Výsledková listina dvacátého prvního ročníku KSP po páté sérii

		škola	*ročník*	*sérií*	*2151*	*2152*	*2153*	*2154*	*2155*	*2156*	*série*	*celkem*
1.	Vojtěch Kolář	G Neratov	3	11	10	10	7	7	10	14	44,0	200,0
2.	Vítězslav Plachý	GJiříPoděb	3	5	3	3	7	0	10	10	33,9	188,6
3.	Filip Hlásek	GMikul23PL	2	10	10		7		10	14	41,2	182,0
4.	Petr Čermák	GEBenešeKL	3	5	3	6	6	2	4	14	35,2	177,3
5.	Michal Bilanský	GLepařovJČ	3	5	3	5	7	10	8		35,1	157,3
6.	Pavel Veselý	G Strakon	4	17	10		7	4		14	31,8	144,9
7.	Jitka Novotná	G Bílovec	4	6							0,0	129,5
8.	Karel Tesař	SPŠE Plzeň	3	7	6						7,2	114,9
9.	Vlastimil Dort	GŠpitálsPH	3	15	10		7				16,1	113,9
10.	Jiří Cidlina	GVoděraPH	4	3							0,0	106,7
11.	Lukáš Ptáček	GJAKŽeliez	3	3							0,0	92,5
12.	Alexander Mansurov	GNVPlániPH	0	3							0,0	87,3
13.	Martin Zikmund	G Turnov	1	4							0,0	79,6
14.	Filip Štědronský	GMikul23PL	2	9	6						6,7	78,6
15.	Pavel Taufer	ArcibisGPH	3	7	3						4,2	76,9
16.	Jiří Setnička	G25březnPH	2	6	2	8					12,1	75,1
17.	David Věčorek	GTNovákBO	3	3							0,0	72,6
18.	Štěpán Šimsa	GJungmanLT	0	4	1	5	7		9		27,0	68,2
19.	Karel Král	G Most	3	5	3		6				12,2	56,8
20.	Jan Vaňhara	G Holešov	4	3							0,0	56,2
21.	Petr Pecha	SPŠSVsetín	2	6			7				7,9	54,5
22.	Alžběta Pechová	SPŠSVsetín	4	8		1	5		1		8,7	48,1
23.	Barbora Janů	GKepleraPH	2	4							0,0	44,3
24.	Jan Veselý	G Strakon	2	2		3					5,7	41,3
25.	Kateřina Lorenzová	G Česká ČB	2	5			9		1		10,9	38,6
26.	Libor Plucnar	GBezručeFM	4	11							0,0	38,2
27.	Filip Sládek	GNámestovo	3	1							0,0	38,1
28.	Radim Cajzl	GNoMěsNMor	2	20			6		3		4,7	37,7
29.	Karel Kolář	GŠpitálsPH	4	3							0,0	36,3
30.	Tomáš Pikálek	GBoskovice	2	1							0,0	35,6
31.	Lukáš Chmela	GJŠkodyPŘ	0	1							0,0	35,2
32.	Ondřej Pelech	GJNerudyPH	4	1							0,0	33,4
33.	Petr Zvoníček	G Slavičín	3	4	1						2,0	32,4
34.	David Formánek	GJarošeBO	2	2							0,0	27,9
35. – 36.	Karolína Burešová	G ČesLípa	2	1							0,0	27,7
	Jan Kostecký	VOŠŠumperk	2	3	2	3	7				17,7	27,7
37.	Milan Rybář	GJungmanLT	4	3							0,0	27,4
38.	Pavol Rohár	G Košice	3	2		3	8		4		21,2	27,2
39.	Honza Žerdík	G Příbor	4	1							0,0	25,7
40.	Jan Škoda	GMikul23PL	2	4							0,0	23,3
41.	Alena Bušáková	G Trutnov	2	1							0,0	22,6
42.	Hynek Jemelík	GJarošeBO	2	3	0						0,0	19,0
43.	Jakub Sochor	G Bílovec	4	1							0,0	17,1
44. – 45.	Martina Vaváčková	GCoubTábor	3	1	1		7		3		17,0	17,0
	David Vondrák	GDašickáPA	3	3	1						2,2	17,0
46.	Pavel Kratochvíl	VOŠGSvetla	1	7			3		1		5,7	15,9
47.	Martin Holec	G Slavičín	2	3							0,0	15,5
48.	Mirek Jarolím	GMikul23PL	3	4							0,0	15,3
49.	Petr Babička	VOŠGSvetla	4	8							0,0	11,6
50. – 52.	Jiří Daněk	GKřenováBO	3	1							0,0	10,0
	Martin Holeček	GMikul23PL	3	1	10						10,0	10,0
	Jan Matějka	G Jírov ČB	4	5							0,0	10,0
53.	Dominik Smrž	GOhradníPH	0	3							0,0	8,7
54.	Anna Chejnovská	GBNěmcovHK	2	1			7				8,5	8,5
55.	Jiří Keresteš	SPŠE Plzeň	3	5							0,0	7,4
56.	Jakub Zíka	GNadAlejPH	2	1	3						6,0	6,0
57.	Stanislav Fořt	GCoubTábor	1	8							0,0	5,9
58.	Igor Koníček	G UherBrod	3	1							0,0	5,7
59.	Ladislav Maxa	GKepleraPH	3	1							0,0	5,5

Milí řešitelé a řešitelky!

S pěnou u úst a utahaným, ale šťastným výrazem v očích jsme opravili i pátou sérii, zatímco vy si už zhusta začínáte užívat prázdniny. Zároveň začínáme připravovat soustředění, nejlepším z vás přijdou pozvánky. Máte se na co těšit.

Zatímco odpočíváte po letošním ročníku, můžete třeba komentovat letošní úlohy, pochválit pilné organizátory a mudrovat nad tím, co bychom pro vás ještě mohli učinit, abyste se v KSPéčku cítili jako doma.

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

Všechny vaše komunikační touhy si můžete splnit na fóru *http://ksp.mff.cuni.cz/forum/* a případně záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz*.

Vzorová řešení páté série dvacátého prvního ročníku KSP

21-5-1 Polomáčené mrakodrapy

Očekávali jsme více správných řešení – na rozdíl od minulé úlohy si tato úloha žádala spíše selský rozum než algebraické triky.

Z úlohy bylo patrné, že všechna maxima byla příliš velká na to, aby mělo triviální kvadratické řešení jakoukoli šanci. Stejně tak rozsah velikostí věžáků a rozsah dnů, na které mohl být položen dotaz, byl příliš velký, než aby s ním šlo dělat cokoli rozumného.

Nicméně si lze všimnout, že zde pracujeme jen s tisícinou položek z celého rozsahu. Zkusíme si tu množinu tedy předzpracovat, aby se nám s ní lépe pracovalo.

Způsob, který si popíšeme, využívá zajímavého panelákového pozorování: jediné budovy, které zvyšují nebo snižují počet ostrůvků ve městě, jsou ty, které jsou „lokálními extrémy“ – jinak řečeno, jsou to takové budovy, jejichž oba sousedé jsou buď zároveň větší nebo menší než ona sama. Když totiž uvážíme posloupnost rostoucích (nebo klesajících) paneláků, voda je postupně zaplavuje jeden po druhém a počet ostrovů se nezmění. Navíc si všimneme, že „vršek“ po zatopení od počtu ostrůvků odečte sám sebe (právě se zatopil), zatímco „údolí“ po zatopení oddělí svou levou a pravou část, a tedy jeden ostrůvek přibude.

Výška budovy přesně určuje den, kdy se přičte nebo odečte jednička z výsledku. Zjistit o každém domě, zda je „vrškem“ či „údolím“, zvládneme jedním průchodem vstupními daty. Při tomto průchodu si tedy rovnou budeme ukládat neuspořádaně dvojice (Den, Změna), kde Den bude výška domu, který procházíme a Změna bude $+1$ nebo -1, podle toho, jakého typu byl onen dům.

Nyní začneme ze vstupu načítat (velice příhodně setříděné) dny a budeme podle $+1$ a -1 upravovat aktuální hodnotu. Abychom toto mohli dělat rychle, stačí si naše pomocná data setřídit podle hodnoty Den vzestupně (postačí i rychlý kvadratický algoritmus jako Quicksort) a pak obě pole procházet naráz.

Paměti nám stačilo lineárně mnoho, dokonce jsme využili jen jedno pole celých čísel ($+1$ bit, ale ten bychom dovedli zakódovat i dovnitř) o velikosti N. Pokud jsme nepoužili žádné „nefér“ praktiky jako přihrádkové třídění, pak nám celý algoritmus seběhl v čase $\mathcal{O}(N \log N)$, respektive $\mathcal{O}(N^2)$ (s rychlým kvadratickým tříděním).

Na závěr doporučujeme všem řešitelům praktických úloh, pokud používáte libovolný jazyk vyjma Pascalu, pak tento jazyk obsahuje zabudovanou třídící knihovnu, která pro velkou většinu vstupů bude alespoň tak rychlá jako libovolný algoritmus, který napíšete. Když ji budete používat (v praktických úlohách), tak nic nezkazíte a navíc si ušetříte spoustu písmenek a potenciálních chyb.

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

21-5-2 Banky

Z došlých řešení je zřejmé, že většina řešitelů nemá ráda chamtivé bankéře. Značná část z vás se je pokoušela zmást nesprávnými výsledky. Někteří pravděpodobně měli v plánu zopakovat Chuliův trik, a tak bance nabídli velmi líné programy, aby měli dostatek času na přípravu a provedení svých plánů. A jen pár jedinců překonalo svou nevoli vůči finančníkům a předvedli nejen funkční, ale i rozumně rychlé algoritmy. A jak že vůbec našel díru v systému Chulio? Podívejte se sami:

Systém převodů měn je vlastně orientovaný ohodnocený graf. Jednotlivé měny jsou vrcholy a pokud mezi dvěmi měnami existuje převodní kurz c, tak mezi odpovídajícími vrcholy existuje orientovaná hrana s ohodnocením c. Posloupnost převodů, která je pro banku prodělečná, odpovídá v grafu orientovanému cyklu s hranami $e_1, e_2, \ldots, e_k$, ve kterém je součin ohodnocení těchto hran větší než jedna, tedy

$$c(e_1) \cdot c(e_2) \cdot \cdots \cdot c(e_k) > 1.$$

Zadaná úloha se dost podobá úloze hledání cyklu, který má součet ohodnocení hran záporný, pro kterou je známý rychlý algoritmus. Na ni naši úlohu převedeme tak, že výše uvedenou nerovnici zlogaritmujeme a vynásobíme -1:

$$-\log(c(e_1) \cdot c(e_2) \cdot \cdots \cdot c(e_k)) < -\log 1,$$
$$(-\log c(e_1)) + (-\log c(e_2)) + \cdots + (-\log c(e_k)) < 0.$$

Každou hranu e tedy místo $c(e)$, což je převodní kurz mezi odpovídajícími měnami, ohodnotíme $-\log c(e)$. Toto nové ohodnocení budeme pro zjednodušení následujícího textu nazývat délkou hrany a délka sledu pak bude součet délek jeho jednotlivých hran. (Sled je posloupnost vrcholů taková, že mezi sousedními vrcholy sledu vede v grafu hrana ve správném směru. Na rozdíl od cesty se v něm mohou opakovat jak vrcholy, tak hrany.)

V upraveném grafu zjišťujeme, jestli tam existuje cyklus (sled, který má první a poslední vrchol stejný) záporné délky. Jak na to? Použijeme Bellman-Fordův algoritmus, který hledá nejkratší sledy z nějakého vrcholu u do všech ostatních. Funguje tak, že si pro každý vrchol pamatuje délku zatím nejkratšího nalezeného sledu z u do něj. V každém kroku pro každou hranu vw určí, jestli není součet délky zatím nejkratšího nalezeného sledu z u do v a délky hrany vw menší, než zatím nejkratší nalezený sled z u do w. Pokud ano, tak ji na tuto délku sníží. Na začátku nastavíme vzdálenost u na nulu a všech ostatních vrcholů na ∞.

Po provedení i-tého kroku budeme znát pro každý vrchol délku nejkratšího sledu obsahujícího nejvýše i hran z u do tohoto vrcholu. Pokud z vrcholu u není dosažitelný žádný cyklus záporné délky, tak nejpozději po $n-1$ krocích (n je počet vrcholů grafu) najde algoritmus pro každý vrchol nejkratší sled z u (každý z těchto sledů bude cesta) a pokud provedeme ještě n-tý krok, tak proběhne „naprázdno“, tj. nenajde žádný nový kratší sled. Pokud graf obsahuje cyklus záporné délky dosažitelný z u, tak v každém kroku (a tedy i v n-tém) najde nějaký nový kratší sled.

Zbývá ještě vymyslet, který vrchol zvolíme za počáteční vrchol u. Vzhledem k tomu, že nemusí existovat žádný vrchol, ze kterého by byly dosažitelné všechny ostatní, bude u nový vrchol, ze kterého povedou hrany do všech ostatních, jejich délku zvolíme třeba nulovou. Protože do nového vrcholu nevedou žádné hrany, nepřidali jsme do grafu žádný cyklus a výsledek jsme tedy nezměnili. Kromě prvního kroku můžu hrany vedoucí z u a tedy i samotný vrchol u, ignorovat, protože určitě nezpůsobí změnu vzdálenosti žádného vrcholu. Naopak v prvním kroku stačí počítat jen s hranami vedoucími z u. Protože po prvním kroku budou mít všechny vrcholy vzdálenost nulovou, nemusím v programu ani vrchol u ani hrany z něj vedoucí nijak reprezentovat a stačí jen nastavit vzdálenost všech vrcholů na nulu.

Nakonec si uvědomíme, že převod kurzů na délky, který jsme provedli na začátku, je vlastně zbytečný a můžeme tedy počítat přímo s kurzy. Pro každý vrchol si budeme pamatovat prozatím nejlepší (maximální) kurzový součin na sledu z u a kurzový součin ve vrcholu w budeme upravovat, pokud je menší než součin kurzového součinu vrcholu v a kurzu hrany vw. Kurz na hranách z u můžeme nastavit třeba na jedničku (a ve zkrácené podobě prvního kroku teda nastavit všem vrcholům kurzový součin na jedničku).

V prvním kroku jen nastavíme hodnotu kurzového součinu všem vrcholům. V každém dalším kroku projdeme všechny hrany a pro každou provedeme $\mathcal{O}(1)$ operací. Celková časová složitost tedy je $\mathcal{O}(n+nm) = \mathcal{O}(nm)$, kde n je počet vrcholů a m je počet hran v grafu. V paměti máme informace o každé hraně a vzdálenost každého vrcholu, takže paměťová složitost je $\mathcal{O}(n+m)$.

Petr Onderka

21-5-3 Krávy

Naším úkolem je přehradit všechny krávy v kravíně Z závorami, aby součet jejich délek byl minimální. Na tento problém se dá koukat dvěma způsoby.

První způsob: Na začátek si představme, že před každou krávou je jedna závora o délce jeden telemetr. Pokud je počet použitých závor rovný nebo menší zadanému číslu Z, jsme s problémem hotovi. Pokud jsme ale použili víc závor, než můžeme, musíme některé sousední závory „spojit“, čímž se nám sníží počet použitých závor o jedna. Protože součet délek závor má být minimální, spojujeme vždy co nejkratší úsek mezi kravami, až bude počet použitých závor rovný Z. Proč tento způsob funguje? V první řadě si musíme uvědomit, že každá kráva musí být ohrazena. Takže pokud jsme měli dostatek závor, je naše začáteční rozmístění určitě minimální. Pokud ale dostatek závor nemáme, musíme nutně jednou závorou přehradit více krav. No a jedné závory se zbavíme tak, že spojíme dvě sousední závory, čímž ale přehradíme taky mezeru mezi nimi. Proto je nejvýhodnější v každém kroku spojit závory s co nejkratší mezerou mezi sebou (mezera mezi sousedícími závorami má délku 0).

Druhý způsob: Na problém se ale můžeme dívat i z jiného hlediska. Mějme na začátku jednu dlouhou závoru přes celý kravín. Na začátek ořežeme kus před první a po poslední krávě. Pak už jen zbývá najít $Z-1$ nejdelších mezer mezi kravami, a když je „vyřežeme“, zbyde nám Z závor, a jejich celková délka je určitě minimální. Proč i tento způsob funguje? Začáteční ohrazení je určitě korektní. Taky ořezání kusu před první a za poslední krávou řešení nepokazí. Takže máme všechny krávy ohrazené, ale je možné, že nám zbyly nějaké závory. No a z jedné závory udělám dvě tak, že z ní „vyřežu“ nějaký kus ze středu. A aby byl součet délek nových dvou závor minimální, musím vyřezat co největší kus. Tento kus ale nesmí obsahovat žádnou krávu, protože by už nebyla ohrazena. Takže se snažím najít co největší kus, který neobsahuje žádnou krávu, což je přesně to, že se snažím najít co největší mezeru mezi kravami, která ještě nebyla vyřezaná. No a protože mám Z závor, tak si můžu dovolit vyřezat až Z-1 kusů, no a tyto kusy musí být co největší.

Nechť vstup vypadá následovně: Na začátek jsou mezerami oddělená čísla N K Z, a pak následuje K čísel, které značí pozice krav v kravíně. Pro jednoduchost ať jsou pozice seřazeny vzestupně. Takže teď nám jenom zbývá najít $K-Z$ nejkratších, případně $Z-1$ nejdelších mezer mezi kravami. Rozeberme třeba hledání $Z-1$ největších. To můžeme udělat několika způsoby:

1. Všechny mezery setřídit podle velikosti a vybrat $Z-1$ největších. Tento způsob má při použití třídícího algoritmu QuickSort časovou složitost $O(K \log K)$ a paměťovou $O(K)$. Více informací o QuickSortu naleznete v Kuchařce 21-2 Rozděl a panuj.

2. Postupně načítat pozice krav, vždy spočíst mezeru a za použití struktury halda vybrat $Z-1$ největších. Halda je binární strom, kde pro každý prvek platí, že je menší než jeho následníci. Pro více podrobností viz Kuchařka 20-4 Halda, heapsort a Dijsktrův algoritmus. Stručně se algoritmus dá popsat následovně: Na začátek zatřídíme prvních $Z-1$ mezer do haldy. Pak vždy když načteme nový neobydlený úsek, porovnáme jeho délku s minimem na vrcholu haldy. Když je délka menší nebo rovna, úsek zahodíme, protože ostatní délky úseků v haldě jsou zjevně taky větší nebo rovny. Když je ale současný úsek delší než minimum v haldě, tohle minimum vyhodíme, a zařadíme nový prvek do struktury. Na konci nám zůstane halda, ve které máme $Z-1$ největších úseků mezi kravami. Tento algoritmus běží v časové složitosti $O(K \log Z)$ a paměťové $O(Z)$.

3. Nejrychlejší způsob - použijeme algoritmus na hledání K-tého nejmenšího prvku z Kuchařky 21-2 Rozděl a panuj. V našem případě jenom otočíme nerovnosti a budeme hledat $(Z-1)$-ní největší prvek. Tenhle algoritmus nám během počítání generuje rovnou všechny větší prvky, které sice nejsou vzájemně setříděné, ale to nám nijak nevadí. Tento algoritmus má časovou i paměťovou složitost $O(K)$.

Mária Vamošová

21-5-4 Zákony

Ze spleti zákonů a soudních pří by se ti z vás, co se o to pokusili, nakonec vymotali, leč většinou by Chosému nechali dostatek času na složení šavle a museli by k ujasnění sporu o farmu (a Ochechulínu) využít jiných prostředků.

Při řešení úlohy si nejdříve můžeme všimnout, že ji lze přeformulovat ještě tak, že z lesa chceme dostat libovolně (nekonečně) malou kouli (tedy vlastně bod), ale mezi dvojice


```
for (j = 0; j <= velikost_pouzdra; j++) {
    /* Pokud jsme narazili na značku,
     * tak zkusíme přidat další díl */
    if (pouzdro[aktualni][j] == i) {
        /* Přidávaný díl nesmí vyčnívat
         * z pouzdra na jedné straně */
        if (j - segment >= 0) {
            /* Do pouzdra pro příští tah
             * přidáme značku konce */
            pouzdro[dalsi][j - segment] = i+1;
            zmena = 1;
        }
        /* Přidávaný díl nesmí vyčnívat
         * ani na druhé straně */
        if (j + segment <= velikost_pouzdra) {
            pouzdro[dalsi][j + segment] = i+1;
            zmena = 1;
        }
    }
}
```

```
        }
    }
    /* Prohodíme pole pouzdra */
    j = aktualni; aktualni = dalsi; dalsi = j;

    /* Pokud segment nikam nepřidáme,
     * tak se šavle nedá složit */
    if (zmena == 0) {
        printf("Tuto šavli do pouzdra nesložíte.\n");
        return 0;
    }
}

printf("Tuto šavli dokážeme do pouzdra složit.\n");
return 0;
```

```
   if (na > nb) {
      median = rand()%na;
      med_smer = a[median].s[1]/a[median].s[0];
   } else {
      median = rand()%nb;
      med_smer = b[median].s[1]/b[median].s[0];
   }
   /* Rozdělení na části a rekurze */
   int ma = bi;
   for (int i = 0; i < na; i++)
      if (med_smer >= a[i].s[1]/a[i].s[0])
         buf[bi++] = a[i];
   int mb = bi;
   for (int i = 0; i < na; i++)
      if (med_smer < a[i].s[1]/a[i].s[0])
         buf[bi++] = a[i];
   int mc = bi;
   for (int i = 0; i < nb; i++)
      if (med_smer >= b[i].s[1]/b[i].s[0])
         buf[bi++] = b[i];
   int md = bi;
   for (int i = 0; i < nb; i++)
      if (med_smer < b[i].s[1]/b[i].s[0])
         buf[bi++] = b[i];
   unsigned char vysledek =
      spojene(buf+ma, buf+md, mb-ma, bi-md, 1, 0) |
      spojene(buf+mb, buf+mc, mc-mb, md-mc, 1, 0) |
      spojene_kolem_pocatku(buf+ma, buf+mc, mb-ma, md-mc) |
      spojene_kolem_pocatku(buf+mb, buf+md, mc-mb, bi-md) ;
   bi = ma;
   return vysledek;
}

/* Projde graf bloků a rozhodne, jestli je počátek uvnitř
 * nějakého polygonu tvořeného hranami cyklu */
bool projit_graf(int a, bool pruseciku) {
   if (bloky[a].navstiveno)
      return bloky[a].pruseciku != pruseciku;
   bloky[a].navstiveno = true;
   for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
         if (!bloky[a].hrany[i][j]) continue;
         if (bloky[a].hrany[i][j] == 3) return true;
         if (projit_graf(najit_blok(bloky[a].s[0]+2*j-4,
                                    bloky[a].s[1]+2*i-4),
                 pruseciku != (bloky[a].hrany[i][j]&1)))
            return true;
      }
   }
   return false;
}
int main(void)
{
   /* Načteme vstup a setřídíme body do skupin podle bloků,
    * do nichž patří; jednotlivé skupiny pak lexikograficky
    * podle souřadnic */
   float r, poc[2];
   scanf("%d%f%f%f", &n, &r, poc, poc+1);
   for (int i = 0; i < n; i++) {
      for (int s = 0; s < 2; s++) {
         scanf("%f", body[i].s+s);
         body[i].s[s] -= poc[s]; body[i].s[s] /= r/2;
         body[i].b[s] = (int)(body[i].s[s]/2)*2
                  + (body[i].s[s] < 0 ? -1 : 1);
      }
   }
   body[n].b[0] = body[n].b[1] = INT_MAX;
        /* zarážka, která zjednoduší podmínky dále */
   qsort(body, n, sizeof(*body), porovnat_bloky);
   /* Nejprve budeme hledat spojení s dvěma bloky napravo
    * postupně od každého bloku, takže body uvnitř bloku
    * potřebujeme mít seřazené podle y-ové souřadnice */
   por_sour = 1;
   for (int i = 0, j = 1; j <= n; j++) {
      body[j-1].blok = i;
      if (body[j].b[0] != body[j].b[0] ||
                          body[i].b[1] != body[j].b[1]) {
         bloky[i].s[0] = body[i].b[0];
         bloky[i].s[1] = body[i].b[1];
         bloky[i].bodu = j-i;
         qsort(body+i, j-i, sizeof(*body), porovnat_sour);
         for (int dx = -2; dx; dx++) {
            int ii = najit_blok(body[i].b[0]+2*dx,
                                          body[i].b[1]);
            if (ii < 0) continue;
            bloky[i].hrany[2][2+dx] =
                bloky[ii].hrany[2][2-dx] =
                     spojene(body+i, body+ii, bloky[i].bodu,
                        bloky[ii].bodu, 0, bloky[i].s[0]-1);
         }
         i = j;
      }
   }
  /* Teď budeme u každého bloku hledat spojení se zbývajícími
   * 10 bloky nad ním, k tomu setřídíme body podle x */
   por_sour = 0;
   for (int i = 0, j = 1; j <= n; j++) {
      if (body[i].b[0] != body[j].b[0] ||
                          body[i].b[1] != body[j].b[1]) {
         qsort(body+i, j-i, sizeof(*body), porovnat_sour);
         for (int dy = -2; dy; dy++) {
            for (int dx = -2; dx <= 2; dx++) {
               int ii = najit_blok(body[i].b[0]+2*dx,
                                   body[i].b[1]+2*dy);
               if (ii < 0) continue;
               bloky[i].hrany[2+dy][2+dx] =
                        bloky[ii].hrany[2-dy][2-dx] =
                  /* Hledáme-li hranu vedoucí "přes" počátek, */
                     (bloky[i].s[1] == 1 && dy == -1) &&
                     ((bloky[i].s[0] == -1 && dx == 1) ||
                           (bloky[i].s[0] == 1 && dx == -1)) ?
                           /* zavoláme speciální funkci, */
                           spojene_kolem_pocatku(
                                       body+i, body+ii,
                                       bloky[i].bodu,
                                       bloky[ii].bodu) :
                           /* jinak standardní */
                           spojene(body+i, body+ii,
                                       bloky[i].bodu,
                                       bloky[ii].bodu, 1,
                                       bloky[i].s[1]-1);
            }
         }
         i = j;
      }
   }
   for (int i = 0; i < n; i++) {
      if (!bloky[i].navstiveno && projit_graf(i, 0)) {
         printf("Nelze se obhájit.\n");
         return 0;
      }
   }
   printf("Je možné se obhájit.\n");
   return 0;
}
```

Úloha 21-5-5 – Cestovní šavle – program

```
#include <stdio.h>

#define MAXL 10001

int pouzdro[2][MAXL];

int main() {
    int pocet_segmentu, velikost_pouzdra;
    int i, j, segment, zmena, aktualni, dalsi;
    scanf("%d %d", &pocet_segmentu, &velikost_pouzdra);

    /* Napoprvé vynulujeme pole */
    for (i = 0; i <= velikost_pouzdra; i++) {
        pouzdro[0][i] = 0;
    }
    aktualni = 0; dalsi = 1;

    for (i = 0; i < pocet_segmentu; i++) {
        /* Nemusíme načítat vstup hned na začátku */
        scanf("%d", &segment);
        /* Musíme si pamatovat,
         * jestli jsme další díl někam umístili */
        zmena = 0;
```


zákonů, které jsou si blíže než $2R$, postavíme zeď. Máme tedy zadány vrcholy grafu, jen potřebujeme zjistit, mezi kterými vedou hrany, a nakonec se podívat, zda je naše výchozí pozice uvnitř nějakého cyklu.

Pokud bychom se spokojili s kvadratickým časem, stačí pro nalezení hran jednoduše vyzkoušet všechny dvojice vrcholů. Nepříjemné je, že rychleji to ani nejde, neboť až kvadratický může být počet hran, které chceme najít; pročež nezbývá než si graf trochu upravit.

Pro jednoduchost budeme dále předpokládat, že počáteční pozice je v bodě (0,0) a zadaný poloměr roven 2 (jinými slovy, přepočítáme souřadnice tak, aby to platilo). Rozdělíme si celou rovinu na čtverce se stranou 2 tak, aby počátek ležel v rohu nějakého z nich. Body uvnitř každého z těchto bloků jsou pak vždy všechny spojené hranou (tvoří úplný podgraf) a můžeme je sloučit do jednoho se souřadnicemi středu čtverce. Musíme si jen dát pozor, když takto slučujeme hrany, které procházejí kolem počátku z různých stran (přesněji to můžou být ty, které protínají kladnou poloosu y, a ty ostatní); v takovém případě zřejmě z lesa zákonů uniknou nemůžeme. Nicméně to vyřešíme později a prozatím předpokládejme, že k tomu nedojde.

Nový graf má až N vrcholů (s celočíselnými lichými souřadnicemi; bloky ale podle nich indexovat nemůžeme, neboť kombinací souřadnic může být až kvadraticky, proto je podle souřadnic budeme mít jen lexikograficky uspořádané a v případě potřeby binárním půlením v logaritmickém čase najdeme blok pro dané souřadnice, nebo zjistíme, že tam žádný není; to bude potřeba jen konstantně-krát pro každý blok), ale každý vrchol může mít nejvýše 24 sousedů, takže počet hran je již nejvýše lineární, jen jejich nalezení bude o něco složitější. Hrana mezi dvěma bloky vede právě, když z nich lze vybrat dva vrcholy (každý z jednoho), mezi nimiž je vzdálenost menší než 4.

Máme tedy vždy dvě množiny bodů (A a B) oddělené nějakou přímkou (bez újmy na obecnosti to budiž osa y) a chceme najít dva body takové, že jejich vzdálenost je nejmenší možná. Všimněme si, že podíváme-li se na průsečík jejich spojnice a osy y, musí mu být oba body blíž než libovolný jiný z jejich skupiny. Pro každý bod na ose y si tedy určíme, který bod z A a který z B je mu nejblíže. To uděláme tak, že si body ve skupině setřídíme podle y-ové souřadnice a v tomto pořadí je budeme přidávat a hledané údaje postupně upravovat: na začátku máme jeden bod, který je nejblíže všem bodům osy y; kdykoli přidáme další, podíváme se, kterým bodům osy y bude blíže než bod předchozí (třeba tak, že najdeme průsečík osy jejich spojnice s osou y) a pokud na ten už žádné nezbudou, odstraníme jej a porovnáme nový bod s jeho dalším předchůdcem atd., jinak pokračujeme přidáváním. Po setřídění nám na toto stačí lineární čas, neboť každý bod jednou přidáme a porovnáváme s předchůdcem bez odstraňování a nejvýše jednou odstraníme. Pak už zbývá jen projít osu y a pro každý bod porovnáme dvojici bodů, které jsou mu z každé ze skupin nejblíže (přesněji, projdeme ji po úsecích, kde se tato dvojice nemění). Pokud najdeme nějakou dvojici, která je blíže než $2R$, zapamatujeme si navíc, jestli protíná kladnou poloosu y.

Spojení bloků $(-1,-1)$ s $(1,1)$ a $(-1,1)$ s $(1,-1)$ – to jsou jediná, která mohou sdružovat hrany vedoucí z různých stran kolem počátku – vyřešíme speciálně: budeme chtít vědět nejen, jestli jsou spojeny, ale také z jaké strany (případně že z obou) to spojení vede. Opět máme dvě množiny A a B a body v nich setříděné podle nějaké ze souřadnic. Pokud je některá prázdná nebo obě jednoprvkové, je řešení triviální, jinak si vybereme nějaký bod m a rozdělíme obě množiny na body, které (když si je představíme jako vektory) mají směrnici menší než m (množiny A_1 a B_1) a ty zbylé (A_2 a B_2). Zřejmě dvojice A_1–B_2 může být spojená jen jedním typem hran, obdobně A_2–B_2; kterým a jestli spojené jsou, můžeme zjistit výše popsaným způsobem v lineárním čase. Nalezení spojení mezi A_1–B_1 a A_2–B_2 je pak původní problém na menších množinách. Aby se nám rekurze zastavila brzy, konkrétně po $\mathcal{O}(\log N)$ iteracích, potřebujeme dané množiny rozdělovat rovnoměrně, nejlépe půlit, k tomu je potřeba, aby m byl mediánem některé z nich v uspořádání podle směrnic; ten můžeme nalézt v lineárním čase (jak na to se lze dočíst v letošní kuchařce ze druhé série), takže celý postup zabere čas $\mathcal{O}(N \log N)$.

Nakonec už jen zbývá projít celý vytvořený graf a podívat se, zda je počátek uvnitř nějakého cyklu v něm. Zjistit, zda je nějaký bod vnitřním bodem polygonu, lze třeba tak, že pošleme „paprsek“ z tohoto bodu libovolným směrem a spočítáme, kolik hran tuto polopřímku protne – pokud jich bude lichý počet, je bod uvnitř, jinak vně. Tím paprskem bude kladná poloosa y, průsečíky s níž už máme předpočítané. Začneme v libovolném vrcholu (bloku), graf budeme procházet do hloubky a přitom si počítat, kolikrát jsme paprsek protli, jakmile narazíme na vrchol, který jsme už viděli (tedy cyklus), jednoduchým rozdílem těchto hodnot z této a předchozí návštěvy zjistíme, zda se počátek nachází uvnitř mnohoúhelníku z hran cyklu. Nemusíme si ani pamatovat počty průsečíků s paprskem, neboť nás zajímá jen parita.

Nalezení hran grafu bloků včetně potřebného třídění zvládneme v čase $\mathcal{O}(N \log N)$ a na jeho prohledání stačí čas lineární a lineární je i velikost spotřebované paměti.

Roman Smrž

21-5-5 Cestovní šavle

Taková cestovní šavle je pěkně zapeklitá záležitost, při skládání si totiž nemůžeme být jisti, jestli zrovna tento dílek už máme složit, anebo ho ještě nechat narovnaný.

Mějme pouzdro délky L a do něj bychom chtěli složit šavli z N dílů. Řešení, které asi napadne každého, je vždy zkusit obě možnosti. Dílek nejdříve zkusíme složit a přesuneme se na zbylé dílky. Když se nám je žádným způsobem nepovede složit do pouzdra, tak se vrátíme, původní dílek zkusíme nechat narovnaný a opět stejný postup použijeme na zbylé dílky. Pokud ani tato cesta nevede k cíli, pak šavli nejde složit. Algoritmus má ovšem exponenciální časovou složitost $\mathcal{O}(2^N)$.

Naivní algoritmus je exponenciální, protože se v něm spousta kroků opakuje. To z této úlohy dělá problém typický pro řešení pomocí dynamického programování. Pusťme se tedy do něj. Téměř vše, co budeme potřebovat, je pole délky $L+1$. To bude po k-té fázi obsahovat značky právě tam, kde všude může prvních k dílků končit.

Fází výpočtu tak bude celkem N. V každé budeme zpracovávat jeden dílek. Projdeme celé pole a od místa, kde by mohl končit dílek předchozí (tzn. v poli na indexu i máme značku) zkusíme přidat dílek nový. Na počátku máme značku ve všech prvcích pole, protože umístění začátku prvního dílku není ničím omezeno. Konec nově přidaného dílku pak může být na indexech $i-d$ a $i+d$, kde d je jeho délka. Samozřejmě, že nový konec musí být v mezích pole. Povede-li

se nám najít alespoň jeden možný konec pro všechny dílky, tak víme, že šavli složit jde.

Pokud bychom používali pouze jedno pole, budou se nám plést nově přidané značky se značkami z minulé fáze. Je tedy nutné mít pole dvě. Z jednoho budeme číst a do druhého si budeme připravovat značky pro další fázi. Po každé fázi pole navzájem vyměníme. Dobrý nápad je také mít pro každou fázi jinou značku(např. číslo fáze), díky tomu nebudeme muset před každou fází druhé pole nulovat.

Pro každý dílek projdeme celé pole, takže zjištění, jestli šavle složit jde, bude mít časovou složitost $\mathcal{O}(N \cdot L)$ a paměťová složitost bude $\mathcal{O}(L)$.

David Marek

21-5-6 Kržnc

Způsobů, jak tuto úlohu vyřešit v dostatečně těsném prostoru, je vícero. Například bychom mohli použít obyčejné prohledávání do hloubky a jeho zásobník šikovně zkomprimovat – tak se můžeme snadno dostat až na dva bity na vrchol. My si ale předvedeme trochu jiné, technicky daleko jednodušší řešení.

Definujeme *navigační posloupnost,* což bude posloupnost n čísel $a_1, \ldots, a_n$ (kde n je počet vrcholů grafu) v rozsahu 0 až 3. Každé navigační posloupnosti odpovídá nějaká „procházka po grafu“ (v grafové terminologii sled délky n): začneme v nultém vrcholu grafu, z něj se vydáme a_1-tou z jeho čtyř hran, z dalšího vrcholu pak a_2-tou hranou a tak dále.

Všimněme si, že pokud v grafu existuje nějaká kružnice, na které leží všechny vrcholy, pak je určitě popsána alespoň jednou navigační posloupností. Budeme tedy postupně generovat všechny navigační posloupnosti a pro každou z nich ověřovat, jestli popisuje hledanou kružnici.

Posloupnosti budeme přímočaře kódovat čísly. Na každé a_i nám stačí dva bity, takže do jednoho 32-bitového čísla schováme hned 16 prvků posloupnosti. Libovolné a_i pak dokážeme pomocí konstantního počtu aritmetických a bitových operací z tohoto kódu přečíst. Navíc se na celý kód můžeme dívat jako na jedno dlouhé $2n$-bitové číslo a jeho postupným zvyšováním o jedničku snadno vygenerovat všechny posloupnosti.

Když už umíme z posloupnosti číst, můžeme také snadno simulovat procházení po grafu a odpovídat na otázky typu „jaký je i-tý vrchol, který potkáme?“ nebo „potkáme cestou vrchol v?“. Obojí sice stojí čas $\mathcal{O}(n)$, ale o ten nám vůbec nejde. Hlavní je, že si vystačíme s konstantním množstvím pracovní paměti.

Teď už stačí umět poznat posloupnost, která popisuje hledanou kružnici. To je také snadné: pro každý vrchol grafu vyzkoušíme, zda jím posloupnost projde, a pak ještě zkontrolujeme, že se na konci vrátíme zpět do vrcholu 0.

Toto řešení spotřebuje $\lfloor n/16 \rfloor + \mathcal{O}(1)$ buněk paměti a má časovou složitost $\mathcal{O}(4^n \cdot n^2)$. Existuje totiž 4^n kódů posloupností, pro každý z nich $\mathcal{O}(n)$-krát hledáme vrchol, což pokaždé stojí $\mathcal{O}(n)$.

Grafy stupně 3: Pokud z každého vrcholu vedou pouze 3 hrany, není potřeba na prvek posloupnosti obětovat celé 2 bity. Nabízí se použít trojkovou soustavu, ale pak bychom neuměli bez dodatečné paměti z posloupnosti číst. Použijeme místo toho „křížence“ mezi dvojkovou a trojkovou soustavou: nadále budeme kód dělit na 32-bitová slova a do každého uložíme trojkově co nejvíce prvků. Vejde se jich až $\lfloor \log_3 2^{32} \rfloor$, čili 20. Spotřeba paměti tedy klesne na $\lfloor n/20 \rfloor + \mathcal{O}(1)$.

Ještě šetrnější způsob: Bystří řešitelé si všimli, že i tento docela hustý popis procházek pomocí posloupností je stále dost marnotratný. Když přijdeme do vrcholu se čtyřmi hranami, nechceme se přeci vracet po hraně, po níž jsme právě přišli, takže máme na výběr pouze ze tří možností. Předchozí trik s trojkovou soustavou tedy můžeme použít i pro původní verzi úlohy. V grafech stupně 3 si dokonce vystačíme se dvěma možnostmi, což nám dává jediný bit na stav, a tedy $\lfloor n/32 \rfloor + \mathcal{O}(1)$ buněk paměti. Strýček Skrblík by měl radost a pan Cowmess jistě také.

Martin Mareš

Úloha 21-5-2 – Banky – program

```
program banky;

const
  MaxN = 182;
  MaxM = 32942;

type
  tkurz = record
    odkud, kam : Integer;
    kurz : Real;
  end;

var
  meny : array [1..MaxN] of real;
  kurzy : array [1..MaxM] of tkurz;
  men, kurzu : integer;
  i, j : integer;
  novaHodnota : real;
  zmena : boolean;

begin
  read(men, kurzu);

  for i := 1 to kurzu do begin
    with kurzy[i] do
      read(odkud, kam, kurz);
  end;

  for i := 1 to men do
    meny[i] := 1;

  for i := 1 to men do begin
    zmena := false;
    for j := 1 to kurzu do begin
      novaHodnota := meny[kurzy[j].odkud] * kurzy[j].kurz;
      if (novaHodnota > meny[kurzy[j].kam]) then begin
        meny[kurzy[j].kam] := novaHodnota;
        zmena := true
      end
    end;
    if not zmena then break
  end;

  if zmena then
    writeln('Existuje pro banku prodělečná posloupnost.')
  else
    writeln('Banka nikdy neprodělá (pokud nepřijde krize).')
end.
```


Úloha 21-5-4 – Zákony – program

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include <math.h>
#define N_MAX 100000
int n;
struct bod {
   float s[2]; /* souřadnice bodu v rovině */
   int b[2];   /* souřadnice bloku, do nějž patří */
   int blok;
} body[N_MAX];

struct {
   int s[2];         /* souřadnice */
   int bodu;         /* počet bodů v bloku */
   unsigned char hrany[5][5];
      /* se kterými z možných 24 bloků sousedí */
   bool navstiveno;
      /* zda byl již navštíven při procházení */
   bool pruseciku;
      /* počet průsečíků s "paprskem" na cestě
       * od počátečního vrcholu (bloku) sem */
} bloky[N_MAX];

/* dvě porovnávací funkce pro qsort */
int porovnat_bloky(const void *va, const void *vb)
{
   const struct bod *a = va, *b = vb;
   return a->b[1] == b->b[1] ?
      a->b[0] - b->b[0] : a->b[1] - b->b[1];
}
int por_sour; /* porovnávaná souřadnice */
int porovnat_sour(const void *va, const void *vb)
{
   const struct bod *a = va, *b = vb;
   return (a->s[por_sour]>b->s[por_sour]) -
      (a->s[por_sour]<b->s[por_sour]);
}

/* Najdeme první bod v bloku o daných souřadnicích
 * pomocí binárního vyhledávání; index tohoto bodu
 * bude zároveň sloužit jako index celého bloku */
int najit_blok(int bx, int by)
{
   int min = 0, max = n-1;
   while (min != max) {
      int i = (min+max)/2;
      if (body[i].b[1] < by) min = i+1;
      else if (body[i].b[1] > by) max = i;
      else if (body[i].b[0] < bx) min = i+1;
      else max = i;
   }
   if (body[min].b[0] == bx && body[min].b[1] == by)
      return min;
   return -1;
}

/* Funkce dostane ukazatele na dvě skupiny bodů,
 * počet bodů v každé z nich, x-ovou souřadnici přímky
 * rovnoběžné s osou x, která skupiny rozděluje; vrátí
 * 1 pokud najde dvojici, která je od sebe vzdálená
 * méně než 4 a neprotíná kladnou poloosu y,
 * 2 pokud ji protíná, jinak 0. Pokud je s = 1, počítáme
 * se souřadnicemi prohozenými oproti popisu. */
unsigned char spojene(struct bod *ba, struct bod *bb,
      int na, int nb, int s, float x)
{
   static int pred[2][N_MAX], nasl[2][N_MAX];
            /* předek a následník daného bodu */
   static float zacatek[2][N_MAX];
         /* y-ová souřadnice, od níž je
          * bod rozdělující přímce nejblíž */
   struct bod *skup[2] = { ba, bb };
   int vel[2] = { na, nb };
   float a, b, c, y;
   for (int t = 0; t < 2; t++) {
      pred[t][0] = nasl[t][0] = n;
      zacatek[t][0] = -INFINITY;
      for (int i = 1; i < vel[t]; i++) {
         pred[t][i] = i-1; nasl[t][i] = n;
         for (int j = i-1; ; j = pred[t][j]) {
          /* najdeme osu spojnice danou rovnicí ax+by+c=0 */
            a = skup[t][i].s[s] - skup[t][j].s[s];
            b = skup[t][i].s[!s] - skup[t][j].s[!s];
            c = - (a*(skup[t][i].s[s]+skup[t][j].s[s])/2) -
               (b*(skup[t][i].s[!s]+skup[t][j].s[!s])/2);
            y = -(c+a*x)/b;
            /* průsečík osy a rozdělující přímky */
            if (y < zacatek[t][j]) {
             /* Jestliže předchozí bod není nikde nejbližší
              * rozdělující přímce, odstraníme jej, */
              pred[t][i] = pred[t][j];
              nasl[t][pred[t][i]] = i;
            } else {
              /* jinak si zapamatujeme,
               * odkud je nejbližší současný */
              zacatek[t][i] = y;
              break;
            }
         }
      }
   }
   for (int i = 0, j = 0; i < na && j < nb; ) {
      float dx = skup[0][i].s[s] - skup[1][j].s[s];
      float dy = skup[0][i].s[!s] - skup[1][j].s[!s];
      if (dx*dx + dy*dy < 16) {
         if (   /* Jsou li body na různých stranách osy y */
            ((skup[0][i].s[0]>0) != (skup[1][j].s[0]>0)) &&
            /* a platí (a_y-b_y) * a_x / (a_x-a_y) < a_y,
             * tedy směrnice vektoru a-b je menší než
             * směrnice vektoru a-0 pro kladná a_x
             * a větší pro záporná a_x, */
            ((skup[0][i].s[1]-skup[1][j].s[1])
             * (skup[0][i].s[0])
             / (skup[0][i].s[0]-skup[1][j].s[0])
             < skup[0][i].s[1])
            ) return 1;
            /* protíná jejich spojnice kladnou poloosu y. */
         return 2;
      }
      /* Posuneme se na další bod buď ve skupině A, nebo B,
       * podle toho, který se mění dřív */
      if (nasl[0][i] != n) {
         if (nasl[1][j] != n) {
            if (zacatek[nasl[0][i]] < zacatek[nasl[1][j]])
              i = nasl[0][i];
            else j = nasl[1][j];
         } else i = nasl[0][i];
      } else {
         if (nasl[1][j] != n)
            j = nasl[1][j];
         else return 0;
      }
   }
   return 0;
}

/* pomocné pole pro dočasné skupiny bodů */
struct bod buf[N_MAX*2];
int bi = 0;
/* Funkce hledá hrany mezi skupinami bodů, které mohou být
 * spojeny oběma typy hran. Vrátí 1 při nalezení pouze
 * kladnou poloosu y neprotínajících spojení, 2 najde-li
 * jen ty, které ji protínají, 3, pokud najde obě,
 * a jinak 0 */
unsigned char spojene_kolem_pocatku(struct bod *a,
                  struct bod *b, int na, int nb)
{
   if (!nb || !na) return 0;
   if (na == 1 && nb == 1)
      return spojene (a, b, na, nb, 0, 0);
   /* Pro jednoduchost zde zvolíme medián směrnic
    * (tedy spíše pivot, protože to ve většině případů
    * medián nebude) náhodně, což dopadne dobře v průměrném
    * případě, jak najít medián spolehlivě v lineárním čase
    * se dočtete v kuchařce ze druhé série. */
   int median;
   float med_smer;
```