**Výsledková listina šestnáctého ročníku KSP po druhé sérii**

| | | škola | ročník | 1621 | 1622 | 1623 | 1624 | 1625 | suma | celkem |
|---|---|---|---|---|---|---|---|---|---|---|
| 1. | Marek Jančuška | G Nitra | 4 | 10 | 11 | 10 | 8 | | 39 | 88 |
| 2. | Peter Perešíni | GJGTajov | 2 | 6 | 10 | 10 | 10 | 0 | 36 | 86 |
| 3. | Miroslav Cicko | GJGTajov | 3 | 6 | 5 | 10 | 10 | 2 | 33 | 81 |
| 4. | Jan Bulánek | G Klatovy | 3 | 9 | 5 | 10 | 6 | | 30 | 80 |
| 5. | Petr Škoda | GÚstavní | 4 | 5 | 9 | 6 | 10 | 3 | 33 | 79 |
| 6. | Kryštof Hoder | GKptJaroš | 4 | 6 | 8 | 10 | 10 | | 34 | 65 |
| 7. | Pavel Motloch | GPBezruče | 1 | 5 | 3 | 2 | 7 | 1 | 18 | 63 |
| 8. | Michal Repovský | | 4 | 10 | 5 | 10 | 1 | | 26 | 61 |
| 9. | Jana Kravalová | G VKlobou | 4 | | 3 | 6 | 10 | | 19 | 58 |
| 10. | Miroslav Klimoš | G Lanškr | 0 | 3 | 5 | 7 | 10 | | 25 | 55 |
| 11. – 13. | Ondřej Bílka | G Zlín | 2 | 2 | 3 | 9 | 8 | | 22 | 51 |
| | Zbyněk Falt | GNeumannov | 3 | 2 | 3 | 6 | | | 11 | 51 |
| | Jan Hrnčíř | GFXŠaldy | 2 | 2 | 3 | 10 | 6 | | 21 | 51 |
| 14. | David Matoušek | GZborov | 4 | | 2 | 9 | 8 | | 19 | 48 |
| 15. | Peter Černo | GLŠtúra | 3 | 6 | | 6 | | | 12 | 46 |
| 16. | Martin Podloucký | G Strážnic | 3 | 2 | 1 | 10 | 3 | | 16 | 45 |
| 17. – 18. | Jana Fabriková | GKptJaroš | 4 | 2 | 5 | 6 | 10 | 2 | 25 | 44 |
| | Ondřej Garncarz | G Příbor | 3 | 3 | 3 | 6 | 5 | 1 | 18 | 44 |
| 19. | Tomáš Gavenčiak | G Bílovec | 4 | | | | | | 0 | 43 |
| 20. – 21. | Michal Bečka | G MTřebová | 4 | | 3 | 4 | 8 | | 15 | 42 |
| | Daniel Marek | GZborov | 1 | | 10 | | 10 | | 20 | 42 |
| 22. | Martin Koníček | G UBrod | 3 | | 3 | 6 | 7 | | 16 | 38 |
| 23. | Martin Křivánek | GKptJaroš | 2 | | | | | | 0 | 37 |
| 24. – 26. | Martin Čech | G UBrod | 3 | | 2 | 6 | | | 8 | 36 |
| | Peter Šufliarsky | G NZámky | 4 | | 3 | 10 | | | 13 | 36 |
| | Ján Záhornadský | GZborov | 3 | 2 | 2 | 5 | 5 | | 14 | 36 |
| 27. – 29. | Pavel Klavík | G Chrudim | 1 | 2 | 1 | 5 | 9 | 1 | 18 | 35 |
| | Petr Soběslavský | GJHeyrovs | 3 | 0 | 3 | 6 | 7 | 1 | 17 | 35 |
| | Petr Švec | G Beroun | 4 | 2 | 3 | 10 | | | 15 | 35 |
| 30. – 31. | Marek Blahuš | G UHradi | 3 | 3 | 3 | 10 | 8 | | 24 | 34 |
| | Filip Šauer | G Klatovy | 3 | | 2 | 6 | | | 8 | 34 |
| 32. – 33. | Jindřich Flídr | G Lanškr | 4 | | | | | | 0 | 33 |
| | Stanislav Haviar | G Klatovy | 3 | | 3 | 7 | | | 10 | 33 |
| 34. – 35. | Jaroslav Havlín | G Sedlča | 4 | 3 | 3 | | | | 6 | 32 |
| | Martina Tomisová | GZborov | 4 | 3 | 2 | 2 | 4 | | 11 | 32 |
| 36. | Petr Kortánek | G Sedlča | 2 | | 2 | | | | 2 | 31 |
| 37. – 38. | Marek Ludha | GJGTajov | 4 | | | | | | 0 | 30 |
| | Adam Přenosil | GSladkNám | 2 | 2 | 5 | | 8 | | 15 | 30 |
| 39. – 40. | Stanislav Basovník | G Kroměříž | 3 | | | | | | 0 | 24 |
| | Jan Křetínský | GMLercha | 4 | | | | | | 0 | 24 |
| 41. | Eva Schlosáriková | G Piešťany | 3 | 2 | 2 | 6 | 5 | | 15 | 22 |
| 42. | Benjamin Vejnar | G Nymburk | 4 | | | | | | 0 | 20 |
| 43. | Milan Dvořák | G NMnMor | 1 | | | | | | 0 | 16 |
| 44. – 45. | Cyril Hrubiš | G Bílovec | 2 | | | | | | 0 | 14 |
| | Petr Kratochvíl | | 1 | | 3 | 9 | | | 12 | 14 |
| 46. – 47. | Jiří Bělohradský | | 2 | | | | | | 0 | 13 |
| | Jan Richter | G Příbor | 3 | | 2 | | | | 2 | 13 |
| 48. – 49. | Kristýna Knapová | G Jičín | 4 | | 2 | 10 | | | 12 | 12 |
| | Petr Musil | G MBuděj | 2 | | 1 | | 5 | | 6 | 12 |
| 50. | Martin Kupec | GMendel | 2 | | | | | | 0 | 11 |
| 51. | Michal Potfaj | G NMnVáh | 4 | | | | | | 0 | 10 |
| 52. | Martin Schmid | G ČTřebová | 0 | | | | | | 0 | 9 |
| 53. – 55. | David Irschik | G Ledeč | 3 | | | | | | 0 | 7 |
| | Petr Paščenko | GDašická | 4 | | | | | | 0 | 7 |
| | Jaromír Vojíř | G Ledeč | 3 | | | | | | 0 | 7 |
| 56. | Radoslav Sopoliga | G Svidník | 4 | | 2 | | 0 | | 2 | 6 |
| 57. | Tomáš Herceg | | 1 | | | | | | 0 | 4 |
| 58. | Jindřich Pergler | G Klatovy | 3 | | 3 | | | | 3 | 3 |
| 59. | Aleš Razým | | 3 | | | | | | 0 | 2 |



*Milí řešitelé!*

Do ruky se Vám dostává již třetí letošní zadání našeho semináře. S touto sérií přichází také několik změn. Jednak jsme se rozhodli, že letošní ročník bude mít pouze čtyři série, a proto jsme zvedli počet úloh v sérii na *šest*.

Další změna se týká toho, že jsme se rozhodli přijímat Vámi vypracovaná řešení i elektronickou cestou. V žádném případě se ovšem **nejedná** o nahrazení stávajícího způsobu odesílání Vašich řešení! Spíše jde o jakýsi zkušební provoz a podle toho, jak dopadne, se zařídíme v časech budoucích.

Pokud se rozhodnete využít tohoto způsobu odevzdávání řešení, pak vězte, že úlohy se budou odevzdávat přes webové rozhraní na *http://ksp.mff.cuni.cz/submit/*. Zde se také dozvíte o konkrétních pravidlech pro odevzdávání Vašich řešení. Doporučujeme Vám si tato pravidla přečíst, pokud hodláte odevzdat svá řešení přes webové rozhraní, abyste nebyli na poslední chvíli zaskočeni.

Ať už pošlete řešení jakkoliv, zpět Vám přijdou normální poštou. Řešení, která přijdou elektronickou cestou, si totiž vytiskneme a opravíme tradičním způsobem. Ale ještě jednou bychom rádi zopakovali, že vypracovaná řešení můžete i nadále posílat normální poštou na naši nezměněnou adresu (najdete ji na konci zadání této série).

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

**Zadání třetí série šestnáctého ročníku KSP**

## 16-3-1 Fyzikova blecha — 10 bodů

Newtoon trénuje svoji blechu na bleší turnaj. Ten probíhá na svislé stěně, na které jsou vodorovné plošinky. Cílem blechy je slézt ze startovací polohy co nejdříve na podlahu. Pohyb blechy závisí na tom, zda je blecha na nějaké plošince, či padá. Pokud padá, klesne za jednu blechovteřinu o jeden blechometr. Pokud je na plošince, posune se za jednu blechovteřinu o jeden blechometr vlevo či vpravo.

Závod tedy probíhá tak, že blecha padá, padá, až dopadne na plošinku. Pak se rozhodne (nebo jí její majitel přikáže), zda půjde doleva nebo doprava, a jde, dokud nedojde na konec plošinky. Z ní pak seskočí a zase padá, dokud se nedostane na podlahu. Vítězí blecha, která přistane na podlaze jako první. Ovšem je nutné, aby žádná blecha nespadla z větší výšky než $v$ blechometrů, jinak se totiž po dopadu urazí a odmítne pokračovat v závodě.

Newtoon již vytrénoval svou blechu tak, že ho poslouchá na slovo. Problém je ten, že sám neví, jak blechu navigovat, aby prošla bludištěm nejkratší možnou cestou. Protože Vás ale zajímá, jak vypadá blecha, která poslouchá, rozhodli jste se Newtoonovi pomoci.

Na vstupu dostanete jednak počáteční souřadnice blechy (v celých blechometrech), $v$, což je největší výška, ze které může blecha spadnout, aby se neurazila, a $N$, což je počet plošinek na stěně. Dále dostanete popis $N$ plošinek, u každé plošinky souřadnice jejího horního dolního rohu a její šířku (vše opět v celých blechometrech). Všechny plošinky jsou vysoké jeden blechometr a žádné dvě se nedotýkají. Blecha je na podlaze, pokud se nachází na souřadnicích $[x; 0]$, kde $x$ je libovolné celé číslo.

Výstupem Vašeho programu je nejmenší počet blechovteřin, které bude blecha potřebovat, aby se dostala na podlahu. Kromě tohoto počtu vypište i počet plošinek, na které blecha dopadne, a u každé plošinky (v pořadí, jak na ně blecha dopadá) rozhodněte, zda má blecha jít vlevo či vpravo. Pokud úloha nemá řešení (moc malé $v$), vypište odpovídající zprávu.

*Příklad:* Blecha se nachází na souřadnicích $[5; 12]$, $v = 4$, $N = 3$. Plošinky jsou $[3; 8]; 5$, $[3; 4]; 5$ a $[7; 6]; 3$ ([souřadnice levého horního rohu]; délka). Nejkratší cesta trvá 17 blechovteřin, blecha navštíví všechny tři plošinky a půjde vpravo na první, vlevo na druhé a vpravo na třetí navštívené plošince.

## 16-3-2 Historikova past — 10 bodů

Váš přítel historik Vykopávka si na Vás přichystal, aby vylepšil Vaše (podle něj poněkud zanedbané) historické vzdělání, následující hru. Hraje se ve dvourozměrném bludišti $N \times M$ polí. V bludišti se (samozřejmě kromě zdí, ty se vyskytují v každém pořádném bludišti) nachází Theseus a Minotaurus. Abyste uspokojili svého přítele Vykopávku, musíte si tato jména nejprve zapamatovat. Theseus a Minotaurus. Theseus a Minotaurus…

Vy budete ovládat Thesea a budete se snažit dostat z bludiště ven, čili dostat se na hranici bludiště a následujícím krokem z bludiště utéct. Ovšem nikdy nesmíte narazit na Minotaura, sic bídně zhynete. Na Minotaura narazíte, pokud s ním sdílíte políčko.

Jeden tah probíhá následovně: nejprve se hýbe Minotaurus a táhne $k$-krát následujícím způsobem. Pokud nejsou postavy Minotaura a Thesea ve stejném sloupci, chce se Minotaurus pohnout o jedno políčko vlevo nebo vpravo, aby se Theseovi přiblížil. Pokud není na políčku, kam se Minotaurus chce posunout, zeď, skutečně se tam posune. Pokud nejsou obě postavy na stejném řádku bludiště, chce se Minotaurus pohnout nahoru či dolů opět směrem k Theseovi. Opět se na zvolené políčko Minotaurus přesune jen tehdy, není-li tam zeď. V jednom kroku provádí Minotaurus tyto pohyby v zadaném pořadí a může provést oba dva, čili se může dostat na jedno z okolních osmi políček.

Po $k$ takovýchto krocích Minotaura se hýbe Theseus, a to na jedno ze čtyř okolních volných polí. Takto se oba střídají na tahu, dokud buď Theseus neuteče z bludiště, nebo dokud Minotaurus nedohoní Thesea.

Vaším úkolem bude najít pro Thesea nejkratší posloupnost pohybů vlevo, vpravo, nahoru, dolů, aby se dostal bezpečně z bludiště, případně říci, že to není možné.

*Příklad:* Pro $k = 2$ a bludiště o rozměrech $8 \times 10$

```
XXXXXXXXXX
X.M......X
X........X
X........X
X...X....X    . - volné políčko
X.XXX....X    X - zeď
X....T....    M - Minotaurus
XXXXXXXXXX    T - Theseus
```

je nejkratší cesta z bludiště ←←→→→→→→→→.

## 16-3-3 Genetikova evoluce — 10 bodů

Když si Blátošlap bral ponožky ze svého prádelníku, zjistil, že se mu v něm vyvinula celá kolonie neznámých živočichů. Protože to byl genetik, jal se hned tyto živočichy zkoumat a zjistil, že i když se jedná o jediný druh, jeho zástupci jsou velmi rozmanití. Každý jedinec má totiž 30 znaků, které ho charakterizují a které se mohou měnit pouze mutací. Dalším výzkumem zjistil, že v prádelníku má $N$ různých poddruhů (poddruhy se navzájem liší alespoň v jednom znaku), i když jeden z nich převažuje.

Ihned se dovtípil, že v jeho prádelníku došlo k evoluci. A protože ho zajímá její průběh, byli jste požádáni o vyřešení tohoto problému.

Blátošlap Vám zadá $N$ a dále popis jednotlivých poddruhů. Každý poddruh je charakterizovaný číslem $0..2^{30}-1$, přičemž $i$-tý bit tohoto čísla vyjadřuje, zda je $i$-tý z 30 znaků, které poddruh charakterizují, přítomen. Pokud bychom si číslo poddruhu napsali ve dvojkové soustavě, $i$-tý bit tohoto čísla je $i$-tá cifra (počítáno zprava od nuly) v tomto zápisu. Jinak řečeno, pokud číslo $i$-krát vydělíme dvěma a spočteme zbytek po dělení tohoto čísla dvěma, výsledek je $i$-tý bit původního čísla.

Vaším úkolem je zjistit, jaký poddruh se vyvinul z jakého, a počet mutací, ke kterým muselo při tomto vývoji dojít. Předpokládejte, že vývoj probíhal tak, že každý poddruh kromě toho, který Vám Blátošlap zadal jako první (to je ten nejpočetnější), se vyvinul právě z jednoho jiného poddruhu. Navíc první zadaný poddruh je původním prapředkem všech poddruhů v prádelníku. Dále předpokládejte, že evoluce probíhala nejjednodušší možnou cestou, a tedy počet mutací v evoluci je nejmenší možný. Počet mutací je součet všech rozdílných znaků mezi každým poddruhem (kromě prvního zadaného) a jeho předkem.

Navíc žádný poddruh v Blátošlapově prádelníku nevymřel, všechny evolucí vzniklé poddruhy přežily až do dnešní doby.

*Příklad:* Pro $N = 4$ a poddruhy 0,3,7,12 je hledaná evoluce tato: poddruhy 3 a 12 se vyvinuly z poddruhu 0, poddruh 7 se vyvinul z poddruhu 3. Počet mutací, ke kterým muselo v evoluci dojít, je roven 5.

## 16-3-4 Ekonomova odměna — 10 bodů

Dlouhoprst byl velmi bohatý ekonom. Jednoho dne si umínil, že by zase mohl zvýšit své jmění. Ale jak? Vždyť všichni lidé už zjistili, že jeho finanční poradenství je spíše finančním proradenstvím. A tak se rozhodl dojít za ďáblem a upsat mu svou duši za pár stříbrňáků.

Po cestě do pekla potkal skupinku pirátů, kteří se handrkovali nad truhlou plnou stříbrňáků. Dlouhoprstovi to nedalo, dal se s nimi do řeči a zjistil, že se piráti domlouvají, jak si poklad rozdělí. Již se shodli na následujícím způsobu dělení.

Řekněme, že v truhle je $S$ stříbrňáků a pirátů je $P$. Piráti se očíslují, čili každý dostane právě jedno číslo od 1 do $P$. Pirát s pořadovým číslem $P$ navrhne způsob rozdělení všech stříbrňáků mezi piráty (každému pirátovi včetně sebe přidělí nezáporný počet stříbrňáků). Ostatní o tomto rozdělení hlasují a pokud je jich alespoň polovina pro, návrh je přijat. V opačném případě je pirát $P$ zabit a navrhuje pirát s číslem $P-1$.

Při hlasování se piráti řídí těmito pravidly: nejdůležitější je přežít. Pokud pirát ví, že v budoucnu určitě zemře, hlasuje vždy pro, nezávisle na výši svého podílu. Druhé pravidlo je zisk. Pokud má pirát v budoucnu šanci, že získá víc, než kolik je mu právě nabídnuto, hlasuje proti (ovšem jen pokud ví, že přežije). Třetí pravidlo je touha popravit co nejvíc pirátů, čili pokud pirát ví, že v budoucnu bude moci určitě získat stejný obnos, jaký je mu nabídnut, hlasuje proti. Důležité je, že se všichni piráti řídí stejnými pravidly a navzájem to o sobě vědí.

Dlouhoprst vycítil příležitost, a protože piráti se doteď nedohodli na tom, jak se očíslují, nabídl jim své finanční poradenství. Očísluje je, ale za odměnu se bude moci do rozdělování zapojit, a to s pořadovým číslem $P+1$. A piráti souhlasili. Dlouhoprst ovšem zjistil, že je to i nad jeho síly, a poprosil Vás o pomoc.

Váš program dostane na vstupu čísla $S$ (počet stříbrňáků v pokladu) a $P$ (počet pirátů). Musíte zjistit, zda může Dlouhoprst vůbec přežít a pokud ano, poraďte mu návrh rozdělení stříbrňáků mezi $P$ pirátů takový, aby byl jeho zisk (počet stříbrňáků, které nemusí mezi piráty rozdělit) co největší.

*Příklad:* Pro $S = 100$ a $P = 2$ Dlouhoprst přežít může a jeho zisk je 100 stříbrňáků. Jeho návrh je dát oběma pirátům nula stříbrňáků. Pirát číslo 2 pro jeho návrh hlasuje, protože jinak (kdyby byl Dlouhoprst popraven) by pirát 1 hlasoval proti libovolnému jeho návrhu a on sám by byl popraven.

## 16-3-5 Botanikova setba — 10 bodů

Známý botanik Konopník se rozhodl, samozřejmě z čistě vědeckých důvodů, zasít svojí nejoblíbenější rostlinu. Bohužel je pronásledován davy laiků, kteří by se rádi podíleli na sklizni, a tak je nucen zasít na poli, kde už je zaseta kukuřice, aby jeho úroda nebyla lidem na očích.

Protože ale Konopník chce, aby úroda byla co největší, rozdělil kukuřičné pole na pravidelnou síť oblastí a u každé oblasti zjistil její předpokládanou úrodnost. Ta může být i záporná, pokud daná oblast brání v pěstování na okolních oblastech. Konopník by chtěl osít jediné pravoúhlé území takové, aby součet úrodností na tomto území byl největší možný. Ovšem protože je biolog, obrátil se s tímto problémem na Vás.

Na vstupu dostanete $N$ a $M$, čili šířku a délku kukuřičného pole. Dále dostanete matici $N \times M$, jejíž hodnoty jsou úrodnosti jednotlivých oblastí kukuřičného pole. Vaším úkolem je najít podmatici (souvislou pravoúhlou oblast) takovou, že součet jejích prvků je největší možný. Pokud je takových podmatic víc, najděte takovou, která má nejmenší obsah.

*Příklad:* Pro pole o rozměrech $4 \times 5$ a úrodnostech

$$\begin{pmatrix} 1 & -1 & 4 & 5 & 3 \\ -2 & 3 & -2 & -5 & 8 \\ 2 & -4 & 2 & 1 & -7 \\ 4 & -3 & 2 & 1 & 1 \end{pmatrix}$$

má hledaná podmatice levý roh v prvním řádku a druhém sloupci, pravý dolní roh v druhém řádku a pátém sloupci.

## 16-3-6 Matematikova sonda — 10 bodů

Matematici jsou zvláštní lidé a jako takovým se jim občas v hlavě zhmotní zvláštní nápady. Jako třeba vyslat sondu na Mars.

K tomuto problému je samozřejmě potřeba přistupovat systematicky. Náš matematik si nejprve studiem odborné literatury a předchozích publikovaných prací v tomto oboru zjistil, že existují dva postupy.



```
        guess = 0;                                  /* Půlení intervalů pro zbylé možnosti */
        for (bit=1; bit<nrest; bit += bit) {
            qlen = 0;
            for (c=0; c<nrest; c++)
                if (c & bit)
                    query[qlen++] = rest[c];
            if (ask_dragon ())
                guess |= bit;
        }
        guess = rest[guess];                        /* Jdeme na jisto */
    }
    printf ("Ha, Tve cislo je %d!\n", guess+1);
    return 0;
}
```

```
function Delka(Obal:PSloup):real;
var Vystup:real;
begin
  Vystup:=0;
  while Obal^.Dalsi<>nil do begin
    Vystup:=Vystup+sqrt(sqr(Obal^.X-Obal^.Dalsi^.X)+sqr(Obal^.Y-Obal^.Dalsi^.Y));
    Obal:=Obal^.Dalsi;
  end;
  Delka:=Vystup;
end;

begin
  Sloupy:=Nacti;
  MergeSort(Sloupy);
  PridejKopii(Sloupy);
  Sloupy:=KonvexniObal(Sloupy);
  writeln('Delka plotu je:',Delka(Sloupy):10:3);
end.
```

**Úloha 16-2-4 – Křížový král – program**

```
#include <stdio.h>

#define MAX 1048576                          /* Do kolika umí draci počítat */
int query[MAX];                              /* Na co se ptáme (my počítáme od 0, drak od 1) */
int qlen;

int ask_dragon (void)                        /* Zeptáme se draka, copak nám řekne? */
{
        int i;
        for (i=0; i<qlen; i++)
                printf ("%d ", query[i]+1);
        printf ("?");
        scanf ("%d", &i);
        return i;
}
int main (void)
{
        int N;                               /* Hledané číslo je mezi 0 až N-1 */
        int guess;                           /* Jaké jsme zatím uhádli */
        int bit;                             /* Který bit zkoumáme */
        int c;                               /* Číslo, které zrovna zkoumáme */

        printf ("Velevazeny draku, rci N! ");
        scanf ("%d", &N);

        guess = 0;                           /* 1. fáze: Půlení intervalů na 0..N-1 */
        for (bit=1; bit<N; bit += bit) {
                qlen = 0;
                for (c=0; c<N; c++)
                        if (c & bit)
                                query[qlen++] = c;
                if (ask_dragon ())
                        guess |= bit;
        }

        query[0] = guess;                    /* Kontrola (pro N=1 ji nemusíme dělat) */
        qlen = 1;
        if (N != 1 && !ask_dragon ()) {      /* Někdy lhal, takže ve 2. fázi najdeme, kdy */
                int rest[8*sizeof (int)+1];  /* Jaké jsou možnosti? */
                int nrest;

                nrest = 0;                   /* Špatně byla kontrola */
                rest[nrest++] = guess;
                for (bit=1; bit<N; bit += bit)   /* Nebo jednotlivé bity */
                        rest[nrest++] = guess ^ bit;
```



- Může si za v zásadě nepatrnou částku (kterou si označíme jako jednu jednotku) nechat sondu vyrobit a vyslat. Je ovšem možné, že se něco nezdaří a pokus bude potřeba opakovat.
- Může si ji vlastnoručně sestrojit za $k > 1$ jednotek. Samozřejmě si je naprosto jistý, že v něčem tak triviálním by přece on nemohl udělat chybu.

Chtěl by samozřejmě zvolit levnější variantu, nicméně která to je, záleží na tom, kolik koupených sond se rozbije. Proto vymyslel následující algoritmus:

Prvních $k-1$ sond si koupí. Pokud žádná z nich neuspěje, sondu si sám vyrobí.

Tento algoritmus mu zajišťuje, že nikdy nezaplatí víc než $(2-1/k)$ krát tolik, kolik by musel: Nechť uspěje $n$-tá sonda. Pak buď $n \leq k-1$, pak zaplatí $n$ jednotek, což je optimální. Nebo $n \geq k$, pak zaplatí $2k-1$ jednotek, zatímco optimální řešení by bylo vyrobit si sondu rovnou a zaplatit pouze $k$ jednotek.

A také si okamžitě dokázal, že tento postup je nejlepší možný. Zjevně jediná možnost, jak si volit strategii, je nějakým způsobem určit počet sond $d$, po jejichž zničení si sondu vyrobí. Nechť uspěje $(d+1)$-ní sonda, pak zaplatí $k+d$ jednotek. Pokud by $d$ bylo větší nebo rovno $k$, optimum je $k$, tedy by zaplatil alespoň dvakrát tolik, kolik musel. Naopak kdyby $d$ bylo menší než $k-1$, optimum je $d+1$ a $\frac{k+d}{d+1} \geq \frac{(d+2)+d}{d+1} \geq 2$.

Po chvíli si však uvědomil, že by možná mohl ušetřit alespoň v průměrném případě, kdyby jednu z mincí našetřených na pořízení sondy použil jako generátor náhodných čísel. Nicméně protože je to starý zkušený profesor, přenechal vám vyřešení této úložky jako jednoduché domácí cvičení.

Vaším úkolem je tedy nalézt pravděpodobnostní strategii, pro kterou by existovala nějaká konstanta $c$ (pokud možno co nejmenší) taková, že ať už se porouchá libovolný počet sond, profesor zaplatí v průměru nanejvýš $c$-krát tolik, než kolik musí.

**Recepty z programátorské kuchařky**

V dnešním, v pořadí již třetím, dílu kuchařky popíšeme problém minimální kostry a ukážeme si, jak ho řešit. Také popíšeme datovou strukturu *Disjoint-Find-Union* (její název je často zkracován na DFU), kterou šikovně použijeme právě na hledání minimální kostry grafu.

Co je to *graf* jsme si vysvětlili již dříve. *Podgraf* nějakého grafu vznikne z původního grafu odebráním libovolných vrcholů a libovolných hran. *Cesta* v grafu je posloupnost vrcholů taková, že každý vrchol je s následujícím spojen hranou a vrcholy se na cestě neopakují (často za cestu považujeme i tyto hrany). *Kružnice* v grafu je podobná posloupnost vrcholů, pouze první vrchol je shodný s posledním a nikde jinde se opět vrcholy neopakují. (Pro úplnost: *tah* bychom tomu říkali tehdy, kdyby se vrcholy mohly opakovat, ale hrany nikoliv, a *sled* tehdy, pokud by se mohlo opakovat cokoliv.)

Pokud v grafu mezi každými dvěma vrcholy existuje cesta (což je totéž, jako když mezi nimi existuje sled, rozmyslete si, proč), říkáme takovému grafu *souvislý*. Pokud graf souvislý není, můžeme ho rozdělit na podgrafy, které již souvislé jsou a nevedou mezi nimi žádné hrany. Takovým podgrafům se pak říká *komponenty souvislosti*. U souvislého grafu za komponentu souvislosti považujeme celý graf.

*Strom* je graf, který je souvislý a zároveň *acyklický* (čili neexistuje v něm žádná kružnice). *List* stromu je takový vrchol, ze kterého vede jen jedna hrana. Každý konečný strom o dvou či více vrcholech má vždy alespoň dva listy. Můžeme je nalézt třeba tak, že sestrojíme ve stromě nejdelší cestu (rozmyslete si, proč bude mít tato cesta konečnou délku) a všimneme si, že koncové vrcholy této cesty jsou hledané listy (kdyby z koncových vrcholů cesty vedla více než jedna hrana, pak bychom ji buďto o ni buďto mohli cestu prodloužit nebo by tato hrana vedla do vrcholu, který na cestě již leží, jenže tím by tvořila kružnici).

Zajímavé je to, že strom s $N$ vrcholy má právě $N-1$ hran. To můžeme dokázat indukcí podle počtu vrcholů stromu: Strom s jedním vrcholem neobsahuje hranu žádnou. Pokud máme strom s $N > 1$ vrcholy, víme, že má alespoň dva listy. Vezmeme tedy libovolný list a ze stromu ho odebereme. Tím získáme opět strom (souvislost jsme porušit nemohli a kružnici jsme také nevytvořili) a jeho počet vrcholů je o 1 menší, čili podle indukčního předpokladu má o 1 méně hran než vrcholů. Nyní list „přilepíme“ zpět, čímž zvýšíme počet vrcholů i hran o 1 a tvrzení stále platí.

A nyní k slibovaným kostrám. *Kostra* grafu budeme říkat podgrafu, který obsahuje všechny vrcholy a nejmenší počet hran takový, aby cesta mezi dvěma různými vrcholy existovala v kostře grafu právě tehdy, existuje-li v grafu původním. Všimněte si, že kostra souvislého grafu je sama souvislá a navíc neobsahuje žádnou kružnici (jinak bychom mohli libovolnou hranu ležící na kružnici z kostry beze škody odebrat, čímž bychom získali menší kostru, a to nám definice zakazuje.) Čili každá kostra souvislého grafu je strom a jelikož všechny stromy na $N$ vrcholech mají $N-1$ hran, všechny kostry také. Pokud je graf nesouvislý, stačí tuto úvahu provést pro každou komponentu zvlášť a zjistíme, že kostra je *les* (to je graf, jehož každá komponenta je strom) s $N-k$ hranami, kde $k$ je počet komponent souvislosti.

Pokud každé hraně grafu přiřadíme nějaké *ohodnocení,* což bude nějaké číslo (pro naše potřeby vždy kladné), dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* a to je taková, pro kterou je součet ohodnocení jejích hran nejmenší možný.

**Algoritmus hledání minimální kostry**

Náš algoritmus na hledání minimální kostry (tzv. hladový algoritmus) bude velmi jednoduchý. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, ale toto mlčky předpokládáme). To, že nalezená kostra je minimální, si ukážeme později. Nyní se podívejme na časovou složitost našeho algoritmu: Pokud vstupní graf má $N$ vrcholů a $M$ hran, pak úvodní setřídění hran vyžaduje čas $O(M \log M)$ (použijeme některý z rychlých třídících algoritmů popsaných v minulém díle) a poté se pokusíme přidat každou z $M$ hran. V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude $M$ testů toho, zda mezi dvěma vrcholy vede hrana, trvat méně než $O(M \log M)$. Celková časová složitost našeho algorit-

mu je tedy $O(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $O(M)$.

**Důkaz správnosti hladového algoritmu**

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou vesměs různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídícím algoritmem zůstalo zachováno. Tím se kostra nalezená naším algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme $T_{\text{alg}}$ kostru nalezenou naším algoritmem a $T_{\text{min}}$ minimální kostru. Nechť $e_1, \ldots, e_{N-1}$ jsou hrany kostry $T_{\text{alg}}$ v pořadí, v jakém byly přidány naším algoritmem do kostry. Pokud jsou kostry $T_{\text{alg}}$ a $T_{\text{min}}$ různé, pak existuje hrana $e_i$, která není obsažena v kostře $T_{\text{alg}}$. Zvolme nejmenší takové $i$, tzn., že kostra $T_{\text{min}}$ obsahuje všechny hrany $e_1, \ldots, e_{i-1}$. Váha každé hrany kostry $T_{\text{min}}$ kromě $i-1$ hran $e_1, \ldots, e_{i-1}$ je ostře větší než váha hrany $e_i$: Kdyby tomu tak nebylo, pak by kostra $T_{\text{min}}$ obsahovala hranu $f$ s menší vahou než $e_i$ a náš algoritmus by hranu $f$ přidal k hranám $e_1, \ldots, e_{i-1}$ při vytváření kostry $T_{\text{alg}}$. Hrana $f$ pak by musela být mezi hranami $e_1, \ldots, e_{i-1}$, protože její váha je menší než váha hrany $e_i$, ale tomu tak není. Tedy taková hrana $f$ neexistuje.

Přidejme nyní hranu $e_i$ ke kostře $T_{\text{min}}$. Takto vzniklý podgraf vstupního grafu zjevně obsahuje kružnici (už před přidáním hrany $e_i$ existovala v kostře $T_{\text{min}}$ cesta mezi koncovými vrcholy hrany $e_i$, protože kostra $T_{\text{min}}$ je souvislá). Tuto kružnici si označme $C$. Protože $T_{\text{alg}}$ neobsahuje žádnou kružnici, kružnice $C$ obsahuje alespoň jednu hranu $e'$, která není v kostře $T_{\text{alg}}$. Protože hrana $e'$ není žádnou z hran $e_1, \ldots, e_{i-1}$, je její váha ostře větší než váha hrany $e_i$. Odstraňme nyní hranu $e'$. Označme $T'$ výsledný podgraf vstupního grafu, tj. graf získaný z kostry $T_{\text{min}}$ záměnou hrany $e'$ za hranu $e_i$. Protože $e'$ a $e_i$ ležely ve společné kružnici $C$, je $T'$ souvislý podgraf, a tedy kostra vstupního grafu. Součet vah hran kostry $T'$ je však ostře menší než součet vah hran kostry $T_{\text{min}}$, což není možné, neboť $T_{\text{min}}$ je minimální kostra vstupního grafu. Tedy neexistuje žádné $i$ takové, že hrana $e_i$ není obsažena v kostře $T_{\text{min}}$ a kostry $T_{\text{alg}}$ a $T_{\text{min}}$ jsou shodné.

**Disjoint-Find-Union**

Datová struktura DFU slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do jedné podmnožiny rozkladu, pokud mezi nimi již ve vytvářené kostře existuje cesta, čili podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find**: Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v našem případě odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti. Tato operace bývá často implementována tak, že vrací nějakého pevně zvoleného reprezentanta podmnožiny. Test, zda jsou dva prvky ve stejné podmnožině se pak provede porovnáním příslušných reprezentantů. V této podobě by tato operace odpovídala níže popsané funkci `root`.
- **union**: Sloučení dvou podmnožin do jedné. Tato operace bude v naší aplikaci odpovídat přidání hrany mezi dvě různé komponenty grafu.

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele, trochu nezvykle, od listů ke kořenu. Operaci `find` lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci `union` provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme si ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do $N$. Ukazatele ve stromě si pak pamatujeme v poli `parent`, kde 0 znamená, že prvek nemá rodiče, tj. je kořenem svého stromu. Funkce `root(v)` vrací kořen stromu, který obsahuje prvek $v$.

```
var parent:array[1..N] of integer;

procedure init;
  var i:integer;
  begin
    for i:=1 to N do parent[i]:=0;
  end;

function root(v: integer):integer;
  begin
    if parent[v]=0 then root:=v
    else root:=root(parent[v]);
  end;

function find(v,w:integer):boolean;
  begin
    find:=(root(v)=root(w));
  end;

procedure union(v,w:integer);
  begin
    v:=root(v);
    w:=root(w);
    if v<>w then parent[v]:=w;
  end;
```

S právě předvedenou implementací operací `find` a `union` by se nám mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“ a pokud budou obsahovat $N$ prvků, nalezení kořene zabere čas $O(N)$. Tedy operace `find` a `union` vyžadují čas až $O(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank**: Každý prvek má přiřazen `rank`. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace `union`, připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.



```
    KterySeznam:=not(KterySeznam);
    Co:=DalsiPrvek;
  end;
  MergeSort(Seznam[false]);
  MergeSort(Seznam[true]);
  PosledniPrvek:=nil;
  while (Seznam[false]<>nil) and (Seznam[true]<>nil) do begin
    KterySeznam:=Seznam[true]^.X<Seznam[false]^.X;
    if abs(Seznam[true]^.X-Seznam[false]^.X)<Presnost then
      KterySeznam:=Seznam[true]^.Y>Seznam[false]^.Y;      {X stejné, rozhoduje Y}
    DalsiPrvek:=Seznam[KterySeznam];
    Seznam[KterySeznam]:=Seznam[KterySeznam]^.Dalsi;
    if PosledniPrvek=nil then
      Co:=DalsiPrvek                                     {Výstup je zatím prázdný}
    else
      PosledniPrvek^.Dalsi:=DalsiPrvek;                  {Přidáme na konec}
    DalsiPrvek^.Dalsi:=nil;                              {Opravíme konec}
    PosledniPrvek:=DalsiPrvek;
  end;
  PosledniPrvek^.Dalsi:=Seznam[Seznam[false]=nil];
end;                                                     {...a je to setříděné}

procedure PridejKopii(Sloupy:PSloup);
var Kopie,Nova:PSloup;
begin
  Kopie:=nil;
  while Sloupy^.Dalsi<>nil do begin
    new(Nova);
    Nova^:=Sloupy^;
    Nova^.Dalsi:=Kopie;
    Kopie:=Nova;
    Sloupy:=Sloupy^.Dalsi;
  end;
  Sloupy^.Dalsi:=Kopie;                                  {...a přidáme kopii na konec}
end;

function KonvexniObal(ZCeho:PSloup):PSloup;
var Obal,DalsiSloup:PSloup;
   Vektor1,Vektor2:record X,Y:real; end;
begin
  DalsiSloup:=ZCeho;
  ZCeho:=ZCeho^.Dalsi;
  DalsiSloup^.Dalsi:=nil;
  Obal:=ZCeho;
  ZCeho:=ZCeho^.Dalsi;
  Obal^.Dalsi:=DalsiSloup;                               {Vložíme první 2 body do posloupnosti}
  while ZCeho<>nil do begin
    DalsiSloup:=ZCeho;
    ZCeho:=ZCeho^.Dalsi;
    DalsiSloup^.Dalsi:=Obal;
    Obal:=DalsiSloup;
    while Obal^.Dalsi^.Dalsi<>nil do begin               {Máme alespoň 3 body?}
      Vektor1.X:=Obal^.Dalsi^.X-Obal^.Dalsi^.Dalsi^.X;
      Vektor1.Y:=Obal^.Dalsi^.Y-Obal^.Dalsi^.Dalsi^.Y;
      Vektor2.X:=Obal^.X-Obal^.Dalsi^.X;                 {Vektor2 určuje směr predposl. a posl. sloupu}
      Vektor2.Y:=Obal^.Y-Obal^.Dalsi^.Y;                 {Vektor1 je totéž vzhledem k předchozím dvěma}

      if (Vektor1.X*Vektor2.Y-Vektor1.Y*Vektor2.X)>Presnost then break;
      DalsiSloup:=Obal^.Dalsi;                           {Bod je třeba odebrat}
      Obal^.Dalsi:=Obal^.Dalsi^.Dalsi;
      dispose(DalsiSloup);
    end;
  end;
  KonvexniObal:=Obal;
end;
```

```
  nejmensi := 0;                                  {Jdeme zkoušet hody}
  while l > 0 do begin
    P := (P−C+C*k div l) div 2;                   {Přepočteme P(k,l) na P(k−1,l−1)}
    Write('Hod vejce z patra ',nejmensi+P−1,'. Rozbilo se?');
    Read(rozbilo);
    if rozbilo = 1 then begin
      C := C*k div l;                             {Prepočteme (l nad k) na (l−1 nad k−1)}
      Dec(k);                                     {P už je správně přepočteno...}
    end
    else begin
      nejmensi := nejmensi + P;
      C := C*(l−k) div l;                         {Přepočteme (l nad k) na (l−1 nad k)}
      P := P + C;                                 {Přepočteme P(k−1,l−1) na P(k,l−1)}
    end;
    Dec(l);
  end;
  if nejmensi = n then
    WriteLn('Vejce se nerozbije z zadneho patra.')
  else
    WriteLn('Vejce se rozbije z patra ',nejmensi,'.');
end.
```

**Úloha 16-2-3 – Král Potvorník – program**

```
program KralPotvornik;
const Presnost = 1E−10;                           {U reálných čísel musíme počítat s chybou}
type PSloup = ^TSloup;
     TSloup = record
              X,Y:real;
              Dalsi:PSloup;
              end;
var Sloupy:PSloup;

function Nacti:PSloup;
var NovySloup:PSloup;
    Vystup:PSloup;
    PocetSloupu:longint;
begin
   Vystup:=nil;
   Readln(PocetSloupu);
   if PocetSloupu<3 then begin
     writeln('A z tohohle chces vytvorit konvexni obal?!');
     halt;
   end;
   while PocetSloupu>0 do begin
     new(NovySloup);
     readln(NovySloup^.X,NovySloup^.Y);
     NovySloup^.Dalsi:=Vystup;                    {Přidáme do seznamu sloupů}
     Vystup:=NovySloup;
     dec(PocetSloupu);
   end;
   Nacti:=Vystup;
end;

procedure MergeSort(var Co:PSloup);
var Seznam:array[boolean] of PSloup;
    DalsiPrvek,PosledniPrvek:PSloup;
    KterySeznam:boolean;
begin
   if Co^.Dalsi=nil then exit;                    {Není co řešit}
   KterySeznam:=false;
   Seznam[false]:=nil;Seznam[true]:=nil;
   while Co<>nil do begin
     DalsiPrvek:=Co^.Dalsi;
     Co^.Dalsi:=Seznam[KterySeznam];
     Seznam[KterySeznam]:=Co;
```



- **path compression**: Ve funkci `root(v)` přepojíme všechny prvky na cestě od prvku $v$ ke kořeni, rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní funkce `root` a `union`:

```
var parent:array[1..N] of integer;
    rank:array[1..N] of integer;

procedure init;
  var i:integer;
  begin
    for i:=1 to N do
      begin
        parent[i]:=0;
        rank[i]:=0;
      end;
  end;

{zmena kvuli path compression}
function root(v: integer):integer;
  begin
    if parent[v]=0 then root:=v
    else begin
      parent[v]=root(parent[v]);
      root:=parent[v];
    end;
  end;

{stejna jako minule}
function find(v,w:integer):boolean;
  begin
    find:=(root(v)=root(w));
  end;

{zmena kvuli union by rank}
procedure union(v,w:integer);
  begin
    v:=root(v);
    w:=root(w);
    if v=w then exit;
    if rank[v]=rank[w] then
      begin
        parent[v]:=w;
        rank[w]:=rank[w]+1;
      end
    else
      if rank[v]<rank[w] then
        parent[v]:=w
      else
        parent[w]:=v;
  end;
```

Zaměřme se nyní blíže na metodu „union by rank“. Nejprve učiníme následující pozorování: Pokud je prvek $v$ s rankem $r$ kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň $2^r$ prvků. Naše pozorování dokážeme indukcí podle $r$. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku $v$ mění z $r-1$ na $r$, slučujeme dva stromy, jejichž kořeny mají rank $r-1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň $2^{r-1}$ prvků a tedy výsledný strom má alespoň $2^r$ prvků, jak jsme požadovali. Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem $r$ je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen „union by rank“, je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Tím pádem ale nalezení reprezentanta (čili kořene) libovolné skupiny trvá $O(\log N)$ a tedy operace `find` a `union` budou vyžadovat čas $O(\log N)$.

**Amortizovaná časová složitost**

Abychom mohli pokračovat dále, musíme si vysvětlit, co znamená *amortizovaná* časová složitost. Pokud řekneme, že nějaká operace vyžaduje amortizovaně čas $O(t)$, pak provedení $k$ takových operací vyžaduje čas nejvýše $O(kt)$, ale provedení jedné konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak ve výsledném odhadu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Předveďme si to na příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že přičíst jedničku k tomuto číslu $N$-krát nám zabere čas $O(N)$, pak můžeme říci, že přičíst k číslu jedničku trvá amortizovaně $O(1)$.

Jak tedy ukážeme, že $N$ přičtení jedničky k číslu zabere čas $O(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek a pokud jich na $N$ operací použijeme jen $O(N)$, uspějeme. Předpokládejme tedy, že každá jednička v dvojkovém zápisu daného čísla už má jeden svůj penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Každé jedničce, kterou chceme přičíst, dáme ze začátku dva penízky. Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na poslední bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu... dokud nenajde nulu. Takto zachová podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek.

Tedy $N$ přičítání jednotky nás stojí $2N$ penízků a tím pádem čas $O(N)$. Ačkoli jedno konkrétní přičtení jedničky k danému číslu může trvat déle než konstantní čas, amortizovaná složitost přičtení jedničky k číslu je konstantní.

**Dokončení analýzy DFU**

Pokud bychom prováděli pouze „path compression“ a nikoliv „union by rank“, dalo by se dokázat, že každá z operací `find` a `union` vyžaduje amortizovaně čas $O(\log N)$. To ale nebudeme dokazovat, protože tím bychom si nijak oproti samotnému „union by rank“ nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou současně dosáhneme mnohem lepšího amortizovaného času $O(\alpha(N))$ na jednu operaci `find` nebo `union`, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty $N$ nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

My si zde předvedeme poněkud horší, ale technicky jednodušší časový odhad $O((N+L)\log^* N)$, kde $L$ je počet provedených operací `find` nebo `union` a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme

funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla $N$ je nejmenší přirozené číslo $k$ takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo $N$ opakovaně logaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU s metodami „union by rank“ a „path compression“. Prvky si rozdělíme do skupin podle jejich ranku: $K$-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (K-1)) + 1$ a $2 \uparrow K$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = O(\log^* N)$ skupin. Odhadněme shora počet prvků v $K$-té skupině:

$$\frac{N}{2^{(2 \uparrow (K-1))+1}} + \cdots + \frac{N}{2^{2 \uparrow K}} = \frac{N}{2^{2 \uparrow (K-1)}} \cdot \left( \sum_{i=1}^{2 \uparrow K - 2 \uparrow (K-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2 \uparrow (K-1)}} \cdot 1 = \frac{N}{2 \uparrow K}.$$

Teď můžeme provést časovou analýzu funkce `root(v)`. Čas, který spotřebuje funkce `root(v)`, je úměrný délce cesty od prvku $v$ ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce `root(v)`, a ty, které zahrneme do faktoru $O(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce `root(v)` započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $O(\log^* n)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek $v$ v $K$-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku $v$ vzroste. Tedy po $2 \uparrow K$ přepojeních je rodič prvku $v$ v $(K+1)$-ní nebo vyšší skupině. Pokud $v$ je prvek v $K$-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce `root(v)` nejvýše $(2 \uparrow K)$-krát. Protože $K$-tá skupina obsahuje nejvýše $N/(2 \uparrow K)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše $N$. A protože počet skupin je nejvýše $O(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce `root(v)`, nejvýše $O(N \log^* N)$. Protože funkce `root(v)` je volána $2L$-krát, plyne časový odhad $O((N+L) \log^* N)$ z právě dokázaných tvrzení.

**Inverzní Ackermannova funkce** $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz $A_k^i$ zastupuje složení $i$ funkcí $A_k$, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce $A(k)$ je pak rovna hodnotě $A_k(2)$, čili $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd… Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pak nejmenší přirozené číslo $k$ takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

Dnešní menu vám servírovali
*Dan Kráľ, Martin Mareš a Milan Straka*

---

Svá řešení nám zasílejte do 15. března 2004 na adresu:

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

---

## Vzorová řešení druhé série šestnáctého ročníku KSP

### 16-2-1 Král Eeek

Jednociferný základ vyřešíme zvlášť: jestliže všechny cifry jsou menší než základ, máme jednu jedinou možnost, jinak nemáme žádnou.

Nechť $n$ je počet cifer na vstupu, $s[i]$ jsou vstupní cifry.

Řešení v čase $O(n^3)$ je jednoduché: V poli $p[i]$ si budeme udržovat počet možností, které by existovaly, kdyby na vstupu chybělo prvních $i$ čísel. Pro všechny délky základu $z$ provedeme následující výpočet:

- $p[n-z-1] = 1$, protože jednociferné číslo je určité menší než alespoň 2-ciferný základ (tohle platí samozřejmě pouze tehdy, nezačíná-li základ cifrou nula).
- Nechť máme všechna $p[i+1]$, $p[i+2]$, … a chceme spočítat $p[i]$. Zjistíme, kolikaciferné číslo může být na této pozici (dle konkrétních cifer to může být $z$ nebo $z-1$, nebo pouze 1-ciferné, pokud ta cifra je 0; toto řeší funkce `kolik_cifer`), a počet možností pro $p[i]$ bude $p[i+1]$ (použijeme jednociferné číslo) + $p[i+2]$ (nebo dvouciferné) + $p[i+3]$ (nebo trojciferné) + … + $p[i+z-1]$ (a možná + $p[i+z]$).

Pokud základ nezačínal nulou (viz. dříve), můžeme k celkovému řešení přičíst $p[0]$. Na každé cifře provedeme $z$ kroků, cifer je $n-z$, všech základů je $n$, čili algoritmus pracuje v čase $O(n^3)$.

Časovou složitost lze vylepšit:

a) potřebujeme rychle zjišťovat, jestli na daném místě smí být $z$ nebo $z-1$ cifer
b) potřebujeme počítat součet $p[i+1] + p[i+2] + \ldots + p[i+z-1]$ nějak inteligentně (tj. v konstantním čase)

a) jde vyřešit vyhledávacím automatem, ale to je tak trochu kanón na vrabce. Lepší je jít s délkou základu postupně od dvojky nahoru a pamatovat si, jestli číslo bylo v minulém kroku dlouhé $z$ nebo $z-1$. Jestliže nejlevější cifra zkoumaného čísla je menší než nejlevější číslice základu, není co řešit a zkoumané číslo určitě může být plné délky. Jestliže nejlevější cifra je větší, také není co řešit – číslo musí být kratší. Jestliže se nejlevější cifry shodují, stačí se podívat, jak to dopadlo v minulém kroku; v tomto kroce dopadne porovnání stejně.
b) zavedeme pomocné pole $c[i]$, přičemž bude platit, že $c[i] = \sum_{j=i}^{n-z-1} p[j]$. Když budeme potřebovat zjistit součet $p[a] + \ldots + p[b]$, spočítáme ho jako $c[a] - c[b+1]$.

Tímto algoritmus urychlíme na $O(n^2)$, paměťová složitost zůstane $O(n)$.

*Pavel Šanda*

### 16-2-2 Král Ovopole

Většina řešitelů této úlohy nevýslovně zkoušela královu tr-



```
int main (void)
{
        int c[MAXN], p[MAXN];
        int i, z, result;

        gets (s);
        n = strlen (s);

        for (i=0; i<n-1; i++) {
                mensi[i] = 0;
        }

        result = 1;                                   /* Vyřešíme jednociferný základ */
        for (i=0; i<n-1; i++) {
                if (s[i] >= s[n-1])
                        result = 0;
                if (s[i] < s[n-1])
                        mensi[i] = 1;
        }
        for (z=2; z<n; z++) {                         /* A ještě víceciferné základy */
                c[n-z] = 1; c[n-z+1] = 0; p[n-z] = 1;

                for (i=n-z-1; i>=0; i--) {
                        int h = kolik_cifer (i, z);
                        p[i] = c[i+1] - c[i+h+1];
                        c[i] = c[i+1] + p[i];
                }
                if (s[n-z] != '0')                    /* Pokud základ nezačíná nulou, bereme ho */
                        result += p[0];
                for (i=0; i<n; i++)
                        mensi[i] = nove_mensi[i];
        }
        printf ("%d\n", result);
        return 0;
}
```

---

### Úloha 16-2-2 – Král Ovopole – program

```
program KingsField;
var
  n, k, l : Integer;                    {Počet pater; počet vajec; počet pokusů}
  P, C : Integer;                       {Spočítaný max. počet pater; Predpočítané komb. číslo}
  nejmensi : Integer;                   {Nejnižší patro, ze kterého se vejce může rozbít}
  rozbilo : Integer;                    {Rozbilo se hodem vejce?}
begin
  Write('Pocet pater:');
  Read(n);
  Write('Pocet vajec:');
  Read(k);
  l := 0;
  P := 1;
  C := 1;                               {v C bude uloženo (l nad k), pokud l>=k}

  while P < n+1 do begin                {Najdeme minimální nutný počet pokusů}
   if l < k then                        {Dokud je dost vajec, půlíme}
    P := 2*P
   else begin
    P := 2*P - C;
    C := C*(l+1) div (l+1-k);           {Přepočteme (l nad k) na (l+1 nad k)}
   end;
   Inc(l);
  end;
```

$y$ vyskytuje jako požadavek, nebo s pravděpodobností 3/4, pokud se tak nevyskytuje ani jedno z nich (3/4 je zřejmě větší než $1-p/2$). Snadno si rozmyslíme, že ve všech ostatních případech bude pravděpodobnost splnění požadavku také alespoň $1-p/2$. Dohromady tedy v průměru splníme alespoň $\min(p, 1-p/2)$ z požadavků. Toto číslo bude největší, pokud $p = 1-p/2$, čili když $p = 2/3$. V tomto případě jsme schopni splnit průměrně alespoň 2/3 požadavků.

Navíc se králi doneslo, že v sousedním království, kde měli podobný problém, si královští synové dokázali naklást takové požadavky, že z nich opravdu více než 2/3 splnit nešly. Tajná služba uvedla, že tyto požadavky byly natolik složitě formulované, že je není vhodné králi detailně prezentovat, a on se s tím spokojil – tedy i vy budete muset.

Zbývá vyřešit to, že ani k tomuto řešení král nehodlá propůjčit svou korunu. Chtěli bychom se tedy obejít bez generátoru náhodných čísel.

Nejprve si rozmysleme, jak přesně spočítáme, kolik požadavků v průměru splníme: Označme si $p_x$ pravděpodobnost, že $x$ bude `true`. Pak pravděpodobnost, že splníme požadavek

- $(x)$ je $p_x$.
- $(x$ `or` $y)$ je $1 - p_x p_y$.
- $(x$ `or not` $y)$ je $1 - p_x(1-p_y)$.
- (`not` $x$ `or` $y)$ je $1 - (1-p_x)p_y$.
- (`not` $x$ `or not` $y)$ je $1 - (1-p_x)(1-p_y)$.

Když všechny tyto pravděpodobnosti sečteme, dostaneme hledaný průměrný počet splněných požadavků, označme si ho $P$.

Nyní řekněme, že bychom si napevno zvolili $x$ jako `true` (tedy položili $p_x = 1$) a ostatní proměnné volili stále náhodně se stejnými pravděpodobnostmi. Stejným postupem si spočítáme průměrný počet splněných požadavků v tomto případě a označíme si ho $P_{\texttt{true}}$. Analogicky, pokud si napevno zvolíme $x$ jako `false`, v průměru splníme $P_{\texttt{false}}$. Nyní nahlédneme, že $P = p_x \cdot P_{\texttt{true}} + (1-p_x) \cdot P_{\texttt{false}}$. Řekněme, že volbu, které provincie dostane který syn, provádíme postupně počínajíce provincií $x$. S pravděpodobností $p_x$ se rozhodneme dát provincii $x$ Petrovi, v tom případě splníme průměrně $P_{\texttt{true}}$ požadavků. S pravděpodobností $(1-p_x)$ ji naopak dáme Pavlovi a splníme průměrně $P_{\texttt{false}}$ požadavků. Dohromady tedy průměrně splníme $p_x \cdot P_{\texttt{true}} + (1-p_x) \cdot P_{\texttt{false}}$ požadavků, což musí být rovno $P$.

$P$ je tedy vážený průměr $P_{\texttt{true}}$ a $P_{\texttt{false}}$, jedno z $P_{\texttt{true}}$ a $P_{\texttt{false}}$ tedy musí být alespoň tak velké jako $P$, protože kdyby byly obě menší než $P$, i jejich průměr by musel být menší než $P$. To nám říká, komu přidělit provincii $x$ – pokud je $P_{\texttt{true}} \geq P$, dáme ji Petrovi, jinak Pavlovi. Nyní si za $x$ do všech požadavků dosadíme zvolenou hodnotu a tento postup opakujeme s takto modifikovanými požadavky pro další proměnnou.

Algoritmus bude tedy pracovat takto: Na začátku nastavíme $p_x$ pro všechny proměnné podle popsaného postupu (buď 1/2 nebo 2/3). Poté spočteme $P$ (součet pravděpodobností splnění všech požadavků). Nyní vybereme jednu proměnnou $x$, které jsme ještě nedali žádnou hodnotu, a spočteme $P_{\texttt{true}}$ a $P_{\texttt{false}}$. Tyto hodnoty se počítají jako $P$, jen jednou použijeme $p_x =$ `true` a jednou $p_x =$ `false`. Víme, že buď $P_{\texttt{true}}$ nebo $P_{\texttt{false}}$ je větší než $P$. Pokud tedy $P_{\texttt{true}} > P_x$, dáme proměnné $x$ hodnotu `true` a $p_x = 1$. Jinak položme $x =$ `false` a $p_x = 0$. Se změněnou hodnotou $p_x$ přepočítáme $P$, vybereme jinou proměnnou, které jsme ještě nedali hodnotu, a tu zpracujeme stejným způsobem. Algoritmus skončí, pokud jsme již všem proměnným přidělili nějaké hodnoty, čili jsme rozdělili všechny provincie.

V celém tomto postupu si nikde nepotřebujeme volit žádná náhodná čísla (pouze provádíme výpočty s pravděpodobnostmi), čímž jsme se vyhnuli nutnosti otloukat královskou korunu.

Časová složitost přímočaré implementace tohoto algoritmu je $O(N^2)$, kde $N$ je počet požadavků, protože spočtení $P$ nám trvá $O(N)$ a počítáme ho třikrát pro každou proměnnou (kterých je nanejvýš dvakrát tolik, co požadavků). Ovšem pokud si všimneme toho, že při změně pouze jedné $p_x$ se $P$ „moc“ nezmění, a trochu se zamyslíme, můžeme časovou složitost snížit na $O(N)$. Paměťovou složitost dokážeme také, při troše snahy, udržet na $O(N)$.

*Zdeněk Dvořák a Král Dan*

**Úloha 16-2-1 – Král Eeek – program**

```
#include <stdio.h>
#define MAXN 10240
#define min (a, b) ( ( (a) < (b)) ? (a) : (b))

char s[MAXN];
int mensi[MAXN], nove_mensi[MAXN], n;

int kolik_cifer (int i, int z)
{
        int m = -1, res;

        if (s[i] < s[n-z])
                m = 1;
        else if (s[i] == s[n-z]) m = mensi[i+1];
                else m = 0;

        nove_mensi[i] = m;
        if (s[i] == '0')                        /* připadá v úvahu jediná cifra - nula */
          return 1;
        res = min (z-1+m, n-z-i);               /* jinak máme nanejvýš res cifer */
        return res;
}
```



pělivost, když jejich řešení vyžadovala více pokusů, než bylo nezbytně nutné. Své počáteční rozhodnutí, že všichni švindlíři budou setnuti nebo přinejmenším vsazeni do věže musel Ovopole brzy poněkud pozměnit pro nedostatek popravčích mistrů a věží. I bombardování pukavci krále brzy přestalo bavit, a tak většina špatných řešitelů byla potrestána jen nízkým bodovým ohodnocením. Abychom ale nebyli jednostranní, je třeba uvést, že se našlo i několik pěkných řešení, z nichž jedno dokonce předčilo očekávání organizátorů. Na snad všechna špatná řešení fungoval následující protipříklad (proto ho uvádím zde a neopisoval jsem ho do každého špatného řešení): Mějme dvě vejce a 28 pater. Pro tuto konfiguraci je správné (viz algoritmus níže) hodit nejdříve vejce ze šestého patra (patra číslujeme od nuly), pokud se nerozbije, tak ze dvanáctého, pokud se nerozbije, tak ze sedmnáctého, pak z dvacátého prvního, dvacátého čtvrtého, dvacátého šestého a nakonec z dvacátého sedmého patra (předpokládáme, že v nějakém patře se vejce rozbít musí – diskuse tohoto předpokladu je též uvedena níže). Pokud se vejce při některém pokusu rozbije, tak prostě budeme procházet patra od posledního, kde se vejce nerozbilo, směrem vzhůru, až najdeme hledané patro. V nejhorších případech potřebuje tento algoritmus pouze 7 pokusů na nalezení hledaného patra.

A nyní již jak má vypadat správné řešení: Protože v zadání nebylo zcela jasně řečeno, jestli se vejce musí rozbít při pádu z nejvyššího patra a jestli se nerozbije při pádu z nejnižšího patra, budeme předpokládat, že oba tyto případy mohou nastat. Abychom je nemuseli speciálně ošetřovat, přidáme si nad poslední patro ještě jedno s tím, že při pádu z tohoto patra se vejce zaručeně rozbije. Nyní už můžeme začít hledat nejnižší patro, při pádu z nějž se vejce rozbije, protože víme, že takové patro zaručeně existuje.

Budeme zkoumat, mezi kolika nejvýše patry je možné rozeznat naše hledané patro, když máme k dispozici $k$ vajec a $l$ pokusů – tento počet pater si označíme $P(k,l)$. Je zřejmé, že $P(0,l) = 1$ – víme že z nějakého patra se vejce rozbije, nemáme žádný pokus, takže naše patro lze rozeznat pouze pokud není na výběr. Dále víme, že $P(k,k) = 2^k$, protože můžeme použít půlení intervalu a lépe to nejde. Interval pater, ve kterém hledáme, si vždy udržujeme tak, aby zaručeně obsahoval naše hledané patro. Tedy pokud se vejce nerozbije z patra $i$, tak počátek intervalu nastavíme na $i+1$, pokud se vejce rozbije, tak konec nastavíme na $i$. Pro $k > l$ je zřejmě $P(k,l) = P(l,l)$. Dalším důležitým pozorováním je, že $P(k,l) = P(k-1,l-1) + P(k,l-1)$. Tato rovnost plyne z toho, že pokud vejce pustíme z nějakého patra a ono se rozbije, tak lze naše hledané patro nalézt mezi $P(k-1,l-1)$ patry (rozbili jsme jedno vejce a udělali jeden pokus), pokud se vejce nerozbije, tak lze naše patro nalézt mezi $P(k,l-1)$ patry. A nyní přijde kouzlo (pro matematiky znalejší to asi nebude až takové kouzlo, ale dlužno poznamenat, že mě též nenapadlo, ale uviděl jsem ho v jednom řešení): Tvrdíme, že $P(k,l) = \sum_{i=0}^{k} \binom{l}{i}$. Ověření této rovnosti je snadná aplikace matematické indukce (využije se, že $\binom{l}{i} + \binom{l}{i+1} = \binom{l+1}{i+1}$), a proto ho vynecháme. Pro ty, kdo neví, co znamená $\binom{n}{k}$: je to tzv. *kombinační číslo* a říká, kolik různých $k$-prvkových podmnožin lze vybrat z množiny velikosti $n$. Je určeno následujícím vzorečkem:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!},$$

kde $n! = n \cdot (n-1) \cdot (n-2) \cdot \ \cdots \ \cdot 2 \cdot 1$.

Dále se nám bude hodit, že z $P(k,l)$ umíme rychle spočítat:

$$P(k-1,l) = P(k,l) - \binom{l}{k}$$

$$P(k,l+1) = P(k,l) + P(k-1,l) = 2 \cdot P(k,l) - \binom{l}{k}$$

$$P(k-1,l-1) = \left(P(k-1,l) + \binom{l-1}{k-1}\right)/2 =$$

$$= \left(P(k,l) - \binom{l}{k} + \binom{l-1}{k-1}\right)/2$$

*Pozn.:* Jak zajistíme, že máme vždy po ruce spočítané potřebné kombinační číslo, si můžete nalézt v programu.

Matematickou přípravu již máme za sebou a nyní je čas ukázat, jak to vše využijeme v našem algoritmu. Nejdříve musíme nalézt nejmenší $l$ takové, že $P(k,l) \geq n+1$ ($n$ je počet pater, máme jedno patro přidané nahoře). To snadno uděláme tak, že budeme zkoušet $l$ od 0 a průběžně počítat $P(k,l)$ – využíváme, že umíme spočítat $P(k,l+1)$ z $P(k,l)$. Když toto nejmenší $l$ nalezneme, začnou skutečné pokusy. V proměnné `nejmensi` si budeme uchovávat nejnižší číslo patra, ze kterého se vejce může rozbít (na počátku `nejmensi`$= 0$). Vejce vždy pustíme z patra $nejmensi + P(k-1,l-1) - 1$. Pokud se vejce rozbilo, tak snížíme $k$ i $l$ o jedna, přepočítáme $P$ a pokračujeme dalším pokusem. Pokud se vejce nerozbilo, zvýšíme `nejmensi` o $P(k-1,l-1)$, snížíme $l$ o jedna, přepočítáme $P$ (na přepočítávání $P$ používáme rovnosti uvedené výše) a opět pokračujeme dalším pokusem. Z definice $P$ takto zjevně po $l$ krocích nalezneme hledané patro a počet kroků byl nejmenší možný.

Paměťová složitost algoritmu je $O(1)$, časová $O(l)$, kde $l$ je nejmenší možný počet pokusů potřebný k nalezení patra.

*Honza Kára*

*Pro zájemce:* proč je $P(k,k) = 2^k$? Tento zápis znamená, že na $k$ pokusů umíme najít hledané patro, jen pokud ho hledáme nanejvýš mezi $2^k$ patry (víme, že jedno z nich je ono hledané). Pokud použijeme půlení intervalu, opravdu tohoto výsledku dosáhneme (při každém pokusu se můžeme zbavit poloviny pater). Ale nejde to lépe?

Jeden pokus skončí jednou ze dvou různých možností (vajíčko se rozbije/nerozbije). Dva pokusy skončí jednou ze čtyř různých možností (první vajíčko se rozbije/nerozbije a druhé se rozbije/nerozbije). Obecně $k$ pokusů skončí jednou z $2^k$ možností. Hledané patro určíme podle toho, jakou možností našich $k$ provedených pokusů skončilo. Jedné takové možnosti (posloupnosti výsledků) může odpovídat nanejvýš jedno patro, protože jinak bychom po provedení $k$ pokusů stále nebyli rozhodnuti, které patro je to hledané. Protože ale možností je $2^k$, není možno najít hledané patro mezi více než $2^k$ patry.

*Milan Straka*

## 16-2-3 Král Potvorník

Idea algoritmu je jednoduchá. Vezmeme body, setřídíme je dle $X$, vezmeme provázek, přivážeme ho k nejlevějšímu bodu a budeme jím naší množinu sloupů obtahovat po směru hodinových ručiček. Konvexní obal tedy bude po celou dobu „zatáčet“ doprava. Nyní nechť máme zpracované body $1..N$ zleva a známe jejich konvexní obal. Chceme přidat další bod. Připojíme ho tedy k seznamu konvexního obalu a „zatáhneme“ za provázek. Ten se vzdálí ode všech sloupů u nichž „zatáčel vlevo“. Za povšimnutí stojí, že tyto sloupy tvoří v konvexním obalu souvislý úsek, který končí před právě přidaným bodem (úsek může být i prázdný). Takže

v programu se stačí kouknout na 3 poslední body nynějšího obalu a dokud tam obal „zatáčí“ vlevo (čili je nekonvexní), předposlední bod vyhazovat. Protože každý bod můžeme vložit, resp. vyhodit nejvýše jednou, $N$ kroků trvá $O(N)$.

Pokud započítáme i třídění, složitost celého algoritmu je $O(N \log N)$. Ale má to drobný háček. Takhle bychom dopadli jako v jistém večerníčku, protože bychom vytvořili jen horní část konvexního obalu. Nicméně není těžké si rozmyslet, že stejně lze zkonstruovat i dolní polovinu (nebo k „vstupní“ posloupnosti bodů připojit na konec posloupnost stejných bodů, které budou setříděné obráceně, a pustit výše popsaný algoritmus na tuto prodlouženou posloupnost. Zřejmě pak bude provázek stále „zatáčet doprava“, ale jelikož jsme se „otočili“ o 180 stupňů, vytvoří i dolní část obalu).

Program je jen implementací výše uvedeného, časová náročnost je $O(N \log N)$ a paměťová $O(N)$.

Na závěr bych přidal jednu poznámku k došlým řešením. Relativně hodně jich běhalo v čase $O(N \log N)$, ale používalo při tom funkce typu `ArcTan` apod... Proč? Tyto funkce nepatří, přiznejme si, zrovna k nejrychlejším a program lze napsat bez nich stejně jednoduše. Ideální je použít vektorový součin.

Vektorový součin třísložkových vektorů $A$ a $B$ je takový třísložkový vektor $C$, pro který platí:

1) Velikost $C$ je rovna ploše rovnoběžníku definovaného vektory $A$ a $B$. Matematicky: $|C| = |A| \cdot |B| \cdot sin(\varphi)$, kde $\varphi$ je úhel, který svírají vektory $A$ a $B$.
2) Vektor $C$ je kolmý na vektory $A$ a $B$ (zřejmě je-li $B$ násobkem vektoru $A$, vektor $C$ nemůže být určen jednoznačně (vektorů kolmých na $A$ a $B$ je mnoho). V tomto případě je naštěstí – dle bodu 1 – vektor $C$ nulový).
3) Vektor C má (není-li nulový) směr, který odpovídá směru palce pravé ruky,pokud jí přiložíme k vektoru $A$ a prsty „stočíme“ po menším úhlu k vektoru $B$. Tato podmínka vlastně pouze určuje znaménko vektoru $C$.

Tohle stačí k tomu, abychom výsledný vektor jednoznačně definovali. Jak se tedy vektorový součin třísložkových vektorů $A$ a $B$ počítá?

$$A \times B = (A_y \cdot B_z - A_z \cdot B_y, A_z \cdot B_x - A_x \cdot B_z, A_x \cdot B_y - A_y \cdot B_x)$$

My použijeme vektorový součin takto: pro tři body $A$,$B$,$C$, které leží v tomto pořadí na obvodu zahrady, nám vektorový součet řekne, zda $|\angle ABC| < 180°$, měřeno od polopřímky $BA$ po směru hodinových ručiček. Vlastně nám poví, zda bude provázek v bodě $B$ „zatáčet“ doleva ($|\angle ABC| > 180°$) či doprava ($|\angle ABC| < 180°$), tedy zda bod $B$ patří nebo nepatří do konvexního obalu.

Výpočet provedeme takto: vektor $U$ bude vektor od bodu $B$ k bodu $A$, vektor $V$ bude vektor od bodu $B$ k bodu $C$. Spočteme $z$-ovou souřadnici součinu $U \times V$ (všimněte si, že při tomto výpočtu nepotřebujeme $z$-ové složky vektorů $U$ a $V$, takže tyto vektory mohou být pouze dvousložkové – to ale chceme, neboť zadané body leží v rovině a tedy mají pouze dvě souřadnice). Bude-li $z$-ová souřadnice součinu kladná, $|\angle ABC| < 180°$, bude-li záporná, $|\angle ABC| > 180°$ (rozmyslete si, jak to plyne z bodu 3).

*Pavel Čížek*

## 16-2-4 Křížový král

Hojnost neoptimálních řešení, která jsme dostali, zdá se nasvědčovat tomu, že drak je v posledních letech výjimečně sytý a dobře naladěný, či že mu možná posledních pár set let nejde počítání pokusů tak bystře, jako ve starých dobrých časech udatných a hlavně křupavých těch, no, rytířů. Zde budeme ctěnému publiku demonstrovati zaručeně spolehlivý návod, kterak draka důmyslem svým přechytračiti (čili tzv. návod na draka).

Abychom si zjednodušili práci, předpokládejme bez újmy na obecnosti, že nehádáme čísla od 1 do $N$, nýbrž od 0 do $N-1$ a že $N$ je mocnina dvojky (kdyby nebylo, stačí $N$ zvýšit, čímž si, jak uvidíme, pohoršíme jen nepatrně).

Kdyby drak nelhal (což je podobně pravděpodobné, jako kdyby peklo zamrzlo a turecký med byl zadarmo... ale potěšme se tou představou aspoň chviličku), bylo by všechno jednodušší: stačilo by se postupně vyptat na hodnoty jednotlivých bitů čísla, tj. v $i$-tém kroku hry nabídnout drakovi ta čísla, jejichž $i$-tý bit je jedničkový. Takto bychom číslo uhodli po $\log N$ krocích (log bude vždy značit dvojkový logaritmus). Podobnost s půlením intervalů není nikterak náhodná. Také si všimněte, že naše strategie nijak nespoléhá na to, že hádáme číslo z intervalu – kdybychom volili mezi nějakými obecnými možnostmi $a_0, \ldots, a_N$, postupovali bychom úplně stejně, až na to, že bychom se nerozhodovali podle bitů čísel $a_i$, nýbrž podle bitů jejich indexů $i$.

Jenže drak samozřejmě lže (a peklo není studené a turecký med je nekřesťansky drahý...), takže když se zeptáme stejně jako předtím, nedostaneme jednoznačný výsledek, nýbrž více možností: jednak tu dřívější, odpovídající tomu, že drak nelhal, a dále $\log N$ dalších, odpovídajících všem možným otázkám, na které drak mohl odpovědět lživě. Ale jaképak s tím ciráty, použijeme ještě jednou stejnou strategii, tentokráte ovšem pouze na zbylých $\log N + 1$ možností (zde použijeme trik s indexy). Na to budeme potřebovat $\lceil \log(1 + \log N) \rceil$ otázek.

Leč drak nám také mohl lhát během této druhé fáze, proto mezi první a druhou fázi přidáme ještě jeden dotaz, kterým si ověříme, zda výsledek první fáze byl správný a pokud byl, druhou fázi ani nespustíme.

Nyní si rozmysleme, jak by nám drak mohl lhát: Pokud lže v první fázi, ověření neprojde a druhá fáze (určitě bez lži) najde správný výsledek. Pokud lže v ověření, jsou všechny ostatní odpovědi správně a druhá fáze opět vydá správný výsledek. Kdyby se vyskytovala lež v druhé fázi, znamenalo by to, že odpovědi jak v první fázi, tak během ověření byly pravdivé, což by ovšem znamenalo, že jsme se do druhé fáze ani nedostali.

Celkem jsme potřebovali (včetně zaokrouhlení na mocninu dvojky) $\lceil \log N \rceil + \lceil \log(1 + \lceil \log N \rceil) \rceil + 1$ otázek, což pro $N = 1\,000\,000$ opravdu dá kýžených 26. Otázky a odpovědi zvládneme zpracovat jednoduchým programem s časovou složitostí $O(N \cdot \log N)$ a paměťovou $O(N)$.

*Pozor, matematika!* Pro odolnější povahy připojíme navíc důkaz (skoro)optimality našeho řešení. Představme si, že máme program hádající číslo pomocí binárních otázek, přičemž každá otázka je jednoznačně určena odpověďmi na předcházející otázky. Předpokládejme navíc, že program pokaždé použije stejný počet $r$ otázek (pokud nepoužije, můžeme vždy na konec doplnit zbytečné otázky, které nijak neovlivní výsledek). Přiřaďme každému číslu $x$ posloupnost $p_1(x), \ldots, p_r(x)$ odpovědí (kódovaných 0=ne, 1=ano), které vedou k tomuto číslu bez lhaní, a dále posloupnosti $p_1^i(x), \ldots, p_r^i(x)$ pro $i = 1, \ldots, r$ popisující odpovědi v případě, že drak zalhal při $i$-té otázce.



Snadno nahlédneme, že všech $r+1$ posloupností odpovídajících témuž číslu musí být navzájem různých (dvěma různými lžemi není možné dojít k témuž výsledku) a že libovolná posloupnost pro číslo $x$ musí být různá od libovolné posloupnosti pro každé jiné číslo $y$ (protože jinak bychom nebyli schopni z posloupnosti odpovědí jednoznačně zrekonstruovat drakovo číslo).

Existuje tedy $N \cdot (r+1)$ různých posloupností odpovědí, ty se ovšem musí vejít do množiny všech možných binárních posloupností délky $r$, kterých je $2^r$, a proto musí platit $N \cdot (r+1) \leq 2^r$, čili $N \leq 2^r/(r+1)$, což nám pro každý počet dotazů $r$ říká, kolik nejvýše čísel jsme schopni rozlišit. Pokud zkusíme nerovnost „obrátit“, vyjde nám opravdu přibližně $r \geq \log N + \log\log N$, čili náš algoritmus je velice blízko k optimu.

Pro ty, jimž by nedalo spát, že jsme jim nabídli řešení jenom *skoro* optimální, přidáváme konstrukci fungující pro libovolné $r = 2^z - 1$, která opravdu dosáhne maximálního možného $N$, které nám z naši nerovnosti vyšlo, tedy $N = 2^r/(r+1) = 2^{r-z}$. Nejdříve si sestrojíme *binární kód se vzdáleností 3*, čili množinu $N$ dvojkových posloupností délky $r$ (těm budeme říkat slova) takových, že každé dvě slova se liší v alespoň třech prvcích. K tomu stačí vzít všechny možné $(r-z)$-bitové posloupnosti (těch je $N$) a za každou z nich přidat $z$ *kontrolních bitů,* abychom dosáhli požadované vzdálenosti. Původním bitům budeme říkat *datové* a oindexujeme si je $z$-ciferными dvojkovými čísly, přičemž vynecháme nulu a všechna čísla obsahující právě jednu jedničku (takže nám zbude $2^z - z - 1 = r - z$ indexů, jak potřebujeme). Kontrolní bity pak spočítáme takto: $i$-tý kontrolní bit bude určovat paritu datových bitů, jejichž index má $i$-tý bit jedničkový (parita je 0, pokud je mezi vybranými datovými bity sudý počet jedniček, jinak je 1). Příklad pro $N = 11$ a $z = 4$ následuje:

| $d_0$ | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ | $d_6$ | $d_7$ | $d_8$ | $d_9$ | $d_{10}$ | $c_0$ | $c_1$ | $c_2$ | $c_3$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | ↵ | | | |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | | ↵ | | |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | | ↵ | |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | ↵ |

Každá dvě sestrojená slova se budou lišit alespoň jedním bitem v datové části. Pokud se liší právě jedním, budou se lišit i alespoň dvěma kontrolními bity (protože v indexu změněného datového bitu jsou alespoň dvě jedničky). Kdyby se datové části lišily dvěma bity, musí mít jejich indexy alespoň jeden bit rozdílný, takže alespoň jeden kontrolní bit se také liší. Čili každá dvě slova se liší v alespoň třech bitech, jak jsme slíbili.

Naši množinu slov lze použít jako samoopravný kód na opravu jedné chyby: pokud nám někdo pošle libovolné ze slov a při přenosu nastane nejvýše jedna chyba, jsme schopni jednoznačně určit, které slovo bylo vysláno. (Pokud by se slovo $x$ změněné v jednom bitu shodovalo se slovem $y$ změněným rovněž nejvýše v jednom bitu, znamenalo by to, že slova $x$ a $y$ se liší v max. dvou bitech, což víme, že není možné.)

Stejným způsobem ovšem můžeme komunikovat s drakem: číslům 0 až $N-1$ přiřadíme našich $N$ slov a v $i$-tém kole se draka zeptáme na $i$-tý bit slova (vyjmenujeme čísla, jejichž slova mají tento bit jedničkový), čímž dostaneme hledané slovo s maximálně jedním bitem změněným a ten je možno opravit. Máme tedy strategii, která je optimální pro vybraná $r$ (zkuste nahlédnout, že funguje a je optimální i pro ostatní $r$), je ji možno naprogramovat v čase $O(N \cdot \log N)$ a paměti $O(N)$ a dokonce nám i spolehlivě odhalí, zda drak zalhal. Náš program se nicméně bude držet mnohem jednoduššího prvního řešení, které je vždy o nejvýše 2 dotazy horší než optimální.

*Martin Mareš*

## 16-2-5 Král Popleta

Nejprve si úlohu převedeme do řeči logických výrazů, čili výrazů nad proměnnými, které mohou nabývat pouze hodnot `true` a `false`. S těmito proměnnými můžeme provádět různé logické operace. Pro náš příklad budou důležité pouze dvě – negace, definovaná vztahy `not false` = `true` a `not true` = `false`, a disjunkce, definovaná vztahy `false or false` = `false`, `false or true` = `true`, `true or false` = `true` a `true or true` = `true`.

Pro každou provincii si zavedeme jednu proměnnou. Ta bude nabývat hodnoty `true`, pokud provincii získá Petr, a `false`, pokud ji získá Pavel.

Jak snadno nahlédneme, požadavky druhého a třetího typu jsou ve skutečnosti stejné – oba jsou splněny, pokud Petr (resp. Pavel) dostane alespoň jednu ze zmiňovaných provincií. Všechny požadavky tedy jdou vyjádřit jedním z těchto způsobů:

- Jestliže Petr požaduje provincii $x$, odpovídá tomu výraz $(x)$.
- Jestliže Pavel požaduje provincii $x$, odpovídá tomu výraz (`not` $x$).
- Jestliže Petr požaduje provincii $x$ nebo provincii $y$, odpovídá tomu výraz ($x$ `or` $y$).
- Jestliže Pavel požaduje provincii $x$ nebo provincii $y$, odpovídá tomu výraz (`not` $x$ `or not` $y$).

Úkolem této úlohy je nalézt takové přiřazení hodnot `true` a `false` proměnným, aby co nejvíce z těchto výrazů mělo hodnotu `true`.

Protože každé tři požadavky lze splnit, platí:

- Pro žádnou proměnnou $x$ se mezi požadavky nevyskytuje jak $(x)$, tak (`not` $x$). Můžeme tedy bez újmy na obecnosti předpokládat, že všechny požadavky, které obsahují pouze jednu proměnnou, jsou ve tvaru $(x)$ – pokud je požadavek ve tvaru (`not` $x$), ve všech požadavcích prohodíme $x$ za `not` $x$ a naopak. Tím zjevně nezměníme počet splnitelných požadavků – pokud by řešení nově vzniklé úlohy dávalo $x$ hodnotu `false`, dáme mu v řešení původní úlohy hodnotu `true` a naopak.
- Nemůže se stát, že by (`not` $x$ `or not` $y$), $(x)$ a $(y)$ byly požadavky. Tedy platí, že pro každý požadavek tvaru (`not` $x$ `or not` $y$) se buď $(x)$ nebo $(y)$ nevyskytují jako požadavek.

Nyní se vraťme k řešení úlohy. Řešení, které se králi nelíbilo, bylo dát každé proměnné hodnotu `true` s pravděpodobností 1/2. Tím by splnil požadavky zmiňující dvě provincie s pravděpodobností 3/4, nicméně kazí mu to požadavky na jednotlivé provincie, které splní s pravděpodobností pouze 1/2. Zkusme tedy pomoci těmto požadavkům – dejme proměnné $x$, která se vyskytuje sama jako požadavek, hodnotu `true` s pravděpodobností $p > 1/2$. Ostatní proměnné, které se nevyskytují jako samostatné požadavky, dále volme s pravděpodobností 1/2.

Požadavky typu $(x)$ na jednotlivé provincie splníme s pravděpodobností $p$. Požadavky typu (`not` $x$ `or not` $y$) splníme s pravděpodobností $1 - p/2$ pokud se právě jedno z $x$,