Další informatické hádanky

PETR OSIČKA – PETR KRAJČA

Přírodovědecká fakulta UP, Olomouc

Článek je pokračováním informatických hádanek publikovaných v předchozích dílech časopisu Matematika–Fyzika–Informatika.

1. Ponožkový kvíz

Hádanka je z oblasti interaktivních dokazovacích systémů. K jejímu vyřešení není zapotřeby hlubokých znalostí matematiky, postačí elementární znalost pravděpodobnosti.

Uvažujme osobu A, která je barvoslepá, a osobu B, která má různobarevné ponožky. Osoba B chce o tom, že má různobarevné ponožky, přesvědčit osobu A. Ta je ovšem barvoslepá, a ponožky jí připadají stejné. Úkolem je vymyslet scénář, ve kterém figurují jenom A, B a předměty, které mají lidé běžně u sebe, ve kterém B může přesvědčit A o různobarevnosti svých ponožek s pravděpodobností libovolně se blížící jedné.

Řešení

Bob skutečně může přesvědčit Alici o různobarevnosti svých ponožek s pravděpodobností blížící se jedné. Stačí k tomu mince. Bob dá ponožky Alici, Alice drží každou ponožku v jedné ruce. Poté se Bob otočí k Alici zády tak, aby neviděl na ponožky. Alice si hodí mincí a pokud padne panna, tak ponožky prohodí (tj. ponožku, kterou drží v levé ruce, si dá do pravé ruky a ponožku z pravé ruky bude držet levou rukou). Bob se otočí zpět a Alice se jej zeptá, jestli ponožky prohodila. Pokud jsou ponožky skutečně různobarevné, Bob to pozná s pravděpodobností 1. Pokud nejsou různobarevné, Bob si pravděpodobností $1/2$ správný výsledek tipne. Předchozí pokus lze opakovat, řekněme, že jsme jej zopakovali n-krát. Pokud ponožky nejsou různobarevné, je pravděpodobnost, že Bob odpoví n-krát správně pouhým tipováním $1/2^n$. Například pro $n = 10$ už je to méně než $1/1000$. Pokud tedy Bob odpoví ve všech opakováních správně, přesvědčí Alici s pravděpodobností $1 - 1/2^n$.

2. Velmi záporné číslo

Může existovat hodnota, která je tak moc záporná, že i její absolutní hodnota bude záporná?Přeformulována do jazyka C by naše hádanka mohla vypadat následovně.

```
#include <stdlib.h>

int main() {
    int x = /* ??? */;
    if (abs(x) < 0) {
        printf("Kdo by to byl řekl, že?\n");
    }
    return 0;
}
```

Otázka zní, může existovat nějaká hodnota `x`, kdy se vykoná daná podmínka. Pokud ano, jaká hodnota to je?

Řešení

Z matematického pohledu tato otázka nedává příliš smysl, protože absolutní hodnota nějakého celého čísla musí být vždy nezáporná. Když se na tuto hádanku podíváme ale technickým pohledem, už to začne být zajímavější.

V první řadě je potřeba si uvědomit, že rozsah čísel, která je schopen pojmout datový `int` je omezen. Dále je potřeba si uvědomit, že pro reprezentaci záporných hodnot typu `int` se v soudobých počítačích používá dvojkový doplněk. To nám dává, že proměnná typu `int` je schopna pojmout hodnoty od -2^{n-1} do $2^{n-1}-1$, kde n odpovídá počtu bitů určených pro typ `int`, typicky je to 32 bitů.

Teď už by mělo být zřejmé v čem je problém – záporných čísel, která typ `int` pojme, je přesně o jedno víc, než těch kladných. Není proto způsob, jak uložit do proměnné typu `int` absolutní hodnotu čísla -2^{n-1}, někdy též označovanou jako `INT_MIN`. Pojďme se podívat, jak se tedy s tímto problémem vypořádat.

Jazyk C, resp. specifikace jeho standardní knihovny, si nad tímto problémem myje ruce a říká, že v případě, kdy se výsledek funkce `int abs(int)` nevejde do typu `int`, je výsledek nedefinován.

Vezmeme-li si přirozený zápis funkce `int abs(int)`, který lze najít ve standardní knihovně, dostaneme odpoveď na naši hádanku.

```
int abs(int i) {
  return (i < 0 ? -i : i);
}
```

Předáme-li jako argument `i` hodnotu -2^{31} (`INT_MIN`), dostaneme pro `-i`, díky vlastnostem dvojkového doplňku, opět hodnotu `INT_MIN`. Proto v tomto případě bude platit `(abs(INT_MIN)) < 0`.

2.1. Jak je to v ostatních jazycích

Jazyk C je proslulý celou řadou takových zákoutí, ale jak jsou na tom ostatní jazyky? Nejlépe jsou na tom jazyky, které umí pracovat s čísly s libovolnou přesností jako jsou Lisp, Scheme nebo Python, tam tento problém odpadá. Jazyk Java se chová stejně jako C, s tím rozdílem, že toto podivné chování je řádně zdokumentované a je součástí specifikace. Naproti tomu jazyk C# nabízí o poznání čistější řešení. Pokud je jako argument předána hodnota -2^{31}, skončí metoda `Math.abs(int)` výjimkou `OverflowException`. To dává docela dobře smysl, jen programátor musí myslet na to, že i takto jednoduchá funkce může skončit výjimkou. Řešení, které nabízí jazyk PHP může přinést programátorovi taky nepříjemné překvapení. Pokud jako argument předáme hodnotu -2^{63} typu `int` (na 64bitovém systému), interpretr se s tím vypořádá tak, že vrátí sice kladnou hodnotu ale jako číslo typu `float`, tj. číslo s plovoucí řádovou čárku. Takže místo skutečné absolutní hodnoty dostaneme jen její aproximaci.

2.2. Proč by nás to mělo zajímat?

Na první pohled se může zdát, že se jedná o okrajový problém a zajímavou hříčku. V reálných programech může mít neuvědomění si této okrajové vlastnosti zásadní důsledky, včetně dopadů na bezpečnost. Uvažujme následující kód, který pracuje s obrázkem patrně v nějakém omezeném bufferu.

```
void process_image(image *img) {
  int size = abs(img->width * img->height);
  if (size > IMG_MAX_W * IMG_MAX_H) {
     ERROR("Image too large");
  }
  /* ... */
}
```

Pokud bychom, jako ve výše zmíněném příkladu, zapomněli na to, že absolutní hodnota může být i záporná, mohl by útočník vhodně nastavenými parametry obejít kontrolu maximální velikosti obrázku. Následně to využít k vyvolání přetečení bufferu a ke spuštění libovolného kódu. Tento typ bezpečnostního problému se proto někdy označuje jako IO2BO (Integer Overflow to Buffer Overflow). Pokud by takový kód byl například ve webovém prohlížeči (a to skutečně byl), stačilo by útočníkovi, aby se prohlížeč pokusil otevřít vhodně upravený obrázek a útočník by mohl spustit libovolný kód na vašem počítači.

3. Svoboda za minci

Vězni A a B si zahrají se žalářníkem následující hru. Žalářník připraví šachovnici o velikosti 8×8 políček. Na každé políčko umístí minci, některé z mincí jsou nahoru orlem, jiné hlavou. Poté žalářník zavolá k šachovnici vězně A (vězeň B v ten moment šachovnici nevidí) a ukáže na jednu minci na šachovnici. Vězeň A si poté také jednu minci vybere, obrátí ji a odejde. Potom zavolá žalářník vězně B a požádá ho, aby našel minci, na kterou žalářník ukázal před vězněm A. Pokud vězeň B tuto minci najde, jsou oba vězni propuštěni. Vězni si mohou dopředu připravit strategii, po zahájení hry už ovšem spolu nesmí komunikovat. Existuje strategie, se kterou vězni vždy zvítězí?

Řešení

Vězeň B může poznat, kterou minci žalářník vězni A ukázal, pouze ze situace na šachovnici, například ze seznamu míst, na kterých je mince otočená hlavou nahoru. Potřebujeme tedy nějakou funkci, která množině takových míst přiřadí místo, které označil žalářník. Úlohou vězně A bude otočit minci na šachovnici tak, aby funkce vrátila správné místo, vězeň B ji pak spočítá. Ve zbytku článku si ukážeme příklad takové funkce.

Očíslujme políčka na šachovnici čísly 0 až 63. Každé z těchto čísel můžeme reprezentovat pomocí řetezce 6 bitů. Dále předpokládejme, že na šachovnici je n míst s mincí otočenou hlavou nahoru. Bitové reprezentace těchto míst označíme pomocí T_i pro $i = 1, \ldots, n$. Jako J označíme bitový zápis místa, které vybral žalářník, a jako X bitový zápis místa, na kterém hráč A otočí minci. Naše funkce je prostou $\oplus$ operací XOR aplikovanou po bitech na místa, na kterých je hlava. Hráč A tedy musí nalézt hodnotu X v rovnici

$$(T_1 \oplus T_2 \oplus \cdots \oplus T_n) \oplus X = J.$$

Řešením je

$$X = (T_1 \oplus T_2 \oplus \cdots \oplus T_n) \oplus J.$$

Pokud je na místě X před otočením mince orel, protokol je zřejmě správně. Pokud je ovšem na tomto místě před otočením mince hlava, mohlo by se zdát, že předchozí trik nefunguje. Uvědomíme-li si však, že v tomto případě máme $X = T_i$ pro nějaké i a navíc pro libovolný bitový řetězec S platí $S \oplus S = 0$, pak vidíme, že

$$\begin{aligned} J &= (T_1 \oplus \cdots \oplus T_i \oplus \cdots \oplus T_n) \oplus T_i \\ &= T_1 \oplus \cdots \oplus T_i \oplus T_i \oplus \cdots \oplus T_n \\ &= T_1 \oplus \ldots T_{i-1} \oplus T_{i+1} \oplus \cdots \oplus T_n, \end{aligned}$$

a tedy hráč B spočítá správnou pozici.

4. Třikrát XOR

Uhodnete, jaký smysl má výraz v „záhadné“ funkci? Víte, proč byste se mu měli vyhnout? Uvažujme následující kód v jazyce C se záhadným výrazem na druhém řádku. Dokážete určit, co dělá?

```
1 void mystery(int x, int y) {
2    x ^= y ^= x ^= y;
3    printf("%i %i\n", x, y);
4 }
```

Řešení

Než se dostaneme k samotnému řešení hádanky, odpovíme si na druhou otázku. *Víte, proč byste se mu měli vyhnout?*

Výrazům tohoto typu byste se měli za každou cenu vyhnout, protože z něj není jasně patrné, jaký má smysl.[1)] Je důležité mít na paměti, že s programem nebude pracovat jen počítač, ale často i jiní lidé. Nesrozumitelný kód nejenom může okrádat o cenný čas, ale může být zdrojem chyb, pokud jiný programátor pochopí váš kód špatně.

Je dobrým programátorským zvykem, že jeden řádek odpovídá jednomu kroku výpočtu. Toto pravidlo zvyšuje čitelnost kódu a usnadňuje pozdější krokování výpočtu. V našem případě je toto pravidlo porušeno hned třikrát. Přepišme si proto výraz do ekvivalentní posloupnosti tří operací.

[1)] Na druhou stranu, pokud by to bylo zjevné, hádanka by postrádala smysl.

```
x ^= y;
y ^= x;
x ^= y;
```

Kód vypadá o něco jednodušeji, ale stále není poznat, co je jeho smyslem. V každém případě můžeme už u tohoto kódu lépe studovat, co dělá.

4.1. Krok po kroku

Po provedení prvního řádku budou proměnné `x` a `y` obsahovat hodnoty:

- `x` ... $x \oplus y$,[2)]
- `y` ... y.

V druhém kroku se nám změní hodnota proměnné `y` na $(y \oplus (x \oplus y))$. Tento výraz lze dále zjednodušit. Víme, že operace $\oplus$ je komutativní a asociativní. Proto můžeme prohodit operandy a následně přesunout závorky. Platí tedy $y \oplus (x \oplus y) = (x \oplus y) \oplus y = x \oplus (y \oplus y)$. Protože $y \oplus y = 0$, bude proměnná `y` obsahovat hodnotu $x \oplus 0$, což odpovídá přímo hodnotě x. Lze tedy říct, že po druhém kroku budou proměnné `x` a `y` obsahovat hodnoty:

- `x` ... $x \oplus y$,
- `y` ... x

V posledním kroku nastavíme hodnotu `x` na $(x \oplus y) \oplus x$. Podobým postupem (s využitím komutativity a asociativity) získame $(x \oplus y) \oplus x = (y \oplus x) \oplus x = y \oplus (x \oplus x)$. Protože $x \oplus x = 0$, budeme mít v proměnné `x` uloženou hodnotu $(y \oplus 0)$, tedy hodnotu y. Po provedení všech tří operací máme v proměnných:

- `x` ... y,
- `y` ... x.

Můžeme tedy říct, že daný výraz prohodí obsah dvou proměnných, aniž by byla potřeba další proměnná.

[2)] Symbolem $\oplus$ označujeme operaci XOR.

4.2. Proč to (ne)používat

V minulosti podobné triky patřily do výbavy *skutečných programátorů*™, protože dokázaly ušetřit cenné byty paměti a patrně i nějaký hodinový cyklus procesoru.

Z dnešního pohledu takový kód lze považovat za velmi špatný. Jak již bylo zmíněno v úvodu, znepřehledňuje program a zakrývá skutečné záměry programátora.

Co hůř, dopad na rychlost bude dnes spíše negativní než pozitivní. Moderní procesory již nelze chápat jako obvod, který postupně načítá jednotlivé instrukce z paměti a pak je mechanicky jednu po druhé provádí. V dnešních procesorech jsou jednotlivé instrukce načteny a poté rozloženy na tzv. mikroperace, které jsou pak přiděleny odpovídajícím jednotkám procesoru. I když to tak navenek nevypadá, procesor provádí více mikrooperací souběžně a často i s předstihem.

Do jisté míry by se dalo říct, že procesor obsahuje „překladač“, který překládá strojový kód na jednodušší operace a stará se o uspořádání instrukcí tak, aby všechny jednotky byly co nejlépe vytíženy. Z toho důvodu se počet registrů, které procesor nabízí k použití a které skutečně má, může významně lišit. Například u nejnovějších procesorů AMD s architekturou Zen 2 má programátor k dispozici pouze 16 registrů (obecně použitelných celočíselných), ale interně CPU využívá až 180 fyzických registrů. V procesoru pak funguje mapování jednotlivých registrů na registry fyzické.

V kontextu našeho příkladu to znamená, že pokud má procesor provést operaci `mov rbx, rax`, která přiřadí obsah registru `rax` do `rbx`, ve skutečnosti žádnou operaci provést nemusí a jen interně změní mapování registrů, tj. nastaví, že registry `rax` a `rbx` ukazují na stejný fyzický registr. Proto prohození dvou hodnot běžným způsobem

```
t = a;
a = b;
b = t;
```

bude na soudobých procesorech rychlejší než použití tří zřetězených logických operací.

Podobně se v minulosti používala konstrukce `x ^= x`, která, jak jsme už ukázali, nastaví hodnotu proměnné na 0. V minulosti to na některých procesorech mělo mírný efekt. U dnešních procesorů využití logických operací pro vynulování hodnoty nedává smysl a navíc to komplikuje interní optimalizace. Procesor proto interně takovou operaci automaticky převádí

na `x = 0`.

Závěrem lze říct, že pokud neprogramujete v prostředí, které je značně omezeno dostupnými prostředky, jako jsou paměť a rychlost procesoru, měli byste se popsaným konstrukcím vyhnout, psát kód co nejsrozumitelnější a optimalizace nechat na překladači a procesoru.

5. Čtvrtá možnost ze tří

Konstrukce `switch-case` v jazycích typu C umí být pěkně záludná. Zapomenutý nebo špatně umístěný příkaz `break` potrápil už nejednoho programátora a způsobil nemalé škody.[3] V této hádance[4] se však záludnost skrývá zcela jinde.

Následující kód lze z pohledu jazyka C považovat za validní a lze jej přeložit. Dokážete určit, co bude výstupem programu?

```
#include <stdio.h>
int main() {
    int n = 3;
    switch (n) {
        case 1:
            printf("foo\n");
            break;
        case 2:
            printf("bar\n");
            break;
        deufalt:
            printf("baz\n");
    }
    return 0;
}
```

Pokud vaše odpověď zní, že program vypíše na standardní výstup některý z uvedených řetězců, jedná se o odpověď chybnou. Tento program totiž nevypíše nic. Program obsahuje chybu, přesněji překlep v klíčovém slově `default`, a proto příkaz `switch` neobsahuje větev výpočtu, která se provede v případě, že neexistuje vhodná větev `case`. Nabízí se otázka, proč

[3] http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html
[4] Inspirováno http://www.gowrikumar.com/c/index.php

jde program přeložit. V případě jazyka C je onen překlep, tj. `deufalt:` intepretován jako návěští pro skok pomocí příkazu `goto`, a proto je kód chápán překladačem jako zcela v pořádku.

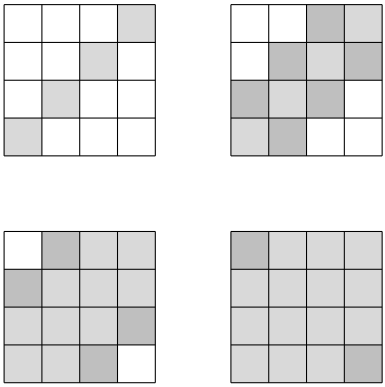
Na druhou stranu překladač vás nepřímo může na tento typ chyby upozornit. Pokud jsou zapnutá upozornění[5)], zobrazí informaci, že existuje návěští, na které nevede žádný příkaz `goto`, nebo že `switch` nepokrývá všechny možnosti.

5.1. Nakažená šachovnice

Představme si, že na zobecněné šachovnici, která má stranu dlouhou n políček, se mohou jednotlivá políčka nakazit virem. Z nakaženého políčka virus nikdy nezmizí. Do nenakaženého políčka se rozšíří v případě, že alespoň dvě jeho sousední políčka jsou nakažená. Uvažujeme přitom pouze horizontální a vertikální sousedy. Jaký je minimální počet políček, která musíme na začátku nakazit, aby se nákaza rozšířila na celou šachovnici?

Řešení

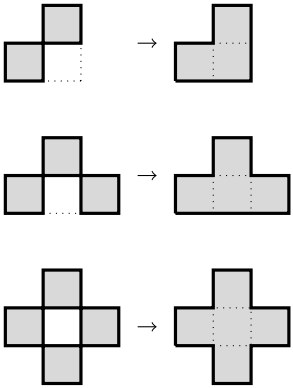
Po chvilce experimentování zjistíme, že lze najít n políček, ze kterých se nákaza úspěšně rozšíří. Můžeme vybrat například diagonálu. Viz následující příklad na šachovnici se stranou dlouhou 4 políčka.



Zbývá ověřit, že na začátku nelze vybrat méně než n políček. K tomu postačí následující úvaha. Změříme obvod souvislých nakažených oblastí, za jednotku zvolíme hranu jednoho políčka. Podíváme-li se na předchozí

[5)] např. přepínač -Wall v gcc

obrázky, tento obvod je u všech 4 šachovnic vždy 16 hran. To není náhoda. Nakazíme-li totiž nenakažené políčko, obvod nakažené části se nemůže zvětšit. Nakažením vždy z obvodu odebereme nejméně dvě hrany (hrany mezi nenakaženým políčkem a jeho nakaženými sousedy) a přidáme maximálně dvě hrany (hrany mezi nenakaženým políčkem a jeho nenakaženými sousedy). Viz příklady na následujícím obrázku.



Na konci má nakažená část obvod $4n$ hran, na začátku tedy musíme nakazit minimálně n políček.

6. Dělení mincí

Máme k dispozici herní desku s dostatečně dlouhou řadou políček uspořádaných zleva doprava. Do prvního políčka položíme sloupec mincí. Potom opakovaně provádíme následující tahy: vezmeme dvě mince z nějakého sloupečku, jednu minci zahodíme a tu zbývající umístíme na sloupeček o jedna vpravo. Toto opakujeme tak dlouho, dokud nemáme na desce jenom políčka obsahující nejvýše jednu minci. Zajímá nás: a) Záleží rozmístění mincí na konci hry na tom, v jakém pořadí vybíráme sloupečky pro přemístění mincí? b) Kolik políček potřebujeme, abychom dohráli hru s n mincemi? c) Kolik přemístění mincí musíme ve hře s n mincemi provést?

Řešení

Přiřadíme políčkům na herní desce indexy, jako by to bylo pole. První políčko má index 0, druhé políčko má index 1 atd. Pomocí b_i označíme počet mincí, které se na konci hry nachází na políčku i, a předpokládáme, že k je maximální index, pro který je $b_i = 1$. Například pro $n = 5$ a

následující průběh hry

```
5
3  1
1  2
1  0  1
```

máme $b_0 = 1$, $b_1 = 0$, $b_2 = 1$ a $k = 2$.

Pro pochopení hry je klíčový následující postřeh: pokud je někdy v průběhu hry na políčku i mince, museli jsme v některém minulém tahu vzít dvě mince z políčka $i-1$, a pokud je $i-1 > 0$, museli jsme vzít čtyři mince z políčka $i-2$, atd. Abychom na konci hry docílili $b_i = 1$, spotřebujeme k tomu 2^i mincí z prvního políčka. Pokud má být také $b_j = 1$ (pro $j \neq i$), pak spotřebujeme dalších 2^j mincí. Dostáváme tak, že

$$n = \sum_{i=0}^{i} 2^k$$

a $b_k b_{k-1} \dots b_0$ je zápis n ve dvojkové soustavě.

Teď můžeme snadno odpovědět na první dvě otázky. Protože zápis čísla ve dvojkové soustavě je pro každé číslo unikátní, na výběru políček pro provední tahů nezáleží. Počet políček, která potřebujeme, odpovídá délce zápisu ve dvojkové soustavě a ten je roven $\lfloor \ln(n) \rfloor + 1$.

Abychom nalezli odpověď na třetí otázku, všimněme si, že provedením jednoho tahu se zmenší počet mincí na desce o jedna. Provedeme tak $n - \sum_{i=0}^{k} b_i$ tahů. Suma ve výrazu je počet jedniček v zápisu n ve dvojkové soustavě.

Literatura

[1] *Levitin, A., Levitin, M.*: Algorithmic puzzles. Oxford University Press, New York, 2011.