Úloha I.S ... Rozjezdová

10 bodů; průměr 5,42; řešilo 33 studentů

a) *Upravte výraz* $\sqrt{x+1} - \sqrt{x}$ *tak, aby nebyl náchylný k problémům cancellation, ordering a smearing. Ke kterým z těchto problémů byl původně náchylný a proč? Jaký je rozdíl ve výsledku původního a opraveného výrazu, pokud jej vyčíslíme v double precision pro* $x = 1{,}0 \cdot 10^{10}$*?*

b) *Popište funkci následujícího kódu. Jaký je rozdíl mezi funkcemi* `a()` *a* `b()`*? Pro jaké hodnoty* `x` *je lze použít? Nebojte se kód spustit a hrát si s hodnotou proměnné* `x`*. Určete také asymptotickou časovou složitost programu v závislosti na proměnné* `x`*.*

```
def a(n):
  if n == 0:
    return 1
  else:
    return n*a(n-1)
def b(n):
  if n == 0:
    return 1.0
  else:
    return n*b(n-1)
x=10
print("{} {} {}".format(x, a(x), b(x)))
```

c) *Označme* o_k *a* O_k *obvod vepsaného a opsaného pravidelného* k*-úhelníku ke kružnici. Pak pro ně platí rekurentní vztahy*

$$O_{2k} = \frac{2o_k O_k}{o_k + O_k}\,, \; o_{2k} = \sqrt{o_k O_{2k}}\,.$$

Napište program, který pomocí těchto vztahů vypočítá hodnotu π*, začněte přitom s opsaným a vepsaným čtvercem. S jakou přesností dokážete* π *takto aproximovat? Obdobu tohoto postupu původně navrhl a použil Archimedes.*

d) *Lukáš a Mirek hrají hru. Házejí férovou mincí a když padne orel, dá Mirek Lukášovi jedno FYKOSí tričko, když padne panna, dá jedno tričko Lukáš Mirkovi. Oba dohromady mají* t *triček, z toho* l *patří Lukášovi a* m *Mirkovi. Pokud jednomu z hráčů dojdou trička, hra končí.*

1. *Nechť* $m = 3$ *a Lukášova zásoba triček je nekonečná. Určete* nejpravděpodobnější *dobu trvání hry, tedy počet hodů mincí, po nichž hra skončí (protože Mirkovi dojdou trička).*
2. *Nechť* $m = 10$, $l = 20$. *Proveďte simulaci pomocí generátoru pseudonáhodných čísel a nalezněte pravděpodobnost, že Mirek vyhraje všechna Lukášova trička. Celou hru nechejte proběhnout alespoň 100krát (čím více opakování, tím lépe).*
3. *Jak se změní výsledek předchozí úlohy, jestliže Mirek minci „vylepší“ a panna nyní padá s pravděpodobností* $5/9$*?*

Bonus *Vypočtěte pravděpodobnosti analyticky a porovnejte výsledek se simulací.*

e) *Mějme lineární kongruenční generátor s parametry* $a = 65\,539$, $m = 2^{31}$, $c = 0$.

1. *Vygenerujte alespoň 1 000 čísel a spočtěte jejich střední hodnotu a rozptyl. Porovnejte se střední hodnotou a rozptylem rovnoměrného rozdělení na stejném intervalu.*
2. *Nalezněte vztah, který vyjádří číslo v generované sekvenci jako lineární kombinaci čísel na dvou předchozích pozicích, tj. nalezněte koeficienty* A, B *v rekurentním vztahu* $x_{k+2} =$

$= Ax_{k+1} + Bx_k$. Pokud budeme považovat každá tři po sobě následující čísla za souřadnice bodu ve trojrozměrném prostoru, jak rekurentní vztah ovlivní prostorové rozložení těchto bodů?

Bonus *Vygenerujte sekvenci alespoň* 10 000 *čísel a vykreslete 3D bodový graf, který ilustruje význam uvedeného rekurentního vztahu.*

Mirek a Lukáš oprašovali staré učební texty.

a) Výraz nemůže být náchylný k problémům ordering a smearing, neboť se v něm nevyskytují žádné mnohokrát se opakující operace. Je ale náchylný ke cancellation, protože se v něm odečítají dvě čísla, která jsou si pro velká x poměrně blízká. Výraz se pokusíme upravit tak, aby se v něm toto odečítání nevyskytovalo. Platí

$$\sqrt{x+1} - \sqrt{x} = \frac{(\sqrt{x+1} - \sqrt{x})(\sqrt{x+1} + \sqrt{x})}{\sqrt{x+1} + \sqrt{x}} = \frac{x+1-x}{\sqrt{x+1} + \sqrt{x}} = \frac{1}{\sqrt{x+1} + \sqrt{x}}\,,$$

kde jsme použili vztah $(a+b)(a-b) = a^2 - b^2$. Pokud výrazy vyčíslíme pro $x = 1{,}0 \cdot 10^{10}$, dostaneme výsledek $4{,}999994416721165 \cdot 10^{-06}$ pro původní výraz a $4{,}999999999875 \cdot 10^{-06}$ pro opravený výraz. Jejich rozdíl tedy je zaokrouhleně $5{,}58 \cdot 10^{-12}$. Pokud bychom x zvětšovali, chyba by ještě rostla, až by při $x = 1{,}0 \cdot 10^{16}$ byla chyba stejného řádu, jako výsledek.

b) Funkce `a()` i `b()` počítají faktoriál svého argumentu pomocí rekurentních vztahů $0! = 1$ a $n! = n(n-1)!$. Obě funkce pochopitelně fungují pouze pro celá $n \geq 0$. Jediný rozdíl mezi funkcemi je v tom, že zatímco funkce `a()` počítá mezivýsledky i vrací konečný výsledek jako celé číslo, funkce `b()` místo toho používá čísla s plovoucí desetinnou čárkou. Odtud pak plynou veškeré odlišnosti v chování.

Pokud program spustíme pro $x \geq 19$, zjistíme, že $x! \gg 10^{16}$. Takto velké číslo již nelze přesně vyjádřit pomocí typu `double`, výsledek vrácený funkcí `b()` už nemusí tedy být přesný. Oproti tomu celé číslo vrácené funkcí `a()` je stále reprezentováno přesně, jazyk Python totiž dokáže pracovat s neomezeně velkými celými čísly. Sami si můžete ověřit, že již neplatí podmínka `a(x)+1 == b(x)+1`.

Nicméně máme štěstí, faktoriál „velkých" čísel končí na několik nul[1], které, když je reprezentace čísla s plovoucí desetinnou čárkou ořízne, výsledek nijak nezmění. Stále tedy platí `a(x) == b(x)`. Tato podmínka přestane platit až pro $x \geq 23$, kdy se na jednu z ořezávaných pozic dostane nenulová číslice.

Další změna nastane pro $x \geq 171$, kdy $x! \gg 10^{308}$. Takto velká čísla již vůbec nelze uložit do typu `double`[2] a funkce `b()` místo čísla začne vracet speciální hodnotu `'inf'`, tedy kladné nekonečno. To můžeme vidět ve výpisu na obrazovku, případně si můžeme otestovat platnost podmínky `b(x) == float('inf')`.

Python je v tom, že dokáže ukládat libovolně velká celá čísla, poněkud výjimečný. Většina programovacích jazyků má pevně daný (v daném standardu) počet bytů a tedy pevně danou maximální velikost celočíselných datových typů. Asi nejběžnějším typem je typ o délce 4 byty (32 bitů), do nejž lze uložit čísla v rozsahu $-2\,147\,483\,648$ – $2\,147\,483\,647$ (znaménkový typ) nebo 0 – 4 294 967 295 (bezznaménkový typ). Pokud bychom implementovali funkci `a()` s použitím 32bitového bezznaménkového celočíselného datového typu, pak by se pro $x \geq 13$ již výsledek nevešel do rozsahu daného typu[3] a došlo by k tzv. přetečení (integer overflow).

---

[1] otázka na zamyšlení: jak rychle zjistit, kolik nul?

[2] Takto velký exponent se již do tohoto typu nevejde.

[3] $13! \doteq 6 \cdot 10^9$

To, co se přesně s číslem stane, závisí na konkrétní bitové reprezentaci čísla v počítači a bitové implementaci operace, která přetečení způsobila (nebo to nemusí být definováno vůbec). Nejjednodušší situace je při sčítání, kdy po překročení horní hranice rozsahu datového typu pokračujeme opět od spodní hranice.[4] Obecně ale dostaneme číslo v rozsahu daného datového typu, které nemá žádnou blízkou spojitost se skutečným výsledkem. Protože stále dostáváme nějaké číslo a výpočet dále pokračuje, jako by se nic nestalo, je přetečení obzvlášť zákeřné a při definici proměnných musíme volit takový typ, aby se do nich vešla všechna čísla, která se v nich mohou v průběhu výpočtu vyskytnout.

Všimněme si také, že v tomto případě přetečení celočíselného typu nastane dříve, než typ s plovoucí desetinnou čárkou začne ztrácet přesnost, a o hodně dříve, než se v něm vyskytne hodnota nekonečno. V jazycích, kde nemáme libovolně velké celočíselné typy jako v Pythonu, se tedy mnohdy vyplatí pro výpočty s hodně velkými celými čísly používat typ s plovoucí desetinnou čárkou, pokud nepotřebujeme znát výsledek zcela přesně.[5] Jednou z alternativ je napsat si vlastní neomezený datový typ, se kterým jsou výpočty ale pomalé a žerou hodně paměti.

Na závěr zmiňme, že pro $x \geq 998$[6] program selže s chybou `RuntimeError: maximum recursion depth exceeded in comparison`. V těle funkcí `a()` i `b()` totiž voláme (pro $n \neq 0$) tu samou funkci s argumetem o jedna menším, ta zase volá sama sebe, ..., až se zavolá funkce s argumentem 0. Toto se nazývá *rekurzivní volání funkce*. Program si při zavolání funkce musí zapamatovat, odkud ji zavolal a kam se má po jejím vykonání vrátit.[7] Všechny tyto informace si ukládá do oblasti paměti zvané zásobník (stack, tady konkrétně call stack), který má omezenou velikost. Při hodně hlubokém rekurzivním volání se pak může stát, že se zásobník zcela naplní a není už kam uložit další iteraci. Tomuto stavu se říká přetečení zásobníku (stack overflow) a jde o kritickou chybu, která nastala i zde.[8] Přetečení zásobníku lze v tomto případě zabránit dvěma způsoby. První spočívá v nastavení větší velikosti zásobníku. Druhý spočívá v přepsání programu tak, aby se v něm nevyskytovaly příliš hluboké rekurze. Například v našem programu lze rekurzi snadno nahradit cyklem, jak vidíme níže.

```
def a_opravena(n):
  faktorial = 1
  for i in range(1,n+1):
    fact *= i
  return faktorial
```

Takto přepsaná funkce by měla fungovat pro libovolně velký argument.

Zmiňme ještě časovou složitost algoritmu. Pro funkce `a()` a `b()` je potřeba pro vstup n danou funkci rekurzivně zavolat $(n+1)$krát a v každé iteraci provést konstantní počet operací (1 násobení a 1 odčítání). Asymptotická časová složitost je tedy $O(n)$. Situace je obdobná i u funkce `a_opravena()`, kde je pro vstup n potřeba provést n iterací cyklu, v každé iteraci pak provést konstantní počet operací. Složitost je tedy také lineární. Pro úplnost dodejme, že složitost celého programu je také lineární v závislosti na velikosti `x`, protože program jen jednou zavolá funkce `a()` a `b()`, a ty jsou lineární.

[4] Odtud název přetečení.

[5] Tento trik se tedy nehodí např. pro kryptografické operace, pro numerické fyzikální výpočty jej ale můžeme použít téměř vždy.

[6] Hodnota se může lišit v závislosti na konkrétním nastavení prostředí.

[7] A vedle toho si musí pamatovat i kontext, např. hodnoty lokálních proměnných ve volajícím místě.

[8] Přesněji zde byla překročena maximální hloubka rekurze, což je hodnota, kterou Python hlídá, aby nenastalo právě přetečení zásobníku.

c) Víceméně přesným přepisem rekurentního postupu ze zadání dostaneme kód níže.

```
import math
pocet_iteraci=100
# inicializace ops. a veps. ctvercem
r = 0.5 # polomer kruznice, s touto hodnotou bude obvod = pi
O = 4*2*r
o = 4*r*math.sqrt(2)
k = 4
# rekurentni vypocet
for i in range(pocet_iteraci):
  print("{} {} {}".format(k,o,O))
  O = (2.*o*O)/(o+O)
  o = math.sqrt(o*O)
  k *= 2
```

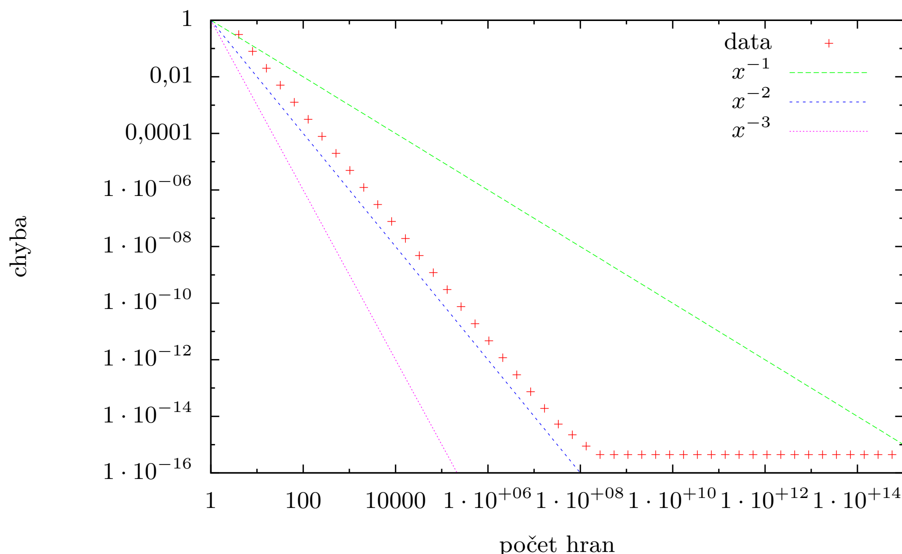
Může být matoucí, jak v těle cyklu používáme pod stejným označením o a O hodnoty z minulého i aktuálního kroku. Vězte, že lze program přepsat i následovně.

```
import math
pocet_iteraci=100
r = 0.5
O_k = 4*2*r
o_k = 4*r*math.sqrt(2)
k = 4
for i in range(pocet_iteraci):
  print("{} {} {}".format(k,o_k,O_k))
  O_2k = (2.*o_k*O_k)/(o_k+O_k)
  o_2k = math.sqrt(o_k*O_2k)
  o_k = o_2k
  O_k = O_2k
  k *= 2
```

Takto ale zbytečně potřebujeme dvě proměnné navíc.

Pokud program spustíme, zjistíme, že nám skutečně vypisuje hodnoty blízké hodnotě π, což je dobrá zpráva. Vypišme si nyní rozdíly vypočtených hodnot od skutečné hodnoty π. Toho docílíme modifikací řádku s výpisem na obrazovku do podoby `print("{} {} {}".format(k,math.pi-o,math.pi-O))`. Vidíme, že se zvětšujícím se k oba rozdíly velmi rychle klesají, pro hodnoty $k \geq 268\,435\,456$ konvergence ustane a rozdíly se zastaví na přibližné hodnotě $4 \cdot 10^{-16}$. Tato hodnota odpovídá naakumulované numerické chybě. Jde ale o chybu v řádu strojové přesnosti, což lze při numerickém výpočtu jednoznačně považovat za úspěch.

Možná jste si všimli, že v jedné iteraci je hodnota $\pi - O$ přesně nulová. To, že je přesně nulová, lze nejspíš považovat za náhodu (stejně to je nula jenom v rámci strojové přesnosti). Náhoda ale není, že jde o nejmenší hodnotu rozdílu, že před touto hodnotou rozdíl klesá a po ní roste či stagnuje, a konečně není náhoda, že se tak děje zrovna pro toto k. Došlo zde totiž ke kombinaci nízké chyby metody, která klesá s rostoucím k, a nízké zaokrouhlovací chyby, která s rostoucím počtem operací, a tedy i s rostoucím k, roste. Tímto jevem se budeme podrobněji zabývat v příštím díle.



Obr. 1: Závislost chyby metody na počtu hran použitého mnohoúhelníka.

Vykresleme si nyní chyby v závislosti na počtu hran k, jak je ukázáno v log-log grafu[9] 1. Vidíme, že body v log-log grafu leží na přímce, což znamená, že chyba klesá polynomiálně. Tuto užitečnou transformaci si hned vysvětlíme. Uvažujme polynomiální závislost

$$y = ax^b\,.$$

Pokud rovnici zlogaritmujeme a označíme $X = \log x$ a $Y = \log y$, dostaneme

$$\log y = \log\left(ax^b\right)$$
$$\log y = \log a + b\log x$$
$$Y = bX + \log a\,,$$

což je rovnice přímky. Ukázali jsme tedy, že v log-log grafu se polynomiální závislosti zobrazují jako přímky, jejichž směrnice odpovídá exponentu závislosti. Poznamenejme, že obdobně lze převést na přímku i exponenciální závislost, jen použijeme graf, který má logaritmickou škálu pouze na svislé ose.

Spolu s daty jsou v grafu 1 vyneseny i závislosti $y = x^{-1}$, $y = x^{-2}$ a $y = x^{-3}$. Vidíme, že data klesají s přibližně stejnou směrnicí jako $y = x^{-2}$,[10] což znamená, že tato metoda je tzv. *kvadratická* v závislosti na počtu hran k.

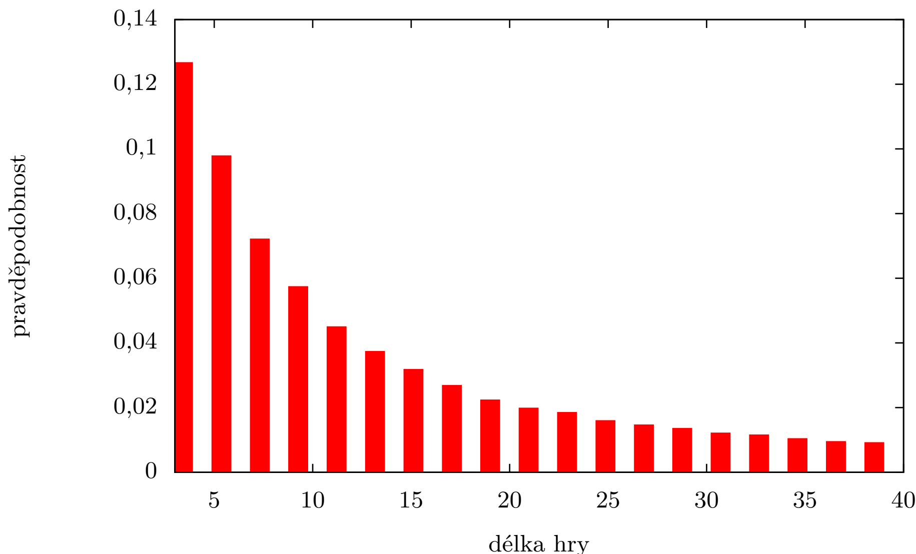
Zdůraznit závislost na k je v tomto případě důležité, neboť běžně se rychlost konvergence udává v závislosti na počtu iterací. Snadným přepočtem zjistíme, že tato rychlost by byla

[9] Log-log graf je graf, který má na obou osách logaritmickou škálu.

[10] Zdůrazňuji, že nám jde o směrnici, ne o přesné proložení dat, neboť závislost může být posunuta o konstantu.

v našem případě exponenciální. V každém případě jde o velmi rychle konvergující metodu, neboť mnoho metod, se kterými se v praxi setkáte, konverguje lineárně nebo pomaleji. Tomuto tvrzení odpovídá fakt, že jsme se k výsledku na úrovni strojové přesnosti dobrali po několika málo iteracích.

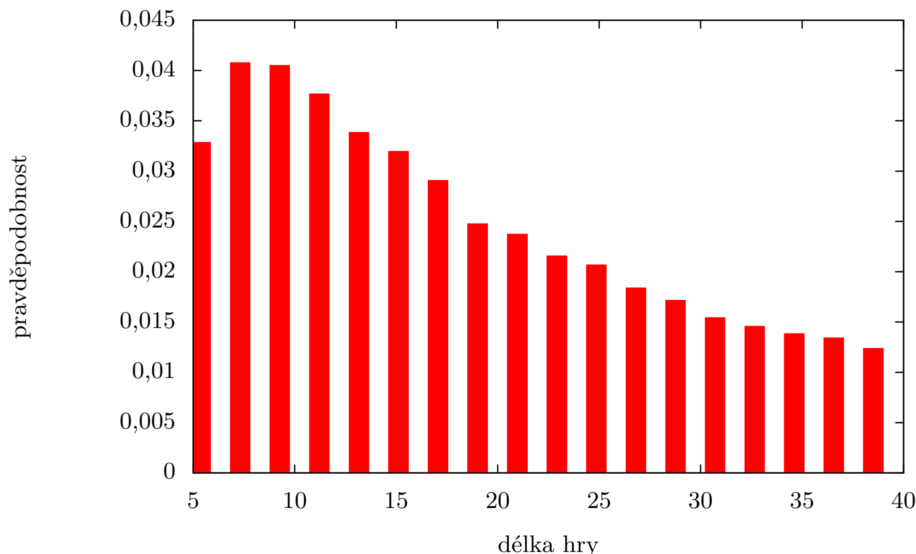
d) 1. Tuto úlohu lze poměrně snadno vyřešit pomocí simulace pro libovolné m. Necháme počítač odehrát velký počet partií a u každé partie zaznamenáme její délku. Ze získaných délek potom vykreslíme normovaný histogram, tj. graf zobrazující pravděpodobnosti ukončení hry po daném počtu hodů. V našem případě jsme použili $m = 3$ a $m = 5$ a celkový počet partií 100 000. Výsledné histogramy jsou na obrázcích 2 a 3. Vidíme, že pro zadané $m = 3$ skončí hra s největší pravděpodobností již po třech hodech, což je také nejmenší počet hodů potřebných k ukončení hry. Oproti tomu pro $m = 5$ je nejpravděpodobnější délkou hry 7 hodů – to již není triviálně výsledek. Také si všimneme, že hra musí skončit vždy po lichém počtu hodů. Protože po každém kroku Mirek buď jedno tričko získá, nebo jedno ztratí, má celkový počet jeho triček vždy opačnou paritu než v předchozím kroku. Jestliže začíná s lichým počtem triček, dostane se na nulu (sudé číslo) jedině po lichém počtu hodů.



Obr. 2: Histogram pravděpodobnosti ukončení hry po určitém počtu hodů. Počet Mirkových triček na začátku hry je 3.

Řešení této úlohy bez pomoci počítače již není tak triviální. Omezíme se pouze na zadané $m = 3$ a pokusíme se o kombinatorický odhad – pokud si s kombinatorikou netykáte, můžete zbytek řešení přeskočit.

Nejprve si uvědomme následující: Pravděpodobnost, že dojde k realizaci jedné konkrétní hry o délce n hodů, je $1/2^n$. Realizací hry zde myslíme unikátní sekvenci zaznamenávající



Obr. 3: Histogram pravděpodobnosti ukončení hry po určitém počtu hodů. Počet Mirkových triček na začátku hry je 5.

počet Mirkových triček v každém hodu, která končí první nulou. Např.

$$3 \to 2 \to 3 \to 2 \to 1 \to 0\,. \tag{1}$$

V této konkrétní hře Mirek získal tričko v druhém hodu, v ostatních pouze ztrácel. Jelikož pravděpodobnost hození panny, nebo orla, je 1/2, násobíme za každou šipku tímto faktorem – kdyby byla mince upravená jako v další úloze, výsledek $1/2^n$ by neplatil. Nyní předpokládejme, že Mirek může mít i záporný počet triček. Poté si všechny možné realizace hry můžeme zakreslit pomocí orientovaného grafu jako na obrázku 4. Vrcholy grafu jsou označeny čísly, která říkají, kolika možnými způsoby je možné se dostat do toho daného stavu hry (stavem rozumíme počet triček a počet odehraných hodů, viz osy grafu v obrázku 4). Hodnoty ve vrcholech odpovídají číslům z Pascalova trojúhelníku. Zaměřme se nyní na vrcholy odpovídající stavu, kdy má Mirek jen jedno tričko. Pokud opět zakážeme záporná trička, tak můžeme tvrdit, že hodnoty v těchto vrcholech budou menší nebo rovny těm vyobrazeným, protože přijdeme o některé hrany grafu (pro hru o více než třech hodech jde o ostrou nerovnost). A pokud budeme chtít, aby hra v následujícím kroku skončila, tak existuje jen jedna hrana vedoucí k tomuto cíli. A jelikož $k$té číslo na $n$tém řádku Pascalova trojúhelníku je určeno binomickým koeficientem

$$\binom{n}{k}\,,$$

bude hodnota ve výše zmíněných vrcholech

$$\binom{n-1}{(n-3)/2}$$

pro délku hry n. Dále budeme uvažovat následovně: jestliže ukážeme, že pro $n+2$ je hodnota ve zkoumaných vrcholech menší než čtyřnásobek hodnoty pro n, a to pro každé $n \geq 3$, pak nejpravděpodobnější délkou hry musí být $n = 3$. Čtyřnásobek zde vystupuje proto, že horní odhad pravděpodobnosti získáme vynásobením hodnoty kombinačního čísla pro n pravděpodobností $1/2^n$, s každým prodloužením hry o dva hody se tedy ještě sníží pravděpodobnost jednotlivých realizací hry násobným faktorem $1/2^2$. Protože skutečná pravděpodobnost je pro $n = 3$ stejná jako tento horní odhad, dokážeme tím, že skutečná pravděpodobnost pro obecné $n > 3$ je taky menší než pro $n = 3$.
Pokud ještě provedeme substituci $n = r+3$ (r je sudé), tak se snažíme o důkaz nerovnosti

$$\frac{\binom{r+4}{(r+2)/2}}{\binom{r+2}{r/2}} \leq 4 \quad \forall r \geq 0\,.$$

Binomické koeficienty rozepíšeme do tvaru

$$\frac{(r+4)!(r/2)!(r/2+2)!}{(r/2+1)!(r/2+3)!(r+2)!} = \frac{(r+3)(r+4)}{(r/2+1)(r/2+3)}$$

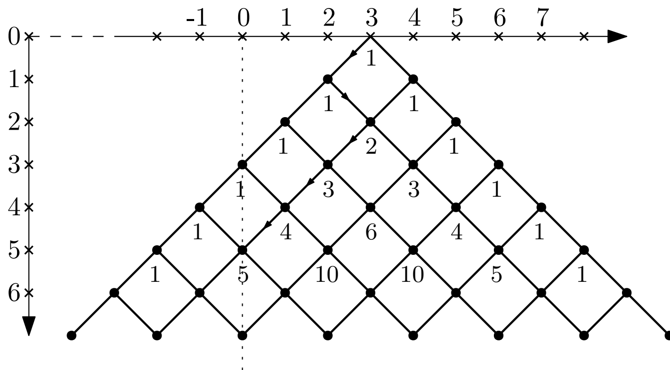
dosazením do nerovnosti a odstraněním jmenovatele získáme

$$r^2 + 7r + 12 \leq r^2 + 8r + 12\,,$$

z čehož už ihned plyne $r \geq 0$ □.

2. Tato úloha byla velice přímočará. Drobný problém při jejím řešení vám mohla způsobit skutečnost, že hra může skončit po libovolném počtu kroků, takže některé partie mohou být poměrně dlouhé – pokud jste tedy ve svém kódu nastavili velký počet opakování, je možné, že jste si na výsledek museli počkat.
My jsme volili délky her 100, 1 000, 10 000 a 100 000. Pro tyto hodnoty jsme postupně dostali pravděpodobnost výhry 0,37, 0,342, 0,3366 a 0,33293. Ačkoli při každém běhu programu dostaneme trochu jiný výsledek, lze z výsledků simulace odhadnout, že přesná hodnota hledané pravděpodobnosti bude 1/3. Kód použitý k řešení úlohy naleznete na našich webových stránkách (psán v Pythonu 3).
K vyřešení bonusu použijeme Větu o úplné pravděpodobnosti, která je uvedena v prvním dílu seriálu. Značme $a = m + l$ a dále si zaveďme pravděpodobnost, že Mirek vyhraje ze stavu, kdy má právě m triček, označme ji $P_\mathrm{M}(m)$. Zřejmě platí $P_\mathrm{M}(0) = 0$ a $P_\mathrm{M}(a) = 1$, jde jen o symbolickou formulaci podmínek výhry a prohry (matematicky bychom řekli, že jde o okrajové podmínky). Věta o úplné pravděpodobnosti nyní říká, že pravděpodobnost výhry $P_\mathrm{M}(m)$ je rovna součtu pravděpodobnosti výhry za podmínky, že padne panna (krát pravděpodobnost, že padne panna) s pravděpodobností výhry za podmínky, že padne orel (krát pravděpodobnost, že padne orel). Toto tvrzení zapíšeme ve tvaru diferenční rovnice

$$P_\mathrm{M}(m) = \mu P_\mathrm{M}(m+1) + \lambda P_\mathrm{M}(m-1)\,,$$



Obr. 4: Graf, z nějž můžeme vyčíst pravděpodobnosti jednotlivých realizací hry za předpokladu, že je povolen záporný počet triček. Vodorovná osa zobrazuje aktuální počet triček, svislá osa počet odehraných hodů. Graf je zkonstruován pro počáteční hodnotu 3 trička. Malými šipkami je v grafu naznačen příklad hry (1).

kde $\mu = 1/2$, $\lambda = 1/2$ označují pravděpodobnosti, že padne panna, resp. orel. Řešení této rovnice snadno uhodneme, když si dosadíme za μ a λ. Poté dostaneme

$$P_{\mathrm{M}}(m) = \frac{P_{\mathrm{M}}(m+1) + P_{\mathrm{M}}(m-1)}{2}\,,$$

Jinými slovy: pravděpodobnost, že Mirek vyhraje s m tričky, že dána aritmetickým průměrem pravděpodobností, že vyhraje s $m+1$ tričky a s $m-1$. Funkce $P_{\mathrm{M}}(m)$ by tedy mohla ležet na přímce, což zapíšeme formálně jako

$$P_{\mathrm{M}}(m) = A + Bm\,,$$

přičemž neznámé koeficienty A, B určíme podmínek výhry/prohry, tj. dosazením $m = 0$ a $m = a$ a vyřešením soustavy dvou rovnic o dvou neznámých. Získáme $A = 0$, $B = 1/a$. Takže

$$P_{\mathrm{M}}(n) = \frac{n}{a} = \frac{m}{m+l}$$

a pro zadané hodnoty $m = 10$, $l = 20$ dostaneme pravděpodobnost výhry $P_{\mathrm{M}}(10) = 1/3$, což je v souladu s naším předpokladem založeným na výsledcích simulace.

3. Úpravou jedné hodnoty v kódu, tedy přepsáním pravděpodobnosti padnutí panny z 1/2 na 5/9, jsme dostali výsledek 0,89276 (pro 100 000 her). Na této hodnotě by nás mělo zaujmout především to, že ačkoli se pravděpodobnost padnutí panny zvýšila pouze o 1/18, Mirkova šance na výhru markantně vzrostla. Tohoto faktu využívají např. kasina v ruletě, kdy šance na výhru při sázce na černou (nebo červenou) je jen o fous menší než 1/2, kvůli zelené nule. Proto, i kdyby mělo kasino méně peněz než hráč (ale řádově více, než je v jedné sázce), tak hráče přesto s velmi vysokou pravděpodobností zruinuje.
K řešení bonusové části zde využijeme jeden matematiky hojně používaný trik, se kterým se ale možná ještě někteří z vás nesetkali. Máme opět rovnici

$$P_{\mathrm{M}}(m) = \mu P_{\mathrm{M}}(m+1) + \lambda P_{\mathrm{M}}(m-1)\,,$$

ale nyní jsou μ a λ různá, existuje mezi nimi pouze vztah $\lambda = 1 - \mu$. Tento typ rovnic se řeší pomocí metody charakteristického polynomu. Budeme předpokládat obecné exponenciální řešení ve tvaru α^m a dosazením do rovnice a krácením α^{m-1} získáme kvadratickou rovnici

$$\mu\alpha^2 - \alpha + \lambda = 0\,,$$

jejíž levá strana se nazývá charakteristický polynom. Řešením této rovnice jsou $\alpha_1 = 1$, $\alpha_2 = \lambda/\mu$. Obecné řešení diferenční rovnice získáme jako lineární kombinaci[11]

$$A\alpha_1^m + B\alpha_2^m\,,$$

tedy

$$P_\mathrm{M}(m) = A + B\left(\frac{\lambda}{\mu}\right)^m\,.$$

Podobně jako výše získáme pomocí okrajových podmínek řešení

$$P_\mathrm{M}(m) = \frac{1 - \left(\frac{1-\mu}{\mu}\right)^m}{1 - \left(\frac{1-\mu}{\mu}\right)^{(m+l)}}\,.$$

Dosazením $m = 10$, $l = 20$, $\mu = 5/9$ a $\lambda = 4/9$ získáme číselný výsledek $P_\mathrm{M}(10) \doteq 0{,}89373$.

e) 1. Lineární kongruenční generátor, který jsme použili k řešení úlohy, najdete na našem webu. Pro ty z vás, co programovat ještě neumějí, zde uvedeme, jak takový generátor sestrojit v Excelu.

Do buňky A1 vepíšeme seed, do B1 až D1 postupně hodnoty a, c, m. Do buňky A2 poté vepíšeme vzorec

```
=MOD(($B$1*A1 + $C$1);$D$1)
```

a „vytáhneme" sloupec A tak daleko, jak potřebujeme. Toť vše.

Pomocí generátoru sepsaného v Pythonu jsme z náhodného seedu x_0 (získaného ze systémového času) vygenerovali 10^6 pseudonáhodných čísel a vypočetli průměrnou hodnotu $\bar{x} \doteq 0{,}499596$ a rozptyl $\sigma^2 \doteq 0{,}083209$ (směrodatná odchylka je potom $\sigma \doteq 0{,}28846$).

Srovnejme nyní výsledky simulace s přesnými výrazy pro střední hodnotu a rozptyl uvedenými v seriálu. Střední hodnota je

$$\frac{0 + \left(2^{31} - 1\right)}{2}\,,$$

po normalizaci $N = 2^{31}$

$$\bar{x} = \frac{0 + \left(2^{31} - 1\right)}{2 \cdot 2^{31}} \doteq 0.49999999977\,.$$

Když do vztahu pro rozptyl

$$\sigma^2 = \sum_i (x_i - \mu)^2 p_i$$

[11] pokud se vám nezdá, že pro $\lambda = \mu$ jsme dosazovali lineární funkci a tady exponenciální, vězte, že případ se stejnými kořeny je speciální – právě díky tomu, že z nich neuděláme lineární kombinaci

dosadíme normalizované hodnoty, vidíme, že normalizovat rozptyl znamená dělit N^2, takže pro rovnoměrné rozdělení máme

$$\sigma^2 = \frac{(2^{31}-1)^2}{12 \cdot 2^{62}} \doteq 0.08333333325572312\,.$$

Ověřili jsme tedy, že vygenerované rozdělení má podobnou střední hodnotu a rozptyl jako rovnoměrné rozdělení. Ještě poznamenejme, že střední hodnota a rozptyl obecně nestačí pro jednoznačné určení pravděpodobnostního rozdělení (museli bychom znát všechny jeho momenty). A už vůbec nám shoda v těchto dvou veličinách nezaručuje, že generátor má dobré náhodné vlastnosti. Pokud jste si nechali vypsat pseudonáhodná čísla na obrazovku, tak jste si určitě všimli, že jsou všechna lichá, což nám indikuje, že se nejedná o dobrý generátor. Více viz následující úloha.

2. Zásadním krokem v této úloze je uvědomit si, že číslo 65 539 lze zapsat ve tvaru $2^{16}+3$. Poté můžeme libovolné číslo v sekvenci napsat ve tvaru

$$\begin{aligned} x_{i+2} &= \left(\left(2^{16}+3\right)\cdot x_{i+1}\right) \bmod 2^{31} = \left(\left(2^{16}+3\right)\cdot \left(\left(2^{16}+3\right)x_i\right) \bmod 2^{31}\right) \bmod 2^{31} \\ &= \left(2^{16}+3\right)^2 x_i \bmod 2^{31}\,. \end{aligned}$$

Postupnými úpravami dostaneme, že modulo 2^{31} platí

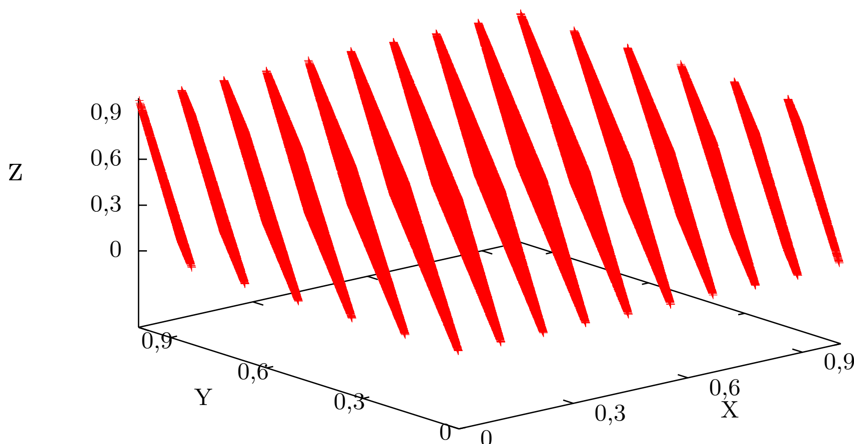
$$\begin{aligned} x_{i+2} &= \left(2^{32}+6\cdot 2^{16}+9\right)x_i = \left(6\cdot 2^{16}+9\right)x_i \\ &= \left(6\left(2^{16}+3\right)-9\right)x_i = 6x_{i+1}-9x_i\,, \end{aligned}$$

kde jsme v druhém kroku využili $2^{32} \bmod 2^{31} = 0$. Pokud nyní budeme tři následující čísla v sekvenci chápat jako kartézské souřadnice $x = x_i$, $y = x_{i+1}$, $z = x_{i+2}$, pak je odvozený rekurentní vztah rovnicí roviny

$$z = 6y - 9x\,.$$

V trojrozměrném prostoru tedy budou všechna vygenerovaná pseudonáhodná čísla ležet v rovině. Ještě si můžeme rozmyslet, kolikrát se v boxu $1\times 1\times 1$ rovina díky modulární aritmetice „zlomí“. V průmětech do rovin yz ($x=0$) a xz ($y=0$) dostaneme rovnice přímek $z=6y$ a $z=-9x$. Stěna boxu ležící na ose y bude tedy protnuta šestkrát a stěna ležící na ose x devětkrát. Celkem tedy bude v boxu patnáct rovin. To si koneckonců můžete potvrdit na řešení bonusové úlohy, které vidíte na obrázku 5.

Možná vás napadlo, že jelikož sekvence generovaná libovolným LCG (lineárním kongruenčním generátorem) má konečnou periodu, tak vždy bude možné najít nějakou korelaci mezi následujícími členy, která významně naruší náhodnost. Tzv. Marsagliův teorém říká, že pokud čísla generovaná lineárním kongruenčním generátorem vykreslíme jako body $n$dimenzionálního prostoru, tak budou spadat do nejvýše $(n!m)^{1/n}$ nadrovin o dimenzi $n-1$. Proto, zatímco v nízkých dimenzích jsou LCG obvykle spolehlivé (i zde používaný generátor, známý pod názvem RANDU, se chová dobře ve dvou dimenzích, viz obr. 6), ve vyšších dimenzích postupně selhávají.



Obr. 5: 1 000 000 pseudonáhodných čísel vykreslených ve 3D. Jistě byste nechtěli, aby takovýto generátor používala v bezpečnostních systémech vaše banka.

Poznámky k došlým řešením

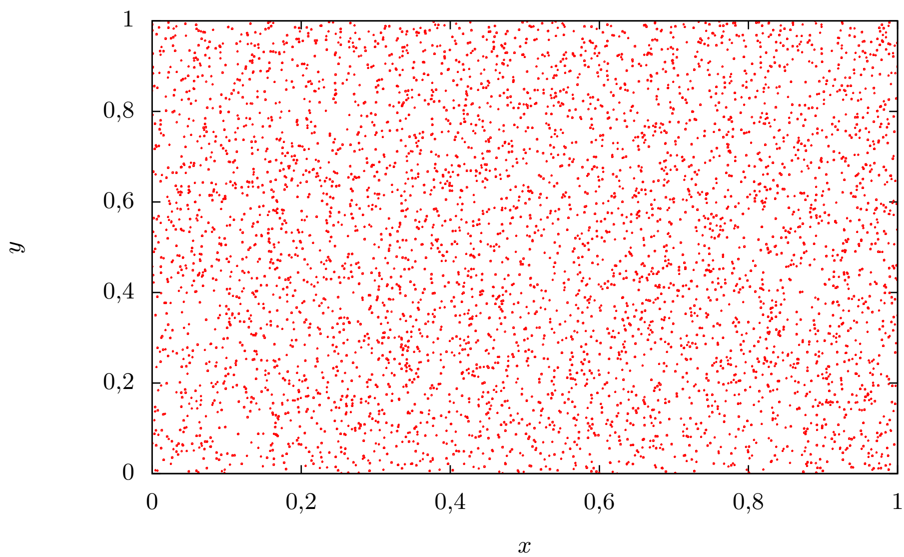
Oceňujeme řešení Miroslava Hrabala a Viktora Fukaly, kteří si všimli, že operace nad dvěma libovolně velkými celými čísly nemůže být provedena v konstantním čase, tudíž v bodu b nebude mít pro velká n funkce `a(n)` časovou složitost $O(n)$, ale o něco horší.

Lukáš Timko
`lukast@fykos.cz`

Miroslav Hanzelka
`mirek@fykos.cz`

Fyzikální korespondenční seminář je organizován studenty MFF UK. Je zastřešen Oddělením pro vnější vztahy a propagaci MFF UK a podporován Ústavem teoretické fyziky MFF UK, jeho zaměstnanci a Jednotou českých matematiků a fyziků.

Toto dílo je šířeno pod licencí Creative Commons Attribution-Share Alike 3.0 Unported.
Pro zobrazení kopie této licence navštivte `http://creativecommons.org/licenses/by-sa/3.0/`.



Obr. 6: 10 000 pseudonáhodných čísel vykreslených ve 2D. V této dimenzi se již polohy bodů nezdají být předvídatelné.