1. Randomizace

Výhodou algoritmů je, že se na ně můžeme spolehnout. Jsou to dokonale deterministické předpisy, které pro zadaný vstup pokaždé vypočítají tentýž výstup. Přesto může být zajímavé vpustit do nich pečlivě odměřené množství náhody. Získáme tím takzvané pravděpodobnostní neboli randomizované algoritmy. Ty mnohdy dovedou dospět k výsledku daleko rychleji než jejich klasičtí příbuzní. Přitom budeme stále schopní o jejich chování ledacos dokázat.

Randomizace nám pomůže například s výběrem pivota v algoritmech Quicksort a Quickselect z předchozí kapitoly. Také zavedeme hešovací tabulky – velice rychlé datové struktury založené na náhodném rozmisťování hodnot do přihrádek.

1.1. Pravděpodobnostní algoritmy

Nejprve rozšíříme definici algoritmu z kapitoly o složitosti, aby umožňovala náhodný výběr. Zařídíme to tak, že do výpočetního modelu RAM doplníme novou instrukci X `:= random(`Y`,`Z`)`, kde X je odkaz do paměti a Y a Z buďto odkazy do paměti, nebo konstanty.

Kdykoliv stroj při provádění programu narazí na tuto instrukci, vygeneruje náhodné celé číslo v rozsahu od Y do Z a uloží ho do paměťové buňky X. Všechny hodnoty z tohoto rozsahu budou stejně pravděpodobné a volba bude nezávislá na předchozích voláních instrukce `random`. Bude-li $Y > Z$, program skončí běhovou chybou podobně, jako kdyby dělil nulou.

Ponechme stranou, zda je možné počítač s náhodným generátorem skutečně sestrojit. Ostatně, samu existenci náhody v našem vesmíru nejspíš nemůžeme nijak prokázat. Je to tedy spíš otázka víry. Teoretické informatice na odpovědi nezáleží – prostě předpokládá výpočetní model s ideálním náhodným generátorem, který se řídí pravidly teorie pravděpodobnosti. V praxi si pak pomůžeme generátorem *pseudonáhodným,* který generuje prakticky nepředvídatelná čísla. Dodejme ještě, že existují i jiné modely náhodných generátorů, než je naše funkce `random`, ale ty ponechme do cvičení.

Algoritmům využívajícím náhodná čísla se obvykle říká *pravděpodobnostní* nebo *randomizované.* Výpočet takového algoritmu pro konkrétní vstup může v závislosti na náhodě trvat různě dlouho a dokonce může dospět k různým výsledkům. Doba běhu, případně výsledek pak nejsou konkrétní čísla, ale náhodné veličiny. U těch se obvykle budeme ptát na střední hodnotu, případně na pravděpodobnost, že překročí určitou mez.

Připomenutí teorie pravděpodobnosti

Pro analýzu randomizovaných algoritmů budeme používat aparát matematické teorie pravděpodobnosti. Zopakujme si její základní pojmy a tvrzení. Pokud jste se s nimi dosud nesetkali, naleznete je například v Kapitolách z diskrétní matematiky [**?**].

Budeme pracovat s *diskrétním pravděpodobnostním prostorem* (Ω, P). Ten je tvořen nejvýše spočetnou množinou Ω *elementárních jevů* a funkcí $P : \Omega \to [0,1]$, která elementárním jevům přiřazuje jejich *pravděpodobnosti.* Součet pravděpodobností všech elementárních jevů je roven 1. *Jev* je obecně nějaká množina elementárních jevů $A \subseteq \Omega$. Funkci P můžeme přirozeně rozšířit na všechny jevy: $P(A) = \sum_{e \in A} P(e)$. Pravděpodobnosti můžeme také připisovat výrokům: $\Pr[\varphi(x)]$ je pravděpodobnost jevu daného množinou všech elementárních jevů x, pro které platí výrok $\varphi(x)$.

Pro každé dva jevy A a B platí $P(A \cup B) = P(A) + P(B) - P(A \cap B)$. Pokud $P(A \cap B) = P(A) \cdot P(B)$, řekneme, že A a B jsou *nezávislé.* Pro více jevů $A_1, \ldots, A_k$ rozlišujeme *nezávislost po dvou* (každé dva jevy A_i a A_j jsou nezávislé) a *plnou nezávislost* (pro každou podmnožinu indexů $\emptyset \neq I \subseteq \{1, \ldots, k\}$ platí $P(\cap_{i \in I} A_i) = \prod_{i \in I} P(A_i)$).

Náhodné veličiny (proměnné) jsou funkce z Ω do reálných čísel, přiřazují tedy reálné hodnoty možným výsledkům pokusu. Můžeme se ptát na pravděpodobnost, že veličina má nějakou vlastnost, například $\Pr[X > 5]$. *Střední hodnota* $\mathbb{E}[X]$ náhodné veličiny X je průměr všech možných hodnot vážený jejich pravděpodobnostmi, tedy $\sum_{x \in \mathbb{R}} x \cdot \Pr[X = x] = \sum_{e \in \Omega} X(e) \cdot P(e)$. Často budeme používat následující vlastnost střední hodnoty:

Fakt (linearita střední hodnoty): Nechť X a Y jsou náhodné veličiny a α a β reálná čísla. Potom $\mathbb{E}[\alpha X + \beta Y] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y]$.

Tak dlouho se chodí se džbánem ...

Než se pustíme do pravděpodobnostní analýzy algoritmů, začneme jednoduchým příkladem: budeme chodit se džbánem pro vodu tak dlouho, než se utrhne ucho. To při každém pokusu nastane náhodně s pravděpodobností p $(0 < p < 1)$, nezávisle na výsledcích předchozích pokusů. Kolik pokusů v průměru podnikneme?

Označme T počet kroků, po kterém dojde k utržení ucha. To je nějaká náhodná veličina a nás bude zajímat její střední hodnota $\mathbb{E}[T]$. Podle definice střední hodnoty platí:

$$\mathbb{E}[T] = \sum_i \left(i \cdot \Pr[\text{ucho se utrhne při } i\text{-tém pokusu}]\right) =$$

$$= \sum_i \left(i \cdot (1-p)^{i-1} \cdot p\right).$$

U každé nekonečné řady se sluší nejprve zeptat, zda vůbec konverguje. Na to nám kladně odpoví třeba podílové kritérium. Nyní bychom mohli řadu poctivě sečíst, ale místo toho použijeme jednoduchý trik založený na linearitě střední hodnoty.

V každém případě provedeme první pokus. Pokud se ucho utrhne (což nastane s pravděpodobností p), hra končí. Pokud se neutrhne (pravděpodobnost $1 - p$), dostaneme se do přesně stejné situace, jako předtím – náš ideální džbán totiž nemá žádnou paměť. Z toho vyjde následující rovnice pro $\mathbb{E}[T]$:

$$\mathbb{E}[T] = 1 + p \cdot 0 + (1-p) \cdot \mathbb{E}[T].$$

Vyřešíme-li ji, dostaneme $\mathbb{E}[T] = 1/p$. (Zde jsme nicméně potřebovali vědět, že střední hodnota existuje a je konečná, takže úvahy o nekonečných řadách byly nezbytné.)

Tento výsledek se nám bude často hodit, formulujme ho proto jako lemma:

Lemma (o džbánu): Čekáme-li na náhodný jev, který nastane s pravděpodobností p, dočkáme se ve střední hodnotě po $1/p$ pokusech.

Cvičení

1. *Ideální mince:* Mějme počítač, jehož náhodným generátorem je ideální mince. Jinými slovy, máme instrukci `random_bit`, ze které na každé zavolání vypadne jeden rovnoměrně náhodný bit vygenerovaný nezávisle na předchozích bitech. Jak pomocí takové funkce generovat rovnoměrně náhodná přirozená čísla od 1 do n? Minimalizujte průměrný počet hodů mincí.
2. Ukažte, že v předchozím cvičení nelze počet hodů mincí v nejhorším případě nijak omezit, leda kdyby n bylo mocninou dvojky.
3. *Míchání karet:* Popište algoritmus, který v lineárním čase vygeneruje náhodnou permutaci množiny $\{1, 2, \ldots, n\}$.
4. V mnoha programovacích jazycích je k dispozici funkci `random`, která nám vrátí rovnoměrně (pseudo)náhodné číslo z pevně daného intervalu. Lidé ji často používají pro generování čísel v rozsahu od 0 do $n-1$ tak, že spočtou `random` mod n. Jaký se v tom skrývá háček? Jak ho obejít?
5. *Náhodná k-tice:* Máme-li obrovský soubor a chceme o něm získat alespoň hrubou představu, hodí se prozkoumat náhodnou k-tici řádků. Vymyslete algoritmus, který ji vybere tak, aby všechny k-tice měly stejnou pravděpodobnost. Vstup se celý nevejde do paměti a jeho velikost ani předem neznáme; k-tice se do paměti spolehlivě vejde. Zvládnete to na jediný průchod daty?

6.* *Míchání podruhé:* Vasil Vasiljevič míchá karty takto: připraví si n prázdných přihrádek. Pak postupně umisťuje čísla $1, \ldots, n$ do přihrádek tak, že vždy vybere rovnoměrně náhodně přihrádku a pokud v ní již něco je, vybírá znovu. Kolik pokusů bude v průměru potřebovat?

7. Lemma o džbánu můžeme dokázat i sečtením uvedené nekonečné řady. Ta je ostatně podobná řadě, již jsme zkoumali při rozboru konstrukce haldy v oddílu **??**. Zkuste to.
8. Představte si, že hodíme 10 hracími kostkami a počty ok sečteme. V jakém pravděpodobnostním prostoru se tento pokus odehrává? O jakou náhodnou veličinu jde? Jak stanovit její střední hodnotu?
9. V jakém pravděpodobnostním prostoru se odehrává lemma o džbánu?

1.2. Náhodný výběr pivota

U algoritmů založených na výběru pivota (Quickselect a Quicksort) jsme spoléhali na to, že pokud budeme pivota volit náhodně, bude se algoritmus „chovat dobře.“ Nyní nastal čas říci, co to přesně znamená, a přednést důkaz.

Mediány, skoromediány a Quickselect

Úvaha o džbánu nám dává jednoduchý algoritmus, pomocí kterého umíme najít skoromedián posloupnosti n prvků: Vybereme si *rovnoměrně náhodně* jeden z prvků posloupnosti; tím *rovnoměrně* myslíme tak, aby všechny prvky měly stejnou pravděpodobnost. Pak ověříme, jestli vybraný prvek je skoromedián. Pokud ne, na vše zapomeneme a postup opakujeme.

Kolik pokusů budeme potřebovat, než algoritmus skončí? Skoromediány tvoří alespoň polovinu prvků, tedy pravděpodobnost, že se do nějakého strefíme, je minimálně $1/2$. Podle lemmatu o džbánu tedy střední hodnota počtu pokusů bude nejvýše 2. (Počet pokusů v nejhorším případě ovšem neumíme omezit nijak – při dostatečné smůle můžeme stále vybírat nejmenší prvek. To ovšem nastane s nulovou pravděpodobností.)

Nyní už snadno spočítáme časovou složitost našeho algoritmu. Jeden pokus trvá $\Theta(n)$, střední hodnota počtu pokusů je $\Theta(1)$, takže střední hodnota časové složitosti je $\Theta(n)$. Obvykle budeme zkráceně mluvit o *průměrné* časové složitosti.

Pokud tento výpočet skoromediánu použijeme v Quickselectu, získáme algoritmus pro hledání k-tého nejmenšího prvku s průměrnou složitostí $\Theta(n)$. Dobrá, tím jsme se ale šalamounsky vyhnuli otázce, jakou průměrnou složitost má původní Quickselect s rovnoměrně náhodnou volbou pivota.

Tu odhadneme snadno: Rozdělíme běh algoritmu na *fáze*. Fáze bude končit v okamžiku, kdy za pivota zvolíme skoromedián. Fáze se skládá z *kroků* spočívajících v náhodné volbě pivota, lineárně dlouhém výpočtu a zahození části vstupu. Už víme, že skoromedián se průměrně podaří najít za dva kroky, tudíž jedna fáze trvá v průměru lineárně dlouho. Navíc si všimneme, že během každé fáze se vstup zmenší alespoň o čtvrtinu. K tomu totiž stačil samotný poslední krok, ostatní kroky mohou situaci jedině zlepšit. Průměrnou složitost celého algoritmu pak vyjádříme jako součet průměrných složitostí jednotlivých fází: $\Theta(n)+\Theta((3/4)\cdot n)+\Theta((3/4)^2\cdot n)+\ldots=\Theta(n)$.

Indikátory a Quicksort

Podíváme-li se na výpočet v minulém odstavci s odstupem, všimneme si, že se opírá zejména o linearitu střední hodnoty. Časovou složitost celého algoritmu jsme vyjádřili jako součet složitostí fází. Přitom fázi jsme nadefinovali tak, aby se již chovala dostatečně průhledně.

Podobně můžeme analyzovat i Quicksort, jen se nám bude hodit složitost rozložit na daleko více veličin. Mimo to si všimneme, že Quicksort na každé porovnání provede jen $\mathcal{O}(1)$ dalších operací, takže postačí odhadnout počet provedených porovnání.

Očíslujeme si prvky podle pořadí v setříděné posloupnosti $y_1,\ldots,y_n$. Zavedeme náhodné veličiny C_{ij} pro $1\le i<j\le n$ tak, aby platilo $C_{ij}=1$ právě tehdy, když během výpočtu došlo k porovnání y_i s y_j. V opačném případě je $C_{ij}=0$. Proměnným, které nabývají hodnoty 0 nebo 1 podle toho, zda nějaká událost nastala, se obvykle říká *indikátory*. Počet všech porovnání je tudíž roven součtu všech indikátorů C_{ij}. (Zde využíváme toho, že Quicksort tutéž dvojici neporovná vícekrát.)

Zamysleme se nyní nad tím, kdy může být $C_{ij} = 1$. Algoritmus porovnává pouze s pivotem, takže jedna z hodnot y_i, y_j se těsně předtím musí stát pivotem. Navíc všechny hodnoty $y_{i+1}, \ldots, y_{j-1}$ se ještě pivoty stát nesměly, jelikož jinak by y_i a y_j už byly rozděleny v různých částech posloupnosti. Jinými slovy, C_{ij} je rovno jedné právě tehdy, když se z hodnot $y_i, y_{i+1}, \ldots, y_j$ stane jako první pivotem buď y_i nebo y_j. A poněvadž pivota vybíráme rovnoměrně náhodně, má každý z prvků $y_i, \ldots, y_j$ stejnou pravděpodobnost, že se stane pivotem jako první, totiž $1/(j-i+1)$. Proto $C_{ij} = 1$ nastane s pravděpodobností $2/(j-i+1)$.

Nyní si stačí uvědomit, že když indikátory nabývají pouze hodnot 0 a 1, je jejich střední hodnota rovna právě pravděpodobnosti jedničky, tedy také $2/(j-i+1)$. Sečtením přes všechny dvojice (i, j) pak dostaneme pro počet všech porovnání:

$$\mathbb{E}[C] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1} \le 2n \cdot \sum_{2 \le d \le n} \frac{1}{d}.$$

Nerovnost na pravé straně platí díky tomu, že rozdíly $j-i+1$ se nacházejí v intervalu $[2, n]$ a každým rozdílem přispěje nejvýše n různých dvojic (i, j). Poslední suma je tzv. harmonická suma, která sečte na $\Theta(\log n)$. Jelikož se s ní při analýze algoritmů potkáváme často, vyslovíme o ní samostatné lemma.

Lemma (o harmonických číslech): Pro součet harmonické řady $H_n = 1/1 + 1/2 + \ldots + 1/n$ platí $\ln n \le H_n \le \ln n + 1$.

Důkaz: Sumu odhadneme pomocí integrálu

$$I(n) = \int_1^n 1/x \, \mathrm{d}x = [\ln x]_1^n = \ln n - \ln 1 = \ln n.$$

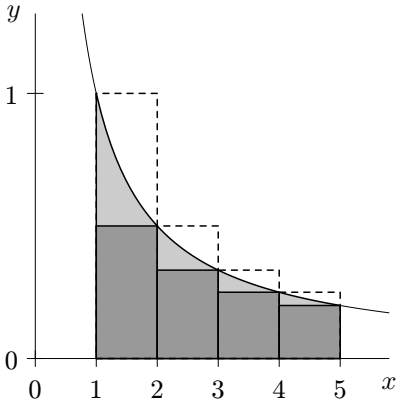
Sledujme obrázek 1.1. Funkce $I(n)$ vyjadřuje obsah plochy mezi křivkou $y = f(x)$, osou x a svislými přímkami $x = 1$ a $x = n$. Součástí této plochy je tmavé „schodiště“, jehož obsah je $1 \cdot (1/2) + 1 \cdot (1/3) + \ldots + 1 \cdot (1/n) = H_n - 1$. Proto musí platit $H_n - 1 \le I(n) = \ln n$, což je horní odhad z tvrzení lemmatu.

Dolní odhad dostaneme pomocí čárkovaného schodiště. Jeho obsah je $1 \cdot (1/1) + 1 \cdot (1/2) + \ldots + 1 \cdot (1/(n-1)) = H_n - 1/n$. Plocha měřená integrálem je součástí tohoto schodiště, pročež $\ln n = I(n) \le H_n - (1/n) \le H_n$. □

Důsledek: Střední hodnota časové složitosti Quicksortu s rovnoměrně náhodnou volbou pivota je $\mathcal{O}(n \log n)$.

Chování na náhodném vstupu

Když jsme poprvé přemýšleli o tom, jak volit pivota, všimli jsme si, že pokud volíme pivota pevně, náš algoritmus není odolný proti zlomyslnému uživateli. Takový uživatel může na vstupu zadat vychytrale sestrojenou posloupnost, která algoritmus donutí vybrat v každém kroku pivota nešikovně, takže poběží kvadraticky dlouho. Tomu jsme se přirozeně vyhnuli náhodnou volbou pivota – pro sebezlotřilejší vstup doběhneme v průměru rychle. Hodí se ale také vědět, že i pro pevnou volbu pivota je špatných vstupů málo.



Obr. 1.1: K důkazu lemmatu o harmonických číslech

Zavedeme si proto ještě jeden druh složitosti algoritmů, tentokrát opět deterministických (bez náhodného generátoru). Bude to *složitost v průměru přes vstupy.* Jinými slovy algoritmus bude mít pevný průběh, ale budeme mu dávat náhodný vstup a počítat, jak dlouho v průměru poběží.

Co to ale takový náhodný vstup je? U našich dvou problémů to docela dobře vystihuje *náhodná permutace* – vybereme si rovnoměrně náhodně jednu z $n!$ permutací množiny $\{1, 2, \ldots, n\}$.

Jak Quicksort, tak Quickselect se pak budou chovat velmi podobně, jako když měly pevný vstup, ale volily náhodně pivota. Pokud je na vstupu rovnoměrně náhodná permutace, je její prostřední prvek rovnoměrně náhodně vybrané číslo z množiny $\{1, 2, \ldots, n\}$. Vybereme-li ho za pivota a rozdělíme vstup na levou a pravou část, obě části budou opět náhodné permutace, takže se na nich algoritmus bude opět chovat tímto způsobem. Můžeme tedy analýzu z této kapitoly použít i na tento druh průměru se stejným výsledkem.

Cvičení

1. Průměrnou časovou složitost Quicksortu lze spočítat i podobnou úvahou, jakou jsme použili u Quickselectu. Uvažujte pro každý prvek, kolika porovnání se účastní a jak se mění velikosti úseků, v nichž se nachází.

2.* Ještě jeden způsob, jak analyzovat průměrnou složitost Quicksortu, je použitím podobné úvahy jako v důkazu Lemmatu o džbánu. Sestavíme rekurenci pro průměrný počet porovnání: $R(n) = n + \frac{1}{n}\sum_{i=1}^{n}(R(i-1) + R(n-i))$, $R(0) = R(1) = 0$. Dokažte indukcí, že $R(n) \leq 4n \ln n$.

3. *Náhodné stromy:* Uvažujme binární vyhledávací strom, který vznikl postupným vkládáním hodnot $1, \ldots, n$ v náhodném pořadí, bez jakéhokoliv vyvažování. Dokažte, že střední hodnota průměrné hloubky vrcholu je $\mathcal{O}(\log n)$. (Pro jistotu: průměr je obyčejný aritmetický, nijak v něm nefiguruje náhoda; z těchto prů-

měrů pak počítáme střední hodnotu přes všechny možné průběhy algoritmu.) Napovíme, že náhodné stromy souvisí s možnými průběhy Quicksortu.

4. *Úsporný medián:* Nalezněte k-tý nejmenší z n prvků, máte-li k dispozici pouze paměť asymptoticky menší než k. Pokuste se dosáhnout lepší časové složitosti než $\Theta(kN)$.

5. *Eratosthenovo síto* je pradávný algoritmus na hledání prvočísel. Začne se seznamem čísel $2, \ldots, n$. V i-tém kroku zkontroluje číslo i: pokud není škrtnuté, nahlásí ho jako prvočíslo a vyškrtá všechny jeho násobky. Časová složitost tohoto algoritmu je poněkud překvapivě $\mathcal{O}(n \log \log n)$. Zkuste dokázat alespoň slabší odhad $\mathcal{O}(n \log n)$.

1.3. Hešování s přihrádkami

Lidé už dávno zjistili, že práci s velkým množstvím věcí si lze usnadnit tím, že je rozdělíme do několika menších skupin a každou zpracujeme zvlášť. Příklady najdeme všude kolem sebe: Slovník spisovného jazyka českého má díly A až M, N až Q, R až U a V až Ž. Katastrální úřady mají svou působnost vymezenu územím na mapě. Padne-li v Paříži smog, smí v některé dny do centra jezdit jenom auta se sudými registračními čísly, v jiné dny ta s lichými.

Informatici si tuto myšlenku také oblíbili a pod názvem *hešování* ji často používají k uchovávání dat.

Mějme nějaké univerzum $\mathcal{U}$ možných hodnot, konečnou množinu přihrádek $\mathcal{P} = \{0, \ldots, m-1\}$ a *hešovací funkci,* což bude nějaká funkce $h : \mathcal{U} \to \mathcal{P}$, která každému prvku univerza přidělí jednu přihrádku. Chceme-li uložit množinu prvků $X \subseteq \mathcal{U}$, rozstrkáme její prvky do přihrádek: prvek $x \in X$ umístíme do přihrádky $h(x)$. Budeme-li pak hledat nějaký prvek $u \in \mathcal{U}$, víme, že nemůže být jinde než v přihrádce $h(u)$.

Podívejme se na příklad: Univerzum všech celých čísel budeme rozdělovat do 10 přihrádek podle poslední číslice. Jako hešovací funkci tedy použijeme $h(x) = x \bmod 10$. Zkusíme uložit několik slavných letopočtů naší historie: 1212, 935, 1918, 1948, 1968, 1989:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		1212			935			1918	1989
								1948	
								1968	

Hledáme-li rok 2015, víme, že se musí nacházet v přihrádce 5. Tam je ovšem pouze 935, takže hned odpovíme zamítavě. Hledání roku 2016 je dokonce ještě rychlejší: přihrádka 6 je prázdná. Zato hledáme-li rok 1618, musíme prozkoumat hned 3 hodnoty.

Uvažujme obecněji: kdykoliv máme nějakou hešovací funkci, můžeme si pořídit pole p přihrádek, v každé pak „řetízek“ – spojový seznam hodnot. Tato jednoduchá datová struktura je jednou z možných forem *hešovací tabulky.*

Jakou má hešovací tabulka časovou složitost? Hledání, vkládání i mazání sestává z výpočtu hešovací funkce a projití řetízku v příslušné přihrádce. Pokud bychom uvažovali „ideální hešovací funkci“, kterou lze spočítat v konstantním čase a která zadanou n-prvkovou množinu rozprostře mezi m přihrádek dokonale rovnoměrně, budou mít všechny řetízky n/m prvků. Zvolíme-li navíc počet přihrádek $m = \Theta(n)$, vyjde konstantní délka řetízku, a tím pádem i časová složitost operací.

Praktické hešovací funkce

Ideální hešovací funkce patří spíše do kraje mýtů, podobně jako třeba ideální plyn. Přesto nám to nebrání hešování v praxi používat – ostřílení programátoři znají řadu funkcí, které se pro reálná data chovají „prakticky náhodně“. Autorům této knihy se osvědčily například tyto funkce:

- *Lineární kongruence:* $x \mapsto ax \bmod m$
  Zde m je typicky prvočíslo a a je nějaká dostatečně velká konstanta nesoudělná s m. Často se a nastavuje blízko $0.618m$ (další nečekaná aplikace zlatého řezu z oddílu **??**).
- *Vyšší bity součinu:* $x \mapsto \lfloor (ax \bmod 2^w)/2^{w-\ell} \rfloor$
  Pokud hešujeme w-bitová čísla do $m = 2^\ell$ přihrádek, vybereme w-bitovou lichou konstantu a. Pak pro každé x spočítáme ax, ořízneme ho na w bitů a z nich vezmeme nejvyšších ℓ. Vzhledem k tomu, že přetečení ve většině programovacích jazyků automaticky ořezává výsledek, stačí k vyhodnocení funkce jedno násobení a bitový posun.
- *Skalární součin:* $x_0, \ldots, x_{d-1} \mapsto (\sum_i a_i x_i) \bmod m$
  Chceme-li hešovat posloupnosti, nabízí se zahešovat každý prvek zvlášť a výsledky sečíst (nebo vyxorovat). Pokud prvky hešujeme lineární kongruencí, je heš celé posloupnosti její skalární součin s vektorem konstant, to vše modulo m. Pozor, nefunguje používat pro všechny prvky tutéž konstantu: pak by výsledek nezávisel na pořadí prvků.
- *Polynom:* $x_0, \ldots, x_{d-1} \mapsto \left(\sum_i a^i x_i\right) \bmod m$
  Tentokrát zvolíme jenom jednu konstantu a a počítáme skalární součin zadané posloupnosti s vektorem $(a^0, a^1, \ldots, a^{d-1})$. Tento typ funkcí bude hrát důležitou roli v Rabinově-Karpově algoritmu na vyhledávání v textu v oddílu **??**.

U všech čtyř funkcí je experimentálně ověřeno, že dobře hešují nejrůznější druhy dat. Nemusíme se ale spoléhat jen na pokusy: v oddílu 1.5 vybudujeme teorii, pomocí které o chování některých hešovacích funkcí budeme schopni vyslovit exaktní tvrzení.

Pokud chceme hešovat objekty nějakého jiného typu, nejprve je zakódujeme do čísel nebo posloupností čísel. U floating-point čísel se například může hodit hešovat jejich interní reprezentaci (což je nějaká posloupnost bytů, kterou můžeme považovat za jedno celé číslo).

Přehešovávání

Hešovací tabulky dovedou vyhledávat s průměrně konstantní časovou složitostí, použijeme-li $\Omega(n)$ přihrádek. Jaký počet přihrádek ale zvolit, pokud n předem neznáme? Pomůže nám technika amortizovaného nafukování pole z oddílu **??**.

Na počátku založíme prázdnou hešovací tabulku s nějakým konstantním počtem přihrádek. Kdykoliv pak vkládáme prvek, zkontrolujeme poměr $\alpha = n/m$ – tomu se říká *faktor naplnění* neboli *hustota* tabulky a chceme ho udržet shora omezený konstantou, třeba $\alpha \leq 1$. Pokud už je tabulka příliš plná, zdvojnásobíme m a všechny prvky přehešujeme. Jedno přehešování trvá $\Theta(n)$ a jelikož mezi každými dvěma přehešováními vložíme řádově n prvků, stačí, když každý prvek přispěje konstantním časem. Při mazání prvků můžeme tabulku zmenšovat, ale obvykle nevadí ponechat ji málo zaplněnou.

Sestrojili jsme tedy datovou strukturu pro reprezentaci množiny, která dokáže vyhledávat, vkládat i mazat v průměrně konstantním čase. Pokud předem neumíme odhadnout počet prvků množiny, je v případě vkládání a mazání tento čas pouze amortizovaný.

Cvičení

1. Mějme množinu přirozených čísel a číslo x. Chceme zjistit, zda množina obsahuje dvojici prvků se součtem x.
2. Chceme hešovat řetězce 8-bitových znaků. Výpočet můžeme zrychlit tak, že čtveřice znaků prohlásíme za 32-bitová čísla a zahešujeme posloupnost čtvrtinové délky. Naprogramujte takovou hešovací funkci a nezapomeňte, že délka řetězce nemusí být dělitelná čtyřmi.
3.* *Bloomův filtr* je datová struktura pro přibližnou reprezentaci množiny. Skládá se z pole bitů $B[1 \ldots m]$ a hešovací funkce h, která prvkům univerza přiřazuje indexy v poli. INSERT(x) nastaví $B[h(x)] = 1$. MEMBER(x) otestuje, zda $B[h(x)] = 1$. Vložme nyní do filtru nějakou n-prvkovou množinu M. Pokud $x \in M$, MEMBER(x) vždy odpoví správně. Pokud se ale zeptáme na $x \notin M$, může se stát, že $h(x) = h(y)$ pro nějaké $y \in M$, a MEMBER(x) odpoví špatně. Spočítejte, s jakou pravděpodobností se to pro dané m a n stane.
4.* Spolehlivost Bloomova filtru můžeme zvýšit tak, že si pořídíme k filtrů s různými hešovacími funkcemi. INSERT bude vkládat do všech, MEMBER se zeptá všech a odpoví ANO pouze tehdy, když se na tom všechny filtry shodnou. Je-li pravděpodobnost chyby jednoho filtru p, pak kombinace k filtrů chybuje s pravděpodobností pouhých p^k. Vymyslete, jak nastavit m a k pro případ, kdy chceme ukládat 10^6 prvků s pravděpodobností chyby nejvýše 10^{-9}. Minimalizujte spotřebu paměti.

1.4. Hešování s otevřenou adresací

Ještě ukážeme jeden způsob hešování, který je prostorově úspornější a za příznivých okolností může být i rychlejší. Za tyto výhody zaplatíme složitějším chováním a pracnější analýzou. Tomuto druhu hešování se říká *otevřená adresace.*

Opět si pořídíme pole s m přihrádkami $A[0], \ldots, A[m-1]$, jenže tentokrát se do každé přihrádky vejde jen jeden prvek. Pokud bychom tam potřebovali uložit další, použijeme náhradní přihrádku, bude-li také plná, zkusíme další, a tak dále.

Můžeme si to představit tak, že hešovací funkce každému prvku $x \in \mathcal{U}$ přiřadí jeho *vyhledávací posloupnost* $h(x,0), h(x,1), \ldots, h(x,m-1)$. Ta určuje pořadí přihrádek, do kterých se budeme snažit x vložit. Budeme předpokládat, že posloupnost obsahuje všechna čísla přihrádek v dokonale náhodném pořadí (všechny permutace přihrádek budou stejně pravděpodobné). Vkládání do tabulky bude vypadat následovně:

Algoritmus OpenInsert (vkládání s otevřenou adresací)
Vstup: Prvek $x \in \mathcal{U}$

1. Pro $i = 0, \ldots, m-1$:
2. $\quad j \leftarrow h(x,i)$ ◁ *číslo přihrádky, kterou právě zkoušíme*
3. $\quad$ Pokud $A[j] = \emptyset$:
4. $\quad\quad$ Položíme $A[j] \leftarrow x$ a skončíme.
5. Ohlásíme, že tabulka je už plná.

Při vyhledávání budeme procházet přihrádky $h(x,0)$, $h(x,1)$ a tak dále. Zastavíme se, jakmile narazíme buď na x, nebo na prázdnou přihrádku.

Algoritmus OpenFind (hledání s otevřenou adresací)
Vstup: Prvek $x \in \mathcal{U}$

1. Pro $i = 0, \ldots, m-1$:
2. $\quad j \leftarrow h(x,i)$ ◁ *číslo přihrádky, kterou právě zkoušíme*
3. $\quad$ Pokud $A[j] = x$, ohlásíme, že jsme x našli, a skončíme.
4. $\quad$ Pokud $A[j] = \emptyset$, ohlásíme neúspěch a skončíme.
5. Ohlásíme neúspěch.

Mazání je problematické: kdybychom libovolný prvek odstranili z tabulky, mohli bychom způsobit, že vyhledávání nějakého jiného prvku skončí předčasně, protože narazí na přihrádku, která v okamžiku vkládání byla plná, ale nyní už není. Proto budeme prvky pouze označovat za smazané a až jich bude mnoho (třeba $m/4$), celou strukturu přebudujeme. Podobně jako u zvětšování tabulky je vidět, že toto přebudovávání nás stojí amortizovaně konstantní čas na smazaný prvek.

Odvodíme, kolik kroků průměrně provedeme při neúspěšném vyhledávání, což je současně počet kroků potřebných na vložení prvku do tabulky. Úspěšné hledání nebude pomalejší.

Věta: Pokud jsou vyhledávací posloupnosti náhodné permutace, průměrný počet přihrádek navštívených při neúspěšném hledání činí nejvýše $1/(1-\alpha)$, kde $\alpha = n/m$ je faktor naplnění.

Důkaz: Nechť x je hledaný prvek a $h_1, h_2, \ldots, h_m$ jeho vyhledávací posloupnost. Označme p_i pravděpodobnost toho, že během hledání projdeme alespoň i přihrádek.

Do přihrádky h_1 se jistě podíváme, proto $p_1 = 1$. Jelikož je to náhodně vybraná přihrádka, s pravděpodobností n/m v ní je nějaký prvek (různý od x, neboť vyhledávání má skončit neúspěchem), a tehdy pokračujeme přihrádkou h_2. Proto $p_2 = n/m = \alpha$.

Nyní obecněji: pakliže přihrádky $h_1, \ldots, h_i$ byly obsazené, zbývá $n - i$ prvků a $m - i$ přihrádek. Přihrádka h_{i+1} je tedy obsazena s pravděpodobností $(n-i)/(m-i) \leq n/m$ (to platí, jelikož $n \leq m$). Proto $p_{i+1} \leq p_i \cdot \alpha$. Indukcí dostaneme $p_i \leq \alpha^{i-1}$.

Nyní počítejme střední hodnotu S počtu navštívených přihrádek. Ta je rovna $\sum_i i \cdot q_i$, kde q_i udává pravděpodobnost, že jsme navštívili právě i přihrádek. Jelikož $q_i = p_i - p_{i+1}$, platí

$$S = \sum_{i \geq 1} i \cdot (p_i - p_{i+1}) = \sum_{i \geq 1} p_i \cdot (i - (i-1)) = \sum_{i \geq 1} p_i \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha^i.$$

To je ovšem geometrická řada se součtem $1/(1-\alpha)$. □

Pokud se tedy faktor naplnění přiblíží k jedničce, začne se hledání drasticky zpomalovat. Pokud ale ponecháme alespoň čtvrtinu přihrádek prázdnou, navštívíme během hledání průměrně nanejvýš 4 přihrádky. Opět můžeme použít přehešovávání, abychom tento stav udrželi. Tak dostaneme vyhledávání, vkládání i mazání v amortizovaně konstantním průměrném čase.

Zbývá vymyslet, jak volit prohledávací posloupnosti. V praxi se často používají tyto možnosti:

- *Lineární přidávání:* $h(x,i) = (f(x) + i) \bmod m$,
  kde $f(x)$ je „obyčejná“ hešovací funkce. Využíváme tedy po sobě jdoucí přihrádky. Výhodou je sekvenční přístup do paměti (který je na dnešních počítačích rychlejší), nevýhodou to, že jakmile se vytvoří souvislé bloky obsazených přihrádek, další vkládání se do nich často strefí a bloky stále porostou (viz obrázek 1.2).

  Bez důkazu uvádíme, že pro neúspěšné hledání platí pouze slabší odhad průměrného počtu navštívených přihrádek $1/(1-\alpha)^2$, a to pouze, jestliže je f dokonale náhodná. Není-li, chování struktury obvykle degraduje.
- *Dvojité hešování:* $h(x,i) = (f(x) + i \cdot g(x)) \bmod m$,
  kde $f : \mathcal{U} \to \{0, \ldots, m-1\}$ a $g : \mathcal{U} \to \{1, \ldots, m-1\}$ jsou dvě různé hešovací funkce a m je prvočíslo. Díky tomu je $g(x)$ vždy nesoudělné s m a posloupnost navštíví každou přihrádku právě jednou. Je známo, že pro dokonale náhodné funkce f a g se dvojité hešování chová stejně dobře, jako při použití plně náhodných vyhledávacích posloupností. Dokonce stačí vybírat f a g ze silně univerzálního systému (viz příští oddíl). Tato tvrzení též ponecháváme bez důkazu.

		42		14	75	36	24	95	17
0	1	2	3	4	5	6	7	8	9

Obr. 1.2: Hešování podle poslední číslice s lineárním přidáváním. Stav po vložení čísel 75, 36, 14, 42, 24, 95, 17.

Cvičení

1.* Úspěšné hledání v hešovací tabulce s otevřenou adresací je o něco rychlejší než neúspěšné. Spočítejte, kolik přihrádek průměrně navštíví. Předpokládejte, že hledáme náhodně vybraný prvek tabulky.

2. Uvažujme hešování řízené obecnější lineární posloupností $h(x,i) = (f(x) + c \cdot i) \bmod m$, kde c je konstanta nesoudělná s m. Srovnejte jeho chování s obyčejným lineárním přidáváním.

1.5.* Univerzální hešování

Zatím jsme hešování používali tak, že jsme si vybrali nějakou pevnou hešovací funkci a „zadrátovali“ ji do programu. Ať už je to jakákoliv funkce, nikdy není těžké najít n čísel, která *zkolidují* v téže přihrádce, takže jejich vkládáním strávíme čas $\Theta(n^2)$. Můžeme spoléhat na to, že vstup takhle nešikovně vypadat nebude, ale co když nám vstupy dodává zvědavý a potenciálně velmi škodolibý uživatel?

Raději při každém spuštění programu zvolíme hešovací funkci náhodně – nepřítel o této funkci nic neví, takže se mu sotva povede vygenerovat dostatečně ošklivý vstup. Nemůžeme ale náhodně vybírat ze všech možných funkcí z $\mathcal{U}$ do $\mathcal{P}$, protože na popis jedné takové funkce bychom potřebovali $\Theta(|\mathcal{U}|)$ čísel. Místo toho se omezíme na nějaký menší systém funkcí a náhodně vybereme z něj. Aby to fungovalo, musí tento systém být dostatečně bohatý, což zachycuje následující definice.

Značení: Často budeme mluvit o různých množinách přirozených čísel. Proto zavedeme zkratku $[k] = \{0, \ldots, k-1\}$. Množina přihrádek bude typicky $\mathcal{P} = [m]$.

Definice: Systém $\mathcal{H}$ funkcí z univerza $\mathcal{U}$ do $[m]$ nazveme *c-univerzální* pro konstantu $c \geq 1$, pokud pro každé dva různé prvky $x, y \in \mathcal{U}$ platí $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq c/m$.

Co si pod tím představit? Kdybychom funkci h rovnoměrně náhodně vybírali z úplně všech funkcí z $\mathcal{U}$ do $[m]$, kolidovaly by prvky x a y s pravděpodobností $1/m$. Nezávisle na tom, kolik vyšlo $h(x)$, by totiž bylo všech m možností pro $h(y)$ stejně pravděpodobných. A pokud místo ze všech funkcí vybíráme h z c-univerzálního systému, smí x a y kolidovat nejvýše c-krát častěji.

Navíc budeme chtít, aby šlo funkci $h \in \mathcal{H}$ určit malým množstvím parametrů a na základě těchto parametrů ji pro konkrétní vstup vyhodnotit v konstantním čase. Například můžeme říci, že $\mathcal{H}$ bude systém lineárních funkcí tvaru $x \mapsto ax \bmod m$ pro $\mathcal{U} = [U]$ a $a \in [U]$. Každá taková funkce je jednoznačně určena parametrem a, takže náhodně vybrat funkci je totéž jako náhodně zvolit $a \in [U]$. To zvládneme v konstantním čase, stejně jako vyhodnotit funkci pro dané a a x.

Za chvíli ukážeme, jak nějaký c-univerzální systém sestrojit. Předtím ale dokážeme, že splní naše očekávání.

Lemma: Buď h funkce náhodně vybraná z nějakého c-univerzálního systému. Nechť $x_1, \ldots, x_n$ jsou navzájem různé prvky univerza vložené do hešovací tabulky a x je libovolný prvek univerza. Potom pro střední počet prvků ležících v téže přihrádce jako x platí:

$$\mathbb{E}[\#i : h(x) = h(x_i)] \leq cn/m + 1.$$

Důkaz: Pro dané x definujeme indikátorové náhodné proměnné:

$$Z_i = \begin{cases} 1 & \text{když } h(x) = h(x_i), \\ 0 & \text{jindy.} \end{cases}$$

Jinými slovy Z_i říká, kolikrát padl prvek x_i do přihrádky $h(x)$, což je buď 0, nebo 1. Celkový počet kolidujících prvků je $Z = \sum_i Z_i$ a díky linearitě střední hodnoty je hledaná hodnota $\mathbb{E}[Z]$ rovna $\sum_i \mathbb{E}[Z_i]$. Přitom $\mathbb{E}[Z_i] = \Pr[Z_i = 1]$, což je podle definice c-univerzálního systému nejvýše c/m, pokud $x_i \neq x$. Pokud $x_i = x$, pak $\mathbb{E}[Z_i] = 1$, což ale může nastat pro nejvýše jedno i. Tedy $\mathbb{E}[Z]$ je nejvýše $cn/m+1$. □

Důsledek: Nechť k hešování použijeme funkci vybranou rovnoměrně náhodně z nějakého c-univerzálního systému. Pokud už hešovací tabulka obsahuje n prvků, bude příští operace nahlížet do přihrádky s průměrně nanejvýš $cn/m+1$ prvky. Udržíme-li $m = \Omega(n)$, bude tedy průměrná velikost přihrádky omezena konstantou, takže průměrná časová složitost operace bude také konstantní.

Mějme na paměti, že neprůměrujeme přes možná vstupní data, nýbrž přes možné volby hešovací funkce, takže tvrzení platí pro libovolně škodolibý vstup. Také upozorňujeme, že tyto úvahy vyžadují oddělené přihrádky a nefungují pro otevřenou adresaci.

Ve zbytku tohoto oddílu budeme předpokládat, že čtenář zná základy lineární algebry (vektorové prostory a skalární součin) a teorie čísel (dělitelnost, počítání s kongruencemi a s konečnými tělesy).

Konstrukce ze skalárního součinu

Postupně ukážeme, jak upravit praktické hešovací funkce z oddílu 1.3, aby tvořily univerzální systém. Nejsnáze to půjde se skalárními součiny.

Zvolíme nějaké konečné těleso $\mathbb{Z}_p$ pro prvočíselné p. Pořídíme si $m = p$ přihrádek a očíslujeme je jednotlivými prvky tělesa. Univerzem bude vektorový prostor $\mathbb{Z}_p^d$ všech d-složkových vektorů nad tímto tělesem. Hešovací funkce bude mít tvar skalárního součinu s nějakým pevně zvoleným vektorem $\mathbf{t} \in \mathbb{Z}_p^d$.

Věta: Systém funkcí $\mathcal{S} = \{h_\mathbf{t} \mid \mathbf{t} \in \mathbb{Z}_p^d\}$, kde $h_\mathbf{t}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x}$, je 1-univerzální.

Důkaz: Mějme nějaké dva různé vektory $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^d$. Nechť i je nějaká souřadnice, v níž je $x_i \neq y_i$. Jelikož skalární součin nezáleží na pořadí složek, můžeme složky přečíslovat tak, aby bylo $i = d$.

Nyní volíme $\mathbf{t}$ náhodně po složkách a počítáme pravděpodobnost kolize (rovnost modulo p značíme $\equiv$):

$$\Pr_{\mathbf{t}\in\mathbb{Z}_p^d}[h_{\mathbf{t}}(\mathbf{x}) \equiv h_{\mathbf{t}}(\mathbf{y})] = \Pr[\mathbf{x}\cdot\mathbf{t} \equiv \mathbf{y}\cdot\mathbf{t}] = \Pr[(\mathbf{x}-\mathbf{y})\cdot\mathbf{t} \equiv 0] =$$
$$= \Pr\left[\sum_{i=1}^{d}(x_i-y_i)t_i \equiv 0\right] = \Pr\left[(x_d-y_d)t_d \equiv -\sum_{i=1}^{d-1}(x_i-y_i)t_i\right].$$

Pokud už jsme $t_1,\ldots,t_{d-1}$ zvolili a nyní náhodně volíme t_d, nastane kolize pro právě jednu volbu: Poslední výraz je lineární rovnice tvaru $az = b$ pro nenulové a a ta má v libovolném tělese právě jedno řešení z. Pravděpodobnost kolize je tedy nejvýše $1/p = 1/m$, jak požaduje 1-univerzalita. □

Intuitivně náš důkaz funguje takto: Pro nenulové $x \in \mathbb{Z}_p$ a rovnoměrně náhodně zvolené $a \in \mathbb{Z}_p$ nabývá výraz ax všech hodnot ze $\mathbb{Z}_p$ se stejnou pravděpodobností. Proto se d-tý sčítanec skalárního součinu chová rovnoměrně náhodně. Ať už má zbytek skalárního součinu jakoukoliv hodnotu, přičtením d-tého členu se z něj stane také rovnoměrně rozložené náhodné číslo.

Příklad: Kdybychom chtěli hešovat 32-bitová čísla do cca 250 přihrádek, nabízí se zvolit $p = 257$ a každé číslo rozdělit na 4 části po 8 bitech. Jelikož $2^8 = 256$, můžeme si tyto části vyložit jako 4-složkový vektor nad $\mathbb{Z}_{257}$. Například číslu $123\,456\,789 = 7\cdot2^{24}+91\cdot2^{16}+205\cdot2^8+21$ odpovídá vektor $\mathbf{x} = (7, 91, 205, 21)$. Pro $\mathbf{t} = (1, 2, 3, 4)$ se tento vektor zahešuje na $\mathbf{x}\cdot\mathbf{t} \equiv 7\cdot1+91\cdot2+205\cdot3+21\cdot4 \equiv 7+182+101+84 \equiv 117$.

Poznámka: Jistou nevýhodou této konstrukce je, že počet přihrádek musí být prvočíselný. To by nám mohlo vadit při přehešovávání do dvojnásobně velké tabulky. Zachrání nás ovšem *Bertrandův postulát,* který říká, že mezi m a $2m$ vždy leží alespoň jedno prvočíslo. Pokud budeme zaokrouhlovat počet přihrádek na nejbližší vyšší prvočíslo, máme zaručeno, že pokaždé tabulku nejvýše zčtyřnásobíme, což se stále uamortizuje. Teoreticky nás může brzdit hledání vhodných prvočísel, v praxi si na 64-bitovém počítači pořídíme tabulku 64 prvočísel velikostí přibližně mocnin dvojky.

Konstrukce z lineární kongruence

Nyní se inspirujeme lineární kongruencí $x \mapsto ax \bmod m$. Pro prvočíselné m by stačilo náhodně volit a a získali bychom 1-univerzální systém – jednorozměrnou obdobu předchozího systému se skalárním součinem. My ovšem dáme přednost trochu komplikovanějším funkcím, které zato budou fungovat pro libovolné m.

Budeme se pohybovat v univerzu $\mathcal{U} = [U]$. Pořídíme si nějaké prvočíslo $p \geq U$ a počet přihrádek $m < U$. Budeme počítat lineární funkce tvaru $ax + b$ v tělese $\mathbb{Z}_p$ a výsledek dodatečně modulit číslem m.

Věta: Nechť $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Potom systém funkcí $\mathcal{L} = \{h_{a,b} \mid a, b \in [p], a \neq 0\}$ je 1-univerzální.

Důkaz: Mějme dvě různá čísla $x, y \in [U]$. Nejprve rozmýšlejme, jak se chovají lineární funkce modulo p bez dodatečného modulení číslem m a bez omezení $a \neq 0$. Pro

libovolnou dvojici parametrů $(a,b) \in [p]^2$ označme:

$$r = (ax + b) \bmod p,$$
$$s = (ay + b) \bmod p.$$

Každé dvojici $(a,b) \in [p]^2$ tedy přiřadíme nějakou dvojici $(r,s) \in [p]^2$. Naopak každá dvojice (r,s) vznikne z právě jedné dvojice (a,b): podmínky pro a a b dávají soustavu dvou nezávislých lineárních rovnic o dvou neznámých, která musí mít v libovolném tělese právě jedno řešení. (Explicitněji: Odečtením rovnic dostaneme $r-s \equiv a(x-y)$, což dává jednoznačné a. Dosazením do libovolné rovnice pak získáme jednoznačné b.)

Máme tedy bijekci mezi všemi dvojicemi (a,b) a (r,s). Nezapomínejme ale, že jsme zakázali $a = 0$, což odpovídá zákazu $r = s$.

Nyní vraťme do hry modulení číslem m a počítejme *špatné* dvojice (a,b), pro něž nastane $h_{a,b}(x) = h_{a,b}(y)$. Ty odpovídají dvojicím (r,s) splňujícím $r \equiv s$ modulo m. Pro každé r spočítáme, kolik možných s s ním je kongruentních. Pokud množinu $[p]$ rozdělíme na m-tice, dostaneme $\lceil p/m \rceil$ m-tic, z nichž poslední je neúplná. V každé úplné m-tici leží právě jedno číslo kongruentní s r, v té jedné neúplné nejvýše jedno. Navíc ovšem víme, že $r \neq s$, takže možných s je o jedničku méně.

Celkem tedy pro každé r existuje nanejvýš $\lceil p/m \rceil - 1$ kongruentních s. To shora odhadneme výrazem $(p+m-1)/m - 1 = (p-1)/m$. Jelikož možností, jak zvolit r, je přesně p, dostáváme maximálně $p(p-1)/m$ špatných dvojic (r,s). Mezi dvojicemi (a,b) a (r,s) vede bijekce, takže špatných dvojic (a,b) je stejný počet. Jelikož možných dvojic (a,b) je $p(p-1)$, pravděpodobnost, že vybereme nějakou špatnou, je nejvýše $1/m$. Systém je tedy 1-univerzální. □

Konstrukce z vyšších bitů součinu

Nakonec ukážeme univerzalitu hešovacích funkcí založených na vyšších bitech součinu. Univerzum budou tvořit všechna w-bitová čísla, tedy $\mathcal{U} = [2^w]$. Hešovat budeme do $m = 2^\ell$ přihrádek. Hešovací funkce pro klíč x vypočte součin ax a „vykousne“ z něj bity na pozicích $w-\ell$ až $w-1$. (Bity číslujeme obvyklým způsobem od nuly, tedy i-tý bit má váhu 2^i.)

Věta: Nechť $h_a(x) = \lfloor (ax \bmod 2^w)/2^{w-\ell} \rfloor$. Potom systém funkcí $\mathcal{M} = \{h_a \mid a \in [2^w], a \text{ liché}\}$ je 2-univerzální.

Důkaz: Mějme nějaké dva různé klíče x a y. Bez újmy na obecnosti předpokládejme, že $x < y$. Označme i pozici nejnižšího bitu, v němž x od y liší. Platí tedy $y-x = z \cdot 2^i$, kde z je nějaké liché číslo.

Chceme počítat pravděpodobnost, že $h_a(x) = h_a(y)$ pro rovnoměrně náhodné a. Jelikož a je liché, můžeme ho zapsat jako $a = 2b+1$, kde $b \in [2^{w-1}]$, a volit rovnoměrně náhodné b.

Prozkoumejme, jak se chová výraz $a(y-x)$. Můžeme ho zapsat takto:

$$a(y-x) = (2b+1)(z \cdot 2^i) = bz \cdot 2^{i+1} + z \cdot 2^i.$$

Podívejme se na binární zápis:

- Člen $z \cdot 2^i$ má v bitech 0 až $i-1$ nuly, v bitu i jedničku a vyšší bity mohou vypadat jakkoliv, ale nezávisí na b.
- Člen $bz \cdot 2^{i+1}$ má bity 0 až i nulové. V bitech $i+1$ až $w+i$ leží $bz \bmod 2^w$, které nabývá všech hodnot z $[2^w]$ se stejnou pravděpodobností: Jelikož z je liché, a tedy nesoudělné s 2^w, má kongruence $bz \equiv d \pmod{2^w}$ pro každé d právě jedno řešení b (viz cvičení **??**). Všechna b nastávají se stejnou pravděpodobností, takže všechna d také.
- Součet těchto dvou členů tedy musí mít bity 0 až $i-1$ nulové, v bitu i jedničku a v bitech $i+1$ až $w+i$ rovnoměrně náhodné číslo z $[2^w]$ (sečtením rovnoměrně náhodného čísla s čímkoliv nezávislým vznikne modulo 2^w opět rovnoměrně náhodné číslo). O vyšších bitech nic neříkáme.

Vraťme se k rovnosti $h_a(x) = h_b(y)$. Ta nastane, pokud se čísla ax a ay shodují v bitech $w-\ell$ až $w-1$. Využijeme toho, že $ay = ax + a(y-x)$, takže ax a ay se určitě shodují v bitech 0 až $i-1$ a neshodují v bitu i. Rozlišíme dva případy:

- Pokud $i \geq w-\ell$, bit i patří mezi bity vybrané do výsledku hešovací funkce, takže $h_a(x)$ a $h_a(y)$ se určitě liší.
- Je-li $i < w-\ell$, pak jsou všechny vybrané bity v $a(y-x)$ rovnoměrně náhodné. Kolize x s y nastane, pokud tyto bity budou všechny nulové, nebo když budou jedničkové a navíc v součtu $ax + a(y-x)$ nastane přenos z nižšího řádu. Obojí má pravděpodobnost nejvýš $2^{-\ell}$, takže kolize nastane s pravděpodobností nejvýš $2^{1-\ell} = 2/m$. □

Vzorkování a silná univerzalita*

Hešovací funkce se hodí i pro jiné věci, než je reprezentace množin. Představte si počítačovou síť, v níž putují pakety přes velké množství routerů. Chtěli byste sledovat, co se v síti děje, třeba tak, že necháte každý router zaznamenávat, jaké pakety přes něj projdou. Jenže na to je paketů příliš mnoho. Nabízí se pakety *navzorkovat,* tedy vybrat si z nich jen malou část a tu sledovat.

Vzorek ale nemůžeme vybírat náhodně: kdyby si každý router hodil korunou, zda daný paket zaznamená, málokdy u jednoho paketu budeme znát celou jeho cestu. Raději si pořídíme hešovací funkci h, která paketu p přiřadí nějaké číslo $h(p) \in [m]$. Kdykoliv router přijme paket, zahešuje ho a pokud vyjde méně než nějaký parametr t, paket zaznamená. To nastane s pravděpodobností t/m a shodnou se na tom všechny routery po cestě.

Nabízí se zvolit funkci h náhodně z nějakého c-univerzálního systému. Jenže pak nebudeme vzorkovat spravedlivě: například v našem systému $\mathcal{S}$ odvozeném ze skalárního součinu padne nulový vektor vždy do přihrádky 0, takže ho pro každé t vybereme do vzorku. Aby vzorkování fungovalo, budeme potřebovat silnější definici univerzality.

Definice: Systém $\mathcal{H}$ funkcí z univerza $\mathcal{U}$ do $[m]$ nazveme *silně c-univerzální* pro konstantu $c \geq 1$, pokud pro každé dva různé prvky $x, y \in \mathcal{U}$ a každé dvě přihrádky $a, b \in [m]$ (ne nutně různé) platí $\Pr_{h \in \mathcal{H}}[h(x) = a \wedge h(y) = b] \leq c/m^2$.

Podobně jako u obyčejné (neboli slabé) univerzality, i zde vlastně říkáme, že funkce náhodně vybraná z daného systému je nejvýše c-krát horší než úplně náhodná funkce: ta by s pravděpodobností $1/m$ zahešovala x do přihrádky a a nezávisle na tom y do přihrádky b.

Důsledek: Pro prvek x a konkrétní přihrádku a je $\Pr_h[h(x) = a] \leq c/m$. Proto náš způsob vzorkování každý paket zaznamená s pravděpodobností nejvýše c-krát větší, než by odpovídalo rovnoměrně náhodnému výběru.

Navíc ukážeme, že malými úpravami již popsaných systémů funkcí z nich můžeme udělat silně univerzální. Pro systém $\mathcal{L}$ to vzápětí dokážeme, ostatní nechme jako cvičení 7 a 8.

Věta: Nechť $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Potom systém funkcí $\mathcal{L}' = \{h_{a,b} \mid a, b \in [p]\}$ je silně 4-univerzální.

Důkaz: Z důkazu 1-univerzality systému $\mathcal{L}$ víme, že pro pevně zvolené x a y existuje bijekce mezi dvojicemi parametrů $(a, b) \in \mathbb{Z}_p^2$ a dvojicemi $(r, s) = (ax \bmod p, bx \bmod p)$. Pokud volíme parametry rovnoměrné náhodně, dostáváme i (r, s) rovnoměrně náhodně. Zbývá ukázat, že závěrečným modulením m se rovnoměrnost příliš nepokazí.

Potřebujeme, aby pro každé $i, j \in [m]$ platilo ($\equiv$ značí kongruenci modulo m):

$$\Pr_{r,s}[r \equiv i \wedge s \equiv j] \leq \frac{4}{m^2}.$$

Jelikož $r \equiv i$ a $s \equiv j$ jsou nezávislé jevy, navíc se stejnou pravděpodobností, stačí ověřit, že $\Pr_r[r \equiv i] \leq 2/m$.

Čísel $r \in [p]$ kongruentních s i může být nejvýše $\lceil p/m \rceil = \lfloor (p + m - 1)/m \rfloor \leq (p + m - 1)/m = (p - 1)/m + 1$. Využijeme-li navíc toho, že $m \leq p$, získáme:

$$\Pr_r[r \equiv i] = \frac{\#r : r \equiv i}{p} \leq \frac{p-1}{p \cdot m} + \frac{1}{p} \leq \frac{1}{m} + \frac{1}{m} = \frac{2}{m}.$$

Tím jsme větu dokázali. □

Cvičení

1. Dostali jste hešovací funkci $h : [U] \to [m]$. Pokud o této funkci nic dalšího nevíte, kolik vyhodnocení funkce potřebujete, abyste našli k-tici prvků, které se všechny zobrazí do téže přihrádky?
2. Ukažte, že pokud bychom v „lineárním“ systému $\mathcal{L}$ zafixovali parametr b na nulu, už by nebyl 1-univerzální, ale pouze 2-univerzální. Totéž by se stalo, pokud bychom připustili nulové a.
3. Ukažte, že pokud bychom v „součinovém“ systému $\mathcal{M}$ připustili i sudá a, už by nebyl c-univerzální pro žádné c.

4. Studujme chování polynomiálního hešování z minulého oddílu. Uvažujme funkce $h_a : \mathbb{Z}_p^d \to \mathbb{Z}_p$, přičemž $h_a(x_0, \ldots, x_{d-1}) = \sum_i x_i a^i \bmod p$. Dokažte, že systém $\mathcal{P} = \{h_a \mid a \in \mathbb{Z}_p\}$ je d-univerzální. Mohou se hodit vlastnosti polynomů z oddílu **??**.

5. Dokažte, že je-li nějaký systém funkcí silně c-univerzální, pak je také (slabě) c-univerzální.

6.* O systému $\mathcal{L}'$ jsme dokázali, že je silně c-univerzální pro $c = 4$. Rozmyslete si, že pro žádné menší c to neplatí.

7.* Ukažte, že systém $\mathcal{S}$ odvozený ze skalárního součinu není silně c-univerzální pro žádné c. Ovšem pokud ho rozšíříme na $\mathcal{S}' = \{h_{\mathbf{t},r} \mid \mathbf{t} \in \mathbb{Z}_p^d, r \in \mathbb{Z}_p\}$, kde $h_{\mathbf{t},r}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x} + r$, už bude silně 1-univerzální.

8.** Podobně systém $\mathcal{M}$ není silně c-univerzální pro žádné c, ale jde to zachránit, dokonce bez zavádění dalších parametrů: $\mathcal{M}' = \{h_a \mid a \in [2^w], a \text{ liché}\}$, přičemž $h_a(x) = \lfloor (ax \bmod 2^{w+\ell})/2^w \rfloor$. Jinak řečeno výsledkem hešovací funkce jsou bity w až $w + \ell - 1$ v součinu ax. Dokažte, že systém $\mathcal{M}'$ je silně 2-univerzální.

9. Dokažte, že je-li $\mathcal{H}$ nějaký c-univerzální systém funkcí z $\mathcal{U}$ do $[m]$, pak pro $m' \leq m$ je $\{x \mapsto h(x) \bmod m' \mid h \in \mathcal{H}\}$ $2c$-univerzální z $\mathcal{U}$ do $[m']$. Vyslovte podobné tvrzení pro silnou univerzalitu. Jakou roli hraje tento fakt v rozboru systémů $\mathcal{L}$ a $\mathcal{L}'$? Lze dosáhnout lepší konstanty než $2c$?