36. ročník matematické olympiády na středních školách

Kategorie P

In: František Zítek (editor); Leo Boček (editor); Karel Horák (editor); Pavel Töpfer (editor): 36. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 1986/87. 28. mezinárodní matematická olympiáda. (Czech). Praha: Státní pedagogické nakladatelství, 1989. pp. 156–213.

Kategorie P

Ve 36. ročníku matematické olympiády se již podruhé soutěžilo také v kategorii P (programování). Tato kategorie je určena všem žákům středních škol bez rozdílu věku, úlohy zde zadávané jsou zaměřené na tvorbu a analýzu algoritmů. Oproti loňskému ročníku zájem o kategorii P značně vzrostl. Zatímco ve 35. ročníku MO se do soutěže zapojilo asi 250 studentů, o rok později jich řešilo úlohy domácího kola více než 400.

Soutěž je organizována tříkolově, soutěží se v domácím, krajském a celostátním kole. Řešení úloh domácího kola museli soutěžící odevzdat do 5. 2. 1987. Úspěšní řešitelé byli pozváni do krajského kola, které se konalo ve středu 8. 4. 1987. Padesát nejlepších účastníků krajského kola se sešlo ve dnech 14.—17. 5. 1987 v Praze na celostátním kole MO kategorie P, jehož pořadatelem byl KV MO Praha.

Po odborné stránce je kategorie P matematické olympiády zajišťována odbornými centry na vysokých školách - na MFF UK Praha, PřF UJEP Brno a MFF UK Bratislava. Pracovníci těchto center opravovali řešení úloh ve všech třech kolech soutěže, podíleli se na přípravě studentů na soutěž, na zabezpečení krajských kol i kola celostátního. Odborným garantem kategorie P 36. ročníku matematické

olympiády bylo centrum na katedře kybernetiky a informatiky MFF UK Praha, ze kterého také pocházejí zadání všech úloh.

Tabulka 7

Počty žáků soutěžících v kategorii P 36. ročníku MO

Kraj	Kolo					
	domácí		krajské		celostátní	
	S	U	S	U	S	U
Praha	56	42	42	24	8	7
Středočeský	25	10	9	3	3	1
Jihočeský	26	13	13	3	3	1
Západočeský	6	0	0	0	0	0
Severočeský	21	8	8	3	3	1
Východočeský	32	15	13	6	3	3
Jihomoravský	24	17	15	7	4	3
Severomoravský	35	28	20	14	6	2
Bratislava	53	31	30	16	12	6
Západoslovenský	32	14	14	2	2	2
Středoslovenský	43	23	23	3	1	1
Východoslovenský	61	34	34	9	5	2
ČSR	225	133	120	60	30	18
SSR	189	102	101	30	20	11
ČSSR	414	235	221	90	50	29

S — počet všech soutěžících
U — počet úspěšných řešitelů

VÝSLEDKY CELOSTÁTNÍHO KOLA MO KATEGORIE P

Vítězové

1. *Pavel Kozlovský*, 3., G Jindřichův Hradec
2. *Vladan Majerech*, 4., G Pardubice
3. *Branislav Stríženec*, 3., G J. Hronca, Bratislava
4. *Vladimír Veselý*, 4., G J. Hronca, Bratislava
5. *Peter Klein*, 4., G A. Markuša, Bratislava
6. *Pavol Kolník*, 4., G Nové Mesto nad Váhom

7.—8. *Rudolf Burcl*, 4., G Trnava
Rastislav Senderák, 4., G Prešov

9.—12. *Ilja Martišovitš*, 2., G J. Hronca, Bratislava
Marcel Polakovič, 4., G A. Markuša, Bratislava
Vladimír Solnický, 3., G Opava
Petr Steinmetz, 4., G Brno, Koněvova ul.

Další úspěšní řešitelé

13.—14. *Petr Brož*, 2., G W. Piecka, Praha
Robert Hetka, 4., G Benešov

15.—18. *Tibor Bartoš*, 3., G A. Markuša, Bratislava
Petr Čížek, 2., G W. Piecka, Praha
Lucie Kárná, 4., G W. Piecka, Praha
Petr Mandík, 4., G Děčín

19. *Radek Porazil*, 3., G Bílovec

20.—22. *Václav Bohdanecký*, 2., G W. Piecka, Praha
Arnošt Kobylka, 2., G W. Piecka, Praha
Jan Sochor, 4., G W. Piecka, Praha

23.—29. *Michal Dostál*, 4., G W. Piecka, Praha
Petr Fencl, 4., G Pardubice
Jiří Jaruška, 4., G Pardubice
Pavel Kafka, 4., G Třebíč
Richard Krajčoviech, 4., G Považská Bystrica
Rado Mróz, 3., G Spišská Nová Ves
Marek Velešík, 2., G Brno, Koněvova ul.

POŘADÍ ÚSPĚŠNÝCH ŘEŠITELŮ KRAJSKÉHO KOLA MO KATEGORIE P

V seznamu je uvedeno nejvýše prvních deset úspěšných řešitelů z každého kraje. Typ školy není uváděn, všichni jsou studenty gymnázia.

Praha

Michal Dostál, 4., W. Piecka, Praha 2, *Jan Sochor*, 4., W. Piecka, Praha 2, *Arnošt Kobylka*, 2., W. Piecka, Praha 2, *Filip Pejša*, 4., Voděradská, Praha 10, *Petr Čížek*, 2., W. Piecka Praha 2, *Petr Brož*, 2., W. Piecka, Praha 2, *Václav Bohdanecký*, 2., W. Piecka, Praha 2, *Lucie Kárná*, 4., W. Piecka, Praha 2, *Michal Kopecký*, 4., W. Piecka, Praha 2, *Jan Dvořák*, 3., Sladkovského nám., Praha 3.

Středočeský kraj

Petr Vyhňák, 3., Mladá Boleslav, *Robert Hetka*, 4., Benešov, *Vladimír Šolc*, 1., Beroun.

Jihočeský kraj

Pavel Kozlovský, 3., Jindřichův Hradec, *Martin Zítek*, 4., Milevsko, *Jakub Čermák*, 1., Jírovcova, České Budějovice.

Severočeský kraj

Dan Lukeš, 2., Partyzánská, Liberec, *Petr Mandík*, 4., Děčín, *Jiří Martinek*, 3., Ústí nad Labem.

Východočeský kraj

Jiří Jaruška, 4., Pardubice, *Vladan Majerech*, 4., Pardubice, *Petr Fencl*, 4., Pardubice, *Petr Kousal*, 3., Tylovo náb., Hradec Králové, *Petr Krákora*, 3., Trutnov, *Radko Martínek*, 3., Jičín.

Severomoravský kraj

Radek Porazil, 3., Bílovec, *David Jedelský*, 3., Ostrava-Hrabůvka, *Vladimír Solnický*, 3., Opava, *Jan Hřebíček*, 4., Valašské Meziříčí, *Michal Heřmanský*, 4., Valašské Meziříčí, *Tomáš Látal*, 3., Olomouc-Hejčín, *Radek Vingrálek*, 4., Olomouc, *David Šindler*, 2., Bílovec, *Marek Hübner*, 3., Karviná, *Antonín Dvořák*, 3., Přerov.

Jihomoravský kraj

Pavel Kafka, 4., Třebíč, *Petr Steinmetz*, 4., Koněvova, Brno' *Miloslav Hledík*, 3., Ivančice, *Marek Velešík*, 2., Koněvova' Brno, *Jan Dvořák*, 4., Moravské Budějovice, *Radek Švenda*' 2., Uherský Brod, *Zdeněk Vonský*, 2., Ivančice, *Robert Maar*' 4., Žďár nad Sázavou, *Miroslav Minárik*, 4., Jihlava, *Petr Kolenčík*, 1., Koněvova, Brno.

Bratislava

Vladimír Veselý, 4., J. Hronca, Novohradská, Bratislava,

Branislav Stríženec, 3., J. Hronca, Bratislava, *Ilja Martišovitš*, 2., J. Hronca, Bratislava, *Stanislav Párnický*, 4., A. Markuša, Červenej armády, Bratislava, *Tibor Bartoš*, 3., A. Markuša, Bratislava, *Miroslav Šrol*, 3., J. Hronca, Bratislava, *Anton Belan*, 4., A. Markuša, Bratislava, *Martin Bujdák*, 3., A. Markuša, Bratislava, *Marcel Polakovič*, 4., A. Markuša, Bratislava, *René Pázman*, 2., J. Hronca, Bratislava.

Západoslovenský kraj

Pavol Kolník, 4., Nové Mesto nad Váhom, *Rudolf Burcl*, 4., Trnava.

Středoslovenský kraj

Richard Krajčoviech, 4., Považská Bystrica, *Silvia Badáková*, 4., Prievidza, *Jozef Gomela*, 3., Prievidza.

Východoslovenský kraj

Mário Drosc, 4., Michalovce, *Rado Mróz*, 3., Spišská Nová Ves, *Roman Soták*, 4., Šmeralova, Košice, *Martin Lieskovský*, 4., Prešov, *Rastislav Senderák*, 4., Prešov, *Peter Fekete*, 4., Michalovce, *Zdeno Kálnássy*, 3., Prešov, *Robert Mráz*, 3., Poprad, *Ladislav Šteffko*, 4., Šrobárova, Košice.

P - I - 1

Konečnou posloupnost čísel nazveme symetrickou, jestliže se nezmění, když zapíšeme její prvky v obráceném pořadí. Nalezněte a dokažte (co nejlepší) algoritmus, který pro libovolnou konečnou posloupnost čísel určí délku jejího nejdelšího souvislého úseku, který je symetrickou posloupností.

Př.: Pro posloupnost 3, 1, 2, 3, 2, 1, 4, 2, 1 je tato délka 5 (úsek 1, 2, 3, 2, 1).

Řešení. Počet čísel v zadané konečné posloupnosti označíme N. Rychlý algoritmus vykoná při prohledávání posloupnosti délky N řádově N^2 operací. Většina řešitelů nalezla některý z dalších správných algoritmů, ovšem se složitostí N^3.

Symetrická podposloupnost maximální délky může obsahovat buď lichý, nebo sudý počet čísel. Podle toho je jejím středem buď některý z prvků zadané posloupnosti, nebo pozice mezi sousedními prvky. Budeme postupně vyšetřovat všech $2N - 1$ míst (N prvků, $N - 1$ mezer mezi nimi), kde může ležet střed maximální symetrické podposloupnosti. Pro každé z těchto míst určíme délku maximálního symetrického úseku se středem v tomto místě. Toto určení je snadné: porovnáváme postupně od zvoleného středu směrem k oběma okrajům

posloupnosti dvojice čísel tak dlouho, dokud jsou porovnávaná čísla shodná nebo dokud nenarazíme na okraj posloupnosti. Pro každý ze zvolených středů se vykoná maximálně $N/2$ porovnání. Výsledkem algoritmu je maximum z délek, které jsme získali pro jednotlivé středy symetrické podposloupnosti.

Algoritmus zapíšeme v programovacím jazyce Pascal. Budeme předpokládat, že proměnná N obsahuje délku posloupnosti a že v poli A jsou jako prvky $A[1], \ldots, A[N]$ uloženy prvky posloupnosti. Dále předpokládáme deklaraci celočíselných proměnných I a MAX. Proměnná I je pomocná, výsledná délka bude uložena v proměnné MAX.

```
procedure TEST (J, K: integer);
var OKRAJ: Boolean;
    DELKA: integer;
begin
  OKRAJ := false;
  while (A[J] = A[K]) and not OKRAJ do
    begin
    {zde vždy platí: úsek A[J],..., A[K] je symetrický}
    J := J - 1; K := K + 1;
    if (J = 0) or (K = N + 1) then OKRAJ := true
    end; {zde platí: úsek A[J + 1],..., A[K - 1] je
    symetrický, ale již ho nelze prodloužit}
  DELKA := K - J - 1;
  if DELKA > MAX then MAX := DELKA
end;
begin
  if N = 0 then MAX := 0
  else if N = 1 then MAX := 1
```

```
  else
    begin
    MAX := 0;
    for I := 1 to N - 1 do
      begin TEST (I, I); {lichá délka}
            TEST (I, I + 1) {sudá délka}
      end
    end
end
```

Uvedený algoritmus je možné ještě dále vylepšovat, ale zrychlení výpočtu již nebude příliš významné a zápis algoritmu se stane složitějším. Je možné například vybírat středy podposloupností od středu celé posloupnosti souběžně směrem k oběma okrajům a výpočet ukončit již ve chvíli, kdy momentální hodnota proměnné *MAX* dosáhne dvojnásobku vzdálenosti právě uvažovaného středu podposloupnosti od okraje posloupnosti (neboť dále už není možné hodnotu proměnné *MAX* zlepšit).

Důkaz správnosti algoritmu vyplývá přímo z jeho popisu. Výpočet podle algoritmu je konečný, neboť algoritmus je tvořen pouze cykly s pevně omezeným počtem opakování (**for**-cyklus v hlavním programu se provede přesně $N-1$krát, **while**-cyklus v proceduře *TEST* se vykoná vzhledem k omezení délky posloupnosti hodnotou N maximálně $N/2$krát). Zadaná posloupnost jistě obsahuje nějakou maximální symetrickou podposloupnost. Má-li tato podposloupnost lichý počet členů a jejím středem je prvek $A[S]$, bude nalezena při S-tém průchodu **for**-cyklem (pro $I = S$) při vyvolání *TEST*(I, I). Má-li sudý počet členů a její střed leží mezi

prvky $A[S]$ a $A[S + 1]$, bude nalezena při S-tém průchodu **for**-cyklem při vyvolání procedury $TEST(I, I + 1)$. Do proměnné MAX se ukládá maximum z délek všech nalezených symetrických podposloupností, které již nelze prodloužit. Také nejdelší z nich bude jednou nalezena a v proměnné MAX proto zůstane na závěr algoritmu její délka.

P - I - 2

Sjednocení konečné množiny uzavřených intervalů na reálné přímce je množina, která je složena z konečného počtu disjunktních souvislých úseků, z nichž každý je opět uzavřený interval.

Nalezněte a dokažte (co nejlepší) algoritmus, který pro libovolnou konečnou množinu uzavřených intervalů zjistí počet disjunktních souvislých úseků jejich sjednocení (tj. skutečný počet intervalů). Intervaly jsou zadány výčtem dvojic čísel, které v daném pořadí určují jejich dolní a horní mez.

Řešení. Označme posloupnost dolních mezí D a horních mezí H, obě mají délku N. K řešení úlohy lze užít několika odlišných algoritmů. Základem těch nejlepších je vhodné setřídění zadaných intervalů. K tomu je třeba provést řádově $N.\log_2 N$ operací (to je časová složitost nejlepších třídících algoritmů). Tímto výrazem je pak určena i efektivita celého algoritmu, neboť po setřídění stačí jednou sekvenčně projít všechny intervaly a při tomto průchodu provést příslušné výpočty.

Algoritmus řešení popíšeme slovně:

1. Pokud $N = 1$, polož $POCET = 1$ a jdi na 8.
2. Setřiď zadané intervaly vzestupně podle hodnoty dolní

meze. Má-li více intervalů stejnou dolní mez, na jejich pořadí nezáleží. Po setřídění tedy bude platit: $D[1] \leqq D[2] \leqq \ldots \leqq D[N]$. *Pozn.:* třídíme nejen dolní meze, ale celé intervaly, tzn. souběžně se změnami pořadí v posloupnosti D přemísťujeme i hodnoty horních mezí v posloupnosti H.

3. Polož $POCET = 1$, $HORMEZ = H[1]$, $I = 2$.
4. Jestliže $D[I] > HORMEZ$, zvětši $POCET$ o 1.
5. Jestliže $H[I] > HORMEZ$, polož $HORMEZ = H[I]$.
6. Zvětši I o 1.
7. Jestliže $I \leqq N$, jdi na 4.
8. Výsledný počet souvislých disjunktních intervalů je v proměnné *POCET*. Konec.

Algoritmus zvlášť řeší případ jediného intervalu na vstupu - - bod 1. Je-li na vstupu více intervalů, provede jejich setřídění podle hodnot dolních mezí - bod 2. Třídící algoritmus uvedeme na konci řešení. Potom postupně prochází všechny intervaly. Dokud intervaly patří do jednoho souvislého úseku, pouze aktualizuje horní mez jejich sjednocení (proměnná *HORMEZ*) - bod 5. Jakmile má některý interval prázdný průnik s intervaly zpracovávaného úseku, algoritmus zaregistruje ukončený souvislý úsek (proměnná *POCET*) a začne vytvářet další - bod 4.

Zbývá podat zdůvodnění správnosti algoritmu. Algoritmus jistě skončí výpočet po konečně mnoha krocích, neboť setřídění N intervalů je konečný proces a v cyklu (body algoritmu 4 až 7) roste při každém průchodu hodnota proměnné I od 2 do N, takže průchodů bude méně než N.

Je-li zadán jediný interval ($N = 1$), bude mít proměnná *POCET* správnou hodnotu 1 z bodu 1. Tvoří-li všechny za-

dané intervaly jediný souvislý úsek, bude mít proměnná *POCET* hodnotu 1 z bodu 3. Hodnota proměnné *POCET* se zvětšuje o 1 pouze v bodu 4, a to přesně tehdy, když zpracovávaný *I*-tý interval má dolní mez větší, než je maximum horních mezí všech předchozích intervalů (uložené v proměnné *HORMEZ*). Vzhledem k vzestupnému setřídění dolních mezí intervalů platí nerovnosti

$$HORMEZ < D[I] \leqq D[I + 1] \leqq \ldots \leqq D[N]$$

čili také všechny následující intervaly mají dolní mez větší, než je hodnota *HORMEZ*. Proto *I*-tým intervalem skutečně začíná další souvislý úsek.

Na závěr se vraťme k problému třídění. Máme za úkol uspořádat N čísel vzestupně podle velikosti. Třídících algoritmů existuje celá řada. Nejrychlejší z nich vyžadují provedení řádově $N.\log_2 N$ porovnání. Ukážeme si jeden z takových třídících algoritmů. Chceme setřídit čísla $D[1], D[2], \ldots, D[N]$ uložená v poli D. Rozdělíme pole D do dvou úseků $D[1], \ldots, D[(N + 1)$ **div** $2]$ a $D[(N + 1)$ **div** $2 + 1], \ldots, D[N]$. Oba úseky mají délku stejnou nebo lišící se jen o 1. Každý z těchto úseků zvlášť setřídíme rekurzívním vyvoláním téhož třídícího algoritmu. Ze setříděných úseků pak již snadno vytvoříme jedinou uspořádanou posloupnost všech čísel. Do výsledné posloupnosti vždy zařadíme to z prvních čísel v úsecích, které je menší. Toto číslo zároveň vynecháme z jeho úseku. Postup opakujeme, dokud do posloupnosti není zařazeno všech N čísel.

Popsaný algoritmus zapíšeme v Pascalu ve tvaru rekurzívní procedury *SORT*. Pro utřídění celého pole D bude tato

procedura volána s parametry *SORT*(1, *N*). Dále předpokládáme, že je deklarováno pomocné globální pole *A* stejného typu jako je pole *D*.

```
procedure SORT (J, K: integer);
var P, I, I1, I2: integer;
begin
  if J < K then
    begin
    P := (J + K) div 2;
    SORT (J, P);
    SORT (P + 1, K);
    I := J;
    I1 := J; I2 := P + 1;
      while I <= K do
        if D[I1] < D[I2] then
          begin
          A[I] := D[I1];
          I := I + 1; I1 := I1 + 1;
          if I1 > P then
            while I <= K do
              begin
              A[I] := D[I2];
              I := I + 1; I2 := I2 + 1;
              end
          end
        else
          begin
          A[I] := D[I2];
          I := I + 1; I2 := I2 + 1;
```

```
        if I2 > K then
          while I <= K do
            begin
            A[I] := D[I1];
            I := I + 1; I1 := I1 + 1
            end
        end;
    for I := J to K do D[I] := A[I]
    end
end;
```

Procedura *SORT* je rekurzívně volána vždy k setřídění úseku přibližně poloviční délky. Začínáme-li od pole délky N, znamená to, že rekurze se bude provádět do hloubky asi $\log_2 N$. Na každé úrovni hloubky rekurze vyžaduje spojení setříděných úseků čísel do úseků dvojnásobné délky nejvýše N porovnání čísel. Celkem je tedy k setřídění N čísel uvedeným algoritmem zapotřebí řádově $N.\log_2 N$ porovnání.

P - I - 3

Je dán Euklidův algoritmus pro výpočet největšího společného dělitele celých čísel A a B, kde $A > B > 0$.

PASCAL	BASIC
while B<>0 **do**	10 IF B = 0 THEN 60
begin C := A **mod** B;	20 LET C = A MOD B
A := B;	30 LET A = B
B := C	40 LET B = C
end;	50 GOTO 10
NSD := A	60 LET NSD = A

Operace A MOD B znamená zbytek po celočíselném dělení čísla A číslem B. Dokažte, že počet opakování cyklu v algoritmu je vždy menší než $2.\log_2 A$.

Řešení. Mezi správnými řešeními této úlohy se objevily v podstatě dva základní postupy. My si zde ukážeme názornější a jednodušší z nich, který je založen na úvahách o změnách hodnoty proměnné A v průběhu výpočtu. Druhý postup řešení vychází ze srovnání posloupnosti hodnot proměnné A během výpočtu s Fibonacciho posloupností a dále Fibonacciho posloupnosti s posloupností $(2^{n/2})$.

Nejprve dokážeme pomocné tvrzení: Jestliže A, B jsou libovolná přirozená čísla, $A > B > 0$, potom $A \bmod B < A/2$. Důkaz provedeme rozborem případů:

a) $B > A/2$ ... pak $A \bmod B = A - B < A - A/2 = A/2$,

b) $B = A/2$ ... pak $A \bmod B = 0$ a z předpokladu $A > 0$ plyne výsledek $A \bmod B < A/2$,

c) $B < A/2$ ... vždy platí $A \bmod B < B$, tedy pro $B < A/2$ přímo vyplývá výsledek.

Nyní již můžeme přikročit k důkazu tvrzení ze zadání úlohy. Nebudeme zde zabíhat do formálních podrobností a technických detailů, zaměříme se jen na hlavní myšlenky důkazu. Na základě našeho pomocného tvrzení je možné snadno ukázat, že vždy po dvou průchodech cyklem v Euklidově algoritmu se hodnota proměnné A zmenší na méně než polovinu své původní hodnoty. Proměnná A totiž po dvou průchodech cyklem nabude hodnoty $A \bmod B$, která je podle pomocného tvrzení vždy menší než $A/2$. Kdyby se hodnota proměnné A zmenšovala po dvou průchodech cyklem vždy jen na polovinu až do hodnoty 1 (tj. na začátku by bylo A tvaru $A = 2^k$

a pak vždy po dvou průchodech 2^{k-1}, 2^{k-2}, ..., 1), bylo by těchto dvojic průchodů cyklem třeba vykonat $k = \log_2 A$. Celkem by se tedy vykonalo $2.\log_2 A$ průchodů cyklem. Ve skutečnosti se hodnota proměnné A zmenšuje rychleji vzhledem k ostré nerovnosti A mod $B < A/2$. Navíc výpočet nemusí končit až při $A = 1$, může i dříve. Proto se při celém výpočtu provede celkem méně než $2.\log_2 A$ průchodů cyklem.

P - I - 4

Zapište ve zjednodušeném jazyce LISP definice následujících funkcí:

a) MEMBER [X;S]
   S musí být seznam
   hodnotou funkce je T, jestliže výraz X je prvkem seznamu S, jinak je hodnotou F

b) DELETE [X;S]
   S musí být seznam
   hodnotou funkce je seznam, který vznikne vypuštěním prvku X ze seznamu S (vypouští se pouze první výskyt); není-li X prvkem seznamu S, je hodnotou funkce původní seznam S.

Při řešení můžete nejprve definovat jednodušší pomocné funkce a s jejich využitím pak funkce hlavní.

Poznámka. V každém kole obdrželi soutěžící shodný krátký studijní text o zjednodušené verzi programovacího jazyka LISP. Tento text nyní uvádíme v plném znění. U úloh P-II-4 a P-III-4 zde v ročence studijní text již neopakujeme.

Zjednodušený LISP

Definujeme zjednodušenou verzi programovacího jazyka LISP. Data v tomto jazyce mají tvar tzv. symbolických výrazů. Nejjednoduššími výrazy jsou atomy, které jsou dvou typů: číselné a nečíselné. Zápis číselných atomů je stejný jako zápis celých čísel, např. −125, +3, 10. Nečíselné atomy jsou shodné s identifikátory.

Složitější symbolické výrazy nazýváme seznamy. Seznam může být prázdný nebo neprázdný. Neprázdný seznam je tvořen jedním nebo více prvky, uzavřenými v okrouhlých závorkách. Přitom záleží na pořadí prvků a stejný prvek se může v seznamu vyskytovat několikrát. Každý prvek seznamu je sám výrazem, buď je to atom, nebo opět seznam. Prázdný seznam neobsahuje žádný prvek a zapisuje se znaky ( ).

Příklady seznamů:

( )	prázdný seznam
(X)	seznam obsahující jediný prvek - atom X
(X Y Z)	tříprvkový seznam tvořený atomy X, Y a Z
((X) (Y Z))	dvouprvkový seznam tvořený dvěma prvky, z nichž první je jednoprvkový seznam obsahující atom X, a druhý je dvouprvkový seznam tvořený atomy Y a Z

Pro zpracovávání symbolických výrazů je v jazyce LISP zavedeno několik elementárních funkcí:

CAR[S]	S musí být neprázdný seznam, hodnotou funkce je první prvek seznamu S
CDR[S]	S musí být neprázdný seznam, hodnotou funk-

	ce je seznam, který vznikne z S vynecháním prvního prvku
CONS[R;S]	S musí být seznam (neprázdný nebo i prázdný); hodnotou CONS[R;S] je nově vytvořený seznam, jehož prvním prvkem je výraz R a všechny další prvky vzniknou překopírováním všech prvků seznamu S
EQ[R;S]	R, S musí být atomy; jsou-li atomy R a S identické, je hodnotou funkce atom T (true), jinak je hodnotou atom F (false)
ATOM[S]	hodnotou je T, je-li S atom, jinak je hodnotou F
NULL[S]	argumentem funkce musí být seznam; hodnotou funkce je T, když seznam je prázdný, jinak je hodnotou F

Příklady:

CAR[(A B C)] = A
CDR[(A B C)] = (B C)
CONS[A;(B C)] = (A B C)
CONS[(A B); (C D)] = ((A B) C D)
EQ[1;2] = F
EQ[X;0] = T, jestliže X = 0, F jinak
ATOM[(A)] = F
CONS[A;( )] = (A)
CAR[(A)] = A
CDR[(A)] = ( )

Z elementárních funkcí lze složit užitím podmíněného výrazu a definice funkce složitější funkce. Podmíněný výraz zapisujeme ve tvaru

$$[P1 \rightarrow E1;\ P2 \rightarrow E2;\ \ldots;\ Pn \rightarrow En],$$

kde P1, P2, ..., Pn jsou symbolické výrazy, které mají význam podmínek a které smějí nabývat pouze hodnot T a F.

Tyto podmínky jsou vyhodnocovány postupně zleva doprava tak dlouho, dokud se nenarazí na první podmínku, která má hodnotu T. Nechť je to Pk. Pak hodnotou podmíněného výrazu je hodnota Ek. To může být atom, seznam, podmíněný výraz nebo volání funkce. Aby měl podmíněný výraz smysl, musí aspoň jeden z výrazů Pk nabývat hodnoty T. K zajištění tohoto požadavku bývá zvykem na místě poslední podmínky Pn psát přímo atom T.

Novou funkci zadefinujeme tak, že napíšeme:

NÁZEV [SEZNAM PARAMETRŮ] = VÝRAZ

Parametry v seznamu oddělujeme středníky. Výraz v definici musí být podmíněný výraz nebo volání funkce. V definicích funkcí lze libovolně užívat rekurzívního volání funkcí. Je to obrat velmi častý a u řady funkcí nezbytný. Znamená to, že v definici funkce můžeme použít libovolné elementární i složitější definované funkce včetně té, kterou právě definujeme, jak to ukazuje i následující příklad.

Př.: Chceme definovat funkci EQUAL[X;Y] takovou, že její hodnotou je T, jsou-li X a Y stejné symbolické výrazy, jinak je hodnotou F.

Definice:

$$\begin{aligned} \text{EQUAL}[X;Y] = [\text{ATOM}[X] \rightarrow [\text{ATOM}[Y] \rightarrow \text{EQ}[X;Y];\\ T \rightarrow F];\\ \text{ATOM}[Y] \rightarrow F; \end{aligned}$$

NULL[X] → NULL[Y];
NULL[Y] → F;
EQUAL[CAR[X]; CAR[Y]] →
    EQUAL[CDR[X]; CDR[Y]];
T → F]

Význam jednotlivých částí definice funkce je následující:

1. Je-li X atom, pak rozlišujeme dva případy. Je-li také Y atom, pak výsledek závisí na tom, zda jsou si rovny, jinak (není-li Y atom) je hodnota F.
2. Jestliže nenastal případ 1 (není-li X atom) a přitom Y je atom, je výsledek F.
3. Víme již, že X a Y jsou seznamy. Je-li první seznam prázdný, tak výsledek závisí na tom, je-li prázdný i druhý seznam.
4. Jestliže nenastal případ 3 a druhý seznam je prázdný, pak je výsledkem F (první seznam byl neprázdný).
5. Víme již, že X a Y jsou neprázdné seznamy. Porovnáme jejich první prvky a jestliže se rovnají, je nutno porovnat i zbytky seznamů. (V obou případech využíváme rekurzívního volání právě definované funkce. Jejími argumenty jsou však kratší seznamy.)
6. Poslední případ může nastat jedině tehdy, když se první členy seznamů X a Y nerovnají, takže výsledkem je F.

Řešení soutěžní úlohy P-I-4. V řešení úlohy budeme využívat pomocnou funkci EQUAL [X;Y]. Její hodnotou je atom T, jsou-li X a Y stejné symbolické výrazy, jinak je hodnotou F. Popis této funkce je uveden jako příklad ve studijním textu, proto ho zde neopakujeme.

a) Definujeme funkci MEMBER[X;S]; předpokládáme, že S je seznam:

```
MEMBER[X;S] = [NULL[S] → F;
               EQUAL[X;CAR[S]] → T;
               T → MEMBER[X;CDR[S]]]
```

1. Je-li S prázdný seznam, neobsahuje žádný prvek, tedy ani X. Hodnotou funkce MEMBER je proto F.
2. Pokud se výraz X rovná prvnímu prvku seznamu S, pak X je v S obsažen a hodnotou funkce je T.
3. V opačném případě budeme zkoumat, jestli se X nachází ve zbytku S po vynechání prvního prvku. Funkce MEMBER je rekurzívně volána na kratší seznam.

b) Definujeme funkci DELETE[X;S], předpokládáme, že S je seznam:

```
DELETE[X;S] = [NULL[S] → S;
               EQUAL[X;CAR[S]] → CDR[S];
               T → CONS[CAR[S];
                 DELETE[X;CDR[S]]]]
```

1. Je-li S prázdný seznam, jistě X neobsahuje a hodnotou funkce je opět prázdný seznam.
2. Jestliže se výraz X rovná prvnímu prvku seznamu S, vypustíme ho. Hodnotou funkce DELETE pak bude zbytek seznamu S bez prvního prvku.
3. V opačném případě bude hodnotou funkce seznam, který vznikne spojením prvního prvku původního seznamu S a zbytku seznamu S s vypuštěným prvním výskytem výrazu X. Vypuštění prvního výskytu X ze zbytku S dosáhneme rekurzívním voláním funkce DELETE na zkrácený seznam CDR[S].

Jestliže výraz X nebude v seznamu S vůbec obsažen, bude

hodnotou funkce seznam totožný s původním seznamem S. (Nikdy se neuplatní příkaz na 2. řádku v definici funkce.)

Existuje i celá řada zcela odlišných správných řešení úlohy. Většinou jsou ale podstatně složitější na zápis i z hlediska rychlosti výpočtu, často jsou také složitější na pochopení. Nemůžeme je zde všechny rozepisovat, ale uvedeme ještě alespoň základní myšlenku, jak je také možné zapsat funkci DELETE. Můžeme využít tří pomocných funkcí následujících vlastností. První pomocná funkce na základě parametrů S a X vytvoří seznam, který vznikne jako část seznamu S od prvního prvku až do prvního výskytu výrazu X (již bez výrazu X) nebo až do konce, pokud se X v S nevyskytuje. Druhá pomocná funkce naopak z S vypustí tento počáteční úsek až do prvního výskytu X a vypustí ještě i samotný prvek X (pokud se v S vyskytuje). Jestliže se výraz X v seznamu S nevyskytuje, bude výsledkem této pomocné funkce prázdný seznam. Konečně třetí pomocná funkce provádí spojení dvou seznamů za sebe. Příklad takové funkce najdeme v řešení úlohy P-III-4. V definici funkce DELETE je pak volána tato třetí pomocná funkce tak, že za její parametry jsou dosazeny seznamy vzniklé výpočtem prvních dvou pomocných funkcí (s hodnotami parametrů X a S).

P - II - 1

Nalezněte a dokažte (co nejlepší) algoritmus, který pro libovolnou konečnou posloupnost čísel nalezne maximální K tak, že existuje nějaká posloupnost délky K, která se v zadané posloupnosti vyskytuje alespoň na dvou různých místech jako souvislý úsek. Tyto úseky se mohou částečně překrývat.

Př.: Pro posloupnost 6, 2, 3, 2, 3, 2, 3, 1, 7 je $K = 4$. (Dvakrát se opakuje posloupnost 2, 3, 2, 3.)

Řešení. Úlohu je možné řešit více různými algoritmy. Jednoduchý algoritmus (s časovou složitostí N^3) jistě napadne každého. Stačí brát postupně všechny dvojice čísel v posloupnosti a pro každou takovou dvojici zjišťovat, jak dlouhé shodné úseky těmito zvolenými čísly začínají. Ze všech takto nalezených délek se vezme maximum - a to je hledané číslo K.

Předvedeme zde jiný algoritmus, jehož zápis a popis je o trochu složitější. Z hlediska efektivity je to ale lepší algoritmus, má časovou složitost úměrnou N^2. Základní myšlenka algoritmu je následující. Předpokládejme, že zadanou posloupnost čísel máme uloženou v poli $A[1..N]$. Budeme

postupně měnit vzdálenost (tj. rozdíl indexů v poli A) mezi porovnávanými prvky, a to v cyklu od 1 do $N - 1$. Pro každou takovou pevnou vzdálenost projdeme celou posloupnost pomocí dvou souběžně zvětšovaných indexů. Při tomto průchodu stále počítáme délky souvislých shodných úseků. Maximum z takto získaných délek se ukládá do proměnné K a je výsledkem úlohy.

Algoritmus zapíšeme v programovacím jazyce Pascal. U obou příkazů while-cyklu jsou v komentářích uvedeny podmínky, kterými je možné výpočet ještě mírně zrychlit. Tyto podmínky využívají již získanou hodnotu K.

```
begin
K := 0; VZDAL := 1;
while VZDAL < N do {zde je možné drobné vylepšení:
                    VZDAL < N - K}
  begin
  I := 1; POC := 0;
  while I <= N - VZDAL do {zde je možné drobné
                           vylepšení: I <= N -
                           VZDAL - K + POC}
    begin
    if A[I] = A[I + VZDAL] then
      begin
      POC := POC + 1; {zde vždy platí: úseky délky
                       POC začínající prvky A[I]
                       a A[I + VZDAL] jsou
                       shodné}
      if POC > K then K := POC
      end
```

```
    else POC := 0;
    I := I + 1
    end;
  VZDAL := VZDAL + 1
  end
end
```

Popsaný algoritmus jistě skončí výpočet po konečně mnoha krocích. V těle vnitřního cyklu je zvyšována řídicí proměnná *I* a bez ohledu na průběh výpočtu se vzhledem k podmínce v příkazu while bude tento cyklus opakovat nejvýše *N*-krát. Obdobně ve vnějším cyklu se zvyšuje hodnota proměnné *VZDAL*, tělo vnějšího cyklu se bude provádět méně než *N*-krát. Počet průchodů jednotlivými příkazy programu je tedy předem omezen.

Hodnota proměnné *K* je měněna vždy v okamžiku, když algoritmus najde dvojí výskyt souvislého úseku délky větší než *K*. Na závěr výpočtu tedy v *K* bude skutečně délka nejdelšího souvislého úseku, který byl nalezen s dvojím výskytem v dané posloupnosti. Zbývá ukázat, že algoritmus při svém výpočtu skutečně najde dvojí výskyt podposloupnosti maximální délky, jaká se v dané posloupnosti nachází. To ale je patrné přímo z popisu algoritmu. Oba tyto výskyty musí být vůči sobě posunuty o jistý počet prvků a proměnná *VZDAL* této hodnoty při výpočtu jednou nabude (nabývá všech hodnot od 1 do $N - 1$). Pro tuto hodnotu *VZDAL* se pak ve vnitřním cyklu shodné úseky maximální délky jistě naleznou (prochází se v něm celá posloupnost).

P - II - 2

Nalezněte a dokažte (co nejlepší) algoritmus, který pro libovolnou konečnou množinu uzavřených intervalů na reálné přímce nalezne číslo, které patří do maximálního počtu zadaných intervalů, a určí tento počet. Intervaly jsou zadány výčtem svých dolních a horních mezí.

Řešení. Stejně jako u ostatních úloh na návrh algoritmu existuje celá řada různých řešení. Nejefektivnější z nich jsou založeny, podobně jako v úloze P - I - 2, na vhodném setřídění vstupních dat a mají proto časovou složitost úměrnou $N.\log_2 N$ (= složitost nejlepších třídicích algoritmů). Je možné navrhnout jednoduché algoritmy bez třídění, ovšem s časovou složitostí přinejlepším úměrnou N^2. Stačí například brát postupně horní (nebo dolní) meze všech intervalů, každé z těchto N čísel testovat do kolika ze zadaných intervalů patří, a z takto získaných hodnot vzít maximum. My si zde ukážeme rychlejší algoritmus.

Algoritmus řešení této úlohy zapíšeme slovně:

1. Setřiď vzestupně všech $2N$ čísel ze vstupu, dolní i horní meze intervalů dohromady (kde N je počet zadaných intervalů). Přitom si u každého čísla pamatuj, zda jde o horní nebo o dolní mez nějakého intervalu. Není třeba pamatovat si, která horní mez patří ke které dolní mezi intervalu. Opakuje-li se nějaká hodnota mezi zadanými $2N$ čísly vícekrát, budou v setříděné posloupnosti dolní meze umístěny před horními.
2. Setříděnou posloupnost $2N$ čísel postupně procházej od nejmenších k největším číslům. Přitom v proměnné *POCET* eviduj počet právě otevřených intervalů. Maxi-

mální dosaženou hodnotu proměnné *POCET* udržuj v proměnné *MAX*. Vždy současně se zvětšením hodnoty proměnné *MAX* aktualizuj hodnotu proměnné *CISLO*, ve které se udržuje jedno z čísel patřících do maximálního počtu intervalů.

3. Po ukončení výpočtu jsou požadované výsledné hodnoty v proměnných *CISLO* a *MAX*.

Bod 1 představuje některý ze standardních třídicích algoritmů, bod 3 je triviální. Rozepíšeme proto podrobněji již jen bod 2. Budeme dále předpokládat, že hodnoty mezí jsou po setřídění uloženy v poli $M[1..2N]$.

2.1. Polož $POCET = 0$, $MAX = 0$, $I = 1$.
2.2. Jestliže $M[I]$ je dolní mez, polož $POCET = POCET + 1$ a jdi na 2.6.
2.3. Jestliže $POCET \leqq MAX$, jdi na 2.5.
2.4. Polož $MAX = POCET$, $CISLO = M[I]$
2.5. Polož $POCET = POCET - 1$.
2.6. Polož $I = I + 1$.
2.7. Jestliže $I \leqq 2N$, jdi na 2.2.

Algoritmus skončí výpočet po konečně mnoha krocích, neboť intervalů je konečně mnoho, třídicí algoritmus je konečný a dále se už jen sekvenčně procházejí všechny meze všech intervalů. Průchod setříděnou posloupností $2N$ čísel v bodu 2 popsaného algoritmu vlastně představuje projití celé číselné osy s vyznačenými intervaly. V proměnné *POCET* je přitom stále uložen údaj, do kolika ze zadaných intervalů patří ten bod číselné osy, ve kterém se právě nacházíme. Ke změnám tohoto údaje může dojít jedině v počátečních a koncových bodech intervalů, stačí proto průchod provádět pouze

přes tyto body. Proměnná *MAX* nabývá největší hodnoty, jakou proměnná *POCET* při průchodu získala (*MAX* se testuje a případně zvyšuje pokaždé, kdy se má snížit hodnota proměnné *POCET*). Zároveň se zvýšením hodnoty proměnné *MAX* se dosazuje nová hodnota do proměnné *CISLO*, takže po ukončení výpočtu obsahuje dvojice proměnných *CISLO* a *MAX* správné výsledky podle zadání úlohy.

Je třeba zmínit se ještě o situaci, že dolní mez nějakého intervalu I_1 je rovna horní mezi jiného intervalu I_2. Podle zadání se jedná o uzavřené intervaly, takže I_1 a I_2 mají společný bod a při průchodu přes tento jejich společný bod je třeba zvýšit hodnotu proměnné *POCET* (a hned ji zase snížit, ale mezitím to může ovlivnit hodnotu proměnných *MAX* a *CISLO*). Toto je zajištěno uspořádáním zadaných $2N$ čísel podle bodu 1 algoritmu - dolní meze jsou umístěny před horními mezemi téže hodnoty.

P - II - 3

Je dán následující úsek programu se vstupními celočíselnými proměnnými *X* a *Y*. Zjistěte, jaká hodnota bude po provedení algoritmu v proměnné *Z*, a svoje tvrzení dokažte.

PASCAL:
```
Z := 1;
while Y > 0 do
  begin
    if Y mod 2 = 1 then Z := Z*X;
    Y := Y div 2;
    X := X*X
  end
```

```
BASIC:  10 LET Z = 1
        20 IF Y < = 0 THEN 70
        30 IF Y MOD 2 = 1 THEN LET Z = Z * X
        40 LET Y = Y DIV 2
        50 LET X = X * X
        60 GO TO 20
        70 REM KONEC
```

Operace A DIV B znamená celočíselný podíl čísel A a B, operace A MOD B znamená zbytek po celočíselném dělení. Tedy platí A = (A DIV B)*B + (A MOD B)

Řešení. Pokud bude mít proměnná Y hodnotu zápornou nebo nulovou, z celého popsaného algoritmu se provede pouze dosazovací příkaz $Z := 1$ a po ukončení algoritmu bude proto $Z = 1$. Ukážeme, že pro $Y > 0$ nabude proměnná Z hodnoty X^Y.

Počáteční hodnotu proměnné Y (označme ji y) si můžeme představit zapsanou ve dvojkové soustavě:

$$y = \sum_{i=0}^{n} a_i . 2^i, \quad a_i = 0 \text{ nebo } 1 \text{ pro } i = 0, 1, \ldots, n-1,$$
$$a_n = 1.$$

K násobení $Z := Z*X$ dochází při j-tém průchodu cyklem právě tehdy, když ve dvojkovém zápisu čísla y je v řádu $j - 1$ cifra 1 (tj. když $a_{j-1} = 1$). Test Y mod 2 totiž zjišťuje hodnotu poslední cifry dvojkového zápisu momentální hodnoty proměnné Y a příkaz $Y := Y$ div 2 mění hodnotu proměnné Y tak, že odebírá z jejího dvojkového zápisu poslední cifru.

Při každém průchodu cyklem se provede příkaz $X := X^*X$. Označíme-li počáteční hodnotu proměnné X jako x, bude v proměnné X na začátku j-tého průchodu cyklem hodnota $x^{2^{j-1}}$. Dohromady to znamená, že proměnná Z s počáteční hodnotou 1 je přinásobena číslem x^{2^j} právě tehdy, když $a_j = 1$ pro $j = 0, \ldots, n$. Pro hodnotu proměnné Z po ukončení výpočtu proto platí:

$$Z = \prod_{j=0}^{n} a_j . x^{2^j}$$

Tento výraz můžeme dále upravit (připomínáme, že koeficienty a_j, $j = 0, \ldots, n$ jsou cifry dvojkové soustavy, tedy čísla 0 nebo 1):

$$Z = x^{\sum_{j=0}^{n} a_j 2^j}$$

V exponentu jsme získali výraz $\sum_{j=0}^{n} a_j . 2^j$, který je přímo roven vyjádření počáteční hodnoty y proměnné Y. Platí tedy skutečně, že po ukončení výpočtu uvedeného algoritmu je

$$Z = x^y.$$

P - II - 4

Definici zjednodušeného jazyka LISP nyní ještě doplníme. Další elementární funkce slouží k provádění aritmetických výpočtů s celými čísly:

ADD[A;B] — součet A + B
SUB[A;B] — rozdíl A − B
MUL[A;B] — součin A.B
DIV[A;B] — podíl A div B
MOD[A;B] — zbytek po dělení, tj. A mod B
GT[A;B] — porovnání $A > B$
GE[A;B] — porovnání $A \geqq B$

Argumenty A, B všech těchto funkcí musí být číselné atomy. Funkční hodnotou je číselný atom, jehož hodnota odpovídá výsledku příslušné aritmetické operace, v případě funkcí GT, GE pak atom T nebo F podle platnosti porovnání.

Zadání úlohy

Zapište ve zjednodušeném jazyce LISP definice následujících funkcí:

a) AVER[S]

S je neprázdný seznam tvořený číselnými atomy, hodnotou funkce je jejich aritmetický průměr.

b) MAX[S]

S je neprázdný seznam tvořený číselnými atomy, hodnotou funkce je hodnota největšího z nich.

c) PREVOD[X;S]

X, S jsou číselné atomy s hodnotou $X \geqq 0$, $S > 1$; hodnotou funkce je seznam cifer, který reprezentuje zápis čísla X v číselné soustavě se základem S. Prvním prvkem seznamu je cifra nejvyššího řádu.

Př.: Pro X = 11, S = 2 je hodnotou funkce seznam (1 0 1 1).

Řešení. a) Funkci AVER[S] pro výpočet aritmetického průměru číselných atomů v seznamu S nejsnáze zapíšeme pomocí dvou pomocných funkcí:

DELKA[S] — kde S je jako v zadání úlohy, hodnotou funkce je počet prvků v seznamu S:

$$\text{DELKA}[S] = [\text{NULL}[S] \rightarrow 0;\ T \rightarrow \text{ADD}[1; \text{DELKA}[\text{CDR}[S]]]]$$

SOUCET[S] — kde S je jako v zadání úlohy, hodnotou funkce je číselný atom, jehož hodnota odpovídá součtu hodnot prvků v seznamu S:

$$\text{SOUCET}[S] = [\text{NULL}[S] \rightarrow 0;\ T \rightarrow \text{ADD}[\text{CAR}[S]; \text{SOUCET}[\text{CDR}[S]]]]$$

Potom stačí napsat

$$\text{AVER}[S] = \text{DIV}[\text{SUM}[S]; \text{DELKA}[S]].$$

Upozorňujeme, že do zjednodušeného jazyka LISP jsme zavedli práci pouze s celými čísly, takže funkce AVER[S] počítá vlastně celou část z aritmetického průměru.

b) $$\text{MAX}[S] = [\text{NULL}[\text{CDR}[S]] \rightarrow \text{CAR}[S];\ \text{GT}[\text{CAR}[S]; \text{CAR}[\text{CDR}[S]]] \rightarrow \rightarrow \text{MAX}[\text{CONS}[\text{CAR}[S]; \text{CDR}[\text{CDR}[S]]]];\ T \rightarrow \text{MAX}[\text{CDR}[S]]]$$

Maximální hodnotou jednoprvkového seznamu je hodnota jeho jediného prvku. Jinak se vynechá menší z prvních dvou prvků seznamu. Tím vznikne seznam o jeden prvek kratší

(se stejným maximem hodnot prvků, jaké měl původní seznam), na který se rekurzívně zavolá funkce MAX.

c) Nejprve definujeme pomocnou funkci APPEND[S;X], kde S je seznam. Výsledkem této funkce je seznam, který vznikne přidáním výrazu X do seznamu S na konec (tj. zařazením za poslední prvek seznamu S):

```
APPEND[S;X] = [NULL[S] → CONS[X;( )];
               T → CONS[CAR[S];
                 APPEND[CDR[S]; X]]]
```

Nyní již můžeme přistoupit k řešení zadané úlohy. Poslední číslicí zápisu čísla X v soustavě se základem S získáme jako zbytek po celočíselném dělení čísla X číslem S, tedy jako X mod S. Předcházející číslice jsou rovny cifrám zápisu čísla X div S v soustavě se základem S. Postup převádění čísla X do soustavy se základem S proto vypadá následovně: Postupně číslo X celočíselně dělíme hodnotou S a zaznamenáváme zbytky po těchto celočíselných děleních. Tyto zbytky jsou odzadu jednotlivými ciframi hledaného zápisu. Uvedený postup snadno zapíšeme v programovacím jazyce LISP:

```
PREVOD[X;S] = [GT[S;X] → CONS[X;( )];
               T → APPEND[PREVOD[DIV[X;S];S];
                 MOD[X;S]]]
```

Je-li $X < S$, je číslo X v číselné soustavě se základem S reprezentováno číslicí X. Jinak uplatňujeme výše uvedený postup výpočtu. Ke skládání jednotlivých číslic do výsledného seznamu se používá pomocná funkce APPEND. Zde není možné použít standardní funkci CONS, neboť cifry je třeba do seznamu připojovat na konec.

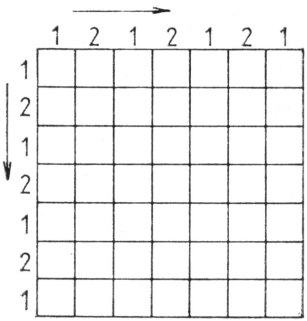
P - III - 1

Nalezněte a dokažte (co nejlepší) algoritmus, který pro libovolnou konečnou posloupnost čísel a čísla K, L určí, zda daná posloupnost obsahuje souvislý úsek délky K, který se v ní vyskytuje alespoň L-krát. Jednotlivé výskyty se mohou částečně překrývat.

Př.: Pro posloupnost 1, 2, 1, 2, 1, 2, 1 a $K = 3$, $L = 3$ je odpověď »ANO«, protože se v ní třikrát vyskytuje úsek 1, 2, 1.

Řešení. Označíme-li délku zadané posloupnosti čísel jako N, je možné úlohu vyřešit provedením řádově N^2 operací porovnání dvou čísel. Je zajímavé, že při soutěži na tento algoritmus nikdo nepřišel, všechna odevzdaná správná řešení obsahovala algoritmy pomalejší.

Nejprve vysvětlíme základní myšlenky algoritmu. Představme si čtvercovou tabulku o rozměrech $N \times N$, jejíž řádky i sloupce jsou označeny postupně jednotlivými čísly ze zadané posloupnosti (shora dolů a zleva doprava). Pro příklad ze zadání úlohy tedy bude situace vypadat podle obr. 38. Nyní tuto tabulku vyplníme čísly 0 a 1 podle jednoduchého pra-



Obr. 38

vidla. Číslo 1 bude v tabulce zapsáno v těch políčkách, která mají stejně označené řádky a sloupce. V ostatních políčkách tabulky bude nula. V našem příkladu ukazuje vyplnění tabulky obr. 39. V takto vyplněné tabulce budou jistě na hlavní diagonále samé jedničky. Dále se budeme zajímat pouze o polovinu tabulky nad hlavní diagonálou (polovina pod hlavní

	1	2	1	2	1	2	1
1	1	0	1	0	1	0	1
2	0	1	0	1	0	1	0
1	1	0	1	0	1	0	1
2	0	1	0	1	0	1	0
1	1	0	1	0	1	0	1
2	0	1	0	1	0	1	0
1	1	0	1	0	1	0	1

Obr. 39

diagonálou je s horní polovinou symetrická podle diagonály a nepřináší proto žádnou další informaci).

Pro řešení naší úlohy nás budou zajímat ty souvislé řady jedniček v tabulce, které mají směr rovnoběžný s hlavní diagonálou (klesají odshora dolů zleva doprava). Příklad takové řady jedniček ukazuje obr. 40. Pomocí těchto šikmých řad

	1	2	1	2	1	2	1
1	1	0	1	0	1	0	1
2		1	0	1	0	1	0
1			1	0	1	0	1
2				1	0	1	0
1					1	0	1
2						1	0
1							1

Obr. 40

jedniček v tabulce již můžeme snadno zformulovat, jak získáme výsledek řešené úlohy. Jedničkám v jednom řádku tabulky totiž odpovídají opakující se výskyty čísel v zadané posloupnosti, klesajícím souvislým šikmým řadám jedniček proto odpovídají opakující se úseky v zadané posloupnosti. Jestliže existuje takových K po sobě jdoucích řádků tabulky, že jimi prochází alespoň L souvislých šikmých řad jedniček, algoritmus má odpovědět »ANO«, jinak je výsledkem »NE«. V našem příkladu na obr. 40 pro zadané $K = 3$, $L = 3$ bude odpovědí skutečně »ANO«, prvními třemi řádky tabulky procházejí tři šikmé řady jedniček (jedna je na hlavní diagonále,

jedna je vyznačena, další leží uprostřed mezi prvními dvěma).

Další úpravy jsou již jen technického rázu, jejich cílem je usnadnit činnost algoritmu. Naši tabulku budeme vyplňovat po řádcích. Přitom můžeme do tabulky zapisovat nejen znaky 0 a 1, ale obecně celá nezáporná čísla, a tím můžeme (vždy s využitím obsahu předchozího řádku) zároveň počítat délky souvislých šikmých řad »jedniček«. Vyplněnou tabulku v našem příkladu po této úpravě ukazuje obr. 41.

	1	2	1	2	1	2	1
1	1	0	1	0	1	0	1
2		2	0	2	0	2	0
1			3	0	3	0	3
2				4	0	4	0
1					5	0	5
2						6	0
1							7

Obr. 41

Pravidlo pro vyplňování tabulky je tedy nyní následující. Vyplňujeme-li políčko $(1, j)$, $1 \leqq j \leqq N$, zapíšeme do něj 1, jestliže označení j-tého sloupce je stejné jako označení prvního řádku, jinak zapíšeme 0. Vyplňujeme-li políčko (i, j), $1 < i \leqq N$, $i \leqq j \leqq N$, zapíšeme do něj hodnotu políčka $(i - 1, j - 1)$ zvětšenou o 1, jestliže označení i-tého řádku a j-tého sloupce jsou stejné, jinak zapíšeme 0.

Určení výsledku je potom snadné. Algoritmus odpoví »ANO« právě tehdy, jestliže je v některém řádku tabulky

alespoň L čísel větších nebo rovných K. Jak je vidět, pro stanovení odpovědi není ani třeba procházet znovu celou tabulku, hodnoty větší nebo rovné K je možné na každém řádku počítat již při zaplňování tabulky (a v případě nalezení odpovědi »ANO« ani není třeba vyplňovat zbytek tabulky). Důsledkem těchto úvah je ještě jedno vylepšení algoritmu. Jestliže už nemusíme zaplněnou tabulku znovu procházet, není ani nutné celou ji ukládat. Vždy stačí pamatovat si pouze obsah předchozího řádku. Nebudeme proto vytvářet čtvercovou tabulku s velkým paměťovým nárokem N^2, stačí pracovat pouze s jedním vektorem délky N, do kterého budeme postupně počítat a ukládat jednotlivé řádky naší tabulky.

Popsaný algoritmus řešení zadané úlohy nyní ještě zapíšeme v programovacím jazyce Pascal. Budeme předpokládat, že danou posloupnost čísel délky N máme uloženou v poli $A[1..N]$ a že proměnné K, L obsahují vstupní údaje podle zadání úlohy. Jednotlivé řádky tabulky se budou vytvářet v poli $T[1..N]$. Tyto řádky jsou počítány a ukládány do T odzadu (zprava doleva) proto, abychom si hodnotami nově vytvářeného řádku nepřepsali ty hodnoty předchozího řádku, které ještě budeme potřebovat. Zároveň je v proměnné *POCET* sledován počet čísel větších nebo rovných K na řádku. Najde-li se jich L, výpočet ihned skončí. Proměnné I a J jsou pomocné, určují vždy řádkový a sloupcový index právě počítaného políčka v tabulce.

```
begin
I := 1; J := N; POCET := 0;
while (I < = N) and (POCET < L) do
  begin
```

```
    POCET := 0;
    while (J >= I) and (POCET < L) do
      begin
      if A[I]<>A[J] then T[J] := 0
      else if I = 1 then T[J] := 1
      else T[J] := T[J - 1] + 1;
      if T[J] >= K then POCET := POCET + 1;
      J := J - 1
      end;
    I := I + 1; J := N
    end;
if POCET >= L then write ('ANO') else write ('NE')
end
```

Výpočet podle algoritmu jistě skončí, neboť postupně se zaplňuje a testuje méně než N^2 políček tabulky. Každý z cyklů v uvedeném programu se bude opakovat nejvýše N-krát. Zbývá ukázat, že popsaný algoritmus skutečně najde L výskytů podposloupnosti délky K v zadané posloupnosti. Nechť tyto shodné podposloupnosti mají následující indexy v poli A:

první: $p_1, p_1 + 1, \ldots, p_1 + K - 1$
druhá: $p_2, p_2 + 1, \ldots, p_2 + K - 1$
...
L-tá: $p_L, p_L + 1, \ldots, p_L + K - 1$

přičemž všechny tyto indexy mají hodnoty mezi 1 a N (včetně) a platí $p_1 < p_2 < \ldots < p_L$ (podposloupnosti jsou vypsány v pořadí jejich výskytu).

Při vyplňování naší tabulky dojdeme k řádku p_1 (v programu $I = p_1$). Na tomto řádku se jistě objeví kladná čísla ve sloupcích $p_1, p_2, \ldots, p_L$, neboť je $A[p_1] = A[p_2] = \ldots$

$\ldots = A[p_L]$ (v programu to znamená, že hodnoty $T[p_1]$, $T[p_2]$, $\ldots$, $T[p_L]$ budou kladné). V následujícím řádku $p_1 + 1$ (v programu: pro $I = p_1 + 1$) nastane shoda $A[p_1 + 1] = A[p_2 + 1] = \ldots = A[p_L + 1]$, čísla ve sloupcích $p_1 + 1$, $p_2 + 1$, $\ldots$, $p_L + 1$ proto budou kladná a vzhledem k obsazení předchozího p_1-tého řádku budou větší nebo rovna 2 (v programu: hodnoty $T[p_1 + 1]$, $T[p_2 + 1]$, $\ldots$, $T[p_L + 1]$ budou $\geqq 2$). Tato situace se opakuje i pro další řádky tabulky až do řádku $p_1 + K - 1$ včetně. V řádku číslo $p_1 + K - 1$ budou ve sloupcích $p_1 + K - 1$, $p_2 + K - 1$, $\ldots$, $p_L + K - 1$ zapsána čísla všechna větší nebo rovna hodnotě K (v programu: $T[p_1 + K - 1]$, $T[p_2 + K - 1]$, $\ldots$, $T[p_L + K - 1]$ budou mít po vyhodnocení řádku $I = p_1 + K - 1$ hodnotu větší nebo rovnu K). V řádku $p_1 + K - 1$ tedy bude nalezeno L čísel větších nebo rovných K a algoritmus proto správně odpoví »ANO«. Stejným způsobem se ukáže, že neexistuje-li v zadané posloupnosti L výskytů podposloupnosti délky K, algoritmus nikdy v jednom řádku nenapočítá L čísel větších nebo rovných K a odpoví proto »NE«.

P - III - 2

Je dán následující úsek programu se vstupními reálnými proměnnými X a A, kde $X > 1$, $A > 1$.

```
PASCAL: Z := A;
        K := 1;
        Y := 0;
        while X > = Z do
          begin
          Z := Z*Z;
```

```
    K := 2*K
    end;
  while K > 1 do
    begin
    {X*A ↑ Y = konst. & X < Z}
    Z := SQRT(Z);
    K := K div 2;
    if X >= Z then
      begin
      X := X/Z;
      Y := Y + K
      end
    end
```

BASIC:

```
 10 LET Z = A
 20 LET K = 1
 30 LET Y = 0
 40 IF X < Z THEN 80
 50 LET Z = Z * Z
 60 LET K = 2 * K
 70 GOTO 40
 80 IF K <= 1 THEN 160
 90 REM X * A ↑ Y = KONST. & X < Z
100 LET Z = SQR (Z)
110 LET K = K / 2
120 IF X < Z THEN 150
130 LET X = X / Z
140 LET Y = Y + K
150 GOTO 80
160 REM KONEC
```

Dokažte, že výraz $X^*A \uparrow Y$ má stejnou hodnotu při všech průchodech výpočtu místem programu, kde je v komentáři zapsán, a že v tomto místě je zároveň vždy splněna podmínka $X < Z$. Na základě toho zjistěte, jaké hodnoty nabývá proměnná Y po ukončení výpočtu, a svoje tvrzení dokažte.

Řešení (upravené řešení Radka Porazila z gymnázia v Bílovci).

Nejprve ukážeme, že po úvodní inicializaci na řádku 10 a 20, po změnách na řádku 50 a 60 i na řádku 100 a 110 stále platí vztah

$$Z = A^K, \tag{1}$$

kde symboly Z, A, K označují po řadě momentální hodnoty proměnných Z, A, K. Obsah proměnné A se v programu nemění. Uvedený vztah platí vždy po změně obou proměnných Z, K (provedení dvou po sobě jdoucích přiřazovacích příkazů).

Dokážeme platnost vztahu $Z = A^K$ pro změny na řádcích 40 a 50. K tomu označme K_0, Z_0 hodnoty proměnných K, Z před prvním testováním podmínky na řádku 40. Dále označme K_n, Z_n hodnoty proměnných K, Z po n-tém průchodu řádkem 60. Dokazujeme matematickou indukcí podle n:

1. Pro výchozí hodnoty K_0, Z_0, A vztah $Z_0 = A^{K_0}$ zřejmě platí, neboť podle dosazení na řádku 10 a 20 je $Z_0 = A$, $K_0 = 1$.
2. Nechť platí $Z_n = A^{K_n}$ pro $n \geqq 0$. Chceme dokázat, že potom $Z_{n+1} = A^{K_{n+1}}$. Podle řádku 50 je $Z_{n+1} = Z_n^2$, podle řádku 60 platí $K_{n+1} = 2.K_n$. Odtud plyne:

$$Z_{n+1} = Z_n^2 = (A^{K_n})^2 = A^{2 \cdot K_n} = A^{K_{n+1}}$$

Tím je tvrzení dokázáno.

Obdobně bude vypadat důkaz vztahu $Z = A^K$ pro změny na řádku 100 a 110. Nyní bude značit K_0, Z_0 hodnoty proměnných K, Z před prvním testováním podmínky na řádku 80 a K_n, Z_n hodnoty proměnných K, Z po n-tém průchodu řádku 110. Opět dokazujeme matematickou indukcí:

1. Platnost vztahu pro K_0, Z_0 plyne z předchozího důkazu, neboť K_0, Z_0 jsou poslední hodnoty proměnných K, Z, které vznikly při změnách na řádku 40 a 50.
2. Indukční předpoklad: $Z_n = A^{K_n}$ pro nějaké $n \geqq 0$. Máme dokázat, že $Z_{n+1} = A^{K_{n+1}}$. Podle řádku 100 je $Z_{n+1} = Z_n^{1/2}$, podle řádku 110 je $K_{n+1} = K_n/2$. Platí:

$$Z_{n+1} = Z_n^{1/2} = (A^{K_n})^{1/2} = A^{K_n/2} = A^{K_n+1},$$

což jsme měli dokázat.

Nyní můžeme přikročit k důkazu invariantu uvedeného v zadání úlohy. Nejprve ukážeme, že $X.A^Y = \text{konst.}$ Počáteční hodnoty proměnných X, Y označíme po řadě X_0, Y_0, platí $Y_0 = 0$ (podle dosazení na ř. 30). Při prvním vyhodnocení invariantu (tj. prvním průchodu řádkem 90) mají proměnné X, Y hodnoty X_0, Y_0. Výraz $X.A^Y$ nabývá proto hodnoty $X_0.A^{Y_0} = X_0$ (neboť $Y_0 = 0$). Budeme proto dokazovat platnost tvrzení

$$X_n.A^{Y_n} = X_0, \tag{2}$$

kde X_n, Y_n označují hodnoty proměnných X, Y po n-tém průchodu tělem cyklu (řádky 100 až 150). Tvrzení dokážeme matematickou indukcí:

1. Pro X_0, Y_0 jsme platnost vztahu (2) dokázali již při jeho odvození.
2. Nechť platí $X_n . A^{Y_n} = X_0$ pro jisté $n \geqq 0$. Dokážeme, že také $X_{n+1} . A^{Y_{n+1}} = X_0$. Při $(n + 1)$-ním průchodu tělem cyklu na řádku 100 až 150 mohou nastat dvě možnosti:
   a) platí $X_n < Z'$ (kde Z' je momentální hodnota proměnné Z) ... pak se hodnoty proměnných X, Y nezmění, bude $X_{n+1} = X_n$, $Y_{n+1} = Y_n$ a platnost dokazovaného vztahu plyne přímo z indukčního předpokladu.
   b) platí $X_n \geq Z'$ ... potom podle řádku 130 bude $X_{n+1} = X_n / Z'$ a podle řádku 140 platí $Y_{n+1} = Y_n + K'$, kde K', Z' jsou hodnoty proměnných K, Z v okamžiku průchodu řádky 130 a 140. Podle (1) platí rovnost $Z' = A^{K'}$ Nyní bude s využitím uvedených rovností:

$$X_{n+1} . A^{Y_{n+1}} = (X_n / Z') . A^{Y_n + K'} = (X_n / A^{K'}) . A^{Y_n} . A^{K'} =$$

$$= X_n . A^{Y_n} = X_0$$

Tím je vztah (2) dokázán.

Dále dokážeme druhou část invariantu, totiž to, že při každém průchodu řádkem 90 platí

$$X < Z. \tag{3}$$

Symboly X_0, Z_0 budou značit hodnoty proměnných X, Z při prvním průchodu řádkem 90, symboly X_n, Z_n hodnoty X, Z

při $(n + 1)$-ním průchodu. Dokazujeme opět matematickou indukcí:

1. Jistě platí $X_0 < Z_0$, neboť na řádek 80 se výpočet dostane až po splnění této podmínky (test na řádku 40).
2. Nechť platí $X_n < Z_n$ pro jisté $n \geqq 0$. Dokážeme, že $X_{n+1} < Z_{n+1}$. Na řádku 100 se v těle cyklu nejprve určí nová hodnota proměnné Z: $Z_{n+1} = Z_n^{1/2}$. Dále mohou nastat dvě možnosti jako v předchozím důkazu:
   a) $X_n < Z_{n+1}$ ... potom bude $X_{n+1} = X_n$, a tedy jistě také $X_{n+1} < Z_{n+1}$.
   b) $X_n \geqq Z_{n+1}$ ... potom $X_{n+1} = X_n/Z_{n+1}$ (podle řádku 130). Můžeme psát:

$$X_{n+1} = X_n/Z_{n+1} = X_n/Z_n^{1/2} = X_n . Z_n^{1/2}/Z_n =$$
$$= X_n . Z_{n+1}/Z_n = (X_n/Z_n) . Z_{n+1} < Z_{n+1},$$

   což jsme měli dokázat.

Poznámka. poslední nerovnost plyne z indukčního předpokladu.

Tím je ukončena první část úlohy - důkaz platnosti invariantu. Na základě dokázaného invariantu můžeme vyslovit následující hypotézu: V proměnné Y bude po ukončení výpočtu hodnota celé části logaritmu X při základu A. Tuto hypotézu nyní dokážeme.

Při posledním průchodu řádkem 90 platí $K_{n-1} = 2$, $Z_{n-1} = A^2$ a $X_{n-1} . A^{Y_{n-1}} = X_0$, přičemž $X_{n-1} < A^2$ (vyplývá z (2) a (3)), kde X_{n-1}, Y_{n-1}, K_{n-1} Z_{n-1} jsou hodnoty proměnných X, Y, K, Z při posledním průchodu řádkem 90. Těsně před provedením řádku 120 je již $K = 1$, $Z = A$.

a) Je-li $X_{n-1} < A$, jistě také platí $\log_A X_{n-1} < 1$. V tomto případě bude $X_n = X_{n-1}$, $Y_n = Y_{n-1}$ (na základě testu na řádku 120). Platí $X_n . A^{Y_n} = X_0$, odtud po zlogaritmování dostáváme

$$\log_A X_n + Y_n = \log_A X_0.$$

Jedná se o poslední průchod tělem cyklu, takže Y_n je výsledná hodnota proměnné Y. Hodnota proměnné Y se v programu vytváří jako součet celých čísel, Y_n je tedy také celé číslo. Protože $\log_A X_n$ je nezáporné číslo menší než 1, je jistě Y_n celou částí hodnoty $\log_A X_0$, což jsme měli dokázat.

b) Jestliže $X_{n-1} \geqq A$ (a z invariantu $X_{n-1} < A^2$), dostaneme po zlogaritmování nerovnosti

$$\log_A A \leqq \log_A X_{n-1} < \log_A A^2$$

$$1 \leqq \log_A X_{n-1} < 2$$

Můžeme tedy $\log_A X_{n-1}$ zapsat ve tvaru $1 + v$, kde $0 \leqq v < 1$. Dále podle řádku 140 je $Y_n = Y_{n-1} + 1$ a zlogaritmováním vztahu $X_{n-1} . A^{Y_{n-1}} = X_0$ dostaneme

$$\log_A X_{n-1} + Y_{n-1} = \log_A X_0.$$

Dosadíme-li do této rovnosti za $\log_A X_{n-1}$ výraz $1 + v$ a za Y_{n-1} výraz $Y_n - 1$, dostaneme

$$Y_n + v = \log_A X_0.$$

Protože Y_n je celé číslo a hodnota v je nezáporná menší než 1, je hodnotou Y_n celá část z $\log_A X_0$.

Tím je hypotéza o výsledné hodnotě proměnné Y dokázána.

P - III - 3

Nalezněte a dokažte (co nejlepší) algoritmus, který z libovolné konečné množiny uzavřených intervalů na reálné přímce vybere skupinu po dvou disjunktních invervalů, která ze všech takových skupin obsahuje největší možný počet intervalů.

Řešení (upravené řešení Pavla Kozlovského z gymnázia v Jindřichově Hradci).

Dolní a horní meze intervalů uložíme do pole $M[1..N, 1..2]$, kde N je počet intervalů a druhý index určuje druh meze invervalu (1 - dolní, 2 - horní). Potom intervaly zapsané v poli M setřídíme podle hodnot dolní meze vzestupně, tzn. bude platit $M[1,1] \leqq M[2, 1] \leqq \ldots \leqq M[N, 1]$. Další částí algoritmu je jednoprůchodové prohledání pole M. Zvlášť jsou přitom vždy uschovány hodnoty mezí »posledního« zpracovaného intervalu (na počátku výpočtu to jsou meze prvního intervalu, tj. $M[1, 1]$ a $M[1, 2]$). Při zpracování dalšího intervalu v pořadí může nastat několik možností:

1. Zpracovávaný interval je obsažen v posledním intervalu. Do výsledné skupiny bude výhodnější vybrat menší z nich (má méně společných bodů s dalšími intervaly). Poslední interval proto zapomeneme a zpracovávaný interval prohlásíme za poslední.
2. Zpracovávaný a poslední interval jsou disjunktní. Protože

pole je setříděné podle dolních mezí, jsou všechny dosud nezpracované intervaly disjunktní s posledním intervalem. Poslední interval zapíšeme do výstupu a zpracovávaný interval prohlásíme za poslední.

3. Zpracovávaný a poslední interval mají neprázdný průnik, ale zároveň neplatí inkluze z bodu 1. Do výsledné skupiny nelze vybrat oba intervaly. Vybereme proto ten, který má nižší hodnotu horní meze (vzhledem k umístění dosud nezpracovaných intervalů je to výhodnější). Tím je jistě poslední interval, jinak by nastal případ 1. Poslední interval tedy zůstává beze změn, zpracovávaný interval zapomeneme.

Po zpracování všech intervalů se ještě poslední interval zapíše do výstupu.

Rychlost výpočtu tohoto algoritmu je dána rychlostí třídění a je tedy řádově $N.\log_2 N$, další zpracování intervalů má již lineární složitost.

Algoritmus nyní zapíšeme ve tvaru úseku programu v programovacím jazyce Pascal. Předpokládejme, že N udává počet intervalů na vstupu a že máme deklarováno dvourozměrné pole $M[1..N, 1..2]$ tak, jak je uvedeno v úvodu řešení. Dále budeme předpokládat, že v poli M jsou uloženy hodnoty mezí intervalů a že tyto intervaly jsou již setříděné vzestupně podle dolních mezí. K tomuto setřídění by se použil některý ze standardních třídicích algoritmů (zde neuvádíme). Platí tedy $M[1, 1] \leqq M[2, 1] \leqq \ldots \leqq M[N, 1]$ a $M[K, 1]$, $M[K, 2]$ jsou dolní a horní mez K-tého intervalu pro celá K, $1 \leqq K \leqq N$. Pomocné proměnné L, H slouží k uložení dolní a horní meze »posledního« intervalu.

```
L := M[1, 1];
H := M[1, 2];
for I := 2 to N do
  if M[I, 2] < H then
    begin {případ 1}
    L := M[I, 1];
    H := M[I, 2];
    end
  else if M[I, 1] > H then
    begin {případ 2}
    writeln (L, H);
    L := M[I, 1];
    H := M[I, 2];
    end; {v případě 3 se nic neprovádí}
writeln (L, H);
```

Zdůvodnění správnosti algoritmu:

1. Výpočet programu skončí, neboť počet provádění jednotlivých operací je předem omezen hodnotou N (tělo cyklu se opakuje méně než N-krát).

2. Vybraná skupina obsahuje pouze po dvou disjunktní intervaly. Interval poznamenaný jako »poslední« (v proměnných L, H) je totiž zapsán na výstup právě tehdy, je-li disjunktní s právě zpracovávaným intervalem, a vzhledem k uspořádání intervalů podle hodnot dolní meze je proto disjunktní i se všemi dalšími dosud nezpracovanými intervaly. Dále platí, že interval je prohlášen za »poslední«, buď je-li disjunktní s intervalem právě zapisovaným na výstup, nebo je-li podmnožinou dosud posledního intervalu. Protože je pole intervalů setříděno, je poslední interval disjunktní i se všemi již vybranými a vytištěnými intervaly.

3. Vybraná skupina intervalů obsahuje největší možný počet intervalů. Při průchodu všemi setříděnými intervaly totiž interval do výsledné skupiny nevybereme a »zapomeneme« ho pouze ve dvou případech, kdy je to nezbytné, aby vybraná skupina obsahovala pouze po dvou disjunktní intervaly:

a) Je-li některý z intervalů částí jiného, je nutné jeden z nich vynechat. Algoritmus vynechá větší z obou intervalů, což je jistě výhodnější z hlediska možných společných bodů s dalšími intervaly.

b) Mají-li dva intervaly neprázdný průnik, ale nenastává přitom případ a), musí být jeden z nich vynechán. Algoritmus vynechá ten z obou intervalů, který má větší horní (a tudíž i dolní) mez, neboť to je výhodnější z důvodu případných společných bodů s následujícími intervaly (konflikty s předchozími intervaly jsou již vyřešeny, neboť intervaly jsou při procházení setříděné).

P - III - 4

Zapište ve zjednodušeném jazyce LISP definice následujících funkcí:

a) REVERSE [X]

X musí být seznam

hodnotou funkce je seznam tvořený stejnými prvky jako seznam X, ale v opačném pořadí.

b) DIFELEM [S]

S je seznam atomů

hodnotou funkce je seznam všech různých prvků seznamu S.

c) LINEAR [S]

S musí být seznam

hodnotou funkce je seznam všech atomů, které se vyskytují někde v S bez ohledu na úroveň vnoření seznamů a jejich pořadí.

Řešení. a) Nadefinujeme pomocnou funkci PRESUN [X; Y], kde X, Y jsou seznamy. Tato funkce přesune všechny prvky ze seznamu X v opačném pořadí na začátek seznamu Y:

$$\begin{aligned}\text{PRESUN}\,[X;\,Y] = [&\text{NULL}\,[X] \to Y;\\ &T \to \text{PRESUN}\,[\text{CDR}[X];\,\text{CONS}\\ &\qquad[\text{CAR}[X];\,Y]]]\end{aligned}$$

S využitím této funkce již snadno zapíšeme požadovanou funkci REVERSE [X]:

$$\text{REVERSE}\,[X] = \text{PRESUN}\,[X;\,()]$$

b) Pomocná funkce MEMBER [X; S] zjišťuje, zda je atom X prvkem seznamu atomů S (viz úloha P - I - 4):

$$\begin{aligned}\text{MEMBER}\,[X;\,S] = [&\text{NULL}\,[S] \to F;\\ &\text{EQ}\,[X;\,\text{CAR}[S]] \to T;\\ &T \to \text{MEMBER}\,[X;\,\text{CDR}[S]]]\end{aligned}$$

Dále definujeme pomocnou funkci DIFEL [S; Y], kde S, Y jsou seznamy atomů. Tato funkce zpracovává seznam atomů S a vytváří pomocný seznam atomů Y následujícím způsobem: do seznamu Y přesouvá ty prvky ze seznamu S, které ještě v Y nejsou. Jestliže se projde celý seznam S, vydá se vytvořený seznam Y jako výsledek funkce DIFEL:

```
DIFEL [S; Y] = [NULL [S] → Y;
                MEMBER [CAR[S]; Y] →
                → DIFEL [CDR[S]; Y];
                T → DIFEL [CDR[S];
                CONS [CAR[S]; Y]]]
```

Výslednou funkci DIFELEM[S] požadovaných vlastností nyní snadno vyjádříme jako volání funkce DIFEL s prázdným seznamem dosazeným za parametr Y:

$$\text{DIFELEM}[S] = \text{DIFEL}[S; ()]$$

Pro zajímavost uvádíme již bez podrobnějšího komentáře ještě jiné řešení úlohy, které nepoužívá pomocnou funkci DIFEL a nevyužívá triku s pomocným, na počátku prázdným seznamem. Zápis tohoto druhého řešení je kratší, ale výpočet by byl o něco pomalejší:

```
DIFELEM [S] = [NULL [S] → S;
               MEMBER [CAR[R];
               CDR[S]] → DIFELEM [CDR[S]];
               T → CONS [CAR[S];
               DIFELEM [CDR[S]]]]
```

c) Nejprve nadefinujeme pomocnou funkci CONNECT [R; S], jejíž hodnotou je seznam vzniklý spojením seznamů R a S za sebe:

```
CONNECT [R; S] = [NULL [R] → S;
                  T → CONS [CAR[R];
                  CONNECT [CDR [R] ; S]]]
```

S použitím této pomocné funkce pak již zapíšeme výslednou funkci LINEAR [S]. Postupně procházíme všechny prvky seznamu S. Je-li takový prvek seznamu S atom, je připojen přímo do vytvářeného seznamu atomů a funkce LINEAR je rekurzívně volána na zpracování zbytku seznamu S bez prvního prvku. Je-li prvním prvkem seznamu S seznam, je funkce LINEAR volána dvakrát - na první prvek seznamu S i na zbytek seznamu S - a oba takto vzniklé seznamy atomů jsou propojeny pomocí funkce CONNECT:

```
LINEAR [S] = [NULL [S] → S;
              ATOM[CAR[S]] → CONS[CAR[S];
              LINEAR[CDR[S]]];
              T → CONNECT [LINEAR[CAR[S]];
              LINEAR[CDR[S]]]]
```

Úlohu je možné řešit i mnoha jinými způsoby. Pro ilustraci uvádíme, již bez komentářů, ještě jedno řešení:

```
LINE [S; Y] = [NULL [S] → Y;
               ATOM[CAR[S]] → LINE[CDR[S];
               CONS[CAR[S]; Y]];
               T → LINE[CDR[S];
               LINE[CAR[S]; Y]]]
LINEAR [S] = LINE [S; ()]
```

MEZINÁRODNÍ OLYMPIÁDA V PROGRAMOVÁNÍ

Ve školním roce 1986/87 se uskutečnil také 1. ročník mezinárodní olympiády v programování. Tato soutěž vznikla z iniciativy slovenských organizátorů kategorie P matematické olympiády. Mezinárodní olympiády v programování se zúčastnila družstva ze SSSR (8 účastníků), z BLR (6 účastníků) a z ČSSR (7 účastníků), přitomni dále byli pozorovatelé z NDR, MLR a z PLR. Olympiáda probíhala v Bratislavě a v Modre ve dnech 23.—30. 8. 1987. Vlastní soutěž se konala po dva dny, v každém z těchto dnů měli soutěžící čtyři hodiny času na vyřešení dvou soutěžních úloh. Výběr i opravování úloh probíhaly pod vedením mezinárodní jury stejným způsobem, jaký je obvyklý u mezinárodní matematické olympiády.

SOUTĚŽNÍ ÚLOHY

1. Pro přirozená čísla X a Y budeme říkat, že X se vyskytuje v Y, jestliže se dá binární zápis čísla X získat z binárního zápisu Y vyškrtnutím (vyloučením) žádné, jedné nebo více cifer. (Například $X = 1010$ se vyskytuje v $Y = 1001100$.)

Vytvořte algoritmus, který pro daná dvě přirozená čísla A a B najde maximální číslo C, které se vyskytuje v A i v B.

2. Je dáno n karet, které jsou očíslovány 1, 2, ..., n (každé číslo se vyskytuje právě jednou). Vytvořte algoritmus, který pro libovolnou posloupnost $A1$, $A2$, ..., An těchto karet najde nejmenší počet K prostých výměn karet nutný na jejich uspořádání podle rostoucích hodnot jejich čísel. (Pod prostou výměnou se rozumí vzájemná záměna pozic libovolných dvou karet.)

Př.: Pro posloupnost 1, 5, 3, 2, 4 je výsledek $K = 2$, protože prostými výměnami karet s číslem 5 a 2 a pak karet s číslem 4 a 5 dostaneme uspořádanou posloupnost 1, 2, 3, 4, 5.

3. Je dáno prvních N^2 přirozených čísel $(N > 2)$ $1, 2, \ldots, N^2$. Sestavte algoritmus, který rozdělí tato čísla do N skupin tak, aby byly současně splněny následující tři podmínky:

1. každá skupina obsahuje právě N čísel,
2. každé číslo se nachází právě v jedné skupině,
3. součet čísel v každé skupině je stejný.

4. Uvažujme následující hru. Pro pevně dané přirozené číslo $N > 1$ hráč A zvolí přirozené číslo X, $1 \leqq X \leqq N$. Cílem hráče B je uhodnout toto číslo X pomocí dotazů typu »Je X větší, nebo rovno K?«, kde K je libovolné přirozené číslo. Hráč A musí odpovídat na dotazy pravdivě a nesmí během hry měnit X. Hráč B platí hráči A za každou odpověď. Za odpověď »ANO« platí 2 Kčs, za odpověď »NE« platí 1 Kčs.

Určete pro dané N nejmenší množství Kčs $P(N)$, které zaručeně stačí na uhodnutí libovolného přirozeného čísla X, $1 \leqq X \leqq N$.

Vytvořte takový algoritmus, podle kterého má hráč B klást dotazy tak, aby uhodl číslo X a zaplatil přitom nejvýše $P(N)$ Kčs (tj. algoritmus určující vhodné K v dotazech).

Řešení každé úlohy bylo hodnoceno maximálně 10 body. Následující tabulka ukazuje, kolik z 21 účastníků soutěže získalo jaký počet bodů za řešení jednotlivých úloh.

Úloha	Počet bodů										
	0	1	2	3	4	5	6	7	8	9	10
1	2	2	0	2	1	3	3	4	0	3	1
2	0	0	0	2	1	2	2	4	3	5	2
3	1	0	1	1	3	3	2	1	1	3	5
4	11	0	0	0	0	1	2	2	3	2	0

Průměrné bodové hodnocení jednotlivých úloh bylo následující:

Úloha	Průměrné hodnocení
1	5,24
2	7,10
3	6,48
4	3,48

Nejlepším řešitelům udělila jury celkem dvě I. ceny, tři II. ceny a pět III. cen. Naši studenti si vedli v soutěži velice dobře, získali jednu I. cenu, jednu II. cenu a dvě III. ceny.

Soutěžící ze Sovětského svazu obdrželi jednu I. cenu a tři III. ceny, bulharští studenti pak dvě II. ceny. Bodové zisky jednotlivých zúčastněných družstev jsou shrnuty do následující tabulky.

ČSSR	celkem 168 bodů	průměr 24 bodů na žáka
SSSR	celkem 176 bodů	průměr 22 bodů na žáka
BLR	celkem 126 bodů	průměr 21 bodů na žáka

SLOŽENÍ A VÝSLEDKY ČESKOSLOVENSKÉHO DRUŽSTVA

Vladan Majerech, 4., G Pardubice
— 1. místo, 38 bodů (9, 10, 10, 9), I. cena
Pavel Kozlovský, 3., G Jindřichův Hradec
— 3. místo, 33 bodů (10, 6, 10, 7), II. cena
Pavol Kolník, 4., G Nové Mesto nad Váhom
— 7. místo, 23 bodů (1, 9, 5, 8), III. cena
Rudolf Burcl, 4., G Trnava
— 10. místo, 21 bodů (4, 5, 6, 6), III. cena
Peter Klein, 4., G A. Markuša, Bratislava
— 12.—15. místo, 19 bodů (1, 5, 7, 6)
Rastislav Senderák, 4., G Prešov
— 12.—15. místo, 19 bodů (7, 3, 9, 0)
Branislav Stríženec, 3., G J. Hronca, Bratislava
— 20. místo, 15 bodů (5, 8, 2, 0)