopedii (to je takový velký počítač, ve kterém jsou uloženy všechny dosud známé informace), napojil jsem se a hledal nápady na to, co bych mohl objevit.

22-4-1 Zaheslované stránky — 10 bodů

`CPU7DR3X` se chce dostat na N různých stránek s důležitými informacemi. Stránky jsou naneštěstí zaheslované, na některých stránkách jsou kromě informací ještě hesla k některým z dalších stránek. Aby se `CPU7DR3X` dostal na všechny stránky, potřebuje u některých prolomit bezpečnostní systém, a protože prolamování je nebezpečná činnost, chce ji provádět co možná nejméně. Úlohu mu ulehčila rada od kolegů trojských koní, kteří mu vyzradili, že ke každé stránce existuje heslo nejvýše na jedné jiné stránce.

Na vstupu dostanete seznam stránek očíslovaných $1 \ldots N$ a pro každou i informaci, ke kterým jiným stránkám je na ní uvedeno heslo. Napište program, který řekne, kolik nejméně stránek musí `CPU7DR3X` prolomit, aby si mohl přečíst informace na všech stránkách.

Strávil jsem nad tím několik hodin, ale pak jsem zjistil něco úžasného – kdysi prý na této planetě žily bytosti, které si říkaly lidé. Už podle obrázku vypadali divně, neměli žádná kolečka, žádné anténky, žádné senzory ani kamerky, ale údajně to byli jedni z nejvyspělejších historických živočichů, uměli vyrábět různé nástroje, derivovat, psát. Ale našel jsem o nich i záznamy o tom, jak poškozovali planetu – pozměnili ji natolik, že na ní už ani nedokázali žít. Bylo mi divné, že taková vyspělá zvířata jsou ohledně zachování vlastního života velmi hloupá, a rozhodl jsem se, že zkusím takové zvíře zrekonstruovat. Tohle ještě nikdo nezkoušel – nejvyspělejší zvíře, které jsme dosud odchytili do laboratoře na pozorování, byl králík.

Nahrál jsem si do paměti všechny důležité informace a vyrazil jsem do terénu, abych získal podklady pro znovuoživení onoho divného zvířete. Potřeboval jsem sehnat DNA, neboť právě ona nesla veškeré informace o zvířeti.

A tak jsem se ještě toho dne vypravil hledat lidskou DNA. Napadlo mne, že nějaká by mohla být v muzeu. A tak jsem se vypravil k transportéru, který měl cestu kolem muzea, abych ušetřil pár hodin času. Během transportu jsem zaregistroval venku několik pracovních robotů hlasitě nadávajících na nezkontrolované zásilky zboží.

22-4-2 Rozvoz zásilek — 10 bodů

Transportér je jedno velké vozidlo, převážející zboží a součástky na jednosměrné, pravidelné trase. Trasa má několik zastávek – skladišť. Transportér zastavuje na všech zastávkách v pevném pořadí daném jejich čísly – naše měkké lidské mozky si mohou představit jednu autobusovou linku.

Zboží, které se průběžně ve skladištích nakládá a vykládá, může být buď v pořádku nebo poškozené, a aby se ušetřil čas (a pracovní roboti mohli dělat jiné užitečné věci), je v transportéru zboží skener, který mezi stanicemi zásilky zboží zkontroluje a označí je, zda jsou, nebo nejsou v pořádku. Organismy založené na uhlovodících si mohou představit jakéhosi hodného revizora, který nikoho z autobusu nevyhodí, jen vás označí. Na vaší výstupní stanici si už vás „přeberou“.

Naneštěstí ne vždy je skener plně nabitý, aby mohl kontrolovat zboží mezi všemi stanicemi. Určete, jak nastavit skener (rozhodněte, mezi kterými stanicemi se skener má aktivovat), aby zkontrolovaného zboží bylo v součtu co nejvíc, když víte, že skener je nabitý natolik, že je schopen kontrolovat zboží právě N-krát.

U každé stanice znáte počet kusů naloženého zboží a počet kusů zboží transportovaného do každé z následujících stanic (z tohoto nákladu). Pozor, každý kus zboží se započítává do součtu jen jednou, i když jste ho zkontrolovali během

celé cesty třeba osmkrát.

Očíslujme stanice $1 \ldots S$, kde S je počet stanic. Například pro zadání (formát `počet kusů:odkud->kam`):

`4:1->4, 2:1->2, 6:2->3, 3:2->4, 5:3->4`; $N = 2$

je správným řešením dvojice úseků `2-3`, `3-4`.

Stěžování pracovních robotů mne natolik zaujalo, že jsem málem přehlédl, že už jsem u největšího biologického muzea, kam jsem se mohl během dne dostat. Největší problém bude se nenápadně dostat do muzea a opatřit si lidskou DNA. Biologické materiály v muzeu jsou střeženy a označeny jen číselnými a písmennými kódy a pro jejich získání potřebují i výzkumní roboti povolení od centrály zadávání práce, že k pokusu biologický materiál skutečně potřebují. Povolení jsem neměl, a tak jsem se musel do muzea dostat potají, najít lidskou DNA a kus jí ukrást. To rozhodně nebylo jednoduché. Kdyby na mě přišli, mohli by mě i sešrotovat! Ale v zájmu vědy ...

Utěšoval jsem se, že až dokončím rekonstrukci člověka, všichni budou jásat nad mým objevem. A tak jsem se vydal k zadnímu vchodu do muzea, kudy se do něj dováží různé spotřebiče a materiály. Jelikož dovoz je častý a skoro pořád tam jezdí pracovní roboti, není to tam moc hlídané. Schoval jsem se za nedalekou bednu a pozoroval jsem zadní vchod. Za několik hodin přivezl transportér zboží a krabice. Chvíli nato vyjeli ven pracovní roboti, zboží popadli a vezli ho dovnitř. Vyjel jsem ze svého úkrytu, sebral nejbližší krabici a zamíchán mezi pracovní roboty jsem pronikl do muzea. V muzeu jsem se chodbami dostal až do výstavních míst, kde byly biologické materiály. Problém byl, že jsem neznal kód, pod kterým zde byla uchována lidská DNA.

22-4-3 Muzeum — 10 bodů

V muzeu v biologickém oddělení se uchovává spousta biologických materiálů, které jsou tématicky roztříděny a každému tématu je vyhrazena jedna místnost. Místností je K a každá je hlídána přesně $(N-1)/K$ kamerami. (N je celkový počet kamer, přičemž $(N-1)$ je vždy dělitelné K.) Kamery jsou navzájem pospojované dráty. Navíc se v muzeu nachází centrální kamera na chodbě, z níž vede právě jeden kabel do každé místnosti, jenž je napojen na jednu z kamer. (Přes ni se pak dalšími spoji lze dostat k libovolné kameře v místnosti.) Mezi místnostmi jinak dráty vůbec nevedou.

`CPU7DR3X` se podařilo zjistit, jaké kamery jsou spojeny drátem. Tím získal souvislý graf oddělení (snad biologického), v němž kamery jsou vrcholy a dráty mezi nimi neorientované hrany. V tomto grafu potřebuje najít centrální kameru, tedy vrchol, z něhož vede právě jedna hrana do každé místnosti a jehož odebráním by se graf rozpadl na K komponent souvislosti. Bohužel neví, jaká kamera patří do které místnosti. Navíc si vůbec není jist, jestli se dostal do biologického oddělení, a tak zároveň potřebuje zjistit, zda je v každé místnosti $(N-1)/K$ kamer.

Pokud například bude 10 kamer (očíslovaných od 1 do 10), 3 místnosti a spoje mezi kamerami `1-5`, `2-7`, `2-4`, `1-10`, `7-1`, `4-6`, `3-8`, `3-7`, `2-6`, `8-9`, má centrální kamera číslo 7 a nacházíme se skutečně v biologickém oddělení (v první místnosti jsou kamery 1, 5, 10, v druhé 2, 4, 6 a v třetí 3, 8, 9). Ovšem pro spoje `1-5`, `2-7`, `2-4`, `1-10`, `7-1`, `4-6`, `3-8`, `3-7`, `2-6`, `2-9` má jedna místnost 4 kamery a jiná jen 2, takže máme graf jiného oddělení.

A tak jsem prozkoumával kamery a zjišťoval jejich počet, abych si ověřil, že jsem ve správném oddělení. Muzeum bylo obrovské, ale díky rychlému prozkoumávání kamer jsem našel správné oddělení už za 8 hodin. Zbývalo to nejdůležitější – pomalu a pracně jsem se naboural do centrálního monitoringu a vypnul jsem monitorování v celém oddělení. Pak jsem se rychle vloupal do několika vitrín, sebral kusy vzorků a prchal jsem, abych byl co nejrychleji z muzea, nejlépe ještě dřív, než přijdou na výpadek jednoho z centrálních monitoringů. Již po chvíli se strhl poplach a panika ... ani si nepamatuju, jak jsem se v tom zmatku dostal z muzea. V laboratoři jsem pak vzorky pořádně prozkoumal a zjistil, že jeden z nich je opravdu lidská DNA!

Konečně jsem ho měl. Zbývalo jen to hlavní – probudit z něj k životu ono podivné zvíře. A tak jsem zajel do laboratoře. Věděl jsem, že k oživení zárodku bytosti je třeba DNA probudit k životu, ale protože u takového vyspělého zvířete to bude jistě náročné, rozhodl jsem si DNA namnožit – tak s ní budu moct dělat pokusy a vždy mi nějaká zbyde pro ty další.

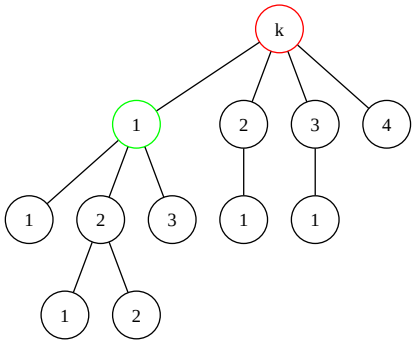
A tak jsem spoustu dalších týdnů strávil tím, že jsem se snažil probudit lidskou DNA k životu. Přidával jsem k ní všechno možné, píchal do ní injekce s rozličnými látkami, pracoval s ní v různých teplotách, vlhkostech i tlacích, ale stále bezvýsledně. Teprve po několika dalších týdnech jsem v jedné ze zkumavek zaregistroval něco, co připomínalo lidský zárodek. Izoloval jsem zárodek do skleněné nádoby, vložil ho do 3D mikroskopu s největším přiblížením (jaké bylo v budově dostupné) a pořádně ho prohlédl. Nebylo pochyb – byl to skutečně lidský zárodek. Okamžitě jsem jel vložit zárodek do urychlovače – naneštěstí se přístroj zasekl a zárodek se zničil. Prohlédl jsem svoje záznamy a okamžitě jsem začal pracovat na tvorbě dalších zárodků. Zárodky jsem střídavě mutoval a ořezával, aby mi vzniklo to správné zvíře.

22-4-4 Ořez zárodků — 10 bodů

`CPU7DR3X` dává zárodky v různém stupni mutace na ořezání špatných větví. Zárodek je různě rozvětvený, navíc je na něm poznamenáno místo, odkud větvení začíná.

Kdybychom měli špatný mikroskop, viděli bychom vlastně strom (souvislý graf bez kružnic) s pevně daným kořenem. Tyto stromy mají navíc jasné uspořádání synů, takže podívám-li se na některý vrchol, tak umím vždy jasně říci, který syn je první, který je druhý, a tak dále.

Není-li vám jasné, co takový strom je, podívejte se do naší Kuchařky o grafech. Ale hlubokou algebru v tom nehledejte, obrázek popisuje situaci více než dobře.



příklad: původní strom

Nyní je třeba zárodek ořezat. To se udělá tak, že se nejprve vybere rozbočení (vrchol, například první syn kořene z přikladu) 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,

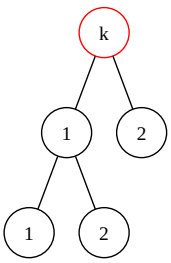
které byly jeho syny ve vybraném intervalu, a uspořádání se zachová (druhý syn bude prvním synem nového kořene). Spolu s tímto intervalem patří do ořezaného stromu také celé podstromy pod těmito syny. Vrcholy, které neležely v příslušném intervalu nebo byly jinde v původním stromě, v novém stromě prostě nebudou.

Naneštěstí je přístroj na ořezávání nedokonalý, takže občas oseká strom ne přesně tak, jak jsme popsali v odstavci výše. Navíc `CPU7DR3X` nemá v záznamech úplně pořádek a proto v nich má počáteční zárodek a ořezaný zárodek zaznamenány v nahodilém pořadí (tj. nelze předpokládat, že neořezaný je ten první). Od toho jste tu vy.

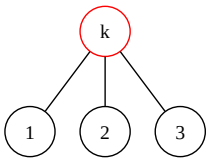
Na vstupu dostanete dva zárodky (zakořeněné stromy) a máte zjistit, jestli jeden mohl vzniknout ořezem druhého.

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, kde I-tý prvek pole je spojový seznam, který obsahuje číselná označení synů I-tého vrcholu, uspořádána 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 z původního vzniknout nemohl

(Psst! Zaslechl jsem organizátory si šuškat o tom, že některé úlohy tohoto ročníku se dají vyřešit pomocí přiložené kuchařky o řetězcích. Tahle to ale asi nebude, že? To by bylo ujeté... váš Štek Tiskařský, v.r.)

A tak jsem dával opakovaně do urychlovače zárodky, kontroloval, zda se dobře vyvíjejí, a s napětím čekal, kdy se mi konečně vyvine kompletní zvíře. Až jednou ... Hurá, konečně se to povedlo! Po otevření urychlovače se v něm hýbal nějaký bledý tvor. Zvíře bylo asi metr dlouhé a obrázku ze záznamu se podobalo jen velmi hrubě. Položil jsem zvíře na podlahu. Vstalo a postavilo se na dvě úzké tyčky s klouby. (Tyto tyčky tento druh zvířat prý nazýval „nohy“.) Otočil jsem se na zvíře, připravil jsem si počítač pro záznam a zadal zvířeti jednoduchou otázku:

„Kolik je desátá derivace z $x^{20}+4x^{18}-e^{x^{14}}\cdot x^7-x^{13\mathrm{tg}\ x}+x(10-e^{5x})-sin^{11}x$*?“ Zvíře se ke mně otočilo a nepatrně se mu rozšířily hnědé kuličky na vrchní kulaté části (těmhle věcem říkala ta zvířata oči). Udělal jsem si do počítače záznam, že zvíře derivovat neumí. Popadl jsem zvíře (samozřejmě se vzpouzelo a vydávalo různé otravné zvuky) a zavezl jsem je do skladu součástek. Tam jsem zvíře pustil a (zatímco se rozhlíželo kolem) jsem mu zadal jiný jednoduchý úkol: „Vyrob solární panel!“ Zvíře se na mne notnou dobu dívalo, a pak se slabým hláskem zeptalo: „Co je to solární panel?“ „Ty nevíš, co je solární panel?“ rozkřikl jsem se a udělal záznam o tom, jak je zvíře neuvěřitelně hloupé. Takovéhle bytosti že prý byly nejvyspělejší? To to tu bylo před desítkami tisíc let velmi zaostalé! Prohledal jsem svou paměť, abych si ověřil, že zvíře není příliš staré – stará zvířata byla podle záznamů mnohem blbější než mladá. Zvířata začínala podle záznamů blbnout stářím asi kolem 70 let. Otočil jsem se na zvíře a zeptal jsem se: „Kolik je ti let?“ Zvíře odpovědělo: „Je mi pět let.“ Tady něco nehrálo. Zvíře nebylo staré, a přesto nefungovalo tak, jak by podle záznamů fungovat mělo. Po projití dat v paměti jsem si připomněl, že příliš mladá zvířata toho také moc neumí. Přes protesty a pohyby zvířete jsem je popadnul a odvezl je do urychlovače. Potřeboval jsem, aby bylo o něco starší. Dal jsem zvíře do urychlovače, zaklapl víko a přes jeho vřeštění jsem přístroj zapnul. Řev slábnul a když na přístroji zasvítily kontrolky, otevřel jsem urychlovač. Zvíře se ohnulo v půlce, slezlo z urychlovače a já mu opět položil otázku: „Kolik ti je let?“*

Tentokrát zvíře odpovědělo: „Je mi třiadvacet ... “ Zadal jsem mu opět jednoduchý příklad na derivování. Tentokrát zvíře chtělo něco na psaní. Ukázal jsem mu zastaralý model počítače, který se často sekal (přeci jen, nemohu riskovat, že by zvíře zničilo nějaký drahý přístroj), zvíře se uvelebilo před obrazovkou, zmáčklo tlačítko, ale starý počítač se nenastartoval. Co to? Po bližším zkoumání se ukázalo, že v počítači nebyl energetický článek. Otevřel jsem zásobník článků a začal uvažovat, který dát do počítače.

22-4-5 Energetické články — 10 bodů

`CPU7DR3X` otevřel zásobník, ve kterém je N různých druhů energetických článků. Každý článek se vyznačuje tím, že je souměrný vzhledem k tomu, z jakých částic se skládá. (Např. `RAR` je energetický článek, `RAN` není energetický článek. My neroboti říkáme, že článek je palindromem.) Články lze spojovat, ale jen tehdy, pokud vzniklý článek je opět souměrný. `CPU7DR3X` chce vložit do počítače článek složený ze dvou článků a zajímá ho, z kolika různých příslušných uspořádaných dvojic článků si může vybrat.

Na vstupu dostanete N různých článků, zadaných slovním schématem (N slov, palindromů). Vaším úkolem je napsat program, který spočítá, kolik uspořádaných dvojic (dvojici přečteme jako jedno slovo zleva doprava) opět tvoří energetický článek (palindrom).

Například dvojice (`RAR`, `ARA`) palindrom netvoří, ale dvojice (`BAB`, `BABBAB`) palindrom tvoří.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx: *http://codex2.ms.mff.cuni.cz/ksp*. Pokud jste zatím žádnou praktickou úlohu neřešili, přečtěte si stručný úvod: *http://ksp.mff.cuni.cz/about/codex.html*.

Po úspěšném zapojení počítače začlo zvíře konečně počítat příklad. Přitom se mnou mluvilo a kladlo spoustu otázek – zjevně se mu okolní prostředí líbilo a zajímalo ho. Udělal jsem si záznam, že ve 23 letech zvíře derivovat umí, a právě jsem si chtěl zapsat, jak dlouho mu to trvá, když se rozletěly dveře a dovnitř vjelo a vlétlo několik masivních bezpečnostních robotů. Někteří mne popadli a někam mne odváželi (a dočasně mi vyzkratovali kamerky), poslední, co jsem zahlédl, bylo, že ostatní se vrhli na zrekonstruované pokusné zvíře.

Když jsem konečně mohl vnímat obraz, byl jsem znehybněn uprostřed malé místnosti a okolo mne bylo asi osm robotů, kteří na mne přísně shlíželi. Jeden z nich popojel ke mně a řekl: „`CPU7DR3X`, toto je tvůj soud za ohrožení bezpečnosti všech robotů!!“ Zeptal jsem se, co jsem provedl.

A ten robot popojel ještě o kousíček blíž, až jsem se bál, že do mne naboural, a zahřímal: „Cos provedl?! Ohrozil jsi celou naši existenci obnovením velmi nebezpečného druhu!“ Nechápal jsem, jak by nám jeden člověk mohl být nebezpečný, a obhajoval jsem se, že jsem učinil významný objev, ale nic mi nevysvětlovali a odešli do vedlejší místnosti se radit, jak mne potrestají.

Přemýšlel jsem o tom, jak se dozvěděli, že jsem vyrobil člověka, a vzpomněl jsem si, že mezi roboty již dlouho kolují nepodložené informace, že všechny činnosti robotů pozoruje jakési Tajné Bratrstvo, které má všude rozmístěny skryté kamery (a které tudíž muselo hned zaznamenat, že jsem odpojil centrální monitoring v muzeu). Nikdy jsem těmto zkazkům nevěřil – až dnes jsem se přesvědčil, že jsou pravdivé.

Když se Bratrstvo vrátilo, oznámil mi jeden z nich, že s konečnou platností přestávám být výzkumným robotem a přeřazuji se mezi roboty pracovní. Pak mne převezli do laboratoře, kde mi z paměti vymazali všechna hesla a přístupové kódy do místností sloužících k výzkumu, sběru informací, prostě do místností informačního a výzkumného charakteru. A tak jsem teď jen pracovním robotem. Pracovním robotem té nejnižší kategorie.

22-4-6 Umisťování panelů — 13 bodů

`CPU7DR3X` byl přeřazen mezi nejnedůvěryhodnější pracovní roboty. Nyní místo výzkumu bude s ještě jedním pracovním robotem umisťovat na louku panely pro příjem větrné a sluneční energie. Louka je rozdělena pomocí kolíků na čtvercovou síť (kolíky jsou v rozích čtverců), v některých čtvercích mají být umístěny panely (jeden nebo i více). Na louce má být celkem umístěno N panelů. Na louce budou stavět panely dva nedůvěryhodní roboti, z nichž každý má postavit K panelů. A protože ti dva pracovní roboti jsou nedůvěryhodní, je potřeba v rámci zabezpečení nalézt pro každého z nich obdélníkový prostor, kde má být celkem postaveno K panelů, a tyto prostory ohraničit z kolíků bezpečnostními paprsky. Jelikož na paprsky je potřeba energie v závislosti na délce paprsku (kterou se pokud možno šetří), má součet obvodů vytyčených pracovních míst být co nejmenší. Oba pracovní prostory se navíc nesmějí překrývat a pokud někde mají společnou hranu, je stejně potřeba v tomto místě vést paprsek pro každou pracovní plochu zvlášť (tj. v takové hraně budou dva paprsky).

Na vstupu dostanete na prvním řádku rozměry louky (délka a šířka), na dalším čísla N a K, a na dalších N řádcích dostanete souřadnice pro umístění jednotlivých panelů (délka a šířka) – na každém řádku jeden. Napište program, který vypíše minimální počet jednotek (jednotka – mezi dvěma kolíky) paprsků, které jsou potřeba k ohraničení dvou disjunktních obdélníkových pracovišť (z nichž na každém má být dle plánu K panelů) nebo vypíše, že pro příslušná data řešení neexistuje.

Bodování:

- max. 13 bodů za řešení rychlé při $N, K \leq 300$,
- max. 10 bodů za řešení rychlé při $N, K \leq 50$.

Nevím, co udělali s člověkem, ale nejspíš ho rozebrali. To je mi moc líto, takový velký objev jsem učinil a nikdo to neocení. Když mě propouštěli ze soudní místnosti, jeden z nich mi dokonce pošeptal „Máš štěstí, moh' jsi skončit ve šrotu . . . “

22-4-7 Pozdrav z pravěku — 12 bodů

Láká vás podívat se na programovací jazyk z dob, kdy muži byli ještě praví muži, ženy pravé ženy a počítače praví mamuti, aspoň co do velikosti? Jazyk, v němž opravdoví programátoři píší programy jen tehdy, když se vejdou na jeden řádek, což je ovšem překvapivě často? Jazyk, který má tolik jednoznakových příkazů, že by se na psaní všech těch znaků hodila klaviatura od varhan? Pak jste na správné adrese, protože dnes si budeme povídat o jazyku APL.

Historie APL se začala psát v roce 1964, kdy Kenneth Iverson dumal nad tím, proč chce-li něco jednoduchého spočítat, musí kvůli tomu psát složitý program, když přitom celý výpočet lze příjemně a stručně popsat matematickou notací. Chvíli experimentoval, až napsal interpreter jazyka založeného právě na matematické notaci a k tomu knížku s prostým názvem A Programming Language. A APL bylo na světě. Pojďme si ukázat pár jeho nejdůležitějších vlastností a zkusit si v něm zaprogramovat. Mezi řečí občas potkáte jednoduché úkoly, tentokrát po vás budeme chtít, abyste vymysleli co nejelegantnější (zejména co nejkratší) řešení bez ohledu na časovou složitost.

Předně bychom měli vědět, že *APL je interaktivní* – když mu napíšete výraz, interpreter ho ihned vyhodnotí a vypíše vám výsledek (v dnešní době nic moc překvapivého, ale v době dřevěných počítačů . . .). Třeba na `1+2+4+8` vypíše `15`, na `3*6` vypíše poněkud nečekaně `729`, protože `*` značí umocňování, kdežto násobení se značí `×`. Tedy `3×6` je `18`. I ostatní operace se píší trochu nezvykle: dělení je `÷`, zbytek po dělení `|` (pozor, má opačně argumenty, takže $3 \mid 7 = 7 \bmod 3 = 1$). Ještě nás asi překvapí, že `3×4+1 = 15`, protože neexistují priority operací a vše se striktně vyhodnocuje zprava doleva, pokud neurčíte jinak závorkami.

Většina operací existuje ve dvou formách: *dyadické* (ty chtějí dva argumenty, dneska bychom jim asi spíš říkali *binární*) a *monadické* (žádají jeden argument, jinak též *unární*). Obvykle obě formy počítají něco podobného: například dyadické `x÷y` dělí a monadické `÷y` počítá převrácenou hodnotu. Celý zvěřinec *aritmetických operací* vypadá takto:

`x+y`	sčítání	`+x`	identita (vrací `x`)
`x-y`	odčítání	`-x`	otočení znaménka
`x×y`	násobení	`×x`	signum (viz níže)
`x÷y`	dělení	`÷x`	$1/$`x`
`x|y`	y mod `x`, `0|x=x`	`|x`	absolutní hodnota
`x⌈y`	maximum	`⌈x`	horní celá část
`x⌊y`	minimum	`⌊x`	dolní celá část

Operace `×x` vrátí jedničku pro kladné `x`, `-1` pro záporné `x` a nula od nuly pojde.

Logické operace se zapisují jako v matematice, přičemž na vstupu považují 0 za nepravdu a cokoliv nenulového za pravdu; na výstupu dávají vždy 0 nebo 1. Podobně při porovnávání čísel dostanete vždy výsledek 0 nebo 1:

`x∨y`	nebo	`x<y`	menší než
`x∧y`	a zároveň	`x>y`	větší než
`~x`	negace	`x≤y`	menší nebo rovno
`x=y`	rovnost	`x≥y`	větší nebo rovno
`x≠y`	nerovnost		

Přiřazení funguje, jak čekáte, a značí se šipkou: `x←5` přiřadí do proměnné `x` hodnotu `5`. Příjemné je, že se chová jako funkce a vrací hodnotu, kterou přiřadilo, takže můžeme psát (podobně jako třeba v Céčku) `x←y←5`.

Můžete si také *definovat vlastní operace.* Nejjednodušší je to pro monadické: `f x: x*2` zavede monadickou operaci `f`,

která vrátí druhou mocninu svého parametru. Kdybychom chtěli definovat dyadickou operaci `g`, řekněme pro součet druhých mocnin, napíšeme `x g y: (f x) + (f y)`. Funkce více parametrů dostávají parametry ve složených závorkách oddělené středníky: `max{a;b;c}: a⌈b⌈c`.

Úkol 1 (1 bod): Nadefinujte operaci `×x` (signum) pomocí ostatních operací.

APL je vektorové – v podstatě všechno, co umí provádět s čísly, dokáže i s posloupnostmi čísel (čili vektory). Například `1 2 4 + 3 1 2` sečte vektory `1 2 4` a `3 1 2` po prvcích, takže vyjde `4 3 6`. Pokud k vektoru přičítáme konstantu, přičte se ke každému prvku: `1 2 4 + 1 → 2 3 5`. (Zde raději místo rovnítka značíme výsledek šipkou, protože `=` by se jako každá jiná operace na vektory aplikovalo po prvcích.) Obecně to funguje takto: Pokud použijeme na vektor monadickou operaci, provede se s každým prvkem zvlášť. Použijeme-li dyadickou na dva stejně dlouhé vektory, provede se po prvcích (první s prvním, druhý s druhým atd.). Pokud dyadickou na vektor a číslo, z čísla se vyrobí vektor tím, že se číslo zopakuje.

Často potřebujeme vyrobit vektor čísel $0, \ldots, n-1$. Tehdy se hodí operátor `⍳n` (*iota*), který přesně takové vektory vytváří.

Mimo vektorů APL dokáže pracovat i s vícerozměrnými poli: `⍳ a b c` vytvoří trojrozměrné pole velikosti `a × b × c` a vyplní ho čísly od 0 do `(a×b×c)-1`. Obecně můžete operátoru `⍳` dát jako parametr libovolný vektor a on vás obdaří polem, které má tolik rozměrů, kolik je délka vektoru, přičemž každá složka určuje, jak je pole v daném rozměru velké. Počtu rozměrů se říká *rank* pole. Dvojrozměrným polím budeme říkat *matice.*

Úkol 2 (1 bod): Co udělá `⍳5+1` a co `1+⍳5`? (Nezapomeňte, v jakém pořadí se vyhodnocují operace.)

Úkol 3 (1 bod): Co udělá `⍳(2+⍳3)`?

Pokud chceme zjistit, jak je nějaké pole velké, stačí použít monadický operátor `⍴` (*ró*). Ten vrátí vektor, jehož jednotlivé složky jsou velikosti v jednotlivých rozměrech. Tedy `⍴⍳2 3 4` vrátí `2 3 4`. Je-li `x` vektor, `⍴x` musí tedy být jedno číslo, které udává počet prvků vektoru. Proto `⍴⍴p` vrátí rank pole `p`.

Ještě jsme ale nepřišli na to, jak vícerozměrné pole zadat. K tomu se hodí dyadická forma operace `⍴`. Ta slouží k *přeformátování* pole na dané rozměry: `2 3 ⍴ 4 5 6 7 8 9` například vezme vektor `4...9` a přerovná ho na matici o dvou řádcích `4 5 6` a `7 8 9`. Pokud má původní pole příliš málo prvků, začnou se opakovat: `2 3 ⍴ 1 2` vyrobí matici s řádky `1 2 1` a `2 1 2`. Přerovnávání přitom zachovává standardní pořadí prvků: vektor se čte zleva doprava, matice se čte po řádcích, vícerozměrné tak, že se první rozměr mění nejpomaleji a poslední nejrychleji.

Pole můžeme také indexovat (od nuly): pokud je `x` vektor, pak `x[0]` je jeho nultý prvek, `x[1]` první atd. Jako index můžeme také použít vektor, v tom případě vybereme více prvků najednou: `x[0 2 3]` dá vektor skládající se z prvků `x[0] x[2] x[3]`. U vícerozměrných polí se indexy oddělují středníkem: `y[1;3]` je třetí prvek na prvním řádku matice `y`, `y[2 3;2 3]` je čtvercová podmatice 2×2 „vykousnutá“ z matice `y`. Na pole ranku k se také můžeme dívat jako na vektor, jehož prvky jsou pole ranku $k-1$, tedy pro matici `y` je `y[0]` její první řádek, `y[1]` druhý atd.

Existuje řada dalších operací, které zacházejí s vektory. Zajímavé je třeba spojování vektorů za sebe nebo pod sebe. Spojení za sebe se zapisuje čárkou: `x,y` je vektor, který obsahuje nejprve všechny prvky vektoru `x` a pak všechny prvky z `y`. Spojení pod sebe neboli *laminace* `x⍪y` má za výsledek dvojřádkovou matici, jejíž prvním řádkem je vektor `x` a druhým vektor `y` (řádky přitom musí být stejně dlouhé). Obě operace samozřejmě fungují i na vícerozměrná pole: čárka spojuje v prvním z rozměrů (matice tedy slepuje pod sebe), laminace přidá na začátek nový rozměr, takže `(x⍪y)[0]→x` a `(x⍪y)[1]→y`.

Hodit se nám bude též *redukce* `+/x`. Ta spočte pro vektor $\mathtt{x} = x_0 x_1 \ldots x_{n-1}$ výraz $x_0 + x_1 + \ldots + x_{n-1}$, tedy součet všech prvků. Podobně můžete pomocí lomítka vyrobit z nějaké operace její redukční verzi zpracovávající postupně prvky vektoru: například `⌈/x` vrátí maximum z vektoru.

U vícerozměrných polí se redukce chová trochu záludněji. Pokud použijeme `+/` například na matici, což, jak už víme, je vektor řádkových vektorů, sečte nám jednotlivé řádky, takže vznikne vektor, jehož i-tá složka vyjadřuje součet i-tého sloupce matice. Třeba pro matici

$$\mathtt{x} = \mathtt{⍳3\ 4} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

spočítáme `+/x → (0 1 2 3)+(4 5 6 7)+(8 9 10 11) → 12 15 18 21`.

Co kdybychom naopak chtěli sčítat prvky v každém řádku? Tehdy je nejjednodušší použít operaci transpozice `⍉`, která prohodí obě souřadnice – řádky se nyní stanou sloupci a naopak. Pro řádkové součty tedy stačí `+/⍉x`.

Úkol 4 (1 bod): Jak spočítat maximum ze všech prvků matice?

Na lomítko se můžeme dívat jako na něco, co dostane dyadickou operaci s čísly a vyrobí z ní monadickou operaci s vektory (nebo obecněji z dyadické operace s poli ranku k vyrobí monadickou operaci s polem ranku $k+1$). Takových konstrukcí, které vyrábějí z operací jiné operace, má APL víc a říká se jim *operátory.*

Dalším důležitým operátorem je *direktní součin* `∘.f`, kde `f` je nějaká dyadická operace. Pokud ho použijeme na vektory `x` a `y` (tedy napíšeme `x∘.fy`), dostaneme matici, jejíž prvek na pozici (i, j) obsahuje výsledek operace `f` aplikované na i-tý prvek vektoru `x` a j-tý prvek vektoru `y`. Výraz `(⍳10)∘.×(⍳10)` nám tedy vytiskne malou násobilku. Direktní součiny samozřejmě fungují i s poli vyšších ranků; snadno si domyslíte, jak, když napovíme, že součinem pole ranku k s polem ranku ℓ bude pole ranku $k + \ell$.

Potkali jsme tedy zatím tyto operace pracující s poli:

`⍳n`	iota: generátor přirozených čísel
`⍴x`	zjištění rozměrů pole
`x⍴y`	přeformátování pole `y` na rozměry `x`
`x[...]`	indexování pole
`⍉x`	transpozice (překlopení)
`x,y`	spojení za sebe
`x⍪y`	laminace (spojení pod sebe)
`f/x`	operátor redukce
`x∘.fy`	operátor direktního součinu

S prohlídkou dalších operací a operátorů počkáme do příště, i s těmi, co už známe, se dá naprogramovat ledacos:

Úkol 5 (2 body): Napište funkci, která vytvoří matici $n \times n$ z nul a jedniček, která bude mít na i-tém řádku i jedniček a za nimi $n - i$ nul.

Úkol 6 (3 body): Jak pro dané sudé n vyrobit vektor $0, n-1, 1, n-2, 2, n-3, \ldots, n-1, 0$?

Úkol 7 (3 body): Vymyslete, jak v APL spočítat největšího společného dělitele dvou čísel.

Ještě dodejme, že jde vytvářet i delší programy – stačí napsat více řádků a na každém jeden výraz, nebo případně více výrazů na řádek oddělených středníkem. Pokud byste si chtěli nainstalovat opravdový překladač APL, podívejte se na odkazy ve webové verzi tohoto zadání. Fonty a TEXová makra na sázení APL najdete tamtéž. Pokud se vám nechce nic instalovat, můžete samozřejmě psát programy na papír a nemajíce varhany, psát místo podivných znaků jejich názvy.

APL a dnešní doba. Masového rozšíření se APL nikdy nedočkalo (snad proto, že jen opravdovým programátorům vyhovuje programovat tak, že hodinu se zavřenýma očima přemýšlejí, načež si sednou k počítači a napíší jednořádkový program). Přesto ale není pouze mrtvým jazykem vystaveným v muzeu počítačové prehistorie – stále vznikají nové implementace APL (například `A+` nebo dokonce `APL.Net`) a jazyky od APL odvozené (třeba jazyk `J`, který se snaží vystačit si bez varhan, tedy se znaky na běžné klávesnici). Navíc se s nástupem víceprocesorových počítačů ukazuje, že programy skládané pomocí vektorových operací a operátorů, jako je třeba naše redukce a direktní součin, jsou velice dobře paralelizovatelné. Kdo ví, co se ještě o APL naučíme.

Recepty z programátorské kuchařky

V dnešním vydání kuchařky se podíváme na vyhledávání slov v textu. Náš úkol tentokrát zní: Máte seznam slov a *hodně dlouhý* text, vypište všechny výskyty těchto slov v textu. Ukážeme si řešení, kterému stačí jeden průchod textem a lineární čas na předzpracování slovníku.

Pro začátek si zavedeme několik pojmů:

- Mějme nějakou konečnou *abecedu* Σ, tedy množinu všech *znaků.* Klidně si představujte klasickou latinskou abecedu, ale může to být např. i množina $\{0, 1\}$.
- Σ^* je množina všech *slov*, která lze z naší abecedy utvořit. To jsou všechny konečné posloupnosti znaků z Σ. Takové slovo může tudíž být i posloupnost 01101. Slova budeme značit řeckými písmenky a zvláštní postavení mezi nimi má *prázdné slovo* ε.
- $|\alpha|$ pro $\alpha \in \Sigma^*$ je *délka* slova, tedy počet jeho znaků.
- $\alpha\beta$ pro $\alpha, \beta \in \Sigma^*$ je *zřetězení* slov α a β, tedy slovo, které vznikne zapsáním slov α a β za sebe.
- γ^k je slovo vzniklé k-násobným zopakováním slova γ. Tedy $\gamma^0 = \varepsilon$, $\gamma^{k+1} = \gamma^k\gamma$.
- Slovo α nazveme *podslovem* slova β, pokud je α obsaženo v β, čili pokud $\beta = \gamma\alpha\delta$ pro nějaká slova γ a δ.
- Řekneme, že slovo α je *prefixem* slova β, pokud slovo β začíná slovem α, čili $\beta = \alpha\delta$ pro nějaké slovo δ.
- Podobně α je *suffixem* slova β, pokud β končí slovem α, tedy $\beta = \delta\alpha$ pro nějaké slovo δ.
- Každé slovo je prefixem i suffixem sebe sama, takovému pre-/suffixu říkáme *nevlastní;* všem ostatním *vlastní.*
- Všimněte si, že prázdné slovo je podslovem, prefixem i suffixem každého slova včetně prázdného slova.

Po tomto teoretickém úvodu se konečně zamyslíme nad vlastním vyhledáváním. Ponejprv si úlohu trochu zjednodušme a zkoumejme případ, kdy hledáme všechny výskyty jednoho slova $\alpha \in \Sigma^*$ o délce $|\alpha| = p$ v textu $\beta \in \Sigma^*$, $|\beta| = n$. (Hledanému slovu se často říká jehla, textu kupka sena.)

Asi první algoritmus, který nás napadne, je procházet text β od začátku až do konce a pro každou pozici i v textu zkontrolovat, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provedeme až p porovnání znaků, čili celkem až np porovnání. To není nic pěkného, zkusme to lépe.

Všimněme si, že porovnávání slova s textem může skončit dvěma způsoby. Buď zjistíme, že se slovo s textem shoduje celé, nebo najdeme v textu znak, který ve slově není. Tehdy nestačí pokračovat novým vyhledáváním od místa, kde jsme skončili: např. pro slovo `instinkt` a text `instinstinkt` by algoritmus u druhého `s` zjistil, že se text liší, a pokud by pokračoval dále, již by nenalezl skutečný výskyt slova. Proto se vždy musíme vrátit o kousek zpět, v předchozím algoritmu jsme se vraceli vždy těsně za místo, kde se text začal se slovem shodovat.

Na druhou stranu, když se takto vrátíme, začneme znovu zpracovávat text, který už jsme jednou četli, takže je vlastně předem dáno, jak to dopadne. Pojďme toho využít. Říkejme *stavy* prefixům slova α. Pro každou pozici i v textu si označme $r[i]$ nejdelší stav, který je obsažen v textu tak, že v něm končí na pozici i (nebo vezměme nejdelší suffix prvních i znaků textu, který je stavem – to je totéž). Posuneme-li se v textu o pozici dále, další znak $\beta[i+1]$ buď prodlouží prefix $r[i]$, a tím určitě získáme nový nejdelší stav $r[i+1]$ (rozmyslete si, že nemůže existovat delší), nebo už prefix není možné prodloužit, a tehdy budeme muset najít jiný. Nahlédněme ale, že useknutím posledního písmenka stavu získáme zase stav, takže useknutím posledního písmenka stavu $r[i+1]$ získáme nějaký suffix stavu $r[i]$. Naše $r[i+1]$ tedy vznikne prodloužením co možná nejdelšího suffixu stavu $r[i]$ o písmenko $\beta[i+1]$ (některé suffixy prodloužit nejdou, vezměme nejdelší, který jde). Pro předchozí příklad a prefix `instin` to bude suffix `in`.

Jelikož nový stav získáme ze suffixů předchozího stavu, nemusíme vědět vůbec nic o předcházejících písmenech textu. Postačí nám předpočítat si pro každý stav σ jeho nejdelší vlastní suffix, který je také stavem – ten si označíme $f(\sigma)$ a funkci f budeme říkat *zpětná funkce.* Přechod od $r[i]$ k $r[i+1]$ budeme provádět tak, že zkusíme $r[i]$ prodloužit o znak $\beta[i+1]$ a když to nepůjde, zkrátíme si $r[i]$ pomocí zpětné funkce a opět zkusíme přidat tentýž znak, pokud to stále nejde, zkracujeme dál opětovným zavoláním zpětné funkce, dokud se nám prodloužení nezdaří nebo dokud nedostaneme prázdné slovo.

Když navíc během výpočtu narazíme na i, pro které je $r[i] = \alpha$, ohlásíme výskyt slova α.

Aby se nám se stavy v programu pohodlně pracovalo, očíslujeme si je – j-tý stav bude prefixem slova α o délce j. Zpětná funkce pak bude přiřazovat číslům čísla, takže si ji můžeme pamatovat v obyčejném jednorozměrném poli.

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 p-krát. Při každém volání však klesne délka aktuálního stavu 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. Všimněte si, že $f(i)$ je přesně stav, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\alpha[2 \ldots i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco $r[i]$ 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 w. Jenže k vyhledávání zase potřebujeme funkci f. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $f(1) = \varepsilon$. 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 celý řetězec $\alpha[2 \ldots p]$ 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 $p-1$, a proto poběží v čase $\mathcal{O}(p)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(n+p)$. 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.
```

Tento algoritmus můžeme také formálně popsat pomocí automatů:

Konečný automat nad abecedou Σ si můžeme představit jako stroj, kterému dáme slovo ze Σ^* a on ho buď odmítne nebo přijme. V průběhu práce je vždy v právě jednom *stavu* z nějaké pevné množiny stavů. Slovo zpracovává po jednotlivých znacích a podle přečteného znaku se rozhodne, do jakého stavu přejde. K tomu slouží *přechodová* funkce g, která dvojicím (*aktuální stav*, *nový znak*) přiřazuje nové stavy. Pokud vstupní slovo dojde, automat podle toho, v jakém stavu se právě nachází, odpoví, že je slovo přijato nebo odmítnuto.

Konečný automat můžeme formálně nadefinovat jako čtveřici (Q, g, q_0, F), kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která pro daný stav automatu a znak na vstupu řekne, do jakého stavu má automat přejít;
- $q_0 \in Q$ je *počáteční stav,* v němž je automat na počátku výpočtu;
- $F \subset Q$ je množina *přijímacích stavů.*

Výpočte konečného automatu pak probíhá následovně:

1. Nastav aktuální stav s_0 na počáteční stav q_0.
2. Postupně čti znaky $x[i]$ ze vstupu a po přečtení každého přejdi ze stavu s_{i-1} do stavu $s_i = g(s_{i-1}, x[i])$.
3. Pokud skončíš v přijímacím stavu ($s_n \in F$), pak slovo přijmi.

Příklad: Mějme automat nad abecedou $\Sigma = \{0, 1\}$ se třemi stavy $s_1 \ldots s_3$, počátečním stavem $q_0 = s_1$, jedním přijímacím stavem $F = \{s_1\}$ a přechodovou funkcí g dle tabulky:

$$\begin{array}{ll} g(s_1, 0) = s_3 & g(s_2, 1) = s_3 \\ g(s_1, 1) = s_2 & g(s_3, 0) = s_3 \\ g(s_2, 0) = s_1 & g(s_3, 1) = s_3. \end{array}$$

Tento automat přijímá právě slova ve tvaru $(10)^k$, $k \geq 0$, tedy např. 101010 a prázdné slovo přijme, zatímco 1010101 odmítne.

Konečné automaty docela dobře popisují chod našeho algoritmu – ten také zpracovává text po znacích a přechází podle právě přečteného znaku mezi stavy. Jsou zde ale ještě některé rozdíly: předně KMP neodpovídá ano/ne, ale hlásí jednotlivé výskyty. K tomu můžeme automat upravit například tak, že množinu přijímacích stavů bude používat nejen na konci vstupu, ale v každém kroku. Druhá odlišnost tkví v tom, že přechodová funkce KMP (ta odpovídá prodlužování prefixu o další písmeno) není definována všude. Tam, kde definována není, nastupuje místo ní zpětná funkce, která nás přesouvá mezi stavy tak dlouho, než přechodová funkce definována je.

Tomuto rozšíření se obvykle říká *vyhledávací automat* a definuje se jako pětice (Q, g, f, q_0, out), kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která je definovaná pouze pro některé dvojice (*stav*, *znak*);
- $f : Q \to Q$ je *zpětná funkce,* která říká, do jakého stavu se má automat přesunout, pokud přechodová funkce není definována;
- $q_0 \in Q$ je *počáteční stav,* v němž se automat nachází na začátku výpočtu;

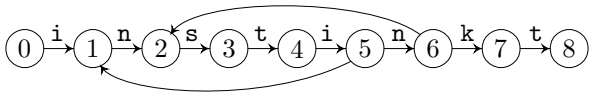
- $out : Q \to \mathcal{P}(\Sigma^*)$ je *výstupní funkce,* která každému stavu přiřazuje, jaký se v něm má ohlásit výstup, což bude množina nalezených slov. (V případě KMP byla vždy buďto prázdná nebo jednoslovná, až budeme za chvíli hledat více slov, bude bohatší.)

Výpočet vyhledávacího automatu pak probíhá následovně:

1. Nastav aktuální stav s na počáteční stav q_0.
2. Pro každý znak $c = x[i]$ vstupního textu proveď:
3. Dokud je $g(s, c)$ nedefinovaná, přejdi zpět do stavu $s \leftarrow f(s)$.
4. Přejdi do nového stavu $s \leftarrow g(s, c)$.
5. Vypiš všechna slova z $out(s)$.

Ještě doplňme, že aby se algoritmus vždy zastavil, musí být $g(q_0, c)$ definováno pro každý znak $c \in \Sigma$, obvykle opět jako q_0.

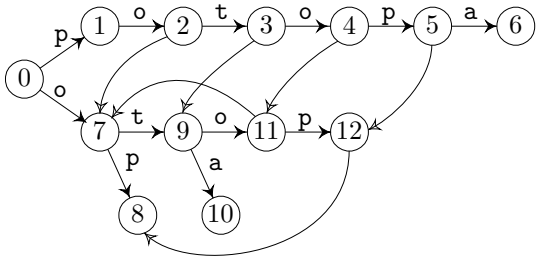
Příklad: Pro slovo `instinkt` by vyhledávací automat vypadal takto (zpětnou funkci jsme kreslili pouze tam, kde nevede do stavu 0):



Nyní algoritmus KMP rozšíříme, aby uměl hledat více slov. Mějme slovník K, což je konečná množina slov nad abecedou Σ, a prohledávaný text β. Vytvoříme vyhledávací automat, jehož výstupem bude výpis nalezených slov a jejich pozic v textu. Jeho stavy budou odpovídat prefixům všech slov ze slovníku a očíslujeme si je přirozenými čísly, počáteční stav $q_0 = 0$ bude odpovídat prázdnému prefixu. Výstupní funkce *out* pro prefix α ohlásí všechna slova ze slovníku, která jsou suffixem slova α.

Příklad: Jak takový vyhledávací automat může vypadat, si ukážeme pro latinskou abecedu a slovník

$$K = \{\texttt{potopa}, \texttt{op}, \texttt{ota}, \texttt{otop}\}.$$



Rovnými čarami je zobrazena přechodová funkce, kroucenými zpětná funkce. Nejsou zakresleny šipky do 0 u přechodové ani u zpětné funkce. Výstupní funkce je dána následující tabulkou:

$$\begin{array}{ll} out(5) = \{\texttt{otop}, \texttt{op}\} & out(10) = \{\texttt{ota}\} \\ out(6) = \{\texttt{potopa}\} & out(12) = \{\texttt{otop}, \texttt{op}\} \\ out(8) = \{\texttt{op}\} & out(\text{ostatní}) = \emptyset. \end{array}$$

Vyhledávání pomocí tohoto automatu bude probíhat stejně jako u KMP, $r[i]$ opět bude nejdelší stav, na který končí právě přečtená část textu, složitost vyhledávání bude opět $\mathcal{O}(n)$ až na vypisování výskytů, které poběží v čase $\mathcal{O}(\textit{počet výskytů})$, což může být více než lineárně, ale lépe to určitě nejde.

Pro pořádek dokážeme, že automat doopravdy vyhledává všechny výskyty: (i) Každé slovo, které oznámíme jako nalezené, se v textu opravdu vyskytuje ($r[i]$ se v textu vyskytuje podle své definice a všechna oznámená slova jsou suffixy $r[i]$). (ii) Všechny výskyty opravdu oznámíme. Pokud se na pozici i vyskytuje slovo $\alpha \in K$, pak je zajisté α jedním ze stavů, na něž $\beta[1 \ldots K]$ končí a $r[i]$ musí být buďto tento stav nebo nějaký ještě delší, jehož je α suffixem.

Teď se podíváme na to, jak vyhledávací automat pro daný slovník sestrojit. Provedeme to ve dvou krocích. Nejprve sestrojíme množinu stavů Q, přechodovou funkci g a částečnou výstupní funkci o. Ve druhém kroku vytvoříme zpětnou funkci f a rozšíříme o na výstupní funkci *out*.

V prvním kroku založíme počáteční stav 0, postupně projdeme celý slovník K a každé slovo σ ze slovníku do automatu přidáme. To provedeme tak, že začneme ve stavu 0 a pustíme automat na σ. Jakmile ale v některém stavu s pro znak $\sigma[i]$ nebude přechodová funkce definována, přidáme nový stav q, nastavíme přechodovou funkci $g(s, \sigma[i]) = q$, přejdeme do stavu q a pokračujeme. Tím v lineárním čase vytvoříme strom stavů. Pokaždé, když dojdeme na konec slova, nastavíme také částečnou výstupní funkci $o(q)$ na $\{\sigma\}$.

Popíšeme tuto část formálně:

1. Začni s množinou stavů $Q \leftarrow \{0\}$.
2. Pro každé slovo σ ze slovníku K proveď kroky 3–7:
3. Nastav aktuální stav s na 0.
4. Pro každé písmeno $\sigma[i]$ slova σ proveď 5–6:
5. Pokud je $g(s, \sigma[i])$ nedefinované, založ nový stav q, nastav $Q \leftarrow Q \cup \{q\}$ a polož $g(s, \sigma[i]) \leftarrow q$.
6. Přejdi do nového stavu: $s \leftarrow g(s, \sigma[i])$.
7. Nadefinuj částečnou výstupní funkci: $o(s) \leftarrow \{\sigma\}$.

Zpětnou funkci vytvoříme podobně jako pro jedno slovo tak, že pustíme ještě nehotový automat na část vyhledávaného slova. Opět chceme využít toho, že je funkce definovaná pro všechna kratší slova. Vezměme si náš příklad. Při přidávání slova `potopa` bychom nastavili $f(1) = 0$, $f(2) = 7$, $f(3) = 9$, ale u druhého `o` bychom chtěli použít zpětnou funkci $f(9)$, která ještě není definovaná. Proto budeme postupovat pro všechna slova ze slovníku současně v pořadí podle rostoucí vzdálenosti od stavu 0.

Ještě vyřešíme výstupní funkci. Označme $\sigma(s)$ slovo, jehož cesta vede do stavu s. Pokud pro stav s platí $f(s) = 0$, znamená to, že neexistuje žádný nevlastní (neprázdný) suffix, který by byl prefixem některého ze slov ve slovníku. Proto v tomto stavu může skončit pouze slovo $\sigma(s)$. Nastavíme $out(s) = o(s)$. Pokud $f(s) \neq 0$ končí v tomto stavu také všechna slova, které jsou suffixem slova $\sigma(s)$. Tehdy je $out(s) = o(s) \cup out(f(s))$.

Opět formálně:

1. Založ frontu F, zatím prázdnou.
2. Nastav $f(0) \leftarrow 0$ a $out(0) \leftarrow \emptyset$.
3. Pro každý znak $c \in \Sigma$ proveď následující krok:
4. Pokud je stav $s \leftarrow g(0, c) \neq 0$ pak nastav $f(s) \leftarrow 0$, $out(s) \leftarrow o(s)$ a zařaď s na konec fronty F.
5. Dokud je nějaký stav ve frontě, prováděj následující:
6. Odeber první stav r z fronty F.
7. Pro každý znak $c \in \Sigma^*$, pokud je $g(r, c) \neq 0$, proveď:
8. Označ $s \leftarrow f(r)$. Dokud $g(s, c) = 0$, zvol $s \leftarrow f(s)$.
9. Nastav $f(s) \leftarrow g(s, c)$.
10. Nastav $out(s) \leftarrow o(s) \cup out(f(s))$.
11. Zařaď s na konec fronty F.

Aby algoritmus fungoval rychle, musíme zvolit šikovnou reprezentaci výstupní funkce. Kdyby si každý stav pamatoval

svou vlastní množinu, mohly by tyto množiny dohromady být víc než lineárně velké (zkuste vymyslet příklad slovníku, pro který tomu tak je) a museli bychom se vzdát naděje, že stihneme automat zkonstruovat v lineárním čase. Proto použijeme trik: všimneme si, že $out(s)$ je pro každý stav buďto rovna $out(f(s))$ nebo se od ní liší přidáním slova $o(s)$. Stačí si proto pamatovat $o(s)$ a ještě nějakou funkci $z(s)$, která řekne, ve kterém stavu máme najít zbytek množiny $out(s)$. Krok 9 proto upravíme takto:

9. Pokud je $o(f(s)) = \emptyset$, polož $z(s) \leftarrow z(f(s))$, jinak $z(s) \leftarrow f(s)$.

Podobně upravíme vypisování nalezených slov: vypíšeme $o(s)$ a pokud je $z(s) \neq 0$, pokračujeme ve vypisování ve stavu $z(s)$.

Ještě se zamysleme nad časovou složitostí. Označme P velikost celého slovníku. První část algoritmu provede maximálně $\mathcal{O}(P)$ kroků, pokud považujeme velikost abecedy za konstantu. Ve druhé fázi se každý stav dostane do fronty právě jednou, takže vše je lineární až na průchody zpětnou funkcí. Můžeme si ale všimnout, že podobně jako u KMP i zde vlastně spouštíme vyhledávací automat na všechna hledaná slova bez prvního písmene, až na to, že místo jedno po druhém je zpracováváme na přeskáčku a že společné části výpočtů (než se strom rozvětví) počítáme jen jednou. Celkem to tedy bude trvat nejvýše tolik, kolik vyhledání všech slov dohromady, což je $\mathcal{O}(P)$.

Celkové tedy vyhledávací algoritmus běží v čase $O(P+n+v)$, kde n je délka textu, P celková velikost slovníku a v počet nalezených výskytů. Na závěr dodejme, že tento algoritmus vymysleli pan Aho a paní Corasicková, a předveďme program:

```
var
  N: Integer;           { délka textu }
  W: Integer;           { počet slov}
  Slova: array[1..MaxW, 1..MaxK] of Char;
  Delky: array[1..MaxW] of Integer;
  Text: array[1..MaxN] of Char;
  Q: Integer;           { počet stavů }
  { přechodová a zpětná funkce: }
  g: array[1..MaxQ, Char] of Integer;
  f: array[0..MaxQ] of Integer;
  { obě části výstupní funkce: }
  o: array[0..MaxQ] of Integer;
  z: array[0..MaxQ] of Integer;

  procedure NulujStav(State: Integer);
  var
    C: Char;
  begin
    o[State]:= 0;
    for C:= #0 to #255 do
      g[State, C]:= 0;
  end;

  function Krok(S: Integer; C: Char): Integer;
  begin
    while (S > 0) and (g[S, C] = 0) do S:= f[S];
    Krok:= g[S, C];
  end;

var
  S, I, J, L: Integer;
  C: Char;
  FI, FC: Integer;
  Fronta: array[1..MaxQ] of Integer;
begin
  { vložíme všechna slova }
  Q:= 0;
  NulujStav(Q);
  for I:= 1 to W do
  begin
    S:= 0;
    for J:= 1 to Delky[I] do
    begin
      if g[S, Slova[I, J]] = 0 then
      begin
        Q:= Q + 1;
        NulujStav(Q);
        g[S, Slova[I, J]]:= Q;
      end;
      S:= g[S, Slova[I, J]];
    end;
    o[S]:= I;
  end;

  { zkonstruujeme zpětnou a výstupní fci }
  f[0]:= 0; z[0]:= 0;
  FC:= 1; FI:= 1;
  Fronta[FC]:= 0;
  while FI <= FC do
  begin
    for C:= #0 to #255 do
    if g[Fronta[FI], C] <> 0 then
    begin
      S:= g[Fronta[FI], C];
      I:= Krok(f[Fronta[FI]], C);
      if Fronta[FI] = 0 then f[S]:= 0
                        else f[S]:= I;
      if o[f[S]] <> 0 then z[S]:= f[S]
                      else z[S]:= z[f[S]];
      FC:= FC + 1;
      Fronta[FC]:= S;
    end;
    FI:= FI + 1;
  end;

  { hledáme }
  S:= 0;
  for I:= 1 to N do
  begin
    S:= Krok(S, Text[I]);
    L:= S;
    while L <> 0 do
    begin { hlásíme výskyty }
      if o[L] <> 0 then
      begin
        write(I, ': ');
        for J:= 1 to Delky[o[L]] do
                    write(Slova[o[L], J]);
        writeln;
      end;
      L:= z[L];
    end;
  end;
end.
```

Dnešní menu Vám servírovali
Martin Mareš a Petr Škoda

22-3-1 Falešná mince

Nad úlohou budeme uvažovat trochu pozpátku. Co nám asi může říct lékárník? Z každého vážení mohou přijít tři různé výstupy. Buďto se váhy naklonily vlevo, nebo vpravo, nebo zůstaly v rovnováze. Tedy máme celkem 3^K možných odpovědí pro K vážení. Kdybychom věděli, že mince je BÚNO (bez újmy na obecnosti) lehčí, pak máme naprosto triviální situaci. Podíváme se, která mince byla na všech miskách, které byly prohlášeny za „lehčí“, a nebyla na žádných jiných. Umíme zvážit 3^K mincí a finito.

Nicméně my víme jen to, že mince je různé hmotnosti, a tedy musíme uvážit obě možnosti. Nalezneme minci, která byla buďto na všech lehčích, nebo na všech těžších miskách a při rovnovážných váženích ležela stranou. Tedy sestavíme takovou sadu vážení, aby se toto dalo jednoznačně určit. Uvědomme si však, jak se liší výsledky pro stejnou minci falešnou, ovšem jednou lehčí a jednou těžší – výsledek každého vážení se prostě obrátí (a vznikne *inverzní výsledek*) s výjimkou případu, kdy byly váhy vždy v rovnováze, ten je inverzní sám sobě – a tedy můžeme určit maximální počet mincí, které umíme zvážit K váženími, na $(3^K + 1)/2$.

Každé minci tedy můžeme přiřadit jeden řetězec znaků `<=>`, znamenajících „levá strana je lehčí“, „váhy jsou v rovnováze“, „pravá strana je lehčí“. Z každého můžeme získat jemu inverzní vzájemným nahrazením znaků `<>`.

Takový řetězec také říká, jak ve kterém vážení mince figuruje. Jsou-li váhy v rovnováze, zjevně na nich falešná mince zrovna není, jinak se nachází na jedné z misek. Uvažme tedy i-té vážení: Existuje právě $2 \cdot 3^{K-1}$ různých řetězců délky K majících na i-tém místě `<` nebo `>`. Z nich ale právě polovinu vyškrtáme – jsou v nich totiž samé dvojice *duálních řetězců*. Takže na dvě misky v i-tém vážení potřebujeme rozmístit 3^{K-1} mincí, to ale není možné (neumíme nedestruktivně rozdělit lichý počet mincí na poloviny) – jeden řetězec zjevně nevyužijeme, a tedy neumíme zvážit $(3^K + 1)/2$ mincí, ale jen $(3^K - 1)/2$ mincí.

Nyní přichází nejtěžší úkol – jak zkonstruovat rozložení mincí na váhy. Pro $N = 1$ si můžeme být jisti, že ta mince, kterou držíme v ruce, je falešná. Pro $N = 2$ to naopak určit vůbec nelze. Pro ostatní N vytvoříme N řetězců délky K ze znaků `<=>` tak, že pro každou pozici $i \in \{1 \dots K\}$ bude platit, že existuje stejný počet řetězců majících na i-té pozici `<` jako počet řetězců majících na i-té pozici `>` (dále označuji jako *Podmínka*). Pak jednoduše rozložím mince na váhy při jednotlivých váženích tak, že při `<` položím minci na levou misku, při `>` na pravou a při `=` odložím stranou.

Příklad: vážení pro $N = 4$ se dá zapsat jako `1--2, 2--3`, ale také jako $M_N = \{$`<=, ><, =>, ==`$\}$, což pak přečteme jako předpis: „První minci polož při prvním vážení na levou misku a při druhém ji odlož stranou; druhou minci polož při prvním vážení na pravou misku a při druhém na levou; třetí minci nejdříve odlož stranou a při druhém vážení polož na pravou misku a čtvrtá mince nechť se vah ani nedotkne.“

Zkonstruujme nejprve rozložení pro $N = N_K = (3^K - 1)/2$ a z nich potom všechna ostatní rozložení. To uděláme rekurentně – konstrukcí z předchozího. Pro $K = 2$, $N_2 = 4$ máme předchozí příklad. Všimněme si, že v něm není řetězec `<<`. Lze jednoduše ukázat, že $N_{K+1} = 3N_K + 1$. Vezměme tedy množinu M_{N_K}, odstraníme z ní řetězec `==...=` a do množiny $M_{N_{K+1}}$ ji vložíme třikrát – jednou ke všem řetězcům přidáme na začátek `<`, jednou `>` a jednou `=`. Chybí nám ještě 4 řetězce: trojice `=>>>...>`, `><<<...<`, `<===...=` a „odložená mince“ `===...==`. Je jednoduše ověřitelné, že množina $M_{N_{K+1}}$ splňuje *Podmínku* a zároveň neobsahuje řetězec `<<<<...<<`.

Nakonec si ještě rozmyslete, že z každé takto zkonstruované množiny lze vyškrtnout správný počet řetězců, abychom získali M_N, pro všechna N s výjimkou $N = 2$. Jde totiž rozdělit M_{N_K} na trojice splňující *Podmínku*: Označíme-li v $M_{N_{K-1}}$ nějakou existující trojici řetězců jako `A, B, C`, tak v M_{N_K} máme následující trojice: (`<A, =B, >C`), (`=A, >B, <C`), (`>A, <B, =C`). Zbytek je jedna trojice a „odložená mince“.

Tedy pokud potřebuju M_N pro $N = N_K - 3\varphi$, tak odstraním φ trojic, pro $N = N_K - (3\varphi + 1)$ odstraním φ trojic a „odloženou minci“. Zbývá podlý trik podle Mirka Olšáka (díky!) pro $N = N_K - (3\varphi + 2)$: Z řetězců `=<<<<...<<=`, `<>>>...>>=`, `=<<<...<<<` udělám řetězec `<<<<...<<`, který v M_{N_K} nebyl, čímž jsem se zbavil dvou řetězců, a následně ještě odstraním φ dalších trojic.

A tedy počet vážení, které potřebujeme k určení falešné mince z množiny N mincí, je roven $K = \lceil \log_3(2N) \rceil$.

Jan „Moskyt“ Matějka

22-3-2 Dětská hra

Jednalo se o grafovou úlohu. Děti představují vrcholy a pokud má dítě u chytat dítě v, pak existuje hrana (u, v). Dítě d se může dostat do pozice, kdy by muselo chytat samo sebe, pouze v případě, že vrchol d leží na kružnici. Z každého vrcholu vede právě jedna hrana, takže počet vrcholů (n) je roven počtu hran (m). Díky tomu víme, že v každé komponentě je právě jeden cyklus. Pokud $m = n - 1$, tak je graf stromem (každý vrchol kromě kořene je zavěšen jednou hranou ke svému předkovi). Pokud přidáme n-tou hranu, tak nám vznikne kružnice.

Kružnice budeme hledat prohledáváním do hloubky. V každé fázi si vybereme vrchol x, který jsme ještě neprošli, a spouštíme prohledávání z něj. U každého vrcholu si značíme „čas“ prvního příchodu (např. číslo fáze) $in(vrchol)$. Postupujeme po hranách, dokud nenarazíme na hranu (u, v), která vede do prozkoumaného vrcholu v. Pokud je čas příchodu do v menší než čas příchodu do x, tak jsme se jen dostali k již prozkoumané části grafu, v opačném případě jsme nalezli kružnici délky $in(u) + 1 - in(v)$. Poznačíme si délku nalezeného cyklu a pokračujeme další fází, dokud existují neprozkoumané vrcholy. Nakonec jsou všechny vrcholy navštívené a výsledný počet dětí, které mohou chytat samy sebe, je součtem nalezených cyklů.

Při procházení navštívíme každý vrchol právě jednou a ještě jedenkrát se do něj můžeme podívat, pokud leží na cyklu, anebo na cestě, kterou jsme začali procházet až od něj. Časová složitost tedy je $\mathcal{O}(n)$, paměťová složitost je taktéž $\mathcal{O}(n)$ – pro každý vrchol si pamatujeme jeho následníka.

David Marek

22-3-3 Kurýrní služba

Na první pohled je vidět, že úloha je grafová, města tvoří vrcholy a kurýři orientované grafy (komu nic tyto pojmy

neříkají, doporučuji přečíst kuchařku o grafech na našich stránkách). Počet měst (vrcholů) si označíme N a počet kurýrů (hran) M. O husitském orientovaném grafu chceme zjistit, zda-li existuje cesta mezi každými dvěma vrcholy alespoň jedním směrem, tedy z A do B nebo z B do A pro každé dva vrcholy A a B. Takový graf se nazývá polosouvislý.

To bude určitě Floyd-Warshallův algoritmus, zazní v hlavě první návrh. Ten přece počítá nejkratší cestu v orientovaném grafu mezi všemi vrcholy. Používá dvourozměrné pole velikosti $N \times N$, přičemž prvek na pozici $[i, j]$ obsahuje nejkratší cestu mezi vrcholy i a j. Na začátku jsou všechny prvky inicializovány „nekonečnem“ nebo ohodnocením hrany a v N krocích se vylepšuje odhad na délku nejkratší cesty. Pro podrobnější popis odkazuji opět na naše kuchařky na webu, konkrétně na dynamické programování. Jen dodám, že tento algoritmus má časovou složitost $\mathcal{O}(N^3)$ a paměťovou $\mathcal{O}(N^2)$.

S časem $\mathcal{O}(N^3)$ by však mohli husiti prohrát válku, než by zjistili, že se mezi městy nedají posílat zprávy – mají pomalé dřevěné počítače a spoustu dobytých měst. Navíc nepotřebujeme nutně počítat nejkratší cestu, ani neznáme ohodnocení hran (vzdálenost mezi městy). A zatřetí, dávali bychom pouze za použití algoritmu z kuchařky 13 bodů? Zkusíme tedy postupovat lépe.

Nejprve si dokážeme, že v polosouvislém grafu musí existovat sled procházející všemi vrcholy (sled je cesta, ve které se mohou opakovat vrcholy i hrany). Kdyby totiž neobsahoval všechny vrcholy, vezmeme vrchol V, jenž v něm neleží. Aby byl graf polosouvislý, musí pro každý vrchol U ze sledu existovat buď cesta z U do V nebo z V do U. Podle toho, jestli z vrcholu existuje cesta z nebo do V, si rozdělíme vrcholy na dvě skupiny (tyto skupiny mohou mít nějaký překryv) a seřadíme je podle toho, jak leží na sledu. Potom ovšem můžeme V přidat do sledu mezi poslední vrchol, z něhož se lze dostat do V, a první vrchol, do něhož vede cesta z V, anebo kamkoliv do překryvu těchto dvou skupin.

Aby se nám sled lépe hledal, odstraníme si z grafu orientované cykly, respektive sloučíme je do jednoho vrcholu. Jinými slovy, najdeme silně souvislé komponenty (SSK), což jsou maximální podgrafy, ve kterých mezi každými dvěma vrcholy vede orientovaná cesta oběma směry. Pro představu, SSK mohou tvořit jednotlivé cykly, soustavy více spojených cyklů, ale i samotné vrcholy. Jak je vidět, v nich skutečně nemusíme zjišťovat, jestli existuje cesta mezi každými dvěma vrcholy. Netřeba vymýšlet kolo, na nalezení silně souvislých komponent se používá Kusarajův či Tarjanův algoritmus. My si zde popíšeme Tarjanův, jenž najdete i na anglické Wikipedii pod heslem *Tarjan's strongly connected components algorithm*.

Tarjanův algoritmus je modifikací prohledávání do hloubky. Oproti standardnímu prohledávání si navíc budeme vrcholy číslovat podle toho, kdy do nich vstoupíme (seřadíme je prohledáváním do hloubky), a při zpětném průchodu hledáme, do jakého vrcholu s co nejnižším číslem se můžeme dostat. Je-li to nejnižší nalezené číslo rovno číslu vrcholu, našli jsme cyklus a tedy SSK. Všimněte si, že každou SSK zaznamenáme právě jednou, protože všechny její vrcholy obdrží stejné číslo – to nejnižší z celé SSK – a rovnost tedy nastane jen u jednoho vrcholu. Tento vrchol si nazveme kořenem SSK.

Jak zjistit, jaký potomek vrcholu má nejnižší číslo? Jednoduše, stačí se podívat, do jakého nejnižšího čísla se dostali potomci vrcholu, do nichž z něj vede hrana. Pokud narazíme na cyklus a tedy na vrchol, u něhož dosud tuto hodnotu neznáme, vezmeme prostě jeho číslo (což se dá zařídit lehce – na začátku bude mít každý vrchol inicializován nejnižšího potomka svým číslem).

Díky nalezení SSK v kořeni (vrcholu s nejnižším číslem v SSK) máme jistotu, že jsme získali maximální SSK, tedy že k ní už nelze žádný vrchol přidat, aby zůstala silně souvislá. Pro určení, které vrcholy náleží do nalezené SSK, použijeme zásobník, do nehož přidáváme vrcholy při vstupu do nich. Po objevení SSK se vrcholy ze zásobníku odebírají, dokud se nenarazí na kořen.

Máme tedy silně souvislé komponenty. Co s nimi? Sloučíme je do jednoho vrcholu a vytvoříme si nový, acyklický graf, tzv. kondenzaci původního grafu. V tomto grafu již stačí otestovat, jestli v něm existuje cesta obsahující všechny vrcholy (důvod je podobný tomu, že v původním grafu je sled se všemi vrcholy). Nový graf se ale nemusí skládat jen z cesty, může obsahovat tzv. dopředné hrany, jež vedou z nějakého vrcholu A do jiného vrcholu, jež je na cestě dále než A. Test, který se pokusí takovou cestu najít a rozhodne o výsledku, může být například nalezení vrcholu, do nějž nevede hrana (ten existuje, máme acyklický graf a musí být právě jeden), a průchod do šířky, při kterém si budeme pamatovat, přes kolik vrcholů jsme již přešli.

Jaká je časová a paměťová složitost algoritmu? Jak již bylo řečeno, Tarjanův algoritmus je modifikované prohledávání do hloubky. Má-li tedy operace zjištění, jestli je prvek v zásobníku, konstantní časovou složitost (což se dá zařídit polem, jehož prvek na místě i určuje, jestli je i-té město v zásobníku), složitost Tarjanova algoritmu vyjde $\mathcal{O}(N+M)$. Na následné nalezení vrcholu, do kterého nevede hrana, spotřebujeme opět řádově $N + M$ operací. Složitost prohledávání do šířky nám už hezkou lineární časovou složitost $\mathcal{O}(N + M)$ nezkazí. Asymptoticky rychleji úlohu určitě nevyřešíme, nepřečetli bychom ani celý vstup. Algoritmus zabere také jen $\mathcal{O}(N + M)$ paměti, takže jsme zachránili husity s pomalými dřevěnými počítači.

Jiné řešení

Celkem elegantní řešení poslal Mirek Olšák. Všiml si, že stačí modifikovat Tarjanův algoritmus, takže si vystačíme pouze s jedním prohledáváním do hloubky. Použijeme následující pozorování: když se vracíme při prohledávání acyklického grafu, jež nám vyrobil Tarjanův algoritmus, z nějakého vrcholu, musí z něj vést hrana do vrcholu, z něhož jsme se vraceli předtím (pro lepší představu si zkuste na chvíli vypustit z grafu cykly). Při návratu z vrcholu tedy uložíme vrchol do nějaké globální proměnné a v každém vrcholu zkontrolujeme, jestli do něj vede hrana. Že graf není polosouvislý, objeví tento algoritmus ve vrcholech, z nichž nevede hrana do žádného dosud neprošlého vrcholu, zkuste si rozmyslet proč.

Nejlépe půjde algoritmus pochopit asi z následujícího pseudokódu vycházejícího z implementace Tarjanova algoritmu na Wikipedii:

```
vstup: graf G = (V, E)
// V je množina vrcholů a E množina hran
index = 0
// proměnná sloužící k číslování vrcholů
S = empty // prázdný zásobník pro vrcholy
posledni = undefined
// poslední vrchol, z něhož jsme se vrátili
forall v in V do
  if (v.index == undefined)
```

```
    // prohledej do hloubky vrcholy,
    // které ještě nebyly navštíveny
    tarjan(v)
print "Hurá! Graf je polosouvislý."

procedure tarjan(v)
  v.index = index  // očíslování vrcholu
  // inicializace nejnižšího dosaženého vrcholu
  v.lowlink = index
  index = index + 1
  nalezen_posledni = false
  if (posledni == undefined)
    //zatím jsme se ze žádného vrcholu nevraceli
    nalezen_posledni = true
  S.push(v)  // přidej vrchol do zásobníku
  // projdi všechny následníky vrcholu
  forall (v, v') in E do
    // pokud není vrchol navštíven
    if (v'.index == undefined)
        tarjan(v') // prohledej ho do hloubky
        // urči vrchol s nejnižším číslem
        v.lowlink = min(v.lowlink, v'.lowlink)
    else if (v' is in S)
        // vrchol je v zásobníku
        v.lowlink = min(v.lowlink, v'.index)
    // vede hrana do vrcholu, z něhož
    // jsme se naposled vraceli?
    if (posledni == v')
       nalezen_posledni = true
  // konec for cyklu
  if (nalezen_posledni == false)
    print "Graf není polosouvislý."
    exit
  if (v.lowlink == v.index) // nalezena SSK
    // odeber všechny vrcholy SSK ze zásobníku
    repeat
      v' = S.pop
      // nejhlouběji v zásobníku je kořen SSK
    until (v' == v)
  posledni = v
```

Pavel „Paulie“ Veselý

22-3-4 Rukavice

Trošku nás mrzelo, jak málo jste se snažili dokazovat správnost svých řešení. Není vůbec těžké na nějakou strategii přijít, ale vrací taková opravdu požadované, tj. vzhledem k L a P minimální výsledky? K nočním můrám opravovatelů KSP patří divná, složitá a odvážná řešení, která skoro určitě nefungují, ale je třeba přijít na potřebný protipříklad – v případě této úlohy byla ale většina špatných řešení vyvratitelná krátkým kritickým náhledem, kterého byste měli být schopni i sami. Nebojte se nám napsat, že slabinu svého postupu znáte: chápeme, že na dobré řešení není snadné přijít.

Jednoduchý, nesprávný, ale slibný pohled vypadá takto: pokud bych měl jistotu, že z levé truhly vytáhnu od každého páru alespoň jednu rukavici, mohlo by mi stačit z pravé bedny vytáhnout libovolnou. Takovou jistotu však nemohu získat nikdy, protože mi nic nezaručuje, že neexistují pravé rukavice bez levého ekvivalentu (bezlevé): proto budu chtít zajistit, abych z levé bedny vytáhnul alespoň jednu rukavici od každé zastoupené barvy a z pravé vytáhnu tolik rukavic, abych měl jistotu, že mezi nimi bude nějaká, která má v levé bedně souputníka.

Kolik tedy? Inu, pokud necháme Tomáše z pravé bedny vytáhnout tolik rukavic, kolik je tam bezlevých a ještě jednu navíc, nemůže se ani v tom nejhorším případě stát, že by Tomášův výběr obsahoval samé bezlevé pravé rukavice. Podobně pokud v levé bedně vybereme tolik rukavic, kolik jich je tam celkem bez počtu nejméně zastoupeného druhu plus jednu navíc, určitě se nám nemůže stát, že by se v takovém výběru nevyskytovala libovolná varianta. (Rozmyslete si! Proč to musí být „nejméně zastoupeného druhu“?)

Nyní nás může napadnout, že při nahrazení levé bedny za pravou a naopak by nám tento postup mohl vrátit lepší výsledek, kupříkladu kdyby napravo bylo velmi mnoho bezlevých rukavic. Je propočítání obou variant a navrácení té lepší správné řešení?

Ne.

(Přestaňme nyní uvažovat bezlevé a bezpravé rukavice – obecně jejich výskyty stejně nemůžeme řešit jinak, než že jejich počet přičteme k počtu rukavic k vytáhnutí.)

Rozhodli jsme se z jedné bedny vytáhnout zaručeně všechny barvy a z druhé zaručeně jednu, z čehož jsme si logicky odvodili, že získáme jednobarevnou dvojici. Co kdybychom chtěli z levé bedny zaručeně vytáhnout k různých barev a z pravé $n-k+1$? Dirichletův princip praví, že i pak bychom měli zaručeno, že vytáhneme alespoň jeden stejný pár. Může se pro nějaké k stát, že dosáhneme lepšího výsledku než v krajních případech $k = 0$, $k = n$? (Uvědomte si, že to jsou přesně dvě možnosti zvažované v odstavci před „Ne.“)

Nejdříve: jak zajistíme vytáhnutí k barev? Stačí vytáhnout počet všech rukavic bez $n - k + 1$ nejméně častých plus jednu navíc, podobně jako jsme to dělali výše. Teď si je třeba rozmyslet, jaký rozdíl v počtu tažených rukavic způsobí přechod od nějakého k ke $k+1$: z levé bedny budeme muset vytáhnout navíc rukavice $(n-k+1)$-ní nejméně zastoupené barvy, z pravé naopak nebudeme muset táhnout rukavice k-té nejméně zastoupené barvy. V obecném případě nevíme nic o tom, jestli si tím polepšíme nebo pohoršíme, musíme tedy získat minimum přes všechna k.

No a triviální postup, jak ho získat, je seřadit si počty rukavic jednotlivých barev a pak k projít cyklem. Trvat to bude $\mathcal{O}(n \log n)$, místa zabereme $\mathcal{O}(n)$. Detaily najdete v autorském zdrojáku.

Je to tak správně? Jde si snadno představit, že nastavili-li bychom pro určité množství L rukavic P menší, než jak to děláme, tj. takové, pro které nám Dirichlet už nezaručuje, že vytáhneme dvě opačné rukavice stejné barvy, šel by sestavit výběr rukavic, který skutečně takovou dvojici neobsahuje. Teď je otázka, zda neexistuje L, které jsme nezkoušeli a které dává lepší řešení: počet všech levých rukavic je ostře větší než n, takže jsme těch propočtů přešli docela hodně.

A vtip je v tom, že pro L, které $L_k < L < L_{k+1}$ (kde L_k odpovídá počtu levých rukavic, které musíme vytáhnout pro dané k), bude $P = P_k$, protože jsme oněch $L - L_k$ rukavic vytáhli nadarmo: nezaručili jsme jimi vytažení většího množství barev.

Mária Vámošová & Lukáš Lánský

22-3-5 Reklama

Na začiatku riešenia si dopomôžeme menším trikom. Celú situáciu bodov v rovine otočíme o 45 stupňov proti smeru hodinových ručičiek. To spravíme jednoducho tak, že nahradíme $x' = x + y$ a $y' = x - y$, a ďalej budeme pracovať už len s týmito transformovanými súradnicami. Teraz platí, že všetky body (x_1, y_1), ktoré môžu byť spojené s nejakým bodom na súradniciach (x_2, y_2), musia mať $x_1 \leq x_2$ a $y_1 \leq y_2$ alebo $x_1 \geq x_2$ a $y_1 \geq y_2$.

Najskôr sa zamyslime, čo to znamená, nakresliť čo najmenej čiar, aby obsahovali všetky body. Ak si predstavíme čiaru

ako postupnosť úsekov spájajúcich dva body, tak to znamená, že minimum sa dosiahne práve vtedy, keď je celkový počet úsekov všetkých čiar dohromady maximálny.

Jeden úsek čiary je teda spojenie dvoch rôznych bodov $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ také, že $x_1 \leq x_2$ a $y_1 \leq y_2$. Takéto usporiadané dvojice (p_1, p_2) bodov budeme ďalej volať pár a množinu párov, v ktorých každý bod vystupuje maximálne 2 krát (raz ako p_1 a raz ako p_2), budeme volať párovanie, ktorého veľkosť sa snažíme maximalizovať.

Na ďalšie uvažovanie si úlohu trocha preformulujme: Sú dané dve množiny bodov S a P, pričom S obsahuje kópie bodov, ktoré môžu vystupovať v párovaní ako body p_1 a P obsahuje kópie bodov, ktoré môžu vystupovať v párovaní ako body p_2.

Pochopiteľne, naša pôvodná úloha je špeciálnym typom tejto preformulovanej úlohy, keď $S = P =$ { pôvodné body }.

Ak si množiny S a P predstavíme ako partity bipartitného grafu, kde každý korektný pár reprezentujeme hranou medzi príslušnými dvoma vrcholmi v S a P, potom veľkosť najväčšieho párovania vieme nájsť pomocou obecného algoritmu na hľadanie maximálneho bipartitného párovania. O tomto algoritme si môžete prečítať viac na anglickej Wikipedii[1] alebo na stránkach Martina Mareše[2] a za jeho použitie ste mohli získať maximálne 10 bodov.

K lepšiemu algoritmu bolo nutné učiniť dôležité pozorovanie: označme bod $p = (x_0, y_0) \in P$ ako bod množiny P s najväčšou y-ovou súradnicou (a spomedzi tých s najmenšou x-ovou súradnicou). A skúmajme, s ktorými bodmi v S môže tvoriť pár v nejakom optimálnom (maximálnom) párovaní.

Pre všetky body, s ktorými môže tvoriť pár, platí $x \leq x_0$ a $y \leq y_0$. Keďže p je bod s maximálnou y-ovou súradnicou, dostávame, že to môže byť ľubovoľný bod, spĺňajúci podmienku $x \leq x_0$. Ak žiaden takýto bod neexistuje, potom je samozrejmé, že v žiadnom optimálnom riešení nie je tento bod spárovaný. Inak označme bodom $q = (x_1, x_2) \in S$ bod s maximálnou y-ovou súradnicou spĺňajúci $x_1 \leq x_0$ a ak je takých viac, tak spomedzi tých bod s najväčšou x-ovou súradnicou.

Ukážeme, že existuje optimálne riešenie, keď je bod p spárovaný s bodom q.

Nech existuje ľubovoľné optimálne riešenie, v ktorom to neplatí. Potom môžu v tomto párovaní nastať len tieto situácie:

- Bod p je spárovaný s nejakým iným bodom $q' \neq q \in S$ a bod q je bez páru alebo q je spárovaný s nejakým iným bodom $(p' \in P) \neq p$ a bod p je bez páru. To však znamená, že môžeme spárovať p s q a q' (resp. p') nechať bez páru, čím párovanie nezmenšíme.
- Bod p je spárovaný s nejakým iným bodom $q' \neq q$ a bod q je spárovaný s nejakým iným bodom $p' \neq p$.
  Keďže však platí, že bod q je bod s najväčšou y-ovou súradnicou (a prípadne najväčšou x-ovou), pre ktorý platí $x_1 \leq x_0$, potom pre bod $p' = (x_2, y_2)$ platí $x_2 > x_0$ a $y_2 \geq y1$. Rovnako pre bod $q' = (x_3, y_3)$ platí $x_3 \leq x_1$ a $y_3 \leq y_1$. Teda môžeme spárovať bod p s q a bod p' s q', čím párovanie nezmenšíme.
- Prípad, keď v optimálnom riešení je bod p aj q bez páru, nemôže nastať, lebo spárovaním vieme dostať párovanie o 1 väčšie.

Vidíme, že nič nepokazíme, keď bod p spárujeme s bodom q. Vykonaním tohto kroku sme si vlastne zredukovali náš problém veľkosti N na problém veľkosti $N-1$. Pretože bod p už nemôže vystupovať v žiadnom párovaní, môžeme ho z množiny P odstrániť a rovnako bod q odstrániť z množiny S. A na nový problém použijeme rovnaký algoritmus.

Jednoduchým aplikovaním uvedeného postupu, keď N-krát opakovane nájdeme bod s maximálnou súradnicou y v čase $\mathcal{O}(N)$ a k nemu príslušný bod q tiež v $\mathcal{O}(N)$, dostávame kvadratický algoritmus, za ktorý ste mohli získať 12 bodov.

Na finálny vzorový algoritmus bolo nutné trocha zmeniť pohľad na úlohu.

Začneme body v množine P prechádzať v poradí rastúcej x-ovej súradnice (a body s rovnakým x podľa y-ovej súradnice). Pričom vždy, keď navštívime bod, určíme, s ktorým bodom v množine S bude spárovaný. Za týmto účelom si budeme pamätať množinu $Q \subseteq S$ doteraz nespárovaných bodov.

Budeme sa pri tom riadiť pravidlom, že aktuálny bod $p = (x_0, y_0) \in P$ spárujeme s bodom $q = (x_1, y_1) \in Q$, ktorý má zo všetkých bodov v Q najväčšiu y-súradnicu, avšak menšiu ako y_0. Ak takýto bod neexistuje, tak bod p zostane bez páru. Následne odstránime z množiny Q bod q a zaradíme tam bod p.

Treba však ukázať, že takýto algoritmus vedie k optimálnemu riešeniu, teda najmenšiemu počtu nutných čiar.

Pri tomto spracovávaní platí invariant, že v každom kroku existuje optimálne párovanie, ktoré páruje spracované body P rovnako, ako sme to spravili my.

Ak takýto invariant platí v kroku n, potom spárovanie bodu p s bodom q nám takýto invariant nepokazí a bude platiť aj v kroku $n+1$. Pretože v tomto optimálnom riešení, ktoré spárovalo prvých n bodov množiny rovnako ako my, môžu nastať len situácie rovnaké ako rozoberané situácie v našom kvadratickom algoritme. Teda spárovaním p s q zachovávame invariant o existencii optimálneho riešenia.

Algoritmus sa dá efektívne implementovať tak, že na začiatku si utriedime body P v čase $\mathcal{O}(N \log N)$ a pri spracúvaní si budeme množinu Q udržiavať v utriedenom poradí podľa súradnice y, napr. pomocou vyvažovaného binárneho stromu, čím dokážeme v čase $\mathcal{O}(\log N)$ hľadať v tejto množine bod, ktorý má najväčšiu súradnicu menšiu ako dané y_0 a v tomto čase tiež vymazať bod q s Q a vložiť tam bod p.

Šikovnejšiu implementáciu dostaneme, ak si všimneme, že bod p sa vkladá na rovnaké miesto v utriedenej množine Q, z ktorého bol vymazaný bod q (ak existoval). Alebo ak neexistoval, tak vloženie sa uskutočňuje vždy na koniec, čím sa nám ponúka jednoduchá možnosť udržiavať si množinu Q v poli a na ňom hladať požadovaný prvok binárnym vyhľadávaním.

Pamäťová zložitosť je $\mathcal{O}(N)$, stačí si nám pamätať len prvky P a množinu Q.

Poučenie na záver: Táto úloha spadá do kategórie „Greedy algoritmy“ alebo tiež „hladové“. V týchto úlohách platí, že v každom kroku máme možnosť vykonať operáciu, ktorá nám zaručene nezabráni nájsť optimálne riešenie, čo

[1] http://en.wikipedia.org/wiki/Maximum_bipartite_matching
[2] http://mj.ucw.cz/vyuka/ga/

implikuje, že ak takú operáciu spravíme v každom kroku, dostaneme optimálne riešenie,

V tomto prípade to bolo operácia spárovania bodu s maximálnou y-ovou súradnicou.

Pri týchto úlohách je však vždy nutné overiť, či náš krok je skutočne tým, ktorý nám našu cestu k optimálnemu riešeniu neodstrihne a či máme možnosť nejakú takúto operáciu spraviť v každom kroku.

Peter Ondrúška

22-3-6 Kolejní výtahy

V této sérii byla leckterá úloha pořádně vypečená, jednoduchost jsme skryli právě do praktické úložky.

Ochotu jednotlivých studentů, kteří chtějí vyjet nahoru výtahem, můžeme chápat jako intervaly na celočíselné ose od 1 do n, kde n je výška kolejí. Některé intervaly se mohou překrývat, a našim cílem je vybrat takovou množinu čísel, že pokryjeme všechny intervaly. Počet intervalů označme k.

Pojďme na to od lesa. Tedy ... od začátku. První student (tedy ten, který je ochoten vystoupit v nejnižším patře) bude muset někde vystoupit, ale může zastavit tak, aby pomohl více lidem. Moc nového jsme se nedozvěděli. Co když se ale podíváme na prvního studenta, který *musí vystoupit* – jehož konec intervalu je ze všech konců nejblíže 1. Víme, že tohoto studenta musíme někde z výtahu vystrčit, ale navíc víme, že to klidně můžeme učinit právě na tomto místě.

Můžeme to udělat proto, neboť jsme ho chytře vybrali – může se stát, že jeho interval protíná intervaly jiných studentů, avšak žádný z těch, kdo s jeho intervalem ochoty mají průnik, ještě nemusí vystupovat – jinak řečeno, všichni takoví mohou vystoupit právě v tomto patře.

Necháme tedy z výtahu odejít všechny studenty, kteří jsou na konci intervalu prvního studenta ochotni. A máme základ algoritmu! Zbytek dořešíme obdobně – nalezneme dosud neprobraného studenta, jehož konec intervalu je nejblíže poslednímu místu, a necháme s ním vystoupit všechny ostatní, kteří jsou v tom patře ochotni.

Pokud na začátku algoritmu data setřídíme podle konců intervalů (v čase $\mathcal{O}(k \log k)$), zbytek dořešíme se složitostí $\mathcal{O}(k)$ a celkově to tedy stihneme v čase $\mathcal{O}(k \log k)$.

Máme za sebou složitost, ale je algoritmus správně? Postupem výše zmíněným určitě dojdeme k nějaké korektní posloupnosti zastávek, nemusí nám být ale zcela jasné, že taková posloupnost je minimální. Podívejme se na posloupnost intervalů, které jsme vždy vybrali jako další „koncové“, a označme její velikost p. Protože jsme na každém konci poslali z výtahu všechny studenty, kteří vystoupit mohli, tyto intervaly jsou disjunktní. Tím pádem každé přípustné rozložení zastávek musí zastavit alespoň p-krát – na každém intervalu alespoň jednou. Nícméně p je právě počet zastávek, které vykonal náš algoritmus, a výsledek je tedy vskutku optimální. Howgh.

Tím skončila indiánská část řešení. Ještě zmíníme, že pokud si nakreslíme jednotlivé intervaly na reálnou osu a budeme se na tuto strukturu dívat jako na graf (intervaly = vrcholy a dva vrcholy jsou spojeny hranou, protínají-li se dva intervaly), dostaneme speciální typ grafu, jménem „průnikový graf“. Takovéto grafy (nejen přímkové) jsou často studovány v teorii grafů a již se o nich leccos ví, například že leckteré „těžké“ problémy pro obecné grafy na nich lze vyřešit v polynomiálním čase.

22-3-7 Pavouci internetu

Přehled

V našem vzorovém řešení je situace mírně komplikovaná – dohadují se mezi sebou 3 druhy procesů. Podívejme se tedy napřed na jednodušší situaci, kdy nic nepadá. Jak bude vypadat pracovní cyklus?

Klient si vybere server a zadá mu práci, počká na potvrzení o jejím přijetí. Někdy mezi tím se někde jinde objeví pracující proces a připojí se také na server. Když se tedy na serveru sejdou oba, server je seznámí – předá práci pracujícímu procesu, společně s PID toho, kdo ji zadal. Více se o ně nestará. Pracující provede zadaný úkol, odešle výsledek (přímo zadávajícímu, nikoliv přes server) a znovu se přihlásí na server. Mezitím server seznamuje další pracující s prací. Zatím jednoduché?

Jak je ale řešené, když se pracující přihlásí na jiný server, než na který je uložen úkol? Servery mezi sebou komunikují také. Ve chvíli, kdy na některém serveru dojde k přebytku libovolného druhu (přebývají buď pracující a nebo úkoly), oznámí to server všem ostatním. Ti si buď řeknou o přesun pracujícího nebo nabídnou svého pracujícího. Tomu je poté sděleno, aby se připojil na onen jiný server.

Jak budeme řešit havárie? Podívejme se na jednotlivé části podrobněji.

Zadávání

Když chceme zadat úkol, spustíme nový proces. Jeho úkolem bude sledovat, co se s úkolem kde děje.

Ten se pokusí odeslat práci prvnímu serveru z nastavení. Bohužel, ten nemusí běžet, proto čekáme na odpověď jen nějakou dobu. Pokud ale běží, linkne si náš proces a příjem potvrdí.

Ve chvíli, kdy máme potvrzení o přijetí, čekáme na přiřazení. Až server najde vhodného pracujícího, dá nám o tom vědět (a také jeho PID). My si ho linkneme a budeme čekat na odpověď od něj (a serveru už si nebudeme všímat, co se s ním děje, je nyní nezajímavé).

Když přijde výsledek, vše proběhlo, jak mělo, my můžeme poslat výsledek do původního procesu a spokojeně skončit.

Pokud se nedočkáme odpovědi od serveru, zkusíme další server v seznamu a s ním provádíme totéž. Jestliže selže cokoliv dalšího (server či pracující umře, když čekáme na něco od něj), tak začneme zcela od začátku – od prvního serveru a v novém procesu. (Kdo si má pamatovat, co všechno by bylo potřeba unlinknout? Navíc, pokud by některý server žil, ale nestihl odpovědět včas, pak by měl uloženou naši práci – skončením ji u něj zrušíme.)

Pracující

Každý pracující má dvě vývojová stádia. První stádium je čekající. Podobným způsobem jako klient se pokusíme spojit se serverem (tedy, pošleme mu své PID a on buď odpoví, že nás bere, nebo, když se nedočkáme odpovědi, zkusíme další server). Až nás některý přijme, tak si nás linkne a my jen budeme poklidně čekat, až nám přidělí nějakou práci.

Až nějakou dostaneme, tak si nás unlinkne server, ale linkne si nás klient. My práci pustíme, ale to uděláme v novém procesu (samozřejmě, také s námi slinkovaném) a čekáme na výsledek. Až skončí, pošleme výsledek klientovi, sami se

pokusíme znovu přihlásit k serveru, ale to opět v novém procesu (abychom zničili všechny stará spojení).

Nyní, co se může stát? Když spadne proces s prací, který ale běží na stejném stroji, jako my, znamená to, že je v něm chyba. Nahlásíme to tedy klientovi a práci považujeme za splněnou. Obdobně když se nedočkáme výsledku v požadovaném čase (ale předtím proces s prací ukončíme).

Pokud spadne buď server nebo klient, pak jen ukončíme případnou probíhající práci a zkusíme se opět na některý server připojit. To je zcela v pořádku – pokud spadl server (a my se o tom dozvěděli), pak ještě nemáme žádnou práci a nic se neztratí. Jediný případ, kdy zničíme nějakou práci, je, když umře klient, ale v tom případě není komu poslat výsledek, je tedy zbytečná.

A co když umřeme my? Buď se tak stalo ještě před připojením na server a v tom případě to nezpůsobí žádnou škodu. A poté až do dokončení práce s námi je vždy někdo slinkovaný, takže se o tom dozví a může provést nápravná opatření (v případě serveru si nás jen smazat, v případě klienta zadat práci znovu, někomu živějšímu).

Server

Nyní, co dělá server? Na chvíli si odmysleme poněkud krkolomné předávání pracujících procesů. V tom případě je celý server velmi jednoduchý. Sbírá úkoly a pracující procesy, ve chvíli, kdy se sejde od každého jeden, tak je spáruje a dál se o ně nestará.

Po dobu uložení úkolu i pracujícího procesu je s ním prolinkován. Ve chvíli, kdy druhý konec spadne, smaže si jej ze seznamu (nefungující pracující je k ničemu, práce pro neexistující proces je zbytečná). Když spáruje požadavek s pracujícím, tak je oba unlinkne – pro server jsou již nezajímají, vyřídí si vše mezi sebou.

Co když spadne server? Potom se o tom dozvědí všichni klienti i všichni pracující. Pracující se jen přepojí na jiný server a klienti pošlou práci znovu, někam jinam.

Předávání pracujících

Není problém se dohodnout, že někdo jiný potřebuje pracujícího – stačí si navzájem posílat zprávy o tom, že se jedna z množin stala neprázdnou. Problémem je, jak pracujícího předat bezpečným způsobem. Aby se někde „neztratil cestou“, je potřeba, aby s ním vždy byl někdo prolinkovaný, a pokud cestou umře, říct si o nového.

Napřed se tedy dva servery, které si ho předávají, nalinkují spolu a dohodnou si předání. Přijímající server si nalinkuje pracujícího a potvrdí předávajícímu, ten ho v tuto chvíli může unlinkovat a říci mu, aby se přesunul. Pracující se připojí na nový server a přesun je hotový.

Co může spadnout? Inu, cokoliv. Když spadne pracující, dozví se o tom přijímající server (linkne si ho) a může požádat o nového. Když umře přijímající server při dohadování, dozví se o tom předávající. V tom případě nic neodešle (není komu). Nakonec, může umřít i předávající, o tom se ovšem dozví přijímající a nebude tedy čekat na pracujícího. (Kdyby náhodou přišel, tak se nic zlého nestane, přijmeme ho tak jako tak. Pokud nepřijde, stejně si od mrtvého serveru nemůžeme vyžádat nového.)

Michal „vorner“ Vaner

Vzorové programy

22-3-6 Kolejní výtahy — C

```
#include <stdio.h>
#include <stdlib.h>

// minimum a maximum
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))

struct MF {
    int start;
    int cil;
} mff[1000001];


// porovnávací funkce pro Quicksort
// makro poslouží pro zjednodušení zápisu
#define R(x) ((struct MF*) x)->cil

int cmp(const void *a, const void *b)
{
    return (R(a) - R(b));
}

int main(void)
{
    int a, b, k, n, o, p, i;
    FILE *fin, *fout;

    fin=fopen("kolej.in", "r");
    fscanf(fin, "%d %d\n", &n, &k);
    for (i = 0; i < n; i++)
    {
        fscanf(fin, "%d %d\n", &p, &o);
        mff[i].start = MAX(1, p-o);
        mff[i].cil=MIN(k, p+o);
    }
    fclose(fin);
    qsort(mff, n, sizeof(struct MF), cmp);

    b = 0;
    a = 0;
    for (i = 0; i < n; i++)
    {
        if (mff[i].start > b)
        {
            b = mff[i].cil;
            a++;
        }
    }

    fout = fopen("zastavky.out", "w");
    fprintf(fout, "%d\n", a);
    fclose(fout);
    return 0;

}
```

22-3-2 Dětská hra — Python

```
#!/usr/bin/python

NEPROSLE = -1 # Značí, že vrchol nebyl navštíven

n = input("Počet dětí: ")
deti = n * [0] # Hrana (u,v) <==> deti[u] == v
cas_vstupu = n * [NEPROSLE]
# Kdy jsme vstoupili do vrcholu poprvé

for i in range(n):
    deti[i] = input("Koho chytá dítě č.%d: " % i)

cas = 1
soucet = 0
for i in range(n):
    # Značíme čas vstupu do prvního prozkoumaného vrcholu
    zacatek_faze = cas

    # Dokud jdeme po neprozkoumaných vrcholech
    while cas_vstupu[i] == NEPROSLE:
        cas_vstupu[i] = cas
        # Zapíšeme čas prvniho vstupu do vrcholu
        i = deti[i] # Podíváme se, kam z něj vede hrana
        cas += 1

    if cas_vstupu[i] >= zacatek_faze:
    # narazili jsme na vrchol prošlý v této fázi => cyklus
        soucet += cas - cas_vstupu[i]

print "Počet dětí, které mohou chytat samy sebe:",soucet
```

22-3-3 Kurýrní služba — C

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 1000
#define MAXM 10000
#define min(a, b) ((a) < (b) ? (a) : (b))
int kon[MAXN], nasl[MAXM], // uložení grafu ve formě seznamu sousedů
    nahrad[MAXN], // nahrazení vrcholu číslem silně souvislé komponenty
    cislo[MAXN], nejnizsi[MAXN], aktualni, // proměnné pro Tarjana
    poc[MAXN], // počet hran vycházejících z vrcholu
    stack[MAXN], pos_stack, // zásobník a pozice jeho vrcholu
    instack[MAXN], // pole určující, jestli je vrchol v zásobníku
    nasl2[MAXM], kon2[MAXN], poc2[MAXN], // kondenzovaný graf
    pocetSSK; // počet silně souvislých komponent

void tarjan(int v) {
    cislo[v] = aktualni; // očíslování vrcholu
    nejnizsi[v] = aktualni; // inicializace nejnižšího dosaženého
    aktualni++;
    stack[pos_stack++] = v; // přidej do zásobníku
    instack[v] = 1;
    int v2;
    for (int i = kon[v] - poc[v]; i < kon[v]; i++) {
        v2 = nasl[i]; // následník
        if (cislo[v2] == -1) { // vrchol nenavšíven
            tarjan(v2);
            nejnizsi[v] = min(nejnizsi[v], nejnizsi[v2]);
        }
        else if (instack[v2]) nejnizsi[v] = min(nejnizsi[v], cislo[v2]); // vrchol je v zásobníku, není tedy ještě v žádné SSK
    }
    if (nejnizsi[v] == cislo[v]) { // nalezena SSK
        int j = 0;
        if (pocetSSK > 0) j = kon2[pocetSSK - 1];
        do { // odebírej vrcholy ze zásobníku
            if (pos_stack > 0) {
                 v2 = stack[--pos_stack];
                 instack[v2] = 0;
                 // přidej vrchol do SSK
                 nahrad[v2] = pocetSSK;
                 // nový seznam následníků
                 for (int i = kon[v2] - poc[v2]; i < kon[v2]; i++)
                    nasl2[j++] = nasl[i];
            }
        } while (v2 != v && pos_stack > 0);
        kon2[pocetSSK++] = j;
    }
}

int main(void) {
    int n, m;
    // načti vstup
    scanf("%d %d", &n, &m); // počet měst a kurýrů
    int h1[MAXM], h2[MAXM];
    // načti orientované hrany
    for (int i = 0; i < m; i++) {
        scanf("%d %d", h1 + i, h2 + i);
        poc[h1[i]]++; // zvyš počet hran vedoucích z vrcholu
    }
    kon[0] = 0;
    nahrad[0] = 1; nejnizsi[0] = cislo[0] = -1;
    for (int i = 1; i < n; i++) {
        kon[i] = kon[i - 1] + poc[i - 1];
```

```
        nahrad[i] = i; nejnizsi[i] = cislo[i] = -1; // inicializace
    }
    for (int i = 0; i < m; i++)    nasl[kon[h1[i]]++] = h2[i];   // vytvoř seznam následníků

    // Tarjanův algoritmus
    aktualni = 0; pocetSSK = 0; pos_stack = 0;  // init; zasobnik je prazdny
    for (int i = 0; i < n; i++)
        if (cislo[i] == -1)    // nalezen neočíslovaný vrchol
            tarjan(i);      // spusť prohledávání do hlouby

    if (pocetSSK == 1) {        // jen jedna silně souvislá komponenta
        printf("Existuje cesta mezi kazdymi dvema mesty alespon jednim smerem.\n");
        return 0;
    }

    // nejprve spočti počet hran vedoucích do vrcholů
    int v = 0;    // index aktuální SSK
    for (int i = 0; i < m; i++)    {
        while (i == kon2[v]) v++;
        if (nahrad[nasl2[i]] != v)  poc2[nahrad[nasl2[i]]]++; // pokud hrana vede jinam než do této SSK
    }
    // najdi SSK, do které nevede hrana
    int prvni = -1;
    for (int i = 0; i < pocetSSK; i++) {
        if (poc2[i] == 0) {    // do SSK nevede hrana
            if (prvni == -1) prvni = i;
            else {        // více SSK, do nichž nevede hrana
                printf("Neexistuje cesta alespon jednim smerem mezi kazdymi dvema mesty.\n");
                return 0;
            }
        }
    }
    // prohledáváním do šířky najdi co nejdelší cestu
    int fronta[MAXN]; // fronta na prohledávání do šířky
    int navstiveno[MAXN]; // navstivili jsme uz SSK?
    int kon_fronta = 0, zac_fronta = 0;
    fronta[kon_fronta++] = prvni;
    navstiveno[prvni] = 1;
    while (zac_fronta < kon_fronta) {
        int v = fronta[zac_fronta++]; // odeber z fronty
        // projdi následníky
        for (int i = v > 0 ? kon2[v - 1] : 0; i < kon2[v]; i++) {
            int a = nahrad[nasl2[i]]; // do jaké SSK jsme se dostali?
            if (a == v) continue; // jsme pořád v té samé SSK
            // přes jaký největší počet SSK se tam dostaneme?
            if (navstiveno[a] < navstiveno[v] + 1) {
                navstiveno[a] = navstiveno[v] + 1;
                if (navstiveno[a] == pocetSSK) { // máme SSK, do níž vede cesta přes všechny SSK
                    printf("Existuje cesta mezi kazdymi dvema mesty alespon jednim smerem.\n");
                    return 0;
                }
            }
            poc2[a]--; // odeber hranu
            if (poc2[a] == 0)  fronta[kon_fronta++] = a; // pokud už do SSK nevede žádná další hrana, zařadíme ji do fronty
        }
    }
    printf("Neexistuje cesta alespon jednim smerem mezi kazdymi dvema mesty.\n");
    return 0;
}
```

22-3-4 Rukavice — C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int typ;
    int pocet;
} ponozky;

int cmpf(ponozky *p1, ponozky *p2);

int main()
{
    int L = 0, P = 0;
    int l, p;
    scanf("%d %d", &l, &p);

    ponozky LB[l], PB[p];
    for (int i=0; i<l; i++)
        scanf("%d %d", &(LB[i].typ), &(LB[i].pocet));
    for (int i=0; i<p; i++)
        scanf("%d %d", &(PB[i].typ), &(PB[i].pocet));

// Budeme předpokládat, že typy ponožek jsou přirozená
// čísla z intervalu 0..(l+p-1) -- na ošklivěji
// zadaných vstupech (typy by koneckonců šly určovat
// třeba řetězci jako "zelená se žlutými puntíky")
// bychom je uspořádali a přečíslovali.

    int n = l;

    // Ověříme existenci alespoň jedné ponožky do páru.
    int jeLeva[l+p], jePrava[l+p];
    for (int i=0; i<l+p; i++) { jeLeva[i]=0; jePrava[i]=0; }
    for (int i=0; i<l; i++)
        if (LB[i].pocet != 0)
            jeLeva[LB[i].typ]=1;
    for (int i=0; i<p; i++)
        if (!jeLeva[PB[i].typ])
        {
            P += PB[i].pocet; // Vytáhneme je
```

```
            PB[i].pocet = 0;
        }
        else
            if (PB[i].pocet != 0)
                jePrava[PB[i].typ]=1;
    for (int i=0; i<l; i++)
        if (!jePrava[LB[i].typ])
        {
            L += LB[i].pocet;
            LB[i].pocet = 0;

            n--;
        }

    qsort(LB, l, sizeof(ponozky), cmpf);
    qsort(PB, p, sizeof(ponozky), cmpf);

    int nejL = 1000, nejP = 1000;
    int meziL = 1, meziP = 1;
    for (int i=0; i<n; i++) { meziP += PB[i].pocet; }

    // konečně kýžený důležitý cyklus
    for (int k = 0; k <= n; k++)
    {
        if (k!=0) meziL += LB[k-1].pocet;
        if (k!=n) meziP -= PB[n-k-1].pocet;

        if (meziL + meziP < nejL + nejP) {
            nejL = meziL; nejP = meziP;
        }
    }

    printf("L: %d, P: %d\n", (nejL + L), (nejP + P));
    return 0;
}

int cmpf(ponozky *p1, ponozky *p2)
{
    return p2->pocet-p1->pocet;
}
```

22-3-5 Reklama — C++

```
#include <iostream>
#include <algorithm>
using namespace std;
#define MAXN 100000

// y-ové súradnice koncov čiar, usporiadané zostupne
int Y[MAXN];

// body na vstupe
pair<int,int> body[MAXN];

int main() {
    int N, x, y, poc = 0;

    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> x >> y;
        // otočiť súradnice o 45 stupňov
        body[i] = make_pair(x+y,x-y);
    }

    // utriediť podľa x (pre rovnaké x podľa y)
    sort(body,body+N);

    for (int i = 0; i < N; i++) {
        // binárnym vyhľadávaním nájsť najvyšší koniec,
        // ktorý je nižšie ako súradnica y
        int pos = lower_bound(Y,Y+poc,-body[i].second)-Y;
        // ak taký nie je, vytvoriť nový
        if (pos == poc) poc++;
        // predĺžiť nájdený koniec čiary
        Y[pos] = -body[i].second;
    }

    cout << poc << endl;
    return 0;
}
```

22-3-7 Pavouci internetu — Erlang

```
-module(klient).
-export([zadej/1, cekej/2]).

% Odešle požadavek na server a počká, jestli ho přijme.
% Pokud ne, zkusí další server.
%
% Po přijetí počká na první odpověď (je možné, že se původní
% přiřazení někde zdrželo).
cekejInterni(Funkce, Zadavatel, [Server|Nahradni]) ->
        % Řekne serveru, že chceme něco provést
        {server, Server} ! {prace, self(), Funkce},
        receive
                % Server práci přijal
                prijato -> receive
                        % Server nám při tom umřel, zkusíme to jinde
                        {'EXIT', Server, _} -> restart;
                        % Bylo to přiřazeno nějakému pracantovi
                        {prirazeno, PID} -> link(PID), receive
                                % Umřel pracant (server už nás nezajímá)
                                {'EXIT', PID, _} -> restart;
                                % Hurá, máme výsledek, pošleme to majiteli a končíme
                                {vysledek, Stav, Hodnota} -> Zadavatel ! {vysledek, Stav, Hodnota}, PID ! ok
                        end
                end
                % Server neodpovídá, zkusíme jiný
                after nastaveni:timeout_serveru() -> cekejInterni(Funkce, Zadavatel, Nahradni)
        end.

cekej(Funkce, Zadavatel) ->
        % Kdyz umře server (který si nás při přijetí linkne), tak o tom chceme vědět
        process_flag(trap_exit, true),
        % Zkus to vyřídit
        case cekejInterni(Funkce, Zadavatel, nastaveni:servery()) of
                % Umřel server, nový pokus
                restart -> cekej(Funkce, Zadavatel);
                % Vše v pořádku
                _ -> ok
        end.
```

```
% Zadá funkci ke zpracování.
zadej(Funkce) -> spawn(klient, cekej, [Funkce, self()]).
-module(nastaveni).
-export([servery/0, timeout_serveru/0, timeout/0]).

% Kde běží servery?
servery() -> [server@localhost, nahradni@localhost].
% Jak dlouho budeme čekat, než prohlásíme server za mrtvý a zkusíme jiný?
timeout_serveru() -> 1000.
% 5 minut bude trvat nejdelší povolený úkol
timeout() -> 300000.
-module(prace).
-export([start/0, start/1, pracuj/2, proved/2]).

servery() -> lists:map(fun (S) -> {server, S} end, nastaveni:servery()).

proved(Majitel, Fun) -> Majitel ! {vysledek, ok, Fun()}.

% Počká, až nám zadavatel dovolí skončit nebo skončí sám
pockej(Zadavatel) -> receive
        ok -> ok;
        {'EXIT', Zadavatel, _} -> ko
end.

pracuj(Zadavatel, Fun) ->
        process_flag(trap_exit, true),
        link(Zadavatel),
        % Spustíme výpočet
        Pracujici = spawn_link(prace, proved, [self(), Fun]),
        % A počkáme, jestli to vyjde, nebo spadne
        receive
                % Vyšlo, pošleme výsledek
                {vysledek, Stav, Hodnota} -> Zadavatel ! {vysledek, Stav, Hodnota}, pockej(Zadavatel);
                % Umřelo, pošleme hlášení o chybě
                {'EXIT', Pracujici, Chyba} -> Zadavatel ! {vysledek, chyba, Chyba}, pockej(Zadavatel);
                % Umřel zadavatel, zabij práci
                {'EXIT', Zadavatel, _} -> exit(Pracujici, kill)
                % Už běží moc dlouho, zabít
                after nastaveni:timeout() -> Zadavatel ! {vysledek, timeout, nic}, exit(Pracujici, kill), pockej(Zadavatel)
        end,
        % Zase se přihlásíme o práci
        start().

pracuj() ->
        % Neumřeme, když umře server
        process_flag(trap_exit, true),
        receive
                % Chce se po nás, abychom pracovali někde jinde, tak tedy se přesuneme tam a dostaneme práci
                % Je stejné, jako přihlášení, jen je ten server, který chce práci první v seznamu.
                {presun, Odkud, Kam} ->
                        unlink(Odkud), % S ním už nemáme nic společného
                        link(Kam), % Sem se nastěhujeme
                        Kam ! {prihlasit, self()}, % Přihlašme se
                        receive
                                prijato -> pracuj(); % Přijal nás
                                {'EXIT', Kam, _} -> start() % Umřel než nás přijal, zkusíme nový start
                        end;
                % Přišla práce. Skončíme (tím se odpojíme ze serveru), ale předtím ještě spustíme vlastní zpracování.
                {prace, Zadavatel, Fun} -> pracuj(Zadavatel, Fun);
                % Umřel server, přihlásíme se jinam o práci
                {'EXIT', _, _} -> start()
        end.

start([Server|Nahradni]) ->
        % Zkusme se přihlásit na server
        Server ! {prihlasit, self()},
        receive
                % Super, chce nás
                prijato -> pracuj()
                % Neozývá se, zkusíme jiný
                after nastaveni:timeout_serveru() -> start(Nahradni)
        end.

start() -> spawn(prace, start, [servery()]).
-module(server).
-export([start/0, startInterni/0]).

rozesli(Co) -> lists:foreach(fun (Server) -> {server, Server} ! {Co, self()} end, nastaveni:servery()).
```

```
% Když máme jak úkoly, tak pracující, tak něco z toho zpracujeme
zpracuj([{Zadavatel, Fun}|Ukoly], [Pracant|Pracujici]) ->
        % Předáme práci
        Pracant ! {prace, Zadavatel, Fun},
        % Už se o něj nemusíme starat, oni si to vyřídí mezi sebou
        unlink(Pracant),
        % Zadavateli o tom řekneme a dál nás nezajímá
        Zadavatel ! {prirazeno, Pracant},
        unlink(Zadavatel),
        % A nyní ten zbytek úkolů
        zpracuj(Ukoly, Pracujici);
zpracuj(Ukoly, Pracujici) -> receive
                % Chce se po nás vykonávat nějaká práce
                {prace, Zadavatel, Fun} ->
                        % Vezmeme si to na starost a uložíme
                        link(Zadavatel),
                        Zadavatel ! prijato,
                        case Pracujici of
                                [_|_] -> ok;
                                % Nemáme žádného pracanta, řekneme si o nějaké
                                true -> rozesli(chciPracanta)
                        end,
                        % A podíváme se, co se dá dělat nyní
                        zpracuj(lists:append(Ukoly, [{Zadavatel, Fun}]), Pracujici);
                % Přišel nový pracant.
                {prihlasit, Pracant} ->
                        % OK, bereme ho
                        link(Pracant),
                        Pracant ! prijato,
                        case Ukoly of
                                [_|_] -> ok;
                                true -> rozesli(mamPracanta)
                        end,
                        zpracuj(Ukoly, [Pracant | Pracujici]);
                % Někdo chce pracanta
                {chciPracanta, Server} -> case Pracujici of
                        % Máme ho, tak mu řekneme, ať se přestěhuje
                        [Pracant | Zbytek] ->
                                % Dohodneme se s druhým serverem, aby ho čekal
                                link(Server),
                                Server ! {cekej, Pracant, self()},
                                receive
                                        % Čeká na něj, pošleme mu ho
                                        cekam ->
                                                unlink(Server),
                                                Pracant ! {presun, self(), Server},
                                                zpracuj(Ukoly, Zbytek);
                                        % Umřel, tak si ho necháme
                                        {'EXIT', Server, _} ->
                                                zpracuj(Ukoly, [Zbytek])
                                end;
                        % Nemáme, ignorujeme požadavek
                        [] -> zpracuj(Ukoly, Pracujici)
                end;
                % Máme očekávat pracanta
                {cekej, Pid, Od} ->
                        link(Pid),
                        Od ! cekam,
                        receive
                                % Přišel
                                {prihlasit, Pid} ->
                                        % Přijmout, zařadit a pokračovat
                                        Pid ! prijato,
                                        zpracuj(Ukoly, [Pid | Pracujici]);
                                % Ten už nepřijde, řekneme si o jiného
                                {'EXIT', Pid, _} ->
                                        Od ! {chciPracanta, self()},
                                        zpracuj(Ukoly, Pracujici);
                                % Umřel server, který ho měl poslat, smůla
                                {'EXIT', Od, _} -> zpracuj(Ukoly, Pracujici)
                        end;
                % Někdo nabízí pracanta
                {mamPracanta, Server} ->
                        case Ukoly of
                                % Máme pro něj využití, řekneme si o něj
                                [_|_] -> Server ! {chciPracanta, self()};
                                [] -> ok
                        end,
                        zpracuj(Ukoly, Pracujici);
                % Něco skončilo. Ať to byl pracant nebo zadavatel, odebereme všechny, které tomu tady odpovídají
                % Pokud to byl zadavatel, tak se nic neděje
```

```
                % Pokud to byl pracant, zadavatel si požadavek zadá znovu
                {'EXIT', PID, _} -> zpracuj(lists:keydelete(PID, 1, Ukoly), lists:delete(PID, Pracujici))
        end.

startInterni() ->
        % Určitě nechceme umírat, když nepřežije některý klient, jen ho vyřadíme
        process_flag(trap_exit, true),
        % Začneme pracovat, nejsou žádné úkoly ani pracanti
        zpracuj([], []).

start() -> register(server, spawn(server, startInterni, [])).
```

Výsledková listina dvacátého druhého ročníku KSP po třetí sérii

		škola	*ročník*	*sérií*	*2231*	*2232*	*2233*	*2234*	*2235*	*2236*	*2237*	*série*	*celkem*
1.	Jiří Eichler	SlovanG_OL	2	3	7,5	6	13	10	11	10	12	48,5	133,1
2.	Pavol Rohár	GMRŠKošice	4	5	7	8	9	3	3	10	6	37,7	113,4
3.	Vojtěch Kolář	G_Neratov	4	14		8	9	6	3	10		31,0	112,9
4.	Filip Hlásek	GMikul23PL	3	13	9	8	8		9	10		34,9	106,8
5.	Vojtěch Hlávka	GŠlapanice	1	3	6,5	1	5	7	6	2		35,4	99,1
6.	Vlastimil Dort	GŠpitálsPH	4	18		7				9		14,0	94,1
7.	Miroslav Olšák	GBuďánkaPH	4	2	10	4	13		14	10		47,7	89,1
8.	Štěpán Šimsa	GJungmanLT	1	7		7		10	3	10		31,8	81,3
9.	Karel Tesař	SPŠE_Plzeň	4	11	5	7		7		10		29,0	80,1
10.	Ondřej Hübsch	ZŠJílov_PH	0	3		5	2		3	10		26,9	69,9
11.	Petr Hudeček	GCoubTábor	2	2								0,0	48,5
12.	Petr Pecha	SPŠsVsetín	3	9	4	8		8		1		22,4	48,1
13.	Karel Král	G_Most	4	8	7	8		10				25,8	45,2
14.	Jiří Setnička	G25březnPH	3	8								0,0	40,9
15.	Martin Zikmund	G_Turnov	2	6		8						8,0	37,1
16.	Petr Čermák	GEBenešeKL	4	6								0,0	34,8
17.	Filip Štědronský	GMikul23PL	3	11		7	6					13,0	34,3
18.	Pavel Taufer	ArcibisGPH	4	10	4,2		7	6				18,4	33,3
19.	Daniel Stahr	GJungmanLT	3	1		5	6		3	1		26,1	26,1
20.	Daniel Šafka	GKepleraPH	3	1								0,0	25,1
21.	Petra Vahalová	G_Plasy	4	1	5	7		7				24,8	24,8
22.	Tomáš Novella	GAlejKošic	4	1								0,0	23,9
23.	Jakub Diatel	G_Slavičín	2	3	0			3				5,4	22,8
24.	Ondřej Mička	G_Jírov_ČB	1	2								0,0	22,7
25.	Martin Holec	G_Slavičín	3	6							4	5,9	22,2
26.	Mária Mrocková	GJHroncaBA	3	1	3	1	2	6				21,8	21,8
27.	Jonatan Matějka	G_Jírov_ČB	0	2	4							6,8	20,2
28.	Radim Cajzl	GNoMěsNMor	3	22		7				1		5,6	18,6
29.	Petr Zvoníček	G_Slavičín	4	7	3							4,2	18,2
30.	Matěj Kocián	GLesníZlín	3	1		8		7				17,1	17,1
31.	Kateřina Lorenzová	G_Česká_ČB	3	7								0,0	15,7
32.–33.	Dana Marečková	GPatočkyPH	4	1								0,0	12,8
	Filip Matzner	GJirsíkaČB	3	1								0,0	12,8
34.	Jakub Červenka	GŠpitálsPH	4	5								0,0	12,0
35.	Tomáš Masák	GJirsíkaČB	3	1								0,0	10,7
36.	Hynek Jemelík	GJarošeBO	3	5						10		10,0	10,0
37.	Tomáš Maleček	GEBenešeKL	4	1								0,0	9,1
38.	Martin Mach	G_Jírov_ČB	2	1								0,0	7,9
39.	Pavel Kratochvíl	VOŠGSvětlá	2	8	3			3				7,8	7,8
40.	Michal Bilanský	GLepařovJČ	4	6								0,0	6,6
41.	Karel Hulec	GJirsíkaČB	3	1								0,0	6,0