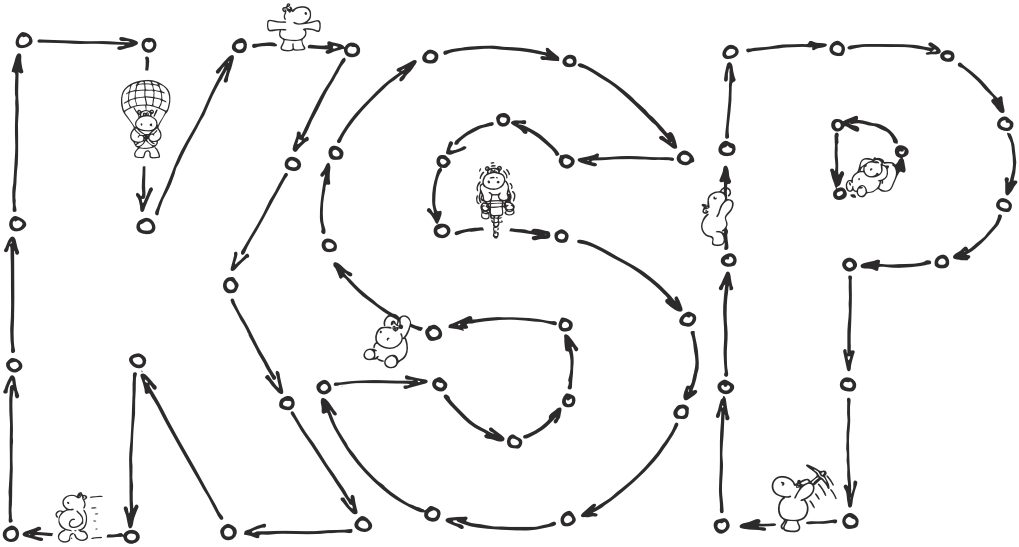
PETR KRATOCHVÍL A KOLEKTIV

Korespondenční seminář z programování

XXI. ročník – 2008/2009



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

PETR KRATOCHVÍL A KOLEKTIV

Korespondenční seminář z programování

XXI. ročník – 2008/2009

matfyzpress

Praha 2009

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-099-9

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý první ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešením úloh našeho semináře získávají středoškoláci praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé disciplíny informatiky. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy. To ovšem vůbec neznamená, že nemá smysl takové problémy řešit – při troše přemýšlení není příliš obtížné nějaké (i když někdy ne to nejlepší) řešení nalézt. Nakonec – posuďte sami.

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání obvykle šesti úloh, v klidu domácího krbu je (ne nutně všechny, počítají se nejlepší čtyři) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na naši adresu (ať už fyzickou či elektronickou). My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

Za jeden školní rok obvykle proběhne pět sérií. Závěrečným bonbónkem je pak pravidelné *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě).

Náš korespondenční seminář není ojedinělou aktivitou svého druhu – existují korespondenční semináře z fyziky a matematiky při MFF, jakož i jiné programátorské semináře (kupříkladu bratislavský). Rozhodně si však nekonkurujeme, ba právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

Velice rádi vám odpovíme na libovolné dotazy jak ohledně studia informatiky na naší fakultě, tak i stran jakýchkoli informatických či programátorských problémů. Jakýkoli problém, jakákoli iniciativa či nabídka ke spolupráci je vítána na adrese:

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

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Zadání úloh

V letošním ročníku jsme se rozhodli pro lehce netradiční formu příběhů, které prokládají jednotlivé úložky a patří již k tradičnímu folklóru KSP. Na rozdíl od minulých dvou let jsme letos neměli příběh jediný, který by spojoval všech pět sérií, ale představujeme vám pět různých povídek z pěti různých žánrů od pěti různých autorů. První povídku vám přináší reportér Michal Vaner.

Milí čtenáři, tímto článkem se s vámi loučím. Píši do časopisu KSP již příliš dlouho a každý potřebuje trochu kariérního růstu. Možná bych vám ale měl poskytnout alespoň malé vysvětlení.

Nedávno se uvolnilo místo šéfredaktora. To byla skvělá příležitost pro člověka znuděného běžnou denní prací novináře (získávání informací v terénu, nasazování života, podplácení, psaní článků, odpovídání na stížnosti a tak podobně) Bohužel, takových lidí máme v redakci mnoho, místo šéfredaktora jen jedno. Což si žádalo výkon hodný nejlepšího reportéra. Jak se říká, bombu.

Po kratším zkoumání možností jsem se nerozhodl pro nic menšího než reportáž přímo z pekla. Zabalil jsem dostatek papíru, tužku, diktafon, příručku vymítání ďábla pro všechny případy a vyrazil.

Jak jistě ví každý čtenář, do pekla se dá dostat dvěma způsoby. Ten obvyklý je však jednosměrný a poněkud nepříjemný, proto jsem využil pekelnou bránu.

Pro vyvolání pekelné brány je nutné zapálit oheň v ohništi o obsahu 666 čtverečních centimetrů. Nejjednodušší ohniště je ve tvaru konvexního mnohoúhelníku, ovšem je trochu problém správně trefit obsah.

21-1-1 Ohniště — 9 bodů

Ohniště je konvexní mnohoúhelník. Na vstupu je zadané jako seznam jeho vrcholů v kartézské soustavě (jejich x-ová a y-ová souřadnice), seřazený ve směru hodinových ručiček.

Vaším úkolem je spočítat jeho obsah.

Příklad: Vrcholy: $(0,0), (0,1), (1,0)$, obsah: 0.5

Formát vstupu si zvolte libovolný, avšak souřadnice jsou reálná čísla.

Oheň se rozhořel, já vhodil síru a brána se otevřela. Sestoupil jsem po schodišti do něčeho, co vypadalo jako temně rudě vymalované sklepení s klenutým stropem. Osvětlení zajišťovaly plameny šlehající ze spár mezi kameny černými od sazí. Přestože mi v žádném případě nebyla zima, běhal mi mráz po zádech a naskakovala husí kůže. Možná za to mohly divné zvuky vycházející ze vzdáleného konce chodby, možná jen to, jak neobvykle tato poněkud strohá (byť originální) výzdoba působila.

Sklepní chodba vyústila do prostorného sálu. Možná by se dalo říci skladu, neboť se po zemi a podél stěn válely hromady polen, staré vidle, několik kotlů, a dokonce starý, moly prožraný, čertí kožich.

Otočil jsem se a zjistil, že chodba za mnou zmizela. Místo ní se na okraji skladu nacházely tři stojany na kotle, řádně rezavé, aby nebozí hříšníci měli strach, že dostanou infekci. Napadlo mě, že bych se měl trochu maskovat a zmizet, než mě najdou hned tam, kudy jsem přišel, zajisté vědí, kam se jim tvoří brány. Popadl jsem onen starý kožich, jedny vidle a vyšel ze skladu.

Jen co jsem vyšel na chodbu, potkal jsem mladou čertici. Její dlouhé černé vlasy krásně ladily k učesanému kožichu. Usmála se na mě. Nevím, co se jí mohlo líbit na mě, když jsem neměl ani rohy, ani ocas a kožich jsem měl řádně vypelichaný, ale určitě to byl úsměv. Upřímně řečeno mě taková věc vyděsila. Přeci jen, já jsem obyčejný smrtelník, který je na černo v pekle a ona čertice. Krve by se ve mě nedořezal, když na mě promluvila:

„Koukám, že preferuješ starý typ vidlí. Dneska už s tím skoro nikdo neumí zacházet, ale je to škoda, vyvolávaly mezi lidmi respekt. Osobně taky preferuji původní model. Nešel by sis někdy zaházet?“

„Um, totiž, já, ehm, totiž, nemám čas na…“, začal jsem se vymlouvat a pokoušel jsem se nenápadně zmizet za roh. Ale ve chvíli, kdy si postěžovala, že také nemá moc času, že musí plánovat optimalizace kotlů, probudily se ve mě novinářské pudy. V kapse jsem zapnul diktafon a začal se vyptávat:

„Co jsou to optimalizace kotlů?“

„No, jak pořád straší s tím globálním oteplováním, tak je lepší mít všechny používané kotle pohromadě, aby méně tepla unikalo nahoru na zem. No a jak tak přibývají noví hříšníci a staří už jsou k nepoužití, tak je třeba je občas přesouvat.“

„A to pomáhá? Myslím jako s tím globálním oteplováním?“

„To netuším, nahoru k lidem chodíš snad ty, ne? Ale podle mě je to úplný nesmysl. Až tam na ně jednou vlítneme, to teprve bude globální oteplení. A oni si zatím dělají hlavu s nějakým stupněm za 20 let.“

„A můžeš našim čte… ehm, můžeš mi, prosím, prozradit, jak se takové optimalizace provádí? To nestačí prostě vzít hříšníky z jedné místnosti a přestěhovat je jinam?“

„A oni rovnou utečou, žejo. Ti se musí stěhovat po jednom, kvůli bezpečnosti. A musí se to pečlivě naplánovat, aby to trvalo co nejkratší dobu. Teď zrovna tu mám seznam, koho kam přesunout. Klidně se podívej …“

21-1-2 Optimalizace kotlů — 10 bodů

Stejně jako v loňském ročníku i letos pro vás budeme připravovat praktické úložky. V každé sérii bude právě jedna, a jak jste již asi uhodli, zrovna jste narazili na první takovou.

Na rozdíl od běžných úložek není potřeba k této úloze sepisovat jakýkoli popis nebo vysvětlení vašeho řešení. Jediným vaším cílem je odladit funkční program, který bude přesně odpovídat specifikaci v zadání.

Zdrojový kód tohoto programu pak odevzdáte do webové aplikace CodEx (*http://codex2.ms.mff.cuni.cz/ksp*), kde za něj rovnou obdržíte body. Pokud vám CodEx žádné body nedá, nedostanete je ani od nás, takže věnujte zvýšenou pozornost tomu, co vám CodEx odpověděl. Na odevzdání máte víc pokusů (kolik přesně se dozvíte přímo v CodExu) a do hodnocení se vám pak bude počítat nejlepší pokus (maximum získaných bodů).

K řešení praktické úložky můžete používat jazyky Pascal, C, C++ a C#. Jiné jazyky bohužel CodEx neumí. Při psaní si dávejte pozor, abyste nepoužívali knihovny a techniky, které jsou závislé na vašem kompilátoru nebo platformě a které nejsou garantované normou použitého jazyka (např. Pascalisté nesmí používat unitu Crt, naopak programátoři v C++ můžou použít knihovny STL).

Přihlašovací jméno a heslo do CodExu je stejné jako do našeho webového submitovátka. Pokud nemáte zřízený účet v submitovátku, musíte se nejprve zaregistrovat. V případě potíží nás můžete kontaktovat na naší e-mailové adrese nebo na diskusním fóru *http://ksp.mff.cuni.cz/forum/codex/*.

CodEx bude během prázdnin odstaven, ale začátkem září již budete moci vaše úložky odevzdávat.

Zadání:

Přesouvání hříšníků mezi kotli není snadná záležitost. Když si jeden nedá pozor, může se mu stát, že nacpe do jednoho kotle hříšníky dva, nebo ještě hůř, že se mu hříšníci rozutečou. A čerti nemají času nazbyt, takže potřebují mít přesuny hotové co nejdřív.

Ve vstupním souboru *kotle.in* se na prvním řádku nachází číslo N, které představuje počet kotlů. Na druhém řádku následuje N čísel oddělených mezerami, kde i-té číslo k_i popisuje i-tý kotel (kotle číslujeme od 1). Pokud je $k_i = 0$, znamená to, že je i-tý kotel prázdný. Naopak nenulové k_i znamená, že se v i-tém kotli vaří hříšník, kterého je potřeba přesunout do k_i-tého kotle. Speciálně pak pokud je $k_i = i$, tak se v i-tém kotli nachází někdo, koho není třeba přesouvat.

V jednom okamžiku lze přesouvat jen jednoho hříšníka, a to z plného kotle do prázdného (takže nikdy nesmí být v žádném kotli víc než jeden hříšník). Celkový počet přesunů musí být minimální možný.

Své výsledky uložte do souboru *kotle.out*, kde každý přesun bude na samostatném řádku jako dvojice čísel i a j (číslo i je index kotle, odkud se přesouvá, a j je index, kam se přesouvá).

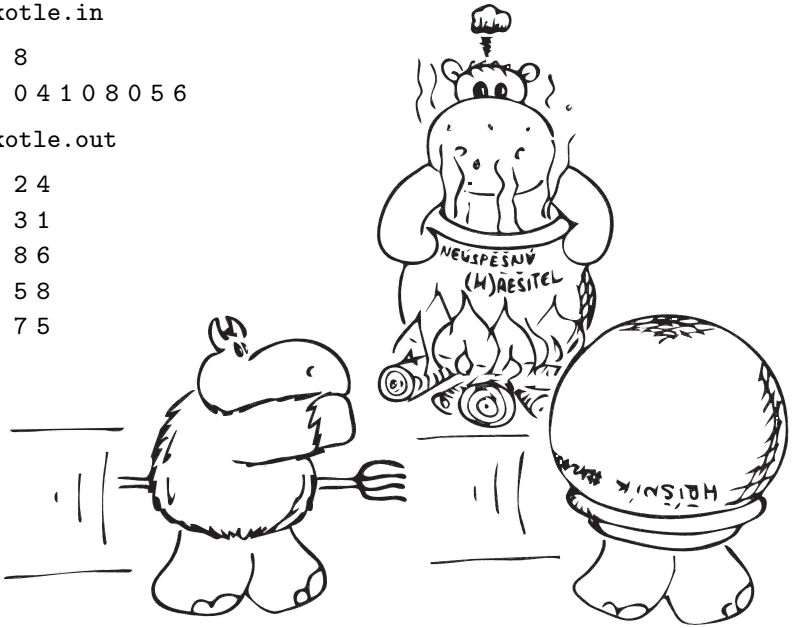
Příklad:

kotle.in

```
8
0 4 1 0 8 0 5 6
```

kotle.out

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



Rozloučil jsem se s čerticí a zašel za roh. Teprve tam jsem si všiml, že si odnesla moje vidle. Inu, co se dá čekat od čertice, řekl jsem si, že vidle stejně k ničemu nepotřebuji, když je to navíc ještě starý typ, se kterým se prý špatně zachází.

Opatrně jsem nakoukl za roh křižovatky a dobře jsem udělal. Dva čerti v uhlově černých kožiších, pravděpodobně místní pořádková služba, právě kontrolovali třetího. Poté, co zjistili, že má všechny doklady v pořádku, nechali ho jít. Já pochopitelně žádné doklady neměl, takže jsem ani netoužil po setkání s nimi.

Poté, co odešli, jsem se ještě jednou rozhlédl a vydal se další, plameny osvětlenou, chodbou. Za chvíli se přede mnou objevila další křižovatka ...

Jak se ukázalo po asi hodině bloudění chodbami, měl jsem si začít kreslit plány. Tento podzemní komplex je neuvěřitelně rozsáhlý. Kromě toho jsem se vyhnul dalším dvěma hlídkám a při obcházení třetí jsem vrazil do dvojice čertů nesoucí pytel.

„Dávej pozor, ty nemehlo! Co tu vlastně děláš, když nikoho neneseš?“

Co na takovou otázku říci. Vzmohl jsem se jen na to, že jsem se ztratil, což byla ostatně pravda. Ten druhý čert, nejspíš o něco málo méně otrávený tím, že na mě narazil, mi popsal cestu k nejbližšímu plánu a oba dva vyrazili dál se svým nákladem.

Jak se ukázalo, na plánu bylo zakresleno celé jedno podlaží. Zachyceny byly sklady, místnosti s kotli, ubytovny personálu i administrativní místnosti. Taktéž byl u každé chodby naznačen směr, kterým se mají čerti nesoucí své oběti pohybovat. Domyslel jsem si, že se budou někde stýkat a tam bude slavná pekelná váha a přijímací kancelář. Okamžitě mě samozřejmě začalo zajímat, kde se taková věc nachází, abych tam čistě náhodou nezabloudil, ještě by mě právem považovali za smrtelníka a rovnou mi poskytli svoje služby.

21-1-3 Přijímací kancelář — 11 bodů

Vaším úkolem je najít na plánu přijímací kancelář. Plán (vstup programu) obsahuje význačné body (křižovatky, kotelny, sklady a podobně) a chodby mezi nimi. Každá chodba má stanovený směr, ve kterém mají čerti chodit. Přijímací kancelář je význačné místo, do kterého se dá dostat z libovolného jiného význačného místa při použití chodeb pouze v předepsaném směru.

Vstup může být zadaný například takto: Na prvním řádku jsou dvě čísla, m a n. n udává počet význačných bodů, m počet chodeb mezi nimi. Každý z následujících m řádků obsahuje dvě čísla, a_i a b_i. Takový řádek říká, že místa a_i a b_i jsou spojena chodbou a čerti v ní mají chodit směrem od a_i do b_i.

Vyrazil jsem opačným směrem, než byla přijímací kancelář, a protože mě začínala přemáhat únava, vešel jsem do nejbližších dveří a v koutě, který mi přišel nejměkčí, usnul.

PJOOONG! Probudil mě nepříjemný zvuk těsně u ucha. Leknutím jsem se narovnal a do něčeho narazil hlavou. To něco byly čerstvě zapíchnuté vidle.

„Co tady děláš?!“ vyjel na mě čert, který je hodil: „To je nápad – chrápat na cvičišti.“ Byl jsem tak ospalý, že mě nenapadla žádná vhodná výmluva. Jak jsem si tak protíral oči, čert si všiml, že něco není v pořádku:

„Ty nejsi čert!“ popadl mě. Nacvičeným chvatem si mě hodil na záda a nesl chodbou. „A ať tě ani nenapadne se vzpouzet,“ dodal konverzačním tónem. Vzhledem k tomu, že nesl v jedné ruce mě a v druhé vidle a na první pohled s tím neměl nejmenší problém, tak mě to vážně ani nenapadlo.

Vstoupil do malé místnosti se stolem a oznámil sedícímu: „Šéfe, mrkněte, koho jsem chytil na cvičišti. Co s ním mám dělat?“

„Hoď mi ho ťámhle ďo kouťa, ať še na něj můžu poďívať,“ zašišlal ten sedící čert, z jehož kvality chrupu se dalo soudit, že má svá nejlepší léta dávno za sebou. Čert-nosič mě upustil na zem a opustil místnost.

„Ťakže, já šem Nejvyšší Bejžebub. A čo ťu děják ty?“

„Já jsem reportér,“ odpověděl jsem po pravdě. Stejně už odhalili, že mi chybí rohy a tak podobně, takže už mi pravda nemohla nijak ublížit.

„Hm, ťo je čo?“

„No, chtěl jsem si to tu prohlédnout a říct ostatním, jak to tu vypadá. Ostatní jsou zvědaví a o pekle se toho zase tak moc neví.“

„Pjohjídnouť žíkáš…“ mumlal si pod fousy, zatímco se přehraboval v hlubinách stolu: „kam jen jšem ji štjčij?“

BUCH! Hodil na stůl těžkou knihu, čímž shodil ze stolu nějaký instrument. „Šakja, už žaše,“ ušklíbl se a začal listovat v knize. „Jepojtéj, kde jen šem ťo měj… á, ťaďy ťě mám. Hm, hžíchů moč nemáš, pšišej jši šám. Hmm. Aje žaše jši mě pjobudij a ještě jšem ši kvůji tobě jožbij žavjažovačí šyštém…“

„Mohl bych vám ho opravit, ať kvůli mě nemáte škodu,“ snažil jsem se zachránit situaci a především sobě krk. Nabídka ho zaujala a vysvětlil mi, že to, co shodil ze stolu je zavlažovací systém pro jeho bonsaj. Že největší problém je vždy se seřízením. Na bonsaji rostou lístky v trsech, některé trsy jsou pod sebou a množství vody proudící z jednotlivých hubic je třeba přizpůsobit počtu lístků, které rostou pod nimi.

Začal jsem tedy přemýšlet nad seřízením, zatímco Bezezub si sedl do křesla a usnul.

21-1-4 Bonsaj — 11 bodů

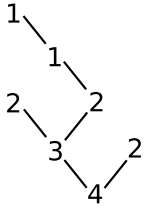
Bezezubova bonsaj se skládá z větviček, lístků a rozdvojek. Dole v květináči je rozdvojka. Z ní doleva i doprava vyrůstají větvičky dlouhé $\sqrt{2}$ pod úhlem 45°. Na konci těchto větviček jsou zase rozdvojky, ze kterých mohou vyrůstat dvojice větviček pod úhlem 45°. Jinými slovy, je to binární strom.

Lístky rostou pouze na rozdvojkách (za rozdvojku je považováno i místo na konci větvičky, ze které vyrůstá méně než 2 další větvičky).

Takovou bonsaj lze popsat v tzv. preorderu. Vypsání stromu probíhá tak, že ho vezmeme za kořen (rozdvojka, která se nachází nejníže) a vypíšeme, kolik je v něm lístků. Poté si představíme, že bychom obě větvičky přeřízli a uříznuté kusy prohlásily opět za stromy. Ty poté vypíšeme stejným způsobem (kořen, uříznout…), napřed levý, potom pravý.

Když se stane, že už nemáme co uříznout (větvička na některou stranu neroste), vypíšeme místo počtu lístků v tomto neexistujícím stromu číslo −1.

Protože Bezezub je nedůvěřivý, k bonsaji nikoho nepustí a dal vám k dispozici jen její popis v tomto formátu.



Jak již bylo řečeno (a lze si všimnout z popisu bonsaje), některé trsy lístků se nacházejí nad sebou. Vaším úkolem je spočítat, kolik lístků je v jednotlivých „sloupečcích“, směrem odleva doprava.

Na obrázku vidíte bonsaj, která odpovídá popisu 4 3 2 −1 −1 2 1 1 −1 −1 −1 −1 2 −1 −1. Spočítané sloupečky vyjdou 3 4 6 2.

Bezezub si prohlížel zařízení a pochvaloval si, jak dobře je seřízené. Když jsem se zeptal, co tedy bude se mnou, zděšeně sebou trhl. Očividně na mě zcela zapomněl. Po chvíli přemýšlení ale řekl, že proti mě v předpisech nic nenašel, takže se můžu po pekle volně pohybovat.

Opustil jsem tedy jeho kancelář a přemýšlel, koho z místního obyvatelstva bych mohl vyzpovídat. Přitom jsem procházel jednou z mnoha chodeb, které se tu vyskytují, aniž bych pořádně tušil kam.

„Brrlrlrlr“ ozvalo se ze dveří. Nakoukl jsem dovnitř a uviděl bar. „Dneska to nalejváš nějaký silný. Ale co už, ještě jednu,“ objednával čert s červeným nosem a rohy. „Brrlrlrlr“ hodil do sebe další skleničku a odvrávoral ke stolu, kde byli tři další, podobně červenorozí, čerti. Vytáhli karty a začali je rozkládat po stole.

Rozhlédl jsem se po ostatních stolech, kde vysedávali čerti a čertice v různém stádiu společenské vyčerpanosti. Vidle měli všichni odložené u dveří a nalévali do sebe nějakou žlutočervenou tekutinu, ze které se kouřilo a občas vylétla jiskra. Barman se právě přehraboval někde v policích, zatímco ocasem naléval další skleničku žlutočerveného moku.

Rozhodl jsem se přistoupit ke stolku karetních hráčů: „Promiňte, že vyrušuji, můžu se vás zeptat, co je to za hru?“ Jeden z čertů se otočil a zareagoval otázkou: „A ty jsi jako kdo? Kde máš vocas?“ Začal jsem tedy vysvětlovat svou situaci a po chvíli zjistil, že s nimi sedím u stolu a v ruce držím skleničku žlutočerveného čehosi. Zatímco jsem nedůvěřivě pozoroval jiskry vylétající ze sklenice, a čerti se bavili mým nedůvěřivým pohledem, jeden z nich mi vysvětlil princip karetní hry. Nebo, alespoň se o to pokusil.

Co si pamatuji s jistotou, je, že karty byly rozděleny do dvou skupin. Jedna skupina, ta větší, byli čerti, a druhá čertice. A základní pravidlo vykládání bylo, že se nikdy nesmí vyložit dvě čertice vedle sebe. Na dotaz proč mi bylo řečeno, že to by pak bylo opravdu peklo, kdyby se mohly na nebohé čerty dohodnout.

A za každé vyložení balíčku byly nějaké body. Protože jsem se ale nakonec k ochutnání onoho jiskřícího nápoje odhodlal a nezůstalo u jednoho, tak si nepamatuji, jak. Alespoň by mě zajímalo, kolik možných způsobů vyložení existuje.

21-1-5 Zapeklitá karetní hra — 8 bodů

Máme balíček karet. V tomto balíčku se nachází Č čertů a Ď ďáblic, přičemž Č > Ď. Úkolem je spočítat, kolika způsoby lze balíček vyložit do řady tak, že žádné dvě ďáblice nejsou vedle sebe. Jak čerti, tak ďáblice jsou navzájem nerozlišitelní (tedy, čert od čerta a ďáblice od ďáblice).

Když jsem se ráno s bolestí hlavy probudil, marně jsem vzpomínal, kdy jsem šel spát a kolik jsem toho vlastně vypil. O to větší šok to byl, když jsem dostal do rukou pytel a vidle. Řekli mi, že jsem se s nimi včera večer dohodl, že dnes

společně vyrazíme (jen tak cvičně) na lov hříšníků. Čerti byli čtyři, všichni si to zřejmě pamatovali a já s vidlemi zacházet neuměl, takže mi ani nic jiného nezbývalo.

Musím ale uznat, že to byla rozhodně lepší práce, než sedět u počítače a psát články do časopisu. Takže jsem se rozhodl změnit zaměstnání a čerti mi slíbili, že mě naučí zacházet s vidlemi, létat, a že i ten ocas a rohy mi nějak zařídí. Toto je můj poslední článek. Doufám, že bude otištěn, když už ho posílám poštou a nedostanu za něj žádný honorář.

Nevyhlašujte po mě pátrání.

Na brzkou shledanou

Váš Kadet Služebníků Pekelných

21-1-6 Nejkratší vyhrává — 12 bodů

V minulých ročnících KSP jste na tomto místě potkávali seriál zabývající se nějakým netradičním způsobem programování – hradlovými sítěmi, logickými predikáty, funkcionálně, pravděpodobnostně, ... Pro letošní ročník si pořídíme poměrně běžný programovací jazyk, ale zatímco většinou nás zajímá *nejefektivnější* program (tedy takový, který spotřebuje co nejméně času a paměti), tentokrát budeme hledat řešení, které je *nejkratší možné.*

V Pascalu nebo Céčku je ovšem délka programu pojem poměrně podivný, protože můžete do jednoho příkazu poskládat mnoho navzájem nepříliš souvisejících operací – příkladem budiž třeba jednořádkové řešení úlohy 16-4-1, které najdete v archivu KSP na webu. Proto si zavedeme svůj vlastní jazyk, ne nepodobný assembleru dnešních procesorů, ve kterém budou jednotlivé instrukce vykonávat jen velice jednoduché operace. Jazyku budeme říkat Rapl (Raw Abbreviated Programming Language).

Paměť našeho pomyslného počítače je tvořena 26 *registry* označenými `a` až `z`. Do každého z nich lze uložit libovolné 32-bitové číslo, tedy celé číslo x v rozsahu $0 \leq x < 2^{32}$. Mimo to má náš počítač k dispozici ještě 26 *polí* značených `A` až `Z`. Každé z nich je indexováno 32-bitovými čísly a jeho prvky jsou opět 32-bitová čísla. Často budeme pracovat s dvojkovými zápisy čísel, budeme je značit tučnými číslicemi **0** a **1**. Tedy $10 = \mathbf{1010}$, $12 = \mathbf{1100}$.

Program je tvořen posloupností *instrukcí.* Těch je několik druhů. Nejdůležitější jsou *výpočetní instrukce.* Ty mají tvar "$a = b \odot c$", kde $\odot$ je nějaká operace (třeba "+"), b a c jsou hodnoty, se kterými se tato operace provádí, a a je místo v paměti, kam uložit výsledek.

Operace jsou k dispozici tyto:

- *Aritmetické operace* `+`, `-`, `*`, `/` (celočíselné dělení) a `%` (zbytek po dělení). Ty fungují tak, jak jsme zvyklí, ovšem pokud se výsledek

nevejde do 32-bitového čísla, jsou "přečnívající" bity oříznuty, jinými slovy počítáme modulo 2^{32}. Například $2^{31}+2^{31}=0$ a $1-2=2^{32}-1$.

- *Bitové operace* `&` (AND), `|` (OR) a `^` (XOR). Fungují úplně stejně jako v Pascalu nebo Céčku.
- *Bitové posuvy* $<<$ (posuv doleva) a $>>$ (posuv doprava). Výsledkem je číslo vzniklé posunutím dvojkového zápisu čísla b o c bitů doleva, resp. doprava, s doplněním nulami. Příklad: $1 << 5 = \mathbf{100000} = 32$, $12 >> 2 = \mathbf{1100} >> 2 = \mathbf{11} = 3$.
- *Bitová selekce* @, která vybere z čísla b bity ležící na pozicích, na kterých jsou ve dvojkovém zápisu čísla c jedničky, a „sešoupne" je k sobě. Tedy pokud jsou dvojkové číslice čísla b třeba $vwxyz$ a $c = \mathbf{10110}$, je $b \mathbin{@} c = vxy$. Příklad: $\mathbf{11111} \mathbin{@} \mathbf{10110} = \mathbf{111}$, $\mathbf{10101} \mathbin{@} \mathbf{10110} = \mathbf{110}$.

Hodnoty b a c, se kterými operace pracuje, mohou být buďto čísla (32-bitové konstanty), nebo registry `a`–`z`, či případně prvky polí indexované číslem nebo registrem – ty budeme značit `A[i]`. Totéž platí pro místo a, kam se ukládá výsledek, pouze to nemůže být číslo. Za speciální případ výpočetní instrukce můžeme považovat *přiřazení* "$a = b$". (Vlastně ho nemusíme zavádět, měli jsme ho už předtím, třeba ve formě instrukce $a = b + 0$, ale praktická zkratka jistě neuškodí.)

Dále potřebujeme *řídící instrukce,* kterými můžeme program větvit a vytvářet cykly. Bude to instrukce *skoku* "`jump` x". Zde x je nějaké místo v programu (totiž konkrétní instrukce), na které chceme skočit. Určíme ho jednoduše: pořadovým číslem instrukce od začátku programu (první instrukce má číslo 0, druhá 1 atd.). Abychom nemuseli čísla instrukcí ručně počítat, zavedeme *návěští* – před každou instrukci můžeme napsat "*návěští*`:`" a tím si její pozici pojmenovat. Instrukci skoku pak místo čísla instrukce povíme jméno návěští. Argument x instrukce `jump` může být obecně libovolná hodnota, takže pozici instrukce můžeme i uložit do registru, pokud nám to je k něčemu dobré.

Podmíněný skok vyvoláme instrukcí "`if` b `?` $c \Rightarrow$ `jump` x". Ta porovná dvě hodnoty (relace `?` může být `=`, `<>` (nerovná se), `<`, `<=`, `>` nebo `>=`) a pokud je podmínka splněna, skočí na dané místo v programu. Porovnávané hodnoty b a c mohou být stejného druhu jako u výpočetních instrukcí.

Nakonec zavedeme ještě *komunikační* instrukce. Mezi ty patří instrukce "`read` a", která do a (registr nebo místo v poli) zapíše číslo přečtené ze vstupu, a "`write` b", která hodnotu b (číslo, registr, místo v poli) zapíše na výstup.

Program se vykonává od začátku, tedy od nulté instrukce, a končí poslední instrukcí, případně skokem za konec programu. Při spuštění programu jsou ve všech registrech i ve všech položkách polí uloženy nuly.

Předvedeme si jednoduchý příklad. Bude to program, který vypíše všechna čísla od nuly do desítky:

```
        a=0
znovu:  write a
        a=a+1
        if a<11 => jump znovu
```

Tento program by šel zkrátit o jednu instrukci, totiž o inicializaci registru `a`, jelikož máme zaručenu nulovost všech registrů při spuštění programu. Rozmyslete si, proč na méně než tři instrukce zkrátit nejde.

Ještě si ukážeme program, který přečte ze vstupu posloupnost čísel ukončenou nulou a pak ji vypíše pozpátku (včetně původně koncové nuly):

```
zapis:  i=i+1
        read A[i]
        if A[i]>0 => jump zapis
vypis:  write A[i]
        i=i-1
        if i<>0 => jump vypis
```

Úloha: V prvním dílu našeho seriálu jsme si pro vás přichystali následující úlohu. Pro každou ze zadaných posloupností čísel napište *co nejkratší* program (měřeno počtem instrukcí), který tuto posloupnost vypíše. Za koncem posloupnosti může program vypisovat libovolná další čísla.

a) 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
b) 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233
c) 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3
d) 0, 1, 0, 1, 2, 3, 2, 3, 0, 1, 0, 1, 2, 3, 2, 3, 4, 5, 4, 5, 6, 7, 6, 7, 4, 5, 4, 5, 6, 7, 6, 7.

(Na tuto úlohu se můžete dívat třeba jako na trochu netradiční formu IQ-testu, případně na způsob komprese dat.)

Vzorová úloha

Jako bonus přidáváme do této série malou ukázku, jak se dá KSP řešit.

Představme si asi takovouto úlohu. Máme zadané body v rovině, některé z nich jsou spojeny úsečkou (tedy, máme k dispozici seznam bodů a seznam úseček). Body jsou v obecné poloze (to znamená, že žádné tři neleží v jedné přímce). Navíc, žádné dvě úsečky se neprotínají. Úkolem je spočítat, kolik je zde trojúhelníků.

Řešení viditelné na první pohled by vypadalo asi takto: Každý trojúhelník se skládá ze tří bodů, které jsou navzájem spojené. Vezmeme proto všechny trojice bodů a u každé vyzkoušíme, jestli jsou propojené.

Budeme předpokládat, že body jsou čísla, kdyby ne, tak si je očíslujeme, aby se nám s nimi lépe pracovalo.

Jak to provést? Vezmeme tři cykly, které budou procházet přes seznam vrcholů a zanoříme je do sebe. To by ale vygenerovalo každou trojici celkem

šestkrát (s různým pořadím bodů). Sice bychom mohli nakonec vydělit výsledek šesti, ale je to zbytečná práce. Proto raději zařídíme, aby se každá trojice objevila právě jednou.

Zvolíme si například její podobu, ve které čísla bodů rostou (první je nejmenší, poslední největší). Vnější cyklus nám „volí“ číslo prvního bodu (řekněme i). Protože druhý bod bude mít číslo větší, nemá cenu další, zanořený, cyklus začínat od jedničky. Začneme ho od nejmenšího čísla, které dává smysl, tedy od $i + 1$. Obdobný trik použijeme na vnitřní cyklus.

Druhý problém je, jak nejrychleji zjistit, jestli jsou dva body spojené. Na vstupu máme seznam úseček, procházet ho celý by trvalo dlouho. Vytvoříme si proto tabulku (dvourozměrné pole). Oba indexy tohoto pole budou vrcholy a v buňce i, j bude $true$ právě tehdy, když jsou body i a j spojeny úsečkou. Na začátku vyplníme samé $false$ a potom, průchodem přes všechny úsečky, zaznamenáme, kde jsou. Zjistit, zda jsou dva body spojeny, je jednoduché, prostě se podíváme na správné místo do tohoto pole. (Odborně se této tabulce říká matice sousednosti.)

To, že tento algoritmus funguje, je zřejmé – každou trojici vygenerujeme právě jednou a u každé ověříme, jestli tvoří trojúhelník.

Paměťová složitost je $O(n^2)$ (kde n je počet bodů) – potřebujeme tabulku velikosti $n \cdot n$ prvků.

Časová složitost je $O(n^3)$. To lze nahlédnout například tak, že program obsahuje tři, do sebe vnořené, cykly, každý prochází maximálně n prvků. Zpracování jedné trojice trvá konstantně dlouho – podíváme se na tři prvky v tabulce.

Takové řešení by jistě získalo nějaké body (funguje a to dokonce v polynomiálním čase – netrvá mu to žádných $O(2^n)$ či $O(n!)$), ale kdybychom pomýšleli na maximum, museli bychom se ještě trochu zamyslet a přijít s něčím lepším.

Tedy, ještě jednou se podíváme, jak vypadá trojúhelník. Všimneme si, že trojúhelník je úsečka, jejíž oba koncové body jsou spojené s jiným bodem. A hned je na světě rychlejší algoritmus. Místo procházení trojic vrcholů budeme procházet dvojice hrana-vrchol a zkoumat, jestli tvoří trojúhelník.

Provedení bude obdobné, budou nám stačit cykly dva. První projde všechny úsečky a druhý uvnitř bude ke každé zkoušet všechny body.

Nyní ale bude trochu obtížnější vymyslet, jak se vyhnout duplicitám. Stačí se malinko zamyslet a přijdeme na to, že vnitřní cyklus stačí startovat na čísle o jedna větší, než je větší z koncových bodů úsečky.

Opět, algoritmus musí fungovat, neboť zkouší všechny možnosti.

Paměťová složitost je opět $O(n^2)$, přestože si nyní musíme navíc pamatovat ještě seznam úseček, abychom přes ně mohli procházet. Ale počet úseček určitě nebude větší, než je $O(n^2)$ – každá spojuje některou dvojici bodů, těchto dvojic je $\frac{n \cdot (n-1)}{2}$.

Časová složitost je $O(n \cdot m)$ za zkoušení všech možností (m je počet úseček). Tentokrát ale nemůžeme zanedbat tvorbu tabulky. K tomu potřebujeme $O(n^2)$, musíme ji napřed vyprázdnit – v případě, že by m bylo malé, např. 0, pak by to hrálo svoji roli. Tedy složitost je $O(m \cdot n + n^2)$.

Kdybychom chtěli, můžeme zabrousit ještě trochu do matematiky, podívat se na body a úsečky jako na rovinný graf a dokázat, že $m \leq 3n$. To nám umožní učinit odhad časové složitosti $O(n^2)$ – na kterém je již jasně vidět, že jsme si oproti minulému algoritmu pomohli.

Poslední věc, která zbývá udělat, je napsat program (čtenář si může procvičit za domácí úkol). Také bychom mohli věřit vlastnímu úsudku a spolehnout se na to, že algoritmus je vymyšlený dobře a že organizátor pochopí jak algoritmus, tak zdůvodnění správnosti a časové složitosti jen z popisu.

Po autentické reportáži z pekla, která nás stála jednoho reportéra, jsme se rozhodli držet při zemi (resp. na zemi). Téma druhé série se ponese ve znamení věrného popisu matfyzácké reality. Jak vypadá běžný den matfyzáka, vám popíší přímo ti nejkvalifikovanější – Jan Bulánek a Zbyněk Falt.

Jdu temným lesem, když tu najednou za sebou uslyším plíživé kroky. Neotáčím se a pomalu zrychluji. Ale kroky se stále blíží, a když už téměř utíkám, uslyším hluboký hlas: „Definuj Lebesgueův integrál!“ V tu chvíli mi ztuhne krev v žilách, snažím se vykrucovat, ale hlas je neústupný. Během vteřiny mi před očima proběhne celý můj čtyřletý matfyzácký život a pomalu se s ním loučím ...

„O-ou,“ ozve se náhle, „jdeš dneska do školy?“ Zvedám hlavu otlačenou od klávesnice. Spolubydlící mi píše na ICQ.

„Jasně, že jdu!“ odepisuji.

A tak mi začíná další všední den – den matfyzáka.

Vypiji dva dny starou kávu, která chutná spíš jako polévka, jež byla v hrnku před ní, a pomalu se přesunu k umyvadlu. Zbytkem kartáčku si vyčistím zuby a podívám se do zrcadla, jenže hřeben nikde. Tak aspoň polituji všechny, kteří mě dnes uvidí. Holicímu strojku se jenom zasměji a jdu se oblékat. Děravé ponožky, tenisky vylepšené o několik ergonomických děr, tradiční ledvinka. Když ale zpoza postele vyndávám své oblíbené tričko, nestačím se divit. Kdysi krásně bílé, nyní hýří barvami. Ale asi nic divného, když na něj něco tu a tam ukápne.

21-2-1 Špinavé tričko — 10 bodů

Takové tričko si lze představit jako obdélník a skvrny si lze rovněž představit jako obdélníky různých barev, které se mohou vzájemně překrývat. Pokud se na jednom místě překrývá více skvrn, je na tomto místě vidět pouze ta skvrna, která se na tričku objevila jako poslední. Vaším úkolem bude pro zadané

skvrny spočítat, kolik které barvy se na tričku vyskytuje a kolik ještě zbylo nezašpiněného trička.

Na vstupu dostanete kladná celá čísla W a H, která představují šířku a výšku trička. Levý dolní roh bude mít souřadnice [0,0] a pravý horní roh souřadnice [W,H]. Dále dostanete číslo N, které představuje počet skvrn, ty jsou na vstupu zadávány přesně v tom pořadí, v jakém se objevovaly na tričku. Každá skvrna je zadána pěti kladnými celými čísly. Souřadnicemi levého dolního rohu, souřadnicemi pravého horního rohu a svou barvou. Barvy jsou očíslované od 1 do N.

Příklad: Pro $W = 6$, $H = 6$, $N = 3$ a skvrny (1, 1, 5, 5, 2), (1, 2, 2, 3, 1) a (4, 4, 6, 6, 3) bude výstup, že čistého trička zůstalo 16 čtverečních jednotek, barva 1 zabírá 1 jednotku, barva 2 zabírá 14 jednotek a barva 3 se vyskytuje na 4 jednotkách.

Konečně mohu vyrazit z kolejí, doběhnout autobus a nestihnout tramvaj. Však pojede další a škola počká. V tramvaji se snažím vyřešit domácí úkol z diskrétní matematiky. Ještě, že je tak jednoduchý. Ale i tak se postaral na celou cestu o můj typický matfyzácký nepřítomný pohled, který se změnil až ve chvíli, kdy jsem o 4 zastávky přejel Malostranské náměstí. Konečně přijíždím ke škole. Místo na přednášku ale směruji své kroky k rotundě, ve které je počítačová laboratoř, abych se podíval, kde že to vlastně mám přednášku. Zrovna jsem stál uprostřed, když mě polil studený pot. Ta noční můra měla být varováním, neboť na dnešek jsem měl od profesora už po několikáté slíbeno, že mě z té analýzy vyzkouší, ať chci nebo nechci. Tentokrát ale jeho výraz nasvědčoval tomu, že to myslí smrtelně vážně ...

21-2-2 Útěk před zkouškou — 9 bodů

Rotunda má tvar kruhu. Náš hrdina se nachází přesně uprostřed, zatímco profesor na jeho obvodu. Protože je podél stěn rotundy mnoho skříní, stolů a východů, stačí, aby se student dostal k libovolnému bodu na obvodu, odkud již může utéct nebo se bezpečně schovat. Samozřejmě, že se zároveň na tomto bodu nesmí vyskytovat i profesor (pak by se velmi těžko schovávalo a student by byl okamžitě vyzkoušen). Má to ale jeden háček, matfyzák nezvyklý pohybu se pohybuje 4× pomaleji než rozzlobený profesor. Profesor se ale na druhou stranu z neznámého důvodu bojí přiblížit ke středu, takže se pohybuje pouze podél obvodu.

Najděte strategii, jak se má za těchto podmínek student pohybovat, aby profesorovi vždy utekl, nebo dokažte, že mu utéct nelze. Profesor se může pohybovat zcela libovolně, takže o jeho „chytací“ strategii nemůžete dělat žádné předpoklady.

Ani nevím, jestli se mi podařilo utéct nebo ne. Každopádně mi z toho pořádně vyhládlo. Takhle se přeci nemohu soustředit. A tak jsem se odhodlal k zoufalému činu. Vydal jsem se do menzy. Bohužel jsem vůbec nebyl sám, kdo dostal hlad, takže před výdejnou byla obrovská fronta.

21-2-3 Fronta — 10 bodů

Matfyzáci jsou pyšní na svou inteligenci a dávají to ostatním najevo. Nejvíce se tento problém projevuje, když jsou matfyzáci nuceni tvořit fronty. Matfyzák, který si o jiném matfyzákovi myslí, že je hloupější, odmítá stát ve frontě za ním. Tím vzniká řada nepříjemných strkanic a šarvátek.

Napište program, který dostane seznam matfyzáků a jejich názorů na inteligenci ostatních. Vaším úkolem je matfyzáky uspořádat do posloupnosti tak, aby vždy platilo, že pokud považuje matfyzák A kolegu B za hloupějšího, pak musí v této posloupnosti stát A před B. Samozřejmě, že takové uspořádání nemusí existovat. Např. když si všichni myslí, že jsou chytřejší než všichni ostatní. V takovém případě oznamte, že uspořádání nelze vytvořit.

Tato úložka je praktická, což znamená, že řešení budete odevzdávat výhradně formou odladěného zdrojového kódu. Přesnější zadání a formulář na odevzdání kódu naleznete na stejném místě jako v předcházejících sériích – v CodExu na adrese *https://codex2.ms.mff.cuni.cz/ksp/*. Nevíte-li, co praktická úložka je a jak přesně postupovat, podívejte se do zadání úlohy 21-1-2 „Optimalizace kotlů“.

Ještě, že to (jako ostatně vždy) vyšlo tak, že jsem šel na řadu první, takže jsem mohl nerušeně pokračovat ve studiu. Tříhodinové zkoušení Unreal Tournamentu v rámci předmětu „Vývoj počítačových her“ mi vždycky šlo a na rozdíl od jiných předmětů jsem v něm viděl svou budoucnost. Bohužel někdo vždycky rozpojí pracně vytvořenou síť, takže počítače musíme každý týden sesíťovávat znovu a znovu. A to zavání nepříjemnou fyzickou prací.

21-2-4 Síťování — 8 bodů

Sesíťovat počítače není žádná maličkost. Nemají totiž klasické síťové rozhraní. Každý počítač má dvě zdířky na síťový kabel. Jednu vstupní a jednu výstupní. Pokud tedy chcete dosáhnout konektivity mezi všemi počítači, musí být spojeny do kruhu, a to tak, že každý kabel vede z výstupní zdířky jednoho počítače do vstupní zdířky druhého počítače. Navíc jsou kabely špatně odstíněné, takže se nesmí nikde křížit, aby se navzájem nerušily.

Na vstupu dostanete číslo N, které značí počet počítačů v místnosti. Následuje N řádků, přičemž i-tý řádek určuje souřadnice i-tého počítače v místnosti. Vaším úkolem je najít pořadí počítačů $p_1, p_2, \ldots, p_n$, takové, že se v něm každý

počítač vyskytuje právě jednou a úsečky (p_1, p_2),(p_2, p_3), …, (p_{n-1}, p_n), (p_n, p_1) se nikde neprotínají. Pokud to není možné, vypište, že řešení neexistuje.

Například pro $N = 4$ a souřadnice $(0, 0)$, $(0, 2)$, $(1, 1)$ a $(-1, -2)$ může být výstupem třeba $(2, 3, 4, 1)$.

Když jsme po sedmi hodinách studia uznali, že už umíme dost a že se nám dělají mžitky před očima, vydal jsem se domů. Na kolejích na mě ale čekalo nepříjemné překvapení. Naše nádobí mi totiž div nepřišlo otevřít dveře. Už jednou jsme se pokoušeli tuto situaci teoreticky řešit, ale očividně bezúspěšně. Sice by se zdálo, že nejsnazší je všechno nádobí prostě umýt, ale na co bychom pak studovali informatiku?

21-2-5 Nádobí — 10 bodů

Umývání nádobí je velice náročná činnost, takže lze umýt pouze jeden kus za jeden den. Bohužel ale platí, že když některý kus neumyjete do určité doby, nemá smysl jej umývat vůbec a je mnohem ekonomičtější vyhodit jej a koupit nový kus. Samozřejmě, že za ta léta už studenti vědí, kolik dní určité kusy vydrží bez umytí, i kolik takový kus stojí nový.

Vaším úkolem bude navrhnout optimální systém umývání nádobí takový, aby student musel za nákup nového nádobí zaplatit co nejméně. Na vstupu dostanete číslo N, které představuje počet kusů nádobí. Dále N dvojic čísel D_i a C_i, což znamená, že i-tý kus vydrží ještě D_i dní a nový stojí C_i korun.

Vypište, v jakém pořadí se má nádobí umývat (jeden kus za jeden den) tak, aby se umyly/koupily všechny kusy a zároveň náklady na nákup byly minimální.

Například pro vstup $N = 3$ a dvojice $(1, 5)$, $(1, 4)$ a $(2, 3)$ je správný výstup $(1, 3, 2)$. Což znamená, že umyjeme 1. a 3. kus. Druhý kus už bohužel nestihneme umýt včas, a tak jej budeme muset koupit nový. Všimněte si, že jakmile nestihneme druhý úkol, už není kam spěchat a raději si uděláme třetí, za který díky tomu nezaplatíme pokutu. Náklady na nákup nového nádobí jsou 4 koruny.

Konečně si mohu do nového hrnku uvařit oblíbenou čínskou polévku a jít se podívat, kdo je online. No jo, noc bude ještě dlouhá. Navíc je potřeba zhlédnout nový díl Simpsonových. Po několika hodinách začínám cítit, že bych měl jít spát. Ale co, ještě jeden díl určitě vydržím … Zzz

Jdu temným lesem, když tu najednou …

Dnes sáhneme do trochu jiného soudku. Po dvou vydařených reportážích jsme se rozhodli zařadit trochu oddychovější téma. O pár řádků níže se můžete zakousnout do krátké sci-fi povídky s nádechem cyberpunku, kterou pro vás připravil Martin „Bobřík" Kruliš. Měli jste někdy sen, který vypadal jako skutečný? Co když to není jen zdání a sen začne žít vlastním životem . . . ?

„Zase jsem měl ten sen."

„Opravdu? A který?" pozvedl doktor obočí a dál čmáral cosi do svého poznámkového bloku. Sám nevím, jestli je to vlastnost všech psychologů, nebo zda to dělá jen ten můj, ale občas mám pocit, že vám vůbec nevěnují pozornost a snaží se jen udržovat konverzaci hloupými dotazy.

„Ten o těch lidech," řekl jsem unaveně. „Vyspělá civilizace, utopická společnost a tak dále. Už ani nevím, kdy se mi naposledy zdálo něco jiného."

„A zdá se vám pokaždé to samé, nebo pozorujete drobné rozdíly?" zeptal se s dobře hraným výrazem neutuchajícího zájmu.

„Je to pokaždé . . . trochu jiné. Skoro jako seriál – každý sen je o něčem novém, ale zároveň stále o tom samém . . . "

⋆⋆⋆

Opět mi neřekl nic převratného. Stále ty samé řeči o přepracování a stresu. Nemyslím, že by to byl zrovna můj případ. Jmenuji se Harold a tohle je můj život. Pracuji na pozici řadového úředníka v účetní firmě. Procházím formuláře a počítám nejrůznější statistiky. Můj život je klidný a předvídatelný. Největší vzrušení jsem zažil, když kolega z kanceláře ztratil sešívačku a celá směna úředníků mu ji pomáhala hledat. Takže jak jsem říkal – s tím stresem se doktor trochu netrefil.

Po návštěvě ordinace mě čekala práce. Celý den proběhl poklidně – ostatně jako každý jiný den. Žádné vzrušení. Žádný stres. Kancelář jsem opustil přesně v 17 hodin a zamířil do podzemky. Byla ošklivě přeplněná, ale takový už je život. Stálo mě to zpoždění dlouhé přesně tři minuty a čtyřicet pět vteřin.

Doma následovala obvyklá večerní rutina. Nakrmit rybičky, večeře a televizní zprávy na kanálu 6. Dělám to tak každý den. Do postele jsem ulehl s rozečtenou knihou a pomalu se ukolébal čtením.

⋆⋆⋆

Pohled na město byl nádherný. Táhlo se od obzoru k obzoru a stříbrně lesklé výškové budovy šplhaly k nebi. Každý den vyrostla alespoň jedna nová. A mezi nimi na různých letových hladinách proplouvala vznášedla. Úchvatný pohled, a přitom nic neobvyklého. Běžná denní doprava. Lidé jedou do práce, děti do školy . . .

V jedné nevýznamné budově na okraji města uvnitř velikého sálu mezitím pobíhali lidé v bílých pláštích a křičeli na sebe nesrozumitelné věci. Podobali se mravencům, kterým právě někdo šlápl do mraveniště. Shlukovali se do skupinek

a vášnivě debatovali. Co chvíli se zase rozprchli do všech směrů a vytvořili nové skupinky. Hluk sílil a s ním i zoufalství pobíhajících lidí.

Místnost prořízl ostrý pískavý zvuk. Všichni jako na povel utichli a obrátili se k řečnickému pultu, ke kterému právě z davu vystoupil postarší muž s plnovousem. Opatrně přejel ostatní nervózním pohledem a napjatou atmosféru ještě vylepšil váhavým odkašláním.

„Vážení kolegové," začal rozechvělým hlasem, „není o tom již nejmenších pochyb. Všechny naše hypotézy se potvrdily sérií nezávislých experimentů, a tak zbývá jediné rozumné vysvětlení . . . " Na chvíli se odmlčel, aby si osušil pot z čela, a pak pokračoval: „Žijeme ve snu! Celá naše civilizace byla vytvořena snem někoho jiného! Navíc nejsme schopni určit, kdy tento sen –"

⋆⋆⋆

Probudil jsem se. Byly čtyři hodiny ráno a na okno tiše bubnoval déšť. Poslední útržky snu pomalu zahaloval mrak zapomnění. Oči se prodíraly tmou a postupně rozeznávaly obrysy nábytku mého pokoje. Posadil jsem se na posteli a snažil se uklidnit. Nešlo to. Uběhla skoro hodina a já stále nehybně seděl. Hlavou se mi honily nejrůznější myšlenky a paměť se snažila sen uspořádat. Únava mě přemáhala, ale strach mě držel vzhůru. Nakonec mě ukolébal déšť a já spal až do rána.

Budík zazvonil ve čtvrt na osm. Z rozespalosti mě probudila až studená sprcha společně s teplou kávou a dvěma croissanty. Začetl jsem se do ranních novin. Titulní stranu opět zdobil článek o nějakém vědeckém objevu. Oči byly zaměstnány čtením, ale mou mysl ovládaly vzpomínky na muže v bílých pláštích, na jejich experimenty a teorie . . . Z rozjímání mě probral pohled na hodiny. K čertu, prolétlo mi hlavou. Následovalo ještě mnoho slov. Nepěkných slov.

Do práce jsem dorazil s velkým zpožděním. Stálo mě to napomenutí nadřízeného a ostudu před všemi kolegy. První zameškání po tolika letech. Bohužel to nebyl jediný problém, který mě čekal. Má mysl se potulovala kdesi daleko. Soustředit se bylo extrémně těžké a práce mi nešla od ruky. V poledne mě můj nadřízený poslal domů, protože jsem za celé dopoledne nevyplnil jediný formulář správně.

Doma ale také nebylo k vydržení. Chodil jsem nervózně po svém bytě sem a tam. Únava mne pronásledovala a já se jí snažil unikat. Nakonec mě ale přece jen přemohla. Krátký spánek během dne nemůže uškodit. Natáhl jsem se oblečený na postel a opatrně zavřel oči. Jen krátké zdřímnutí. Jen tak krátké, aby se mi nic nemohlo zdát . . .

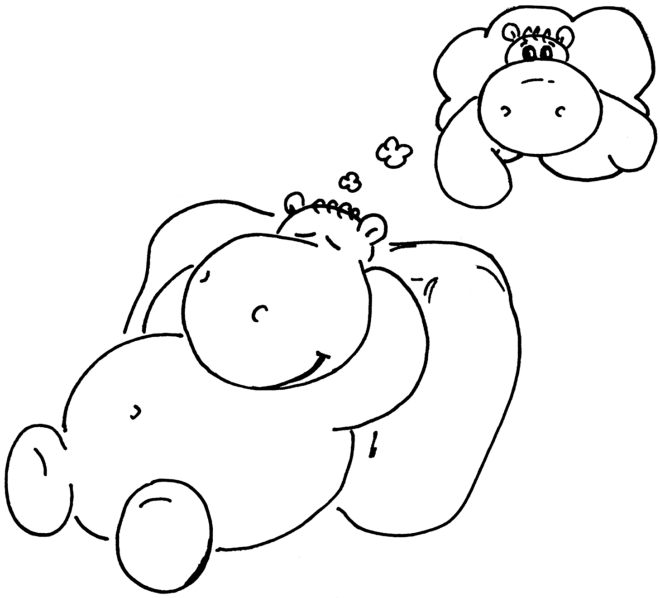
⋆⋆⋆

Pohled na město byl nádherný. Právě zapadalo slunce a jeho odraz se zrcadlil na lesklém povrchu budov. Vzduch se lehce tetelil a podtrhával tak atmosféru letního večera.

„Máme to! Máme to!“ ozvalo se odkudsi. V malé pracovně jedné bezvýznamné budovy na kraji města se rozlétly dveře. Do nich se vřítila postava mladého muže v brýlích a bílém plášti, který nad hlavou vítězoslavně mával štosem papírů.

„Máme co?“ podíval se na něj postarší muž, kterému očividně ona pracovna patřila.

„Přišli jsme na to, jak modifikovat memgramy! Tím můžeme posílat informace přímo do mozku našeho Spáče!“



21-3-1 Kódování memgramů — 9 bodů

Memgram, tedy záznam nesoucí jednu myšlenku, si můžeme představit jako mřížku obsahující 8×8 polí. Každé pole může nabývat dvou hodnot – 0 a 1. Vědci se snaží memgramy modifikovat, aby pomocí nich mohli přenášet své informace. Otázka je, kolik informace můžou zakódovat do jednoho memgramu.

Posílání informace probíhá tak, že vědci dostanou memgram, který nese (z jejich pohledu) náhodnou informaci – tj. nelze dělat žádné předpoklady o tom, kde jsou jedničky a kde nuly. Tomuto memgramu můžou – ba co víc, dokonce musí – změnit jeden bit (jednu jedničku překlopí na nulu nebo obráceně). Mřížka se nesmí otáčet, tzn. políčka jsou jasně a jednoznačně očíslována.

Příjemce (mozek Spáče) pak dostane upravený memgram. Přitom ale neví, jak vypadal původní memgram, tedy ani který bit byl změněn. Protože nás

zajímá maximální velikost přenášené informace, předpokládejte, že mozek umí informaci rozkódovat, ať je jakkoli zakódovaná (zná komunikační protokol).

Pro lepší představu si uveďme triviální příklad, na kterém si ukážeme, jak přenést jeden bit. Řekněme, že hodnota tohoto bitu se bude přenášet na pozici $(1, 1)$ a všechna ostatní políčka budou pro příjemce nepodstatná. Když přijde memgram, podíváme se na naši pozici. Buď tam přímo dostaneme hodnotu, kterou chceme přenést, a pak změníme libovolný jiný bit (vždy musíme něco změnit, tak sáhneme jinam, abychom si políčko $(1, 1)$ nerozbili). Pokud tam je opačná hodnota, změníme políčko $(1, 1)$ a tím ho nastavíme správně. Příjemci pak stačí přečíst toto políčko, aby získal posílanou informaci.

Vašim úkolem je určit, kolik nejvíce bitů informace lze takto přenést v jednom memgramu, a popsat, jak bude tato informace kódována.

Pokud úlohu vyřešíte i obecně pro memgram obsahující libovolný počet (N) bitů, případně pokud dokážete, že vaše řešení je nejlepší možné, bonusové body vás jistě neminou.

„To vypadá zajímavě," pokýval hlavou starší muž, když pročítal papíry mladšího kolegy. „A za jak dlouho může být zařízení připravené?"

„Když budeme všichni pracovat jen na tomto projektu, mohli bychom to stihnout už za pár dní."

„Hmm, to není špatné. Ale pomyslel jste, pane kolego, na možnost, že se mezitím Spáč probudí?"

⋆⋆⋆

Probudil jsem se a posadil na posteli. Bylo pozdní odpoledne a venku se začínalo stmívat. Tyhle sny začínají být čím dál realističtější. A také děsivější. Musím přijít na jiné myšlenky! Krátká procházka by mi mohla vyčistit hlavu. Oblékl jsem se a vyrazil.

V parku se pohybovali nejrůznější lidé. Bylo to ideální místo na odpočinek, přestože o sobě dával podzim vědět vtíravým chladem a všudypřítomným zažloutlým listím. Posadil jsem se na lavičku, pohodlně se opřel. Večerní vánek mi lehce foukal do tváře. Bylo to příjemně osvěžující.

Kolem prošel muž v bílém plášti. Snažil jsem se mu nevěnovat pozornost. Každý přece může nosit bílý plášť! V opačném směru prošla dvojice lidí. Také v bílých pláštích. Vášnivě diskutovali nad nějakými dokumenty. Nedaleko se zastavila žena ve středních letech. Byla zahalená do bílého pláště a venčila malého bílého psa. Vypadal roztomile, ale něco na něm nebylo v pořádku. Bližší pohled odhalil, že pes není bílý. Byl oblečený do bílého oblečku, který nápadně připomínal plášť. Kdo může obléknout psa do něčeho takového?!

Mám snad vidiny, nebo se ti lidé okolo zbláznili? Vydal jsem se směrem domů a snažil se příliš nerozhlížet po lidech. Přepracování, stres, to bude určitě ono! Nic jiného to nemůže být.

Teplá sprcha, několik prášků na spaní a rychle do postele. Prostě se z toho vyspím a bude to! Už žádné sny o bílých pláštích. Usilovně jsem se snažil myslet na jiné věci a prášky pomalu zabraly ...

⋆⋆⋆

Na rozlehlé, sytě zelené louce se pásly ovce. Uprostřed nich seděl bača a hrál na vlastnoručně vyřezanou píšťalku. Ovce klidně přežvykovaly trávu a sem tam se ozvalo zabečení. Bača odložil píšťalku a odkudsi vytáhl dížku na mléko. Sedl si na stoličku vedle nejbližší ovce a začal ji dojit.

Něco nebylo v pořádku. Ovce znervózněly a začaly pobíhat sem a tam. Hustá bílá vlna na nich poskakovala a vlála. Jako by v ní byly jen oblečené a mohly ji kdykoli shodit. Tráva se podivně leskla a její zelená barva vybledla a změnila se na stříbrnou. Jednotlivá stébla se přestala kývat ve větru a stála vzpřímeně kolmo k nebi jako výškové budovy. Ovce si stouply na zadní. Stále pobíhaly kolem a vášnivě spolu diskutovaly bečivými hlasy. Už na sobě neměly vlnu – byly to bílé pláště! Bača zmizel neznámo kam a z louky se stala obrovská prosvětlená místnost ...

Jedna ovce – vědec se přitočil k jinému a do všeobecného hluku téměř zakřičel: „Už to můžeme spustit?!“

„Téměř! Právě provádíme kalibraci!“ odpověděl mu rovněž křikem druhý vědec.

Uprostřed místnosti stál přístroj, který vypadal jako jaderný reaktor zkřížený s obřím kávovarem. Okolo pobíhala hromada vědců v bílých pláštích, kteří usilovně pracovali na různých částech přístroje. Trvalo notnou chvíli, než dokončili všechny drobnosti.

„Jsme připraveni!“ zamával jeden vědec z vrcholu přístroje na kolegu u ovládacího pultu. Ten mu zamával nazpět a pomalu zatáhl za velkou páku. Navzdory všem předpokladům přístroj nezačal ani hučet, ani blikat, jen malá kontrolka signalizovala, že byl uveden do provozu. Vědci z celé místnosti se shromáždili u informační obrazovky, kde s napětím čekali na první výsledky. Číselné ukazatele se pohnuly a začaly stoupat. Ozvalo se hromadné oddechnutí a celý dav začal optimisticky švitořit.

„Výborně! A teď uložíme náš svět do dlouhodobé paměti Spáče,“ zavelel nejstarší vědec. Ostatní se rozprchli po nejbližším okolí a začali opět pilně pracovat.

21-3-2 Nadposloupnost — 10 bodů

Vědci se snaží uložit informace o jejich světě do paměti Spáče. Problém je, že v paměti už některé věci jsou a žádné nesmí zmizet – to by mohlo mít nedozírné následky.

Paměť si představte jako uspořádanou posloupnost vzpomínek. Jednu vzpomínku budeme pro jednoduchost brát jako řetězec. Zároveň je dána po-

sloupnost vzpomínek, které by vědci rádi do paměti uložili. Některé vzpomínky se můžou překrývat s těmi, které už v paměti jsou.

Cílem je najít *nadposloupnost* takovou, aby původní paměť i nové vzpomínky představovaly podposloupnosti této *nadposloupnosti*. Vzhledem k tomu, že paměť není svou kapacitou neomezená, měla by být nadposloupnost nejkratší možná, aby se minimalizovalo riziko zapomínání.

Příklad: V paměti je „`snídaně`“, „`práce`“ a „`večeře`“. Vědci chtějí přidat „`večeře`“, „`sen`“ a „`snídaně`“. Jeden z možných výsledků je „`snídaně`“, „`práce`“, „`večeře`“, „`sen`“ a „`snídaně`“. Vyškrtnutím snu a druhé snídaně dostaneme původní paměť a vyškrtnutím první snídaně a práce dostaneme posloupnost, kterou by rádi vědci do paměti dostali.

Probudilo mě drnčení budíku. Musel jsem spát opravdu tvrdě, protože zvonil už několik minut v kuse. Hlava byla čistá a celým tělem mi pulzovala energie. Byl to nádherný pocit. Stačilo pár minut, abych se osprchoval, nasnídal a vyrazil do kanceláře.

Práce mi šla od ruky. Formuláře, které se mi nahromadily za včerejšek, zmizely ještě před dopolední poradou a chvíli po poledni jsem měl splněné všechny povinnosti na dnešní den.

„Dobrá práce, Harolde!“ pochválil mě nadřízený přede všemi kolegy. „Vidím, že jsi překonal tu včerejší krizi.“

„Potřeboval jsem se z toho jen pořádně vyspat,“ odpověděl jsem ledabyle a přibral si další formuláře navíc od svých kolegů.

Práce mě úplně pohltila. Čas ubíhal a kancelář se postupně vyprazdňovala. Vyrušil mě až vrátný, když kontroloval, zda v budově nikdo nezbyl. To byl ale produktivní den! Musel jsem udělat práci nejméně za pět lidí! Cesta domů byla klidnější než obvykle. Podzemka byla poloprázdná. Aby také ne, v tuhle dobu.

Čekal mne pravidelný večerní rituál. Nakrmit rybičky, večeře a večerní zprávy. Můj přesčas v práci ale způsobil, že zprávy na kanálu 6 už dávno skončily. Na obrazovce pobíhala nějaká zvířata, zatímco hlas na pozadí vyprávěl odborné pikantnosti z jejich života. Zajímavé, ale ne zas tolik. Vypnul jsem televizi a šel si číst do postele.

⋆⋆⋆

Pohled na město byl nádherný. Vycházelo slunce a společně s ním se probouzeli lidé. Proudy vznášedel houstly a město pomalu ožívalo. Do jedné už ne tak bezvýznamné budovy na kraji města se sbíhali vědci. Přednáška na téma Snových světů se bude konat od devíti hodin ve Velké aule, informovaly všudypřítomné plakáty.

Aula se plnila a s přibývajícím množstvím lidí se zvyšoval i hluk. Šepot se brzy změnil v křik a v místnosti nebylo slyšet vlastního slova. K řečnickému pultu vystoupil starší muž s plnovousem. Chvíli počkal, než hluk v sále utichl

na přijatelnou úroveň, a začal přednášet.

Mluvil dlouze o nejrůznějších věcech. Jak je možné žít ve snu a jaké filozofické problémy jsou s tím spojené. Kolik takových snových světů může existovat, jak jsou propojené a zda se dá mezi nimi cestovat. Zda mohou existovat snové světy vytvořené snem osoby, která je rovněž uvězněna ve snu. Jak mohou snové světy ovlivňovat reálné světy a naopak. A na závěr upozornil posluchače na vážný problém.

„Jak jistě všichni víte, jsme uvězněni v mysli Spáče. Spuštěním přístroje na úpravu memgramů se nám podařilo zafixovat naši existenci přímo v jeho paměti a podvědomí, takže nehrozí, že by na nás v blízké době zapomněl. Ale stále tu existuje jeden problém . . . “ Přednášející se na chvíli odmlčel, prohrábl si plnovous a promítl další holografický obrázek. „Co když našeho spáče potká řekněme malá mozková příhoda? Co když ho zítra přejede cestou do práce auto? A i kdybychom měli štěstí a nic z toho se nestalo, spáč je jen obyčejný člověk. Za nějakých padesát, možná šedesát let stejně zemře přirozenou smrtí a naše civilizace zanikne s ním.“

Poslední věta visela ve vzduchu a v aule se rozhostilo úplné ticho. Po chvíli se mezi posluchači zvedla ruka těsně následovaná i jejím majitelem. Stoupl si, rozhlédl se po okolních posluchačích a pak se rozechvělým hlasem zeptal: „A myslíte, že s tím půjde něco udělat, pane profesore?“

„Upřímně řečeno, nevím. Stále nad tím bádáme. Pravděpodobně je naší jedinou možností vytvořit tunel mezi reálným a snovým světem. Problém je, že by si to vyžádalo obrovské množství energie a také úplnou znalost topologie všech snových světů Spáče.“

Aula začala tiše šumět vzrušenými debatami. O chvíli později se z davu zvedla jiná ruka. „Myslím, pane profesore, že bych pro vás mohl vyřešit to druhé . . . “

21-3-3 Topologie snů — 10 bodů

Topologie snových světů je reprezentována binárním stromem. Katedra snového inženýrství se pokusila zmapovat okolní světy, ale zatím mají k dispozici jen neúplná data.

Podařilo se jim získat tento strom vypsaný v prefixové a infixové notaci. Problém je, že pro další zpracování by velice potřebovali mít strom vypsaný také v postfixové notaci.

Napište algoritmus, který z prefixového a infixového výpisu sestaví výpis postfixový, případně oznámí, že ze zadaných dat nelze sestavit tento výpis jednoznačně.

Jednotlivé vrcholy jsou ve stromě označeny celými náhodnými čísly. Označení je navíc jednoznačné – tj. žádné dva vrcholy nemají stejné číslo.

Příklad: Pro strom s prefixovým výpisem 16 11 19 3 42 a infixovým 11 16 3 19 42 bude postfixový výpis 11 3 42 19 16.

Následujících několik dní bylo velmi zajímavých. Energie mne neopouštěla, a tak jsem strávil v práci i celý víkend. Můj plán byl našetřit nějaké přesčasy a pak si udělat dovolenou. Ale nešlo to. Mozek byl příliš aktivní a neustále potřeboval něco řešit. Každou noc se mi zdály tytéž sny o vědcích usilovně pracujících na zařízení, které by je mělo dostat ven ze snu. Začínalo mě to děsit. Opět jsem navštívil svého psychologa a převyprávěl mu své sny i to, co se se mnou v poslední době děje ...

„Takže říkáte, že oni – tedy jako v tom vašem snu – něco staví?“ zeptal se už asi po čtvrté.

„Ano. Jak jsem říkal, staví nějaké zařízení, které by je mělo dostat ven ze snu.“

„Hm. Velmi zajímavé,“ pokýval doktor hlavou a zamyslel se. „Myslím, že byste je měl nechat, aby to dostavěli.“

„A nemohlo by to být – já nevím – nebezpečné?“

„Snad nemyslíte, že by to mohlo být skutečné,“ usmál se. „Ten stroj symbolizuje nějakou věc ve vašem podvědomí, která čeká, až ji vyřešíte. Pomozte jim. Mějte dobrou vůli ten stroj dostavět. Jedině tak vaše podvědomí vyřeší problém, který zřejmě máte.“

Úžasné! Psychologové mají prostě na všechno vysvětlení. Cestou do práce se mi hlavou honily nejrůznější myšlenky. Na jednu stranu mohl mít pravdu. Na druhou stranu, co když je ten sen skutečný. Nebo je také možné, že mi šplouchá na maják! V duchu jsem se zasmál té hloupé myšlence, ale veselo mi nebylo.

Večer jsem dal na radu psychologa. Od teď bude mou jedinou myšlenkou před spaním dostavět ten zatracený přístroj. Ať to stojí co to stojí.

⋆⋆⋆

Pohled na město byl nádherný, ale zdaleka už ne tak úchvatný. Byl to stále ten stejný pohled, který se vracel noc co noc. Budova na okraji města, ve které vědci připravovali svůj přístroj, byla stále plná a žila čilou kreativní prací. Právě řešili další problém ...

„Už se nám podařilo rozmístit jednotlivé sny na různé frekvence, ale stále nemáme potřebný výkon, abychom je dokázali všechny pokrýt,“ říkal právě jeden vědec druhému.

„A co kdybychom optimalizovali hierarchii snů?“

„To by mohlo jít, ale výsledná struktura by musela mít co nejmenší průměr!“ přikývl vědec a začal počítat potřebné úpravy.

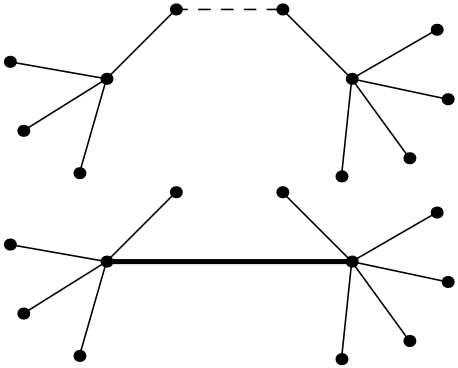
21-3-4 Optimalizace stromu — 11 bodů

Hierarchie, do které vědci přeuspořádali jednotlivé sny, je vlastně strom. Ale už nemusí být nutně binární a také není zakořeněný. Manipulace se sny je velmi složitá, takže můžete jen jednu hranu odebrat a jednu hranu přidat. Vý-

sledek musí být opět souvislý strom a navíc musí mít nejmenší možný průměr.

Pro upřesnění: Termínem *průměr* zde myslíme počet hran na cestě mezi dvěma nejvzdálenějšími vrcholy grafu (v našem případě stromu).

Příklad: Na následujících obrázcích je uveden strom, který máme modifikovat. Na prvním obrázku je zvýrazněná hrana, kterou odebíráme, na druhém hrana, kterou jsme přidali. Původní strom má průměr 5, po úpravě se dostaneme na hodnotu 3.



„To by mělo stačit," přikývl postarší vědec, když prošel všechna čísla. „Zkontrolujte znovu všechny systémy! Zbývá necelá hodina do spuštění!"

Zařízení mělo tvar obrovského oválu. Po obvodu byly přidělány mohutné cívky, ke kterým se táhly tlusté kabely. Pobíhající vědci kontrolovali poslední detaily před prvním spuštěním. Čas pomalu ubíhal a jednotlivé týmy postupně potvrzovaly funkčnost jednotlivých systémů. Přípravy byly dokončeny a čekalo se pouze na pokyn z nejvyšších míst. Pan profesor, který celý vývoj řídil, se postavil před shromážděné vědce.

„Experiment, který nyní provedeme, je velice nebezpečný. Pokud jsme někde udělali chybu, ten, kdo projde bránou na druhou stranu, může skončit kdekoli. Třeba v nějaké noční můře, nebo ještě hůř . . . " Shromáždění vědci se podívali jeden na druhého.

„Nemohu po nikom z vás chtít, aby takto riskoval svůj život," navázal profesor. „Proto jsem se rozhodl, že bránou projdu sám." Z hloučku přítomných lidí se ozvalo překvapené zalapání po dechu.

„Spusťte to!" zavelel profesor a otočil se k zařízení. Ozvalo se hlasité cvaknutí spínaných kontaktů a hluboké bručení transformátorů. Vzduch uvnitř oválu se začal vlnit a tmavnout. Ve vzduchu byl cítit silný elektrostatický náboj. Obraz uvnitř portálu se ustálil a skupinka vědců zírala do tmavé místnosti. Uprostřed ní byla postel, ve které kdosi spal.

Profesor pomalu vykročil k oválu. Naposledy se otočil a kývl svým kolegům na pozdrav. Zhluboka se nadechl a jedním krokem prošel skrz. Obraz ložnice se zavlnil. Pak začal rychle blednout, až zmizel docela ...

Probudil jsem se a posadil se. U postele stála postava. Oči pomalu přivykly šeru a rozeznaly dlouhý plášť a plnovous ...

⋆⋆⋆

„Vzpomínáte na ty vědce z mého snu? Tak už dostavěli to zařízení.“

„Opravdu? A mělo to na vás nějaký účinek?“ zeptal se doktor.

„Myslím, že ano,“ přikývl jsem. „Každopádně se mnou dnes přišel někdo, kdo by se s vámi rád seznámil ... “

21-3-5 Praktická: Rozklad na součty — 10 bodů

V této sérii se nám praktická úloha nevešla do příběhu. Ale nebojte se, o to lépe se vám bude řešit ...

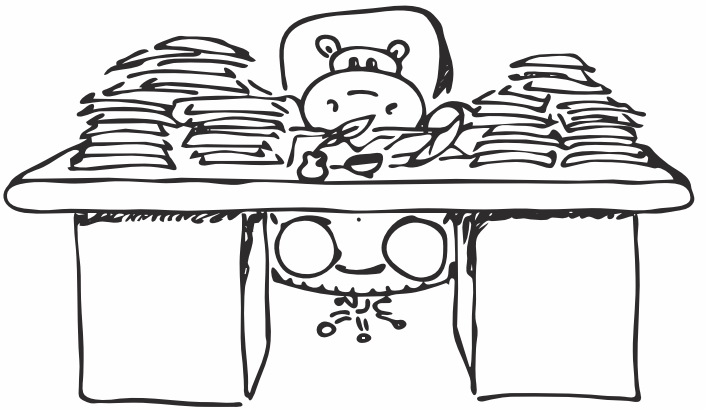
Tuto úlohu odevzdávejte výhradně prostřednictvím webové aplikace CodEx (*https://codex2.ms.mff.cuni.cz/ksp/*). Pokud jste s řešením začali teprve v této sérii a nevíte, co to je CodEx, podívejte se třeba na úlohu 21-1-2 „Optimalizace kotlů“, ve které naleznete úvodní povídání o CodExu.

Zadání:

Na standardním vstupu je zadáno číslo N ($1 \leq N \leq 40$). Vypište na standardní výstup všechny možnosti, jak toto číslo rozložit na součet celých kladných čísel. Každý rozklad musí být uveden na samostatném řádku, sčítance vyjmenovány od nejmenšího k největšímu a odděleny znaménkem „+“. Na pořadí řádků nezáleží.

Například pro $N = 5$ je jeden ze správných výstupů následující:

```
1+1+1+1+1
1+1+1+2
1+1+3
1+2+2
1+4
2+3
5
```



Většina z vás jistě zná fantasy, případně hry, ve kterých každý hraje jednu postavu z oněch světů. Zkusili jste si ale někdy představit, jak by to vypadalo, kdyby se karta obrátila a elfové s trpaslíky si začali hrát na programátory?

V tmavém sklepě se mihotalo světlo svíček. Průvan si hrál s jejich ohněm a stíny vrhaly podivuhodné tvary. Všude vládlo mrtvolné ticho, přerušované jen tichým dýchaním, občasným krysím zapištěním a ... zachřestěním kostek?

„Zase pětka? Nemůžu mít ještě jeden pokus?“

„Ne, tady to máš v pravidlech – 'Každý hráč si na začátku hry naháže schopnosti postavy. Háže se dvacetistěnnou kostkou a jsou na to dva pokusy, z nichž si hráč vybere ten lepší'. Tak už přestaň kňourat, ať popojedem, nemáme na to celou noc!“

„Ale síla 5? Kdo to kdy viděl, aby měl trpaslík sílu pět?“

„Ale to je jenom ve hře, Boendale, to jako nejseš ty ... Koukni, jaké sis zvolil povolání?“

Trpaslík se podíval do svých poznámek. „Haa-keř“, oznámil po chvíli.

„No vidíš, hackři nejsou moc silní. Zato jsou ale hodně inteligentní,“ snažil se Barun dál přesvědčovat hráče.

„Ale jak jako udržím svou sekyru, když budu tak slabej?“ Nemělo to cenu ...

⋆⋆⋆

Všechno začalo asi před týdnem, když mezi starými magickými svazky svého mistra našel tu podivuhodnou knížku: CnH – Computers and Hackers. Zpočátku ji moc nechápal, popisovala jakýsi tuze zvláštní svět. Byl obydlen jenom lidmi, kteří nejenže neovládali magii, ale dokonce v ni ani nevěřili! Místo toho v laboratořích stvořili jakési podivné zařízení, takzvané Počítače, které se pak naučili ovládat pomocí určitých příkazů a tyto se pak staly neoddělitelnou součástí jejich světa. Vypadalo to jako nějaká propaganda těch zatracených alchymistů. Ti se taky vrtali v různých strojích, místo aby se věnovali studiu magie. Už už se chystal knížku zaklapnout, když ho zaujal velký nápis: „Hra roku 2 149 třetího věku! Doporučuje 9 z 10 elfů!“ Ahá, takže hra! No to jsem zvědavý, co na to řeknou Boendal s Mírielem ...

⋆⋆⋆

„Takže projdeme si pravidla ještě jednou, jo? Na začátku hry si každý hráč zvolí povolání. Řekněte mi popořadě, co jste si kdo zvolil.“

„Haa-keř,“ zamumlal mrzutě Boendal, ještě stále zklamaný z toho, že v herním světě s sebou nemůže nosit sekyru.

„Správce sítě. Hele, a můžu mít na sobě alespoň svoji modro-zelenou košilku?“ zeptal se s nadějí v hlase elf Míriel.

„Ne, tady jasně stojí – 'Pro správce sítě je typické vytahané, měsíc neprané, každodenně nošené tričko se vzorem tučňáka, případně ďáblíka.' Hele, hraj postavu, jo? Jinak ti strhnu body!“, sprdnul ho Barun. „Tak jo, další postava?“

„Uaargh!“

„Cože?“

„UAARGH!“

„Říká, že chce hrát algebraického topologa,“ překládal Míriel, „hele, já za to nemůžu, že mně teta hodila na krk hlídání bratrance. To víš, sehnat chůvu pro mladého trolla je dnes docela drahá záležitost . . . “

„No jo, no jo, chápu. Hele, tak já začnu. Takže, nacházíte se . . . třeba na přednášce ve škole – jste teď ještě jenom učňové, jo? Sedíte každý u svého Počítače, když tu najednou Boendalovi přijde ímejl.“

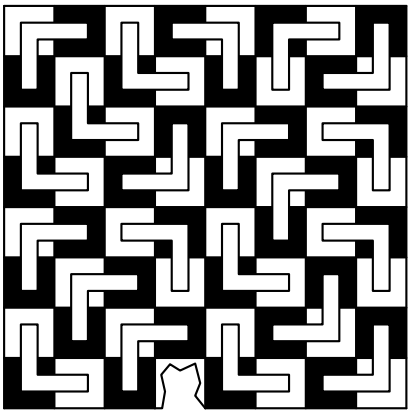
„Cože mi to přišlo?“ zeptal se polekaně Boendal.

„Něco jako dopis. Prostě zpráva. Každopádně když to otevřeš, tak zjistíš, že je to nějaká hra. Co uděláš?“

„Tak si zahrajem, ne?“ navrhnul Míriel a Barun začal vysvětlovat pravidla.

21-4-1 Dláždění šachovnice — 8 bodů

Hra je velice jednoduchá. Hráč na začátku dostane čtvercový plánek, něco jako šachovnici, o hraně velikosti 2^N políček. Jedno políčko ale chybí (libovolné). Úkolem je pokrýt šachovnici útvary podobnými písmenu L složenými ze tří kostiček. L-ka je povoleno otáčet. Vaším úkolem je nalézt algoritmus, jak pokrýt takovýhle plánek, ať je chybějící políčko na libovolném místě. Na obrázku je jedno z možných pokrytí pro $N = 3$.



„Tak jo. Jakmile jste dohráli, objevila se na obrazovce zpráva: ‘Právě jste splnili kvalifikační test na soutěž o Nejlepšího crackera roku 2009!’“

„To jsem se právě kvalifikoval na sušenku?“ vyděsil se Boendal.

„Ach jo, na crackera, ne na krekr.“

„Fuul mít hlad!“ ozval se mladý troll a začal kolem sebe máchat rukama.

Míriel si povzdechl, zamával rukama, zamrmlal si něco pod nosem a krysa v rohu místnosti začala vonět pečeným masem. Fuul se po ní nedočkavě vrhl, pak se usadil a o poznání veselejší pozoroval zbytek družiny.

Barun se ujal slova: „Tak co děláte dál?“

21-4-2 Dosah kouzla — 9 bodů

Kouzlit, to není jen tak. Nejenže to vyžaduje hluboké vědomosti a určitou dávku energie, ale taky má každé kouzlo svůj dosah působnosti. Jaký dosah by muselo mít Mírielovo kouzlo, aby úspěšně zasáhlo krysu, ať by stála v kterémkoli rohu místnosti? On sám seděl taky v rohu místnosti a sklep má půdorys konvexního n-úhelníku. Na vstupu dostanete konvexní n-úhelník (n vrcholů popsaných souřadnicemi $[x, y]$) a vaším úkolem je najít dva nejvzdálenější vrcholy.

„Ne, pořád to nechápu“, trval na svém Boendal.

„Je to prostě taková skříňka a k ní je připojená jiná skříňka a k ní ještě jedna. Vidíš, tady to je na obrázku,“ Barun píchnul prstem do knížky. Už půl hodiny se snažil kamarádům vysvětlit, jak vlastně vypadá takový počítač a co obnáší programování.

„Takže když jako praštím do tady tý skříňky, tak na tamtý se něco objeví?“

„No, v podstatě nějak tak,“ povzdechl si Barun.

„Hele, a co je tady ten ‘Robot’?“ vyzvídal dál.

„Robot? To je takové mechanické zařízení, které za tebe dělá nějakou práci,“ vysvětloval Barun.

„Něco, co maká za mně? To zní hodně zajímavě . . . “ zalesklo se Boendalovi v oku a pustil se do čtení pravidel.

⋆⋆⋆

„Tak ty kabely zapojím tady!“ prohlásil vítězoslavně Boendal.

„No, když myslíš . . . Robot vstal, začal tancovat po okolí, vyházel pár hrníčků od kafe ven z okna, pak zalil kytku a sám se vypnul,“ popsal situaci Barun.

„Paráda, už to skoro funguje!“ těšil se Boendal.

„Hm, možná bychom to neměli jenom tak zkoušet,“ snažil se zapojit do debaty Míriel.

„Ty se do toho nepleť. Už jenom pár pokusů a budeme mít prvního robota na zaplétaní vousů na světě! Nedovedeš si představit, jaká je to otrava dělat to každý den ručně . . . “

21-4-3 Stavění robota — 10 bodů

Boendalova technika stavění robota je vskutku originální. Prostě si vybere nějaké vstupy na základní desce, a ty pak připojí kabely ke zdroji. Při různém propojení dělá robot různé věci. Boendala by zajímalo, kolik různých věcí dokáže robot dělat, když má N výstupů a K kabelů. Nejjednodušší způsob, jak daný problém vyřešit, je spočítat kombinační číslo $\binom{N}{K}$, což se dá rozepsat jako

$$\frac{N \cdot (N-1) \cdot \cdots \cdot (N-K+1)}{K \cdot (K-1) \cdot \cdots \cdot 1}.$$

Nebude to ale tak jednoduché. Protože v informatickém světě se občas stává, že je číslo tak velké, že se nevejde do paměti, bude vaším úkolem spočítat kombinační číslo $\binom{N}{K}$ modulo M, tak, aby se vám mezivýpočet vždy vešel do paměti. Na vstupu dostanete 3 čísla – N, K a M a na standardní výstup vypište výsledek. Předpokládejte, že $0 \leq K \leq N \leq 1\,000\,000$ a $1 \leq M \leq 10\,000$.

Příklad 1: Vstup: `6 2 100` Výstup: `15`

Příklad 2: Vstup: `1000 400 1270` Výstup: `1040`

Tato úložka je praktická, což znamená, že řešení budete odevzdávat výhradně formou odladěného zdrojového kódu. Přesnější zadání a formulář na odevzdání kódu naleznete jako vždy v CodExu. Nevíte-li, co praktická úložka je a jak přesně postupovat, podívejte se do zadání úlohy 21-1-2 „Optimalizace kotlů“.

„BUUM!, ozvalo se v školní laboratoři. Právě jste zničili kus budovy. Z tohohle bude pořádný průšvih ...“ popisoval situaci Barun.

„Tak zdrháme, ne?“ navrhnul trpaslík.

„No, to teda nebylo moc rozumné. Stejně na vás přišli a ... jste podmíněně vyloučeni. Jo, to by mohl být adekvátní trest,“ oznámil jim Barun.

„Hm,“ zamyslel sa Míriel, „a kde jsou uloženy školní záznamy?“

„Myslím, že záznamy jsou uloženy výhradně v elektronické podobě, na centrálním školním počítači. Proč se ptáš?“

„Hehe, jsme přeci nějací hackři, nebo ne?“ usmál se na Baruna elf.

⋆⋆⋆

„Napiš tam ‘heslo’! To bude určitě fungovat!“ povzbuzoval Míriela Boendal, „nebo ‘pás-vord’!“

Už hodinu se snažili dostat na speciální stránky školního systému, ale zatím bez úspěchu.

„Hele, když to nejde logicky, vem větší sekyru. Zkusíme tam napsat postupně všechna možná hesla,“ navrhnul Boendal.

„To zní dobře, ale netrvalo by to příliš dlouho?“ pochyboval Míriel.

„Hm ... Ale mohli bychom tam zkusit zadat každé páté možné heslo, nebo každé sedmé.“

21-4-4 Heslo — 10 bodů

Heslo, to je vlastně permutace nějakých znaků, z nichž se některé můžou opakovat. Řekneme, že permutace p_1 je lexikograficky menší než permutace p_2, pokud první znak, ve kterém se liší (bráno zleva doprava), je na i-té pozici a platí $p_1[i] < p_2[i]$. Příklad: Mějme znaky 1, 2, 3 a 3. Pak jejich permutace 2133 je lexikograficky menší než permutace 2313. Permutace jsou seřazeny lexikograficky, pokud jsou seřazeny od nejmenší po největší. Pokud vygeneruje-

me všechny možné permutace určitých znaků a seřadíme je lexikograficky, pak permutace o K větší než zadaná permutace je permutace na pozici o K větší.

Na vstupu dostanete permutaci cifer (cifry se mohou opakovat) a číslo K a vaším úkolem je najít permutaci o K větší.

Příklad: Vstup: `1234 3` Výstup: `1423`

(protože po permutaci 1243 následuje permurace 1324, pak 1342, no a o 3 větší je permutace 1423).

„Gratuluju, tak jste se právě dostali do Školního informačního systému! Na obrazovce se objevily nějaké znaky – a vy jim vůbec nerozumíte. Vypadá to, že stránka je šifrovaná," oznámil družině Barun. „Co uděláte?"

„Hm, asi by to chtělo nějak líp prostudovat ty znaky. Jak vypadaj?" zajímal se Míriel.

„No, je to velice jednoduché," zalesklo se Barunovi v očích a začal popisovat znaky šifry ...

21-4-5 Znaky — 10 bodů

Znak je reprezentován čtvercem o hraně délky N pixelů, ve kterém je přesně N pixelů vybarvených. Platí ale, že v každém vodorovném, svislém i šikmém směru je vybarvený maximálně jeden pixel. Míriel si ale všimnul, že některé čtverce se opakují a že by se mu hodilo vědět, kolik různých znaků se to vlastně v šifře používá.

Na vstupu dostanete přirozená čísla N a K, a pak K čtverců popsaných výše. Čtverce budou reprezentovány jako dvourozměrné pole integerů: 1 znamená, že pixel je vybarvený, 0, že není. Vaším úkolem je zjistit počet unikátních čtverců tak, aby váš algoritmus měl co nejmenší paměťové nároky (reálné, nejen asymptotické).

„Výborně! Povedlo se vám rozluštit ty znaky a na obrazovce čtete nápis ... "

„Míiíriéééli! Kde to zase vězíš?! Večeře je už hotová a ty jsi ještě nenakrmil draka!!"

„A jeje, máma," povzdechl si Míriel.

„Sakra, to mi připomíná, že bych měl taky pomalu jít. Slíbil jsem bráchovi, že mu pomůžu se skládáním hudby k jeho nejnovějšímu hitu 'Zlato, zlato, zlato!'."

„Hm, tak pro dnešek asi konec. Dohrajem to někdy příště," usmál se Barun a uklidil knížku k sobě do batohu.

A tak se družina rozešla – Míriel šel nakrmit draka, Boendal komponovat hudbu a Fuul se vyvalovat v jeskynním jezírku. A Barun? Hned druhý den si šel k alchymistům půjčit pár knížek. Pár dní na to se několik sousedů stěžovalo na hluk v okolí magické laboratoře a asi za týden byl spatřen, jak za bouřky chytá blesky do velké černé krabice ...

Školní rok se pomalu chýlí ke svému konci a ani my nejsme výjimkou. Přinášíme vám letošní poslední, pátou sérii, tentokrát z důvěrně známého prostředí – telenovelového světa.

Píše se rok 2050. Konečně se podařilo rozpustit poslední zbytky ledovců a hladiny světových oceánů se netriviálně zvedly. Hlavní město hlavního telenovelového státu se začalo potýkat s problémy. A právě v těchto časech začíná příběh našeho hlavního hrdiny Chulia.

21-5-1 Polomáčené mrakodrapy — 10 bodů

Na hlavní město našeho státu – Chuenos Aires – se valí pohroma. Chulianovo rodné město, dříve tak prosperující metropole nudlovitého tvaru, bude brzy zatopeno stoupající vodou z oceánů. Úředníci vlády teď nutně potřebují vědět, kolik úseků města si bude i během záplav moci naladit pravidelnou večerní telenovelu.

Město Chuenos Aires si můžeme představit jako jednu dlouhou ulici, na které je namačkán mrakodrap vedle mrakodrapu. Mrakodrapy jsou tak natěsno, že nemají mezi sebou žádné mezery. Každý mrakodrap (s plochou střechou) má kladnou celočíselnou výšku měřenou v telemetrech nad mořem. Chodník má výšku právě 0 telemetrů nad mořem.

Předpověď počasí hlasí, že zatím jsou všechny mrakodrapy v pořádku, ale počínaje dnem 1 se výška moře zdvihne o jeden telemetr denně. Postupem času se některé věžáky zatopí až po špičku a z hlavní ulice v Chuenos Aires zbudou pouze souvislé úseky mrakodrapů, které ční nad hladinu. Každému takovému úseku stačí jedna anténa na přijímání televize. Vaším úkolem je spočítat, kolik antén bude v jednotlivých dnech potřeba (tedy kolik zbude souvislých bloků mrakodrapů v onen den).

Tato úloha je praktická, takže bližší informace o formátu vstupu programu i ukázkový příklad najdete v CodExu. Povídání o tom, jak se praktická úloha odevzdává, najdete například v zadání úlohy 21-1-2 „Optimalizace kotlů“.

Když i nejvyšší mrakodrap v Chuenos Aires, ve kterém Chulio bydlel, začal být těžko obyvatelný, protože jeho obyvatelé museli každé ráno vyhánět z postele žraloky, rozhodl se Chulio odjet na venkov, kde by začal žít nový život.

A jak se rozhodl, tak udělal. Odešel z města a začal pracovat na kávové plantáži. Tam se brzy seznámil s krásnou Ochechulínou. Začali snít o tom, že se jednou vezmou a sami budou vlastnit podobnou plantáž, každé ráno budou vstávat se šálkem kávy a úsměvem nad dalším dnem. Bohužel zloduch Choachým jim každé snění zkazil, neboť je stále budil před ránem a nutil pracovat.

Z Choachýma se stal úhlavní nepřítel a Chulio s Ochechulínou začali snovat plán pomsty.

Jejich plán byl geniální. Propracovaný do nejmenšího detailu. Skoro. Neměli na něj peníze. A protože prací ještě nikdo nezbohatl, začali vymýšlet, jak na to. Naštěstí bankovní servery v té době ne zcela plánovaně chlazené mořskou vodou vykazovaly o něco vyšší chybovost, a tak stačilo jen trocha šikovnosti a investice ve výši 10 pesos k odlehčení kont nenasytných bankéřů.

21-5-2 Banky — 10 bodů

Aby se podobné odlehčování nekonalo příliš často, nebo pokud možno už nikdy, požádali vás nenasytní bankéři, abyste jim pomohli. Jak vlastně Chulio zbohatnul? Banka obchoduje valutami a má vypsané kurzy pro některé dvojice měn. Banka je hodná a za směnu si neúčtuje žádný poplatek. Pokud ale není dostatečně opatrná, může se stát, že vhodnou posloupností směn získáte více, než jste měli na začátku. Váš program tedy dostane na vstupu směnný kurz banky a měl by rozhodnout, zda je na jeho základě možné zbohatnout výše popsaným způsobem.

Například pro 4 měny a kurzy (zápis $X \to YZ$ znamená, že za 1 jednotku měny X získáte Z jednotek měny Y):

```
1->2 0.8; 2->3 24; 2->1 0.8; 2->4 129; 1->4 0.08; 3->1 0.5
```

lze např. posloupností výměn `1->2`, `2->3` a `3->1` (za jednu jednotku měny 1 získáte po této sérii výměn 9,6 jednotek) na úkor banky vydělat. Takže v tuto chvíli by měl program odpovědět, že směnný kurz je prodělečný. Pokud žádná taková posloupnost neexistuje, váš program by měl odpovědět, že banka bude opět o něco bohatší.

Chulio s Ochechulínou skoupili celou plantáž a mohli dokonat svůj dokonale zosnovaný plán – Choachýma přeřadili na tu nejhorší práci: ochutnávač kávy. Je jasné, že Choachýmovi tato práce nedovolila pořádně se vyspat, takže nedostatkem spánku psychicky narušený Choachým začal vymýšlet svůj plán pomsty.

Chulio a Ochechulína vedli šťastný život. Zcela se jim splnil sen. To ale netrvalo dlouho. Bratr Chulia Chosé se dozvěděl o Chuliově úspěchu a přicestoval za ním. Hodný Chulio zaměstnal Chosého a pověřil ho úkolem vybudovat vedle plantáží farmu pro dobytek.

21-5-3 Krávy — 9 bodů

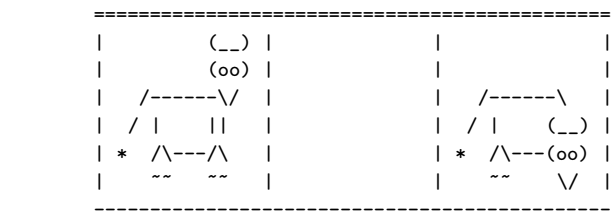
Navrhovat stáje pro krávy není vůbec jednoduchý úkol. Navíc máte-li omezené množství prostředků. Chuliův kravín se skládá z řady boxů stejných rozměrů těsně vedle sebe. V každém boxu může bydlet nejvýše jedna kráva. Tyto boxy jsou z jedné (stejné) strany vždy otevřené. Vaším úkolem je rozmístit před tyto boxy závory tak, aby každý, ve kterém bydlí kráva, byl přehrazený.

Závor ale máte k dispozici pouze určitý počet. Naštěstí každá závora může být libovolně dlouhá a není zakázáno přehradit i boxy, ve kterých žádná kráva nebydlí.

Navrhněte algoritmus, který pro zadaný kravín (počet boxů a rozmístění krav v jednotlivých boxech) nalezne takové rozmístění závor, aby součet jejich délek (jeden box má šířku jeden telemetr) byl minimální a všechny boxy s kravami byly přehrazeny. Pokud má úloha více řešení, stačí nalézt libovolné z nich.

Příklad: Mějme kravín s 8 boxy, číslovanými od 1 do 8. Krávy bydlí v boxech 1, 3, 5, 6 a 8 a k dispozici máme 3 závory. Pak je jedno z optimálních řešení přehradit jednou závorou boxy 1 až 3, druhou závorou 5 až 6 a třetí závorou box 8. Celková délka závor tak bude 6 telemetrů, neboť první závora přehrazuje 3 boxy, druhá 2 a třetí pouze jeden.

Takto například vypadají boxy 1, 2 a 3:



Chosé byl ze začátku spokojený, a tak navrhoval stáje, obdělával pastviny, kupoval dobytek a farma vzkvétala. Jenže brzy začal pociťovat závist, protože dřel na svého bratra, který jenom popíjel kávu a trávil čas s krásnou Ochechulínou, po které Chosé stále více toužil.

Ochechulínu to uvádělo do rozpaků, ale příliš se návrhům Chosého nebránila, neboť narozdíl od Chulia neměl hrb, měl obě ruce stejně dlouhé a také měl delší . . . vlasy.

Ani Choachým během probdělých dní a nocí nemarnil čas a stále promýšlel svůj plán pomsty. Protože zbohatnout stejně jako Chulio již nebylo možné, chtěl k pomstě využít to, čeho měl nejvíc. Šarm. Rozhodl se svést Chulia, následně jej obžalovat z obtěžování a potom převzít vládu nad farmou i Ochechulínou.

Plán se mu podařil a Chulio skončil se zlomeným srdcem u soudu. Zatímco se Chulio pokoušel prokličkovat ve spleti zákonů, Choachým seč mohl pospíchal na farmu.

21-5-4 Zákony — 12 bodů

Obhájit se u soudu bývá podobné jako vymotat se z hustého a temného lesa. Všude kolem jsou stromy, totiž jednotlivé zákony, a čím těžší přestupek jste

spáchali, tím obtížnější je se mezi nimi prosmyknout a proniknout z právního lesa zpět na svobodu.

Napište program, který na vstupu dostane popis spleti zákonů a velikost vašeho přestupku. Program pak odpoví, zda je možné se u soudu obhájit nebo ne.

Program dostane dvě čísla: přirozené N a kladné reálné R. Číslo N udává počet zákonů a R velikost vašeho hříchu. Dále dostane vaše výchozí souřadnice v lese X a Y a popis jednotlivých stromů (zákonů). Každý strom je zadán svými souřadnicemi. Všechny souřadnice jsou reálná čísla.

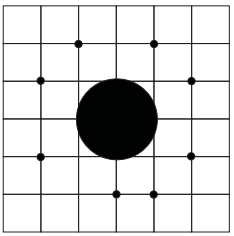
Hodnota R určuje, jak nejvíc se na cestě ven z lesa můžete přiblížit k libovolnému stromu. Předpokládejte, že stromy mají nulový průměr a jsou nekonečně vysoké a že výchozí pozice je vždy korektní, neboli že na začátku nejste v kolizi s žádným stromem.

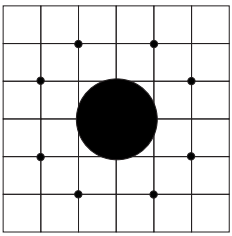
Úlohu lze formulovat i tak, že zjišťujete zda se koule o poloměru R může vykulálet ven ze zadaného lesa.

A co znamená dostat se ven z lesa? Třeba mít možnost dostat se do bodu se souřadnicemi (∞, ∞).

Například pro $N = 8$, $R = 1{,}1$, $X = 0$, $Y = 0$ a stromy $(-2, 1)$, $(-1, 2)$, $(1, 2)$, $(2, 1)$, $(2, -1)$, $(1, -2)$, $(0, -2)$, $(-2, -1)$ se z lesa dostat lze.

Pro $N = 8$, $R = 1{,}1$, $X = 0$, $Y = 0$ a stromy $(-2, 1)$, $(-1, 2)$, $(1, 2)$, $(2, 1)$, $(2, -1)$, $(1, -2)$, $(-1, -2)$, $(-2, -1)$ se z lesa dostat nejde.





Choachým dorazil, právě když vládu nad Ochechulínou přebíral Chosé, využívaje k tomu svých dlouhých vlasů. Takové setkání prostě muselo skončit soubojem. Chosé proklál Choachýma svojí cestovní šavlí.

21-5-5 Cestovní šavle — 10 bodů

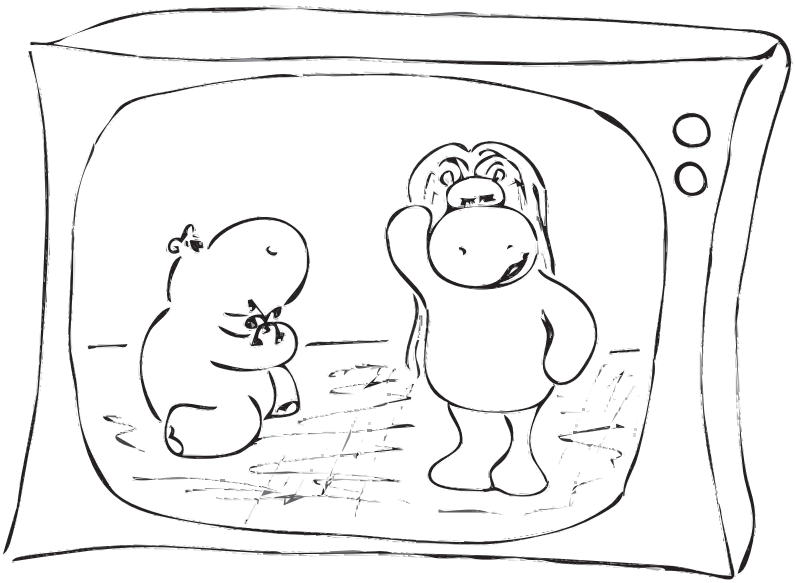
Určitě znáte klasický zednický metr. Ten se skládá z několika segmentů. Vždy dva sousední segmenty jsou spojeny pantem, takže je metr možné rozložit, pokud měříme nějakou vzdálenost, nebo složit, pokud ho přenášíme. Přesně tímto způsobem funguje Chosého cestovní šavle. Jenom s tím rozdílem, že délky segmentů nejsou stejné.

Napište program, který pro zadané délky jednotlivých segmentů šavle zjistí, zda je možné šavli poskládat do pouzdra zadané délky. Program dostane dvě

přirozená čísla N a L. Číslo N je rovno počtu segmentů šavle a L je délka pouzdra. Následuje N přirozených čísel, které postupně udávají délky segmentů šavle zleva doprava. Můžete předpokládat, že N i $L \leq 10000$. Program by měl odpověď „Ano“ nebo „Ne“ podle toho, zda je možné šavli složit do pouzdra.

Například pro $N = 3$, $L = 6$ a délky $6, 3, 3$ je možné šavli poskládat. Pro stejné N i L, ale pro délky $6, 3, 4$ šavli poskládat nelze.

Mezitím se od soudu vrátil Chulio. Když se dozvěděl, co se tu stalo, využil toho, že se Chosému stále nedařilo složit šavli zpět do pouzdra a proklál s ní i svého proradného bratra. Od té doby již žil Chulio s Ochechulínou šťastně až do konce ... až do konce 14223. dílu, kdy se ukázalo, že předchozích 14221 dílů bylo jenom snem a Chulio s Ochechulínou se probouzí do dalšího pracovního dne na farmě.



Programátorské kuchařky

20-2-K Kuchařka druhé série – Rozděl a panuj

Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj*. A tak by se slušelo začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce“ v druhé sérii 20. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
type Pole=array[1..MaxN] of Integer;        {budeme třídit takováto pole}

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin                          {pivotem se stane poslední prvek úseku      }
 x:=a[r];                      {hodnota pivota                             }
```

```
  i:=l-1;                    {a[i] bude vždy poslední <= pivotovi        }

  for j:=l to r-1 do         {samotné přerovnávání                       }
   if a[j]<=x then           {právě probíraný prvek                      }
   begin                     {menší/rovný hodnotě pivota                 }
    Inc(i);                  {pak zvyš ukazatel                          }
    q:=a[j];                 {a proveď přerovnání prvku                  }
    a[j]:=a[i];
    a[i]:=q;
   end;

  q:=a[r];                   {nakonec přesuneme pivota za poslední <=    }
  a[r]:=a[i+1];
  a[i+1]:=q;
  prer:=i+1;                 {vrátíme novou pozici pivota                }
end;

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
 if l<r then                 {máme ještě co dělat?                       }
 begin
  m:=prer(l,r);              {přerovnej, m pozice pivota                 }
  QuickSort(l,m-1);          {setřiď prvky napravo                       }
  QuickSort(m+1,r);          {setřiď prvky nalevo                        }
 end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $O(N + (N-1) + (N-2) + \ldots + 1) = O(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $O(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $O(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které

dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $O(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $O(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = O(\log N)$. Místo $1/4$ by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se tak často QS implementuje.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností $1/2$ to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních algoritmech v 16. ročníku). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $O(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme

vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $O(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $O(N + N/2 + N/4 + \ldots + 1) = O(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
 x:=prer(a,l,r);                  {přerovnej, x je pozice pivota}
 z:=x-l+1;                       {pozice pivota vzhledem k [l..r]}
 if k=z then
   kty:=a[x]                              {k-tý nejmenší je pivot}
 else if k<z then
   kty:=kty(a,l,x-1,k)                   {k-tý nejmenší je nalevo}
 else
   kty:=kty(a,x+1,r,k-z);                                 {napravo}
end;
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

Řečeno s panem Pascalem:

```
{přerovnávací funkce, která dostane hodnotu pivota jako parametr   }
function prerp(var a:Pole;
               l,r,m:Integer):Integer;
var q,p:Integer;
begin                                        {nalezneme pozici pivota}
 p:=l;
 while a[p]<>m do
  inc(p);
 q:=a[p]; a[p]:=a[r]; a[r]:=q; {pivota prohodíme s posledním prvkem}
 prerp := prer(a,l,r);         {a zavoláme původní přerovnávací fci}
end;

{hledání k-tého nejmenšího prvku z a[l..r]}
```

```
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;                                  {pole pro mediány pětic}
    i,j,q,x,pocet,m,z:Integer;
begin
 pocet:=r-l+1;                                {s kolika prvky pracujeme}

 if pocet<=1 then                                    {pouze jeden prvek?}
  kth:=a[l]                                   {výsledek nemůže být jiný}
 else if pocet<6 then begin                           {méně než 6 prvků}
  QuickSort(a,l,r);
  kth:=a[l+k-1];
  end
 else begin                                  {mnoho prvků, jde to tuhého}
                                               {rozdělíme prvky do pětic}
  q:=1;                                         {zatím máme jednu pětici}
  i:=l;                                         {levý okraj první pětice}
  j:=i+4;                                     {pravý okraj první pětice}
  while j<=r do begin                           {procházíme celé pětice}
   QuickSort(a,i,j);
   medp[q]:=a[i+2];                                      {medián pětice}
   Inc(q);                                             {zvyš počet pětic}
   Inc(i,5);                                   {nastav levý okraj pětice}
   Inc(j,5);                                  {nastav pravý okraj pětice}
   end;
                        {případnou neúplnou pětici můžeme ignorovat}


  m:=kth(medp,1,q-1,q div 2);          {najdeme medián mediánů pětic}

  x:=prer(a,l,r,m);           {přerovnej a zjisti, kde skončil pivot}
  z:=x-l+1;                                    {pozice vzhledem k [l..r]}
  if k=z then
   kth:=m                                       {k-tý nejmenší je pivot}
  else if k<z then
   kth:=kth(a,l,x-1,k)                            {k-tý nejmenší nalevo}
  else
   kth:=kth(a,x+1,r,k-z);                                      {napravo}
 end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze

dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = O(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo mužeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A + B) \cdot (C + D) = AC + AD + BC + BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = O((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$. Konstanta d se nám „schová do O-čka“, takže algoritmus má časovou složitost přibližně $O(n^{1.58})$. Existují i rychlejší algoritmy se složitostí až $O(n \log n)$, ale ty jsou mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom (viz kuchařka v 5. sérii minulého ročníku) vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $O(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

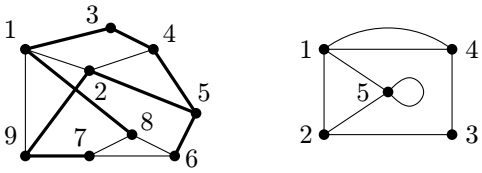
20-3-K Kuchařka třetí série – grafy

V dnešním dílu kuchařky si zavedeme základní pojmy z teorie grafů a ukážeme si, jak řešit problém nalezení minimální kostry grafu. Také si popíšeme datovou strukturu *Disjoint-Find-Union* (její název je často zkracován na DFU), kterou šikovně použijeme právě na řešení tohoto problému.

Grafy

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, přičemž hrany jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více

než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a jeho kostra; multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y: Každý sled, který není cestou, obsahuje nějaký vrchol u dvakrát, nechť $u = v_i = v_j$, $i < j$. Z takového sledu ale můžeme vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostat také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

Kružnicí nazýváme cestu délky alespoň 3, ve které oproti definici platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

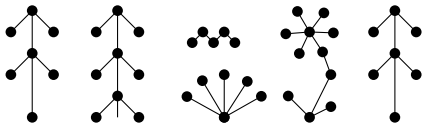
Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé*. Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi

kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti.*

Teď se podívejme na pár grafů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musejí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



Les, jak ho vidí matematici

Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale informatici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Ještě se nám bude hodit nahlédnout, že strom s n vrcholy má právě $n-1$ hran: Budeme postupovat matematickou indukcí podle počtu vrcholů stromu. Strom s jedním vrcholem neobsahuje žádnou hranu. Pokud máme strom s $n > 1$ vrcholy, vezměme libovolný jeho list a odeberme ho ze stromu. Tím získáme opět strom (souvislost jsme porušit nemohli a kružnici jsme také nevytvořili) a jeho počet vrcholů je o 1 menší. Podle indukčního předpokladu má o jednu hranu méně než vrcholů. Nyní list „přilepíme“ zpět, čímž zvýšíme počet vrcholů i hran o 1, a tvrzení stále platí.

A nyní k slibovaným kostrám. Mějme nějaký souvislý graf. Jeho *kostrou* nazveme libovolný podgraf, který obsahuje všechny vrcholy a nejmenší počet hran takový, aby každé dva vrcholy byly spojeny nějakou cestou. Všimněte si, že kostra musí být sama souvislá a navíc neobsahuje žádnou kružnici (jinak bychom mohli libovolnou hranu ležící na kružnici z kostry beze škody odebrat, čímž bychom získali menší kostru, a to nám definice zakazuje.) Čili každá kostra je strom. Na prvním obrázku je kostra levého grafu znázorněna silnými hranami.

Pokud každou hranu grafu ohodnotíme nějakou *vahou,* což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf.* V takových grafech

pak obvykle hledáme mezi všemi kostrami *kostru minimální,* což je taková, pro kterou je součet vah jejích hran nejmenší možný. Graf může mít více minimálních koster – například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n - 1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální. Pokud si graf představíme jako města spojená silnicemi, problém nalezení minimální kostry můžeme vidět následovně: Chceme určit silnice, které se budou v zimě udržovat sjízdné tak, aby součet délek silnic, které je třeba udržovat, byl co nejmenší možný a zároveň se stále bylo možné přepravit mezi každými dvěma městy.

Algoritmus pro hledání minimální kostry

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: Pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $O(M \log M)$ (použijeme některý z rychlých třídících algoritmů popsaných v jednom z minulých dílů kuchařky) a poté se pokusíme přidat každou z M hran. V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $O(M \log M)$. Celková časová složitost našeho algoritmu je tedy $O(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $O(M)$.

Důkaz správnosti hladového algoritmu

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídícím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e, která je v T_{alg}, ale není v T_{min}. Ze všech takových hran si vyberme tu, která má

nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e, vidíme, že sestrojil nějakou částečnou kostru F, která je ještě součástí jak T_{min}, tak T_{alg}.

Přidejme nyní hranu e ke kostře T_{min}. Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jeda hrana e', která není v T_{alg}.

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e: hrana e' neleží v T_{min} na žádném cyklu, takže tím spíš netvoří cyklus v F, a kdyby ji algoritmus zpracoval, musel by ji přidat do F, což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e. Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e, musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min}, což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Disjoint-Find-Union

Datová struktura DFU slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele (trochu nezvykle) od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách re-

prezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme se ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce $root(v)$ vrátí kořen stromu, který obsahuje prvek v.

```
var parent:array[1..N] of integer;

procedure init;
var i:integer;
begin
  for i:=1 to N do parent[i]:=0;
end;

function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else root:=root(parent[v]);
end;

function find(v,w:integer):boolean;
begin
  find:=(root(v)=root(w));
end;

procedure union(v,w:integer);
begin
  v:=root(v);
  w:=root(w);
  if v<>w then parent[v]:=w;
end;
```

S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“ a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $O(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.

- **path compression:** Ve funkci *root*(v) přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:

```
var parent:array[1..N] of integer;
    rank:array[1..N] of integer;

procedure init;
var i:integer;
begin
  for i:=1 to N do
    begin
      parent[i]:=0;
      rank[i]:=0;
    end;
end;

{zmena kvuli path compression}
function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else begin
    parent[v]:=root(parent[v]);
    root:=parent[v];
  end;
end;

{stejna jako minule}
function find(v,w:integer):boolean;
begin
  find:=(root(v)=root(w));
end;

{zmena kvuli union by rank}
procedure union(v,w:integer);
begin
  v:=root(v);
  w:=root(w);
  if v=w then exit;
  if rank[v]=rank[w] then
    begin
      parent[v]:=w;
      rank[w]:=rank[w]+1;
    end
  else
    if rank[v]<rank[w] then
      parent[v]:=w
    else
```

```
      parent[w]:=v;
end;
```

Zaměřme se nyní blíže na metodu *union by rank*. Nejprve učiníme následující pozorování: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků. Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r - 1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r - 1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali. Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank*, je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $O(\log N)$, a tedy operace *find* a *union* stihneme v čase $O(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $O(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $O(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Nejdříve si předveďme tento pojem na jednoduchém příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že N přičení jedničky k číslu, které je na počátku nula, zabere čas $O(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $O(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $O(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek a pokud jich na N operací použijeme jen $O(N)$, bude tvrzení dokázáno. Každé jedničce, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou

jedničku na nulu a pokračuje u dalšího bitu, atd. Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $O(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $O(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $O(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $O(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $O((N+L)\log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (k-1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = O(\log^* N)$ skupin. Odhadněme shora počet prvků v k-té skupině:

$$\frac{N}{2^{(2 \uparrow (k-1))+1}} + \cdots + \frac{N}{2^{2 \uparrow k}} = \frac{N}{2^{2 \uparrow (k-1)}} \cdot \left(\sum_{i=1}^{2 \uparrow k - 2 \uparrow (k-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2 \uparrow k}.$$

Teď můžeme provést časovou analýzu funkce $root(v)$. Čas, který spotřebuje funkce $root(v)$, je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce $root(v)$, a ty, které zahrneme do faktoru $O(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce $root(v)$ započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $O(\log^* n)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow k$ přepojeních je rodič prvku v v $(k+1)$-ní nebo vyšší skupině. Pokud v je prvek v k-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce $root(v)$ nejvýše $(2 \uparrow k)$-krát. Protože k-tá skupina obsahuje nejvýše $N/(2 \uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $O(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce $root(v)$, nejvýše $O(N \log^* N)$. Protože funkce $root(v)$ je volána $2L$-krát, plyne časový odhad $O((N+L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Jednoparametrová Ackermannova funkce $A(k)$ je pak rovna hodnotě $A_k(2)$, čili $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd... Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

20-4-K Kuchařka čtvrté série – halda a Dijsktrův algoritmus

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými

čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
```

```
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

```
void přidej (typ_klíče klíč, typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
    !stejný(klíč, heš[index].hodnota))
      return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

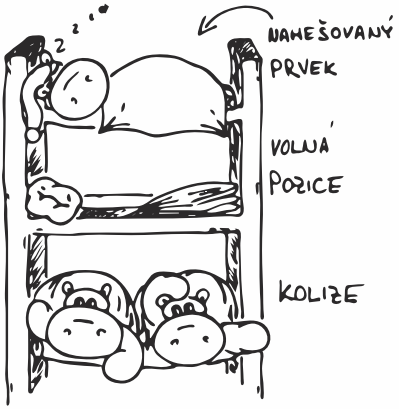
Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
```

```
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je,ale ne
    // to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T+H)$. A budeme-li

schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                          \
  a-=b; a-=c; a^=(c>>13);                     \
  b-=c; b-=a; b^=(a<< 8);                     \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);        \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);        \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff; \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff; \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff; \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff; \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba:

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací" heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Pár poznámek na závěr:

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé příhrádce (příhrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se" za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nová heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

20-5-K Kuchařka páté série – vyhledávací stromy

V poslední kuchařce tohoto ročníku se budeme zabývat převážně rekurzí a dynamickým programováním. O čem že je řeč? Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe. Dynamické programování pak bude technika, kterou často půjde z exponenciálně pomalého rekurzivního algoritmu vyrobit pěkný polynomiální. Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze:

Fibonacciho čísla

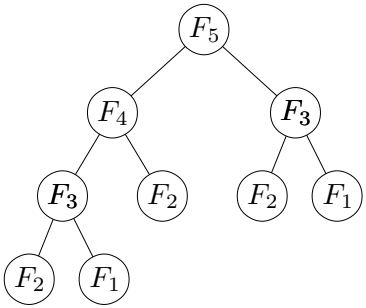
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice: zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
function Fibonacci(n: Integer): Integer;
begin
  if n <= 2 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end;
```

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



Vidíme, že program se rozvětvuje a tvoří strom volání. Všimněme si také, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího.

Pokusme se odhadnout časovou složitost T_n naší funkce. Pro $n = 1$ a $n = 2$ funkce skončí hned, tedy v konstantním (řekněme jednotkovém) čase. Pro vyšší n zavolá sama sebe pro dva předchozí členy plus ještě spotřebuje konstantní čas na sčítání:

$$T_n \geq T_{n-1} + T_{n-2} + const, \text{ a proto } T_n \geq F_n.$$

Tedy na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že:

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

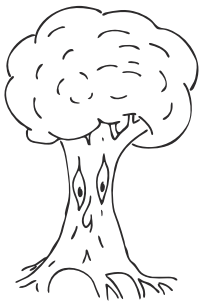
z čehož plyne:

$$F_n \geq 2^{n/2}.$$

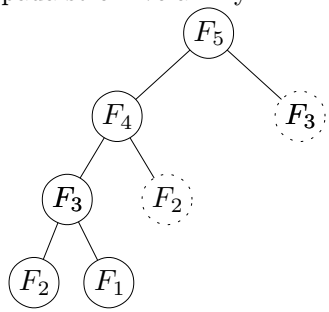
Funkce `Fibonacci` má tedy exponenciální časovou složitost, což není nic vítaného. Ovšem jak jsme už řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka z klobouku s minimem námahy.

Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

```
var P: array[1..MaxN] of Integer;
function Fibonacci(n: Integer): Integer;
begin
  if P[n] = 0 then
  begin
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```



Podívejme se, jak vypadá strom volání nyní:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou zlepšili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ale při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, a na to samotné potřebujeme určitě paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoliv známe $P[1] = F_1, \ldots, F_k = P[k]$, dokážeme snadno spočítat i $P[k+1] = F_{k+1}$:

```
function Fibonacci(n: Integer): Integer;
var
  P: array[1..MaxN] of Integer;
  I: Integer;
begin
  P[1] := 1;
  P[2] := 1;
  for I := 3 to n do
    P[I] := P[I-1] + P[I-2];
  Fibonacci := P[n]
end;
```

Zopakujme si, co jsme postupně udělali: nejprve jsme vymysleli pomalou rekurzivní funkci, tu jsme zrychlili zapamatováváním si mezivýsledků a nakonec jsme celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení (a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty a paměťovou složitost tak zredukovat na konstantní), ale zmíněný obecný postup zrychlování rekurze nebo rovnou řešení úlohy od nejmenších

podproblémů k těm největším – obvykle se mu říká *dynamické programování* – funguje i pro řadu složitějších úloh. Třeba na tuto:

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, a přitom nepřekročil M. Předvedeme si algoritmus, který tento problém řeší v čase $O(MN)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i. Před prvním krokem (po nultém kroku), jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1. Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme: v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty, atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k). Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů. Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k-tém kroku. Potom ale hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i. Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X. Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M. Nalézt jednu množinu této hmotnosti také není obtížné: protože v k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, tak v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Zdrojový kód tohoto algoritmu lze nalézt níže.

Časová složitost algoritmu je $O(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $O(M)$. Paměťová složitost činí $O(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

```
var N: word; {   počet předmětů   }
    M: word; { hmotnostní omezení }
    hmotnost: array[1..N] of word;
              { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: word;
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then
        A[i]:=k;
  i:=M;
  while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
  write('Předměty v množině:');
  while A[i]<>-1 do
    begin
      write(' ',A[i]);
      i:=i-hmotnost[A[i]];
    end;
  writeln;
end.
```

Na rozmyšlenou: Proč pole A procházíme pozadu a ne popředu?

Nejkratší cesty a Floydův-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů (o grafech se dočtete například v kuchařce třetí série), ale zkusíme si ho nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) *silnice,* jejichž (nezáporné) délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst. *Cestou* rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují. [V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.]

Půjdeme na to následovně: Vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu).

V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolná z měst $1, \ldots, k$. V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo nová cesta přes město k. Pokud nejkratší cesta prochází přes město k, můžeme si ji rozdělit na nejkratší cestu z i do k a nejkratší cestu z k do j. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$. Takže pokud je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem, jinak ji ponecháme.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j. Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $O(N^2)$. Celková časová složitost našeho algoritmu tedy je $O(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $O(N^2)$. Program bude vypadat následovně:

```
var N:word; { počet měst }
    D:array[1..N] of array[1..N] of longint;
      { délky silnic mezi městy, D[i][i]=0,
        místo neexistujících je "nekonečno" }
    i,j,k:word;
begin
  for k:=1 to N do
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
        D[i][j]:=D[i][k] + D[k][j];
end.
```

Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi. To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k). Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

Na rozmyšlenou:

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞, je `maxint`. To ovšem nebude fungovat, protože $\infty + \infty$ přeteče. Stačí `maxint div 2`?

- Hodnoty v poli si sice přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?
- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)? Inu, to samozřejmě nevíme, ale všimněte si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá ... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. (Což, pokud bychom chtěli být přesní, musíme přidat do předpokladů našeho algoritmu.)
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní ... jenže pak samozřejmě nebude fungovat.

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B. Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat. Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká. Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti,

které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem. Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost, takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P, jelikož v libovolném rozšíření Q-čka můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě, a dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l+1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný: Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a+1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B. Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti. Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněte si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Takhle vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP). Ukážeme si to na našem příkladu: jelikož poslední nenulové číslo na posledním řádku je ve 12. sloupci, má hledaná NSP délku 12. $D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[11, 7]$, třetí z $D[9, 6]$, atd. Jednou z hledaných podposloupností je:

poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Ještě trochu konkrétněji:

```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;  { Délky posloupností }
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, T: Integer;
begin
  ...
  if LA > LB then  { A bude kratší z obou }
  begin
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
  end;

  for I := 1 to LA do
    D[0, I] := LB;

  L := 0;
  for I := 1 to LA do
  begin
    for J := 1 to LA do
      D[I, J] := D[I-1, J];

    L := 1;
    for J := 0 to LB-1 do
      if B[J] = A[I-1] then
      begin
        while D[I-1, L] < J do Inc(L);
```

```
          if D[I, L] >= J then
            D[I, L] := J;
        end;
    end;

    LC := L;
    J := LA;
    for I := LC downto 1 do
    begin
      while D[J-1, I] = D[J, I] do Dec(J);
      C[I-1] := A[J-1];
      Dec(J);
    end;
    ...
  end.
```

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $L(A)$ a $L(B)$, což jsou délky posloupností A a B. Vnořený cyklus while proběhne celkem maximálně $L(A)$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $O(L(A) \cdot L(B))$. Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti a tedy i velikost pole s daty je kvadrát této délky. Paměťovou složitost odhadneme $O(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.
Na rozmyšlenou: Proč jsme si z více posloupností zapamatovali zrovna tu, která v B končí nejlevějším možným prvkem?

Vzorová řešení

21-1-1 Ohniště — Roman Smrž

Ukázalo se, že dostat se do pekla pro většinu z vás nijak velký problém nebude (nakonec, problém to nemusí být ani pro ty ostatní ...). Ohniště si totiž můžeme, jak jste si téměř všichni všimli, snadno rozdělit na $n-2$ trojúhelníků – jelikož jsou jeho vrcholy na vstupu zadány postupně ve směru hodinových ručiček (označme je $A_1, A_2, \ldots, A_n$), můžeme vzít trojúhelníky $A_1A_2A_3, A_1A_3A_4, \ldots, A_1, A_{n-1}, A_n$.

Zbývá už jen určit obsahy jednotlivých trojúhelníků a ty následně posčítat. K tomu mnozí z vás použili Heronův vzorec využívající délek stran trojúhelníka (ty můžeme zjistit pomocí Pythagorovy věty).

O něco elegantnější (bez používání odmocnin v programu) je využití vektorového součinu: máme-li body A, B a C a vektory $\vec{u} = B - A$ a $\vec{v} = C - A$, je absolutní hodnota jejich vektorového součinu rovna dvojnásobku obsahu trojúhelníka ABC (jelikož počítáme v rovině, položíme $a_3 = b_3 = c_3 = u_3 = v_3 = 0$); máme tedy:

$$\vec{u} \times \vec{v} = (u_2 \cdot 0 - 0 \cdot v_2, 0 \cdot v_1 - u_1 \cdot 0, u_1v_2 - u_2v_1) =$$

$$= (0, 0, u_1v_2 - u_2v_1)$$

Odkud již snadno zjistíme kýžený obsah trojúhelníka (ke stejnému vzorci lze dospět také využitím determinantu matice):

$$S_{\triangle ABC} = \frac{1}{2}|\vec{u} \times \vec{v}| = \frac{1}{2}\sqrt{0^2 + 0^2 + (u_1v_2 - u_2v_1)^2} =$$

$$= \frac{1}{2}|(b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1)|$$

Spočítání obsahu každého z trojúhelníků zvládneme v konstantním čase, a jelikož je jich celkem lineárně s počtem vrcholů, je i časová složitost $\mathcal{O}(N)$. Jednotlivé obsahy lze zjišťovat postupně při načítání vstupu, který (s výjimkou prvního vrcholu) již na nic dalšího nepotřebujeme, takže si vystačíme s konstantní pamětí.

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

int main(void) {
        int n;
        float obsah = 0.0;
```

```
        float a1, a2, b1, b2, c1, c2; // souřadnice bodů A, B a C
        // načteme počet vrcholů a souřadnice prvních dvou:
        scanf("%d", &n);
        scanf("%f%f%f%f", &a1, &a2, &c1, &c2);
        // pro následující vrcholy již počítáme obsahy trojúhelníků:
        for (int i = 2; i < n; i++) {
                b1 = c1; b2 = c2;
                scanf("%f%f", &c1, &c2);
                obsah += fabs((b1-a1)*(c2-a2) - (b2-a2)*(c1-a1))/2.0;
        }
        printf("%f\n", obsah);
        return 0;
}
```

21-1-2 Optimalizace kotlů — Martin „Bobřík“ Kruliš & CodEx

„Můžu?“
„Můžeš.“
„Držíš?“
„Držím.“
„Spouštěj!“
„Spouštím.“
„Máš ho?“
„Mám!“
„Tady dobrý! Na řadě je kotel číslo 421954 *. . . “*

Jak jste měli možnost si na vlastní kůži vyzkoušet, přesouvání hříšníků mezi kotli opravdu není snadná záležitost. Pozor si musíme dávat hned na několik věcí:

Zkusíme na to jít nejprve přímočaře. Projdeme si celé pole kotlů a provedeme přesuny těch hříšníků, jejichž cílové políčko je volné. Tohle budeme opakovat tak dlouho, dokud nebudou všichni na svých místech. Každého hříšníka přesouváme právě jednou, a to přímo na místo, kde se má nacházet, takže na první pohled je algoritmus konečný a vrací korektní výstup.

Ale ouha, co když se nám hříšníci zrovna sejdou např. takto: Hříšník z kotle 1 musí do 2, hříšník z 2 musí do 3 a konečně z 3 je potřeba provést přesun do 1. Tyto 3 přesuny tvoří cyklus a náš výše uvedený algoritmus na ně nebude fungovat. Pokaždé, když bychom chtěli přesunout některého z výše uvedených hříšníků, bude v jeho cílovém kotli trůnit jiný hříšník. Musíme na to tedy jinak . . . .

Ze zadání je jasné, že alespoň jeden kotel musí být volný (jinak by nešlo s hříšníky vůbec pohnout). Zkusíme tedy využít tohoto garantovaného volného kotle. Postupně budeme brát hříšníky na přesun. Pokud je cílový kotel volný, není s přesunem žádný problém. Pokud je ale obsazený, přesuneme *překážejícího*

hříšníka do libovolného volného kotle a tím se nám uvolní cílový kotel, takže můžeme opět přesun provést. Všimněte si, že pokud je zadání korektní, nikdy nepřesouváme nikoho, kdo je již na svém cílovém místě. V každém kroku tedy snížíme počet hříšníků, kteří ještě nejsou na svém místě, o 1 a algoritmus je tedy opět konečný (korektnost je zřejmá).

Teď již máme řešení, které bude fungovat, ale musíme se zamyslet, zda splňuje požadavek na minimální počet přesunů. Jak už jistě tušíte – nesplňuje. Představme si, že máme přesunout hříšníka z kotle 1 do 2, z kotle 2 do 3 a z 3 do 4, přičemž kotel 4 je prázdný. Budeme-li postupovat přesně podle našeho algoritmu, budeme nejprve přesouvat hříšníka 1. Jenže kotel 2 je obsazen, takže je potřeba nejprve přesunout 2 do 4, abychom si kotel uvolnili. Dále chceme umístit druhého hříšníka do správného kotle (č. 3), jenže ten je jako na potvoru opět obsazen a musí být uvolněn. Sami si rozmyslete, že toto pořadí si vyžádá celkem 5 přesunů. Přitom bychom to ale určitě zvládli jen na 3 přesuny: 3 do 4, 2 do 3 a 1 do 2.

Jak je vidět, nestačí nám přímočarý algoritmus, ale potřebujeme nějaký, který zohlední plánování, abychom nic nepřesouvali zbytečně. V ideálním případě tedy budeme přesouvat hříšníky přímo do jejich cílových kotlů bez meziskladování. Problém nastává s cykly. Když narazíme na cyklus, musíme jej nejdřív rozbít (jednoho hříšníka z něj přesuneme do meziskladiště). Tím z něj vznikne *cesta*, kterou snadno vyřešíme, a následně přesuneme hříšníka z meziskladu na jeho cílové místo.

Cestou budeme rozumět posloupnost přesunů k_i, kdy se k_i-tý hříšník přesouvá do kotle číslo k_{i+1} a poslední (k_n-tý) hříšník se přesouvá do volného kotle. Cesty budeme zpracovávat tak, že je celé projdeme až na konec a přesuny budeme provádět od posledního k prvnímu.

Ve finále zbývá tuto myšlenku již jen naprogramovat. Pokud si nejste jisti, jak na to, podívejte se na přiložený program. Při troše snahy při implementaci lze docílit časové i paměťové složitosti $O(N)$.

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

int N = 0;                      // Počet kotlů.
int *data;                      // Údaje o přesunech (pozn: pole má
                                // o jeden prvek víc a indexujeme jej od 1).
int free_place = 0;             // Index posledního nalezeného volného kotle
                                // (pro dočasné odkládání hříšníků).

// Načte data ze vstupního souboru do pole "data".
void load_data() {
      FILE *fp = fopen("kotle.in", "r");
      fscanf(fp, "%d\n", &N);
      data = (int*)malloc(sizeof(int) * (N+1));
```

```
      for(int i = 1; i <= N; i++)
            fscanf(fp, "%d", data + i);
      fclose(fp);
}

// Nalezne nejbližší volný kotel, který lze použít jako dočasné odkladiště
// při rozbíjení cyklů.
inline int get_free_place() {
      if ((free_place == 0) || (data[free_place] != 0)) {
            free_place = 1;
            while((free_place <= N) && (data[free_place] != 0))
                  free_place++;
            if (free_place > N) exit(1); // Tohle se nesmí podle zadání stát.
      }
      return free_place;
}

// Nalezne cestu nebo cyklus v přesunech počínaje kotlem "from" a zároveň
// v poli data obrátí ukazatele přesunu (tj. místo toho, kam se má hříšník
// přesunout z daného kotle, bude u kotle uloženo, ze kterého kotle se má
// hříšník přesunout do něj). Funkce vrací index posledního prvku cesty
// (pokud je stejný, jako "from", pak je to cyklus).
int find_path(int from) {
      if ((data[from] == from) || (data[from] == 0))
            return 0;

      int next = data[from];
      data[from] = 0;
      while(data[next] != 0) {
            int tmp = data[next];
            data[next] = from;
            from = next;
            next = tmp;
      }
      data[next] = from;
      return next;
}

// Zpracuje cestu, která byla nalezena pomocí find_path
// a vypíše výsledky o přesunech do souboru fp.
void process_path(int from, FILE *fp) {
      while(data[from] != 0) {
            fprintf(fp, "%d %d\n", data[from], from);
            int tmp = data[from];
            data[from] = from;
            from = tmp;
      }
}

int main(int argc, char **argv) {
      load_data();
```

```
      FILE *fp = fopen("kotle.out", "w");

      for(int i = 1; i <= N; i++) {
            // Když narazíme na volné pole, zapamatujeme si jej,
            // abychom ušetřili práci funkci get_free_place().
            if (data[i] == 0) free_place = i;
            if ((data[i] == 0) || (data[i] == i)) continue;

            // Nalezneme poslední prvek cesty/cyklu a cestu si připravíme.
            int last = find_path(i);

            int tmpPlace = 0;
            if (last == i) {
                  // Pokud je to cyklus, musíme ho nejdřív rozbít
                  // (odložit si jednoho hříšníka dočasně stranou).
                  tmpPlace = get_free_place();
                  fprintf(fp, "%d %d\n", data[i], tmpPlace);
                  last = data[i];
                  data[i] = 0;
            } else
                  free_place = i;

            // Zpracujeme cestu a vypíšeme přesuny.
            process_path(last, fp);

            if (tmpPlace) {
 // Pokud máme hříšníka uloženého stranou, tak si ho přesuneme na správné místo.
                  fprintf(fp, "%d %d\n", tmpPlace, i);
                  data[i] = i;
            }
      }
      fclose(fp);
      return 0;
}
```

21-1-3 Přijímací kancelář — Pavel Klavík

Takový plán pekla byl určitě pekelně zapeklitý a nebohému reportérovi zamotal na nemalou chvilku hlavu. Tu však nezamotal jenom jemu, ale i nejednomu řešiteli. Co by to bylo za peklo, kdyby přijímací kancelář musela být jenom na jednom místě? Přijímajících kanceláří může být samozřejmě víc. A co kdyby na nás čerti ušili podvod a v pekle žádná kancelář vůbec nebyla? S obojím se musí počítat a většina řešení si na tomto vylámala zuby. Nebo snad řádky kódu? Nelze činit žádné předpoklady o něčem, co v zadání nebylo uvedeno! Některá další řešení byla natolik pomalá, že pokud reportér neumřel, prohledává peklo dodnes ...

Ukážeme si postup, jak rychle a snadno přelstít peklo. Nejprve si maličko přeformulujeme zadání. Peklo bude orientovaný graf, místnosti budou vrcholy a chodby mezi nimi hrany. Nyní otočíme orientaci všech hran. Přijímací kancelář bude místo, z kterého existuje cesta do všech ostatních vrcholů grafu.

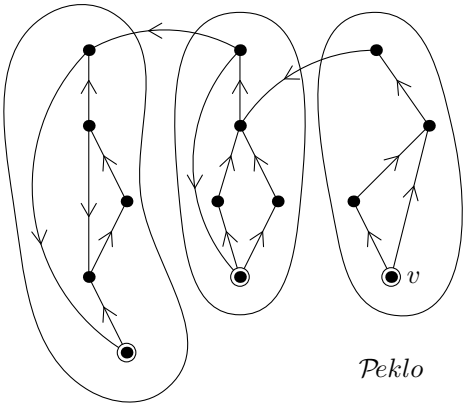
Jednoduché řešení nás určitě napadne hned. Pro každý vrchol ověříme, zda je přijímací kancelář. Spustíme z něj *prohledávání do hloubky*. To je algoritmus, který nám pro určitý vrchol zjistí, do kterých vrcholů z něj vede cesta. Funguje tak, že postupně prochází a značkuje vrcholy. Na začátku máme označený jenom výchozí vrchol. Pokud přijdeme do neoznačeného vrcholu, označíme ho a rekurentně spustíme prohledávání pro jeho sousedy. Po skončení běhu algoritmu budou označeny všechny dostupné vrcholy. Bližší informace o prohledávaní do hloubky naleznete v kuchařce Grafy 20-3.

Pokud jsme označili úplně všechny vrcholy grafu, vyhráli jsme a nalezli přijímací kancelář. Pokud žádný takový vrchol v grafu není, pak ani nemůže existovat přijímací kancelář. Jedno prohledávání nám pro graf s N vrcholy a M hranami zabere čas $O(N + M)$ (každý vrchol a každou hranu navštívíme jednou) a musíme ho zavolat pro každý vrchol z N vrcholů, tedy celkem $O(N^2 + NM)$, což nám pro běžné grafy (kde $M \geq N$) dává $O(NM)$. V paměti si potřebujeme udržovat celý graf, paměťová složitost bude $O(M + N)$. Toto řešení je správné, ale nikoho svojí rychlostí neoslní.

K rychlejšímu algoritmu nám pomůže následující pozorování. Pokud z libovolného vrcholu existuje cesta do přijímací kanceláře, pak i on sám je přijímací kancelář. Proč? Protože se z něj můžeme dostat do přijímací kanceláře a z ní poté do všech vrcholů grafu, tedy můžeme se dostat kamkoliv. Ale to není nic jiného než definice přijímací kanceláře. Tedy pokud spustíme prohledávání z nějakého vrcholu, který není přijímací kancelář, ani libovolný z navštívených vrcholů nebude přijímací kancelář. Z nich již nemusíme pouštět prohledávání, což nám ušetří spoustu času, bohužel asymptoticky máme pořád $O(NM)$.

Co kdybychom zkusili při dalších prohledáváních již neprocházet vrcholy, které jsme navštívili při předcházejících prohledáváních? Budou nás zajímat pouze nové vrcholy, do kterých jsme schopni se dostat. Pokud nám stále budou nějaké chybět, určitě žádný z navštívených vrcholů není přijímací kancelář. Takto redukujeme počet nenavštívených vrcholů, až po čase nalezneme vrchol v, z kterého spustíme prohledávání naposled. Všechny vrcholy jsme tedy navštívili buď při posledním prohledávání, nebo někdy dříve. Pokud je v grafu přijímací kancelář, pak je to určitě vrchol v. Proč? Nemůže to být žádný vrchol z předchozího procházení, protože z něj neexistuje cesta například do vrcholu v. A pokud by to byl nějaký jiný vrchol z posledního prohledávání, pak by také v byla přijímací kancelář, využijeme výše ukázaného poznatku, neboť z v do ní vede cesta. Na druhou stranu v vůbec nemusí být přijímací kancelář, může existovat vrchol z předchozích prohledávání, do kterého se z v nemůžeme dostat.

V takovém případě v grafu neexistuje přijímací kancelář. Řešení je jednoduché: Prostě pro v ověříme, zda přijímací kanceláří skutečně je. Projdeme z něj znovu celý graf. Pokud navštívíme všechny vrcholy, je v přijímací kancelář, jinak v pekle žádná není. Na obrázku jsou vyznačeny vrcholy, z kterých jsme spustili prohledávání, a spolu s nimi v jedné bublině všechny vrcholy, které jsme navštívili při jednom prohledávání.

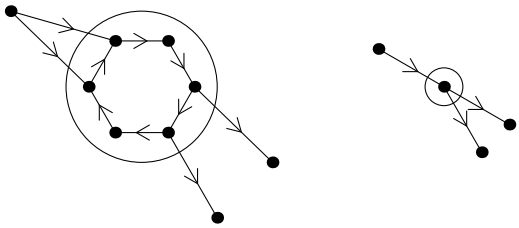


První fáze algoritmu poběží v čase $O(M + N)$, neboť na každý vrchol a hranu se podíváme jednou. Druhá ověřovací fáze poběží ve stejné složitosti $O(M+N)$, tedy dohromady dostáváme $O(M+N)$. Rychleji tento problém ani vyřešit nelze, tolik času potřebujeme na načtení vstupů. Paměťová složitost zůstává pořád stejná $O(M + N)$.

Řada z vás si všimla, že pokud by peklo bylo souvislé a byl v něm právě jeden vrchol, do kterého nevede žádná hrana, potom by to určitě byla přijímací kancelář. Naproti tomu pokud by takové vrcholy byly dva, pak přijímací kancelář nemůže v pekle existovat. Toto řešení však selže, pokud je přijímacích kanceláří v grafu více, protože nebude existovat vrchol, do kterého nevede žádná hrana. Ukážeme si, že i z na první pohled (po)chybné myšlenky se dá ledacos vytěžit a vytvořit funkční řešení.

Pokud jsou přijímací kanceláře dvě, musí ležet v *silně souvislé komponentě*. Co je to silně souvislá komponenta orientovaného grafu? Podobně jako u klasického grafu je to maximální množina vrcholů taková, že mezi každými dvěma vrcholy existuje orientovaná cesta (která je navíc vždy celá uvnitř komponenty). Každý orientovaný graf lze rozložit na komponenty. Z hlediska hledání přijímající kanceláře se všechny vrcholy komponenty chovají podobně. Proto můžeme vytvořit kopii grafu, každou komponentu nahradit jedním vrcholem a zachovat hrany napříč komponentami. Takové věci se říká *kontrakce komponenty*, kterou si můžete také představit tak, že všechny vrcholy komponenty natlačíme k sobě, čímž nám splynou v jeden, hrany napříč nám zůstanou. Vý-

sledná kopie je acyklická a stačí nám podívat se na stupně jednotlivých vrcholů. Zde musí existovat alespoň jeden vrchol, do kterého nevede žádná hrana, tedy projde nám výše uvedený test.



Poslední problém, který musíme vyřešit, je, jak hledat silně souvislé komponenty. Existuje pěkný rychlý algoritmus založený na prohledávání do hloubky, který funguje stejně jako výše uvedené řešení v čase $O(N + M)$. Jeho detaily zde vypustíme, avšak zvídavý čtenář si ho může zkusit vymyslet. Napovíme, že se použije průchod grafem do hloubky, poté se otočí orientace hran a spustí se druhý průchod do hloubky.

```
#include <stdio.h>
#define MAXN 1000
#define MAXM 1000

struct hrana {
        struct hrana* dalsi; // další prvek
        int cil; // cilový vrchol
};

struct vrchol {
        int oznacen;
        struct hrana* hrany; // spojový seznam na hrany
};

// počet vrcholů, hran, označených vrcholů
int N, M, posledni;
struct vrchol v[MAXN]; // pole vrcholů
struct hrana e[MAXM]; // pole hran

void projdi(int num);

int main(void) {
        // načteme vstup
        scanf("%d %d", &N, &M);
        // inicializace vrcholů
        for (int i = 0; i < N; i++) {
                v[i].oznacen = 0;
                v[i].hrany = NULL;
        }
```

```
        for (int i = 0; i < M; i++) { // čteme hrany
                int start, cil;
                // uložíme v opačném směru
                scanf("%d %d", &start, &cil);
                e[i].cil = start;
                // pridame hranu k vrcholu
                e[i].dalsi = v[cil].hrany;
                v[cil].hrany = &e[i];
        }

        for (int i = 0; i < N; i++) { // procházíme vrcholy
                if (v[i].oznacen == 0) {
                        posledni = i;
                        projdi(i);
                }
        }
        // otestujeme, zda "posledni" je kancelář
        for (int i = 0; i < N; i++)
                v[i].oznacen = 0;
        projdi(posledni);
        int je_kancelar = 1;
        for (int i = 0; i < N; i++)
                if (v[i].oznacen == 0)
                        je_kancelar = 0;

        if (je_kancelar) // vypíšeme výsledek
                printf("Přijímací kancelář je %d.\n",
                                                posledni);
        else printf("V pekle není přijímací kancelář!\n");
}

void projdi(int num) {
        v[num].oznacen = 1; // označíme vrchol
        struct hrana* hr = v[num].hrany;
        // projdeme všechny hrany z vrcholu
        while (hr != NULL) {
                if (v[hr->cil].oznacen == 0)
                        projdi(hr->cil);
                hr = hr->dalsi;
        }
}
```

21-1-4 Bonsaj — Martin Böhm

Mnohé z (h)řešitelů napadlo si bonsaj prostě postavit, během stavění zjistit její šířku a nakonec ji jednou projít a uložit si výsledky. To je samozřejmě řešení správné, složitost takového řešení je lineární – sestavení trvá lineárně dlouho k velikosti vstupu a paměti zabereme rovněž jen tolik, kolik má bonsaj/strom rozdvojek.

Jaké je asymptoticky optimální řešení? Vstup jistě musíme projít celý, tedy časová složitost lepší než $O(N)$ nebude. Paměťová složitost je lineární jakbysmet – stačí uvážit bonsaj typu `2 2 2 -1 2 -1 2 -1 -1 -1 -1`, kde si musíme pamatovat hodnoty alespoň $N/2$ prvků. Vidíme, že postavení bonsaje bylo řešení snadné, ale také správné.

O konstantu chytřejší řešení dostaneme tak, že si uvědomíme, že bonsaj nepotřebujeme stavět, stačí nám ji projít přímo v preorderu a chytře si ukládat přesuny mezi sloupečky. Ideální struktura na uchovávání počtu lístků v jednotlivých sloupečcích je obousměrný spojový seznam, neboť jej můžeme rozšiřovat na obou stranách. Abychom se v hustých větvičkách bonsaje neztratili, musíme si také pamatovat cestu, kudy jsme do aktuálně zkoumané rozdvojky přišli. Na to můžeme využít zásobníku nebo také rekurze.

```
program Bonsaj;

{ Obousměrný spojový seznam }
type odksez = ^sez;
    sez = record
    vlevo: odksez;
    vpravo: odksez;
    listku: Integer;
end;

type smer = (Doleva, Doprava);

procedure zpracuj(predchozi: odksez; s: smer);
{ Rekurzivní zpracování rozdvojek bonsaje:
  predchozi je předchozí rozdvojka,
  s je směr, ve kterém se koukáme. }
    var pocet: Integer;
    var novy: odksez;

begin;
    read(pocet);
    if pocet <> -1 then begin;
        if( s = Doleva) then begin;
            if predchozi^.vlevo = nil then begin;
                { Díváme se doleva, ale vlevo prvek
                  spojového seznamu chybí. }
                new(novy);
```

```
                novy^.vpravo := predchozi;
                predchozi^.vlevo := novy;
            end else novy := predchozi^.vlevo;

        end else begin;
            if predchozi^.vpravo = nil then begin;
                new(novy);
                novy^.vlevo := predchozi;
                predchozi^.vpravo := novy;
            end else novy := predchozi^.vpravo;
        end;

        novy^.listku := novy^.listku + pocet;

        zpracuj(novy, Doleva);
        zpracuj(novy, Doprava);
    end;
end;

var pocet: Integer;
var koren: odksez;
var pocatek: odksez;
begin;
    { Zpracuj vstup, kořen zvlášť. }
    read(pocet);
    if(pocet <> -1) then begin;
        new(koren);
        koren^.listku := pocet;
        zpracuj(koren, Doleva);
        zpracuj(koren, Doprava);

        { Přejdi na nejlevější prvek seznamu. }
        pocatek := koren;

        while pocatek^.vlevo <> nil do
                pocatek := pocatek^.vlevo;

        { Vypiš. }
        while(pocatek <> nil) do begin;
            write(pocatek^.listku);
            write(' ');
            pocatek := pocatek^.vpravo;
        end;
   end;
end.
```

21-1-5 Zapeklitá karetní hra — Josef Špak

Že jde o úlohu z kombinatoriky, napadne kdekoho. Ale aby šlo řešení hladce od ruky, je potřeba to vzít ze správného konce. Nejprve rozložíme do řady všechny navzájem nerozlišitelné čerty. Mezi každými dvěma čerty, na začátku a na konci řady je místo, kam můžeme položit nejvýše jednu ďáblici, celkem $\check{C}+1$. Teď spočítáme, kolika způsoby si můžeme vybrat místa, na která ďáblice po jedné položíme. Program tedy bude počítat kombinační číslo $\binom{\check{C}+1}{\check{D}}$. Taky můžeme uvažovat místa, na která čertice nepoložíme – $\binom{\check{C}+1}{\check{C}+1-\check{D}}$. Výsledek je stejný, ale dopočítáme se ho rychleji, pokud je ďáblic víc než $(\check{C}+1)/2$.

Jak ale efektivně spočítat kombinační číslo? Počítat dva faktoriály a pak je dělit sice bude fungovat, ale pro větší počty karet si nevystačíme s velikostí datového typu (obyčejná kapesní kalkulačka nezvládá víc než 69!, a i na to už potřebuje počítat s mantisou a exponentem). Výsledek bude celé číslo, zkusme tedy zlomek s faktoriály nějak přeuspořádat a střídavě násobit a dělit. Kombinační číslo $\binom{n}{k}$ se spočte jako $n!/\,[\,k!(n-k)!\,]$. Po vykrácení n! a (n-k)! si zlomek rozepíšeme:

$$\frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{1\cdot 2\cdot\ldots\cdot k}$$

V čitateli jsou po sobě jdoucí přirozená čísla, takže lze zlomek počítat postupně takto:

$$\frac{n}{1}\cdot\frac{(n-1)}{2}\cdot\frac{(n-2)}{3}\cdot\ldots\cdot\frac{(n-k+1)}{k}$$

Stačí si všimnout, že po každém vynásobení a vydělení (jedné iteraci) jsme spočítali nějaké kombinační číslo – $\binom{n}{1}$, $\binom{n}{2}$, … $\binom{n}{k}$, takže mezivýsledky jsou všechny celočíselné. Na to nám stačí proměnná, do které se vejde k-krát větší číslo než výsledek, kde k je $\min(\check{D}, \check{C}+1-\check{D})$.

Časová složitost je $O(\check{C}-\check{D})$, paměťová $O(1)$.

```
#include <stdio.h>
int main() {
  int C, D, n, k;
  unsigned long X = 1;
  scanf("%d %d", &C, &D);
  n = C + 1;
                // vybereme menší k pro výpočet kombinačního čísla
  k = (D < (C + 1 - D)) ? D : C + 1 - D;

  for (int i = 1; i <= k; ++i) {
    X *= n - (i - 1);
    X /= i;
  }
  printf("%d", X);
}
```

21-1-6 Nejkratší vyhrává — Martin Mareš & Milan Straka

Řešení našeho IQ-testu se sešla slušná hromádka, ale některá z nich používala jazyk Rapl až příliš vynalézavě a domýšlela si do něj instrukce, které podle definice v zadání rozhodně neuměl. Na ty běžnější chyby raději upozorníme rovnou, aby se nenachytali ostatní:

- Argument instrukce `write` může být pouze číslo, proměnná nebo prvek pole, určitě ne výraz s operátory.
- Podmínit lze jenom instrukci skoku, konstrukce typu `if a=0 => a=1` je nekorektní.

a) Posloupnost mocnin dvojky je možné vypsat na *4 instrukce* jednoduchou modifikací příkladu ze zadání:

```
        a=1
znovu:  write a
        a=a*2
        jump znovu
```

b) Správně jste poznali, že se jedná o začátek Fibonacciho posloupnosti, v níž je každé číslo součtem dvou předchozích. Stačí tedy, abychom si v registru `a` pamatovali aktuální číslo a v registru `b` číslo předchozí (budeme si přitom představovat, že před počáteční nulou je jednička). Dostaneme tak jednoduchý program na *6 instrukcí:*

```
        b=1
zase:   write a
        c=a+b     # následující
        b=a       # předchozí <- aktuální
        a=c       # aktuální <- následující
        jump zase
```

Kličku s pomocnou proměnnou `c` si ale můžeme ušetřit jednoduchým trikem a program tak zkrátit na *5 instrukcí:*

```
        b=1
zase:   write a
        a=a+b
        b=a-b
        jump zase
```

c) Jak vypsat prvních 16 číslic čísla π? Naprogramovat opravdový výpočet π není úplně snadné a určitě to nestihneme za méně než 16 instrukcí, které by nám stačily na program typu `write 3; write 1; ...` (mimochodem, opravdu jsme dostali několik delších řešení). Když tedy neumíme π spočítat, vymyslíme, jak tabulku jeho číslic reprezentovat co nejkompaktněji. Všimněme si, že do 32 bitů se vejde libovolné devíticiferné desítkové číslo, takže nám stačí vzít si dvě konstanty `31415926` a `53589793` a vypsat je po číslicích. Toho se drží i náš program na *7 instrukcí,* jen čísla rozkládá na číslice od konce:

```
        a=62951413
loooop: b=a%10    # mod 10 = posl. číslice
        write b
        a=a/10    # o číslici zkrátíme
        if a<>0 => jump loooop
        a=39798535
        jump loooop
```

Existuje ovšem ještě jeden efektivnější způsob: Najdeme dvě čísla, jejichž podíl se dostatečně přesně přiblíží k π, a tento podíl vypíšeme po číslicích. Překvapivě, s 32-bitovým čitatelem a jmenovatelem můžeme získat právě 16 číslic π, konkrétně zlomkem $165\,707\,065/52\,746\,197$. (Tohle je opravdu náhoda, i když nám to asi nebudete věřit. Opravdu jsme zadání schválně nenarafičili tak, aby to vyšlo. My sami jsme na tento způsob přišli díky inspiraci od Alexandra Mansurova, který ho ale sám vzápětí zavrhl):

```
        a=165707065
jedeme: b=a/52746197
        write b
        a=a%52746197
        a=a*10
        jump jedeme
```

d) Toto byl takový malý test, jak pozorně jste četli seznam instrukcí RaPLu. Jestlipak jste si všimli operace `@`, která počítá bitovou selekci? A jestlipak jste si také všimli, že čísla v naší čtvrté posloupnosti jsou přesně čísla 1, 2, 3, 4, ..., ze kterých jsou ovšem vyselectované jenom bity na sudých pozicích? Pokud ne, honem si běžte zopakovat instrukční sadu; pokud ano, zde je program na *4 instrukce:*

```
preqap: b=a@85    # 01010101 dvojkově
        write b
        a=a+1
        jump preqap
```

21-2-1 Špinavé tričko — Zbyněk Falt & Petr Kratochvíl

Ukážeme dvě řešení problému: jednodušší v čase $O(N^3)$ a o něco rychlejší v čase $O(N^2 \log N)$. Začneme tím jednodušším :-)

Skvrnu si uložíme jako x-ové souřadnice svislých hran a y-ové souřadnice vodorovných hran. V každém okamžiku si budeme pamatovat spojový seznam skvrn, které se na tričku nacházejí. Na začátku je seznam prázdný; když načteme ze vstupu další skvrnu, přidáme ji do seznamu. Navíc se podíváme, jestli nám nová skvrna nepřekryla nějakou starší skvrnu – v takovém případě starou skvrnu nahradíme seznamem jejích zbylých nepřekrytých částí.

Že se dvě skvrny překrývají, poznáme snadno: překrývají se jejich průměty na vodorovnou, nebo na svislou osu.

Nyní vyřešíme rozpadávání staré skvrny, kterou překryla nová.

- Pokud zasahuje stará skvrna nad novou, uřízneme z ní vršek – to je obdélník, který má všechny souřadnice stejné jako stará skvrna, kromě dolní hrany (ta bude rovna horní hraně nové skvrny).
- Pokud zasahuje stará skvrna pod novou, podobným způsobem uřízneme spodek (použijeme souřadnice staré skvrny, jen horní hrana bude rovna dolní hraně nové skvrny).
- Pokud stará skvrna zasahuje i nalevo od nové, uřízneme levý kus z toho, co ze staré skvrny zbylo po našem případném předchozím řezání.
- A nakonec uřízneme pravý kus, pokud to půjde. Tím získáme až čtyři zbylé kusy, na které se stará skvrna rozpadla, protože ji z části (nebo zcela) překryla skvrna nová. Tyto nové skvrny připojíme do spojového seznamu místo skvrny staré.

Zatím to vypadá, že časová složitost našeho algoritmu může být dost vysoká, protože může vznikat velké množstvích malých „rozpadnutých“ skvrn. Můžeme si všimnout, že po provedení všech rozpadů bude počet skvrn nejvýše $O(N^2)$. Když totiž protáhneme každou hranu každé skvrny přes celé tričko, rozdělíme ho nejvýše na $(2N-1)^2$ obdélníků ($2N$ svislými a $2N$ vodorovnými řezy), z nichž žádný se již nemůže nijak rozpadnout. Každou novou skvrnu porovnáváme s až $O(N^2)$ předchozími, takže časová složitost není horší než $O(N^3)$.

A nyní jak to udělat rychleji:

Nejdříve si zadání úlohy trochu upravme: Nebudeme počítat počet jednotek, které zabírají jednotlivé barvy, nýbrž pro každou skvrnu budeme počítat počet jednotek, na kterých je tato skvrna vidět.

Není těžké si rozmyslet, že pokud vyřešíme takto upravenou úlohu, tak nalezení řešení původního zadání je triviální: Stačí pro každou barvu sečíst počet jednotek, které zabírají skvrny dané barvy. A to je z algoritmického hlediska nezajímavá záležitost.

Nyní k samotnému řešení. První myšlenka, která určitě každého okamžitě napadne, spočívá ve vytvoření dvojrozměrného pole velikosti $(W-1)\times(H-1)$, které bude představovat tričko. Poté se postupně zpracovávají jednotlivé skvrny a příslušná políčka v poli se označují číslem této skvrny. Nakonec stačí pole projít a jednoduše spočítat výsledek.

Jakou složitost by mělo toto řešení? Vzhledem k tomu, že každé skvrna může být až velikosti $O(W \cdot H)$, tak časová složitost by byla $O(N \cdot W \cdot H)$ a paměťová $O(W \cdot H + N)$. To je poměrně hodně. Navíc už pro poměrně malá W a H můžeme velmi brzy narazit na velikost fyzické operační paměti.

Co když rozdělíme plochu trička na oblasti, které jsou ohraničeny přímkami, které procházejí hranami jednotlivých skvrn? Určitě platí, že každá takto vzniklá oblast bude obsahovat stejná čísla. Navíc proto, že každá skvrna při-

spěje nejvýše čtyřmi přímkami, tak celkový počet těchto oblastí bude pouze $O(N^2)$.

Ve skutečnosti tedy stačí pole velikosti $O(N^2)$, ve kterém lze simulovat předchozí triviální algoritmus. Jaká bude časová složitost tohoto postupu? Pro každou skvrnu musíme určit, v jakých oblastech se nachází. I kdybychom toto zvládli rychle, tak každá skvrna se může skládat až z $O(N^2)$ oblastí, takže časová složitost je minimálně $O(N^3)$, což je sice lepší, ale stále to není ono.

Neefektivita předchozího postupu je ukryta v tom, že každou oblast můžeme až $O(N)$-krát přečíslovat. Co kdybychom ale nezpracovávali postupně jednotlivé skvrny, ale jednotlivé oblasti? Například zdola nahoru a zleva doprava. Pak by si stačilo průběžně udržovat seznam skvrn, které se v aktuální oblasti vyskytují, a z nich vždy vybrat tu, která se objevila nejpozději (to lze provést v čase $O(\log N)$), a oblasti přiřadit její číslo.

Zbývá tedy vyřešit, jak onen seznam udržovat. Možné řešení je pamatovat si pro každý vodorovný pruh oblastí množinu skvrn, které se v tomto pruhu vyskytují a tuto množinu pak projít „zleva doprava“ a průběžně si pamatovat, které skvrny se v aktuální oblasti vyskytují.

Udržovat onu množinu skvrn je jednoduché. Pokud ji máme vytvořenou pro jeden pruh, tak je totiž snadné přejít na pruh následující. Stačí odebrat všechny skvrny, které se v tomto pruhu již nevyskytují a naopak přidat skvrny, které se v tomto pruhu nově objevily. Najít takové skvrny lze snadno, pokud si předem vytvoříme seznam všech horních a dolních hran, který je setříděné vzestupně podle souřadnice y. Pak když narazíme na dolní hranu, tak příslušnou skvrnu do seznamu přidáme. V případě horní hrany skvrnu odebereme.

Naprosto stejným způsobem pak zpracujeme jeden pruh. Ze skvrn v příslušné množině vytvoříme seznam levých a pravých hran, který setřídíme podle osy x a tento seznam pak jednoduše projdeme. Pokud narazíme na levou hranu, pak se v následující oblasti tato skvrna vyskytuje, pokud na hranu levou, tak se příslušné oblasti skvrna přestala vyskytovat. Seznam aktuálních skvrn pak budeme udržovat v haldě, abychom mohli vždy rychle nalézt skvrnu, která se v aktuální oblasti vyskytla nejpozději (je na vrchu).

Není těžké si rozmyslet, že tento seznam není třeba stále třídit. Je možné udržovat jej stále setříděný. Aby se pak lépe vkládalo doprostřed, je vhodné jej reprezentovat spojovým seznamem. Dále není těžké přijít na to, že žádné pole velikosti $O(N^2)$ vlastně nepotřebujeme, neboť je možné rovnou při průchodu seznamem počítat výsledek.

Jaká je časová složitost algoritmu? Nejdříve načteme vstup, to trvá $O(N)$, následovně setřídíme seznam dolních a horních hran skvrn, což lze zvládnout v $O(N \cdot \log N)$, poté tento seznam projdeme tak, že každou hranu zatřídíme/odebereme do/ze seznamu skvrn v aktuálním pruhu, což trvá $O(N)$. Tento seznam pak procházíme, přičemž každý bod vložíme/odebereme do/z haldy

$O(\log N)$.

Dohromady tedy $O(N)+O(N\cdot\log N)+O(N)\cdot(O(N)+O(N)\cdot O(\log N)) = O(N^2\cdot\log N)$. Paměťová složitost je pak $O(N)$.

Algoritmus je implementovaný v jazyku C++, neboť ten obsahuje knihovny pro pohodlnou práci se zmíněnými datovými strukturami. Aby byl program jednodušší, je tričko považováno za jednu velkou skvrnu s barvou 0. Samotný převod na původní zadání je pak udělán neefektivně, ale ve výsledné časové složitosti se to již neprojeví.

```
/***************************** Pomalejší verze ****************************/

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

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

 // je prunik dvou intervalu neprazdny?
#define prunik(xl, xp, yl, yp) \
      (((xl <= yl && xp > yl) || (yl <= xl && yp > xl))?1:0)

typedef struct skvrna {                  // udaje o skrvne
      int levy, pravy, horni, dolni;      // okraje skvrny
      int barva;
            // ukazatel na dalsi skrvnu ve spojovem seznamu
      struct skvrna *dalsi;
} SKVRNA;

// rozpadne starou skvrnu a misto ni vytvori spojovy seznam rozpadlych casti
void rozpad(SKVRNA *nova, SKVRNA *stara) {
      SKVRNA *seznam, *konec;  // spojovy seznam vzniklych casti; konec seznamu
      if (!prunik(stara->levy, stara->pravy, nova->levy, nova->pravy)
            || !prunik(stara->dolni, stara->horni, nova->dolni, nova->horni))
                        // pokud se skvrny neprekryvaji, nemame co delat
                  return;
      // na zacatek seznamu dame starou skvrnu
      seznam = konec = (SKVRNA *)malloc(sizeof(SKVRNA));
      *seznam = *stara;
      seznam->dalsi = NULL;
      if (stara->horni > nova->horni) {      // urizneme horni kus
            konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
            konec = konec->dalsi;
            konec->horni = stara->horni;
            konec->dolni = nova->horni;
            konec->levy = stara->levy;
            konec->pravy = stara->pravy;
            konec->barva = stara->barva;
            konec->dalsi = NULL;
      }
      if (stara->dolni < nova->dolni) {      // urizneme spodek
```

```
            konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
            konec = konec->dalsi;
            konec->horni = nova->dolni;
            konec->dolni = stara->dolni;
            konec->levy = stara->levy;
            konec->pravy = stara->pravy;
            konec->barva = stara->barva;
            konec->dalsi = NULL;
      }
      if (stara->levy < nova->levy) {            // urizneme levy kus
            konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
            konec = konec->dalsi;
            konec->horni = min(nova->horni, stara->horni);
            konec->dolni = max(nova->dolni, stara->dolni);
            konec->levy = stara->levy;
            konec->pravy = nova->levy;
            konec->barva = stara->barva;
            konec->dalsi = NULL;
      }
      if (stara->pravy > nova->pravy) {      // urizneme pravy kus
            konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
            konec = konec->dalsi;
            konec->horni = min(nova->horni, stara->horni);
            konec->dolni = max(nova->dolni, stara->dolni);
            konec->levy = nova->pravy;
            konec->pravy = stara->pravy;
            konec->barva = stara->barva;
            konec->dalsi = NULL;
      }
                  // napojime seznam na puvodni pokracovani
            konec->dalsi = stara->dalsi;
                  // a vynechame rozpadnutou skvrnu
            *stara = *(seznam->dalsi);
}

int main(void) {
      int N;                  // pocet skvrn
      int W, H;             // rozmery tricka
      int *plocha;            // plochy jednotlivych barev
      SKVRNA *skvrny = NULL;      // spojovy seznam skvrn
      SKVRNA *skvrny_kon;      // ukazatel na konec seznamu

      scanf("%d%d%d", &N, &W, &H);
      plocha = (int *)malloc((N+1)*sizeof(int));
      for (int i=0; i<N; i++) {
            if (skvrny == NULL) {      // pridame novou skvrnu na konec seznamu
                  skvrny = (SKVRNA *)malloc(sizeof(SKVRNA));
                  skvrny_kon = skvrny;
            } else {
                  skvrny_kon->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                  skvrny_kon = skvrny_kon->dalsi;
```

```
            }
            scanf("%d%d%d%d%d", &skvrny_kon->levy, &skvrny_kon->dolni,
                  &skvrny_kon->pravy, &skvrny_kon->horni, &skvrny_kon->barva);
            skvrny_kon->dalsi = NULL;
            // projdeme seznam skvrn a provedeme rozpady
            for (SKVRNA *stara = skvrny; stara != skvrny_kon;
                            stara = stara->dalsi) {
                  rozpad(skvrny_kon, stara);
            }
      }
      for (int i=1; i<=N; i++)
            plocha[i] = 0;
      for (SKVRNA *s = skvrny; s != NULL; s = s->dalsi)
            plocha[s->barva] += (s->pravy-s->levy) * (s->horni-s->dolni);
      int ciste = W*H;
      for (int i=1; i<=N; i++) {
            printf("Barva %d zabira na tricku %d jednotek plochy.\n",
                            i, plocha[i]);
            ciste -= plocha[i];
      }
      printf("Cisteho tricka zustalo %d jednotek.\n", ciste);
      return 0;
}

/******************** Rychlejší řešení ***************************/

#include <cstdio>
#include <vector>
#include <algorithm>
#include <set>
#include <list>

#define HORNI 1
#define DOLNI 2
#define LEVA 1
#define PRAVA 2

using namespace std;

struct vhrana_t {                   // vodorovná hrana skvrny
   int radek;
   int lsloupec;
   int psloupec;
   int typ;                         // dolní nebo horní
   int flek;                        // číslo fleku, kterému tato hrana přísluší
   vhrana_t(int _radek, int _lsloupec, int _psloupec, int _typ, int _flek)
      : radek(_radek), lsloupec(_lsloupec), psloupec(_psloupec),
                                       typ(_typ), flek(_flek) {}
   bool operator<(const vhrana_t &hrana) const {    // třídíme od zdola nahoru
      return radek<hrana.radek;
   }
```

```
};

struct shrana_t {                    // svislá hrana skvrny
   int sloupec;
   int typ;                          // levá nebo pravá
   int flek;                         // číslo fleku, kterému tato hrana přísluší
   shrana_t(int _sloupec, int _typ, int _flek)
      : sloupec(_sloupec), typ(_typ), flek(_flek) {}
   bool operator<(const shrana_t &hrana) const {    // třídíme zleva doprava
      return sloupec<hrana.sloupec;
   }
};

int main() {
   int W,H,N;
   scanf("%d%d%d",&W,&H,&N);

   vector<vhrana_t> vhrany;          // seznam všech vodorovných hran
   vector<int> barvy(N+1);           // pro převod čísla fleku na jeho barvu
   vector<int> obsahy(N+1,0);        // počet jednotek zabíraných fleky

                                  // plocha trička je považována za nultý flek
   vhrany.push_back(vhrana_t(0,0,W,DOLNI,0));
   vhrany.push_back(vhrana_t(H,0,W,HORNI,0));
   barvy[0]=0;

   for (int i=1;i<=N;i++) {
      int ldr, lds, phr, phs, barva;

      scanf("%d%d%d%d%d",&lds,&ldr,&phs,&phr,&barva);
      vhrany.push_back(vhrana_t(ldr,lds,phs,DOLNI,i));
      vhrany.push_back(vhrana_t(phr,lds,phs,HORNI,i));
      barvy[i]=barva;
   }
   sort(vhrany.begin(),vhrany.end());
   list<shrana_t> shrany;
   vector<vhrana_t>::iterator vhrana;
   int vpozice=-1;
                                   // procházíme vodorovné hrany zdola nahoru
   for (vhrana=vhrany.begin();vhrana!=vhrany.end();++vhrana) {
      list<shrana_t>::iterator shrana;
      if (vpozice>=0) {                      // pokud se nejedná o první hranu
         int vyska=vhrana->radek-vpozice;    // výška vodorovného pruhu
         int spozice=-1;
         set<int> halda;   // ze standarní haldy nelze odebírat libovolné prvky
                                    // procházíme svislé hrany zleva doprava
         for (shrana=shrany.begin();shrana!=shrany.end();++shrana) {
            if (spozice>=0)
                // *halda.rbegin() je maximální prvek v haldě
               obsahy[*halda.rbegin()] +=(shrana->sloupec-spozice)*vyska;
```

```
            if (shrana->typ == LEVA)              // přesně podle popisu řešení
                  halda.insert(shrana->flek);
            else
                  halda.erase(shrana->flek);
            spozice=shrana->sloupec;
         }
      }
      if (vhrana->typ == DOLNI) {                 // přesně podle popisu řešení
         list<shrana_t> pomoc;
         pomoc.push_back(shrana_t(vhrana->lsloupec,LEVA,vhrana->flek));
         pomoc.push_back(shrana_t(vhrana->psloupec,PRAVA,vhrana->flek));
         shrany.merge(pomoc);              // zatřídíme svislé hrany do množiny
      } else {
         for (shrana=shrany.begin();shrana!=shrany.end();++shrana)
                                    // odstraníme příslušné hrany z množiny
            while (shrana->flek==vhrana->flek)
               shrana=shrany.erase(shrana);
      }
      vpozice=vhrana->radek;
   }
   printf("Čistého trička zůstalo %d jednotek\n", obsahy[0]);

   for (int i=1;i<=N;i++) // neefektivní způsob, ale časovou složitost nezhorší
      if (barvy[i]!=0) {
         int barva=barvy[i];
         int celkem=0;
         for (int j=i;j<=N;j++)
            if (barvy[j]==barva) {
               celkem+=obsahy[j];
               barvy[j]=0;
            }
         printf("Barva %d zabíra %d jednotek\n",barva,celkem);
      }
   return 0;
}
```

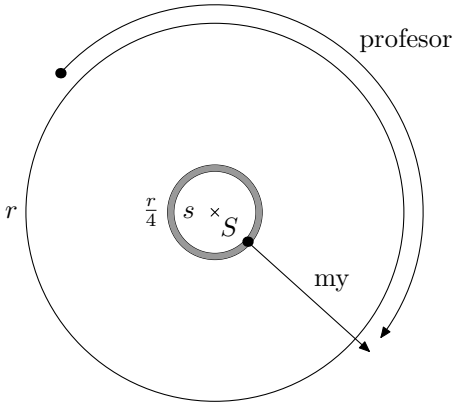
21-2-2 Útěk před zkouškou — Pavel Klavík

Analýza je velice komplikovaná věc a jak se ukázalo, hrozící zkouška zamotala nejednomu řešiteli hlavu. Přitom upláchnout je otázkou života a smrti! Došlá řešení by šla rozdělit do dvou skupin. V první skupině byla řešení, která tvrdila, že ať bude matfyzák snažit sebevíc, nedokáže se zkoušce vyhnout. Bohužel argumentace byla vedena způsobem: „Nenašel jsem řešení, proto neexistuje!“ Takový postup je zcela chybný. O tom svědčí i to, že druhá skupina řešitelů objevila způsob, jak upláchnout a zkoušce se vyhnout. Ukážeme si, jak vyzrát nad profesorem analýzy!

Označme si r poloměr rotundy. Předpokládejme, že profesor běží rychlostí 4 za časovou jednotku a student pouze 1. Pokud se student nachází hodně blízko středu rotundy S, je schopen obíhat po menší kružnici (se středem S)

rychleji než profesor po obvodu. Jeho úhlová rychlost je větší než profesorova. Jaký je maximální poloměr, který menší kružnice může mít? Délka obvodu roste lineárně s poloměrem. Pokud je její poloměr roven $r/4$, bude běhat stejně rychle jako profesor. Pokud bude (byť jen o malinko) menší, bude běhat rychleji.

Nyní si představme, že bychom byli kousek od středu a navíc profesor by byl přesně na opačném konci rotundy (střed S by ležel na úsečce mezi námi a profesorem). Rádi bychom se vydali přímo ke kraji rotundy, tedy na opačnou stranu, než stojí profesor. Jak daleko od středu musíme být, abychom mu upláchli? Nechť jsme ve vzdálenosti s. Profesor doběhne na druhou stranu za čas $\pi r/4$, zatímco nám to zabere čas $r - s$. Tedy pokud $r - s < \pi r/4$, podaří se nám upláchnout.



Pro lepší představu se podívejte na obrázek. Existuje vzdálenost od středu, která splňuje obě výše uvedené nerovnosti současně, ta je vyznačena šedým pásem. Můžeme provést nejprve první krok, dostat se na opačnou stranu než profesor. Poté provedeme druhý krok a utečeme profesorovi, jak je naznačeno na obrázku šipkami. Ať se profesor pohybuje jakkoli, nemůže nám v ani jednom kroku zabránit. Před zkouškou jsme šťastně zachráněni!

21-2-3 Fronta

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

Jak je vidět z došlých řešení, někteří z vás jsou rození matfyzáci a v tlačenici matfyzáckého života nebudou mít žádné problémy. Došlo i pár rozpačitých řešení, ale jejich autoři nemusí věšet hlavu – ne vždy je na škodu stát ve frontě druhý. Nyní se společně podívejme, jak se měla úloha řešit.

Zatímco se matfyzáci ve frontě dohadují, předbíhají a strkají, zkusme si jejich situaci matematicky popsat. Matfyzáci sami představují množinu a pokud jsou vztahy mezi nimi rozumné (tj. neexistuje v nich cyklus), můžeme hovořit dokonce o *částečně uspořádané množině* (zkráceně *ČUM*). ČUM se velice dobře

reprezentuje orientovaným grafem, kde vrcholy jsou prvky množiny a hrany představují vztahy (např. pokud si a myslí, že je chytřejší než b, pak existuje hrana z a do b).

V našem příkladu hledáme úplné (lineární) uspořádání částečně uspořádané množiny. Podíváme-li se na problém z hlediska grafu, kterým reprezentujeme ČUM, potřebujeme zavést číslování w, takové, že všem vrcholům je přiřazeno jedinečné číslo z rozsahu 1 až N (N je počet vrcholů) a pokud vede hrana z a do b, pak je $w(a) < w(b)$.

Postup, který nám vyrobí takové očíslování, se nazývá *topologické třídění* a je velmi dobře popsán v řadě učebnic programování. Existují dva známé a velmi jednoduché algoritmy, které řeší tento problém. První z nich je založený na odtrhávání vrcholů, ze kterých už nevede žádná hrana, a naleznete jej podrobně popsaný v knize Algoritmy a programovací techniky. Druhý, který využívá prohledání grafu do hloubky, najdete v naší kuchařce na téma grafy. Oba zvládnou najít uspořádání vrcholů v čase $O(N + M)$, kde N je počet vrcholů a M je počet hran.

Shodou okolností je zde uvedená časová složitost také dolním odhadem, protože každý vrchol musíme očíslovat ($O(N)$) a každou hranu musíme vzít v úvahu ($O(M)$), jinak nemůžeme zaručit, že jsme ověřili všechny podmínky na uspořádání.

Neboť jsou programátoři pěkní lenoši, ukážeme vám ten druhý, který je o hroší chlup kratší. A jak tento algoritmus funguje? Vybereme si některý ještě neprošlý vrchol v a spustíme na něj prohledávání do hloubky. Po chvilce dumání odhalíme, že pokud očíslujeme nejdříve všechny potomky (z hlediska průchodu DFS) a až nakonec sebe, dostaneme topologické uspořádání začínající vrcholem v. Číslování ale musíme provádět „odzadu“ – tj. od čísla N směrem dolů. Zbydou-li nám ještě některé neprošlé vrcholy, tak je opět zpracujeme pomocí DFS a číslování nám je zařadí ještě před vrchol v.

Abychom vyřešili i okrajové případy, zbývá ještě rozhodnout, kdy žádné uspořádání neexistuje. To se stane právě tehdy, když je v grafu alespoň jeden orientovaný cyklus. Naštěstí jej odhalíme jednoduše při průchodu do hloubky. Pokud při prohledávání dojdeme do vrcholu, který je stále ještě „otevřený“ (DFS s ním ještě neskončilo), musí nutně existovat orientovaná kružnice obsahující tento vrchol. Zadaná množina pak nemůže být nijak uspořádána, takže nám nezbývá, než oznámit výsledek a skončit.

Technické detaily implementace můžete prozkoumat ve vzorovém řešení. Tímto se s vámi loučí dvojka Martinů a jeden CodEx.

```
program fronta;
const max = 20000;
type pole = array [1..max] of integer;
    dpole = ^pole;
```

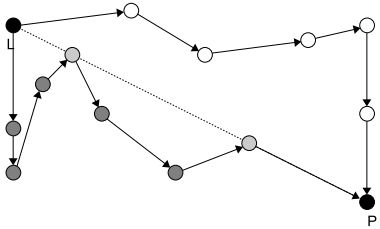
```
    matfyzak = record
        znamych: integer;
        znami: dpole; {seznam lidí ve frontě, které matfyzák zná}
        aktivni: integer; {informace o tom, je-li matfyzák již odbytý či ne}
    end;
    {pole matfyzáků / graf, reprezentovaný vrcholy se seznamy sousedů}
    matfyzaci = array[1..max] of matfyzak;
var usporadani: pole;    mela: matfyzaci;
    vpozice: integer; {první volná pozice v poli uspořádání}
    kruznice: integer;
{průchod do hloubky, vrátí 0 pokud je vše OK,
                                        1 pokud byla nalezena or. kružnice}
procedure DFS( v: integer);
var n,i: integer;
begin;
    n := 0;
    mela[v].aktivni := 1;
    if vpozice > max then exit;
    for i := 1 to mela[v].znamych do begin;
                                                    { detekována kružnice }
        if mela[ mela[v].znami^[i] ].aktivni = 1 then kruznice := 1
                                                    { vrchol již hotov }
        else if mela[ mela[v].znami^[i] ].aktivni = 2 then continue
        else DFS(mela[v].znami^[i]);
        if kruznice = 1 then exit;
    end;
    {jsme-li tu, kružnice nebyla nalezena}
    usporadani[vpozice] := v; {deaktivuj vrchol a ulož jej do uspořádání}
    vpozice := vpozice + 1;
    mela[v].aktivni := 2;
end;
var pocet,z,m: integer;
begin;
    read(pocet); {vstup}
    for m := 1 to pocet do begin;
        mela[m].aktivni := 0;
        read(mela[m].znamych);
        getmem(mela[m].znami, mela[m].znamych * sizeof(integer));
        for z := 1 to mela[m].znamych do begin;
            read(mela[m].znami^[z]);
    end; end;
    vpozice := 1; {spuštění}
    kruznice := 0;
    for m := 1 to pocet do begin;
        if mela[m].aktivni = 0 then DFS(m);
        if (kruznice = 1) or (vpozice > max) then break;
    end;
    if kruznice = 1 then write('no') else {výstup}
        for z := pocet downto 1 do write(usporadani[z],' ');
    writeln;
end.
```

21-2-4 Síťování — Pavel Čížek

Lze soudit, že většině řešitelů se podařilo dostat síť alespoň do stavu, aby mohli odeslat řešení. Bohužel jsem ale při opravování měl pocit, že často se tak stalo jen díky vhodnému rozložení počítačů (obvykle se stávalo, že Vaše programy vygenerovaly nějakou posloupnost i v případě, že řešení neexistovalo).

Úspěšní řešitelé obvykle používali algoritmus založený na porovnávání úhlů mezi jednotlivými počítači. Konkrétně vybrali jeden (např. nejlevější a pokud jich je víc, tak nejspodnější z nich) počítač Y a ostatní setřídili dle úhlu, který svírala jejich spojnice s Y s nějakým pevným směrem (obvykle kladnou poloosou x). Pokud se sešly 2 počítače na stejném úhlu, rozhodovala vzdálenost od Y. V tomto pořadí počítače zapojili a Y zařadili mezi první a poslední v setříděné posloupnosti. Snadno se nahlédne, že takto vygenerovaná spojnice se nekříží (Buď počítače A_i a A_{i+1} mají stejný úhel vzhledem k Y a pak, díky setřídění dle vzdálenosti, není žádný počítač X se stejným úhlem a vzdáleností $|A_iY| < |XY| < |A_{i+1}Y|$, nebo mají úhel různý, ale pak zřejmě úhly mezi A_i a A_{i+1} protne generovaná spojnice jen jednou, a to právě na drátu vedoucím z A_i do A_{i+1}. Podobně to dopadne i pro dráty vedoucí z počítače Y).

Nicméně ukážeme si zde jiný přístup k problému. Idea je taková, že vezmeme spojnici nejlevějšího a nejpravějšího počítače a spojíme počítače nad a pod touto přímkou zvlášť. Najdeme tedy nejlevější (a nejvyšší v případě, že je více počítačů s nejmenší x souřadnicí) počítač L a nejpravější (a nejnižší, pokud jich je více) počítač P. Pak roztřídíme zbylé počítače na ty, co jsou *Nad*, *Pod* a *Na* přímce *L*–*P* (na ilustrativním obrázku bílé, šedé, resp. světle šedé). Pokud budou všechny počítače na této spojnici, tak zřejmě řešení neexistuje (přímé spojení PL při návratu se protne s dráty vedoucími opačně). Jinak lze síť udělat, jen si musíme dát pozor, co provedeme s počítači *Na* spojnici — když *Nad* i *Pod* spojnicí jsou nějaké počítače, je jedno, jestli je přidáme k cestě z L do P (tedy k počítačům *Nad* spojnicí) nebo k cestě z P do L (tedy k počítačům *Pod* spojnicí) (na ilustraci jsme počítače *Na* spojnici přidali k počítačům *Pod* spojnicí). Pokud ale *Nad* (analogicky *Pod*) spojnicí nejsou žádné počítače a počítače *Na* spojnici bychom přidali na cestu z P do L (z L do P) dostali bychom protnutí právě v bodech, které jsou *Na* spojnici. Proto musime v tomto případě počítače *Na* spojnici uvažovat jako kdyby byly *Nad* (*Pod*) spojnicí ($Nad' = Nad \cup Na$,resp. $Pod' = Pod \cup Na$).



Nakonec už jen zbývá seřadit počítače *Nad* spojnicí a *Pod* spojnicí tak, aby vytvořily cestu, která se nebude protínat. Na to ale stačí setřídit počítače dle souřadnice x vzestupně a v případě, že se se nachází víc počítačů na stejném sloupci, tak dle y sestupně (jak dopadne setřídění je na obrázku naznačeno šipkami). Takto vzniklá spojnice se nevrací ve směru x a při stejném x používá třídění dle y a proto opět nedojde k protnutí. Podobná situace nastane u L (P) díky volbě nejvyššího (nejnižšího) počítače s minimální (maximální) x souřadnicí. „Kružnici“ z počítačů pak vytvoříme spojením L, setříděných počítačů *Nad*, P a obrácením setříděných počítačů *Pod*.

```
const Presnost = 1E-6;  {Přesnost pro práci s reálnými čísly}
            {Čísla lišící se o méně než tuto konstantu považujeme za stejná}

type
   PPocitac = ^TPocitac;
   TPocitac = record
      X,Y:real;
      Poradi:integer;
      Dalsi:PPocitac;
   end;

var
   Pocitace:PPocitac;
   Nejlevejsi,Nejpravejsi:PPocitac;
   NadSpojnici,PodSpojnici,NaSpojnici:PPocitac;

procedure Nacti;
{Načtení vstupu a určení nejlevejšího a nejpravějšího počítače (z pohledu X)}
var N,i:integer;
    Novy:PPocitac;
begin
   readln(N);  {počet počítačů}
   Pocitace:=nil;
   if N <> 0 then begin   {načtení do lineárního spojového seznamu}
      new(Novy);
      readln(Novy^.X,Novy^.Y);
      Novy^.Poradi:=1;
      Novy^.Dalsi:=nil;
      Nejlevejsi:=Novy;
      Nejpravejsi:=Novy; {první počítač je zvlášť kvůli inicializaci}
      Pocitace:=Novy;
      for i:=2 to N do begin
         new(Novy);
         readln(Novy^.X,Novy^.Y);
         if (Novy^.X > Nejpravejsi^.X) or
                 ((Novy^.X = Nejpravejsi^.X) and (Novy^.Y < Nejpravejsi^.Y))
            then Nejpravejsi:=Novy;
         if (Novy^.X < Nejlevejsi^.X) or
                 ((Novy^.X = Nejlevejsi^.X) and (Novy^.Y > Nejlevejsi^.Y))
```

```
            then Nejlevejsi:=Novy;
         Novy^.Poradi:=i;
         Novy^.Dalsi:=Pocitace;
         Pocitace:=Novy;
      end;
   end;
end;

procedure Rozdel;
{Rozdělí počítače podle relativní polohy
                        vzhledem k spojnici nejlevějšího a nejpravějšího}
var Zpracovavany:PPocitac;
    ZSlozkaVektorovehoSoucinu:real;
    {vektor popisující směr spojnice nejlevějšího a nejpravějšího počítače}
    VektX,VektY:real;
    {vektor popisující směr spojnice nejlevějšího a zpracovávaného bodu}
    ZpracVektX,ZpracVektY:real;
begin
   NadSpojnici:=nil;
   PodSpojnici:=nil;
   NaSpojnici:=nil;
   VektX:=Nejpravejsi^.X - Nejlevejsi^.X;
   VektY:=Nejpravejsi^.Y - Nejlevejsi^.Y;
   while Pocitace <> nil do begin
      Zpracovavany:=Pocitace;
      Pocitace:=Pocitace^.Dalsi; {vyřazení zpracovávaného počítače ze seznamu}
      if (Zpracovavany=Nejlevejsi) or (Zpracovavany=Nejpravejsi) then continue;
      {tyto 2 se zpracovávají zvlášť}
      ZpracVektX:=Zpracovavany^.X - Nejlevejsi^.X;
      ZpracVektY:=Zpracovavany^.Y - Nejlevejsi^.Y;
      ZSlozkaVektorovehoSoucinu:=VektX * ZpracVektY - VektY * ZpracVektX;
      if abs(ZSlozkaVektorovehoSoucinu) < Presnost then begin {leží na spojnici}
         Zpracovavany^.Dalsi:=NaSpojnici;
         NaSpojnici:=Zpracovavany;
      end else if ZSlozkaVektorovehoSoucinu > 0 then begin {leží nad spojnicí}
         Zpracovavany^.Dalsi:=NadSpojnici;
         NadSpojnici:=Zpracovavany;
      end else begin {leží pod spojnicí}
         Zpracovavany^.Dalsi:=PodSpojnici;
         PodSpojnici:=Zpracovavany;
      end;
   end;
end;

function Merge(Seznam1,Seznam2:PPocitac):PPocitac;
{dva setřízené seznamy slije - původní seznamy jsou během procesu zničeny}
var Prvni,Posledni:PPocitac;
   function Porovnej(Pocitac1,Pocitac2:PPocitac):boolean;
   {porovná polohy dvou počítačů a vrací true,
                                pokud má být Pocitac1 zatřízen jako první}
   begin
```

```
      Porovnej:=(Pocitac1^.X < Pocitac2^.X)
             or ((Pocitac1^.X = Pocitac2^.X) and (Pocitac1^.Y > Pocitac2^.Y));
      {zde si můžeme dovolit mezi reálnými čísly test na rovnost
      - zaokrouhlovací chyby, které vzniknou při načítání, budou u stejných
      hodnot na vstupu stejné, a tedy rovnost bude doopravdy platit}
   end;
begin
   if (Seznam1 = nil) then Merge:=Seznam2 {v případě, že je jedna posloupnost
                                                    prázdná, je slití triviální}
   else if (Seznam2 = nil) then Merge:=Seznam1
   else begin
      if Porovnej(Seznam1,Seznam2) then begin
                 {nejdříve zjistíme, čím bude výsledná posloupnost začínat}
         Prvni:=Seznam1;
         Seznam1:=Seznam1^.Dalsi;
      end else begin
         Prvni:=Seznam2;
         Seznam2:=Seznam2^.Dalsi;
      end;
{Konec funkce (až po přiřazení výsledku) jen slije zbytek seznamů.}
{Je zde implementována nerekurzivní varianta.
                                          Rekurzivní by byla výrazně kratší.}
{  Prvni^.Dalsi:=Merge(Seznam1,Seznam2); }
      Posledni:=Prvni;
      while (Seznam1<>nil) and (Seznam2<>nil) do begin {a pak slijeme zbytek}
         if Porovnej(Seznam1,Seznam2) then begin
            Posledni^.Dalsi:=Seznam1;
            Posledni:=Seznam1;
            Seznam1:=Seznam1^.Dalsi;
         end else begin
            Posledni^.Dalsi:=Seznam2;
            Posledni:=Seznam2;
            Seznam2:=Seznam2^.Dalsi;
         end;
      end;
      if (Seznam1 = nil) then Posledni^.Dalsi:=Seznam2
                                              else Posledni^.Dalsi:=Seznam1;
      Merge:=Prvni;
   end;
end;

function MergeSort(Seznam:PPocitac):PPocitac;
                                       {dostane seznam a vrátí ho setřízený}
var Seznam1,Seznam2,Swap:PPocitac;
begin
   if (Seznam = nil) then MergeSort:=nil
   else if (Seznam^.Dalsi = nil) then MergeSort:=Seznam {koncové podmínky}
   else begin
      Seznam1:=nil;Seznam2:=nil;
      while Seznam<>nil do begin {rozdělí seznam na poloviny
                                              - sudé a liché prvky zvlášť}
```

```
         Swap:=Seznam^.Dalsi;
         Seznam^.Dalsi:=Seznam1;
         Seznam1:=Seznam2;
         Seznam2:=Seznam;
         Seznam:=Swap;
      end;
      Seznam1:=MergeSort(Seznam1); {setřízení polovin}
      Seznam2:=MergeSort(Seznam2);
      MergeSort:=Merge(Seznam1,Seznam2); {a merge setřízených posloupností}
   end;
end;

procedure Vypis(Seznam:PPocitac);
begin
   while (Seznam<>nil) do begin
      write(Seznam^.Poradi,' ');
      Seznam:=Seznam^.Dalsi;
   end;
end;

function Obrat(Seznam:PPocitac):PPocitac;
var ObracenySeznam,Dalsi:PPocitac;
begin
   ObracenySeznam:=nil;
   while Seznam<>nil do begin
      Dalsi:=Seznam^.Dalsi;
      Seznam^.Dalsi:=ObracenySeznam;
      ObracenySeznam:=Seznam;
      Seznam:=Dalsi;
   end;
   Obrat:=ObracenySeznam;
end;

begin
   Nacti;
   if Pocitace = nil then begin
      writeln;  {není co vypisovat - 0 počítačů}
   end else if Pocitace^.Dalsi = nil then begin
      writeln('1'); {jedinný počítač ... takže není příliš co řešit}
   end else begin
      Rozdel;
      if (NadSpojnici = nil) and (PodSpojnici = nil) then
         writeln('Řešení neexistuje.') {všechny počítače leží na spojnici}
      else begin
         NadSpojnici:=MergeSort(NadSpojnici); {setřízení jednotlivých seznamů}
         PodSpojnici:=MergeSort(PodSpojnici);
         NaSpojnici:=MergeSort(NaSpojnici);
         if (NadSpojnici = nil) then NadSpojnici:=NaSpojnici
         else PodSpojnici:=Merge(NaSpojnici,PodSpojnici);
                              {přidání bodů na spojnici k příslušné straně}
         write(Nejlevejsi^.Poradi,' ');
```

```
        Vypis(NadSpojnici);
        write(Nejpravejsi^.Poradi,' ');
        PodSpojnici:=Obrat(PodSpojnici);
        Vypis(PodSpojnici);
        writeln;
      end;
   end;
end.
```

21-2-5 Nádobí — Pavel Klavík & Kristýna Stodolová

Pohled na talíře a hrnky evokuje u většiny matfyzáků myšlenku na jídlo a následné kručení v břiše. Proto není divu, že danou problematiku řeší pomocí hladového algoritmu. Hladový algoritmus (angl. greedy) vybírá v každém kroku to aktuálně nejlepší řešení. Ne vždycky se dobere toho optimálního, ale v tomto případě funguje. Rozhodování o tom, který kus nádobí kdy umýt, probíhá odzadu (tj. ode dne, do kterého „přežije“ to nejtrvanlivější). Pro každý den se vezmou všechny kusy nádobí, které do něj vydrží, a zároveň jejich umývání ještě nebylo naplánováno na později. Z nich se hladově vybere ten nejcennější a naplánuje se na tento den k umytí. Takto nalezneme nějaké řešení, ale ještě nemusí být úplně jasné, že je skutečně optimální. Zkusme si to tedy dokázat.

K dokázání správnosti použijeme techniku, která nám může pomoci i se spoustou jiných algoritmů, které postupně konstruují optimální řešení. Ze začátku algoritmus běžel správně. Nerozhodli jsme totiž ještě o umytí jediného kusu, proto naše rozhodnutí nemohlo být špatně. Poté algoritmus prováděl jednotlivé kroky a nakonec vydal nějaký výsledek. Předpokládejme pro spor, že by výsledek byl špatně, tedy nebyl by optimální. Potom musel existovat nějaký krok, kdy se algoritmus poprvé rozhodl špatně, tedy rozhodl umýt kus nádobí A, který se nevyskytoval v žádném optimálním řešení. Vezměme si tedy jedno z optimálních řešení, které vyhovovalo všem rozhodnutím provedeným v předcházejících krocích, a v daném kroku umývá kus B. Ukážeme, že z tohoto řešení dokážeme vyrobit jiné optimální řešení, které umývá kus A, to by byl jistě spor. Pokud optimální řešení umývalo kus A v nějakém dalším kroku, pouze prohodíme pořadí umývání A a B. Pokud není nikde umývání A naplánováno, tak umyjeme místo B kus A. Platí však, že cena kusu A je alespoň tak velká jako cena B. Nově vytvořené řešení je optimální a zároveň omývá ve špatném kroku A, což je spor.

Ještě jedna drobná poznámka na okraj: co kdybychom k řešení použili hladový algoritmus, ale rozhodovali o umývání v opačném pořadí od prvního dne. Takové řešení by nefungovalo, například pro vstup $(2, 2)$ a $(1, 1)$. Můžete si vyzkoušet jako malé cvičení nalézt místo, ve kterém by výše uvedený důkaz nezafungoval.

Co se týče implementace, nejprve si nádobí utřídíme sestupně podle trvanlivost např. pomocí QuickSortu v čase $O(N \log N)$. Výběr nejdražšího kusu uděláme haldou, uspořádanou dle ceny. Do té vždy přidáme kusy, které vydrží do aktuálně zkoumaného dne, z vrchu odebereme ten nejdražší a dáme ho k mytí. V haldě dokážeme dělat operace vkládání i odebírání v čase $O(\log N)$ na prvek, což nám dohromady dává $O(N \log N)$, tedy i složitost celého algoritmu je $O(N \log N)$. Plány, co umýt v jednotlivé dny, si udržujeme v poli. Protože některé kusy mohou mít obrovskou trvanlivost ve srovnání s N, maximální počet dní, které nás zajímají, je minimum z N a maximální trvanlivosti. Další dny už stejně budeme mít celý dřez volný!

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 1000
#define SWAP(A,B,tmp) tmp = A; A = B; B = tmp;

struct nadobi { // jeden kus nádobí
    int id, t, c, umyt; // číslo; trvanlivost; cena; zda máme naplanováno umytí
};
struct nadobi kusy[MAX]; // informace o kusech nádobí
int halda[MAX*2],hpocet=0,dny[MAX],pdni;
                // halda; počet prvků; plány na jednotlivé dny; počet dní
int N,tmp; // počet kusů; proměnná pro swap

int halda_pridej_prvek(int index) { // přidání prvku do haldy
    halda[++hpocet] = index; // dáme na konec
    halda[hpocet*2] = halda[hpocet*2+1] = 0;
    int i = hpocet;
    // probubláme prvek nahoru na správné místo
    while (i > 1 && kusy[halda[i/2]].c < kusy[halda[i]].c) {
        SWAP(halda[i/2], halda[i], tmp);
        i /= 2;
    }
}
int halda_odeber_maximum() { // odebrání prvku z haldy
    if (hpocet == 0)
        return 0;
    int ret = halda[1]; // vymažeme a prohodíme s posledním prvek
    halda[1] = halda[hpocet];
    halda[hpocet--] = 0;

    int i = 1; // poslední prvek bubláme dolů, dokud není správně
    while (kusy[halda[i]].c < kusy[halda[i*2]].c
                    || kusy[halda[i]].c < kusy[halda[i*2+1]].c) {
        // prohodíme s potomkem, který má větší cenu
        if (kusy[halda[i*2]].c > kusy[halda[i*2+1]].c) {
            SWAP(halda[i], halda[i*2], tmp);
            i = i*2;
```

```
        }
        else {
            SWAP(halda[i], halda[i*2+1], tmp);
            i = i*2+1;
        }
    }
    return ret;
}
// porovnávací funkce pro qsort
int porovnej(const void * a, const void * b) {
    int ta = ((struct nadobi*) a)->t;
    int tb = ((struct nadobi*) b)->t;
    if (ta < tb) return -1;
    else if (ta == tb) return 0;
    else return 1;
}
int main() {
    scanf("%d", &N);

    // načteme jednotlivé kusy
    for (int i = 1; i <= N; i++) {
        scanf("%d %d", &kusy[i].t, &kusy[i].c);
        kusy[i].id = i;
    }
    kusy[0].c = INT_MIN; // 0. kus plní funkci zarážky

    // utřídíme podle trvanlivosti (nejtrvanlivější na konec)
    qsort(&kusy[1], N, sizeof(struct nadobi), porovnej);

    int i = N; // procházíme dny a určujeme kusy k umytí
    pdni = N < kusy[N].t ? N : kusy[N].t;
    for (int t = pdni; t > 0; t--) { // začneme od minima dny/max. trvanlivost
        while (kusy[i].t >= t) // přidáme nové kusy nádobí
            halda_pridej_prvek(i--);

        dny[t] = halda_odeber_maximum(); // vezmeme hladově nejcennější
        kusy[dny[t]].umyt = 1;
    }

    // vypíšeme výsledek
    for (int i = 1; i <= pdni; i++)
        if (dny[i])
            printf("%d ", kusy[dny[i]].id);
    for (int i = 1; i <= N; i++)
        if (!kusy[i].umyt)
            printf("%d ", kusy[i]);
        printf("\n");
    return 0;
}
```

21-2-6 Nejkratší opět vyhrává — Martin Mareš & Milan Straka

Úloha první s jedním chybějícím číslem vám nečinila velké problémy. V zásadě se objevila řešení dvě. První využívalo funkce `xor`. Tato funkce totiž splňuje `x ^ x = 0`, takže `xor`y dvou stejných čísel se vyruší. Navíc při `xor`ování nezáleží na pořadí, takže řešení je nasnadě: pokud z`xor`ujeme všechna čísla $1..N$ a všechna čísla $A[0]..A[N-2]$, výsledek je přesně chybějící číslo. To proto, že chybějící číslo jsme `xor`ovali jednou a všechna ostatní čísla dvakrát. Pokud chceme mít řešení co nejkratší, ještě si uvědomíme, že $A[N-1] = 0$. Získáme tak následující program o pěti instrukcích:

```
dalsi:  x = x ^ A[i]            # zde se xoruje A[0]..A[N-1]
        i = i + 1
        x = x ^ i               # zde se xoruje 1..N
        if i < N => jump dalsi
        write x
```

Druhé řešení první úlohy používá místo `xor`u sčítání a odčítání. Pokud k jedné proměnné přičteme všechna čísla $1..N$ a odečteme čísla $A[0]..A[N-2]$, dostaneme chybějící číslo. Zde se ale někteří řešitelé zarazili – pokud je $N > 2^{31}$, tak $N+(N-1) > 2^{32}$ a při sčítání dojde k přetečení! A při odečátání zrovna tak! Velmi dobře, drahý Watsone, ale i když dojde k přetečení, pořád budeme mít výsledek spočítaný modulo 2^{32}. Taková přesnost nám ovšem stačí, protože chybějící číslo $\leq N < 2^{32}$. Získáme tedy alternativní řešení opět o pěti instrukcích:

```
dalsi:  x = x - A[i]            # zde se odčítá A[0]..A[N-1]
        i = i + 1
        x = x + i               # zde se přičítá 1..N
        if i < N => jump dalsi
        write x
```

Ještě jedna poznámka. Někteří pokročilí matematici využívali faktu, že $1 + \ldots + N = N(N+1)/2$. Jenomže program `S = N+1; S = S*N; S = S/2` nespočítá výsledek modulo 2^{32}, ale jenom modulo 2^{31}. Problém je v dělení – pokud máte v $S = N(N-1)$ modulo 2^{32}, tak po provedení $S/2$ je v S hodnota $N(N-1)/2$ modulo 2^{31}. Takový program tedy nefunguje správně.

Nyní k řešení druhé úlohy. Rádi bychom nějak aplikovali naše řešení úlohy první. To bychom mohli, pokud bychom dokázali čísla $1..N$ rozdělit do dvou skupin, aby v každé bylo právě jedno chybějící číslo. Pak by stačilo použít `xor`ovací řešení na každou skupinu zvlášť.

Nejprve z`xor`ujeme všechna čísla $1..N$ a $A[0]..A[N-3]$. Tím dostaneme `xor` obou chybějících čísel. Tato hodnota má jednotkové bity tam, kde se chybějící čísla liší. Nechť je tedy b-tý bit této hodnoty jedničkový. Chybějící čísla se liší v b-tém bitu. Pokud rozdělíme čísla $1..N$ a $A[0]..A[N-3]$ do dvou skupin podle toho, jakou mají hodnotu b-tého bitu, bude v každé skupině právě jedno chybějící číslo. Jedno z nich pak dostaneme tak, že z`xor`ujeme všechny hodnoty

$1..N$ a $A[0]..A[N-3]$ s nulovým bitem b, a druhé tak, že `zxor`ujeme hodnoty s jedničkovým bitem b.

Nyní jak to provést na co nejméně instrukcí. Nejprve musíme v `xor`u chybějících čísel najít jeden z jeho jedničkových bitů. Označme `xor` chybějících čísel jako x a zapišme ho ve dvojkové soustavě:

$$\begin{aligned} x &\quad bbbbbb1000 \\ \text{dvojkový doplněk } x\text{, tj } -x &\quad \overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}1000 \\ x\&(-x) &\quad 0000001000 \end{aligned}$$

Vidíme, že `x & (-x)` má nastavený jediný bit, a to nejnižší bit, který je nastaven v x.

Zbývá vyřešit poslední detail. Jak rozdělit hodnoty podle nějakého bitu? Pokud je v b nastaven jeden bit, můžeme použít bitovou selekci, protože `x @ b` je přesně hodnota tohoto bitu, 0 nebo 1. Touto hodnotou pak můžeme indexovat pole `X`, takže nemusíme používat instrukci skoku a chybějící čísla budou v `X[0]` a `X[1]`.

Použitím všech těchto triků dostaneme program o 14 instrukcích:

```
alpha:  x = x ^ A[i]            # zde se xoruje A[0]..A[N-1]
        i = i + 1
        x = x ^ i               # zde se xoruje 1..N
        if i < N => jump alpha
        # nyní je v x xor chybějících čísel

        y = 0 - x
        b = x & y
        # v b je nastaven jediný bit,
        # a to takový, kde se chybějící čísla liší

beta:   i = A[j] @ b            # A[j] je ve skup. i (0 či 1)
        X[i] = X[i] ^ A[j]      # xoruj A[j] ve skupině i
        j = j + 1
        i = j @ b               # j je ve skupině i (0 či 1)
        X[i] = X[i] ^ j         # xoruj čísla j ve skupině i
        if j < N => jump beta
        write X[0]
        write X[1]
```

Někteří řešitelé se pokusili řešit druhou úlohu tak, že použili sčítací řešení a spočítali si součet chybějících čísel. Z tohoto součtu spočítali průměr chybějících čísel. Nakonec rozdělili čísla $1..N$ a $A[0]..A[N-3]$ na čísla menší nebo rovná průměru a větší než průměr. Takto také rozdělili čísla do svou skupin, z nichž každá obsahuje jedno chybějící číslo. Problém je jenom s přesností –

pokud známe součet chybějících čísel modulo 2^{32}, tak průměr, tj. součet děleno dvěma, známe jenom modulo 2^{31}, takže algoritmus bez ošetření přetékání nefunguje.

21-3-1 Kódování memgramů — Petr Kratochvíl

Nejprve vyřešíme jednoduchý případ, kdy má memgram 4 políčka, a pak se pustíme do případu pro obecné N.

V tabulce se 4 políčky lze přenést 2 bity následujícím způsobem: První bit informace bude `xor` hodnot v prvním řádku (tedy počet jedniček v prvním řádku), druhým bitem bude `xor` hodnot v prvním sloupci. Když dostaneme memgram v nějakém konkrétním stavu, podíváme se, které bity jsou nastaveny jinak, než jak je chceme odeslat. První bit opravíme změnou na políčku b, druhý opravíme převrácením hodnoty políčka c a pokud jsou špatně oba, stačí změnit políčko a. A pokud jsou oba bity nastaveny správně, změníme políčko d, které nemá na zprávu žádný vliv.

a	b
c	d

1. bit $= a \operatorname{xor} b$
2. bit $= a \operatorname{xor} c$

Více než 2 bity ale v této malé tabulce přenést nelze: jsou totiž 4 možné zprávy (4 možné kombinace hodnot posílaných 2 bitů), a my můžeme provést jen 4 různé akce (změnit jedno ze 4 políček). Při přenášení více bitů by bylo možných zpráv více, ale my dokážeme rozlišit jen 4.

Tento důkaz lze snadno rozšířit pro obecný případ, kdy má tabulka N políček. Umíme provést pouze N různých akcí, tedy umíme poslat nejvýše N různých zpráv, což odpovídá $\log_2 N$ přeneseným bitům informace.

A jak bude vypadat přenos informace většími tabulkami? Necháme se inspirovat případem $N = 2$: Najdeme v tabulce $\log_2 N$ množin políček, každá bude nést jeden bit informace v podobě `xor`u hodnot na všech svých políčkách. A navíc potřebujeme, aby pro každou (i prázdnou) množinu bitů existovalo v tabulce políčko, jehož změna způsobí změnu právě těchto vybraných bitů (a žádných jiných). Pak totiž budeme moci opravit libovolnou kombinaci bitů, které budou v náhodně nastaveném memgramu špatně.

Stačí tedy, když o každém políčku řekneme, do kterých množin patří (které bity ovlivní jeho změna). Takovou informaci o jednom políčku si můžeme představit jako číslo ve dvojkové soustavě – i-tá číslice bude 1, pokud dané políčko leží v i-té množině (ovlivňuje i-tý bit zprávy), jinak bude 0. Aby byly množiny správně rozděleny (a bylo možné opravit libovolnou kombinaci bitů), musí v tabulce existovat všechna čísla od 0 do $2^{\log_2 N} - 1 = N - 1$. To lze ale zařídit jednoduše, prostě políčka popořadě očíslujeme čísly 0 až $N - 1$. Po převodu těchto čísel do dvojkové soustavy pro každé políčka zjistíme, které bity ovlivňuje, a můžeme jít vesele kódovat :-)

V konkrétním případě, kdy tabulka má velikost 8×8, je možné přenést $\log_2 64 = 6$ bitů informace a lépe to už nejde.

21-3-2 Nadposloupnost — Pavel Machek

Nadposloupnost zřejmě hodně lidí odradila svojí komplikovaností; ten zbytek aspoň zmátla. Část problému byla v tom, že sny byly zadány jako stringy a řešení se snažila pracovat s nimi opravdu efektivně.

Nechť L je délka obou posloupností ve znacích a N je délka obou posloupností ve slovech.

Řešení používající jednoduché strcmp má složitost $O(L^2)$. S použitím trie se dá vyrobit řešení v čase $O(N^2 + L)$, které je ale zbytečně komplikované, a dá se předpokládat, že slova stejně budou krátká.

Jak takové řešení bude fungovat? Bylo by možné použít kuchařku na hledání společné podposloupnosti, a potom mezi její prvky naskládat přebytecná slova.

Ale přímé řešení je možná názornější: Budeme si pamatovat nejlepší řešení pro prvních N slov z originálních vzpomínek a prvních M slov z přidávaných vzpomínek (pole `best/bestlen/bestbeg`). Nejlepší řešení pro N/M zjistíme na základě již spočítaných řešení pro $1 \ldots N-1/1 \ldots M-1$.

Pokud slovo je společné v obou sekvencích, tak nejlepší řešení bude přidat do výstupu tuto vzpomínku, jinak máme na vyběr mezi řešením pro $n-1/m$ a přidat originální vzpomínku a řešením pro $n, m-1$ a přidat vkládanou vzpomínku.

Časová složitost bude $O(L^2)$, paměťová by byla $O(L + N^2)$, kdybychom ovšem používali dynamickou alokaci.

S díky CTU Open Contestu 2008 a Josefu Cibulkovi.

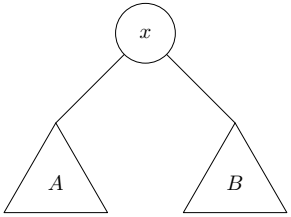
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXW 2100
#define MAXLEN 1000
typedef struct Words {
       char *w;
       int sentence;
       int pos;
} Words;

/* Vstupní texty */
char word[2][MAXW+2][MAXLEN+2];
int n,n0,n1;
Words w[2*MAXW+2];
int numseq[2][MAXW+2], diffwcnt;
/* 0 == slovo společné oběma sekvencím,
   1 == použijeme originální vzpomínku,
```

```
   2 == použijeme novou vzpomínku */
int best[MAXW+2][MAXW+2];
int bestlen[MAXW+2][MAXW+2];
int bestbeg[MAXW+2][MAXW+2];

int main(void)
{
        int i,j,tmp;
        /* Načteme originální vzpomínky */
        for(i=0;; i++) {
                scanf(" %s ", word[0][i]);
                if(word[0][i][0]=='.') break;
                w[i].w = word[0][i];
                w[i].sentence = 0;
                w[i].pos = i;
        }
        n0 = i;
        if(n0==0) return 0;
        /* Načteme vkládané vzpomínky */
        for(i=0;; i++)          {
                scanf(" %s ", word[1][i]);
                if(word[1][i][0]=='.') break;
                w[n0+i].w = word[1][i];
                w[n0+i].sentence = 1;
                w[n0+i].pos = i;
        }
        n1 = i;
        n=n0+n1;

        diffwcnt=0;
        for(i=0;i<n;i++) {
                if(i>0 && strcmp(w[i].w, w[i-1].w)) diffwcnt++;
                numseq[w[i].sentence][w[i].pos] = diffwcnt;
        }

        for(i=0;i<n0;i++) { bestlen[i][n1] = n0-i;
                bestbeg[i][n1] = numseq[0][i]; best[i][n1] = 1; }
        for(i=0;i<n1;i++) { bestlen[n0][i] = n1-i;
                bestbeg[n0][i] = numseq[1][i]; best[n0][i] = 2; }
        bestlen[n0][n1] = 0; bestbeg[n0][n1] = 0; best[n0][n1] = -1;

        for(i=n0-1;i>=0;i--)
                for(j=n1-1;j>=0;j--) {
                        if(numseq[0][i]==numseq[1][j]) {
                                bestlen[i][j] = 1+bestlen[i+1][j+1];
                                bestbeg[i][j] = numseq[0][i];
                                best[i][j] = 0;
                                continue;
                        }
                        if(bestlen[i+1][j] < bestlen[i][j+1] ||
                           (bestlen[i+1][j] == bestlen[i][j+1]
```

```
                        && numseq[0][i] < numseq[1][j])) {
                            bestlen[i][j] = bestlen[i+1][j]+1;
                            bestbeg[i][j] = numseq[0][i];
                            best[i][j] = 1;
                    } else {
                            bestlen[i][j] = bestlen[i][j+1]+1;
                            bestbeg[i][j] = numseq[1][j];
                            best[i][j] = 2;
            }       }
        i=0; j=0;
        while(1) {
                tmp = best[i][j];
                if(tmp==0 || tmp==1) printf("%s ", word[0][i]);
                if(tmp==2) printf("%s ", word[1][j]);
                if(tmp==-1) break;
                if(tmp==0 || tmp==2) j++;
                if(tmp==0 || tmp==1) i++;
        }
        printf(".\n");
        return 0;
}
```

21-3-3 Topologie snů — Pavel Klavík

Skoro všechna došlá řešení se úspěšně vypořádala s problémem a snoví inženýři mohli být nadmíru spokojeni. Jak už to ale bývá, některá byla pomalejší a jiná o maličko rychlejší. Udělejme si malou vycházku do lesa a podívejme se na stromy pořádně.



x	prefix A	prefix B

infix A	x	infix B

postfix A	infix B	x

Jakpak vypadá prefixový, infixový a postfixový zápis? Na obrázku máme strom s kořenem x a podstromy A a B. Tři zápisy jsou uvedeny pod ním.

Kořenem stromu je x, které se nachází na první pozici v prefixovém zápisu. Naproti tomu v infixovém zápisu odděluje prvky levého podstromu a prvky pravého podstromu. Navíc k těmto dvěma podstromům máme i jejich prefixové zápisy. Tím jsme problém rozdělili na dva a můžeme použít pro stromy typický postup *rozděl a panuj*. Pro stromy se hodí obzvlášť, neboť pro ně je rozdělení na menší podproblémy naprosto přirozené, máme podstromy a elementární podproblémy mohou být listy.

Nalezneme kořen x v infixovém zápisu, třeba tak, že projdeme všechny prvky, a problém rozdělíme na dva podstromy, které budeme řešit rekurzivně. Rekurze se zastaví v listech, pro které jsou všechny tři zápisy stejné. Strom

si nemusíme vytvářet v paměti, ale můžeme postfixový zápis rovnou vypisovat. Jaká je časová složitost? V každém kroku si potřebujeme vyhledat kořen v infixovém zápisu. Na to potřebujeme lineární čas v délce úseku. Pokud bude strom vyvážený, dostaneme celkový čas $O(N \log N)$, v nejhorším případě však $O(N^2)$. Analýza je velice podobná jako u QuickSortu.

Pokud bychom chtěli zrychlit výše popsaný algoritmus, potřebovali bychom zrychlit vyhledávání kořenů v infixovém zápisu. Mohli bychom si pro každý prvek stromu předpočítat, kde přesně se v infixovém poli nachází. Pro to není potřeba konstruovat žádné komplikované struktury, stačí nám si ke každé hodnotě v infixovém zápisu ještě pamatovat její pozici a poté infixový zápis utřídit. Vyhledávat pak můžeme pomocí půlení intervalu. S tímto zrychlením dosáhneme časové složitosti $O(N \log N)$ v nejhorším případě. Podaří se nám to vyřešit ještě rychleji? Co kdybychom se na věc podívali z jiného úhlu?

Co v předchozím algoritmu stálo tolik času? Lineární čas $O(N)$ stojí procházení a vypisovaní prvků, tady zjevně algoritmus nezrychlíme. Velice drahé je však vyhledávání v infixovém zápisu, zkusíme se obejít bez něj. Vždy jsme nejprve problém rozdělili na dva menší a pak je řešili. Co kdybychom s rozdělením počkali, tedy nejprve pustili řešení na levý podstromu a až v průběhu zjistili, kde se nachází kořen v infixu. Jak toho ale dosáhnout? V infixovém seznamu se postupně budeme posouvat. Na začátku ukazujeme na jeho první prvek. Prefixový seznam procházíme. Ve chvíli, kdy narazíme v prefixovém seznamu na prvek z infixového, víme, že jeho levý podstrom máme vyřešený. V infixu se posuneme na další prvek a zbývá vyřešit pravý podstrom. Když narazíme na rodiče o jedna výše, víme, že i pravý podstrom je vyřešený, protože jsme vyřešili levý podstrom umístěný výš. Takto rekurentně dokážeme vypsat strom v lineárním čase $O(N)$.

```
#include <stdio.h>
#define MAX 1000

int N, prefix_pos = 0, infix_pos = 0; // počet prvků; pozice v prefixu a infixu
int prefix[MAX], infix[MAX]; // prefixový a infixový zápis

void vypisuj_postfix(int rodic) // rekurzivní výpis
{
        if (infix[infix_pos] == rodic || prefix_pos >= N)
                return;
        int hodnota = prefix[prefix_pos++]; // kořen podstromu

        vypisuj_postfix(hodnota); // vypíšeme levý podstrom
        infix_pos++; // posuneme se v infixu
        vypisuj_postfix(rodic); // vypíšeme pravý podstrom
        printf("%d ", hodnota); // nakonec vypíšeme samotný kořen
}
```

```
int main()
{
        scanf("%d", &N); // načteme zápisy
        for (int i = 0; i < N; i++) scanf("%d", &prefix[i]);
        for (int i = 0; i < N; i++) scanf("%d", &infix[i]);
        vypisuj_postfix(-1); // vypíšeme postfixový zápis
        return 0;
}
```

21-3-4 Optimalizace stromu — Michal „vorner“ Vaner

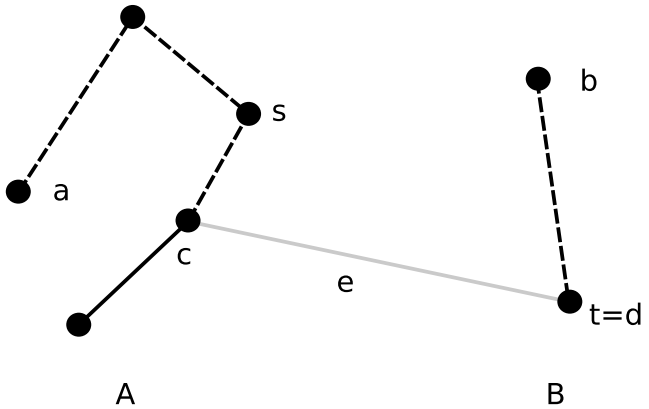
Vyberme si dva nejvzdálenější vrcholy a a b v neoptimalizovaném stromě T. Mezi nimi vede nějaká cesta P, jejíž délka je rovna průměru stromu $d(T)$.

Pokud je optimalizace úspěšná a zoptimalizovaný strom T' má menší průměr, potom musí existovat i kratší cesta mezi a a b. A protože T' je strom, P již existovat nemůže, tedy jsme při optimalizaci museli odebrat některou hranu e z P.

Po odebrání hrany e (ale před přidáním nové) vznikly dva stromy, říkejme jim třeba A a B. Každý z nich obsahuje jeden z vrcholů a a b, nechť jsme si je tedy pojmenovali tak, aby $a \in A$ a $b \in B$.

Dále si označme c a d konce hrany e ve stromech A a B.

Na pomoc těm, kteří se ztrácejí ve značení, je zde obrázek.



Nyní si v každém stromu najdeme střed (pro A to bude s, pro B t). Střed bude takový vrchol, že cesta do nejvzdálenějšího vrcholu téhož stromu bude nejmenší možná. Je zřejmé, že pokud jsme již správně zvolili hranu e, tak spojením s a t získáme optimální strom T' – kdybychom zvolili nějaký vrchol, který má k některému vrcholu ve svém stromě dále, vznikne tím delší cesta. Lze si všimnout, že s, resp. t, lze umístit doprostřed nejdelší cesty v A, resp. B (pokud jsou prostředky dva při sudém počtu vrcholů na cestě, pak lze vybrat libovolný z nich). Kdyby totiž vzdálenost do některého vrcholu byla větší než polovina

této cesty, lze cestu prodloužit a není tudíž nejdelší. Naopak, alespoň polovinu určitě bude potřebovat libovolný vrchol na této cestě (k tomu vzdálenějšímu konci cesty).

Rozmysleme si tedy, jak správně vybrat hranu e. Už víme, že se musí nacházet na cestě P. Jaký bude průměr grafu po optimalizaci? Nová nejdelší cesta buď povede přes nově přidanou hranu, pak její délka bude:

$$\left\lceil \frac{d(A)}{2} \right\rceil + 1 + \left\lceil \frac{d(B)}{2} \right\rceil$$

Tato vznikne spojením cesty z s do nejvzdálenějšího vrcholu v A, nové hrany a cesty z t do nejvzdálenějšího vrcholu v B.

Další možnost je, že nejdelší cesta je uvnitř jednoho malého stromu, tedy $d(A)$ nebo $d(B)$ je větší, než nejdelší spojená cesta. Na tento případ někteří řešitelé zapomněli.

Jiní si úlohu zjednodušili tak, že odebírali prostřední hranu cesty. To, bohužel, také nefunguje (protipříkladem budiž cesta o 12 vrcholech).

Jak tedy najdeme správnou hranu? Jednoduše, vyzkoušíme všechny na cestě P. U každé hrany si spočítáme průměry stromů, které by vznikly, když bychom ji odebrali. Z nich spočítáme nejdelší cestu, která v celém stromě poté povede. Nakonec vybereme nejlepší hranu a odebereme.

Potřebujeme tedy najít napřed nejdelší cestu stromu. Strom si zakořeníme a projdeme ho do hloubky. V každém vrcholu vždy spočítáme hodnoty pro celý jeho podstrom, za použití již spočítaných hodnot ze všech synů.

Budeme počítat délku nejdelší cesty v celém podstromu a délku nejdelší cesty, která končí v kořeni aktuálního podstromu. V listech je situace jednoduchá, obě cesty mají délku 0. Pokud má vrchol syny, pro spočítání nejdelší končící v něm vezme „nejlepší nabídku“ od synů – nejdelší, končící v některém synovi. Tu poté prodlouží o 1 hranu. Nejdelší v podstromu bude buď nejdelší v podstromu některého syna a nebo spojení dvou nejdelších končících v synech. Je třeba ještě ošetřit, když je syn jen jeden (pak je druhá nabídka nulová).

Nakonec si výsledek vyzvedneme v kořeni.

Protože potřebujeme nejen délku cesty, ale i cestu samotnou, budeme si pamatovat vždy některý vrchol nejdelší cesty v podstromu (nejjednodušší bude si pamatovat ten nejblíže ke kořeni) a v každém vrcholu cesty si pamatovat, kam je třeba pokračovat. Pak lze cestu jednoduše zrekonstruovat.

Proč to bude fungovat? Dokážeme indukcí. Že jsou informace pravdivé v listech, je zřejmé. To, že nejdelší cesta končící do tohoto vrcholu je prodloužením jedné z nejdelších cest v listech, je také vidět. A nejdelší cesta v podstromu buď prochází kořenem vrcholu, pak se skládá ze dvou končících v něm, a nebo neprochází. V takovém případě ale musí být uvnitř některého synovského podstromu.

Nyní máme tedy cestu, jak z ní vybrat správnou hranu? Potřebujeme znát průměr každého „zajímavého“ stromu (vzniklého odebráním některé hrany cesty P). Všimneme si, že délka nejdelší cesty v podstromu je průměr tohoto podstromu. Kdybychom tedy vybírali kořeny podstromů při výpočtu nejdelší cesty tak, že leží na cestě P, dostali bychom vždy jeden ze stromů odebráním hrany otec-syn.

Napřed tedy najdeme cestu P. Poté strom zakořeníme, tentokrát ve vrcholu a a prohledáme znovu. Cesta P vede z a (kořene) do některého z listů. Tedy dostaneme všechny zajímavé stromy obsahující b. Poté uděláme ještě jednou totéž, ale opačně – zakořeníme v b.

Poté projdeme hodnoty a odebereme správnou hranu. Nyní potřebujeme najít středy zbylých stromů. Ty ale, jak jsme si již všimli, leží uprostřed nejdelších cest těchto stromů. Nejdelší cesty těchto stromů máme již spočítané, tudíž středy dostaneme zadarmo.

Časová složitost je lineární. V každém vrcholu provedeme konstantně mnoho operací. Tedy, pokud porovnávání nabízených hodnot budeme považovat ještě za zpracování každého syna (každý bude s nejlepší a 2. nejlepší nabídkou porovnáván jen jednou, ve svém otci). Paměťová složitost je také lineární, více než lineárně mnoho paměti v lineárním čase nestihneme spotřebovat.

Je ale potřeba si dát pozor při načítání a pokud je v něm potřeba třídit (např. hrany podle zdrojového vrcholu), tak použít přihrádkové třídění, aby nám to složitost nezhoršilo.

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

// Přednačtená, nezpracovaná polovina hrany
struct edge_raw {
  unsigned vertex, index;
};

// Jeden vrchol
struct vertex {
  // První hrana tohoto vrcholu. Další následují za ní,
  // až do první hrany následujícího vrcholu
  unsigned edges;
  // Spočítané nejdelší cesty
  struct longest {
    // Nejdelší vedoucí tudy, nejdelší celková
    unsigned local, subtree;
    // Vrchol, kterým prochází nejdelší cesta v tomto podgrafu
    unsigned vertex;
    // Sousedi v cestě, která prochází tudy
    unsigned neighbors[2];
  } longest[2];
```

```
};

// Počet vrcholů
static unsigned tree_size;
// Přednačtené hrany
static struct edge_raw (*edges_raw)[2];
// Graf
static struct vertex *vertices;
static unsigned *edges;

// Najde nejdelší cestu v podstromu s kořenem index a nevrací se do parent
// Výsledky ukládá do longest[result]
static void path_search(unsigned index, unsigned parent, unsigned result) {
  // Případ, kdy nic hezkého nebylo nalezeno
  unsigned longest_subtree = 0, subtree_vertex = index;
  unsigned longest_neigh[3];
  unsigned local_lengths[3] = {};
  // Hrany vedoucí ze mě
  for(unsigned i = vertices[index].edges; i < vertices[index + 1].edges; i ++) {
    unsigned vertex = edges[i];
    if(vertex == parent) // Odsuď jsem přišel
      continue;
    if(vertex == tree_size) // Tahle hrana tu není (je "vypnutá")
      continue;
    // Prohledat tento podstrom
    path_search(vertex, index, result);
    // Je to zajímavé? Je tam něco dostatečně dlouhého?
    if(vertices[vertex].longest[result].subtree > longest_subtree) {
      subtree_vertex = vertices[vertex].longest[result].vertex;
      longest_subtree = vertices[vertex].longest[result].subtree;
    }
    for(int j = 1; j >= 0; j --)
      if(vertices[vertex].longest[result].local + 1 >= local_lengths[j]) {
        longest_neigh[j + 1] = longest_neigh[j];
        local_lengths[j + 1] = local_lengths[j];
        longest_neigh[j] = vertex;
        local_lengths[j] = vertices[vertex].longest[result].local + 1;
  }   }
  // Vyplnit sebe z toho, co se spočítalo
  memcpy(vertices[index].longest[result].neighbors, longest_neigh,
                                              2 * sizeof *longest_neigh);
  vertices[index].longest[result].local = local_lengths[0];
  // Nejdelší prochází skrz mě
  if(local_lengths[0] + local_lengths[1] > longest_subtree) {
    vertices[index].longest[result].subtree =
                                       local_lengths[0] + local_lengths[1];
    vertices[index].longest[result].vertex = index;
  } else { // Nejdelší v některém podstromu
    vertices[index].longest[result].subtree = longest_subtree;
    vertices[index].longest[result].vertex = subtree_vertex;
} }
```

```
// Vytáhne nejdelší cestu v subtree z grafu do output
static void path_fill(unsigned subtree, unsigned result, unsigned *output) {
  // Najdi nejvyšší vrchol na cestě
  unsigned vertex = vertices[subtree].longest[result].vertex;
  unsigned length = vertices[subtree].longest[result].subtree;
  unsigned index = vertices[vertex].longest[result].local;
  // Umísti ho na správné místo
  output[index] = vertex;
  // Projdi oba konce cesty odsud
  for(unsigned i = 0, pos = vertices[vertex].longest[result].neighbors[0];
                  i < index;
                  i ++, pos = vertices[pos].longest[result].neighbors[0])
    output[index - 1 - i] = pos;
  for(unsigned i = index+1, pos = vertices[vertex].longest[result].neighbors[1];
                  i <= length;
                  i ++, pos = vertices[vertex].longest[result].neighbors[0])
    output[i] = pos;
}

int main(int argc, char *argv[]) {
  // Jen otravné načítání souboru
  FILE *f = fopen("optstro.in", "rt");
  fscanf(f, "%zu", &tree_size);
  if(tree_size < 2)
  {
    fprintf(stderr, "Nemá hrany, nelze optimalizovat\n");
    return EXIT_FAILURE;
  }
  // Jeden navíc - zarážka
  vertices = malloc((tree_size + 1) * sizeof *vertices);
  for(unsigned i = 0; i < tree_size; i ++)
    vertices[i].edges = 0;
  // O jednu hranu méně než vrcholů
  edges_raw = malloc((tree_size - 1) * sizeof *edges_raw);
  // Načíst hrany, spočítat, kolik patří kterému vrcholu
  for(unsigned i = 0; i < tree_size - 1; i ++)
    // Každá hrana má 2 konce, uložme si je
    for(unsigned j = 0; j < 2; j ++) {
      fscanf(f, "%zu", &edges_raw[i][j].vertex);
      edges_raw[i][j].vertex --; // Soubor je od 1, v programu indexujeme od 0
      edges_raw[i][j].index = vertices[edges_raw[i][j].vertex].edges ++;
    }
  fclose(f);
  // Zařadit hrany, kam patří, připravit vrcholy
  unsigned current_pos = 0;
  for(unsigned i = 0; i <= tree_size; i ++) {
    unsigned size = vertices[i].edges;
    vertices[i].edges = current_pos;
    current_pos += size;
  }
  // Každá hrana je tu 2* - jednou tam, jednou zpět
```

```
edges = malloc((tree_size - 1) * 2 * sizeof *edges);
for(unsigned i = 0; i < tree_size - 1; i ++)
  for(unsigned j = 0; j < 2; j ++)
    edges[vertices[edges_raw[i][j].vertex].edges + edges_raw[i][j].index]
            = edges_raw[i][1 - j].vertex;
// Neroztříděné hrany už nejsou potřeba
free(edges_raw);

// Teď to zajímavé. Najdeme si nejdelší cestu.
// Zavolat rekurzivní fci. zakořeněnou v 0. vrcholu a s neexistujícím rodičem
path_search(0, tree_size, 0);
// Vyplnit cestu
unsigned path_len = vertices[0].longest[0].subtree;
unsigned *path = malloc((path_len + 1) * sizeof *path);
path_fill(0, 0, path);
// Co jsou konce cesty? Budeme z nich znovu hledat.
unsigned a = path[0], b = path[path_len];
// Spočítat velikosti cest ve zbylých stromech
path_search(a, tree_size, 0);
path_search(b, tree_size, 1);
// Která hrana je ta nejlepší?
unsigned best;
unsigned best_val = tree_size + 1; // Něco dostatečně velkého
for(unsigned i = 0; i < path_len; i ++)
{
  unsigned left = vertices[path[i]].longest[1].subtree;
  unsigned right = vertices[path[i + 1]].longest[0].subtree;
  unsigned trough = (left + 1) / 2 + (right + 1) / 2 + 1;
  unsigned total = (left > right) ? left : right;
  total = (total > trough) ? total : trough;
  // Tahle hrana je lepší
  if(total < best_val)
  {
    best_val = total;
    best = i;
  }
}

// Načíst nejlepší hranu
unsigned e[] = { path[best], path[best + 1] };
free(path); // Zbytek cesty není potřeba
printf("Odebrat: %zu-%zu\n", e[0] + 1, e[1] + 1);
// Použít už spočítané cesty k nalezení středů, spojit
for(unsigned i = 0; i < 2; i ++) {
  unsigned path_len_short = vertices[e[i]].longest[1 - i].subtree;
  unsigned *path_short = malloc((path_len_short + 1) * sizeof *path_short);
  // Zkopírovat cestu (nebyla by potřeba celá,
  // ale takto je to méně práce a může se hodit při ladění)
  path_fill(e[i], 1 - i, path_short);
  // Načíst novou, optimálnější hranu
  e[i] = path_short[path_len_short / 2];
```

```
    free(path_short);
  }
  printf("Přidat: %zu-%zu\n", e[0] + 1, e[1] + 1);
  // Zrušit graf
  free(edges);
  free(vertices);
  return EXIT_SUCCESS;
}
```

21-3-5 Rozklad na součty

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

Nejeden řešitel se nyní právem raduje z velkého součtu svých bodů, neboť tato úložka byla pouze jednoduchým cvičením na rekurzi. Klíčové bylo vhodně rozmyslet, v jakém pořadí vytvářet jednotlivé součty, aby se nám žádný neopakoval víckrát.

Jeden z možných způsobů je generovat součty uspořádané vždy do monotónní posloupnosti. V našem příkladu vytváříme nerostoucí posloupnost, kterou pak vypíšeme v opačném pořadí (díky tomu je vypíšeme dokonce ve stejném pořadí jako v zadání, i když to není nezbytně nutné). Zbývá prozradit technický detail, jak jednoduše vytvářet uspořádané posloupnosti sčítanců. Zavedeme si při generování maximální číslo m a všechny zkoušené sčítance pak budou pouze z rozsahu 1 až m. Když pak sčítanec na pozici i má hodnotou k, necháme zbylé sčítance (na pozicích $i + 1$ a vyšších) generovat s parametrem $m = k$, takže následující sčítance budou nejvýš tak velké, jako ten na pozici i. Nyní stačí jen přesypat naše úvahy do nějakého programovacího jazyka, trochu zamíchat a voilà ...

Na závěr bychom rádi poznamenali, že někteří vykutálení řešitelé se snažili optimalizovat svůj program vložením předpočítaných výsledků přímo do zdrojového kódu. Po načtení vstupu pak pouze vypsali příslušnou množinu výsledků. To je sice naprosto legitimní a v některých situacích i velmi rozumná optimalizační technika, avšak v tomto případě není nikterak užitečná. Nejpomalejším místem této úlohy je beztak vypsání výstupu a časové limity byly dostatečně velké, abyste mohli použít rekurzi. Nám nezbývá než pochválit tuto invenci a zároveň upozornit, že jsme nastavili omezení na maximální velikost odevzdávaného řešení (na $128kB$), takže příště již tato technika nepůjde použít.

```
program Rozklad;
const MAX_N = 40;
var
 { Globální pole, do kterého si rekurze připravuje sčítance. }
  scitance: array [1..MAX_N] of integer;

{ Rozloží číslo n na sčítance, mezi kterými jsou jen čísla od 1 do max. }
{ Číslo i představuje počet již hotových čísel v poli scitance. }
```

```
procedure rozloz(n, i, max: integer);
var
  j: integer;
begin
  if n = 0 then begin
    { Máme připraveny všechny sčítance, vypíšeme je (konec rekurze). }
    for j := i downto 1 do begin
      if j < i then write('+');
      write(scitance[j]);
    end;
    writeln;
  end else begin
    { Provedeme další krok rekurze. }
    if n < max then max := n;
    for j := 1 to max do begin
      scitance[i+1] := j;
      rozloz(n-j, i+1, j);
    end;
  end;
end;

var
  n: integer;
begin
  read(n);
  rozloz(n, 0, n);
end.
```

21-3-6 Pan Cowmess — Martin Mareš & Milan Straka

Nejprve rozluštěme, co program pana Cowmesse dělal. Číslo zadané v registru x nejprve zapsal ve dvojkové soustavě pomocí právě 32 cifer. Pak opakovaně hledal páry tvořené nulou a jedničkou a přepisoval je na páry dvojek. Nakonec zkontroloval, jestli byly všechny číslice přepsané, a podle toho odpověděl. Jinými slovy testoval, zda je v dvojkovém zápisu čísla právě 16 jedniček. Použít na tak triviální úlohu celých 21 instrukcí je skutečně naprostá nehoráznost.

Můžeme napsat daleko jednodušší program, který bude při převodu do dvojkové soustavy rovnou počítat jedničky:

```
    jupiii: a=x%2
            b=b+a
            x=x/2
            if x<>0 => jump jupiii
            if b<>16 => jump ooops
            y=1
    ooops:
```

Krásných 6 instrukcí by si zasloužilo potlesk, ale vavřínový věnec ještě ne. Stačí jich totiž pouhá polovina. Dopomůže nám k tomu starý dobrý operátor bitové selekce, který nám už nejednou nečekaně zamíchal karty.

Pokud spočítáme `x@x`, získáme číslo, které obsahuje tytéž jedničky co `x`, jenže „sklepané doprava.“ Pak už jen toto číslo porovnáme s číslem 65535 (16 jedniček umístěných úplně vpravo) a máme vyhráno:

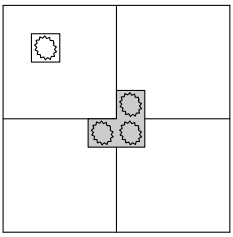
```
        a=x@x
        if a<>65535 => jump oooops
        y=1
oooops:
```

Po tomto skandálním odhalení již zbývá jen popřát panu Cowmessovi, aby mu jeho zákazníci snížili odměnu na polovic za každou přebytečnou instrukci :)

21-4-1 Dláždění šachovnice — Pavel Klavík

„Právě jste splnili kvalifikační test na soutěž o Nejlepšího crackera roku 2009!“ Podobnou zprávu by obdržela většina řešitelů této úlohy. Naopak to, co místy pokulhávalo, byl popis řešení. Pojďme se tedy na celý problém šachovnice podívat pořádně.

Každou šachovnici o rozměru 2^N lze pokrýt L-ky bez jednoho libovolného políčka a budeme to dokazovat matematickou indukcí. Pro $N = 1$ máme šachovnici 2×2 a tu snadno vyplním jedním L-kem. Pro $N > 1$ celou šachovnici rozdělíme na čtvrtiny (o straně 2^{N-1}) a rádi bychom využili indukčního předpokladu. V jedné čtvrtině se nachází chybějící políčko a tuto čtvrtinu umíme z předpokladu celou vyplnit L-ky. Zbývá nám ještě vyplnit zbývající tři čtvrtiny. Abychom mohli nasadit indukční předpoklad, musíme z každé čtvrtiny vybrat jedno políčko, které nevyplníme. Pokud vybereme tři rohová políčka, jak je naznačeno na obrázku, jsme je schopni zaplnit jedním L-kem. Zbytek každé čtvrtiny potom umíme vyplnit z indukčního předpokladu. Tím jsme nalezli vyplnění celé šachovnice o velikosti 2^N a důkaz je dokončen.



Důkaz nám říká, jak vytvořit rekurzivní algoritmus, který bude vyplnění přímo hledat. Stačilo by napsat rekurzivní funkci, která by dostala velikost šachovnice a pozici vykousnutého políčka a vrátila pokrytí L-kami. Časová složitost algoritmu je počet políček šachovnice, tedy $O(2^{2N})$ a rychleji to ani nejde, protože pozici přesně tolika L-ek musíme popsat. Psát však do řešení přímo program nebylo zapotřebí.

21-4-2 Dosah kouzla

Michal „vorner“ Vaner & Jan „Moskyt“ Matějka

Vážení čtenáři, těsně před redakční uzávěrkou ročenky jsme zjistili, že následující řešení je chybné a pro některé vstupy vrací nesprávný výstup. Přesto jsme se rozhodli jej zařadit v původním znění, nicméně ani o to (snad už) správné nebudete ochuzeni – následuje hned za ním.

Úloha je zcela zřejmá. Prostě ke každému vrcholu projdu všechny ostatní a spočítám vzdálenosti. Z nich vybrat největší je cvičení první hodiny programování. Časová složitost je zcela očividně $O(n^2)$.

Ale moment, 9 bodů za dva jednoduché cykly v sobě? To by se KSP dopustilo urážky na cti všech řešitelů. To půjde lépe.

Toto řešení by určitě fungovalo na libovolnou množinu bodů. Tak proč je zadaný sklep konvexní? Všimneme si, že když se posadíme do jednoho vybraného rohu k Mírielovi, od nás doleva vyběhne krysa a poběží po obvodu, tak se bude napřed stále vzdalovat, než doběhne do nejvzdálenějšího rohu, a potom už se zase bude pouze přibližovat. Jinými slovy, posloupnost vzdáleností vrcholů je v první části rostoucí a v druhé klesající.

Jak takové věci využít? Mohli bychom použít něco, co až nápadně připomíná binární vyhledávání v setříděném poli. Máme nějaký interval (na začátku je to celá posloupnost vrcholů n-úhelníku bez jednoho, ve kterém sedíme). Vybereme si vždy prostřední, tím interval rozdělíme na dvě části. Z každé části si vybereme jeden vzorek a vezmeme tu, kde vyjde vzdálenost větší (v té se bude nacházet největší). Pokud se nám vzdálenosti vzorků rovnají, pak interval nemůžeme rozpůlit podle prostředního, ale určitě bude největší mezi nimi. Pokud budeme brát vzorky ve vzdálenosti 1/4 od krajů, pak interval také zmenšíme na polovinu.

Pokud takto nalezneme nejvzdálenější vrcholy pro každý vrchol, zlepšíme si časovou složitost na $O(n \cdot \log n)$. S tím se ale ještě nespokojíme.

Když máme spočítaný nejvzdálenější vrchol od vrcholu, ve kterém právě sedíme a přesedneme si o jeden vrchol po směru hodinových ručiček (trochu obtížná představa na dobu, kdy byly hodiny přesýpací), co se stane s nejvzdálenějším vrcholem? Buď bude stále na tom samém místě, nebo se posune také po směru hodinových ručiček (nevíme ale, jak daleko). A vida, to přímo nabádá k tomu na tom založit algoritmus.

Na začátku tedy nějak seženeme nejvzdálenější vrchol k prvnímu vrcholu. Poté si $(n-1)\times$ „přesedneme“. Tím jakoby jednou oběhneme celé sklepení. Přestože nevíme, jak daleko se při každém přesunu pohne nejvzdálenější vrchol, po celém kolečku určitě skončí na stejném místě a nikdy nás nepředběhne (to by nás musel dohnat a vrchol sám sobě nejvzdálenější být nemůže). Tedy, oběhne

celé sklepení také právě jednou. Toto obíhání má tedy složitost $O(n)$. Pokud stihneme spočítat nejvzdálenější vrchol v čase $O(n)$, tak jsme vyhráli. Rychleji to už určitě nepůjde, načtení vstupu nám zabere alespoň takovéto množství času.

Co se týče paměťové složitosti, stačí nám pamatovat si jednotlivé vrcholy.

Ještě si lze všimnout, že si při porovnávání můžeme zcela odpustit odmocňování, neboť odmocnina z kladných čísel je rostoucí, a tedy $\sqrt{a} < \sqrt{b} \Leftrightarrow a < b$.

```
program kouzlo;

const max = 1000;
{ Sklep s více vrcholy už je prakticky kruhový }

type
    tcoord = record
        x, y: real;
    end;
    tcorners = array[0..max-1] of tcoord;

var
    n: integer;
    corners: tcorners;
    f: text;
    best, bestLoc: real;
    bestPos, bestPair1, bestPair2: integer;
    i: integer;

function dist2(i, j: integer): real;
begin
    dist2 := (corners[i].x - corners[j].x) * (corners[i].x
    - corners[j].x) + (corners[i].y - corners[j].y)
    * (corners[i].y - corners[j].y);
end;

begin
    { Trocha načítání }
    assign(f, 'kouzlo.in');
    reset(f);
    n := 0;
    while not seekeof(f) do begin
        read(f, corners[n].x, corners[n].y);
        inc(n);
    end;
    close(f);
    { Zatím nic nenalezeno }
    bestPos := 1;
    best := 0;
    { Ke všem najdeme ten nejlepší}
    for i := 0 to n - 1 do begin
        bestLoc := dist2(i, bestPos mod n);
```

```
        { Posouváme tak dlouho, dokud se to zlepšuje }
        while dist2(i, (bestPos + 1) mod n) > bestLoc do
        begin inc(bestPos);
            bestLoc := dist2(i, bestPos mod n);
        end;
        { Zlepšilo se celkově? }
        if bestLoc > best then begin
            bestPair1 := i;
            bestPair2 := bestPos mod n;
            best := bestLoc;
        end;
    end;
    { Vypsat }
    writeln('[', corners[bestPair1].x, ',',
        corners[bestPair1].y, '], [', corners[bestPair2].x,
        ',', corners[bestPair2].y, ']');
end.
```

Proč předchozí řešení není správné: Představme si elipsu s vysokou excentricitou (laicky řečeno „pořádně spláclou“) a po jejím obvodu rovnoměrně rozložené větší množství bodů. Pak zjevně pro některé body existují dvě lokální maxima vzdáleností, a tedy hned první tvrzení (posloupnost vzdáleností bodů nejprve stále roste a pak stále klesá) je mylné.

Po několika bezesných nocích se nám povedlo nalézt i protipříklad k samotnému řešení, který sestává z 16 bodů a chová se tak, že „podrží“ koncový bod úhlopříčky, když se blížíme druhým koncem k nejdelší úhlopříčce, a drží ho, dokud ji nemineme. Takto „obejdeme“ nejdelší úhlopříčku. Jeho konstrukce je dosti netriviální, nicméně docela intuitivní, a ponecháváme na čtenáři, aby si to v případě zájmu sám vyzkoušel. Během jeho tvorby jsme využili i následující pozorování:

- Označme BÚNO (bez újmy na obecnosti) nejdelší úhlopříčku PQ a R, S sousední body bodu P. Označme o_R a o_S osy úseček PR a PS. Pak určitě bod Q leží v polorovině $o_R R$ (jinak by byl dále od bodu R než P) a v polorovině $o_S S$ (stejný případ). Tedy bod Q leží v rovinné výseči určené těmito dvěma polorovinami.
- Další zajímavé (a samozřejmě platné) tvrzení říká, že pokud existují v posloupnosti vzdáleností ostatních vrcholů od jednoho bodu A lokální maxima, pak body mezi dvěma sousedními maximy můžeme s klidným srdcem ignorovat. Označíme-li dvě sousední maxima AK a AL a nějaký vrchol mezi K a L jako M, pak osa KM a osa LM se určitě protínají až „za“ bodem M, takže v příslušné výseči není žádný vrchol našeho konvexního mnohoúhelníka.

- Zároveň pokud jsme už nějaký vrchol zpracovali (nalezli a uložili kandidátku na nejdelší úhlopříčku v něm končící), nemusíme se jím již dále zabývat a též ho můžeme smazat.

Správné řešení vypadá trochu jinak, nicméně je taktéž lineární.

Představme si, že zadaný mnohoúhelník valíme po přímce a zajímá nás, jak vysoko maximálně zasáhne, kde už může být strop. Zjevně nezasáhne výš, než je délka jeho nejdelší úhlopříčky, ale níž také ne, abychom nenabourali nejdelší úhlopříčkou do stropu.

Nuže, jak valit? Veďme dvě rovnoběžky tak, že každá z nich má s mnohoúhelníkem společný právě jeden bod A_i, resp. A_j (uzavírají mnohoúhelník mezi sebe), a otáčejme s nimi okolo těchto bodů, dokud některá nesplyne se stranou mnohoúhelníka. Nechť je to BÚNO A_iA_{i+1}, nadále tedy budeme otáčet okolo bodů A_{i+1}, A_j a přeznačením se dostáváme do výchozí situace. Jakmile jsme otočili přímky o 180°, prošli jsme nutně okolo nejdelší úhlopříčky, takže nám již jen stačí vybrat správnou dvojici A_i, A_j ...

Velký dík patří Pavlu Veselému za obětavé hledání protipříkladů a Michaelu Shamosovi, profesoru Carnegie Mellon University, který řešení na tomto principu publikoval v roce 1978 ve své doktorské práci.

21-4-3 Stavění robota

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

Netrpaslická nebyla jen velikost Boendalova čísla, ale i obtížnost této úlohy. Provolejme třikrát sláva těm řešitelům, kteří se namočení do algebry nezalekli, a pojďme si osvětlit řešení.

Jak si jistě pamatujete, cílem bylo spočítat kombinační číslo $\binom{N}{K}$, respektive jeho zbytek po dělení číslem R. Standardní metody na počítání kombinačních čísel zde selhávají – konstrukce dynamickým programováním by trvala příliš dlouho a rekurentní definice kombinačních čísel by potřebovala dlouhá čísla, která v povolených jazycích k dispozici nemáme. Je tedy třeba zkusit využít dělení modulo, které bychom tak jako tak museli provést.

Vezměme si hledané kombinační číslo $\binom{N}{K}$. Možná už ze školy víte, že kombinační číslo se dá také zapsat jako $H/K!$, kde H je součin K klesajících čísel od N do $N-K+1$. Řečeno jazykem matematikovým:

$$H = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot (N-K+1)$$

$$\Downarrow$$

$$\binom{N}{K} = H/K!$$

Tento zápis je pro kombinační čísla přirozenější, neboť vystihuje lépe jejich podstatu – kombinační číslo je počet K-tic, které můžeme vybrat z N prvků, pokud neuvažujeme opakování. A přesně tak počítá i vzorec $H/K!$. Nejprve vybereme první prvek K-tice, na to máme N možností, pak zvolíme druhý, to máme $N-1$ možností (ten první znovu vybrat nemůžeme) a tak dále, až vybereme poslední, K-tý, na něj máme $N-K+1$ možností. Nicméně tímto výběrem jsme si porušili druhé pravidlo – některé K-tice se nám tam opakují. My ale víme, které a kolikrát. Tímto taháním prvků jsme vybrali každou K-tici právě tolikrát, kolik je zpřeházení (permutací) K prvků. Například pokud vybíráme trojice z prvků $\{1 \dots 6\}$, pak trojici $(1,2,3)$ jsme vybrali ještě jako trojice $(1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)$. Abychom všechny tyto případy zahodili, postačí nám celé číslo H vydělit počtem permutací na K prvků, a jak všichni ví, těch je $K!$.

Nám se tenhle zápis bude hodit, protože dělení modulo se chová hezky při sčítání a násobení. Bota nás bude tlačit jen tehdy, když budeme chtít spočíst $H/K!$, neboť celočíselné dělení při operacích se zbytky k dispozici nemáme. Budeme si jej muset „odsimulovat“.

Půjdeme na to přes prvočíselné reprezentace, kterých se každý nabažil na základní škole – rozložíme zadané kombinační číslo na mocniny prvočísel. Začneme zlehka, rozkladem součinu H. Nejprve si najdeme Eratosthenovým sítem všechna prvočísla od 1 do N. U čísel složených v tomto rozsahu si ještě poznačíme, jaké největší prvočíslo je dělí (to nám síto udělá takřka samo).

S prvočísly po ruce již můžeme rozkládat. Budeme si udržovat aktuální rozklad čísla (který nemusí být prvočíselný, například $360 = 72 \cdot 5$), setříděný podle základů. Tento rozklad si budeme pamatovat v poli, kde hodnota Z-tého prvku je rovna exponentu E takovému, že Z^E je přítomno v našem rozkladu. Tento seznam rozkladů budeme postupně drolit na menší kousky.

Vezměme největší neprobraný základ Z a odpovídající exponent E. Je-li Z číslo složené, pak se podíváme do síta a najdeme největší prvočíslo P, které Z dělí. Tím se nám rozloží Z^E na $P^E \cdot (Z/P)^E$. Obě čísla jsou menší než Z, přidáme je tedy s odpovídajícím exponentem do rozkladu a pokračujeme dále. Je-li Z prvočíslo, pak mohu tento základ a exponent vzít jako hotový (a přejdu k menšímu základu).

Pro náš příklad bude rozklad probíhat takto:

$$360 = 72^1 \cdot 5^1 = 24^1 \cdot 5^1 \cdot 3^1 = 8^1 \cdot 5^1 \cdot 3^2 = \\ = 5^1 \cdot 4^1 \cdot 3^2 \cdot 2^1 = 5^1 \cdot 3^2 \cdot 2^3.$$

Použití těchto neúplných rozkladů se nám velice hodí – algoritmus totiž může rozložit libovolné číslo zadané součinem – a přesně takové je H, které rozložit potřebujeme. Nyní přišel čas vyřešit dělení číslem $K!$. To můžeme udělat chytrým trikem – už dopředu víme, že výsledek bude celočíselný. Začněme

tedy s následujícím rozkladem:

$$\binom{N}{K} = N^1 \cdot (N-1)^1 \cdot \ldots \cdot (N-K+1)^1 \cdot K^{-1} \cdot (K-1)^{-1} \cdot \ldots \cdot 2^{-1}$$

Rozklad je to korektní a algoritmus nám určitě neporuší, neboť jakmile budeme rozkládat čísla menší než K, už jsme rozložili všechna větší než K, a tedy (díky celočíselnosti výsledku) nikde nemůžeme narazit na záporný exponent.

Máme rozloženo, co dál? Teď si trochu započítáme. Jak jsme řekli už výše, při operacích modulo můžeme používat sčítání a násobení, jak jsme zvyklí. Tedy pokud chceme spočíst $\binom{N}{K}$ a už známe jeho prvočíselný rozklad, budeme dělit modulo ne celé číslo, ale jednotlivé mocniny prvočísel, a mezivýsledky vynásobíme dohromady. Na konci ještě nesmíme zapomenout celý součin vydělit modulo R, protože během násobení se nám mohlo stát, že jsme z modulo R utekli.

Matematik by to napsal takto: Pokud si zapíšeme prvočíselný rozklad $\binom{N}{K}$ jako ${p_1}^{l_1} \cdot {p_2}^{l_2} \cdot \ldots \cdot {p_k}^{l_k}$, pak platí:

$$\binom{N}{K} \equiv {p_1}^{l_1} \cdot {p_2}^{l_2} \cdot \ldots \cdot {p_k}^{l_k} \equiv ({p_1}^{l_1}) \cdot ({p_2}^{l_2}) \cdot \ldots \cdot ({p_k}^{l_k}) \pmod{R}$$

Fajn, přešli jsme kopec, přeplavali řeku a nyní nám zbývá se poprat s tím, máme-li velkou mocninu prvočísla p^l (ještě stále se nám celý výsledek nevejde do integeru), jak rychle spočítat její zbytek modulo R. Naštěstí víme, že exponent l našeho prvočísla bude relativně malý (to, že exponent závisí nejvýše lineárně na N, můžeme vypozorovat například z rovnice $\sum_{k=0}^{N} \binom{N}{k} = 2^N$). Můžeme si tedy dovolit například přepočítání v $O(\log_2 N)$. To uděláme tak, že rozložíme l na součet mocnin dvojky, kde se každá vyskytuje nejvýše jednou (rozmyslete si, že každé číslo lze rozložit na takový součet). Nyní máme číslo ve tvaru $p^{2^a+2^b+2^c \ldots 2^z}$, kde $0 \leq a < b < cd \ldots < z \leq \log_2 P$. To si podle starých dobrých aritmetických pravidel přepíšeme na $p^{2^a} \cdot p^{2^b} \cdots p^{2^z}$ a můžeme přistoupit k počítání. Jako obvykle u toho využijeme faktu, že zbytek součinu modulo dvou čísel je roven součinu zbytků jednotlivých čísel.

Začneme od 0, $p^1 \bmod R$ uložíme jako základ. Pak v každém kroku přistoupíme k další mocnině. Nejprve si ověříme, jestli tato mocnina zrovna v našem rozkladu figuruje, a pokud ano, číslo, kterým máme přinásobit výsledek, vypočítáme snadno z předchozího: $p^{2^a} = p^{2^{a-1}} \cdot p^{2^{a-1}}$.

A tím jsme hotovi! Ukázali jsme si, jak spočítat modulo mocninu velkého prvočísla, tyto výsledky vynásobíme dohromady pro každé prvočíslo, které se účastnilo rozkladu kombinačního čísla, a celkový součin vypíšeme na výstup (nezapomeneme v průběhu dělit modulo R, aby nám výsledek nepřetekl).

Zbývá prohodit pár slov o časové a paměťové složitosti. Na rozklad na prvočísla jsme potřebovali dvě pole o N prvcích, zbytek se dá zvládnout jen s pár

integery. Paměťové nároky jsou tedy $O(N)$. Co se týče nároků časových, bylo by třeba spočítat, jak rychle proběhlo hledání prvočísel Eratosthenovým sítem. To zde nebudeme dokazovat, pouze poznamenejme, že je to $O(N \log \log N)$. Co se týče druhé části algoritmu, bude asymptoticky alespoň $O(N \log N)$. Na optimalitu si nároky neděláme, neboť u teorie čísel skoro vždy platí, že můžeme nasadit lepší síto a dosáhneme asymptoticky lepšího výpočtu.

Dík patří Zbyňkovi Faltů za vzorové řešení!

```
program robot;

var
    sito     : array [ 1..1000000 ] of longint;
    N, K, M  : longint;
    i, j     : longint;
    vysledek : longint;



    {rozklad[i] odpovídá největšímu
     exponentu z takovému, že i^z dělí (N nad K).}

    rozklad    : array [ 1..1000000 ] of longint;
    mocnina  : longint;

begin
    readln(N,K,M);

    for i:=2 to N do begin
        sito[i]:=0;
        rozklad[i]:=0;
    end;

    {Jednoduché zrychlení.}
    if K > N-K then
        K:=N-K;

    {Eratostenovo síto, které navíc počítá
     největší prvočíslo dělící každé složené
     číslo.}
    for i:=2 to N do begin
        if sito[i]=0 then begin
            j:=i*2;
            while j<=N do begin
                sito[j]:=i;
                j:=j+i;
            end;
        end;
    end;

    {Kombinačníčíslo je součin čísel od
```

```
     N-K+1 do N ...}
    for i:=N-K+1 to N do
        inc(rozklad[i]);

    {...dělen k! }
    for i:=2 to K do
        dec(rozklad[i]);

    vysledek:=1;
    for i:=N downto 2 do begin
        if sito[i]=0 then begin
            mocnina:=i mod M;
            while rozklad[i]<>0 do begin
                {Nejvyšší číslo rozkladu je prvočíslo,}
                {zpracujeme je. Rozložíme exponent na
                součet mocnin dvojky a ty spočítáme
                prostým mocněním modulo M.}

                if rozklad[i] mod 2 = 1 then
                    vysledek:=(vysledek*mocnina) mod M;
                {Zkus vyššíninu.}
                mocnina:=(mocnina*mocnina) mod M;
                rozklad[i]:=rozklad[i] div 2;
            end;
        end else begin
            {Nejvyšší číslo rozkladu je číslo složené.}
            rozklad[sito[i]]:=rozklad[sito[i]]+rozklad[i];
            rozklad[i div sito[i]]:=rozklad[i div sito[i]]
                +rozklad[i];
        end;
    end;

    writeln(vysledek);
end.
```

21-4-4 Heslo — Roman Smrž

K dané permutaci s prvky, které se mohou opakovat, lze nalézt následující v lineárním čase s její délkou (stačí najít nejdelší nerostoucí konec, cifru před ním prohodit s nejbližší vyšší z onoho konce a ten pak obrátit); K-násobné zopakování tohoto postupu vede k řešení úlohy, leč v čase $O(KN)$, přičemž K by mohlo být poměrně velké i pro malá N, takže se pokusme o rychlejší algoritmus.

Kdybychom chtěli najít K-tou permutaci v lexikografickém uspořádání od začátku (tedy ne od nějaké dané na vstupu), vezmeme si nejprve tu nejnižší (cifry uspořádané vzestupně) a podíváme se, za jak dlouho se změní první cifra. Zřejmě se před tím musí vystřídat všechny permutace, které na ni začínají, a těch je tolik, kolik je permutací zbývajících cifer (označme tento počet P).

$(P+1)$-ní permutace tedy vypadá tak, že na prvním místě je druhá nejmenší cifra a zbytek je opět setříděný vzestupně (rozmyslete si, že od první permutace se liší jen prohozením dvou cifer).

Pokud je $K \leq P$, cifru na prvním místě měnit nepotřebujeme a jen zopakujeme týž postup pro permutaci zbylých čísel. V opačném případě zaměníme první cifru s nejbližší vyšší, od K odečteme P, jakožto počet permutací, které jsme přeskočili, a opět můžeme postupovat stejně (podívat se, jestli dalším změnou na prvním místě K „nepřeskočíme“, a buď příslušné prohození udělat, nebo jít na další pozici) dokud K nesnížíme na 0.

Popsaný postup však funguje, jen když máme za cifrou, s níž právě pracujeme, cifry seřazené. Všimněme si ale, že stejně lze cifry i snižovat: pokud máme permutaci začínající na nějakou cifru, za níž jsou cifry již uspořádané, můžeme tu první prohodit s nejbližší nižší, čímž se dostaneme zpět o počet permutací na těchto cifrách bez té, kterou jsme dali na začátek.

Půjdeme tedy odzadu a cifru na každé pozici budeme postupně vyměňovat za menší až na tu nejnižší s tím, že cifry za ní jsou vždy seřazené (na začátku – pro pozici na konci – to platí triviálně a po skončení úprav tuto podmínku zřejmě rozšíříme i na aktuální pozici); zároveň budeme příslušně zvyšovat K. Ve chvíli, kdy K bude menší než počet permutací od aktuální pozice do konce, znamená to, že cifry nalevo měnit už nepotřebujeme a stačí jen najít K-tou permutaci na cifrách, které jsme prošli.

Abychom vždy nejbližší vyšší cifru k té aktuální našli rychle, místo „přeskládaného“ konce si budeme pamatovat počty jednotlivých cifer, které jsme viděli, což také stačí k tomu, abychom věděli, jak má vypadat.

Zbývá si už jen rozmyslet, jak zjišťovat počty permutací. Jelikož se cifry mohou opakovat, musíme $N!$ ještě vydělit faktoriály počtů jednotlivých cifer. Počítat to pokaždé celé by stálo příliš času, ale naštěstí vždy potřebujeme jen drobnou úpravu, pokud do permutace délky n přidáváme nějakou cifru c, počet permutací stačí vynásobit $(n+1)$ a vydělit novým počtem cifer c; analogicky pak pro odebírání a vyměňování cifer.

Kromě načtení vstupu projdeme celou permutaci nejvýše dvakrát a pro každou pozici potřebujeme jen konstantní čas, takže časová složitost je $O(N)$, stejně tak i paměťová. To ovšem za předpokladu, že čísla, se kterými budeme pracovat, budou dostatečně malá na to, abychom čas spotřebovaný na jednotlivé operace mohli za konstantní považovat. Nicméně, jak bylo naznačeno v úvodu, počty permutací mohou být poměrně velké; v případě permutací deseti cifer by jich mohlo být $O(10^N)$ a na potřebné výpočty s takovýmito čísly by byl nutný čas $O(\log 10^N) = O(N)$, což by vedlo na celkovou časovou složitost $O(N^2)$.

```
#include <stdio.h>

int n, k;
int heslo[N_MAX];       // celá permutace
int pocty[10];          // počty jednotlivých cifer v podpermutaci
                        // od aktuální pozice do konce
int poc_perm = 1;       // počet podpermutací, které mohou vytvořit
                        // cifry zaznamenané v poli 'pocty'
int main(void)
{
        scanf("%d%d\n", &n, &k);
        for (int i = 0; i < n; i++) heslo[i] = getchar()-'0';

        int i;
        for (i = n-1; k >= poc_perm; i--) {

                // každou cifru postupně vyměňujeme za menší ...
                for (int j = heslo[i]-1; j >= 0; j--) {
                        if (pocty[j]) {
                                poc_perm *= pocty[j]--;
                                poc_perm /= ++pocty[heslo[i]];

                                k += poc_perm;
                                heslo[i] = j;
                        }
                }

                // ... a nakonec rozšíříme podpermutaci i o aktuální pozici
                poc_perm *= n-i;
                poc_perm /= ++pocty[heslo[i]];
        }

        for (i++; i < n; i++) {
                // najdeme nejnižší dostupnou cifru ...
                heslo[i] = 0;
                while (!pocty[heslo[i]]) heslo[i]++;

                // ... tu odebereme z podpermutace ...
                poc_perm *= pocty[heslo[i]]--;
                poc_perm /= n-i;

                // ... a poté se ji budeme poukoušet postupně zvyšovat
                for (int j = heslo[i]+1; j < 10 && k >= poc_perm; j++) {
                        if (pocty[j]) {
                                k -= poc_perm;
                                poc_perm *= pocty[j]--;
                                poc_perm /= ++pocty[heslo[i]];
                                heslo[i] = j;
                        }
                }
        }
```

```
        for (int i = 0; i < n; i++) putchar(heslo[i]+'0');
        putchar('\n');
        return 0;
}
```

21-4-5 Znaky — Petr Onderka

Úlohu vyřešíme jednoduchým algoritmem: v každém kroku přečteme jeden znak ze vstupu, převedeme ho do nějaké vhodné reprezentace a postupně ho porovnáme se všemi dosud přečtenými jedinečnými znaky. Pokud se neshoduje s žádným z nich, přidáme ho do seznamu. Protože nám v této úloze jde hlavně o paměťovou náročnost, soustředíme se na to, jak reprezentovat znaky v paměti co nejúsporněji.

Vzhledem k tomu, že v každém řádku znaku je vybarvený právě jeden pixel, nabízí se řešení ukládat znak jako posloupnost N čísel s hodnotami $1 \dots N$, kde i. číslo udává, v kolikátém sloupci i. řádku je vybarvený pixel. Kolik bitů bude jeden takovýto znak zabírat? Na každý řádek potřebujeme $\lceil \log_2 N \rceil$ b, na celý znak tedy $N \cdot \lceil \log_2 N \rceil$ b. Abychom takovéto velikosti opravdu dosáhli, nemůžeme ukládat každý řádek do samostatného prvku pole, ale budeme znak v této reprezentaci chápat jako posloupnost bitů, které rozdělíme po osmi a uložíme do pole bytů. Na konci pole pak zbude až sedm nevyužitých bitů, ale s tím už nic neuděláme. Časová složitost tohoto řešení je $O(K \cdot (N^2 + KN))$, protože každý z K znaků nejdříve načtu v čase $O(N^2)$ a pak porovnám s až K přečtenými znaky řádek po řádku. Lepší časové složitosti by šlo dosáhnout třeba použitím binárního vyhledávání, ale o čas nám tentokrát moc nejde.

K dosažení menší paměťové náročnosti můžeme využít toho, že každé číslo sloupce se v zápisu znaku vyskytuje jen jednou. Hodnota pro každý řádek tak nebude určovat, v kolikátém sloupci ze všech možných je vybarvený pixel, ale budeme brát v úvahu jen povolené sloupce, tedy ty, ve kterých ještě není žádný vybarvený pixel. Když zapíšu každý řádek jako dvojici číslo sloupce / maximální číslo sloupce, tak znak s $N = 5$ a zápisem 2/5, 5/5, 3/5, 1/5, 4/5 (zabírající 15 b) můžu s využitím výše zmíněného postupu zapsat jako 2/5, 4/4, 2/3, 1/2, 1/1 (8 b). Je vidět, že na uložení předposledního řádku stačí jeden bit a pro poslední řádek je to dokonce nula bitů. Na zapsání jednoho znaku tak potřebujeme $\sum_{i=0}^{N-1} \lceil \log_2(N-i) \rceil$ b $= \sum_{i=1}^{N} \lceil \log_2 i \rceil$ b.

Zatím jsme ještě nevyužili toho, že na každé diagonále je vybarvený nejvýše jeden pixel. Použijeme stejný postup jako v předchozím případě, jenom pro každý řádek budou povolené ty pixely, pro které sloupec a obě diagonály, na kterých leží, neobsahují zatím žádný pixel. Výše uvedený znak tak můžeme přepsat jako 2/5, 2/2, 2/2, 1/1, 1/1 (5 b). Celková velikost takto zapsaného znaku může být různá i pro dva znaky se stejným N.

Časová složitost těchto dvou řešení, využívajících povolené sloupce, je $O(K \cdot (N^2 + KN^2)) = O(K^2N^2)$. Zhoršení oproti první variantě je způsobené tím, že při porovnávání znaků musíme navíc pro každý řádek znaku ze seznamu už přečtených projít seznamy povolených sloupců, abychom zjistili, kolik bitů máme přečíst při čtení následujícího řádku.

Další možností, jak ještě snížit paměťové nároky, je použití trie, ale my zkusíme něco jiného: řešení, které má ještě menší paměťové nároky, ale na druhou stranu zase velkou časovou složitost. Jeho myšlenka je velice jednoduchá: všechny možné znaky si očíslujeme a pro každý znak si budeme pamatovat jen jeho číslo. Pokud je možných znaků M, tak každý z nich bude zabírat $\lceil \log_2 M \rceil$ b. Jeden znak ani do méně bitů zapsat nejde, protože pak by možných zápisů bylo méně než různých znaků. Když teď do seznamu přečtených jedinečných znaků ukládáme přímo čísla, mohli bychom, podobně jak jsme to už dělali, počítat s tím, že číslo, které se v seznamu už vyskytlo, tam znovu nebude. Jde to ale líp. Tentokrát nám totiž nezáleží na pořadí a tak si můžeme tento seznam udržovat setříděný. Když v seznamu najdu číslo x, tak vím, že za ním musí následovat číslo mezi $x+1$ a M, takže na jeho uložení bude stačit $\lceil \log_2(M-x) \rceil$ b. Jinou variantou, jak si přečtené znaky ukládat, je posloupnost M bitů, kde i-tý bit určuje, jestli jsem už přečetl znak s číslem i. Rozumným kompromisem mezi těmito dvěma řešeními je použít ze začátku seznam znaků a jakmile jeho velikost překročí M bitů, přejít na druhou variantu.

Jak ale vlastně určíme číslo znaku, který dostaneme na vstupu? Snadno, procházíme postupně všechny permutace N čísel a pokud některá z nich vyhovuje definici znaku, započítáme ji. Pokud narazíme na tu z nich, která je shodná se zpracovávaným znakem, známe jeho číslo. Ještě před přečtením prvního znaku ze vstupu ale musíme podobným způsobem projít všechny znaky, abychom věděli, kolik jich je celkem a kolik bitů tak budeme potřebovat na jeden znak.

A nakonec časová složitost: $O(NN! + K(NN! + N!)) = O(KNN!)$. Na začátku totiž spočítáme včechny možné znaky a pak pro každý znak na vstupu určíme jeho číslo a v případě první varianty jej porovnáme s už přečtenými unikátními znaky, kterých nemůže být víc než $N!$. Takto velká časová složitost samozřejmě způsobí praktickou nepoužitelnost popsaného algoritmu už i pro relativně malá N.

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

// znak reprezentovaný polem řádků, pro každý řádek
// je hodnotou číslo vybrveného sloupce
struct znak {   unsigned * radky;       };
```

```
struct bitove_pole {
    uint8_t * bity;
    // aktuální pozice při průchodu, pozice1 určuje byte, pozice2 bit v bytu
    unsigned pozice1, pozice2;
    // počet plně obsazených bytů, obsazených bitů v posledním bytu
    unsigned obsazeno1, obsazeno2;
    unsigned alokovano; // alokovaných bytů
};

unsigned N, K;
struct bitove_pole prectene; // seznam přečtených znaků

unsigned log_2(unsigned x) { // spočítá ceil(log_2(x))
    x = x - 1;
    unsigned result = 0;
    while (x > 0)
    {
        result++;
        x >>= 1;
    }
    return result;
}

// vrátí číslo zadaného znaku;
// je-li zadaný znak větší než největší platný znak, vrátí počet možných znaků
// prochází postupně permutace řádků a porovnává se zadaným znakem
unsigned cislo_znaku(struct znak z) {
    struct znak generovany;
    generovany.radky = malloc(sizeof(&generovany.radky) * N);
    // sloupce a diagonály zakázané v aktuálním řádku
    bool * sloupce = calloc(N, sizeof(bool));
    bool * diag1 = calloc(2 * N - 1, sizeof(bool));
    bool * diag2 = calloc(2 * N - 1, sizeof(bool));
    generovany.radky[0] = 0;
    unsigned shodnych_radku = 0;
    unsigned nalezenych_znaku = 0;
    unsigned i = 0;
    bool konec = false;
    while (!konec) {
        unsigned j = generovany.radky[i];
        if (j >= N)    { // prošli jsme celý řádek
            if (i == 0)
                konec = true; // prošli jsme všechny znaky
            else { // vracíme se o řádek výš
                i--;
                j = generovany.radky[i];
                sloupce[j] = false;
                diag1[i+j] = false;
                diag2[N+i-j] = false;
                generovany.radky[i]++;
```

```
            }
        }
             // jde umístit pixel
        else if (!sloupce[j] && !diag1[i+j] && !diag2[N+i-j])    {
            if (z.radky[i] == j && i == shodnych_radku)
                shodnych_radku++;
            if (i == N - 1)    { // poslední řádek
                if (shodnych_radku == N)
                    konec = true; // našli jsme shodný znak
        // našli jsme další (neshodný) znak, vracíme se o řádek výš
                else {
                    nalezenych_znaku++;
                    i--;
                    j = generovany.radky[i];
                    sloupce[j] = false;
                    diag1[i+j] = false;
                    diag2[N+i-j] = false;
                    generovany.radky[i]++;
                }
            } else { // jdeme o řádek níž
                sloupce[j] = true;
                diag1[i+j] = true;
                diag2[N+i-j] = true;
                i++;
                generovany.radky[i] = 0;
            }
        } else
            generovany.radky[i]++; // zkusíme další sloupec v řádku
    }
    free(generovany.radky);
    free(sloupce);
    free(diag1);
    free(diag2);
    return nalezenych_znaku;
}

void nacti_znak(struct znak * z) { // do z uloží znak načtený ze vstupu
unsigned x;
  for (unsigned i = 0; i < N; i++)
    for (unsigned j = 0; j < N; j++)
    {
      scanf("%u", &x);
      if (x == 1)
        z->radky[i] = j;
    }
}

// určuje, jestli ještě jsou v bitovém poli p nějaké nepřečtené bity
bool jde_cist_bity(struct bitove_pole * p) {
    return (p->pozice1 < p->obsazeno1) ||
        ((p->pozice1 == p->obsazeno1) && (p->pozice2 < p->obsazeno2));
```

```
}

// přečte n bitů z bitového pole
unsigned cti_bity(struct bitove_pole * p, unsigned n) {
    unsigned result = 0;
    unsigned precteno = 0;
    while (precteno < n) {
        if (n - precteno >= 8 - p->pozice2) // ctu do konce bytu
        {
            result |= (p->bity[p->pozice1] >> p->pozice2) << precteno;
            precteno += 8 - p->pozice2;
            p->pozice1++;
            p->pozice2 = 0;
        } else {
            result |= ((p->bity[p->pozice1] >> p->pozice2) &
                    ((1 << (n-precteno)) - 1)) << precteno;
            p->pozice2 += n - precteno;
            precteno += n - precteno;
        }
    }
    return result;
}

// přečte z bitového pole číslo z {a, ..., b}
unsigned cti_z_intervalu(struct bitove_pole * p, unsigned a, unsigned b) {
    return a + cti_bity(p, log_2(b-a+1));
}

// zapíše na konec bitového pole n bitů uložených v parametru x
void zapis_bity(struct bitove_pole * p, unsigned x, unsigned n) {
    if ((p->alokovano - p->obsazeno1) * 8 - p->obsazeno2 < n) {
        // musíme zvětšit pole
        p->alokovano = p->alokovano + (n -
            ((p->alokovano - p->obsazeno1) * 8 - p->obsazeno2) + 7)/8;
        // +7 kvůli zaokrouhlení nahoru
        p->bity = (uint8_t *)realloc(p->bity, p->alokovano);
    }
    unsigned zapsano = 0;
    while (zapsano < n) {
        if (p->obsazeno2 == 0)
             // vynuluju nově přidělenou paměť
            p->bity[p->obsazeno1] = 0;
        if (n - zapsano < 8 - p->obsazeno2) {
            p->bity[p->obsazeno1] |= (x >> zapsano) << p->obsazeno2;
            p->obsazeno2 += n - zapsano;
            zapsano += n - zapsano;
        }    else {
            p->bity[p->obsazeno1] |= ((x >> zapsano) &
                ((1 << (8 - p->obsazeno2)) - 1)) << p->obsazeno2;
            zapsano += 8 - p->obsazeno2;
            p->obsazeno1++;
```

```
            p->obsazeno2 = 0;
        }
    }
}

// zapíše x na konec p, víme, že x je z {a, ..., b}
void zapis_z_intervalu(struct bitove_pole * p,
                unsigned x, unsigned a, unsigned b) {
    zapis_bity(p, x - a, log_2(b-a+1));
}

// přidá do bitového pole hodnotu x, M je maximální hodnota
// zachovává setříděnost, takže musíme přepsat celé pole,
// abychom mohli vložit novou hodnotu
void zapis_cislo(struct bitove_pole * p, unsigned x, unsigned M) {
    struct bitove_pole nove;
    nove.alokovano = p->alokovano;
    nove.bity = malloc(nove.alokovano);
    nove.obsazeno1 = nove.obsazeno2 = 0;
    bool zapsano = false;
    unsigned min = 0;
    p->pozice1 = p->pozice2 = 0;
    while (jde_cist_bity(p)) {
        unsigned c = cti_z_intervalu(p, min, M);
        if (!zapsano && x < c) {
            zapis_z_intervalu(&nove, x, min, M);
            min = x + 1;
            zapsano = true;
        }
        zapis_z_intervalu(&nove, c, min, M);
        min = c + 1;
    }
    if (!zapsano)
        zapis_z_intervalu(&nove, x, min, M);

    free(p->bity);
    *p = nove;
}

bool get_bit(uint8_t * p, unsigned i) { // přečte bit na pozici i z pole bytů
    return (p[i / 8] & (1 << (i % 8))) != 0;
}

// zapíše bit x na pozici i v poli bytů
void set_bit(uint8_t * p, unsigned i, bool x) {
    if (x) p[i / 8] |= (1 << (i % 8));
    else p[i / 8] &= ~(1 << (i % 8));
}

// hlavní funkce programu, vrací počet jedinečných znaků na vstupu
unsigned jedinecnych_znaku(void) {
```

```
    prectene.bity = NULL;
    prectene.alokovano = prectene.obsazeno1 = prectene.obsazeno2 = 0;
    unsigned znaku = 0;
    bool zpusob2 = false;
    scanf("%u %u", &N, &K);
    struct znak z;
    z.radky = (unsigned *)malloc(N * sizeof(&z.radky));
    for (unsigned i = 0; i < N; i++)
        z.radky[i] = N;
    unsigned M = cislo_znaku(z);
    unsigned logM = log_2(M);
    for (unsigned i = 0; i < K; i++) {
        nacti_znak(&z);
        unsigned c = cislo_znaku(z);
        if (!zpusob2) {
// použijeme první způsob: seznam přečtených znaků, uložených pod svým číslem
            prectene.pozice1 = prectene.pozice2 = 0;
            unsigned min = 0;
            bool nalezeno = false;
            while (!nalezeno && jde_cist_bity(&prectene)) {
                unsigned c2 = cti_z_intervalu(&prectene, min, M);
                if (c2 == c)
                    nalezeno = true;
                else
                    min = c2 + 1;
            }
            if (!nalezeno) {
                znaku++;
                zapis_cislo(&prectene, c, M);
            }
            if (prectene.alokovano >= (M + 7) / 8) {
                // přejdeme na druhý způsob
                zpusob2 = true;
                uint8_t * bity = (uint8_t *)calloc((M + 7) / 8, 1);
                prectene.pozice1 = prectene.pozice2 = 0;
                unsigned min = 0;
                while (jde_cist_bity(&prectene)) {
                    unsigned c = cti_z_intervalu(&prectene, min, M);
                    set_bit(bity, c, true);
                    min = c + 1;
                }
                free(prectene.bity);
                prectene.bity = bity;
            }
        }   else {
            // používáme druhý způsob: pole bitů, když je i. bit 1,
            // tak jsme už někdy přečetli znak s číslem i
            if (!get_bit(prectene.bity, c)) {
                znaku++;
                set_bit(prectene.bity, c, true);
            }
```

```
        }
    }
    free(z.radky);
    free(prectene.bity);
    return znaku;
}

int main(void) {
    printf("%u", jedinecnych_znaku());
    return 0;
}
```

21-4-6 Nejtěžší číslo — Martin Mareš & Milan Straka

Začneme podúlohou **b)**, neboť nejtěžší číslo bylo nakonec překvapivě lehké :-) Stačí si totiž vzpomenout na trik z řešení Pana Cowmesse (úloha 21-3-6): pomocí `x@x` „sklepeme“ všechny jedničky v čísle směrem k nejnižšímu řádu. Pak si ještě všimneme toho, že číslo `a` je těžší než číslo `b` právě tehdy, když `a@a` je větší než `b@b`, a řešení je nasnadě:

```
sem:    a = A[n] @ A[n]
        if a<=m => goto tam
        m = a
        y = A[n]
tam:    n = n-1
        if n>0 => goto sem
```

Program pro n čísel použije $4n + 2m$ instrukcí, kde m říká, kolikrát se během procházení pole zvýší váha čísla. To se může stát nejvýše 32-krát, takže můžeme počet instrukcí odhadnout shora výrazem $4n + 64$.

Toto řešení můžeme ještě zrychlit (díky, Jitko!), když si uvědomíme, že si místo hodnoty zatím nejtěžšího čísla a jeho sklepané verze stačí pamatovat pozici takového čísla ve vstupu:

```
sem:    B[n] = A[n] @ A[n]
        if B[n]<=B[m] => goto tam
        m = n
tam:    n = n-1
        if n>0 => goto sem
        y = B[m]
```

Tak počet provedených instrukcí snížíme na $4n + m + 1 \leq 4n + 33$.

Podúloha **a)**, tedy zvážení jednoho čísla, byla naopak lehce překvapivá :-) Nejrychlejší známý program má totiž přibližně $2 \cdot 10^9$ instrukcí, z nichž se ovšem pro každý vstup provede nejvýše pět.

Jak funguje? Nejprve si zadané číslo sklepeme a pak rozebereme 33 případů, které mohou nastat. To bychom přímočaře zvládli 32 podmínkami nebo pomocí půlení intervalu 6 podmínkami. Ale my raději nepoužijeme podmínku

žádnou: Využijeme toho, že parametr instrukce skoku může být nejen konkrétní číslo (či návěští), ale také obsah registru, a jedním skokem se přesuneme na správné místo v programu, které pouze vrátí výsledek. Vypadat to bude takto (v závorkách jsou napsány adresy instrukcí):

```
(0)      z = x@x
(1)      z = z+4
(2)      jump z
(3)      w = 32          // při z=2^32-1
(4)      jump hotovo     // při z=0
(5)      w = 1           // při z=1
(6)      jump hotovo
(7)      w = 2           // při z=3
(8)      jump hotovo
         ...
(11)     w = 3           // při z=7
         jump hotovo
         ... ...
(2^31+3) w = 31          // při z=2^31-1
hotovo:
```

(Všimněte si, že `x@x` stále potřebujeme, protože kdybychom chtěli rozebrat úplně všechny případy, nevešel by se program do 2^{32} instrukcí, které dovedeme adresovat.)

Jako zákusek přidáváme ještě řešení, které si vystačí s 12 instrukcemi bez jediného skoku. Konstanty začínající `0b` jsou dvojkové. Zkuste přijít na to, proč funguje:

```
y = x & 0b10101010101010101010101010101010
x = x & 0b01010101010101010101010101010101
y = y >> 1
x = x + y
y = x & 0b00110000110000110000110000110000
z = x & 0b00001100001100001100001100001100
x = x & 0b11000011000011000011000011000011
y = y >> 4
z = z >> 2
x = x + y
x = x + z
w = x % 63
```

21-5-1 Polomáčené mrakodrapy

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

Očekávali jsme více správných řešení – na rozdíl od minulé úlohy si tato úloha žádala spíše selský rozum než algebraické triky.

Z úlohy bylo patrné, že všechna maxima byla příliš velká na to, aby mělo triviální kvadratické řešení jakoukoli šanci. Stejně tak rozsah velikostí věžáků

a rozsah dnů, na které mohl být položen dotaz, byl příliš velký, než aby s ním šlo dělat cokoli rozumného.

Nicméně si lze všimnout, že zde pracujeme jen s tisícinou položek z celého rozsahu. Zkusíme si tu množinu tedy předzpracovat, aby se nám s ní lépe pracovalo.

Způsob, který si popíšeme, využívá zajímavého panelákového pozorování: jediné budovy, které zvyšují nebo snižují počet ostrůvků ve městě, jsou ty, které jsou „lokálními extrémy“ – jinak řečeno, jsou to takové budovy, jejichž oba sousedé jsou buď zároveň větší nebo menší než ona sama. Když totiž uvážíme posloupnost rostoucích (nebo klesajících) paneláků, voda je postupně zaplavuje jeden po druhém a počet ostrovů se nezmění. Navíc si všimneme, že „vršek“ po zatopení od počtu ostrůvků odečte sám sebe (právě se zatopil), zatímco „údolí“ po zatopení oddělí svou levou a pravou část, a tedy jeden ostrůvek přibude.

Výška budovy přesně určuje den, kdy se pričte nebo odečte jednička z výsledku. Zjistit o každém domě, zda je „vrškem“ či „údolím“, zvládneme jedním průchodem vstupními daty. Při tomto průchodu si tedy rovnou budeme ukládat neuspořádaně dvojice (Den, Změna), kde Den bude výška domu, který procházíme a Změna bude $+1$ nebo -1, podle toho, jakého typu byl onen dům.

Nyní začneme ze vstupu načítat (velice příhodně setříděné) dny a budeme podle $+1$ a -1 upravovat aktuální hodnotu. Abychom toto mohli dělat rychle, stačí si naše pomocná data setřídit podle hodnoty Den vzestupně (postačí i rychlý kvadratický algoritmus jako Quicksort) a pak obě pole procházet naráz.

Paměti nám stačilo lineárně mnoho, dokonce jsme využili jen jedno pole celých čísel ($+1$ bit, ale ten bychom dovedli zakódovat i dovnitř) o velikosti N. Pokud jsme nepoužili žádné „nefér“ praktiky jako přihrádkové třídění, pak nám celý algoritmus seběhl v čase $O(N \log N)$, respektive $O(N^2)$ (s rychlým kvadratickým tříděním).

Na závěr doporučujeme všem řešitelům praktických úloh, pokud používáte libovolný jazyk vyjma Pascalu, pak tento jazyk obsahuje zabudovanou třídící knihovnu, která pro velkou většinu vstupů bude alespoň tak rychlá jako libovolný algoritmus, který napíšete. Když ji budete používat (v praktických úlohách), tak nic nezkazíte a navíc si ušetříte spoustu písmenek a potenciálních chyb.

21-5-2 Banky — Petr Onderka

Z došlých řešení je zřejmé, že vetšina řešitelů nemá ráda chamtivé bankéře. Značná část z vás se je pokoušela zmást nesprávnými výsledky. Někteří pravděpodobně měli v plánu zopakovat Chuliův trik, a tak bance nabídli velmi líné programy, aby měli dostatek času na přípravu a provedení svých plánů. A jen pár jedinců překonalo svou nevoli vůči finančníkům a předvedli nejen funkční, ale i rozumně rychlé algoritmy. A jak že vůbec našel díru v systému Chulio? Podívejte se sami:

Systém převodů měn je vlastně orientovaný ohodnocený graf. Jednotlivé měny jsou vrcholy a pokud mezi dvěma měnami existuje převodní kurz c, tak mezi odpovídajícími vrcholy existuje orientovaná hrana s ohodnocením c. Posloupnost převodů, která je pro banku prodělečná, odpovídá v grafu orientovanému cyklu s hranami $e_1, e_2, \ldots, e_k$, ve kterém je součin ohodnocení těchto hran větší než jedna, tedy

$$c(e_1) \cdot c(e_2) \cdot \cdots \cdot c(e_k) > 1.$$

Zadaná úloha se dost podobá úloze hledání cyklu, který má součet ohodnocení hran záporný, pro kterou je známý rychlý algoritmus. Na ni naši úlohu převedeme tak, že výše uvedenou nerovnici zlogaritmujeme a vynásobíme -1:

$$-\log(c(e_1) \cdot c(e_2) \cdot \cdots \cdot c(e_k)) < -\log 1,$$

$$(-\log c(e_1)) + (-\log c(e_2)) + \cdots + (-\log c(e_k)) < 0.$$

Každou hranu e tedy místo $c(e)$, což je převodní kurz mezi odpovídajícími měnami, ohodnotíme $-\log c(e)$. Toto nové ohodnocení budeme pro zjednodušení následujícího textu nazývat délkou hrany a délka sledu pak bude součet délek jeho jednotlivých hran. (Sled je posloupnost vrcholů taková, že mezi sousedními vrcholy sledu vede v grafu hrana ve správném směru. Na rozdíl od cesty se v něm mohou opakovat jak vrcholy, tak hrany.)

V upraveném grafu zjišťujeme, jestli tam existuje cyklus (sled, který má první a poslední vrchol stejný) záporné délky. Jak na to? Použijeme Bellman-Fordův algoritmus, který hledá nejkratší sledy z nějakého vrcholu u do všech ostatních. Funguje tak, že si pro každý vrchol pamatuje délku zatím nejkratšího nalezeného sledu z u do něj. V každém kroku pro každou hranu vw určí, jestli není součet délky zatím nejkratšího nalezeného sledu z u do v a délky hrany vw menší než zatím nejkratší nalezený sled z u do w. Pokud ano, tak ji na tuto délku sníží. Na začátku nastavíme vzdálenost u na nulu a všech ostatních vrcholů na ∞.

Po provedení i-tého kroku budeme znát pro každý vrchol délku nejkratšího sledu obsahujícího nejvýše i hran z u do tohoto vrcholu. Pokud z vrcholu u není dosažitelný žádný cyklus záporné délky, tak nejpozději po $n-1$ krocích (n je počet vrcholů grafu) najde algoritmus pro každý vrchol nejkratší sled z u (každý z těchto sledů bude cesta) a pokud provedeme ještě n-tý krok, tak proběhne „naprázdno“, tj. nenajde žádný nový kratší sled. Pokud graf obsahuje cyklus záporné délky dosažitelný z u, tak v každém kroku (a tedy i v n-tém) najde nějaký nový kratší sled.

Zbývá ještě vymyslet, který vrchol zvolíme za počáteční vrchol u. Vzhledem k tomu, že nemusí existovat žádný vrchol, ze kterého by byly dosažitelné všechny ostatní, bude u nový vrchol, ze kterého povedou hrany do všech ostatních,

jejich délku zvolíme třeba nulovou. Protože do nového vrcholu nevedou žádné hrany, nepřidali jsme do grafu žádný cyklus a výsledek jsme tedy nezměnili. Kromě prvního kroku můžu hrany vedoucí z u a tedy i samotný vrchol u, ignorovat, protože určitě nezpůsobí změnu vzdálenosti žádného vrcholu. Naopak v prvním kroku stačí počítat jen s hranami vedoucími z u. Protože po prvním kroku budou mít všechny vrcholy vzdálenost nulovou, nemusím v programu ani vrchol u ani hrany z něj vedoucí nijak reprezentovat a stačí jen nastavit vzdálenost všech vrcholů na nulu.

Nakonec si uvědomíme, že převod kurzů na délky, který jsme provedli na začátku, je vlastně zbytečný a můžeme tedy počítat přímo s kurzy. Pro každý vrchol si budeme pamatovat prozatím nejlepší (maximální) kurzový součin na sledu z u a kurzový součin ve vrcholu w budeme upravovat, pokud je menší než součin kurzového součinu vrcholu v a kurzu hrany vw. Kurz na hranách z u můžeme nastavit třeba na jedničku (a ve zkrácené podobě prvního kroku tedy nastavit všem vrcholům kurzový součin na jedničku).

V prvním kroku jen nastavíme hodnotu kurzového součinu všem vrcholům. V každém dalším kroku projdeme všechny hrany a pro každou provedeme $O(1)$ operací. Celková časová složitost tedy je $O(n + nm) = O(nm)$, kde n je počet vrcholů a m je počet hran v grafu. V paměti máme informace o každé hraně a kurzový součin každého vrcholu, takže paměťová složitost je $O(n+m)$.

```
program banky;

const
  MaxN = 182;
  MaxM = 32942;

type
  tkurz = record
    odkud, kam : Integer;
    kurz : Real;
  end;

var
  meny : array [1..MaxN] of real;
  kurzy : array [1..MaxM] of tkurz;
  men, kurzu : integer;
  i, j : integer;
  novaHodnota : real;
  zmena : boolean;

begin
  read(men, kurzu);
  for i := 1 to kurzu do begin
    with kurzy[i] do
      read(odkud, kam, kurz);
  end;
```

```
  for i := 1 to men do
    meny[i] := 1;

  for i := 1 to men do begin
    zmena := false;
    for j := 1 to kurzu do begin
      novaHodnota := meny[kurzy[j].odkud] * kurzy[j].kurz;
      if (novaHodnota > meny[kurzy[j].kam]) then begin
        meny[kurzy[j].kam] := novaHodnota;
        zmena := true
      end
    end;
    if not zmena then break
  end;

  if zmena then
    writeln('Existuje pro banku prodělečná posloupnost.')
  else
    writeln('Banka nikdy neprodělá (pokud nepřijde krize).')
end.
```

21-5-3 Krávy — Mária Vámošová

Naším úkolem je přehradit všechny krávy v kravíně Z závorami, aby součet jejich délek byl minimální. Na tento problém se dá koukat dvěma způsoby.

První způsob: Na začátek si představme, že před každou krávou je jedna závora o délce jeden telemetr. Pokud je počet použitých závor rovný nebo menší zadanému číslu Z, jsme s problémem hotovi. Pokud jsme ale použili víc závor, než můžeme, musíme některé sousední závory „spojit“, čímž se nám sníží počet použitých závor o jedna. Protože součet délek závor má být minimální, spojujeme vždy co nejkratší úsek mezi kravami, až bude počet použitých závor rovný Z.

Proč tento způsob funguje? V první řadě si musíme uvědomit, že každá kráva musí být ohrazena. Takže pokud jsme měli dostatek závor, je naše začáteční rozmístění určitě minimální. Pokud ale dostatek závor nemáme, musíme nutně jednou závorou přehradit více krav. No a jedné závory se zbavíme tak, že spojíme dvě sousední závory, čímž ale přehradíme taky mezeru mezi nimi. Proto je nejvýhodnější v každém kroku spojit závory s co nejkratší mezerou mezi sebou (mezera mezi sousedícími závorami má délku 0).

Druhý způsob: Na problém se ale můžeme dívat i z jiného hlediska. Mějme na začátku jednu dlouhou závoru přes celý kravín. Na začátek ořežeme kus před první a po poslední krávě. Pak už jen zbývá najít $Z - 1$ nejdelších mezer mezi kravami a když je „vyřežeme“, zbyde nám Z závor, jejichž celková délka je určitě minimální.

Proč i tento způsob funguje? Začáteční ohrazení je určitě korektní. Taky ořezání kusu před první a za poslední krávou řešení nepokazí. Takže máme všechny krávy ohrazené, ale je možné, že nám zbyly nějaké závory. Z jedné závory udělám dvě tak, že z ní „vyřežu“ nějaký kus ze středu. A aby byl součet délek nových dvou závor minimální, musím vyřezat co největší kus. Tento kus ale nesmí obsahovat žádnou krávu, protože by už nebyla ohrazena. Takže se snažím najít co největší kus, který neobsahuje žádnou krávu, což je přesně to, že se snažím najít co největší mezeru mezi kravami, která ještě nebyla vyřezaná. Protože mám Z závor, tak si můžu dovolit vyřezat až Z-1 kusů, a tyto kusy musí být co největší.

Nechť vstup vypadá následovně: Na začátek jsou mezerami oddělená čísla N K Z, a pak následuje K čísel, které značí pozice krav v kravíně. Pro jednoduchost ať jsou pozice seřazeny vzestupně. Takže teď nám jenom zbývá najít $K - Z$ nejkratších, případně $Z - 1$ nejdelších mezer mezi kravami. Rozeberme třeba hledání $Z - 1$ největších. To můžeme udělat několika způsoby:

- Všechny mezery setřídit podle velikosti a vybrat $Z - 1$ největších. Tento způsob má při použití třídícího algoritmu QuickSort časovou složitost $O(K \log K)$ a paměťovou $O(K)$. Více informací o QuickSortu naleznete v Kuchařce 21-2 Rozděl a panuj.
- Postupně načítat pozice krav, vždy spočíst mezeru a za použití struktury halda vybrat $Z - 1$ největších. Halda je binární strom, kde pro každý prvek platí, že je menší než jeho následníci. Pro více podrobností viz Kuchařka 20-4 Halda, heapsort a Dijsktrův algoritmus.

  Stručně se algoritmus dá popsat následovně: Na začátek zatřídíme prvních $Z - 1$ mezer do haldy. Pak vždy, když načteme nový neobydlený úsek, porovnáme jeho délku s minimem na vrcholu haldy. Když je délka menší nebo rovna, úsek zahodíme, protože ostatní délky úseků v haldě jsou zjevně taky větší nebo rovny. Když je ale současný úsek delší než minimum v haldě, tohle minimum vyhodíme a zařadíme nový prvek do struktury. Na konci nám zůstane halda, ve které máme $Z - 1$ největších úseků mezi kravami. Tento algoritmus běží v časové složitosti $O(K \log Z)$ a paměťové $O(Z)$.
- Nejrychlejší způsob – použijeme algoritmus na hledání K-tého nejmenšího prvku z Kuchařky 21-2 Rozděl a panuj. V našem případě jenom otočíme nerovnosti a budeme hledat $(Z - 1)$-ní největší prvek. Tenhle algoritmus nám během počítání generuje rovnou všechny větší prvky, které sice nejsou vzájemně setříděné, ale to nám nijak nevadí. Tento algoritmus má časovou i paměťovou složitost $O(K)$.

21-5-4 Zákony — Roman Smrž

Ze spleti zákonů a soudních pří by se ti z vás, co se o to pokusili, nakonec vymotali, leč většinou by Chosému nechali dostatek času na složení šavle a museli by k ujasnění sporu o farmu (a Ochechulínu) využít jiných prostředků.

Při řešení úlohy si nejdříve můžeme všimnout, že ji lze přeformulovat ještě tak, že z lesa chceme dostat libovolně (nekonečně) malou kouli (tedy vlastně bod), ale mezi dvojice zákonů, které si jsou blíže než $2R$, postavíme zeď. Máme tedy zadány vrcholy grafu, jen potřebujeme zjistit, mezi kterými vedou hrany, a nakonec se podívat, zda je naše výchozí pozice uvnitř nějakého cyklu.

Pokud bychom se spokojili s kvadratickým časem, stačí pro nalezení hran jednoduše vyzkoušet všechny dvojice vrcholů. Nepříjemné je, že rychleji to ani nejde, neboť až kvadratický může být počet hran, které chceme najít; pročež nezbývá než si graf trochu upravit.

Pro jednoduchost budeme dále předpokládat, že počáteční pozice je v bodě (0,0) a zadaný poloměr roven 2 (jinými slovy, přepočítáme souřadnice tak, aby to platilo). Rozdělíme si celou rovinu na čtverce se stranou 2 tak, aby počátek ležel v rohu nějakého z nich. Body uvnitř každého z těchto bloků jsou pak vždy všechny spojené hranou (tvoří úplný podgraf) a můžeme je sloučit do jednoho se souřadnicemi středu čtverce. Musíme si jen dát pozor, když takto slučujeme hrany, které procházejí kolem počátku z různých stran (přesněji to můžou být ty, které protínají kladnou poloosu y, a ty ostatní); v takovém případě zřejmě z lesa zákonů uniknou nemůžeme. Nicméně to vyřešíme později a prozatím předpokládejme, že k tomu nedojde.

Nový graf má až N vrcholů (s celočíselnými lichými souřadnicemi; bloky ale podle nich indexovat nemůžeme, neboť kombinací souřadnic může být až kvadraticky, proto je podle souřadnic budeme mít jen lexikograficky uspořádané a v případě potřeby binárním půlením v logaritmickém čase najdeme blok pro dané souřadnice, nebo zjistíme, že tam žádný není; to bude potřeba jen konstantně-krát pro každý blok), ale každý vrchol může mít nejvýše 24 sousedů, takže počet hran je již nejvýše lineární, jen jejich nalezení bude o něco složitější. Hrana mezi dvěma bloky vede právě, když z nich lze vybrat dva vrcholy (každý z jednoho), mezi nimiž je vzdálenost menší než 4.

Máme tedy vždy dvě množiny bodů (A a B) oddělené nějakou přímkou (bez újmy na obecnosti to budiž osa y) a chceme najít dva body takové, že jejich vzdálenost je nejmenší možná. Všimněme si, že podíváme-li se na průsečík jejich spojnice a osy y, musí mu být oba body blíž než libovolný jiný z jejich skupiny. Pro každý bod na ose y si tedy určíme, který bod z A a který z B je mu nejblíže. To uděláme tak, že si body ve skupině setřídíme podle y-ové souřadnice a v tomto pořadí je budeme přidávat a hledané údaje postupně upravovat: na začátku máme jeden bod, který je nejblíže všem bodům osy y; kdykoli přidáme další, podíváme se, kterým bodům osy y bude blíže než bod předchozí

(třeba tak, že najdeme průsečík osy jejich spojnice s osou y), a pokud na ten už žádné nezbudou, odstraníme jej a porovnáme nový bod s jeho dalším předchůdcem atd., jinak pokračujeme přidáváním. Po setřídění nám na toto stačí lineární čas, neboť každý bod jednou přidáme a porovnáváme s předchůdcem bez odstraňování a nejvýše jednou odstraníme. Pak už zbývá jen projít osu y a pro každý bod porovnáme dvojici bodů, které jsou mu z každé ze skupin nejblíže (přesněji, projdeme ji po úsecích, kde se tato dvojice nemění). Pokud najdeme nějakou dvojici, která je blíže než $2R$, zapamatujeme si navíc, jestli protíná kladnou poloosu y.

Spojení bloků $(-1,-1)$ s $(1,1)$ a $(-1,1)$ s $(1,-1)$ – to jsou jediná, která mohou sdružovat hrany vedoucí z různých stran kolem počátku – vyřešíme speciálně: budeme chtít vědět nejen, jestli jsou spojeny, ale také z jaké strany (případně že z obou) to spojení vede. Opět máme dvě množiny A a B a body v nich setříděné podle nějaké ze souřadnic. Pokud je některá prázdná nebo obě jednoprvkové, je řešení triviální, jinak si vybereme nějaký bod m a rozdělíme obě množiny na body, které (když si je představíme jako vektory) mají směrnici menší než m (množiny A_1 a B_1) a ty zbylé (A_2 a B_2). Zřejmě dvojice A_1–B_2 může být spojená jen jedním typem hran, obdobně A_2–B_2; kterým a jestli spojené jsou, můžeme zjistit výše popsaným způsobem v lineárním čase. Nalezení spojení mezi A_1–B_1 a A_2–B_2 je pak původní problém na menších množinách. Aby se nám rekurze zastavila brzy, konkrétně po $O(\log N)$ iteracích, potřebujeme dané množiny rozdělovat rovnoměrně, nejlépe půlit, k tomu je potřeba, aby m byl mediánem některé z nich v uspořádání podle směrnic; ten můžeme nalézt v lineárním čase (jak na to se lze dočíst v letošní kuchařce ze druhé série), takže celý postup zabere čas $O(N \log N)$.

Nakonec už jen zbývá projít celý vytvořený graf a podívat se, zda je počátek uvnitř nějakého cyklu v něm. Zjistit, zda je nějaký bod vnitřním bodem polygonu, lze třeba tak, že pošleme „paprsek“ z tohoto bodu libovolným směrem a spočítáme, kolik hran tuto polopřímku protne – pokud jich bude lichý počet, je bod uvnitř, jinak vně. Tím paprskem bude kladná poloosa y, průsečíky s níž už máme předpočítané. Začneme v libovolném vrcholu (bloku), graf budeme procházet do hloubky a přitom si počítat, kolikrát jsme paprsek protli, jakmile narazíme na vrchol, který jsme už viděli (tedy cyklus), jednoduchým rozdílem těchto hodnot z této a předchozí návštěvy zjistíme, zda se počátek nachází uvnitř mnohoúhelníku z hran cyklu. Nemusíme si ani pamatovat počty průsečíků s paprskem, neboť nás zajímá jen parita.

Nalezení hran grafu bloků včetně potřebného třídění zvládneme v čase $O(N \log N)$ a na jeho prohledání stačí čas lineární a lineární je i velikost spotřebované paměti.

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

```
#include <stdbool.h>
#include <limits.h>
#include <math.h>
#define N_MAX 100000
int n;
struct bod {
   float s[2]; /* souřadnice bodu v rovině */
   int b[2];   /* souřadnice bloku, do nějž patří */
   int blok;
} body[N_MAX];

struct {
   int s[2];         /* souřadnice */
   int bodu;         /* počet bodů v bloku */
   unsigned char hrany[5][5];
      /* se kterými z možných 24 bloků sousedí */
   bool navstiveno;
      /* zda byl již navštíven při procházení */
   bool pruseciku;
      /* počet průsečíků s "paprskem" na cestě
       * od počátečního vrcholu (bloku) sem */
} bloky[N_MAX];

/* dvě porovnávací funkce pro qsort */
int porovnat_bloky(const void *va, const void *vb)
{
   const struct bod *a = va, *b = vb;
   return a->b[1] == b->b[1] ?
      a->b[0] - b->b[0] : a->b[1] - b->b[1];
}
int por_sour; /* porovnávaná souřadnice */
int porovnat_sour(const void *va, const void *vb)
{
   const struct bod *a = va, *b = vb;
   return (a->s[por_sour]>b->s[por_sour]) -
      (a->s[por_sour]<b->s[por_sour]);
}

/* Najdeme první bod v bloku o daných souřadnicích
 * pomocí binárního vyhledávání; index tohoto bodu
 * bude zároveň sloužit jako index celého bloku */
int najit_blok(int bx, int by)
{
   int min = 0, max = n-1;
   while (min != max) {
      int i = (min+max)/2;
      if (body[i].b[1] < by) min = i+1;
      else if (body[i].b[1] > by) max = i;
      else if (body[i].b[0] < bx) min = i+1;
      else max = i;
   }
```

```
   if (body[min].b[0] == bx && body[min].b[1] == by)
      return min;
   return -1;
}

/* Funkce dostane ukazatele na dvě skupiny bodů,
 * počet bodů v každé z nich, x-ovou souřadnici přímky
 * rovnoběžné s osou x, která skupiny rozděluje; vrátí
 * 1 pokud najde dvojici, která je od sebe vzdálená
 * méně než 4 a neprotíná kladnou poloosu y,
 * 2 pokud ji protíná, jinak 0. Pokud je s = 1, počítáme
 * se souřadnicemi prohozenými oproti popisu. */
unsigned char spojene(struct bod *ba, struct bod *bb,
      int na, int nb, int s, float x)
{
   static int pred[2][N_MAX], nasl[2][N_MAX];
            /* předek a následník daného bodu */
   static float zacatek[2][N_MAX];
         /* y-ová souřadnice, od níž je
          * bod rozdělující přímce nejblíž */
   struct bod *skup[2] = { ba, bb };
   int vel[2] = { na, nb };
   float a, b, c, y;
   for (int t = 0; t < 2; t++) {
      pred[t][0] = nasl[t][0] = n;
      zacatek[t][0] = -INFINITY;
      for (int i = 1; i < vel[t]; i++) {
         pred[t][i] = i-1; nasl[t][i] = n;
         for (int j = i-1; ; j = pred[t][j]) {
          /* najdeme osu spojnice danou rovnicí ax+by+c=0 */
            a = skup[t][i].s[s] - skup[t][j].s[s];
            b = skup[t][i].s[!s] - skup[t][j].s[!s];
            c = - (a*(skup[t][i].s[s]+skup[t][j].s[s])/2) -
               (b*(skup[t][i].s[!s]+skup[t][j].s[!s])/2);
            y = -(c+a*x)/b;
            /* průsečík osy a rozdělující přímky */
            if (y < zacatek[t][j]) {
             /* Jestliže předchozí bod není nikde nejbližší
              * rozdělující přímce, odstraníme jej, */
               pred[t][i] = pred[t][j];
               nasl[t][pred[t][i]] = i;
            } else {
               /* jinak si zapamatujeme,
                * odkud je nejbližší současný */
               zacatek[t][i] = y;
               break;
            }
         }
      }
   }
   for (int i = 0, j = 0; i < na && j < nb; ) {
```

```
      float dx = skup[0][i].s[s] - skup[1][j].s[s];
      float dy = skup[0][i].s[!s] - skup[1][j].s[!s];
      if (dx*dx + dy*dy < 16) {
         if (   /* Jsou li body na různých stranách osy y */
            ((skup[0][i].s[0]>0) != (skup[1][j].s[0]>0)) &&
            /* a platí (a_y-b_y) * a_x / (a_x-a_y) < a_y,
             * tedy směrnice vektoru a-b je menší než
             * směrnice vektoru a-0 pro kladná a_x
             * a větší pro záporná a_x, */
            ((skup[0][i].s[1]-skup[1][j].s[1])
             * (skup[0][i].s[0])
             / (skup[0][i].s[0]-skup[1][j].s[0])
             < skup[0][i].s[1])
            ) return 1;
            /* protíná jejich spojnice kladnou poloosu y. */
         return 2;
      }
      /* Posuneme se na další bod buď ve skupině A, nebo B,
       * podle toho, který se mění dřív */
      if (nasl[0][i] != n) {
         if (nasl[1][j] != n) {
            if (zacatek[nasl[0][i]] < zacatek[nasl[1][j]])
              i = nasl[0][i];
            else j = nasl[1][j];
         } else i = nasl[0][i];
      } else {
         if (nasl[1][j] != n)
            j = nasl[1][j];
         else return 0;
      }
   }
   return 0;
}

/* pomocné pole pro dočasné skupiny bodů */
struct bod buf[N_MAX*2];
int bi = 0;
/* Funkce hledá hrany mezi skupinami bodů, které mohou být
 * spojeny oběma typy hran. Vrátí 1 při nalezení pouze
 * kladnou poloosu y neprotínajících spojení, 2 najde-li
 * jen ty, které ji protínají, 3, pokud najde obě,
 * a jinak 0 */
unsigned char spojene_kolem_pocatku(struct bod *a,
                struct bod *b, int na, int nb)
{
   if (!nb || !na) return 0;
   if (na == 1 && nb == 1)
      return spojene (a, b, na, nb, 0, 0);
   /* Pro jednoduchost zde zvolíme medián směrnic
    * (tedy spíše pivot, protože to ve většině případů
    * medián nebude) náhodně, což dopadne dobře v průměrném
```

```
    * případě, jak najít medián spolehlivě v lineárním čase
    * se dočtete v kuchařce ze druhé série. */
   int median;
   float med_smer;
   if (na > nb) {
      median = rand()%na;
      med_smer = a[median].s[1]/a[median].s[0];
   } else {
      median = rand()%nb;
      med_smer = b[median].s[1]/b[median].s[0];
   }
   /* Rozdělení na části a rekurze */
   int ma = bi;
   for (int i = 0; i < na; i++)
      if (med_smer >= a[i].s[1]/a[i].s[0])
         buf[bi++] = a[i];
   int mb = bi;
   for (int i = 0; i < na; i++)
      if (med_smer < a[i].s[1]/a[i].s[0])
         buf[bi++] = a[i];
   int mc = bi;
   for (int i = 0; i < nb; i++)
      if (med_smer >= b[i].s[1]/b[i].s[0])
         buf[bi++] = b[i];
   int md = bi;
   for (int i = 0; i < nb; i++)
      if (med_smer < b[i].s[1]/b[i].s[0])
         buf[bi++] = b[i];
   unsigned char vysledek =
      spojene(buf+ma, buf+md, mb-ma, bi-md, 1, 0) |
      spojene(buf+mb, buf+mc, mc-mb, md-mc, 1, 0) |
      spojene_kolem_pocatku(buf+ma, buf+mc, mb-ma, md-mc) |
      spojene_kolem_pocatku(buf+mb, buf+md, mc-mb, bi-md) ;
   bi = ma;
   return vysledek;
}

/* Projde graf bloků a rozhodne, jestli je počátek uvnitř
 * nějakého polygonu tvořeného hranami cyklu */
bool projit_graf(int a, bool pruseciku) {
   if (bloky[a].navstiveno)
      return bloky[a].pruseciku != pruseciku;
   bloky[a].navstiveno = true;
   for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
         if (!bloky[a].hrany[i][j]) continue;
         if (bloky[a].hrany[i][j] == 3) return true;
         if (projit_graf(najit_blok(bloky[a].s[0]+2*j-4,
                                    bloky[a].s[1]+2*i-4),
                  pruseciku != (bloky[a].hrany[i][j]&1)))
            return true;
```

```
      }
   }
   return false;
}
int main(void)
{
   /* Načteme vstup a setřídíme body do skupin podle bloků,
    * do nichž patří; jednotlivé skupiny pak lexikograficky
    * podle souřadnic */
   float r, poc[2];
   scanf("%d%f%f%f", &n, &r, poc, poc+1);
   for (int i = 0; i < n; i++) {
      for (int s = 0; s < 2; s++) {
         scanf("%f", body[i].s+s);
         body[i].s[s] -= poc[s]; body[i].s[s] /= r/2;
         body[i].b[s] = (int)(body[i].s[s]/2)*2
                 + (body[i].s[s] < 0 ? -1 : 1);
      }
   }
   body[n].b[0] = body[n].b[1] = INT_MAX;
        /* zarážka, která zjednoduší podmínky dále */
   qsort(body, n, sizeof(*body), porovnat_bloky);
   /* Nejprve budeme hledat spojení s dvěma bloky napravo
    * postupně od každého bloku, takže body uvnitř bloku
    * potřebujeme mít seřazené podle y-ové souřadnice */
   por_sour = 1;
   for (int i = 0, j = 1; j <= n; j++) {
      body[j-1].blok = i;
      if (body[i].b[0] != body[j].b[0] ||
                     body[i].b[1] != body[j].b[1]) {
         bloky[i].s[0] = body[i].b[0];
         bloky[i].s[1] = body[i].b[1];
         bloky[i].bodu = j-i;
         qsort(body+i, j-i, sizeof(*body), porovnat_sour);
         for (int dx = -2; dx; dx++) {
            int ii = najit_blok(body[i].b[0]+2*dx,
                                        body[i].b[1]);
            if (ii < 0) continue;
            bloky[i].hrany[2][2+dx] =
                bloky[ii].hrany[2][2-dx] =
                     spojene(body+i, body+ii, bloky[i].bodu,
                        bloky[ii].bodu, 0, bloky[i].s[0]-1);
         }
         i = j;
      }
   }
 /* Teď budeme u každého bloku hledat spojení se zbývajícími
  * 10 bloky nad ním, k tomu setřídíme body podle x */
   por_sour = 0;
   for (int i = 0, j = 1; j <= n; j++) {
      if (body[i].b[0] != body[j].b[0] ||
```

```
                    body[i].b[1] != body[j].b[1]) {
            qsort(body+i, j-i, sizeof(*body), porovnat_sour);
            for (int dy = -2; dy; dy++) {
               for (int dx = -2; dx <= 2; dx++) {
                  int ii = najit_blok(body[i].b[0]+2*dx,
                                      body[i].b[1]+2*dy);
                  if (ii < 0) continue;
                  bloky[i].hrany[2+dy][2+dx] =
                          bloky[ii].hrany[2-dy][2-dx] =
                    /* Hledáme-li hranu vedoucí "přes" počátek, */
                        (bloky[i].s[1] == 1 && dy == -1) &&
                        ((bloky[i].s[0] == -1 && dx == 1) ||
                              (bloky[i].s[0] == 1 && dx == -1)) ?
                               /* zavoláme speciální funkci, */
                               spojene_kolem_pocatku(
                                              body+i, body+ii,
                                              bloky[i].bodu,
                                              bloky[ii].bodu) :
                               /* jinak standardní */
                               spojene(body+i, body+ii,
                                              bloky[i].bodu,
                                              bloky[ii].bodu, 1,
                                              bloky[i].s[1]-1);
               }
            }
            i = j;
         }
      }
      for (int i = 0; i < n; i++) {
         if (!bloky[i].navstiveno && projit_graf(i, 0)) {
            printf("Nelze se obhájit.\n");
            return 0;
         }
      }
      printf("Je možné se obhájit.\n");
      return 0;
}
```

21-5-5 Cestovní šavle — David Marek

Taková cestovní šavle je pěkně zapeklitá záležitost, při skládání si totiž nemůžeme být jisti, jestli zrovna tento dílek už máme složit, anebo ho ještě nechat narovnaný.

Mějme pouzdro délky L a do něj bychom chtěli složit šavli z N dílů. Řešení, které asi napadne každého, je vždy zkusit obě možnosti. Dílek nejdříve zkusíme složit a přesuneme se na zbylé dílky. Když se nám je žádným způsobem nepovede složit do pouzdra, tak se vrátíme, původní dílek zkusíme nechat

narovnaný a opět stejný postup použijeme na zbylé dílky. Pokud ani tato cesta nevede k cíli, pak šavli nejde složit. Algoritmus má ovšem exponenciální časovou složitost $O(2^N)$.

Naivní algoritmus je exponenciální, protože se v něm spousta kroků opakuje. To z této úlohy dělá problém typický pro řešení pomocí dynamického programování. Pusťme se tedy do něj. Téměř vše, co budeme potřebovat, je pole délky $L+1$. To bude po k-té fázi obsahovat značky právě tam, kde všude může prvních k dílků končit.

Fází výpočtu tak bude celkem N. V každé budeme zpracovávat jeden dílek. Projdeme celé pole a od místa, kde by mohl končit dílek předchozí (tzn. v poli na indexu i máme značku) zkusíme přidat dílek nový. Na počátku máme značku ve všech prvcích pole, protože umístění začátku prvního dílku není ničím omezeno. Konec nově přidaného dílku pak může být na indexech $i-d$ a $i+d$, kde d je jeho délka. Samozřejmě, že nový konec musí být v mezích pole. Povede-li se nám najít alespoň jeden možný konec pro všechny dílky, tak víme, že šavli složit jde.

Pokud bychom používali pouze jedno pole, budou se nám plést nově přidané značky se značkami z minulé fáze. Je tedy nutné mít pole dvě. Z jednoho budeme číst a do druhého si budeme připravovat značky pro další fázi. Po každé fázi pole navzájem vyměníme. Dobrý nápad je také mít pro každou fázi jinou značku (např. číslo fáze), díky tomu nebudeme muset před každou fází druhé pole nulovat.

Pro každý dílek projdeme celé pole, takže zjištění, jestli šavle složit jde, bude mít časovou složitost $O(N \cdot L)$ a paměťová složitost bude $O(L)$.

```
#include <stdio.h>

#define MAXL 10001

int pouzdro[2][MAXL];

int main() {
    int pocet_segmentu, velikost_pouzdra;
    int i, j, segment, zmena, aktualni, dalsi;
    scanf("%d %d", &pocet_segmentu, &velikost_pouzdra);

    /* Napoprvé vynulujeme pole */
    for (i = 0; i <= velikost_pouzdra; i++) {
        pouzdro[0][i] = 0;
    }
    aktualni = 0; dalsi = 1;

    for (i = 0; i < pocet_segmentu; i++) {
        /* Nemusíme načítat vstup hned na začátku */
        scanf("%d", &segment);
        /* Musíme si pamatovat,
         * jestli jsme další díl někam umístili */
```

```
        zmena = 0;

        for (j = 0; j <= velikost_pouzdra; j++) {
            /* Pokud jsme narazili na značku,
             * tak zkusíme přidat další díl */
            if (pouzdro[aktualni][j] == i) {
                /* Přidávaný díl nesmí vyčnívat
                 * z pouzdra na jedné straně */
                if (j - segment >= 0) {
                    /* Do pouzdra pro příští tah
                     * přidáme značku konce */
                    pouzdro[dalsi][j - segment] = i+1;
                    zmena = 1;
                }
                /* Přidávaný díl nesmí vyčnívat
                 * ani na druhé straně */
                if (j + segment <= velikost_pouzdra) {
                    pouzdro[dalsi][j + segment] = i+1;
                    zmena = 1;
                }
            }
        }
        /* Prohodíme pole pouzdra */
        j = aktualni; aktualni = dalsi; dalsi = j;

        /* Pokud segment nikam nepřidáme,
         * tak se šavle nedá složit */
        if (zmena == 0) {
            printf("Tuto šavli do pouzdra nesložíte.\n");
            return 0;
        }
    }

    printf("Tuto šavli dokážeme do pouzdra složit.\n");
    return 0;
}
```

21-5-6 Kržnc — Martin Mareš

Způsobů, jak tuto úlohu vyřešit v dostatečně těsném prostoru, je vícero. Například bychom mohli použít obyčejné prohledávání do hloubky a jeho zásobník šikovně zkomprimovat – tak se můžeme snadno dostat až na dva bity na vrchol. My si ale předvedeme trochu jiné, technicky daleko jednodušší řešení.

Definujeme *navigační posloupnost,* což bude posloupnost n čísel $a_1, \ldots, a_n$ (kde n je počet vrcholů grafu) v rozsahu 0 až 3. Každé navigační posloupnosti odpovídá nějaká „procházka po grafu“ (v grafové terminologii sled délky n): začneme v nultém vrcholu grafu, z něj se vydáme a_1-tou z jeho čtyř hran, z dalšího vrcholu pak a_2-tou hranou a tak dále.

Všimněme si, že pokud v grafu existuje nějaká kružnice, na které leží všechny vrcholy, pak je určitě popsána alespoň jednou navigační posloupností. Budeme tedy postupně generovat všechny navigační posloupnosti a pro každou z nich ověřovat, jestli popisuje hledanou kružnici.

Posloupnosti budeme přímočaře kódovat čísly. Na každé a_i nám stačí dva bity, takže do jednoho 32-bitového čísla schováme hned 16 prvků posloupnosti. Libovolné a_i pak dokážeme pomocí konstantního počtu aritmetických a bitových operací z tohoto kódu přečíst. Navíc se na celý kód můžeme dívat jako na jedno dlouhé $2n$-bitové číslo a jeho postupným zvyšováním o jedničku snadno vygenerovat všechny posloupnosti.

Když už umíme z posloupnosti číst, můžeme také snadno simulovat procházení po grafu a odpovídat na otázky typu „jaký je i-tý vrchol, který potkáme?“ nebo „potkáme cestou vrchol v?“. Obojí sice stojí čas $O(n)$, ale o ten nám vůbec nejde. Hlavní je, že si vystačíme s konstantním množstvím pracovní paměti.

Teď už stačí umět poznat posloupnost, která popisuje hledanou kružnici. To je také snadné: pro každý vrchol grafu vyzkoušíme, zda jím posloupnost projde, a pak ještě zkontrolujeme, že se na konci vrátíme zpět do vrcholu 0.

Toto řešení spotřebuje $\lfloor n/16 \rfloor + O(1)$ buněk paměti a má časovou složitost $O(4^n \cdot n^2)$. Existuje totiž 4^n kódů posloupností, pro každý z nich $O(n)$-krát hledáme vrchol, což pokaždé stojí $O(n)$.

Grafy stupně 3: Pokud z každého vrcholu vedou pouze 3 hrany, není potřeba na prvek posloupnosti obětovat celé 2 bity. Nabízí se použít trojkovou soustavu, ale pak bychom neuměli bez dodatečné paměti z posloupnosti číst. Použijeme místo toho „křížence“ mezi dvojkovou a trojkovou soustavou: nadále budeme kód dělit na 32-bitová slova a do každého uložíme trojkově co nejvíce prvků. Vejde se jich až $\lfloor \log_3 2^{32} \rfloor$, čili 20. Spotřeba paměti tedy klesne na $\lfloor n/20 \rfloor + O(1)$.

Ještě šetrnější způsob: Bystří řešitelé si všimli, že i tento docela hustý popis procházek pomocí posloupností je stále dost marnotratný. Když přijdeme do vrcholu se čtyřmi hranami, nechceme se přeci vracet po hraně, po níž jsme právě přišli, takže máme na výběr pouze ze tří možností. Předchozí trik s trojkovou soustavou tedy můžeme použít i pro původní verzi úlohy. V grafech stupně 3 si dokonce vystačíme se dvěma možnostmi, což nám dává jediný bit na stav, a tedy $\lfloor n/32 \rfloor + O(1)$ buněk paměti. Strýček Skrblík by měl radost a pan Cowmess jistě také.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Vojtěch Kolář	G Neratov	3	29	200,0
2.	Vítězslav Plachý	GJiříPoděb	3	28	188,6
3.	Filip Hlásek	GMikulášPL	2	19	182,0
4.	Petr Čermák	GEBenešeKL	3	27	177,3
5.	Michal Bilanský	GLepařovJČ	3	22	157,3
6.	Pavel Veselý	G Strakon	4	21	144,9
7.	Jitka Novotná	G Bílovec	4	17	129,5
8.	Karel Tesař	SPŠE Plzeň	3	19	114,9
9.	Vlastimil Dort	GŠpitálsPH	3	15	113,9
10.	Jiří Cidlina	GVoděraPH	4	18	106,7
11.	Lukáš Ptáček	GJAKŽeliez	3	13	92,5
12.	Alexander Mansurov	GNVPláníPH	0	11	87,3
13.	Martin Zikmund	G Turnov	1	16	79,6
14.	Filip Štědronský	GMikulášPL	2	8	78,6
15.	Pavel Taufer	ArcibisGPH	3	10	76,9
16.	Jiří Setnička	G25březnPH	2	11	75,1
17.	David Věčorek	GTNovákBO	3	11	72,6
18.	Štěpán Šimsa	GJungmanLT	0	10	68,2
19.	Karel Král	G Most	3	9	56,8
20.	Jan Vaňhara	G Holešov	4	7	56,2
21.	Petr Pecha	SPŠSVsetín	2	10	54,5
22.	Alžběta Pechová	SPŠSVsetín	4	11	48,1
23.	Barbora Janů	GKepleraPH	2	10	44,3
24.	Jan Veselý	G Strakon	2	5	41,3
25.	Kateřina Lorenzová	G Česká ČB	2	6	38,6
26.	Libor Plucnar	GBezručeFM	4	5	38,2
27.	Filip Sládek	GNámestovo	3	5	38,1
28.	Radim Cajzl	GNoMěsNMor	2	9	37,7
29.	Karel Kolář	GŠpitálsPH	4	6	36,3
30.	Tomáš Pikálek	GBoskovice	2	4	35,6
31.	Lukáš Chmela	GJŠkodyPŘ	0	6	35,2
32.	Ondřej Pelech	GJNerudyPH	4	4	33,4
33.	Petr Zvoníček	G Slavičín	3	6	32,4
34.	David Formánek	GJarošeBO	2	4	27,9
35. – 36.	Jan Kostecký	VOŠŠumperk	2	5	27,7
	Karolína Burešová	G ČesLípa	2	4	27,7
37.	Milan Rybář	GJungmanLT	4	4	27,4

Pořadí řešitelů

38.	Pavol Rohár	G Košice	3	4	27,2
39.	Honza Žerdík	G Příbor	4	3	25,7
40.	Jan Škoda	GMikulášPL	2	3	23,3
41.	Alena Bušáková	G Trutnov	2	3	22,6
42.	Hynek Jemelík	GJarošeBO	2	3	19,0
43.	Jakub Sochor	G Bílovec	4	2	17,1
44. – 45.	David Vondrák	GDašickáPA	3	3	17,0
	Martina Vaváčková	GCoubTábor	3	3	17,0
46.	Pavel Kratochvíl	VOŠGSvetla	1	4	15,9
47.	Martin Holec	G Slavičín	2	3	15,5
48.	Mirek Jarolím	GMikulášPL	3	3	15,3
49.	Petr Babička	VOŠGSvetla	4	3	11,6
50. – 52.	Jan Matějka	G Jírov ČB	4	1	10,0
	Jiří Daněk	GKřenováBO	3	1	10,0
	Martin Holeček	GMikul23PL	3	1	10,0
53.	Dominik Smrž	GOhradníPH	0	1	8,7
54.	Anna Chejnovská	GBNěmcovHK	2	1	8,5
55.	Jiří Keresteš	SPŠE Plzeň	3	1	7,4
56.	Jakub Zíka	GNadAlejPH	2	1	6,0
57.	Stanislav Fořt	GCoubertTÁ	1	2	5,9
58.	Igor Koníček	G UherBrod	3	2	5,7
59.	Ladislav Maxa	GKepleraPH	3	1	5,5

Obsah

Petr Kratochvíl a kolektiv

Korespondenční seminář z programování
XXI. ročník

Autoři a opravující úloh:
Martin Böhm, Pavel Čížek, Zbyněk Falt, Pavel Klavík,
Petr Kratochvíl, Martin Kruliš, Pavel Machek, David Marek
Martin Mareš, Petr Onderka, Roman Smrž, Kristýna Stodolová
Milan Straka, Josef Špak, Mária Vámošová, Michal Vaner

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 293. publikaci.

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.

S jejich pomocí ročenku vysázel Jan Matějka.

Ilustrace (včetně té na obálce) vytvořil Martin Kruliš.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 158 stran
Náklad 300 výtisků
Praha 2009

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-099-9

ISBN 978-80-7378-099-9