Milí řešitelé!

Do ruky se Vám dostává řešení závěrečné série šestnáctého ročníku KSP, spolu s výsledkovou listinou.

Jistě víte, že několik „nejlepších“ řešitelů zveme na podzimní týdenní soustředení. Kdo to bude se dozvíte až v září, pokud Vám přijde pozvánka. Nicméne my už nyní známe termín soustředění – bude 10. až 16. října (± den na obou koncích).

Ještě bychom Vás chtěli poprosit: myslíme si, že je škoda, že náš seminář řeší docela málo řešitelů. Byli bychom rádi, pokud byste nám na začátku příštího roku pomohli s „reklamou“ – například pověsit zadání první série na školní nástěnku, ...

Aktuální informace o KSP můžete nalézt na Internetu na *http://ksp.mff.cuni.cz/,* kde se také během pár týdnů objeví i zadání první série nového ročníku. Libovolné dotazy organizátorům je možno posílat e-mailem na adresu *ksp@mff.cuni.cz.*

Vzorová řešení čtvrté série šestnáctého ročníku KSP

16-4-1 Mnichova posedlost aneb Hanoi strikes back

Řádění Askejťáka mnichy zmátlo natolik, že nechali nápravu pouze na Vás. Ovšem často jenom čekali a čekali a spali a čekali... Nakonec se na nekonečné posouvání disků nemohl dívat ani sám Bůhdha a rozhodl se přispět svou troškou k nápravě toho, co jeho příbuzný Kazisvět napáchal.

Začněme jednoduchým pozorováním: věž je vždy nejlepší postavit na tom kolíku, kde leží největší disk. Kdybychom ji chtěli postavit na jiném kolíku, museli bychom všech $N-1$ disků přesunout na nějaký volný kolík, pak přesunout největší disk na cílový kolík a pak na něj přesunout zbytek disků. Nicméně v tom stavu, kdy bylo všech $N-1$ disků na nějakém volném kolíku a největší disk nebyl dosud přesunutý, jsme mohli rovnou přesunout $N-1$ disků na ten největší a ušetřit tak jedno přesunutí (největšího disku).

Dále bychom potřebovali vědět, na kolik přesunutí dokážeme přemístit věž o m discích na jiný kolík. Označme tento počet $T(m)$ (evidentně $T(1)=1$). Když tedy chceme přesunout věž o m discích na jiný kolík, musíme nejprve $m-1$ disků přesunout na volný kolík, největší disk přesunout na cílový a zbylých $m-1$ disků vrátit zpátky na největší disk. Toto jednoduché pozorování se dá vyjádřit takto: $T(m)=T(m-1)+1+T(m-1)=2T(m-1)+1$. Po chvilce počítání zjistíme, že $T(m)=2^m-1$.

Vyzbrojení (Bůhdhou) těmito dvěma pozorováními můžeme se směle pustiti do našeho původního problému. Budeme si pro každý disk i počítat, kolik přesunů by nám zabralo postavit věž s disky $i,\ldots,N$ na každý z kolíků $1,2,3$. Tuto hodnotu můžeme značit $\mathit{Tahy}(i,k)$, kde k je číslo kolíku. Ještě si $K(i)$ označme, na jakém kolíku je i-tý disk.

Je jasné, že hledaná hodnota je $\mathit{Tahy}(1,K(1))$. Jak ale tyto hodnoty počítat? Pro jediný disk určitě platí, že na nějakém kolíku už je, nebo ho tam mohu přesunout jediným tahem, tedy $\mathit{Tahy}(N,k=K(N))=1$ a $\mathit{Tahy}(N,k\neq K(N))=0$.

Pokud chceme přesunout disky od i-tého až po nejmenší (N-tý) na kolík k, můžou nastat dva případy. Buď je i-tý disk na kolíku k a pak je nutné přesunout zbylé disky na kolík k, čili $\mathit{Tahy}(i,k=K(i))=\mathit{Tahy}(i+1,k)$. Nebo disk i na tomto kolíku neleží a pak je nutné přesunout disky od $(i+1)$-ního na volný kolík pom (ani k ani $K(i)$), disk i přesunout na kolík k a pak všechny disky od $(i+1)$-ního vrátit na kolík k, čili $\mathit{Tahy}(i,k\neq K(i))=\mathit{Tahy}(i{+}1,\mathit{pom})+1+T(N-i)=\mathit{Tahy}(i+1,\mathit{pom})+2^{N-i}$.

Z toho plyne řešení s lineární časovou složitostí. Budeme počítat jednotlivé hodnoty $\mathit{Tahy}(i,k)$ pro $i{=}N,\ldots,1$ a nakonec vypíšeme hodnotu $\mathit{Tahy}(1,K(1))$. Takto by byla paměťová složitost také lineární. Nicméně můžeme si všimnout toho, že hodnotu $\mathit{Tahy}(i,k)$ potřebujeme jen k výpočtu hodnot $\mathit{Tahy}(i-1,k)$ – čili není nutné si pole Tahy pamatovat celé, ale stačí nám aktuální dva řádky. Pokud by vstupní data byla čísla kolíků, na kterém leží disky $N,\ldots,1$, bude paměťová složitost konstantní.

Ještě poznámka k došlým řešením – málokdo vzal v úvahu, že počet přesunů může být až $2^{64}-1$ a že se tedy nejspíš nevejde do běžného 32-bitového čísla. Nicméně většina dnešních jazyků 64-bitové čísla podporuje a tak stačilo použít příslušný 64-bitový typ.

Navíc mnozí řešitelé používali při výpočtu mocniny dvojky a tak si je předpočítali do pole. To ale není vůbec třeba, protože jak v Pascalu tak v Céčku existují operace bitového posunu, který se dá použít přesně k tomuto výpočtu: $2^i = 1$ `<<` $i = 1$ `shl` i.

Milan Straka

Ve zkratce: víme, kde má skončit největší disk. Končíme tedy tím, že přesuneme druhý největší disk (evidentně také z místa, kde byl na začátku) na první a ze zbylého kolíku pak přestěhujeme všechny zbylé disky nad druhý disk. Jinými slovy potřebujeme 2^{N-2} tahů + tahy na složení zbylých disků na zbylý kolík, což je ale ta samá úloha (oba velké disky nám v tom nemohou nijak překážet). Takže můžeme všechny disky zpracovat pozpátku v konstantním prostoru, lineárním čase a jednom řádku programu (viz). –M.M.

16-4-2 Technologické trable

Kuchařka a počet bodů svádí k tomu, abychom na naše pole prostě napasovali binární vyhledávání a hotovo. Jenže základní problém se světem je ten, že krokodýla štěkat nenaučíš a půlením nekonečna se ke konstantě nedobereš (a tedy ani k hledané hodnotě).

Proto budeme muset nejprve vyřešit tři problémy:

a) Vzhledem k čemu měřit časovou složitost, jestliže pole ve kterém hledáme je nekonečné.

1. Když se standardně uvažuje o časové složitosti operací na poli (orání), bere se vzhledem k jeho velikosti – to v našem případě zřejmě není použitelná cesta, navíc přímo v zadání je dáno, že pole nemáme považovat za vstup.
2. Další možnost by byla měřit ji vzhledem k hodnotě hledaného čísla, které je skutečným vstupem, o což se někteří z vás pokusili. Žel bohu, čísla se mohou v poli opakovat, takže pro libovolně rychlý algoritmus hledání stejného nebo vyššího čísla vymyslím takové pole, že místo výsledku dostanete kopřivku.
3. Zbývá tedy použít index N hledaného čísla x v poli, byť ze zadání nejsme schopní o jeho velikosti nic říci.

b) Žádný algoritmus v principu nemůže být konečný, neboť u nekonečného pole plného jedniček nemáme šanci zjistit, že se v něm hledaná dvojka nevyskytuje. Proto naše odhady budou platit pouze pro případ, kdy algoritmus skončí.

c) Jak naučit krokodýla štěkat. (Haf!)

A jak tedy hledat?

Jediné co o posloupnosti čísel v poli vím, je, že je neklesající. Dále mohu dotazem zjistit konkrétní hodnotu v určité buňce. Z toho však nemohu nic usoudit o rychlosti růstu v neprobádaných oblastech – z tohoto důvodu byly všechny vaše snahy o nástřel pozice hledaného čísla na základě hodnot buněk marné (vzhledem k nejhoršímu možnému odhadu, a o ten nám jde). Zbývá nám využít monotonii – každý dotaz na buňku nám dává pouze informaci, ve které polovině pole vůči buňce se hodnota nalézá – zabývat se druhou částí pole je čiré plýtvání, neb se nic nového nedovíme.

Podtrženo sečteno, jde o to, jak se pomocí půlení intervalu co nejrychleji dostat k hledané hodnotě. Každý dotaz nám řekne, v které polovině pole hledat, což nás dovede k tomu, že první fáze algoritmu se týká nalezení stejné nebo vyšší hodnoty než je hledaná. Všimněte si, že pokud by se jednalo pouze o tuto úlohu, je otázka optimality neřešitelná – ke každému algoritmu najdu ještě rychlejší. Jenomže nám se bude stávat, že najdeme číslo, které je vyšší, a tak nastane druhá fáze – klasické binární vyhledávání v intervalu I ohraničeném posledními dvěma dotazy. Následuje další pozorování – čím rychlejší bude první fáze, tím větší budou intervaly mezi jednotlivými dotazy, a tedy i poslední interval pro druhou fázi (BÚNO předpokládám, že poslední interval není kratší než intervaly předchozí). S pomocí půlení intervalů – a nic víc dotazem na buňku nezjistím – jsem v i-tém kroku schopen zúžit prohledávaný interval I na $I/2^i$, což dává složitost $\log_2 I$ pro druhou fázi.

Nyní rozeberme složitosti pro vybrané strategie v první fázi:

1. Pokud budu hledat pravý konec intervalu přičítáním konstanty k ku indexu, bude nám první fáze trvat $O(N)$, druhá fáze $O(logk) = O(1)$; celkově $O(N)$.
2. Pokud budu hledat pravý konec násobením indexu konstantou k, bude první fáze trvat $\lceil \log_k N \rceil$ a druhá fáze dostane interval délky $\leq (k-1) \cdot N$, na kterém bude hledání trvat $O(\log_2(k-1) + \log_2 N)$; celkově tedy $O(\log N)$.
3. Pokud budu hledat pravý konec v čase $O(F(N))$, kde F je asymptoticky pomalejší než $\log N$, dostane druhá fáze interval I_F a bude trvat $O(\log I_F)$, což se bude rovnat $O(\log N)$ pouze v případě, že I_F není větší než mocnina N. Příkladem strategie, kdy se interval ještě vejde do mocniny, je násobení indexu sebou samým místo konstanty ($O(\log N^2) = O(\log N)$). Příkladem strategie, kdy se do mocniny nevejdeme, budiž výpočet dalšího indexu pomocí Ackermannovy funkce – v takovém případě se nám druhá fáze obecně dostane nad logaritmus.

Z hlediska asymptotického chování je optimum na složitosti $O(\log N)$, šťourové zajímající se o optimalitu počtu dotazů do pole bez zanedbávání multiplikativních konstant mohou začít zkoumat vzájemnou závislost F a I_F.

Někteří z vás řešili, jak je to s paměťovou složitostí, jestliže budu chtít ukládat velké indexy nekonečného pole. Uvědomte si, že tento problém nastává i za normální situace, kdy je velikost pole n – zjevně s velikostí n bude muset růst i rozsah hodnot, které jsme schopni uložit v registru. Na to jsou dvě možné odpovědi:

1. Standardní: předpokládáme nezáludnost programátora a do každého registru mu dovolíme uložit libovolně velké číslo. Nezáludnost spočívá v tom, že ho nenapadne do tohoto registru začít nějak kódovat další informaci – každý program bychom pak mohli spočítat pomocí konstantní paměťové složitosti.
2. Pro záludné přestaneme počítat paměťové buňky na čísla a začneme počítat bity. Každý registr pak dostane paměťovou složitost $\log n$; zvedne se také dosud konstantní složitost na atomické operace s registry atd. Celkově se proto zvýší odhady složitostí jednotlivých algoritmů, nicméně stále nám zůstane schopnost porovnávat rychlosti jednotlivých algoritmů mezi sebou.

V implementaci je použito přímo pole; považuji-li přístupy do něj za dotaz na uživatele, je paměťová složitost konstantní; nenechte se mýlit céčkem, pole začíná od indexu 1.

Pavel Šanda

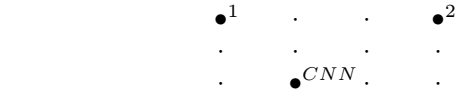
Šťoura se hlásí o slovo. Nejprve si uvědomme, že Šandíkův dvojfázový algoritmus je vlastně jediný možný: položíme-li první dotaz a zjistíme, že číslo v poli je menší než hledaná hodnota, nemá se smysl dále ptát na cokoliv před ní, analogicky pro další dotazy, takže musíme jít doprava, dokud nedostaneme větší hodnotu. Ale jakmile ji dostaneme, zase víme, že hledané číslo je mezi touto hodnotou a předchozím dotazem, a tady už je optimální binární vyhledávání.

Co se optimálního počtu dotazů týče: co to vlastně znamená optimální? Libovolný algoritmus můžeme přeci pro prvních n hodnot zlepšit až na $\lfloor \log_2(n-1) \rfloor + 2$ tím, že první dotaz bude na n-tý prvek a pokud je hledaná hodnota menší, spustíme půlení intervalu, jinak přepneme na původní algoritmus, který jsme tím na ostatních prvcích o konstantu zlepšili. Naopak pokud v uvedeném algoritmu s k-násobením zvolíme větší k, bude nám první fáze trvat $\log_2 k$-krát rychleji a druhou si tím zpomalíme jen o konstantu $\log_2(k-1)$, čili jsme algoritmus zrychlili všude až na prvních několik hodnot. Analogicky místo libovolné funkce F můžeme použít funkci o konstantu pomalejší. Docházíme tak k překvapivému závěru, že ke každému algoritmu můžeme najít takový, který pro vybrané hodnoty bude o něco rychlejší. Ale na druhou stranu, alespoň $\log_2 n$ je určitě potřeba, takže dokud nás nezajímají multiplikativní konstanty, je $O(\log n)$ dozajista optimální. Haf! –M.M.

16-4-3 Stávka programátorů

Stávková reportáž televize CNN nedopadla úplně tak, jak její majitel čekal. Byla to vlastně docela katastrofa. Jedni dali řidičovi filmového vozu radu, ať chvíli počká, že to hned v momentě vyřeší (a jestli neumřeli, řeší dodnes). Jiní ho instruovali, aby si vybral vždy cestu, která je nejkratší, takže ho všichni ostatní (lépe instruovaní) řidiči předjeli a reportáž televize CNN byla odvysílání s velkým zpožděním. Je tedy nutno říci, že stávka programátorů neměla náležitý dopad (když ji televize CNN nestihla během týdne okomentovat) a tak budou muset brát programátoři dál obrovské sumy peněz za tak lehkou práci.

Většina řešení radila řidiči, aby si mezi dvěma stávkami vybral cestu buď přímo vodorovně nebo přímo svisle. A z těchto dvou tu, která je kratší. To je bohužel řešení chybné a dostalo nula bodů. Můžeme si to ukázat na následujícím protipříkladě:



Pokud si vyberete nejkratší cestu na stávku 1, pojedete o jednu ulici doleva. Pak na stávku druhou pojedete o dvě

nahoru a pak o tři zpět do CNN, čili celkem šest. Ovšem pokud byste se ovšem vydali na začátku nahoru o dvě ulice, nafilmovali byste obě stávky a stačilo by se vrátit – celkem tedy pouze 4 ulice.

Úloha se dala řešit několika způsoby. Jeden z nich je prohledávání do šířky. Představme si ne jedno, ale M měst přesně nad sebou, přičemž v prvním je jen CNN a první stávka, v druhém je jen druhá stávka, ... , v M-tém městě je M-tá stávka a opět CNN. Pokud si představíme, že z města i se do $(i+1)$-ního dá dostat jen z ulic v městě i, ze kterých je vidět i-tá stávka, tak hledáme nejkratší cestu z CNN v prvním městě do CNN v městě M-tém. Tato nalezená cesta bude nejkratší a vzhledem k tomu, jak přecházíme mezi městy, bude u každé stávky. Toto řešení má časovou i paměťovou složitost úměrnou velikosti prohledávaného prostoru, čili $O(M \cdot N^2)$.

Jiný pohled na řešení může být tento: pro města i od M-tého k prvnímu si spočteme matici C_i, ve které budeme mít pro každou křižovatku v daném městě nejkratší vzdálenost cesty, která nafilmuje stávky $i, \ldots, M$ a vrátí se do CNN. Pokud tyto matice vzdáleností budeme počítat v popsaném pořadí od M-té k první, můžeme spočítat délku jedné nejkratší cestu (jeden prvek matice) v konstantním čase: v i-tém městě na křižovatce (x, y) vede hledaná nejkratší cesta buď horizontálně nebo vertikálně na ulici, ze které je možné nafilmovat i-tou stávku a a pak pokračuje dále (přičemž délku tohoto pokračování už známe). Čili délka této cesty (pokud i-tá stávka je na křižovatce (i, j)) je $min(abs(x-i) + C_{i+1}(i, y), abs(y-j) + C_{i+1}(x, j))$.

Tímto máme další řešení, které musí počítat $M \cdot N^2$ čísel, každé v konstantním čase. To není o nic lepší. Ale můžeme si všimnout jedné věci – hodnoty matice C_{i+1}, které při výpočtu skutečně použijeme, leží vždy jen na ulicích, ze kterých je vidět i-tá stávka. Takže vlastně nepotřebujeme počítat hodnoty v celé matici C_{i+1}, stačí pouze ty, ze kterých je vidět i-tou stávku. Těch je ale docela málo – přesněji $2N - 1$.

Z toho plyne následující řešení: pro každou stávku od M-té k první si spočteme délku nejkratší cesty, která nafilmuje stávky $i, \ldots, M$ a skončí v CNN. Tyto délky budeme ale počítat jen ve křižovatkách, ze kterých můžeme i-tou stávku nafilmovat. K výpočtu těchto délek nám poslouží již popsaný vzoreček.

Navíc pokud si u každé křižovatky budeme pamatovat, kudy z ní nejkratší cesta vede, můžeme nakonec i vypsat cestu vozu (a popředu – proto jsme hledali nejkratší cesty od poslední stávky). Celkem má naše řešení časovou i paměťovou složitost $O(M \cdot N)$. Paměťová by šla snížit na $O(N)$, ovšem jen pokud bychom nechtěli znát cestu, ale jen její délku.

Milan Straka

Další možnost, i když poněkud zběsilá: Jak víme, hodnoty, které potřebujeme, leží na „kříži“ † aktuální stávky (tak budeme říkat křižovatkám, ze kterých je tato stávka vidět) a počítáme je z hodnot na kříži †′ následující stávky. Každý kříž si ale můžeme rozdělit na svislou a vodorovnou část. Pak hodnoty na vodorovné části † budou hodnoty z vodorovné části †′, pouze zvýšené o vzdálenost obou přímek, a navíc ještě průmět všech hodnot ze svislé části kříže †′ do průsečíku příslušné svislé přímky s naši vodorovnou, který ohodnotíme minimem z (vzdálenost bodu od vodorovné přímky + ohodnocení bodu). To není o mnoho lepší řešení, ale jen do okamžiku, kdy si uvědomíme, že by se celá situace dala udržovat ve dvou vyhledávacích stromech – jednom pro svislý směr a jednom pro vodorovný –, přičemž podobně jako si v úloze 16-4-5 udržujeme minimální rozdíl, bychom si udržovali minimum z (souřadnice ve směru přímky + ohodnocení bodu), a tak bychom dokázali od jednoho kříže k druhému přejít v čase $O(\log N)$, dosahujíce tak celkové časové složitosti $O(M \cdot \log N)$. –M.M.

16-4-4 Dlouhoprstova zapeklitá hra

Většina z vás, zdá se, přála Dlouhoprstovi buďto předlouhý život nebo z pekla štěstí, jelikož většina řešení byla buďto ďábelsky pomalá (pracovala v exponenciálním nebo dokonce faktoriálovém čase) nebo to byly po čertech podivné heuristiky fungující pouze pro některé vstupy a pro jiné se chovající značně démonicky. Ale přeci jen někteří dokázali našemu hrdinovi (neříkám, že kladnému) s dlouhými prsty a krátkýma nohama pomoci. Jde to třeba takto:

Označme si $D(\alpha, \beta)$ nejkratší možnou posloupnost Dlouhoprstových karet (Dlouhoprstova posloupnost, zkráceně DP), která odpovídá Satanovým posloupnostem (SP) α a β. Podle toho, jak α a β začínají, můžeme rozlišit následující případy:

- $D(x\alpha, x\beta) = xD(\alpha, \beta)$ – pokud začínají obě SP stejným písmenem, musí DP začínat také tímto písmenem a za ním bude následovat DP pro původní SP bez tohoto písmena.
- $D(x\alpha, y\beta) = \Psi$ – pokud SP začínají různými písmeny, evidentně příslušná DP neexistuje, takže vrátíme chybu, a tu budeme značit Ψ.
- $D(\varepsilon, \varepsilon) = \varepsilon$ – pokud jsou obě SP prázdné, je DP také (ε budeme značit prázdný řetězec).
- $D(x\alpha, *\beta) = \min(D(x\alpha, \beta), D(\alpha, *\beta))$ – pokud se objeví hvězdička, můžeme ji buďto splnit prázdným řetězcem a nebo ji nechat „spolknout“ první písmenko druhého řetězce (případně i nějaká další, protože $*$ v řetězci ponecháme). Vybereme si samozřejmě kratší z obou variant (jako min značíme minimum řetězcové, které z dvou řetězců vrátí ten kratší a pokud mají stejnou délku, tak libovolný z nich; navíc Ψ je delší než všechny řetězce).
- $D(\varepsilon, *\beta) = D(\varepsilon, \beta)$ – pokud je levá strana prázdná, musí hvězdičce vyhovovat prázdný řetězec, ale ještě musíme pokračovat, protože napravo může být hvězdiček více.
- $D(*\alpha, x\beta)$, $D(*\alpha, \varepsilon)$ – analogicky.
- $D(*\alpha, *\beta) = \min(D(\alpha, *\beta), D(*\alpha, \beta))$ – pokud obě SP začínají na $*$, nemá smysl do DP přidávat znaky, které by měly vyhovovat oběma hvězdičkám – ty by byly zbytečné. Takže chceme jednu z hvězdiček vypustit a druhou ponechat, jen si musíme vybrat tu správnou, pročež zkusíme obě.
- pokud se někde vyskytne otazník, můžeme ho chápat jako písmeno, které se rovná libovolnému jinému písmenu a pokud bychom ho chtěli vypsat do výstupu, vypíšeme místo něj libovolné jedno písmeno.
- $D(x\alpha, \varepsilon)$ a ostatní zbylé případy (jedna SP došla a druhá ještě ne) jsou neřešitelné, a tak odpovíme Ψ.

Rekursivním použitím těchto pravidel už můžeme spočítat D pro libovolnou dvojici Satanových posloupností, ale má to jeden háček: rekurse se nám na hvězdičkách větví, takže může trvat exponenciálně dlouho. Jak z toho ven?

Všimneme si, že všechny řetězce, pro které D počítáme, jsou vždy suffixy původních SP (suffix je část řetězce od nějakého místa až do konce) a rekurse je exponenciální jen proto,

že se pro mnohé dvojice suffixů počítá totéž vícekrát. Budeme si proto již spočtené hodnoty pamatovat v pomocném poli a kdykoliv by po nás někdo chtěl hodnotu spočítat znovu, prostě ji jen vytáhneme z pole jako králíka z klobouku a hned se vrátíme bez dalšího rekursivního volání. Možných dvojic suffixů je jenom $(M+1)\cdot(N+1)$, takže netriviálních volání funkce D může být jen $O(M \cdot N)$.

Již z toho by plynul pěkný algoritmus se složitostí $(M \cdot N \cdot (M+N))$, ale ten by většinu času trávil předáváním řetězců mezi funkcemi. Tak se ho ještě zkusíme zbavit: Všimneme si, že každé $D(\alpha, \beta)$ vždy získáme z nějakého $D(\alpha', \beta')$ (kde α' je nějaký suffix řetězce α a obdobně β') přidáním jednoho nebo žádného znaku na začátek. Naše funkce tedy místo toho, aby vrátila řetězec, jen poznamená do nějakého pomocného pole, jak má hodnota vzniknout a jak bude dlouhá (to zvládneme na konstantní počet operací), a po ukončení výpočtu ten správný řetězec podle těchto poznámek zrekonstruujeme (v lineárním čase).

Celkově má tedy náš algoritmus časovou i paměťovou složitost $O(M \cdot N)$.

Martin Mareš

16-4-5 Obchodníci s deštěm

Velkou část našich řešitelů O. Š .Kubal, věren svému jménu, žel Bůhdhovi, o.š.kubal. Svými pomalými programy totiž nedokázali zjistit, jaký zmetek jim chtějí Kubalovi prodat.

První věci, které si všimneme, je to, že čas potřebný na jednu odpověď (vypsání aktuálního rozdílu po přečtení jednoho čísla) by neměl být závislý na N, ale jenom na K.

Nejjednodušší řešení je, po načtení další hodnoty, spočítat všechny vzdálenosti dvojic posledních K vrcholů a z nich si vybrat tu nejmenší. To určitě zvládneme v $O(N \cdot K^2)$.

Vylepšit to můžeme například tak, že si všimneme, že pokud bychom měli posledních K hodnot setříděných, nemusíme zkoumat $O(K^2)$ vzdáleností, stačí nám spočítat vzdálenosti mezi dvěma sousedními prvky (sousedí v *setříděném* poli). Těch už je jenom $K-1$, nicméně třídění nás stojí zase $O(K \log K)$. Celkem vylepšení na $O(N \cdot K \log K)$.

Další pozorování je, že po načtení jednoho čísla se pole posledních K čísel moc nezmění – určitě nemá cenu ho třídit vždy znova. Pokud máme setříděné pole posledních K čísel a načítáme další, stačí to nejstarší z pole vyhodit $(O(K))$ a nové přidat $(O(K))$ tak, aby pole zůstalo uspořádané. Pak stačí v jednom průchodu nad polem spočítat vzdálenosti sousedních prvků a vypsat nejmenší. Tím jsme na $O(N \cdot K)$.

Vylepšovat ale jde dále. Ukážeme si dvě možná řešení se složitostí $O(N \cdot \log K)$. První z nich je založeno na tomto pozorování: pokud uvažujeme o postupu s lineárním časem, tak počet dvojic, jejichž vzdálenost počítáme, se při načtení jednoho čísla mění velmi málo. Můžeme tedy mít všechny vzdálenosti sousedních prvků (sousedních v setříděném poli) v haldě. Při načtení nového čísla ho zatřídíme do nějaké struktury (použijeme např. AVL stromy z minulé kuchařky), která nám řekne jeho sousedy (většího a menšího) v setříděném poli. Pokud už je máme, z haldy odebereme vzdálenost těchto dvou sousedů a naopak do ní vložíme vzdálenost aktuálního prvku od menšího a vzdálenost aktuálního prvku od většího souseda. Při mazání čísla uděláme podobnou úpravu (zase si najdeme sousedy mazaného prvku, z haldy odebereme dvě hodnoty a dáme tam místo nich jednu [vzdálenost sousedů mazaného prvku]).

Pokud použijeme ke zjišťování sousedů nějaký druh vyvážených stromů (třeba AVL :–), můžeme hledání sousedů, vkládání a mazání provádět v čase $O(\log K)$. Stejnou složitost mají i operace s haldou – a protože všeho tohoto děláme konstantní počet, máme řešení se složitostí $O(N \cdot \log K)$.

To bylo jedno řešení, slíbili jsme ještě druhé: opět použijeme nějaký vyvážený binární strom. Každý jeho vrchol bude odpovídat jednomu z posledních K čísel, nicméně ve vrcholu si kromě hodnoty budeme pamatovat ještě tyto údaje:

- *min* – minimum hodnot v tomto podstromě.
- *max* – maximum hodnot v tomto podstromě.
- *delta* – nejmenší vzdálenost hodnot v tomto podstromě.

Pokud máme vrchol a známe tyto hodnoty u obou jeho synů, můžeme si spočítat i jeho hodnoty v konstantním čase:

- *min* – vezmeme minimum od levého syna.
- *max* – vezmeme maximum od pravého syna.
- *delta* – vezmeme minimum z delt levého a pravého syna, dále ze vzdálenosti hodnoty aktuálního vrcholu od maxima levého syna a ještě rozdíl hodnoty aktuálního vrcholu a minima pravého syna.

Můžeme tedy načtené hodnoty vložit do stromu, přepočítat popsané hodnoty a vypsat deltu kořene. Přepočítání hodnot můžeme provádět tak, že po vložení/smazání prvku budeme stromem procházet od vloženého/smazaného prvku směrem ke kořeni a po cestě upravovat popsané hodnoty. Pokud bude strom opravdu vyvážený, bude mít logaritmickou hloubku a tedy popsané operace budou mít složitost $O(\log K)$ a celé řešení tedy $O(N \cdot \log K)$.

Ve vzorovém řešení jsme schválně nepoužili AVL stromy, ty už znáte. Použili jsme tzv. BB-α stromy, které mají logaritmickou složitost pouze amortizovaně. To nám ale vůbec nevadí, protože nás zajímá složitost N operací a ne jedné.

BB-α strom je normální binární vyhledávací strom takový, že v každém vrcholu platí podmínka, že počet vrcholů v levém a pravém podstromě se liší nanejvíc α-krát. Takový strom má vždy logaritmickou hloubku, protože podstrom nějakého stromu má nanejvýš $\alpha/(\alpha+1)$ vrcholů – počet vrcholů v podstromu tak klesá geometrickou řadou a maximální možná výška stromu je tak $\log_{(\alpha+1)/\alpha} N$.

A jak takovou podmínku dodržet? U každého vrcholu si budeme udržovat počet vrcholů v levém a pravém podstromu. Pokud kdykoliv zjistíme, že se liší více než α-krát, celý podstrom odpojíme, vytvoříme z něj vyvážený strom a vrátíme zpátky. Takové „vybalancování“ určitě trvá lineárně vzhledem k počtu vrcholů ve vybalancovávaném stromečku.

Předpokládejme nyní, že $\alpha = 2$, stejně jako ve vzorovém programu. Kolik stojí jedno vkládání či mazání? Na to, aby se nějaké vybalancování spustilo, se musí lišit hodnoty v levém a pravém podstromu dvakrát, čili od minulého rebalancování muselo dojít k řádově tolika vkládáním a mazáním, kolik je vrcholů ve zkoumaném stromečku. Čili stačilo, aby každé vkládání a mazání přispělo aktuálnímu vrcholu konstantním časem (jedním penízkem), ze kterého se pak vybalancování „uplatí“. Každé vkládání a mazání musí přispět na rebalancování všem vrcholům, přes které projde. Těch je ale nanejvíc tolik, jaká je výška stromu – a ta je logaritmická. Čili amortizovaná složitost vkládání nebo mazání prvku je $O(\log K)$ (amortizovaná znamená, že i když nevíme, jak dlouho bude jedna operace doopravdy trvat, N operací bude trvat nejvýš $O(N \cdot K)$).

Milan Straka

16-4-6 Šikmá věž v Kocourkově

Řešení potíží našich věžechtivých Kocourkovanů v kostce: Klíčové úkony leží na nejdelší cestě v závislostním grafu (acyklickém!) a všechny takové cesty můžeme najít projitím grafu v topologickém pořadí.

Připusťme ale poněkud pomalejší chápání kocourkovských buřtipánů a zkusme tento nápad poněkud rozvést:

Nejdříve si sestrojíme *závislostní graf:* to bude orientovaný graf, jehož vrcholy budou odpovídat úkonům a z u do v povede hrana právě tehdy, když je nutné úkon u dokončit před započetím úkonu v; navíc si každý vrchol ohodnotíme číslem, které bude udávat, jak dlouho příslušný úkon trvá. Ještě přidáme počáteční úkon α odpovídající položení základního kamene stavby (ohodnocena nulou a vedou z ní hrany do všech ostatních úkonů) a úkon ω odpovídající kolaudaci (opět nula [naivní, vím], hrany ze všech ostatních úkonů). *Délkou cesty* budeme nazývat součet ohodnocení vrcholů, které na ni leží, *bez počátečního a koncového vrcholu,* což je sice trochu nesystematické, ale usnadní nám to počítání. *Odlehlostí* z vrcholu (úkonu) u do v pak nazveme délku *nejdelší cesty* z u do v (to je svým způsobem opak vzdálenosti, která je délkou cesty nejkratší).

Pokud je v závislostním grafu cyklus, věž evidentně postavit nelze. Budeme tedy předpokládat, že graf je acyklický a ukážeme, že věž postavit půjde, a to v čase rovném odlehlosti L z α do ω. Ještě trocha značení: pro každý úkon u bude $l(u)$ délka tohoto úkonu, $a(u)$ odlehlost z α do u a $z(u)$ odlehlost z u do ω.

Teď ale na chvíli odbočíme od tématu a zavedeme si *topologické pořadí* vrcholů grafu. Tak budeme říkat takovému pořadí $v_1, v_2, \ldots, v_n$ všech vrcholů, pro které platí, že pokud vede hrana z v_i do v_j, je vždy $i < j$. V libovolném acyklickém grafu takové pořadí existuje, což si dokážeme tak, že si rovnou předvedeme lineární algoritmus, který ho najde.

Pokud je graf acyklický, musí v něm existovat vrchol v_1, kam nevede žádná hrana: stačí vyjít z libovolného vrcholu a stále jít proti směru hran – jelikož graf je konečný, nemůžeme přicházet do stále nových vrcholů, takže časem narazíme buďto na vrchol, do kterého nic nevede, nebo na vrchol, ve kterém jsme už byli, což ovšem není možné, protože bychom tím uzavřeli cyklus. Nalezený vrchol v_1 očíslujeme jedničkou (to je určitě správně, nevede do něj přeci žádná hrana) a z grafu ho odebereme včetně všech hran, které z něj vedou. Tím získáme opět acyklický graf a v něm pokračujeme stejně od dvojky. Až nezbude žádný vrchol, budeme mít hotové topologické pořadí; pokud by nějaký zbyl, znamená to, že se v grafu nacházely cykly. Abychom ale dosáhli lineární časové složitosti, potřebujeme umět takové vrcholy nacházet v konstantním čase. K tomu stačí zapamatovat si pro každý vrchol počet hran, které do něj vedou, při odebírání hran tyto počty aktualizovat a udržovat si frontu vrcholů, které už mají nulu, ale ještě jsme je neodebrali.

[Ve vzorovém programu nepřiřazujeme čísla explicitně, ale využíváme toho, že ve frontě jsou vrcholy uloženy také v topologickém pořadí. Ještě jiný způsob, jak topologické pořadí najít, by byl prohledat graf do hloubky a všimnout si, že pořadí, ve kterém se z vrcholů vracíme, je obrácené topologické pořadí. Zkuste si rozmyslet, proč to platí, v některé z kuchařek v příštím ročníku se na to podíváme podrobněji.]

Hodnoty $a(u)$ pak můžeme spočítat velice snadno indukcí: bereme vrcholy v topologickém pořadí, začínáme s $a(\alpha) = 0$. Pro každý další vrchol využijeme toho, že již známe $a(v)$ pro všechny předchůdce v vrcholu u, čili vrcholy, z nichž vede do u hrana, a položíme $a(u) = \max_v(a(v)+l(v))$ (nejdelší cesta z α do u musí být nutně nejdelší cestou do některého z předchůdců u prodloužená o hranu do u). To zvládneme v lineárním čase a stejně tak můžeme spočítat i $b(u)$ pomocí opačného topologického pořadí a L jako $a(\omega)$ nebo $z(\alpha)$.

(Konec odbočky.) Nyní si všimněme, že žádný úkon nelze začít provádět dříve než v čase $a(u)$: víme totiž, že existuje posloupnost úkonů, které musí být všechny provedeny po sobě a před u a dohromady trvají $a(u)$. Z toho také plyne, že celou věž nemůžeme dostavět dříve než za $L = a(\omega)$. Teď už stačí ukázat, že si můžeme úkony rozvrhnout tak, abychom u dokončili v čase $a(u)+l(u)$, a tedy celou stavbu v čase L. To je snadné: každý úkon u začneme provádět v čase $a(u)$ a vzorec pro $a(u)$ z minulého odstavce vlastně říká přesně to, že všechny předchozí úkoly jsou nejpozději v tomto okamžiku hotové.

Zbývá ještě zjistit, které úkoly jsou klíčové: jsou to ty, které leží na některé z nejdelších cest (tedy ty, pro které je $L_u = L$, kde $L_u = a(u) + b(u) + l(u)$ je délka nejdelší cesty z α do ω, která obsahuje u). Pokud úkon u na nejdelší cestě leží, je nutně klíčový, protože na úkonech na nejdelší cestě pracujeme po celou dobu L bez přestávek, takže pokud libovolný úkon prodloužíme, prodloužíme i celou dobu stavby. Naopak pokud úkon u neleží na nejdelší cestě, můžeme ho prodloužit až o $L - L_u$ a celkovou dobu tím nezměníme.

Program je toliko formálním zápisem našich úvah a má lineární časovou složitost a kvadratickou paměťovou (ale jen proto, že jsme si ušetřili práci s načitáním hran; jinak by byla také lineární).

Martin Mareš

[Všchn thle b s smzřjm dlo smrsknt d jdnh prchdu grfm, le bl b t čtlné as jko thl vta. –M.M.]

Úloha 16-4-1 – Mnichova posedlost aneb Hanoi strikes back

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

int N, kolik[MAX_N];                          /* na jakém kolíku je i-tý disk */
unsigned long long min_tahu[MAX_N+1][3];      /* jak přemístit disky od i-tého na lib. kolík */

int main (void) {
  int k, i, disk;

  printf ("Zadejte počet disků:");
  scanf ("%d", &N);
  for (k=0; k<3; k++) {
    printf ("Zadejte disky na kolíku %d:", k+1);
    while (scanf ("%d", &i), i) kolik[i-1]=k;
  }
```

```
   for (disk=N−1; disk>=0; disk−−)
     for (k=0; k<3; k++)                          /* dáme disky od „disk“-tého na k-tý kolík */
       if (kolik[disk]==k) {                      /* disk je už na místě */
         min_tahu[disk][k]=min_tahu[disk+1][k];
       } else {                                   /* disk není na místě */
         i=3−k−kolik[disk];                       /* pomocný kolík */
         min_tahu[disk][k]=min_tahu[disk+1][i] + (1≪(N−disk−1));
       }
   printf (“Nejlepší to bude na kolík %d, celkem %llu přesunů.\n”, kolik[0]+1, min_tahu[0][kolik[0]]);
   return 0;
}
```

A ještě jedna lahůdka v céčku (pouze do $2^{32}-1$ přesunů, na vstupu jsou čísla kolíků $(0,1,2)$ a čtou se ze vstupu všechna):

```
int n,a,k,_;main(){scanf("%d",&a);while(scanf("%d",&k)>0)_+=_,k!=a?_++,a=3-a-k:0;printf("%d\n",_);}
```

Úloha 16-4-2 – Technologické trable – program

```
int main (void) {
   int l=1, r=1, i;                             /* left, right, index */
   int w[4]={0, 1, 2, 3};                       /* nekonečné pole :-) */
   int x=2;                                     /* hodnota, kterou hledáme */
   while (w[r]<x) l=r+1, r*=2;                  /* pravá zarážka */
   while (l<=r) {                               /* binární vyhledávání */
    i = (l+r)/2;
    if (w[i]==x) return i;
    if (w[i]>x) r=i−1;
    if (w[i]<x) l=i+1;
   }
   return −1;
}
```

Úloha 16-4-3 – Stávka programátorů – program

```
#include <stdio.h>
#define MAX_N 100
#define MAX_M (MAX_N*MAX_N)
#define min (a, b) ( ( (a) < (b)) ? (a) : (b))
#define abs (a) ( ( (a) > 0) ? (a) : − (a))
enum {ROW, COL};

struct krizovatka {
   int x, y;
};

struct krizovatka cnn;
struct krizovatka stavky[MAX_M];                /* pole se stávkami */
int delky[MAX_M][2][MAX_N+1];                   /* délky do řádku a sloupce jdoucí skrz stávku */
int M, N;

int main (void) {
   int i, stavka;
   struct krizovatka m;
   int m_delka;
   printf (“Zadejte N a M:”);
   scanf (“%d %d”, &N, &M);
   printf (“Zadejte souřadnice CNN:”);
   scanf (“%d %d”, &cnn.x, &cnn.y);
   for (i=0; i<M; i++) {
     printf (“Zadejte souřadnice %d. stávky:”, i+1);
     scanf (“%d %d”, &stavky[i].x, &stavky[i].y);
   }
   for (i=1; i<=N; i++) {                       /* počáteční nastavení délky cest do CNN */
     delky[M−1][ROW][i]=abs (i−cnn.x)+abs (stavky[M−1].y−cnn.y);
     delky[M−1][COL][i]=abs (stavky[M−1].x−cnn.x)+abs (i−cnn.y);
   }
```

```
  for (stavka=M−2; stavka>=0; stavka−−) {                    /* pro všechny stávky */
    for (i=1; i<=N; i++) {
      delky[stavka][ROW][i]=min (abs (stavky[stavka].y−stavky[stavka+1].y)+delky[stavka+1][ROW][i],
                                 abs (i−stavky[stavka+1].x)+delky[stavka+1][COL][stavky[stavka].y]);
      delky[stavka][COL][i]=min (abs (stavky[stavka].x−stavky[stavka+1].x)+delky[stavka+1][COL][i],
                                 abs (i−stavky[stavka+1].y)+delky[stavka+1][ROW][stavky[stavka].x]);
    }
  }
  printf ("CNN−>");                                          /* výpis trasy */
  m=cnn;
  m_delka=min (abs (cnn.y−stavky[0].y)+delky[0][ROW][cnn.x],
               abs (cnn.x−stavky[0].x)+delky[0][COL][cnn.y]);
  for (stavka=0; stavka<M; stavka++) {
    if (delky[stavka][ROW][m.x]+abs (m.y−stavky[stavka].y) == m_delka) m.y=stavky[stavka].y;
    else m.x=stavky[stavka].x;
    printf ("(%d,%d)−>", m.x, m.y);
  }
  printf ("CNN\n");
  return 0;
}
```

Úloha 16-4-4 Dlouhoprstova zapeklitá hra

```
#include <stdio.h>

#define MAX 100                                              /* sizeof(Hell) */
#define INF 100000                                           /* Nekonečno */
char a[MAX], b[MAX];                                         /* Satanovy řetězce, tradičně ukončené kódem 0 */
int opt[MAX][MAX];                                           /* Optimální délka řetězce pro danou dvojici suffixů SP */
int optx[MAX][MAX], opty[MAX][MAX];                          /* Poznámky ke konstrukci řetězců */
char optc[MAX][MAX];

int D (int x, int y)                                         /* Spočte funkci D pro zadané suffixy SP */
{
  int ret;                                                   /* Budoucí výsledek */
  int bx=0, by=0;                                            /* Odkud jsme ho vzali */
  int bc=0;                                                  /* A co musíme připsat do výstupu, aby vznikl */
  if (opt[x][y] >= 0)                                        /* Králík z klobouku? */
    return opt[x][y];
  /* Dvě pomocná makra: nastavení výsledku a pokus o jeho vylepšení */
#define SET (xx, yy, cc) do { ret=D (xx, yy); bx=xx; by=yy; bc=cc; } while (0)
#define UPDATE (xx, yy, cc) do { int r2=D (xx, yy); if (r2 < ret) { ret=r2; bx=xx; by=yy; bc=cc; } } while (0)
  if (!a[x] && !b[y])                                        /* Oba řetězce skončily */
    ret = 0;
  else if (a[x] == '*'&& b[y] == '*') {                      /* Dvě hvězdičky */
      SET (x+1, y, 0);
      UPDATE (x, y+1, 0);
  }
  else if (a[x] == '*') {                                    /* Hvězdička vlevo ... */
      SET (x+1, y, 0);
      if (b[y])
        UPDATE (x, y+1, b[y]);
  }
  else if (b[y] == '*') {                                    /* ... nebo vpravo */
      SET (x, y+1, 0);
      if (a[x])
        UPDATE (x+1, y, a[x]);
  }
  else if (a[x] == b[y])                                     /* 2 stejné znaky */
    SET (x+1, y+1, a[x]);
  else if (a[x] == '?'&& b[y])                               /* Otazníky */
    SET (x+1, y+1, b[y]);
  else if (a[x] && b[y] == '?')
    SET (x+1, y+1, a[x]);
  else                                                       /* Jinak nemožno */
    ret = INF;
```

```
    optx[x][y] = bx;
    opty[x][y] = by;
    optc[x][y] = bc;
    return opt[x][y] = ret;
}

int main (void)
{
    int i, j, l;
    scanf ("%s%s", a, b);
    for (i=0; i<MAX; i++)
      for (j=0; j<MAX; j++)
        opt[i][j] = -1;
    if (D (0, 0) >= INF) {
        printf ("Pomooooc! Prohraavaaaaam!\n");
        return 0;
    }
    int x=0, y=0, nx;                                   /* Rekonstrukce výsledného řetězce */
    while (a[x] && b[y]) {
        if (optc[x][y])
          putchar (optc[x][y] == '?' ? 'x': optc[x][y]);
        nx = optx[x][y];
        y = opty[x][y];
        x = nx;
    }
    putchar ('\n');
    return 0;
}
```

Úloha 16-4-5 – Obchodníci s deštěm – program

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

struct node {
    int value;
    int min, max, delta;
    struct node *left, *right, *parent;
    int left_c, right_c;
};

struct node *root=NULL;                                 /* kořen stromu */
int hodnoty[MAX_K];                                     /* posledních K načtených hodnot */
int N, K;

void bbtree_oprav (struct node *);                      /* nadeklarujeme až později */

struct node *bbtree_find (struct node *ptr, int value) {
    if (value < ptr->value && ptr->left) return bbtree_find (ptr->left, value);
    if (value > ptr->value && ptr->right) return bbtree_find (ptr->right, value);
    return ptr;
}

void bbtree_insert (int value) {                        /* vloží vrchol do stromu */
    struct node *new=malloc (sizeof (struct node)), *ptr;
    new->value=new->min=new->max=value;
    new->left=new->right=new->parent=NULL;
    new->delta=new->left_c=new->right_c=0;

    if (!root) root=new;                                /* ještě nemám root */
    else {
      ptr=bbtree_find (root, value);
      new->parent=ptr;                                  /* napojíme na rodiče */
      if (value < ptr->value) ptr->left=new; else ptr->right=new;       /* a rodiče na mě */
      bbtree_oprav (new);
    }
}
```

```
void bbtree_remove (int value) {                                /* odebere vrchol */
  struct node *ptr=bbtree_find (root, value), *son;

  if (!ptr || ptr->value != value) return;                      /* nenašli jsme – to se nám nestane */
  if (ptr->left && ptr->right) {                                /* mám oba syny – najdi za sebe náhradu */
    struct node *nahrada= (ptr->left_c>ptr->right_c) ? ptr->left : ptr->right;
    while ( (ptr->left_c>ptr->right_c) ? nahrada->right : nahrada->left)
      nahrada= (ptr->left_c>ptr->right_c) ? nahrada->right : nahrada->left;
    ptr->value=nahrada->value;
    ptr=nahrada;
  }

  son= (ptr->left) ? ptr->left : ptr->right;
  if (son) son->parent=ptr->parent;                             /* úprava syna */
  if (!ptr->parent) root=son;                                   /* úprava rodiče */
  else if (ptr->parent->left==ptr) ptr->parent->left=son; else ptr->parent->right=son;

  if (ptr->parent) bbtree_oprav (ptr->parent);
  free (ptr);
}
void bbtree_update_vrcholu (struct node *ptr) {                 /* přepočítá min, max, delta, left_c a right_c */
#define test_delta (a) if ( (a) && (ptr->delta==-1 || (a) < ptr->delta)) ptr->delta= (a);
  ptr->min= (ptr->left) ? ptr->left->min : ptr->value;
  ptr->max= (ptr->right) ? ptr->right->max : ptr->value;
  ptr->delta=-1;                                                /* delta neinicializovaná */
  if (ptr->left) {                                              /* pomůže mi levý syn? */
    test_delta (ptr->left->delta);
    test_delta (ptr->value - ptr->left->max);
  }
  if (ptr->right) {                                             /* a co pravý syn? */
    test_delta (ptr->right->delta);
    test_delta (ptr->right->min - ptr->value);
  }
  if (ptr->delta<0) ptr->delta=0;
  ptr->left_c= (ptr->left) ? ptr->left->left_c + 1 + ptr->left->right_c: 0;
  ptr->right_c= (ptr->right) ? ptr->right->left_c + 1 + ptr->right->right_c: 0;
}

  /* tohle vytvoří spoják z vrcholů v stromu s vrcholem ptr a vrátí to první prvek p */
  /* spoják je svázaný ->left pointerama, jenom p->right je konec spojáčku */
  /* a p->right_c je počet prvků ve spojáčku */
struct node *bbtree_collect (struct node *ptr) {
  struct node *begin;
  if (ptr->left) {                                              /* někdo vlevo? */
    begin=bbtree_collect (ptr->left);
    begin->left->right=ptr;
  } else {begin=ptr; begin->right_c=0; }
  begin->left=ptr;
  begin->right_c++;
  if (ptr->right) {                                             /* a co vpravo? */
    begin->left->right=bbtree_collect (ptr->right);
    begin->right_c+=begin->left->right->right_c;
    begin->left=begin->left->right->left;
  }
  return begin;
}

  /* tohle z výše popsaného spojáčku udělá vyvážený strom */
  /* parametr next ukazuje na první dosud nepoužitý prvek z popsaného seznamu */
struct node *bbtree_rebuild (struct node *spojak, struct node **next) {
  int l=spojak->right_c/2;                                      /* počet vrcholů vlevo */
  int r=spojak->right_c-l-1;                                    /* a vpravo */
  struct node *left=NULL, *right=NULL;

  if (l) {spojak->right_c=l; left=bbtree_rebuild (spojak, next); spojak=*next; }
  *next=spojak->right;                                          /* toto je momentálně první nepoužitý prvek */
  if (r) {spojak->right->right_c=r; right=bbtree_rebuild (spojak->right, next); }
```

```
  spojak->left=left;                                  /* spoják bude kořen */
  if (left) left->parent=spojak;                      /* upravit pointery na syny a rodiče */
  spojak->right=right;
  if (right) right->parent=spojak;
  bbtree_update_vrcholu (spojak);
  return spojak;
}

 /* tato funkce se stará o opravu hodnot min,max a delta ve stromě */
 /* podle potřeby přestavuje strom funkcemi bbtree_collect a bbtree_rebuild */
void bbtree_oprav (struct node *ptr) {
  struct node *parent=ptr->parent;
  bbtree_update_vrcholu (ptr);
  if (ptr->left_c/2 > ptr->right_c || ptr->right_c/2 > ptr->left_c) {      /* přestavba stromu? */
    struct node *tmp;
    tmp=bbtree_rebuild (bbtree_collect (ptr), &tmp);
    if (parent && parent->value > ptr->value) parent->left=tmp;
    else if (parent && parent->value < ptr->value) parent->right=tmp;
    else root=tmp;
    tmp->parent=parent;
  }
  if (parent) bbtree_oprav (ptr->parent);             /* propagace výš */
}

int main (void) {
  int i;
  printf ("Zadejte N a K:");
  scanf ("%d %d", &N, &K);
  for (i=0; i<N; i++) {
    int v=i%K;
    if (i >= K) bbtree_remove (hodnoty[v]);           /* budeme mazat? */
    scanf ("%d", hodnoty+v);
    bbtree_insert (hodnoty[v]);
    if (i) printf ("Aktuální nejmenší rozdíl je %d.\n", root->delta);
  }
  return 0;
}
```

Úloha 16-4-6 – Šikmá věž v Kocourkově – program

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

#define MAXN 100

int N;                                                /* Počet úkonů: úkon 0 je α, N je ω */
int l[MAXN];                                          /* Délky úkonů */
int a[MAXN], z[MAXN];                                 /* Délky maximálních cest (viz popis) */
int edges[MAXN][MAXN];                                /* Hrany vedoucí z jednotlivých vrcholů */
int outdeg[MAXN], indeg[MAXN];                        /* Vstupní a výstupní stupeň vrcholů */
int top[MAXN];                                        /* Topologické pořadí vrcholů */

void add_edge (int v, int w)                          /* Přidá hranu do závislostního grafu */
{
  edges[v][outdeg[v]] = w;
  outdeg[v]++;
  indeg[w]++;
}

void read (void)                                      /* Přečte vstup a vytvoří graf */
{
  int v, w;
  scanf ("%d", &N); N++;                              /* Počet úkonů a jejich ceny */
  for (v=1; v<N; v++)
    scanf ("%d", &l[v]);
  while (scanf ("%d%d", &v, &w) == 2 && v > 0)        /* Závislosti, ukončeny (0,0) */
    add_edge (v, w);
```

```
    for (v=1; v<N; v++) {                    /* Závislosti pro vrcholy α a ω */
        add_edge (0, v);
        add_edge (v, N);
    }
}
void topsort (void)                          /* Topologicky setřídí graf */
{
    int i, r, w, u, v;
    top[0] = 0;                              /* První je vrchol α */
    r = 0;
    w = 1;
    while (r < w) {                          /* Probíráme frontu vrcholů (v)stupně 0 */
        u = top[r++];
        for (i=0; i<outdeg[u]; i++) {        /* Projdeme všechny hrany */
            v = edges[u][i];
            if (!--indeg[v])                 /* Snížime stupeň cílového vrcholu */
              top[w++] = v;                  /* a pokud je 0, přidáme do fronty */
        }
    }
    if (w != N+1) {                          /* Objevili jsme cyklus */
        printf ("Nelze.\n");
        exit (0);
    }
}
void paths (void)                            /* Spočte délky nejdelších cest */
{
    int i, j, v, w;
    for (i=0; i<=N; i++) {                   /* a[v] počítáme popředu */
        v = top[i];
        for (j=0; j<outdeg[v]; j++) {
            w = edges[v][j];                 /* hrana (v, w) */
            if (a[w] < a[v]+l[v])
              a[w] = a[v]+l[v];
        }
    }
    for (i=N; i>0; i--) {                    /* z[v] zase pozpátku */
        v = top[i];
        for (j=0; j<outdeg[v]; j++) {
            w = edges[v][j];                 /* hrana (v, w) */
            if (z[v] < z[w]+l[w])
              z[v] = z[w]+l[w];
        }
    }
}
void answer (void)                           /* Vypíše odpověď */
{
    int i;
    printf ("Věž je možno postavit za %d TUK.\n", a[N]);   /* Time Unit of Kocourkov */
    for (i=1; i<N; i++)
      if (a[i] + z[i] + l[i] == a[N])
        printf ("Úkon %d je klíčový.\n", i);
}
int main (void)
{
    read ();
    topsort ();
    paths ();
    answer ();
    return 0;
}
```

Konečná výsledková listina šestnáctého ročníku KSP

		škola	ročník	1641	1642	1643	1644	1645	1646	suma	úloh	celkem
1.	Peter Perešíni	GJGTajov	2	9	6	10	10	10	10	55	20	180
2.	Miroslav Cicko	GJGTajov	3	10	8	10	7	10	6	51	21	177
3.	Petr Škoda	GÚstavníPH	4	10	6	11	9	10	10	56	21	175
4.	Jan Bulánek	G Klatovy	3		8	9		10	9	36	17	147
5.	Kryštof Hoder	GKptJaroBO	4		8	8			10	26	13	120
6.	Jana Kravalová	G VKlobou	4	3	6	8		8	9	34	18	113
7. – 8.	Ondřej Bílka	G Zlín	2	7	8			7	8	30	15	103
	Miroslav Klimoš	G Lanškr	0		4	4	0	10	7	25	18	103
9.	David Matoušek	GZborovPH	4							0	12	93
10.	Pavel Klavík	G Chrudim	1	1	8	3	1	8	9	30	21	92
11.	Marek Jančuška	G Nitra	4							0	9	88
12.	Jan Hrnčíř	GFXŠaldy	2		4	3	2	6	9	24	19	86
13.	Ondřej Garncarz	G Příbor	3	2	6	3	1	5	2	19	22	84
14.	Michal Repovský	G Trebišov	4							0	12	83
15.	Pavel Motloch	GPBezruče	1							0	12	77
16.	Zbyněk Falt	GNeumZdar	3		6	0		6	5	17	12	74
17.	Marek Blahuš	G UHradi	3		8			10		18	10	72
18.	Jana Fabriková	GKptJaroBO	4					2	2	4	14	66
19.	Martin Koníček	G UBrod	3		7			6		13	11	62
20.	Martin Podloucký	G Strážnic	3		5					5	11	60
21.	Eva Schlosáriková	G Piešťany	3		4	8		2	1	15	14	58
22. – 24.	Martin Čech	G UBrod	3		6		0	7		13	13	56
	Petr Kratochvíl	G Světlá	1	3	6	2	0	7	8	26	12	56
	Filip Šauer	G Klatovy	3		7	2		3	1	13	13	56
25.	Peter Šufliarsky	G NZámky	4		6	0				6	11	55
26. – 27.	Stanislav Basovník	G Kroměříž	3	5	8			7	7	27	7	51
	Martin Dobroucký	G MTřebová	3		7		1	9	8	25	8	51
28.	Stanislav Haviar	G Klatovy	3							0	8	50
29.	Michal Bečka	G MTřebová	4							0	7	48
30.	Peter Černo	GĽŠtúra	3							0	6	46
31. – 32.	Petr Kortánek	G Sedlča	2		3	0		4	1	8	11	45
	Martina Tomisová	GZborovPH	4		4	0			1	5	13	45
33. – 34.	Jindřich Flídr	G Lanškr	4		3			6	1	10	7	43
	Tomáš Gavenčiak	G Bílovec	4							0	5	43
35.	Daniel Marek	GZborovPH	2							0	5	42
36.	Petr Soběslavský	GJHeyrovPH	3					3		3	14	41
37. – 38.	Cyril Hrubiš	G Bílovec	2		5		0	4	1	10	11	37
	Martin Křivánek	GKptJaroBO	2							0	5	37
39.	Ján Zahornadský	GZborovPH	3							0	8	36
40.	Petr Švec	G Beroun	4							0	8	35
41.	Jaroslav Havlín	G Sedlča	4							0	5	32
42.	Adam Přenosil	GSladkovPH	2							0	7	31
43.	Marek Ludha	GJGTajov	4							0	5	30
44. – 45.	Jan Křetínský	GMLerchaBO	4							0	3	24
	Martin Kupec	GMendel	2							0	8	24
46.	Benjamin Vejnar	G Nymburk	4		3					3	3	23
47. – 48.	Miroslav Kratochvíl	G Čáslav	2		4		0	3		7	6	17
	Petr Musil	G MBuděj	2							0	6	17
49.	Milan Dvořák	G NMnMor	1							0	3	16
50.	Zbyněk Konečný	GKptJaroBO	1					6		6	2	15
51. – 52.	Jiří Bělohradský	SŠAK HrKr	2							0	2	13
	Jan Richter	G Příbor	3							0	5	13
53. – 54.	Jiří Cabal ml.	SPŠDvKrál	1	3		0	0	3	6	12	5	12
	Kristýna Knapová	G Jičín	4							0	2	12
55.	Michal Potfaj	G NMnVáh	4							0	4	10
56.	Martin Schmid	G ČTřebová	0							0	3	9
57.	Jindřich Pergler	G Klatovy	3							0	2	8
58. – 60.	David Irschik	G Ledeč	3							0	2	7
	Petr Paščenko	GDašická	4							0	2	7
	Jaromír Vojíř	G Ledeč	3							0	3	7
61.	Radoslav Sopoliga	G Svidník	4							0	4	6
62.	Tomáš Herceg	G Třebíč	1							0	1	4
63. – 64.	Aleš Razým	SpGTáborPL	3							0	1	2
	Pavel Vlašánek	SPŠ Brunt	3		2					2	1	2