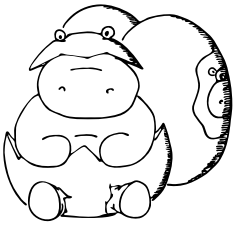
Korespondenční Seminář z Programování

33. ročník **KSP** **Březen 2021**

Milí řešitelé, řešitelky a řešitelčata!



Už je to tak, 33. ročník KSP pokračuje dál a čtvrtá série KSP-H je tu.

Naleznete zde **4 normální úlohy**, **bonusovou těžší úlohu** a pak také pokračování **seriálu o počítačové grafice**. Všechny díly seriálu můžete **odevzdávat v průběhu celého roku**, takže vůbec nevadí, že jste třeba první díl nestihli. Detaily naleznete u zadání seriálu. Připomínáme, že oproti loňskému ročníku se do výsledkové listiny započítávají **všechny úlohy mimo bonusové** a body se již nepřepočítávají podle počtu vyřešených sérií.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Termín série: **neděle 11. dubna 2021 ve 32:00 (tedy další ráno v 8:00)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky

Praktická open-data úloha

Úloha, u které doporučujeme začíst se do kuchařky

Seriálová úloha

Odměna série: Každému, kdo z **bonusové úlohy 33-4-X1 získá alespoň dva body**, pošleme sladkou odměnu.

Čtvrtá série třicátého třetího ročníku KSP

33-4-1 Ohňostroj — 12 bodů

V řadě za sebou je N odpalovacích zařízení na ohňostroj. V každém je odpalovací mikropočítač, který je propojený s počítači sousedních odpalovacích zařízení.

Rádi bychom odpálili ohňostroj ze všech stanic současně. To ale není tak jednoduché, protože komunikace mezi sousedními stanicemi je moc pomalá. Kdyby si tedy jen předávali informaci „Odpal!“, tak by nevznikl hezký správně načasovaný ohňostroj, ale vznikla by mexická vlna. Vašim úkolem bude tento problém vyřešit.

Vymyslete program, který se nahraje do každé odpalovací stanice a nechá se běžet. Po nějakém čase obsluha zmáčkne tlačítko na libovolné ze stanic a je na vašich programech, aby poté v jeden moment odpálily všechny stanice.

Je tu ovšem jeden problém. V každé stanici máme jen omezené množství paměti na konečný počet bitů.

Pro potřeby naší úlohy si tedy můžete alokovat bitů libovolný počet, ale tento počet musí být konstantní, tedy nemůže záviset například na N. Díky tomu si ani nemůžete uložit hodnotu čísla N do každé paměti, protože ta zabere $\log N$ bitů, což již není konstanta.

Algoritmus se vyhodnocuje v ticích, které nastávají současně na všech odpalovacích stanicích. V každém tiku se na každém zařízení spustí vámi vymyšlená funkce. Ta může upravovat svoji paměť a navíc může komunikovat se sousedními mikroprocesory tím, že čte jejich paměť. Umí poznat levého a pravého souseda a také jestli je soused připojen, nebo jestli se stanice nachází na okraji řady a žádného souseda již nemá. Přesněji řečeno čte paměť sousedů ve stavu, v jakém byla na začátku tiku. Krom toho může požádat o odpálení a také číst stav tlačítka. Máte zaručeno, že tlačítko bude stisknuto pouze na jednom zařízení a to po dobu jednoho tiku.

Formálně vzato, každé zařízení se tedy chová jako konečný automat.

Za časovou složitost algoritmu považujeme počet tiků od zmáčknutí tlačítka po odpálení ohňostroje, přitom nás zajímá jen její asymptotický odhad vzhledem k N.

Část bodů dostanete i za řešení, které fungují jen pro nějaká speciální N, ovšem musí jich být nekonečně mnoho. Tedy například násobky 42, nebo třeba každé druhé prvočíslo.

33-4-2 Firewall — 10 bodů

Tato stránka je pro Váš region nedostupná. Tato hra není povolená pro Vaší zemi. Tento kód je platný jen v některých zemích. K tomuto serveru se nemůžeš připojit, protože málo uctíváš hroší bohy ...

S některými z těchto hlášek jste se mohli již setkat, např. některé filmy na Netflixu jsou uvedeny jen pro určité trhy, ať to má jakýkoliv důvod, nebo nějaké Steam klíče jsou geograficky zamčené.

Tyto geografické zákazy se dějí z větší míry pomocí IP adres, u kterých se zjišťuje, jestli je to jedna z povolených adres (*whitelist*), nebo jedna ze zakázaných adres (*blacklist*). Samozřejmě to nemusí být jediné kritérium.

V této úloze si vyzkoušíte napsat filtr na IP adresy, konkrétně na jejich starší variantu IPv4. Budete dostávat tři typy dotazů:

- `A` (allow) – povol danou podsíť IP adres
- `B` (ban) – zakaž danou podsíť IP adres
- `Q` (query) – dotaz: Je tato IP adresa povolená?

Podsítě budou zadány v CIDR formátu. Pokud tento formát znáte, můžete následujících pět odstavců přeskočit.

Na začátek si ujasníme pohled na IPv4 adresy. IPv4 je v podstatě 32bitové číslo. Zpravidla se zapisuje rozdělené do čtyř osmibitových čísel oddělených pro přehlednost tečkami, například 192.168.255.4 (po přepsání do dvojkové soustavy `11000000.10101000.11111111.00000100`).

Počítačové sítě jsou často děleny do podsítí, ve kterých mají všechny adresy stejnou počáteční část (nazývanou *prefix sítě*). Často se podsítě zapisují v CIDR formátu, který vypadá jako IP adresa s číslem za lomítkem. Toto číslo, označme ho K, udává délku prefixu v bitech. Pro IPv4 je v rozmezí 0 až 32. Před lomítkem je pak IP adresa, jejíž prvních K bitů je *prefix sítě* a zbytek je doplněn nulami (aby měla délku 32 bitů).

Například zápis 192.168.3.0/24 označuje podsíť s 256 IP adresami ve tvaru 192.168.3.xxx (což ve dvojkové soustavě vypadá jako `11000000.10101000.00000011.xxxxxxxx`).

Pro prefixy, které jsou násobky 8, je to celkem intuitivní, ale co značí třeba zápis 192.168.2.128/26? Ten označuje podsíť 64 IP adres začínající adresou 192.168.2.128 a končící adresou 192.168.2.191, neboli prvních 26 bitů adresy je pevně určeno prefixem a variabilita je jen v posledních 6 bitech. Zapsáno ve dvojkové soustavě to vypadá takto: `11000000.10101000.00000010.10xxxxxx`.

Abyste si nemuseli vše ručně počítat na papír, tak doporučujeme nástroj `ipcalc`, existuje i jeho online verze.[1]

Na začátku jsou všechny IP adresy zakázané. Jednotlivé operace (povolení a zakázání podsítě) se navzájem přepisují. Například pokud je povolena podsíť 192.168.0.0/16, ale později je zakázána podsíť 192.168.1.0/24, tak IP adresa 192.168.1.1 je po těchto operacích zakázána.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N udávající počet dotazů. Na dalších N řádcích budou následovat jednotlivé dotazy. Dotazy začínají písmenem `A`, `B` nebo `Q`. U dotazů `A` a `B` bude následovat rozsah IPv4 adres v CIDR formátu. U dotazu `Q` bude následovat jen samostatná IP adresa.

Formát výstupu: Pro každý dotaz typu `Q` vypište na samostatný řádek `A`, nebo `N` jako ANO, nebo NE, je-li IP adresa povolena, či ne.

Ukázkový vstup:	*Ukázkový výstup:*
`7`	`N`
`Q 192.168.1.1`	`A`
`A 192.168.0.0/16`	`N`
`Q 192.168.1.1`	`A`
`B 192.168.1.0/24`	
`Q 192.168.1.1`	
`A 192.0.0.0/8`	
`Q 192.168.1.1`	

Příklad: Na začátku nejsou povoleny žádné adresy a první dotaz se tak vyhodnotí na `N`. Potom jsme povolili podsíť 192.168.0.0/16, tudíž IP adresa 192.168.1.1 je povolena. Poté jsme zakázali podsíť 192.168.1.0/24 a IP adresa 192.168.1.1 je znovu zablokována. Nakonec je povolena podsíť 192.0.0.0/8 a IP adresa 192.168.1.1 je povolena.

[1] `http://jodies.de/ipcalc`

33-4-3 Mraveniště na kopci — 11 bodů

Kevinova babička má za chalupou prazvláštní kopec. Na jeho vrcholu je veliké mraveniště, kde vládne chytrá mravenčí královna. K té se jednoho dne doneslo, že na úpatí kopce se děje něco divného, a tak se rozhodla vyslat své mravenčí dělníky na průzkum.

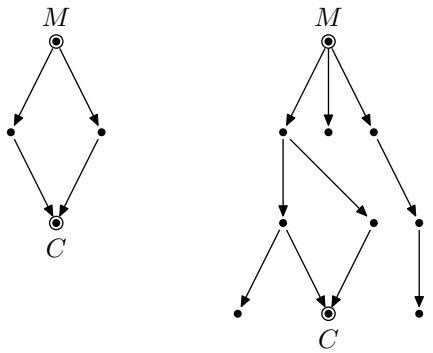
Vyslat mravenčí dělníky na průzkum ale není tak jednoduché – když nemají vytyčenou pachovou stopu z feromonů, tak chodí docela náhodně. Přesněji když přijde mravenčí dělník na křižovatku, ze které vychází N cest, tak si hodí spravedlivou N-stěnnou kostkou (základní výbava každého mravenčího dělníka) a se stejnou pravděpodobností se vydá po jakékoliv z cest dál.

Mravenčí královna alespoň své dělníky instruovala, aby chodili vždy jen z kopce dolů (a tudíž se nemohou zamotat do žádného cyklu) – když mravenec přijde na křižovatku, tak si náhodně vybere jen z cest vedoucích dolů. Celou situaci si tak můžeme představit jako orientovaný graf, kde každý vrchol má nějakou výšku a každá hrana vede z nějakého vrcholu do vrcholu, který je ostře níž. Ještě dodejme, že když mravenec dojde do *listu* (vrcholu, ze kterého již nepokračuje žádná cesta dolů), tak se zde zastaví a zůstane tu.

Na vrcholu kopce je vrchol reprezentující mraveniště a někde u úpatí kopce je list, který královnu zajímá a do kterého by chtěla dostat alespoň K průzkumníků. Vaším úkolem je vymyslet algoritmus, který pro zadaný graf kopce a pro číslo K zjistí, kolik nejméně mravenců musí královna z mraveniště vyslat, aby do označeného listu došlo ve střední hodnotě alespoň K z nich.

Pokud nevíte, co znamená „ve střední hodnotě“, tak se začtěte do přiložené kuchařky.

Příklad: Pro graf vlevo dojdou všichni mravenci vyslaní z mraveniště M do cíle C, takže stačí vyslat právě K mravenců. Pro pravý graf je nutné vyslat $4K$ mravenců, aby jich ve střední hodnotě do cíle došlo K.



Lehčí varianta (za 6 bodů): Vyřešte úlohu pro grafy, které mají podobu stromu s mraveništěm v jeho kořeni.

33-4-4 Prohledávání KSP webu — 12 bodů

Petr si chtěl něco vyhledat na webu KSP, když tu zjistil, že jeho oblíbený globální vyhledávač nefunguje. Samotné stránky KSPčka ale fungovaly a když se ani po pár minutách situace s vyhledávačem nezlepšila, tak se rozhodl napsat si pro web KSP vyhledávač vlastní ...

Toto bude v této méně tradiční open-datové úloze i váš úkol. Budete pro zadaná slova vracet URL stránek na webu KSP,

na kterých se slova nachází. K tomu se bude pravděpodobně hodit znalost, jak si z vašeho programu stáhnout přes HTTP obsah zadaného URL. Ukázky stažení jedné stránky v několika jazycích naleznete ve webové verzi zadání, můžete je použít jako základ vašeho řešení.

Také se vám bude hodit základní znalost HTML.[2] Ve zbytku zadání budeme předpokládat, že víte, co je to *HTML tag* a jak se zapisuje.

Vaším úkolem bude hledat v obsahu zadání, řešení a komentářů všech sérií[3] obou kategorií 0. až 32. ročníku KSP (tedy všechny mimo letošního ročníku). Na webu vás tedy budou zajímat některé části podstromů `/h/ulohy/...` a `/z/ulohy/...`. Obsahem vždy myslíme vnitřek elementu `<div id='content'>` na dané stránce.

Hledat budete *celá slova*, tedy maximální podřetězce tvořené alfanumerickými znaky (tedy písmeny a čísly, včetně písmen s diakritikou) a podtržítky. Pokud bude hledané slovo jen částí slova na stránce, tak se nejedná o shodu (například při hledání slova `nasyta` není slovo `nenasyta` shoda, protože se nejedná o shodu na celém slově). Velká a malá písmena nerozlišujte.

Technické detaily

Při práci s diakritikou si dejte pozor na to, že web KSP servíruje stránky v kódování UTF-8, takže jeden znak nemusí být vždy jeden bajt. Některé programovací jazyky před vámi problém s kódováním skryjí (například Python), v jiných ho budete muset vyřešit ručně.

HTML tagy, komentáře a entity považujeme pro jednoduchost za oddělovače slov. Například řetězec `do<b>psat</b>` tedy představuje dvě slova `do` a `psat`. Podobně řetězec `ne&hellip;nebo ano` obsahuje tři slova `ne`, `nebo` a `ano`.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N udávající počet vyhledávacích dotazů. Na dalších N řádcích dostanete slova (alfanumerické řetězce v UTF-8 bez mezer), která máte za úkol nalézt na KSP webu.

Formát výstupu: Pro každé slovo ze vstupu vypište na samostatný řádek cestu ke stránce na KSP webu, na které se dané slovo nachází (jen cestu bez `http://ksp.mff.cuni.cz`). Při nalezení slova na více stránkách vypište jednu libovolnou z nich. Slibujeme, že každé slovo na vstupu lze na webu nalézt na alespoň jedné stránce.

Ukázkový vstup:	*Ukázkový výstup:*
`3`	`/h/ulohy/24/zadani3.html`
`anekdota`	`/h/ulohy/20/zadani3.html`
`cestář`	`/h/ulohy/25/zadani4.html`
`berňák`	

Pokud byste narazili na nějaké technické problémy při implementaci, ozvěte se nám na našem Discordu, zkusíme vám poradit.

33-4-X1 Koření a recepty — 10 bodů

Chtěli byste si otevřít restauraci a přemýšlíte o tom, jaká jídla připravovat. Znáte recepty $R_1, \ldots R_n$, na jejichž přípravu potřebujete některá z koření $K_1, \ldots, K_m$. Nákup koření K_i stojí cenu k_i, po pořízení ho máte k dispozici libovolné množství. Recept R_j můžete připravit a utržíte za něj částku r_j, pokud máte všechna potřebná koření.

Vymyslete algoritmus, jenž pro zadané ceny koření, výdělky receptů a bipartitní graf závislostí receptů na koření stanoví, která koření pořídit, abychom na přípravě receptů vydělali co nejvíce. Chceme tedy maximalizovat rozdíl tržeb (součet výdělků receptů, které umíte uvařit) a nákladů (součet ceny koření, která jste koupili).

33-4-S Raytracing — 15 bodů

Toto je seriálová úloha, která navazuje na podobné úlohy v minulých sériích. Pokud jste předchozí díly seriálu neřešili, pro pochopení tohoto dílu je dobré si je nejméně přečíst. A pokud si chcete úlohy z minulých dílů také naprogramovat, stále za ně můžete získat polovinu bodů.

Nastal čas vydat se v našich grafických toulkách do hlubokého světa třetí dimenze. Dnešním tématem je *raytracing* a vykreslování prostorových scén.

Raytracing je způsob vykreslování, ve kterém se sledují trasy nějakých paprsků. Typicky se pro každý pixel obrázku vystřelí do scény jeden parsek (jak, to záleží na kameře, kterou simulujeme), sleduje se jeho dráha a zjistí se, kde dojde ke kolizi se scénou. Poté se v tomto místě spočítá osvětlení, které po tomto paprsku doputovalo k pozorovateli, a podle něj se zabarví pixel obrázku.

Z místa kolize můžeme vystřelit druhý paprsek směrem ke světlu a sledovat, jestli do něj dorazí nebo ne. Pokud nedorazí, mezi bodem kolize a zdrojem světla se nachází nějaká překážka a kolize tedy bude ve stínu. Stejně tak můžeme z bodu kolize vystřelit paprsek směrem, kam by se původní odrazil, a tím spočítat zrcadlové odrazy. Pokud chceme vícenásobné zrcadlové odrazy, můžeme tento postup rekurzivně opakovat.

Napíšeme si vlastní jednoduchý raytracer. Jeho nejdůležitější částí je počítání kolizí paprsků se scénou. Pro klasické scény ze složitých polygonových objektů je to velmi výpočetně náročné, a proto se raytracing ve hrách začíná ve velkém využívat až nyní s příchodem specializovaného hardwaru. My ale tento problém můžeme obejít tím, že použijeme nějakou jednodušší reprezentaci scény.

Protože budeme v tomto dílu hodně pracovat s vektory, připomínáme náš úvod do vektorů.[4]

V tomto díle na sebe hodně úkolů postupně navazuje, takže jestli chcete, můžete odevzdat více úkolů dohromady jako jeden soubor s kódem shaderu. Všechny zdrojáky (jako vždy) zabalte do zip souboru.

Signed distance function

Signed distance function, česky „znaménková vzdálenostní funkce“, zkráceně *SDF*, je způsob reprezentace scény pomocí funkce, která pro každý bod v prostoru vrátí vzdálenost k nejbližšímu povrchu. Navíc, pokud se bod nachází uvnitř nějakého objektu, tak je tato vzdálenost záporná a její absolutní velikost udává vzdálenost k nejbližšímu bodu mimo objekt.

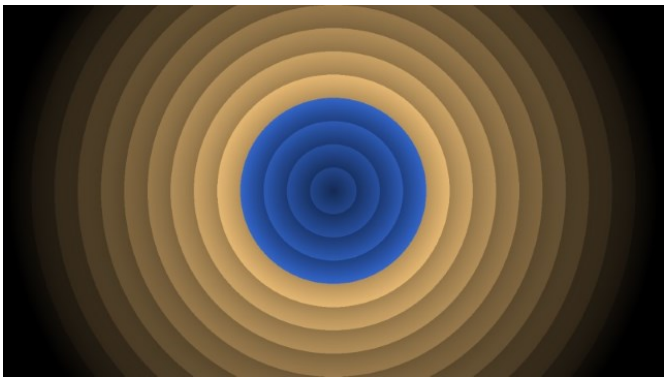
Jak si takovou funkci pořídit? Například pro kouli nebo kruh je to snadné: spočítáme vzdálenost bodu od středu

[2] `https://cs.wikipedia.org/wiki/Hypertext_Markup_Language`

[3] speciální nultou sérii 24. ročníku neuvažujeme

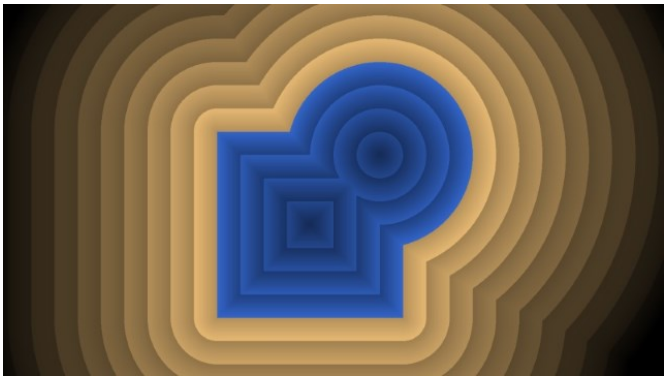
[4] `http://ksp.mff.cuni.cz/encyklopedie/vektory.html`

a odečteme od této hodnoty poloměr. Rozmyslete si, že toto nám opravdu vrátí správnou znaménkovou vzdálenost. Tato funkce by se dala vizualizovat následovně:



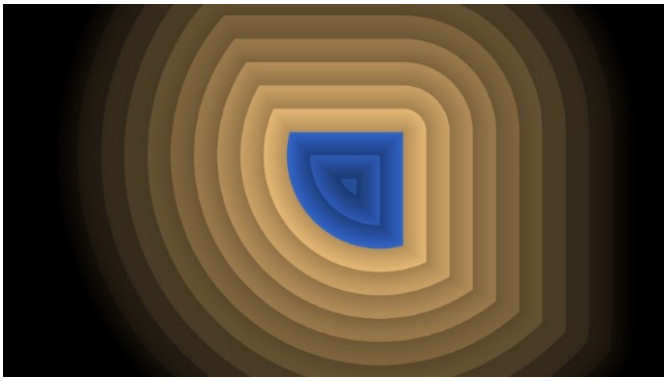
Žluté barvy jsou kladné hodnoty vzdálenosti, modré jsou záporné.

Silnou stránkou SDF je, že na nich můžeme snadno provádět množinové operace. Sjednocení dvou SDF uděláme tak, že použijeme menší z jejich hodnot, tedy minimum:



Vnější vzdálenosti jsou v této kombinaci správně, všimněte si ale, že uvnitř jsou sjednocené vzdálenosti špatně. Kolem ostrých rohů, kde kruh splývá se čtvercem, by měly vnitřní vzdálenosti tvořit kruhové vrstvy. Tento problém nemá žádné jednoduché řešení, ale naštěstí jej můžeme ignorovat. Bude nám stačit mít správné vnější vzdálenosti. A dokonce nemusíme mít ani to. Později narazíme na SDF, které vrací jen dolní odhad vzdálenosti a i to bude pro naše účely dostačující.

Průnik dvou SDF provedeme tím, že z obou hodnot vezmeme maximum:



Tento způsob reprezentace scény je u shaderových raytracerů velmi rozšířený – pokud v Shadertoy najdete shader s nějakou prostorovou scénou, velmi pravděpodobně používá právě SDF. Mnoho užitečných SDF primitiv a technik naleznete na tomto odkazu.[5]

Raymarching

Máme reprezentaci scény a nyní potřebujeme spočítat kolizi s paprskem. To se u SDF dělá takzvaným *raymarchingem*, tedy kráčením po paprsku.

Začneme v počátku paprsku. Pro aktuální pozici zjistíme d – hodnotu vzdálenostní funkce. Jelikož d je vzdálenost k nejbližšímu povrchu (nebo dolní odhad této vzdálenosti), máme zaručeno, že v kouli o poloměru d okolo aktuálního bodu není žádná kolize. Můžeme se tedy pohnout o d po směru paprsku a máme zaručeno, že žádnou potenciální kolizi nepřeskočíme.

Tento postup opakujeme, dokud d neklesne pod nějaké dostatečně malé epsilon, potom prohlásíme, že jsme nalezli kolizi. Pokud naopak uražená vzdálenost překročí nějakou horní hranici, řekneme, že paprsek vyletěl ze scény do nekonečna.

Této variantě raymarchingu, kde v každém bodě známe poloměr koule, ve které se zaručeně nenachází kolize, se někdy říká *sphere tracing*. Raymarching jako takový je obecnější algoritmus, který lze použít například při vykreslování mlhy, kdy skáčeme po krocích pevné velikosti a pro každou navštívenou pozici přečteme hustotu mlhy z nějaké šumové funkce nebo trojrozměrného pole.

Střílíme paprsky

Ještě potřebujeme dvě věci. První z nich je způsob, jak spočítat ze SDF normálu povrchu v nějakém bodě. To provedeme funkcí `computeNormal` (viz kód u prvního úkolu), která je odvozená podobným způsobem jako počítání normály z výškové mapy v minulém díle.

Druhou věcí je vygenerování paprsků samotných pro každý pixel obrazovky. Naše kamera bude definovaná svou pozicí, a směrem, kam se dívá. Všechny paprsky budou mít počátek v bodě, kde se nachází kamera. Někde ve směru, kam se kamera dívá, si představíme obdélník obrazovky. Pro každý pixel vygenerujeme paprsek tak, že protne střed daného pixelu na této pomyslné obrazovky. Také si to můžeme představit tak, že kamera se nachází uprostřed vašeho oka a paprsky z něj míří doprostřed každého pixelu monitoru, na který se (pravděpodobně) díváte.

Obdélník obrazovky bude orientovaný tak, že jeho plocha bude vždy kolmá na osu z jeho středu ke kameře (tedy směr, kam se kamera dívá). Jen pozice a směr pro pořádnou definici kamery nestačí, ještě musíme popsat, jak bude obdélník natočený podle osy směru, kam se kamera dívá. My jej vždy natočíme tak, že jeho horní a dolní hrany budou vodorovné.

Tento styl kamery se běžně používá ve hrách. Její střed je definovaný pozicí hráče a směr kamery ovládáme myší.

Na generování paprsků pro přesně tuto kameru dostanete funkci `getRayDirection`, jejíž kód naleznete u zadání prvního úkolu. Tato funkce umí pro každý pixel obrazu vygenerovat vektor směru paprsku, a to pomocí souřadnic pixelu `uv` (v rozsahu -1 až 1), vektoru `forward` (směru, kam se kamera dívá) a horizontálního zorného úhlu (ve hrách se mu říká *field of view* či *fov*). Vzdálenost mezi kamerou a pomyslným obdélníkem obrazovky vypočítáme tak, aby úhel mezi kamerou a jeho levým a pravým okrajem byl právě zadaný horizontální zorný úhel `hfov`. Velikost obdélníku obrazovky si funkce sama zjistí ze zabudovaných vstupů v Shadertoy.

[5] `https://iquilezles.org/www/articles/distfunctions/distfunctions.htm`

Úkol 1 [2b]:

Naprogramujte raymarchovací funkci `raymarch` do níže uvedené kostry shaderu.

Jelikož to je něco, na čem se dá poměrně snadno spálit, připomínáme adresu *zdrojaky@ksp.mff.cuni.cz*, kde vám s radostí poradíme. Pokud jste na něčem zaseklí déle než půl hodiny, určitě pište!

Raymarchovací funkce dostane souřadnice počátku paprsku (`origin`) a normalizovaný vektor směru paprsku (`direction`) a má vrátit vzdálenost, kterou musí paprsek urazit, než narazí na nejbližší kolizi. Pokud paprsek urazí vzdálenost větší než konstanta `FAR` a na žádnou kolizi nenarazil, tak předpokládáme, že vystřelil do nekonečna a vrátíme libovolnou hodnotu větší než `FAR` (abychom mohli později snadno zjistit, jestli paprsek do něčeho narazil nebo ne pomocí porovnání `raymarch(...) < FAR`).

Ve funkci použijete nějakou smyčku. Je dobrý nápad omezit její maximální počet iterací (třeba na 128). Pokud smyčka skončí kvůli velkému počtu iterací, budeme se tvářit, že paprsek vyletěl do nekonečna a prostě vrátíme vzdálenost `FAR`.

V kódu se nachází jednoduchá SDF scéna tvořená sjednocením tří koulí a roviny - `p.y` je vzdálenostní funkce vodorovné roviny ve výšce 0. Scénu klidně rozšiřte o další objekty, jestli chcete (ale není to součást úkolu).

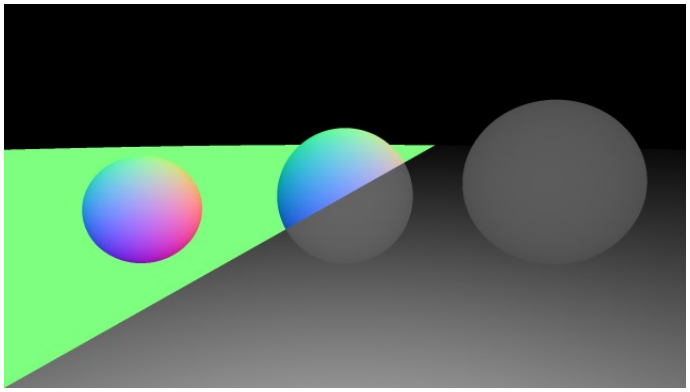
Pokud se vám editor seká, můžete vypnout pravidelné překreslování obrazu pomocí tlačítka „pause“. Když se změní pozice myši nebo překompiluje shader, obraz se stále obnoví.

```
#define PI 3.14159265358979323846264338327950

// Tuto vzdálenost budeme považovat za nekonečno
#define FAR 50.0

float sdfSphere(vec3 center,
                float radius,
                vec3 p)
{
    return length(center - p) - radius;
}

float sceneSdf(vec3 p)
{
    // Podlaha je prostor mezi dvěma rovinami
    float d = max(p.y, -p.y - 0.5);
    d = min(d,
      sdfSphere(vec3(0.0, 1.0, 0.0), 1.0, p));
    d = min(d,
      sdfSphere(vec3(-3.0, 0.8, 0.0), 0.8, p));
    d = min(d,
      sdfSphere(vec3(3.0, 1.2, 0.0), 1.2, p));
    return d;
}

// ... pokračování kódu ve vedlejším sloupci
```

```
// Raymarchovací funkce
// Vstupy:
// origin - počátek paprsku
// direction - směr paprsku (vektor
//             normalizovaný na velikost 1)
// Výstup: vzdálenost, kterou musí paprsek
//         urazit, než narazí na kolizi nebo
//         hodnota větší nebo rovna než FAR,
//         pokud do vzdálenosti FAR na žádnou
//         kolizi nenarazil.
// Nezapomeňte smyčku ukončit, pokud hodnota SDF
// klesne pod nějaké epsilon nebo pokud
// se provede příliš mnoho iterací.

float raymarch(vec3 origin, vec3 direction)
{
    // TODO: sem doprogramujte raymarching
}

vec3 computeNormal(vec3 p)
{
    const float epsilon = 0.001;
    float here = sceneSdf(p);
    return normalize(vec3(
        sceneSdf(p + epsilon * vec3(1, 0, 0))
            - here,
        sceneSdf(p + epsilon * vec3(0, 1, 0))
            - here,
        sceneSdf(p + epsilon * vec3(0, 0, 1))
            - here
    ));
}

// forward: normalizovaný vektor dopředu
// uv: souřadnice pixelu v -1..1
// hfov: horizontální zorný úhel ve stupních
vec3 getRayDirection(vec3 forward,
                     vec2 uv,
                     float hfov)
{
    // Z "forward" vygenerujeme i vektory vpravo
    // a nahoru, vše normalizované a navzájem
    // kolmé
    vec3 right = normalize(
        cross(forward,vec3(0.0, 1.0, 0.0)));
    vec3 up = normalize(cross(right, forward));

    // Převedeme hfov na radiány
    hfov = hfov / 180.0 * PI;

    // Spočítáme vzdálenost roviny od kamery
    float dist = 1.0 / tan(hfov * 0.5);

    // Přeškálujeme "up" tak, aby byl zachován
    // poměr stran
    up *= iResolution.y / iResolution.x;
    vec3 ray = forward * dist + right * uv.x
        + up * uv.y;

    return normalize(ray);
}

// ... pokračování kódu na další stránce
```

```
void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    // Střed souřadnic chceme uprostřed obrazu
    uv = uv * 2.0 - 1.0;

    vec3 cameraPosition = vec3(0.0, 2.0, 6.0);
    vec3 cameraLookAt = vec3(0.0, 1.0, 0.0);

    vec3 rayDirection = getRayDirection(
      normalize(cameraLookAt - cameraPosition),
      uv, 80.0);

    float depth = raymarch(cameraPosition,
        rayDirection);

    // Souřadnice dopadu paprsku
    vec3 hit = cameraPosition
        + rayDirection * depth;
    // Normála povrchu v bodě dopadu
    vec3 normal = computeNormal(hit);

    // Převedeme normálu na barvu
    vec3 color = normal * 0.5 + 0.5;

    // Dolní pravý trojúhelník obrazu
    // vyplníme vizualizací hloubky
    if(uv.x > uv.y)
    {
        color = vec3(1.0 / depth * 2.0);
    }

    if(depth > FAR)
    {
        color = vec3(0.0);
    }

    // Output to screen
    fragColor = vec4(color, 1.0);
}
```

Tento shader bude zobrazovat zároveň normálu v místě dopadu paprsku (levý horní trojúhelník) a zároveň hloubku (pravý dolní). Pokud je vaše raymarchovací funkce správně, výsledek by měl vypadat takto:



Úkol 2 [3b]:

Naprogramujte ovladatelnou kameru pomocí myši, viz zabudovaný vstup `iMouse.xy` - pozor, že je v pixelech a že složka y roste směrem nahoru.

Hezký styl ovládání je, že se kamera pohybuje na povrchu koule, v jejímž středu je bod `cameraLookAt` a kamera míří vždy na něj.

Bude se vám hodit tato funkce, která pro zadaný horizontální a vertikální úhel vrátí souřadnice daného místa na jednotkové kouli. (Úhly si můžete představit jako GPS souřadnice.)

```
// Pro místo na jednotkové kouli zadané
// pomocí úhlů vrátí jeho 3d souřadnice
// Vstup jsou úhly v radiánech
// První úhel je horizontální, druhý vertikální
vec3 anglesToPosition(float hor, float ver)
{
    return vec3(
        cos(ver) * sin(hor),
        sin(ver),
        cos(ver) * cos(hor)
    );
}
```

Gamma korekce

Ještě než do raytraceru přidáme počítání světla z minulého dílu, je třeba se zamyslet nad tím, co počítáme a co zobrazujeme.

Doteď jsme v shaderu počítali jen s barvou, kterou jsme přímo zobrazovali na monitor. Nějak automaticky jsme předpokládali, že vztah mezi zobrazovanou a vnímanou světlostí je lineární: pokud je jedna hodnota barvy dejme tomu dvakrát větší než jiná, tak ji také budeme vnímat jako dvakrát světlejší. Toto při zobrazování barev na monitor zhruba platí.

Lidské vnímání světla ale lineární není: je citlivější na nízké hodnoty. Nicméně vztah mezi zobrazovanou světlostí na monitoru a vnímanou světlostí *je* lineární. Důvodem je, že samotné zobrazování barev na monitor též lineární není. Reálné množství světla, které opouští pixel rozsvícený na barvu i, je přímo úměrné asi $i^{2.2}$. Tato konstanta vyplývá z vlastností starých CRT obrazovek.

Naštěstí pro lidské vidění platí, že pokud do oka dopadá světlo i, vnímáme přibližně světlost $i^{(1.0/2.2)}$. Transformace v monitoru se tedy vyruší s transformací v našich očích a barvy jsou tedy lineární.

Pro nás to ale znamená, že pokud chceme zobrazovat konkrétní intenzitu světla a ne „vnímanou“ barvu, musíme onu „automatickou“ korekci z monitoru sami kompenzovat. Proto před zobrazením barvy na monitor provedeme následující umocnění:

```
fragColor.rgb = pow(color, vec3(1.0/2.2));
```

Protože odteď v seriálu budeme počítat jen intenzity světla, tuto korekci vždy používejte.

Odbočka: toto ale není jediné místo, kde se dá na nelinearitě vnímání světlostí spálit. Ve hrách se jako zdroj difúzní barvy typicky používají textury, ve kterých je třeba fotka nějakého povrchu. A hodnoty v takové textuře budou nejspíš lineární vzhledem k vnímané světlosti. Pokud je chceme interpretovat jako množství světla, které nějaký povrch

odrazí, musíme je převést tak, aby byly lineární vzhledem k intenzitě světla a nikoliv vnímané světlosti. Kód by pak vypadal nějak takto:

```
float gamma = 2.2;

vec3 diffuseColor = texture(diffuseMap, uv).rgb;

// Nyní je diffuseColor lineární vzhledem
// k vnímané světlosti.

diffuseColor = pow(diffuseColor, vec3(gamma));

// Nyní je diffuseColor lineární vzhledem
// k intenzitě světla a můžeme s ním spočítat
// osvětlení (třeba do proměnné light)

// Nakonec opět převedeme na vnímané světlosti
fragColor.rgb = pow(light, vec3(1.0 / gamma));
```

Gamma korekce textur není potřeba v rámci seriálu řešit (na rozdíl od gamma korekce u výstupu).

Úkol 3 [3b]:

Implementujte do raytraceru Phongův osvětlovací model z minulého dílu seriálu. Implementujte osvětlení od slunce, ambientní světlo a oblohu (viz níže).

Scénu osvětlete pomocí falešného „slunce“. Slunce se implementuje tak, že *light vector* i intenzita příchozího světla je stejná pro všechny body scény (předpokládáme totiž, že všechny příchozí paprsky jsou rovnoběžné, tudíž že slunce je nekonečně malý bod nekonečně daleko).

Pokud jste minulý díl neřešili, použijte tento jednoduchý model:

```
// diffuseColor - barva povrchu
// normal - (normalizovaná) normála povrchu
// lightColor - barva světla
// lightDirection - (normalizovaný) směr
//                  ke zdroji světla
vec3 computeLight(vec3 diffuseColor,
                  vec3 normal,
                  vec3 lightColor,
                  vec3 lightDirection)
{
    return diffuseColor
        * max(0.0, dot(normal, lightDirection))
        * lightColor;
}
```

Aby nebyly stíny absolutně temné, použijeme velmi jednoduchou aproximaci globální iluminace: prostě na celou scénu aplikujeme nějaké konstantní „ambientní“ světlo. V praxi k finální hodnotě přičtete toto světlo pronásobené aktuální difúzní barvou povrchu.

Materiál použijte buď konstantní pro celou scénu, nebo jej zkuste udělat různý na základě místa kolize paprsku se scénou. Například takto lze zabarvit rovinu podlahy jinak než objekty na ní ležící.

Pixely, kde paprsek odletěl do nekonečna, zabarvěte nějakým barevným přechodem, který tvoří dojem oblohy (nemusí být nijak složitý, přechod mezi dvěma barvami bohatě stačí).

Úkol 4 [2b]:

Implementujte stíny od slunce.

Bod je osvětlen sluncem právě tehdy, když mezi ním a sluncem neleží žádná překážka. Jak to zjistíme? Vystřelíme směrem ke slunci paprsek! Pokud do něčeho narazí, mezi bodem a sluncem je překážka, a osvětlení od slunce nezapočítáme (ale ambientní světlo ano).

Dejte pozor na to, že pokud byste vystřeliti paprsek z místa kolize, funkce `raymarch` by vám ihned vrátila nulu, protože v místě kolize je vzdálenostní funkce nutně menší než epsilon uvnitř `raymarch` (protože její předchozí volání se v tomto místě zastavilo). Toto lze obejít tím, že výchozí bod nového paprsku posuneme trochu směrem „ven“ z povrchu pomocí normály v tomto bodě, stačí třeba o 0.02.

Úkol 5 [3b]:

Implementujte do raytraceru zrcadlové odrazy.

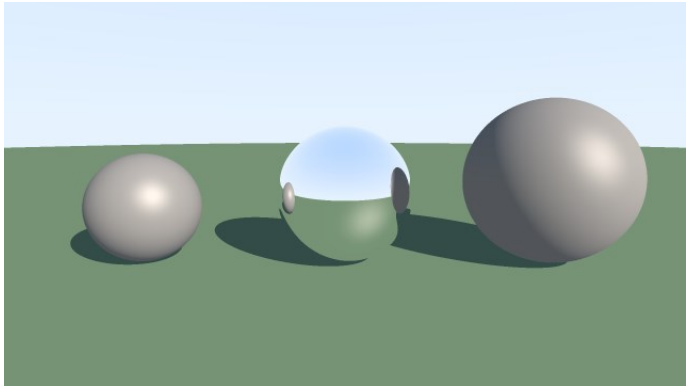
Zde by se hodila rekurze, ovšem shadery rekurzi neumí, tudíž je nutné obejít se bez ní. Také se vám bude hodit zabudovaná funkce GLSL `reflect`.

Opět je třeba vystřelit z místa kolize paprsek, tentokrát tam, kam by se původní paprsek odrazil. Též je potřeba jeho výchozí bod trochu posunout, tentokrát je vhodné použít posunutí ve směru nového paprsku.

Na místě kolize nového paprsku spočítejte osvětlení (klidně i včetně stínů, ale bez dalších vnořených odrazů) a výsledek přičtěte k osvětlení ze současné kolize, ale vynásobený nějakou konstantou.

Aby scéna vypadala zajímavě, udělejte zrcadlo pouze z jedné koule. Zda paprsek dopadl právě na ten správný objekt lze zjistit třeba tak, že pro daný objekt spočítáte SDF v místě dopadu, a pokud vyjde menší než nějaké epsilon (třeba 0.03), tak se místo dopadu nachází na daném objektu. Stejným postupem můžete různé objekty různě zabarvit.

Pokud nastavíte směr ke slunci na `normalize(vec3(0.9, 0.8, 1.0))`, difúzní barvu koulí na `vec3(0.5)`, barvu země na `vec3(0.3, 0.5, 0.3)` a spekulární barvu všeho na `vec3(0.5)` a spekulární „lesklost“ `a` na 8.0, výsledek předchozích tří úkolů by měl vypadat nějak takto:



Fraktály podruhé

Nyní máme hotový jednoduchý renderer SDF scény. Pojďme s ním něco zajímavého udělat!

V prvním díle jsme si hráli s Mandelbrot a Julia fraktály, což byly takové množiny komplexních čísel, kde pro funkci

$$f_c(z) = z^2 + c$$

je posloupnost $f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), \ldots$ omezená. Tyto fraktály můžeme kromě komplexních čísel definovat i pro *kvaterniony*, což je rozšíření komplexních čísel na čtveřici hodnot. Kromě reálné a imaginární složky i je v kvaternionech také j a k.

Důležité je, že i pro kvaterniony tvoří tyto množiny zajímavé obrazce. Tyto obrazce jsou sice čtyřrozměrné, ale můžeme si je promítnout do prostoru tím, že jednu komponentu čtyřrozměrné pozice nastavíme na nulu. Získáme tím jakousi 3D obdobu těchto fraktálů.

K tomu, abychom se na tyto fraktály podívali naším raytracerem, potřebujeme někde získat jejich vzdálenostní funkci. Něco takového naštěstí existuje, nicméně není to pravá vzdálenostní funkce, vrací totiž jen dolní odhad vzdálenosti. To nám ale pro vykreslování stačí. Níže je kód této vzdálenostní funkce, založený na tomto článku.[6]

```
vec4 quaternionMultiply(vec4 q, vec4 r)
{
    vec4 t;
    t.x = r.x*q.x - r.y*q.y - r.z*q.z - r.w*q.w;
    t.y = r.x*q.y + r.y*q.x - r.z*q.w + r.w*q.z;
    t.z = r.x*q.z + r.y*q.w + r.z*q.x - r.w*q.y;
    t.w = r.x*q.w - r.y*q.z + r.z*q.y + r.w*q.x;
    return t;
}

float lengthSquared(vec4 v)
{
    // dot(v, v) = v.x*v.x + v.y*v.y
    // + v.z*v.z + v.w*v.w = length(v)^2
    return dot(v, v);
}

float sdfMandelbrot(vec4 z, vec4 c)
{
    vec4 dz = vec4(1.0, 0.0, 0.0, 0.0);

    const int iterations = 32;

    for (int i = 0; i < iterations; i++)
    {
        dz = 2.0 * quaternionMultiply(z, dz);
        z = quaternionMultiply(z, z) + c;
        if (lengthSquared(z) > 256.0)
            break;
    }

    float distance = sqrt(
            lengthSquared(z) / lengthSquared(dz)
        ) * 0.5 * log(lengthSquared(z));

    return distance * 0.5;
}
```

Úkol 6 [2b]:

Vykreslete prostorový Julia fraktál. Aby byl dobře vidět fraktálovitý interiér, **useknete** horní půlku fraktálu rovinou.

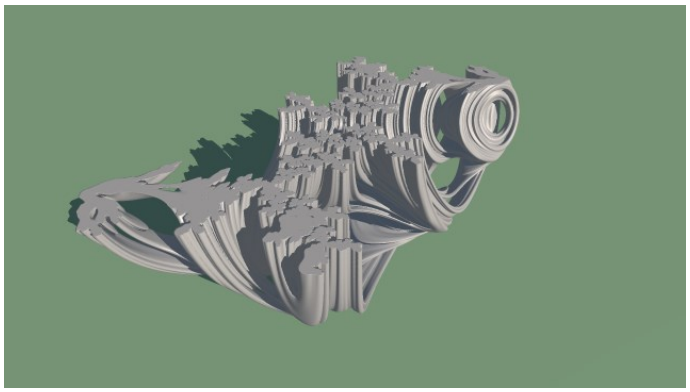
Fraktál rozsekněte pomocí sjednocení nebo průniku s vhodným objektem způsobem popsaným na začátku tohoto dílu.

Parametr z fraktálu nastavíme podle pozice v prostoru. Dobře funguje, když 3D prostor namapujete na 4D kvaterniony prostě tak, že první tři komponenty zkopírujete a poslední komponentu necháte nulovou. Také je třeba nastavit parametr c na něco zajímavého (pro nulu je fraktál jen koule), dobře funguje třeba `vec4(-1.5,7,16.5,-6)/22.0`, ale pokuste se najít vlastní zajímavý bod.

Aby se fraktál vykresloval spolehlivě, je potřeba mírně modifikovat raymarchovací funkci. Konkrétně je třeba zajistit, že se nikdy neprovede krok delší než 2. Jinak by fraktál mohl být přeskočen (jeho vzdálenostní funkce nefunguje dobře, když jste od fraktálu daleko).

Také `epsilon` v raymarchovací funkci má velký vliv na vzhled fraktálu. Nižší hodnoty vedou k detailnějšímu povrchu za cenu zhoršeného výkonu. Výkon můžete ladit též pomocí počtu iterací ve funkci fraktálu.

Pro parametry `cameraPosition = vec3(0.0, 2.0, 2.0)` a `cameraLookAt = vec3(0.0, 0.5, 0.0)` a fraktál se středem ve výšce 1 by měl výsledek vypadat takto:



Kuba Pelc

[6] https://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm

Recepty z programátorské kuchařky: Pravděpodobnost

Náhodná čísla a pravděpodobnost hrají v informatice důležitou roli. Když neumíme pro nějakou úlohu najít algoritmus, který je rychlý i v nejhorším případě, můžeme se spokojit s algoritmem rychlým alespoň v průměru. Nebo naopak s algoritmem, který je sice vždy rychlý, ale někdy odpoví špatně. Často se přitom hodí, aby si algoritmus „házel korunou“, tedy generoval nějaká náhodná čísla. Takovým algoritmům se říká *pravděpodobnostní* nebo také *randomizované.* V této kuchařce nahlédneme do základů teorie pravděpodobnosti a vybudujeme jí dost na to, abychom uměli zkoumat jednoduché randomizované algoritmy.

Po dlouhá staletí lidé přemýšleli nad pravděpodobností bez toho, aby jí rozuměli matematicky. Pravděpodobnost často není intuitivní, a proto i slavní matematici, jako třeba Newton, někdy v úvahách o náhodě chybovali. Naštěstí v minulém století přišel ruský matematik Andrej Kolmogorov s matematickým vysvětlením pravděpodobnosti, které nám dovoluje použít matematiku k přemýšlení o náhodných jevech, a vyhnout se tak chybám.

Jelikož „opravdová“ teorie pravděpodobnosti závisí na poměrně pokročilé matematice (takzvané teorii míry), vybudujeme ji trochu jednodušším způsobem. Bude výrazně intuitivnější než ta kolmogorovská, ale zvládne popsat jen konečné objekty. Budeme tedy moci zkoumat kostky, karty, algoritmy používající náhodná čísla z nějakého daného rozsahu (třeba celá čísla od 1 do 100) a podobně. Nezvládneme naopak korektně uvažovat o náhodných reálných číslech, náhodných bodech v rovině a jiných nekonečných věcech.

Jevy a jejich pravděpodobnost

Pro začátek si pravděpodobnost ukážeme na příkladu: Máme obyčejnou hrací kostku. Pokud kostkou hodíme, můžeme si být jisti, že na ní padne číslo od 1 do 6. Číslům 1 až 6 budeme říkat *elementární jevy.* Obecně, kdykoliv provedeme nějaký *náhodný experiment* (to je třeba hod kostkou), vždy nastane právě jeden elementární jev – v našem příkladu vždy padne jedno z čísel 1 až 6. Každému elementárnímu jevu můžeme přiřadit jeho *pravděpodobnost.* To je nějaké reálné číslo mezi 0 nebo 1, přičemž pravděpodobnosti všech elementárních jevů se sečtou na 1. To odpovídá tomu, že pokaždé nastane právě jeden z elementárních jevů – kostka nezůstane stát na rohu, ani nepadne jednička a dvojka současně. Prozatím tedy řekněme, že pravděpodobnost je funkce, která každému elementárnímu jevu přiřadí číslo od 0 do 1 a že se všechny pravděpodobnosti sečtou na 1.

Co kdybychom chtěli něco říci o pravděpodobnosti toho, že nám padne liché číslo. Takový výsledek pokusu už není elementární jev, ale můžeme ho stále z elementárních jevů poskládat: lichá čísla popíšeme množinou elementárních jevů $L = \{1, 3, 5\}$. Obecně můžeme za (náhodný) *jev* prohlásit jakoukoliv množinu elementárních jevů. Pro kostku to může být třeba jev $A = \{5, 6\}$ (padlo aspoň 5), jev $\emptyset$ (ten nenastane nikdy), nebo $V = \{1, 2, 3, 4, 5, 6\}$ (takovýto jev nastane vždy).

Pravděpodobnost jevu pak můžeme definovat jako součet pravděpodobností těch elementárních jevů, ze kterých se daný jev skládá. Jelikož všechny elementární jevy na kostce mají pravděpodobnost $1/6$, bude pravděpodobnost, že padne liché číslo (to je náš jev L), rovna $1/6+1/6+1/6 = 3/6 = 1/2$. Označíme-li pravděpodobnost jevu J jako $P(J)$, vyjde pro ostatní zmíněné jevy $P(A) = 2/6 = 1/3$, $P(\emptyset) = 0$, $P(V) = 1$.

Jevy jsou tedy množiny a můžeme o nich jako o množinách mluvit. Můžeme například říct, že jevy A a B jsou *disjunktní,* tedy že $A \cap B = \emptyset$. To odpovídá tomu, že tyto dva jevy nemohou nastat najednou. Například jevy „padlo sudé číslo“ a „padlo liché číslo“ jsou zjevně disjunktní, protože žádné číslo není současně sudé a liché. Podobně jev $A \cup B$ odpovídá tomu, že nastane jev A nebo jev B, a jev $A \cap B$ tomu, že nastane současně A a B.

Zkusme si nyní rozmyslet, že pro jakékoliv dva disjunktní jevy A a B platí $P(A \cup B) = P(A) + P(B)$. Tedy pravděpodobnost (disjunktního) sjednocení A a B je součet pravděpodobností A a B. Toto tvrzení platí proto, že jsme definovali pravděpodobnosti A a B jako součty pravděpodobností elementárních jevů v A a B. Když tedy sečteme pravděpodobnosti všech elementárních jevů, které jsou v A, nebo v B (ale ne v obou, protože A a B mají prázdný průnik), dostaneme to samé, jako když sečteme $P(A)$ a $P(B)$.

Princip inkluze a exkluze

Pro každé dvě množiny A a B platí $|A \cup B| = |A| + |B| - |A \cap B|$. To proto, že do $|A| + |B|$ jsme prvky v průniku započítali dvakrát, a musíme tedy odečíst jejich počet, abychom dostali správný počet prvků ve sjednocení. Tomuto tvrzení (respektive jeho zobecnění pro libovolný počet množin) se někdy říká *princip inkluze a exkluze,* česky by asi dalo říci princip zahrnutí a vyloučení.

Podobně pro pravděpodobnosti se dá náhlednout rovnost $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. Vzpomeňme si, že pravděpodobnost jevu je součtem pravděpodobností elementárních jevů, ze kterých se skládá. Pravděpodobnosti elementárních jevů v průniku jsme tedy v $P(A) + P(B)$ započetli dvakrát a musíme je odečíst, abychom dostali pravděpodobnost sjednocení. Všimněme si, že pokud jsou množiny A a B disjunktní, dostaneme $P(A \cup B) = P(A) + P(B)$, jak jsme si rozmysleli před chvílí.

Princip inkluze a exkluze je užitečné pravidlo k počítání pravděpodobností, ale jeho hlavní síla spočívá v následujícím. Uvědomme si, že pravděpodobnosti jsou nezáporná čísla. Platí tedy, že $P(A \cup B) = P(A) + P(B) - P(A \cap B) \leq P(A) + P(B)$, protože $P(A \cap B)$ je nezáporné číslo. Jinými slovy pravděpodobnost toho, že nastane A nebo B je nejvýše součet pravděpodobnosti, že nastane A, a pravděpodobnosti, že nastane B. Tomuto odhadu se říká *odhad sjednocení* (anglicky *union bound*) a bývá obzvláště přesný, pokud jevy, na které ho použijeme, mají malé pravděpodobnosti. (Raději zmiňujeme, že matematici slovem *odhad* míní nerovnost, nikoliv nějaké „odhadnutí od oka“.)

Co kdybychom teď měli sjednocení tří jevů? Pak si to sjednocení správně uzávorkujeme! $A \cup B \cup C = (A \cup B) \cup C$. Poté můžeme použít odhad sjednocení na dvojice jevů.

$$P(A \cup B \cup C) = P((A \cup B) \cup C) \leq P(A \cup B) + P(C) \leq P(A) + P(B) + P(C).$$

Můžeme pokračovat indukcí a dokázat, že pravděpodobnost sjednocení libovolného počtu jevů je nejvýše součet jednotlivých pravděpodobností.

Podmíněná pravděpodobnost

Náš kamarád hodil kostkou a řekl nám, že padlo číslo menší než 4 (padlo tedy 1, 2 nebo 3). Jaká je pravděpodobnost, že padlo liché číslo? V tomto případě není těžké si rozmyslet, že správná odpověď je 2/3. Ale teorie pravděpodobnosti tak, jak jsme si ji zatím vymysleli, nám na podobné úvahy nestačí.

Definujeme tedy *podmíněnou pravděpodobnost.* Řekneme, že $P(A \mid B)$ je pravděpodobnost, že nastal jev A, pokud už víme, že nastal jev B. Čteme to obvykle „pravděpodobnost A za podmínky B“.

Čemu by se ale měla $P(A \mid B)$ rovnat? Předpokládejme, že uděláme hodně náhodných experimentů. $P(B)$ říká, v jakém zlomku z nich nastal jev B. Z nich si vybereme ty, v nichž nastal i jev A. Ty tvoří zlomek $P(A \cap B)$ ze všech experimentů, takže z těch, kdy nastalo B, je to zlomek $P(A \cap B)/P(B)$. Definujeme tedy:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

Ve zmíněném příkladu dostaneme

$$P(\{1,3,5\} \mid \{1,2,3\}) = P(\{1,3\}) \,/\, P(\{1,2,3\}),$$

což se opravdu rovná 2/3, protože všechny elementární jevy mají v našem případě stejnou pravděpodobnost.

Můžeme si to také představit tak, že z množiny všech elementárních jevů vyřadíme ty, o nichž víme, že nenastaly. Zbudou nám tedy ty elementární jevy, které leží v B. Počítáme-li pak pravděpodobnost jevu A, musíme vyřazené jevy smazat, tedy uvážit průnik $A \cap B$. To ovšem nestačí: vyřazením jsme porušili pravidlo, že pravděpodobnosti všech elementárních jevů se sečte na jedničku: nyní se sečtou na $P(B)$. Proto ještě „změníme měřítko“ vydělením všech pravděpodobností číslem $P(B)$.

Nezávislé jevy

Nyní zkusíme hodit dvěma kostkami po sobě a výsledek přečíst jako dvojciferné desítkové číslo. Elementárních jevů je tedy celkem 36 jsou to čísla od 11 do 66. Uvažme tyto jevy:

$A =$ „první číslice je 1“,
$B =$ „druhá číslice je 1“,
$C =$ „číslo je menší než 30“.

Jejich pravděpodobnosti snadno spočítáme jako $P(A) = P(B) = 6/36 = 1/6$ a $P(C) = 12/36 = 1/3$.

Představme si nyní, že víme, že nastal jev A, tedy padlo jedno z čísel 11 až 16. Co jsme schopni říci o tom, zda nastal jev B? Jeho pravděpodobnost zůstává stále stejná: 1/6. Jev A nám tedy o jevu B nedává žádnou informaci. Tehdy říkáme, že jevy A a B jsou *nezávislé.*

Kdybychom naopak věděli, že nastal jev C, pravděpodobnost jevu A by se změnila na 1/2, protože z čísel 11 až 26 celá polovina začíná jedničkou. Tyto jevy jsou tedy *závislé.*

Nezávislost jevů A a B můžeme popsat požadavkem

$$P(A \mid B) = P(A).$$

Dosadíme-li definici podmíněné pravděpodobnosti, dostaneme:

$$P(A \cap B)/P(B) = P(A),$$

čili

$$P(A \cap B) = P(A) \cdot P(B).$$

Za definici nezávislosti se většinou považuje tato poslední rovnost, protože funguje i pro $P(B) = 0$, kdy by podmíněná pravděpodobnost nebyla vůbec definovaná. Také je z ní hned vidět, že nezávislost je symetrická vlastnost. Dodejme ještě, že samozřejmě platí i $P(B \mid A) = P(B)$.

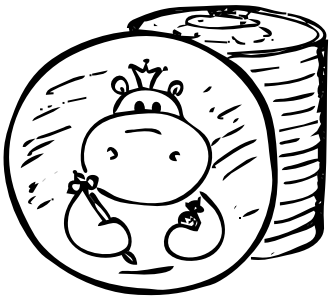
Definovat nezávislost pro více jevů je složitější, obecně nestačí říci, že pravděpodobnost toho, že nastanou současně, je součin jednotlivých pravděpodobností. Je potřeba, aby to platilo pro kteroukoliv podmnožinu jevů.

Náhodné proměnné a střední hodnota

Představme si nyní jednoduchý pravděpodobnostní algoritmus. Funkce *random*(x) bude vracet náhodné celé číslo z rozsahu 0 až $x - 1$.

1. $x \leftarrow$ *random*(2)
2. Pokud $x = 1$:
3. $\quad y \leftarrow$ *random*(2)

Algoritmus si tedy hodí korunou (vygeneruje náhodný bit) a v případě, že padla jednička, hodí ještě jednou. Jak ho pomocí teorie pravděpodobnosti popíšeme? Možné průběhy výpočtu můžeme popsat jako elementární jevy. Jelikož pro pevný vstup (tady dokonce žádný nemáme) je průběh výpočtu jednoznačně určený hodnotami náhodných čísel, můžeme říci, elementární jevy jsou možné posloupnosti náhodných čísel.



Pro náš jednoduchý program to jsou posloupnosti 0, 10 a 11. Přitom 0 má pravděpodobnost 1/2, zbylé dvě 1/4.

Co kdybychom chtěli říci, jak dlouho náš program běží? Pro jednoduchost to budeme počítat v příkazech. V nejhorším případě se provedou 3, ale kdybychom to chtěli říci přesněji, mohli bychom říci, že pro posloupnost 0 se provedou 2, zatímco pro 10 a 11 se provedou 3. Doba běhu randomizovaného algoritmu je hezkým příkladem takzvané náhodné proměnné, kterou teď zavedeme obecně.

Když provedeme náhodný experiment (třeba hodíme kostkou), nastane jeden z elementárních jevů. *Náhodná proměnná* (nebo také *náhodná veličina*) je jedno číslo určené tím, který elementární jev nastal. Příkladem by mohla být náhodná proměnná L, která je 0, pokud padlo na kostce sudé číslo, a 1, pokud liché. Dalším příkladem náhodné proměnné může být třeba „číslo, které padlo na kostce, umocněné na druhou“. Tu označme třeba D.

Všimněte si, že podmínka, ve které figurují náhodné proměnné, vždy určuje nějaký jev: množinu elementárních jevů, pro které podmínka platí. Například $L = 1$ platí pro $\{1,3,5\}$, $D < 10$ pro $\{1,2,3\}$ a $D < 10 \wedge L = 0$ pro $\{2\}$. Proto se můžeme ptát na pravděpodobnost toho, že podmínka platí, což se obvykle značí pomocí hranatých závorek: $P[D < 10] = P(\{1,2,3\}) = 3/6 = 1/2$.

Také se můžeme ptát, jaká je průměrná hodnota náhodné proměnné. Zkusme si to na proměnné D z předchozího příkladu, ale uvažujme obecné pravděpodobnosti stěn kostky $p_1, \ldots, p_6$ (kostka by tedy mohla být falešná). Představme si, že hodíme kostkou N-krát pro nějaké hodně velké číslo N. Jedniček padne přibližně $p_1 \cdot N$, dvojek $p_2 \cdot N$ atd., takže průměr proměnné D vyjde:

$$\frac{1^2 \cdot p_1 \cdot N + 2^2 \cdot p_2 \cdot N + \ldots + 6^2 \cdot p_6 \cdot N}{N}.$$

To také můžeme zapsat jako

$$1^2 \cdot p_1 + 2^2 \cdot p_2 + \ldots + 6^2 \cdot p_6.$$

Je to tedy vážený průměr, v němž roli vah hrají pravděpodobnosti jednotlivých možností. Pro poctivou kostku ($p_1 = \ldots = p_6 = 1/6$) se z toho stane obyčejný aritmetický průměr s hodnotou $91/6$.

Obecně můžeme definovat *střední hodnotu* náhodné proměnné X (značíme $\mathbf{E}[X]$) jako průměr hodnot, které proměnná přiřazuje jednotlivým elementárním jevům, vážený pravděpodobnostmi těchto jevů:

$$\mathbf{E}[X] = \sum_{\omega} X(\omega) \cdot P(\omega),$$

kde suma probíhá přes všechny elementární jevy ω a $X(\omega)$ je hodnota proměnné pro elementární jev ω.

Pokud jsou hodnoty proměnné celá čísla, někdy je praktičtější pro každou hodnotu posbírat všechny elementární jevy, pro které proměnná této hodnoty nabývá. Dostaneme:

$$\mathbf{E}[X] = \sum_{a \in \mathbb{Z}} a \cdot P[X = a].$$

(Nenechte se vyvést z míry tím, že sčítáme nekonečně mnoho členů. Pro konečný počet elementárních jevů je jen konečně mnoho sčítanců nenulových.)

Lze matematicky dokázat (byť my to zde dělat nebudeme), že pokud zprůměrujeme hodně pokusů, bude výsledek blízko $\mathbf{E}[X]$, a pokud budeme průměrovat více a více pokusů, tak se bude přibližovat (jazykem matematické analýzy: průměr bude konvergovat) k $\mathbf{E}[X]$.

Linearita střední hodnoty

Jednou z důležitých vlastností střední hodnoty je, že je lineární. Tím se myslí, že pro libovolné dvě náhodné proměnné platí $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$ a také $\mathbf{E}[cX] = c \cdot \mathbf{E}[X]$ pro každé číslo c. Uvědomte si, že $X + Y$ a cX jsou také náhodné proměnné.

Předtím, než tuto vlastnost dokážeme poctivě, rozmysleme si, že je velmi intuitivní. Představme si, že máme dva randomizované algoritmy používající náhodnost. Označme X a Y náhodné proměnné, které udávají jejich doby běhu. Linearita střední hodnoty pak říká, že když spustíme X a poté Y, bude průměrná celková doba běhu stejná jako součet průměrných dob běhu jednotlivých algoritmů. Podobně pokud algoritmus dvakrát zpomalíme, bude i průměrná doba běhu dvakrát větší. To není nikterak překvapivé.

Nyní matematický důkaz, nejprve pro násobení konstantou. Stačí dosadit do definice střední hodnoty:

$$\begin{aligned}\mathbf{E}[cX] &= \sum_{\omega} (cX)(\omega) \cdot P(\omega) = \sum_{\omega} c \cdot X(\omega) P(\omega) \\ &= c \cdot \sum_{\omega} X(\omega) P(\omega) = c \cdot \mathbf{E}[X].\end{aligned}$$

A teď pro součet:

$$\begin{aligned}\mathbf{E}[X + Y] &= \sum_{\omega} (X + Y)(\omega) \cdot P(\omega) \\ &= \sum_{\omega} (X(\omega) + Y(\omega)) \cdot P(\omega) \\ &= \sum_{\omega} X(\omega) P(\omega) + Y(\omega) P(\omega) \\ &= \left(\sum_{\omega} X(\omega) P(\omega) \right) + \left(\sum_{\omega} Y(\omega) P(\omega) \right) \\ &= \mathbf{E}[X] + \mathbf{E}[Y].\end{aligned}$$

Raději zdůrazníme, že jsme nikde nepotřebovali předpokládat žádný druh nezávislosti: linearita střední hodnoty funguje za všech okolností.

Nyní si ukažme dva příklady využití linearity. Házíme dvěma kostkami, zapisujeme dvojciferný výsledek D a ptáme se, jaká je jeho střední hodnota $\mathbf{E}[D]$. Mohli bychom zajisté rozebrat všech 36 možností, ale existuje jednodušší cesta. Uvážíme náhodné veličiny pro číslo na první kostce X a to na druhé Y. Jistě je $D = 10X + Y$, takže podle linearity $\mathbf{E}[D] = 10\mathbf{E}[X] + \mathbf{E}[Y]$. Ovšem X a Y jsou úplně obyčejné hodnoty na kostkách, takže jejich střední hodnoty vyjdou $(1 + \ldots + 6)/6 = 7/2$. Proto $\mathbf{E}[D] = (10 + 1) \cdot 7/2 = 77/2$.

V druhém příkladu hodíme N-krát kostkou a budeme se ptát, kolik padlo šestek. Označme tuto náhodnou proměnnou S. Elementární jevy jsou posloupnosti hodů, takže jich je 6^N. Ovšem $\mathbf{E}[S]$ spočítáme snadno trikem takřka kouzelnickým. Rozložíme S na součet proměnných $S_1 + \ldots + S_N$, kde S_i říká, kolik padlo šestek v i-tém hodu. To je trochu přihlouplá otázka, protože padla buď jedna, anebo žádná. Ale každopádně se snadno spočítá, že střední hodnota $\mathbf{E}[S_i]$ je $0 \cdot P[S_i = 0] + 1 \cdot P[S_i = 1] = P[S_i = 1]$, což je pravděpodobnost, že v i-tém hodu padla šestka, tedy $1/6$. Proto $\mathbf{E}[S] = \mathbf{E}[S_1 + \ldots + S_N] = \mathbf{E}[S_1] + \ldots + \mathbf{E}[S_N] = N \cdot 1/6 = N/6$. Žádné překvapení se nekoná.

Mimochodem, tato úvaha je speciálním případem obecné techniky: Libovolnému jevu J můžeme přiřadit jeho *indikátor,* což je náhodná proměnná I_J, která nabývá hodnoty 1, pokud jev nastane, a jinak je nulová. Vychází pak $\mathbf{E}[I_J] = P[I_J = 1] = P(J)$.

Lemma o džbánu

Představte si, že opakovaně házíte kostkou, dokud nepadne šestka. Kolikrát v průměru hodíte? Obecněji: opakujeme nějaký pokus, který se pokaždé povede s nějakou pravděpodobností p. Pokud vám to připomíná přísloví o chození se džbánem pro vodu, jste na správné stopě. Suchým vědeckým jazykem jde o tzv. střední hodnotu geometrického rozdělení, ale nám přijde džbán s vodou výstižnější.

Jak to zapadá do naší teorie? Posloupnosti hodů jsou elementární jevy, počet hodů je náhodna proměnná a my se ptáme na její střední hodnotu. Nenechte se prosím znepokojit tím, že elementárních jevů je nekonečně mnoho, s čímž naše teorie nepočítala. Pohybujeme se sice na tenkém ledu, ale slibujeme, že se nepropadneme, protože toto nekonečno je „dostatečně krotké“.

Lemma říká následující: Mějme džbán. Kdykoliv s ním jdeme pro vodu, tak se jeho ucho utrhne s pravděpodobností p, nezávisle (ve smyslu popsaném výše) na ostatních pokusech. Pak ve střední hodnotě půjdeme se džbánem pro vodu $(1/p)$-krát, než se ucho utrhne.

Než lemma dokážeme, rozmysleme si, co říká o našem čekání na šestku: Pravděpodobnost šestky je $1/6$, takže v průměru hodíme $1/(1/6) = 6$-krát.

Nyní důkaz: Nechť U je náhodná proměnná, která nám říká, při kolikátém pokusu se utrhlo ucho. Střední hodnotu budeme chtít spočítat podle definice:

$$\mathbf{E}[U] = \sum_{i=1}^{\infty} i \cdot P[U = i].$$

Potřebujeme tedy znát pravděpodobnosti toho, že ucho se poprvé utrhne v i-tém kroku. Pro $i = 1$ je to triviální: ucho se utrhne hned napoprvé s pravděpodobností p. Pro $i = 2$ uvážíme, že se napoprvé neutrhlo (to má pravděpodobnost $1 - p$) a napodruhé utrhlo (tedy p). Jelikož oba jevy jsou nezávislé, můžeme jejich pravděpodobnosti vynásobit a dostaneme $P[U = 2] = (1 - p) \cdot p$. Pro obecné i se při prvních $i-1$ pokusech ucho neutrhne a v i-tém pokusu utrhne, tedy dostaneme $P[U = i] = (1 - p)^{i-1} \cdot p$.

Teď dosadíme spočtené pravděpodobnosti:

$$\mathbf{E}[U] = \sum_{i=1}^{\infty} i \cdot (1 - p)^{i-1} \cdot p = p \cdot \sum_{i=0}^{\infty} (i + 1)(1 - p)^i.$$

To je nějaká nekonečná suma, u níž budeme věřit, že se sečte na konečné číslo (znalci analýzy mohou použít podílové kriterium konvergence). Jak ji spočítáme?

Označme S hodnotu druhé sumy, platí tedy $\mathbf{E}[U] = pS$. Abychom lépe viděli, co se děje, sumu si rozepíšeme:

$$S = 1 \cdot (1 - p)^0 + 2 \cdot (1 - p)^1 + 3 \cdot (1 - p)^2 + \ldots$$

Podívejme se, co se stane po vynásobení obou stran $1 - p$:

$$(1 - p)S = 1 \cdot (1 - p)^1 + 2 \cdot (1 - p)^2 + 3 \cdot (1 - p)^3 + \ldots$$

To se nepočítá o nic lépe, ale je to docela podobné výrazu pro S: v jednom případě je u $(1 - p)^i$ koeficient $i + 1$, v druhém i. Zkusíme tedy od sebe oba výrazy odečíst:

$$S - (1 - p)S = (1 - p)^0 + (1 - p)^1 + (1 - p)^2 + \ldots$$

Levou stranu můžeme zjednodušit na pS, napravo je nějaká geometrická řada s kvocientem $1 - p$. Tabulkový vzoreček pro součet geometrické řady s kvocientem q říká, že $q^0 + q^1 + q^2 + \ldots = 1/(1 - q)$. V našem případě je $q = 1 - p$, takže $pS = 1/(1 - (1 - p)) = 1/p$. My už ovšem víme, že pS je také rovnou hledané střední hodnotě $\mathbf{E}[U]$. Hotovo.

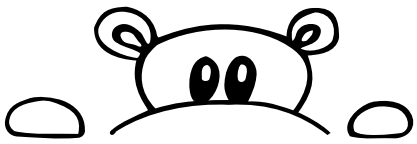
(Mimochodem, kdyby vám vrtalo hlavou, jak se odvozuje vzoreček pro součet geometrické řady, dělá se to trikem velmi podobným tomu našemu: Pokud $G = q^0 + q^1 + \ldots$, pak $qG = q^1 + q^2 + q^3 + \ldots$. Opět obě řady odečteme a získáme $G - qG = q^0 = 1$. Proto $(1-q)G = 1$, a tedy $G = 1/(1-q)$.)

Odbočka k hledání pivota

Lemma o džbánu si vyzkoušíme na analýze algoritmu. V třídícím algoritmu Quicksort potřebujeme najít pivota, který bude přibližně mediánem. Řekněme, že v setříděném pořadí prvků má pivot ležet v prostřední třetině. Pak by stále platilo, že Quicksort poběží v čase $\mathcal{O}(n \log n)$. Jak ale takového pivota najít? Pomůže jednoduchý pravděpodobnostní algoritmus: vybereme pivota náhodně ze zadaných prvků, spočítáme, kolik prvků je menších a kolik větších, a pokud pivot neleží v prostřední třetině, algoritmus prostě zopakujeme.

Jeden průchod algoritmu trvá $\mathcal{O}(n)$. Algoritmus může uspět hned v prvním průchodu, takže v nejlepším případě skončí v lineárním čase. Také by se mohlo stát, že budeme mít smůlu a pokaždé vybereme minimum, a tehdy se algoritmus nezastaví nikdy – to nás ovšem nemusí trápit, tento průběh algoritmu má pravděpodobnost rovnu 0.

Ukážeme, že průměrná časová složitost je také lineární. Střední hodnota času výpočtu je určitě $\mathcal{O}(n)$ krát střední hodnota počtu průchodů (všimněte si, jak jsme zde použili linearitu střední hodnoty). Průchod uspěje, pokud se trefí do prostřední třetiny. Jelikož všechny prvky vybíráme se stejnou pravděpodobností, nastane to s pravděpodobností $1/3$. Podle lemmatu o džbánu tedy musíme udělat v průměru 3 průchody, než se ucho utrhne a my najdeme prvek v prostřední třetině.



Zesilování pravděpodobnosti

Představme si nakonec, že máme nějaký pravděpodobnostní algoritmus, který vždy doběhne rychle, ale nezaručuje správný výsledek. Typickým příkladem je třeba takzvaný Rabinův-Millerův test prvočíselnosti. Nebudeme ho vysvětlovat detailně, podrobnosti najdete například v Medvědových Algoritmech okolo teorie čísel.[7] Důležité ale je, že se chová takto: Dáme-li mu na vstupu prvočíslo, vždy správně odpoví „ANO, je to prvočíslo.“ Složené číslo odhalí s pravděpodobností aspoň $1/2$: pokud mu ho dáme, odpoví s pravděpodobností alespoň $1/2$ NE, jinak ANO.

Poloviční pravděpodobnost chyby nezní moc užitečně, ale můžeme ji snadno vylepšit tím, že algoritmus k-krát zopakujeme. Číslo budeme považovat za prvočíslo jen tehdy, když se na tom shodlo všech k opakování algoritmu.

Jaká je nyní pravděpodobnost chyby? Pokud je číslo doopravdy prvočíslem, pokaždé to zjistíme správně, takže celkově odpovíme ANO. Pokud je složené, odpověděli bychom špatně (tedy ANO) jen tehdy, kdyby se každé z k spuštění algoritmu zmýlilo. Jelikož tato selhání jsou navzájem nezávislá (algoritmus si pokaždé vygeneruje nová náhodná čísla nezávisle na těch předchozích), pravděpodobnost k selhání je nejvýš $(1/2)^k = 1/2^k$. Už pro $k = 10$ je to méně než $1/1000$.

Tomuto triku se říká *zesilování pravděpodobnosti* (anglicky *probability amplification*). Zatím jsme ho použili pro stlačení pravděpodobnosti chyby pod libovolně nízkou (ale kladnou) konstantu. Někdy se ovšem hodí umět víc.

Co kdybychom na prvočíselnost testovali N čísel? S původním Rabinovým-Millerovým testem bychom se mýlili v průměrně $N/2$ případech (pokud by všechna čísla byla složená). Pokud bychom chtěli stlačit průměrný počet chyb pod konstantu (řekněme pod 1), museli bychom pravděpodobnost chyby v jednom testu stlačit pod $1/N$. Toho dosáhneme, pokud zvolíme $k > \log_2 N$. Tím jsme algoritmus asymptoticky zpomalili ($\mathcal{O}(\log N)$-krát), ale to je poměrně malá cena za tak podstatné snížení chybovosti. Další zvětšování k snižuje chybovost exponenciálně: už pro $k = 2 \log_2 N$ vyjde pravděpodobnost chyby $1/N^2$.

[7] `http://mj.ucw.cz/papers/numth.pdf`

Pár slov závěrem

Pokud se chcete dozvědět o pravděpodobnosti více, můžeme vřele doporučit skvělé přednášky z Harvardu. Jejich videozáznamy jsou dostupné na YouTube.[8]

Až si někdy budete číst o pravděpodobnosti, nejspíš se setkáte s axiomy pravděpodobnosti zavedenými jinak, než jak jsme je zavedli my. Ty naše kuchařkové jsou intuitivnější, ale bohužel se nedají jednoduše zobecnit na nekonečné množiny. O „plnotučných“ Kolmogorových axiomech pravděpodobnosti se můžete dočíst ve Wikipedii.[9]

Další jednoduché randomizované algoritmy a datové struktury najdete v knížce Průvodce labyrintem algoritmů.[10] Inspirovat vás také může seriál z 16. ročníku KSP.[11]

Pokud by vám i tak něco nebylo jasné, ptejte se na našem fóru.

Martin „Medvěd“ Mareš & Jakub Tětek

[8] https://www.youtube.com/playlist?list=PL2SOU6wwxB0uwwH80KTQ6ht66KWxbzTIo
[9] https://en.wikipedia.org/wiki/Probability_axioms
[10] http://pruvodce.ucw.cz/
[11] https://ksp.mff.cuni.cz/h/ulohy/16/