Milí řešitelé!

Po kratší pauze již jistě netrpělivě čekáte na závěrečný díl našeho seriálu na pokračování. V dnešní epizodě se dozvíte, jak to dopadne s Felixem, co ošklivého v útrobách hory našel a jak se s tím naše družinka vypořádá. Pohodlně se usaďte ve svých křesílkách, pohádka začíná…

Termín odeslání Vašich řešení je pro pátou sérii stanoven na 5. května 2008. Řešení můžete odevzdávat elektronicky na *http://ksp.mff.cuni.cz/submit/,* nebo klasickou poštou na známou adresu:

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

Aktuální informace naleznete na stránkách *http://ksp.mff.cuni.cz/,* diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz.*

Pátá série dvacátého ročníku KSP

„Domňauhajzlůůůcosetosakratadyseněcopálííí… “

Z úzké chodby vyletěl ohořelý kocour a těsně za ním vyšlehl plamen. Felix přistál na chladných kamenech venku a tiše doutnal. Přitom vrhal ošklivé pohledy do širokého a dalekého okolí a speciální dávku věnoval samotnému mágovi.

„Je tam drak,“ pronesl suše po nervózní pauze, ve které se každý snažil vyhnout kocourově pohledu.

„Drak,“ opakoval nepřítomně mág a pohladil si plnovous. „Tak to musí být Ohnivý drak!“

„Nevím, nestačil jsem si všimnout,“ ušklíbl se sarkasticky Felix a začal si olizovat popálený kožich.

„Ale, vždyť ve Škytánii žádní draci nejsou!“ vykulil oči Vilda.

„To nejsou … ehm – nebyli,“ ohradil se mág, když spatřil kocourův výraz. „Ale každopádně by mě zajímalo, jak se sem dostal. To musel přiletět až z Velkého pohoří … “

20-5-1 Dračí cestování — 8 bodů

Dračí cestování není tak jednoduché, jak by se mohlo zdát. Let je pro tvora velikosti malého dopravního letadla namáhavý, a proto se musí drak dobře živit, aby měl dost sil na mávání křídly.

Největší pochoutkou (a zároveň jedinou potravinou, která má dostatečný energetický potenciál) je pro draka síra. Při letu drak spotřebuje 1 kg síry na 1 km.

Síra se bohužel ve Škytánii vyskytuje pouze u Ohnivé hory, takže veškerou síru na cestu si musel drak nést z Velkého pohoří. Délka trasy, kterou musel uletět, je 4200 km. Navíc si drak chtěl přivést co nejvíce síry s sebou, protože nevěděl, jak kvalitní a rozsáhlé budou místní sirné zásoby.

Zjistěte, jaké maximální množství síry si drak mohl přinést, když má nosnost 42 metráků (4200 kg) a ve Velkém pohoří bylo zrovna k dispozici 14 tun (14 000 kg) síry.

Drak si pochopitelně může cestou dělat překladiště síry, kde si kousek nákladu odloží a pak se pro něj vrátí.

„…no ovšem, to je jasné,“ prohlásil spokojeně mág. „Stačilo, aby si udělal překladiště u Knůllu a pak ještě dvě nebo tři v Troudském lese a měl vystaráno … “

„To je úplně fuk, jak se sem dostal!“ vykřikl Felix. „Podstatné je, že je tady a teď. Viděli jsme, slyšeli jsme a teď bychom mohli takticky ustoupit. Beztak s ním nic nezmůžeme.“

„Krá!“ přitakal Kiri, který doteď plnil pouze funkci težítka mágova ramene.

„Máte pravdu, tady nemůžeme zůstat,“ pokýval hlavou mág. Sotva to dořekl, přeletěl jim nad hlavami oborvský stín. Mág zavřel oči a pozvedl ruce. Čelo se mu nakrabatilo hlubokým soustředěním. Mezi jeho dlaněmi se objevila modrá koule a rychle se zvětšovala. V mžiku pohltila celou družinu a s tichým „pop“ zmizela.

O pár set metrů dál se zavlnil vzduch. Ozvalo se opět tiché „pop“ následované hlasitým heknutím, když povedená čtveřice dopadla na zem. Mág si otřel čelo a roztřesenýma rukama si přihnul z měchu s vodou, aby se trochu uklidnil.

„Na toho draka sami nestačíme,“ posteskl si Vilda.

„To si pište, že nestačíte,“ ozval se jim za zády skřehotavý hlas. Všichni se jako na povel otočili. Stála tam shrbená stařena oblečená celá do černých šatů. Nebyla to temná čerň, kterou by ji mohla závidět i noc. Jen obyčejná všední vybledlá čerň, která o své nositelce prozrazovala maximálně to, že nerada pere. Na hlavě měla špičatý klobouk propíchlý několika jehlicemi.

„Kdo jsi a co tu děláš?“ obořil se na ni mág. Žena na něj vrhla ošklivý pohled. Mág luskl prsty a zeptal se znovu: „Kdo jsi a co tu děláš?“ a jeho hlas ukapával jako med z včelí plástve.

„Pch,“ oznámila mu stařena. „Si myslíš, že jsi nějakej mág, nebo co? Že si tady luskneš prsty a všechny tu omámíš? Já jsem Čarodějka! Na mě tyhle laciné triky neplatí!“

„Možná že neplatí, ale i tak jsi mi zodpověděla půlku mé otázky,“ usmál se mág.

Čarodějka se zamračila: „Kdyby ses pořádně podíval, všiml by sis, že mám klobouk, a nemusel by ses ptát.“

Mág už otvíral pusu, aby kontroval nějakou peprnou narážkou, ale pak ji zase zavřel. Hádání mu tady nijak neprospěje. On se potřebuje dostat do dračí jeskyně a nasbírat dostatek Temných kamenů. „Poslyš,“ začal mág opatrně, „ty o tom drakovi něco víš?“

„Jistě, že vím. Vím to nejdůležitější, co se o něm dá vědět: Jeden by se měl od něj držet co nejdál!“

„No ne, vážně?“ upřel na ni nevinný pohled Felix a při tom žoviálně pohupoval ohořelým ocasem na všechny strany.

„Ale my se potřebujeme dostat dovnitř a nasbírat nějaké Temné kameny,“ pokýval smutně hlavou mág.

„Pak bych tu možná měla něco, co by vám mohlo pomoct. Ale něco za něco … “

Družinka následovala čarodějku až k jejímu domku uprostřed lesa. Když zastavili, ukázala čarodějka směrem na půdu: „Na půdě se mi přemnožili skřítci a já už nevím, co s nimi.“

„To znám, jednou se mi to také přihodilo,“ přikývl mág. „Na to je nejlepší 'Piškorcův deratizátor'!“

„Žádná taková bylinka tady neroste. To bych musela vědět.“

„Ale ne, to je zaklínadlo!“ vysvětloval mág s převahou znalce. „Má ale jeden háček … “

„To mají zaklínadla vždycky,“ ušklíbla se čarodějka.

„Deratizovat se musí přesný počet skřítků. Když jich je víc, tak nějací přežijí a pak se dál množí. A když je jich míň, tak se musí přebytečná magenergie někam uvolnit a to bývá často … nepříjemné,“ dokončil neohrabaně mág.

„Myslíš tak nepříjemné jako tenkrát, kdy ti narostly oslí uši a celý týden jsi nemohl vyjít z ložnice?“ pochechtával se Felix.

„Ne, myslím tak nepříjemné, že by přebytečná magenergie deratizovala jiné tvory v blízkém okolí. A začala by těmi menšími … “ zmrazil kocourovu zábavu mág.

20-5-2 Piškorcův deratizátor — 10 bodů

Je potřeba, aby při sesílání kouzla byl počet skřítků alespoň N. Na začátku je skřítků K, kde $K < N$. Každý den se přesně o půnoci počet skřítků ztrojnásobí. Čarodějka umí každý den povolat nového skřítka nebo jednoho skřítka nechat zmizet. Samozřejmě nemusí dělat nic a nechat populaci takovou, jaká je.

Mág umí seslat (i několikrát za den) deratizační kouzlo, při kterém zmizí právě N skřítků. Aby ho mohl seslat, musí být skřítků alespoň N, jinak by se mohly stát ošklivé věci.

Navrhněte postup, jak skřítky každý den přidávat, odebírat a případně deratizovat, aby na konci nezbyl žádný. Musíte mága šetřit, aby se úplně nevyčerpal, neboť ho ještě čeká souboj s drakem, takže vámi navržený postup musí obsahovat co nejméně deratizací. Navíc by vytvoření vašeho plánu mělo trvat co nejkratší dobu, aby se jím stihli čaroděj s čarodějkou vůbec řídit.

Čarodějka i mág mohou kouzlit hned první den. Nemusí tedy čekat, až se jim K ještě ztrojnásobí.

Příklad: Na začátku mějme 4 skřítky a deratizanční kouzlo jich zlikviduje 7. Sledujme populaci po jednotlivých dnech (v závorkách jsou počty skřítků):

1 . (4) zmiz skřítka (3)
2 . (9) deratizace (2)
3 . (6) přidej skřítka (7) deratizace (0)

„Jo, všechna ta havěť je pryč,“ usmál se pod vousky Felix, když pečlivě prošmejdil celou půdu. „Ale mohli jste mi nechat jednoho na hraní … “

„To by tak ještě chybělo,“ odbyl ho mág. „Ty jsi neviděl, jak se ty potvory rychle množí?“

„Výborně, chlapci,“ pochválila je čarodějka a postavila před Felixe misku s mlékem.

„Krá, krá,“ ozval se Kiri a dožadoval se také nějaké odměny, ale pro havrana se v domě nenašel jediný pamlsek.

Čarodějka otevřela jednu ze svých truhel a podala mágovi zažloutlý svitek převázaný pečetí. Na pečeti byl symbol draka uzavřený do kruhu. Mág z části sfoukl a z části sklepal vrstvu prachu, která svitek pokrývala. „Co to je?“ zeptal se podezřívavě.

„To je svitek ochrany před draky. Po rozlomení pečeti se kolem svitku vytvoří neviditelná bariéra o poloměru 42 metrů, do které není žádný drak schopen vstoupit,“ vysvětlovala trpělivě čarodějka.

„A jak dlouho to vydrží?“

„Asi hodinu. Doufám. Alchymista, který mi ten svitek věnoval, byl celkem roztržitý … “

Mág poděkoval a celá družina se vydala zpět k Ohnivé hoře. Hora stála na svém místě a dýmala. Po drakovi nebylo ani vidu ani slechu. Vzduch byl nehybný a všude vládlo naprosté ticho. Klid před bouří, pomyslel si mág. Teď musíme najít vchod do jeskyně, kde bydlí ten drak. Konec konců, musel se přece nějak dostat ven.

Několik hodin chodili po úbočí hory, když si Vilda všiml velké díry vysoko ve skále. Tou by se určitě protáhl i drak. Družina se s funěním a hekáním vyškrábala až k výklenku.

„Ťuk, ťuk,“ řekl potichu Felix, sotva popadl dech. „Vidíte, nikdo není doma, tak můžeme zas jít, ne?“ Mág ho ale odstrčil z cesty a pevným krokem vykročil vpřed. Tunel se pozvolna rozšiřoval, až vyústil do obrovské sluje, do které by se vešlo … opravdu hodně draků. Naštěstí tam byl jen jeden. Ležel na hromadě horkých kamenů a spal.

Mág se rozhlédl po jeskyni. Na spoustě míst ležely Temné kameny, ale jeskyně byla příliš velká, než aby ji celou pokryl kouzlem ze svitku …

20-5-3 Ochrana před draky — 13 bodů

Pro naše potřeby si úlohu trochu zjednodušíme. Představíme si pouze dvourozměrný půdorys dračí jeskyně. Máme seznam souřadnic, kde se nachází Temné kameny. Svitek má dosah N metrů a mág se s ním snaží pokrýt co nejvíc kamenů. Kámen je považován za pokrytý, pokud je jeho vzdálenost od svitku menší nebo rovna N.

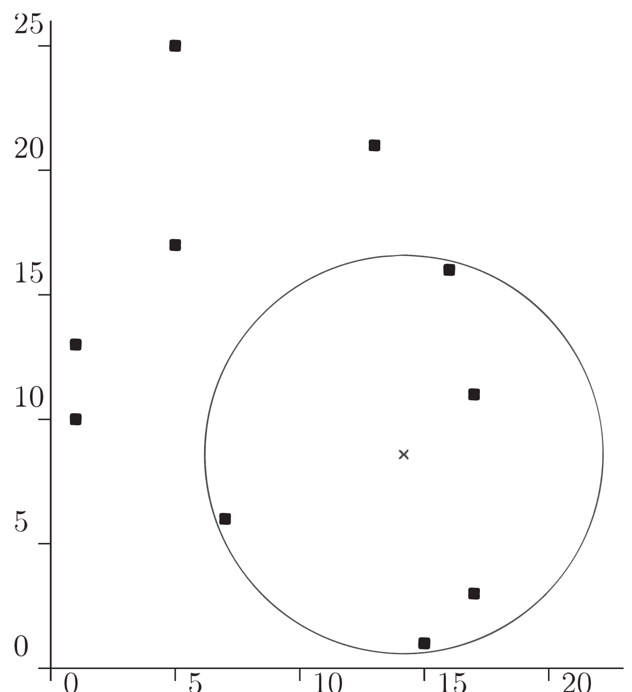
Napište algoritmus, který dostane seznam bodů v rovině a nalezne bod takový, že když v něm nakreslíme kruh o poloměru N, bude tento kruh pokrývat maximální možný počet bodů. Střed kruhu může být kdekoli, ne nutně v nějakém zadaném bodě.

Pokud existuje takových bodů víc, stačí vypsat jeden libovolný z nich. Souřadnice se uvádějí jako reálná čísla (a vejdou se do nějakého float typu v počítači).

Příklad: Řekněme, že poloměr svitku bude 8 metrů a v jeskyni je 10 kamenů na následujících souřadnicích:

1.0, 13.0 5.0, 17.0 7.0, 6.0 13.0, 21.0 17.0, 3.0 5.0, 25.0 15.0, 1.0 16.0, 16.0 17.0, 11.0 1.0, 10.0

Pokud umístíme svitek na souřadnice $[14.178125, 8.58475]$, pokryjeme maximální počet – 5 kamenů.



Mág došel doprostřed jeskyně a rozhlédl se. Ano, tady je nejlepší místo. Vytáhl z vaku svitek a pečlivě si ho ještě jednou prohlédl. Drak otevřel oko.

„Á, návštěva!“ zaburácel drak a udělal krok k mágovi.

Jestli to teď nebude fungovat, tak je s námi konec, pomyslel si mág. Pozvedl svitek a zavolal: „Neprojdeš dál!“ Na ta slova rozlomil pečeť. Mágovi se naježily vousy i vlasy a vzduch se naplnil mazlavým zápachem použité magenergie.

„A kam bych jako neměl projít?“ zeptal se drak a udělal další dva kroky. Chtěl udělat ještě jeden, ale narazil na neviditelnou bariéru a rozplácl se na ní, jako moucha na předním skle závodního létajícího koštěte.

„Už chápu,“ brblal si pro sebe, když si masíroval naražený čumák. „To je zajímavá věcička . . . “

K mágovi zatím přiběhl Vilda a začal sbírat Temné kameny.

„Tak by mě zajímalo,“ nahodil drak konverzačním tónem, „jestli vás to kouzlo taky ochrání před předměty, které bych mohl třeba neopatrně upustit dovnitř . . . “ Mág ustaraně pozoroval, jak drak sebral středně velký kámen do pařátů a ledabyle ho prohodil magickou bariérou.

„Hups,“ usmál se drak. „A taky by mě zajímalo, jestli ten svitek bude fungovat i potom, co shoří,“ řekl a z nosu mu vyšel obláček kouře. Mág odhodil svitek na zem a o několik kroků ustoupil.

„Jen nevím, kam si vás vystavím,“ pokračoval drak. „Mám tu barbary drakobijce, trpaslíky drakobijce, dokonce i pár rytířů . . . ale kouzelník, zombie, kočka a pták . . . na to si budu muset založit samostatnou sbírku.“

Vilda už měl v torně několik Temných kamenů a tak začal ustupovat společně s mágem. Drak nasměroval svůj chřtán přímo na svitek a úzkým kontrolovaným plamenem ho v mžiku přeměnil na uhel. Ozvalo se slabé zapraskání a magická bariéra povolila.

„Ehm,“ odkašlal si mág, „my jsme vám nepřišli nijak . . . ublížit . . . “

„Ale ovšem, že ne. Nechte mě hádat – vy jste přišli na koblihu a šálek čaje, že?“ zašklebil se na něj drak.

„No, ve skutečnosti jsme přišli . . . jen pro pár Temných kamenů.“ vypravil ze sebe mág a přitom horečnatě přemýšlel, jak z téhle nepříjemné situace ven.

„Ach tak. Takže vás zajímá pár obyčejných šutrů, zatímco drak a jeho poklad jsou vám úplně lhostejní, že?“

„Jaký poklad?“ podíval se na něj mág.

„Hmm. Tady něco nesedí,“ zarazil se drak. „Jenom abych si to ujasnil: Vy jste sem přišli s nějakou magickou ochranou před draky, ale ve skutečnosti jste neměli v úmyslu mi nijak ublížit a několik bezcenných šutrů vás zajímá víc, než můj poklad,“ přemýšlel nahlas.

„Jo, přesně tak to bylo,“ přikyvoval horlivě Vilda.

„Krá,“ přispěchal mu na pomoc Kiri.

„Já jim hned říkal, že to neprojde,“ přidal se Felix. „Co na mě všichni tak zíráte? Říkal jsem vám to! Nebo snad ne?!“

„Takže, přišli, seslali a nechtějí poklad,“ mumlal si pro sebe drak, jako by s tou myšlenkou měl potíže. „Pořád tady jedné věci nerozumím – proč?“

Mág mu začal vysvětlovat, jaké to je, být pánem Temného hvozdu, a co všechno musí dělat, aby si udržel potřebný image. Pak tu byla ta patálie s temnou lucernou a Temné kameny se špatně shánějí. Vyprávěl mu, jak museli projít Temným hvozdem, zjistit, kde vlastně takové kameny hledat, a pak se dotrmácet až sem přes všechny ty močály a druidy . . .

Drak se zaujetím poslouchal a občas vypustil proužek dýmu. A protože mág uměl každý příběh podat jako nikdo jiný, začal drak dojetím slzet.

„To je tak dojebdý příběh,“ řekl drak a popotáhl. „Debáte dáhodou khapesník?“

Mág s Vildou se na sebe podívali. „Máme jen tohle,“ řekl Vilda, vytáhl z vaku obrovskou deku a podal ji drakovi.

„Ďhekuju,“ odvětil drak a hlasitě se vysmrkal, až to zatřáslo jeskyní. Deka mu shořela v pařátcích na troud a rozsypala se.

„Víte, já jsem se sem přestěhoval z daleka,“ navázal drak, když se trochu sebral. „Žil jsem s rodiči ve Velkém pohoří, ale znáte to. Přijde čas, kdy se potřebujete trochu osamostatnit. Vzlétnout na vlastní křídla, jak se říká. Myslel jsem, že tady to bude lepší. Nová země, čerstvá síra . . . ale ve skutečnosti je to tady hrozné. V horách jsem měl klid. Jenže sem mi pořád někdo leze. Většinou je to pořád dokola to samé – hrdina sem přijde, vyzve mě na souboj a já pak jen po něm uklidím ohořelé boty a meč s nápisem 'Drakobijec', nebo tak nějak. Problém je, že hora je přímo protkaná velkým množstvím nejrůznějších tunelů a jeskyní, které tu vytvořila láva. Některé tunely vedou i na povrch a pak mi sem lezou lidé. Chtěl jsem některé tunely zasypat, ale nevím které. A taky si nechci zasypat svůj poklad . . . “ dokončil smutně drak.

„S tím bych možná mohl pomoct,“ usmál se mág.

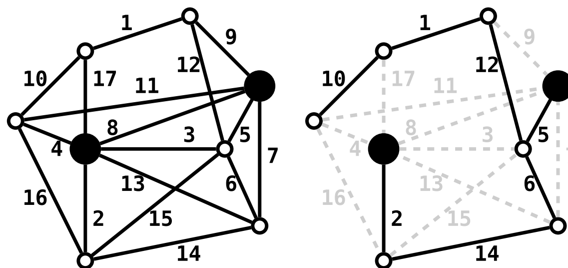
20-5-4 Dračí chodbičky — 11 bodů

Spleť dračích chodeb a jeskyní si představíme jako graf, kde vrcholy jsou jeskyně nebo křižovatky a hrany jsou tunely. Graf nemusí být nutně rovinný, protože hora je velká a některé tunely se mohou křížit mimoúrovňově.

Drak by rád co nejvíc chodeb zasypal, ale zároveň chce, aby se dostal do všech jeskyní (vrcholů). Také vám dává seznam míst, ve kterých má část pokladu. K takovým místům by chtěl nechat pouze jednu přístupovou chodbu (tj. z těchto vrcholů mají být listy). Navrhněte, které chodby by měl drak zachovat, aby součet délek zasypaných chodeb byl největší možný.

Můžete předpokládat, že zadaný problém má řešení (tzn. z vrcholů s pokladem lze udělat listy, aniž by se graf rozpadl na více komponent).

Příklad: Vlevo je obrázek současného stavu tunelů v Ohnivé hoře (místa s pokladem jsou vyznačena černě). Vpravo pak vidíte výsledek (zasypané chodby jsou čárkované).



Celá hora se třásla a všude se ozýval ohlušující zvuk padajícího kamení.

„To byla poslední,“ pohladil si mág spokojeně plnovous, když hluk ustal.

„To je úžasné,“ rozplýval se drak nad provedenými stavebními úpravami. „A tady bych si mohl zřídit konferenční salónek. Až přiletí naši na návštěvu, ti budou koukat . . . “

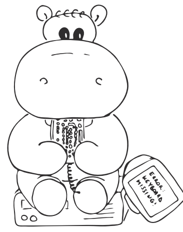
Družinka se rozloučila s drakem a vydala se na dlouhou cestu zpět do Temného hvozdu. Putování jim zabralo hezkých pár týdnů, ale nakonec dorazili všichni ve zdraví domů.

Před Temnou věží se už tísnilo několik rádobydobrodruhů, kteří se zbraní v ruce čekali na mága. Mág jim pokynul na pozdrav a pozval je, jako obvykle, na šálek čaje.

A všechno bylo zase jako dřív. Mág měl svou temnotu, hvozd měl svého pána a dobrodruhové celé Škytánie měli opět kam chodit za hrdinstvím … a na čaj.

20-5-5 Roztržitý matematik 15

Milí řešitelé a řešitelky.



Všechno, co má začátek, má i svůj konec. A tak se i nám pomalu blíží konec jubilejního 20. ročníku KSP. Ale ještě než se stane nevyhnutelné, můžete si vyřešit poslední praktickou úložku. Je taková … ze života.

Způsob odevzdávání a všechny ostatní detaily zůstávají stejné jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Roztržitý matematik tráví většinu svého času ve své malé pracovně na Karlíně. Po stole, po zemi, po stěnách a občas i po stropě se povalují nejrůznější papíry s nedokončenými výpočty, rozečtenými články a sem tam se objeví i seznam s nákupem nebo lísteček z čistírny. Není divu, že se matematikovi těžce pracuje, když neustále něco hledá …

Všechny matematikovy papíry (včetně obalů od svačiny) jsou očíslované. Matematik má také svůj odkládací systém, ve kterém se sice nevyznáme, ale pro zjednodušení budeme předpokládat, že všechny papíry leží v řadě za sebou. Když matematik nějaký papír použije, vyndá jej z řady, chvíli do něho údivně zírá mumlaje si pod vousy nesrozumitelné věci, načež tento papír položí na začátek řady (ostatní papíry se posunou). Na začátku matematikovy práce to šlo pěkně, neboť všechny papíry byly seřazeny podle čísel $(1, 2, \ldots N)$. Teď už jsou ale hodně přeházené a matematik nemůže najít ani svoji tramvajenku. Naštěstí si ještě pamatuje, kolikátý od začátku řady byl každý papír, se kterým pracoval. A v tomto okamžiku nastupujete do vzniklého chaosu vy, abyste matematika zachránili před jistou smrtí vyčerpáním.

Ve vstupním souboru `papiry.in` jsou na prvním řádku dvě čísla N a K, kde N $(1 \leq N \leq 500000)$ představuje počet papírů a K $(1 \leq K \leq 500000)$ počet operací, které matematik udělal. Na druhém řádku je posloupnost K čísel, kde každé číslo x_i představuje i-tou operaci, při které matematik vzal x_i-tý papír od začátku řady a posunul ho na první místo. Před započetím všech operací byly papíry seřazeny vzestupně od 1 do N.

Výstup uložte do souboru `papiry.out` tak, že na prvním řádku bude N čísel představujících permutaci dokumentů po provedení všech K operací.

Příklad:

`papiry.in`

```
8 3
5 1 4
```

`papiry.out`

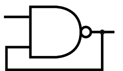
```
3 5 1 2 4 6 7 8
```

20-5-6 Hrady, hrádky, hradla 12 bodů

Milí řešitelé,

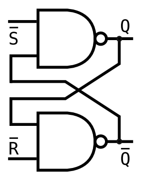
dnes bude naše povídání tak trošku z jiného soudku. V celém seriálu se naše obvody chovaly kombinačně, což znamená, že nebyly závislé na předchozím stavu a na jeden vstup, který si třeba můžeme představit jako celé číslo zapsané binárně, jsme s jistotou dostali vždy stejný výstup (jiné celé číslo zapsané binárně). Nyní se nám věci začnou komplikovat, neboť naučíme obvody „pamatovat si" a díky tomu výstup obvodu nebude nutně záležet pouze na vstupu, ale výsledek může ovlivnit i vnitřní stav obvodu. Vnitřní stav obvodu je to, co si obvod pamatuje z minulých vstupů, tedy obvod může vydat jiný výstup na stejný vstup, pokud jsme posloupnost zkoušených vstupů přeházeli. U skutečných obvodů je počáteční stav, tedy stav po zapnutí přístroje nedefinovaný (převážně díky fyzikálním efektům). My si pro jednoduchost počáteční stav, tedy stav na začátku výpočtu, sami nadefinujeme.

Například by takový obvod mohl počítat průběžnou paritu, na vstupu by byla buď jednička nebo nula, a na výstupu parita aktuální části binárního čísla reprezentovaného posloupností bitů na vstupu. Než se ale do takového obvodu pustíme, musíme vyřešit jeden drobný problém a to, jak má takový obvod poznat dva po sobě jdoucí jedničkové bity (rozmyslete si, že po sobě jdoucí nulové bity v tomto případě nemění výsledek a proto je nemusíme umět rozlišit.) Problém se řeší jednoduše, přidáme si do vstupu další bit, který se bude měnit při každém novém vstupu, tedy tři po sobě jdoucí jedničky budou vypadat třeba jako $10, 11, 10$. V elektronice se podobný signál obvykle označuje jako hodinový (Clock), s tím rozdílem, že reálně se za změnu vstupu považuje chvíle, kdy se hodinový signál přehoupne z nuly na jedničku (náběžná hrana). V následujícím textu budeme hodinový signál považovat za aktivní na náběžné hraně. Problém jsme vyřešili, tedy nechť si obvod na začátku pamatuje, že průběžná parita, tj. parita již načtené části čísla je nula. Pak obvod pro každou nulu na vstupu pošle zapamatovanou paritu na výstup, a pro každou jedničku provede negaci zapamatované dosavadní parity a tuto hodnotu pošle na výstup. Je jistě vidět, že se obvod pro jedničku na vstupu chová různě a rozhoduje se podle vnitřního stavu.

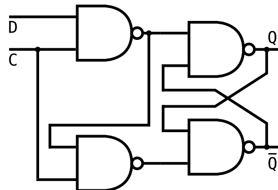


Nebudeme už dále chodit kolem horké kaše a ukážeme, jak se taková paměť vytvoří. Musíme vymyslet, jak uchovat nějakou hodnotu. Když vezmeme hradlo NAND a zapojíme jeho výstup na jeden ze dvou vstupů, dostaneme obvod, který si umí zapamatovat, že na vstupu byla jednička. Nechť je výstup nastaven na jedničku, pak jeden ze vstupů je nastaven také na jedničku a obvod v tomto stavu vydrží, dokud nenastavíme druhý vstup hradla na jedničku. Tím se výstup hradla přepne a bude již mít trvale na výstupu nulu.

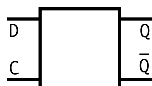
Takové hradlo je sice velice zajímavé, i když pramálo užitečné, nám by se hodilo umět jednak výstup nastavit, ale i vynulovat. Vezmeme tedy hradla NAND dvě a zapojíme výstup jednoho na vstup druhého a naopak. Takové zapojení se jmenuje klopný obvod RS. Funguje jednoduše, máme dva vstupy. Vstup Set, který nastavuje výstup na jedničku a vstup Reset, který nastavuje Výstup na nulu (odtud se také vzalo RS v pojmenování). Vstupy jsou negované, tedy považujme nezapojený vstup, za vstup na kterém je jednička. Připojením právě jednoho vstupu na nulu se buď obvod přepne, nebo neudělá nic (to záleží na vnitřním stavu). Výstup je na výstupu hradla označen písmenem Q, zatímco Q s pruhem je jeho negace.



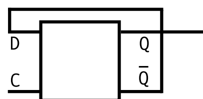
Odtud není daleko ke klopnému obvodu D, který je základem všech paměťových obvodů. Narozdíl od klopného obvodu RS je řízen hodinovým signálem. Funguje tak, že se vstup D „zkopíruje“ na výstup Q, v okamžiku, kdy se na hodinovém vstupu nastaví jednička. V řeči elektroniky bychom řekli, že se vstup zapíše na výstup na náběžné hraně hodinového signálu. Na následujícím obrázku je klopný obvod typu D, který ovšem nereaguje na náběžnou hranu, ale kopíruje vstup D, na výstup Q, když je vstup C nastavený na jedničku.



Obvod který reaguje na náběžnou hranu je o něco složitější a proto ho budeme kreslit následující schématickou značkou:



V sekvenčních obvodech se často využívá zpoždění na hradle, které nás v předchozích úlohách trápilo. Pro nás je teď důležité, že signál projde hradlem pomaleji než drátem (rozumně krátkém, kdybychom natáhli drát kolem země, bude samozřejmě signál hradlem procházet rychleji, nehledě na to, že se v tak dlouhem drátu po cestě ztratí). To znamená, že když za sebe zapojíme dvě hradla NOT, čímž dostaneme původní signál, a vedle natáhneme drát zapojený do téhož vstupu, bude na výstupu těchto hradel jednička ještě chvíli poté, co na vedlejším drátu bude už nula a naopak. S tímto efektem a klopným obvodem D lze vyrobit takzvanou děličku. To je obvod, jenž pro vstupní signál, kde se pravidelně střídají jedničky a nuly (hodinový signál) vyrobí signál, kde se pravidelně střídají dvě jedničky a dvě nuly (dělí frekvenci v původním signálu dvěma).



Vaším úkolem bude vymyslet:

1) zmíněný obvod na průběžnou paritu [5 bodů]

2) čítač, to jest obvod, který má na vstupu hodinový signál a postupně s každou jedničkou na vstupu zvýší hodnotu na výstupu (reprezentovanou binárním N-bitovým číslem) o jedna. [7 bodů]

Recepty z programátorské kuchařky

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

Binární vyhledávání. Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho x_m) a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

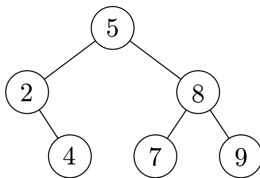
```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;   { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N\rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být k nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $\mathcal{O}(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do $\mathcal{O}$-čka.“]

Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

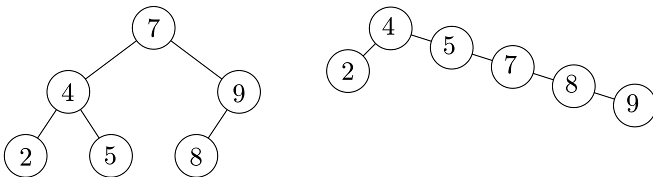
Zkusme ale provést jednoduchý myšlenkový pokus:

Vyhledávací stromy. Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání a, jak za chvilku uvidíme, mnohých dalších operací.

Definice. Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
     vrchol = record
       l, r : pvrchol;  { levý a pravý syn }
       x    : integer;  { hodnota }
     end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil.*

Find. V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
  end;
  TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $\mathcal{O}(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Insert. Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil.* A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

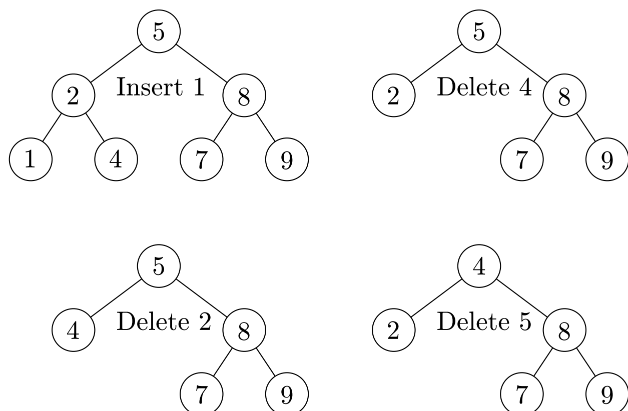
```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení,
  vrátí nový kořen. }
begin
  if v=nil then begin
    { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then  { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then      { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

Delete. Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
```

```
  if v=nil then exit          { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin                  { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin
      TreeDel := nil;         { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.r;        { jen pravý syn }
      dispose(v);
    end else if v^.r=nil then begin
      TreeDel := v^.l;        { jen levý }
      dispose(v);
    end else begin            { má oba syny }
      w := v^.l;              { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
      v^.x := w^.x;           { prohazujeme }
      { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $\mathcal{O}(h)$. Ale pozor, jejich používáním může h nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy h dosáhne až N.

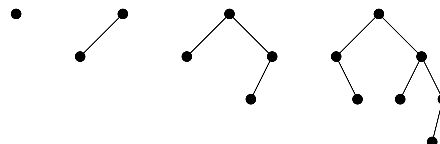
Procházení stromu. Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

```
procedure TreeShow(v:pvrchol);
begin
  if v=nil then exit;  { není co dělat }
  TreeShow(v^.l);
  writeln(v^.x);
  TreeShow(v^.r);
end;
```

Vyvážené stromy. S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy. Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $\mathcal{O}(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

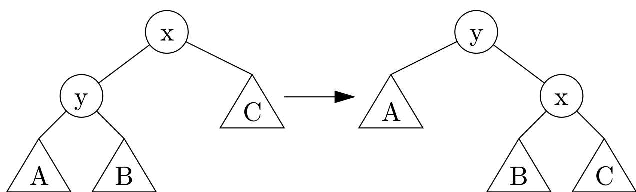
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = \mathcal{O}(\log N)$. *Q.E.D.*

AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně –

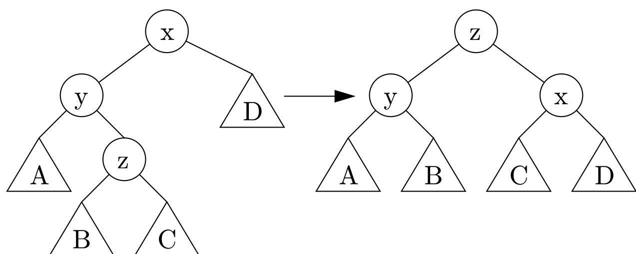
stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

Rotace. Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace. Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka. Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom hlubší a + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

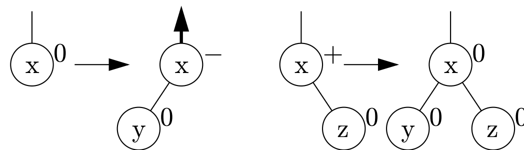
Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

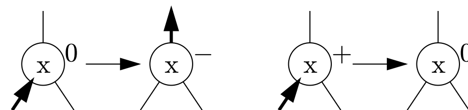
Vyvažování po Insertu. Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

Nejprve přidání listu samotné:

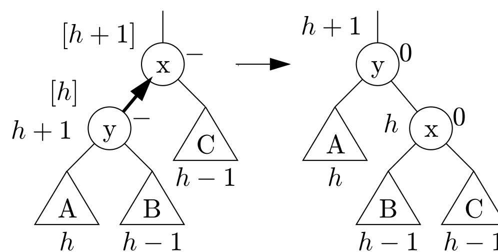


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

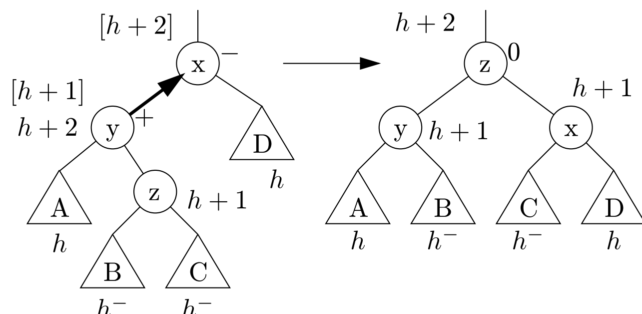


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

Další možnost je y jako $\oplus$:

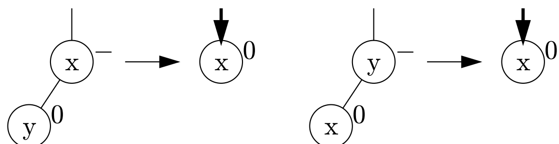


Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku.

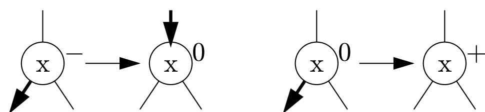
Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

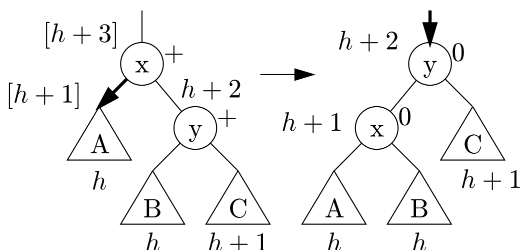
Vyvažování po Deletu. Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

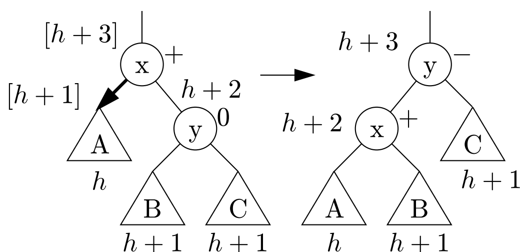


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



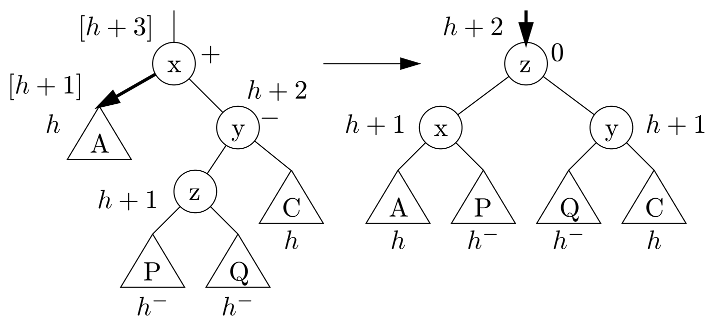
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end. Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů. AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.)
- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název), a navíc přidáme pravidlo, že všechny listy jsou na téže hladině. Hloubka vyjde opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

  Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $\mathcal{O}(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $\mathcal{O}(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných). To u většiny použití stačí – datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromad – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často

hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.

- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0,1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně, pokud tedy jsou všechny váhy navzájem různé, což skoro jistě jsou). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $\mathcal{O}(\log N)$.
- *BB-α stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $\mathcal{O}(\log N)$ na operaci.

Cvičení. Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-α stromy
3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $\mathcal{O}(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a,b\rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase …

Několik poznámek na závěr.

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP (nebo se můžete podívat do kuchařky u 4. série 19. ročníku). Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adeľsona-Veľského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Dnešní menu vám servírovali
Martin Mareš a Tomáš Valla

Vzorová řešení druhé série dvacátého ročníku KSP

20-3-1 Platba koně

Napřed provedeme malý trik. Každá hodnota je s přesností na 2 desetinná místa. Když všechny hodnoty vynásobíme 100 (tedy, převedeme z Korun na Haléře), dostaneme celá čísla, se kterými se mnohem lépe pracuje. Mimo to, ne každý handlíř umí pracovat s desetinnými čísly.

Nyní si na chvíli představme, že handlíř je naprosto chudý a nemá ani vindru (tedy, jeho hromádka na vracení je prázdná). Mimo to, je rozumné mít pouze kladné mince a kladnou cenu koně (i když, u některých koní…).

Jak by se dal řešit takový problém? Půjdeme na to od lesa. Rozdělíme to na fáze a chceme po k-té fázi vědět, které všechny obnosy lze zaplatit pomocí k prvních mincí.

Stav na začátku, tedy po nulté fázi, je jasný. Dokážeme zaplatit jedinou částku a to 0.

V každé fázi vezmeme minci o hodnotě h a vedle každé částky zaplatitelné pomocí $k-1$ prvních mincí přidáme do naší množiny ještě částku zvětšenou o h.

Z tohoto již lze snadno poznat, že daná částka jde zaplatit. Jednoduše se bude vyskytovat v množině. Jak ale poznat, kterými mincemi ji máme zaplatit? Při přidávání každé částky do množiny si k ní připíšeme také, ze které částky vznikla. Odečtením získáme poslední použitou minci a když se podíváme na onu předchozí částku, tak můžeme obdobným způsobem zrekonstruovat předchozí minci, až se dostaneme na začátek.

Musíme vyřešit, jak budeme reprezentovat tuto množinu. Všimneme si, že veškeré hodnoty jsou celá čísla a proto i jejich součty budou celá čísla. Navíc, nikdy nebude menší než 0 a větší, než součet všech Vildových mincí s_v. Můžeme použít pole, jehož indexy budou čísla $0 \ldots s_v$, v každé fázi toto pole projít a poznamenat do něj hodnoty nové.

Dále je třeba zajistit, aby námi vybraná možnost byla ta, která má nejméně použitých mincí. Inu, zavedeme tuto podmínku do každé fáze, tedy na konci k-té fáze budeme mít všechny zaplatitelné obnosy a každý na nejmenší počet mincí. Když budeme chtít poznamenat, že lze zaplatit nějaká hodnota a tato hodnota již zaplatit jde, tak ji přepíšeme na novou jen v případě, že je tato nová na méně mincí. Zřejmě to funguje, neboť ta s nejmenším počtem možností buď používá k-tou minci a nebo nepoužívá, což je přesně to, co

ověřuje tato podmínka.

Poslední problém, který je třeba vyřešit, jsou handlířovy mince. Jednoduchým trikem je sesypeme do stejného pytle jako mince Vildovy, jen je všechny vynásobíme číslem -1. Tento trik vypadá jednoduše, ale je třeba vyřešit několik detailů. Hlavním z nich je rozsah pole. Již není pravda, že by součet některých mincí nemohl být záporný. Avšak, když vyzkoušíme napřed všechny kladné mince a potom nám už zbudou jen záporné, tak zaznamenávat, že umíme zápornou částku k ničemu není, neboť ji již nikdy nad nulu nedostaneme.

Stejně tak si rozmyslíme horní hranici. Určitě se nikdy nedostaneme nad součet všech Vildových mincí. Ale také nemá nikdy cenu ukládat částky, které přesahují cenu koně a součet handlířových mincí dohromady, takové částky již nedokážeme dostatečně snížit, i kdyby handlíř vrátil vše, co měl. Stačí tedy vzít minimum z těchto dvou hodnot.

Označme toto minimum jako μ. Potom paměťová složitost je očividně $\mathcal{O}(\mu + M + N)$, kde N a M jsou počty mincí na jednotlivých hromádkách. Časová složitost je $\mathcal{O}(\mu \cdot (N + M))$, neboť pro každou minci proběhne jedna fáze a v každé fázi projdeme celé pole.

Michal „vorner“ Vaner

20-3-2 Dva lupiči

Máme čtyři výroky lupičů, a každý z nich nějakým způsobem omezuje možné dvojice čísel. Projdeme si tyto omezující podmínky jednu po druhé. Označme si velitelova čísla jako x, y; víme, že $2 \leq x, y \leq 99$.

První věta nám říká, že součin xy jde rozložit na součin dvou čísel více než jedním způsobem. To znamená, že xy je součin alespoň tří prvočísel, která nejsou všechna stejná. Označme A_1 množinu všech těchto povolených součinů.

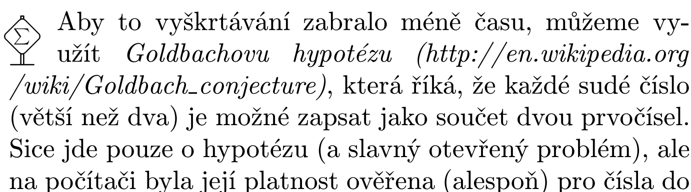
Druhá věta nám říká, že ať součet $x + y$ rozložíme na součet dvou čísel jakkoliv (čímž získáme řekněme čísla $\alpha, \beta \in \langle 2, 99 \rangle$, kde $\alpha + \beta = x + y$), vždycky bude součin $\alpha\beta$ ležet v množině A_1. Množinu všech součtů, které toto splňují, označme B_1.

Z třetí věty víme, že součin xy jde rozložit na součin dvou čísel α, β tak, aby $\alpha+\beta \in B_1$, právě jedním způsobem. Množinu všech součinů, které tuhle podmínku splňují, označíme A_2.

A poslední věta nám říká, že součet $x + y$ jde rozložit na součet dvou čísel α, β tak, aby $\alpha\beta \in A_2$, právě jedním způsobem. Množinu všech součtů, které to splňují, označíme B_2.

Chceme teď najít všechna čísla x, y taková, že $xy \in A_1 \cup A_2$ a $x + y \in B_1 \cup B_2$. Tím jsme úspěšně formulovali problém matematicky, ale jak ho vyřešit? Inu, pomocí čtyřech uvedených podmínek vyškrtáme zakázané součiny a součty a uvidíme, co zbude.

Povolené součty jsou v intervalu $\langle 4, 198 \rangle$, a chceme z nich vyškrtat ty, které se dají zapsat jako součet dvou prvočísel, nebo jako součet prvočísla a jeho druhé mocniny.



Aby to vyškrtávání zabralo méně času, můžeme využít *Goldbachovu hypotézu (http://en.wikipedia.org /wiki/Goldbach_conjecture)*, která říká, že každé sudé číslo (větší než dva) je možné zapsat jako součet dvou prvočísel. Sice jde pouze o hypotézu (a slavný otevřený problém), ale na počítači byla její platnost ověřena (alespoň) pro čísla do 10^{18}. Takže nám stačí škrtnout všechna sudá čísla, a z lichých ta, co jsou o 2 větší než nějaké prvočíslo. A součet prvočísla a jeho druhé mocniny je vždycky sudý.

Pozor, je tu jedna záludnost. Potřebujeme, aby ta dvě prvočísla byly přípustné hodnoty čísel x, y, tedy aby byla v intervalu $\langle 2, 99 \rangle$. Může se nám stát, že číslo sice rozložíme na součet dvou prvočísel, ale pokud jedno z těch prvočísel bude moc velké, stále se jedná o přípustný součet. Na druhou stranu součiny jako $99 \cdot 99$ přípustné nejsou, i když jde o součin alespoň tří prvočísel, která nejsou všechna stejná.

Teď projdeme všechny dvojice čísel z intervalu $\langle 2, 99 \rangle$ (možné součiny). Rovnou škrtneme součin dvou prvočísel a třetí mocninu prvočísla. A nakonec projdeme všechny dělitele d možného součinu s a podíváme se, jestli existuje právě jeden dělitel d tak, že součet $d + s/d$ je v množině B_1 povolených součtů.

A zbývá aplikovat poslední podmínku. Projdeme si množiny povolených součtů a pro každý z nich si najdeme všechny jeho rozklady na součet dvou čísel α, β z intervalu $\langle 2, 99 \rangle$. Pokud mezi všemi rozklady existuje právě jeden, pro který je součin $\alpha\beta$ povolený (nevyškrtnutý), našli jsme řešení.

Pro ruční procházení je těch možností docela hodně, a tak se vyplatilo napsat si program. Pro výpočetní sílu počítače je však jejich počet nepatrný, a proto dáme přednost jednoduššímu kódu před optimalizacemi pomocí prvočísel.

Nuže konečně uvedu dlouho očekávaný výsledek, úlohu řeší právě dvojice čísel $\{4, 13\}$.

Petr Kratochvíl

20-3-3 Cesta z kopce

Úloha jde nelépe vyřešit, jak si drtivá většina z vás všimla, v čase $\mathcal{O}(N)$. Budeme hledat nejdelší podposloupnost končící nějakým členem posloupnosti A, která splňuje zadání (v dalším textu podposloupnost znamená podposloupnost splňující zadání, tedy obsahující nejvýše k stoupání). Začneme s podposloupností obsahující jen zvolený prvek a její začátek postupně posunujeme směrem k počátku posloupnosti A. Dokud podposloupnost obsahuje méně než k stoupání, je všechno v pořádku. Jakmile ji ale rozšíříme tak, že už má právě k stoupání, musíme pokračovat opatrně, abysme nepřidali další stoupání. Skončíme tedy těsně za nějakým stoupáním (na druhém prvku z rostoucí dvojice), které by už překročilo limit, případně na začátku celé posloupnosti A. Pokud takto najdeme nejdelší podposloupnost pro každý prvek z A, bude mezi nimi určitě hledaná celkově nejdelší. Obdobnou úvahou při posunování konce podposloupnosti místo začátku zjistíme, že konec hledané podposloupnosti bude těsně před nějakým stoupáním, případně na konci celé posloupnosti.

Když obě úvahy spojíme dohromady, zjistíme, že není třeba hledat začátek a konec podposloupnosti jinde než u stoupání a na úplném začátku nebo konci. Najdeme si tedy všechna stoupání v posloupnosti A a pro každé z nich hledáme nejdelší podposloupnost, která končí těsně před ním. Pokud právě zkoumané stoupání, bude i-té, tak pro začátek podposloupnosti musíme přeskočit k stoupání a bude tedy ležet hned za $(i - k - 1)$-tým. Hodnoty $i \leq k$ vůbec nemusíme uvažovat, protože u nich končící podposloupnosti budou začínat prvním prvkem A, stejně jako pro $i = k + 1$, pro nějž bude délka větší než pro všechna menší i.

Z výše uvedeného vyplývá, že pro zjištění výsledku si vůbec nemusíme pamatovat hodnoty A, kromě dvou posled-

ních k zjištění stoupání. Navíc stačí znát jen $k+1$ stoupání před aktuálním, jelikož stoupání $k+1$ míst před právě zkoumaným potřebujeme v tomto kroku a následující budeme potřebovat v dalších krocích. Hledání nejdelší posloupnosti pak může probíhat přímo při hledání stoupání. Nejdříve si zapamatujeme prvních $k+1$ stoupání a pak vždy když najdeme další, zkontolujeme délku podposloupnsti, která u něj končí a začíná u nejstaršího stoupání, které si pamatujeme, což je právě to $k+1$ míst před aktuálním. Toto nejstarší stoupání pak zapomeneme a zapamatujeme si právě nalezené. Taková datová struktura se nazývá fronta. Nakonec si ještě musíme dát pozor, pokud je počet stoupání nejvýše k, abychom za řešení přijali celou posloupnost.

Časová složitost je $\mathcal{O}(N)$, protože procházíme jedenkrát zadanou posloupnost a pro každý její prvek provádíme konstantně mnoho operací. Paměťová složitost je $\mathcal{O}(k)$, kvůli frontě délky $k+1$.

Petr Onderka

20-3-4 Orientace na mapě

Nejprve si nejspíš uvědomíme, že v acyklickém orientovaném grafu musí být alespoň jeden vrchol, do kterého nevede žádná hrana – zdroj. Z každého vrcholu (které není zdroj) můžeme cestou proti směru hran dojít do nějakého zdroje. Proto při hledání vrcholů, mezi nimiž vede nejvíce cest,můžeme předpokládat, že počáteční vrchol je zdroj – kdyby cesty vycházely z jiného vrcholu, můžeme všechny prodloužit až do nějakého zdroje. Tím se jejich počet určitě nezmenší. (Z podobného důvodu bychom také mohli hledat koncové vrcholy pouze ve stocích – vrcholech z nichž nevede žádná hrana.)

Vzápětí si uvědomíme, že zdrojů v grafu může být mnoho, takže nám tohle pozorování práci neušetří a algoritmus nezlepší, ale využít ho můžeme… Z každého zdroje tedy spočítáme cesty do jednotlivých vrcholů.

Máme-li pro nějaký vrchol v spočíst počet cest z určitého zdroje, lze to udělat tak, že sečteme počty cest do všech vrcholů, ze kterých vede hrana do v. K tomu ovšem musíme tyto počty cest znát. Proto je nutné počítat cesty do vrcholů ve správném pořadí – v topologickém pořadí (o němž se píše v kuchařce ke třetí sérii). Topologické pořadí určíme například tak, že projdeme graf ze zdroje do hloubky a pamatujeme se pořadí, ve kterém jsme naposled opouštěli jednotlivé vrcholy - tímto způsobem je dostaneme setříděné pozpátku – zdroj bude úplně poslední, takže musíme počítat počty cest do vrcholů od konce. Když máme spočítané cesty do všech vrcholů, zapamatujeme si maximální počet cest (a kam vedly) a prozkoumáme cesty z dalšího zdroje. Pak už stačí jenom vybrat zdroj, z něhož vede nejvíce cest.

Jak to všechno bude složité? Na jednotlivé průchody do hloubky potřebujeme $O(N+M)$ času. Počet potřebných průchodů závisí na počtu zdrojů v grafu, může být až $O(N)$. Celkem se dostáváme na časovou složitost $O(N\cdot(N+M))$. Paměťová složitost vzorového řešení je $O(N^2)$, protože si seznamy sousedů pamatuje ve zbytečně dlouhých polích, při šikovnějším způsobu by stačilo $O(N+M)$.

Tereza Klimošová

20-3-5 Asfaltování

Zdravím všechny řešitele upatlané od asfaltu. Mám pro vás dobrou zprávu: Nebojte, za pár měsíců se to oloupe. A nyní vám přinášíme exkluzivně algoritmus od samotného vrchního cestáře, který nám jej (samozřejmě pod pohrůžkou mučení) vyzradil.

Nejprve si naši úlohu převedeme do řeči grafů. Města představují vrcholy, cesty mezi nimi budou hrany, a protože lze cestovat po celé Hipopotámii, bude graf souvislý. Chceme najít párování hran takové, aby každá hrana byla spárovaná a dvě spárované hrany měly společný vrchol. Nedá mnoho přemýšlení, že zmíněné párování nemůže existovat, pokud je počet hran lichý. Od teď se tedy budeme zabývat pouze grafy, které mají sudý počet hran.

Klíčem k řešení našeho problému bude algoritmus prohledávání do hloubky, též známý jako DFS (Depth First Search). Podívejme se, jak bude vypadat situace vstoupíme-li při procházení do vrcholu u. Nejprve zpracujeme všechny dosud nenavštívené sousedy vrcholu u, jak nám káže algoritmus DFS. Následně se podíváme, s kolika nespárovanými hranami vrchol inciduje. Je-li jich sudý počet, můžeme spárovat hrany libovolně mezi sebou. Pokud jich je lichý počet, necháme hranu vedoucí k otci (k vrcholu, ze kterého jsme do u přišli při DFS) nespárovanou (taťka se nám o ni postará) a zbývající hrany, kterých už je sudě, opět libovolně spárujeme.

Zbývá ukázat, že u vrcholu s, ve kterém jsme prohledávání započali, budeme mít na konci algoritmu sudý počet nespárovaných hran. Kdyby tomu tak nebylo, měli bychom problém, protože s už nemá žádného „taťku“, který by mu pomohl vyřešit problémy s lichou hranou. Naštěstí ale víme, že na začátku máme sudý počet (nespárovaných) hran. Při párování nám ubývají nespárované hrany vždy po dvou, takže se zachovává jejich sudý počet. V okamžiku, kdy se vrátíme v DFS zpět do vrcholu s, můžou být nespárované pouze hrany incidující s s. A protože jich je celou dobu sudý počet, můžeme je vesele spárovat.

Budeme-li reprezentovat graf seznamem sousedů, je časová i paměťová složitost algoritmu $\mathcal{O}(N+M)$, kde N je počet vrcholů a M je počet hran.

Přeji vám pěkný den a pokud možno hladké asfaltové cesty.

Martin „Bobřík“ Kruliš

20-3-6 Hrady, hrádky, hradla

V minulé části seriálu jste měli za úkol vymyslet obvod, který zjistí, zda-li je číslo na vstupu dělitelné třemi. Než začneme s vymýšlením obvodu, podíváme se na to, jaké zbytky po dělení třemi dávají číslice ve dvojkovém zápisu.

pozice číslice	0	1	2	3	…
hodnota desítkově	2^0	2^1	2^2	2^3	…
modulo třemi	1	2	1	2	…
zapsáno dvojkově	1	10	1	10	…

Vidíme, že se zbytek opakuje periodicky a pro liché pozice dostáváme zbytek $1_{10} = 01_2$ a pro sudé zbytek $2_{10} = 10_2$. Formálně lze toto pozorování dokázat indukcí, pro liché pozice dostáváme $n_0 = 2^0 = 1$, $n_k = 2^{(2k+1)}, n_{k+1} = 2^{(2k+3)}$, pak $n_{k+1} = 4\cdot 2^{(2k+1)} = 3\cdot 2^{(2k+1)} + 2^{(2k+1)}$ Vidíme tedy, že pro další číslici platí, že je součtem něčeho krát tři, což má jistě zbytek po dělení třemi nula, plus předchozí číslice, aplikováno "rekurzivně" dostáváme, že všechny číslice na lichých pozicích mají stejný zbytek modulo třemi. Pro sudé pozice je důkaz podobný. A teď už dost formalismu a pojďme se podívat dál.

Vidíme, že když si budeme brát vstup po dvojicích, ty sečteme, pak bude toto číslo dělitelné třemi právě tehdy když bylo dělitelné třemi číslo původní. Což odpovídá tomu, že se-

čteme zbytky na lichých pozicích plus zbytky na sudých pozicích krát dva $a = a_0 + 2 \cdot a_1 + 4 \cdot a_2 + \ldots + 2^{n-1} \cdot a_{n-1} + 2^n \cdot a_n$, pak součet zbytků po dělení třemi je $S = a_0 + 2 \cdot a_1 + a_2 + 2 \cdot a_3 + \ldots + a_{n-1} + a_n = a_0, a_1 + a_2, a_3 + \ldots + a_{n-1}, a_n$, kde a, b znamená binární číslo poskládané z binárních číslic a a b, neboť ve dvojkovém zápisu je násobení dvěma posunutí doleva (obdobně jako násobení desíti v soustavě desítkové).

Teď nám už zbývá jenom vymyslet obvod, který sečte dvě dvoubitová čísla modulo třemi. Číslo 00 má stejný zbytek po dělení třemi jako 11. Sčítání je komutativní a proto nám nezáleží na pořadí sčítání, tato operace je tedy jednoznačně určena následující tabulkou. Značka „$\equiv$" zde znamená, že čísla mají stejný zbytek po dělení třemi.

Vstup A	Vstup B	Výstup
01	01	10
10	10	01
01	10	$00 \equiv 11$
01	$00 \equiv 11$	01
10	$00 \equiv 11$	10
$00 \equiv 11$	$00 \equiv 11$	$00 \equiv 11$

Když se na tuto operaci pozorně podíváme, zjistíme, že se nápadně podobá obyčejnému sčítání, až na to, že se přenos znovu přičte k výsledku.

Takže máme obvod, který má na vstupu čtyři bity, dvě dvoubitová čísla a na výstupu dva bity, jedno dvoubitové číslo. Nyní stačí tyto obvody poskládat tak, že na každé výpočetní hladině zmenšíme počet dvoubitových zbytků na polovinu. Vstup jako obvykle doplníme dostatečným počtem nul. Číslo pak bude dělitelné třemi, když nám na konci zbyde 11 nebo 00. Jelikož obvod na sčítání má konstantní hloubku má celé zapojení asymptoticky logaritmickou složitost stejně jako obvod na počítání parity z předchozího seriálu.

Rozmyslete si, že pro dělitelnost třemi v zápise BCD bude fungovat stejný postup.

Cyril Hrubiš

Úloha 20-3-1 – Platba koně – program

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

typedef struct {
	int predchozi;
	int minci;
} castka_t;

#define INFINITY 0x4fffffff //Takto se pozná něco, co nejde zaplatit

void pridej( castka_t castky[], int mince, int index, int limit ) {
	int nova = index + mince;
	if( nova < 0 || nova > limit ) //mimo rozsah
		return;
	if( castky[ index ].minci + 1 < castky[ nova ].minci ) {
		castky[ nova ].predchozi = index;
		castky[ nova ].minci = castky[ index ].minci + 1;
	}
}

int main( int argc, const char *argv[] ) {
	int n, m, p, vil_celkem, han_celkem;
	float prep; //Dočasné, na nehaléřové hodnoty
	scanf( "%d%d%f", &n, &m, &prep );
	p = ( int ) ( prep * 100 ); //Převod na haléře
	int mince[ m + n ];
	vil_celkem = han_celkem = 0;
	for( int i = 0; i < m + n; ++ i ) {
		float nova;
		scanf( "%f", &nova );
		nova *= 100; //Převod na haléře
		mince[ i ] = ( int ) nova;
		if( i >= n ) {
			han_celkem += mince[ i ];
			mince[ i ] *= -1; //Handléřovy mince "vracejí"
		} else {
			vil_celkem += mince[ i ];
		}
	}
	int limit = han_celkem + p; //Vybrat vhodný rozsah pole
	if( vil_celkem < limit )
		limit = vil_celkem;
	if( p > limit ) {
		printf( "Na to Vilda nemá\n" );
		return 0;
	}
	if( p < 0 ) {
		printf( "Nepodporujeme záporné koně\n" );
		return 0;
	}
	castka_t castky[ limit + 1 ];
	for( int i = 1; i <= limit; ++ i ) {
```

```
                castky[ i ].minci = INFINITY;
        }
        castky[ 0 ].predchozi = 0;
        castky[ 0 ].minci = 0;
        for( int i = 0; i < n; ++ i )
                //Opačně, abychom nenacházeli z této fáze
                for( int j = limit; j >= 0; -- j )
                        pridej( castky, mince[ i ], j, limit );
        for( int i = n; i < n + m; ++ i )
                //Záporné mince -> tímto směrem
                for( int j = 0; j <= limit; ++ j )
                        pridej( castky, mince[ i ], j, limit );
        if( castky[ p ].minci == INFINITY ) {
                printf( "Koně zaplatit nelze\n" );
                return 0;
        }
        while( p ) {
                printf( "%f\n", ( p - castky[ p ].predchozi ) / 100.0 );
                p = castky[ p ].predchozi;
        }
        return 0;
}
```

Úloha 20-3-1 – Platba koně – program v Haskellu :-)

```
-- zaplat vildova_hromádka handlířova_hromádka cena_koně
-- Nothing -- nedá se zaplatit
-- Just seznam_mincí -- čím zaplatit
zaplat :: [ Float ] -> [ Float ] -> Float -> Maybe [ Float ]
zaplat vilda handlir kun = if vysledek == Nothing then Nothing else
        let Just v = vysledek in Just $ map ( \m -> ( fromInteger $ toInteger m ) / 100.0 ) v
        where
                vildai = map zhaleruj vilda
                handliri = map zhaleruj handlir
                vysledek = vyres ( zhaleruj kun ) ( sum vildai ) ( sum handliri ) ( vildai ++ map ( (*) ( -1 ) ) handliri )

zhaleruj :: Float -> Int
zhaleruj m = round ( 100.0 * m )

polozka :: Int -> ( Int, Int, Int )
polozka i = ( i, 0, infinity )

vyres :: Int -> Int -> Int -> [ Int ] -> Maybe [ Int ]
vyres cena vil_sum han_sum mince =
        vyber cena $ foldl pridej ( ( 0, 0, 0 ):map polozka [ 1 .. min vil_sum $ han_sum + cena ] ) mince

infinity :: Int
infinity = 1000000

priplat :: ( ( Int, Int, Int ), ( Int, Int, Int ) ) -> ( Int, Int, Int )
priplat ( ( index, pred_puv, min_puv ), ( pred_nov, _, min_nov ) )
        | min_puv <= min_nov + 1 = ( index, pred_puv, min_puv )
        | True = ( index, pred_nov, min_nov + 1 )

pridej :: [ ( Int, Int, Int ) ] -> Int -> [ ( Int, Int, Int ) ]
pridej moznosti mince = [ m | m <- map priplat $ zip
        ( ( map polozka [ mince .. -1 ] ) ++ moznosti )
        ( ( map polozka [ 1 .. mince ] ) ++ moznosti ++ ( map polozka [ 1 .. ] ) ),
        let ( index, _, _ ) = m in index >= 0 ]

vyber :: Int -> [ ( Int, Int, Int ) ] -> Maybe [ Int ]
vyber 0 _ = Just []
vyber cena moznosti = if vysledek == [] then Nothing else Just $ mince:zbytek
        where
                vysledek = [ predchozi | ( index, predchozi, minci ) <- moznosti, index == cena, minci < infinity ]
                Just zbytek = vyber predchozi moznosti
                mince = cena - predchozi
                [ predchozi ] = vysledek
```

Úloha 20-3-2 – Dva lupiči – program

```
#include <stdio.h>

int vysledek_x, vysledek_y;

int a_nevi(int soucin) {
        int rozklad = 0;
        for (int i = 2; (i <= 99) && (i*i <= soucin); i++)
                if ((soucin % i == 0) && (soucin/i <= 99))
                        rozklad++;

        if (rozklad >= 2)
```

```
                return 1;
        else return 0;
}

int b_vi_ze_a_nevi(int soucet) {
        for (int i = 2; (i <= 99) && (2*i <= soucet); i++)
                if (!a_nevi(i*(soucet-i)))
                        return 0;
        return 1;
}

int a_uz_vi(int soucin) {
        int rozklad = 0;
        if (!a_nevi(soucin))
                return 0;
        for (int i = 2; (i <= 99) && (i*i <= soucin); i++) {
                if ((soucin % i == 0) && (b_vi_ze_a_nevi(soucin/i + i)))
                        rozklad++;
                if (rozklad > 1)
                        break;
        }
        if (rozklad == 1)
                return 1;
        else return 0;
}

int b_taky_vi(int soucet) {
        int rozklad = 0;
        if (!b_vi_ze_a_nevi(soucet))
                return 0;
        for (int i = 2; (i <= 99) && (2*i <= soucet); i++)
                if (a_uz_vi(i*(soucet-i))) {
                        rozklad++;
                        vysledek_x = i;
                        vysledek_y = soucet - i;
                }
        if (rozklad == 1)
                return 1;
        else return 0;
}

int main(void) {
        for (int i=2; i<=198; i++)
                if (b_taky_vi(i))
                        printf("(%d, %d)\n", vysledek_x, vysledek_y);
        return 0;
}
```

Úloha 20-3-3 – Cesta z kopce – program

```
program cestaZkopce;

const MaxK = 20;

var N, k, a, b, i, j, pred, akt: integer;
    stoupani: array [0..MaxK] of integer;

begin
 a := 1;
 b := 1;
 stoupani[0] := 1;
 j := 1;
 read(N, k);
 read(pred);
 for i:=2 to N do begin
  read(akt);
  if pred < akt then begin                      {máme stoupání}
   if j <= k then                               {ale zatím málo}
    stoupani[j] := i
   else begin                                   {teď už dost}
    if i-1 - stoupani[j mod (k+1)] > b - a then begin
{zbytek po dělení používám proto, aby pole stoupani bylo zatočené do kruhu}
     a := stoupani[j mod (k+1)];                {zatím nejdelší podposloupnost}
     b := i - 1;
    end;
    stoupani[j mod (k+1)] := i;                 {uložíme do fronty}
   end;
   inc(j);
  end;
  pred := akt;
 end;
```

```
 if N - stoupani[j mod (k+1)] > b-a then begin  {ještě zkontrolujeme poslední podposloupnost}
  if j <= k then a := 1                         {vezmeme celou posloupnost}
  else a := stoupani[j mod (k+1)];              {nebo jen od prvního stoupání z fronty}
  b := N;
 end;
write('a = ', a, ' b = ', b);
end.
```

Úloha 20-3-4 – Orientace na mapě – program

```
program mapa;


const C=6;
type vrchol = record
                hrany:array [1..C] of integer;
                deg:integer;
                zdroj:boolean;
                prosel:boolean;
                cesty:integer;
                end;

var
  graf:array [1..C] of vrchol;
  topol:array[1..C] of integer;
{cisla vrcholu v topologickem poradi}
  maxvrchol,maxhodnota:integer;{kde najdeme maximum a jaka je jeho hodnota}
  zdr:integer; {pocet zdroju}
  top:integer;{pocet vrcholu v topologickem usporadani z aktualniho zdroje}
  i,j,k,N,u,v:integer;

  zdroje:array [1..C] of record
                          start,cil,cesty:integer;
                          end;

procedure pruchod(i:integer);
  var m:integer;
  begin
    m:=1;
    if not graf[i].prosel then begin
     while graf[i].deg>=m do
        begin
                  writeln(i,graf[i].hrany[m]);
          pruchod(graf[i].hrany[m]);
          inc(m);
        end;
     graf[i].prosel:=TRUE;
         inc(top);
         writeln(top);
     topol[top]:=i;
         end;
  end;


begin
{nacteni}
   readln(N);
   for i:=1 to N do begin
     graf[i].deg:=0;
     graf[i].zdroj:=TRUE;
     end;
   readln(u,v);
   while u<>0 do begin
     inc(graf[u].deg);
     graf[v].zdroj:=FALSE;
     graf[u].hrany[graf[u].deg]:=v;
     readln(u,v);
     end;
{nalezeni zdroju}
   zdr:=0;
   for i:=1 to N do begin
     if graf[i].zdroj then begin
                              inc(zdr);
                              zdroje[zdr].start:=i;
                            end;
   end;
   writeln(1);
{pruchod do hloubky - topologicke trideni ze zdrojovych vrcholu}
    for i:=1 to zdr do begin
      for j:=1 to N do graf[j].prosel:=FALSE;
      top:=0;
```

```
      pruchod(zdroje[i].start);
      for j:=1 to N do graf[j].cesty:=0;
      graf[topol[top]].cesty:=1;
      for  j:= top downto 1 do begin
        v:=topol[j];
        for k:=1 to graf[v].deg do begin
          u:=graf[v].hrany[k];
          graf[u].cesty:=graf[u].cesty+graf[v].cesty;
          end;
        end;
     maxhodnota:=0;
      for j:=1 to N do if graf[j].cesty>maxhodnota
        then begin
          maxhodnota:=graf[j].cesty;
          maxvrchol:=j;
        end;
      zdroje[i].cil:=maxvrchol;
      zdroje[i].cesty:=maxhodnota;
      end;
          writeln('z=',zdr);
      maxhodnota:=0;
      for i:=1 to zdr do if zdroje[i].cesty>maxhodnota
        then begin
          maxhodnota:=zdroje[i].cesty;
          maxvrchol:=i;
        end;
      writeln('Mezi vrcholy ',zdroje[maxvrchol].start,' a ',zdroje[maxvrchol].cil,'vede ',zdroje[maxvrchol].cesty,'cest.');
      {for s:=1 to N do
        if cesty[s]>max then
          begin
            maxs:=s;
            maxz:=zdroje[i];
            max:=cesty[s];
          end;}
    readln;
end.
```

Úloha 20-3-5 – Asfaltování – program

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

#define MAXN    100000
#define MAXM    500000

#define SWAP_INT(x, y) { int tmp = x; x = y; y = tmp; }


/* Struktura uchovávající údaje o jedné hraně */
struct s_edge {
        int a, b;       // Mezi kterými vrcholy hrana vede.
        int pair;       // Se kterou hranou je spárovaná (-1, pokud ještě není).
};

/* Pocatek hran od daneho vrcholu v seznamu hran a stupen vrcholu */
struct s_vertex {
        int start;              // Index první hrany v seznamu hran (resp. poli "ep"), která inciduje s tímto vrcholem.
        int deg;                // Stupeň vrcholu.
        int visited;    // Zda již byl vrchol navštíven.
};


/* Globální proměnné */
struct s_edge edges[MAXM];              // Seznam hran (tak jak je načten ze souboru).
struct s_vertex vertices[MAXN]; // Seznam vrcholů.
int ep[2*MAXM];                                 // Přeuspořádání hran (aby bylo možné jednoduše držet seznamy sousedů).
int N, M;                                               // Počet vrcholů a hran.


/* Načte vstupní data a vytvoří reprezentaci grafu. */
void read_input(void)
{
        /* Otevřeme soubor */
        FILE *fp = fopen("asfalt.in", "r");
        if (!fp) {
                perror("Cannot open input file");
                exit(1);
        }

        /* Přečteme počet vrcholů a hran. */
```

```
	fscanf(fp, "%d %d", &N, &M);

	/* Ošetříme speciální případ, kdy je M liché a úloha tak nemá řešení. */
	if ((M % 2) == 1) {
		FILE *fout = fopen("asfalt.out", "w");
		if (!fout) {
			perror("Cannot open output file");
			exit(1);
		}
		fputs("no\n", fout);
		exit(0);
	}

	/* Načteme seznam hran */
	for (int i = 0; i < M; i++) {
		fscanf(fp, "%d %d", &edges[i].a, &edges[i].b);
		edges[i].a--;  edges[i].b--;    // V našem programu indexujeme města od 0 do N-1.
		edges[i].pair = -1;                          // -1 značí, že hrana ještě není spárovaná.
		vertices[edges[i].a].deg++;
		vertices[edges[i].b].deg++;
	}
	fclose(fp);

	/* Vybudujeme si index ep nad polem hran, díky kterému budeme mít jednoduchý seznam sousedů. */
	for (int i = 1; i < N; i++) {
		vertices[i].start = vertices[i-1].start+vertices[i-1].deg;
		vertices[i-1].deg = 0;
	}
	vertices[N-1].deg = 0;
	for (int i = 0; i < M; i++) {
		ep[ vertices[edges[i].a].start + vertices[edges[i].a].deg++ ] = i;
		ep[ vertices[edges[i].b].start + vertices[edges[i].b].deg++ ] = i;
	}
}


/* Vrátí druhý konec hrany edge incidující s vrcholem vertex. */
inline int edge_end(int vertex, int edge)
{
	if (edges[edge].a != vertex)
		return edges[edge].a;
	return edges[edge].b;
}


/* Uloží výsledky do výstupního souboru. */
void print_out(void)
{
	/* Otevřeme výstupní soubor. */
	FILE *fout = fopen("asfalt.out", "w");
	if (!fout) {
		perror("Cannot open output file");
		exit(1);
	}

	/* Vytiskneme spárované hrany. */
	for (int i = 0; i < M; i++)
		if (edges[i].pair > i) {
			int res[4] = { edges[i].a, edges[i].b, edges[edges[i].pair].a, edges[edges[i].pair].b };

			if ((res[0] == res[2]) || (res[0] == res[3]))
				SWAP_INT(res[0], res[1])

			if ((res[0] == res[3]) || (res[1] == res[3]))
				SWAP_INT(res[2], res[3])

			fprintf(fout, "%d %d %d\n", res[0]+1, res[1]+1, res[3]+1);
		}
	fclose(fout);
}


/* Prohledávání do hloubky na párování hran. */
void dfs(int actVertex, int parent)
{
	vertices[actVertex].visited = 1;        // Takhle označíme prohledávaný vrchol za navštívený.

	/* Zavolame se na vsechny sousedni vrcholy, ve kterych jsme dosud nebyli */
	for (int i = vertices[actVertex].start; i < vertices[actVertex].start + vertices[actVertex].deg; i++)
		if (!vertices[edge_end(actVertex, ep[i])].visited)
			dfs(edge_end(actVertex, ep[i]), actVertex);
```

```
        /* Nyní spárujeme zbylé hrany. */
        int pair = -1;
        for (int i = vertices[actVertex].start; i < vertices[actVertex].start + vertices[actVertex].deg; i++)
                if ((edges[ep[i]].pair == -1) && edge_end(actVertex, ep[i]) != parent) {
                        if (pair == -1)         // Zatím nemame hranu ke spárování...
                                pair = ep[i];
                        else {                          // Už na nás jedna nespárovaná hrana čeká.
                                edges[ep[i]].pair = pair;
                                edges[pair].pair = ep[i];
                                pair = -1;
                        }
                }

        /* Pokud jedna hrana zbyla plonková - spárujeme ji s hranou k rodiči. */
        if (pair != -1) {
                /* Bohužel hranu k otci musíme nejdřív najít... */
                int i = vertices[actVertex].start;
                while (edge_end(actVertex, ep[i]) != parent) i++;

                /* Index hrany k otci je teď uložen v i. */
                edges[ep[i]].pair = pair;
                edges[pair].pair = ep[i];
        }
}


/* Vstupní bod aplikace. */
int main(void)
{
        read_input();
        dfs(0, 0);
        print_out();
        return 0;
}
```

Výsledková listina dvacátého ročníku KSP po třetí sérii

		škola	*ročník*	*sérií*	*2031*	*2032*	*2033*	*2034*	*2035*	*2036*	*série*	*celkem*
1.	Peter Ondrúška	SPŠDubnica	4	3	12			10	11	8	43,4	129,2
2.	Jan Michelfeit	G HBrod	4	8			8	10	9	3	31,6	110,0
3.	Alena Skálová	GNaVPláni	4	4			8	8	3	12	34,4	109,9
4.	Filip Hlásek	GMikuláš	1	3		8	8	7		5	32,9	108,0
5.	Petr Malý	GSladkNám	4	3		7	8	6	2	8	34,3	99,2
6.	Libor Plucnar	GPBezruče	3	8	12		6	7	3		30,4	97,8
7.	Trung Ha duc	GMasaryk	2	8			6	8		12	27,2	91,7
8.	Vlastimil Dort	GŠpitálsPH	2	8			8	7	3	3	23,8	91,5
9.	Tomáš Toufar	G Bílovec	4	3		6			2	2	15,5	83,5
10.	Vojtěch Tůma	G Jihlava	4	7			6	9		5	22,7	81,3
11.	Filip Štědronský	GMikuláš	1	3			8			1	10,2	81,1
12.	Štěpán Weber	GBuďánka	3	2							0,0	78,9
13.	Pavel Veselý	G Strakon	3	10	1		5	9	2		17,7	72,9
14.	Stanislav Fořt	G Tábor	0	7			1	3			5,7	60,2
15.	Jan Škoda	GMikuláš	1	3			6				7,3	59,1
16.	Jitka Novotná	G Bílovec	3	2							0,0	58,6
17.	Lukáš Kripner	G Litvínov	2	6							0,0	57,7
18.	Jakub Hrnčíř	GFXŠaldyLI	1	3			4				6,0	55,9
19.	David Marek	SPŠ Zlín	4	4							0,0	49,2
20.	Petr Babička	G SvětláNS	3	3				6	2		12,3	43,2
21.	Radim Cajzl	G NMnMor	1	13						12	12,0	39,7
22.	Jan Matějka	GJírovco	3	4							0,0	35,4
23.	Jakub Kaplan	GJKTylaHK	4	16							0,0	34,4
24.	Pavel Kratochvíl	ZŠSvětlá	0	3		1				2	6,2	34,3
25.	Jiří Zárevúcky	SŠInformFM	3	1							0,0	32,5
26.	Tomáš Sýkora	G VKlobou	4	11			6				6,0	30,8
27.	David Brázdil	G Zlín	3	6							0,0	29,8
28.	Martin Vlach	G Jihlava	4	1							0,0	29,7
29.	Milan Rybář	GJJunman	3	1							0,0	29,3
30.	Karel Tesař	SPŠEPlzeň	2	2							0,0	28,4
31. – 32.	Jiří Keresteš	SPŠEPlzeň	2	4							0,0	28,2
	Petr Sokola	SPŠ Zlín	4	2							0,0	28,2
33.	Adam Streck	G Hořice	4	1							0,0	27,6
34.	Vojtěch Kolář	G Neratov	3	1							0,0	27,3
35.	Jakub Červenka	GŠpitálsPH	2	4			8				8,0	26,0
36.	Pavel Taufer	GArcibisk	2	1							0,0	25,7
37.	Martin Patera	GArabská	2	1							0,0	25,5
38. – 39.	Jakub Suchý	GMikuláš	1	2							0,0	24,1
	Jan Žák	G HBrod	3	6							0,0	24,1
40.	Alžběta Pechová	SPŠSVsetín	3	3							0,0	20,6
41.	Roman Smrž	GOhradní	4	1			8		11		19,0	19,0
42.	Jan Vaňhara	G Holešov	3	1	3		3	3			17,8	17,8
43.	Petr Holášek	G Příbor	4	5			6		2		10,5	10,5
44.	Nikolas Zigmund	ZŠHavířov	1	1							0,0	8,9
45.	Peter Smatana	EkoGLabsBO	3	1							0,0	6,4
46.	Tomáš Jakl	G MTřebová	4	3			3				5,0	5,0
47.	Miroslav Klimoš	G Bílovec	3	20							0,0	4,4
48.	Lukáš Timko	G Tábor	0	1	1	0,5					3,9	3,9