

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 našťástí 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 liché, můžeme rovnou skončit a oznámit nedočkavému uživateli, že tyto bankovky opravdu rozdělit na polovinu nejdu. 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 psocítáno).

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 polozku $V[s]$. 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 $\mathcal{O}(s) = \mathcal{O}(2^B)$, 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 remizujeme. 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 remizuje, 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 nepřijedeme. 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 odebrali větší číslo (tedy je o něco těžší přijít s protipříkladem, ale 4,2,1,6,8,5 zabere). Takových figlů se dá vymyslet spousta, ale neznám žádný, který opravdu funguje.

Půjdeme na to tedy trochu jinak: Nebudeme se snažit jenom vyhrát, ale dokonce soupeře obrát 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, \dots, A_{N-1} , budou všechny pozice v průběhu hry vždy nějaké intervaly $A_i, \dots, A_{i+\ell-1}$. 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. (Tí zkušenější už samozřejmě správně většinu 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:

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

Z těchto dvou možností si samozřejmě vybereme tu, ve které nahamtaté 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 indukci 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

```
{ nyní slijeme setridené poloviny }
a := 1; b := mid+1;
res := i;
while (a <= mid) and (b <= j) do begin
  if (tmp[al] > tmp[b]) then begin
    count := count + (mid-a+1);
    main[res] := tmp[b];
    inc(b);
  end else begin
    main[res] := tmp[a];
    inc(a);
  end;
  inc(res);
end;
{ zkopirujeme zbytky slévaných částí (jsou-li nějaké) }
while (a <= mid) do begin main[res] := tmp[al]; inc(a); inc(res); end;
while (b <= j) do begin main[res] := tmp[b]; inc(b); inc(res); end;
merge := count;

var
  F : Text;
  N, i : LongInt;
  cisla, cisla2 : TCisla;
begin
  Assign(F, 'cisla.in');
  Reset(F);
  ReadLn(F, 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) );
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)).
q(S,K,X):-setof(H,(member(H,S),H>K),[X|_]).
r(S,K,X):-sort([K|S],R),nextto(K,X,R).

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

zretez(A=B, B=C, A=C).
vloz(Fronta-[NovyPrvek|Y], NovyPrvek, Fronta-Y).
vloz2(X-X, NovyPrvek, [NovyPrvek|X]-X) :- var(X).
vloz2(StaraFronta-X, NovyPrvek, NovaFronta) :- zretez(StaraFronta-X, [NovyPrvek|Y]-Y, NovaFronta).
odeber(X-X) :- var(X), fail.
odeber([H|Zbytek]-X, Zbytek-X, H).
je_prazdna(X-X) :- var(X).

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

% databaze.pl:
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').
hadej([q(Otazka,Ano)|Nel]):-
  write(Otazka), nl, read(Odpoved),
  (Odpoved = ano, hadej(Ano)) ;
  (Odpoved = ne, hadej(Nel)).
hadej(Zvire) :- write('Myslis si zvire '), write(Zvire).
```

```

{ Hlavní algoritmus - zjistí, jak dosáhnout všech možných součtů (naplní pole "Vybrane") }
procedure spocitSoucty(Bankovy: 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
    { zkúsíme všechny bankovky }
    if Bankovy[item] > Sum then Exit;           { našli 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 + Bankovy[item] <= Sum) and (Vybrane[i + Bankovy[item]] = 0) then
        Vybrane[i + Bankovy[item]] := item;    { umím součet i => umím také součet i + nová bankovka }

    if Vybrane[i + Bankovy[item]] = 0 then Vybrane[i + Bankovy[item]] := item;
    max := max + Bankovy[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;

{ Vytáhá z pole Vybrane bankovky a vypíše je na výstup }
procedure zapsiVysledky(Bankovy: 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.');
```

Úloha 19-5-4 – Lodní mrazáky – program

```

unit Podpalubi;

uses Mrazaky;
var Prichozí, Odchozí: Mrazak;           { Naše dva mrazáky }

procedure uloz( Co: Potravinina );
begin
  vlozDoMrazaku( Co, Prichozí ); end;

function snez: Potravinina;
begin
  if prazdnyMrazak( Odchozí ) then begin
    if prazdnyMrazak( Prichozí ) then begin { Hladomor }
      snez := HrnicekOdMedu;
      exit;
    end;
    while not prazdnyMrazak( Prichozí ) do vlozDoMrazaku( vyndejZMrazaku( Prichozí ), Odchozí );
  end;
  snez := vyndejZMrazaku( Odchozí );
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 - tríděný 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 tríděného intervalu a metodou rozdělí a panuj spočítáme podúlohy }
  mid := (i+j) div 2;
  count := merge(tmp, main, i, mid);
  count := merge(tmp, main, mid+1, j) + count;
```

$H_{i,\ell}$. 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]; // Číslička, se kterými hrajeme
int H[MAX][MAX]; // Maximální zisk v každém úseku
int T[MAX][MAX]; // Vyhraňující tah pro úsek
```

```

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

Ostatně, k těmúž výsledku bychom se také mohli dobrat tak, že bychom si napsali rekurzivní funkci, která by počítala $H_{i,\ell}$ podle našich vzoreček. 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 úlozku 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 $\mathcal{O}(n\sqrt{n}) = \mathcal{O}(n^{3/2})$. Mezi-lehlé 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 $\mathcal{O}(n^2p)$ 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, řekjeme mu třeba přichozí, budou potraviny seřazeny tak, že čím hlouběji v mrazáku bude, tím starší budou.
- 2) Druhý mrazák, budíž nazýván odchozí, bude seřazen přesně naopak, tedy čím hlouběji, tím novější potraviny.
- 3) Libovolná potravina v odchozím mrazáku bude starší, než libovolná v přichozím.

Teď zbývá vytvořit operace uložit a sněžit tak, aby žádnou z předchozích podmínek neporušily. Operace uložit je jednoduchá, prostě vložíme potravinu do přichozího 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 nedotkne, ani podmínku 3), protože tato potravina je určitě novější, než cokoliv v odchozím mrazáku.

Pokud bude v odchozím mrazáku alespoň jedna potravina, je operace sněžit také triviální. Potravina navrchu tohoto mrazáku je nejstarší, protože je starší než cokoliv pod ní a je starší než cokoliv v přichozím 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 prázdnou? Předpokládáme, že je alespoň jedna potravina v přichozím mrazáku, protože jinak by na lodi panoval hladomor a nebylo by možné provést operaci sněžit. Celkově nejstarší potravina je tedy nejstarší potravina v přichozím mrazáku, která je dle podmínky 1) vespod. Proto přeházíme všechny potraviny z přichozího do odchozího mrazáku. Tím se otočí pořadí potravin, takže nejstarší bude navrchu a postupně budou novější a novější. Tím je splněna třetí podmínka a zařizeno, že lze provést operaci sněžit tak, jak byla popsána výše. První a druhá jsou splněny také, protože přichozí mrazák je prázdný.

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

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

Michal „vornér“ Vaner

19-5-5 Praktická úlozka – Počet inverzí

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

Náš postup je založen na známém trídícím algoritmu mergesort, neboli trídění pomocí slévání. Tento algoritmus pracuje na principu rozdělí a panuj. Tríděnou posloupnost rozdělí

na dvě poloviny (tedy podúlohy menšího rozsahu), které setřídí rekurzivním použitím stejného algoritmu. Setříděné poloviny následně slijí 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) dojde 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 dojde 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 zvorovém 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 vhodné 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 úlohy. 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 záležitosti, a protože se našlo jenom několik odváživců, rozhodla jsem se nešfouat 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;           /* čas příjezdu do další stanice */
    int prijezd;             /* cena cesty do další stanice */
    int cena;
    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 *odjedy[1440];
struct prijezd *prijezdy[1440];
struct prijezd *cesty[MAX_STANIC];    /* Aktuálně nejlépejší 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\n", &d, &n, &s, &c);
    for (i=0; i<n; i++) {
        int k, mzst, mcas, mcena, szst, scas, scena, ch, cm;
        scanf("%d %d %d %d", &k, &mzst, &ch, &cm, &mcena);
        mcas = 60*ch+cm;
        for (j=i; j<k; j++) {
            scanf(" %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 = odjedy[mcas]; odjedy[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++) {
        for (p = prijezdy[t]; p; p=p->dalsi)
            if (!(cesty[p->stanice] || cesty[p->stanice]->cena > p->cena)
                cesty[p->stanice] = p;

        for (o = odjedy[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]) {
        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 procedura 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;
```