```
        // vypíšeme výsledek
        for (int i = 1; i <= pdni; i++)
                if (dny[i])
                        printf("%d ", kusy[dny[i]].id);
        for (int i = 1; i <= N; i++)
                if (!kusy[i].umyt)
                        printf("%d ", kusy[i]);
                printf("\n");
        return 0;
}
```

Výsledková listina dvacátého prvního ročníku KSP po druhé sérii

		škola	*ročník*	*sérií*	*2121*	*2122*	*2123*	*2124*	*2125*	*2126*	*série*	*celkem*
1.	Vítězslav Plachý	GJiříPoděb	3	2	9	9	10		2	10	40,1	81,3
2.	Filip Hlásek	GMikul23PL	2	7		9	10		10	10	39,8	79,0
3.	Petr Čermák	GEBenešeKL	3	2	6	9	5	4	6	9	36,8	77,5
4.	Michal Bilanský	GLepařovJČ	3	2	8		7	1	7	9	38,4	76,7
5.	Vojtěch Kolář	G Neratov	3	8	7	9	10	7	10	6	36,8	75,1
6.	Lukáš Ptáček	GJAKŽeliez	3	2	5	2	6	0		9	31,2	69,9
7.	Jiří Cidlina	GVoděraPH	4	2	4	2	6	5	1	4	29,4	68,8
8.	Jitka Novotná	G Bílovec	4	4		2		7	6	4	25,8	61,2
9.	Pavel Veselý	G Strakon	4	14		2	10		10	3,5	24,0	60,0
10.	Jan Vaňhara	G Holešov	4	3	5	9		7	6		32,4	56,2
11.	Vlastimil Dort	GŠpitálsPH	3	2			5	7	2	5	28,0	55,4
12.	Martin Zikmund	G Turnov	1	2	4	2	3	3	2	4	25,1	55,1
13.	David Věčorek	GTNovákBO	3	2	4			7		1	16,9	54,7
14.	Karel Tesař	SPŠE Plzeň	3	2	4	2	0	8	2	3,5	25,7	54,4
15.	Jiří Setnička	G25březnPH	2	2		9			5		16,7	44,3
16.	Filip Štědronský	GMikul23PL	2	2						12	12,0	40,5
17.	Alexander Mansurov	GNVPlániPH	0	1							0,0	39,7
18.	Filip Sládek	GNámestovo	3	1							0,0	38,1
19.	Libor Plucnar	GBezručeFM	4	10							0,0	36,2
20. – 21.	Tomáš Pikálek	GBoskovice	2	1							0,0	35,6
	Jan Veselý	G Strakon	2	1							0,0	35,6
22.	Lukáš Chmela	GJŠkodyPŘ	0	1							0,0	35,2
23.	Pavel Taufer	ArcibisGPH	3	4				8			8,0	34,4
24.	Barbora Janů	GKepleraPH	2	2	4	1	0	0	1		11,5	34,3
25.	Ondřej Pelech	GJNerudyPH	4	1							0,0	33,4
26.	Petr Pecha	SPŠSVsetín	2	2	4,5			0	1		9,7	33,3
27.	Milan Rybář	GJungmanLT	4	1							0,0	29,5
28.	Štěpán Šimsa	GJungmanLT	0	2						2,5	5,2	29,2
29.	David Formánek	GJarošeBO	2	2	4	9					15,8	27,9
30.	Karolína Burešová	G ČesLípa	2	1							0,0	27,7
31.	Petr Zvoníček	G Slavičín	3	2	4				2		11,0	26,5
32.	Honza Žerdík	G Příbor	4	1	4				5	8	25,7	25,7
33.	Jan Škoda	GMikul23PL	2	1							0,0	25,1
34.	Alena Bušáková	G Trutnov	2	1							0,0	22,6
35.	Karel Král	G Most	3	2		9					9,0	20,6
36.	Radim Cajzl	GNoMěsNMor	2	17	4					4	4,5	19,7
37.	Alžběta Pechová	SPŠSVsetín	4	5							0,0	19,3
38.	Hynek Jemelík	GJarošeBO	2	2			7				9,0	19,0
39.	Jakub Sochor	G Bílovec	4	1							0,0	17,1
40.	Karel Kolář	GŠpitálsPH	4	2	5	9					16,7	16,7
41.	Mirek Jarolím	GMikul23PL	3	4							0,0	15,3
42.	David Vondrák	GDašickáPA	3	2			5				7,7	14,8
43.	Kateřina Lorenzová	G Česká ČB	2	2		2					4,1	14,3
44.	Martin Holec	G Slavičín	2	2					1		2,4	11,4
45.	Jiří Daněk	GKřenováBO	3	1			10				10,0	10,0
46.	Dominik Smrž	GOhradníPH	0	1							0,0	8,8
47.	Petr Babička	VOŠGSvetla	4	1							0,0	8,1
48.	Stanislav Fořt	GCoubTábor	1	1							0,0	7,5
49.	Jiří Keresteš	SPŠE Plzeň	3	5							0,0	7,4
50.	Pavel Kratochvíl	VOŠGSvetla	1	5				6			7,0	7,0
51.	Igor Koníček	G UherBrod	3	1							0,0	5,7
52.	Ladislav Maxa	GKepleraPH	3	1							0,0	5,5
53.	Jan Kostecký	VOŠŠumperk	2	1							0,0	4,5

Milí řešitelé a řešitelky!

Jaro je sice ještě daleko před námi, ale medvědi i jiní pilní organizátoři KSP se již probudili z tradičního zimního spánku a přinášejí Vám další sérii úložek.

Svá řešení posílejte do pondělí 30. 3. 2008 elektronicky na *http://ksp.mff.cuni.cz/submit/*, nebo klasickou poštou na známou adresu:

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

Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz*.

Čtvrtá série dvacátého prvního ročníku KSP

V tmavém sklepě se mihotalo světlo svíček. Průvan si hrál s jejich ohněm a stíny vrhaly podivuhodné tvary. Všude vládlo mrtvolné ticho, přerušované jen tichým dýcháním, občasným krysím zapištěním a . . . zachřestěním kostek?

„Zase pětka? Nemůžu mít ještě jeden pokus?“

„Ne, tady to máš v pravidlech – 'Každý hráč si na začátku hry naháže schopnosti postavy. Háže se dvacetistěnnou kostkou a jsou na to dva pokusy, z nichž si hráč vybere ten lepší'. Tak už přestaň kňourat, ať popojedem, nemáme na to celou noc!“

„Ale síla 5? Kdo to kdy viděl, aby měl trpaslík sílu pět?“

„Ale to je jenom ve hře, Boendale, to jako nejseš ty . . . Koukni, jaké sis zvolil povolání?“

Trpaslík se podíval do svých poznámek. „Haa-keř“, oznámil po chvíli.

„No vidíš, hackři nejsou moc silní. Zato jsou ale hodně inteligentní,“ snažil se Barun dál přesvědčovat hráče.

„Ale jak jako udržím svou sekyru, když budu tak slabej?“ Nemělo to cenu . . .

* * *

Všechno začalo asi před týdnem, když mezi starými magickými svazky svého mistra našel tu podivuhodnou knížku: CnH – Computers and Hackers. Zpočátku ji moc nechápal, popisovala jakýsi tuze zvláštní svět. Byl obydlen jenom lidmi, kteří nejenže neovládali magii, ale dokonce v ni ani nevěřili! Místo toho v laboratořích stvořili jakési podivné zařízení, takzvané Počítače, které se pak naučili ovládat pomocí určitých příkazů a tyto se pak staly neoddělitelnou součástí jejich světa. Vypadalo to jako nějaká propaganda těch zatracených alchymistů. Ti se taky vrtali v různých strojích, místo aby se věnovali studiu magie. Už už se chystal knížku zaklapnout, když ho zaujal velký nápis: „Hra roku 2 149 třetího věku! Doporučuje 9 z 10 elfů!“ Ahá, takže hra! No to jsem zvědavý, co na to řeknou Boendal s Mírielem . . .

* * *

„Takže projdeme si pravidla ještě jednou, jo? Na začátku hry si každý hráč zvolí povolání. Řekněte mi popořadě, co jste si kdo zvolil.“

„Haa-keř,“ zamumlal mrzutě Boendal, ještě stále zklamaný z toho, že v herním světě s sebou nemůže nosit sekyru.

„Správce sítě. Hele, a můžu mít na sobě alespoň svoji modro-zelenou košilku?“ zeptal se s nadějí v hlase elf Míriel.

„Ne, tady jasně stojí – 'Pro správce sítě je typické vytahané, měsíc neprané, každodenně nošené tričko se vzorem tučňáka, případně ďábláka.' Hele, hraj postavu, jo? Jinak ti strhnu body!“, sprdnul ho Barun. „Tak jo, další postava?“

„Uaargh!“

„Cože?“

„UAARGH!“

„Říká, že chce hrát algebraického topologa,“ překládal Míriel, „hele, já za to nemůžu, že mně teta hodila na krk hlídání bratrance. To víš, sehnat chůvu pro mladého trolla je dnes docela drahá záležitost . . . “

„No jo, no jo, chápu. Hele, tak já začnu. Takže, nacházíte se . . . třeba na přednášce ve škole – jste teď ještě jenom učňové, jo? Sedíte každý u svého Počítače, když tu najednou Boendalovi přijde ímejl.“

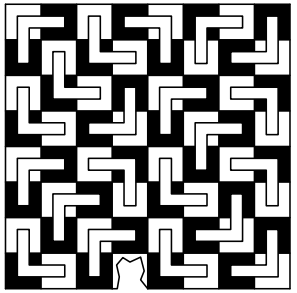
„Cože mi to přišlo?“ zeptal se polekaně Boendal.

„Něco jako dopis. Prostě zpráva. Každopádně když to otevřeš, tak zjistíš, že je to nějaká hra. Co uděláš?“

„Tak si zahrajem, ne?“ navrhnul Míriel a Barun začal vysvětlovat pravidla.

21-4-1 Dláždění šachovnice — 8 bodů

Hra je velice jednoduchá. Hráč na začátku dostane čtvercový plánek, něco jako šachovnici, o hraně velikosti 2^N políček. Jedno políčko ale chybí (libovolné). Úkolem je pokrýt šachovnici útvary podobnými písmenu L složenými ze tří kostiček. L-ka je povoleno otáčet. Vaším úkolem je nalézt algoritmus, jak pokrýt takovýhle plánek, ať je chybějící políčko na libovolném místě. Na obrázku je jedno z možných pokrytí pro $N = 3$.



„Tak jo. Jakmile jste dohráli, objevila se na obrazovce zpráva: 'Právě jste splnili kvalifikační test na soutěž o Nejlepšího crackera roku 2009!'“

„To jsem se právě kvalifikoval na sušenku?“ vyděsil se Boendal.

„Ach jo, na crackera, ne na krekr.“

„Fuul mít hlad!“ ozval se mladý troll a začal kolem sebe máchat rukama.

Míriel si povzdechl, zamával rukama, zamrmlal si něco pod nosem a krysa v rohu místnosti začala vonět pečeným masem. Fuul se po ní nedočkavě vrhl, pak se usadil a o poznání veselejší pozoroval zbytek družiny. Barun se ujal slo-

va: „Tak co děláte dál?“

21-4-2 Dosah kouzla — 9 bodů

Kouzlit, to není jen tak. Nejenže to vyžaduje hluboké vědomosti a určitou dávku energie, ale taky má každé kouzlo svůj dosah působnosti. Jaký dosah by muselo mít Mírielovo kouzlo, aby úspěšně zasáhlo krysu, ať by stála v kterémkoli rohu místnosti? On sám seděl taky v rohu místnosti a sklep má půdorys konvexního n-úhelníku. Na vstupu dostanete konvexní n-úhelník (n vrcholů popsaných souřadnicemi $[x, y]$) a vaším úkolem je najít dva nejvzdálenější vrcholy.

„Ne, pořád to nechápu“, trval na svém Boendal.

„Je to prostě taková skříňka a k ní je připojená jiná skříňka a k ní ještě jedna. Vidíš, tady to je na obrázku,“ Barun píchnul prstem do knížky. Už půl hodiny se snažil kamarádům vysvětlit, jak vlastně vypadá takový počítač a co obnáší programování.

„Takže když jako praštím do tady tý skříňky, tak na tamtý se něco objeví?“

„No, v podstatě nějak tak,“ povzdechl si Barun.

„Hele, a co je tady ten ‘Robot’?“ vyzvídal dál.

„Robot? To je takové mechanické zařízení, které za tebe dělá nějakou práci,“ vysvětloval Barun.

„Něco, co maká za mně? To zní hodně zajímavě . . . “ zalesklo se Boendalovi v oku a pustil se do čtení pravidel.

* * *

„Tak ty kabely zapojím tady!“ prohlásil vítězoslavně Boendal.

„No, když myslíš . . . Robot vstal, začal tancovat po okolí, vyházel pár hrníčků od kafe ven z okna, pak zalil kytku a sám se vypnul,“ popsal situaci Barun.

„Paráda, už to skoro funguje!“ těšil se Boendal.

„Hm, možná by jsme to neměli jenom tak zkoušet,“ snažil se zapojit do debaty Míriel.

„Ty se do toho nepleť. Už jenom pár pokusů a budeme mít prvního robota na zaplétaní vousů na světě! Nedovedeš si představit, jaká je to otrava dělat to každý den ručně . . . “

21-4-3 Stavění robota — 10 bodů

Boendalova technika stavění robota je vskutku originální. Prostě si vybere nějaké vstupy na základní desce, a ty pak připojí kabely ke zdroji. Při různém propojení dělá robot různé věci. Boendala by zajímalo, kolik různých věcí dokáže robot dělat, když má N výstupů a K kabelů. Nejjednodušší způsob, jak daný problém vyřešit, je spočítat kombinační číslo $\binom{N}{K}$, což se dá rozepsat jako

$$\frac{N \cdot (N-1) \cdot \cdots \cdot (N-K+1)}{K \cdot (K-1) \cdot \cdots \cdot 1}.$$

Nebude to ale tak jednoduché. Protože v informatickém světě se občas stává, že je číslo tak velké, že se nevejde do paměti, bude vaším úkolem spočítat kombinační číslo $\binom{N}{K}$ modulo M, tak, aby se vám mezivýpočet vždy vešel do paměti. Na vstupu dostanete 3 čísla – N, K a M a na standardní výstup vypište výsledek. Předpokládejte, že $0 \leq K \leq N \leq 1\,000\,000$ a $1 \leq M \leq 10\,000$.

Příklad 1: Vstup: `6 2 100` Výstup: `15`

Příklad 2: Vstup: `1000 400 1270` Výstup: `1040`

Tato úložka je praktická, což znamená, že řešení budete odevzdávat výhradně formou odladěného zdrojového kódu. Přesnější zadání a formulář na odevzdání kódu naleznete jako vždy v CodExu. Nevíte-li, co praktická úložka je a jak přesně postupovat, podívejte se do zadání úlohy 21-1-2 „Optimalizace kotlů“.

„BUUM!, ozvalo se v školní laboratoři. Právě jste zničili kus budovy. Z tohohle bude pořádný průšvih . . . “ popisoval situaci Barun.

„Tak zdrháme, ne?“ navrhnul trpaslík.

„No, to teda nebylo moc rozumné. Stejně na vás přišli a . . . jste podmíněně vyloučeni. Jo, to by mohl být adekvátní trest,“ oznámil jim Barun.

„Hm,“ zamyslel sa Míriel, „a kde jsou uloženy školní záznamy?“

„Myslím, že záznamy jsou uloženy výhradně v elektronické podobě, na centrálním školním počítači. Proč se ptáš?“

„Hehe, jsme přeci nějací hackři, nebo ne?“ usmál se na Baruna elf.

* * *

„Napiš tam ‘heslo’! To bude určitě fungovat!“ povzbuzoval Míriela Boendal, „nebo ‘pás-vord’!“

Už hodinu se snažili dostat se na speciální stránky školního systému, ale zatím bez úspěchu.

„Hele, když to nejde logicky, vem větší sekyru. Zkusíme tam napsat postupně všechna možná hesla,“ navrhnul Boendal.

„To zní dobře, ale netrvalo by to příliš dlouho?“ pochyboval Míriel.

„Hm . . . Ale mohli bychom tam zkusit zadat každé páté možné heslo, nebo každé sedmé.“

21-4-4 Heslo — 10 bodů

Heslo, to je vlastně permutace nějakých znaků, z nichž se některé můžou opakovat. Řekneme, že permutace p_1 je lexikograficky menší než permutace p_2, pokud první znak, ve kterém se liší (bráno zleva doprava) je na i-té pozici a platí $p_1[i] < p_2[i]$. Příklad: Mějme znaky 1, 2, 3 a 3. Pak jejich permutace 2133 je lexikograficky menší než permutace 2313. Permutace jsou seřazeny lexikograficky, pokud jsou seřazeny od nejmenší po největší. Pokud vygenerujeme všechny možné permutace určitých znaků a seřadíme je lexikograficky, pak permutace o K větší než zadaná permutace je permutace na pozici o K větší.

Na vstupu dostanete permutaci cifer (cifry se mohou opakovat) a číslo K a vaším úkolem je najít permutaci o K větší.

Příklad: Vstup: `1234 3` Výstup: `1423`

(protože po permutaci 1243 následuje permurace 1324, pak 1342, no a o 3 větší je permutace 1423).

„Gratuluju, tak jste se právě dostali do Školního informačního systému! Na obrazovce se objevily nějaké znaky – a vy jim vůbec nerozumíte. Vypadá to, že stránka je šifrovaná,“ oznámil družině Barun. „Co uděláte?“

„Hm, asi by to chtělo nějak líp prostudovat ty znaky. Jak vypadaj?“ zajímal se Míriel.

„No, je to velice jednoduché,“ zalesklo se Barunovi v očích a začal popisovat znaky šifry . . .

21-4-5 Znaky — 10 bodů

Znak je reprezentován čtvercem o hraně délky N pixelů, ve kterém je přesně N pixelů vybarvených. Platí ale, že

Úloha 21-2-5 – Nádobí – program

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 1000
#define SWAP(A,B,tmp) tmp = A; A = B; B = tmp;

struct nadobi { // jeden kus nádobí
        int id, t, c, umyt; // číslo; trvanlivost; cena; zda máme naplanováno umytí
};

struct nadobi kusy[MAX]; // informace o kusech nádobí
int halda[MAX*2],hpocet=0,dny[MAX],pdni; // halda; počet prvků; plány na jednotlivé dny; počet dní
int N,tmp; // počet kusů; proměnná pro swap

int halda_pridej_prvek(int index) { // přidání prvku do haldy
        halda[++hpocet] = index; // dáme na konec
        halda[hpocet*2] = halda[hpocet*2+1] = 0;
        int i = hpocet;
        // probubláme prvek nahoru na správné místo
        while (i > 1 && kusy[halda[i/2]].c < kusy[halda[i]].c) {
                SWAP(halda[i/2], halda[i], tmp);
                i /= 2;
        }
}

int halda_odeber_maximum() { // odebrání prvku z haldy
        if (hpocet == 0)
                return 0;
        int ret = halda[1]; // vymažeme a prohodíme s posledním prvek
        halda[1] = halda[hpocet];
        halda[hpocet--] = 0;

        int i = 1; // poslední prvek bubláme dolů, dokud není správně
        while (kusy[halda[i]].c < kusy[halda[i*2]].c || kusy[halda[i]].c < kusy[halda[i*2+1]].c) {
                // prohodíme s potomkem, který má větší cenu
                if (kusy[halda[i*2]].c > kusy[halda[i*2+1]].c) {
                        SWAP(halda[i], halda[i*2], tmp);
                        i = i*2;
                }
                else {
                        SWAP(halda[i], halda[i*2+1], tmp);
                        i = i*2+1;
                }
        }
        return ret;
}

// porovnávací funkce pro qsort
int porovnej(const void * a, const void * b) {
        int ta = ((struct nadobi*) a)->t;
        int tb = ((struct nadobi*) b)->t;
        if (ta < tb) return -1;
        else if (ta == tb) return 0;
        else return 1;
}

int main() {
        scanf("%d", &N);

        // načteme jednotlivé kusy
        for (int i = 1; i <= N; i++) {
                scanf("%d %d", &kusy[i].t, &kusy[i].c);
                kusy[i].id = i;
        }
        kusy[0].c = INT_MIN; // 0. kus plní funkci zarážky

        // utřídíme podle trvanlivosti (nejtrvanlivější na konec)
        qsort(&kusy[1], N, sizeof(struct nadobi), porovnej);

        int i = N; // procházíme dny a určujeme kusy k umytí
        pdni = N < kusy[N].t ? N : kusy[N].t;
        for (int t = pdni; t > 0; t--) { // začneme od minima dny/max. trvanlivost
                while (kusy[i].t >= t) // přidáme nové kusy nádobí
                        halda_pridej_prvek(i--);

                dny[t] = halda_odeber_maximum(); // vezmeme hladově nejcennější
                kusy[dny[t]].umyt = 1;
        }
```

```
   end;
end;

function MergeSort(Seznam:PPocitac):PPocitac; {dostane seznam a vrátí ho setřízený}
var Seznam1,Seznam2,Swap:PPocitac;
begin
   if (Seznam = nil) then MergeSort:=nil
   else if (Seznam^.Dalsi = nil) then MergeSort:=Seznam {koncové podmínky}
   else begin
      Seznam1:=nil;Seznam2:=nil;
      while Seznam<>nil do begin {rozdělí seznam poloviny - sudé a liché prvky zvlášť}
         Swap:=Seznam^.Dalsi;
         Seznam^.Dalsi:=Seznam1;
         Seznam1:=Seznam2;
         Seznam2:=Seznam;
         Seznam:=Swap;
      end;
      Seznam1:=MergeSort(Seznam1); {setřízení polovin}
      Seznam2:=MergeSort(Seznam2);
      MergeSort:=Merge(Seznam1,Seznam2); {a merge setřízených posloupností}
   end;
end;

procedure Vypis(Seznam:PPocitac);
begin
   while (Seznam<>nil) do begin
      write(Seznam^.Poradi,' ');
      Seznam:=Seznam^.Dalsi;
   end;
end;

function Obrat(Seznam:PPocitac):PPocitac;
var ObracenySeznam,Dalsi:PPocitac;
begin
   ObracenySeznam:=nil;
   while Seznam<>nil do begin
      Dalsi:=Seznam^.Dalsi;
      Seznam^.Dalsi:=ObracenySeznam;
      ObracenySeznam:=Seznam;
      Seznam:=Dalsi;
   end;
   Obrat:=ObracenySeznam;
end;

begin
   Nacti;
   if Pocitace = nil then begin
      writeln;  {není co vypisovat - 0 počítačů}
   end else if Pocitace^.Dalsi = nil then begin
      writeln('1'); {jedinný počítač ... takže není příliš co řešit}
   end else begin
      Rozdel;
      if (NadSpojnici = nil) and (PodSpojnici = nil) then
         writeln('Řešení neexistuje.') {všechny počítače leží na spojnici}
      else begin
         NadSpojnici:=MergeSort(NadSpojnici); {setřízení jednotlivých seznamů}
         PodSpojnici:=MergeSort(PodSpojnici);
         NaSpojnici:=MergeSort(NaSpojnici);
         if (NadSpojnici = nil) then NadSpojnici:=NaSpojnici
         else PodSpojnici:=Merge(NaSpojnici,PodSpojnici); {přidání bodů na spojnici k příslušné straně}
         write(Nejlevejsi^.Poradi,' ');
         Vypis(NadSpojnici);
         write(Nejpravejsi^.Poradi,' ');
         PodSpojnici:=Obrat(PodSpojnici);
         Vypis(PodSpojnici);
         writeln;
      end;
   end;
end.
```


v každém vodorovném, svislém i šikmém směru je vybarvený maximálně jeden pixel. Míriel si ale všimnul, že některé čtverce se opakují a že by se mu hodilo vědět, kolik různých znaků se to vlastně v šifře používá.

Na vstupu dostanete přirozená čísla N a K, a pak K čtverců popsaných výše. Čtverce budou reprezentovány jako dvourozměrné pole integerů: 1 znamená, že pixel je vybarvený, 0, že není. Vaším úkolem je zjistit počet unikátních čtverců, tak, aby váš algoritmus měl co nejmenší paměťové nároky (reálné, nejen asymptotické).

„Výborně! Povedlo se vám rozluštit ty znaky a na obrazovce čtěte nápis . . . “

„Míříriéééli! Kde to zase vězíš?! Večeře je už hotová a ty jsi ještě nenakrmil draka!!“

„A jeje, máma,“ povzdechl si Míriel.

„Sakra, to mi připomíná, že bych měl taky pomalu jít. Slíbil jsem bráchovi, že mu pomůžu se skládáním hudby k jeho nejnovějšímu hitu ‘Zlato, zlato, zlato!’.“

„Hm, tak pro dnešek asi konec. Dohrajem to někdy příště,“ usmál se Barun a uklidil knížku k sobě do batohu.

A tak se družina rozešla – Míriel šel nakrmit draka, Boendal komponovat hudbu a Fuul se vyvalovat v jeskynním jezírku. A Barun? Hned na druhý den si šel k alchymistům půjčit pár knížek. Pár dní na to se několik sousedů stěžovalo na hluk v okolí magické laboratoře, a asi za týden byl spatřen, jak za bouřky chytá blesky do velké černé krabice . . .

21-4-6 Nejtěžší číslo — 12 bodů

Tuto úlohu musíte řešit v programovacím jazyce Rapl, jehož popis najdete v zadání úlohy 21-1-6 z první série.

Tentokrát budeme ovšem místo nejkratšího programu hledat program *nejrychlejší*. Nebude nás ale zajímat asymptotická časova složitost programu, nýbrž skutečný počet provedených instrukcí Raplu.

Vše se bude točit okolo vážení čísel. *Vahou čísla* nazveme počet jedniček v jeho dvojkovém zápisu. Nula má tedy váhu 0 (a je to jediné číslo s nulovou vahou), mocniny dvojky mají váhu 1 a třeba takové číslo 1 000 je těžké 6 jednotek, protože jeho dvojkový zápis je `1111101000`.

a) Napište co nejrychlejší program, který je spuštěn s číslem v registru `x` a odpoví jeho vahou uloženou v registru `w`.

b) Vymyslete co nejrychlejší program, který dostane `n` čísel `A[1]` až `A[n]` ($1 \le n \le 2^{32} - 1$) a odpoví v registru `y` nejtěžším z těchto čísel. Pokud je nejtěžších čísel více, může si vybrat libovolné jedno z nich.

Recepty z programátorské kuchařky

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \le x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

```
void přidej (typ_klíče klíč, typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
    !stejný(klíč, heš[index].hodnota))
      return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
```

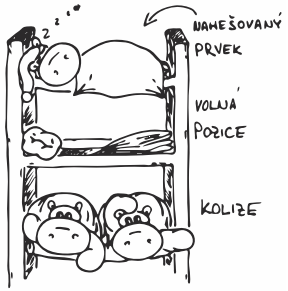
```
  return 1;
}
```

Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}

int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je,ale ne
    // to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T + H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                          \
  a-=b; a-=c; a^=(c>>13);                     \
  b-=c; b-=a; b^=(a<< 8);                     \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);        \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);        \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff;  \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff;  \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff;  \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff;  \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff;  \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba:

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce časteji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost,


```
            then Nejpravejsi:=Novy;
          if (Novy^.X < Nejlevejsi^.X) or ((Novy^.X = Nejlevejsi^.X) and (Novy^.Y > Nejlevejsi^.Y))
            then Nejlevejsi:=Novy;
          Novy^.Poradi:=i;
          Novy^.Dalsi:=Pocitace;
          Pocitace:=Novy;
        end;
    end;
end;

procedure Rozdel;
{Rozdělí počítače podle relativní polohy vzhledem k spojnici nejlevějšího a nejpravějšího}
var Zpracovavany:PPocitac;
    ZSlozkaVektorovehoSoucinu:real;
    VektX,VektY:real; {vektor popisující směr spojnice nejlevějšího a nejpravějšího počítače}
    ZpracVektX,ZpracVektY:real; {vektor popisující směr spojnice nejlevějšího a zpracovávaného bodu}
begin
   NadSpojnici:=nil;
   PodSpojnici:=nil;
   NaSpojnici:=nil;
   VektX:=Nejpravejsi^.X - Nejlevejsi^.X;
   VektY:=Nejpravejsi^.Y - Nejlevejsi^.Y;
   while Pocitace <> nil do begin
      Zpracovavany:=Pocitace;
      Pocitace:=Pocitace^.Dalsi; {vyřazení zpracovávaného počítače ze seznamu}
      if (Zpracovavany = Nejlevejsi) or (Zpracovavany = Nejpravejsi) then continue;
      {tyto 2 se zpracovávají zvlášť}
      ZpracVektX:=Zpracovavany^.X - Nejlevejsi^.X;
      ZpracVektY:=Zpracovavany^.Y - Nejlevejsi^.Y;
      ZSlozkaVektorovehoSoucinu:=VektX * ZpracVektY - VektY * ZpracVektX;
      if abs(ZSlozkaVektorovehoSoucinu) < Presnost then begin {leží na spojnici}
         Zpracovavany^.Dalsi:=NaSpojnici;
         NaSpojnici:=Zpracovavany;
      end else if ZSlozkaVektorovehoSoucinu > 0 then begin {leží nad spojnicí}
         Zpracovavany^.Dalsi:=NadSpojnici;
         NadSpojnici:=Zpracovavany;
      end else begin {leží pod spojnicí}
         Zpracovavany^.Dalsi:=PodSpojnici;
         PodSpojnici:=Zpracovavany;
      end;
   end;
end;

function Merge(Seznam1,Seznam2:PPocitac):PPocitac;
{dva setřízené seznamy slije - původní seznamy jsou během procesu zničeny}
var Prvni,Posledni:PPocitac;
   function Porovnej(Pocitac1,Pocitac2:PPocitac):boolean;
   {porovná polohy dvou počítačů a vrací true, pokud má být Pocitac1 zatřízen jako první}
   begin
      Porovnej:=(Pocitac1^.X < Pocitac2^.X) or ((Pocitac1^.X = Pocitac2^.X) and (Pocitac1^.Y > Pocitac2^.Y));
      {zde si můžeme dovolit mezi reálnými čísly test na rovnost - zaokrouhlovací chyby, které vzniknou}
      {při načítání budou u stejných hodnot na vstupu stejné a tedy rovnost bude doopravdy platit}
   end;
begin
   if (Seznam1 = nil) then Merge:=Seznam2 {v případě, že je jedna posloupnost prázdná je slití triviální}
   else if (Seznam2 = nil) then Merge:=Seznam1
   else begin
      if Porovnej(Seznam1,Seznam2) then begin {nejdříve zjistíme čím bude výsledná posloupnost začínat}
         Prvni:=Seznam1;
         Seznam1:=Seznam1^.Dalsi;
      end else begin
         Prvni:=Seznam2;
         Seznam2:=Seznam2^.Dalsi;
      end;
{Konec funkce (až po přiřazení výsledku) jen slije zbytek seznamů.}
{Je zde implementována nerekurzivní varianta. Rekurzivní by byla výrazně kratší.}
{  Prvni^.Dalsi:=Merge(Seznam1,Seznam2); }
      Posledni:=Prvni;
      while (Seznam1<>nil) and (Seznam2<>nil) do begin {a pak slijeme zbytek}
         if Porovnej(Seznam1,Seznam2) then begin
            Posledni^.Dalsi:=Seznam1;
            Posledni:=Seznam1;
            Seznam1:=Seznam1^.Dalsi;
         end else begin
            Posledni^.Dalsi:=Seznam2;
            Posledni:=Seznam2;
            Seznam2:=Seznam2^.Dalsi;
         end;
      end;
      if (Seznam1 = nil) then Posledni^.Dalsi:=Seznam2 else Posledni^.Dalsi:=Seznam1;
      Merge:=Prvni;
```

```
procedure DFS( v: integer);
var n: integer;
    i: integer;
begin;
    n := 0;
    mela[v].aktivni := 1;
    if vpozice > max then exit;
    for i := 1 to mela[v].znamych do begin;
        if mela[ mela[v].znami^[i] ].aktivni = 1 then kruznice := 1 { detekována kružnice }
        else if mela[ mela[v].znami^[i] ].aktivni = 2 then continue { vrchol již hotov }
        else DFS(mela[v].znami^[i]);
        if kruznice = 1 then exit;
    end;
    {jsme-li tu, kružnice nebyla nalezena}
    usporadani[vpozice] := v; {deaktivuj vrchol a ulož jej do uspořádání}
    vpozice := vpozice + 1;
    mela[v].aktivni := 2;
end;
var pocet: integer;
    z: integer;
    m: integer;
begin;
    read(pocet); {vstup}
    for m := 1 to pocet do begin;
        mela[m].aktivni := 0;
        read(mela[m].znamych);
        getmem(mela[m].znami, mela[m].znamych * sizeof(integer));
        for z := 1 to mela[m].znamych do begin;
            read(mela[m].znami^[z]);
        end;
    end;
    vpozice := 1; {spuštění}
    kruznice := 0;
    for m := 1 to pocet do begin;
        if mela[m].aktivni = 0 then begin;
            DFS(m);
        end;
        if (kruznice = 1) or (vpozice > max) then break;
    end;
    if kruznice = 1 then write('no') else {výstup}
        for z := pocet downto 1 do write(usporadani[z],' ');
    writeln;
end.
```

Úloha 21-2-4 – Síťování – program

```
const Presnost = 1E-6;  {Přesnost pro práci s reálnými čísly}
                        {Čísla lišící se o méně než tuto konstantu považujeme za stejná}

type
   PPocitac = ^TPocitac;
   TPocitac = record
      X,Y:real;
      Poradi:integer;
      Dalsi:PPocitac;
   end;

var
   Pocitace:PPocitac;
   Nejlevejsi,Nejpravejsi:PPocitac;
   NadSpojnici,PodSpojnici,NaSpojnici:PPocitac;

procedure Nacti;
{Načtení vstupu a určení nejlevějšího a nejpravějšího počítače (z pohledu X)}
var N,i:integer;
    Novy:PPocitac;
begin
   readln(N);  {počet počítačů}
   Pocitace:=nil;
   if N <> 0 then begin   {načtení do lineárního spojového seznamu}
      new(Novy);
      readln(Novy^.X,Novy^.Y);
      Novy^.Poradi:=1;
      Novy^.Dalsi:=nil;
      Nejlevejsi:=Novy;
      Nejpravejsi:=Novy; {první počítač je zvlášť kvůli inicializaci}
      Pocitace:=Novy;
      for i:=2 to N do begin
         new(Novy);
         readln(Novy^.X,Novy^.Y);
         if (Novy^.X > Nejpravejsi^.X) or ((Novy^.X = Nejpravejsi^.X) and (Novy^.Y < Nejpravejsi^.Y))
```


sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Pár poznámek na závěr:

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé příhrádce (příhrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nová heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

Vzorová řešení druhé série dvacátého prvního ročníku KSP

21-2-1 Špinavé tričko

Ukážeme dvě řešení problému: jednodušší v čase $\mathcal{O}(N^3)$, a o něco rychlejší v čase $\mathcal{O}(N^2 \log N)$. Začneme tím jednodušším :-)

Skvrnu si uložíme jako x-ové souřadnice svislých hran a y-ové souřadnice vodorovných hran. V každém okamžiku si budeme pamatovat spojový seznam skvrn, které se na tričku nacházejí. Na začátku je seznam prázdný; když načteme ze vstupu další skvrnu, přidáme ji do seznamu. Navíc se podíváme, jestli nám nová skvrna nepřekryla nějakou starší skvrnu – v takovém případě starou skvrnu nahradíme seznamem jejích zbylých nepřekrytých částí.

Že se dvě skvrny překrývají, poznáme snadno: překrývají se jejich průměty na vodorovnou, nebo na svislou osu.

Nyní vyřešíme rozpadávání staré skvrny, kterou překryla nová.

Pokud zasahuje stará skvrna nad novou, uřízneme z ní vršek – to je obdélník, který má všechny souřadnice stejné jako stará skvrna, kromě dolní hrany (ta bude rovna horní hraně nové skvrny).

Pokud zasahuje stará skvrna pod novou, podobným způsobem uřízneme spodek (použijeme souřadnice staré skvrny, jen horní hrana bude rovna dolní hraně nové skvrny).

Pokud stará skvrna zasahuje i nalevo od nové, uřízneme levý kus z toho, co ze staré skvrny zbylo po našem případném předchozím řezání.

A nakonec uřízneme pravý kus, pokud to půjde. Tím získáme až čtyři zbylé kusy, na které se stará skvrna rozpadla, protože ji z části (nebo zcela) překryla skvrna nová. Tyto nové skvrny připojíme do spojového seznamu místo skvrny staré.

Zatím to vypadá, že časová složitost našeho algoritmu může být dost vysoká, protože může vznikat velké množstvích malých „rozpadnutých“ skvrn. Můžeme si všimnout, že po provedení všech rozpadů bude počet skvrn nejvýše $\mathcal{O}(N^2)$. Když totiž protáhneme každou hranu každé skvrny přes celé tričko, rozdělíme ho nejvýše na $(2N-1)^2$ obdélníků ($2N$ svislými a $2N$ vodorovnými řezy), z nichž žádný se již nemůže nijak rozpadnout. Každou novou skvrnu porovnáváme s až $\mathcal{O}(N^2)$ předchozími, takže časová složitost není horší než $\mathcal{O}(N^3)$.

A nyní jak to udělat rychleji:

Nejdříve si zadání úlohy trochu upravme: Nebudeme počítat počet jednotek, které zabírají jednotlivé barvy, nýbrž pro každou skvrnu budeme počítat počet jednotek, na kterých je tato skvrna vidět.

Není těžké si rozmyslet, že pokud vyřešíme takto upravenou úlohu, tak nalezení řešení původního zadání je triviální: Stačí pro každou barvu sečíst počet jednotek, které zabírají skvrny dané barvy. A to je z algoritmického hlediska nezajímavá záležitost.

Nyní k samotnému řešení. První myšlenka, která určitě každého okamžitě napadne, spočívá ve vytvoření dvojrozměrného pole velikosti $(W-1) \times (H-1)$, které bude představovat tričko. Poté se postupně zpracovávají jednotlivé skvrny a příslušná políčka v poli se označují číslem této skvrny. Nakonec stačí pole projít a jednoduše spočítat výsledek.

Jakou složitost by mělo toto řešení? Vzhledem k tomu, že každé skvrna může být až velikosti $\mathcal{O}(W \cdot H)$, tak časová složitost by byla $\mathcal{O}(N \cdot W \cdot H)$ a paměťová $\mathcal{O}(W \cdot H + N)$. To je poměrně hodně. Navíc už pro poměrně malá W a H můžeme velmi brzy narazit na velikost fyzické operační paměti.

Co když rozdělíme plochu trička na oblasti, které jsou ohraničeny přímkami, které procházejí hranami jednotlivých skvrn? Určitě platí, že každá takto vzniklá oblast bude obsahovat stejná čísla. Navíc proto, že každá skvrna přispěje nejvýše čtyřmi přímkami, tak celkový počet těchto oblastí bude pouze $\mathcal{O}(N^2)$.

Ve skutečnosti tedy stačí pole velikosti $\mathcal{O}(N^2)$, ve kterém lze simulovat předchozí triviální algoritmus. Jaká bude časová složitost tohoto postupu? Pro každou skvrnu musíme určit, v jakých oblastech se nachází. I kdybychom toto zvládli rychle, tak každá skvrna se může skládat až z $\mathcal{O}(N^2)$ oblastí, takže časová složitost je minimálně $\mathcal{O}(N^3)$, což je sice lepší, ale stále to není ono.

Neefektivita předchozího postupu je ukryta v tom, že každ

dou oblast můžeme až $\mathcal{O}(N)$-krát přečíslovat. Co kdybychom ale nezpracovávali postupně jednotlivé skvrny, ale jednotlivé oblasti? Například zdola nahoru a zleva doprava. Pak by si stačilo průběžně udržovat seznam skvrn, které se v aktuální oblasti vyskytují, a z nich vždy vybrat tu, která se objevila nejpozději (to lze provést v čase $\mathcal{O}(\log N)$), a oblasti přiřadit její číslo.

Zbývá tedy vyřešit, jak onen seznam udržovat. Možné řešení je pamatovat si pro každý vodorovný pruh oblastí množinu skvrn, které se v tomto pruhu vyskytují a tuto množinu pak projít „zleva doprava“ a průběžně si pamatovat, které skvrny se v aktuální oblasti vyskytují.

Udržovat onu množinu skvrn je jednoduché. Pokud ji máme vytvořenou pro jeden pruh, tak je totiž snadné přejít na pruh následující. Stačí odebrat všechny skvrny, které se v tomto pruhu již nevyskytují a naopak přidat skvrny, které se v tomto pruhu nově objevily. Najít takové skvrny lze snadno, pokud si předem vytvoříme seznam všech horních a dolních hran, který je setříděné vzestupně podle souřadnice y. Pak když narazíme na dolní hranu, tak příslušnou skvrnu do seznamu přidáme. V případě horní hrany skvrnu odebereme.

Naprosto stejným způsobem pak zpracujeme jeden pruh. Ze skvrn v příslušné množině vytvoříme seznam levých a pravých hran, který setřídíme podle osy x a tento seznam pak jednoduše projdeme. Pokud narazíme na levou hranu, pak se v následující oblasti tato skvrna vyskytuje, pokud na hranu levou, tak se příslušné oblasti skvrna přestala vyskytovat. Seznam aktuálních skvrn pak budeme udržovat v haldě, abychom mohli vždy rychle nalézt skvrnu, která se v aktuální oblasti vyskytla nejpozději (je na vrchu).

Není těžké si rozmyslet, že tento seznam není třeba stále třídit. Je možné udržovat jej stále setříděný. Aby se pak lépe vkládalo doprostřed, je vhodné jej reprezentovat spojovým seznamem. Dále není těžké přijít na to, že žádné pole velikosti $\mathcal{O}(N^2)$ vlastně nepotřebujeme, neboť je možné rovnou při průchodu seznamem počítat výsledek.

Jaká je časová složitost algoritmu? Nejdříve načteme vstup, to trvá $\mathcal{O}(N)$, poté setřídíme seznam dolních a horních hran skvrn, což lze zvládnout v $\mathcal{O}(N \cdot \log N)$, poté tento seznam projdeme tak, že každou hranu zatřídíme/odebereme do/ze seznamu skvrn v aktuálním pruhu, což trvá $\mathcal{O}(N)$. Tento seznam pak procházíme, přičemž každý bod vložíme/odebereme do/z haldy $\mathcal{O}(\log N)$.

Dohromady tedy $\mathcal{O}(N) + \mathcal{O}(N \cdot \log N) + \mathcal{O}(N) \cdot (\mathcal{O}(N) + \mathcal{O}(N) \cdot \mathcal{O}(\log N)) = \mathcal{O}(N^2 \cdot \log N)$. Paměťová složitost je pak $\mathcal{O}(N)$.

Algoritmus je implementovaný v jazyku C++, neboť ten obsahuje knihovny pro pohodlnou práci se zmíněnými datovými strukturami. Aby byl program jednodušší, je tričko považováno za jednu velkou skvrnu s barvou 0. Samotný převod na původní zadání je pak udělán neefektivně, ale ve výsledné časové složitosti se to již neprojeví.

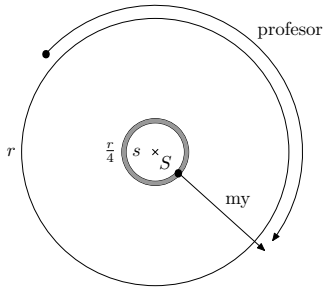
Zbyněk Falt & Petr Kratochvíl

21-2-2 Útěk před zkouškou

Analýza je velice komplikovaná věc a jak se ukázalo, hrozící zkouška zamotala nejednomu řešiteli hlavu. Přitom upláchnout je otázkou života a smrti! Došlá řešení by šla rozdělit do dvou skupin. V první skupině byla řešení, která tvrdila, že ať bude matfyzák snažit sebevíc, nedokáže se zkoušce vyhnout. Bohužel argumentace byla vedena způsobem: „Nenašel jsem řešení, proto neexistuje!“ Takový postup je zcela chybný. O tom svědčí i to, že druhá skupina řešitelů objevila způsob, jak upláchnout a zkoušce se vyhnout. Ukážeme si, jak vyzrát nad profesorem analýzy!

Označme si r poloměr rotundy. Předpokládejme, že profesor běží rychlostí 4 za časovou jednotku a student pouze 1. Pokud se student nachází hodně blízko středu rotundy S, je schopen obíhat po menší kružnici (se středem S) rychleji než profesor po obvodu. Jeho úhlová rychlost je větší jak profesorova. Jaký je maximální poloměr, který menší kružnice může mít? Délka obvodu roste lineárně s poloměrem. Pokud je její poloměr roven $r/4$, bude běhat stejně rychle jako profesor. Pokud bude (byť jen o malinko) menší, bude běhat rychleji.

Nyní si představme, že bychom byli kousek od středu a navíc profesor by byl přesně na opačném konci rotundy (střed S by ležel na úsečce mezi námi a profesorem). Rádi bychom se vydali přímo ke kraji rotundy, tedy na opačnou stranu než stojí profesor. Jak daleko od středu musíme být, abychom mu upláchli? Nechť jsme ve vzdálenosti s. Profesor doběhne na druhou stranu za čas $\pi r/4$, zatímco nám to zabere čas $r - s$. Tedy pokud $r - s < \pi r/4$, podaří se nám upláchnout.



Pro lepší představu se podívejte na obrázek. Existuje vzdálenost od středu, která splňuje obě výše uvedené nerovnosti současně, ta je vyznačena šedým pásem. Můžeme provést nejprve první krok, dostat se na opačnou stranu než profesor. Poté provedeme druhý krok a utečeme profesorovi, jak je naznačeno na obrázku šipkami. Ať se profesor pohybuje jakkoliv, nemůže nám v ani jednom kroku zabránit. Před zkouškou jsme šťastně zachráněni!

Pavel Klavík

21-2-3 Fronta

Jak je vidět z došlých řešení, někteří z vás jsou rození matfyzáci a v tlačenici matfyzáckého života nebudou mít žádné problémy. Došlo i pár rozpačitých řešení, ale jejich autoři nemusí věšet hlavu – ne vždy je na škodu stát ve frontě druhý. Nyní se společně podívejme, jak se měla úloha řešit.

Zatímco se matfyzáci ve frontě dohadují, předbíhají a strkají, zkusme si jejich situaci matematicky popsat. Matfyzáci sami představují množinu a pokud jsou vztahy mezi nimi rozumné (tj. neexistuje v nich cyklus), můžeme hovořit dokonce o *částečně uspořádané množině* (zkráceně *ČUM*). ČUM se velice dobře reprezentuje orientovaným grafem, kde vrcholy jsou prvky množiny a hrany představují vztahy (např. pokud si a myslí, že je chytřejší než b, pak existuje hrana z a do b).

V našem příkladu hledáme úplné (lineární) uspořádání částečně uspořádané množiny. Podíváme-li se na problém z hle-


```
   vhrany.push_back(vhrana_t(0,0,W,DOLNI,0));       // plocha trička je považována za nultý flek
   vhrany.push_back(vhrana_t(H,0,W,HORNI,0));
   barvy[0]=0;

   for (int i=1;i<=N;i++) {
      int ldr, lds, phr, phs, barva;

      scanf("%d%d%d%d%d",&lds,&ldr,&phs,&phr,&barva);
      vhrany.push_back(vhrana_t(ldr,lds,phs,DOLNI,i));
      vhrany.push_back(vhrana_t(phr,lds,phs,HORNI,i));
      barvy[i]=barva;
   }
   sort(vhrany.begin(),vhrany.end());
   list<shrana_t> shrany;
   vector<vhrana_t>::iterator vhrana;
   int vpozice=-1;
   for (vhrana=vhrany.begin();vhrana!=vhrany.end();++vhrana) {  // procházíme vodorovné hrany zdola nahoru
      list<shrana_t>::iterator shrana;
      if (vpozice>=0) {                             // pokud se nejedná o první hranu
         int vyska=vhrana->radek-vpozice;           // výška vodorovného pruhu
         int spozice=-1;
         set<int> halda;                            // ze standarní haldy nelze debírat libovolné prvky
         for (shrana=shrany.begin();shrana!=shrany.end();++shrana) {    // procházíme svislé hrany zleva doprava
            if (spozice>=0)
               obsahy[*halda.rbegin()]              // *halda.rbegin() je maximální prvek v haldě
                                          +=(shrana->sloupec-spozice)*vyska;

            if (shrana->typ == LEVA)                // přesně podle popisu řešení
                  halda.insert(shrana->flek);
            else
                  halda.erase(shrana->flek);
            spozice=shrana->sloupec;
         }
      }
      if (vhrana->typ == DOLNI) {                   // přesně podle popisu řešení
         list<shrana_t> pomoc;
         pomoc.push_back(shrana_t(vhrana->lsloupec,LEVA,vhrana->flek));
         pomoc.push_back(shrana_t(vhrana->psloupec,PRAVA,vhrana->flek));
         shrany.merge(pomoc);                       // zatřídíme svislé hrany do množiny
      } else {
         for (shrana=shrany.begin();shrana!=shrany.end();++shrana)
            while (shrana->flek==vhrana->flek)      // odstraníme příslušné hrany z množiny
               shrana=shrany.erase(shrana);
      }
      vpozice=vhrana->radek;
   }
   printf("Čistého trička zůstalo %d jednotek\n", obsahy[0]);

   for (int i=1;i<=N;i++)                           // neefektivní způsob, ale časovou složitost nezhorší
      if (barvy[i]!=0) {
         int barva=barvy[i];
         int celkem=0;
         for (int j=i;j<=N;j++)
            if (barvy[j]==barva) {
               celkem+=obsahy[j];
               barvy[j]=0;
            }
         printf("Barva %d zabíra %d jednotek\n",barva,celkem);
      }
   return 0;
}
```

Úloha 21-2-3 – Fronta – program

```
program fronta;
const max = 20000;
type pole = array [1..max] of integer;
    dpole = ^pole;
    matfyzak = record
        znamych: integer;
        znami: dpole; {seznam lidí ve frontě, které matfyzák zná}
        aktivni: integer; {informace o tom, je-li matfyzák již odbytý či ne}
    end;
    {pole matfyzáků / graf, reprezentovaný vrcholy se seznamy sousedů}
    matfyzaci = array[1..max] of matfyzak;
var usporadani: pole;
    mela: matfyzaci;
    vpozice: integer; {první volná pozice v poli uspořádání}
    kruznice: integer;
{průchod do hloubky, vrátí 0 pokud je vše OK, 1 pokud byla nalezena or. kružnice}
```

```
        int *plocha;            // plochy jednotlivych barev
        SKVRNA *skvrny = NULL;  // spojovy seznam skvrn
        SKVRNA *skvrny_kon;     // ukazatel na konec seznamu

        scanf("%d%d%d", &N, &W, &H);
        plocha = (int *)malloc((N+1)*sizeof(int));
        for (int i=0; i<N; i++) {
                if (skvrny == NULL) {   // pridame novou skvrnu na konec seznamu
                        skvrny = (SKVRNA *)malloc(sizeof(SKVRNA));
                        skvrny_kon = skvrny;
                } else {
                        skvrny_kon->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                        skvrny_kon = skvrny_kon->dalsi;
                }
                scanf("%d%d%d%d", &skvrny_kon->levy, &skvrny_kon->dolni,
                        &skvrny_kon->pravy, &skvrny_kon->horni, &skvrny_kon->barva);
                skvrny_kon->dalsi = NULL;
                // projdeme seznam skvrn a provedeme rozpady
                for (SKVRNA *stara = skvrny; stara != skvrny_kon; stara = stara->dalsi) {
                        rozpad(skvrny_kon, stara);
                }
        }
        for (int i=1; i<=N; i++)
                plocha[i] = 0;
        for (SKVRNA *s = skvrny; s != NULL; s = s->dalsi)
                plocha[s->barva] += (s->pravy-s->levy) * (s->horni-s->dolni);
        int ciste = W*H;
        for (int i=1; i<=N; i++) {
                printf("Barva %d zabira na tricku %d jednotek plochy.\n", i, plocha[i]);
                ciste -= plocha[i];
        }
        printf("Cisteho tricka zustalo %d jednotek.\n", ciste);
        return 0;
}
```

Úloha 21-2-1 – Špinavé tričko – rychlejší program

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <set>
#include <list>

#define HORNI 1
#define DOLNI 2
#define LEVA 1
#define PRAVA 2

using namespace std;

struct vhrana_t {                                   // vodorovná hrana skvrny
   int radek;
   int lsloupec;
   int psloupec;
   int typ;                                         // dolní nebo horní
   int flek;                                        // číslo fleku, kterému tato hrana přísluší
   vhrana_t(int _radek, int _lsloupec, int _psloupec, int _typ, int _flek)
      : radek(_radek), lsloupec(_lsloupec), psloupec(_psloupec), typ(_typ), flek(_flek) {}
   bool operator<(const vhrana_t &hrana) const {    // třídíme od zdola nahoru
      return radek<hrana.radek;
   }
};

struct shrana_t {                                   // svislá hrana skvrny
   int sloupec;
   int typ;                                         // levá nebo pravá
   int flek;                                        // číslo fleku, kterému tato hrana přísluší
   shrana_t(int _sloupec, int _typ, int _flek)
      : sloupec(_sloupec), typ(_typ), flek(_flek) {}
   bool operator<(const shrana_t &hrana) const {    // třídíme zleva doprava
      return sloupec<hrana.sloupec;
   }
};

int main() {
   int W,H,N;
   scanf("%d%d%d",&W,&H,&N);

   vector<vhrana_t> vhrany;                         // seznam všech vodorovných hran
   vector<int> barvy(N+1);                          // pro převod čísla fleku na jeho barvu
   vector<int> obsahy(N+1,0);                       // počet jednotek zabíraných fleky
```


diska grafu, kterým reprezentujeme ČUM, potřebujeme zavést číslování w, takové, že všem vrcholům je přiřazeno jedinečné číslo z rozsahu 1 až N (N je počet vrcholů) a pokud vede hrana z a do b, pak je $w(a) < w(b)$.

Postup, který nám vyrobí takové očíslování, se nazývá *topologické třídění* a je velmi dobře popsán v řadě učebnic programování. Existují dva známé a velmi jednoduché algoritmy, které řeší tento problém. První z nich je založený na odtrhávání vrcholů, ze kterých už nevede žádná hrana, a naleznete jej podrobně popsaný v knize Algoritmy a programovací techniky. Druhý, který využívá prohledání grafu do hloubky, najdete v naší kuchařce na téma grafy. Oba zvládnou najít uspořádání vrcholů v čase $\mathcal{O}(N + M)$, kde N je počet vrcholů a M je počet hran.

Shodou okolností je zde uvedená časová složitost také dolním odhadem, protože každý vrchol musíme očíslovat ($\mathcal{O}(N)$) a každou hranu musíme vzít v úvahu ($\mathcal{O}(M)$), jinak nemůžeme zaručit, že jsme ověřili všechny podmínky na uspořádání.

Neboť jsou programátoři pěkní lenoši, ukážeme vám ten druhý, který je o hroší chlup kratší. A jak tento algoritmus funguje? Vybereme si některý ještě neprošlý vrchol v a spustíme na něj prohledávání do hloubky. Po chvilce dumání odhalíme, že pokud očíslujeme nejdříve všechny potomky (z hlediska průchodu DFS) a až nakonec sebe, dostaneme topologické uspořádání začínající vrcholem v. Číslování ale musíme provádět „odzadu“ – tj. od čísla N směrem dolů. Zbydou-li nám ještě některé neprošlé vrcholy, tak je opět zpracujeme pomocí DFS a číslování nám je zařadí ještě před vrchol v.

Abychom vyřešili i okrajové případy, zbývá ještě rozhodnout, kdy žádné uspořádání neexistuje. To se stane právě tehdy, když je v grafu alespoň jeden orientovaný cyklus. Naštěstí jej odhalíme jednoduše při průchodu do hloubky. Pokud při prohledávání dojdeme do vrcholu, který je stále ještě „otevřený“ (DFS s ním ještě neskončilo), musí nutně existovat orientovaná kružnice obsahující tento vrchol. Zadaná množina pak nemůže být nijak uspořádána, takže nám nezbývá, než oznámit výsledek a skončit.

Technické detaily implementace můžete prozkoumat ve vzorovém řešení. Tímto se s vámi loučí dvojka Martinů a jeden CodEx.

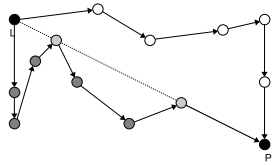
Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

21-2-4 Síťování

Lze soudit, že většině řešitelů se podařilo dostat síť alespoň do stavu, aby mohli odeslat řešení. Bohužel jsem ale při opravování měl pocit, že často se tak stalo jen díky vhodnému rozložení počítačů (obvykle se stávalo, že Vaše programy vygenerovaly nějakou posloupnost i v případě, že řešení neexistovalo).

Úspěšní řešitelé obvykle používali algoritmus založený na porovnávání úhlů mezi jednotlivými počítači. Konkrétně vybrali jeden (např. nejlevější a pokud jich je víc, tak nejspodnější z nich) počítač Y a ostatní setřídili dle úhlu, který svírala jejich spojnice s Y s nějakým pevným směrem (obvykle kladnou poloosou x). Pokud se sešly 2 počítače na stejném úhlu, rozhodovala vzdálenost od Y. V tomto pořadí počítače zapojili a Y zařadili mezi první a poslední v setřízené posloupnosti. Snadno se nahlédne, že takto vygenerovaná spojnice se nekříží (Buď počítače A_i a A_{i+1} mají stejný úhel vzhledem k Y a pak, díky setřízení dle vzdálenosti není žádný počítač X se stejným úhlem a vzdáleností $|A_iY| < |XY| < |A_{i+1}Y|$, nebo mají úhel různý ale pak zřejmě úhly mezi A_i a A_{i+1} protne generovaná spojnice jen jednou a to právě na drátu vedoucím z A_i do A_{i+1} . Podobně to dopadne i pro dráty vedoucí z počítače Y).

Nicméně ukážeme si zde jiný přístup k problému. Idea je taková, že vezmeme spojnici nejlevějšího a nejpravějšího počítače a spojíme počítače nad a pod touto přímkou zvlášť. Najdeme tedy nejlevější (a nejvyšší v případě, že je více počítačů s nejmenší x souřadnicí) počítač L a nejpravější (a nejnižší, pokud jich je více) počítač P. Pak roztřídíme zbylé počítače na ty, co jsou *Nad*, *Pod* a *Na* přímce *L*–*P* (na ilustrativním obrázku bílé, šedé, resp. světle šedé). Pokud budou všechny počítače na této spojnici, tak zřejmě řešení neexistuje (přímé spojení PL při návratu se protne s dráty vedoucími opačně). Jinak lze síť udělat, jen si musíme dát pozor, co provedeme s počítači *Na* spojnici — když *Nad* i *Pod* spojnicí jsou nějaké počítače, je jedno, jestli je přidáme k cestě z L do P (tedy k počítačům *Nad* spojnicí) nebo k cestě z P do L (tedy k počítačům *Pod* spojnicí) (na ilustraci jsme počítače *Na* spojnici přidali k počítačům *Pod* spojnicí). Pokud ale *Nad* (analogicky *Pod*) spojnicí nejsou žádné počítače a počítače *Na* spojnici bychom přidali na cestu z P do L (z L do P) dostali bychom protnutí právě v bodech, které jsou *Na* spojnici. Proto musime v tomto případě počítače *Na* spojnici uvažovat jako kdyby byly *Nad* (*Pod*) spojnicí ($Nad' = Nad \cup Na$,resp. $Pod' = Pod \cup Na$).



Nakonec už jen zbývá seřadit počítače *Nad* spojnicí a *Pod* spojnicí tak, aby vytvořili cestu, která se nebude protínat. Na to ale stačí setřídit počítače dle souřadnice x vzestupně a v případě, že se se nachází víc počítačů na stejném sloupci, tak dle y sestupně (jak dopadne setřízení je na obrázku naznačeno šipkami). Takto vzniklá spojnice se nevrací ve směru x a při stejném x používá třízení dle y a proto opět nedojde k protnutí. Podobná situace nastane u L (P) díky volbě nejvyššího (nejnižšího) počítače s minimální (maximální) x souřadnicí. „Kružnici“ z počítačů pak vytvoříme spojením L, setřízených počítačů *Nad*, P a obrácením setřízených počítačů *Pod*.

Pavel Čížek

21-2-5 Nádobí

Pohled na talíře a hrnky evokuje u většiny matfyzáků myšlenku na jídlo a následné kručení v břiše. Proto není divu, že danou problematiku řeší pomocí hladového algoritmu. Hladový algoritmus (angl. greedy) vybírá v každém kroku to aktuálně nejlepší řešení. Ne vždycky se dobere toho optimálního, ale v tomto případě funguje. Rozhodování o tom, který kus nádobí kdy umýt, probíhá odzadu (tj. ode dne, do kterého „přežije“ to nejtrvanlivější). Pro každý den se vezmou všechny kusy nádobí, které do něj vydrží, a zároveň jejich umývání ještě nebylo naplánováno na později. Z nich se hladově vybere ten nejcennější a naplánuje se na tento den k umytí. Takto nalezneme nějaké řešení, ale ještě nemusí být úplně jasné, že je skutečně optimální. Zkusme si to tedy dokázat.

K dokázání správnosti použijeme techniku, která nám může pomoci i se spoustou jiných algoritmů, které postupně konstruují optimální řešení. Ze začátku algoritmus běžel správně. Nerozhodli jsme totiž ještě o umytí jediného kusu, proto naše rozhodnutí nemohlo být špatně. Poté algoritmus prováděl jednotlivé kroky a nakonec vydal nějaký výsledek. Předpokládejme pro spor, že by výsledek byl špatně, tedy nebyl by optimální. Potom musel existovat nějaký krok, kdy se algoritmus poprvé rozhodl špatně, tedy rozhodl umýt kus nádobí A, který se nevyskytoval v žádném optimálním řešení. Vezměme si tedy jedno z optimálních řešení, které vyhovovalo všem rozhodnutím provedených v předcházejících krocích, a v daném kroku umývá kus B. Ukážeme, že z tohoto řešení dokážeme vyrobit jiné optimální řešení, které umývá kus A, to by byl jistě spor. Pokud optimální řešení umývalo kus A v nějakém dalším kroku, pouze prohodíme pořadí umývání A a B. Pokud není nikde umývání A naplánováno, tak umyjeme místo B kus A. Platí však, že cena kusu A je alespoň tak velká jako cena B. Nově vytvořené řešení je optimální a zároveň omývá ve špatném kroku A, což je spor.

Ještě jedna drobná poznámka na okraj: co kdybychom k řešení použili hladový algoritmus, ale rozhodovali o umývání v opačném pořadí od prvního dne. Takové řešení by nefungovalo, například pro vstup $(2, 2)$ a $(1, 1)$. Můžete si vyzkoušet jako malé cvičení nalézt místo, ve kterém by výše uvedený důkaz nezafungoval.

Co se týče implementace, nejprve si nádobí utřídíme sestupně podle trvanlivost např. pomocí QuickSortu v čase $\mathcal{O}(N \log N)$. Výběr nejdražšího kusu uděláme haldou, uspořádanou dle ceny. Do té vždy přidáme kusy, které vydrží do aktuálně zkoumaného dne, z vrchu odebereme ten nejdražší a dáme ho k mytí. V haldě dokážeme dělat operace vkládání i odebírání v čase $\mathcal{O}(\log N)$ na prvek, což nám dohromady dává $\mathcal{O}(N \log N)$, tedy i složitost celého algoritmu je $\mathcal{O}(N \log N)$. Plány, co umít v jednotlivé dny, si udržujeme v poli. Protože některé kusy mohou mít obrovskou trvanlivost ve srovnání s N, maximální počet dní, které nás zajímají, je minimum z N a maximální trvanlivosti. Další dny už stejně budeme mít celý dřez volný!

Pavel Klavík & Kristýna Stodolová

21-2-6 Nejkratší opět vyhrává

Úloha první s jedním chybějícím číslem vám nečinila velké problémy. V zásadě se objevila řešení dvě. První využívalo funkce `xor`. Tato funkce totiž splňuje `x ^ x = 0`, takže `xor`y dvou stejných čísel se vyruší. Navíc při `xor`ování nezáleží na pořadí, takže řešení je nasnadě: pokud `zxor`ujeme všechna čísla $1..N$ a všechna čísla $A[0]..A[N-2]$, výsledek je přesně chybějící číslo. To proto, že chybějící číslo jsme `xor`ovali jednou a všechna ostatní čísla dvakrát. Pokud chceme mít řešení co nejkratší, ještě si uvědomíme, že $A[N-1] = 0$. Získáme tak následující program o pěti instrukcích:

```
dalsi:  x = x ^ A[i]            # zde se xoruje A[0]..A[N-1]
        i = i + 1
        x = x ^ i               # zde se xoruje 1..N
        if i < N => jump dalsi
        write x
```

Druhé řešení první úlohy používá místo `xor`u sčítání a odčítání. Pokud k jedné proměnné přičteme všechna čísla $1..N$ a odečeteme čísla $A[0]..A[N-2]$, dostaneme chybějící číslo. Zde se ale někteří řešitelé zarazili – pokud je $N > 2^{31}$, tak $N + (N-1) > 2^{32}$ a při sčítání dojde k přetečení! A při odečátání zrovna tak! Velmi dobře, drahý Watsone, ale i když dojde k přetečení, pořád budeme mít výsledek spočítaný modulo 2^{32}. Taková přesnost nám ovšem stačí, protože chybějící číslo $\leq N < 2^{32}$. Získáme tedy alternativní řešení opět o pěti instrukcích:

```
dalsi:  x = x - A[i]            # zde se odčítá A[0]..A[N-1]
        i = i + 1
        x = x + i               # zde se přičítá 1..N
        if i < N => jump dalsi
        write x
```

Ještě jedna poznámka. Někteří pokročilí matematici využívali faktu, že $1 + \ldots + N = N(N+1)/2$. Jenomže program `S = N+1; S = S*N; S = S/2` nespočítá výsledek modulo 2^{32}, ale jenom modulo 2^{31}. Problém je v dělení – pokud máte v $S = N(N-1)$ modulo 2^{32}, tak po provedení $S/2$ je v S hodnota $N(N-1)/2$ modulo 2^{31}. Takový program tedy nefunguje správně.

Nyní k řešení druhé úlohy. Rádi bychom nějak aplikovali naše řešení úlohy první. To bychom mohli, pokud bychom dokázali čísla $1..N$ rozdělit do dvou skupin, aby v každé bylo právě jedno chybějící číslo. Pak by stačilo použít `xor`ovací řešení na každou skupinu zvlášť.

Nejprve z`xor`ujeme všechna čísla $1..N$ a $A[0]..A[N-3]$. Tím dostaneme `xor` obou chybějících čísel. Tato hodnota má jednotkové bity tam, kde se chybějící čísla liší. Nechť je tedy b-tý bit této hodnoty jedničkový. Chybějící čísla se liší v b-tém bitu. Pokud rozdělíme čísla $1..N$ a $A[0]..A[N-3]$ do dvou skupin podle toho, jakou mají hodnotu b-tého bitu, bude v každé skupině právě jedno chybějící číslo. Jedno z nich pak dostaneme tak, že z`xor`ujeme všechny hodnoty $1..N$ a $A[0]..A[N-3]$ s nulovým bitem b, a druhé tak, že z`xor`ujeme hodnoty s jedničkovým bitem b.

Nyní jak to provést na co nejméně instrukcí. Nejprve musíme v `xor`u chybějících čísel najít jeden z jeho jedničkových bitů. Označme `xor` chybějících čísel jako x a zapišme ho ve dvojkové soustavě:

$$\begin{aligned} x &\ \ bbbbbbb1000 \\ \text{dvojkový doplněk } x\text{, tj } -x &\ \ \overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}1000 \\ x\&(-x) &\ \ 0000001000 \end{aligned}$$

Vidíme, že `x & (-x)` má nastavený jediný bit, a to nejnižší bit, který je nastaven v x.

Zbývá vyřešit poslední detail. Jak rozdělit hodnoty podle nějakého bitu? Pokud je v b nastaven jeden bit, můžeme použít bitovou selekci, protože `x @ b` je přesně hodnota tohoto bitu, 0 nebo 1. Touto hodnotou pak můžeme indexovat pole `X`, takže nemusíme používat instrukci skoku a chybějící čísla budou v `X[0]` a `X[1]`.

Použitím všech těchto triků dostaneme program o 14 instrukcích:

```
alpha:  x = x ^ A[i]            # zde se xoruje A[0]..A[N-1]
        i = i + 1
        x = x ^ i               # zde se xoruje 1..N
        if i < N => jump alpha
        # nyní je v x xor chybějících čísel

        y = 0 - x
        b = x & y
        # v b je nastaven jediný bit,
        # a to takový, kde se chybějící čísla liší

beta:   i = A[j] @ b            # A[j] je ve skup. i (0 či 1)
        X[i] = X[i] ^ A[j]      # xoruj A[j] ve skupině i
        j = j + 1
        i = j @ b               # j je ve skupině i (0 či 1)
        X[i] = X[i] ^ j         # xoruj čísla j ve skupině i
        if j < N => jump beta
        write X[0]
        write X[1]
```

Někteří řešitelé se pokusili řešit druhou úlohu tak, že použili sčítací řešení a spočítali si součet chybějících čísel. Z tohoto součtu spočítali průměr chybějících čísel. Nakonec rozdělili čísla $1..N$ a $A[0]..A[N-3]$ na čísla menší nebo rovná průměru a větší než průměr. Takto také rozdělili čísla do svou skupin, z nichž každá obsahuje jedno chybějící číslo. Problém je jenom s přesností – pokud známe součet chybějících čísel modulo 2^{32}, tak průměr, tj. součet děleno dvěma, známe jenom modulo 2^{31}, takže algoritmus bez ošetření přetékání nefunguje.

Martin Mareš & Milan Straka

Úloha 21-2-1 – Špinavé tričko – pomalejší program

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

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))
#define prunik(xl, xp, yl, yp) (((xl <= yl && xp > yl) || (yl <= xl && yp > xl))?1:0) // je prunik dvou intervalu neprazdny?

typedef struct skvrna {                 // udaje o skrvne
        int levy, pravy, horni, dolni;  // okraje skvrny
        int barva;
        struct skvrna *dalsi;           // ukazatel na dalsi skrvnu ve spojovem seznamu
} SKVRNA;

void rozpad(SKVRNA *nova, SKVRNA *stara) { // rozpadne starou skvrnu a misto ni vytvori spojovy seznam rozpadlych casti
        SKVRNA *seznam, *konec; // spojovy seznam vzniklych casti; konec seznamu
        if (!prunik(stara->levy, stara->pravy, nova->levy, nova->pravy)
                        || !prunik(stara->dolni, stara->horni, nova->dolni, nova->horni))
                        return;         // pokud se skvrny neprekryvaji, nemame co delat
        seznam = konec = (SKVRNA *)malloc(sizeof(SKVRNA));      // na zacatek seznamu dame starou skvrnu
        *seznam = *stara;
        seznam->dalsi = NULL;
        if (stara->horni > nova->horni) {       // urizneme horni kus
                konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                konec = konec->dalsi;
                konec->horni = stara->horni;
                konec->dolni = nova->horni;
                konec->levy = stara->levy;
                konec->pravy = stara->pravy;
                konec->barva = stara->barva;
                konec->dalsi = NULL;
        }
        if (stara->dolni < nova->dolni) {       // urizneme spodek
                konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                konec = konec->dalsi;
                konec->horni = nova->dolni;
                konec->dolni = stara->dolni;
                konec->levy = stara->levy;
                konec->pravy = stara->pravy;
                konec->barva = stara->barva;
                konec->dalsi = NULL;
        }
        if (stara->levy < nova->levy) {         // urizneme levy kus
                konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                konec = konec->dalsi;
                konec->horni = min(nova->horni, stara->horni);
                konec->dolni = max(nova->dolni, stara->dolni);
                konec->levy = stara->levy;
                konec->pravy = nova->levy;
                konec->barva = stara->barva;
                konec->dalsi = NULL;
        }
        if (stara->pravy > nova->pravy) {       // urizneme pravy kus
                konec->dalsi = (SKVRNA *)malloc(sizeof(SKVRNA));
                konec = konec->dalsi;
                konec->horni = min(nova->horni, stara->horni);
                konec->dolni = max(nova->dolni, stara->dolni);
                konec->levy = nova->pravy;
                konec->pravy = stara->pravy;
                konec->barva = stara->barva;
                konec->dalsi = NULL;
        }
                konec->dalsi = stara->dalsi;    // napojime seznam na puvodni pokracovani
                *stara = *(seznam->dalsi);      // a vynechame rozpadnutou skvrnu
}

int main(void) {
        int N;                  // pocet skvrn
        int W, H;               // rozmery tricka
```