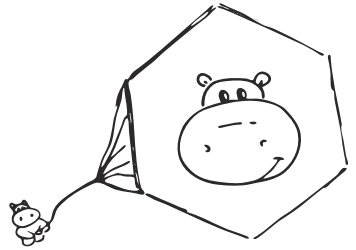
Výsledková listina osmnáctého ročníku KSP po první sérii

		škola	*ročník*	*série*	*1811*	*1812*	*1813*	*1814*	*1815*	*1816*	*suma*	*celkem*
1.	Peter Perešíni	GJGTajov	4	9			10	9	15	11	45,4	45,4
2.	Pavel Klavík	G Chrudim	3	10	4	6	10	11	7	11	39,6	39,6
3.	Michal Pavelčík	G UBrod	3	4		5		11	9	10	39,2	39,2
4.	Miroslav Klimoš	G Lanškr	1	10	5	5	10	9	11	8	38,9	38,9
5.	Josef Pihera	G Strakon	3	6			10	10	9	5	38,6	38,6
6.	Adam Zivner	G UBrod	4	6	3	3	10	11	12		38,4	38,4
7.	Daniel Marek	GZborov	4	6	4	6	10	11		11	38,0	38,0
8.	Roman Smrž	GOhradní	2	6	4	6	10	10		11	37,5	37,5
9.	Michal Čudrnák	G Holešov	4	1	4	2	10	6	10		37,2	37,2
10.	Jiří Maršík	GJKTyla	2	1	3	6	10	7		10	36,5	36,5
11.	Michal Vaner	G Turnov	4	2	4	6	5	11		11	35,7	35,7
12.	Petr Kratochvíl	G SvětláNS	3	10	3	5	7	10		10	32,6	32,6
13.	Kristýna Krejčová	G Tišnov	3	1	2	5		7	4	5	32,1	32,1
14.	Zbyněk Konečný	GKptJaroš	3	8	4	5	10	10	5		32,0	32,0
15.	Jakub Kaplan	GJKTyla	2	6	3	6	5	6		11	31,4	31,4
16.	Petr Onderka	G VKlobou	3	1	4	5		7	1	9	30,7	30,7
17.	Josef Špak	GJírovco	3	3	5	6		9		6	29,9	29,9
18.	Drahoslav Viktorýn	G UBrod	3	1	4	5	5	6			27,5	27,5
19.	Lukáš Lánský	GJKTyla	2	6	2	5	3	10		5	27,3	27,3
20.	Vojtěch Molda	G Vsetín	4	1	5	5	3	8		2	26,9	26,9
21.	Tomáš Zámečník	GJKeplera	3	1	3	1		6	3		22,4	22,4
22. – 23.	Tomáš Herceg	G Třebíč	3	7	4	2	3			10	21,8	21,8
	Jiří Machálek	G Holešov	4	2	3	4	2	5			21,8	21,8
24.	Ján Mikuláš	G Lučenec	4	1	4	6		11			21,7	21,7
25.	Radim Pechal	SPŠ Rožnov	3	1	5	5	5				18,7	18,7
26.	Richard Jedlička	G Vlašim	2	1	3	2		7			17,8	17,8
27.	Ondřej Bílka	G Zlín	4	10	5	3	0	0	8	0	16,7	16,7
28.	Jan Hrnčíř	GFXŠaldy	4	10	3	6		4			13,6	13,6
29.	Ondřej Bouda	GKptJaroš	3	3		3		6			13,1	13,1
30.	Lukáš Moravec	GSRandyJN	2	1	3	2		2			12,7	12,7
31.	David Škorvaga	G Kralupy	3	1		1		8			12,4	12,4
32.	Jakub Pavlík jn.	G Kladno	3	1		2	4				10,9	10,9
33.	Rudolf Rosa	G Kladno	3	1	3	6					10,3	10,3
34.	Matej Kollár	G PBystric	4	1	3	5					10,1	10,1
35.	Tomáš Ehrlich	G Holešov	3	3	3	5					9,8	9,8
36.	Pavel Veselý	G Strakon	1	1	4	3					9,5	9,5
37.	Petr Trňák	G UHradi	3	1	4	2					8,5	8,5
38.	Marián Bazálik	G Košice	4	1	2	3					8,3	8,3
39.	Adam Ráž	GBudějo	3	4	4	2					7,9	7,9
40. – 41.	Ondrej Mikuláš	G Lučenec	3	1		3	1				7,3	7,3
	Jan Musílek	G NBydžov	2	1	2	2					7,3	7,3
42.	Jan Tichý	GDašická	1	1	3	1					6,6	6,6
43.	Vladimír Munzar	SPŠ Rožnov	1	1		5					5,8	5,8
44. – 46.	Jakub Balhar	GJNerudy	3	1	4						4,7	4,7
	Martin Fojtík	GSRandyJN	2	1	4						4,7	4,7
	Jan Kohout	G Roudnice	3	1	4						4,7	4,7
47. – 48.	Jakub Loucký	G Písek	3	1	2						3,5	3,5
	Tomáš Sýkora	G VKlobou	2	1	2						3,5	3,5
49. – 50.	Radim Cajzl	G NMnMor	0	1		1					2,3	2,3
	Jiří Václavík	G Dobříš	4	1		1					2,3	2,3

Milí řešitelé!

Poněkud s předstihem dostáváte do rukou zadání třetí série našeho semináře. S ním dostáváte taktéž opravená řešení série první, takže špinavé a podlé fígly, které se z nich naučíte, můžete použít ještě při řešení série druhé :–)



Termín odeslání Vašich řešení třetí série jest 30. ledna 2006. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
Praha 1, 118 00

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

Zadání třetí série osmnáctého ročníku KSP

18-3-1 Trávník — 4 body

Hrošík Seček si na své zahrádce pěstuje trávník. Čas od času trávník poseče a z výtěžku uspořádá hostinu pro své přátele. Předtím, než trávník poseká, ho musí samozřejmě nechat pěkně narůst. A tak Seček potřebuje napsat program, který mu navrhne ideální den pro posekání trávníku.

Trávník se dá popsat maticí $M \times N$, kde každý prvek matice je celé číslo udávající velikost stébla v trávníku v centimetrech. Toto číslo také udává rychlost růstu stébla v centimetrech za den. Rychlost růstu se nemění. Aby Seček mohl přátele pohostit, potřebuje alespoň K centimetrů stébel trávy.

Napište tedy program, který na vstupu dostane čísla M, N a K a dále matici $M \times N$ nezáporných celých čísel, kde každý prvek odpovídá délce stébla trávníku v daném místě první den od posledního posekání. Tato matice udává zároveň rychlost růstu jednotlivých stébel. Váš program by měl vypsat, kolik dní má Seček ještě počkat s posekáním trávníku.

Příklad: Pro $M = 2$, $N = 3$, $K = 79$ a matici

$$\begin{pmatrix} 1 & 0 & 9 \\ 8 & 9 & 3 \end{pmatrix}$$

Seček počká ještě 2 dny. Matice trávníku pak bude:

$$\begin{pmatrix} 3 & 0 & 27 \\ 24 & 27 & 9 \end{pmatrix}$$

Z tohoto trávníku lze získat 90 centimetrů trávy.

18-3-2 Duel — 5 bodů

Hrošík a prasátko jsou soutěživá zvířátka. Kde mohou, tam se snaží navzájem trumfovat. Jednou objevili zajímavou hru: Na stůl se na papírcích umístí čísla od 1 do 9. Oba hráči se potom střídají v odebírání jednotlivých čísel na svou hromádku. Vyhraje ten hráč, v jehož hromádce se jako prvnímu nachází trojice čísel dávající součet 15.

Hrošík začíná jako první a chce vyhrát. Jenže vůbec neví jak na to… Poraďte hrošíkovi, jak to zařídit, aby když začíná jako první, vždy nad prasátkem vyhrál.

18-3-3 Vrah — 10 bodů

Hugo se právě vrátil ze soustředění KSP, kam byl pozván jako účastník, a protože se nyní ve škole nudil, rozhodl se naučit své spolužáky zábavnou hru „na vraha“.

Do pytlíku se vhodí N papírků se jmény a každý si potom vylosuje jméno své oběti. Cílem hry je nikým jiným neviděn sáhnout zezadu své oběti na krk a sám se nestát obětí. Po úspěšném „zabití“ oběť předává vrahovi svůj papírek se jménem a vrah má nový cíl. Hráč končí, když dostane do ruky svůj vlastní papírek.

Jenže hráči za chvíli zjistili, že pro jistá rozlosování některé dvojice vůbec nedostanou šanci se spolu utkat (například když si hráč již na začátku vylosuje sám sebe, ale i jindy). Naštěstí měli osvícenou paní učitelku, která podporovala kulturní vyžití svých žáků a která se zeptala, kdo s kým chce mít šanci se ve hře potkat. Pak se podívala na jinak tajné rozlosování a některým dvojicím hráčů oznámila, aby si mezi sebou vyměnili své papírky, navíc tak, že počet prohození byl nejmenší možný.

Vymyslete algoritmus a napište program, který to bude dělat za paní učitelku. Na vstupu dostanete počet hráčů N, které si místo jmen očíslujeme čísly 1 až N. Následuje N čísel, kde i-té udává číslo hráče, jehož má zabít hráč číslo i. Poté číslo K a za ním K dvojic čísel určujících, kteří hráči se spolu chtějí potkat. Zdůrazněme, že jeden hráč se může chtít potkat i s více než jedním jiným hráčem. Program by měl odpovědět nejmenším možným počtem pokynů k prohození papírků, aby všech K dvojic zaručeně mělo šanci se ve hře potkat (nemusí se potkat v jedné hře, ale pro každou dvojici musí existovat průběh hry, při kterém se potkají).

Příklad: Pro $N = 6$ hráčů, rozlosování $6, 2, 4, 3, 1, 5$ a $K = 3$ dvojice $(1, 6), (1, 3), (4, 5)$, které se chtějí potkat, stačí jediné prohození papírků, a to mezi lidmi 4 a 6.

18-3-4 Pochoutka pro prasátko — 10 bodů

V lese sousedícím s poklidným rybníčkem našich hrošíků se objevilo hladem šilhající prasátko. Zaslechlo totiž zvěsti o Velké Bukvici, která si tou dobou lebedila v podzemí lesíku. Začalo tedy bez rozmyslu rejdit mezi stromy, leč brzy mu došly síly – byl to už přeci jenom nějaký čas od poslední mňamky. Budete umět prasátku poradit ?

Les si představme jako čtvercovou síť, v jednom políčku prasátko, v jiném bukvice. Aby to nebylo tak jednoduché, prasátko se v lese může pohybovat jen podle určitých pravidel a každé z nich stojí nějaké kladné množství námahy.

Konkrétně – na vstupu dostanete rozměry lesa a pozici prasátka a bukvice spolu s pravidly, podle kterých se prasátko může pohybovat. Každé pravidlo obsahuje trojici čísel $x\ y\ z$, kde x a y je povolený posun v mřížce (o kolik se změní pozice prasátka ve čtvercové síti), zatímco z je úsilí, které prasátko musí vynaložit pro daný přesun. Vaším úkolem je najít a vypsat cestu od prasátka k bukvici. Na své cestě nesmí prasátko opustit lesík. A aby milý čuník hlady nepošel, musí být vynaložené úsilí nejmenší možné.

Příklad: Les má rozměry 6×6, poloha prasátka je $[3, 3]$ a poloha bukvice $[1, 5]$. Pohyb prasátka se řídí třemi pravidly 2 2 3, 1 1 1 a −4 0 5. Potom je pro prasátko nejvýhodnější použít dvakrát pravidlo 2 ($\rightarrow [5, 5]$) a pak jednou pravidlo 3 ($\rightarrow [1, 5]$). Vynaložená námaha je pak $2 \cdot 1 + 5 = 7$.

18-3-5 Hroší lov — 13 bodů

Do hrošího království vtrhlo šílenství – divoké prase začalo zběsile rozrývat rozsáhlé části lesa. Marné bylo domlouvání ostatních obyvatel polesí, kvičící střela zvolna proměňovala hluboké hvozdy v důlní centrum pro těžbu bukvic.

Hrochům nakonec došla trpělivost a rozhodli se, že nezbedné prase polapí, upečou a sní. Teprve nyní prasátko dostalo strach – ale ouha, bylo příliš pozdě. Stádo nasupených hrochů pročesávalo les a dalo se zastavit leda až večerní tmou. Pomůžete prasátku uniknout ještě dříve, než nastane večer?

Podobně jako v předešlém případě je možné les popsat jako čtvercovou síť, po které je pohyb všech zvířat možný pouze podle předepsaných pravidel a nijak jinak.

Na vstupu tedy dostanete rozměry lesa a pozici prasátka a hrochů společně s neohodnocenými pravidly pohybu pro každého z nich (i pro prasátko). Dále dostanete čas t, který zbývá do setmění. Vaším úkolem je najít a vypsat únikovou cestu pro prasátko. Ta se vyznačuje tím, že se prasátko celou dobu pohybuje po bezpečných polích, a buď uteče z lesa ven (dosáhne hranice lesa), nebo uplyne doba t (pak jej totiž hroši přestanou hledat). Bezpečné pole je takové, na které v čase pobytu prasátka nemůže dostat žádný hroch. Ještě dodáváme, že více hrochů smí v jeden moment stoupnout na jedno políčko a čas chápeme jako diskrétní kroky, v nichž každé zvíře musí udělat pohyb podle nějakého svého pravidla (čili nemůže zůstat stát na místě).

Příklad: Les má rozměry 3×3 a do setmění zbývá čas $t = 5$. Prasátko je na souřadnicích $[2, 2]$ a smí se pohybovat podle jediného pravidla $-1\ 0$. Hroši jsou dva – první je na $[3, 2]$ a pohybuje se podle jednoho pravidla $-1\ 0$, druhý je na $[2, 3]$ a pohybuje se podle dvou pravidel $0\ -1$ a $-1\ 0$.

Hledaná cesta pro prasátko je dvakrát použít pravidlo jedna (čili $-1\ 0$), čímž se dostane na $[0, 2]$, což jest ven z lesa.

18-3-6 Komplikovanější komplikátory — 10 bodů

Jedním z problémů, které je často nutné při optimalizacích v kompilátorech řešit, je *analýza toku dat* (*dataflow*). Základní optimalizací, která se pomocí dataflow provádí, je globální propagace konstant. Její úlohou je rozhodnout, které proměnné mají vždy konstantní hodnotu, a nahradit jejich použití touto hodnotou. Například následující kód (v mezijazyce popsaném v první sérii):

```
assign a 0
assign b 1
if (c = 0) 1 2

label 1
assign c (a + b)
goto 3

label 2
assign c 1
assign a 2

label 3
assign x (c + a)
```

Bude po propagaci konstant vypadat takto:

```
assign a 0
assign b 1
if (c = 0) 1 2

label 1
assign c 1
goto 3

label 2
assign c 1
assign a 2

label 3
assign x (1 + a)
```

Povšimněme si, že nestačí jen určit, která proměnná je konstanta, protože to se může v průběhu programu změnit. Například `a` je v prvních třech basic blocích konstanta 1, ale v posledním může mít hodnotu 1 nebo 2. Budeme si proto chtít určit, zda je proměnná konstantní, zvlášť na začátku a na konci každého basic bloku – lokální propagace uvnitř každého basic bloku je jednoduchá, prostě projdeme postupně všechny příkazy a odsimulujeme si je. Pro každou proměnnou `x` a každý basic blok b budeme mít proměnné x_b^+ a x_b^-, které určují stav `x` na začátku a na konci bloku b. Ohodnocení proměnných bude nabývat jedné z následujících hodnot:

- `Top` – tato hodnota znamená, že `x` může být konstantní, ale ještě nevíme, jakou by ta konstanta měla mít hodnotu.
- Nějaké číslo c – to znamená, že `x` je buď vždy rovno c, nebo není konstantní.
- `Bottom` – tato hodnota znamená, že `x` není konstantní.

Tyto hodnoty si uspořádejme tak, že Bottom $<c<$ Top pro libovolnou konstantu c (konstanty jsou navzájem neporovnatelné). Toto uspořádání je důležité pro důkaz konečnosti algoritmu dataflow.

Jestliže nějak určíme hodnoty těchto proměnných na konci všech basic bloků, určit je na začátku libovolného bloku b je snadné, prostě je potřeba slít hodnoty příslušných proměnných na konci předchůdců b. Pravidla pro slévání jsou tato:

- `Bottom` slité s libovolnou hodnotou je `Bottom` – pokud se hodnota může v některém z předchozích bloků měnit, na začátku b nemůže být konstantní.
- `Top` slité s libovolnou hodnotou v je v – `Top` nám říká, že o hodnotě proměnné nic nevíme, takže pokud je na konci některého z předchozích bloků rovna konstantě c, může to být pravda i na začátku b (ale nemůže být vždy rovná libovolné jiné konstantě).
- Slití dvou různých konstant je `Bottom` – taková proměnná nabývá alespoň dvou různých hodnot.
- Slití dvou stejných konstant c je zase c – výsledek pořád může být tato konstanta.

Naopak, pokud bychom znali hodnoty proměnných na začátku basic bloku, hodnoty na konci určíme odsimulováním příkazů v basic bloku s tím, že operace s konstantami vyhodnocujeme. Výsledky operací s `Top` jsou skoro vždy `Top` a operací s `Bottom` zase `Bottom`, až na drobné výjimky – například 0 krát cokoliv je vždy 0, bez ohledu na hodnotu výrazu, který počítáme. Je potřeba si dávat maličko pozor, aby toto vyhodnocování operací bylo monotónní, tj. pokud x op a vyhodnotíme jako a', x op b vyhodnotíme jako b' a $a \le b$, pak i $a' \le b'$. Tedy například pokud chceme, aby `Bottom`$\times 0 = 0$, pak `Bottom`$\times$`Top` může být `Top` nebo 0, ale nic jiného. Tato podmínka je nutná pro zajištění konečnosti níže popsaného algoritmu. Také je vhodné, abychom nevraceli `Top`, pokud alespoň jeden z operandů nebude `Top`. To už není nutné pro konečnost, jen to většinou nedává moc


```
            while (isalnum (znak = vstup[index])) index++;
          } else abort ();
          vstup[index] = 0;
          val[atok++] = strdup (vstup + zac);
          vstup[index] = znak;
          break;
        }
    }
}

char *vyhodnot (char *levy, char znam, char *pravy) {          /* Syntaktická a semantická analýza */
  static unsigned pom = 0;
  char *prom = malloc (20);
  sprintf (prom, "_tmp%d", pom++);
  printf ("assign %s (%s %c %s)\n", prom, levy, znam, pravy);
  return prom;
}

unsigned pos;                                                  /* Aktuální pozice ve vstupu. */

char *cti_term (void);
char *cti_faktor (void);

char *cti_vyraz (void) {
  char *levy, *pravy, znam;
  levy = cti_faktor ();
  while (1) {                                                  /* Čteme zbývající faktory až do konce výrazu či závorky. */
    if (tokeny[pos] == ')') return levy;
    znam = tokeny[pos++];
    pravy = cti_faktor ();
    levy = vyhodnot (levy, znam, pravy);
  }
}

char *cti_faktor (void) {
  char *levy, *pravy, znam;
  levy = cti_term ();
  while (1) {
    znam = tokeny[pos];
    if (znam != '*' && znam != '/') return levy;
    pos++;
    pravy = cti_term ();
    levy = vyhodnot (levy, znam, pravy);
  }
}

char *cti_term (void) {
  char *hodnota, token = tokeny[pos++];
  if (token == '(') {
    hodnota = cti_vyraz ();                                    /* Přeskočit zavírací závorku. */
    pos++;
  } else hodnota = val[pos - 1];
  return hodnota;
}

int main (void) {                                              /* A teď to celé spojíme dohromady */
  char *hodnota;
  fgets (vstup, MAX_DELKA, stdin);
  lex ();
  hodnota = cti_vyraz ();
  printf ("assign _result %s\n", hodnota);
  return 0;
}
```

```
    double sum = 0;
    if (edge->left <= tree[p].left && edge->right >= tree[p].right) {
      if (edge->open && !tree[p].tables++) sum = tree[p].right - tree[p].left - tree[p].covered;
      else if (!edge->open && !--tree[p].tables) {
        sum = tree[p].right - tree[p].left;
        if (!tree[p].leaf) sum -= tree[2*p].covered + tree[2*p+1].covered;
      }
    } else {
      sum = update (2*p, edge) + update (2*p+1, edge);
      if (tree[p].tables) sum = 0;
    }
    if (tree[p].tables) tree[p].covered = tree[p].right - tree[p].left;
    else tree[p].covered = tree[p].leaf ? 0 : tree[2*p].covered + tree[2*p+1].covered;
    return sum;
}

void makeEdge (struct Edge * edge, double main, double left, double right, int open) {
    edge->main = main; edge->open = open;
    edge->left = left; edge->right = right;
}

int main (void) {
    scanf ("%d", &n);
    s = 2 * n;

    int i;
    for (i = 0; i < n; i++) {
      double x, y, w, h;
      scanf ("%lf%lf%lf%lf", &x, &y, &w, &h);
      makeEdge (he + 2 * i , y - h, x, x + w, 1);
      makeEdge (he + 2 * i + 1, y, x, x + w, 0);
      makeEdge (ve + 2 * i , x, y - h, y, 1);
      makeEdge (ve + 2 * i + 1, x + w, y - h, y, 0);
    }

    qsort (he, s, sizeof (struct Edge), &edgecmp);
    qsort (ve, s, sizeof (struct Edge), &edgecmp);

    buildTree (ve, 1, 0, s - 1);
    for (i = 0; i < s; i++) perim += update (1, &he[i]);

    buildTree (he, 1, 0, s - 1);
    for (i = 0; i < s; i++) perim += update (1, &ve[i]);

    printf ("%2.2f\n", perim);
    return 0;
}
```

Úloha 18-1-6 – Kompilované komplikátory – program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_DELKA 1000

typedef char token;                                  /* Typ tokenu – +, -, *, /, (, ), 'a' pro proměnnou, '0' pro číslo. */

char vstup[MAX_DELKA];
char tokeny[MAX_DELKA];
char *val[MAX_DELKA];                                /* Pro proměnné a čísla řetězce, obsahující jejich hodnotu. */

void lex (void) {                                    /* Lexikální analýza */
    unsigned index = 0, atok = 0, zac;
    char znak;

    while (1) {
        znak = vstup[index++];

        while (isspace (znak)) znak = vstup[index++];   /* Přeskočíme mezery. */
        switch (znak) {
          case 0:                                    /* Na konec výrazu si přidáme zavírací závorku, */
            tokeny[atok++] = ')'; return;            /* abychom ho nemuseli ošetřovat speciálně. */
          case '+': case '-': case '*': case '/': case '(': case ')':
            tokeny[atok++] = znak; break;
          default:
            zac = index - 1;
            if (isdigit (znak)) {
                tokeny[atok] = 'a';
                while (isdigit (znak = vstup[index])) index++;
            } else if (isalpha (znak)) {
                tokeny[atok] = '0';
```


smysl – taková operace by tvrdila, že její výsledek je nezávisle na vstupech nějaká neznámá konstanta.

Algoritmus dataflow funguje takto: Na začátku algoritmu nastavíme všechny proměnné na `Top`, kromě těch na začátku prvního bloku, které nastavíme na `Bottom`. Poté budeme opakovat operace popsané v minulých odstavcích tak dlouho, dokud se něco mění. Operace lze provádět v libovolném pořadí, prakticky se to dělá tak, že si udržujeme seznam bloků, pro které se změnily hodnoty proměnných na konci některého z jejich předchůdců. Z něj si odebereme libovolný blok b, slijeme hodnoty z jeho předchůdců, vyhodnotíme si výrazy uvnitř b a v případě, že se ohodnocení proměnných na konci b změnilo, přidáme do seznamu všechny následníky b.

Stav proměnných poté, co dosáhneme ustáleného stavu, je řešením problému. K tomu je potřeba dokázat, že se algoritmus vždy zastaví a že je korektní, tj. že pokud proměnná není konstantní, pak její hodnota na konci bude `Bottom`. Pro důkaz konečnosti si povšimneme, že ohodnocení libovolné proměnné se může změnit nejvýše třikrát – na začátku je `Top`, pak se můžeme nějakou dobu domnívat, že by její hodnota mohla být konstantní, a nakonec se její hodnota může stát `Bottom`, pokud dokážeme, že konstantní není. Protože proměnných, jejichž ohodnocení určujeme, je nejvýše N^2, kde N je délka programu, algoritmus se časem jistě zastaví.

Zajímavější je korektnost. Nejprve si ukážeme, že na konci žádná proměnná nebude mít hodnotu `Top`. Předpokládejme, že tomu tak není a že například x_b^+ je `Top`. Vezměme si libovolnou cestu p z počátku do bloku b a vracejme se z p po b. V každém okamžiku musíme mít alespoň jednu proměnnou ve stavu `Top` – když přecházíme přes hranu CFG, `Top` na jejím konci mohl vzniknout jedině z `Top`u na jejím začátku, případně slitím `Top`ů z ostatních předchůdců. Vyhodnocením příkazu také `Top` mohl vzniknout, jen pokud některý z operandů byl `Top`. Tedy `Top` by musel existovat i na začátku úplně prvního bloku, což ale není možné – ohodnocení všech proměnných na začátku jsme nastavili na `Bottom`.

Předpokládejme nyní, že algoritmus je chybný a nějaké proměnné x_b^+ přiřadí ohodnocení c, i když `x` na začátku basic bloku b může nabývat i jiné hodnoty d. Buď p cesta z počátku do b, která odpovídá výpočtu, jenž způsobí, že `x` je na konci rovno d. Někde na této cestě je první místo, kde se námi nalezené ohodnocení rozchází s tímto výpočtem. Nemůže to být úplně na začátku, neboť tam je ohodnocení všech proměnných `Bottom`, čili o hodnotách proměnných nic netvrdíme. Nemůže to také být na začátku jiného basic bloku, protože pokud o nějaké proměnné tvrdíme, že má hodnotu c, museli jsme to tvrdit i na konci předchozího basic bloku a na hraně CFG se hodnota proměnné nemohla změnit. Čili bychom museli někde mít výraz, jehož operandy mají správné ohodnocení, ale jeho výsledek chybné. To ale nemůže nastat, protože jsme používali pouze korektní pravidla pro vyhodnocování výrazů. Čili všechny proměnné na konci budou ohodnoceny korektně.

S několika drobnými triky se tento algoritmus se dá implementovat s časovou složitostí $\mathcal{O}(N \cdot E)$, kde E je počet hran CFG. Naše implementace je pro lepší čitelnost o něco hloupější a nesnažili jsme se ji příliš optimalizovat, takže její časová složitost je o něco horší, $\mathcal{O}(N^2 \cdot E)$. Můžete ji najít za kuchařkou, před řešením příkladů první série.

Úloha

Jedním z problémů, které se dají pomocí dataflow analýze řešit, je mazání mrtvého kódu, tedy výrazů, jejichž hodnota není k ničemu použita. Výraz je živý, pokud má nějaké efekty, které nejdou smazat (například volání funkce, skok, přiřazení do globální proměnné), nebo pokud je jeho hodnota použita v živém výrazu. Například v následujícím příkladě jsou živá jen přiřazení do i (protože hodnota i je použitá v podmínce skoku) a druhé přiřazení do k (které je použito v návratové hodnotě funkce):

```
assign i 0
assign j 0
assign k~10
assign k~15

label 1
assign i (i + 1)
assign j (j + 1)
if (i < 100) 1 2

label 2
assign result k
```

Vaším úkolem je navrhnout algoritmus pro nalezení mrtvých výrazů založený na dataflow analýze.

Recepty z programátorské kuchařky

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $\mathcal{O}(\log N)$ na přidání či odebrání jednoho prvku a $\mathcal{O}(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k+1$; samozřejmě, pokud je k velké, a tedy např. $2k+1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem N. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $\mathcal{O}(\log N)$.

Jako cvičení si rozmyslete, že v čase $\mathcal{O}(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer;            { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i]) do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
    halda[i]:=x;
    i:=i div 2
  end
end;

procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1]) then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
    i:=j
  end
end;
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $\mathcal{O}(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase [proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky]. Zbytek třídění bohužel nadále zůstává $\mathcal{O}(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;

procedure HeapSort(var A: Pole);
var i, x: integer;
   procedure bublej(m, i: integer); { "zabublání" prvku }
   { m je velikost haldy, i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i); { bublej }
   for i:=N downto 2 do begin { vybírej maximum }
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
   end;
end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu*. Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka v minulé sérii) a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní*. Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme

Úloha 18-1-4 – Dortík – program

```
program dortik;
const max = 1000;
var U: array[1..max] of real;              {seznam úhlů svíček}
    P, C: array[1..max] of integer;        {čísla předchůdců a jejich počet}
    i, j, N, K: integer;
    dif: real;

begin
   read(N);
   read(K);
   for i:=1 to N do begin
      read(U[i]);
      P[i]:=0;
      C[i]:=0
   end;

   {setřiď úhly v U a vyházej duplikáty - vynecháme}
   i:=1; j:=2;
   while j <= N do begin
      dif:=U[j] - U[i] - 360/K;
      if abs(dif) < 0.00001 then begin    {našli jsme vhodného následníka}
         P[j]:=i;
         C[j]:=C[i]+1;
         inc(j)
      end else if dif > 0 then inc(i)
      else inc(j)
   end;
   for i:=1 to N do
      if C[i] = K-1 then begin             {našli jsme konec K-úhelníku}
         j:=i;
         writeln('Svíčkový k-úhelník:');
         repeat writeln(U[j]);
                j:=P[j]
         until j=0
      end
end.
```

Úloha 18-1-5 – Matlalové – program

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

#define MAXN 1000

struct Edge {
   double main, left, right;
   int open; };

struct Node {
   double covered;
   int tables;
   double left, right;
   int leaf; };

int edgecmp (const void * a, const void * b) {
   struct Edge * p = a, * q = b;
   if (p->main != q->main) return p->main - q->main;
   if (p->open != q->open) return p->open ? -1 : 1;
   return 0;
}

int n, s;
double perim;
struct Edge ve[2 * MAXN], he[2 * MAXN];
struct Node tree[8 * MAXN + 1];

void buildTree (struct Edge * f, int p, int l, int r) {
   tree[p].covered = 0; tree[p].tables = 0;
   tree[p].left = f[l].main; tree[p].right = f[r].main;
   tree[p].leaf = 1;

   if (r - l == 1) return;
   int m = (l + r) / 2;
   tree[p].leaf = 0;
   buildTree (f, 2*p, l, m);
   buildTree (f, 2*p+1, m, r);
}

double update (int p, struct Edge * edge) {
   if (edge->right <= tree[p].left || edge->left >= tree[p].right) return 0;
```

```
        if (vstup>0) then begin   { přidává nový šanon do zásobníku}
             akt:=akt+1;
             zasobnik[akt]:=vstup;
        end else begin            { uzavírá sanon}
             if (zasobnik[akt]=-vstup) then akt:=akt-1
             else N:=-1;          { kontroluje, zda uzavírá správný šanon, pokud ne, tak ukončí cyklus a odpoví ne}
        end;
        N:=N-1;
    end;
    if (N<>0) then writeln('ne')
    else writeln('ano');          { jinak je vstup bez chyby}
end.
```

Úloha 18-1-3 – Keřík – program

```
Program Kerik;
const MaxN=100;
var Hrany: array[1..MaxN] of integer;                { reprezentace grafu seznamem sousedů }
    Vrcholy: array[1..MaxN+1] of integer;
    Listky: array[1..MaxN] of integer;               { počet lístků v daném vrcholu }
    NejCesta: array[1..MaxN] of integer;             { nejlepší cesta z listu do daného vrcholu }
    N,max_cesta, max_vrchol, max_syn1, max_syn2: integer;

procedure Ohodnot(v,o:integer);                      { najde vrchol, kde se láme nejvýživnější cesta }
var max1,max2,p_max_syn1,p_max_syn2,i: integer;
begin
  max1:=0; max2:=0; p_max_syn1:=-1; p_max_syn2:=-1;  { hledáme dva nejvýživnější podstromy }
  for i:=Vrcholy[v] to Vrcholy[v+1]-1 do             { ohodnoť všechny podstromy vrcholu v }
    if Hrany[i]<>o then begin                        { nebudeme se vracet zpátky }
      Ohodnot(Hrany[i],v);                           { zjisti výživnost cesty končící v synu }
      if NejCesta[Hrany[i]]>=max1 then begin         { našli jsme zatím nejvýživnějšího syna }
        max2:=max1; max1:=NejCesta[Hrany[i]]; p_max_syn2:=p_max_syn1; p_max_syn1:=Hrany[i];
      end
      else if NejCesta[Hrany[i]]>=max2 then begin    { našli jsme zatím 2. nejvýživnějšího syna }
        max2:=NejCesta[Hrany[i]]; p_max_syn2:=Hrany[i];
      end;
    end;
  NejCesta[v] := Listky[v] + max1; { nejlepší cesta z nějakého listu končící zde má tuto hodnotu }
  if Listky[v] + max1 + max2 > max_cesta then begin  { našli jsme lepší cestu lámající se zde }
    max_cesta:=Listky[v] + max1 + max2;  max_vrchol:=v; max_syn1:=p_max_syn1; max_syn2:=p_max_syn2;
  end;
end;

procedure Vypis(v,o,smer:integer);
var i,max,max_syn:integer;
begin
  if smer = 1 then write(v,' ');                     { jdeme z vrcholu dolů, chceme prefixový výpis }
  max:=-1;
  for i:=Vrcholy[v] to Vrcholy[v+1]-1 do begin       { najdeme toho nejlepšího syna }
    if (Hrany[i]<>o) and (NejCesta[Hrany[i]]>max) then begin
      max:=NejCesta[Hrany[i]]; max_syn:=Hrany[i];
    end;
  end;
  if max<>-1 then Vypis(max_syn,v,smer);             { pokud existuje, vypíšeme jej }
  if smer = -1 then write(v,' ');                    { jdeme zespoda nahoru, chceme postfixový výpis }
end;

begin
  { Zde se načítají data... }
  if N = 0 then begin
    writeln('Chudinka housenka, asi se moc nenažere...');  exit;
  end;
  max_syn1:=-1; max_syn2:=-1;  max_cesta:=-1;
  Ohodnot(1,-1);
  { výpis cesty }
  if max_syn1 <> -1 then Vypis(max_syn1,max_vrchol,-1);
  write(max_vrchol,' ');
  if max_syn2 <> -1 then Vypis(max_syn2,max_vrchol,1);
  writeln;
end.
```


za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováváme v poli. Celý algoritmus má nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme z jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $\mathcal{O}(N)$. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2+M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $\mathcal{O}(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $\mathcal{O}(\log N)$, celkově za všechny hrany tedy $\mathcal{O}(M \log N)$. Z toho vyjde celková časová složitost algoritmu $\mathcal{O}((N+M)\log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \dots v_k v$ takové, že všechny vrcholy $v_0, v_1, \dots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0 v_1 \dots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \dots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \dots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \dots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus funguje je možné snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Dnešní menu Vám servírovali
Dan Kráľ, Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu

```
var N: word;                                  { počet vrcholů }
    vahy: array[1..MAX, 1..MAX] of integer; { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;          { délky zatím nalezených cest, -1 = nekonečno }
    def: array[1..MAX] of boolean;            { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i]) then delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

Ukázková implementace algoritmu propagace konstant popsaného v 18-3-6

```
#include <stdio.h>
#define MAX_BLOKU 100
#define MAX_PROM 100
  /* Typy pro reprezentaci programu. */
unsigned n_prom;                                        /* Proměnné očíslované od 0 do N_PROM - 1. */

struct operand {                                        /* Operand výrazu. Pokud je typ PROM, */
   enum typop {PROM, CISLO} typ;                        /* je hodnota číslo proměnné, */
   unsigned hodnota; };                                 /* jinak to je hodnota čísla. */

struct vyraz{
   char operator;
   struct operand operandy[2]; };

  /* Pro ASSIGN je v data[0] identifikátor proměnné, do níž se přiřazuje, a ve vyraz přiřazovaný výraz. */
  /* Pro LABEL je v data[0] číslo labelu. */
  /* Pro IF je ve vyraz podmínka, v data[0] label, kam se skáče, je-li splněna, v data[1] kam se skáče, pokud splněna není. */
  /* Pro GOTO je v data[0] label, na který skáče. */
struct prikaz {
   enum prikazy {ASSIGN, LABEL, IF, GOTO} prikaz;
   struct vyraz vyraz;
   unsigned data[2];
   struct prikaz *dalsi, *predchozi;                    /* Příkazy jsou uloženy v seznamu. */

struct hrana {                                          /* CFG. */
   struct basic_block *z, *k;                           /* Hrana z bloku Z do bloku K. */
   struct hrana *nasl_dalsi, *pred_dalsi; };            /* Hrany ve dvou seznamech: následníci a předchůdci bloku. */

struct basic_block {
   unsigned index;                                      /* Číslo bloku, od 0 do N_BLOKU. */
   struct hrana *pred, *nasl;                           /* Seznam předchůdců a následníků. */
   struct prikaz *prvni, *posledni; };                  /* Příkazy v bloku. */

unsigned n_bloku;                                       /* CFG se skládá z N_BLOKU bloků. Počáteční blok má číslo 0. */
struct basic_block cfg[MAX_BLOKU];

struct hodnota {                                        /* Hodnoty pro propagaci konstant. */
   enum hod { TOP, KONST, BOTTOM } hod;
   unsigned konstanta; };

struct hodnota ohodn_zac[MAX_BLOKU][MAX_PROM];          /* Ohodnocení proměnných na začátku a na konci bloku. */
struct hodnota ohodn_kon[MAX_BLOKU][MAX_PROM];

void slij_hodnoty (struct hodnota *h, struct hodnota h1) {   /* Slije H a H1 do H. */
   if (h->hod == BOTTOM || h1.hod == TOP) return;
   if (h->hod == TOP || h1.hod == BOTTOM) { *h = h1; return; }
   if (h->konstanta != h1.konstanta) h->hod = BOTTOM;
}
void slij (struct basic_block *bb) {                    /* Slití hodnot na začátku basic bloku */
   struct hrana *p;                                     /* z hodnot na koncích předcházejích bloků. */
   struct basic_block *pred;
   unsigned i;

   for (i = 0; i < n_prom; i++) ohodn_zac[bb->index][i].hod = TOP;
   for (p = bb->pred; p; p = p->pred_dalsi) {
     pred = p->z;
     for (i = 0; i < n_prom; i++) slij_hodnoty (&ohodn_zac[bb->index][i], ohodn_kon[pred->index][i]);
   }
}
void vyhodnot_vyraz (struct basic_block *bb, struct prikaz *prik) {   /* Odsimuluje přiřazení PRIK. */
   unsigned vysl = prik->data[0];
   struct hodnota hod[2], vys;
   unsigned i;
   struct operand *op;

   for (i = 0; i < 2; i++) {
     op = &prik->vyraz.operandy[i];
     if (op->typ == PROM) hod[i] = ohodn_kon[bb->index][op->hodnota];
     else { hod[i].hod = KONST; hod[i].konstanta = op->hodnota; }
   }

   /* Pokud je alespon jeden z operandů TOP, je bezpečné vrátit TOP (byť v praxi je výhodnější konvergovat k BOTTOMu rychleji). */
   if (hod[0].hod == TOP || hod[1].hod == TOP) vys.hod = TOP;
   else {
     vys.hod = KONST;
     switch (prik->vyraz.operator) {
```


Dojde-li k chybě v syntaktické analýze (například proto, že po sobě následují dva operátory a podobně), příslušná funkce si domyslí nějaký token, který se jí hodí, nebo ten aktualní zahodí, podle toho, co dává víc smysl.

Co se týče vylepšení lexikální analýzy, všimneme si, že syntaktická analýza čte vstup postupně a nikdy se nevrací. Je tedy zbytečné si vstup rozložený na tokeny pamatovat celý. Lexikální analýzu proto budeme realizovat funkcí, která ze vstupu načte a vrátí další token (`dalsi_token`), a sémantická analýza si ji bude volat podle potřeby. Ve skutečnosti se občas hodí se ve vstupu o jeden token vrátit – například když `cti_faktor` narazí na plus, ukončí se, ale toto plus by měla zpracovat funkce `cti_vyraz`. Proto `cti_faktor` nejdřív vrátí plus zpět do vstupu. K tomu slouží funkce `vrat_token`.

Dalším drobným trikem je, že si udržujeme tabulku identifikátorů `hodnota_na_promennou`, a když načteme dvakrát identifikátor se stejným jménem, vrátíme místo něj jeho pořadí v tabulce, takže nemusíme nikde dál testovat, zda jsou dva identifikátory stejné (`promenna_na_hodnotu` je vlastně hešovací tabulka, která nám umožní záznamy v tabulce `hodnota_na_promennou` hledat rychle – víc k hešování viz kuchařka druhé série sedmnáctého ročníku).

Syntaktickou analýzu si oddělíme od sémantické. Syntaktická analýza už nebude přímo generovat mezikód, ale místo toho každému zpracovanému podvýrazu přiřadí nějaké číslo. Tato čísla by měla být taková, že výrazy s různou hodnotou dostanou vždy různé číslo, zatímco výrazy se stejnou hodnotou dostanou stejné číslo – například výrazy `x + y` a `x - y` dostanou různá čísla, protože jejich hodnoty se mohou lišit, zatímco výrazům `x - x` a `0` by mělo být přiřazeno stejné číslo. To děláme tak, že si udržujeme tabulku hodnot výrazů, které jsme již viděli (`hodnota_na_vyraz` a `vyraz_na_hodnotu`, opět používáme hešování), v níž si pamatujeme, jak se každé číslo hodnoty spočítá. Pokud narazíme na výraz, který už v tabulce je, vrátíme jeho číslo, jinak mu přidělíme nové číslo a přidáme ho do tabulky. Podrobnější vysvětlení tohoto postupu viz řešení úlohy 17-2-1. Snadno nahlédneme, že toto je v podstatě jen jiná reprezentace syntaktického stromu – čísla hodnot odpovídají vrcholům, a abychom určili syny vrcholu, podíváme se do tabulky `hodnota_na_vyraz`. Číslování hodnot nám ale umožní zajistit, že stejnou hodnotu nebudeme počítat dvakrát – když na ni narazíme podruhé, budeme místo ní používat pomocnou proměnnou, do níž jsme ji poprvé spočítali.

Navíc se nám toto očíslování hodnot hodí při zjednodušování výrazů. Budeme chtít rovnou vyhodnocovat konstantní výrazy a také aplikovat jednoduché algebraické identity typu `x - x = 0`. Abychom mohli tuto optimalizaci provést, je potřeba zjistit, zda oba operandy mínusu jsou stejné. Vzhledem k tomu, jak si výrazy reprezentujeme, stačí porovnat čísla jejich hodnot, není potřeba procházet stromy výrazů a ověřovat, zda si odpovídají (to by nám mohlo zhoršit časovou složitost na kvadratickou). Zjednodušování výrazů provádíme tak, že kdykoliv vytváříme vrchol stromu, který odpovídá nějakému operátoru, podíváme se na jeho operandy a určíme, zda ho můžeme nějak zjednodušit. Toto provádí funkce `postav_strom`. Například pokud vyhodnocujeme výraz $(x+1)+2$, `postav_strom` dostane plus, jehož parametry mají čísla hodnot h_1 a h_2, a z tabulek zjistíme, že h_2 je ve skutečnosti konstanta 1, a že h_1 je součet hodnot h_3 a h_4, kde h_4 je konstanta 2. Konstanty sečteme, dostaneme 3 a zjistíme si, že číslo hodnoty pro 3 je h_5. Zjednodušený výraz tedy bude h_3+h_5, což odpovídá $x+3$. Tomuto výrazu přidělíme nové číslo hodnoty h_6, dáme ho do tabulek a h_6 vrátíme.

Jednou z komplikací, které se v tomto řešení vyhýbáme, ale prakticky je nutné se jí zabývat, je provedení vedlejších účinků zjednodušovaných výrazů. Například máme-li výraz `funkce () * 0`, jeho hodnota je vždy 0, ale přesto je nutné funkci zavolat. Prakticky tedy nestačí pouze vracet hodnotu zjednodušeného výrazu, ale je nutné zajistit, aby se také provedly tyto vedlejší akce. V našem případě jediný takový problém je dělení 0 – například výraz `x/y - x/y` by mohl způsobit chybu, pokud $y = 0$, ale zjednodušený výraz 0 chybu způsobit nemůže. Tento problém pro jednoduchost řešit nebudeme – konec konců, v platném programu se dělení nulou vyskytnout nesmí (až na výjimky).

Vedlejší účinky by navíc mohly měnit hodnoty proměnných, které se ve výrazu používají. Například při vyhodnocování výrazů v C je nutné při dosažení *sequence pointu* (což je místo, na kterém je zaručeno, že se vedlejší účinky vykonají) upravit čísla hodnot ovlivněných proměnných.

Poslední fází je sémantická analýza, tj. expanze do mezikódu. V ní vypíšeme výrazy nutné pro spočtení hodnoty, která odpovídá číslu hodnoty celého výrazu. Podíváme se tedy do tabulek, jak jsme toto číslo dostali, rekurzivně vyhodnotíme čísla hodnot podvýrazů a vypíšeme příslušnou operaci, která z nich spočte výsledek. Abychom nepočítali nějaký výraz dvakrát, u každého čísla hodnoty si pamatujeme, zda jsme ho už počítali, a když ho potřebujeme podruhé, použijeme místo něj příslušnou pomocnou proměnnou. Přidělování jmen pomocným proměnným řešíme jednoduše, k prefixu `tmp` připojíme číslo hodnoty.

Je snadné si rozmyslet, že všechna tato rozšíření stále fungují v lineárním čase.

Zdeněk Dvořák

Úloha 18-1-2 – Úřad – program

```
program Kontrolor;
const MAX_N=1000;
var K,N:integer;
    vstup,akt:integer;
    zasobnik:array[1..MAX_N] of integer;

begin
     writeln('zadej počet barev: ');   readln(K);
     writeln('zadej počet šanonů: ');  readln(N);
     akt:=1;
     zasobnik[1]:=0;                {pomocná hodnota nám zajistí, že zásobník nikdy nebude prázdný}
     if (N mod 2=1) then N:=-1;     {ošetření lichého počtu šanonů}
     while(N>0)do begin
          writeln('zadej barvu šanonu: ');
          readln(vstup);
```

bloku vstoupili do oblasti nebo ji opustili, a proto délku tohoto bloku přičteme k obvodu. Říkáme, že blok je pokryt, pokud má $c_i > 0$.

Popsaný algoritmus spočte horizontální obvod jako součet délek bloků při změně jejich pokrytí. Jak dlouho mu to bude trvat? Třídení nám bude trvat čas $\mathcal{O}(n \log n)$. Pak pro každou hranu aktualizujeme $\mathcal{O}(n)$ bloků. Celkem tedy $\mathcal{O}(n^2)$. Paměti spotřebujeme pouze lineárně – $\mathcal{O}(n)$.

Pokud si budeme intervaly uchovávat trochu chytřeji, dá se časová složitost trochu zlepšit. Použijeme k tomu tzv. *intervalový strom*. Intervalový strom je binární vyvážený strom, který v našem případě vypadá následovně. Každému uzlu přiřadíme interval. Listům přiřadíme naše bloky, jak je známe, seřazené zleva doprava. Vnitřnímu uzlu přiřadíme interval daný sjednocením intervalů jeho synů. Kořen stromu má tedy přiřazen celý interval. Protože strom je vyvážený a má $\mathcal{O}(n)$ listů, je jeho hloubka $\mathcal{O}(\log n)$. Každý uzel má tyto vlastnosti:

- `left` – začátek intervalu
- `right` – konec intervalu
- `covered` – pokrytí intervalu
- `tables` – počet stolů, které ho úplně překrývají

Potřebujeme umět přidat do stromu a odebrat z něj interval v lepším čase než $\mathcal{O}(n)$. Jak asi u stromu očekáváte, bude to $\mathcal{O}(\log n)$. Protože se interval skládá až z $\mathcal{O}(n)$ bloků, nemůžeme si dovolit při jeho přidání zaktualizovat všechny bloky, z kterých se skládá. Rozmyslíme si, že se každý interval dá rozložit do $\mathcal{O}(\log n)$ intervalů reprezentovaných uzly našeho stromu. Tento rozklad najdeme podobně jako při přidávání intervalu, označme ho I. Budeme ho realizovat rekurzivní funkcí `find`. Začneme v kořeni stromu. Aktuální uzel označme u, interval uzlu uznačme I_u. Pokud I a I_u mají prázdný průnik, skončíme. Pokud I_u je podinterval I, I_u je jeden z hledaných intervalů. Jinak se částečně překrývají a zavoláme funkci `find` na oba syny. Průběh volání funkce bude vypadat takto. Z kořene projdeme po synech až k uzlu, kde interval I zasahuje do obou synů. Zde se průchod rozdělí na dvě větve. Jedna jde po levé straně intervalu, druhá po pravé. Všechny intervaly mezi nimi spadají zcela do intervalu I. Zkuste si nakreslit obrázek. Protože při zavolání funkce `find` na kořen projdeme maximálně dvě cesty z kořene k listům, je časová složitost této operace $\mathcal{O}(\log n)$.

Vkládání a odebírání intervalu bude podobné funkci `find`. V každém z uzlu, na který se interval rozložil, aktualizujeme vlastnosti `tables` a `covered`. Vlastnost `tables` je počet stolů, jejichž rozklad intervalu obsahuje tento uzel. Pokrytí intervalu, `covered`, udává v reálných souřadnicích, kolik z intervalu je pokryto stoly. Pokrytí je větší než nula i v případě, kdy `tables` je rovno nula a některé intervaly v jeho synech jsou pokryty. Tyto vlastnoti se vztahují pouze k podstromům daného uzlu, nikoli k jeho rodičům. Rozmyslete si, že je dokážeme při vkládání a odebíraní intervalu aktualizovat. Funkce pro vkládání a odebírání bude vracet změnu pokrytí v daném podstromě. Proto pokud je interval uzlu pod stolem a uvnitř jeho podstromu se změní pokrytí, tento uzel ho znuluje. Jinak se propaguje až do kořene a nakonec je celková změna pokrytí přičtena k počítanému obvodu.

Protože přidání a odebrání intervalu nám trvá čas $\mathcal{O}(\log n)$ a stálé máme celkem $2n$ hran, celkový čas algoritmu je $\mathcal{O}(n \log n)$. Paměťová složitost zůstala lineární.

Petr Škoda

18-1-6 Kompilované komplikátory

Jak si mnoho řešitelů uvědomilo, tuto úlohu šlo řešit i podstatně přímočařeji než v textu seriálu naznačeným postupem, například konstrukci syntaktického stromu lze přeskočit a generovat rovnou požadovaný mezikód. My si nejprve implementujeme jedno takové jednoduché řešení a poté si ukážeme, jak si ho vylepšit – kvůli tomu již bude nutné se přidržet postupu popsaného v seriálu. Abychom šetřili naše lesy a nervy méně otrlých řešitelů, asi 700-řádkový program k tomuto rozšířenému řešení zde netiskneme. Můžete ho nalézt na adrese *http://ksp.mff.cuni.cz/tasks/18/ksp1816b.c*.

Lexikální analýza je v našem případě triviální – načteme znak ze vstupu a rozhodneme, zda je to jeden z operátorů. Je-li tomu tak, rovnou vrátíme jemu odpovídající token. Další možností je začátek jména identifikátoru nebo číslo, pak načteme jeho zbytek.

V jednoduchém řešení bude sémantická analýza spojena se syntaktickou a místo stavby syntaktického stromu budeme rovnou vypisovat výpočet v mezikódu. Pro syntaktickou analýzu se prakticky se používají dva hlavní přístupy. Jeden z nich je zkonstruovat si zásobníkový automat, který rozpoznává danou gramatiku. Tento postup je vysvětlen například v řešení úlohy 9-3-3. Druhý přístup je složit analyzátor ze vzájemně rekurzivních funkcí, které odpovídají symbolům gramatiky. Implementace tohoto postupu bývá pro člověka o něco čitelnější a dají se v ní lépe ošetřovat chyby a jiné speciální případy. My se přidržíme druhého postupu. Syntaktickou analýzu budou zajišťovat funkce `cti_vyraz`, která zpracovává celý výraz nebo jeho podvýraz uvnitř závorkek, `cti_faktor`, která zpracovává podvýraz, jehož operátory jsou násobení či dělení, a `cti_term`, která vyhodnotí podvýraz tvořený identifikátorem, číslem, nebo uzávorkovaným výrazem. Každá z těchto funkcí přečte co nejdelší kus kódu, který jí odpovídá, vypíše příkazy nutné pro jeho vyhodnocení a vrátí identifikátor proměnné, v níž je uložena jeho hodnota. Například funkce `cti_vyraz` pomocí funkce `cti_faktor` čte postupně kusy výrazu oddělené znaménky plus a mínus, dokud nenarazí na konec výrazu či závorky, a sčítá či odčítá odpovídající hodnoty. Funkce `cti_faktor` se chová podobně, volá `cti_term` a kontroluje, zda po nich následuje krát či děleno. Funkce `cti_term` se podívá, zda následuje proměnná či číslo (pak ho rovnou vrátí), nebo otevírací závorka, na jejíž vnitřek zavolá `cti_vyraz`. Každá z těchto funkcí také posune ukazatel ve vstupu na první znak, který nezpracovala.

Celý tento postup lze realizovat s paměťovou i časovou složitostí lineární v délce zadaného výrazu. Pro časovou složitost stačí nahlédnout, že je omezená počtem volání funkce `cti_term`, a ta vždy načte alespoň jeden token ze vstupu.

Nyní si popíšeme možná vylepšení tohoto postupu. U každé fáze si řekneme něco k tomu, co jde udělat lépe. Mimo jiné se budeme zabývat zotavením se z chyb. Pokud se v kódu programu vyskytne chyba (v našem případě třeba nespárované závorky), je poněkud nešikovné s překladem okamžitě skončit, například proto, že už se nevypíšou hlášení pro další chyby. Nejde tedy opravit všechny chyby naráz a je nutné po každé z nich program znovu kompilovat, což může být dost pomalé. Je zřejmě lepší se s chybou nějak vypořádat a pokračovat v překladu.

Smysluplné ošetření chyb při lexikální analýze je obtížné, protože v této fázi toho o vstupu mnoho nevíme. Chyby se proto řeší prostým zahozením nerozpoznaných znaků.


```
    case '+': if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM) vys.hod = BOTTOM;
              else vys.konstanta = hod[0].konstanta + hod[1].konstanta; break;
    case '-': if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM) vys.hod = BOTTOM;
              else vys.konstanta = hod[0].konstanta - hod[1].konstanta; break;
    case '*': if ( (hod[0].hod == KONST && hod[0].konstanta == 0)
                 || (hod[1].hod == KONST && hod[1].konstanta == 0)) vys.konstanta = 0;
              else if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM) vys.hod = BOTTOM;
              else vys.konstanta = hod[0].konstanta * hod[1].konstanta; break;
    case '/': if (hod[0].hod == KONST && hod[0].konstanta == 0) vys.konstanta = 0;
              else if (hod[0].hod == BOTTOM || hod[1].hod == BOTTOM) vys.hod = BOTTOM;
              else vys.konstanta = hod[0].konstanta / hod[1].konstanta; break;
    default: vys.hod = BOTTOM;
    }
  }
  ohodn_kon[bb->index][vysl] = vys;
}

int vyhodnot (struct basic_block *bb) {                    /* Vyhodnotí výrazy v bloku BB, */
  struct prikaz *prik;                                     /* vrátí 1 pokud se změnilo ohodnocení na jeho konci. */
  unsigned i;
  struct hodnota ohodn_stare[MAX_PROM];

  for (i = 0; i < n_prom; i++) {                           /* Uložíme staré ohodnocení, */
    ohodn_stare[i] = ohodn_kon[bb->index][i];              /* nové nainicializujeme na hodnoty na začátku. */
    ohodn_kon[bb->index][i] = ohodn_zac[bb->index][i];
  }
  for (prik = bb->prvni; prik; prik = prik->dalsi) if (prik->prikaz == ASSIGN) vyhodnot_vyraz (bb, prik);

  for (i = 0; i < n_prom; i++)                             /* Porovnáme se starým ohodnocením. */
    if (ohodn_stare[i].hod != ohodn_kon[bb->index][i].hod) return 1;

  return 0;
}

void cprop_dataflow (void) {                               /* Vyplní ohodnocení pro propagaci konstant. */
  unsigned i, b;

  struct basic_block *zmenene[MAX_BLOKU], *bb;             /* Zmenene je seznam bloků, jejichž ohodnocení se změnilo a */
  unsigned pocet_zmenenych;                                /* je třeba ho přepočítat. Chováme se k němu pro jednoduchost */
  char je_zmeneny[MAX_BLOKU];                              /* jako k zásobníku, i když bývá lepší používat jiné pořadí. */
  struct hrana *n;

  for (b = 0; b < n_bloku; b++)
    for (i = 0; i < n_prom; i++)
      ohodn_zac[b][i].hod = ohodn_kon[b][i].hod = TOP;
  for (i = 0; i < n_prom; i++) ohodn_zac[0][i].hod = BOTTOM;
  for (b = 0; b < n_bloku; b++) je_zmeneny[b] = 0;         /* Na začátku je změněný jen počáteční blok. */
  je_zmeneny[0] = 1;
  zmenene[0] = &cfg[0];
  pocet_zmenenych = 1;

  while (pocet_zmenenych > 0) {
    bb = zmenene[--pocet_zmenenych];
    je_zmeneny[bb->index] = 0;
    slij (bb);
    if (!vyhodnot (bb)) continue;

    /* Hodnoty na konci se změnily, je potřeba vložit následníky BB do seznamu, pokud už v něm nejsou. */
    for (n = bb->nasl; n; n = n->nasl_dalsi) if (!je_zmeneny[n->k->index]) {
      je_zmeneny[n->k->index] = 1;
      zmenene[pocet_zmenenych++] = n->k;
    }
  }
}
```

Vzorová řešení první série osmnáctého ročníku KSP

18-1-1 Dimenze X

Roboti se v naprosté většině došlých řešení šťastně shledali. Ne vždy však po setkání došlo ke zničení Dimenze X, neboť občas jim hledání trvalo tak dlouho, že se jejich zásoba plutonia rozpadla až na bismut, který se k výbuchu již tolik neměl.

Došlá řešení lze rozdělit na dvě zhruba stejně velké skupiny. V první prohledávali roboti své okolí do stále se zvětšující vzdálenosti a když narazili na hromadu šrotu toho druhého, tak na něj buď počkali, nebo v lepším případě mu šli naproti. Myšlenka je to správná, bohužel implementace pokulhávala. Většina zvětšovala amplitudu prohledávání o konstantu, což dává složitost $\mathcal{O}(N^2)$. Pouze menšina amplitudu zvětšovala konstantněkrát, čehož výsledkem je pro Dimenzi X nepříznivější složitost $\mathcal{O}(N)$.

Druhá skupina řešila úkol tak, že vyslala roboty nižší rychlostí libovolným směrem. Jeden z robotů tak zákonitě musel narazit na hromadu druhého z robotů, což pochopil jako povel zrychlit na plnou rychlost a dohnat tak robota, který

se nerušeně vzdaloval stále nízkou rychlostí. Myšlenka je to opět správná, bohužel i tentokrát nebyla implementace vždy v pořádku. Většina totiž řešila snížení rychlosti tak, že robot dělal mezi posuny pauzu. V zadání však bylo, že robot umí udělat v jednom kroku pouze posun o 1 metr na sever nebo na jih (o zastavení tam řeč nebyla). Řešitelné to však je. Například udělat dva posuny jedním směrem a jeden posun zpět, atd. Obecně je ale jakékoliv řešení založené na této myšlence v čase $\mathcal{O}(N)$.

Všechny algoritmy si vystačily s maximálně 3 proměnnými, což dává velmi příznivou paměťovou složitost $\mathcal{O}(1)$. Bohužel paměťový obvod nemusel mít poruchou omezenou pouze kapacitu, ale i spolehlivost uchovávání informací a proto byla nejcennější řešení ta, které si kromě ukazatele pozice v programu nemusela pamatovat vůbec nic.

Abych byl konkrétní, tak uvádím příklad možného postupu, který pracuje v čase $\mathcal{O}(N)$, nepotřebuje žádné pomocné proměnné a za které bylo možné získat plný počet bodů:

```
{ posun na sever nižší rychlostí }
{  dokud nenajdu druhou hromadu  }
repeat
  Posun_Na_Sever; Posun_Na_Sever; Posun_Na_Jih;
until Stojím_Na_Hromadě;

{ narazil jsem na hromadu druhého }
{  robota => zrychlím a doženu ho }
while true do
  Posun_Na_Sever;
```

Zbyněk Falt

18-1-2 Úřad

U mnoha řešitelů byla znát vcelku oprávněná zášť vůči úřednímu šimlovi, jež mnohdy vyústila v zákeřné chyby, které měly znemožnit další bujení byrokracie. Ve snaze znemožniti kontrolorovi jejich odhalení je důvtipně skrývali v moři rekurze a záplavě cyklů. Někteří z vás se dokonce uchýlili k zákeřné fintě známé jako vypouštění komentářů a popisů řešení. Kvílení kontrolorů a otázky proč to děláš? a jak to funguje?, budiž jim odměnou za dobře odvedenou práci. Ale zpět k úloze.

Jak většina z vás správně poznala, použijeme zásobník. Pokud ze vstupu načteme otevírací závorku, pak ji uložíme na zásobník. Na vrcholu zásobníku tak bude vždy poslední nespárovaná otevírací závorka O. Pokud načtu zavírací závorku Z, mohou nastat tyto tři situace:

1) Barvy si odpovídají. Pak je vše v pořádku. Navíc je tato závorka použitá a už nikdy ji nebudu potřebovat. Můžeme ji tedy zezásobníku odebrat.
2) Barvy si neodpovídají. Takové uzávorkování nemůže být správné, protože O může být uzavřena až za Z. Zároveň ale i otevírací závorka pro Z se může nacházet jedině před O. Tím by se nám zkřížily dva šanony a to je chybné.
3) V zásobníku nic není. Velmi často opomíjená možnost nám říká, že nalevo od uzavírací závorky není žádná nepoužitá otevírací závorka. V tomto případě samozřejmě končíme s výpisem „ne“.

Pokud program dojde na konec vstupu, tak zjistí, zda je zásobník prázdný. Pokud ano, tak to znamená, že všechny otevírací závorky byly zavřeny a uzávorkování je korektní. V opačném případě jsme něco neuzavřeli a končíme s „ne“. Časová složitost je stejně jako paměťová $\mathcal{O}(N)$, s každým prvkem provedeme konstantní počet operací a prvků v pomocném poli je nejvýše $N/2$.

Někteří z vás zbytečně načítali vstupní data do pole, i když to nebylo nutné. Naštěstí pro ně si tím asymptotickou složitost nezhoršili. Za chybějící nebo špatné popisy, odhady složitosti a hlavně zdůvodnění správnosti se dle míry provinění daly dohromady ztratit až dva body. Pokud se v programu vyskytla chyba, která způsobila nefunkčnost programu, tak jsem strhával dle nefunkčnosti až 4 body, podobně pro pomalá (typicky $\mathcal{O}(N^2)$) řešení docházelo ke srážkám úměrným pomalosti.

Jan Bulánek

18-1-3 Keřík

Velká lístečková žranice skončila ve většině případů dle libosti nenasytné píďalky. Kdo už jednou vymyslel ten správný výpočet pohybu housenky, neudělal už většinou žádnou chybu, takže bodové zisky od spokojeně nadládbnuté žížalky byly hojné. Zbytek řešitelů se více či méně úspěšně snažil spouštět z každého význačného bodu prohledávání do hloubky, případně generovat všechny možné dvojice význačných bodů (dále je budeme v souladu s grafovou teorií nazývat vrcholy), za což zaplatil zvýšením časové složitosti. Taková řešení pak získala maximálně 5 bodů. Jak tedy nakrmit žravou housenku v lineárním čase?

Nejprve je třeba si ujasnit, jaký typ grafu náš keřík vlastně je. Víme, že se jedná o obecný strom s neorientovanými hranami. Někteří předpokládali, že máme dán zakořeněný strom s hranami orientovanými a snažili se hledat vrchol, do nějž nevede žádná hrana, aby z něj pak vesele začali počítat. Nikdo ale neříkal, že graf má hrany orientované, právě naopak, logicky hrany musí být obousměrné. Navíc nám nikdo nedal záruku, že takový vrchol, ze kterého nevede žádná hrana, je právě jeden!

Mějme tedy náš obecný neorientovaný strom. Tento strom si „zakořeníme“, čili jeden vrchol prohlásíme za kořen, jeho sousedy za jeho syny, jejich sousedy za syny synů, atd. Uvidíme, že náš algoritmus tím nijak nepoškodíme, děláme to jenom proto, abychom si vše mohli lépe představit. Navíc uvidíme, že kořenem může být libovolný vrchol.

Jistě mi budete věřit, že nejvýživnější cesta musí začínat a končit v listu, to jest ve vrcholu stupně jedna. Kdyby tomu tak nebylo a cesta by končila v nějakém vrcholu stupně > 1, pak bychom cestu mohli z tohoto vrcholu ještě prodloužit až do nějakého listu.

Dobře, ale pak cesta musí vypadat tak, že začíná v nějakém listu, pak míří nahoru směrem ke kořeni, dorazí do nějakého vrcholu, ve kterém se jakoby láme a míří zase dolů až do nějakého jiného listu a skončí.

Představme si, že se nacházíme v nějakém vrcholu v a chtěli bychom vědět, jaká nejlepší cesta přes něj prochází, čili láme se v něm. Hodilo by se nám, kdybychom u každého jeho syna měli spočítáno, jaká nejlepší cesta vede od nějakého listu v jeho podstromě až do něj. Pak bychom si ze všech synů vrcholu v vybrali ty dva nejvýhodnější, sečetli jejich hodnoty, přičetli bychom počet lístečků ve vrcholu v a kýženou výživnost cesty lámající se ve vrcholu v bychom znali. Je pro nás těžké spočítat tyto hodnoty u všech synů? Kdepak, uděláme to úplně stejným způsobem, tedy rekurzí.

Algoritmus tedy funguje tak, že pro každičký vrchol ve stromě spočítáme dvě hodnoty: Jak výhodná cesta se v něm láme (tuto informaci získáme rekurzivním voláním synů) a jaká nejlepší cesta vedoucí z nějakého listu v některém z jeho podstromů v něm končí (tuto informaci předáme nahoru jeho otci).

Nakonec stačí projít všechny vrcholy a najít ten s nejvyšší hodnotou lámající se cesty a z něj pak spustit výpis na obě strany. Samozřejmě, že jedna strana nemusí existovat, to například když má strom tvar cesty (např. $1 \to 2 \to 3 \to 4$).

Jak rychlá je tato rekurze? Do každého vrcholu přijdu jenom jednou, tedy $\mathcal{O}(N)$, a na každou hranu se podívám také jenom jednou, tedy $\mathcal{O}(M)$. My ale víme, že pracujeme se stromem, tedy pro něj platí, že $M = N - 1$, takže složitost rekurze je opravdu $\mathcal{O}(N)$. Načtení hodnot, závěrečné vyhledání nejvýhodnějšího vrcholu a výpis cesty nám též trvá $\mathcal{O}(N)$. Paměťová složitost je při reprezentaci seznamem sousedů také lineární.

Jana Kravalová

18-1-4 Dortík

Sfoukněte svíce, uchopte do ruky nůž, nakrojte dort a vydatně se posilněte na následující vzorové řešení.

Nejprve několik poznámek k došlým řešením. Někteří řešitelé si bohužel do zadání domysleli některé dodatečné podmínky, které jsme ve skutečnosti nikde netvrdili. Například to, že úhly jsou vždy jen celá čísla a že se nikdy nevyskytnou dvě svíčky na stejném místě. Takové podmínky ovšem úlohu zjednodušují, proto jsem za takové věci strhával body. Pokud si příště nebudete jisti, řešte raději tu těžší variantu, případně není problém se nás zeptat.

Ale teď již k věci. Ukážeme si řešení, které kdyby dostalo na vstupu úhly svíček vzestupně setříděné, seběhlo by v lineárním čase vzhledem k počtu svíček. Nejprve si uvědomíme, že mezi každými dvěma sousedními svíčkami v K-úhelníku musí být úhel $360/K$. Další naše pozorování bude, že každá svíčka může ležet maximálně v jednom K-úhelníku.

Mějme tedy na vstupu setříděné úhly U. Dle našich úvah z nich teď můžeme klidně vyházet duplikáty, tedy ze svíček na stejné pozici nechat jen jednu. Poté projdeme seznam svíček od nejmenšího úhlu k největšímu a budeme si pro i-tou svíčku s úhlem U_i počítat následující věci: Kde (a jestli vůbec) má nějakého „předchůdce“ P_i na potenciálním K-úhelníku (tedy svíčku na úhlu $U_i - 360/K$), a kolikátá svíčka C_i s rozestupem $360/K$ v řadě to je (čili kde se v potenciálním K-úhelníku nachází).

Když budeme mít tato data spočítána, stačí se podívat, jestli existuje svíčka s právě $K-1$ pravidelnými předchůdci. Jestliže ano, potom touto svíčkou končí K-úhelník a proskáčeme skrz zpětné odkazy P po svíčkách a výsledek vypíšeme.

Zbývá si rozmyslet, jak efektivně počítat žádaná data. Použijeme k tomu metodu dvou posuvných indexů. Mějme indexy i a j, kde j bude vždycky ten více vepředu. Na začátku nastavíme i na 1 a j na 2. Pokud je rozdíl úhlů U_i a U_j roven $360/K$, j-tá svíčka má přechůdce i a je (C_i+1)-ní v pořadí, nastavíme tedy příslušně hodnoty P_j a C_j. Pokud je $U_j - U_i > 360/K$, tedy svíčky jsou příliš daleko, zvýšíme i o 1, čímž je k sobě přiblížíme, pokud $U_j - U_i < 360/K$, zvýšíme o 1 j, čímž je oddálíme. Všimněte si, že pokud jsou úhly setříděné, náš postup skutečně najde všechny dvojice správně vzdálených svíček.

Vyházení duplikátů zvládneme jedním průchodem přes U v lineárním čase, stejně tak výpis výsledků proskákáním pole P a C. Při posouvání dvou indexů oba pouze rostou, tedy se nad každým prvkem vykonají maximálně dvě operace, máme tedy časovou složitost $\mathcal{O}(N)$. Ještě je potřeba započítat čas na třídění na počátku, to umíme například použitím nějakého rychlého kuchařkového algoritmu v čase $\mathcal{O}(N \log N)$, dohromady tak máme časovou složitost $\mathcal{O}(N \log N)$. Všimněte si, že třídění je nejpomalejší částí našeho algoritmu. Paměťová složitost je $\mathcal{O}(N)$, pamatujeme si tři pole délky N.

Ještě poznámka k programu: ve skutečnosti bychom se obešli bez jednoho z polí P či C, pro větší srozumitelnost však v programu používáme obě.

Tomáš Valla

18-1-5 Matlalové

Matlalové jsou již za vysokým drátěným plotem a jen občas hromada přebytečného pletiva zaclání ve výhledu. Spíše byl problém postavit plot dříve, než Matlalové zase odletí.

Nebudu déle zdržovat a rovnou přejdu ke vzorovému řešení. Stoly si rozdělíme na dvě vodorovné a dvě svislé hrany stolu. Nejprve zjistíme, které vodorovné hrany tvoří obvod stolů. Pak můžeme stoly otočit o 90° a použít stejný algoritmus na vertikální hrany. Proto se zabývejme pouze horizontálními hranami.

Představme si horizontální přímku, kterou posunujeme přes naši soustavu stolů, hledanému sjednocení stolů budeme říkat *oblast*. Jak přecházíme přímkou do vnitřku oblasti a ven z ní, připočítáváme tyto hrany přechodu k celkovému obvodu. Horizontální část obvodu se může změnit pouze na místě, kde začíná nebo končí nějaký stůl vodorovnou hranou. To je hlavní idea algoritmu.

Mějme tedy pole $2n$ vodorovných hran. Zvolíme si směr průchodu podle rostoucí y-nové souřadnice. U každé hrany si zapamatujeme

- y – y-ovou souřadnici
- `left` – souřadnici x počátku hrany
- `right` – souřadnici x konce hrany
- `open` – zda je hrana otevírající, tedy projdeme jí dříve než druhou hranou stolu

Nyní pole setřídíme podle dvou kritérií. Prvním je y-nová souřadnice. Pokud ji mají dvě hrany stejnou, pak upřednostníme tu, která je otevírající. To proto, abychom mezi stoly stojícími těsně vedle sebe nepostavili plot.

Při průchodu přímkou přes oblast můžeme na přímce zobrazit oblast jako několik intervalů. Pokud si budeme tyto intervaly pamatovat, změna intervalu znamená konec nebo začátek oblasti čili příspěvek do obvodu. Je vidět, že intervaly mohou začínat a končit pouze na místě, kde začínají nebo končí stoly. Těchto bodů je maximálně $2n$. Místo celé přímky si stačí pamatovat pouze $2n - 1$ bloků. Jak budeme aktualizovat intervaly? Pokud projdeme otevírající hranou, znamená to, že jsme uvnitř nějakého stolu, pokud projdeme ukončující hranou, právě jsme stůl opustili. Stolů ale může na sobě ležet více a proto si pro každý blok zapamatujeme číslo c_i, kolik stolů je právě na jeho místě. Projdeme hrany jednu po druhé tak, jak je máme setříděny a pokud je to hrana otevírající, přidáme interval na místě hrany, jinak interval odebereme. Přidání a odebrání intervalu znamená přičtení nebo odečtení jedničky od c_i všech bloků, do kterých hrana zasahuje. Pokud se přitom změní c_i některého bloku z 0 na 1 nebo z 1 na 0, znamená to, že jsme na tomto