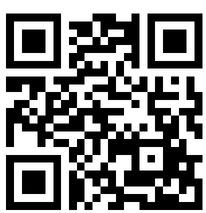
Korespondenční Seminář z Programování

38. ročník | **KSP** | **Září 2025**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám první číslo hlavní kategorie 38. ročníku KSP.



Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

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 (hlavní 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, nálepku na notebook a možná i další překvapení.

Termín série: **neděle 2. listopadu 2025 ve 32:00 (tedy další ráno v 8:00)**

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

Značky úloh:
- Lehčí úloha (či její část) vhodná pro začátečníky
- Praktická open-data úloha
- Úloha, u které doporučujeme začíst se do kuchařky
- Seriálová úloha
- Experimentální (neobvyklá) úloha

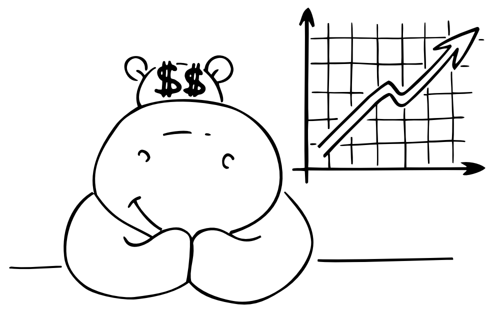
Odměna série: **Odznáček do profilu na webu** si vyslouží ten, kdo **z každé úlohy série získá alespoň 2 body**.

První série třicátého osmého ročníku KSP

38-1-1 One-shot burza — 11 bodů

Trpaslík Plesnivous se rozhodl, že drakův poklad je pro něj moc práce a musí si vydělat jinak. Jenže vydělávat standardní cestou je zaprvé náročné, zadruhé neefektivní a zatřetí nic pro něj. Proto se rozhodl jít za Lady Hrochtenzií – místním orákulem. Ta mu sdělila cenu uhlí pro každý z příštích N dní.

Bohužel si ale důlní úřad kontroluje velké finančně-uhelné transakce, proto smí Plesnivous za svých K sesterciů pouze jednou nakoupit libovolné kladné celé množství uhlí a jednou ho všechno prodat. Chce při tom postupovat se značnou obezřetností, aby si vydělal co nejvyšší sumu. Pomozte Plesnivousovi zjistit, kdy má uhlí nakoupit a kdy prodat, aby skončil s co nejvíce sestercii.



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 dostanete N, počet dní, na které Lady Hrochtenzie předpověděla ceny uhlí, a K, Plesnivousův kapitál v sesterciích. Na druhém řádku dostanete N kladných celých čísel reprezentující ceny uhlí v daných dnech.

Formát výstupu: Na jednom řádku vypište dvě čísla – den, ve který má Plesnivous uhlí nakoupit a kdy prodat. Dny indexujeme jako správní trpaslíci od nuly. Pokud je takových řešení více, vypište libovolné z nich. Pokud Plesnivous na uhlí nemůže nic vydělat, nebo by dokonce prodělal, vypište namísto toho `-1`. Pozor, Plesnivousův výdělek se nemusí vejít do 32-bitového čísla.

Ukázkový vstup:
```
8 6
3 7 2 4 9 1 5 4
```

Ukázkový výstup:
```
5 6
```

Když by Plesnivous nakoupil v den 5, nakupoval by uhlí za 1 sestercius, tedy by jich mohl koupit 6. 6 uhlí potom v den 6 prodá každé za 5 sestercií, čímž získá peněžní obnos v hodnotě $6 \cdot 5 = 30$ sestercií. Vydělal tedy $30 - 6 = 24$ sestercií, což je pro daný vstup maximální.

38-1-2 Bambulus — 12 bodů

Matúšovi se konečně podařilo rozjet jeho kariéru motivačního řečníka. O jeho sérii přednášek „Bambulus: tři pilíře životního štěstí?“ je nebývalý zájem a nezřídka bývají představení vyprodaná. Nově nabytá sláva s sebou však nese jistá úskalí: Matúšovi obdivovatelé se nemůžou shodnout na tom, který z pilířů jeho učení je nejdůležitější, a rozdělili se na tři navzájem rozhádané tábory. Jedni věří, že klíčem ke všemu je *ladění* (harmonie se sebou i okolím), druzí vidí hlavní smysl v *řádění* (následování svého vnitřního dítěte a žití v okamžiku), třetí nedají dopustit na *paradění* (hledání vnitřní i vnější krásy ve všedních věcech).

Tyto rozbroje jsou problém pro Matúšova vystoupení, jelikož příslušníci rozhádaných táborů si odmítají sednout vedle sebe. Po minulém představení, kde kvůli tomu zůstala půlka míst volná, už Matúšovi došla trpělivost. Přicházející diváci prostě dostanou nějaké místo přidělené, a basta. Jenže jak, aby byli pořád všichni spokojení, ale přitom volných míst zbylo co nejméně? Matúš je zaneprázdněn googlením motivačních citátů, a tak tento úkol připadl na vás.

Hlediště je tvořené N židlemi umístěnými v jedné řadě. Na začátku je hlediště prázdné a pak do něj po jednom přicházejí diváci. Vaším cílem je navrhnout algoritmus, který dostává události tvaru „právě přišel divák vyznávající 1./2./3. pilíř“ a na každou z nich odpovídá místem, na které si daný divák má sednout. Chceme přitom, aby vedle sebe nikdy neseděli dva diváci vyznávající různé pilíře.

Váš algoritmus je *online:* diváci přicházejí jeden po druhém, a nový divák přijde až potom, co umístíme předchozího. Již usazené diváky nejde přemisťovat.

Kvalitu vašeho řešení budeme hodnotit podle *ztráty* – množství volných míst, které zbývají v okamžiku, kdy už nejde umístit dalšího diváka. Stejně jako u časové složitosti nás zde nezajímá konkrétní číslo, ale pouze asymptotické chování: zdali je volných míst $\Theta(N)$, $\Theta(\sqrt{N})$, nebo nějaká jiná funkce v N. Počet volných míst vyhodnocujeme na nejhorším možném vstupu. Můžete si představit, že před vchodem stojí Ríša, který Matúšovi nepřeje úspěch, a který dovnitř pouští diváky v takovém pořadí, aby váš algoritmus selhal co nejdříve. Ríša zná zdrojový kód vašeho algoritmu a má opravdu hodně času na vymýšlení co nejzapeklitějšího vstupu.

Příklad: Uvažme algoritmus, který každého přicházejícího diváka umístí na co nejlevější použitelné místo. Takový algoritmus nechá až $N/2$ míst volných, a to v situaci, kdy se nám přicházejí střídavě vyznavači 1. a 2. pilíře – pak budou na sudých místech sedět diváci a lichá místa budou prázdná. Naopak si jde rozmyslet, že více než $N/2$ volných míst nikdy nezůstane. Ztráta tohoto algoritmu je tedy $\Theta(N)$.

Cílíme na algoritmus, který v nejhorším případě ponechá $\Theta(\log N)$ míst volných, ale částečné body získáte i za asymptoticky horší závislost na N (třeba $\Theta(\sqrt{N})$). Ve vašem řešení nezapomeňte zdůvodnit, proč je jeho ztráta zrovna taková, jaká tvrdíte.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

38-1-3 Mediánový poddaný — 14 bodů

Za sedmero horami a sedmero řekami vládl moudrý král Ríša. Ten si udržoval svoje rádce, kteří mu každoročně počítali a plánovali chod království. Nicméně aby rádci mohli připravit své plány, potřebovali by sehnat reprezentativního poddaného…

Ríšovo království je rozděleno na K krajů. Protože Ríša má rád pravidelnost, v každém kraji je právě N poddaných, kde každý poddaný má unikátní celočíselnou pracovitost P_i ($1 \leq P_i \leq P_{\max}$). Rádci by chtěli sehnat poddaného, pro něhož se počty poddaných s vyšší pracovitostí a s nižší liší nejvýš o 1. Tedy poddaného, který je mediánem všech poddaných v celém království.

Nicméně Ríšovo království je velmi velké a rozlehlé, takže se záznamy o pracovitosti poddaných udržují pouze v krajích, kde žijí. Ríša ale může vyslat svého věrného posla s *konstantně** dlouhým rozkazem do jednoho z krajů. Tam může zkonzultovat archivy a vrátit se s *konstantně* dlouhým výsledkem. Mezi vysláním posla a jeho návratem uplyne vždy jeden týden. Na základě všech předchozích výsledků může poté Ríša posla vyslat do dalšího kraje.

Například můžeme poslat posla:

- Zjistit nejmenší pracovitost v daném kraji.
- Zjistit počet poddaných s pracovitostmi mezi A a B.

Ale nemůžeme:

- Vyslat ho se seznamem největších pracovitostí v každém kraji, neboť rozkaz je dlouhý $\Theta(K)$.
- Nechat si poslat seznam $N/2$ nejméně pracovitých poddanných v kraji, neboť výsledek je dlouhý $\Theta(N)$.

Pomozte Ríšovi zjistit medián pracovitosti. Vaše řešení budou hodnocena primárně podle celkového počtu vyslání Ríšova věrného posla. Při návrhu algoritmu předpokládejte, že K je řádově menší než N, a to je řádově menší než $P_{\max}$.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

38-1-4 Jedno slovo — 8 bodů

Kevinovi přišla s novou sérií od orgů KSP i nějaká podivně vypadající zpráva… a kdyby jen jedna!

Všiml si, že každý vstup obsahuje na prvním řádku číslo vstupu, zbytku ale nerozumí a potřebuje vaši pomoc.

Nalezněte řešení.

Rada: pokud jste ještě nikdy neluštili šifry, může se vám hodit přečíst si třeba začátek tohoto návodu,[1] po část „Kódování znaků“. Nehledejte však v řešení veliké složitosti, orgové KSP jsou především informatici a spíš než vlajková abeceda je zajímají způsoby kódování, které s informatikou přirozeně souvisejí.

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

* Konstantně dlouhá zpráva smí obsahovat jen konstantně mnoho „rozumně velkých“ čísel – čísel velkých nejvýše polynomiálně k číslům na vstupu. Délka zprávy v bitech tedy musí být v $\mathcal{O}(\log K + \log N + \log P_{\max})$.

[1] `https://technoplaneta.cz/2019/informace/jak-lustit-sifry`

38-1-S Hoďte si kostkou 15 bodů

V letošním seriálu se spolu vydáme do říše *pravděpodobnostních algoritmů.* To jsou takové, které si během výpočtu „hází kostkou“ – tedy generují nějaká náhodná čísla. To dokáže být překvapivě užitečné. Často jsou takové algoritmy daleko jednodušší, nebo dokonce rychlejší (aspoň v průměru) než jejich determinističtí (nenáhodní) příbuzní.

Abychom uměli o náhodě uvažovat, bude se nám hodit matematická *teorie pravděpodobnosti.* O té už v KSPčku máme kuchařku (přiloženou na konci tohoto letáku). Před řešením úloh si ji prosím přečtěte.

Dnešní díl seriálu se bude týkat generování náhody. Budeme používat:

- *Poctivé mince* – ty generují náhodný bit `0` nebo `1` tak, že obě možnosti mají stejnou pravděpodobnost $1/2$.
- *Obecné mince* – ty generují `1` s nějakou obecnou pravděpodobností p a `0` s pravděpodobností $1 - p$.
- *Poctivé k-stěnné kostky* – na nich padají čísla od 0 do $k-1$, všechna se stejnou pravděpodobností $1/k$. Poctivá mince je tedy ekvivalentní s poctivou 2-stěnnou kostkou. Na konkrétním rozsahu hodnot samozřejmě nezáleží, stejně dobře si můžeme pořídit kostku třeba s čísly 1 až k nebo 5 až $k + 4$.
- *Obecné k-stěnné kostky* – jednotlivá čísla mají pravděpodobnosti $p_0, p_1, \ldots, p_{k-1}$, přičemž $p_0 + \ldots + p_{k-1} = 1$.

Stačí ovšem mít k dispozici libovolnou kostku nebo minci, a hned umíme simulovat všechny ostatní. Nejdřív dokážeme, že pomocí poctivé mince dokážeme simulovat poctivou k-stěnnou kostku pro libovolné $k \geq 2$. Simulací myslíme algoritmus, který bude mít k dispozici funkci na hod mincí a jeho výstupem bude hod kostkou.

Kdyby počet stěn k byl mocnina dvojky 2^b pro nějaké $b > 0$, stačilo by b-krát hodit mincí a získané bity přečíst jako číslo ve dvojkové soustavě. Takto budou mít všechna čísla od 0 do $k - 1$ stejnou pravděpodobnost $1/2^b = 1/k$. (Například pro $k = 8$ bude $b = 3$, hodíme mincemi `101` a výsledek bude 5.)

Pokud k není mocnina dvojky, najdeme si nejbližší mocniny dvojky, tedy $2^{b-1} < k < 2^b$. Pak stejně jako předtím b-krát hodíme mincí, a tím získáme rovnoměrně náhodné číslo x od 0 do $2^b - 1$ (*rovnoměrně náhodné* znamená, že všechny možnosti jsou stejně pravděpodobné).

Je-li $x < k$, máme vyhráno a oznámíme výsledek x. V případě $x \geq k$ se nabízí oznámit $x \bmod k$, ale pozor, to není rovnoměrné: například pro $k = 5$ je $b = 3$ a výsledek 0 oznámíme jak pro hody `000` ($x = 0$), tak pro `101` ($x = 5$). Výsledek 0 má tedy pravděpodobnost $2/8 = 1/4$ místo požadované $1/5$.

Lepší je v případě $x \geq k$ prostě na všechno zapomenout a házet znovu. Když i napodruhé bude $x \geq k$, zkusíme to potřetí atd. Tím zajistíme rovnoměrnou náhodnost, ale v nejhorším případě selže *každý* pokus a algoritmus se nikdy nezastaví. Dokážeme ovšem, že průměrný počet hodů mincí (a tím i průměrná časová složitost) jsou konečné.

Spočítejme pravděpodobnost, že se pokus podaří (tedy bude $x < k$). To nastane v k případech z 2^b, takže pravděpodobnost úspěchu bude $p = k/2^b$. Jelikož $k > 2^{b-1} = 2^b/2$, bude $p > 1/2$. Podle lemmatu o džbánu z naší kuchařky vychází střední hodnota počtu pokusů do prvního úspěšného $1/p < 2$. Každý pokus spotřebuje b hodů mincí, takže náš algoritmus potřebuje v průměru (ve střední hodnotě) méně než $2b$ hodů mincí. To je nejvýše $2 \log_2 k + 2$, jelikož $b \leq \log_2 k + 1$. Pokud nám stačí asymptotický odhad, je to prostě $\mathcal{O}(\log k)$ hodů v průměru.

To je poměrně typická situace: náš pravděpodobnostní algoritmus je v průměru rychlý, ale v nejhorším případě velmi pomalý – máme-li smůlu, dokonce se nemusí zastavit. Podobně Quicksort s náhodnou volbou pivota, zmiňovaný v kuchařce, má průměrnou složitost $\mathcal{O}(n \log n)$, ale v nejhorším případě kvadratickou.

Úkol 1 [4b]: Ukažte, jak pomocí poctivé k-stěnné kostky simulovat poctivou ℓ-stěnnou.

Úkol 2 [3b]: Ukažte, jak pomocí obecné mince simulovat poctivou. Pravděpodobnost jedničky na obecné minci ovšem neznáte.

Úkol 3 [4b]: Ukažte, jak pomocí poctivé mince simulovat obecnou. Pravděpodobnost p jedničky na obecné minci dostanete na vstupu. Je to nějaké reálné číslo mezi 0 a 1. Můžete předpokládat, že váš počítač umí provádět aritmetické operace s reálnými čísly v konstantním čase.

Úkol 4 [4b]: Dokažte, že neexistuje algoritmus, který by pomocí poctivé mince simuloval poctivou 3-stěnnou kostku, a měl konečnou časovou složitost v nejhorším případě.

Vaše řešení úkolů 1–3 by mělo obsahovat algoritmus, důkaz správnosti (tedy že všechny možné výsledky mají požadované pravděpodobnosti) a výpočet složitosti v nejhorším případě a v průměru.

Recepty z programátorské kuchařky: Základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[2]

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[3]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu ($+, -, *, /$).
- Vyhodnocení nějaké konkrétní podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáme s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[4]

Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme

[2] `http://ksp.mff.cuni.cz/study/odkazy.html`

[3] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

[4] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit například při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, je snadné spočítat umístění konkrétní přihrádky. Proto když se počítače zeptáme na obsah přihrády `pole[42]`, vrátí nám hodnotu ihned.

Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[5] nejdříve však doporučujeme dočíst tuto kuchařku.

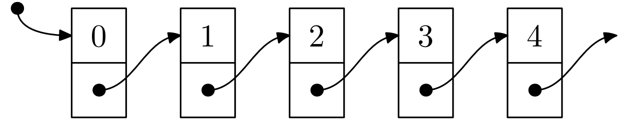
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N (čili pole obsahující N prvků) trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

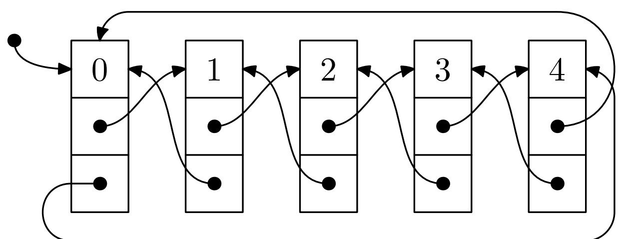


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
```

[5] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

```
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}

// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n", aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def vypis(self, aktualni):
        if aktualni is not None:
            print(aktualni.hodnota)
            self.vypis(aktualni.dalsi)

    def vloz_po(self, prvek, za_prvek = None):
        if za_prvek is not None:
            prvek.dalsi = za_prvek.dalsi
            prvek.predchozi = za_prvek
            za_prvek.dalsi = prvek

        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = prvek

        if self.koren is None:
            self.koren = prvek

    def odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.vloz_po(prvekB)
seznam.vloz_po(prvekD, prvekB)
seznam.vloz_po(prvekC, prvekD)
seznam.vloz_po(prvekA, prvekC)
seznam.odstran(prvekC)

seznam.vypis(seznam.koren)
```

Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, také první přijde na řadu. Trochu jinak si ji můžeme představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru do něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu, konkrétně na poslední prvek.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást určitých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce

vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

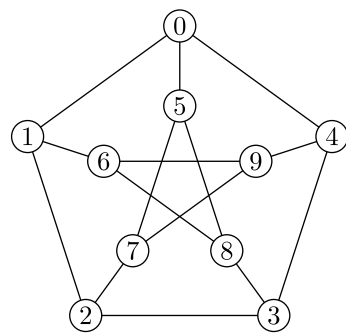
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy" a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (v popisech bude n značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (nebo jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.
- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je ale dobré o ní vědět.

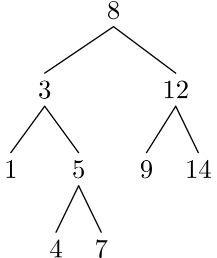
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[6]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný

[6] `http://ksp.mff.cuni.cz/kucharky/`

strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, jeho synové jsou právě vrcholy s indexy $2i+1$ a $2i+2$. Do pole níže je zapsaný binární strom z obrázku výše.

index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromu menší než hodnota tohoto vrcholu a hodnoty v jeho pravém podstromu naopak větší.

V takovém stromu pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v sekci *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, …) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Třeba stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Kromě rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každé volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit), a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, … pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

V Pythonu:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n == 0) return 0;

        int a = 0; int b = 1;
        while (n > 1) {
                int pomocna = a + b;
                a = b;
                b = pomocna;
                n--;
        }
```

```
        return b;
}
```

V Pythonu:

```
def fib2(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        (a, b) = (b, a+b)
        n -= 1
    return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do podkapitolky Předpočítané mezivýsledky).

Rekurzivní varianta v tomto případě může běžet až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$. Dá se ale celkem lehce rozmyslet úprava rekurzivní varianty, aby běžele také v $\mathcal{O}(n)$, zkuste si rozmyslet jak.

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
    if castka == 0:
        return True
    elif castka < 0:
        return False
    elif rozloz(castka-5):
        print(" 5 Kc")
        return True
    elif rozloz(castka-3):
        print(" 3 Kc")
        return True
    else:
        return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě třikorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představme si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, maximálně po $\log n$ krocích se dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[7]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;
```

[7] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

```
do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```

Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:

```
def bin_vyhled(pole, hledane,
               levy_index=0, pravy_index=None):
    if pravy_index is None:
        pravy_index = len(pole)
    while levy_index < pravy_index:
        prostredni = (levy_index +
                      pravy_index) // 2
        x = pole[prostredni]
        if x < hledane:
            levy_index = prostredni + 1
        elif x > hledane:
            pravy_index = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print(bin_vyhled([1,2,5,7, 8,12,16,42], 1))
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

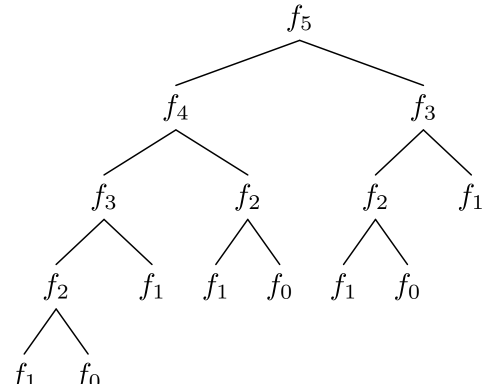
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[8]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme připomenout naši rekurzivní implementaci počítání Fibonacciho čísel zmíněnou výše.

Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

Dynamické programování

Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[9]

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 $(4, 5$ a $2, 7)$. Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat sou-

[8] http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj
[9] http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani

čet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

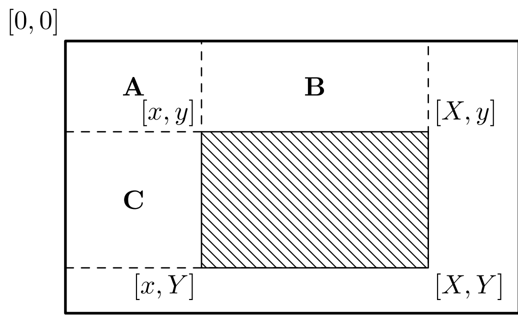
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A+B$ a jednou $A+C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, předvýpočet má smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, nejedná se již o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

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 $\emptyset$ (prázdná množina, 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.

$$\begin{aligned} P(A \cup B \cup C) = P((A \cup B) \cup C) &\leq P(A \cup B) + P(C) \\ &\leq P(A) + P(B) + P(C). \end{aligned}$$

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 jaké části z nich nastal jev B. Z nich si vybereme ty, v nichž nastal i jev A. Ty tvoří část $P(A \cap B)$ 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:

$$\begin{aligned} A &= \text{„první číslice je 1"}, \\ B &= \text{„druhá číslice je 1"}, \\ C &= \text{„číslo je menší než 30"}. \end{aligned}$$

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

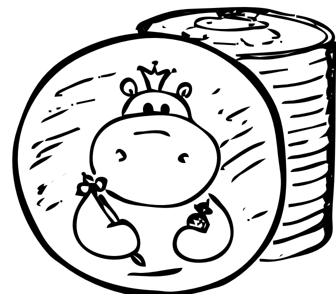
Definovat nezávislost pro více jevů je složitější, obecně nestačí říci, že pravděpodobnost toho, že nastanou současně, je součin jednotlivých pravděpodobností. Je potřeba, aby to platilo pro kteroukoliv podmnožinu jevů.

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

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:

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

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 $G = 1/(1-q)$.)

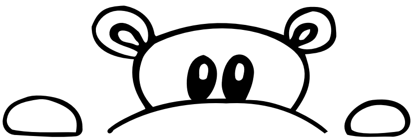
Odbočka k hledání pivota

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.[10] 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 $1/N^2$.

Pár slov závěrem

Pokud se chcete dozvědět o pravděpodobnosti více, můžeme vřele doporučit skvělé přednášky z Harvardu. Jejich videozáznamy jsou dostupné na YouTube.[11]

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.[12]

Další jednoduché randomizované algoritmy a datové struktury najdete v knížce Průvodce labyrintem algoritmů.[13] Inspirovat vás také může seriál z 16. ročníku KSP.[14]

Martin „Medvěd" Mareš & Jakub Tětek

[10] `http://mj.ucw.cz/papers/numth.pdf`
[11] `https://www.youtube.com/playlist?list=PL2SOU6wwxB0uwwH80KTQ6ht66KWxbzTIo`
[12] `https://en.wikipedia.org/wiki/Probability_axioms`
[13] `http://pruvodce.ucw.cz/`
[14] `https://ksp.mff.cuni.cz/h/ulohy/16/`

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*