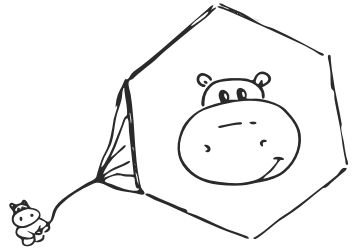
```
        while(i < count) {
                if (i != resCount) courses[resCount] = courses[i];
                last = courses[resCount].finish;
                resCount++;  i++;
                while((i < count) && (courses[i].start <= last)) i++;   // hladově vybereme další přednášku
        }

        FILE *fp = fopen("rozvrh.out", "w");                            // výpis nalezených přednášek do souboru
        fprintf(fp, "%d\n", resCount);
        for(i = 0; i < resCount; i++) fprintf(fp, "%d ", courses[i].idx);
        fclose(fp);
}

int main() {
        struct SCourse *courses;
        int count;

        count = loadCourses(&courses);
        qsort(courses, count, sizeof(struct SCourse), courseCompare);
        greedy(courses, count);
        return 0;
}
```

Výsledková listina dvacátého ročníku KSP po první sérii

		škola	*ročník*	*sérií*	*2011*	*2012*	*2013*	*2014*	*2015*	*2016*	*série*	*celkem*
1.	Peter Ondrúška	SPŠDubnica	4	1	9	11	9	2	10	12	42,8	42,8
2.	Štěpán Weber	GBuďánka	3	1	9	11		1	10	8	38,8	38,8
3.	Alena Skálová	GNaVPláni	4	2	9		8		10	11	38,0	38,0
4.	Jan Michelfeit	G HBrod	4	6	9	11			10	7	37,5	37,5
5.	Jakub Kaplan	GJKTyla	4	16	7	12			5	12	36,0	36,0
6.	Filip Hlásek	GMikuláš	1	1	9		8		7	11	35,0	35,0
7.	Jitka Novotná	G Bílovec	3	1	7	4	6	0		12	32,6	32,6
8.	Vlastimil Dort	GŠpitálsPH	2	6	9	11				12	32,5	32,5
9.	Filip Štědronský	GMikuláš	1	1	7		6		7	12	32,0	32,0
10.	Tomáš Toufar	G Bílovec	4	1	9				10	12	31,0	31,0
11.	David Marek	SPŠ Zlín	4	3	9	4	8		5	6	30,0	30,0
12. – 13.	Trung Ha duc	GMasaryk	2	6	9		8	0		11	28,0	28,0
	Libor Plucnar	GPBezruče	3	6	9				8	11	28,0	28,0
14.	David Brázdil	G Zlín	3	6	7	4	8	2	0	7	27,9	27,9
15.	Jiří Zárevúcky	SŠInfoTech	3	1	9		7		2	9	27,0	27,0
16.	Milan Rybář	GJJunman	3	1	3	2	8		2	11	26,7	26,7
17.	Jan Škoda	GMikuláš	1	1	7		8		7	4	26,0	26,0
18.	Martin Vlach	G Jihlava	4	1	9	1	6			8	25,6	25,6
19. – 20.	Lukáš Kripner	G Litvínov	2	5	7		8			10	25,0	25,0
	Petr Malý	GSladkNám	4	1			8		5	12	25,0	25,0
21.	Stanislav Fořt	G Tábor	0	5	6	1	4	0	5	8	23,0	23,0
22.	Martin Patera	GArabská	2	1	7	1	8	0	5		22,6	22,6
23. – 24.	Vojtěch Tůma	G Jihlava	4	5	2		8		0	12	22,0	22,0
	Jan Žák	G HBrod	3	1	7		7			8	22,0	22,0
25.	Vojtěch Kolář	G Neratov	3	1	2		7	2		9	20,0	20,0
26.	Pavel Veselý	G Strakon	3	8	9				0	10	19,0	19,0
27.	Adam Streck	G Hořice	4	1	6		8		3		17,0	17,0
28. – 29.	Jakub Hrnčíř	GFXŠaldy	1	1	3		4			9	16,0	16,0
	Tomáš Sýkora	G VKlobou	4	9	3			5		8	16,0	16,0
30.	Petr Sokola	SPŠ Zlín	4	1	4					11	15,0	15,0
31.	Jakub Suchý	GMikuláš	1	1					2	12	14,0	14,0
32.	Radim Cajzl	G NMnMor	1	11			1			12	13,0	13,0
33.	Petr Babička	G SvětláNS	3	1				0		12	12,0	12,0
34.	Karel Tesař	SPŠEPlzeň	2	1	7					3	10,0	10,0
35.	Jan Matějka	GJírovco	3	3	9						9,0	9,0
36.	Pavel Kratochvíl	ZŠSvětlá	0	1						8	8,0	8,0
37. – 38.	Miroslav Klimoš	G Bílovec	3	20					7		7,0	7,0
	Alžběta Pechová	GPodStrání	3	2						7	7,0	7,0
39.	Jiří Keresteš	SPŠEPlzeň	2	3						6	6,0	6,0
40.	Nikolas Zigmund	ZŠHavířov	1	1	0	1	0	0		3	5,6	5,6
41.	Peter Smatana	EkoGLabsBO	3	1			4				4,0	4,0

Milí řešitelé!

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



Termín odeslání Vašich řešení třetí série jest 28. ledna 2008. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

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

Třetí série dvacátého ročníku KSP

„Á, konečně jsem to našel," zvolal mág nadšeně a vítězoslavně pozvedl knihu nad hlavu. „Tady se píše, že se musíme vydat směrem k Lávové hoře."

Mág položil knihu na stůl a už se chystal provést složité kopírovací kouzlo, když se do věci vložil Felix: „Tak to vám tedy opravdu nezávidím. To budete mít docela štreku."

„Kráá?" ozval se nechápavě Kiri.

„Hm, na tom něco bude. Lávová hora není zrovna za rohem a létat na koštěti, to už na mě není. Průvan mi nedělá dobře na klouby."

„No vidíte! Já říkal hned, že jsme měli zůstat doma," pochvaloval si kocour náhlou změnu plánu. Mág mu ale hbitě zkazil náladu: „Vildo, zajdeš na trh a pořídíš koně. Já tu zatím zkopíruji tu knihu, ať můžeme vyrazit ještě dnes!"

Vilda zašel k místnímu handlíři. Mezi dveřmi mu kývl na pozdrav a hned přešel k věci: „Rád bych si pořídil jednoho koně. Nemusí jezdit moc rychle, ale musí vydržet dlouhou cestu. Jo, a taky aby se neplašil, pokud uvidí něco neobvyklého."

Handlíř vypadal velmi vyděšeně při pohledu na nazelenalého zákazníka. Byl by vzal nohy na ramena, ale z jedné strany mu v tom bránila zeď a mezi dveřmi stál Vilda.

„T-t-třeba t-t-tohohle?" zakoktal handlíř, když se mu vrátila řeč a ukázal přitom prstem na krásného bělouše.

„A v černé by nebyl?"

„B-bílý táhne první. To je výhoda, ne?" vyblekotal handlíř první argument, který mu přišel na mysl. V tu chvíli si ale Vildy všiml i bělouš, vzepjal se na zadní a málem tak převrhl koryto s vodou.

„Máte pravdu, tento není ten pravý," změnil handlíř názor.

Ale jak se ukázalo, žádný z přítomných koní nebyl dostatečně klidný, aby snesl pohled na Vildu. „Možná bych to mohl ještě stihnout, než..." zamumlal si handlíř pod vousy a zmizel v přilehlém uzenářství. Za chvíli se vrátil a táhl za sebou něco, co budeme z nedostatku lepší terminologie nazývat koněm.

Kůň nevypadal, že by snesl cestu třeba jen za město, ale Vildův vzhled mu zřejmě nevadil. Dokonce se mu přiblížil natolik, že mu dokázal vytáhnout z kapsy svačinu.

„Ale vždyť je slepý na jedno oko!" všiml si Vilda.

„Ano. A ani za to nebudu chtít příplatek."

„Prosím?"

„Když uvidíte něco opravdu neobvyklého, prostě okolo toho projdete zleva."

Netrvalo dlouho a obchod byl uzavřen. Ale když došlo k placení, nastal problém. Cena nepředstavovala žádné potíže, na té se shodli k oboustranné spokojenosti. Zato se ukázalo, že vybrat správné mince je těžký úkol.

20-3-1 Platba koně — 12 bodů

Je zadán obnos peněz P, požadovaný za koně, a dvě hromádky mincí (jedna Vildova a jedna handlířova). V každé hromádce mincí jsou mince různých hodnot a od každé hodnoty může být více mincí. Na Vildově hromádce je celkem N mincí, na handlířově M. Každá hodnota je desetinné číslo s přesností na 2 desetinná místa. Vaším úkolem je nalézt způsob, jak zaplatit přesně P na co nejmenší počet vyměněných mincí, nebo zjistit, že to provést nelze.

Pozn: Hodnoty mincí nemusí tvořit žádný systém běžných platidel ani být jinak „rozumné".

Příklad: Pro $P = 15{,}00$, Vildovu hromádku obsahující dvě mince o hodnotách 20,00 a 45,73 a handlířovu hromádku s mincemi o hodnotách 5,00 a 15,00 je správné řešení takové, že Vilda použije minci o hodnotě 20 a handlíř mu vrátí 5. Pro $P = 16{,}16$ a stejné hromádky mincí řešení neexistuje.

Naložili koně, mág se nechal vysadit do sedla a vyrazili na cestu. Cestovali dva dny, když se v křoví vedle cesty ozvaly hlasy. Družinka se zastavila a všichni pozorně naslouchali.

„Dneska byl nějaký nesdílný. Asi mi už nevěří," stěžoval si první hlas.

„Taky? Prý máme přepadat u severní cesty. Ale číslo stromu mi neřekl. Za prvním se nedá schovat, ale od druhého až k devadesát devátému je to stále ještě hrozně moc možností," rozumoval někdo další.

První si přisadil: „Hm, mně řekl jen součet čísel stromů."

„Mně součin. Ale to mi moc nepomůže, z toho se nedají určit," kňoural druhý.

„I z toho součtu mi bylo hned jasné, že součin nikomu k ničemu nebude," souhlasil první.

„To ti z toho bylo jasné? Tak to já už vím, která čísla to jsou! Jdu si pro luk," zajásal druhý.

„Vážně? Tak v tom případě to vím také, kam jen jsem dal meč?" pookřál první.

Mág sesedl z koně a celá družinka se ukryla za nedalekým kamenem. Sotva se jim podařilo přimět koně, aby si lehl, vyskočili lupiči na cestu.

„Už odešli. Tak bychom mohli jít také," prohlásil Felix, kterému se celá záležitost vůbec nelíbila.

„Ano, ale potřebujeme jít okolo nich," zaškaredil se Vilda.

„Vildo, ty jsi nezabalil čaj!" postěžoval si mág, když se prohraboval sedlovou brašnou. „No, co se dá dělat. Vypadá to, že je budu muset proměnit v ježky."

„Stejně tak, jako všechno ostatní, co se u každého stromu jen pohne?" neodpustil si rýpnutí Felix. Kiri při představě

proměny zděšeně opustil strom, na kterém seděl, a s mírným žuchnutím přistál vedle koně.

„No tak ježků je tu stejně málo, tak by to tak moc nevadilo. Ale alespoň bych si ušetřil práci, kdybych věděl, za kterými stromy jsou…“

20-3-2 Dva lupiči — 8 bodů

Úkolem této úlohy je najít všechny možné dvojice stromů, za kterými mají lupiči čekat. Nestačí řešení pouze najít, je třeba také zdůvodnit, proč je správné a proč žádné další dvojice neexistují.

Pozn: předpokládejme, že lupiči jsou dokonalí matematici (mají cvik v počítání lupu) a jsou z takových informací opravdu schopni zjistit svá čísla stromů v podstatě okamžitě. Tedy prohlášení: „Tak to já už vím, která čísla to jsou,“ není myšleno ironicky, ale opravdu znamená, že z dostupných informací dokáže odvodit ona čísla.

Pro případ, že by vám nějaké drobnosti utekly, uvedeme zde všechny informace ještě jednou: Velitel lupičů si myslí dvě celá čísla mezi 2 a 99. Lupiči A prozradí jejich součin, lupiči B prozradí jejich součet. Navíc A ví, že B zná součet, jen neví, kolik to je (a naopak). Rozhovor pak probíhá následovně:

A: „Nevím, jaká čísla si myslí náš velitel.“
B: „Věděl jsem, že to nebudeš vědět.“
A: „Opravdu? V tom případě už vím, co je to za čísla.“
B: „V tom případě to vím taky.“

Felix opovržlivě přičichl ke zlatě, které vyplašení ježci nechali za sebou při zoufalém úprku. Ani tolik spěchat nemuseli, mezi našimi hrdiny nebyl nikdo ochotný běhat a mág prohlásil, že bodliny na zádech jako jejich trest zcela stačí.

„Tak to bychom měli,“ pochvaloval si mág potěšen, že se mu tak složité kouzlo podařilo.

„A dej si patentovat tenhle způsob výroby ostnatého drátu,“ navrhl Felix, který si při pozorování zlata všiml žížaly, která se nemohla zahrabat, protože měla po celém těle bodliny.

„Tomu se říká aktivní obranný systém. Teď ji alespoň nesezobne kos,“ vysvětlil mág a začal listovat v knize. „Utábořím se tu na noc,“ dodal po chvíli čtení.

Vilda rozdělal oheň, vybalil deky a uvařil večeři. Noc uběhla rychle a až na vzdálené dupání ježků nic nerušilo mírumilovné ticho. Vildu ráno probralo sluníčko a rosa.

„Vaše Mágstvo, kam pokračujeme dál?“ zeptal se Vilda, sotva si protřel oči. Podíval se na mága a znejistěl: „Vždyť vypadáte jako kdybyste celou noc nespal!“ A opravdu. Mág měl červené oči, které držel sotva otevřené. Seděl u ohniště a okolo sebe měl rozloženou hromadu papírů popsanou divnými symboly.

„Uáááh, cožeto?“ zívl ospalý mág. „Aha, cesta. To je právě ten problém. Tady v knize sice píšou, že musíme najít Svítící bažinu. Ale celé je to popsané těmi starodávnými hádankami, takže nemám ponětí, kam dál.“

Hádanka Vildovi tak tajemná nepřipadala. Stálo v ní: „Cesta nejdelší – stále z kopce, jen osmkrát vykročit nahoru. A na konci cesty té, stoč se vpravo k obzoru.“

„A já nevím, jak takovou nejdelší cestu, co vede stále z kopce, jen osmkrát kousek do kopce, najít. A věštění nepomohlo,“ postěžoval si smutně mág.

„To se divím. Posledně se ty vosy vykouřit podařilo,“ ohodnotil jeho věštecké schopnosti Felix.

Kiri se probudil a slétl z větve, na které spal. Dopadl doprostřed papírů a vytvořil v nich malý kráter. Zoufale krákal a máchal křídly, aby se z té hromady dostal. Vzduch se naplnil poletujícími papíry a havraním peřím.

Vilda se natáhl, aby havrana vytáhl, když v tom mu přímo na nose přistál papír s nějakými čísly. Byly na něm vypsané výšky jednotlivých bodů na cestě.

„Ale vždyť stačí vybrat nejdelší takovou posloupnost čísel na tomhle papíru,“ zaradoval se a začal počítat.

„Kdybych nebyl tak ospalý, tak bych na to přišel sám…“ zamumlal mág, když usínal.

20-3-3 Cesta z kopce — 8 bodů

Je zadána posloupnost A obsahující N čísel a nějaké číslo k. Úkolem je nalézt indexy a, b ($a \leq b$) takové, že $b - a$ je největší možné a posloupnost $A_a, \ldots, A_b$ obsahuje maximálně k dvojic těsně po sobě jdoucích prvků A_i, A_{i+1} takových, že $A_i < A_{i+1}$. Pro všechny ostatní dvojice platí $A_i \geq A_{i+1}$. Tedy až na k míst je posloupnost nerostoucí.

Příklad: Pro $k = 1$ a posloupnost $9, 7, 6, 4, 8, 6, 3, 9, 1$ je správným výsledkem $a = 1$ a $b = 7$, tj. podposloupnost $9, 7, 6, 4, 8, 6, 3$.

„Tady to vypadá strašidelně,“ postěžoval si Vilda, když se blížili k bažině.

„Samá voda. Alespoň nějaká lávka, kdyby tu byla,“ remcal Felix.

„Lávka tu sice není, ale jsou tu provazové mosty. A kolik,“ pochvaloval si mág.

A opravdu. Mezi stromy vedly stovky provazových mostů. Mág ze své kopie knihy vytáhl mapu Svítící bažiny a s pomocí ostatních se vydrápal na nejbližší most.

Procházeli bažinou křížem krážem, od jednoho stromu k druhému. Felix několikrát navrhl, že na ně počká na té či oné křižovatce, než se na ni zase vrátí.

„Rozhled na rašelinu bude z tamté křižovatky sice nádherný, ale klidně si ho nechám ujít.“

Když i mágovi došla trpělivost a konstatoval unaveně: „Ta mapa je špatně. Tady už jsme byli nejméně šestnáctkrát a na tamté křižovatce třináctkrát. Bloudíme mezi nimi stále tam a zpět. Mezi támhletou zatracenou křižovatkou a tímhle prokletým ztrouchnivělým stromem vede určitě nejvíce cest v celé bažině. Ale nemůžu je najít na mapě.“

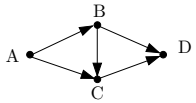
„To bude tím, že ta mapa je pěkně stará. Provazové mosty zůstaly natažené mezi stejnými stromy, ale tohle je bažina a ty stromy v ní nedrží. Prostě se od té doby přemístily,“ mudroval Vilda a strčil do nejbližšího stromu. Strom se posunul o dobrý metr.

„V tom případě stačí zjistit, mezi kterými vede nejvíce různých cest. Z toho pak dokážu zorientovat mapu…“

20-3-4 Orientace na mapě — 10 bodů

Na vstupu váš program dostane popis orientovaného grafu znázorňujícího mapu. Víte, že tento graf neobsahuje žádný orientovaný cyklus, čili že neexistuje žádná orientovaná cesta délky alespoň 1, která by začínala a končila ve stejném vrcholu. Úkolem programu je vypsat dvojici vrcholů, mezi kterými vede nejvíce různých cest v celém grafu. Za různé jsou považovány libovolné dvě cesty, které se liší alespoň jednou hranou. Pokud je takových dvojic vrcholů více, stačí libovolná z nich.

Příklad: Pro tento graf je řešením dvojice vrcholů A a D:




```
  for i:=1 to d do                                  {Nahlédneme do každé díry a zalepíme ji}
   if not diry[i].patch then begin                  {díra není zazáplatována?}
     diry[i].patch:=true;
     minx:=diry[i].x; miny:=diry[i].y; maxx:=minx; maxy:=miny;
     fcnt:=1;                                       {Díru dáme do fronty}
     f[1]:=i; fi:=1;
     while fi<=fcnt do begin                        {Projdeme díry ve frontě}
        for k:=1 to 4 do begin                      {Zkontrolujeme, jestli nemá díra sousedky}
          j:=find(diry[f[fi]].x+movex[k],diry[f[fi]].y+movey[k]);
          if j<>-1 then                             {našli jsme nějakou vedle sebe?}
           if not diry[j].patch then begin          {přidej do fronty a rozšiř záplatu}
              inc(fcnt);
              f[fcnt]:=j;
              diry[j].patch:=true;
              if minx>diry[j].x then minx:=diry[j].x;
              if maxx<diry[j].x then maxx:=diry[j].x;
              if miny>diry[j].y then miny:=diry[j].y;
              if maxy<diry[j].y then maxy:=diry[j].y;
           end;
        end;
        inc(fi);
      end;
     vysledek:=vysledek+(maxx-minx+1)*(maxy-miny+1);
   end;
  writeln(vysledek);                                {A je to}
end.
```

Úloha 20-1-4 – Kormidlo – program

```
program kormidlo;

var N, V, S, Total, i: longint;
begin
        writeLn( 'Jak velké je kormidlo?' );
        readLn( N );
        V := 0;
        S := 1;
        Total := N*N;
        for i := 1 to N - 1 do begin
                inc( V, S );
                inc( S, V );
                inc( Total, ( N - i ) * ( N - i ) * V );
        end;
        writeLn( 'Kormidlo je možné sestavit ', Total, ' způsoby' );
end.
```

Úloha 20-1-5 – Studentův rozvrh – program

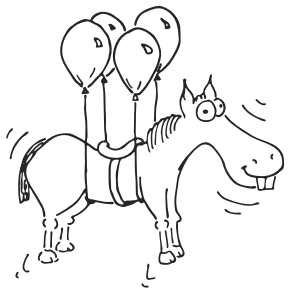
```
#include <stdio.h>
#include <stdlib.h>

struct SCourse {                                        /* Struktura udržující informace o jedné přednášce. */
        int idx;        // index přednášky
        int start;      // čas začátku
        int finish;     // čas konce
};

int loadCourses(struct SCourse **courses) {             /* Funkce načítající přednášky ze souboru do pole courses. */
        int count;
        FILE *fp = fopen("prednasky.in", "r");
        fscanf(fp, "%d\n", &count);
        *courses = (struct SCourse*)malloc(count * sizeof(struct SCourse));

        for(int i = 0; i < count; i++) {
                fscanf(fp, "%d %d\n", &((*courses)[i].start), &((*courses)[i].finish));
                (*courses)[i].idx = i+1;
        }
        fclose(fp);
        return count;
}

int courseCompare(const void *course1, const void *course2) {   /* Funkce porovnávající přednášky dle času konce. */
        return ((struct SCourse*)course1)->finish - ((struct SCourse*)course2)->finish;
}

/* Implementace hladového algoritmu nad přednáškami. Vybrané přednášky rovnou ukládá do výstupního souboru. */
void greedy(struct SCourse *courses, int count) {
        if (count < 1) return;

        int i = 0, resCount = 0, last = -1;
```

```
        for (int i = N; i > 0; i--) {
                printf("Úkol %d: %d minut\n",N-i+1,m[i][zbyva]);        // kolik minut přidelím úkolů i,
                zbyva -= m[i][zbyva];                                   //když zbylo "zbývá" minut
        }
        printf("Celková pravděpodobnost spokojenosti: %f\n",p[N][M]);
        return 0;
}
```

Úloha 20-1-3 – Oprava lodi – program

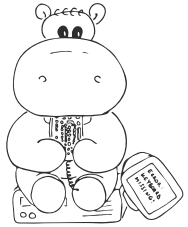
```
program Zaplaty;
type  TDira = record x,y: LongInt; patch: boolean; end;
const MaxD=1000000;
      movex:array[1..4] of integer=(-1,1,0,0); {doleva, doprava, ..., ...}
      movey:array[1..4] of integer=(0,0,-1,1); { ..., ..., nahoru, dolů}
var diry: array[1..MaxD] of TDira;
    m,n,d: LongInt; {rozměry m n, počet děr}
    i,j,k:LongInt;
    vysledek: LongInt;
    f:array[1..MaxD] of LongInt; {fronta děr}
    fi,fcnt:LongInt; {index ve frontě, počet děr ve frontě}
    minx,miny,maxx,maxy:LongInt;

procedure ReadInp;
var x,y:LongInt;
begin
  readln(m,n,d);
  for i:=1 to d do
  begin
    readln(y, x);
    diry[i].x:= x;   diry[i].y:= y;   diry[i].patch:= false;
  end;
end;

function Compare(a,b:TDira):LongInt;
begin
  if a.x>b.x then Compare:=1 else
   if a.x<b.x then Compare:=-1 else
    if a.y>b.y then Compare:=1 else
     if a.y<b.y then Compare:=-1 else
      Compare:=0;
end;

procedure sort_diry;
 procedure qsort(l,p:LongInt);
 var i,j,m:LongInt;
     t:TDira;
 begin
   i:=l-1;
   m:=(l+p) div 2;
   t:=diry[m]; diry[m]:=diry[p]; diry[p]:=t;
   for j:=l to p-1 do
    if Compare(diry[j],diry[p])<0 then
     begin inc(i); t:=diry[i]; diry[i]:=diry[j]; diry[j]:=t; end;
   t:=diry[p]; diry[p]:=diry[i+1]; diry[i+1]:=t;
   if i>l then qsort(l,i);
   if i+2<p then qsort(i+2,p);
 end;
begin
  qsort(1,d);
end;

function find(x,y:LongInt):LongInt;
var pt:TDira;
    l,p,m:LongInt;
begin
  pt.x:=x; pt.y:=y;
  l:=1; p:=d;
  repeat
   m:=(l+p) div 2;
   if Compare(pt,diry[m])=1 then
    l:=m+1 else p:=m-1;
  until (Compare(pt,diry[m])=0) or (l>p);
  if Compare(pt,diry[m])=0 then find:=m else find:=-1;
end;

begin
  ReadInp;
  sort_diry;                                          {Roztřídíme díry}
  vysledek:=0;
```


Konečně vylezli z bažiny na správnou stranu. Zajásali, rozvázali nebohého koně a mág z něho sňal levitační kouzlo. Vyškrábali se na nejbližší kopec a rozhlédli se po okolí. V dálce spatřili dýmající horu zahalenou v temných mracích. Lávová hora byla konečně na dohled…



20-3-5 Asfaltování — 11 bodů

Milí řešitelé a řešitelky,



blíží se zima a CodEx už se těší na vaše dárečky k Vánocům v podobě řešení úlohy třetí série. Způsob odevzdávání a všechny ostatní detaily zůstávají stejné, jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Obyvatelé Hipopotámie si již dlouho stěžovali na nekvalitní silnice. Když si jednou i vrchní cestář při cestě do práce v kočáře vyrazil zub, rozhodl se k radikální akci. Doslechl se, že v sousední zemi začali na cesty používat novinku zvanou asfalt, a myšlenka na vyasfaltování všech silnic v Hipopotámii byla na světě. A jak si vrchní cestář usmyslel, tak se i stalo. Při realizaci nápadu se ale nižší cestáři museli potýkat s nemilým problémem: asfalt se dovážel ze sousední země v ohromných barelech, ve kterých bylo asfaltu tak akorát na dvě cesty (byly to opravdu ohromné barely). Potíž byla v tom, že jakmile se barel narazil a asfalt z něj začal vytékat, nedal se už proud asfaltu ničím zastavit a bylo tedy nutné vyasfaltovat najednou dvě na sebe navazující cesty. Jenže jak rozvrhnout asfaltování, aby cestáři neskončili ve městě s poloplným barelem a se všemi cestami vedoucími do města už vyasfaltovanými? Důsledky zalití náměstí a několika přilehlých čtvrtí do asfaltu by byly pro cestáře jistě nemilé…

Napište program, který pro zadané propojení měst cestami rozhodne, zda je možno cesty podle popsaných pravidel vyasfaltovat, a pokud ano, vypíše jednu z možností, jak rozvrhnout, která dvojice cest bude asfaltována ze kterého barelu.

Na prvním řádku vstupního souboru `asfalt.in` se nacházejí dvě celá čísla N a M oddělená mezerou ($1 \leq N \leq 10000$ a $1 \leq M \leq 40000$), kde N určuje počet měst a M počet cest v Hipopotámii. Dále ve vstupním souboru následuje M řádků popisujících jednotlivé cesty. Každý řádek obsahuje dvě celá čísla A a B oddělená mezerou – čísla měst (města číslujeme od jedné do N), mezi kterými cesta vede. Předpokládejte, že mezi každými dvěma městy se lze po cestách dostat (pokud ne přímo, tak přes jiná města).

Výstupní soubor `asfalt.out` bude buď obsahovat jediný řádek s textem `no`, pokud neexistuje způsob, jak vyasfaltovat všechny cesty a neskončit s poloplným barelem v nějakém městě, nebo bude obsahovat $M/2$ řádků s popisem postupu asfaltování. Každý řádek postupu bude popisovat využití jednoho barelu s asfaltem, tj. bude obsahovat tři celá čísla oddělená mezerou, která představují po řadě: číslo města, ve kterém má asfaltování začít, číslo města, do kterého se má pokračovat a číslo města, ve kterém má asfaltování skončit.

Nezapomeňte, že každá cesta má být vyasfaltována právě jednou!

Příklad 1:

`asfalt.in`
```
3 3
1 2
2 3
3 1
```

`asfalt.out`
```
no
```

Příklad 2:

`asfalt.in`
```
5 8
1 5
1 4
1 3
1 2
3 2
3 4
4 5
4 2
```

`asfalt.out`
```
5 1 4
5 4 3
4 2 3
3 1 2
```

20-3-6 Hrady, hrádky, hradla — 12 bodů

Milí řešitelé, nyní už víme, jak vlastně obvody fungují, jak jsou rychlé a energeticky náročné. V dnešním pokračování našeho seriálu si povíme něco o reprezentaci dat v binární podobě, tj. pomocí jedniček a nul.

Reprezentací dat zde myslíme nějaký způsob, jakým převést čísla nebo i jiné údaje do nul a jedniček, se kterými už umíme pracovat. Nejednodušší způsob, jak to udělat je dvojková neboli binární číselná soustava. Čísla se pak mezi desítkovou a dvojkovou soustavou převádějí následovně:

Dvojková → Desítková

Vezmeme číslo zapsané ve dvojkové soustavě a budeme postupně brát cifry po jedné zprava doleva. Každou z nich vynásobíme dvojkou umocněnou na počet cifer, které jsou od aktuální cifry vpravo. Pro číslo 101010 to bude vypadat následovně: $0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 = 2 + 8 + 32 = 42$.

Jelikož $a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots + a_n \cdot 2^n$ lze přepsat jako $a_0 + 2(a_1 + 2(a_2 + 2(\ldots + 2a_n)\ldots)))$, funguje i následující jednodušší převodní algoritmus: začneme nejlevější číslicí, znásobíme ji dvěma, přičteme další číslici, výsledek znásobíme dvěma, a tak dále.

Desítková → Dvojková

Pokud je zadané číslo sudé, zapíšeme si nulu, v případě, že číslo je liché, zapíšeme jedničku a od čísla jedničku odečteme. Pak číslo vydělíme dvěma a postup zopakujeme. Jedničky a nuly budeme zapisovat zprava doleva tak dlouho, až

se naše číslo stane nulou. Číslo 13 bychom převáděli takto:

číslo desítkové	*číslo dvojkové*	*výpočet*
13	1	$(13-1)/2$
6	01	$6/2$
3	101	$(3-1)/2$
1	1101	$(1-1)/2$

Všimněte si, že tento postup je vlastně předchozí postup pro převod z dvojkové soustavy spuštěný pozpátku.

Jiný používaný způsob, jak kódovat čísla do jedniček a nul, je kód BCD (Binary Coded Decimal). Myšlenka je následující: Když se rozhodneme trochu plýtvat, klidně můžeme kódovat desítková čísla po číslicích. Přitom na každou číslici použijeme 4 bity, a to tak, že využijeme 10 kombinací z 16 možných. Číslo 1394 bychom zapsali jako 0001 0011 1001 0100. Díky mírnému plýtvání jsme si značně zjednodušili práci, tedy aspoň v případech, kdy čísla daleko častěji rozebíráme na číslice, než s nimi počítáme. (Pěkný příklad jsou třeba digitální hodinky – ty musí umět číslo zobrazovat na displeji, ale jinak stačí umět přičítat jedničku.)

Někdy se také potřebujeme vypořádat s tím, že pokud přenášíme nějaký údaj do jiného obvodu (třeba po počítačové síti, a nebo ho ukládáme na disk), může se cestou trochu poškodit – některé bity se mohou změnit z nul na jedničky nebo naopak. Tehdy se samozřejmě hodí umět chybu odhalit, nebo dokonce automaticky opravit. Zkusme si třeba na konec binárního čísla přidat nulu nebo jedničku tak, aby celkový počet jedniček v čísle byl sudý (tomu se říká paritní bit). Pokud v takovém čísle nastane jedna chyba, ihned ji odhalíme, jelikož vždy vznikne číslo s lichým počtem jedniček. Existuje spousta sofistikovanějších kódů, které dokáží detekovat nebo opravovat větší počty chyb, ale my si už místo dalšího povídání raději naservírujeme nové úložky.

Úložky

Vaším úkolem nyní bude vymyslet obvod na detekci chyb, který bude testovat dělitelnost třemi (taková kontrola chyb je o něco silnější než parita).

1) Navrhněte a nakreslete obvod, který pro číslo zadané v binární reprezentaci zjistí, zda je dělitelné třemi. [6 bodů]

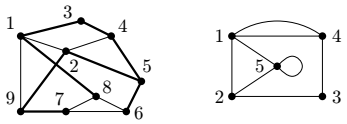
2) Úkol je stejný, ovšem čísla jsou na vstupu v BCD reprezentaci. [6 bodů]

Recepty z programátorské kuchařky

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé*. Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti*.

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

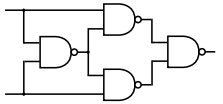
Grafům bez kružnic budeme obecně říkat *lesy*, jelikož každá komponenta souvislosti takového grafu je strom.

Existuje ještě obecnější důkaz správnosti hladových algoritmů, ale k tomu bychom potřebovali vyložit matroidy a na to zde bohužel není prostor. Takže snad někdy příště. . . :o)

Martin „Bobřík“ Kruliš

20-1-6 Hrady, hrádky, hradla

Prvním úkolem bylo vymyslet hradlo XOR z hradel NAND a nakreslit takový obvod. Asi nejjednodušší je tento obvod:



Úkolem druhým bylo najít všechny dvouvstupové funkce. Je několik způsobu, jak si spočítat, kolik jich vlastně je. Vstupem funkce jsou čtyři různé dvojice (00, 10, 01, 11). Máme dvě možnosti jako odpovědět na 00, dvě jak odpovědět na 10 . . . , celkem nám vychází $2 \cdot 2 \cdot 2 \cdot 2 = 2^4$ možných funkcí. K tomuto číslu se dá také dobrat jinou úvahou: Budeme sčítat počty čtyřprvkových posloupností složených z nul a jedniček a to tak, že si je roztřídíme do skupin podle toho, kolik obsahují jedniček. Zavedeme k od 0 do 4, které označuje počet jedniček v posloupnosti. Pro $k = 0$ je to právě jedna možnost, a to čtyři nuly. Pro $k = 1$ jsou to 4 možnosti: představme si, že jednička postupně prochází všechny pozice. Pro $k = 2$ je to 6 možností, přijít se na to dá třeba takhle: pokud je levá z obou jedniček na první pozici, jsou 3 možnosti, kam dát druhou, pokud na druhé pozici, tak 2, a pro třetí pozici jen jedna. Pro $k = 3$ a $k = 4$ je situace stejná jako pro $k = 1$ a $k = 0$, stačí zaměnit jedničky a nuly. Tedy celkem $1+4+6+4+1$. Což je překvapivě součet pátého řádku v Pascalově trojúhelníku.

Dle obou výpočtů nám vyšlo, že máme celkem 16 dvouvstupových funkcí. Najdete je v následující tabulce, kde jsme si je roztřídili do několika skupin: nejprve dvě funkce konstantní, pak čtyři s jednou jedničkou, čtyři s jednou nulou a konečně šest funkcí se dvěma jedničkami a dvěma nulami:

X	Y	‘0’	‘1’	AND	$>$	$<$	NOR	OR	$\geq$	$\leq$	NAND
1	1	0	1	1	0	0	0	1	1	1	0
1	0	0	1	0	1	0	0	1	1	0	1
0	1	0	1	0	0	1	0	1	0	1	1
0	0	0	1	0	0	0	1	0	1	1	1

X	Y	X	Y	XNOR	XOR	$\neg Y$	$\neg X$
1	1	1	1	1	0	0	0
1	0	1	0	0	1	1	0
0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	1

Úkolem třetím bylo dokázat, že všechny dvouvstupové funkce jdou postavit z hradel NAND.

Nejprve si všimneme, že všechny funkce postavíme z hradel AND, OR a NOT. Konstantní nula je x AND $\neg x$. Funkce obsahující právě jednu jedničku jsou (v pořadí podle naší tabulky) x AND y, x AND $\neg y$, $\neg x$ AND y a $\neg x$ AND $\neg y$. Funkce s více jedničkami dostaneme jako OR funkcí pro jednotlivé jedničky. Například x XOR $y = (x$ AND $\neg y)$ OR$(\neg x$ AND $y)$.

Teď už stačí dokázat, že z hradel NAND postavíme AND, OR a NOT. To je snadné: $\neg x = x$ NAND x, x AND $y = \neg(x$ NAND $y)$ a x OR $y = (\neg x)$ NAND$(\neg y)$.

Totéž by se dalo dokázat i jinak: Všimneme si, že každá funkce má svůj negovaný protějšek, takže stačí z hradel NAND postavit NOT a 8 vhodně vybraných funkcí, s nimiž rovnou dostaneme i jejich negace. Hradlo NOT opět složíme jako x NAND x, NAND už máme (a získáme AND), 0 je x NAND x (z toho 1), OR jest $\neg x$ NAND $\neg y$ (z toho NOR), XOR jest $(x$ OR $y)$ AND$(x$ NAND $y)$ (z toho XNOR), $>$ jest x AND $\neg y$ (z toho $\leq$) a $<$ jest y AND $\neg x$ (a máme i $\geq$). Tím je důkaz hotov.

Mimochodem, všechny dvouvstupové funkce by také šly vyrobit jen z hradel NOR. Vskutku: $\neg x = x$ NOR x, OR získáme jako $\neg(x$ NOR $y)$ a AND coby $(\neg x)$ NOR$(\neg y)$. Z těchto funkcí už umíme získat všechny zbylé.

Kdybychom počítali n-vstupové funkce, měla by naše tabulka 2^n řádků, takže by existovalo 2^{2^n} způsobů, jak ji vyplnit. Všimněte si, že trik vyjádřením všech funkcí pomocí AND, OR a NOT by stále fungoval, takže by stále stačil samotný NAND nebo NOR, jen by obvody byly trochu složitější.

Zkuste si dokázat, že hradla NAND a NOR jsou jediná takto univerzální. Třeba AND takový není proto, že každý obvod, ve kterém jsou jenom ANDy, na vstup ze samých nul odpoví zase nulou, takže nelze postavit (třeba) negaci.

Cyril Hrubiš

Úloha 20-1-2 – Příprava na cestu – program

```
#include <stdio.h>

int main() {
        int N, M;
        scanf("%d %d",&N,&M);
        double a[N+1][M+1], p[N+1][M+1];
        int m[N+1][M+1];

        for (int i = 0; i < N; i++)
                for (int j = 0; j <= M; j++) scanf("%lf ",&a[N-i][j]);  // trik - obrátíme pořadí pro jednodušší vypisování

        for (int i = 0; i <= M; i++) p[0][i] = 1;                       // neexistují žádné úkoly
        for (int i = 1; i <= N; i++)                                    // dynamika
                for (int j = 0; j <= M; j++) {
                        int maxk = 0;                                   // přidělíme úkolu i 0 minut
                        for (int k = 1; k <= j; k++)                    // zkusíme úkolu i přidělit k=1..j minut
                                if (a[i][k] * p[i-1][j-k] >= a[i][maxk] * p[i-1][j-maxk]) maxk = k;
                        p[i][j] = a[i][maxk] * p[i-1][j-maxk];          // nejlepší uložíme
                        m[i][j] = maxk;                                 // zapamatujeme si, kolik jsme úkolu přidělili minut
                }

        int zbyva = M;
```

Mohlo by se stát, že se nám taková časová složitost nelíbí. V takovém případě se pokusíme zbavit počítání multigrafů s násobnými hranami tím, že přepíšeme vzorečky, aby používaly pouze V_N^1 a μ_N^1. Postupně budeme rozkládat vše, co má horní index různý od 1. Tedy, $V_N^k = k \cdot V_{N-1}^1 + V_{N-1}^{k+1} = k \cdot V_{N-1}^1 + (k+1) \cdot V_{N-2}^1 + V_{N-3}^{k+2} = \ldots$. Zastavíme se, až budeme mít V_1^{k+N-1}, což je, jak jsme si rozmysleli výše, $k+N-1$. Obdobně to uděláme pro μ_N^k. Doporučuji si to napřed rozepsat pro třeba $N = 4$, je z toho hezky vidět, co vyjde.

Protože již k nepotřebujeme, pro zkrácení si označme V_N jako ekvivalent V_N^1. Obdobně pro μ_N a μ_N^k. Až práci s tužkou a papírem dokončíme, vyjde nám, že $V_N = 1 \cdot V_{N-1} + 2 \cdot V_{N-2} + \ldots + (N-1) \cdot V_1 + N$. Pro celá kormidla to vyjde $\mu_N = 1^2 \cdot V_{N-1} + 2^2 \cdot V_{N-2} + \ldots + (N-1)^2 \cdot V_1 + N^2$.

Kdybychom nám někdo dal všechny $V_1, \ldots, V_{N-1}$, není problém v lineárním čase spočítat μ_N sečtením všech sčítanců.

Zbývá tedy spočítat všechny vějíře, pokud možno také v lineárním čase. Kdybychom měli čísla $S_l := 1 + \sum_{i=1}^{l} V_i$ a $V_l = l + \sum_{i=1}^{l-1} (l-i) \cdot V_i$, jejich sečtením získáme V_{l+1} (čtenář si může ověřit sečtením). S_{l+1} získáme tak, že k S_l přičteme V_{l+1} (které již nyní máme také). Stačí doplnit startovní hodnoty. V_1 je jedna (vějířek s jedním krajním bodem je jen hrana), S_1 spočteme na 2. Všechny tedy zvládneme spočítat v $O(N)$.

Nyní si už stačí jen všimnout, že každé V_l potřebujeme jen k přičtení k celkovému výsledku (samozřejmě vynásobené správným číslem). Toto přičtení můžeme udělat okamžitě, tudíž ho již příště nepotřebujeme a není třeba uchovávat pole se všemi. Tím k lineární časové složitosti získáme jako bonus konstantní paměťovou.

Program si můžeme zjednodušit dopočítáním V_0 a S_0 (na 0 a 1), čímž zjednodušíme chování cyklu a celkový součet můžeme přepočítat už po spočtení V_1.

Michal „vorner“ Vaner

Poznámka M.M.: Každý pravověrný matematik samozřejmě věří, že na libovolný „počítací“ problém existuje chytrý vzoreček. Někdy je i hezký :) Pokud na formulky pro μ_N z našeho vzorového řešení použijete techniku zvanou metoda vytvořujících funkcí (ta je moc pěkně popsaná ve starých dobrých Kapitolách z diskrétní matematiky), dostanete následující pěkný vztah (časem – ono dá docela dost práce se tím vším propočítat, takže detaily si pro tentokrát odpustíme):

$$\mu_N = \alpha^N + \beta^N - 2,$$

kde α a β jsou konstanty definované takto:

$$\alpha = \frac{3+\sqrt{5}}{2}, \quad \beta = \frac{3-\sqrt{5}}{2}.$$

Pro počítání v programu to žádná velká výhra není, protože stěží dovedeme iracionální odmocniny z pěti reprezentovat dost přesně. Můžeme si ale pomoci drobným úskokem: Podobně jako se počítá s komplexními čísly jako s výrazy typu $a + b\sqrt{-1}$, my budeme počítat s dvousložkovými čísly ve tvaru $a + b\sqrt{5}$, kde a a b jsou racionální. Jelikož součet, rozdíl i součin takových čísel je opět číslo v tomto tvaru, můžeme vše počítat v nich a na konci pouze vypsat první složku. (Víme totiž, že výsledek je přirozené číslo, a tak musí být druhá složka nulová. Navíc díky symetrii bude první složka u α^N stejné jako u β^N, takže stačí počítat jen jednu z nich). Ještě si vzpomeneme na trik na rychlé umocňování (viz třeba řešení úlohy 18-4-1) a vyloupne se následující program, který μ_N spočítá v čase $\mathcal{O}(\log N)$.

```
/* Dvojsložková čísla a jejich násobení */
typedef struct { int i, j; } num;
num mul(num x, num y)
{ return (num){ x.i*y.i + 5*x.j*y.j,
                x.i*y.j + x.j*y.i }; }

int M(int n)
{
  num x={3,1}, y={1,0};    // x=2*alfa
  for (int i=n; i; i/=2)   // počítáme y=x^n
    {
      if (i%2)
        y = mul(y,x);
      x = mul(x,x);
    }
  return ((2*y.i) >> n) - 2;
}
```

20-1-5 Praktická – Studentův rozvrh

K řešení tohoto problému použijeme jednoduchý *„hladový“* (*„greedy“*) algoritmus. Na začátku si nejprve všechny přednášky setřídíme vzestupně podle času konce f_i (pozor, pokud je setřídíme jinak - např. podle času začátku s_i, tak to nebude fungovat). V průběhu algoritmu si budeme navíc držet čas konce poslední dosud vybrané přednášky (označíme ho F a na počátku bude nastaven na hodnotu mínus nekonečno).

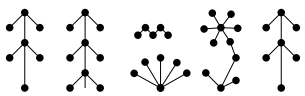
Samotný algoritmus je vlastně pouze jedním cyklem přes setříděné přednášky, při kterém si hladově vybereme, které přednášky vezmeme, a které ne. Na každou přednášku i se podíváme a porovnáme její začátek s číslem F. Pokud je $F \leq s_i$, pak si přednášku vybereme a aktualizujeme hodnotu $F = f_i$. V opačném případě víme, že se přednáška nevejde do rozvrhu a tak se s ní nemusíme dále zdržovat.

Jak vidíte, algoritmus je v zásadě lehký. Zbývá ukázat, že také funguje. Množina přednášek, kterou náš program vydá, je určitě nezávislá (žádné přednášky se v ní nepřekrývají). To je vidět na první pohled přímo z definice hladového výběru. Ovšem není úplně jisté, jestli je tato množina také maximální možná (tzn. jestli neexistuje jiná nezávislá množina, ve které by bylo víc přednášek).

Abychom měli jistotu, že je naše množina také maximální, musíme vědět, že nám výběr některé přednášky nezablokuje lepší řešení. Řekněme, že jsme právě vybrali do našeho rozvrhu nějakou přednášku a chceme ukázat, že tato přednáška nezablokuje výběr dvou (nebo více) přednášek, které bychom mohli zapsat místo ní.

Budeme postupovat sporem. Nechť jsme vybrali přednášku x, která nám zablokovala výběr přednášek y a z (pro nástin důkazu nám dvě stačí, ale počet by šel libovolně rozšířit). Protože přednášky vybíráme setříděné podle času konce, musí rozhodně platit: $f_x \leq f_y$ a $f_x \leq f_z$. Navíc přednáška x blokuje přednášky y a z, což lze zapsat jako $s_y \leq f_x$ a $s_z \leq f_x$. Nemusíme mít doktorát z matematiky, abychom si všimli, že přednášky y a z spolu musí také kolidovat (minimálně prochází obě časovým bodem f_x). To je ovšem spor s předpoklady, protože přednášky y a z nelze obě zařadit do rozvrhu místo x.

Tím jsme ukázali, že rozvrh spočítaný naším algoritmem bude nejen korektní z hlediska nezávislosti, ale také největší možný.



Les, jak ho vidí matematici

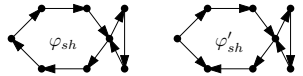
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf.* Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“, je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny.* Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů Kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i, j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	1	1
2	1	0	1	1	0	0	0	0	1
3	1	0	0	1	0	0	0	0	0
4	0	1	1	0	1	0	0	0	0
5	0	1	0	1	0	1	0	0	0
6	0	0	0	0	1	0	1	1	0
7	0	0	0	0	0	1	0	1	1
8	1	0	0	0	0	1	1	0	0
9	1	1	0	0	0	0	1	0	0

- *seznam sousedů* je obvykle tvořen dvěma poli: polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $O(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7

i	1	2	3	4	5	6	7	8	9	10
$Z[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `Sousedi`, poli Z `Zacatky` a nadeklarujeme si je takto:

```
var N, M: Integer;  { počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of Integer;
    Sousedi: array[1..MaxM] of Integer;
```

Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá

first in, first out, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme do zásobníku a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M, čili $O(N+M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Oznacen: array[1..MaxN] of Boolean;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V] := True;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if not Oznacen[Sousedi[I]] then
      Projdi(Sousedi[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $O(N_i+M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $O(N+M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $O(N+M)$.

```
var Komponenta: array[1..MaxN] of Integer;
    NovaKomponenta: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V] := NovaKomponenta;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Komponenta[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
end;

var I: Integer;
begin
  ...
  for I := 1 to N do Komponenta[I] := -1;
  NovaKomponenta := 1;
  for I := 1 to N do
    if Komponenta[I] = -1 then
      begin
        Projdi(I);
        Inc(NovaKomponenta);
      end;
  ...
end.
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany*. *Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.

Mějme tedy přihrádkově setříděné dva sousední sloupečky děr podle řádků a seznam děr pro každý z obou sloupečků (seznamy vznikly při prvním setřídění děr podle souřadnice X). Nyní se stačí pro každou z děr v seznamech podívat do přihrádek, jestli má sousedku nad sebou nebo pod sebou (pro $x \geq 2$ to stačí pro díry ze sloupce $x+1$, protože sloupec x jsme vyřešili v předchozím kroku). Takto jsme doplnili svislé hrany do grafu (hrany odpovídající sousedství dvou děr ve svislém směru). Dále budeme hledat sousedící díry z právě zpracovávaných dvou sloupečků ve vodorovném směru. Pro každou díru ze sloupce x na řádku y (a tedy v přihrádce y pro tento sloupec) se podíváme, zda je nějaká díra v přihrádce y sloupečku $x+1$. Tím, že projdeme všechny sousední dvojice sloupečků, máme zajištěno, že nalezneme všechny vodorovné hrany našeho grafu. Pro každou díru si budeme v celku pamatovat max. 4 odkazy na sousedy (paměť $\mathcal{O}(D)$).

Všimněte si, že se tento postup opět podobá prvnímu, ale s tím rozdílem, že nemáme „napřihrádkovaný“ úplně celý trup, ale jen tu část, kterou právě potřebujeme, a zbytek – ostatní sloupečky – jen částečně – tj. neřešíme jejich rozmístění děr na řádcích. Celková paměťová složitost odpovídá časové, tj. $\mathcal{O}(M+N+D)$

Josef Pihera

20-1-4 Kormidlo

Zdá se, že tato úloha byla těžší, než se z počátku zdálo. Správných řešení přišlo pomálu, ty rychlé v podstatě žádné, takže Vildovi nezbylo než přibýt místo kormidla jeden obdélníkový kus dřeva, který mu zbyl z opravy.

Úkolem je vlastně spočítat počet koster na daném grafu. Co je to kostra a graf se dočtete kupříkladu v kuchařce ke třetí sérii, kterou jste právě dostali.

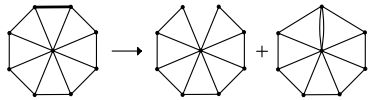
Vzorec na výpočet počtu koster na úplném grafu nám nepomůže, protože kormidlo není úplný graf. Stejně tak postupy pro obecné grafy jsou trochu jako nukleární bomba na vrabce. Jde to jednodušeji.

Tedy, naší úlohou je najít počet koster určeného grafu. Úlohu si mírně zobecníme. V grafu je obvykle zakázané mít násobné hrany (více hran spojující stejnou dvojici vrcholů). My toto zakazovat nebudeme, čímž dostaneme multigraf. K čemu nám to bude dobré si povíme později.

Máme tedy multigraf M. Vyberme si jednu multihranu (multihrana jsou všechny „normální“ hrany, které spojují stejné 2 vrcholy). Rozdělíme si množinu koster grafu M podle této multihraně na dvě (disjunktní) podmnožiny.

První podmnožina bude obsahovat všechny kostry, které neobsahují žádnou hranu z této multihrany. Velikost takové množiny je zjevně stejná, jako velikost množiny všech koster grafu M^-, který vznikne z M odebráním celé této multihrany.

Druhá podmnožina je ten zbytek, tedy všechny kostry, kde použijeme právě jednu hranu z této multihrany (více jich vést nemůže, to by nebyla kostra). Kdyby naše multihrana nebyla násobná (byla by to jen obyčejná hrana), velikost této podmnožiny by byla stejná jako počet koster na grafu $M^{\Rightarrow\Leftarrow}$, který z M vznikne odstraněním této multihrany a sloučením vrcholů touto multihranou spojených do jednoho (toto je proč celou dobu pracujeme s multigrafy – tady mohou vznikat multihrany).



Jak to ale bude vypadat, když námi vybraná hrana bude h-násobná? Úplně stejně, jako s jednoduchou, jen použitou hranu můžeme vybrat h způsoby, tedy výsledkem bude $h \cdot M^{\Rightarrow\Leftarrow}$.

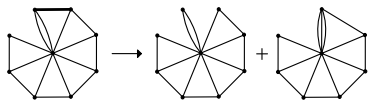
Protože jsou tyto dvě podmnožiny disjunktní a dohromady dávají celou množinu koster (nic jiného, než že tam hrana je a že tam není, se stát nemůže), můžeme velikosti těchto dvou podmnožin jednoduše sečíst.

Tímto převedeme problém počtu koster na multigrafu na dva stejné problémy, ale na menších multigrafech (čímž jsme mimochodem dokázali, že algoritmus je konečný, neboť počet koster na jednovrcholovém grafu je roven jedné a počet koster na nesouvislém grafu je nula). Nyní stačí už jen využít toho, že vstupní graf není jen tak ledajaký, ale že je to naše pěkné kormidlo.

Podívejme se, na co se rozloží kormidlo velikosti N. Vybereme si jednu hranu na jeho obvodu. Když hranu vynecháme, vznikne něco, co by se dalo nazvat vějířem (viz obrázek). Když hranu použijeme, vznikne skoro totéž, jako kormidlo velikosti $N-1$, jen s tím rozdílem, že jedna hrana do středu je dvojitá.

Kormidlo velikosti N s jednou k-násobnou hranou se rozloží na vějíř velikosti N s jednou $(k+1)$-násobnou hranou na kraji (vybereme si opět hranu sousedící s onou k-násobnou hranou) a jedno kormidlo velikosti $N-1$ s jednou $(k+1)$-násobnou hranou.

Co uděláme s vějířem velikosti N a k-násobnou krajní hranou? Vybereme si vnější hranu, která sousedí s tou k-násobnou. Když ji použijeme, dostaneme vějíř velikosti $N-1$ s jednou $k+1$-násobnou hranou. Když ji nepoužijeme, dostaneme vějíř velikosti $N-1$ na násobné stopce. Protože do toho vrcholu na konci stopky vede už jen tato multihrana, musíme ji použít a počet koster takové kostry bude stejný jako počet koster vějíře velikosti N vynásobeným k (máme k způsobů, jak připojit stopkový vrchol).



Nyní, kdy toho necháme? Vějíře se jednou stáhnou až do jedné k-násobné hrany (na které najdeme k různých koster). Když nám nebude vadit myšlenka existence kormidla velikosti 1 s jednou k-násobnou hranou, všimneme si, že je to opět hrana samotná (spojující „krajní“ se „středovým“ vrcholem).

Nyní trocha počtů. Označme μ_N^k počet koster korμdla o velikosti N s jednou k-násobnou hranou. Stejně tak V_N^k budiž počet koster vějíře velikosti N s jednou k-násobnou hranou. Pomocí našeho rozkládacího pravidla si vyjádříme, že $V_N^k = V_{N-1}^{k+1} + k \cdot V_{N-1}^1$. Stejně tak $\mu_N^k = V_N^k + \mu_{N-1}^{k+1}$. Toto je jen přepis výše zmíněných rozkladů na menší podproblémy.

Kdybychom nyní iterovali přes všechny potřebná N a k (všimněme si, že k bude nejvýše N – až na nějaké malé konstanty okolo), tak se zajisté dobereme k výsledku. Když si budeme mezivýsledky ukládat (některé budeme potřebovat vícekrát), tak se dostaneme na časovou složitost $O(N^2)$.

$a_{ik} \cdot p_{i-1,j-k}$. Toto nejlepší k si také uložíme, protože pro nás znamená nejlepší přidělené minuty pro tento konkrétní úkol a na konci ho budeme potřebovat vypsat.

Ještě zbývá vymyslet, jak zaplníme první řádek matice P. Můžeme si pomoct tím, že si na začátek přidáme nultý řádek, který bude na pozicích p_{00} až p_{0M} zaplněn samými jedničkami, což si také můžeme vyložit tak, že hodnoty označují mágovu spokojenost, když nebude žádný úkol a bude pro něj k dispozici $0..M$ minut. Potom můžeme uvedený algoritmus nastartovat od prvního řádku matice (od prvního úkolu).

Protože musíme zaplnit matici P, která má N řádků a $M+1$ sloupců a protože v každém políčku musíme udělat $\mathcal{O}(M)$ testů, má celý algoritmus časovou složitost $\mathcal{O}(NM^2)$.

Jana Kravalová

20-1-3 Oprava lodi

V naprosté většině případů měl Vilda naději, že nebude spolu s lodí muset prozkoumávat dno, což mu udělalo velkou radost. Proto si všechno o záplatování trupu pečlivě zaznamenal:

K řešení využijeme toho, že žádné dvě záplaty se nebudou překrývat, pokud mají nejmenší možnou velikost. Rovněž víme, že záplata musí pokrývat celou souvislou oblast všech děr. Nemusíme tedy příliš hloubat, abychom objevili, že rozměr i poloha záplaty jsou dány nejlevější, nejhořejší, nejpravější a nejspodnější dírou celé souvislé oblasti, ty v tomto pořadí přesně určují levý, horní, pravý a dolní okraj záplaty. Cokoliv menšího je málo, cokoliv většího by bylo zbytečné. Ke spočtení velikosti záplat nám tedy stačí projít všechny oblasti a určit jejich horní, dolní, levé a pravé okraje.

Teď se musíme vypořádat s tím, jak hledat souvislé oblasti děr. Začněme tím, že všechny díry jsou v dvojrozměrném poli *diry*. Prvek $diry[x, y]$ je logická hodnota odpovídající tomu, zda na souřadnicích $[x, y]$ díra je nebo není.

Ještě než začneme se samotným hledáním, připomeneme si jednu datovou strukturu – *frontu*. Fronta není nic jiného než nějaký seznam prvků, avšak dvě operace jsou pro ni specifické – přidání prvku na konec a odebrání prvku ze začátku. Prvky jsou tedy z fronty odebírány ve stejném pořadí, v jakém do ní byly přidány a programátorova fronta se od fronty v obchodě liší pouze tím, že se v ní nesmí předbíhat.

Nyní zpět k původnímu problému. Souvislé oblasti budeme hledat *hledáním do šířky*. Těm z vás, kteří o tomto algoritmu slyší poprvé, vřele doporučuji se s ním blíže obeznámit (je popsán v aktuální kuchařce). Na počátku si vybereme nějakou libovolnou dosud nezazáplatovanou díru a uložíme si ji do fronty. Nyní budeme postupně odebírat díry z fronty a pro každou z nich se podíváme, zda nemá vedle sebe sousedku. Sousedku hledáme prostě tak, že se v poli s dírami podíváme na políčka kolem sebe. Pokud takovou díru nalezneme, přidáme si ji na konec fronty a v poli děr si ji poznačíme jako zpracovanou, abychom ji nevkládali do fronty vícekrát. Povšimněme si, že záplatu můžeme určovat už během procházení frontou. Na počátku řekneme, že záplata zakrývá právě jednu díru, a to tu první, kterou jsme nalezli a přidali do fronty. S každou další dírou nám stačí záplatu „roztáhnout“, pokud z ní díra vybočuje. Při dosažení konce fronty jsme určitě nalezli všechny díry v dané oblasti a velikost záplaty pro tuto oblast přičteme k celkovému výsledku. Zbývá už jen po nalezení všech oblastí vypsat celkový součet.

Náš algoritmus potřeboval pole popisující celkem $M \cdot N$ políček, tedy paměťová složitost je $\mathcal{O}(M \cdot N)$. Každé políčko zpracujeme pouze konstantně-krát, navíc přinejmenším na počátku muselo být inicializováno, takže s každým jsme pracovali alespoň jednou. Tedy časová složitost je $\mathcal{O}(M \cdot N)$. Povšimněte si, že $D \leq M \cdot N$, proto ho můžeme „schovat“ do $\mathcal{O}(M \cdot N)$.

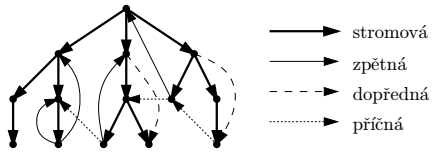
Ještě se zamyslíme, jestli neexistuje vylepšení algoritmu pro D, která jsou mnohem menší než $M \cdot N$. Pokusme se zbavit závislosti na M a N, která mohou být neúměrně větší než D i bez toho, že by se mezi sebou vynásobili. Odstraňme proto ono dvojrozměrné pole pro díry a ponechme si jen posloupnost děr takovou, jakou jsme ji načetli ze vstupu. Pole, kterého jsme se zbavili, jsme používali k tomu, abychom uměli rychle naleznout sousední díry v průběhu hledání do šířky. Jak ale teď nalezneme sousední díry? Abychom to uměli rychle, uspořádáme si na počátku všechny díry podle souřadnice X a pokud se v ní shodují, tak podle Y. Nyní řekněme, že máme díru se souřadnicemi $[x, y]$. Pak její sousedky mohou být pouze $[x-1, y]$, $[x+1, y]$, $[x, y-1]$ nebo $[x, y+1]$. Jsou tedy celkem čtyři možnosti, které musíme vyzkoušet. Tedy hledáme čtyři hodnoty v uspořádané posloupnosti, což umíme půlením intervalu v logaritmickém počtu kroků vzhledem k délce takové posloupnosti. Pokud taková díra existuje, pak jsme k ní zřejmě připojeni a to už algoritmu stačí.

Celý algoritmus potřebuje pouze množinu všech děr, kterých je D, a frontu pro průchod do šířky, která může rovněž obsahovat až D prvků. Paměťová složitost je tedy $\mathcal{O}(D)$. Časovou složitost nejvíce ovlivňuje setřídění děr a prohledávání souvislých oblastí, kde pro každý prvek provádíme půlení intervalu v logaritmickém čase. Obě hlavní fáze a tedy celý algoritmus běží v čase $\mathcal{O}(D \cdot \log D)$. Mezi zdrojovými kódy vzorových řešení naleznete pouze tuto druhou variantu algoritmu, která je ve skutečnosti poměrně jednoduchou úpravou varianty první.

Ještě existuje jedno řešení, kterého si správně všiml Peter Ondrúška. Úlohu lze řešit v čase $\mathcal{O}(M+N+D)$. Zvídavější z vás si nejspíše uvědomili, že prohledáváme graf a hledáme slabě souvislé komponenty. To samo o sobě umíme rychle, ale v tomto případě nám nejdéle trvá konstrukce grafu. Na počátku si přihrádkově setřídíme všechny díry podle souřadnice X, tedy je rozdělíme do sloupečků(zvládneme v čase $\mathcal{O}(N+D)$). Pomocí tohoto rozdělení dokážeme později lépe určit vodorovně sousedící díry. Začneme postupně zpracovávat všechny sloupečky x a $x+1$ pro x od 1 do $M-1$. Díry těchto dvou sloupečků si vždy setřídíme podle řádků - opět přihrádkově a oba sloupečky zvlášť. Zde musíme dát pozor, abychom netřídili pro každou dvojici sloupečků v čase $\mathcal{O}(M)$. Budeme opakovaně používat dvě přihrádková pole velikosti M (tedy co řádek, to přihrádka). Před prvním tříděním podle řádků si je musíme vynulovat ($\mathcal{O}(M)$). V dalších krocích (pro další dvojice sloupečků) předpokládáme, že pole jsou vynulována, a my proto děláme zápisy jen pro každou díru, nikoliv pro každou přihrádku – to je zřejmé, protože používáme jen ty přihrádky, které potřebujeme. Až nebudeme obsah polí potřebovat, opět vymažeme pouze ty díry, které jsme přidali, tedy nebudeme nulovat všechny přihrádky v poli – v důsledku třídění pro všechny dvojice sloupečků a všechny díry na plánu stihneme v $\mathcal{O}(M+D)$.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n+1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $O(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
var Fronta, H: array[1..MaxN] of Integer;
    I, V, Prvni, Posledni: Integer;
    PocatecniVrchol: Integer;
begin
  ...
  for I := 1 to N do H[I] := -1;
  Prvni := 1;
  Posledni := 1;
  Fronta[Prvni] := PocatecniVrchol;
  H[PocatecniVrchol] := 0;

  repeat
    V := Fronta[Prvni];
    for I := Zacatky[V] to Zacatky[V+1]-1 do
      if H[Sousedi[I]] < 0 then begin
        H[Sousedi[I]] := H[V]+1;
        Inc(Posledni);
        Fronta[Posledni] := Sousedi[I];
      end;
    Inc(Prvni);
  until Prvni > Posledni;  { Fronta je prázdná }
  ...
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp. z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což nelze splnit.

Cyklus je ovšem to jediné, co muže existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme

po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále a nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $O(N + M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili nižší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $O(N + M)$.

```
var Ocislovani: array[1..MaxN] of Integer;
    Posledni: Integer;
    I: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Ocislovani[V] := 0; { zatím V jen označíme }
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Ocislovani[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
  Inc(Posledni);
  Ocislovani[V] := Posledni;
end;


begin
  ...
  for I := 1 to N do
    Ocislovani[I] := -1;
  Posledni := 0;
  for I := 1 to N do
    if Ocislovani[I] = -1 then Projdi(I);
  ...
end.
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most*.

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“ pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany*.

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $O(N+M)$. Zde jsou důležité části programu:

```
var Hladina, Spojeno: array[1..MaxN] of Integer;
    DvojSouvisle: Boolean;
    I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I, W: Integer;
begin
  Hladina[V] := NovaHladina;
  Spojeno[V] := Hladina[V];

  for I := Zacatky[V] to Zacatky[V+1]-1 do
  begin
    W := Sousedi[I];
    if Hladina[W] = -1 then
    begin { stromová hrana }
      Projdi(W, NovaHladina + 1);
      if Spojeno[W] < Spojeno[V] then
        Spojeno[V] := Spojeno[W];
      if Spojeno[W] > Hladina[V] then
        DvojSouvisle := False; { máme most }
    end else { zpětná nebo dopředná hrana }
    if (Hladina[W] < NovaHladina-1) and
       (Hladina[W] < Spojeno[V]) then
      Spojeno[V] := Hladina[W];
  end;
end;

begin
  ...
  for I := 1 to N do
    Hladina[I] := -1;
  DvojSouvisle := True;
  Projdi(1, 0);
  ...
end.
```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

Dnešní menu Vám servírovali
Martin Mareš, David Matoušek a Petr Škoda

Vzorová řešení první série dvacátého ročníku KSP

20-1-1 Temná šachovnice

„Ach jo, co si to ti mágové dnes nevymyslí,“ povzdechl si Vilda a začal vysvětlovat temnému pánovi, proč jeho příkazu nemůže vyhovět:

Na šachovnici 4×4, a vůbec na všech šachovnicích o sudém rozměru, se při výchozím obarvení nachází sudý počet jak černých, tak bílých políček. Zaměřme se třeba na černá. („No proto!“ pochvaloval si mág.)

Dokážu, že když je na takové šachovnici před tahem počet černých políček sudý, po tahu bude sudý také. Při libovolném tahu mohu změnit počet černých políček následovně: V daném sloupci/řádku byl počet černých políček sudý, což znamená, že bílých bylo také sudě. Takže když se po tahu tyto počty prohodí, zůstanu pořád na sudém počtu černých políček v tomhle sloupci/řádku. Protože byl ale celkový počet černých políček před tahem sudý, tak když odečtu sudý počet původně černých políček a přičtu sudý počet nově černých políček, dostanu opět sudé číslo.

Naopak, mohlo se mi stát, že počet černých políček v daném sloupci/řádku byl lichý, což také znamená, že bílých políček bylo také liše. Když se po tahu tyto počty prohodí, celkem dostanu opět sudý počet černých políček (od původního počtu odečtu liché číslo, ale pak zase jiné liché číslo přičtu).

„Z toho bohužel vyplývá, že šachovnici na požadovaný tvar upravit nedokážu – ve výsledku mám dostat lichý počet černých políček, což opravdu nejde,“ dokončil svůj výklad Vilda. „Hm,“ zamyslel se mág, „chápu, že pro šachovnice o sudém rozměru to nejde, nepůjde to ale pro šachovnice s lichým počtem políček na straně?“ snažil se přesvědčit Vildu a začal si připravovat pilku pro případné ořezání šachovnice.

„No, to bohužel nepůjde také,“ zklamal ho Vilda a pokračoval ve vysvětlování:

Každá šachovnice o rozměru větším než 2 má v levém horním rohu *podšachovnici* o rozměru 2×2. Tuto podšachovnici potřebujeme obarvit tak, aby levé horní políčko bylo bíle a zbylá tři černá. Když se na chvíli zamyslíme, zjistíme, že na změnu libovolného políčka podšachovnice mají vliv jenom inverze prvních dvou sloupců/řádků. Odmysleme si zbytek těchto sloupců/řádků, jsme opět tam kde jsme byli – snažíme se přebarvit šachovnici o straně 2 a o té už víme, že přebarvit nejde.

„Mám však taky jednu dobrou zprávu,“ rozjasnil se Vilda, „šachovnici 1×1 lze obarvit velice snadno!“ načež si od mága vypůjčil pilku, vyřezal levé horní políčko a obarvil ho bíle. Mág chvíli na šachovnici hleděl silně podezřívavým pohledem, pak sáhl do kapsy, vytáhl figurku dámy a položil ji na políčko. „Tak, zahrajem si?“

Mária Vámošová

20-1-2 Příprava na cestu

Každého jistě napadne, že bychom v KSP nevypisovali úlohu za 12 bodů, kdyby se měla řešit prostým prohledáváním všech možností neboli obyčejným backtrackem. Nezachrání nás ani různé heuristiky a pokusy sem tam odřezat neperspektivní větev výpočtu. My si ukážeme, že pomocí šikovného ukládání již spočítaných výsledků dokážeme najít řešení v polynomiálním čase.

Celá myšlenka spočívá v následující úvaze: Představme si, že bychom věděli, jak nejlépe vyřešit tuto úlohu pro $N-1$ úkolů a zadaný počet minut. Tedy že pro $N-1$ úkolů a daný počet minut víme, kolik minut věnovat kterému úkolu a jaká bude výsledná pravděpodobnost, že bude mág s Vildovou prací spokojen. (Zatím nemusíme moc přemýšlet nad tím, jak jsme k této znalosti přišli, prostě ji máme.) Kdyby nám teď někdo přidal N-tý, poslední úkol a dal nám na vyřešení všech úkolů M minut, stačí udělat jednoduchou věc: zkusíme poslednímu N-tému úkolu postupně přidělit všechny možné počty minut, které přichází v úvahu (tedy $k = 0..M$ minut) a na zbylých $N-1$ úkolů použít zbylých $M-k$ minut. Pravděpodobnost, že bude mág spokojen, bude součinem čísla a_{Nk} (na úkol N jsme použili k minut) a nejlepší pravděpodobnosti, kterou můžeme na $N-1$ úkolech při $M-k$ minutách dosáhnout (toto číslo máme dopředu spočítané). Pak nám stačí ze všech možných k vybrat to nejlepší.

Dobře, ale jak spočítáme nejlepší pravděpodobnost, které můžeme dosáhnout na $N-1$ úkolech pro nějaký zadaný počet minut? No, představme si, že toto číslo už známe pro $N-2$ úkolů...

A teď si algoritmus představíme pořádně:

Máme zadánu matici $A = aij$ kde prvek a_{ij} udává, jaká je pravděpodobnost, že bude mág s Vildovým řešením úkolu i spokojen, pokud mu Vilda věnuje j minut. Zavedeme si matici $P = pij$, kde prvek p_{ij} bude znamenat nejlepší možnou pravděpodobnost, jakou můžeme dosáhnout na úkolech $1..i$ při přidělených minutách j. Hodnota p_{23} tedy například udává, jaká je nejlepší možná pravděpodobnost mágovy spokojenosti pro úkoly 1 a 2 s přiděleným počtem minut 3.

Matici P budeme zaplňovat po řádcích od levého horního rohu, který odpovídá nejlepší pravděpodobnosti pro 1 úkol a 0 minut, až do pravého dolního rohu, který odpovídá nejlepší pravděpodobnosti pro N úkolů a M minut a je vlastně řešením naší úlohy.

Chceme zaplnit pole p_{ij}, které by mělo odpovídat nejlepší možné pravděpodobnosti pro úkoly $1..i$ a přidělené minuty j. Trošku jsme to naznačili už v úvodní myšlence. Vyzkoušíme všechna možná přidělení minut $k = 0..j$ pro poslední i-tou úlohu a do hodnoty p_{ij} dosadíme maximum z hodnot