Výsledná výsledková listina sedmnáctého ročníku KSP

		škola	ročník	1751	1752	1753	1754	1755	suma	celkem
1.	Miroslav Klimoš	G Lanškr	0	8	9	11	10	10	45	234
2.	Josef Pihera	G Strakon	2	12	9	9	7	10	45	212
3.	Ondřej Bílka	G Zlín	3	3	4	6	10	9	32	185
4.	Jan Pelc	G UBrod	3	3	8	8		10	29	164
5.	Pavel Klavík	G Chrudim	2	5	9	5	1	4	23	154
6. – 7.	Zbyněk Konečný	GKptJaroš	2	3	2	7	1	5	18	149
	Peter Perešíni	GJGTajov	3						0	149
8.	Adam Zivner	G UBrod	3		9	9	0	10	28	148
9.	Peter Černo	GLŠtúra	4						0	130
10.	Miroslav Cicko	GJGTajov	4						0	123
11.	Jan Bulánek	G Klatovy	4		8				8	107
12.	Roman Smrž	GOhradní	1	8	2	8		5	20	102
13.	Jakub Kaplan	GJKTyla	1	3	0	6		8	17	100
14.	Martin Koníček	G UBrod	4			7			7	93
15.	Jan Hrnčíř	GFXŠaldy	3	3	2	7			12	89
16.	Lukáš Lánský	GJKTyla	1	3	5	1		6	15	81
17.	Petr Kratochvíl	G SvětláNS	2	3	7			6	16	77
18.	Cyril Hrubiš	G Bílovec	3	8	5	7			17	60
19.	Eva Schlosáriková	G Piešťany	4						0	59
20.	Martin Čech	G UBrod	4						0	52
21.	Tomáš Herceg	G Třebíč	2	3	2				5	47
22.	Stanislav Basovník	G Kroměříž	4						0	43
23.	Tereza Klimošová	G Lanškr	3			8		10	18	42
24.	Daniel Marek	GZborov	3			7			7	34
25.	Martin Kupec	GMendel	3						0	33
26.	Zbyněk Falt	GNeumannov	4						0	32
27.	Michal Pavelčík	G UBrod	2	3	2	6			11	31
28.	Adam Ráž	GBudějo	2	5				5	9	28
29.	Josef Špak	GJírovco	2						0	25
30.	Lukáš Špalek	G Čadca	4						0	24
31.	Ondřej Bouda	GKptJaroš	2						0	18
32.	Marian Kaluža	GHavlíčkov	2						0	16
33. – 35.	Jiří Cabal	SPŠ DvKrál	2						0	15
	Ondřej Garncarz	G Příbor	4						0	15
	Martin Kahoun	GJNerudy	2						0	15
36.	Jan Palenčar	G Martin	2						0	14
37.	Martin Podloucký	G Strážnic	4						0	12
38. – 41.	Jakub Jenis	GsvCyrMet	1						0	11
	Hana Klempová	GUBalvanJN	4						0	11
	Jakub Porod	G Týn nV	2						0	11
	Ján Zahornadský	GZborov	4						0	11
42. – 44.	Lukáš Beleš	G Čadca	4						0	10
	Jakub Benda	GJNerudy	2						0	10
	Michal Vaner	G Turnov	3						0	10
45. – 47.	Kateřina Böhmová	G Rožnov	3						0	8
	Jiří Machálek	G Holešov	3						0	8
	Petr Soběslavský	GJHeyrovs	4						0	8
48. – 49.	Daniel Sedláček	SPŠE Hav	1						0	7
	Filip Šauer	G Klatovy	4						0	7
50.	Jiří Nohavec	G Domažl	4						0	6
51. – 55.	Dalibor Adamčík	SPŠE Preš	2						0	5
	Tomáš Ehrlich	G Holešov	2	3					3	5
	Petr Musil	G MBuděj	3						0	5
	Jan Staněk	GKptJaroš	3						0	5
	Zdeněk Vilušínský	G Turnov	4						0	5
56. – 57.	Florián Danko	SPŠEtech	2			1			1	2
	Martin Vařák	G Bílovec	2						0	2
58. – 59.	Tamara Kuštárová	GBiling	0						0	1
	Petr Zimčík	G UBrod	1						0	1
60. – 62.	Miroslav Hovorka	GJateční	4						0	0
	Adrián Lachata	G Svidník	3						0	0
	Michal Onderko	SPŠ Karviná	3						0	0

Milí řešitelé!

Touto opravenou pátou sérií jsme zakončili sedmnáctý ročník našeho semináře. Ale přeci ne úplně, na podzim bude ještě soustředění pro naše nejlepší řešitele. Soustředení se letos bude konat nejspíš v termínu 16. až 22. října a pozveme na něj přibližně třicet řešitelů z první čtyřicítky.

Ještě bychom Vás chtěli poprosit: myslíme si, že je škoda, že se našeho semináře účastní tak málo řešitelů. Byli bychom rádi, pokud byste nám na začátku příštího roku pomohli s „reklamou“ – například pověšením zadání první série na školní nástěnku, ...

Aktuální informace o KSP naleznete na stránkách *http://ksp.mff.cuni.cz/*, dotazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz.*

Vzorová řešení páté série sedmnáctého ročníku KSP aneb Tractatus brevis De hippopotamis

17-5-1 Velkovezír

Došlá řešení se dala rozdělit na tři skupiny, a to na jednak na triviální kvadratická (k nim není co dodat, pomocí částečných součtů řady se zkusily všechny možné úseky a vybral se ten s nejlepším průměrem), na řešení se složitostí $O(KN)$ (také se zkouší všechny možnosti, ale uvědomíme si, že mezi všemi posloupnostmi s maximálním průměrem existuje alespoň jedna, která má délku menší než $2K$, viz druhé pozorování) a na lineární. Jak na to?

Začněme *pozorováním*: Pokud máme nějakou posloupnost s průměrem d a rozdělíme ji na dvě části, alespoň jedna z těchto částí musí mít stejný nebo větší průměr než původní posloupnost. Nechť má jedna část délku l_1 a druhá l_2. Kdyby toto tvrzení neplatilo (čili průměr první části $d_1 < d$ a taktéž $d_2 < d$), byl by průměr celé posloupnosti:

$$\frac{l_1 \cdot d_1 + l_2 \cdot d_2}{l_1 + l_2} < \frac{l_1 \cdot d + l_2 \cdot d}{l_1 + l_2} = d$$

ostře menší než d, což není možné, protože d je její průměr. Navíc stejné pozorování se dá provést i pro opačnou nerovnost, že tedy průměr jedné z částí musí být menší nebo roven průměru celé posloupnosti.

Přimíchejme ještě toto *pozorování*: Označme $\varphi_{a\dots b}$ průměr čísel s indexy a až b a $\varphi_{\max}$ největší průměr nějaké posloupnosti. Předpokládejme, že posloupnost s maximálním průměrem končí prvkem s indexem $L > i+K$ a že průměr všech posloupností, které končí prvkem i, není maximální (matematik by řekl, že pro každé $1 \le j < i$ platí $\varphi_{j\dots i} < \varphi_{\max}$). Potom posloupnost s maximálním průměrem začíná na čísle s indexem větším než i.

Rozepišme, jak dopadne průměr posloupnosti začínající prvkem $j \le i$:

$$\begin{aligned}\varphi_{j\dots L} &= \frac{\varphi_{j\dots i} \cdot (i-j+1) + \varphi_{i+1\dots L} \cdot (L-i)}{L-j+1} \le \\ &\le \frac{\varphi_{j\dots i} \cdot (i-j+1) + \varphi_{\max} \cdot (L-i)}{L-j+1} < \\ &< \frac{\varphi_{\max} \cdot (i-j+1) + \varphi_{\max} \cdot (L-i)}{L-j+1} = \varphi_{\max}.\end{aligned}$$

Tedy žádné z čísel s indexem $1 \dots i$ nemůže být obsaženo v posloupnosti s maximálním průměrem.

Dost už bylo přihazování pozorování, pojďme z nich nyní vařit algoritmus. Na začátku vezmeme prvních K prvků, které budou tvořit zpracovávanou podposloupnost. V každém kroku algoritmu posuneme pravý konec zpracovávané posloupnosti o jeden prvek doprava. Její levý konec už nemusíme posouvat zpátky doleva, můžeme ho nechat tam, kde je (a druhé pozorování nám říká, že nepřijdeme o žádnou posloupnost s maximálním průměrem). Levý konec tedy nemusíme posouvat doleva, ale může se nám stát, že ho budeme muset posouvat doprava, abychom zvětšili průměr zpracovávané posloupnosti. Jak, to vyřešíme za chvilku. Takto v tomto kroku najdeme posloupnost s největším průměrem, která má pevný pravý konec a vznikla zkracováním posloupnosti z minulého kroku. (Netvrdíme, že tato posloupnost má největší průměr ze všech, které končí tímto pravým okrajem, ale z druhého pozorování víme, že jsme nezapomněli na posloupnost s maximálním průměrem, a to nám stačí.) Když si z těchto posloupností (pro každý pravý okraj máme jednu) vybereme tu s největším průměrem, najdeme určitě posloupnost s maximálním průměrem.

Jak tedy přesně vypadá krok algoritmu? Na začátku $L := 1$ a $P := K$, zpracovávaná posloupnost je prvních K prvků. V každém dalším kroku uděláme

- $P := P + 1$
- dokud existuje $L' > L$, že $P - L' + 1 \ge K$ (nezkrátíme moc) a $\varphi_{L'\dots P} > \varphi_{L\dots P}$ (zlepšíme průměr), pokládáme $L := L'$. Navíc pokud použijeme naše první pozorování a trochu se zamyslíme, zjistíme, že podmínka $\varphi_{L'\dots P} > \varphi_{L\dots P}$ (část posloupnosti má větší průměr) je stejná jako podmínka $\varphi_{L\dots L'-1} < \varphi_{L\dots P}$ (druhá část posloupnosti má menší průměr). Pro nás bude druhá varianta lepší.

Zbývá tedy vyřešit, jak zjistit, když máme L a P, jestli existuje prvek L', aby průměr posloupnosti prvků $L \dots L'-1$ byl menší než průměr prvků $L \dots P$ ($\varphi_{L\dots L'-1} < \varphi_{L\dots P}$). Použijeme k tomu datovou strukturu, která bude kombinací fronty a zásobníku (bude umět data přidávat na jeden konec a odebírat je z konců obou), říkejme jí *frobník*. Ve frobníku budeme mít uloženou informaci o tom, jak vypadají průměry posloupností mezi prvky L a $P-K$. Přesněji řečeno v něm budou uloženy průměry posloupností $X_0 \dots X_1 - 1, X_1 \dots X_2 - 1, \dots, X_{S-1} \dots X_S - 1$ s tím, že $X_0 = L$ (začínáme vždy v L) a $X_S = P - K + 1$ (končíme vždy v $P-K$). Navíc bude vždy platit, že průměr jedné posloupnosti je menší než průměry všech posloupností, které se nacházejí ve frobníku (a tedy i v původní posloupnosti) *za* ní, matematicky řečeno $\varphi_{X_{i-1}\dots X_i-1} < \varphi_{X_i\dots X_{i+1}-1}$.

Data budeme ve frobníku udržovat následujícím způsobem. Na začátku v něm není nic, protože $P - K = 0$. Pak vždy, když zvětšujeme P, přidáme na konec frobníku průměr jednoprvkové posloupnosti s prvkem na indexu $P - K$ (když přidáváme prvek do zkoumané posloupnosti, přidáme do frobníku prvek, který ještě může být v posloupnosti končící prvkem P). To nám ale mohlo pokazit vlastnost zvětšujících se průměrů. Pokud se tak stalo ($\varphi_{X_{S-1}\dots X_S-1} \ge \varphi_{P-K\dots P-K} = P - K$), budeme slučovat dvě poslední posloupnosti ve frobníku do jedné, dokud nebude platit, že průměr poslední posloupnosti ve frobníku je větší než průměr posloupnosti předposlední, případně dokud nesloučíme všechny posloupnosti do jedné.

Když jsme tedy takto upravili frobník, můžeme zkusit najít hledané L', aby průměr prvků $L \ldots L'-1$ byl menší než průměr prvků $L \ldots P = \varphi_{L\ldots P}$. Vezmeme první posloupnost z frobníku (je to posloupnost $L \ldots L'-1$) a pokud má průměr menší než $\varphi_{L\ldots P}$, je to naše hledaná posloupnost s menším průměrem, tedy položíme $L = L'$, a tuto posloupnost z frobníku odebereme. Takto pokračujeme, dokud je průměr první posloupnost ve frobníku menší než $\varphi_{L\ldots P}$ nebo dokud frobník nevyprázdníme.

Nyní už víme, jak algoritmus pracuje, zbývá zjistit složitost. Program se skládá z N kroků, v každém zvětšujeme P, upravujeme L (frobník) a testujeme, zda je nalezená posloupnost lepší než dosud nalezená. Kromě úprav frobníku jsou všechny tyto operace konstantní, tedy časová složitost algoritmu bez úprav frobníku je lineární. Do frobníku vložíme nanejvýš N posloupností, každá se může nanejvýš jednou sloučit a jednou vyndat, takže všechny operace s frobníkem trvají dohromady také jenom lineárně dlouho. (V jednom kroku algoritmu sice můžeme ve frobníku najednou sloučit až $O(N)$ posloupností, ale uvědomte si, že sloučit mohu jenom to, co jsem do frobníku dal, takže slučování je celkem nanejvýš $N-K$.) Tedy časová složitost celého algoritmu je lineární, paměťová taktéž.

Pavel Čížek a Milan Straka

17-5-2 Ranní hroše

Při řešení každé úlohy je nejprve nutno pochopit zadání. To se mnohým řešitelům této úložky nepovedlo i přesto, že úloha byla zadána poměrně srozumitelně. Úkolem bylo zjistit, zda existuje nějaký interval h z množiny H a v z V, že h začíná i končí dříve než v. Tedy jedná se o úlohu zjišťovací, na kterou stačilo odpovědět ano nebo ne. Vypsání hledané dvojice intervalů navíc samozřejmě není nijak na škodu, ale nebylo potřeba. Co však již vadí, je situace, ve které řešitel vypisuje všechny vyhovující dvojice intervalů, to vede na triviální kvadratický algoritmus. Takováto řešení nechť do budoucna odradí ne víc než dva body za funkčnost.

Tato úloha se dá optimálně pořešit rozličnými přístupy. Zkusme nejprve setřídit obě množiny dohromady podle počátků intervalů. Nyní procházejme po setříděné posloupnosti. Budeme si přitom pamatovat jeden interval z množiny H, říkejme mu kandidát, který má takovou vlastnost, že pokud existuje nějaká dvojice vyhovujících intervalů, pak existuje i vyhovující dvojice, ve které se nachází náš kandidát. Na počátku si poznačme, že kandidáta ještě nemáme. Pokud tedy při průchodu narazíme na interval z množiny H, pak ho porovnáme s kandidátem. V případě, že nově nalezený interval je „lepší“, označíme ho za nového kandidáta. Nově nalezený interval A je „lepší“ než interval B právě, když jeho konec je menší.

Nyní rozeberme případ, kdy narazíme na interval z množiny V. Pokud ještě nemáme kandidáta, znamená to, že neexistuje žádný interval z množiny H takový, který má menší začátek, tehdy jdeme v posloupnosti dále. Pokud však již máme nějakého kandidáta, pak jeho začátek je díky setříděnosti menší než začátek nově nalezeného intervalu. Stačí tedy porovnat konec kandidáta s koncem nového intervalu a v případě, že kandidát má konec intervalu menší, skončíme, protože jsme právě nalezli vyhovující dvojici.

Linearita průchodu po setřídění i jeho konečnost je zřejmá. Celková časová složitost algoritmu je tedy ovlivněna tříděním, které pro neceločíselné intervaly umíme při nejlepším v čase $O(N \log N)$, kde N budiž součet velikostí množin H a V. Paměťová složitost je vůči stejnému N lineární.

Zbývá ukázat, že pokud řešení existuje, náš algoritmus ho najde. Ať tedy dvojice intervalů splňující zadání existuje, označme ji h a v, kde h je z H, v z V, pak při průchodu posloupností narazíme na h dříve než na v. Tedy jistě dojde k porovnání h s kandidátem, v případě, že je kandidát „lepší“, pak dvojice kandidát a v také splňuje zadané podmínky. V případě, že kandidát není „lepší“, pak dojde k nahrazení kandidáta intervalem h. Tedy po průchodu přes h splňuje kandidát podmínky pro hledané intervaly. Zřejmě pokud v budoucnu se kandidát zlepší, stále bude kandidát splňovat podmínky. A konečně až narazíme při průchodu na v, porovnáme jej s kandidátem a program skončí.

Na závěr zmíním, že řešení, která měla po setřídění již jen lineární průchody, tedy taková, která by se v případě celočíselných intervalů dala napsat i se setříděním v lineárním čase, byla o maličko lépe ohodnocena než ostatní řešení.

David Matoušek

17-5-3 Nouze V-dáli-hrocha

Poslyšte příběh kratochvilný o tom, jak hrošík v nouzi poprosil o pomoc kmotřičku Rekurzi (ťuky ťuk!) a jak to dopadlo.

Maličký hrošík to asi ještě neví, ale to, co potřebuje, je vypsat všechny *permutace* množiny $\{1, \ldots, N\}$ (čili všechna možná uspořádání těchto čísel) tak, aby se sousední permutace lišily prohozením právě jedné dvojice prvků (máte-li rádi cizí slovíčka, tak jednou *transpozicí*). A my vyřešíme rovnou těžší variantu úlohy, která po nás žádá, aby se jedinou transpozicí lišila i první a poslední permutace.

Jak na to? Všimněme si, že všechny permutace N prvků získáme z permutací $N-1$ prvků tak, že do každé původní permutace vložíme prvek N postupně na všechna možná místa. Čili pokud už víme, že pro $N=2$ existují permutace 12 a 21, pak pro $N=3$ můžeme z 12 získat 312, 132 a 123, zatímco z 21 získáme 321, 231 a 213. Když si uvědomíme, že míst, kam vložit nový prvek, je vždy N, dostaneme ihned vzoreček, který nám řekne, že permutací N prvků je $p(N) = N \cdot p(N-1)$, čili $p(N) = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1$ (obvykle značíme $N!$, tedy faktoriál z N).

Vyzbrojeni tímto pozorováním získáme ihned jednoduchý algoritmus, který nám všechny permutace vygeneruje. Pokud je $N=1$, vrátíme ihned jedinou permutaci, a to jedničku samu. Pokud je $N>1$, rekurzivním zavoláním našeho algoritmu vyřešíme úlohu pro $N-1$, čímž získáme nějaké permutace p_1 až $p_{(N-1)!}$. Nyní do nich budeme vkládat N-tý prvek, přičemž do p_1 ho vložíme nejdříve na poslední pozici, pak na předposlední, atd. až na první, zatímco u p_2 budeme postupovat popředu, u p_3 opět pozpátku atd.

Nejlépe to asi bude vidět na příkladu:

Pro $N=1$: 1
Pro $N=2$: 12 21
Pro $N=3$: 123 132 312 321 231 213

Tak dosáhneme, že se sousední permutace liší jen jednou transpozicí: mezi dvěma sousedními permutacemi vzniklými z jedné p_i se přesunulo pouze N na sousední pozici, zatímco mezi poslední vzniklou z p_i a první z p_{i+1} zůstalo N na místě a změnily se pouze ostatní prvky, ovšem podle stejného algoritmu, takže také s jedinou transpozicí [ejhle, důkaz indukcí].

První vygenerovaná permutace bude určitě $12 \ldots N$ (každý prvek startuje vpravo). Jak ale bude vypadat ta poslední?


```
static void vypln_L (void)                          /* Vyplní tabulku L. */
{
  int u, v, l, z;

  for (v = 0; v < N; v++)
    {
      L[v][v][0] = L[v][v][1] = 0;
      predposledni[v][v][0] = predposledni[v][v][1] = -1;
    }

  for (l = 1; l < N; l++)
    for (u = 0; u < N; u++)
      {
        v = (u + l) % N;
        for (z = 0; z < 2; z++)
          vypln_polozku (u, v, z);
      }
}

static void vypis_cestu (int u, int v, int x)       /* Vypíše cestu pokrývající interval ⟨u,v⟩ */
{                                                   /* a končící vrcholem x (x je buď u nebo v). */
  int z, predp;
  int u1, v1;

  u1 = (u + 1) % N;
  v1 = (v + N - 1) % N;

  z = (v == x);
  predp = predposledni[u][v][z];
  if (predp == -1)
    {
      printf ("%d", body[x].cislo + 1);
      return;
    }

  if (z)
    vypis_cestu (u, u1, predp);
  else
    vypis_cestu (u1, v, predp);

  printf (" %d", body[x].cislo + 1);
}

int main (void)                                     /* Hlavní program. */
{
  int v, minv, min;

  nacti ();
  vypln_L ();

  min = L[0][N - 1][0];
  minv = 0;
  for (v = 1; v < N; v++)
    if (L[v][v - 1][0] < min)
      {
        min = L[v][v - 1][0];
        minv = v;
      }

  if (min == NEKONECNO)
    printf ("Cesta neexistuje.\n");
  else
    {
      printf ("Cesta mající délku %d:\n", min);
      vypis_cestu (minv, (minv + N - 1) % N, minv);
      printf ("\n");
    }

  return 0;
}
```

```
  dxa = ba->x - body[0].x;
  dya = ba->y - body[0].y;
  dxb = bb->x - body[0].x;
  dyb = bb->y - body[0].y;
  return - (dxa * dyb - dya * dxb);
}
static void nacti (void)                                  /* Načte body a vzdálenosti a setřídí body podle  */
{                                                         /* pořadí podél obvodu mnohoúhelníka.  */
  int u, v, l;
  scanf ("%d", &N);
  for (v = 0; v < N; v++)
    {
      scanf ("%d%d", &body[v].x, &body[v].y);
      body[v].cislo = v;
    }
  qsort (body + 1, N - 1, sizeof (struct bod), porovnej_smery);
  for (u = 0; u < N; u++)
    for (v = 0; v < N; v++)
      vzdalenost[u][v] = NEKONECNO;
  while (scanf ("%d%d%d", &u, &v, &l) == 3)
    vzdalenost[u - 1][v - 1] = vzdalenost[v - 1][u - 1] = l;
}
static void rozsir_usek (int u, int v, int dalsi, int *delka, int *predp)      /* Rozšíří nejlepším způsobem interval  */
{                                                         /* ⟨u, v⟩ přidáním dalšího vrcholu cesty dalsi.  */
  int cislo_u = body[u].cislo;                            /* Délku rozšířené cesty uloží do delka  */
  int cislo_v = body[v].cislo;                            /* a index předposledního vrcholu do predp.  */
  int cislo_dalsi = body[dalsi].cislo;
  int lu, lv;
  if (L[u][v][0] == NEKONECNO || vzdalenost[cislo_u][cislo_dalsi] == NEKONECNO)
    lu = NEKONECNO;
  else
    lu = L[u][v][0] + vzdalenost[cislo_u][cislo_dalsi];
  if (L[u][v][1] == NEKONECNO || vzdalenost[cislo_v][cislo_dalsi] == NEKONECNO)
    lv = NEKONECNO;
  else
    lv = L[u][v][1] + vzdalenost[cislo_v][cislo_dalsi];
  if (lu < lv)
    {
      *delka = lu;
      *predp = u;
    }
  else
    {
      *delka = lv;
      *predp = v;
    }
}
static void vypln_polozku (int u, int v, int z)           /* Vyplní L[u][v][z].  */
{
  int u1, v1;
  u1 = (u + 1) % N;
  v1 = (v + N - 1) % N;
  if (z)
      rozsir_usek (u, v1, v, &L[u][v][z], &predposledni[u][v][z]);      /* Určujeme L(u, v, v).  */
  else
      rozsir_usek (u1, v, u, &L[u][v][z], &predposledni[u][v][z]);      /* Určujeme L(u, v, u).  */
}
```


Jelikož pro $N > 2$ je $(N-1)!$ sudé číslo, budeme v $p_{(N-1)!}$ posouvat N-kem zleva doprava, takže N skončí na poslední pozici. Indukcí nahlédneme, že tak dopadnou i všechny ostatní prvky až na jedničku a dvojku, které zůstanou prohozené. Proto poslední permutace bude $2134\ldots N$, přesně, jak potřebujeme.

Náš algoritmus má ale jeden velký háček: potřebuje z rekurze vracet celý vygenerovaný seznam permutací, takže má paměťovou složitost $O(N \cdot N!)$. To věru není dobré, ovšem můžeme to snadno napravit: připravíme si už na začátku celou počáteční permutaci $12\ldots N$ a použijeme rekurzivní proceduru, která pro dané i proskáče aktuální permutaci prvkem i a mezi každými dvěma skoky zavolá sama sebe pro $i+1$. Jen si pro každý prvek potřebujeme pamatovat, jestli jsme jím naposledy skákali doleva nebo doprava, abychom správně střídali směry, a také se nám bude hodit udržovat si *inverzi* permutace, čili pole, které nám pro každý prvek řekne, kde se zrovna v permutaci nachází.

Všimněte si, že při proskakování i-čkem nás prvky větší než i nebudou rušit, protože jsou bezpečně uklizeny na jednom či druhém kraji permutace, takže opravdu stačí i-čkem posouvat na sousední pozici ve správném směru. Tím jsme vlastně mimoděk splnili mnohem víc, než po nás zadání chtělo: používáme jen transpozice *sousedních* prvků.

Zbývá rozebrat složitost: Naše rekurzivní procedura potřebuje jen konstantní čas na vymyšlení jedné permutace, leč na její vypsání je potřeba čas lineární. Proto pokud chceme vypisovat celé permutace, časová složitost nutně dosáhne $O(N \cdot N!)$ a lépe to ani nejde, protože je to čas lineární ve velikosti výstupu; kdyby nám stačilo vypsat posloupnost prohazování, zvládli bychom to v čase $O(N!)$ a opět to lépe nemůže jít. Paměti spotřebováváme lineární množství (lineárně velká globální pole a konstantně na každou z N úrovní rekurze).

Pozorný čtenář si asi povšimne, že argument s velikostí výstupu na jednu nohu pokulhává, protože výstupem je přeci *desítkový* zápis permutace, ve kterém na každý prvek spotřebujeme řádově $\log N$ číslic, takže celý zápis musí měřit $O(N \log N)$ namísto $O(N)$. To je i není pravda, tato potíž je totiž důsledkem toho, že jsme si nikdy paměťovou složitost (ani velikost vstupu a výstupu) nezavedli precizně. Dá se zavést dvěma způsoby: buďto můžeme počítat složitosti na chlup přesně a měřit velikost vstupu a výstupu i zabranou paměť v bitech (což „opravdová“ teorie složitosti skutečně dělá a zde ji vyjde $N \cdot \log N$), nebo si zvolíme jako základní jednotku jeden integer (rozumné velikosti, řekněme polynomiální v N, abychom předešli trikům à la naskládání celého vstupu do jediného integeru) a vše měříme v integerech. V KSP obvykle používáme ten druhý, výrazně jednodušší (i když někdy zbytečně hrubozrnný) přístup, a ten nám v tomto případě říká, že výstup je velký jenom $O(N)$. Podobně je to s časovou složitostí: v druhém případě považujeme každou aritmetickou operaci za konstantně rychlou, v prvním její složitost závisí na počtu bitů čísel, se kterými operace počítá. Toto téma ještě nakousneme v úvodu k dalšímu ročníku KSP.

Martin Mareš

17-5-4 Kudy tudy cestička

Začneme tím, že si určíme, v jakém pořadí po sobě následují zadané body na obvodu mnohoúhelníka (řekněme proti směru hodinových ručiček). Jeden ze způsobů jak to udělat je vybrat si nejlevější bod bod A a ostatní body setřídit sestupně podle úhlu, který svírá jejich spojnice s bodem A s osou x. Možná o něco jednodušší než si pro každý takový vektor počítat tento úhel, je pouze umět pro libovolné dva takové vektory rozhodnout, který z nich je má tento úhel větší. To poznáme podle znaménka jejich vektorového součinu: Pokud je tento součin kladný, leží druhý vektor v levé polorovině určené prvním vektorem, a tedy je úhel druhého vektoru větší. Povšimněte si, že pokud vektory porovnáváme tímto způsobem, můžeme si bod A vybrat libovolně, tj. nemusíme ani hledat nejlevější zadaný bod.

Nyní uvažme počáteční úsek P' libovolné nekřížící se cesty P, která projde všechny vrcholy. Vrcholy navštívené P' tvoří souvislý interval I na obvodu mnohoúhelníka (bráno cyklicky, tedy za N-tým vrcholem následuje první). Kdyby tomu tak nebylo a P' by nějaký vrchol w přeskočil, cesta P by se bez křížení nemohla dostat zároveň do w a do ostatních vrcholů. Z toho plyne, že poslední vrchol u úseku P' musí být jeden z krajních bodů intervalu I, a následující vrchol v cesty P je jeden z maximálně dvou vrcholů, které sousedí s krajními body I na obvodu mnohoúhelníka (samozřejmě u a v musí být také spojeny pěšinkou).

Nabízí se řešení zvolit si nějaký vrchol a poté procházet všechny cesty, které v něm začínají. Nicméně podle pozorování z předchozího odstavce může být možné každý počáteční úsek rozšířit dvěma způsoby, tedy všech takových cest může být řádově až 2^N. To je příliš mnoho a přímočarý program založený na této myšlence by byl neúnosně pomalý.

Označme si $\ell(y, z)$ délku pěšinky mezi dvěma vrcholy y a z (tato hodnota je ∞, pokud mezi vrcholy y a z pěšinka není). Procházet všechny cesty je beznadějné, ale všimli jsme si, že počáteční úsek libovolné nekřížící se cesty odpovídá nějakému intervalu. Intervalů není mnoho (jen řádově N^2), zkusme toho využít. Je pro nás celkem nepodstatné, jak přesně cesta vypadá uvnitř intervalu, zajímá nás pouze její délka. Budeme si tedy pro každý interval mezi vrcholy s čísly u a v počítat délku nejkratší cesty, která pokryje interval u a v a navíc skončí v předepsaném vrcholu x (kde x je buď u nebo v). Označme si délku nejkratší takové cesty $L(u, v, x)$. Nechť bez újmy na obecnosti $x = v$. Předposlední vrchol této cesty potom je buď u nebo $v-1$, a z těchto možností si chceme vybrat tu lepší. Takto dostáváme, že

$$L(u,v,v) = \min(L(u,v-1,u) + \ell(u,v),\\ L(u,v-1,v-1) + \ell(v-1,v)).$$

Z tohoto vztahu již snadno všechny hodnoty $L(u, v, x)$ spočítáme – k jejich výpočtu potřebujeme znát hodnoty L pro kratší intervaly, uděláme si tedy tabulku, do níž budeme počítat tyto hodnoty postupně podle rostoucí délky intervalu. Ještě zmiňme, že pro intervaly délky 0 (tj. jednotlivé vrcholy) si nastavíme $L(v, v, v) = 0$. Tato tabulka bude mít velikost $2N^2$ a každé její políčko spočítáme z předchozích hodnot v konstantním čase. Délku nejkratší cesty, která projde všechny vrcholy, pak dostaneme jako minimum z hodnot $L(v, v-1, v)$ pro všechny vrcholy v.

Zbývá přijít na to, jak najít tuto cestu, ne jen její délku. Při výpočtu hodnoty $L(u, v, x)$ si můžeme zapamatovat, který vrchol má být předposlední. Pak snadno cestu zkonstruujeme odzadu – začneme posledním vrcholem v_N, k němu si najdeme předposlední v_{N-1}, k tomu pak v_{N-2}, atd., až dokud nedojdeme k prvnímu vrcholu.

Třídění vrcholů na obvodu mnohoúhelníka nám zabere čas $O(N \log N)$, délku cesty si určíme vyplněním tabulky v čase

$O(N^2)$ a cestu samotnou nalezneme v čase $O(N)$, výsledná časová složitost je tedy $O(N^2)$. Paměťová složitost je dána velikostí tabulek L a ℓ a je tedy $O(N^2)$. Povšimněte si, že pokud bychom chtěli znát pouze délku nejkratší cesty P a nepotřebovali bychom P vypsat, stačilo by nám pole L velikosti $O(N)$ – počítali bychom si hodnoty $L(u,v,x)$ podle vzrůstající délky k intervalu mezi vrcholy u a v a pamatovali bychom si pouze dva řádky tabulky – $(k-1)$-ní a k-tý.

Zdeněk Dvořák

17-5-5 Jazykozpytec se loučí

Nebude zřejmě na škodu, když si ještě jednou pořádně rozmyslíme, co jsou to vlastně gramatiky. Gramatika je nástroj, který se používá pro přesný formální popis jistého jazyka. Abychom mohli o určité gramatice diskutovat, jaký jazyk vlastně popisuje, hodí se na chvíli na ni pohlížet jako na stroj, který postupně generuje slova podle přepisovacích pravidel. Takový výpočet je v principu nedeterministický, typicky totiž bývá na výběr několik pravidel, které se v daném okamžiku mohou použít. Některé větve výpočtu jsou slepé – pokud se v nich expandované slovo ocitne, nikdy z něj již nevymizí neterminální symboly a výpočet se nezastaví. Všechny možné větve výpočtu, které naopak úspěšně skončí odstraněním všech neterminálů, potom svým výsledným slovem tvoří jazyk gramatiky. Chceme-li sestrojit gramatiku popisující nějaký jazyk L, musíme jednak zajistit, aby se sadou přepisovacích pravidel bylo možno vytvořit všechna slova jazyka L, ale hlavně ukázat, že všechny ostatní větve výpočtu jsou slepé a gramatika tak netvoří slova nepatřící do L.

Úloha 1: První úloha je snadná, gramatiku pro jazyk $\{a^i b^j c^k;\ 1 \le i \le j \le k\}$ vytvoříme úpravou příkladu ze zadání. Nosná myšlenka bude následující: kromě původních pravidel z příkladu, které zajišťovaly namnožení stejného počtu symbolů a, b a c, dodáme ještě pravidlo na přimnožení libovolného množství symbolů b a c, od obou ovšem stejný počet, a konečně ještě jedno pravidlo pro přimnožení libovolného počtu samotných symbolů c. Zjevně tak bude v každém okamžiku platit $1 \le i \le j \le k$, a každá platná kombinace počtů i, j, k bude naší gramatikou pokryta.

Tedy přesně, sestrojíme gramatiku $G = (V_N, V_T, S, P)$, kde množina neterminálů bude $V_T = \{a, b, c\}$, množina neterminálů $V_N = \{S, B, C, X\}$, startovní symbol S a sada přepisovacích pravidel tato:

$$S \to aSBC \mid abC$$
$$CB \to BC \quad (CB \to XB,\ XB \to XC,\ XC \to BC)$$
$$bB \to bb$$
$$bC \to bc \mid bbCC \mid bCC \quad \ldots \text{množíme } bc \text{ a } c$$
$$cC \to cc$$

Abychom byli poctiví, bylo by ještě třeba si přesně zdůvodnit, že gramatika nevydá nepatřičná slova a nechtěné větve výpočtu tedy skončí jako slepé. Věříme však, že máte již dostatek zkušeností a znalostí, abyste si to po chvíli dívání na sadu pravidel bez problémů uvědomili sami.

Úloha 2: Druhá úloha se ukázala být pro některé řešitele oříškem, a občas tvrdili, že gramatiku popisující jazyk $\{a^{2^n};\ n \in N\}$ nelze sestrojit, kupodivu i s „důkazem“. To samozřejmě není pravda, příslušná gramatika existuje a my si jednu takovou ukážeme.

V první fázi nejdříve vytvoříme jeden symbol A. V každé další fázi potom rozmnožíme všechny symboly a na dvojnásobný počet. Problém ovšem je, jak toho v rámci jedné fáze dosáhnout. Jediné pravidlo typu $A \to AA$ by zjevně násobným použitím produkovalo i jiné počty, než 2^n.

Pomůžeme si speciálním neterminálním symbolem K, „kurzorem“, který bude běhat po slovu, vždy zleva doprava, přeskočí každé A a zdvojí ho při tom. Budeme při tom potřebovat poznačit si, kde aktuální slovo začíná a končí, obalíme si ho tedy na začátku symboly L a R, které se mohou změnit na A. Zbývá chytře navrhnout sadu přepisovacích pravidel P tak, aby gramatika nevydávala špatná slova.

$S \to a$ ... ošetři jednopísmenné slovo
$S \to LR$ ... obal slovo mezemi
$L \to A$ ... meze jsou v podstatě skryté A
$R \to A$
$L \to LAK$ ... vytvoř vlevo nový kurzor
$KA \to AAK$ ... přeskoč A a zdvoj ho; podvádíme
$KR \to AR$ ... kurzor dorazil na konec, zruš ho
$A \to a$ ... a nahraď neterminály terminály

Všimněme si, že kurzorů může být v jednom okamžiku ve slově i více, ale protože zanikají až na konci slova, ničemu to nepřekáží a správně zdvojnásobí všechna A. Zdvojovací pravidlo není kontextové, ve skutečnosti tedy použijeme známý trik:

$$KA \to XA$$
$$XA \to XK$$
$$XK \to AAK$$

Formálně bude naše gramatika G čtveřice (V_N, V_T, S, P), kde $V_N = \{S, A, L, R, K, X\}$ a $V_T = \{a\}$.

Protože kurzor může zaniknout pouze až když dorazí úplně napravo, dojde tak ke správnému počtu zdvojnásobení všech symbolů A, gramatika tedy negeneruje nežádoucí slova. Naopak, pro slovo délky 2^n stačí gramatice vytvořit $n-1$ kurzorů, které už se postarají o umocnění.

Tomáš Valla

Úloha 17-5-1 – Velkovezír – program

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

struct PolozkaFrobniku {
    int Delka;
    int Soucet;
};

int N, K, Cisla[MaxN];
int AktualniDelka, AktualniSoucet;
int Levy, Pravy;                        /* aktuálně zpracovávané řady */
```



```
    dir:array [index] of -1..1;         { Kterým směrem putuje který prvek }
    i:index;                            { Pomocná proměnná prchavého významu }

procedure show;                         { Vypíše aktuální pořadí }
var k:index;
begin
   for k:=1 to N do write(a[k], ' ');
   writeln;
end;

procedure swap(i,j:index);              { Prohodí dva prvky a vypíše nové pořadí }
var k:index;
begin
   k:=a[i]; a[i]:=a[j]; a[j]:=k;
   b[a[j]]:=j; b[a[i]]:=i;
   show;
end;

procedure Vdalihroch(i:index);          { The core of the poodle: posouvá i-tý prvek }
var j:index;
begin
   if i>N then exit;                    { Ale vždyť tolik jich nemáme! }
   for j:=1 to i-1 do begin             { Posouvat ho budeme celkem (i-1)-krát }
      Vdalihroch(i+1);                  { Mezitím vždy prostrkáme prvky s většími čísly }
      swap(b[i], b[i]+dir[i]);          { A posuneme ve správném směru }
      end;
   Vdalihroch(i+1);                     { Ještě jednou pro poslední pozici }
   dir[i] := -dir[i];                   { Pamatuj, příště půjdeme opačně }
end;

begin
   read(N);                             { Velikost městečka }
   for i:=1 to N do begin               { Prostoduchá inicializace }
      a[i] := i;
      b[i] := i;
      dir[i] := -1;
      end;
   show;                                { Ukážeme, kde jsme začali }
   Vdalihroch(1);                       { Go! Go! Go! }
end.
```

Úloha 17-5-4 – Kudy tudy cestička – program

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

#define MAXN 100
#define NEKONECNO 100000

struct bod
{
   int x, y;                            /* Souřadnice bodu. */
   int cislo;                           /* Číslo bodu v pořadí podél obvodu mnohoúhelníka. */
};

static int N;

static struct bod body[MAXN];
static int vzdalenost[MAXN][MAXN];

static int L[MAXN][MAXN][2];            /* Pole L(u,v,x): L[u][v][0] pokud x = u, L[u][v][1] pokud
                                           x = v. */
static int predposledni[MAXN][MAXN][2]; /* Předposlední vrchol cesty, indexováno jako L. */

static int porovnej_smery (const void *a, const void *b)
                                        /* Porovná směry vektoru od bodu a k body[0] */
{                                       /* a od bodu b k body[0]. */
   const struct bod *ba = a;
   const struct bod *bb = b;
   int dxa, dya, dxb, dyb;
```

```c
typedef struct                                  /* struktura intervalu */
{
   OWNER owner;                                 /* vlastník intevalu */
   int start, end;                              /* začátek, konec intervalu */
} INTERVAL;
int compare (const void *a, const void *b)      /* porovnávací procedura pro intervaly do qsortu */
{
   return ( ( (INTERVAL *)a)->start -  ( (INTERVAL *)b)->start);
}
int main (void)
{
   int h_count, v_count, hv_count;              /* počet intervalů hrošíkových, zajícových a obou */
   scanf ("%d%d", &h_count, &v_count);          /* načteme vstup do pole hv za sebe, potom setřídíme */
   hv_count=h_count+v_count;
   INTERVAL hv[hv_count];                       /* množiny H a V dohromady */
   for (int i=0; i<h_count; i++)                /* hrošíkovy intervaly */
     {
       scanf ("%d%d", &hv[i].start, &hv[i].end);
       hv[i].owner=HIPPI;
     }
   for (int i=0; i<v_count; i++)                /* zajícovy intervaly */
     {
       scanf ("%d%d", &hv[h_count+i].start, &hv[h_count+i].end);
       hv[h_count+i].owner=HARE;
     }
   qsort (hv, hv_count, sizeof (INTERVAL), compare);   /* setřídíme */
   /* projdeme a hledáme vhodné intervaly  */
   INTERVAL hippi_min={.owner=NOBODY};          /* hrošíkův kandidát, zpočátku žádný  */
   for (int i=0; i<hv_count; i++)
   {
      /* pokud ještě nemáme kandidáta, pak kadidujeme první hrošíkův interval */
      /* nebo pokud jsme našli lepší interval, změníme kandidáta */
      if ( (hv[i].owner==HIPPI) &&
           ( (hippi_min.owner==NOBODY) ||  (hippi_min.end>hv[i].end)))
      {
        hippi_min.owner=hv[i].owner;            /* nastavime hodnoty nového kandidáta */
        hippi_min.start=hv[i].start;
        hippi_min.end=hv[i].end;
        continue;
      }
      /* v případě zajícova intervalu porovnáme s kandidátem, pokud máme */
      if ( (hv[i].owner==HARE) &&  (hippi_min.owner!=NOBODY) &&
           ( (hippi_min.start<hv[i].start) &&  (hippi_min.end<hv[i].end)))
      {
        printf ("Hledané intervaly existují:\n");
        printf ("Zajíček: [%d,%d]\n", hv[i].start, hv[i].end);
        printf ("Hrošík: [%d,%d]\n", hippi_min.start, hippi_min.end);
        return 0;
      }
   }
   printf ("Zadané podmínky nesplňují žádné dva intervaly.\n");
   return 0;
}
```

Úloha 17-5-3 – Nouze V-dáli-hrocha – program

```
program NouzeJezNaucilaVDaliHrochaRekurzi;

type index=1..16;                       { Rozsah čísel prvků }
var N:index;                            { Počet prvků }
    a:array [index] of index;           { Právě zpracovávaná permutace }
    b:array [index] of index;           { Jeji inverze (a[b[i]]=i) }
```



```c
struct PolozkaFrobniku Frobnik[MaxN];           /* je vysvětlen v popisu řešení */
int Vrchol, Dno;                                /* vrchol a dno zásobníku */
double NejPrumer;
int NejLevy, NejPravy;                          /* nejlepší úsek - levý, pravý konec a průměr */
int main (void) {
    int index;
    printf ("Zadej počet čísel:"); scanf ("%d", &N);
    printf ("Zadej K:"); scanf ("%d:", &K);
    printf ("Zadej čísla:");
    for (index = 0; index < N; index++) scanf ("%d", &Cisla[index]);      /* načteno */
    Vrchol = 0; Dno = 0;                        /* inicializace frobníku */
    AktualniDelka = K;
    AktualniSoucet = 0;
    for (index = 0; index < K; index++) AktualniSoucet += Cisla[index];
    /* Aktuální kus řady je prvních K čísel */
    Levy = 0; Pravy = K-1;                      /* konce zkoumané posloupnosti */
    NejLevy = 0; NejPravy = K-1; NejPrumer = AktualniSoucet /  (double) K;      /* a je to zatím nejlepší úsek */
    for (Pravy = K; Pravy < N; Pravy++) {       /* teď projdeme všechny možné pravé konce */
        AktualniSoucet += Cisla[Pravy];
        AktualniDelka++;                        /* přidáme číslo na konec posloupnosti */
        Frobnik[Vrchol].Delka = 1;
        Frobnik[Vrchol].Soucet = Cisla[Pravy - K];      /* přidáme číslo na frobník */
        Vrchol++;
        /* a teď frobník opravíme */
        while (  (Vrchol - Dno > 1) &&          /* dokud tam jsou alespoň 2 prvky */
           (Frobnik[Vrchol-1].Soucet * Frobnik[Vrchol-2].Delka <
            Frobnik[Vrchol-2].Soucet * Frobnik[Vrchol-1].Delka))    /* a ve frobníku je chyba */
        {
            Vrchol--;
            Frobnik[Vrchol-1].Soucet += Frobnik[Vrchol].Soucet;
            Frobnik[Vrchol-1].Delka += Frobnik[Vrchol].Delka;       /* slučujeme */
        }
        /* nyní jdeme opravit kandidáta na maximum */
        while ( (Vrchol != Dno) &&              /* dokud je něco ve frobníku */
           (Frobnik[Dno].Soucet * AktualniDelka <       /* a dokud vylepšujeme průměr */
            AktualniSoucet * Frobnik[Dno].Delka))
        {
            AktualniDelka -= Frobnik[Dno].Delka;
            Levy += Frobnik[Dno].Delka;
            AktualniSoucet -= Frobnik[Dno].Soucet;
            Dno++;
        }
        if (AktualniSoucet /  (double) AktualniDelka > NejPrumer) {
            /* našli jsme něco lepšího  */
            NejLevy = Levy;
            NejPravy = Pravy;
            NejPrumer =  (double)AktualniSoucet / AktualniDelka;
        }
    }
    printf ("Nejvyšší průměr %g má úsek od %d do %d.", NejPrumer, NejLevy+1, NejPravy+1);
    return 0;
}
```

Úloha 17-5-2 – Ranní hroše – program

```c
#include <stdio.h>
#include <stdlib.h>
typedef enum                                    /* vlastník intervalu */
{
   HARE,                                        /* zajíc */
   HIPPI,                                       /* hroch */
   NOBODY                                       /* neplatný interval */
} OWNER;
```

```c
typedef struct                                  /* struktura intervalu */
{
   OWNER owner;                                 /* vlastník intevalu */
   int start, end;                              /* začátek, konec intervalu */
} INTERVAL;
int compare (const void *a, const void *b)      /* porovnávací procedura pro intervaly do qsortu */
{
   return ( ( (INTERVAL *)a)->start -  ( (INTERVAL *)b)->start);
}
int main (void)
{
   int h_count, v_count, hv_count;              /* počet intervalů hrošíkových, zajícových a obou */
   scanf ("%d%d", &h_count, &v_count);          /* načteme vstup do pole hv za sebe, potom setřídíme */
   hv_count=h_count+v_count;
   INTERVAL hv[hv_count];                       /* množiny H a V dohromady */
   for (int i=0; i<h_count; i++)                /* hrošíkovy intervaly */
     {
       scanf ("%d%d", &hv[i].start, &hv[i].end);
       hv[i].owner=HIPPI;
     }
   for (int i=0; i<v_count; i++)                /* zajícovy intervaly */
     {
       scanf ("%d%d", &hv[h_count+i].start, &hv[h_count+i].end);
       hv[h_count+i].owner=HARE;
     }
   qsort (hv, hv_count, sizeof (INTERVAL), compare);   /* setřídíme */
   /* projdeme a hledáme vhodné intervaly  */
   INTERVAL hippi_min={.owner=NOBODY};          /* hrošíkův kandidát, zpočátku žádný  */
   for (int i=0; i<hv_count; i++)
   {
      /* pokud ještě nemáme kandidáta, pak kadidujeme první hrošíkův interval */
      /* nebo pokud jsme našli lepší interval, změníme kandidáta */
      if ( (hv[i].owner==HIPPI) &&
           ( (hippi_min.owner==NOBODY) ||  (hippi_min.end>hv[i].end)))
      {
        hippi_min.owner=hv[i].owner;            /* nastavime hodnoty nového kandidáta */
        hippi_min.start=hv[i].start;
        hippi_min.end=hv[i].end;
        continue;
      }
      /* v případě zajícova intervalu porovnáme s kandidátem, pokud máme */
      if ( (hv[i].owner==HARE) &&  (hippi_min.owner!=NOBODY) &&
           ( (hippi_min.start<hv[i].start) &&  (hippi_min.end<hv[i].end)))
      {
        printf ("Hledané intervaly existují:\n");
        printf ("Zajíček: [%d,%d]\n", hv[i].start, hv[i].end);
        printf ("Hrošík: [%d,%d]\n", hippi_min.start, hippi_min.end);
        return 0;
      }
   }
   printf ("Zadané podmínky nesplňují žádné dva intervaly.\n");
   return 0;
}
```

Úloha 17-5-3 – Nouze V-dáli-hrocha – program

```
program NouzeJezNaucilaVDaliHrochaRekurzi;

type index=1..16;                       { Rozsah čísel prvků }
var N:index;                            { Počet prvků }
    a:array [index] of index;           { Právě zpracovávaná permutace }
    b:array [index] of index;           { Jeji inverze (a[b[i]]=i) }
```



```c
struct PolozkaFrobniku Frobnik[MaxN];           /* je vysvětlen v popisu řešení */
int Vrchol, Dno;                                /* vrchol a dno zásobníku */
double NejPrumer;
int NejLevy, NejPravy;                          /* nejlepší úsek - levý, pravý konec a průměr */
int main (void) {
    int index;
    printf ("Zadej počet čísel:"); scanf ("%d", &N);
    printf ("Zadej K:"); scanf ("%d:", &K);
    printf ("Zadej čísla:");
    for (index = 0; index < N; index++) scanf ("%d", &Cisla[index]);      /* načteno */
    Vrchol = 0; Dno = 0;                        /* inicializace frobníku */
    AktualniDelka = K;
    AktualniSoucet = 0;
    for (index = 0; index < K; index++) AktualniSoucet += Cisla[index];
    /* Aktuální kus řady je prvních K čísel */
    Levy = 0; Pravy = K-1;                      /* konce zkoumané posloupnosti */
    NejLevy = 0; NejPravy = K-1; NejPrumer = AktualniSoucet /  (double) K;      /* a je to zatím nejlepší úsek */
    for (Pravy = K; Pravy < N; Pravy++) {       /* teď projdeme všechny možné pravé konce */
        AktualniSoucet += Cisla[Pravy];
        AktualniDelka++;                        /* přidáme číslo na konec posloupnosti */
        Frobnik[Vrchol].Delka = 1;
        Frobnik[Vrchol].Soucet = Cisla[Pravy - K];      /* přidáme číslo na frobník */
        Vrchol++;
        /* a teď frobník opravíme */
        while (  (Vrchol - Dno > 1) &&          /* dokud tam jsou alespoň 2 prvky */
           (Frobnik[Vrchol-1].Soucet * Frobnik[Vrchol-2].Delka <
            Frobnik[Vrchol-2].Soucet * Frobnik[Vrchol-1].Delka))    /* a ve frobníku je chyba */
        {
            Vrchol--;
            Frobnik[Vrchol-1].Soucet += Frobnik[Vrchol].Soucet;
            Frobnik[Vrchol-1].Delka += Frobnik[Vrchol].Delka;       /* slučujeme */
        }
        /* nyní jdeme opravit kandidáta na maximum */
        while ( (Vrchol != Dno) &&              /* dokud je něco ve frobníku */
           (Frobnik[Dno].Soucet * AktualniDelka <       /* a dokud vylepšujeme průměr */
            AktualniSoucet * Frobnik[Dno].Delka))
        {
            AktualniDelka -= Frobnik[Dno].Delka;
            Levy += Frobnik[Dno].Delka;
            AktualniSoucet -= Frobnik[Dno].Soucet;
            Dno++;
        }
        if (AktualniSoucet /  (double) AktualniDelka > NejPrumer) {
            /* našli jsme něco lepšího  */
            NejLevy = Levy;
            NejPravy = Pravy;
            NejPrumer =  (double)AktualniSoucet / AktualniDelka;
        }
    }
    printf ("Nejvyšší průměr %g má úsek od %d do %d.", NejPrumer, NejLevy+1, NejPravy+1);
    return 0;
}
```

Úloha 17-5-2 – Ranní hroše – program

```c
#include <stdio.h>
#include <stdlib.h>
typedef enum                                    /* vlastník intervalu */
{
   HARE,                                        /* zajíc */
   HIPPI,                                       /* hroch */
   NOBODY                                       /* neplatný interval */
} OWNER;
```