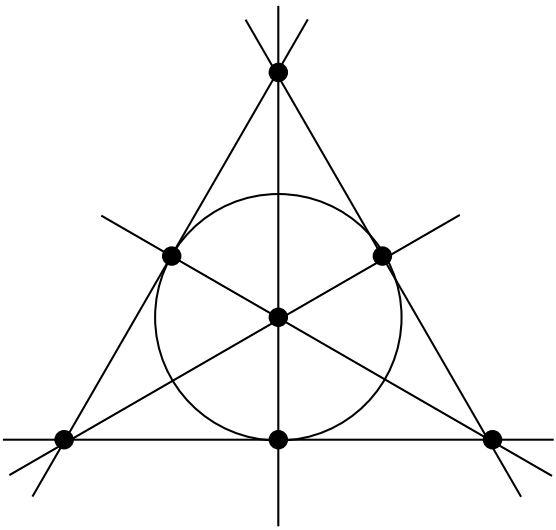
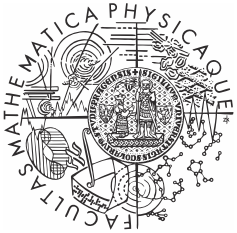
JAN KÁRA A KOLEKTIV

Korespondenční seminář z programování

XII. ročník – 1999/2000





Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

JAN KÁRA A KOLEKTIV

Korespondenční seminář z programování

XII. ročník – 1999/2000

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvanáctý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešíce úlohy našeho semináře, středoškoláci získávají praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé discipliny informatiky. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy, MFF z toho nevyjímaje. 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 několika (čtyř či pěti) úloh, v klidu domácího krbu je (ne nutně všechny) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na níže uvedenou adresu. My je poté (víceméně obratem) 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 proběhnou čtyři série, v letech hojnějších pak pět. 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 jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

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

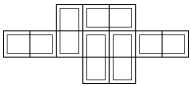
e-mail: *ksp@mff.cuni.cz*
www: *http://atrey.karlin.mff.cuni.cz/ksp/*

Zadání úloh

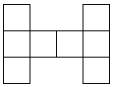
12-1-1 Dlaždiči — 10 bodů

Firma Kostka a syn se zabývá pokládáním dlažeb všeho druhu. Momentálně má firma bohaté zásoby velmi kvalitních dlaždiček velikosti 1×2 dm. Dlaždičky jsou natolik kvalitní, že dokonce ani nejdou dělit (za což si prodejce samozřejmě naúčtoval řádnou přirážku), a proto je třeba dlaždičky naskládat tak, aby nebylo potřeba je dělit. To je ale samozřejmě pro řadového dlaždiče poněkud náročný úkol, a tak si firma najala vás, abyste napsali program, který danou úlohu vyřeší (potřebuje totiž zjistit, jestli je danou zakázku vůbec možno splnit). Máte tedy zadánu plochu ke dláždění (zadanou jako matici $m \times n$, kde `1` označuje plochu ke dláždění a `0` trávník okolo) a máte odpovědět, zda je plochu možno vydláždit.

Příklad:



lze vydláždit



nelze vydláždit

12-1-2 Parník — 10 bodů

Paroplavební společnost Tit & Nick se rozhodla obohatit svůj strojový park o další parník. Chce ale, aby tento byl schopen doplout z libovolného místa do libovolného. K tomu ale potřebuje vědět, která dvě místa jsou v říční síti nejvzdálenější. A to už je úkol pro vás. Říční síť je zadána pomocí N význačných bodů na řekách a splavných úseků mezi nimi. Význačnými body jsou prameny, soutoky či ústí do moře. Splavný úsek je zadán jako dvojice význačných bodů, které spojuje. U každé dvojice je též zadáno, jak je charakterizovaný úsek řeky dlouhý. Dále víte, že říční síť tvoří strom, tedy že mezi každými dvěma význačnými body existuje právě jedna cesta. Výsledkem vašeho snažení by pak měla být dvojice nejvzdálenějších bodů v říční síti.

Příklad: Síť má 5 význačných bodů. Úseky řek jsou následující: $(1, 2)$ o délce 3, $(2, 3)$ o délce 4, $(3, 4)$ o délce 2 a $(2, 5)$ o délce 5. Nejvzdálenější body jsou 1 a 4.

12-1-3 A sort of sort — 10 bodů

Na vstupu máte N kladných celých čísel. Vaším úkolem není nic těžšího, než je co nejrychleji vzestupně uspořádat. Navíc o každém z nich víte, že je menší nebo rovno N^3.

12-1-4 PRAM — 10 bodů

S rozvojem techniky se neustále rozšiřuje možnost provádět paralelní výpočty. Z teoretického hlediska je bezpochyby rozumné zavést standardizovaný model paralelního počítače – nejrozšířejnějším takovým modelem je `PRAM`, kterému se budeme v letošním seriálu věnovat. Nutno však poznamenat, že tento model má význam spíše v teoretické informatice – současné paralelní počítače a výpočty na nich prováděné jsou tomuto modelu velmi vzdálené.

Parallel Random Access Machine (`PRAM`) je počítač tvořený k identickými procesory. Tyto procesory sdílejí společnou (*globální*) paměť. Ta obsahuje paměťové buňky číslované od nuly, jejichž počet není omezen. Do každé buňky je možno uložit libovolné celé číslo. Na začátku výpočtu je v několika (obvykle prvních) z nich uloženo zadání úlohy, obsah ostatních buněk není definován. Na konci výpočtu je pak v několika (opět obvykle prvních) buňkách uložen výsledek výpočtu. Každý z procesorů dále má své vlastní *lokální* paměťové buňky pro celočíselné hodnoty, které jsou stejně jako buňky globální paměti indexovány čísly od nuly.

Program, podle kterého probíhá výpočet, je společný pro všechny procesory a v každém kroku provede každý procesor právě jednu instrukci. Výpočet na všech procesorech začíná ve stejném okamžiku. Ke společné paměti přistupují procesory přes pseudopole `gmem[i]`, kde i je celé nezáporné číslo, do lokální paměti přes pseudopole `lmem[i]` a dále si může každý z k procesorů přečíst své číslo (od 1 do k) z pseudokonstanty `cpuid`. Kromě toho může každý z procesorů použít (nejvýše jednonásobnou) nepřímou adresaci globální paměti, tj. lze jako index do pseudopole představujícího globální pamět použít některý z výše uvedených výrazů. Tedy jsou správné např. výrazy typu `gmem[gmem[i]]`, `gmem[lmem[i]]`, `gmem[cpuid]`, ale nikoliv `gmem[gmem[cpuid]]`. Rovněž tak nelze použít jako adresu ve sdílené paměti jakýkoliv aritmetický výraz (kupř. `mem[cpuid*3]`). Nechť `X`, `Y` a `Z` představují některý z výše uvedených výrazů pro přístup do paměti a k registrům, `X` není `cpuid`, `Y` a `Z` mohou být i celočíselné konstanty, potom instrukce `PRAM`u jsou tvaru:

- Přiřazení: `X:=Y`
- Unární minus: `X:=-Y`
- Součet: `X:=Y+Z`
- Rozdíl: `X:=Y-Z`
- Součin: `X:=Y*Z`
- Celočíselné dělení: `X:=Y/Z`
- Zbytek po celočíselném dělení: `X:=Y%Z`
- Bitová konjunkce: `X:=Y&Z`
- Bitová disjunkce: `X:=Y|Z`
- Bitová nonekvivalence: `X:=Y^Z`

- Bitový posun doleva: `X:=Y<<Z`
- Bitový posun doprava: `X:=Y>>Z`
- Prázdná instrukce: `nop`
- Ukončení výpočtu: `halt`
- Nepodmíněný skok: `goto LABEL`
- Podmíněný příkaz: `if LOGEXPR then INSTR`

Bitové operace lze provádět pouze s nezápornými operandy, dělení pouze s nenulovým dělitelem. `LABEL` je návěští (jako v Pascalu, pouze jej není nutno deklarovat), definuje se napsáním `LABEL:` před instrukci. Doba provádění podmíněného příkazu nezávisí na splnění jeho podmínky a je stejná jako doba provádění libovolné jiné instrukce. `INSTR` je libovolná instrukce mimo podmíněného příkazu a `LOGEXPR` je jeden z následujících logických výrazů:

- Test rovnosti: `Y=Z`
- Negace testu rovnosti: `Y<>Z`
- Test ostré nerovnosti: `Y<Z`, případně `Y>Z`
- Test neostré nerovnosti: `Y<=Z`, případně `Y>=Z`

Komentáře lze psát do programu na libovolný řádek za instrukci; před začátek komentáře se píše středník.

Dosud jsme však opominuli jednu důležitou otázku: Co se stane, když se více procesorů v jednom kroku pokusí zapsat několik různých hodnot do stejné paměťové buňky? Mohou různé procesory v jednom kroku číst ze stejné paměťové buňky? Zavedeme si několik různých typů `PRAM`ů podle odpovědí na tyto otázky:

- Priority Concurrent-Read Concurrent-Write (zkráceně `P-CRCW`, jinak též *prioritní*) `PRAM` – hodnotu ze stejné paměťové buňky může číst a zapisovat do ní libovolný počet procesorů; do buňky bude uložena ta hodnota, kterou do ní chtěl zapsat procesor s nejnižším číslem.
- Concurrent-Read Exclusive-Write (zkráceně `CREW`, jinak také *částečně konfliktní*) `PRAM` – hodnotu ze stejné paměťové buňky může číst libovolný počet procesorů, ale nejvýše jeden do ní může hodnotu v jednom kroku zapisovat. V případě porušení tohoto pravidla, dojde k obdobné chybě jako např. při dělení nulou.
- Exclusive-Read Exclusive-Write (`EREW`, *bezkonfliktní*) `PRAM` – z libovolné paměťové buňky může číst v jednom kroku nejvýše jeden procesor a nejvýše jeden procesor se může pokusit do paměťové buňky nějakou hodnotu zapsat. V případě, že hodnota se z buňky čte a je do ní zapisována v jednom kroku, musí toto čtení a zápis provádět ten samý procesor.

Nyní si ukážeme příklady programů pro různé typy PRAMů. Jako vstup dostanou posloupnost n čísel nula a jedna uloženou v paměťových buňkách 1 až n; číslo n je uloženo v paměťové buňce číslo 0. Úkolem programu je zapsat do nulté paměťové buňky pořadí první jedničky v této posloupnosti. Pokud posloupnost obsahuje pouze nuly, bude tato buňka obsahovat hodnotu nula. Na P-CRCW PRAMu lze úlohu vyřešit pomocí n procesorů následujícím jednoduchým programem:

```
gmem[0]=0
if gmem[cpuid]=1 then gmem[0]=cpuid
halt
```

Tento program nelze na CREW PRAMu použít, neboť např. při druhé instrukci zapisují všechny procesory příslušející jedničkám naráz do nulté paměťové buňky. To obejdeme následujícím trikem (opět používáme n procesorů): Představme si, že jsme si očíslovali prvky posloupnosti ne od jedné, ale od nuly. V k-té etapě bude l-tý prvek (pro $2^k \mid l$) posloupnosti obsahovat nejmenší číslo jedničky v úseku délky 2^k, který jím počíná. Nyní se procesory příslušející prvkům, jejichž číslo l je dělitelné 2^{k+1}, a takovým, že v úseku délky 2^k počínajícím l-tým prvkem jednička není, podívají na hodnotu prvku $l+2^k$ a v případě její nenulovosti ji prohlásí za svou konečnou hodnotu. V samotném programu je ještě třeba dbát na ošetření „plusmínusjedničkových“ chyb, které by mohly snadno vzniknout.

Program pro CREW PRAM:

```
   lmem[0]:=gmem[0]     ; tady budeme mít délku posloupnosti
   lmem[1]:=cpuid       ; tady budeme mít další testovanou buňku
   lmem[2]:=cpuid-1     ; své číslo minus jedna
   lmem[3]:=1           ; posun k další buňce
   if gmem[cpuid]=1 then gmem[cpuid]:=cpuid
   if gmem[cpuid]<>0 then goto 2
1: lmem[4]:=lmem[2]&lmem[3]        ; Budeme končit?
   if lmem[4]<>0 then goto 2       ; Končíme - dopočítali jsme
   lmem[1]:=cpuid+lmem[3]
   lmem[3]:=lmem[3]<<1
   if lmem[1]>lmem[0] then goto 2  ; Jsme za koncem posloupnosti
   if gmem[lmem[1]]<>0 then gmem[cpuid]:=gmem[lmem[1]]
   if gmem[cpuid]=0 then goto 1    ; Ještě jsme nenašli nenulu
2: if cpuid=1 then gmem[0]=gmem[1] ; Zapsat výsledek.
   halt
```

Modifikaci programu pro EREW PRAM si jistě již každý dokáže vytvořit sám.

Při řešení zadaných úloh lze předpokládat, že je spuštěno (ve stejný okamžik) právě tolik procesorů, kolik požadujete. Navíc se při řešení úlohy snažte dodržovat následující pravidla (v uvedeném pořadí!):

1) Počet použitých procesorů závisí na velikosti úlohy nejvýše polynomiálně.
2) Celková doba výpočtu, měřená jako doba výpočtu procesoru, který skončil jako poslední, je co nejkratší.
3) Velikost prostoru použitého při výpočtu, měřená jako součet počtu všech použitých buněk (v globální i lokálních pamětech), je co nejmenší.
4) Počet použitých procesorů je co nejmenší (to je částečně zahrnuto v předchozí podmínce).

A nyní již lze zformulovat zadání první úlohy v tomto roce: Vytvořte program, který jako vstup obdrží dvě nula – jedničkové posloupnosti délky n, které představují binární čísla (nejnižší bit je poslední). Délka posloupností je uložena v `gmem[0]`, první posloupnost se nachází v `gmem[1]` až `gmem[n]`, druhá pak v `gmem[n+1]` až `gmem[2n]`. V případě, že první posloupnost představuje menší číslo než ta druhá, měla by být na konci výpočtu v `gmem[0]` uložena jednička, pokud je větší, tak dvojka, a pokud jsou si rovny, tak nula. Úlohu postupně vyřešte na `P-CRCW PRAM`u, `CREW PRAM`u a `EREW PRAM`u.

12-2-1 Osmisměrka — 10 bodů

Jistě všichni znáte křížovkářskou hříčku zvanou osmisměrka. My si ji pro účely naší úlohy lehce upravíme: Je dána sada slov a obdélník tvaru $M \times N$ složený z písmenek. Řešení osmisměrky probíhá tak, že postupně bereme jednotlivá slova ze slovníku, pro každé slovo najdeme všechny jeho výskyty v osmisměrce ve všech směrech (slovo se v osmisměrce vůbec vyskytovat nemusí) a použitá písmenka vyškrtneme (vyškrtnuté písmeno může být ale klidně použito znovu). Slovo $S = s_1 \ldots s_l$ se vyskytuje na pozici a, b, pokud $o[a, b] = s_1, o[a + dx, b + dy] = s_2, \ldots, o[a + l \cdot dx, a + l \cdot dy] = s_l$, kde dx a dy určují jeden z osmi možných směrů – formálně tedy $dx, dy \in \{-1, 0, 1\}$ a buď $dx \neq 0$ nebo $dy \neq 0$. Když takto probereme všechna slova, nevyškrtaná písmena čtená v přirozeném (pro průměrného středoevropana) pořadí nám dají výsledný text.

Vaším úkolem není nic snazšího, než napsat program řešící osmisměrku. Váš program si nejdříve načte slovník a ten si libovolně předzpracuje (můžete předpokládat, že slovník se vám vejde do paměti). Potom na vstup dostane jednotlivé osmisměrky. Na ně by měl v co nejkratším čase odpovídat výslednými texty. Důraz je kladen především na rychlost vyřešení jedné osmisměrky. Délka předzpracování je méně významná.

Příklad: Ve slovníku se nachází 5 slov: `maso`, `eso`, `mele`, `sokl`, `lyko`. Osmisměrka tvaru 4×4 vypadá následovně:

```
meso
esok
lyko
elly
```

Zbylý text je `kly`.

12-2-2 Šnečí maraton — 9 bodů

Šneci se jednoho slunného léta rozhodli uspořádat maraton – tedy plaz na 42 metrů. Na start se dostavilo N závodníků. Podle pořadí, v jakém vyráželi na trať, jim byla přiřazena startovní čísla od jedné do N. I přes nepříjemnosti, jako třeba houba znenadání vyrostlá na trati nebo ztráta domečku při prudkém zatáčení, se nakonec podařilo doplazit se do cíle všem hlemýžďům. Problém ovšem je, že každý z hlemýžďů si pamatuje pouze to, kolik soupeřů s nižším startovním číslem předběhl. Napište program, který z těchto informací rekonstruuje pořadí, v němž šneci dobíhali do cíle.

12-2-3 Jednosměrky — 10 bodů

Bylo jednou jedno království. A v onom království se začal rozmáhat motorismus. Silnice byly přecpány kočáry, trakaři i vozíky, křižovatky byly neprůjezdné. A tak se král rozhodl vylepšit nevábnou situaci v dopravě a poněkud zvýšit plynulost provozu. Rádce, jemuž byla realizace této velkolepé myšlenky svěřena, dostal nápad, že provoz by šlo bez větších nákladů zlepšit tak, že by se ze všech silnic udělaly jednosměrky. I najal si vás, abyste mu pro danou silniční síť zjistili, zda je možno ze silnic udělat jednosměrky tak, že se stále ještě půjde dostat z libovolného místa do libovolného (bez porušení předpisů samozřejmě). A pokud to možné je máte i vypsat, jak to učinit.

Váš program dostane na vstupu počet křižovatek N a počet silnic mezi křižovatkami M. Dále pak pro každou silnici dostane čísla dvou křižovatek, které spojuje (křižovatky jsou očíslovány od jedné do N). Na výstup má pak váš program buď vypsat, že ze silnic jednosměrky udělat nejde, nebo vypsat silniční síť s jednosměrkami – tedy seznam spojení křižovatek ve tvaru *počáteční křižovatka* $\longrightarrow$ *koncová křižovatka*.

Příklad 1: Máme 4 města a 4 silnice – $\{1,2\}$, $\{3,2\}$, $\{3,4\}$, $\{1,4\}$. Silniční síť s jednosměrkami může vypadat třeba následovně: $(2,3), (3,4), (4,1), (1,2)$.

Příklad 2: Máme 4 města a 4 silnice – $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1,4\}$. Silniční síť s jednosměrkami neexistuje.

12-2-4 Vysílače — 11 bodů

Společnost Shumm & Brumm se zabývá výstavbou sítí vysílačů. Protože signály z jednotlivých vysílačů se ruší, musí společnost zajistit, aby vysílače měly dostatečně malý výkon a k rušení tedy nedocházelo. Vývojové oddělení firmy po nákladném výzkumu zjistilo, že nejvíce se ruší nejbližší vysílače. Z toho poté již v relativně krátké době odvodilo, že pokud se nebudou rušit dva nejbližší vysílače, nebudou se rušit ani žádné dva jiné vysílače. Způsob, jak nalézt dva nejbližší vysílače, už ovšem oddělení nevyřešilo. A tak jste byli najati, abyste tento problém vyřešili vy.

Na vstupu tedy máte dány souřadnice N vysílačů a vaším úkolem je určit, které dva vysílače jsou nejblíže.

Příklad: Máme pět vysílačů o souřadnicích $(0,0)$, $(1,0)$, $(-0.5,0)$, $(0,2)$, $(1,1)$. Nejblíže jsou si vysílače jedna a tři.

12-2-5 PRAM — 11 bodů

V minulém kole jste vymysleli, jak na `PRAM`u porovnávat dvě celá čísla zadaná po bitech. Nyní se naučíme sčítat:

Vytvořte program, který jako vstup obdrží dvě posloupnosti délky n složené z nul a jedniček, které představují binární čísla (nejnižší bit je poslední). Délka posloupností je uložena v `gmem[0]`, první posloupnost pak v `gmem[1]` až `gmem[n]`, druhá v `gmem[n+1]` až `gmem[2n]`. Zápis čísel nemusí začínat jedničkou, např. první číslo může být 1100 a druhé 0110. Obsah paměťových buněk před začátkem výpočtu by pro tato dvě čísla tedy byl: $4,1,1,0,0,0,1,1,0$.

Program má tato dvě čísla sečíst a výsledek uložit do paměťových buněk `gmem[0]` až `gmem[n]` tak, aby tyto obsahovaly jednotlivé cifry výsledného součtu ve formátu obdobném formátu vstupu programu; např. paměťová buňka `gmem[0]` obsahuje jedničku, pokud výsledný součet má $n+1$ cifer, jinak obsahuje 0. Obsah paměťových buněk po ukončení výpočtu by pro naše zadání měl tedy být: $1,0,0,1,0$.

Úlohu postupně vyřešte pro `EREW PRAM`, `CREW PRAM` a `P-CRCW PRAM`.

12-3-1 Strýček Skrblík — 9 bodů

Pan Skrblík se jednoho dne začal zajímat, proč má stále tak zoufale málo peněz. Vzal si tedy své účetnictví a jal se zkoumat, ve kterém období prodělal nejvíce. Po chvilce to ale vzdal, protože záznamů bylo příliš mnoho a rozhodl se, že obětuje tak dva tři centy za program.

Vaším úkolem je tedy program napsat. Na vstupu program dostane počet záznamů a potom jednotlivé záznamy. Každý záznam je vlastně jedno celé číslo. Kladné, pokud pan Skrblík vydělal, záporné, pokud prodělal. Program má zjistit souvislý úsek s minimálním součtem a vypsat jeho počátek a konec.

12-3-2 Vypočítavý Robin — 11 bodů

Určitě všichni znáte Robina Hooda. Byl známý tím, že uměl skvěle střílet z luku. Jednoho dne se zamyslel nad příslovím zabít pět much jednou ranou a přišel na to, že on by mohl dělat to samé – jedním šípem zabít co nejvíc nepřátel. A tak se rozhodl to vyzkoušet v praxi. Jednoho dne zastavil v lese skupinu šerifových nejlepších rytířů a hned se dal do experimentování. Protože však neměl k dispozici moderní techniku a rytíři nechápali jeho volání: „Postavte se do řady, posloužíte vědě!“ (neboť jim věda byla ukradená), skončil nevalně. Než určil, jak zamířit, rytíři ho zajali. Protože Vy nechcete skončit stejně a máte k dispozici moderní techniku, napište si program, který zjistí, máte-li zadáno souřadnicemi N bodů v rovině, maximální počet bodů ležících v nějaké přímce.

12-3-3 Palindromický rozklad — 10 bodů

Mějme řetězec P. Řekneme, že P je palindrom, pokud je stejný při čtení zleva i zprava (např. řetězce „a“, „oko“ nebo „abcddcba“ jsou palindromy). Na vstupu máte daný řetězec S. Zjistěte minimální počet palindromů, na které je možno řetězec rozložit, případně napište, že to nejde.

Příklad: Řetězec „abcbebe“ lze rozložit na 3 palindromy.

12-3-4 Skákací panák — 12 bodů

Anička si jednoho dne všimla, že na chodník před domem někdo nakreslil skákacího panáka. Není to ale jen tak ledajaký panák, je to stromový panák. Ten vypadá tak, že jsou určena místa, kam se může šlápnout a ta jsou propojena čarami. Skákat se může pouze po těchto čarách. Navíc platí, že mezi každými dvěma místy vede po čarách právě jedna cesta. Aničku zajímalo, jestli dokáže celého panáka obskákat tak, aby na každé místo šlápla právě jednou a přitom se vrátila zase na počátek. Protože je ale Anička ještě malá, nedokáže skočit přes více jak tři čáry.

Vaším úkolem je napsat program, který pro panáka daného počtem míst a spojnicemi mezi nimi, nalezne způsob, jak přeskákat po všech polích, vrátit se zpět a přitom nešlápnout na žádné pole dvakrát (vyjma prvního). Navíc se může skákat pouze přes tři čáry.

Příklad:
Počet polí: 6
Spojnice: (1, 2), (2, 3), (3, 4), (3, 5), (5, 6)
Návod pro Aničku: 1, 2, 4, 5, 6, 3, 1

12-3-5 PRAM — 10 bodů

Vytvořte program, který jako vstup obdrží posloupnost n celých čísel a určí její maximum. Délka posloupnosti bude při spuštění uložena v `gmem[0]`

a posloupnost sama pak v buňkách `gmem[1]` až `gmem[n]`. Obsah paměťových buněk před začátkem výpočtu by tedy například mohl být: $5, 4, 0, 4, -2, 3$. Program určí největší prvek z této posloupnosti a zapíše jej do buňky `gmem[0]`. V našem příkladě by do `gmem[0]` tedy zapsal 4.

Úlohu postupně vyřešte na `P-CRCW`, `CREW` a `EREW PRAMu`.

12-4-1 Magický zapletenec — 10 bodů

Jak víte, nebo možná nevíte, čarodějové si schovávají svou moc do různých talismanů, prstenů či náhrdelníků. Nejinak to dělal i náš čaroděj Kern. Jedno z jeho kouzel se mu ale poněkud vymklo z ruky a mimo jiné způsobilo, že se v okolí objevil kvákající kocour, mňoukající hrnec a ušatá žába. Nejvíce ale čarodějovi vadilo, že se mu úplně zašmodrchal jeho kouzelný řetízek a stal se tak naprosto nepoužitelným. Protože Kern nehodlal riskovat opravu náhrdelníku magickými cestami, jal se rozmotávat náhrdelník ručně. Brzy ale zjistil, že kouzlo mu pospojovalo některé kroužky, které spojeny nebyly a naopak rozpojilo kroužky, které spojeny byly. Nezbývá tedy, než některé kroužky otevřít, přesunout a pak zase uzavřít. Protože se tím ale řetízek poškozuje, je třeba otevřít co nejmenší počet kroužků. A to už je úloha pro vás. Na vstupu je dán počet kroužků N a zašmodrchaný řetízek. Ten je popsán dvojicemi kroužků, které jsou spojeny (kroužky si budeme číslovat od 1 do N). Vaším úkolem je zjistit minimální počet kroužků, které je třeba rozpojit, aby ze zamotaného řetízku vznikl jeden jednoduchý řetízek.

Příklad 1:
Počet kroužků: 4
Propojení: (1, 2), (2, 3), (2, 4)
Je třeba rozpojit alespoň 1 kroužek.
(např. 4, který se pak připojí za 3)

Příklad 2:
Počet kroužků: 5
Propojení: (1, 2), (2, 3), (3, 4), (4, 1), (1, 5)
Je třeba rozpojit alespoň 1 kroužek.
(např 1 – 4 z něj pak můžeme vyvléknout)

12-4-2 Pyrus Achras — 10 bodů

Pan Hruška měl sad. Jednoho dne se rozhodl, že si do sadu vysadí novou odrůdu svého nejoblíbenějšího stromu – hrušně – a vyhlédl si pro ni pěkné místo. Bohužel se koupí nového stromu poněkud finančně vyčerpal, a tak již nemá na přikoupení pletiva na plot, který si kolem svého sadu staví. Potřeboval by tedy zjistit, jestli jím vybrané místo leží uvnitř projektovaného oplocení nebo nikoliv. A to je již úkol pro vás.

Na vstupu je počet stromů N a jejich souřadnice. Dále jsou dány souřadnice vybraného místa. Oplocení je projektováno jako nejmenší možné – tedy nejmenší konvexní k-úhelník obsahující všechny dané stromy. Váš program má zjistit, zda vybrané místo leží uvnitř nebo vně příslušného k-úhelníka.

Příklad 1:
Stromů: 5
Pozice: (0, 0), (2, 0), (0, 2), (1, 1), (2, 3)
Nový strom: (2.5, 0.5)
Nový strom leží mimo.

Příklad 2:
Stromů: 3
Pozice: (0, 0), (2, 0), (1, 2)
Nový strom: (1, 1)
Nový strom leží uvnitř.

12-4-3 Šachová šlamastika — 10 bodů

Bílá dáma si objednala dláždění do své komnaty. Jak jinak než po vzoru šachovnice. Dlaždiči ovšem místo krásné a pravidelné šachovnice vydláždili na podlahu jiné ornamenty a dáma je teď nešťastná. Bílý král si vás tedy najal, abyste nalezli v komnatě největší oblast, kde by se královna mohla cítit jako doma (tedy na šachovnici).

Na vstupu jsou dány rozměry komnaty M a N a dále popis vydláždění komnaty. Vydláždění je popsáno maticí $M \times N$ obsahující nuly a jedničky. Nula je na místě bílého políčka, jednička na místě černého. Vaším úkolem je napsat program, který v dané matici nalezne obdélník s největším obsahem takový, že se v něm budou střídat barvy jako na šachovnici.

Příklad:
Rozměry: 5, 6
00101
11001
01010
10001
11111
Největší šachová podmatice má obsah 6.
Levý horní roh: 4, 2. Rozměry: 2, 3.

12-4-4 Barevné intervaly — 10 bodů

Jsou dány neprázdné intervaly $(a_1, b_1) \ldots (a_n, b_n)$. Každý z intervalů se pokusíme obarvit nějakou barvou z dané sady tak, aby žádné dva intervaly

s neprázdným průnikem neměly stejnou barvu. Je asi zřejmé, že ne vždy to jde. Například intervaly $(1,3),(2,4)$ a $(1,4)$ pomocí dvou barev obarvit nelze.

Vaším úkolem je napsat program, který pro dané intervaly zjistí nejmenší počet barev k takový, že intervaly jdou obarvit a dále nalezne příslušné obarvení (barvy pro jednoduchost označujte čísly jedna až k).

Příklad:
Intervaly: (1, 3), (2, 4), (1, 4), (2, 3), (0, 2)
Jsou potřeba 4 barvy.
Možné obarvení je: (1, 3) = 1; (2, 4) = 2; (1, 4) = 3; (2, 3) = 4; (0, 2) = 2

12-4-5 PRAM — 10 bodů

V této sérii budeme řešit úlohy dvě, ale zato pouze na `P-CRCW PRAM`u.

1) Vytvořte program, který dostane zadanou posloupnost n celých čísel a spočítá posloupnost jejích částečných součtů, tj. $a_1, a_1 + a_2, a_1 + a_2 + a_3, \ldots$ Délka posloupnosti bude při spuštění uložena v `gmem[0]` a posloupnost sama pak v buňkách `gmem[1]` až `gmem[n]`. Obsah paměťových buněk před začátkem výpočtu by tedy mohl být například: 5, 4, 0, 4, -2, 3. Obsah paměťových buněk po skončení výpočtu by pak měl být: 5, 4, 4, 8, 6, 9.
2) Vytvořte program, který jako vstup obdrží číslo n a spočte horní celou část jeho dvojkového logaritmu. Vstupní číslo je zadáno v `gmem[0]`; výsledek by měl být po skončení výpočtu uložen též v `gmem[0]`. Váš algoritmus by měl používat právě n procesorů. Je ale samozřejmě možné, že některé z nich nemusí vůbec počítat a mohou se tedy hned v prvním kroku zastavit instrukcí `halt`.

Vzorová řešení

12-1-1 Dlaždiči — Jan Kára

Dobrých řešení této úlohy bylo přibližně jako dinosaurů (šafrán je v našich krajích příliš častý). Je ovšem třeba přiznat, že je to i naše chyba, protože jsme špatně odhadli obtížnost této úlohy… S výjimkou tří dobrých řešení pomocí párování v bipartitním grafu se nám zde sešla pěkná řádka heuristik a backtrackingů. Chybou autorů heuristik bylo často neexistující nebo nekorektní zdůvodnění správnosti (jinak byste totiž zjistili, že váš algoritmus nefunguje). Také odhady složitosti zjevně dělaly problémy mnohým řešitelům.

A teď už k řešení úlohy. Náš problém s dlaždicemi převedeme na poměrně známý problém perfektního párování v bipartitním grafu. Na ten už existují rychlé a osvědčené algoritmy. Abychom si řešení mohli vysvětlit, uvedeme si nejdříve několik definic:

- *Graf* G je uspořádaná dvojice (V, E), kde $E \subseteq \binom{V}{2}$. Prvky množiny V budeme nazývat vrcholy, prvky množiny E hrany. A teď lidsky: *Graf* je nějaká sada bodů, přičemž některé body jsou spojeny čarami — klasickým příkladem je třeba silniční síť. Křižovatky jsou vrcholy, silnice mezi křižovatkami hrany.
- *Bipartitní graf* je takový graf, kde vrcholy lze rozdělit na dvě skupiny tak, že hrany vedou pouze mezi těmito skupinami. Tedy žádná hrana nevede mezi dvěma vrcholy stejné skupiny.
- *Párování v grafu* je taková množina hran, že žádné dvě hrany nesdílí vrchol. O párování řekneme, že je *perfektní*, jestliže v každém vrcholu končí jedna z vybraných hran.

A jak tyto pojmy souvisí s naším problémem? Naše plocha k vydláždění je vlastně bipartitní graf. Políčka k vydláždění budou vrcholy a hrana mezi dvěma vrcholy povede tehdy, jestliže odpovídající políčka sousedí hranou. To, že to bude graf bipartitní plyne z faktu, že pokud si představíme šachovnicové obarvení plochy a do jedné skupiny zařadíme „bílé“ vrcholy a do druhé „černé“ vrcholy, tak zajisté povedou všechny hrany pouze mezi vrcholy s různými barvami. Dlaždice jsme na plochu mohli položit tam, kde byla dvě sousední políčka volná. To je tedy přesně tam, kde nyní mezi odpovídajícími vrcholy vede hrana. Rozložení dlaždic na ploše tedy odpovídá nějakému párování – požadavek na to, aby hrany neměly společný vrchol přesně odpovídá požadavku, aby se žádné dvě dlaždice nepřekrývaly. Tedy problém, zda lze plochu vydláždit jsme převedli na problém, zda existuje párování zahrnující všechny vrcholy – tedy perfektní párování.

Algoritmus na hledání perfektního párování bude pracovat tak, že bude postupně (v každém kroku o 1) zvětšovat velikost párování. Pokud již párování nepůjde zvětšit, zjistíme, zda pokrývá všechny vrcholy. Pokud ano, plochu vydláždit jde, pokud ne, plochu vydláždit nejde. Aktuální párování (na počátku třeba prázdné – nejsou spárovány žádné vrcholy) budeme vylepšovat následovně: Nalezneme cestu, která začíná v nespárovaném vrcholu, pokračuje po hraně, která není v párování, pak po hraně, která v párování je, a tak dále až do vrcholu, který spárování není. Pokud na této cestě (zjevně musela mít lichou délku, protože začínala i končila hranou mimo párování a všude se hrany pravidelně střídaly) zaměníme hrany v párování a mimo něj, zjevně získáme párování o jednu hranu větší. To zase můžeme stejným způsobem zlepšovat. Pozorování, že takováto cesta bude vždy existovat, pokud párování jde zvětšit vám necháváme k rozmyšlení (návod: zkuste to sporem a podívejte se na graf zachycující rozdíly mezi párováním, ve kterém jste cestu už nenašli, a maximálním párováním).

Zbývá ještě otázka, jak najít onu kýženou cestu. To jde ale velmi snadno. Použijeme lehce modifikované prohledávání do šířky. Na začátku budeme mít ve frontě ke zpracování nespárovaný vrchol. Krok hledání proběhne tak, že vždy z fronty odebereme vrchol a podíváme se na jeho sousedy. Pokud je nějaký soused nespárovaný, máme hledanou cestu. Pokud jsou všichni spárovaní, přidáme do fronty odpovídající sousedy sousedů v párování (samozřejmě přidáváme pouze vrcholy, které jsme v tomto prohledávání dosud nezpracovali). Když se fronta vyprázdní a nenalezli jsme cestu, do tohoto vrcholu zjevně hledaná cesta nevede a snadno si můžete dokázat, že ani nikdy vést nemůže, takže vydláždění neexistuje.

Algoritmus v každém kroku zvětší párování o jednu hranu. Kroků tedy může být nejvýše N (počet vrcholů). Jeden krok nám trvá $O(M)$, kde M je počet hran. Celkem by čas tedy byl $O(M \cdot N)$. Když si ale navíc povšimneme, že hran je nejvýše $2 \cdot N$, získáváme lepší odhad časové složitosti – $O(N^2)$. Paměťová složitost algoritmu je $O(N)$.

Správnost algoritmu byla ukázána již v jeho popisu. Program je přímou imlementací algoritmu.

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

#define MAXSIZE 20

 /* Struktura pro hranu */
struct edge {
    int to;                          /* Cílovy vrchol hrany */
    char inmatch;                    /* Je hrana v párování? */
};

int m, n;                            /* Rozměry plochy */
int map[MAXSIZE][MAXSIZE];           /* Mapa plochy */
```

```
int V[2];                                   /* Počty vrcholů v partitách */
struct edge E[MAXSIZE*MAXSIZE][4];          /* Hrany */
int Deg[MAXSIZE*MAXSIZE];                   /* Stupně jednotlivých vrcholů */
const int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};   /* Jednotlivé směry */
int From[MAXSIZE*MAXSIZE];                  /* Pole s vrcholy, odkud jsme přišli */

void ReadArea (void)
{
  int i, j;

  printf ("Velikost plochy:␣");
  scanf ("%d %d", &m, &n);
  getchar ();
  for (i = 0; i < m; i++) {
    printf ("Radek %d:\n", i+1);
    for (j = 0; j < n; j++) {
      if (getchar () == '1') {              /* Místo k vydláždění? */
        V[ (i+j)&1]++;                      /* Máme nový vrchol do partity */
        map[i][j] = V[0] + V[1];            /* Poznamenáme si číslo vrcholu */
      }
    }
    getchar ();
  }
}

/* Propojí jednotlivé vrcholy hranami */
void CreateGraph (void)
{
  int i, j, k;

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      if (map[i][j])
        for (k = 0; k < 4; k++)
          /* Nevylézáme z mapy a je políčko volné? */
          if (i+dy[k] >= 0 && i+dy[k] < m && j+dx[k] >= 0 && j+dx[k] < n &&
              map[i+dy[k]][j+dx[k]])
            /* Přidáme hranu */
            E[map[i][j]][Deg[map[i][j]]++].to = map[i+dy[k]][j+dx[k]];
}

/* Zjistí, zda je vrchol v párování */
int InMatching (int V)
{
  int k;

  for (k = 0; k < Deg[V]; k++)
    if (E[V][k].inmatch)
      return k;
  return -1;
}
/* Vrátí index hrany U↔V v U */
int EdgeIndex (int U, int V)
{
  int k;
```

```
    for (k = 0; k < Deg[U] && E[U][k].to != V; k++);
    return k;
}

/* Zamění hrany v párování a mimo něj na cestě */
void AlternateWay (int V)
{
    int inmatch = 1;

    while (From[V]) {
        E[V][EdgeIndex (V, From[V])].inmatch = inmatch;
        E[From[V]][EdgeIndex (From[V], V)].inmatch = inmatch;
        V = From[V];
        inmatch = !inmatch;
    }
}

/* Zjistí, zda existuje perfektní párování */
int PerfectMatching (void)
{
    int i, k;
    int Q[MAXSIZE*MAXSIZE];                 /* Fronta s vrcholy ke zpracování */
    int QS, QE;                             /* Počátek a konec fronty */
    int MatchSize;
    int AV;

    for (MatchSize = 0; MatchSize < V[0]; MatchSize++) {
        memset (From, 0, sizeof (int) * MAXSIZE * MAXSIZE);
        for (i = 1; i < V[0] + V[1]; i++)
            if (InMatching (i) == -1)       /* Vrchol je nespárovaný? */
                break;
        QS = QE = 0;
        for (Q[QE++] = i; QE > QS; QS++) {  /* Je něco ve frontě? */
            AV = Q[QS];
            for (k = 0; k < Deg[AV]; k++)
                if (!From[E[AV][k].to]) {   /* Ještě nedosažený vrchol? */
                    From[E[AV][k].to] = AV;
                    /* Je i soused nespárovaný? */
                    if (InMatching (E[AV][k].to) == -1) {
                        AlternateWay (E[AV][k].to);
                        goto cont;          /* Začneme hledat další cesty */
                    }
                    /* Pridame do fronty vrchol z konce sparovane hrany */
                    Q[QE] = E[E[AV][k].to][InMatching (E[AV][k].to)].to;
                    From[Q[QE++]] = E[AV][k].to;
                }
        }
        /* Nenalezli jsme žádnou zlepšující cestu - vydláždění neexistuje */
        return 0;
cont:
    }
    return 1;
}
```

```
int main (void)
{
    ReadArea ();                          /* Načteme popis plochy */
    if (V[0] != V[1]) {                   /* Různý počet vrcholů v partitách -
                                             párování neexistuje */
      puts ("Tak tohle tedy nevydlazdite.");
      return 0;
    }
    CreateGraph ();                       /* Propojí jednotlivé vrcholy hranami */
    if (PerfectMatching ())               /* Zjistí, zda existuje perf. párování */
      puts ("Plocha vydlazit lze.");
    else
      puts ("Plocha vydlazdit nelze.");
    return 0;
}
```

12-1-2 Parník — Martin Mareš

Tento příklad je možno řešit neobyčejně mnoha různými algoritmy. Zde jsme si vybrali jeden velice elegantní, který ovšem vyžaduje trošku přemýšlení o tom, proč vlastně funguje...

Úlohu si přeformulujme do řeči teorie grafů: máme zadánu nějakou množinu *vrcholů* (význačné body říční sítě), které jsou spojeny *hranami* (splavné úseky řeky) a každé hraně je přiřazeno kladné *ohodnocení* (délka úseku). Navíc víme, že mezi každými dvěma vrcholy vede právě jedna *cesta* (posloupnost na sebe navazujících hran taková, že první hrana posloupnosti začíná v prvním z vrcholů, poslední hrana končí v druhém z vrcholů a žádný vrchol není navštíven vícekrát). Takovým grafům říkáme *stromy* a nejprve si je pořádně prohlédneme:

Pozorování 1: Každý strom má alespoň jeden *list,* to jest vrchol, z nějž vede pouze jedna hrana, a oba krajní vrcholy nejdelší cesty jsou listy. [Nejdelší cesta jistě existuje a kdyby libovolný z jejích krajních vrcholů nebyl list, vedla by z něj ještě alespoň jedna nevyužitá hrana a o tu bychom mohli cestu prodloužit ... jenže všechny hrany mají kladné délky, tudíž nová cesta by byla ještě delší než ta nejdelší, což je spor.]

Pozorování 2: Strom o $n \geq 1$ vrcholech má $n - 1$ hran. [To dokážeme třeba indukcí. Pro strom na jednom vrcholu to jistě je pravda, pokud je $n > 1$ a pro všechna menší n již tvrzení platí, odebereme ze stromu jeden list (jak již víme, vždycky nějaký existuje), tím jsme získali strom na $n-1$ vrcholech, který má podle indukčního předpokladu $n - 2$ hran, načež list připojíme zpět, čímž přibyl jeden vrchol a jedna hrana, takže hran je celkem $n - 1$, a to je přesně, co jsme chtěli.]

Značení: $\langle xy \rangle$ budeme značit cestu mezi vrcholy x a y, $d(x, y)$ pak bude délka této cesty. Jelikož náš graf je neorientovaný, platí $d(x, y) = d(y, x)$.

Tvrzeníčko: Zvolme si libovolný vrchol x a vrchol y od x nejvzdálenější. Potom jedna z nejdelších cest v našem stromu končí vrcholem y.

Důkaz: Ve skutečnosti dokážeme ještě silnější tvrzení: máme-li ve stromu vrchol x, od něj nejvzdálenější vrchol y a cestu $\langle ab\rangle$, pak $d(y,b) \geq d(a,b)$ (jinými slovy cesta $\langle yb\rangle$ je alespoň stejně dlouhá jako $\langle ab\rangle$). Rozlišme dva případy:

- Cesta $\langle xy\rangle$ protíná cestu $\langle ab\rangle$. Označme si p ten ze společných vrcholů těchto cest, který leží nejblíže k b. Potom z toho, že $d(x,y) \geq d(x,a)$ plyne také $d(p,y) \geq d(p,a)$ [protože $d(x,y) = d(x,p)+d(p,y)$ a analogicky pro $d(x,a)$], a tedy i $d(y,b) \geq d(a,b)$ [$d(a,b) = d(a,p) + d(p,b)$ a stejně tak pro $d(y,b)$].
- Není tomu tak. Buď q vrchol na cestě $\langle ab\rangle$ nejbližší k vrcholu x. Pak cesta $\langle xy\rangle$ nutně protne cestu $\langle xb\rangle$ (například ve vrcholu x). Nechť r je poslední z průsečíků těchto cest ve směru od x. Pak jelikož $d(x,y) \geq d(x,a)$, je také $d(r,y) \geq d(r,a)$, tím spíše $d(q,y) \geq d(q,a)$ [k levé straně jsme přičetli a od pravé odečetli kladné číslo $d(r,q)$] a z toho stejně jako v předchozím případě dostaneme $d(y,b) \geq d(a,b)$.

Z tohoto tvrzeníčka již snadno odvodíme algoritmus pro hledání kýžené nejdelší cesty: Zvolíme si libovolný vrchol x, nalezneme od něj nejvzdálenější vrchol y (pokud jich je více, tak kterýkoliv z nich) a pak od vrcholu y nejvzdálenější vrchol z, načež cestu $\langle yz\rangle$ prohlásíme za nejdelší. Z výše uvedeného důkazu plyne, že to vždy bude pravda.

Nejvzdálenější vrchol budeme v obou případech hledat prohledáním stromu do hloubky. Graf budeme representovat seznamy sousedů jednotlivých vrcholů. Začneme ve vrcholu, z nějž vzdálenosti měříme, rekursivním voláním téže funkce nalezneme pro každého syna zpracovávaného vrcholu nejvzdálenější vrchol v podstromu určeném tímto synem a poté z těchto mezivýsledků vybereme nejvzdálenější a ten prohlásíme za výsledek. Jelikož náš graf je souvislý a každou hranu máme reprezentovanou v obou orientacích (při rekursivním volání si ovšem pamatujeme předchozí vrchol, abychom se nezacyklili), projdeme jej nutně celý.

Časová složitost našeho algoritmu je $O(N)$ (při každém z obou prohledávání grafu každý vrchol a každou hranu navštívíme právě jednou; vrcholů je N, hran podle Pozorování 2 je $N-1$). Paměťová složitost činí $O(N^2)$, ale to pouze kvůli neúsporné reprezentaci stromu v paměti, kterou bychom snadno mohli upravit tak, aby si vystačila s pamětí $O(N)$.

```
#include <stdio.h>

#define MAXN 20
int d[MAXN][MAXN];          /* Maximální počet vrcholů */
int e[MAXN][MAXN];          /* Sousedé vrcholů */
int deg[MAXN];              /* Délky hran do sousedů */
                            /* Stupně vrcholů (počty sousedů) */
```

```
int solve (int x, int from, int *leaf)          /* Hledání nejvzdálenějšího vrcholu */
{
        int i, l, max, v;

        *leaf = x;
        max = 0;
        for (i=0; i<deg[x]; i++)
                if (d[x][i] != from) {      /* Nechceme se zacyklit */
                        l = solve (d[x][i], x, &v) + e[x][i];
                        if (l > max) {
                                max = l;
                                *leaf = v;
                        }
                }
        return max;
}
int main (void)
{
        int x, y, l;
        while (scanf ("%d%d%d", &x, &y, &l) == 3) {
                e[x][deg[x]] = l; d[x][deg[x]++] = y;
                e[y][deg[y]] = l; d[y][deg[y]++] = x;
        }
        solve (0, -1, &x);
        l = solve (x, -1, &y);
        printf ("%d -> %d = %d\n", x, y, l);
        return 0;
}
```

12-1-3 A sort of sort — Pavel Machek

Pomůcky: n prázdných papírků
n kbelíků očíslovaných $0..n-1$
čas $3n$
3 svíce

O třetí noci po slunovratu vezmi čísla,
každé z nich o jedničku zmenši,
do n-kové soustavy přepiš,
na připravené papírky zapiš.

Vezmi papírky popořadě,
na poslední cifru se soustřeď,
do kbelíku s číslem řádným
potom papírek zařaď.

Až rozdáš papírky všechny,
první svíci zapal,
potom papírky postupně

z kbelíků 0 až $n-1$ vybal
(Pozor však musíš dát:
ten papírek, který první vložen byl,
první musí být vyňat)

Vezmi papírky popořadě,
na střední cifru se soustřeď,
do kbelíku s číslem řádným
potom papírek zařaď.

Až rozdáš papírky všechny,
druhou svíci zapal,
potom papírky postupně
z kbelíků 0 až $n-1$ vybal.

Vezmi papírky popořadě,
na první cifru se soustřeď,
do kbelíku s číslem řádným
potom papírek zařaď.

Až rozdáš papírky všechny,
poslední svíci zapal,
potom papírky postupně
z kbelíků 0 až $n-1$ vybal.

Nyní máš papírky s čísly a hle!
čísla popořadě jdou.

Správnost algoritmu je nasnadě. Čísla budou zřejmě korektně seřazena vzhledem k první cifře. Korektní seřazení vzhledem k druhé cifře nám zaručí předchozí průchod. Ten totiž seřadil čísla podle druhé číslice. Když jsme tedy potom čísla rozdělovali podle první cifry, tak první přišla na řadu čísla s nižší druhou cifrou, a tedy v každém kbelíku budou čísla seřazena podle druhé číslice. Když je tedy z kbelíků na konci vybereme, budou korektně seřazena nejen vzhledem k první, ale i vzhledem k druhé cifře. Obdobnou úvahu můžeme provést při řazení podle poslední cifry, čímž dojdeme k závěru, že čísla budou skutečně korektně seřazena.

Program používá pole velikosti $O(N^2)$, takže skutečná časová složitost je $O(N^2)$. Nebyl by ovšem problém toto pole nahradit dynamicky alokovanými položkami, čímž by se časová složitost stala skutečně lineární.

```
#include <stdio.h>
#define MAXN 40                          /* Maximální počet čísel  */
int buckets[MAXN][MAXN];                 /* Přihrádky na čísla  */
int bused[MAXN];                         /* Počet čísel v přihrádce  */
```

```
int A[MAXN];                                  /* Posloupnost čísel */
int N;                                        /* Počet čísel */

int main (void)
{
   int i, j, D = 1, k;                        /* ; Aktuální číslice; N umocněné na j; */
   int ActN;                                  /* Počet sebraných čísel */

   /* Nacteme vstup */
   scanf ("%d", &N);
   for (i = 0; i < N; i++) {
      scanf ("%d", &A[i]);
      A[i]--;
   }
   for (j = 0; j < 3; j++) {                  /* Tři průchody... */
      for (i = 0; i < N; i++)                 /* Inicializace přihrádek */
         bused[i] = 0;
      for (i = 0; i < N; i++)                 /* Rozdělíme čísla do přihrádek */
         buckets[A[i]/D%N][bused[A[i]/D%N]++] = A[i];
      ActN = 0;
      for (i = 0; i < N; i++)                 /* Sebereme čísla z přihrádek */
         for (k = 0; k < bused[i]; k++)
            A[ActN++] = buckets[i][k];
      D *= N;
   }
   for (i = 0; i < N; i++)
      printf ("%d\n", A[i]+1);
   return 0;
}
```

12-1-4 PRAM — Daniel Kráľ

Myšlenka, jak vyřešit zadanou úlohu na `P-CRCW PRAM`u, napadla každého z Vás. Použijeme tolik procesorů, kolik mají zadaná čísla cifer; každý z procesorů bude mít na starosti dvojici cifer stejného řádu, porovnají je a pokud se liší, pak svůj výsledek ve stejný okamžik zapíší do `gmem[0]` – procesory s nižším číslem (vyšší prioritou) budou porovnávat cifry vyššího řádu a tedy se do `gmem[0]` zapíše výsledek porovnání rozdílných cifer nejvyššího možného řádu. Bohužel již ne ve všech řešeních byla tato myšlenku úspěšně realizována: Především je třeba před porovnáním cifer obou čísel uložit do gmem nulu, aby byl výsledek výpočtu správný i v případě, že se obě čísla shodují. Výsledek porovnání navíc procesory musí zapsat ve stejný okamžik, tj. nelze nejprve testovat, zda cifry prvního čísla jsou menší než cifry druhého a v kladném případě zapsat jedničku, a dalším příkazem testovat opak a zapisovat dvojku (viz porovnávání čísel 101_2 a 110_2). Samotný program, pokud máme na paměti tyto dvě věci, je snadné napsat – počet použitých procesorů je $O(N)$, časová složitost je $O(1)$ a paměťová je $O(N)$, kde N je počet cifer porovnávaných čísel.

Vyřešení zadané úlohy na CREW PRAMu bylo velmi snadné; program pro CREW PRAM, uvedený jako ukázkový v zadání první série, totiž řešil následující úlohu: V zadané posloupnosti nul a jiných čísel (v ukázkovém programu čísel procesorů) totiž našel první nenulové číslo a to prohlásil za výsledek; pokud byla celá posloupnost nulová, byla výsledkem nula. Náš program tedy bude fungovat následovně: Nejprve každý z procesorů porovná dvojici bitů stejného řádu a výsledek uloží na příslušnou pozici do globální paměti; zde je třeba dát pozor, že výsledek nelze do globální paměti ukládat rovnou, neboť je nejprve potřeba provést obě porovnání – následující totiž nefunguje (jak každý jistě sám snadno zjistí pro dvojici cifer 0 a 1):

```
lmem[0]:=cpuid+gmem[0]
if gmem[cpuid]<gmem[lmem[0]] then gmem[cpuid] := 1
if gmem[cpuid]>gmem[lmem[0]] then gmem[cpuid] := 2
if gmem[cpuid]=gmem[lmem[0]] then gmem[cpuid] := 0
```

Napsat nyní program s využitím postupu použitého ve vzorovém programu v zadání první série je již triviální – počet použitých procesorů je $O(N)$, časová složitost je $O(logN)$ a paměťová je $O(N)$, kde N je počet cifer porovnávaných čísel.

Nyní obraťme svoji pozornost k EREW PRAMu. V zadání první série jsme úmyslně zatajili technické detaily, jak vzorovou úlohu přesně na EREW PRAMu vyřešit; bohužel to znamenalo, že většina z Vás se nad úlohou příliš nepokusila zamyslet. Jediný rozdíl oproti CREW PRAMu totiž spočívá v tom, že všechny procesory nemohou v jediném kroku zjistit počet cifer zadaných čísel – to by totiž vyvolalo konflikt při čtení (různé procesory by se pokusily číst tutéž hodnotu); řešení založené na tom, že na začátku proběhne cyklus délky N, v jehož i-té iteraci si i-tý procesor přečte gmem[0], je zcela neparalelní (a tudíž hodnoceno odpovídajícím počtem bodů) – jenom tento cyklus zabere čas $O(N)$ a tedy ve stejném čase bychom mohli celý výpočet provést i s jediným procesorem. Konflikt vyřešíme tak, že si hodnotu rozkopírujeme v globální paměti. Cifry prvního čísla si uschováme bokem, a před první kopírovací vlnou bude znát délku posloupnosti pouze první procesor, který ji zapíše do gmem[1]. Po první vlně budou znát délku dva procesory a bude uložena ve gmem[1] a gmem[2], po druhé vlně čtyři procesory a délka posloupnosti se bude již nacházet v paměťových buňkách gmem[1] až gmem[4] atd. Obecně v i-té vlně prvních 2^{i-1} procesorů zkopíruje obsah gmem[cpuid] o 2^{i-1} buněk dál. Po logaritmicky mnoha krocích již délku posloupnosti znají všechny procesory a můžeme pokračovat stejně jako na CREW PRAMu. Vytvoření programu pro PRAM je nyní již triviální - počet použitých procesorů je $O(N)$, časová složitost je $O(logN)$ a paměťová je $O(N)$, kde N je počet cifer porovnávaných čísel.

Za vyřešení úlohy pro P-CRCW, CREW a EREW PRAM bylo možné po řadě získat 2, 4 a 4 body.

Program pro P-CRCW PRAM:

```
lmem[0]:=cpuid+gmem[0]
lmem[1]:=0
gmem[0]:=0
if gmem[cpuid]<gmem[lmem[0]] then lmem[1]:=1
if gmem[cpuid]>gmem[lmem[0]] then lmem[1]:=2
if lmem[1]<>0 then gmem[0]:=lmem[1]
halt
```

Program pro CREW PRAM:

```
   lmem[0]:=gmem[0]              ; tady bude délka posloupnosti
   lmem[1]:=cpuid+gmem[0]        ; index prvku v druhém čísle
   lmem[2]:=0
   if gmem[cpuid]<gmem[lmem[1]] then lmem[2]:=1
   if gmem[cpuid]>gmem[lmem[1]] then lmem[2]:=2
   gmem[cpuid]:=lmem[2]
   if gmem[cpuid]<>0 then goto 2 ; není se třeba dívat dál
   lmem[1]:=cpuid                ; další testovaná buňka
   lmem[2]:=cpuid-1              ; moje číslo minus 1
   lmem[3]:=1                    ; posun k další buňce
1: lmem[4]:=lmem[2]&lmem[3];     ; budeme končit?
   if lmem[4]<>0 then goto 2     ; končíme
   lmem[1]:=cpuid+lmem[3];
   lmem[3]:=lmem[3]<<1
   if lmem[1]>lmem[0] then goto 2   ; jsme za koncem posloupnosti
   if gmem[lmem[1]]<>0 then gmem[cpuid]:=gmem[lmem[1]]
   if gmem[cpuid]=0 then goto 1     ; ještě jsme nenašli nenulu
2: if cpuid=1 then gmem[0]:=gmem[1] ; a zapíšeme výsledek
   halt
```

Program pro EREW PRAM:

```
   if cpuid=1 then lmem[0]:=gmem[0] ; délka posloupnosti
   lmem[1]:=gmem[cpuid]             ; schováme si obsah paměti
   gmem[cpuid]:=0                   ; znulujeme paměť
   if cpuid=1 then gmem[cpuid]:=gmem[0]
   lmem[2]:=1                       ; počet buňek s délkou posl.
1: lmem[3]:=cpuid+lmem[2]
   lmem[0]:=gmem[cpuid]
   if lmem[3]<=lmem[0] then gmem[lmem[3]]:=gmem[cpuid]
   lmem[2]:=lmem[2]<<1
   lmem[3]:=0                       ; budeme ještě kopírovat?
   if gmem[cpuid]=0 then lmem[3]:=1
```

```
   if gmem[cpuid]>lmem[2] then lmem[3]:=1
   if lmem[3]=1 then goto 1
   lmem[0]:=gmem[cpuid]              ; délka posloupnosti
   lmem[2]:=cpuid+lmem[0]            ; index prvku v druhém čísle
   lmem[3]:=0
   if lmem[1]<gmem[lmem[2]] then lmem[3]:=1
   if lmem[1]>gmem[lmem[2]] then lmem[3]:=2
   gmem[cpuid]:=lmem[3]
   if gmem[cpuid]<>0 then goto 3     ; není se třeba se dívat dál
   lmem[1]:=cpuid                    ; další testovaná buňka
   lmem[2]:=cpuid-1                  ; moje číslo minus 1
   lmem[3]:=1                        ; posun k další buňce
2: lmem[4]:=lmem[2]&lmem[3];         ; budeme končit?
   if lmem[4]<>0 then goto 3         ; končíme
   lmem[1]:=cpuid+lmem[3];
   lmem[3]:=lmem[3]<<1
   if lmem[1]>lmem[0] then goto 3    ; jsme za koncem posloupnosti
   if gmem[lmem[1]]<>0 then gmem[cpuid]:=gmem[lmem[1]]
   if gmem[cpuid]=0 then goto 2      ; ještě jsme nenašli nenulu
3: if cpuid=1 then gmem[0]:=gmem[1] ; a zapíšeme výsledek
   halt
```

12-2-1 Osmisměrka — Aleš Přívětivý

Vyřešit tuto úlohu nečinilo nikomu velký problém, horší to už bylo s efektivitou navržených algoritmů. Většina z řešitelů zvolila ten nejjednodušší algoritmus, spočívající v myšlence, že pro každé políčko osmisměrky probereme postupně všechna slova a zjišťujeme, zda na tomto políčku nezačínají. Pokud tomu tak je, označíme si u políček, které tvoří nalezené slovo, že jsou vyškrtnutá. Nakonec projdeme celou osmisměrku a vypíšeme všechna nezaškrtnutá políčka. Označíme-li si rozměry osmisměrky M a N a počet písmen ve všech slovech slovníku P, vychází nám časová složitost $O(MNP)$ – políček je $M \times N$, pro každé z nich musím zkontrolovat všechna slova o celkovém počtu P písmen. Paměťová složitost je $O(MN + P)$. Každý asi pochopí proč za toto řešení nesklidil plný počet bodů.

V čem je problém a co by se dalo zlepšit? Asi není nejlepší nápad testovat pro každé políčko všechna slova, ale vytvořit si nějakou strukturu, která nám umožní vyhledat slovo ve slovníku průchodem textu od daného políčka v jednom z osmi směrů. Vyskytly se návrhy stromových struktur, které toto umožňovaly, ale většinou nebyly dotaženy do konce. Toto řešení ale vynechejme, protože si ukážeme ještě lepší algoritmus, který se od tohoto zase tak moc neliší. Hledání ve stromové struktuře probíhá v čase úměrném délce nejdelšího

slova slovníku, označme tuto délku D, tedy celková časová složitost takového algoritmu by byla $O(MND)$ plus čas na zkonstruování stromové struktury $O(P)$ a paměťová $O(MN + P)$. To je sice mnohem lepší než předchozí algoritmus, jelikož D je mnohem menší než P, ale naším cílem bude dosáhnout algoritmu pracujícího v čase $O(MN + P)$.

Uvažujme jednodušší případ – máme řádek textu a chceme v něm najít a vyškrtat výskyty určitých slov v lineárním čase vzhledem k jeho délce. Pokud toto budeme umět, můžeme to provést na všechny řádky, sloupce a diagonály a budeme mít kýžený výsledek. Jak to ale provést? K tomu nám dopomohou konečné automaty – takový konečný automat si lze zjednodušeně představit jako stroj, který se může vyskytovat v různých stavech (ale vždy právě v jednom), který ze vstupu vezme jedno písmeno a přejde z aktuálního stavu do jiného (cílový stav závisí na přečteném písmenu), pak vezme další písmeno a zase přejde do nějakého stavu atd. Automat má také určité zvláštní stavy, do kterých když se dostane, znamená to, že posledních k písmen tvoří nějaké hledané slovo. Např.: Mějme automat rozpoznávající slova `LES`, `ALES`, `ESO`, pak při analýze textu `PRALESOV` po zpracování šestého písmene se automat dostane do stavu, který znamená, že zatím přečtený text končí na slova `ALES` a `LES`, a po zpracování sedmého písmene, do stavu indikujícího, že na konci prozatím přečteného textu je slovo `ESO`.

Pro náš případ bude postačující, když si automat po zpracování každého písmene někam poznamená délku nejdelšího z hledaných slov, které tvoří konec (příponu) prozatím přečteného textu – jestliže žádné takové slovo neexistuje, bere délku nulovou. Tedy po skončení práce obdržíme od konečného automatu pole, ve kterém bude pro každé písmeno textu délka nejdelšího slova ze všech hledaných, která jsou obsažena v textu a končí v tomto písmeni – pro výše uvedený příklad bude výsledek $(0, 0, 0, 0, 0, 0, 4, 3, 0)$. Podle takového pole už určitě dokážeme v lineárním čase vzhledem k jeho délce (a tedy i délce analyzovaného textu) vyškrtat písmena obsažená v nalezených slovech. Zbývá ještě ukázat, že dokážeme efektivně zkonstruovat automat pro libovolný slovník. To lze pomocí algoritmu Aho-Corasickové s časovou a paměťovou složitostí $O(P)$, tedy celý algoritmus na luštění osmisměrek využívající konečných automatů má celkovou časovou i paměťovou složitost $O(MN + P)$. Že to lépe nelze, je vidět z faktu, že tento čas potřebujeme už jen k načtení slovníku a osmisměrky. Správnost algoritmu, zde dokazovat nebudu, ale poznatků chtivý čtenář se může podívat do studijního textu z knihy *J. Pavelka, M. Chytil: Algoritmy* uvedeného v upravené podobě na konci řešení nebo si přečíst úlohy týkající se konečných automatů v osmém ročníku KSP.

```
#include <stdio.h>
#include <string.h>
#define MAX_SLOVNIK 8192                /* pocet pismen slovniku  */
#define MAX_ROZMER 80                   /* maximalni rozmer osmismerky  */
```

```
#define UNDEFINED (-1)                    /* konstanta pro nedefinovanou hodnotu
                                             fce g */
#define max (x, y) ( (x)> (y) ? (x): (y))
#define min (x, y) ( (x)< (y) ? (x): (y))
int g[MAX_SLOVNIK][256];                  /* prechodova fce g: stavy × abeceda →
                                             stavy */
int o[MAX_SLOVNIK];                       /* maximalni delka prijmuteho slova v
                                             tomto stavu */
int f[MAX_SLOVNIK];                       /* zpetna fce f: stavy → stavy */
int osmismerka[MAX_ROZMER][MAX_ROZMER];
int skrt[MAX_ROZMER][MAX_ROZMER];
int m, n;

void vytvor_automat (char *jmeno)
{
    FILE *fin;
    char word[80], p;
    int i, j, r, s, t, q;
    int queue[MAX_SLOVNIK];
    int qtail, qhead;

    fin=fopen (jmeno, "r");
    /* zalozime inicialni vrchol 0, fce g je nedifinovana pro kazde pismeno */
    q=1; o[0]=0;
    for (i=0; i<256; i++) g[0][i]=UNDEFINED;
    /* kazde prectene slovo vlozime do stromu automatu normalne i zrcadlove */
    while (fscanf (fin, "%s", word)==1)
      for (r=0; r<2; r++)                 /* kvuli zrcadleni ... */
      {
        /* nejdrive jdeme ve stromu tak dlouho dokud existuje cesta */
        s=0;
        for (j=0; j<strlen (word); j++)
            if (g[s][word[j]]!=UNDEFINED) s=g[s][word[j]]; else break;
        /* a pak pripojime vetev z novych stavu pro zbytek vkladaneho slova */
        while (j<strlen (word))
        {
            g[s][word[j]]=q;
            for (i=0; i<256; i++) g[q][i]=UNDEFINED;
            o[q]=0;
            s=q++; j++;
        }
        /* pro stav odpovidajici konci vkladaneho slova si pamatujme v o, ze zde
           konci slovo delky vkladaneho slova a pak prezrcadli */
        o[s]=strlen (word);
        for (j=0; j<strlen (word)/2; j++)
        {p=word[j]; word[j]=word[strlen (word)-j-1]; word[strlen (word)-j-1]=p;
        }
      }
    /* a dodefinujme korektne g pro inicialni stav 0 */
    for (i=0; i<256; i++) if (g[0][i]==UNDEFINED) g[0][i]=0;
    /* vytvorme frontu stavu a definujme f(0) */
    qtail=0; qhead=0;
```

```
    f[0]=0; o[0]=0;
    /* pro inicialni stav umistme potomky do fronty a fce f pro ne bude 0 */
    for (i=0; i<256; i++)
        if (g[0][i]>0)
          {
              f[g[0][i]]=0;
              queue[qtail++]=g[0][i];
          }
     /* vsem stavum ve fronte zpracuj jejich potomky a definuj jim o a f */
     while (qhead<qtail)
       {
          r=queue[qhead++];
          for (i=0; i<256; i++)
            if (g[r][i]!=UNDEFINED)
              {
                 s=g[r][i];
                 t=f[r];
                 while (g[t][i]==UNDEFINED) t=f[t];
                 f[s]=g[t][i];
                 o[s]=max (o[s], o[f[s]]);
                 queue[qtail++]=s;
              }
       }
     fclose (fin);
}
void vyhledej ( int x, int y, int dx, int dy, int delka)
{
    int mez, i, s=0;
    int intervaly[MAX_ROZMER];
    /* prochazime text a pro kazde pismeno si do pole intervaly ulozime, kolik pismen se
       ma pred nim skrtat */
    for (i=0; i<delka; i++)
    {
      while (g[s][osmismerka[x][y]]==UNDEFINED) s=f[s];
      s=g[s][osmismerka[x][y]];
      x+=dx; y+=dy;
      intervaly[i]=o[s];
    }
    /* pomoci pole intervaly vyskrtame z pole nalezena slova */
    mez=delka;
    for (i=delka-1; i>=0; i--)
    {
      x-=dx; y-=dy;
      mez=min (mez, i-intervaly[i]);
      if (i>mez) skrt[x][y]=1;
    }
}
void skrtej (void)
{
    int i, j;
```

```
    for (i=0; i<n; i++) vyhledej (i, 0, 0, 1, m);    /* skrtej sloupce zhora */
    for (i=0; i<m; i++) vyhledej (0, i, 1, 0, n);    /* skrtej radky zleva */
    /* a trocha cerne magie pro diagonaly */
    for (i=-m+1; i<n; i++) lookup (i>0 ?i:0, i>0 ?0: (-i), 1, 1,
                                   min (n- (i>0 ?i:0), m- (i>0 ?0: (-i))));
    for (i=0; i<m+n-1; i++) lookup (i<m ?0: (i-m-1), i<m ?i: (m-1), 1, -1,
                                   min (n- (i<m ?0: (i-m-1)), 1+ (i<m ?i:
                                       (m-1))));
    /* vypiseme co zbylo */
    for (i=0; i<m; i++)
      for (j=0; j<n; j++) if (skrt[j][i]==0) putc (osmismerka[j][i], stdout);
    puts ("");
}
void nacti_osmismerku (char *jmeno)
{
    FILE *fin;
    int i, j;
    char buf[80];

    fin=fopen (jmeno, "r");
    fscanf (fin, "%d%d", &m, &n);
    fgets (buf, 80, fin);
    for (i=0; i<m; i++)
      {
        for (j=0; j<n; j++) osmismerka[j][i]=fgetc (fin);
        fgets (buf, 80, fin);
      }
    fclose (fin);
    bzero (skrt, sizeof (skrt));                    /* nulovani ... */
}
int main (void)
{
    vytvor_automat ("slovnik.in");
    nacti_osmismerku ("osm.in");
    skrtej ();
    return 0;
}
```

Studijní text

Definice

Konečný automat budeme dále nazývat vyhledávací stroj. Naším cílem bude pro slovník K o celkové délce P písmen sestavit vyhledávací stroj v čase $O(P)$, který umožní vyhledání v textu délky n všech slov z K v čase $O(n)$. Vyhledávací stroj pro abecedu S, konečnou množinu K slov v S definujme jako čtveřici $M = (Q, g, f, out)$, kde

- $Q = 0, \ldots, q$ je konečná množina stavů,
- $g : Q \times S \longrightarrow Q \cup \{\perp\}$ je tzv. přechodová funkce, která každé dvojici $\langle stav, písmeno \rangle$ přiřadí buď nový stav, nebo symbol

„⊥“, který znamená „nedefinováno“, přičemž platí, že $g(0, s)$ je definováno pro všechna $s \in S$.

- $f : Q \longrightarrow Q$ je tzv. zpětná funkce, pro kterou platí $f(0) = 0$
- $out : Q \longrightarrow P(K)$ je výstupní funkce, která každému stavu přiřadí podmnožinu K.

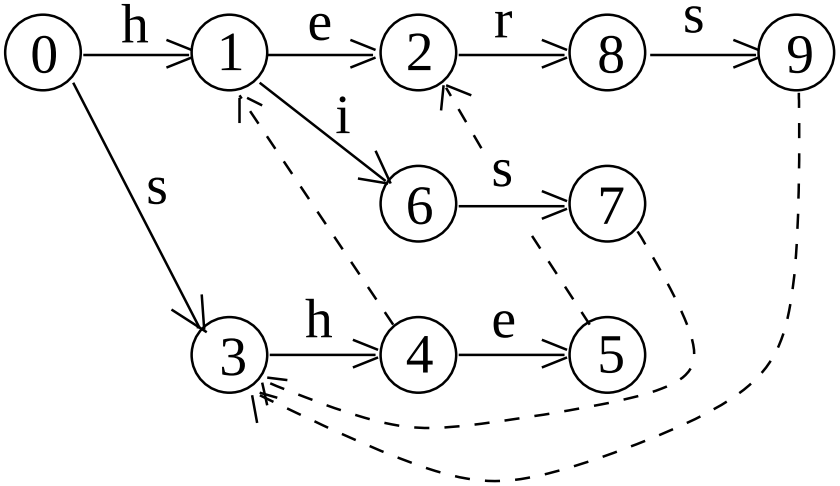
Činnost vyhledávacího stroje

Algoritmus 1 (interpret vyhledávacího stroje)

Vstup: text $x = x_1x_2x_3 \ldots x_n$, vyhledávací stroj $M = (Q, g, f, out)$ pro S a $K = y_1, y_2, \ldots y_m$
Výstup: posloupnost dvojic $\langle i, y_p \rangle$, kde $i = 1..n$, $p = 1..k$

```
begin
  state := 0;
  for i:= 1 to n do begin
    while  g(state,x[i]) nedefinován do state := f(state);
    state := g(state, x[i]);
    for y in out(state) do Report(i,y)
  end;
end.
```

Příklad: S je latinská abeceda, $K = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$. Na následujícím obrázku je vyznačena stavová množina vyhledávacího stroje spolu se všemi přechody prostřednictvím fce g (plné šipky) i prostřednictvím funkce f (přerušované šipky), pro něž cílovým stavem není stav 0 (pro tento stav dodefinujeme $g(0, x_i) = 0$ pro všechna x_i různá od `h` a `s` a $f(0) = 0$). Funkce *out* je zadána tabulkou.



Stav	*out*
0	{}
1	{}
2	{`he`}
3	{}
4	{}
5	{`she`,`he`}
6	{}
7	{`his`}
8	{}
9	{`hers`}

Při reprezentaci stavu výpočtu dvojicí $\langle i, s\rangle$, kde i je délka dosud přečteného úseku textu a s je okamžitý stav, bude historie zpracování textu $x =$ `ushers` vypadat takto: $\langle 0,0\rangle$, $\langle 1,0\rangle$, $\langle 2,3\rangle$, $\langle 3,4\rangle$, $\langle 4,5\rangle$ {vydá `she`,`he`}, $\langle 4,2\rangle$, $\langle 5,8\rangle$, $\langle 6,9\rangle$ {vydá `hers`}.

Vyhledávací stroj tedy pracuje tak, že sleduje shodu prohledávaného textu s co nejdelším vyhledávaným vzorkem a v okamžiku, kdy narazí na jeho konec nebo kdy dojde k neshodě, použije „záchrannou funkci“ f k tomu, aby nalezl stav, ze kterého již bude moci pokračovat. Okamžitý stav reprezentuje veškerou informaci, kterou vyhledávací stroj získal z dosud přečteného úseku textu a který je relevantní pro hledání výskytu vzorků. Při pohybu mezi stavy jsou v každém stavu hlášeny vzorky s tímto stavem spojené. Dále se samozřejmě budeme zajímat o korektní vyhledávací stroje, t.j. o ty, které pro každé $x = x_1x_2x_3\ldots x_n$, vydají v i-té iteraci hlavního cyklu interpretu (t.j. po přečtení předpony $x_1x_2x_3\ldots x_i$ prohledávaného textu) právě ty dvojice $\langle i, y_p\rangle$, kde y_p je příponou $x_1x_2x_3\ldots x_i$.
Správná funkce vyhledávacího stroje plyne z následujících faktů:

1) Každé vyhledávané slovo je v automatu reprezentováno nějakou cestou z počátečního stavu. Všimněte si, že stavy stroje spolu s přechodovou funkcí g tvoří strom, a tedy cesty se nemohou nijak spojovat.
2) Každá cesta z počátečního stavu je počátkem nějakého hledaného slova.
3) Zpětná funkce vede do stavu jehož odpovídající řetězec je zakončením řetězce stavu, ze kterého funkce vede. Navíc skáčeme do takového stavu, že jeho řetězec je nejdelší možný.
4) Vždy když je nějaké hledané slovo zakončením řetězce nějakého stavu, je toto slovo zahrnuto ve výstupní funkci pro daný stav.

Pokud máte zájem o poněkud úplnější a formálnější zdůvodnění funkce automatu, můžete si jej vyhledat ve výše uvedené knize.

Časová složitost interpretace tohoto automatu bude zřejmě $O(n)$. V každém kroku se totiž posuneme buď o stav dál, nebo se budeme vracet. Pokud se vracíme, museli jsme někdy předtím postupovat kupředu a to alespoň tolikrát, kolikrát se teď vracíme. Tedy celkový počet návratů je menší nebo roven počtu postupů kupředu, a tedy návraty nám nemohou zhoršit časovou složitost.

Konstrukce vyhledávacích strojů

Při konstrukci vyhledávacího stroje k dané množině slov K se přidržíme obrázku popisujícímu fci g vyhledávacího stroje pro slova {`hers`, `she`, `he`, `his`}. Nejprve algoritmem 2 sestrojíme stavovou množinu $Q = 0, 1, \ldots, q$, přechodovou funkci g a „polotovar“ o výstupní funkce *out*. Poté algoritmem 3 sestrojíme zpětnou funkci f a rozšíříme funkci o na výstupní funkci *out*.

Algoritmus 2

Vstup: Množina vzorků $K = \{y_1, y_2, \ldots, y_k\}$ neprázdných slov v S
Výstup: $Q = \{0, \ldots, q\}$, $g : Q \times S \longrightarrow Q \cup \{\perp\}$, $o : Q \longrightarrow P(K)$
Metoda: Založíme stavový strom s kořenem 0 a postupně k němu budeme připojovat cesty odpovídající jednotlivým prvkům K tak, aby Q a g měly požadované vlastnosti.

```
var q: integer;
procedure Enter (x[m]$);
{procedura pro připojení slova y = x[1],x[2],x[3]...x[m]}
begin
  s := 0; j := 1;
  while  j < m  and  g(s, x[j]) definováno begin
    s := g(s, x[j]);
    j := j + 1;
  end;
  while  j < m do begin
    q := q + 1;              {přidej nový stav}
    g(s, x[j]) := q;         {rozšiř definici}
    s := q;                  {pokroč do nového stavu}
    j := j + 1;              {pokroč k dalšímu písmeni}
  end;
  o(s) := x; {koncovému stavu s dáme k výstupu příslušné slovo}
end;

begin                                {tělo algoritmu}
  q := 0;                            {založ strom}
  for p := 1 to k do Enter(y[k]);    {připoj větve}
  for all x in S do                  {dodefinuj přechody z kořene}
```

```
    if g(0,x) nedefinováno then g(0,x) := 0;
end.
```

Není těžké se přesvědčit, že Algoritmus 2 pracuje v čase $O(P)$ a jeho výstupy mají vlastnosti popsané ve zdůvodnění správnosti.

Algoritmus 3

Vstup: $Q = 0, \ldots, q$, $g : Q \times S \longrightarrow Q \cup \{\perp\}$, $o : Q \longrightarrow P(K)$
Výstup: $f : Q \longrightarrow Q$, $out : Q \longrightarrow P(K)$
Metoda: Začneme od kořene stavového stromu a postupně probíráme stavy v pořadí rostoucí hloubky. Každý stav, ve kterém definujeme funkce f a out, zařadíme do fronty, v níž si pamatujeme stavy, jejichž potomci ještě nemají tyto funkce definovány. Algoritmus končí vyčerpáním všech stavů.

```
begin
  vytvoř prázdnou frontu queue;
  definuj f(0) = 0; definuj out(0) = []; {prázdná množina}
  for all x in S begin                   {zpracuj potomky 0}
    s := g(0,x);
    if s <> 0 then begin
      definuj f(s) := 0;
      definuj out(s) := o(s);
      zařaď s na konec queue
    end;
  end;
  while queue neprázdná do begin
    r := první prvek queue; vyřaď r z queue;
    for all x in S do begin              {zpracuj potomky s}
      if g(r,x) definováno then begin
        s := g(r,x);                     {potomek}
        t := f(r);                       {má definováno f}
        while g(t,x) nedefinováno do t:= f(t);
        definuj f(s) := g(t,x);
        definuj out(s) := o(s) + out(f(s));
        zařaď s na konec queue;
      end;
    end;
  end;
end.
```

I zde se můžeme lehce přesvědčit, že Algoritmus 3 pracuje v čase $O(P)$ a jeho výstupy tvoří vyhledávací stroj korektně vzhledem k množině vzorků K.

Celková časová a paměťová složitost konstrukce vyhledávacího stroje algoritmem Aho-Corasickové je $O(P)$, přičemž na reprezentaci vyhledávacího stroj potřebujeme paměť o velikosti $O(P)$.

12-2-2 Šnečí maraton — Robert Špalek

Ke zjištění celkového pořadí budeme postupně umísťovat šneky podle rostoucích pořadových čísel. Pak se každý šnek umísťuje až po umístění všech šneků s menším pořadovým číslem a mám tedy jednoznačně danou jeho pozici vůči těmto šnekům.

První šnek nemohl zřejmě nikoho s menším pořadovým číslem předběhnout, zařadíme ho tedy na první místo. Obecně předběhnul-li šnek s pořadovým číslem k celkem s_k šneků s menším pořadovým číslem, vložíme ho na $k - s_k$-té místo v rekonstruovaném pořadí. Umísťování dalších šneků s vyššími pořadovými čísly doprostřed tohoto pořadí už neovlivní vzájemné pořadí již umístěných šneků podle zadání úlohy. Toto pořadí je tedy korektní a jediné možné.

Nyní už zbývá jediná otázka: volba použité datové struktury pro průběžné ukládání rekonstruovaného pořadí šneků. Máme několik možností:

1) pole – pak ovšem musíme při vložení každého šneka doprostřed posunout všechny další šneky o 1 místo doprava, což zabere $O(n)$ času,
2) spojový seznam – pak snadno vložíme šneka doprostřed seznamu, zato nám však bude dlouho trvat nalezení příslušné pozice, zabere nám to také $O(n)$ času,
3) binární strom – pokud bude tento strom vyvážený, zabere nám jak vyhledání příslušné pozice, tak vložení šneka $O(\log n)$ času.

Pořadí tedy budeme reprezentovat binárním stromem. V každém uzlu `u` budou uloženy tyto informace: číslo šneka `u->cislo`, počet šneků v levém podstromu `u->vlevo` a ukazatele na levého a pravého syna `u->l`, `u->p`.

Při umísťování k-tého šneka nejprve vypočítáme jeho průběžné pořadí $p = k - s_k$. Dokud nenalezneme list stromu, provádíme tyto operace:

1) Je-li $p \leq$ `u->vlevo`, musíme šneka umístit do levého podstromu. Má-li `u` levého syna, posuneme se na něj (`u = u->l`). V opačném případě jsme nalezli list stromu a končíme hledání.
2) Je-li $p >$ `u->vlevo`, musíme šneka umístit do pravého podstromu. Předtím však musíme odečíst počet vrcholů v levém podstromu a ještě jedničku za aktuální uzel, neboť tyto šneky jsme právě skokem do pravého podstromu přeskočili. Přiřadíme tedy `p -= u->vlevo+1`. Má-li `u` pravého syna, posuneme se na něj (`u = u->p`), jinak končíme hledání.

Nyní známe budoucího otce vytvářeného uzlu. Vytvářený uzel bude listem našeho stromu, vynulujeme tedy ukazatele na oba syny a počet vrcholů v levém podstromu. Uložíme do něj číslo ukládaného šneka a připojíme jej k otci. Zbývá nám jen aktualizovat počty vrcholů `u->vlevo` ve všech uzlech od našeho listu až ke kořenu stromu. Pokaždé, když jsme se při hledání pozice přesunuli na levého syna, vložili jsme 1 uzel navíc do jeho levého podstromu, takže ve všech takových vrcholech po cestě ke kořenu inkrementujeme `u->vlevo`.

Oba průchody (dolů při hledání pozice a nahoru při aktualizaci `u->vlevo`) budeme implementovat jednou rekurzivní procedurou `vloz_sneka`.

Pozn. pro puntičkáře: Tohle všechno nám ještě nezaručuje logaritmickou nářočnost operací hledání a vkládání. Pokud bude pořadí šneků nešťastně zvolené, pak nám strom může degenerovat až na spojový seznam – např. tehdy, když bude vstupní pořadí šneků již setříděné a my budeme muset vkládat uzly vždy jen do levého resp. pravého podstromu. Pro tyto účely byly navrženy různé modifikace binárních stromů, které strom různými způsoby vyvažují – např. AVL stromy nebo 2–3 stromy. Bohužel implementace operací INSERT a FIND je u nich o něco složitější a je nad rámec tohoto příkladu. Volba použité datové struktury nicméně nic nemění na charakteru řešení, takže budeme tento problém ignorovat. Stejně je v průměrném případě časová složitost logaritmická i u obyčejných binárních stromů, takže toho moc nezkazíme.

Časová náročnost algoritmu je tedy $O(n \log n)$, paměťová $O(n)$.

```
/* Šnečí maraton */
#include <stdio.h>
#include <stdlib.h>

struct uzel {
    int cislo;
    int vlevo;
    struct uzel *l, *p;
};
/* Do rekurzivní procedury předáváme ukazatel na ukazatel, protože nám to umožní snadno
     připojit nový list k otci, ale také nám to zjednoduší vytváření vůbec prvního uzlu,
     který nebude mít otce. */
void vloz_sneka (struct uzel **otec, struct uzel *koho, int poradi)
{
    struct uzel *kam=*otec;                    /* Aktuální uzel. */

    if (!kam) {                                /* Jsme v listu stromu? */
      *otec=koho;                              /* Připojíme nový vrchol k otci. */
    } else {
      if (poradi<=kam->vlevo) {                /* Vkládáme do levého podstromu. */
        vloz_sneka (&kam->l, koho, poradi);
        kam->vlevo++;                          /* Inkrementujeme počet vrcholů v levém
                                                    podstromu. */
      } else {
        vloz_sneka (&kam->p, koho, poradi-kam->vlevo-1);
      }
```

```
    }
}
void vypis_sneky (struct uzel *snek)
{
    if (!snek)
        return;
    vypis_sneky (snek->l);
    printf ("%d␣", snek->cislo);
    vypis_sneky (snek->p);
}
void dealokuj_sneky (struct uzel *snek)
{
    if (!snek)
        return;
    dealokuj_sneky (snek->l);
    dealokuj_sneky (snek->p);
    free (snek);
}
int main (void)
{
    int i, pocet;
    struct uzel *koren=NULL;

    printf ("Zadej počet šneků:␣");
    scanf ("%d", &pocet);
    for (i=0; i<pocet; i++){
        int kolik;
        struct uzel *snek;

        printf ("Kolik šneků s menším poř. číslem předběhl šnek č. %d?␣", i+1);
        scanf ("%d", &kolik);
        snek = calloc (1, sizeof (struct uzel));
        snek->cislo = i+1;
        vloz_sneka (&koren, snek, i-kolik);
    }
    printf ("Pořadí šneků:␣");
    vypis_sneky (koren);
    printf ("\n");
    dealokuj_sneky (koren);
    return 0;
}
```

12-2-3 Jednosměrky — Jan Kára

Úlohu si nejdříve (víceméně z cvičných důvodů) přeformulujeme na problém z teorie grafů. Je dáno N vrcholů (v zadání křižovatek) a hrany mezi nimi (v zadání cesty mezi křižovatkami). Úkolem je hrany zorientovat tak, aby se šlo po zorientování dostat z každého vrcholu do každého (říkáme, že graf má být silně souvislý).

A nyní jak tedy danou orientaci hran nalézt (případně zjistit, že taková orientace neexistuje). Graf budeme z libovolného (třeba prvního) vrcholu prohledávat do hloubky (vezmeme vrchol a na každého souseda, ve kterém jsme dosud nebyli se rekurzivně zavoláme. Když jsme takto zpracovali všechny sousedy, vrátíme se). Hrany budeme orientovat ve stejném směru, v jakém jsme přes ně přešli (případně se přes ně dívali na sousední vrcholy).

Navíc si u každého vrcholu budeme pamatovat, jako kolikátý jsme ho navštívili. Při prohledávání si pak budeme pamatovat nejnižší pořadí vrcholu, do kterého jsme se pokoušeli přejít (ale nepřešli jsme, protože jsme v něm už byli). Vždy když se vracíme z nějakého vrcholu, tak zkontrolujeme, jestli je nejnižší nalezené pořadí menší, než pořadí vrcholu, ze kterého se vracíme. Pokud ne, tak orientace hran nemůže existovat. To plyne z následujícího: Protože nejnižší nalezené pořadí je větší než nebo stejné jako pořadí vrcholu, ze kterého se vracíme, nemůže existovat hrana z oblasti „pod“ tímto vrcholem do oblasti „nad“ tímto vrcholem (když se vracíme, prošli jsme již celou dosažitelnou oblast a vyzkoušeli všechny hrany z ní). Jediná hrana propojující tyto oblasti je tedy ta, po které se nyní vracíme. Ať ji ale zorientujeme jakkoliv, vždy se nepůjde dostat buď z oblasti „nad“ do oblasti „pod“, nebo opačně.

Když prohledávání skončí, otestujeme, zda jsme se dostali do všech vrcholů. Pokud ne, graf nebyl ani souvislý a síť tedy zřejmě neexistuje. Pokud ano, námi přiřazená orientace bude hledaná orientace. Proč? Protože pokud mezi dvěma vrcholy vedla hrana, tak se mezi nimi půjde dostat i po orientaci — vždy když jsme se z nějakého vrcholu po nějaké hraně vraceli, tak jsme zkontrolovali, že se z oblasti „pod“ tímto vrcholem dostaneme do oblasti „nad“, a tedy speciálně že se z tohoto vrcholu dokážeme dostat do vrcholu, do kterého se nyní vracíme proti směru hrany. Jedním směrem se tedy půjde dostat po hraně, druhým směrem po nějaké cestě. Protože původní graf byl souvislý, existovala mezi každými dvěma vrcholy cesta — posloupnost hran. Protože každé dva vrcholy spojené hranou v původním grafu jsou spojené nějakou orientovanou cestou v našem zorientovaném grafu, můžeme orientovanou cestu mezi libovolnými dvěma vrcholy získat jako spojení cest odpovídajících jednotlivým hranám na neorientované cestě (pokud bychom chtěli být korektní, je třeba říci, že spojením nezískáme cestu, ale sled — posloupnost hran, kde se mohou jednotlivé hrany opakovat. Z toho lze ale cesta snadno vytvořit tak, že vypustíme opakující se hrany).

Algoritmus má časovou složitost $O(M + N)$, kde M je počet hran a N je počet vrcholů. Program je přímou implementací algoritmu.

```
#include <stdio.h>

#define MAXKRIZ 100

struct kriz {
   int cn;                                  /* Počet cest z dané křižovatky  */
```

```
    int byl;                                  /* Byli jsme již v dané křižovatce? */
    int h;                                    /* Hodnota přidělená křižovatce při
                                                 prohledávání do hloubky */
    int s[MAXKRIZ];                           /* Čísla křižovatek, se kterými je tato
                                                 spojena */
};
int n, m;                                     /* Počet křižovatek; Počet silnic */
struct kriz k[MAXKRIZ];                       /* Jednotlivé křižovatky */
/* Načte vstup */
void nacti (void)
{
    int i;
    int a, b;

    printf ("Pocet krizovatek:␣");
    scanf ("%d", &n);
    printf ("Pocet silnic:␣");
    scanf ("%d", &m);
    for (i = 0; i < n; i++) {
        k[i].cn = 0;
        k[i].byl = 0;
    }
    for (i = 0; i < m; i++) {
        printf ("Zadejte cestu:␣");
        scanf ("%d %d", &a, &b);
        a--; b--;
        /* Přidáme silnice do seznamu */
        k[a].s[k[a].cn++] = b;
        k[b].s[k[b].cn++] = a;
    }
}
/* Vypíše síť jednosměrek */
void vypis (void)
{
    int i, j;

    for (i = 0; i < n; i++)                   /* Pro všechny křižovatky */
        for (j = 0; j < k[i].cn; j++)         /* Pro všechny silnice z ní */
            if (k[i].h < k[k[i].s[j]].h)      /* Abychom silnici vypsali jen jednou... */
                /* Šli jsme po této silnici při prohledávání přímo? */
                if (k[i].h + 1 == k[k[i].s[j]].h)
                    printf ("%d -> %d\n", i+1, k[i].s[j]+1);
                else
                    printf ("%d -> %d\n", k[i].s[j]+1, i+1);
}
/* Vrátí, zda jsme byli všude */
int vsude (void)
{
    int i;

    for (i = 0; i < n; i++)
        if (!k[i].byl)
            return 0;
```

```
    return 1;
}
/* Prohledávání do hloubky – z vrcholu V, předchozí vrchol byl PREV, prohledali jsme již
     H vrcholů */
int prohledej (int v, int prev, int h)
{
    int i, minh = h, acth;                    /* ; Minimální dosud vrácená hodnota;
                                                 Aktuálně vrácená hodnota */
    k[v].byl = 1;
    k[v].h = h;
    for (i = 0; i < k[v].cn; i++) {           /* Projdeme všechny sousedy */
      if (k[k[v].s[i]].byl) {                 /* Byli jsme už v daném vrcholu? */
        /* Dostali jsme se výše (a nevraceli jsme se)? */
        if (k[k[v].s[i]].h < minh && k[v].s[i] != prev)
          minh = k[k[v].s[i]].h;              /* Zapamatujeme si lepší hodnotu */
      }
      else {
        /* Necháme prohledat souseda */
        acth = prohledej (k[v].s[i], v, h + 1);
        if (acth == -1)                       /* Není už nějaká větev problematická? */
          return -1;
        if (acth < minh)
          minh = acth;
      }
    }
    if (minh >= h)                            /* Nevedla žádná cesta nad nás? */
      return -1;
    return minh;
}
int main (void)
{
    nacti ();
    if (!k[0].cn && n != 1)                   /* Izolovaná křižovatka ve větší síti? */
      puts ("System jednosmerek neexistuje.");
    else {
      /* V kořeni prohledávání jsme už byli... */
      k[0].byl = 1;
      k[0].h = 0;
      /* Byl problém někde níže, nebo je síť nesouvislá? */
      if (prohledej (k[0].s[0], 0, 1) == -1 || !vsude ())
        puts ("System jednosmerek neexistuje.");
      else
        vypis ();
    }
    return 0;
}
```

12-2-4 Vysílače — Zdeněk Dvořák

Drtivá většina řešitelů užila triviální algoritmus – probrat všechny dvojice vysílačů. Pochopitelně se v žádném z těchto řešení nevyskytly vážnější chyby; asi polovina z řešitelů si uvědomila, že při porovnávání vzdáleností dvojic se stačí zabývat jejich druhými mocninami, což vede k jistému zrychlení, i když časová složitost je pořád $O(n^2)$.

K řešení použijeme metodu Rozděl a panuj. Zadané body si nejprve setřídíme podle souřadnice x.

Nyní je rozdělíme přímkou rovnoběžnou s osou y na dvě stejně velké části. V obou nalezneme stejným algoritmem nejbližší body, jejich vzdálenosti si označme d_1, d_2. Pokud by se nejbližší body nacházely oba ve stejné polovině, jsme hotovi; zbývá tedy možnost, že jeden z nich leží na jedné straně dělící přímky a druhý na druhé. Nechť $d = \min(d_1, d_2)$. Zřejmě stačí vyšetřit body, jejichž vzdálenost od dělící přímky je nejvýše d. Navíc nás pro každý bod zajímají pouze body, jež jsou na ose y vzdáleny nejvýše d. Tedy pro daný bod z jedné poloviny musíme určit vzdálenost k bodům v oblasti o rozměrech $d \times 2d$. Rozdělíme si tento obdélník na šest menších o rozměrech $\frac{d}{2} \times \frac{2d}{3}$. V každém z těchto obdélníčků se může vyskytovat nejvýše jeden bod, protože největší vzdálenost, jakou by v nich dva body mohly mít, je

$$\sqrt{\frac{d^2}{4} + \frac{4d^2}{9}} = \frac{5}{6}d$$

což je menší než d. Tedy těchto bodů je nejvýše šest. Abychom dosáhli skutečně této konstantní časové složitosti na jeden bod, setřídíme si body v obou polovinách podle y souřadnice (provádíme to průběžně MergeSortem) a zatímco body v jedné polovině procházíme po řadě, ve druhé si pro každý z nich nalezneme první spadající do vyšetřované oblasti nebo za ni (jestliže si pamatujeme, kde to bylo pro minulý bod, nezabere nám to dohromady pro všechny body více než $O(n)$ porovnání) a určujeme vzdálenost k následujícím bodům, dokud se nacházejí v ní.

Správnost algoritmu je zřejmá z popisu; konečnost plyne z toho, že v každém kroku se nám velikost vyšetřované množiny bodů zmenší alespoň o 1. Složitost (n je počet vysílačů):

- Čas: V každém kroku potřebujeme kromě rekurzivního vyvolání sebe sama lineární čas, tedy $t(n) = k \cdot n + 2t(n/2) = k \cdot n + 2k \cdot n/2 + 4t(n/4) = \ldots = k \cdot n \log n + nt(1)$, tedy $O(n \log n)$.
- Paměť: Kromě rekurze logaritmické hloubky užíváme pouze dvě pole lineární velikosti, tedy $O(n)$.

#include <stdio.h>

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

#define maxp 100
#define maxfol 5

typedef struct
{
    float x, y;
    int id;
} bod;

#define sqr (X)  ( (X)* (X))
#define dist2 (X, Y)  (sqr ( (X).x- (Y).x)+sqr ( (X).y- (Y).y))
                                                    /* Vzdálenost bodů */

bod temp[maxp+1], body[maxp];                       /* ; Načtené body */
/* Pomocná makra, abychom mohli třídit podle X nebo Y */
#define dMerge (x)\
void Merge##x (bod co[], int od, int s, int k)\
{\
    int i1=od, i2=s, kam=od; \
    while (kam<k) {\
       while (i1<s&& (i2==k| |co[i1].x<co[i2].x)) temp[kam++]=co[i1++]; \
       while (i2<k&& (i1==s| |co[i1].x>=co[i2].x)) temp[kam++]=co[i2++]; \
    }\
    for (i1=od; i1<k; i1++) co[i1]=temp[i1]; \
}

dMerge (x)
dMerge (y)
/* Třídění MergeSortem */
void MergeSort (bod co[], int od, int k)
{
    if (od+1<k) {
       int div= (od+k)/2;
       MergeSort (co, od, div);
       MergeSort (co, div, k);
       Mergex (co, od, div, k);
    }
}

float mdist2=FLT_MAX;
int p1, p2;
/* Nalezne nejbližší dva body */
void GetBest (bod co[], int od, int k)
{
    if (od+1 < k) {
       int div= (od+k)/2;
       float dist=sqrt (mdist2);
       float delx=co[div].x;
       float alen;
       int i, lleft, lright, ltested, atested;
       /* Nalezneme body v jednotlivých polovinách */
       GetBest (co, od, div);
```

```
        GetBest (co, div, k);
        /* Vybereme body, se kterými má smysl se zabývat */
        lleft=od; lright=div;
        for (i=od; i<div; i++) if (fabs (co[i].x-delx)<dist) temp[lleft++]=co[i];
        for (; i<k; i++) if (fabs (co[i].x-delx)<dist) temp[lright++]=co[i];
        ltested=div;
        temp[lright].y=1e30;
        /* Procházíme po řadě body na levé straně */
        for (i=od; i<lleft; i++) {
          /* Nalezneme dostatečně blízký bod na pravé straně */
          while (temp[ltested].y<temp[i].y-dist) ltested++;
          if (ltested==lright) break;              /* Body došly? */
          /* Otestujeme, zda nějaký z bodů není blíže */
          for (atested=ltested; temp[atested].y<temp[i].y+dist; atested++)
            if ( (alen=dist2 (temp[i], temp[atested]))<mdist2) {
              p1=temp[i].id; p2=temp[atested].id; mdist2=alen;
            }
        }
        /* Sloučíme seznamy setříděné dle Y */
        Mergey (co, od, div, k);
    }
}
int main (void)
{
    int i, n;
    /* Načteme vstup */
    printf ("Pocet vysilacu:␣");
    scanf ("%d", &n);
    for (i=0; i<n; i++) {
      printf ("Souradnice vysilace cislo %d:␣", i+1);
      scanf ("%f %f", &body[i].x, &body[i].y);
      body[i].id=i+1;
    }
    MergeSort (body, 0, n);
    GetBest (body, 0, n);
    printf ("Nejblize jsou vysilace %d a %d, vzdalenost %f.\n", p1, p2, sqrt (mdist2));
    return 0;
}
```

12-2-5 PRAM — Daniel Kráľ

Nejprve zkusme vyřešit zadanou úlohu pomocí pouze jednoho procesoru. Použijeme klasický školní algoritmus pro sčítání (samozřejmě upravený pro binární čísla) – sečteme dvě cifry nejnižšího řádu, pokud je součet větší než jedna, odečteme dvojku a „zapamatujeme“ si jedničku. Poté sečteme dvě předposlední cifry, pokud si „pamatujeme“ jedničku, zvětšíme součet o jedna, a použijeme stejný postup. Časová složitost řešení je lineární s délkou čísel (délku sčítaných čísel dále označujme n). Většina došlých řešení měla taktéž časovou složitost

lineární, ale používala n procesorů, a tedy byla horší než řešení neparalelní. Byla dle toho i patřičně obodována. Obecně platí: Jednoprocesorové stejně rychlé řešení je lepší než řešení s více procesory.

Z algoritmu pro jeden procesor je vidět, že kdybychom již měli spočítané přenosy pro všechny řády, pak bychom dokázali vypočítat součet na libovolném typu PRAMu v konstantním čase. Zkusme si tedy rozmyslet, jak lze rychle spočítat všechny přenosy. Označme a_i cifry prvního čísla (a_n je cifra nejnižšího řádu, jedniček), b_i cifry druhého čísla a c_i přenos mezi i-tým a $(i+1)$-vým řádem; i-tá cifra výsledku je tedy $(a_i + b_i + c_i) \bmod 2$. Zkusme zformulovat jasněji pravidla pro výpočet c_i:

- Pokud je a_i i b_i nula je c_{i-1} nula.
- Pokud je a_i i b_i jedna je c_{i-1} jedna.
- Pokud je jedno z a_i a b_i nula a jedno jedna je c_{i-1} rovno c_i, pro $i = n$ je c_i nula.

Tyto podmínky lze rovněž přeformulovat následovně: c_i je jedna právě tehdy, pokud mezi daným řádem a nejbližší dvojicí jedniček na nižším řádu není žádná dvojice nul; formálně: právě tehdy, pokud existuje $j > i$ takové, že $a_j = b_j = 1$ a pro všechna $i < k < j$ je právě jedno z a_k a b_k rovno jedné. To je návod na algoritmus pro výpočet přenosu v konstantním čase s $O(n^2)$ procesory. Procesorům budou přiřazeny dvojice čísel $[k, l]$ $(0 \leq k < l \leq n)$, procesory s větším l budou mít nižší prioritu (větší číslo). Procesor prohlásí c_k rovno nule, pokud $a_l = b_l = 0$ a rovno jedné pokud $a_l = b_l = 1$. Správnost algoritmu plyne ihned z naší reformulace podmínky pro výpočet přenosů. Časová složitost je konstantní, paměťová je $O(n^2)$ a počet použitých procesorů je $O(n^2)$.

Při vytváření verze pro CREW a EREW použijeme osvědčený trik s rozkopírováváním jedné hodnoty na n kopií v čase $O(\log n)$. Nejprve namnožíme `gmem[0]` všem procesorům (teď je jich sice kvadraticky mnoho, ale $\log n^2 = 2 \log n = O(\log n)$). Pak procesory spočtou hodnoty -1,0,1 (obsah `lmem[4]` v programu pro CRCW), které určují, zda přenos přes danou cifru „prochází“, „zastaví se“ nebo zde „vznikne“. Tyto hodnoty se spočtou a rozkopírují do n identických polí délky $n + 1$. Následně se určí pro všechna $0 \leq i \leq n$ první hodnota v této posloupnosti za pozicí i různá od -1; ta určuje přenos na pozici i. Vypočítat součet je již pak triviální. Časová složitost takto navrženého algoritmu je $O(\log n)$, paměťová složitost je $O(n^2)$ a počet použitých procesorů je $O(n^2)$. Poznamenejme, že opatrnějším výpočtem přenosů lze docílit paměťové složitosti $O(n)$ s $O(n)$ použitými procesory. Napsat program je nyní již (nudnou) technickou záležitostí. Rozkopírovávací části jsme nahradili komentářem; každý si je jistě zvládne v případě potřeby doplnit sám (viz např. vzorové řešení úlohy z první série).

Program pro jeden procesor:

```
   lmem[0]:=0                        ; přenos (pamatování jedničky)
   lmem[1]:=gmem[0]                  ; pozice v prvním čísle
   lmem[2]:=2*gmem[0]                ; pozice v druhém čísle
1: if lmem[1]=0 then goto 2          ; jsme na konci čísla?
   lmem[0]:=lmem[0]+gmem[lmem[1]]
   lmem[0]:=lmem[0]+gmem[lmem[2]]    ; přičteme obě cifry k přenosu
   gmem[lmem[1]]:=lmem[0]%2          ; vypočítáme cifru výsledku
   lmem[0]:=lmem[0]/2                ; vypočítáme přenos
   lmem[1]:=lmem[1]-1
   lmem[2]:=lmem[2]-1                ; posuneme se
   goto 1
2: gmem[0]:=lmem[0]
   halt
```

Program pro CRCW PRAM:

```
   lmem[0]:=cpuid-1
   lmem[0]:=lmem[0]/gmem[0]        ; vypočteme "k"
   lmem[1]:=cpuid-1
   lmem[1]:=lmem[1]%gmem[0]
   lmem[1]:=1+lmem[1]              ; vypočteme "l"
   if lmem[0]>=lmem[1] then halt   ; pokračujeme pouze pokud k<l
   lmem[2]:=gmem[0]+lmem[1]        ; l+n, pozice druhé cifry
   lmem[3]:=gmem[0]+lmem[0]
   lmem[3]:=gmem[0]+lmem[3]        ; registr pro přenos c[k]
   lmem[4]:=gmem[lmem[1]]+gmem[lmem[2]]
   if lmem[4]=1 then lmem[4]:=-1   ; pokud jsou cifry různé
                                     nepřiřazujeme nic
   if lmem[4]=2 then lmem[4]:=1    ; pokud jsou obě jedna,
                                     bude přenos jedna
   lmem[3]:=lmem[3]+2
   gmem[lmem[3]]:=0                ; je třeba znulovat i c[n],
                                     které se vlastně nepočítá
   lmem[3]:=lmem[3]-1
   gmem[lmem[3]]:=0
   if lmem[4]>=0 then gmem[lmem[3]]:=lmem[4]
   if lmem[0]<>0 then halt         ; pokračuje jen O(n) procesorů
   gmem[lmem[1]]:=gmem[lmem[1]]+gmem[lmem[2]];
   lmem[2]:=gmem[0]+lmem[2]        ; přičetli jsme druhou cifru
                                     z dvojice
   lmem[2]:=lmem[2]+1
   gmem[lmem[1]]:=gmem[lmem[1]]+gmem[lmem[2]];
   gmem[lmem[1]]:=gmem[lmem[1]]%2 ; přičetli přenos a vypočetli
```

```
                                  zbytek
  if lmem[1]<>1 then halt        ; dál běží jeden procesor,
                                   který spočte gmem[0]
  lmem[2]:=lmem[2]-1
  gmem[0]:=gmem[lmem[2]]         ; přiřadíme prostě přenos
  halt
```

Kostra programu pro EREW PRAM:

```
  lmem[1]:=gmem[cpuid]           ; záložní kopie globální paměti
 { Rozkopírování hodnoty gmem[0], procesory si ji
   uloží do lmem[0] }
  lmem[0]:=lmem[0]+1             ; zvýšíme o jedna
  lmem[2]:=cpuid-1
  lmem[2]:=lmem[2]%lmem[0]
  lmem[2]:=lmem[2]               ; hodnota od 0 do n
  lmem[3]:=cpuid-1
  lmem[3]:=lmem[3]/lmem[0]       ; hodnota od 0 do n
  ; všech (n+1)^2 možných dvojic hodnot přiřazeno
  lmem[4]:=cpuid-lmem[0]
  if lmem[4]>=0 then gmem[lmem[4]]:=lmem[1]
  ; přičteme druhou cifru
  if lmem[3]=0 then lmem[1]:=lmem[1]+gmem[lmem[2]]
  lmem[4]:=cpuid-lmem[0]
  if lmem[4]=0 then lmem[1]:=0   ; přenos $c_n$ je nula
  if lmem[3]=0 then gmem[lmem[2]]:=lmem[1]
 { Rozkopírování hodnot v buňkách gmem[0] až gmem[n] tak,
   že procesor bude znát gmem[lmem[2]] }
  if lmem[3]>lmem[2] then halt   ; tyto procesory nejsou třeba
 { Procesory se stejným lmem[3] budou pracovat na určení prvního
   nejedničkového členu posloupnosti gmem[lmem[3]],...,gmem[n];
   zda je nula nebo dva se uloží do gmem[lmem[3]] }
  if lmem[3]>0 then halt;        ; tyto procesory nebudou třeba
                                 ; nyní platí lmem[2]=cpuid-1
  lmem[4]:=gmem[lmem[2]]         ; máme spočítány přenosy
  gmem[cpuid]:=lmem[1]
  lmem[1]:=gmem[lmem[2]]         ; součet odpovídajících cifer
  if lmem[2]=0 then lmem[1]:=0   ; pozor na začátek, tam je
                                   součet cifer nula
  lmem[4]:=lmem[4]/2             ; přenos (2/2=1, 0/2=0)
  lmem[1]:=lmem[1]+lmem[4]       ; přičteme přenos k součtu cifer
  gmem[lmem[2]]:=lmem[4]%2       ; zapíšeme výsledek
  halt
```

12-3-1 Strýček skrblík — Pavel Nejedlý

Došlá řešení byla většinou správně, ba co víc, dokonce asymptoticky optimální. Jen nekteří řešitelé potřebují data nejprve vložit do pole a pak je sekvenčně číst. Proč?

Většinou jste také zpomínali dokázat, že řešení skutečně najde úsek s minimálním součtem – zde je důkaz celkem jednoduchý (viz níže), ale je potřeba jej udělat. Pokud se to někomu zdá zbytečné, nechť uváží jinou úlohu: najdi nejdelší období s zápornou bilancí. Zdálo by se, že je to uloha velmi podobná, ba až stejná. Ale ouha... všechno je jinak. Přemýšlejte sami, já vyřeším raději původní úlohu.

Předpokládejme, že existuje aspoň jedno nekladné období. Zadefinujme nadějný úsek jako úsek s nekladným součtem takový, že pro každé i menší nebo rovné délce úseku je součet prvních i členů nekladný a nedá se již prodloužit bez porušení předchozích podmínek.
Pak:

i) Žádné dva nadějné úseky se nepřekrývají
ii) Úsek s minimalním součtem leží v nějakém nadějném úseku, a lze jej umístit tak, aby začínal prvním prvkem tohoto úseku.

Důkaz:

i) Nechť první úsek P začíná na i a druhý (D) na j ($i < j$) a D se překrývá s P a není jeho částí. Pak úsek od i do konce D je nadějný a navíc je delší než P. Tedy P nemohl být podle definice maximální v délce a tudíž ani nadějný. Spor.
ii) Pokud budeme rozšiřovat minimální úsek (přesněji úsek s minimálním součtem) na obě strany podle podmínek (nekladnost prefixu), dostaneme nutně nadějný úsek. Pokud neleží začátek min. úseku na začátku tohoto úseku, je součet prefixu nula (jinak by se nejednalo o minimální úsek), tedy můžeme minimální úsek prodloužit až k začátku úseku vnějšího.

Pokud neexistuje žádné nekladné období, správným výstupem je minimum ze zadaných hodnot.

Algoritmus nebude dělat nic jiného, než že bude hledat nadějné úseky a zkoušet, zda součet některého z jejich prefixů nebude náhodou menší než doposud nalezené minimum a případně minimum upraví. Zároveň pokud najde nějaké kladné číslo, porovná jej s minimem a taktéž jej případně upraví.

Hledání nadějných úseků je jednoduché – beru postupně čísla; najdu-li záporné, pak začíná nadějný úsek, ten trvá tak dlouho, dokud je suma po každém přidání čísla nekladná. Pak pokračuje hledáním dalšího nadějného úseku.

Časová složitost je lineární, paměťová konstantní.

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

int main (void)
{
    int N, M, Mb, Me, s, sb, i, h;

    Mb=Me=0; M=INT_MAX;
    s=sb=0;
    scanf ("%d", &N);
    for (i=0; i<N; i++) {
      scanf ("%d", &h);
      if (h>0) {
        if (h<M) {M=h; Mb=Me=i; }
        if (s<M && sb-i) {M=s; Mb=sb; Me=i-1; }
      }
      if (s>0) {s=h; sb=i; } else s+=h;
    }
    if (s<M)
      printf ("minimum je od %d do %d a bylo celkem %d$\n", sb+1, N, s);
    else
      printf ("minimum je od %d do %d a bylo celkem %d$\n", Mb+1, Me+1, M);
    return 0;
}
```

12-3-2 Vypočítavý Robin — Pavel Šanda

Smutně rozjímaje nad družinou zamítnutým nákupem nové houfnice, jež by úlohu krajně zjednodušila, Robin nepokojně bloumal lesem. Bylo mu jasné, že triviální algoritmus s časovou složitostí $O(n^3)$ není přijatelný (každou dvojici bodů spojit se zbylými). Teprve po probdělé noci dostal nápad vybudovat všechny přímky (těch bude řádově n^2), které stačí následně setřídit a zjistit počet shodných přímek. Ještě lépe – setřídit vždy podle směru přímky procházející jedním bodem. Tím se dostal již na přijatelnou úroveň $O(n^2 \log n)$. Třídění mu k sobě dostane stejné přímky, na druhou stranu zbytečně třídí nestejné přímky mezi sebou. Nešlo by to bez něj? Dvé jeho uhlově černých oček zablýsklo – hashování! Nejenže dostane shodné přímky k sobě, ale navíc pro jednotlivou přímku provede „zatřídění“ v konstantním čase, což znamená výsledný algoritmus v $O(n^2)$.

K implementaci: v textu jsme používali výrazu „setřídit přímky“. Pro tuto operaci, stejně jako pro hashování je dobré mít přímku v nějakém rozumném zápisu – zde konkrétně směrnicovém tvaru s tím, že speciálně ošetříme vertikální případ. Hashovat budeme podle hodnoty směrnice, kolize různých přímek budeme řešit dynamickým seznamem pro danou hodnotu hashovací fce. Pro každou položku hashovacího pole zavedeme počítadlo, které nám bude udávat počet přímek v daném směru, procházejících právě zpracovávaným bodem. Po zpracování každého bodu musíme pročistit hashovací pole, což naštěstí časovou

složitost nijak nezhoršuje. Šlo by se tomu vyhnout, pokud bychom hashovali všechny přímky najednou. To by však mělo za následek zhoršení časové složitosti, nehledě na fakt, že by se komplikovalo použití počítadla atd.

Škarohlídi namítnou, že v případě magicky zadaných souřadnic jednotlivých rytířů se v tomto případě dostaneme na $O(n^3)$ - krom toho, že je tento případ výjimečný, můžeme tyto škarohlídy potěšit tím, že lze zkonstruovat takovou hashovací funkci, že algoritmus bude mít $O(n^2)$ vždy.

Pakliže bychom zanedbali hashovací pole, bude paměťová složitost $O(n)$. Konečnost i správnost algoritmu je zřejmá.

Několik slov ohledně strategie rytířů – u nezanedbatelného počtu řešitelů by slavilo úspěch umístění rytířů do jedné vertikální přímky, kterážto nemá směrnicové vyjádření a jejich řešení tuto skutečnost jednoduše ignorovala. (Ponechme stranou návrh sečtělého zbrojnoše na nakoupení košilí od firmy Milan Hendrych, Palackého ulice 156.)

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

#define Max 100
#define HMAX 131
#define X 0
#define Y 1

typedef struct Trect {
            int count;                          /* počet přímek */
            float k;                            /* přímka */
            struct Trect *next;
            } Trect;

int n, Best=1, hu[HMAX], used;                  /* rytířů, v jedné řadě, použité indexy v
                                                   hash */
int K[Max][2];                                  /* celočíselné souřadnice rytířů */
Trect *hash[HMAX];

int main (void)
{
   int i, j, f;
   float k;
   Trect *p, *P;

   scanf ("%d", &n); for (i=0; i<n; i++) scanf ("%d%d", &K[i][X], &K[i][Y]);

   for (i=0; i<n; i++) {
      used=0;
      for (j=i+1; j<n; j++) {
         if (K[i][X]==K[j][X]) {                /* na nule budeme mít i kolmé přímky */
            k=MAXFLOAT; f=0;
         } else {
            k= ( (float)K[j][Y]-K[i][Y])/ (K[j][X]-K[i][X]);
            f= (int) ( (atan (k)+M_PI_2)/M_PI*HMAX);    /* hashovací fce */
```

```
        }
        p=hash[f]; hu[used++]=f;
        if (p==NULL) {
          p= (Trect *)malloc (sizeof (Trect));
          p->next=NULL;
          p->count=1;
          p->k=k;
          hash[f]=p;
          continue;
        }
        do {
          if (p->k==k) {                    /* našli jsme dalšího na přímce */
            Best = Best < ++p->count ? p->count : Best;
            break;
          }
          if (p->next==NULL) {
            p->next= (Trect *)malloc (sizeof (Trect));
            p=p->next;
            p->next=NULL;
            p->count=1;
            p->k=k;
            break;
          }
        } while (p=p->next);
      }
      /* vyčištění starých hodnot */
      for (j=0; j<used; j++) {
        p=hash[hu[j]];
        if (!p) continue;
        while (p->next!=NULL)
          P=p->next, free (p), p=P;
        free (p);
        hash[hu[j]]=NULL;
      }
    }
    printf ("%d rytiru jednou ranou.\n", n==1 ?1:Best+1);
    return 0;
}
```

12-3-3 Palindromický rozklad — Zdeněk Dvořák

V zadání nebylo bohužel dostatečně vysvětleno, co znamená rozložit, načež někteří účastníci řešili úlohu jinou, zcela triviální – nalezení všech palindromů v daném řetězci; aby toto skloubili s uvedeným příkladem, užívali nepřirozených předpokladů typu jedno písmeno není palindrom. S ohledem na to, že v případě nejasností v zadání není problém se nás zeptat, hodnotil jsem podobné snahy max. 2 body.

Nyní k samotnému řešení úlohy tak, jak byla myšlena, tj. nalézt nejmenší množinu palindromů takovou, že se navzájem nepřekrývají a pokrývají zadaný řetězec.

Jedním ze správných řešení je dynamické programování (další možností, která se vyskytla u jednoho řešitele, je převést si úlohu na hledání nejkratší cesty v grafu; jinak též lze použít backtracking s ukládáním mezivýsledků). Základní myšlenka je tato: budeme si postupně počítat nejlepší rozklady začátků zadaného řetězce o délce $1, 2, \ldots, n$ (poslední z nich je hledaný výsledek). Každý rozklad (tedy i minimální) daného řetězce délky k, končícího palindromem délky l lze získat spojením rozkladu řetězce délky $k-l$ a tohoto palindromu. Velikost takovéhoto rozkladu je o jedna větší než velikost příslušného rozkladu délky $k-l$. Velikost nejmenšího rozkladu o délce $k-l$ pro každé možné l už známe, stačí tedy vzít jejich minimum pro všechna l taková, že úsek $k-l+1 \ldots k$ je palindrom. Řešení pochopitelně vždy existuje, neboť řetězec můžeme určitě alespoň poskládat z úseků délky 1.

Při samotné realizaci programu si neustále udržujeme seznam palindromů končících na dané pozici (resp. jejich začátků). Jinak je program prostým přepisem uvedeného algoritmu.

Správnost je dokázána v popisu. Konečnost je zřejmá, neboť horní meze všech cyklů jsou omezeny n.
Složitost (n je délka řetězce):

- Čas: Na každé pozici končí nejvýše n palindromů, tedy $O(n^2)$.
- Paměť: $O(n)$

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

#define MAXD 100
#define min (x, y)  ( (x)< (y) ? (x): (y))

int main (void)
{
    char zadani[MAXD+1];
    int pocet[MAXD+1];
    int palindromy[MAXD+1];
    int palindromu=0, akt, i, ai, n;

    pocet[0]=0;
    scanf ("%s", zadani);
    n=strlen (zadani);
    for  (akt=0; akt<n; akt++) {
       /* prodloužíme palindromy  */
       ai=0;
       for  (i=0; i<palindromu; i++)
          if  (palindromy[i] > 0 && zadani[palindromy[i]-1] == zadani[akt])
             palindromy[ai++]=palindromy[i]-1;
       palindromu=ai;
```

```
        /* palindrom délky 1 */
        palindromy[palindromu++]=akt;
        /* upravíme počet */
        pocet[akt+1]=n;
        for (i=0; i<palindromu; i++)
            pocet[akt+1]=min (pocet[akt+1], pocet[palindromy[i]]+1);
        /* přidáme palindrom nulové délky */
        palindromy[palindromu++]=akt+1;
    }
    printf ("Pocet palindromu, na ktere lze rozlozit %s je %d\n", zadani, pocet[n]);
    return 0;
}
```

12-3-4 Skákací panák — Jan Kára

Skákací panák je vlastně graf G, kde políčka jsou vrcholy a pokud jsou políčka propojena čarou, vede mezi vrcholy hrana. Navíc víme, že graf bude vlastně strom (existence právě jedné cesty mezi každými dvěma vrcholy je jednou z jeho ekvivalentních definic). Úkolem je pak nalézt hamiltonovskou kružnici v grafu H, kde dva vrcholy jsou propojené hranou právě když v grafu G byly spojeny cestou délky nejvýše tři. Kružnici nalezneme následujícím způsobem: Strom si zakořeníme v libovolném (třeba prvním) vrcholu. Z toho začneme strom procházet do hloubky. Pokud má vrchol, ve kterém se momentálně nacházíme, sudou vzdálenost od kořene, vrchol zašlápneme a postupně rekurzivně vyřídíme všechny podstromy. Pokud má vrchol lichou vzdálenost od kořene, nejdříve vyřídíme všechny podstromy a pak až vrchol zašlápneme. Algorimus má lineární časovou i paměťovou složitost vzhledem k počtu vrcholů.

A teď správnost algoritmu. To, že algoritmus zašlápne každé políčko právě jednou je zřejmé. Je ale třeba dokázat, že navržené přeskoky nebudou moc dlouhé. To dokážeme indukcí dle hloubky stromu. Pro stromy hloubky 0 a 1 zjevně žádný přeskok není moc dlouhý. Nechť máme strom hloubky $k+1$. Rozlišíme dvě možnosti:

1) Kořen bude zašlápnutý jako první. Potom proskáčeme jeho první podstrom (z indukčního předpokladu korektně) a skončíme v jeho kořeni. Z něj pak algoritmus chce přeskočit do sousedního podstromu do hloubky dvě. To je skok přes tři, a tedy je korektní. Z indukčního předpokladu opět korektně proskáčeme i tento podstrom a opět přeskočíme přes tři do souseda. Takto korektně proskáčeme všechny podstromy a skončíme v hloubce jedna. Tento strom jsme tedy celý proskákali korektně.
2) Kořen bude zašlápnutý jako poslední. Algoritmus tedy z indukčního předpokladu korektně proskáče jeho první podstrom a skončí v hloubce 2 (tedy pod kořenem podstromu). Z tohoto vrcholu

ale snadno přes tři přeskočíme do kořene sousedního podstromu, který opět korektně proskáčeme z indukčního předpokladu. Takto tedy korektně proskáčeme všechny podstromy a skončíme v hloubce dvě odkud přes dvě hrany skočíme do kořene celého stromu. Tento strom jsme tedy také proskákali korektně.

Výše jsme ukázali, že všechny skoky budou maximálně délky tři, a tedy navržené proskákání panáka bude koretní.

Program je přímou implementací algoritmu.

Poznámka pro zvídavé: Právě zkonstruovaný algoritmus nám říká, že libovolný souvislý graf (nejen strom) lze proskákat, pokud budeme skákat přes tři (zkuste si rozmyslet proč). Tedy problém hamiltonovské kružnice se skákáním přes tři hrany je již triviální. Na druhou stranu se dá ale ukázat, že se skákáním přes dvě hrany je problém stejně těžký (tedy NP-úplný), jako když se skáče pouze přes jednu hranu.

```
#include <stdio.h>

#define MAXV 100                          /* Maximální počet vrcholů */
/* Popis jednoho vrcholu */
struct vertex {
    int deg;                              /* Počet hran z vrcholu */
    int e[MAXV];                          /* Jednotlivé hrany */
    int cross[MAXV];                      /* Odkazy na */
};

struct vertex v[MAXV];                    /* Informace o vrcholech */
int n;                                    /* Počet vrcholů */
/* Načte graf */
void read_inp (void)
{
    int i, a, b;

    printf ("Pocet poli:␣");
    scanf ("%d", &n);
    /* Načte hrany */
    for (i = 0; i < n-1; i++) {
       scanf ("%d %d", &a, &b);
       a--; b--;
       v[a].e[v[a].deg++] = b;
       v[b].e[v[b].deg++] = a;
    }
}
/* Nalezne kružnici začnínající v A; P je číslo rodiče; D je hloubka */
void find_circle (int a, int p, int d)
{
    int i;

    if (! (d&1))                          /* Sudá hloubka */
       printf ("%d␣", a+1);
    for (i = 0; i < v[a].deg; i++)
```

```
    if (v[a].e[i] != p)                    /* Nevracíme se? */
      find_circle (v[a].e[i], a, d+1);     /* Vyřešíme podstrom */
  if (d&1)                                 /* Lichá hloubka */
    printf ("%d␣", a+1);
}
int main (void)
{
  read_inp ();                             /* Načteme strom */
  puts ("Navod:");
  find_circle (0, -1, 0);                  /* Nalezne a vypíše kružnici */
  puts ("");
  return 0;
}
```

12-3-5 PRAM — Daniel Kráľ

Nejprve popíšeme řešení v čase $O(\log n)$, kde n je počet zadaných čísel, s $O(n)$ procesory pro `P-CRCW PRAM`, `CREW PRAM` a `EREW PRAM`. Poté ukážeme rychlejší (a tedy lepší) řešení pro `P-CRCW PRAM`, jehož časová složitost bude konstantní a které bude používat $O(n^2)$ procesorů.

Budeme postupovat podobně jako v ukázkovém programu ve studijním textu a jako při řešení úlohy v první sérii. Nechť jsou prvky posloupnosti očíslované od jedné. Program bude probíhat v několika fázích; počet fází bude $\lceil \log n \rceil$. V k-té fázi bude l-tá buňka paměti, pro $2^k \mid l-1$, obsahovat největší číslo úseku délky 2^k z naší zadané posloupnosti, který začíná l-tým prvkem. Nyní procesory, jejichž číslo l zmenšené o jedna je dělitelné 2^{k+1}, porovnají hodnotu v l-té a $(l+2^k)$-té buňce a větší z nich uloží do l-té paměťové buňky. Je zřejmé, že po $\lceil \log n \rceil$ fázích obsahuje první buňka maximum ze všech čísel v posloupnosti. V samotném programu je ještě třeba dbát na ošetření „plusmínus“ jedničkových chyb, které by mohly snadno vzniknout. Program pro `EREW` je rozšířen pouze o počáteční rozkopírování délky posloupnosti, které jsme již několikráte diskutovali.

Program pro `P-CRCW PRAM` bude založen na následující myšlence: Každé dvojici prvků (x_i, x_j) (dvojici bereme uspořádaně) posloupnosti přiřadíme jeden procesor – ten porovná prvky x_i a x_j a v případě, že $x_i < x_j$, tak tento procesor prohlásí, že prvek x_i není největším prvkem v zadané posloupnosti. Prvek nebo prvky, o kterých žádný z procesorů neprohlásil, že nejsou největší, jsou zřejmě všechny stejné a zároveň jsou největší ze všech prvků v posloupnosti; tyto prvky zbývá již jen uložit do `gmem[0]`. Přiřazení dvojic proměnných procesorům uděláme stejně jako v řešení úlohy z druhé série. Právě popsané řešení potřebuje $O(n^2)$ procesorů a běží v čase $O(1)$.

Program pro CREW a P-CRCW v čase $O(\log n)$:

```
lmem[0]:=gmem[0]              ; tady bude délka posloupnosti
```

```
   lmem[1]:=cpuid               ; tady bude další testovaná buňka
   lmem[2]:=cpuid-1             ; moje číslo minus 1
   lmem[3]:=1                   ; posun k další buňce
1: lmem[4]:=lmem[2]&lmem[3]     ; budeme končit?
   if lmem[4]<>0 then goto 2    ; končíme
   lmem[1]:=cpuid+lmem[3]
   lmem[3]:=lmem[3]<<1
   if lmem[1]>lmem[0] then goto 2   ; jsme za koncem posloupnosti
   if gmem[cpuid]<gmem[lmem[1]] then gmem[cpuid]:=gmem[lmem[1]]
   goto 1
2: if cpuid=1 then gmem[0]:=gmem[1] ; a zapíšeme výsledek
   halt
```

Program pro EREW:

```
   if cpuid=1 then lmem[0]:=gmem[0]  ; délka posloupnosti
   lmem[1]:=gmem[cpuid]              ; schováme si obsah paměti
   gmem[cpuid]:=0                    ; znulujeme paměť
   if cpuid=1 then gmem[cpuid]:=gmem[0]
   lmem[2]:=1                        ; počet buněk s délkou posl.
1: lmem[3]:=cpuid+lmem[2]
   lmem[0]:=gmem[cpuid]
   if lmem[3]<=lmem[0] then gmem[lmem[3]]:=gmem[cpuid]
   lmem[2]:=lmem[2]<<1
   lmem[3]:=0                        ; budeme ještě kopírovat?
   if gmem[cpuid]=0 then lmem[3]:=1
   if gmem[cpuid]>lmem[2] then lmem[3]:=1
   if lmem[3]=1 then goto 1
   lmem[0]:=gmem[cpuid]              ; délka posloupnosti
   gmem[cpuid]:=lmem[1]              ; obnovíme obsah paměti
   lmem[1]:=cpuid                    ; další testovaná buňka
   lmem[2]:=cpuid-1                  ; moje číslo minus 1
   lmem[3]:=1                        ; posun k další buňce
2: lmem[4]:=lmem[2]&lmem[3];         ; budeme končit?
   if lmem[4]<>0 then goto 3         ; končíme
   lmem[1]:=cpuid+lmem[3];
   lmem[3]:=lmem[3]<<1
   if lmem[1]>lmem[0] then goto 3    ; jsme za koncem
                                       posloupnosti?
   if gmem[cpuid]<gmem[lmem[1]] then gmem[cpuid]:=gmem[lmem[1]]
   goto 2
3: if cpuid=1 then gmem[0]:=gmem[1]  ; a zapíšeme výsledek
   halt
```

Program pro P-CRCW v čase $O(1)$:

```
lmem[0]:=gmem[0]            ; délka posloupnosti
lmem[1]:=cpuid-1
lmem[1]:=lmem[1]%gmem[0]
lmem[1]:=lmem[1]+1          ; pořadí prvního čísla v mé dvojici
lmem[2]:=cpuid-1
lmem[2]:=lmem[2]/gmem[0]
lmem[2]:=lmem[2]+1          ; pořadí druhého čísla v mé dvojici
lmem[3]:=gmem[lmem[1]]
lmem[4]:=gmem[lmem[2]]
gmem[lmem[1]]:=1            ; příznak bytí největším číslem
if lmem[3]<lmem[4] then gmem[lmem[1]]:=0 ; tady musí být ostrá
                                           nerovnost
if gmem[lmem[1]]=1 then gmem[0]:=lmem[3] ; ulož největší číslo
halt
```

12-4-1 Magický zapletenec — Pavel Machek

Na tuto úlohu přišlo překvapivě málo řešení, i když jediná metoda řešení (backtracking) je jinak metodou hojně používanou. Opravovatelům není známé asymptoticky lepší řešení než projít všechny možné množiny vrcholů a otestovat o každé z nich, jestli po rozevření dané množiny kroužků vzniknou samé řetízky a že počet řetízků je menší než počet rozevřených kroužků. Navíc je třeba zvlášť otestovat, že zadání již není kružnice! Náš algoritmus by totiž na nezašmodrchaném řetízku odpověděl 1. Časová složitost bude $O(2^N)$, kde N je počet vrcholů.

A teď jak konkrétně budeme backtrackovat:

- Vybereme nějakou množinu vrcholů a označíme tyto vrcholy jako rozevřené (stačilo by nám procházet jen nejvýše $N-2$ prvkové množiny, ale to příliš procházení nezrychlí).
- Otestujeme, že nyní jsou v grafu už jen cesty: budeme postupně procházet všechny komponenty a obarvovat je. Pokud zjistíme stupeň vrcholu větší nez dva, nebo zjistíme kruh (vstupujeme do již obarveného vrcholu), vybraná množina určitě není správná.
- Pokud množina prošla prvním testem, tak zkontrolujeme, že máme dost rozevřených kroužků na to, abychom mohli vše pospojovat v jeden dlouhý řetěz. Pokud otevřených kroužků není dost, dopočítáme, kolik je jich ještě potřeba otevřít (dodatečné otevírání kroužků není nutné pro správnou funkci algoritmu, ale může backtracking výrazně zrychlit).

```
int matice[100][100];                    /* 2^100 uz je nekonecno ;-) */
```

```
int n;
int oznaceni[100];
int obarveni[100];
int rozpojenych = 0;
int min = 9999;
/* Vraci 1 pokud nasel zrejmou chybu */
int obarvi (int vrchol, int barva, int from)
{
    int stupen = 0;
    int i;

    if (obarveni[vrchol] != -1)                    /* Nemame radi cykly */
       return 1;
    obarveni[vrchol]=barva;
    for (i=0; i<n; i++) {
       if (!oznaceni[i] && matice[vrchol][i]) {   /* Jsme s timto krouzkem spojeni? */
          stupen++;
          if (i!=from && obarvi (i, barva, vrchol))   /* Obarvime ho (pokud jsme z nej
                                                         neprisli) */
             return 1;
       }
    }
    return (stupen>2);
}

void pocitej (void)
{
    int i;
    int retizku=0;
    int vysl;

    for (i=0; i<n; i++)
       obarveni[i]=-1;
    for (i=0; i<n; i++) {
       if ( (obarveni[i]==-1) && (!oznaceni[i]))
          if (obarvi (i, retizku++, -1))
             return;
    }
    if (retizku <= rozpojenych)
       vysl = rozpojenych;
    else
       vysl = rozpojenych + ( (retizku - rozpojenych + 1)/2);
    if (vysl < min)
       min = vysl;
}

void generuj (int i)
{
    if (i==n)
       pocitej ();
    else {
       oznaceni[i]=0;
       generuj (i+1);
       rozpojenych++;
```

```
        oznaceni[i]=1;
        generuj (i+1);
        rozpojenych--;
    }
}
int main (void)
{
    int a, b;
    scanf ("%d", &n);
    while (scanf ( "%d %d", &a, &b) == 2) {
        a--; b--;
        matice[a][b] = 1;
        matice[b][a] = 1;
    }
    /* Otestovat jestli nahodou neni retizek nezasmodrchany */
    generuj (0);
    printf ("Ulohu lze vyrezit s rozevrenim %d krouzku\n", rozpojenych);
    return 0;
}
```

12-4-2 Pyrus Achras — Zdeněk Dvořák

Ze zadání úlohy vyplývá, že se snažíme určit, zda nějaký bod (hruška) leží uvnitř konvexního obalu ostatních bodů (plot). Kdo nyní začal konstruovat konvexní obal, skončil s časovou složitostí alespoň $\Omega(n \log n)$ (a často i se 3 stránkami programu, které jsem následně musel opravovat...). Nyní jak by mohlo vypadat řešení:

Všimneme si, že bod leží mimo konvexní obal množiny jiných bodů právě tehdy, když existuje úhel menší než 180 stupňů s vrcholem v tomto bodě obsahující všechny body dané množiny. Tento úhel se pokusíme najít.

Do zadaného bodu umístíme počátek soustavy souřadnic. Dále načteme první z bodů množiny. Tento bod nám určí vektor, který jistě leží uvnitř hledaného úhlu. Budeme nyní určovat levou a pravou (vzhledem k tomuto vektoru) hranici úhlu, obsahujícího všechny dosud načtené body. Na počátku jsou obě tyto hranice totožné s vektorem k prvnímu z bodů. Jestliže již máme zpracovánu část vstupu a načteme další bod, rozlišíme 3 případy:

1) Vektor k načtenému bodu je vlevo od směru k prvnímu bodu. Pak ověříme, zda tento směr je vlevo od levé hranice, a je-li, příslušně ji upravíme.
2) Vektor k načtenému bodu je vpravo od směru k prvnímu bodu. Řešíme analogicky.
3) Oba vektory mají stejný směr; mají-li navíc stejnou orientaci, nic neděláme. Mají-li opačnou orientaci, zadaný bod leží na přímce spojující 2 body a tedy i uvnitř konvexního obalu.

Jestliže na konci leží levá hranice vlevo od pravé, máme hledaný úhel. Jinak takový úhel zjevně nemůže existovat (úhly mezi pravou a levou hranicí a mezi hranicemi a vektorem odpovídajícím prvnímu z bodů jsou menší než 180 stupňů, tedy máme-li libovolnou přímku procházející daným bodem, v obou polorovinách jí určených leží alespoň jeden z nich).

Polohu dvou vektorů určíme na základě jejich vektorového součinu (třetí složku doplníme nulovou). To je totiž vektor k nim kolmý takový, že tvoří kladně orientovanou soustavu. Stačí nám tedy spočítat třetí souřadnici tohoto součinu a podívat se na její znaménko.

Konečnost algoritmu je zřejmá (neobsahuje žádné netriviální cykly).

Časová složitost je lineární k počtu stromů, paměťová je konstantní.

Správnost algoritmu **není** zřejmá z popisu (což se mi bohužel všichni řešitelé, kteří se o něčem takovém jako důkaz zmínili, snažili tvrdit); nicméně uvedené skutečnosti jsou natolik intuitivně zřejmé, že skutečně formální důkaz (který by celou ideu spíše zamlžil) zde neuvedu.

```
#include <stdio.h>

char *v="Strom je uvnitr zahrady";
char *m="Strom neni uvnitr zahrady";

#define sgn (x)  ( ( (x)>0)- ( (x)<0))
#define side (ax, ay, rx, ry)  (sgn ( (ax)* (ry)- (ay)* (rx)))

char *urci (void)
{
    int n, cx, cy, lx, ly, rx, ry, hx, hy, ax, ay;

    scanf ("%d%d", &hx, &hy);
    scanf ("%d", &n);
    if (n--<1) return m;
    scanf ("%d%d", &cx, &cy);
    cx-=hx; cy-=hy;
    if (!cx&&!cy) return v;
    lx=rx=cx;
    ly=ry=cy;
    while (n--)
    {
       scanf ("%d%d", &ax, &ay);
       ax-=hx; ay-=hy;
       switch (side (ax, ay, cx, cy))
       {
          case -1:
             if (side (ax, ay, lx, ly)==-1) {lx=ax; ly=ay; };
             break;
          case 0:
             if (ax*cx<=0&&ay*cy<=0) return v;
             break;
          case 1:
             if (side (ax, ay, rx, ry)==1) {rx=ax; ry=ay; };
```

```
            break;
        }
    }
    return side (lx, ly, rx, ry)>=0 ?v:m;
}
int main (void)
{
    puts (urci ());
    return 0;
}
```

12-4-3 Šachová šlamastika — Pavel Šanda

Pakliže jste měli štěstí, potkali jste na posledním KOSu (Konference odvážných starostů) starostu města Hauntry. Tento starosta byl neobyčejně výřečný a každého volky nevolky seznamoval s jeho vynikajícími výsledky v oblasti vyhledávání strašidelných domů. Ve zkrácené verzi problém spočíval v nalezení největšího možného bloku domů, jež oblažují svojí přítomností strašidla (kdo koho?). Asi vás již napadlo optimální řešení zadaného problému – použijeme dva zásadní triky. Jednak převedeme naši úlohu na známou strašidelnou (přesněji vyhledávání co největší jedničkové podmatice), druhak řešení této úlohy bezostyšně opíšeme.

Převod: Na naši matici položíme 'filtr', který vyznačí jedničkami místa, kde se vyskytuje 'správná' barva vzhledem k hledané šachovnici (správná je ta, kde očekávaná barva = skutečná barva). Tak v matici vzniknou jedničkové podmatice, které máme vyhledat. K našemu úžasu se však v matici objeví nulové podmatice, které odpovídají aplikaci non-filtru pro šachovnici o jedna posunutou (uvědomte si, že jsou jen dva typy hledaných šachovnic, které vzniknou posunutím = negací).

Nyní stačí vyhledat ony maximální jedničkové (nulové) podmatice a jsme hotoví. Druhý špinavý trik bude vyřešen tím, že nahlédnete do již řešené KSP úlohy 9-1-3 , kde také najdete detaily. Abych však neriskoval (u)kamenování, uvedu alespoň myšlenku.

Použijeme opět dvou triků.

První: spočtu si pro danou matici spočtu dvě další A, B – na místa jedniček napíšeme počet jedniček, které se nachází přímo pod danou jedničkou (v matici A) a nad danou jedničkou (matice B) (včetně). Matici B dokáži snadno spočítat průchodem postupně po řádcích shora dolů (při výpočtu počtu jedniček nad aktuální budu využívat již spočteného počtu jedniček na předchozím řádku), matici A průchodem po řádcích zdola nahoru.

Druhý: Hledaná strana maximálního jedničkového obdélníku musí přiléhat buď k okraji matice, nebo k nulovému prvku. Protože jsme si okolo celé ma-

tice vytvořili pás nul, můžeme prohlásit, že strana maximálního jedničkového obdélníka přiléhá k nějakému nulovému prvku.

Hledat obdélníky tedy můžeme tak, že budeme postupně procházet řádky shora dolů, zleva doprava. Vždy když narazíme na jedničku, která následuje po nule, tak začneme nový obdélník. Pro každý obdélník si udržujeme jednak minimum z počtu jedniček od aktuální řádky dolů, jednak minimum z počtu jedniček od aktuální řádky nahoru (tato minima dokážeme díky předpočítaným maticím A a B snadno v konstantním čase upravovat). Zřejmě vzdálenost k začátku obdélníka krát minimum nahoru plus minimum dolů nám dá obsah aktuálního obdélníka, který tedy stačí porovnat s obsahem dosud největšího nalezeného. Když se na aktuálním řádku vyskytne nula, aktuální obdélník již nemůže pokračovat, a proto ho ukončíme.

Celková paměťová i časová složitost je evidentně $O(MN)$, kde N, M jsou rozměry matice, protože na každý prvek jsme sáhli nejvýše konstantněkrát.

Stran vašich řešení: poměrně často se ve vašich řešeních vyskytoval čas $O(NM^2)$ s transponováním matice, pakliže $M > N$. Vyskytlo se dokonce jedno řešení $O(NM^2)$ s paměťovou složitostí $O(M)$.

Stran realizace: Filtr je možné definovat pomocí fce `xor`, posun šachovnice je možný provést „negací“ již zfiltrované matice. Byla by možná úprava algoritmu, jenž by vyhledával nuly a jedničky zároveň. Na asymptotické složitosti ale zřejmě nic nezmění. V programu se držím značení zavedeného v 9-1-3.

12-4-4 Barevné intervaly — Aleš Přívětivý

Úkolem této úlohy bylo najít minimální počet barev potřebný k obarvení zadaných intervalů a jedno takové obarvení tak, aby žádné dva intervaly, které se překrývají, neměly shodnou barvu. Mezi došlými řešeními se vyskytla řada algoritmů – efektivních i méně efektivních. My si zde ukážeme hladový algoritmus, který tuto úlohu řeší v čase $O(n \log n)$, a důkaz jeho správnosti.

Algoritmus: Setřídíme si intervaly v neklesajícím pořadí podle souřadnic jejich počátků. Pak jim postupně v tomto pořadí přidělujeme barvy. Přidělená barva je libovolná již použitá barva různá od barev všech doposud obarvených intervalů, které se překrývají s právě barveným intervalem. Pokud žádná taková barva neexistuje vytvoříme novou barvu a interval obarvíme pomocí ní.

Tvrzení: Algoritmus použije na obarvení intervalů nejmenší možný počet barev a takto zkonstruované obarvení je korektní obarvení.

Důkaz: Kdyby algoritmem zkonstruované obarvení nebylo korektní, existovaly by alespoň dva překrývající se intervaly se stejnou barvou. To ale není možné, vzhledem k výběru barvy při barvení intervalu (vybereme vždy takovou, aby nekolidovala s doposud obarvenými intervaly). Zbývá tedy ukázat, že použitý počet barev je nejmenší možný. Označme si k maximální počet intervalů překrývajících se nad libovolným bodem reálné přímky. Libovolné korektní

obarvení intervalů musí použít alespoň k barev (jinak by se nad některým bodem překrývalo více intervalů než máme barev). Ukažme, že náš algoritmus použije právě k barev, a proto bude počet barev jím použitý nejmenší možný. Kdyby jich použil více, pak jsme při barvení nějakého intervalu našli alespoň k s ním se překrývajících doposud obarvených intervalů, a tedy jsme museli vytvořit novou barvu. Označme takový interval I. Podle volby k ale víme, že intervalů překrývajících se nad jedním bodem je maximálně k, tedy i nad bodem odpovídajícímu počátečnímu bodu I se jich překrývá maximálně k. Všimněme si, že všechny doposud obarvené intervaly, které se překrývají s intervalem I, obsahují také počáteční bod I - barvíme je v pořadí vzrůstajících počátečních bodů intervalů. Tedy jich je nejvíce $k-1$ (bez intervalu I), což je spor s tím, že jsme při barvení intervalu I našli alespoň k doposud obarvených intervalů překrývajících se s I. Dokazované tvrzení tedy platí.

Nyní se zaměřme na efektivní implementaci tohoto algoritmu. Je potřeba umět rychle najít barvu pro nový interval, aby nekolidovala s předchozím obarvením. Pro tento účel si zřídíme zásobník barev. Budeme se pohybovat po hraničních bodech intervalů postupně zleva doprava a aktuální stav zásobníku bude odrážet, které barvy nejsou použity v intervalech obsahující bod, na kterém stojíme. Když bude nějaká barva v zásobníku, znamená to, že jí můžeme obarvit interval začínající v bodě, na kterém právě stojíme. To, že zpracovávaný interval touto barvou obarvíme, znamená, že barva je použitá a musíme ji vyjmout ze zásobníku. Barvu do zásobníku vrátíme v okamžiku, když vstoupíme do koncového bodu intervalu, který jí byl obarven. Abychom mohli takto hezky popořadě postupovat po hraničních bodech intervalů, vytvoříme si pole začátků a konců intervalů a utřídíme ho. Po skončení algoritmu vypíšeme počet použitých barev a obarvení jednotlivých intervalů.

Paměťová složitost je lineární k počtu intervalů n, tj. $O(n)$. Časová složitost zahrnuje čas na setřídění pole bodů (začátků a konců intervalů), a pak čas stravený algoritmem při procházení jednotlivých bodů a buď barvením intervalu nebo uvolněním barvy. Přidělit a uvolnit barvu intervalu umíme v konstatním čase, počet bodů je $2n$, tedy časová složitost druhé fáze je $O(n)$. Čas na utřídění pole bodů je závislý na použitém třídícím algoritmu, v případě HybridSortu je to v průměrném případě $O(n)$, v nejhorším $O(n \log n)$. Ve vzorovém programu je použit QuickSort. Celková časová složitost je tedy $O(n \log n)$.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

struct t_interval {
    double meze[2];                    /* krajni body intervalu */
    int barva;                         /* barva intervalu */
};

struct t_bod {
```

```
    int intrvl;                                  /* interval, jehoz je krajnim bodem */
    int mez;                                     /* je bod levy okraj (0), nebo pravy (1) */
};
struct t_interval interval[MAX];                 /* seznam vsech intervalu */
struct t_bod bod[2*MAX];                         /* seznam vsech krajnich bodu */
int colorstack[MAX], top;                        /* zasobnik barev */
int i, n, barev;
int cmp ( struct t_bod *e1, struct t_bod *e2 )   /* pomocna fce pro qsort */
{
    int d=interval[e1->intrvl].meze[e1->mez] - interval[e2->intrvl].meze[e2->mez];
    return (d==0 ? e2->mez - e1->mez : d );
}
int main (void)
{
/* nejprve nacteme vsechny intervaly, a jejich pocet (n), a usporadame */
/* si v poli bod jejich krajni body do neklesajici posloupnosti */
    scanf ("%d", &n);
    for (i=0; i<n; i++) {
       scanf ("%lf%lf", interval[i].meze, interval[i].meze+1);
       bod[2*i].intrvl=i; bod[2*i].mez=0;
       bod[2*i+1].intrvl=i; bod[2*i+1].mez=1;
    }
    qsort (bod, 2*n, sizeof (struct t_bod), cmp);
/* nyni prochazime poporade pole bodu, a kdyz narazime na koncovy bod */
/* intervalu, barvu vratime na zasobnik, kdyz na zacatecni vezmeme */
/* prvni ze zasobniku, a pokud je zasobnik prazdny vytvorime novou */
    barev=0; top=0;
    for (i=0; i<2*n; i++)
       if (bod[i].mez==1)
          colorstack[top++]=interval[bod[i].intrvl].barva;
       else {
          if (top==0)
             interval[bod[i].intrvl].barva=++barev;
          else
             interval[bod[i].intrvl].barva=colorstack[--top];
       }
/* nakonec vypiseme vysledky naseho snazeni */
    printf ("Pouzito barev: %d\n", barev);
    for (i=0; i<n; i++)
       printf ("Interval [%5.2lf,%5.2lf] ma barvu: %d\n", interval[i] );
    return 0;
}
```

12-4-5 PRAM — Daniel Kráľ

Nejprve si popišme řešení úlohy a). Náš program bude pracovat v několika etapách. Po k etapách bude platit, že i-tá buňka globální paměti obsahuje součet $\min(i, 2^k)$ předcházejících buněk (počítáno včetně jí samé). Tedy například

po první etapě bude šestá buňka obsahovat a_5+a_6, po dvou etapách bude obsahovat $a_3+a_4+a_5+a_6$ a po třech etapách bude obsahovat $a_1+a_2+a_3+a_4+a_5+a_6$. Je zřejmé, že po $\lceil \log n \rceil$ obsahují všechny buňky globální paměti požadované součty (n je počet sčítaných čísel).

Spustíme n procesorů a každý z nich bude počítat částečný součet v buňce `gmem[cpuid]`. Pokud je tento součet již dopočítaný, procesor se zastaví. V k-té etapě přičte procesor k obsahu buňky `gmem[cpuid]` obsah buňky `gmem`$[$`cpuid`$-2^{k-1}]$. Protože před k-tou etapou obsahovala buňka `gmem[cpuid]` součet 2^{k-1} předchozích čísel a v buňce `gmem`$[$`cpuid`$-2^{k-1}]$ byl uložen součet $\min($`cpuid`$-2^{k-1}, 2^{k-1})$ čísel, bude v `gmem[cpuid]` po k-té etapě součet $\min($`cpuid`$, 2^k)$ předchozích čísel, což jsme požadovali.

Vytvoření samotného programu je již snadné. Místo toho, abychom si pamatovali v kolikáté etapě se nacházíme a z tohoto čísla pak počítali příslušnou mocninu dvojky, budeme si udržovat rovnou číslo 2^k. Pokud toto počítadlo dosáhne našeho `cpuid` skončíme. V opačném případě přičteme ke `gmem[cpuid]` obsah globální paměťové buňky na pozici `cpuid`-2^k. Z úvah prvního odstavce vyplývá, že poslední procesor dopočítá po $O(\log n)$ krocích. Počet použitých procesorů je $O(n)$; naše implementace algoritmu běží dokonce i na `CREW PRAM`u.

Řešení úlohy b) je ještě jednodušší než řešení úlohy a). Stačí si totiž uvědomit, že $\lceil \log x \rceil = n$ právě tehdy když $2^{n-1} < x \leq 2^n$. Stačí tedy, když každý z x spuštěných procesorů ověří, že $2^{n-1} < x \leq 2^n$, kde za n zvolí své `cpuid`. Je nasnadě, že jeden z procesorů ověří nerovnosti pro správné n, neboť pro všechna čísla x platí: $\lceil \log x \rceil \leq x$. K ověření dvou výše uvedených nerovností použijeme následující zřejmé ekvivalence:

$$2^{n-1} < x \leftrightarrow 2^{n-1} \leq x-1 \leftrightarrow (x-1) >> (n-1) \neq 0$$

$$x \leq 2^n \leftrightarrow x-1 < 2^n \leftrightarrow (x-1) >> n = 0$$

Vytvořit požadovaný program je nyní již snadné; nesmíme však zapomenout ošetřit případ $x = 1$, kdy je spravný výsledek $\lceil \log 1 \rceil = 0$. Námi navržený algoritmus pracuje i na `CREW PRAM`u v konstantním čase ($O(1)$).

Úloha a):

```
   lmem[1]:=1                                  ; inicializujeme posun
1: if cpuid<=lmem[1] then halt                 ; dopočítáno?
   lmem[0]:=cpuid-lmem[1]                      ; co budeme přičítat
   lmem[1]:=lmem[1]<<1                         ; zvětšíme posun
   gmem[cpuid]:=gmem[cpuid]+gmem[lmem[0]]      ; přičteme
   goto 1                                      ; a opakujeme ...
```

Úloha b):

```
   if gmem[0]=1 then gmem[0]=0  ; ošetříme případ log 1
```

```
if gmem[0]=0 then halt
lmem[0]:=gmem[0]-1          ; dané číslo zmenšené o 1
lmem[1]:=cpuid-1
lmem[1]:=lmem[0]>>lmem[1]
lmem[0]:=lmem[0]>>cpuid
if lmem[1]=0 then halt      ; zkontrolujeme nerovnosti
if lmem[0]<>0 then halt
gmem[0]:=cpuid              ; případně zapíšeme výsledek
halt
```

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Jozef Tvarožek	G J. Hornca, Bratislava	2	19	198
2.	Miroslav Rudišin	G Šrobárova, Košice	3	18	138
3.	Tomáš Valla	G Kralupy n. Vlt.	4	18	118
4.	Martin Hamrle	G Pelhřimov	2	18	110
5.	Martin Zlomek	Purkyňovo G, Strážnice	3	16	87
6.	Jiří Koula	G U Lib. zámku, Praha	3	16	85
7.	Jiří Svoboda	G Zborovská, Praha	2	16	80
8.	Josef Hala	G Uherské Hradiště	3	14	79
9.	Marián Dvorský	G Šrobárova, Košice	3	13	74
10.	Ondřej Zajíček	SPŠ strojnická, Chrudim	3	13	73
11. – 12.	Roman Krejčík	G Zborovská, Praha	3	12	72
	Aleš Wojnar	G Třinec	2	11	72
13.	Milan Vraný	G Mikulášské n., Plzeň	3	11	68
14.	Jiří Cvachovec	G Tř. kpt. Jaroše, Brno	4	10	66
15.	Michal Lichvár	G Ľ. Štúra, Trenčín	2	10	62
16.	Radim Tyleček	G Zborovská, Praha	3	10	61
17.	Jiří Paleček	G Kladno	1	9	53
18.	Zdeněk Bouchner	G Otokara Březiny, Telč	4	9	49
19.	Pavel Čížek	G Kralupy n. Vlt.	3	8	48
20.	Marek Sulovský	G Tř. kpt. Jaroše, Brno	3	8	42
21.	Tomáš Matoušek	G Karlovy Vary	4	8	40
22.	Martin Včelák	G Elgartova, Brno	2	8	38
23.	David Kovář	G Otokara Březiny, Telč	4	8	34
24.	Petr Adam	G Třebíč	4	8	33
25.	Lukáš Rypl	G Vrchlabí	4	8	30
26. – 27.	Andrej Chu	G Čadca	4	7	29
	Martin Nečaský	G Semily	4	7	29
28.	František Němec	G Zborovská, Praha	3	6	25
29.	Jan Matějek	G Kladno	1	6	23
30.	Tomáš Kadlec	G Příbram	3	6	22
31. – 33.	Radek Chromý	G Otokara Březiny, Telč	4	5	21
	Marek Sterzik	SPŠ Ostrov	1	5	21
	Jan Tichý	G Řečice, Brno	3	5	21
34. – 35.	Alexej Sidorenko	G Holešov	2	4	20
	Pavel Šimeček	G Tř. kpt. Jaroše, Brno	4	4	20
36.	Vojtěch Meluzín	G Boskovice	3	4	18
37.	Tomáš Vyskočil	G Lanškroun	4	3	15

38.	Václav Cviček	G Frýdek-Místek	1	3	14
39.	Zdeněk Hrnčíř	G Hradec Králové	3	3	13
40. – 41.	Lukáš Homoľa	G Zbrojničná, Košice	3	3	12
	Jiří Iša	G Ústí nad Labem	3	3	12
42. – 43.	Juraj Matuš	G Sabinov	3	2	11
	Ondrej Sňahničan	G Púchov	1	2	11
44.	Lukáš Horák	G Hradec Králové	3	2	9
45.	Jan Javůrek	SPŠ Jihlava	2	1	5
46.	Stanislav Kopr		?	1	3
47. – 68.	Miroslav Bajtoš	G J. Hornca, Bratislava	4	0	0
	Jiří Bláha	G Český Krumlov	1	0	0
	Zdeněk Bulan	G Benešov	3	0	0
	Marek Čapek	G Děčín	3	0	0
	Pavel Čížek	SPŠ Jedovnice	3	0	0
	Jaroslav Dražan	G Příbram	4	0	0
	Kryštof Hoder	G Tř. kpt. Jaroše, Brno	1	0	0
	Lukáš Hofmann	G Nad Štolou, Praha	3	0	0
	Jan Jakubův	G Vlašim	4	0	0
	Antonín Janec	G Čs. exilu, Ostrava	3	0	0
	Michaela Kloboučková	G Mladá Boleslav	3	0	0
	Aleš Kučík	G Trutnov	3	0	0
	Vojtěch Liška	G Mikulášské n., Plzeň	1	0	0
	Lukáš Nalezenec	G Turnov	2	0	0
	Oto Petřík	G Vrchlaví	-2	0	0
	Jiří Plachý	G Uherské Hradiště	3	0	0
	Jaroslav Pospíchal	G Nad Štolou, Praha	3	0	0
	Milan Straka		?	0	0
	Tereza Sýkorová	G Buďánka, Praha	3	0	0
	Martin Szablatura	G Karviná	?	0	0
	Jaroslav Šťáva	G Matyáše Lercha, Brno	3	0	0
	Michal Tarana	G Veľká Okružná, Žilina	3	0	0

Experti v Linuxu

Společnost SuSE CR, s.r.o., zastoupení jedné z největších světových distribucí Linuxu, německé společnosti SuSE Linux AG, působí v České republice již od loňského září. Jedním z hlavních impulsů vytvoření pražské pobočky byl dostatek velmi zkušených linuxových vývojářů a také zkušených uživatelů SuSE Linuxu v České republice.

Společnost SuSE CR nabízí nejen distribuci SuSE Linuxu, ale především provádí jeho lokalizaci pro český trh. Kromě toho se také podílí na lokalizaci některých linuxových aplikací, jako je např. StarOffice.

Mezi hlavní služby společnosti SuSE CR patří instalační a technická podpora v českém jazyce, poskytování školení jak pro začínající uživatele SuSE Linuxu, tak pro zkušené odborníky.

V září 2000, rok po vzniku pražské pobočky pracuje v SuSE CR čtyřicet zaměstnanců, z nichž 70% tvoří vývojáři. V celosvětovém měřítku má česká pobočka již 10% ze všech zaměstnanců společnosti SuSE a tím tvoří základ celosvětového vývoje společnosti. Pro společnost SuSE CR pracují zkušení linuxoví odborníci jak na plný úvazek, tak i externě. Mnoho vývojářů SuSE Linuxu je z řad studentů MFF. Práce se SuSE Linuxem je zajímavá, dává možnost se uplatnit v celosvětovém měřítku a co je velmi důležité má perspektivní budoucnost.

Více informací o celé společnosti získáte na adrese: `www.suse.cz`

Od roku 2000 je společnost SuSE CR, s.r.o. sponzorem MFF.

Obsah

Jan Kára a kolektiv

Korespondenční seminář v programování XII. ročník

Autoři a opravující úloh:
Martin Mareš, Zdeněk Dvořák, Pavel Machek,
Pavel Nejedlý, Jan Kára, Robert Špalek,
Daniel Kráľ, Aleš Přívětivý, Pavel Šanda

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 2000

Vytisklo Reprografické středisko MFF
Malostranské nám. 25, 118 00 Praha 1

76 stran, 4 obrázky
Písmem Computer Modern v programu TEX vysázel Jan Kára
Vydání první
Náklad 300 výtisků