Výsledková listina dvacátého prvního ročníku KSP po první sérii

		škola	*ročník*	*sérií*	*2011*	*2012*	*2013*	*2014*	*2015*	*2016*	*série*	*celkem*
1.	Vítězslav Plachý	GJiříPoděb	3	1	8	10	5	10,5		10	41,2	41,2
2.	Petr Čermák	GEBenešeKL	3	1	7	6	7	10,5	4	10	40,7	40,7
3.	Vojtěch Kolář	G Neratov	4	2	5	10	7	6	8	12	40,4	40,4
4.	Alexander Mansurov	GNVPláníPH	0	1	8	10	6	6	5	10	39,7	39,7
5.	Jiří Cidlina	GVoděraPH	4	1	7	6	8	10	6	6,5	39,4	39,4
6.	Filip Hlásek	GMikulášPL	2	6		10		10,5	6	11	39,2	39,2
7.	Lukáš Ptáček	GJAKŽeliez	3	1	9	8	7	9	7		38,7	38,7
8.	Michal Bilanský	GLepařovJČ	3	1	5	10	2	8	6	8	38,3	38,3
9.	Filip Sládek	GNámestovo	3	1	9		8	10	6	4,5	38,1	38,1
10.	David Věčorek	GTNovákBO	3	1	9	6	8	8	4		37,8	37,8
11.	Libor Plucnar	GBezručeFM	4	10	9	8	11		8		36,2	36,2
12.	Pavel Veselý	G Strakon	4	13	9	8		11		9	36,0	36,0
13. – 14.	Tomáš Pikálek	GBoskovice	2	1	7	6		10,5		4	35,6	35,6
	Jan Veselý	G Strakon	2	1	4	6		6		9	35,6	35,6
15.	Jitka Novotná	G Bílovec	4	3	9	10		1	5	7	35,4	35,4
16.	Lukáš Chmela	GJŠkodyPŘ	0	1	1	4	5	5	4	10	35,2	35,2
17.	Ondřej Pelech	GJNerudyPH	4	1	6	4		10	6		33,4	33,4
18.	Martin Zikmund	G Turnov	1	1	4	6	2	8	1		30,0	30,0
19.	Milan Rybář	GJungmanLT	4	1		4	4		3	6	29,5	29,5
20.	Karel Tesař	SPŠE Plzeň	3	1	6	2	7		4		28,7	28,7
21.	Filip Štědronský	GMikulášPL	2	1	3			11		11	28,5	28,5
22.	Karolína Burešová	G ČesLípa	2	1	6		2		3	6	27,7	27,7
23.	Jiří Setnička	G25březnPH	2	1	2	8			2	6	27,6	27,6
24.	Vlastimil Dort	GŠpitálsPH	3	1	9				4	12	27,4	27,4
25.	Pavel Taufer	ArcibisGPH	3	3				11	3	8	26,4	26,4
26.	Jan Škoda	GMikulášPL	2	1	9		3			6,5	25,1	25,1
27.	Štěpán Šimsa	GJungmanLT	0	1		2		10		5	24,0	24,0
28.	Jan Vaňhara	G Holešov	4	2	8		6		4		23,8	23,8
29.	Petr Pecha	SPŠSVsetín	2	1	4		4			6	23,6	23,6
30.	Barbora Janů	GKepleraPH	2	1	5			3		5,5	22,8	22,8
31.	Alena Bušáková	G Trutnov	2	1	5				4	5	22,6	22,6
32.	Alžběta Pechová	SPŠSVsetín	4	5				3	4	6,5	19,3	19,3
33.	Jakub Sochor	G Bílovec	4	1	7	6					17,1	17,1
34.	Petr Zvoníček	G Slavičín	3	1	6				6		15,5	15,5
35.	Mirek Jarolím	GMikulášPL	3	4	7	0			6		15,3	15,3
36.	Radim Cajzl	GNoMěsNMor	2	16	9					8	15,2	15,2
37.	David Formánek	GJarošeBO	2	1	3				4		12,1	12,1
38.	Karel Král	G Most	3	1			4		2		11,6	11,6
39.	Kateřina Lorenzová	G Česká ČB	2	1						7	10,2	10,2
40.	Hynek Jemelík	GJarošeBO	2	1		10					10,0	10,0
41.	Martin Holec	G Slavičín	2	1	9						9,0	9,0
42.	Dominik Smrž	GOhradníPH	0	1	8						8,8	8,8
43.	Petr Babička	G Světlá	4	1	6						8,1	8,1
44.	Stanislav Fořt	GCoubertTÁ	1	1	5	0					7,5	7,5
45.	Jiří Keresteš	SPŠE Plzeň	3	5						5	7,4	7,4
46.	David Vondrák	GDašickáPA	3	1		4					7,1	7,1
47.	Igor Koníček	G UherBrod	3	1	3	0					5,7	5,7
48.	Ladislav Maxa	GKepleraPH	3	1					3		5,5	5,5
49.	Jan Kostecký	VOŠŠumperk	2	1		2					4,5	4,5
50.	Karel Kolář	GŠpitálsPH	4	1		0					0,0	0,0

Milí řešitelé a řešitelky!

Ani jsme se neohřáli (koneckonců už je taky podzim) a je tu zadání druhé série. Tentokrát jsme si pospíšili, a tak dostáváte zadání společně s opravenými řešeními série první.

Chtěli bychom ještě požádat řešitele, kteří používají naše submitovátko, aby psali u svých řešení stejnou hlavičku, jako kdyby používali papírovou poštu. Letos máme totiž několik řešitelů, kteří nejen nevědí, kterou úlohu vlastně řeší, ale ani se neumí podepsat :–)

Termín odeslání Vašich řešení druhé série jest pondělí 15. prosince 2008. Řešení můžete odevzdávat 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

Na závěr pro vás máme malé pozvání. V úterý 2. prosince se koná Den otevřených dveří MFF UK, více informací najdete na *http://www.mff.cuni.cz/verejnost/dod/*.

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*.

Druhá série dvacátého prvního ročníku KSP

Po autentické reportáži z pekla, která nás stála jednoho reportéra, jsme se rozhodli držet při zemi (resp. na zemi). Téma druhé série se ponese ve znamení věrného popisu matfyzácké reality. Jak vypadá běžný den matfyzáka vám popíší přímo ti nejkvalifikovanější – Jan Bulánek a Zbyněk Falt.

Jdu temným lesem, když tu najednou za sebou uslyším plíživé kroky. Neotáčím se a pomalu zrychluji. Ale kroky se stále blíží a když už téměř utíkám, uslyším hluboký hlas: „Definuj Lebesgueův integrál!“ V tu chvíli mi ztuhne krev v žilách, snažím se vykrucovat, ale hlas je neústupný. Během vteřiny mi před očima proběhne celý můj čtyřletý matfyzácký život a pomalu se s ním loučím . . .

„O-ou,“ ozve se náhle, „jdeš dneska do školy?“ Zvedám hlavu otlačenou od klávesnice. Spolubydlící mi píše na ICQ.

„Jasně, že jdu!“ odepisuji.

A tak mi začíná další všední den – den matfyzáka.

Vypiji dva dny starou kávu, která chutná spíš jako polévka, která byla v hrnku před ní, a pomalu se přesunu k umyvadlu. Zbytkem kartáčku si vyčistím zuby a podívám se do zrcadla, jenže hřeben nikde. Tak aspoň polituji všechny, kteří mě dnes uvidí. Holicímu strojku se jenom zasměji a jdu se oblékat. Děravé ponožky, tenisky vylepšené o několik ergonomických děr, tradiční ledvinka. Když ale zpoza postele vyndávám své oblíbené tričko, nestačím se divit. Kdysi krásně bílé, nyní hýří barvami. Ale asi nic divného, když na něj něco tu a tam ukápne.

21-2-1 Špinavé tričko — 10 bodů

Takové tričko si lze představit jako obdélník a skvrny si lze rovněž představit jako obdélníky různých barev, které se mohou vzájemně překrývat. Pokud se na jednom místě překrývá více skvrn, je na tomto místě vidět pouze ta skvrna, která se na tričku objevila jako poslední. Vaším úkolem bude pro zadané skvrny spočítat, kolik které barvy se na tričku vyskytuje a kolik ještě zbylo nezašpiněného trička.

Na vstupu dostanete kladná celá čísla W a H, která představují šířku a výšku trička. Levý dolní roh bude mít souřadnice [0,0] a pravý horní roh souřadnice [W,H]. Dále dostanete číslo N, které představuje počet skvrn, ty jsou na vstupu zadávány přesně v tom pořadí, v jakém se objevovaly na tričku. Každá skvrna je zadána pěti kladnými celými čísly. Souřadnicemi levého dolního rohu, souřadnicemi pravého horního rohu a svou barvou. Barvy jsou očíslované od 1 do N.

Příklad: Pro $W = 6$, $H = 6$, $N = 3$ a skvrny (1, 1, 5, 5, 2), (1, 2, 2, 3, 1) a (4, 4, 6, 6, 3) bude výstup, že čistého trička zůstalo 16 čtverečních jednotek, barva 1 zabírá 1 jednotku, barva 2 zabírá 14 jednotek a barva 3 se vyskytuje na 4 jednotkách.

Konečně mohu vyrazit z kolejí, doběhnout autobus a nestihnout tramvaj. Však pojede další a škola počká. V tramvaji se snažím vyřešit domácí úkol z diskrétní matematiky. Ještě, že je tak jednoduchý. Ale i tak se postaral na celou cestu o můj typický matfyzácký nepřítomný pohled, který se změnil až ve chvíli, kdy jsem o 4 zastávky přejel Malostranské náměstí. Konečně přijíždím ke škole. Místo na přednášku ale směruji své kroky k rotundě, ve které je počítačová laboratoř, abych se podíval, kde že to vlastně mám přednášku. Zrovna jsem stál uprostřed, když mě polil studený pot. Ta noční můra měla být varováním, neboť na dnešek jsem měl od profesora už po několikáté slíbeno, že mě z té analýzy vyzkouší, ať chci nebo nechci. Tentokrát ale jeho výraz nasvědčoval tomu, že to myslí smrtelně vážně . . .

21-2-2 Útěk před zkouškou — 9 bodů

Rotunda má tvar kruhu. Náš hrdina se nachází přesně uprostřed, zatímco profesor na jeho obvodu. Protože je podél stěn rotundy mnoho skříní, stolů a východů, stačí, aby se student dostal k libovolnému bodu na obvodu, odkud již může utéct nebo se bezpečně schovat. Samozřejmě, že se zároveň na tomto bodu nesmí vyskytovat i profesor (pak by se velmi těžko schovávalo a student by byl okamžitě vyzkoušen). Má to ale jeden háček, matfyzák nezvyklý pohybu se pohybuje 4× pomaleji než rozzlobený profesor. Profesor se ale na druhou stranu z neznámého důvodu bojí přiblížit ke středu, takže se pohybuje pouze podél obvodu.

Najděte strategii, jak se má za těchto podmínek student pohybovat, aby profesorovi vždy utekl, nebo dokažte, že mu utéct nelze. Profesor se může pohybovat zcela libovolně, takže o jeho „chytací“ strategii nemůžete dělat žádné předpoklady.

Ani nevím, jestli se mi podařilo utéct nebo ne. Každopádně mi z toho pořádně vyhládlo. Takhle se přeci nemohu soustředit. A tak jsem se odhodlal k zoufalému činu. Vydal jsem se do menzy. Bohužel jsem vůbec nebyl sám, kdo dostal hlad, takže před výdejnou byla obrovská fronta.

21-2-3 Fronta — 10 bodů

Matfyzáci jsou pyšní na svou inteligenci a dávají to ostatním najevo. Nejvíce se tento problém projevuje, když jsou matfyzáci nuceni tvořit fronty. Matfyzák, který si o jiném matfyzákovi myslí, že je hloupější, odmítá stát ve frontě za ním. Tím vzniká řada nepříjemných strkanic a šarvátek.

Napište program, který dostane seznam matfyzáků a jejich názorů na inteligenci ostatních. Vaším úkolem je matfyzáky uspořádat do posloupnosti tak, aby vždy platilo, že pokud považuje matfyzák A kolegu B za hloupějšího, pak musí v této posloupnosti stát A před B. Samozřejmě, že takové uspořádání nemusí existovat. Např. když si všichni myslí, že jsou chytřejší než všichni ostatní. V takovém případě oznamte, že uspořádání nelze vytvořit.

Tato úložka je praktická, což znamená, že řešení budete odevzdávat výhradně formou odladěného zdrojového kódu. Přesnější zadání a formulář na odevzdání kódu nalzenete jako vždy v CodExu na https://codex2.ms.mff.cuni.cz/ksp/. Nevíte-li, co praktická úložka je a jak přesně postupovat, podívejte se do zadání úlohy 21-1-2 „Optimalizace kotlů“.

Ještě, že to (jako ostatně vždy) vyšlo tak, že jsem šel na řadu první, takže jsem mohl nerušeně pokračovat ve studiu. Tříhodinové zkoušení Unreal Tournamentu v rámci předmětu „Vývoj počítačových her“ mi vždycky šlo a na rozdíl od jiných předmětů jsem v něm viděl svou budoucnost. Bohužel někdo vždycky rozpojí pracně vytvořenou síť, takže počítače musíme každý týden sesíťovávat znovu a znovu. A to zavání nepříjemnou fyzickou prací.

21-2-4 Síťování — 8 bodů

Sesíťovat počítače není žádná maličkost. Nemají totiž klasické síťové rozhraní. Každý počítač má dvě zdířky na síťový kabel. Jednu vstupní a jednu výstupní. Pokud tedy chcete dosáhnout konektivity mezi všemi počítači, musí být spojeny do kruhu, a to tak, že každý kabel vede z výstupní zdířky jednoho počítače do vstupní zdířky druhého počítače. Navíc jsou kabely špatně odstíněné, takže se nesmí nikde křížit, aby se navzájem nerušily.

Na vstupu dostanete číslo N, které značí počet počítačů v místnosti. Následuje N řádků, přičemž i-tý řádek určuje souřadnice i-tého počítače v místnosti. Vaším úkolem je najít pořadí počítačů $p_1, p_2, \ldots, p_n$, takové, že se v něm každý počítač vyskytuje právě jednou a úsečky (p_1, p_2),(p_2, p_3), …, (p_{n-1}, p_n), (p_n, p_1) se nikde neprotínají. Pokud to není možné, vypište, že řešení neexistuje.

Například pro $N = 4$ a souřadnice $(0, 0)$, $(0, 2)$, $(1, 1)$ a $(-1, -2)$ může být výstupem třeba $(2, 3, 4, 1)$.

Když jsme po sedmi hodinách studia uznali, že už umíme dost a že se nám dělají mžitky před očima, vydal jsem se domů. Na kolejích na mě ale čekalo nepříjemné překvapení. Naše nádobí mi totiž div nepřišlo otevřít dveře. Už jednou jsme se pokoušeli tuto situaci teoreticky řešit, ale očividně bezúspěšně. Sice by se zdálo, že nejsnazší je všechno nádobí prostě umýt, ale na co bychom pak studovali informatiku?

21-2-5 Nádobí — 10 bodů

Umývání nádobí je velice náročná činnost, takže lze umýt pouze jeden kus za jeden den. Bohužel ale platí, že když některý kus neumyjete do určité doby, nemá smysl jej umývat vůbec a je mnohem ekonomičtější vyhodit jej a koupit nový kus. Samozřejmě, že za ta léta už studenti vědí, kolik dní určité kusy vydrží bez umytí, i kolik takový kus stojí nový.

Vaším úkolem bude navrhnout optimální systém umývání nádobí takový, aby student musel za nákup nového nádobí zaplatit co nejméně. Na vstupu dostanete číslo N, které představuje počet kusů nádobí. Dále N dvojic čísel D_i a C_i, což znamená, že i-tý kus vydrží ještě D_i dní a nový stojí C_i korun.

Vypište, v jakém pořadí se má nádobí umývat (jeden kus za jeden den) tak, aby se umyly/koupily všechny kusy a zároveň náklady na nákup byly minimální.

Například pro vstup $N = 3$ a dvojice $(1, 5)$, $(1, 4)$ a $(2, 3)$ je správný výstup $(1, 3, 2)$. Což znamená, že umyjeme 1. a 3. kus. Druhý kus už bohužel nestihneme umýt včas, a tak jej budeme muset koupit nový. Všimněte si, že jakmile nestihneme druhý úkol, už není kam spěchat a raději si uděláme třetí, za který díky tomu nezaplatíme pokutu. Náklady na nákup nového nádobí jsou 4 koruny.

Konečně si mohu do nového hrnku uvařit oblíbenou čínskou polévku a jít se podívat, kdo je online. No jo, noc bude ještě dlouhá. Navíc je potřeba zhlédnout nový díl Simpsonových. Po několika hodinách začínám cítit, že bych měl jít spát. Ale co, ještě jeden díl určitě vydržím … Zzz

Jdu temným lesem, když tu najednou …

21-2-6 Nejkratší opět vyhrává — 12 bodů

Tuto úlohu musíte řešit v programovacím jazyce Rapl, jehož popis najdete v zadání úlohy 21-1-6 z minulé série. Taktéž doporučujeme přečíst si řešení této úlohy, vyvarujete se tak častých chyb.

Úloha 1 [5 bodů]: V libovolném pořadí dostanete čísla v rozsahu 1 až N, každé právě jednou, až na jedno, které chybí. Vaším úkolem je chybějící číslo nalézt.

Číslo $1 \leq N < 2^{32}$ je po spuštění programu uloženo v registru `n` a čísla `A[0]` až `A[N-2]` jsou čísla v rozsahu 1 až N, každé se vyskutuje nejvýše jednou. Vypište to jediné číslo v rozsahu 1 až N, které mezi těmito čísly chybí. Váš program *musí* doběhnout v lineárním čase vzhledem k N a *musí* použít pouze konstantně mnoho paměti nezávisle na velikosti N, přičemž pole `A` je pouze pro čtení. Vaším cílem je napsat nejkratší program, který splňuje všechny tyto podmínky.

Úloha 2 [7 bodů]: Kromě toho, že nechybí jedno, ale dvě čísla, je tato úloha stejná jako předchozí.

Přesně řečeno, na začátku dostanete v registru `n` číslo $2 \leq N < 2^{32}$. Vaším úkolem je v libovolném pořadí vypsat tu jedinou dvojici čísel z rozsahu 1 až N, která se nevyskytuje mezi čísly `A[0]` až `A[N-3]`. Váš program *musí* doběhnout v lineárním čase vzhledem k N a *musí* použít pouze konstantně mnoho paměti nezávisle na velikosti N, přičemž pole `A` je pouze pro čtení. Vaším cílem je napsat nejkratší program, který splňuje všechny tyto podmínky.

Recepty z programátorské kuchařky

Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj*. A tak by se slušelo začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek


```
        projdi(posledni);
        int je_kancelar = 1;
        for (int i = 0; i < N; i++)
                if (v[i].oznacen == 0)
                        je_kancelar = 0;

        if (je_kancelar) // vypíšeme výsledek
                printf("Přijímací kancelář je %d.\n",
                                                posledni);
        else printf("V pekle není přijímací kancelář!\n");
}
```

```
void projdi(int num) {
        v[num].oznacen = 1; // označíme vrchol
        struct hrana* hr = v[num].hrany;
        // projdeme všechny hrany z vrcholu
        while (hr != NULL) {
                if (v[hr->cil].oznacen == 0)
                        projdi(hr->cil);
                hr = hr->dalsi;
        }
}
```

Úloha 21-1-4 – Bonsaj – program

```
program Bonsaj;

{ Obousměrný spojový seznam }
type odksez = ^sez;
    sez = record
    vlevo: odksez;
    vpravo: odksez;
    listku: Integer;
end;

type smer = (Doleva, Doprava);

procedure zpracuj(predchozi: odksez; s: smer);
{ Rekurzivní zpracování rozdvojek bonsaje:
  predchozi je předchozí rozdvojka,
  s je směr, ve kterém se koukáme. }
    var pocet: Integer;
    var novy: odksez;

begin;
    read(pocet);
    if pocet <> -1 then begin;
        if( s = Doleva) then begin;
            if predchozi^.vlevo = nil then begin;
                { Díváme se doleva, ale vlevo prvek
                  spojového seznamu chybí. }
                new(novy);
                novy^.vpravo := predchozi;
                predchozi^.vlevo := novy;
            end else novy := predchozi^.vlevo;

        end else begin;
            if predchozi^.vpravo = nil then begin;
                new(novy);
                novy^.vlevo := predchozi;
                predchozi^.vpravo := novy;
            end else novy := predchozi^.vpravo;
        end;

        novy^.listku := novy^.listku + pocet;

        zpracuj(novy, Doleva);
        zpracuj(novy, Doprava);
    end;
end;

var pocet: Integer;
var koren: odksez;
var pocatek: odksez;
begin;
    { Zpracuj vstup, kořen zvlášť. }
    read(pocet);
    if(pocet <> -1) then begin;
        new(koren);
        koren^.listku := pocet;
        zpracuj(koren, Doleva);
        zpracuj(koren, Doprava);

        { Přejdi na nejlevější prvek seznamu. }
        pocatek := koren;

        while pocatek^.vlevo <> nil do
            pocatek := pocatek^.vlevo;

        { Vypiš. }
        while(pocatek <> nil) do begin;
            write(pocatek^.listku);
            write(' ');
            pocatek := pocatek^.vpravo;
        end;
    end;
end.
```

Úloha 21-1-5 – Zapeklitá karetní hra – program

```
#include <stdio.h>

int main() {
  int C, D, n, k;
  unsigned long X = 1;
  scanf("%d %d", &C, &D);
  n = C + 1;
  k = (D < (C + 1 - D)) ? D : C + 1 - D;  // vybereme menší k pro výpočet kombinačního čísla

  for (int i = 1; i <= k; ++i) {
    X *= n - (i - 1);
    X /= i;
  }
  printf("%d", X);
}
```

```
                data[next] = from;
                from = next;
                next = tmp;
        }
        data[next] = from;
        return next;
}

// Zpracuje cestu, která byla nalezena pomocí find_path a vypíše výsledky o přesunech do souboru fp.
void process_path(int from, FILE *fp) {
        while(data[from] != 0) {
                fprintf(fp, "%d %d\n", data[from], from);
                int tmp = data[from];
                data[from] = from;
                from = tmp;
        }
}

int main(int argc, char **argv) {
        load_data();

        FILE *fp = fopen("kotle.out", "w");

        for(int i = 1; i <= N; i++) {
                // Když narazíme na volné pole, zapamatujeme si jej, abychom učetřili práci funkci get_free_place().
                if (data[i] == 0) free_place = i;
                if ((data[i] == 0) || (data[i] == i)) continue;

                // Nalezneme poslední prvek cesty/cyklu a cestu si připravíme.
                int last = find_path(i);

                int tmpPlace = 0;
                if (last == i) {
                        // Pokud je to cyklus, musíme ho nejdřív rozbít (odložit si jednoho hříšníka dočasně stranou).
                        tmpPlace = get_free_place();
                        fprintf(fp, "%d %d\n", data[i], tmpPlace);
                        last = data[i];
                        data[i] = 0;
                } else
                        free_place = i;

                // Zpracujeme cestu a vypíšeme přesuny.
                process_path(last, fp);

                if (tmpPlace) {
                        // Pokud máme hříšníka uloženého stranou, tak si ho přesuneme na správné místo.
                        fprintf(fp, "%d %d\n", tmpPlace, i);
                        data[i] = i;
                }
        }
        fclose(fp);
        return 0;
}
```

Úloha 21-1-3 – Přijímací kancelář – program

```
#include <stdio.h>
#define MAXN 1000
#define MAXM 1000

struct hrana {
        struct hrana* dalsi; // další prvek
        int cil; // cílový vrchol
};

struct vrchol {
        int oznacen;
        struct hrana* hrany; // spojový seznam na hrany
};

// počet vrcholů, hran, označených vrcholů
int N, M, posledni;
struct vrchol v[MAXN]; // pole vrcholů
struct hrana e[MAXM]; // pole hran

void projdi(int num);

int main(void) {
        // načteme vstup
        scanf("%d %d", &N, &M);

        // inicializace vrcholů
        for (int i = 0; i < N; i++) {
                v[i].oznacen = 0;
                v[i].hrany = NULL;
        }
        for (int i = 0; i < M; i++) { // čteme hrany
                int start, cil;
                // uložíme v opačném směru
                scanf("%d %d", &start, &cil);
                e[i].cil = start;
                // pridame hranu k vrcholu
                e[i].dalsi = v[cil].hrany;
                v[cil].hrany = &e[i];
        }

        for (int i = 0; i < N; i++) { // procházíme vrcholy
                if (v[i].oznacen == 0) {
                        posledni = i;
                        projdi(i);
                }
        }
        // otestujeme, zda "posledni" je kancelář
        for (int i = 0; i < N; i++)
                v[i].oznacen = 0;
```


původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce“ v druhé sérii 20. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{budeme třídit takováto pole}
type Pole=array[1..MaxN] of Integer;

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin
 {pivotem se stane poslední prvek úseku}
 x:=a[r];                          {hodnota pivota}
 i:=l-1;  {a[i] bude vždy poslední <= pivotovi}

 {samotné přerovnávání}
 for j:=l to r-1 do
  if a[j]<=x then  {právě probíraný prvek     }
  begin            {menší/rovný hodnotě pivota}
   Inc(i);         {pak zvyš ukazatel         }
   q:=a[j];        {a proveď přerovnání prvku }
   a[j]:=a[i];
   a[i]:=q;
  end;

 {nakonec přesuneme pivota za poslední <=}
 q:=a[r];
 a[r]:=a[i+1];
 a[i+1]:=q;
 prer:=i+1;      {vrátíme novou pozici pivota}
end;

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
 if l<r then            {máme ještě co dělat?}
 begin
  m:=prer(l,r);   {přerovnej, m pozice pivota}
  QuickSort(l,m-1);     {setřiď prvky napravo}
  QuickSort(m+1,r);      {setřiď prvky nalevo}
 end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N+(N-1)+(N-2)+\ldots+1)=\mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek nejvýše $(1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = \mathcal{O}(\log N)$. Místo 1/4 by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se tak často QS implementuje.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních

algoritmech v 16. ročníku). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
 x:=prer(a,l,r); {přerovnej, x je pozice pivota}
 z:=x-l+1;     {pozice pivota vzhledem k [l..r]}
 if k=z then
   kty:=a[x]             {k-tý nejmenší je pivot}
 else if k<z then
   kty:=kty(a,l,x-1,k) {k-tý nejmenší je nalevo}
 else
   kty:=kty(a,x+1,r,k-z);              {napravo}
end;
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

Řečeno s panem Pascalem:

```
{potřebujeme přerovnávací funkci, která
 dostane hodnotu pivota jako parametr}
function prerp(var a:Pole;
               l,r,m:Integer):Integer;
var q,p:Integer;
begin
 {nalezneme pozici pivota}
 p:=l;
 while a[p]<>m do
  inc(p);
 {pivota prohodíme s posledním prvkem}
 q:=a[p]; a[p]:=a[r]; a[r]:=q;
 {a zavoláme původní přerovnávací fci}
 prerp := prer(a,l,r);
end;

{hledání k-tého nejmenšího prvku z a[l..r]}
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;          {pole pro mediány pětic}
    i,j,q,x,pocet,m,z:Integer;
begin
 pocet:=r-l+1;        {s kolika prvky pracujeme}

 if pocet<=1 then           {pouze jeden prvek?}
  kth:=a[l]           {výsledek nemůže být jiný}
 else if pocet<6 then begin   {méně než 6 prvků}
  QuickSort(a,l,r);
  kth:=a[l+k-1];
  end
```


konkrétně zlomkem 165 707 065/52 746 197. (Tohle je opravdu náhoda, i když nám to asi nebudete věřit. Opravdu jsme zadání schválně nenarafičili tak, aby to vyšlo. My sami jsme na tento způsob přišli díky inspiraci od Alexandra Mansurova, který ho ale sám vzápětí zavrhl):

```
          a=165707065
   jedeme: b=a/52746197
          write b
          a=a%52746197
          a=a*10
          jump jedeme
```

d) Toto byl takový malý test, jak pozorně jste četli seznam instrukcí RAPLu. Jestlipak jste si všimli operace `@`, která počítá bitovou selekci? A jestlipak jste si také všimli, že čísla v naší čtvrté posloupnosti jsou přesně čísla 1, 2, 3, 4, ..., ze kterých jsou ovšem vyselectované jenom bity na sudých pozicích? Pokud ne, honem si běžte zopakovat instrukční sadu; pokud ano, zde je program na *4 instrukce:*

```
   preqap: b=a@85    # 01010101 dvojkově
          write b
          a=a+1
          jump preqap
```

Martin Mareš & Milan Straka

Úloha 21-1-1 – Ohniště – program

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

int main(void) {
        int n;
        float obsah = 0.0;
        float a1, a2, b1, b2, c1, c2; // souřadnice bodů A, B a C
        // načteme počet vrcholů a souřadnice prvních dvou:
        scanf("%d", &n);
        scanf("%f%f%f%f", &a1, &a2, &c1, &c2);
        // pro následující vrcholy již počítáme obsahy trojúhelníků:
        for (int i = 2; i < n; i++) {
                b1 = c1; b2 = c2;
                scanf("%f%f", &c1, &c2);
                obsah += fabs((b1-a1)*(c2-a2) - (b2-a2)*(c1-a1))/2.0;
        }
        printf("%f\n", obsah);
        return 0;
}
```

Úloha 21-1-2 – Optimalizace kotlů – program

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

int N = 0;                      // Počet kotlů.
int *data;                      // Údaje o přesunech (pozn: pole má o jeden prvek víc a indexujeme jej od 1).
int free_place = 0;             // Index posledního nalezeného volného kotle (pro dočasné odkládání hříšníků).

// Načte data ze vstupního souboru do pole "data".
void load_data() {
        FILE *fp = fopen("kotle.in", "r");
        fscanf(fp, "%d\n", &N);
        data = (int*)malloc(sizeof(int) * (N+1));
        for(int i = 1; i <= N; i++)
                fscanf(fp, "%d", data + i);
        fclose(fp);
}

// Nalezne nejbližší volný kotel, který lze použít jako dočasné odkladiště při rozbíjení cyklů.
inline int get_free_place() {
        if ((free_place == 0) || (data[free_place] != 0)) {
                free_place = 1;
                while((free_place <= N) && (data[free_place] != 0))
                        free_place++;
                if (free_place > N) exit(1);    // Tohle se nesmí podle zadání stát.
        }
        return free_place;
}

// Nalezne cestu nebo cyklus v přesunech počínaje kotlem "from" a zároveň v poli data obrátí ukazatele
// přesunu (tj. místo toho, kam se má hříšník přesunout z daného kotle, bude u kotle uloženo, ze kterého kotle se má
// hříšník přesunout do něj).
// Funkce vrací index posledního prvku cesty (pokud je stejný, jako "from", pak je to cyklus).
int find_path(int from) {
        if ((data[from] == from) || (data[from] == 0))
                return 0;

        int next = data[from];
        data[from] = 0;
        while(data[next] != 0) {
                int tmp = data[next];
```

poté se otočí orientace hran a spustí se druhý průchod do hloubky.

Pavel Klavík

21-1-4 Bonsaj

Mnohé z (h)řešitelů napadlo si bonsaj prostě postavit, během stavění zjistit její šířku a nakonec ji jednou projít a uložit si výsledky. To je samozřejmě řešení správné, složitost takového řešení je lineární – sestavení trvá lineárně dlouho k velikosti vstupu a paměti zabereme rovněž jen tolik, kolik má bonsaj/strom rozdvojek.

Jaké je asymptoticky optimální řešení? Vstup jistě musíme projít celý, tedy časová složitost lepší než $\mathcal{O}(N)$ nebude. Paměťová složitost je lineární jakbysmet – stačí uvážit bonsaj typu `2 2 2 -1 2 -1 2 -1 -1 -1 -1`, kde si musíme pamatovat hodnoty alespoň $N/2$ prvků. Vidíme, že postavení bonsaje bylo řešení snadné, ale také správné.

O konstantu chytřejší řešení dostaneme tak, že si uvědomíme, že bonsaj nepotřebujeme stavět, stačí nám ji projít přímo v preorderu a chytře si ukládat přesuny mezi sloupečky. Ideální struktura na uchovávání počtu lístků v jednotlivých sloupečcích je obousměrný spojový seznam, neboť jej můžeme rozšiřovat na obou stranách. Abychom se v hustých větvičkách bonsaje neztratili, musíme si také pamatovat cestu, kudy jsme do aktuálně zkoumané rozdvojky přišli. Na to můžeme využít zásobníku nebo také rekurze.

Martin Böhm

21-1-5 Zapeklitá karetní hra

Že jde o úlohu z kombinatoriky, napadne kdekoho. Ale aby šlo řešení hladce od ruky, je potřeba to vzít ze správného konce. Nejprve rozložíme do řady všechny navzájem nerozlišitelné čerty. Mezi každými dvěma čerty, na začátku a na konci řady je místo, kam můžeme položit nejvýše jednu ďáblici, celkem $\check{\mathrm{C}}+1$. Teď spočítáme, kolika způsoby si můžeme vybrat místa, na která ďáblice po jedné položíme. Program tedy bude počítat kombinační číslo $\binom{\check{\mathrm{C}}+1}{\check{\mathrm{D}}}$. Taky můžeme uvažovat místa, na která čertice nepoložíme – $\binom{\check{\mathrm{C}}+1}{\check{\mathrm{C}}+1-\check{\mathrm{D}}}$. Výsledek je stejný, ale dopočítáme se ho rychleji, pokud je ďáblic víc než $(\check{\mathrm{C}}+1)/2$.

Jak ale efektivně spočítat kombinační číslo? Počítat dva faktoriály a pak je dělit sice bude fungovat, ale pro větší počty karet si nevystačíme s velikostí datového typu (obyčejná kapesní kalkulačka nezvládá víc než 69!, a i na to už potřebuje počítat s mantisou a exponentem). Výsledek bude celé číslo, zkusme tedy zlomek s faktoriály nějak přeuspořádat a střídavě násobit a dělit. Kombinační číslo $\binom{n}{k}$ se spočte jako $n!/[k!(n-k)!]$. Po vykrácení n! a (n-k)! si zlomek rozepíšeme jako

$$\frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{1\cdot2\cdot\ldots\cdot k}$$

V čitateli jsou po sobě jdoucí přirozená čísla, takže lze zlomek počítat postupně jako

$$\frac{n}{1}\cdot\frac{(n-1)}{2}\cdot\frac{(n-2)}{3}\cdot\ldots\cdot\frac{(n-k+1)}{k}$$

Stačí si všimnout, že po každém vynásobení a vydělení (jedné iteraci) jsme spočítali nějaké kombinační číslo – $\binom{n}{1}$, $\binom{n}{2}$, … $\binom{n}{k}$, takže mezivýsledky jsou všechny celočíselné. Na to nám stačí proměnná, do které se vejde k-krát větší číslo než výsledek, kde k je $\min(\check{\mathrm{D}},\check{\mathrm{C}}+1-\check{\mathrm{D}})$.

Časová složitost je $O(\check{\mathrm{C}}-\check{\mathrm{D}})$, paměťová $O(1)$.

Josef Špak

21-1-6 Nejkratší vyhrává

Řešení našeho IQ-testu se sešla slušná hromádka, ale některá z nich používala jazyk Rapl až příliš vynalézavě a domýšlela si do něj instrukce, které podle definice v zadání rozhodně neuměl. Na ty běžnější chyby raději upozorníme rovnou, aby se nenachytali ostatní:

- Argument instrukce `write` může být pouze číslo, proměnná nebo prvek pole, určitě ne výraz s operátory.
- Podmínit lze jenom instrukci skoku, konstrukce typu `if a=0 => a=1` je nekorektní.

a) Posloupnost mocnin dvojky je možné vypsat na *4 instrukce* jednoduchou modifikací příkladu ze zadání:

```
        a=1
znovu:  write a
        a=a*2
        jump znovu
```

b) Správně jste poznali, že se jedná o začátek Fibonacciho posloupnosti, v níž je každé číslo součtem dvou předchozích. Stačí tedy, abychom si v registru `a` pamatovali aktuální číslo a v registru `b` číslo předchozí (budeme si přitom představovat, že před počáteční nulou je jednička). Dostaneme tak jednoduchý program na *6 instrukcí:*

```
        b=1
zase:   write a
        c=a+b     # následující
        b=a       # předchozí <- aktuální
        a=c       # aktuální <- následující
        jump zase
```

Kličku s pomocnou proměnnou `c` si ale můžeme ušetřit jednoduchým trikem a program tak zkrátit na *5 instrukcí:*

```
        b=1
zase:   write a
        a=a+b
        b=a-b
        jump zase
```

c) Jak vypsat prvních 16 číslic čísla π? Naprogramovat opravdový výpočet π není úplně snadné a určitě to nestihneme za méně než 16 instrukcí, které by nám stačily na program typu `write 3; write 1; ...` (mimochodem, opravdu jsme dostali několik delších řešení). Když tedy neumíme π spočítat, vymyslíme, jak tabulku jeho číslic reprezentovat co nejkompaktněji. Všimněme si, že do 32 bitů se vejde libovolné devíticiferné desítkové číslo, takže nám stačí vzít si dvě konstanty `31415926` a `53589793` a vypsat je po číslicích. Toho se drží i náš program na *7 instrukcí,* jen čísla rozkládá na číslice od konce:

```
        a=62951413
loooop: b=a%10    # mod 10 = posl. číslice
        write b
        a=a/10    # o číslici zkrátíme
        if a<>0 => jump loooop
        a=39798535
        jump loooop
```

Existuje ovšem ještě jeden efektivnější způsob: Najdeme dvě čísla, jejichž podíl se dostatečně přesně přiblíží k π, a tento podíl vypíšeme po číslicích. Překvapivě, s 32-bitovým čitatelem a jmenovatelem můžeme získat právě 16 číslic π,

```
 else begin         {mnoho prvků, jde to tuhého}
  {rozdělíme prvky do pětic}
  q:=1;              {zatím máme jednu pětici}
  i:=1;              {levý okraj první pětice}
  j:=i+4;            {pravý okraj první pětice}
  while j<=r do begin   {procházíme celé pětice}
   QuickSort(a,i,j);
   medp[q]:=a[i+2];              {medián pětice}
   Inc(q);                    {zvyš počet pětic}
   Inc(i,5);           {nastav levý okraj pětice}
   Inc(j,5);          {nastav pravý okraj pětice}
  end;
  {případnou neúplnou pětici můžeme ignorovat}

  {najdeme medián mediánů pětic}
  m:=kth(medp,1,q-1,q div 2);

  {přerovnej a zjisti, kde skončil pivot}
  x:=prer(a,l,r,m);
  z:=x-l+1;           {pozice vzhledem k [l..r]}
  if k=z then
   kth:=m                {k-tý nejmenší je pivot}
  else if k<z then
   kth:=kth(a,l,x-1,k)    {k-tý nejmenší nalevo}
  else
   kth:=kth(a,x+1,r,k-z);              {napravo}
 end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10\cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10\cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10\cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c>0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\le 7/10\cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N)\le cN+t(N/5)+t(7/10\cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N)=dN$ pro nějaké $d>0$. Dostaneme:

$$dN\le(c+1/5\cdot d+7/10\cdot d)\cdot N.$$

To platí např. pro $d=10c$, takže opravdu $t(N)=\mathcal{O}(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo můžeme zapsat jako 10^NA+B, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^NA+B)\cdot(10^NC+D)=(10^{2N}AC+10^N(AD+BC)+BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N)=cN+4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N)\approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B)\cdot(C+D)=AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD+BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N)=c'N+3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c'=1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned}t(N)&=N+3(N/2+3t(N/4))=\\&=N+3/2\cdot N+9t(N/4)=\\&=N+3/2\cdot N+9/4\cdot N+27t(N/8)=\ldots=\\&=N+3/2\cdot N+\ldots+3^{k-1}/2^{k-1}\cdot N+3^kt(N/2^k).\end{aligned}$$

Pokud zvolíme $k=\log_2 N$, vyjde $N/2^k=1$, čili $t(N/2^k)=t(1)=$, kde d je nějaká konstanta. To znamená, že:

$$t(N)=N\cdot(1+3/2+9/4+\ldots+(3/2)^{k-1})+3^kd.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem 3/2, čili $((3/2)^k-1)/(3/2-1)=\mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen 3^kd, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k=2^{k\log_2 3}=2^{\log_2 n\cdot\log_2 3}=(2^{\log_2 n})^{\log_2 3}=n^{\log_2 3}\approx n^{1.58}$. Konstanta d se nám „schová do $\mathcal{O}$-čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(n^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(n\log n)$, ale ty jsou mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom (viz kuchařka v 5. sérii minulého ročníku) vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

Dnešní menu Vám servírovali
David Matoušek & Martin Mareš

21-1-1 Ohniště

Ukázalo se, že dostat se do pekla pro většinu z vás nijak velký problém nebude (nakonec, problém to nemusí být ani pro ty ostatní ...). Ohniště si totiž můžeme, jak jste si téměř všichni všimli, snadno rozdělit na $n-2$ trojúhelníků – jelikož jsou jeho vrcholy na vstupu zadány postupně ve směru hodinových ručiček (označme je $A_1, A_2, \ldots, A_n$), můžeme vzít trojúhelníky $A_1A_2A_3, A_1A_3A_4, \ldots, A_1, A_{n-1}, A_n$.

Zbývá už jen určit obsahy jednotlivých trojúhelníků a ty následně posčítat. K tomu mnozí z vás použili Heronův vzorec využívající délek stran trojúhelníka (ty můžeme zjistit pomocí Pythagorovy věty).

O něco elegantnější (bez používání odmocnin v programu) je využití vektorového součinu: máme-li body A, B a C a vektory $\vec{u} = B - A$ a $\vec{v} = C - A$, je absolutní hodnota jejich vektorového součinu rovna dvojnásobku obsahu trojúhelníka ABC (jelikož počítáme v rovině, položíme $a_3 = b_3 = c_3 = u_3 = v_3 = 0$); máme tedy:

$$\vec{u} \times \vec{v} = (u_2 \cdot 0 - 0 \cdot v_2, 0 \cdot v_1 - u_1 \cdot 0, u_1v_2 - u_2v_1) = \\ = (0, 0, u_1v_2 - u_2v_1)$$

Odkud již snadno zjistíme kýžený obsah trojúhelníka (ke stejnému vzorci lze dospět také využitím determinantu matice):

$$S_{\triangle ABC} = \frac{1}{2}|\vec{u} \times \vec{v}| = \frac{1}{2}\sqrt{0^2 + 0^2 + (u_1v_2 - u_2v_1)^2} = \\ = \frac{1}{2}|(b_1 - a_1)(c_2 - a_2) - (b_2 - a_2)(c_1 - a_1)|$$

Spočítání obsahu každého z trojúhelníků zvládneme v konstantním čase a jelikož je jich celkem lineárně s počtem vrcholů, je i časová složitost $\mathcal{O}(N)$. Jednotlivé obsahy lze zjišťovat postupně při načítání vstupu, který (s výjimkou prvního vrcholu) již na nic dalšího nepotřebujeme, takže si vystačíme s konstantní pamětí.

Roman Smrž

21-1-2 Optimalizace kotlů

„Můžu?“

„Můžeš.“

„Držíš?“

„Držím.“

„Spouštěj!“

„Spouštím.“

„Máš ho?“

„Mám!“

„Tady dobrý! Na řadě je kotel číslo 421954 ...“

Jak jste měli možnost si na vlastní kůži vyzkoušet, přesouvání hříšníků mezi kotli opravdu není snadná záležitost. Pozor si musíme dávat hned na několik věcí:

Zkusíme na to jít nejprve přímočaře. Projdeme si celé pole kotlů a provedeme přesuny těch hříšníků, jejichž cílové políčko je volné. Tohle budeme opakovat tak dlouho, dokud nebudou všichni na svých místech. Každého hříšníka přesouváme právě jednou a to přímo na místo, kde se má nacházet, takže na první pohled je algoritmus konečný a vrací korektní výstup.

Ale ouha, co když se nám hříšníci zrovna sejdou např. takto: Hříšník z kotle 1 musí do 2, hříšník z 2 musí do 3 a konečně z 3 je potřeba provést přesun do 1. Tyto 3 přesuny tvoří cyklus a náš výše uvedený algoritmus na ně nebude fungovat. Pokaždé, když bychom chtěli přesunout některého z výše uvedených hříšníků, bude v jeho cílovém kotli trůnit jiný hříšník. Musíme na to tedy jinak ....

Ze zadání je jasné, že alespoň jeden kotel musí být volný (jinak by nešlo s hříšníky vůbec pohnout). Zkusíme tedy využít tohoto garantovaného volného kotle. Postupně budeme brát hříšníky na přesun. Pokud je cílový kotel volný, není s přesunem žádný problém. Pokud je ale obsazený, přesuneme *překážejícího* hříšníka do libovolného volného kotle a tím se nám uvolní cílový kotel, takže můžeme opět přesun provést. Všimněte si, že pokud je zadání korektní, nikdy nepřesouváme nikoho, kdo je již na svém cílovém místě. V každém kroku tedy snížíme počet hříšníků, kteří ještě nejsou na svém místě, o 1 a algoritmus je tedy opět konečný (korektnost je zřejmá).

Teď již máme řešení, které bude fungovat, ale musíme se zamyslet, zda splňuje požadavek na minimální počet přesunů. Jak už jistě tušíte – nesplňuje. Představme si, že máme přesunout hříšníka z kotle 1 do 2, z kotle 2 do 3 a z 3 do 4, přičemž kotel 4 je prázdný. Budeme-li postupovat přesně podle našeho algoritmu, budeme nejprve přesouvat hříšníka 1. Jenže kotel 2 je obsazen, takže je potřeba nejprve přesunout 2 do 4, abychom si kotel uvolnili. Dále chceme umístit druhého hříšníka do správného kotle (č. 3), jenže ten je jako na potvoru opět obsazen a musí být uvolněn. Sami si rozmyslete, že toto pořadí si vyžádá celkem 5 přesunů. Přitom bychom to ale určitě zvládli jen na 3 přesuny: 3 do 4, 2 do 3 a 1 do 2.

Jak je vidět, nestačí nám přímočarý algoritmus, ale potřebujeme nějaký, který zohlední plánování, abychom nic nepřesouvali zbytečně. V ideálním případě tedy budeme přesouvat hříšníky přímo do jejich cílových kotlů bez meziskladování. Problém nastává s cykly. Když narazíme na cyklus musíme jej nejdřív rozbít (jednoho hříšníka z něj přesuneme do meziskladiště). Tím z něj vznikne *cesta*, kterou snadno vyřešíme a následně přesuneme hříšníka z meziskladu na jeho cílové místo.

Cestou budeme rozumět posloupnost přesunů k_i, kdy se k_i-tý hříšník přesouvá do k_{i+1}-ho kotle a poslední (k_n-tý) hříšník se přesouvá do volného kotle. Cesty budeme zpracovávat tak, že je celé projdeme až na konec a přesuny budeme provádět od posledního k prvnímu.

Ve finále zbývá tuto myšlenku již jen naprogramovat. Pokud si nejste jistí jak na to, podívejte se na přiložený program. Při troše snahy při implementaci lze docílit časové i paměťové složitosti $\mathcal{O}(N)$.

Martin „Bobřík“ Kruliš

21-1-3 Přijímací kancelář

Takový plán pekla byl určitě pekelně zapeklitý a nebohému reportérovi zamotal na nemalou chvilku hlavu. Tu však nezamotal jenom jemu, ale i nejednomu řešiteli. Co by to bylo za peklo, kdyby přijímací kancelář musela být jenom na jednom místě? Přijímajících kanceláří může být samozřejmě víc. A co kdyby na nás čerti ušili podvod a v pekle žádná kancelář vůbec nebyla? S obojím se musí počítat a většina řešení si na tomto vylámala zuby. Nebo snad řádky kódu? Nelze činit žádné předpoklady o něčem, co v zadání nebylo uvedeno! Některá další řešení byla natolik pomalá, že pokud reportér neumřel, prohledává peklo dodnes ...

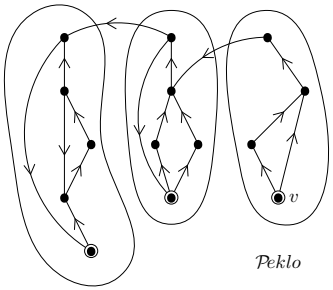
Ukážeme si postup, jak rychle a snadno přelstít peklo. Nejprve si maličko přeformulujeme zadání. Peklo bude orientovaný graf, místnosti budou vrcholy a chodby mezi nimi hrany. Nyní otočíme orientaci všech hran. Přijímací kancelář bude místo, z kterého existuje cesta do všech ostatních vrcholů grafu.

Jednoduché řešení nás určitě napadne hned. Pro každý vrchol ověříme, zda je přijímací kancelář. Spustíme z něj *prohledávání do hloubky*. To je algoritmus, který nám pro určitý vrchol zjistí, do kterých vrcholů z něj vede cesta. Funguje tak, že postupně prochází a značkuje vrcholy. Na začátku máme označený jenom výchozí vrchol. Pokud přijdeme do neoznačeného vrcholu, označíme ho a rekurentně spustíme prohledávání pro jeho sousedy. Po skončení běhu algoritmu budou označeny všechny dostupné vrcholy. Bližší informace o prohledávaní do hloubky naleznete v kuchařce Grafy 20-3.

Pokud jsme označili úplně všechny vrcholy grafu, vyhráli jsme a nalezli přijímací kancelář. Pokud žádný takový vrchol v grafu není, pak ani nemůže existovat přijímací kancelář. Jedno prohledávání nám pro graf s N vrcholy a M hranami zabere čas $\mathcal{O}(N+M)$ (každý vrchol a každou hranu navštívíme jednou) a musíme ho zavolat pro každý vrchol z N vrcholů, tedy celkem $\mathcal{O}(N^2 + NM)$, což nám pro běžné grafy (kde $M \geq N$) dává $\mathcal{O}(NM)$. V paměti si potřebujeme udržovat celý graf, paměťová složitost bude $\mathcal{O}(M+N)$. Toto řešení je správné, ale nikoho svojí rychlostí neoslní.

K rychlejšímu algoritmu nám pomůže následující pozorování. Pokud z libovolného vrcholu existuje cesta do přijímací kanceláře, pak i on sám je přijímací kancelář. Proč? Protože se z něj můžeme dostat do přijímací kanceláře a z ní poté do všech vrcholů grafu, tedy můžeme se dostat kamkoliv. Ale to není nic jiného než definice přijímací kanceláře. Tedy pokud spustíme prohledávání z nějakého vrcholu, který není přijímací kancelář, ani libovolný z navštívených vrcholů nebude přijímací kancelář. Z nich již nemusíme pouštět prohledávání, což nám ušetří spoustu času, bohužel asymptoticky máme pořád $\mathcal{O}(NM)$.

Co kdybychom zkusili při dalších prohledáváních již neprocházet vrcholy, které jsme navštívili při předcházejících prohledáváních? Budou nás zajímat pouze nové vrcholy, do kterých jsme schopni se dostat. Pokud nám stále budou nějaké chybět, určitě žádný z navštívených vrcholů není přijímací kancelář. Takto redukujeme počet nenavštívených vrcholů, až po čase nalezneme vrchol v, z kterého spustíme prohledávání naposled. Všechny vrcholy jsme tedy navštívili buď při posledním prohledávání, nebo někdy dříve. Pokud je v grafu přijímací kancelář, pak je to určitě vrchol v. Proč? Nemůže to být žádný vrchol z předchozího procházení, protože z něj neexistuje cesta například do vrcholu v. A pokud by to byl nějaký jiný vrchol z posledního prohledávání, pak by také v byla přijímací kancelář, využijeme výše ukázaného poznatku, neboť z v do ní vede cesta. Na druhou stranu v vůbec nemusí být přijímací kancelář, může existovat vrchol z předchozích prohledávání, do kterého se z v nemůžeme dostat. V takovém případě v grafu neexistuje přijímací kancelář. Řešení je jednoduché: Prostě pro v ověříme, zda přijímací kancelář skutečně je. Projdeme z něj znovu celý graf. Pokud navštívíme všechny vrcholy, je v přijímací kancelář, jinak v pekle žádná není. Na obrázku jsou vyznačeny vrcholy, z kterých jsme spustili prohledávání, a spolu s nimi v jedné bublině všechny vrcholy, které jsme navštívili při jednom prohledávání.

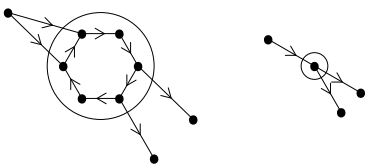


Peklo

První fáze algoritmu poběží v čase $\mathcal{O}(M+N)$, neboť na každý vrchol a hranu se podíváme jednou. Druhá ověřovací fáze poběží ve stejné složitosti $\mathcal{O}(M+N)$, tedy dohromady dostáváme $\mathcal{O}(M+N)$. Rychleji tento problém ani vyřešit nelze, tolik času potřebujeme na načtení vstupů. Paměťová složitost zůstává pořád stejná $\mathcal{O}(M+N)$.

Řada z vás si všimla, že pokud by peklo bylo souvislé a byl v něm právě jeden vrchol, do kterého nevede žádná hrana, potom by to určitě byla přijímací kancelář. Naproti tomu pokud by takové vrcholy byly dva, pak přijímací kancelář nemůže v pekle existovat. Toto řešení však selže, pokud je přijímacích kanceláří v grafu více, protože nebude existovat vrchol, do kterého nevede žádná hrana. Ukážeme si, že i z na první pohled (po)chybné myšlenky se dá ledacos vytěžit a vytvořit funkční řešení.

Pokud jsou přijímací kanceláře dvě, musí ležet v *silně souvislé komponentě*. Co je to silně souvislá komponenta orientovaného grafu? Podobně jako u klasického grafu je to maximální množina vrcholů taková, že mezi každými dvěma vrcholy existuje orientovaná cesta (která je navíc vždy celá uvnitř komponenty). Každý orientovaný graf lze rozložit na komponenty. Z hlediska hledání přijímající kanceláře se všechny vrcholy komponenty chovají podobně. Proto můžeme vytvořit kopii grafu, každou komponentu nahradit jedním vrcholem a zachovat hrany napříč komponentami. Takové věci se říká *kontrakce komponenty*, kterou si můžete také představit tak, že všechny vrcholy komponenty natlačíme k sobě, čímž nám splynou v jeden, hrany napříč nám zůstanou. Výsledná kopie je acyklická a stačí nám podívat se na stupně jednotlivých vrcholů. Zde musí existovat alespoň jeden vrchol, do kterého nevede žádná hrana, tedy projde nám výše uvedený test.



Poslední problém, který musíme vyřešit, je, jak hledat silně souvislé komponenty. Existuje pěkný rychlý algoritmus založený na prohledávání do hloubky, který funguje stejně jako výše uvedené řešení v čase $\mathcal{O}(N+M)$. Jeho detaily zde vypustíme, avšak zvídavý čtenář si ho může zkusit vymyslet. Napovíme, že se použije průchod grafem do hloubky,