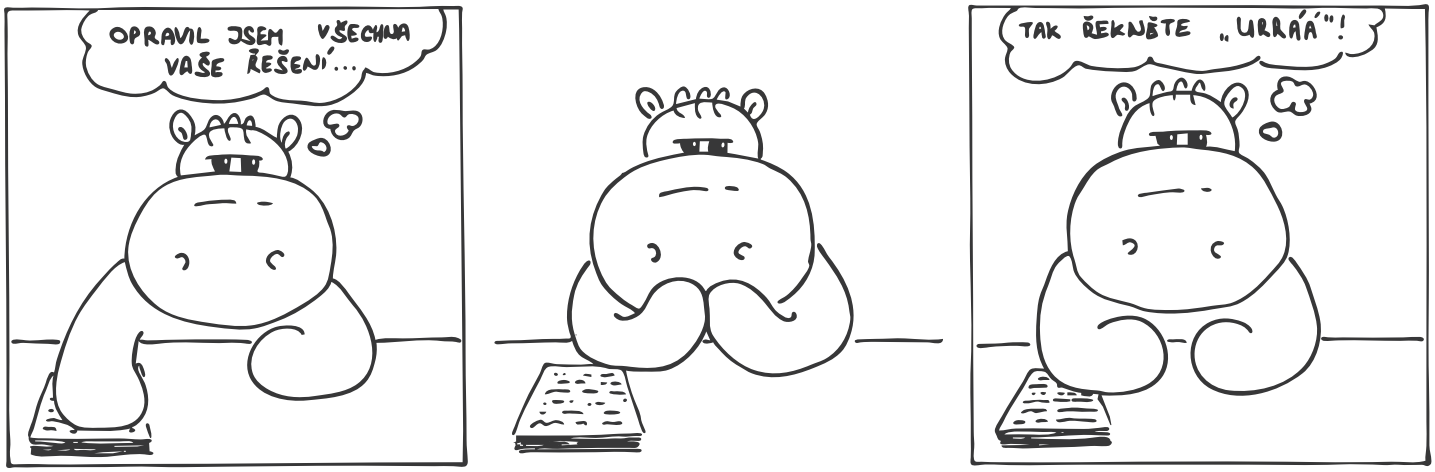
Milí řešitelé!

Touto opravenou pátou sérií jsme zakončili devatenáctý ročník našeho semináře. Na podzim bude samozrejmě ještě soustředění pro naše nejlepší řešitele, kam pozveme přibližně třicet řešitelů z první padesátky. Termín soustředění ještě není znám, ale do konce června bude zveřejněn na naší webové stránce.

Ještě bychom Vás chtěli, jako už několikrát, poprosit: myslíme si, že je škoda, že se našeho semináře účastní tak málo řešitelů. Budeme rádi, když nám na začátku příštího roku pomůžete s „reklamou“ – například pověšením zadání první série na školní nástěnku, ...

Aktuální informace o KSP naleznete na stránkách *http://ksp.mff.cuni.cz/*. Dotazy ohledně zadání můžete posílat na adresu *ksp@mff.cuni.cz*, nebo se ptát přímo na diskusním fóru KSP (*http://ksp.mff.cuni.cz/forum/*).



Vzorová řešení páté série devatenáctého ročníku KSP

19-5-1 Útěk

Na začátek jedna metoda jak neutíkat: Je pravda, že síť MHD připomíná graf, ale udělat co stanici, to uzel a co spoj, to hranu opravdu nejde. Pokud bychom nebyli limitováni penězi (nebo nebyli limitováni časem), Dijkstrův algoritmus by fungoval. Jenže my jsme limitováni obojím – a ani „projdeme všechny cesty“, ani „když dojdou peníze, zkusíme to jinak“ k řešení v polynomiálním čase nevedou.

Je možné stvořit jakýsi „časoprostorový“ graf, ve kterém každá zastávka bude zastoupena tolika vrcholy, kolik z ní odjíždí spojů, a hrany budou reprezentovat autobusy (s váhou ohodnocenou dolárky) a čekání (s váhou nulovou). Na časoprostorovém grafu již Dijkstrův algoritmus najde potřebné spojení, a to i v rozumném čase. (Ano, mohli bychom stvořit i „dolaroprostorový“ graf.)

Programovat Dijkstrův algoritmus se všemi optimalizacemi je ovšem dost práce, a navíc je náš časoprostorový graf poměrně speciálního tvaru. Nabízí se proto jednodušší řešení:

Události buďtež odjezdy autobusů a jejich příjezdy. Setřídíme je podle času vzestupně (umíme to v konstantním čase, protože minut ve dni je 1440). U každé linky a každé stanice si budeme pamatovat, jestli je dostupná *v tomto čase*, a případně za jakou cenu. Nedostupným linkám-stanicím můžeme prostě přiřadit nekonečno dolárků.

Výpočet bude probíhat tak, že půjdeme s časem od půlnoci dopředu, a v každém kroku zpracujeme všechny události. Pokud autobus odjíždí z dostupné stanice, zkontrolujeme, jestli cena linky náhodou není vyšší, než kdybychom zaplatili při nástupu z této stanice. Pokud je vyšší, snížíme cenu linky.

Při příjezdu zkontrolujeme, jestli cena linky plus cena, kterou musíme zaplatit za cestu, není nižší než cena stanice. Pokud ano, snížíme cenu stanice.

Ve chvíli, kdy se cena cíle stane nižší než limit v dolárcích, máme hledané spojení a můžeme ukončit výpočet. Jdeme s časem dopředu, takže spojení určitě bude nejrychlejší.

Časová složitost bude lineární s velikostí vstupu, protože událostí je tolik jako součet zastávek na všech linkách (to je velikost vstupu) a každou zvládneme zpracovat v konstantním čase. Pokud by časy byly zadány reálnými čísly, časová složitost by kvůli třídění událostí stoupne na $\mathcal{O}(Z \log Z)$, kde Z je součet zastávek na všech linkách, tj. velikost vstupu.

Pavel Machek

19-5-2 Nesnadné dělení

Většina řešitelů nenechala Isabelu s Přesprstem na holičkách, a pokud to bylo možné, tak peníze na přesné poloviny rozdělila. Ať už dynamicky s pomocí kuchařky nebo exponenciálně. Algoritmy, které nejdříve bankovky setřídily a pak se je snažily hladově rozdělit jedním průchodem na dvě části, bankovky rovněž sem tam rozdělily na dvě poloviny. Bohužel, občas jedna z nich byla větší než ta druhá :–) No, a jak mělo vypadat správné řešení?

Nejdříve si úlohu přeformulujeme trochu obecněji. Máme posloupnost čísel S (hodnot bankovek) a chceme z ní vybrat podposloupnost, která bude mít nějaký konkrétní součet s (v našem případě je s rovno polovině celkového součtu S).

Jako první každého určitě napadne jednoduchý algoritmus: Vyzkoušíme všechny možné podposloupnosti, každou zkusíme sečíst a podíváme se, jestli má „správný“ součet. Pokud má součet s, tak jsme rovnou našli jednu z vhodných podposloupností a tím i vhodné rozdělení bankovek. Pokud žádná podposloupnost nemá součet s, máme jistotu, že bankovky rozdělit nepůjdou.

Tento algoritmus má jednu vadu. Každý prvek ve vybrané podposloupnosti buď je, nebo není, takže nepotřebujeme

tým matematiků, abychom viděli, že časová složitost bude exponenciální, tj. $\mathcal{O}(2^N)$.

Teď se připravte na to, že přijde špatná zpráva: Problém výběru podposloupnosti s daným součtem je NP-úplný, takže pro obecně zadaná čísla je výše uvedený algoritmus to nejlepší, co umíme. Avšak nezoufejte – záchrana se blíží.

Náš problém naštěstí není tak úplně obecný. Využijeme faktu, že velikosti bankovek (a tedy i čísla z našich posloupností) jsou omezena nějakým relativně malým číslem M. Z toho vyplývá, že náš hledaný součet s bude nejvýše $\mathcal{O}(M \cdot N)$, takže můžeme nasadit rafinovanou metodu, které se říká dynamické programování (viz kuchařka).

V první fázi algoritmu nejprve sečteme všechny prvky (resp. bankovky) a proměnnou s položíme rovnou polovině tohoto součtu. Pokud je součet lichý, můžeme rovnou skončit a oznámit nedočkavému uživateli, že tyto bankovky opravdu rozdělit na polovinu nejdou. Dále si připravíme pole V indexované od 1 do s, které je na počátku celé inicializované na samé nuly. Postupně si do něho budeme ukládat bankovky následujícím způsobem:

Nyní budeme brát bankovky jednu po druhé a s každou provedeme následující. Projedeme celé pole V, a pokaždé když narazíme na nenulový prvek, vezmeme jeho index i (přesně tak – index představuje vlastně celkový dosažený součet), přičteme k němu hodnotu aktuální bankovky b a na pozici s nově získaným indexem $(i + b)$ uložíme naši bankovku (třeba jako její index v posloupnosti S). Co jsme tím vlastně udělali? Nalezený nenulový index i nám říká, že existuje výběr z bankovek (z těch, které předchází b), které mají součet i. Když k nim přidáme bankovku b, tak umíme poskládat i součet $i + b$, takže si na tento index uložíme poslední bankovku, která tento součet tvoří. Pokud je součet $i + b \geq s$, pak jej do pole neuložíme, protože už nás nezajímá (je příliš velký). Stejně tak pokud na pozici $i+b$ už je zapsaná jiná bankovka, necháme ji tam (abychom si neničili, co už máme spočítáno).

Při výše popsaném průchodu polem V musíme postupovat vždy odshora dolů, jinak by mohl nastat zajímavý efekt. Představte si, co by se stalo, kdybychom postupovali opačně a přidávali např. bankovku s hodnotou 1 do prázdného pole. Na první pozici bychom zapsali 1. Následně bychom se podívali na 1. pozici, přičetli hodnotu bankovky (tedy 1) a zapsali ji – na pozici 2. Takto bychom krásně vyplnili celé pole, tj. bankovku 1 bychom vlastně použili opakovaně.

Poslední, co zbývá popsat, je jak ze získaných hodnot pole V zjistit rozdělení bankovek. Nejprve se podíváme na položku $V[s]$. Pokud je nulová (není tam uložena žádná bankovka), tak víme, že neumíme vybrat podposloupnost se součtem s. Dále postupujeme jednoduše. Bankovku na pozici $V[s]$ vypíšeme a podíváme se, jak složit podposloupnost se součtem $i = s - V[s]$. Ale poslední bankovka, která do takové podposloupnosti patří, se přece nachází na pozici $V[i]$. Tak ji vypíšeme a opět si položíme $i = i - V[i]$. Takto pokračujeme, dokud nevypíšeme všechny prvky (tj. i nám neklesne na nulu).

Jak již bylo naznačeno, časová složitost algoritmu bude $\mathcal{O}(N \cdot s)$, kde N je počet bankovek a s jejich součet. Paměťová složitost je o trochu menší, a to sice $\mathcal{O}(N + s)$. Dá se nahlédnout, že pokud bychom neměli žádné omezení na velikost součtu s, byla by časová složitost exponenciální k velikosti vstupu. Na to, abychom zakódovali číslo s, potřebujeme $B = \log_2 s$ bitů, proto by časová složitost byla $\mathcal{O}(s) = O(2^B)$, tedy exponenciální.

Snad se vám z toho příliš nezamotala hlava, a pokud ano, tak vyražte někam ven – třeba k vodě nebo na zmrzlinu a nezapomeňte si vzít s sebou „správný“ obnos bankovek :o))

Zbyněk Falt

19-5-3 Hamtyhamtyhamty

Nejdříve si předvedeme jednoduchou *neprohrávající* strategii, tedy takovou, se kterou pokaždé buď vyhrajeme, nebo aspoň remízujeme. Obarvíme si $2n$ čísel, o která hrajeme, střídavě černě a bíle. Všimneme si, že pokud začínáme černým (bílým) číslem, musí soupeř vždy sebrat bílé (černé) a my dostaneme opět pozici, která má na jednom kraji černé a na druhém bílé číslo. Můžeme tedy snadno soupeře donutit k tomu, aby posbíral všechna bílá nebo všechna černá a my ta druhá. My se samozřejmě rozhodneme podle toho, která mají větší součet, a tím vyhrajeme. Pokud se nám stane, že obě skupiny čísel mají součet stejný, vede naše strategie alespoň k remíze.

Zadání úlohy po nás ovšem chce, aby naše strategie vyhrála, kdykoliv je to možné. Je to opravdu tak, že kdykoliv černobílá strategie remízuje, je remíza nevyhnutelná? Bohužel nikoliv – například pro čísla 1,2,4,2,1,2 můžeme odebráním dvojky dostat soupeře do stavu 1,2,4,2,1, ze kterého musí odebrat jedničku (teď vedeme o jedna), a pokud spustíme černobílou strategii až teď, určitě o svůj náskok nepřijdeme. Tento problém se mnozí z vás pokoušeli obejít tím, že po každém tahu testovali, jestli nezačal být součet černých a bílých různý (to v našem příkladu nepomůže, protože rozhodující je už první tah), nebo třeba při rovnosti odebírali větší číslo (tedy je o něco těžší přijít s protipříkladem, ale 4,2,1,6,8,5 zabere). Takových fíglů se dá vymyslet spousta, ale neznám žádný, který opravdu funguje.

Půjdeme na to tedy trochu jinak: Nebudeme se snažit jenom vyhrát, ale dokonce soupeře obrat o co nejvíce, zkrátka co nejvíce nahamtat – chceme tedy, aby rozdíl mezi tím, co získáme my a co získá soupeř, byl co největší. Když si označíme zadaná čísla $A_0, \ldots, A_{N-1}$, budou všechny pozice v průběhu hry vždy nějaké intervaly $A_i, \ldots, A_{i+\ell-1}$. Pro každý takový interval si spočítáme hodnotu $H_{i,\ell}$, která nám řekne, kolik je do konce hry schopen nahamtat ten, kdo je v tomto okamžiku na tahu. (Ti zkušenější už samozřejmě správně větří dynamické programování.)

Všimneme si, že $H_{i,1} = A_i$ (pokud už je ve hře jen jediné číslo, co zbývá, než ho sebrat). Pokud je interval delší, máme dvě možnosti, jak hrát:

- Buďto sebereme A_i a soupeři předáme $A_{i+1}, \ldots, A_{i+\ell-1}$. Tehdy soupeř, pokud bude hrát nejlépe, jak může, nahamtá $H_{i+1,\ell-1}$, pročež my můžeme celkově nahamtat $A_i - H_{i+1,\ell-1}$.
- Nebo sebereme $A_{i+\ell-1}$ a předáme $A_i, \ldots, A_{i+\ell-2}$, a proto nahamtáme $A_{i+\ell-1} - H_{i,\ell-1}$.

Z těchto dvou možností si samozřejmě vybereme tu, ve které nahamtáme víc. Platí tedy:

$$H_{i,\ell} = \max(A_i - H_{i+1,\ell-1}, A_{i+\ell-1} - H_{i,\ell-1}).$$

Navíc hodnoty pro úseky délky ℓ počítáme z hodnot pro úseky délky $\ell-1$, takže stačí postupovat indukcí od nejmenšího ℓ k největšímu. Podle $H_{0,n}$ pak okamžitě poznáme, jestli je pro nás hra vyhraná nebo prohraná, a v každém stavu hry snadno zjistíme, zda máme odebrat levé nebo pravé číslo podle toho, která z možností nám dala hodnotu

$H_{i,\ell}$. Přesně na tomto principu je založen následující program, který v čase a paměti $\mathcal{O}(n^2)$ pro všechny intervaly spočítá jak $H_{i,\ell}$, tak optimální tah $T_{i,\ell}$.

```
int N;                  // Počet čísel na vstupu
int A[MAX];             // Číslíčka, se kterými hrajeme
int H[MAX][MAX];        // Maximální zisk v každém úseku
int T[MAX][MAX];        // Vyhrávající tah pro úsek

void hamtyhamtyhamty(void)
{
  for (int i=1; i<=N; i++)      // Úseky délky 1
    H[i][1] = A[i];
  for (int l=2; l<=N; l++)      // A teď ty delší
    for (int i=1; i<=N-l+1; i++) {
        int levy = A[i] - H[i+1][l-1];
        int pravy = A[i+l-1] - H[i][l-1];
        if (levy > pravy) {
            T[i][l] = 'L';
            H[i][l] = levy;
        } else {
            T[i][l] = 'P';
            H[i][l] = pravy;
        }
    }
}
```

Ostatně, k témuž výsledku bychom se také mohli dobrat tak, že bychom si napsali rekurzivní funkci, která by počítala $H_{i,\ell}$ podle našich vzorečků. Přímočaře naprogramovaná by běžela v exponenciálním čase, ale mohli bychom si v nějakém pomocném poli pamatovat, které hodnoty jsme už spočítali, a nepočítat je znovu. Tak bychom se dostali opět na kvadratickou časovou a paměťovou složitost.

Nechci tedy nikterak podceňovat naši milou Izabelu, ale tipoval bych, že v jejím úspěchu byla přeci jen trocha začátečnického štěstí, a to jí přineslo posloupnosti, na kterých funguje černobílá strategie snadno spočítatelná zpaměti. Konec konců, není divu, náhodný vstup toto s velkou pravděpodobností splňuje. Ať tedy takové máte i vy :-)

Finta na závěr: Kdybychom se na naši úložku dívali jako na programátorskou úlohu místo původně zamýšlené jednoduché logické hádanky, mohli bychom se ještě pokusit snížit nepříjemně vysokou paměťovou náročnost. Na co tu paměť vlastně potřebujeme? Samotný výpočet hodnot hodnot $H_{i,\ell}$ si vystačí s hodnotami $H_{j,\ell-1}$ a na všechny menší délky můžeme zapomenout. Ovšem potřebujeme si zapamatovat, který tah byl ve které pozici optimální, abychom pak dokázali při libovolném vývoji hry zjistit, jak máme hrát. Jak ušetřit tady? Místo toho, abychom si pamatovali všechny stavy hry, můžeme si třeba uložit jen stavy s délkou úseku dělitelnou nějakým k, a jakmile se dostaneme do intervalu délek mezi ki a $k(i+1)$, spočítáme z uložených výsledků pro délku ki výsledky pro všechny mezilehlé délky. Tak algoritmus příliš nezpomalíme (každou pozici počítáme jen dvakrát) a paměť zredukujeme na $\mathcal{O}(kn + (n/k)n)$, tedy při volbě $k = \sqrt{n}$ na $\mathcal{O}(n\sqrt{n}) = \mathcal{O}(n^{3/2})$. Mezilehlé délky ale můžeme podrozdělovat stejným způsobem a pokud do sebe zasunutých podrozdělení bude p, vyjde, že optimální je podrozdělování na $n^{1/p}$ částí, časová složitost $\mathcal{O}(n^2 p)$ a prostorová $\mathcal{O}(n^{1+1/p})$. Mezním případem je podrozdělení na dvě části, které nám dá hloubku $\mathcal{O}(\log n)$, čas $\mathcal{O}(n^2 \log n)$ a paměť $\mathcal{O}(n \log n)$. To je poměrně univerzální trik, kterým jde u mnoha úloh ušetřit spousta paměti za cenu mírného (obvykle logaritmického) zpomalení.

Martin Mareš

19-5-4 Lodní mrazáky

Nejprve si všimněme, že jeden mrazák nám stačit nebude. Kdybychom použili pouze jeden, věci by se do něj nastrkaly, ty nejstarší by skončily vespod a už by se k nim nikdo nedostal.

Proto použijeme mrazáky dva, a aby se nám to nepletlo, stanovíme si jistá pravidla na uspořádání potravin.

1) V prvním mrazáku, říkejme mu třeba příchozí, budou potraviny seřazeny tak, že čím hlouběji v mrazáku budeme, tím starší budou.

2) Druhý mrazák, budiž nazýván odchozí, bude seřazen přesně naopak, tedy čím hlouběji, tím novější potraviny.

3) Libovolná potravina v odchozím mrazáku bude starší, než libovolná v příchozím.

Teď zbývá vytvořit operace ulož a sněz tak, aby žádnou z předchozích podmínek neporušily. Operace ulož je jednoduchá, prostě vložíme potravinu do příchozího mrazáku. Tím určitě neporušíme podmínku 1) - tato potravina je určitě novější, než vše, co bylo v mrazáku předtím a přišlo to nahoru, ani podmínku 2) - odchozího mrazáku se vůbec nedotkneme, ani podmínku 3), protože tato potravina je určitě novější, než cokoliv v odchozím mrazáku.

Pokud bude v odchozím mrazáku alespoň jedna potravina, je operace sněz také triviální. Potravina navrchu tohoto mrazáku je nejstarší, protože je starší než cokoliv pod ní a je starší než cokoliv v příchozím mrazáku. Tudíž ji stačí jen vzít. Toto samozřejmě neporuší žádnou z podmínek, odebráním libovolné potraviny se nic zkazit nedá.

Co ale v případě, že odchozí mrazák zeje prázdnotou? Předpokládejme, že je alespoň jedna potravina v příchozím mrazáku, protože jinak by na lodi panoval hladomor a nebylo by možné provést operaci sněz. Celkově nejstarší potravinou je tedy nejstarší potravina v příchozím mrazáku, která je dle podmínky 1) vespod. Proto přeházíme všechny potraviny z příchozího do odchozího mrazáku. Tím se otočí pořadí potravin, takže nejstarší bude navrchu a postupně budou novější a novější. Tím je splněna třetí podmínka a zařízeno, že lze provést operaci sněz tak, jak byla popsána výše. První a druhá jsou splněny také, protože příchozí mrazák je prázdný.

Nyní, kolik se provede operací, tedy jaká je časová složitost? Každou potravinu, která projde úschovou v podpalubí, čeká uložení a vyndání z každého ze dvou mrazáků. Celkem tedy 4 operace. Pro zpracování N potravin bude potřeba $4N$ operací, z nichž právě $2N$ budou pomocné. Splnili jsme tedy kapitánovu žádost, aby pomocných operací bylo $O(N)$.

Paměťovou složitost můžeme brát buď dle počtu mrazáků, které potřebujeme 2, nebo podle počtu obsazených mrazících pozic, ale protože každá potravina může být v jeden okamžik uložena nejvýše jednou, je paměťová složitost lineární.

Michal „vorner“ Vaner

19-5-5 Praktická úložka – Počet inverzí

Jak už to tak v životě bývá, způsobů řešení této úlohy je více. Zde si popíšeme jeden velice jednoduchý a naznačíme některé další možné. Posaďte se, prosím, na svá místa, připoutejte se a během startu nekuřte.

Náš postup je založen na známém třídícím algoritmu mergesort, neboli třídění pomocí slévání. Tento algoritmus pracuje na principu rozděl a panuj. Tříděnou posloupnost rozdělí

na dvě poloviny (tedy podúlohy menšího rozsahu), které setřídí rekurzivním použitím stejného algoritmu. Setříděné poloviny následně slije do jedné posloupnosti.

Pro lepší pochopení našeho algoritmu si ještě raději zopakujeme průběh slévání. Řekněme, že máme dvě vzestupně setříděné posloupnosti (uložené jako pole) A a B a chceme je slít do jedné opět vzestupně setříděné posloupnosti C. Vytvoříme si indexy a, b a c, které inicializujeme tak, aby ukazovaly na první prvky jednotlivých posloupností (tj. a ukazuje na první prvek A atd.). Dokud se index a, nebo b nedostane mimo rozsah jeho posloupnosti, budeme provádět následující krok: Porovnáme prvky $A[a]$ a $B[b]$, menší z nich zkopírujeme do $C[c]$ a posuneme index v poli s menším prvkem na další prvek v posloupnosti. Rovněž posuneme c na další volné místo ve výsledné posloupnosti. Když některý z indexů (a, nebo b) dojede za konec své posloupnosti, algoritmus končí, avšak ještě je třeba dokopírovat zbývající prvky z druhé posloupnosti (z té, která ještě nebyla zpracována celá). Např. pokud a dojede za konec A, musí se ještě zpracovat zbytek posloupnosti B.

Nyní zbývá rozmyslet, jak nám tento algoritmus pomůže při počítání inverzí. Celkový počet inverzí v posloupnosti lze spočítat jako součet počtu inverzí v obou polovinách (tj. v obou menších podproblémech) plus počet inverzí, které objevíme při slévání těchto polovin. Z principu fungování algoritmu je jasné, že nám stačí počítat pouze inverze objevené sléváním (o ostatní se postará rekurze).

Máme tedy algoritmus na slévání dvou posloupností popsaný výše. Jako A si označíme první polovinu tříděné posloupnosti a jako B polovinu druhou. Pokud by bylo uspořádání správné (tj. neobsahovalo by žádné inverze), budou všechny prvky z A menší než prvky B. V každém kroku algoritmu nastává právě jedna z možností:

- $A[a] \leq B[b]$ – prvek v první posloupnosti je menší nebo roven prvku ve druhé posloupnosti, takže je vše v pořádku a žádnou inverzi jsme neobjevili.
- $A[a] > B[b]$ – prvek v první posloupnosti je větší než prvek ve druhé posloupnosti. To znamená, že $B[b]$ bude ve výsledku zařazen před všechny zbývající prvky v A, což je rozhodně porucha v uspořádání. Každý zbývající prvek v A je tím pádem v inverzi s prvkem $B[b]$, takže nám stačí přičíst k celkovému počtu inverzí počet zbývajících prvků v A.

Časová složitost tohoto algoritmu je stejná jako časová složitost merge-sortu, tzn. $\mathcal{O}(N \cdot \log_2 N)$. Paměťová složitost je při vhodné implementaci pouze $\mathcal{O}(N)$, neboť nám stačí jedno pole na načtené prvky a jedno pomocné pole na slévání.

Paměťové limity pro jednotlivé testy byly nastaveny tak, abyste mohli použít i staticky alokované pole (jak je to ve vzorovém programu). Na toto pohodlí si ovšem příliš nezvykejte, neboť v dalších praktických úlohách již budete muset alokovat paměť dynamicky podle toho, kolik jí skutečně budete potřebovat.

Závěrem bych ještě zmínil další možné způsoby řešení. Prvním způsobem je použít jiné třídící algoritmy místo merge-sortu. Problém je v tom, že ne každý algoritmus nám bude vyhovovat. Např. quick-sort použít nemůžeme, neboť přehazuje prvky mezi oběma polovinami tříděných dat, a tak nám může během třídění vytvářet inverze, které v původní posloupnosti nebyly. Druhou možností je použít vhodně upravené binární vyhledávací stromy, avšak detailnější popis by si vyžádal poměrně velké množství dalšího textu, a tak si jej dovolím vynechat (kdyby se časem našlo větší množství zájemců o toto řešení, tak jej nějakou formou doplním).

Tímto vám děkuji, že jste se zúčastnili testování praktické úložky. CodEx i já se na vás těšíme v příštím ročníku. CodEx teď zůstane ještě nějakou dobu funkční (pravděpodobně do konce školního roku), takže si ještě můžete vyzkoušet odevzdat nějaká řešení jen tak, nebo si zacvičit v posilovně.

```
return E_OK;
```

Martin „bobřík“ Kruliš

19-5-6 Prolog

1. Nejkratší program

Ukázalo se, že není problém vytvořit krátký program, ale vytvořit funkční program. Řešení, která cyklí při hledání neexistujícího prvku, prostě nemohou získat víc než půl bodu. Při tvoření krátkého programu jste mohli zvolit několik cest, záleželo na tom, jak moc jste chtěli používat vestavěné predikáty Prologu. Všeobecně ale platilo, že nejkratší programy bylo možné napsat na jeden predikát (na jeden řádek).

2. Fronta

Toto cvičení vás mělo navést na řešení pomocí rozdílových seznamů, což většina řešitelů pochopila. Správných řešení ale bylo málo. Prohlédněte si tedy autorské řešení.

3. Expertní systém

Protože se jedná o větší program, ve kterém je potřeba si pořádně rozmyslet datové struktury, spolupráci s uživatelem a jiné záludnosti, a protože se našlo jenom několik odvážlivců, rozhodla jsem se nešťourat do syntaktických chyb a jiných nedostatků. Pokud myšlenka vypadala alespoň trochu použitelně, pokud autor působil dojmem, že ví, co dělá, a pokud se zdálo, že by se nakonec program dal odladit, připsala jsem plný počet bodů. V této úloze nelze žádné řešení prohlásit za jediné optimální. V autorském řešení najdete jedno z nejjednodušších a nejvíce srozumitelných řešení vytvořené pomocí stromu otázek.

Jana Kravalová

Úloha 19-5-1 – Útěk – program

```
/* S díky Romanu Smržovi*/

#include <stdlib.h>
#include <stdio.h>
#define MAX_STANIC 1000
#define MAX_ZASTAVENI 1000

struct odjezd {
        int odkud, kam;
        int prijezd;                    /* čas příjezdu do další stanice */
        int cena;                       /* cena cesty do další stanice */
        int linka;
```

```
        struct odjezd *dalsi;
};
struct prijezd {
        struct prijezd *odkud;                  /* předchozí zastávka */
        int stanice;
        int cena;                               /* celková cena cesty do dané stanice */
        int linka;
        struct prijezd *dalsi;
};
int d,n,s,c;
struct odjezd *odjezdy[1440];
struct prijezd *prijezdy[1440];
struct prijezd *cesty[MAX_STANIC];              /* Aktuálně nejlevnější cesta do dané stanice */

void vypsat_cestu(struct prijezd *p) {
        if (!p) return;
        vypsat_cestu(p->odkud);
        printf("stanice %d linka %d, ", p->stanice, p->linka);
}

int main(int argc, char *argv[]) {
        int i, j, t;
        struct prijezd *p;
        struct odjezd *o;

        scanf("%d %d %d %d\n", &d, &n, &s, &c);
        for (i=0; i<n; i++) {                   /* Načtení odjezdů a jejich přiřazení k časům */
                int k, mzst, mcas, mcena, szst, scas, scena, ch, cm;
                scanf("%d %d %d:%d %d", &k, &mzst, &ch, &cm, &mcena);
                mcas = 60*ch+cm;
                for (j=1; j<k; j++) {
                        scanf( "%d %d:%d %d", &szst, &ch, &cm, &scena);
                        scas = 60*ch+cm;

                        struct odjezd *odjezd = malloc(sizeof(struct odjezd));
                        odjezd->odkud = mzst; odjezd->kam = szst;
                        odjezd->linka = i+1;
                        odjezd->cena = scena-mcena; odjezd->prijezd = scas;
                        odjezd->dalsi = odjezdy[mcas]; odjezdy[mcas] = odjezd;
                        mzst = szst; mcas = scas; mcena = scena;
                }
        }

        struct prijezd start = { 0, s, 0, -1000, NULL };
        cesty[s] = &start;
        for (t = 0; t<24*60; t++) {             /* Postupný průchod časem */
                for (p = prijezdy[t]; p; p=p->dalsi)
                        if (!cesty[p->stanice] || cesty[p->stanice]->cena > p->cena)
                                cesty[p->stanice] = p;

                for (o = odjezdy[t]; o; o=o->dalsi)
                        if (cesty[o->odkud] && cesty[o->odkud]->cena + o->cena <= d) {
                                struct prijezd *prijezd = malloc(sizeof(struct prijezd));
                                prijezd->odkud = cesty[o->odkud];
                                prijezd->stanice = o->kam;
                                prijezd->linka = o->linka;
                                prijezd->cena = cesty[o->odkud]->cena + o->cena;
                                prijezd->dalsi = prijezdy[o->prijezd];
                                prijezdy[o->prijezd] = prijezd;
                        }
                if (cesty[c]) {                 /* Nalezli jsme cestu do cíle */
                        vypsat_cestu(cesty[c]);
                        printf("\n");
                        return 0;
                }
        }
        printf("Cesta neexistuje\n");
        return 0;
}
```

Úloha 19-5-2 – Nesnadné dělení – program

```
type TIntArray = array[1..1000000] of LongInt;
     PIntArray = ^TIntArray;

procedure loadData(var Bankovky: PIntArray; var N: LongInt);
begin
   { tahle procedůra vytvoří pole Bankovky potřebné délky a načte do něj data, také inicializuje proměnnou N }
end;

{ Spočítá a vrátí celkový součet všech prvků v poli Bankovky - tj. celkovou hodnotu všech bankovek }
function soucet(Bankovky: PIntArray; N: LongInt): LongInt;
var res: LongInt;
begin
    res := 0;
    while N > 0 do begin
        res := res + Bankovky^[N];
        dec(N);
    end;
    soucet := res;
end;
```

```
{ Hlavní algoritmus - zjistí, jak dosáhnout všech možných součtů (naplní pole "Vybrane") }
procedure spoctiSoucty(Bankovky: PIntArray; N: LongInt; Vybrane: PIntArray; Sum: LongInt);
var i, max, item: LongInt;
begin
    for i := 1 to Sum do Vybrane^[i] := 0;              { inicializujeme si pole "Vybrane" }
    max := 0;
    for item := 1 to N do begin                         { zkusíme všechny bankovky }
        if Bankovky^[item] > Sum then Exit;             { nalezli jsme bankovku,která je cennější,než půlka...máme smůlu }
        for i := max downto 1 do                        { postupně prověřuji všechny možné součty }
            if (Vybrane^[i] > 0) and (i + Bankovky^[item] <= Sum) and (Vybrane^[i + Bankovky^[item]] = 0) then
                Vybrane^[i + Bankovky^[item]] := item;  { umím součet i => umím také součet i + nová bankovka }

        if Vybrane^[ Bankovky^[item] ] = 0 then Vybrane^[ Bankovky^[item] ] := item;
        max := max + Bankovky^[item];                   { upravím maximální součet, který jsem doposud viděl }
        if (max > Sum) then begin                       { ale stačí mi jen součty do velikosti částky, kterou hledám }
            max := Sum;
            if (Vybrane^[Sum] > 0) then Exit;           { pokud jsem našel způsob, jak dostat hledanou částku, končím }
        end;
    end;
end;

{ Vytahá z pole Vybrane bankovky a vypíše je na výstup }
procedure zapisVysledky(Bankovky: PIntArray; N: LongInt; Vybrane: PIntArray; Sum: LongInt);
var i: LongInt;
begin
    if (Vybrane^[Sum] = 0) then begin                  { neexistuje způsob, jak složit součet velikosti Sum }
        writeln('Bankovky nelze rozdělit na polovinu.');
        Exit;
    end;
    i := Sum;                                          { i představuje částku, kterou je tě zbývá vypsat }
    while i > 0 do begin
        writeln(Vybrane^[i]);                          { vypíšeme vybranou bankovku }
        i := i - Bankovky^[ Vybrane^[i] ];             { zmenšíme akt. částku o právě vypsanou bankovku }
    end;
end;

var N, Sum: LongInt;
    Bankovky, Vybrane: PIntArray;
begin
    loadData(Bankovky, N);
    Sum := soucet(Bankovky, N);

    if (Sum mod 2 = 0) then begin
        Sum := Sum div 2;                              { chceme vybrat bankovky, jejich  součet je právě polovina celkového součtu }
        GetMem(Vybrane, Sum * sizeof(LongInt));
        spoctiSoucty(Bankovky, N, Vybrane, Sum);       { hlavní algoritmus }
        zapisVysledky(Bankovky, N, Vybrane, Sum);      { vytaháme výsledky z pole "Vybrane" a vypí eme je }
    end else writeln('Bankovky nelze rozdělit.');      { je-li celkový součet lichý, určitě nepůjde rozdělit na dvě stejné části }
end.
```

Úloha 19-5-4 – Lodní mrazáky – program

```
unit  Podpalubi;

uses  Mrazaky;
var   Prichozi, Odchozi: Mrazak;                          { Naše dva mrazáky }

procedure uloz( Co: Potravina );
begin   vlozDoMrazaku( Co, Prichozi ); end;

function snez: Potravina;
begin
        if prazdnyMrazak( Odchozi ) then begin
                if prazdnyMrazak( Prichozi ) then begin    { Hladomor }
                        snez := HrnicekOdMedu;
                        exit;
                end;
                while not prazdnyMrazak( Prichozi ) do vlozDoMrazaku( vyndejZMrazaku( Prichozi ), Odchozi );
        end;
        snez := vyndejZMrazaku( Odchozi );
end;
```

Úloha 19-5-5 – Počet inverzí – program

```
program 19-5-5;
const MAX = 100000;
type TCisla = array[1..MAX] of LongInt;

{ Hlavní funkce programu. Pracuje na principu algoritmu merge sort a přitom počítá inverze.
  Parametry: main - hlavní pole s čísly, tmp - pomocné pole, i,j - tříděný rozsah }
function merge(var main, tmp: TCisla; i, j: LongInt) : Integer;
var     mid, count: LongInt;
        a, b, res: LongInt;
begin
        merge := 0;
        if (i >= j) then Exit;                          { prázdný interval => končíme }

        { určíme střed tříděného intervalu a metodou rozděl a panuj spočítáme podúlohy }
        mid := (i+j) div 2;
        count := merge(tmp, main, i, mid);
        count := merge(tmp, main, mid+1, j) + count;
```

```
        { nyní slijeme setříděné poloviny }
        a := i;  b := mid+1;                            { indexy v setříděných polovinách }
        res := i;                                       { index ve výsledném poli }
        while (a <= mid) and (b <= j) do begin          { sléváme, dokud jeden z indexů nedojede do konce svého úseku }
                if (tmp[a] > tmp[b]) then begin         { "porucha" v uspořádání - v druhé půlce je menší prvek }
                        count := count + (mid-a+1);     { přičteme tolik inverzí, kolik prvků zbývá v 1. části }
                        main[res] := tmp[b];
                        inc(b);
                end else begin                          { prvek z 1. části je menší - žádné inverze nepřičítáme }
                        main[res] := tmp[a];
                        inc(a);
                end;
                inc(res);
        end;
        { zkopírujeme zbytky slévaných částí (jsou-li nějaké) }
        while (a <= mid) do begin main[res] := tmp[a]; inc(a); inc(res); end;
        while (b <= j) do begin main[res] := tmp[b]; inc(b); inc(res);end;
        merge := count;
end;

var     F        : Text;
        N, i     : LongInt;
   cisla, cisla2 : TCisla;                              { čísla jsou alokovaná staticky (pro zpřehlednění kódu) }
begin
        Assign(F, 'cisla.in');                          { Načtení vstupu }
        Reset(F);
        ReadLn(F, N);                                   { počet čísel N }
        for i := 1 to N do begin
                Read(F, cisla[i]);
                cisla2[i] := cisla[i];
        end;
        Close(F);

        Write( merge(cisla, cisla2, 1, N) );            { spočítáme počet inverzí a rovnou jej vytiskneme na výstup }
end.
```

Úloha 19-5-6 – Prolog – program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  KSP 19-5-6 1. Nejkratší program  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

p(S,K,X):-sort(S,[Y|T]),(Y>K,X=Y,!;p(T,K,X)).                   % nejlepší řešení účastníků (Roman Smrž)

q(S,K,X):-setof(H,(member(H,S),H>K),[X|_]).                     % původní autorské řešení

r(S,K,X):-sort([K|S],R),nextto(K,X,R).                          % extra řešení (Milan Straka)

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  KSP 19-5-6 2. Fronta  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

zretez(A-B, B-C, A-C).                                          % zřetězení rozdílovych seznamů (potřeba pro vloz2)

vloz(Fronta-[NovyPrvek|Y], NovyPrvek, Fronta-Y).                % vloz(StaraFronta, NovyPrvek, NovaFronta)

vloz2(X-X, NovyPrvek, [NovyPrvek|X]-X) :- var(X).               % kdo nechápe vloz, ať si přečte vloz2
vloz2(StaraFronta-X, NovyPrvek, NovaFronta) :- zretez(StaraFronta-X, [NovyPrvek|Y]-Y, NovaFronta).

odeber(X-X) :- var(X), fail.                                    % odeber(StaraFronta, NovaFronta, Prvek)
odeber([H|Zbytek]-X, Zbytek-X, H).

je_prazdna(X-X) :- var(X).                                      % je_prazdna(Fronta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     KSP 19-5-6 3. Expertní systém     %
%  pěkné a jednoduché řešení Jana Žáka  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% databaze.pl:
q([
        q('Je to kockovita selma?', [
                q('Je to kocka?', kocka),
                q('Je to lev?', lev)
        ]),
        q('Je to psovita selma?',[
                q('Je to pes?', pes),
                q('Je to liska?', liska)
        ])
]).

% program.pl:

hadej :- write('Mysli si zvire...'), nl, q(Otazky), hadej(Otazky).
hadej([]) :- write('Takove zvire neznam').                      % prošli jsme celý strom
hadej([q(Otazka,Ano)|Ne]) :-
        write(Otazka), nl, read(Odpoved),                       % vypiš otázku a přečti odpověď
        (Odpoved = ano, hadej(Ano)) ;                           % kladná odpověď, použij podvětve
        (Odpoved = ne, hadej(Ne)).                              % záporná odpověď, použij další větev
hadej(Zvire) :- write('Myslis si zvire '), write(Zvire).
```

Výsledková listina devatenáctého ročníku KSP po páté sérii

		škola	*ročník*	*sérií*	*1951*	*1952*	*1953*	*1954*	*1955*	*1956*	*série*	*celkem*
1.	Marek Nečada	G Jihlava	3	5	9	10	7		10		37,2	190,6
2.	Vojtěch Kolář	G Neratov	3	5			7	9,5	10	13	40,4	189,5
3.	Jakub Kaplan	GJKTyla	3	15		10	6	9,5	10		34,4	187,0
4.	Jan Michelfeit	G HBrod	3	5	10	10		10	8	12	42,6	183,8
5.	Zbyněk Konečný	GKptJaroš	4	17	8	3	10	10	6	3,5	30,7	174,2
6.	Jan Žák	G HBrod	2	5		9			10	13	32,6	163,6
7.	Trung Ha duc	GMasaryk	1	5		2	7		6	9,5	30,2	161,3
8.	David Brazdil	G Zlín	2	5					6		7,8	147,7
9.	Vlastimil Dort	GŠpitálsPH	1	5			1	9,5	8		20,7	143,5
10.	Petr Onderka	G VKlobou	4	10	8	10		9,5	10	12	41,7	143,1
11.	Petr Kratochvíl	G SvětláNS	4	18							0,0	139,6
12.	Pavel Klavík	G Chrudim	4	19					10		10,0	139,1
13.	Libor Plucnar	GPBezruče	2	5	7	9			10		28,1	138,3
14.	Vojtěch Tůma	G Jihlava	3	4		10	10	9,5	10		39,8	131,6
15.	Tomáš Sýkora	G VKlobou	3	8	3	10		10	3		27,8	106,4
16.	Matěj Korvas	GJSeiferPH	4	3							0,0	99,9
17.	Pavel Veselý	G Strakon	2	7		1		10	3		15,8	96,7
18.	Miroslav Klimoš	G Bílovec	2	19					10		10,0	95,6
19.	Josef Pihera	G Strakon	4	14	8				10		17,3	82,2
20.	Lukáš Kripner	G Litvínov	1	4			6	9	6		24,7	79,6
21.	Karel Pajskr	GJKeplera	2	4							0,0	78,4
22.	Roman Smrž	GOhradní	3	11	10			10	6	4	30,0	75,3
23.	Roman Říha	G Prachat	3	3		10	6	10	6		35,5	73,9
24.	Radim Cajzl	G NMnMor	0	10				9	6		15,4	73,2
25.	Jiří Maršík	GJKTyla	3	8					4		5,0	70,3
26. – 27.	Jakub Balhar	GJNerudy	4	5							0,0	67,4
	Martin Majer	SPŠÚžlabin	2	7		1			10		11,6	67,4
28.	Tomáš Herceg	G Třebíč	4	15							0,0	67,0
29.	Jan Kohout	G Roudnice	4	9							0,0	65,0
30.	Petr Holášek	G Příbor	3	4		1	6		3		14,2	57,0
31.	Tomáš Jakl	G MTřebová	3	2	2	10		9,5	3		29,8	56,7
32.	Marika Ivanová	SPŠ Zlín	3	3		10		5			17,5	55,7
33.	Ondřej Bouda	GKptJaroš	4	9				9,5	6		16,3	53,2
34.	Kristýna Krejčová	G Tišnov	4	5							0,0	52,6
35.	Zbyněk Jiráček	GArabská	3	2							0,0	50,6
36.	Viktor Lucza	G Rožňava	3	2							0,0	50,4
37.	Dušan Rychnovský	G Hranice	3	3							0,0	45,0
38.	Jakub Červenka	GŠpitálsPH	1	2							0,0	43,8
39.	David Marek	SPŠ Zlín	3	2			6	9,5			17,3	40,4
40.	Michal Kozák	G Jihlava	3	1		10	10	9,5	10		39,9	39,9
41.	Rudolf Rosa	G Kladno	4	3							0,0	39,6
42.	Martin Kahoun	GJNerudy	4	7							0,0	36,5
43.	Jan Matějka	GJírovco	2	2					4		6,8	35,8
44.	Lucia Simanová	GGrösslin	3	1							0,0	34,9
45.	Karel Tesař	SPŠEPlzeň	1	1							0,0	34,8
46.	Stanislav Fořt	G Tábor	0	4							0,0	34,5
47.	Martin Němec	G Ledeč	3	2							0,0	31,3
48.	Richard Jedlička	G Vlašim	3	8							0,0	30,4
49.	Mirek Jarolím	GMikuláš	1	3							0,0	29,2
50.	Jakub Slavík	GJKTyla	3	1							0,0	28,8
51.	Jan Krajdl	SPŠÚžlabin	2	6		1			4		7,4	28,5
52.	Amadeo Mareš	SOŠ Blatná	2	2	2	0	1	0			6,5	27,7
53.	Václav Strnad	GArabská	3	1							0,0	27,5
54.	Hana Bendová	G ČLípa	4	1		10		9,5	4		27,0	27,0
55.	Radim Pechal	SPŠ Rožnov	4	3							0,0	26,3
56.	David Škorvaga	G Kralupy	4	2							0,0	24,0
57.	Jan Kučera	G Polička	4	2			2				4,0	23,0
58.	Alena Skálová	GNaVPláni	3	1	3	2			10		20,5	20,5
59. – 60.	Milan Klášterka	SPŠKlatovy	3	3							0,0	18,2
	Jan Sixta	G Brandýs	4	1							0,0	18,2
61.	Jakub Pavlík jn.	G Kladno	4	5							0,0	15,6
62.	Karolína Burešová	G ČLípa	0	1							0,0	13,9
63.	Tomáš Volf	G Tábor	0	1							0,0	12,4
64.	Alžběta Pechová	GPodStrání	2	1	1		1	4			12,1	12,1
65.	Miroslav Jančařík	G UBrod	3	4							0,0	10,5
66.	Matěj Pacovský	G Tábor	3	1					8		9,5	9,5
67.	Jan Papoušek	GKptJaroš	3	1							0,0	8,0
68.	Josef Sedlačík	G UBrod	3	1							0,0	6,2
69.	Martin Pástor	SPŠAlejová	2	1							0,0	6,0
70.	Matyáš Bach	G VKlobou	0	1							0,0	2,6
71. – 72.	Veronika Paštyková	G KHora	1	1			1				2,5	2,5
	Petr Pecha	ZŠŠkValKlo	0	1			1				2,5	2,5
73.	Jan Musílek	G NBydžov	3	2		1					2,4	2,4
74.	Radomír Švihel	G Zlín	3	1							0,0	0,0