Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Rýchle testy prvočíselnosti pre obmedzený rozsah vstupov

Bakalárska práca

2014 **Jakub Jančina**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Rýchle testy prvočíselnosti pre obmedzený rozsah vstupov

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky FMFI
Vedúci práce: RNDr. Michal Forišek, PhD.

2014 **Jakub Jančina**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Jančina
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Rýchle testy prvočíselnosti pre obmedzený rozsah vstupov

Cieľ: Cieľom práce je preskúmať možné implementácie rýchleho testu prvočíselnosti pre vstupy zo zhora obmedzeného rozsahu, s dôrazom pre rozsahy do 2^31, resp. 2^32. Súčasťou výslednej práce by mala byť definícia vhodných kritérií pre porovnávanie efektívnosti implementácií a následne vyhodnotenie známych a v ideálnom prípade aj nájdenie vlastnej implementácie, lepšej pre vyššie spomenuté špecifické rozsahy vstupov.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 28.10.2013

Dátum schválenia: 28.10.2013

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Ďakujem môjmu školiteľovi RNDr. Michalovi Forišekovi za cenné rady a pripomienky. Ďalej ďakujem svojim priateľom za poskytnutú výpočtovú silu a testovanie.

Abstrakt

Táto práca sa zaoberá hľadaním rýchleho deterministického testu prvočíselnosti pre nezáporné čísla menšie ako 2^{32} respektíve menšie ako 2^{64}. Výsledkom tejto práce je 1-kolový deterministický Rabin-Millerov test korektný pre čísla menšie ako 2^{32} s pamäťovou zložitosťou 3808 bitov a 2-kolový deterministický Rabin-Millerov test korektný pre čísla menšie ako 2^{64} s pamäťovou zložitosťou 512 KiB.

Kľúčové slová: rýchly test prvočíselnosti, prvočísla, Rabin-Millerov test

Abstract

The aim of this thesis is to find fast deterministic primality tests for non-negative integers smaller than 2^{32} and smaller than 2^{64}. The result of this thesis is 1-round deterministic Rabin-Miller test which is correct for 32bit integers with memory complexity of 3808bits and 2-round deterministic Rabin-Miller test which is correct for 64bit integers with memory complexity of 512 KiB.

Keywords: fast primality test, prime, Rabin-Miller test

Obsah

Úvod

V tejto práci budeme rozoberať rôzne možnosti implementácie rýchleho testu prvočíselnosti. V prvej časti spomenieme vybrané pravdepodobnostné testy prvočíselnosti, špeciálne Fermatov test, Solovay-Strassenov test, Rabin-Millerov test a Lucasov test. V druhej časti sa pozrieme na determinizáciu Rabin-Millerovho testu pre obmedzený rozsah vstupov pomocou hashovania. Konkrétne budeme hľadať pole báz *base* a hashovaciu funkciu h aby číslo x bolo prvočíslo práve vtedy keď, bude silné pseudoprvočíslo pri báze $base[h(x)]$. Ďalej sa budeme snažiť analyzovať pravdepodobnosť existencie poľa *base* pre náhodne zvolenú hashovaciu funkciu h a pomocou tejto analýzy nájsť čo najmenšie pole *base* pre ktoré bude takýto test korektný. V tretej kapitole sa pozrieme na podobnú konštrukciu pre 64 bitové čísla. Na konci tejto práce uvedieme porovnania voči iným rýchlym testom prvočíselnosti.

Keďže sa v práci zaoberáme len relatívne malými číslami, na odhad časovej zložitosti budeme používať RAM model, v ktorom uvažujeme, že základné aritmetické operácie s $\mathcal{O}(\log n)$-bitovými číslami, kde n je číslo testované na prvočíselnosť, vieme robiť v konštantnom čase.

Najskôr uvedieme dve jednoduché riešenia tohto problému, ktoré nie sú až tak efektívne.

Naivné delenie

Prvočíslo je číslo, ktoré nemá žiadneho netriviálneho deliteľa. Teda je deliteľné len jednotkou a sebou samým. Táto definícia dáva priamočiary algoritmus na dokázanie, že dané číslo je prvočíslo.
Časová zložitosť tohto algoritmu je $\mathcal{O}(n)$ čo je neuspokojivé. Jednoduchou úvahou sa dá tento algoritmus podstatne zlepšiť. Každé zložené číslo má deliteľa menšieho alebo rovného ako $\sqrt{n}$ (lebo inak by jeho kanonický rozklad dával väčšie číslo ako n, čo je spor) a teda stačí vyskúšať len delitele menšie alebo rovné ako $\sqrt{n}$. Tento algoritmus je najjednoduchší možný, avšak nie je až tak efektívny.

Eratostenovo sito

Podobný prístup používa ďalší známy algoritmus nazývaný Eratostenovo sito. Princíp je veľmi podobný, no na rozdiel od jednoduchého delenia sa vyhadzujú násobky. Algoritmus na začiatku prehlási všetky čísla $\leq n$ za prvočísla. Následne toto pole sekvenčne prechádza a pre každý prvok, ktorý je označený ako prvočíslo zavolá cyklus, ktorý prehlási všetky jeho násobky za zložené čísla.
Výsledok tohto algoritmu je zoznam prvočísiel $\leq n$. **Časová zložitosť** tohto algoritmu je $\mathcal{O}(n \cdot \log \log n)$ no pamäťová zložitosť je $\mathcal{O}(n)$. Následné zistenie či je dané číslo ($\leq n$) prvočíslo sa vykonáva v $\mathcal{O}(1)$. Avšak predvýpočet je veľmi pomalý. Ďalšia nevýhoda tohto postupu je, že má lineárnu pamäťovú zložitosť. Na dokazovanie prvočíselnosti nezáporných čísiel $< 2^{32}$ potrebuje (pri lepšej implementácii) 256 MiB pamäte. Ak by sme uvažovali $n = 2^{64}$ tak je to dokonca 1 048 576 TiB, čím sa tento algoritmus stáva nepoužiteľným.

KAPITOLA 1

Pravdepodobnostné testy

Nasledujúce testy nebudú dokazovať prvočíselnosť daného čísla. Namiesto toho ich výstupom bude buď, že číslo nie je prvočíslo, alebo, že číslo je prvočíslo s určitou pravdepodobnosťou. Tieto testy sú oveľa efektívnejšie ako predchádzajúce testy a to z dôvodu, že neoverujú definíciu prvočísla. Namiesto toho overujú vlastnosti, ktoré prvočísla spĺňajú, no väčšina zložených čísel nie. Navyše sú tieto testy skoro vždy úspešné v prvých kolách. V nasledujúcej časti uvedieme niekoľko týchto vlastností aj s algoritmami, ktoré sú na nich založené. Na pochopenie tejto problematiky je nutné uviesť niekoľko viet z teórie čísiel. Vety a lemy, ktoré sú uvedené bez dôkazu alebo bez konkrétneho odkazu na dôkaz sa dajú nájsť v [14]. Niektoré prehľadové časti sú podrobnejšie spracované v [16].

1.1 Frematove pseudoprvočíslo

Lema 1.1.1. *Nech* $p \in \mathbb{P}$. *Potom* $p \mid \binom{p}{k} \quad k \in \{1 \dots p-1\}$

Veta 1.1.1. *(Malá Frematova veta) Nech p je prvočíslo, a ľubovoľné celé číslo. Potom platí* $a^p \equiv p \pmod{p}$. *Ak navyše p nedelí a potom* $a^{p-1} \equiv 1 \pmod{p}$

Definícia 1.1.1. *Nech n je zložené číslo nesúdeliteľné s číslom a. Ak platí* $a^{n-1} \equiv 1 \,(mod\, n)$ *tak povieme, že n je Frematove pseudoprvočíslo pri báze a. (označenie: n je psp(a).*

Aplikovaním tejto vlastnosti dostávame jednoduchý efektívny test zobrazený algoritmom 1.1.

Časová zložitosť algoritmu 1.1: Umocňovať čísla vieme v čase $\mathcal{O}(\log\ n)$ teda časová zložitosť je $\mathcal{O}(\log\ n)$

Algoritmus 1.1 Fermat test

```
1: function IS_PRIME(n, base)
2:     if base^{n-1} ≡ 1 (mod n) then
3:         return true
4:     else
5:         return false
```

Korektnosť algoritmu 1.1: Z malej fermatovej vety (1.1.1) vyplýva, že každé prvočíslo bude prehlásené za prvočíslo pri každej báze. So zloženými číslami je problém. Podľa dát Jána Feitsmu [3] je čísiel, ktoré sú menšie ako 2^{64} a súčasne sú fermatovými pseudoprvočíslami pri báze 2 len 118 968 378. Teda týchto 118 miliónov čísiel bude vyhodnotených zle. Ak by sme vybrali rovnomerne náhodne číslo menšie ako 2^{64}, tak bude s vysokou pravdepodobnosťou vyhodnotené korektne.

Definícia 1.1.2. *Zložené číslo n sa nazýva Carmichaelove číslo práve vtedy, keď je Fermatove pseudoprvočíslo pre všetky nesúdeliteľné bázy s n.*

Podľa [5] je Carmichaelových čísiel nekonečne veľa. Vďaka existencii týchto čísiel Fermatov test nemôže dokazovať prvočíselnosť, pretože nedokáže odlíšiť prvočíslo od Carmichaelovho čísla. Podľa [3] je Carmichaelovych čísiel menších ako 2^{64} len 4279356.

1.2 Eulerove pseudoprvočíslo

Definícia 1.2.1. *(Legendrov symbol) Nech p je prvočíslo a a celé číslo. Potom definujeme*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \exists b \;\; b^2 \equiv a \pmod{p} \\ 0 & (a,p) > 1 \\ -1 & inak \end{cases}$$

Veta 1.2.1. *(Eulerove kritérium) Nech $p > 2$ je prvočíslo, a ľubovoľné celé číslo. Potom platí:*

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod{p}$$

Lema 1.2.1. *Nech p je nepárne prvočíslo a a, b ľubovoľné celé čísla. Potom platia nasledovné rovnosti:*

1. $a \equiv b \pmod{p} \Rightarrow \left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$
2. $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$

Lema 1.2.2. *Nech $p > 2$ je prvočíslo. Potom platí: $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$.*

Veta 1.2.2. *(Kvadratická reciprocita) Nech p, q sú rôzne prvočísla. Potom*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}}$$

Definícia 1.2.2. *(Jacobiho symbol) Nech n, a sú celé čísla. Nech $n = p_0 \dots p_m$ je prvočíselný rozklad čísla n. Potom:*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_0}\right) \dots \left(\frac{a}{p_m}\right)$$

Jacobiho symbol je zovšeobecnenie Legendrovho symbolu na celé čísla.

Lema 1.2.3. *Nech n, a, b sú ľubovoľné celé čísla. Potom platia nasledovné rovnosti:*

1. $a \equiv b \pmod{n} \Rightarrow \left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$
2. $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$

Lema 1.2.4. *Nech n je nepárne prirodzené číslo. Potom platí: $\left(\frac{2}{n}\right) = (-1)^{\frac{n^2-1}{8}}$.*

Nasledujúca veta zovšeobecňuje kvadratickú reciprocitu pre celé čísla.

Veta 1.2.3. *(Kvadratická reciprocita) Nech p, q sú nepárne celé čísla. Potom*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}}$$

Algoritmus na výpočet Jacobiho symbolu

Jacobiho symbol vieme pomocou predchádzajúcich lem efektívne počítať. Chceme zistiť čomu sa rovná hodnota $\left(\frac{a}{b}\right)$. Ak je a párne vieme hodnotu Jacobiho symbolu zapísať ako $\left(\frac{a/2}{b}\right)\left(\frac{2}{b}\right)$ (lema 1.2.3). Hodnotu $\left(\frac{2}{b}\right)$ vieme efektívne zistiť podľa lemy 1.2.4. Ak je b párne vieme to podľa definície Jacobiho symbolu zapísať ako $\left(\frac{a}{b/2}\right)\left(\frac{a}{2}\right)$. Hodnotu $\left(\frac{a}{2}\right)$ vieme jednoducho zistiť podľa parity a. (jediné reziduum modulo 2 je 1). Ak sú a aj b nepárne môžeme použiť kvadratickú reciprocitu (veta 1.2.3) a zredukovať symbol na jednoduchší.

Definícia 1.2.3. *Nech n je nepárne zložené číslo a a celé číslo pričom platí $(a, n) = 1$. Ak platí*

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n} \tag{1.1}$$

tak hovoríme, že n je eulerovo pseudoprvočíslo pri báze a (označenie: n je epsp(a)).

Lema 1.2.5. *Nech n je nepárne zložené číslo. Potom existuje celé číslo a ktoré spĺňa*

$$a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n} \tag{1.2}$$

Dôkaz tejto lemy možno nájst v [15].

Algoritmus 1.2 Solovay Strassen test

```
1: function IS_PRIME(n, k)
2:     for i ← 1, k do
3:         a ←$ {2, ..., n − 1}
4:         x ← (a/n)
5:         if x = 0 ∨ x ≢ a^((n−1)/2) then
6:             return false
7:     return true
```

Zložitosť algoritmu 1.2 Umocňovať čísla vieme v čase $\mathcal{O}(\log n)$ a počítať hodnotu Jacobiho symbolu vieme v čase $\mathcal{O}(\log n)$. Teda je zložitosť $\mathcal{O}(\log n)$

Korektnosť algoritmu 1.2 Ak n je prvočíslo tak podľa vety 1.2.1 je kongruencia určite splnená a vráti sa hodnota true. Ak je n zložené číslo tak podľa lemy 1.2.5 existuje a kedy kongruencia neplatí. Nech $\{a_0, a_1, \dots a_m\}$ sú všetky celé čísla, pre ktoré kongruencia (1.1) platí. Potom

$$(aa_i)^{\frac{n-1}{2}} = a^{\frac{n-1}{2}} a_i^{\frac{n-1}{2}} \equiv a^{\frac{n-1}{2}} \left(\frac{a_i}{n}\right) \not\equiv \left(\frac{a}{n}\right)\left(\frac{a_i}{n}\right) = \left(\frac{aa_i}{n}\right) \pmod{n}$$

Teda ku každému celému číslu, kedy kongruencia (1.1) platí, existuje celé číslo, kedy tá kongruencia neplatí. Nech $\{b_0, \dots b_m\}$ sú všetky celé čísla nekongruentné modulo n kedy kongruencia (1.1) platí a nech a je číslo z lemy 1.2.5. Potom $\{ab_0, \dots ab_m\}$ sú nekongruentné celé čísla modulo n pri ktorých kongruencia 1.1 nie je splnená. Ak by kongruentné boli tak $ab_i \equiv ab_j \pmod{n}$. Keďže $(a, n) = 1$, existuje inverzný prvok vzhľadom na násobenie a po vykrátení dostaneme $b_i \equiv b_j \pmod{n}$ čo je spor. Teda celých čísel, pre ktoré je kongruencia (1.1) splnená, je v grupe Z_n^* najviac $\frac{\varphi(n)}{2}$ a teda pravdepodobnosť, že celé číslo bude prehlásene za prvočíslo je menšia ako $\frac{1}{2}$. Po k cykloch je teda pravdepodobnosť chyby menšia ako 2^{-k}.

1.3 Millerove pseudoprvočíslo

Veta 1.3.1. *Nech p je nepárne prvočíslo $p - 1 = 2^s t$, t je nepárne. Ak p nedelí a*

$$\begin{cases} a^t \equiv 1 \pmod{p} \\ a^{t2^i} \equiv -1 \pmod{p} \quad i \in \{0, \dots, s-1\} \end{cases} \tag{1.3}$$

Dôkaz.

$$\begin{aligned} a^{p-1} \equiv 1 \pmod{p} \Rightarrow p \mid a^{p-1} - 1 \Rightarrow p \mid (a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \\ \Rightarrow p \mid a^{\frac{p-1}{2}} - 1 \quad \vee \quad p \mid a^{\frac{p-1}{2}} + 1 \end{aligned}$$

- $p \mid a^{\frac{p-1}{2}} + 1 \Rightarrow \quad a^{\frac{p-1}{2}} \equiv -1 \pmod{p}$ a teda je splnený prípad 2

- $p \mid a^{\frac{p-1}{2}} - 1 \Rightarrow \quad a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ a teda je buď $\frac{p-1}{2}$ nepárne a platí prípad 1 alebo to znova rozpíšeme.

Definícia 1.3.1. *Nech n je nepárne zložené číslo a b celé číslo. Ak platí 1.3 pre n, b potom hovoríme, že n je silné pseudoprvočíslo pri báze b. (označenie: n je spsp(b)).*

Veta 1.3.2. *(Rabin, Monier) Nech n je nepárne zložené číslo rôzne od 9. Potom*

$$|\{a \mid 1 < a < n \ \wedge \ (a,n) = 1 \ \wedge \ n \text{ je silné pseudoprvočíslo pri báze } a\}| \leq \frac{\varphi(n)}{4}$$

Dôkaz možno nájsť v [11].

Algoritmus 1.3 Rabin Miller test

```
1:  function IS_SPRP(n, e)
2:      a ←$ {2, ..., n − 2}
3:      x ← a^e
4:      if x ≡ 1 (mod n) then
5:          return true
6:      while e ≠ n − 1 do
7:          if x ≡ −1 (mod n) then
8:              return true
9:          x ← x^2
10:         e ← 2e
11:     return false
12: function IS_PRIME(n, k)
13:     t ← n − 1
14:     while t ≡ 0 (mod 2) do
15:         t ← t/2
16:     for i ← 1, k do
17:         if IS_SPRP(n, t) = false then
18:             return false
19:     return true
```

Zložitosť Umocňovať čísla na exponent n vieme v čase $\log n$. Keďže vykonávame k kôl, tak je zložitosť $\mathcal{O}(k \cdot \log n)$.

Korektnosť Ak je n prvočíslo, podľa vety 1.3 ho algoritmus (1.3) vyhodnotí ako prvočíslo pri každej báze, ktorú nedelí. Keďže bázy sú volené zo Z_n táto podmienka je splnená. Ak je n zložené číslo, podľa vety 1.3.2 existuje najviac $\frac{\varphi(n)}{4} < \frac{n}{4}$ báz kedy to silné prvočíslo je. Teda pravdepodobnosť, že sme trafili zlú bázu je najviac $\frac{1}{4}$. Po k iteráciach testu je pravdepodobnosť omylu najviac 4^{-k}.

1.4 Lucasove pseudoprvočíslo

Definícia 1.4.1. *Nech P, Q sú celé čísla. Majme postupnosť $\{a_n\}$ zadanú rekurentným vzťahom:*

$$a_n = Pa_{n-1} - Qa_{n-2} \tag{1.4}$$

Lucasove postupnosti s parametrami P,Q voláme postupnosti $\{u_n\}, \{v_n\}$ pričom $u_0 = 0$, $u_1 = 1$ a $v_0 = 2$, $v_1 = P$ a obe spĺňajú rekurentný vzťah 1.4

Tieto rekurencie možno vyjadriť aj v explicitnom tvare jednoduchým vyriešením parametrickej rekurencie. Rekurenciu si zapíšeme v tvare

$$a_{n+2} - Pa_{n+1} + Qa_n = 0 \tag{1.5}$$

Charakteristická rovnica rekurencie (1.5) je $x^2 - Px + Q$. Nech α a β sú rôzne korene tejto rovnice. Potom všeobecné riešenie rekurencie (1.5) je $a_n = c_1\alpha^n + c_2\beta^n$

$$\begin{aligned}
0 = u_0 &= c_1 + \alpha^0 + c_2\beta^0 = c_1 + c_2 & 2 = v_0 &= c_1 + \alpha^0 + c_2\beta^0 = c_1 + c_2 \\
1 = u_1 &= c_1 + \alpha^1 + c_2\beta^1 = c_1\alpha + c_2\beta & P = v_1 &= c_1 + \alpha^1 + c_2\beta^1 = c_1\alpha + c_2\beta \\
c_1 &= \frac{1}{\alpha - \beta} & c_1 &= 1 \\
c_2 &= \frac{1}{\beta - \alpha} & c_2 &= 1 \\
u_n &= \frac{\alpha^n + \beta^n}{\alpha - \beta} & v_n &= \alpha^n + \beta^n
\end{aligned}$$

Lema 1.4.1. *Platia nasledovné rovnosti:*

$$\begin{aligned}
U_{2n} &= U_n V_n & U_{2n+1} &= (P \cdot U_{2n} + V_{2n})/2 \\
V_{2n} &= V_k^2 - 2Q^k & V_{2n+1} &= (D \cdot U_{2n} + P \cdot V_{2n})/2
\end{aligned}$$

Vďaka tejto leme vieme počítať hodnoty $U_n \pmod x$ a $V_n \pmod x$ v čase $\mathcal{O}(\log n)$ kde x je 32 bitové číslo (aby boli všetky násobenie vykonávané v čase $\mathcal{O}(1)$). Bez tejto lemy to vieme vďaka explicitnému tvaru tiež, ale takto sa nemusíme obávať chýb pri zaokrúhľovaní.

Nasledujúca veta ukazuje ako možno Lucasove postupnosti použiť pri zisťovaní prvočíselnosti.

Veta 1.4.1. *Nech p je nepárne prvočíslo ktoré nedelí Q. Potom platí*

$$u_{p-\left(\frac{D}{p}\right)} \equiv 0 \pmod p \tag{1.6}$$

kde $D = P^2 - 4Q$ (diskriminant charakteristickej rovnice rekurencie (1.5))

Dôkaz možno nájsť v [17].

Definícia 1.4.2. *Lucasove pseudoprvočíslo pre dané P, Q je zložené číslo, pre ktoré platí (1.6)*

Podobne ako sme Fermatov test redukovali na Rabin-Millerov test, môžeme Lucasov test zredukovať na silný Lucasov test.

Veta 1.4.2. *Nech p je prvočíslo, ktoré nedelí $2QD$. Nech $n - \left(\frac{D}{p}\right) = 2^k q$ kde q je nepárne. Potom aspoň jedna z nasledujúcich podmienok je splnená:*

$$\begin{cases} U_q \equiv 0 \pmod{p} \\ V_{2^i q} \equiv 0 \pmod{p} \quad i \in \{0, \ldots, k-1\} \end{cases} \tag{1.7}$$

Veta 1.4.3. *Nech D je celé číslo a n zložené číslo nesúdeliteľné s $2D$ a rôzne od 9. Potom pre všetky D platí, že*

$$\begin{aligned} SL(D,n) = \{(P,Q) \mid\ & 0 \le P, Q < n, \\ & (Q,n) = 1, \\ & P^2 - 4Q \equiv D \pmod{n}, \\ & n \text{ je silné Lucasove pseudoprvočíslo s parametrami } P,Q\} \end{aligned}$$

veľkosť $SL(D,n)$ je menšia ako $\frac{4}{15}n$ okrem prípadu keď $n = (2^{k_1}q_1 + 1)(2^{k_1}q_1 - 1)$ je súčin dvoch prvočíselných dvojčiat kde q_1 je nepárne a $\left(\frac{D}{2^{k_1}q_1 - 1}\right) = -1$ a $\left(\frac{D}{2^{k_1}q_1 + 1}\right) = 1$ Vo všeobecnosti platí, že veľkosť množiny $SL(D,n)$ je menšia ako $\frac{n}{2}$.

Dôkaz tejto vety možno nájsť v [6].

Vďaka platnosti tejto vety môžeme zostrojiť pravdepodobnostný test, ktorý bude mať na vstupe číslo n. Test overí, či je n párne alebo 9. Potom vygeneruje náhodné $P, Q \in Z_n$ pričom $(Q,n) = 1$ a spočíta hodnotu $D = P^2 - 4Q$. Následne overí, či platí veta (1.4.2) pre n s parametrami P, Q. Ak nie, tak n je určite zložené číslo. Ak áno, treba overiť či n nie je súčin prvočíselných dvojčiat teda $n = p(p+2)$. To sa overí napríklad tak, že zistíme či $n+1$ je perfektný štvorec. Ak áno, tak n je zložené číslo. Ak nie, tak n je s pravdepodobnosťou najviac $\frac{4}{15}$ prvočíslo. Ak číslo n k-krát úspešne prejde týmto testom, tak je s pravdepodobnosťou menšou ako $(\frac{4}{15})^k$ zložené číslo.

Všimnime si, že tento test je pomalší ako Rabin-Millerov test, pretože treba overovať viac podmienok a taktiež má horšiu presnosť. Preto sa tento test používa v inom štýle.

1.5 Test prvočíselnosti Baille-PSW

Lucas Selfridge Nasledovný test je upravený Lucasov test. Namiesto špecifikovania hodnôt P, Q, Selfridge navrhol nájsť najprv hodnotu D takú, že platí $\left(\frac{D}{n}\right) = -1$ kde

n je testované číslo. Pôvodný Lucasov test túto podmienku nemá a funguje aj bez nej. Následne zvolí $P = 1$ a dopočíta hodnotu $Q = \frac{\sqrt{1-D}}{4}$. Celý test prebieha nasledovne:

1. Over či n nie je štvorec.
2. Nájdi v postupnosti $A = \{(5+2i)(-1)^i\}_{i=0}$ prvý člen, pre ktorý platí $\left(\frac{a_i}{n}\right) = -1$.
3. Vypočítaj hodnoty P, Q
4. Over či platí kongruencia $U_{n+1} \equiv 0 \pmod{n}$

Prvý krok je nutný pre existenciu člena postupnosti s hodnotou Jacobiho symbolu -1. Platí totiž nasledovná lema.

Lema 1.5.1. *n je perfektný štvorec* $\Leftrightarrow \forall x \in Z_n^* : \left(\frac{x}{n}\right) = 1$

Autori tohto testu navrhli podmienku na Jacobiho symbol -1 preto, lebo pre určité hodnoty P,Q,D je tento test len obyčajný fermatov test. S obmedzením pre Jacobiho symbol táto situácia nenastáva. Autori tiež uvádzajú, že v priemere stačí otestovať najviac 2 hodnoty postupnosti A, preto pri implementácii sa oplatí kroky 1 a 2 vymeniť. Keď dostatočne skoro nenájdeme vhodné D, overíme či náhodou nemáme na vstupe štvorec. V článku [7] je viac dôvodov ako voliť parametre P a Q. Tento postup voľby argumentov P a Q sa dá taktiež použiť pre silnejšiu verziu Lucasovho testu.

BPSW Nasledovný test je kombináciou dvoch predchádzajúcich testov.

1. over či je n párne alebo menšie ako 3
2. over či je n silné pseudoprvočíslo pri báze 2
3. over či je n (silné) Lucasove pseudoprvočíslo s parametrami P, Q podľa Selfridga

Tento test je deterministický a 100%-korektný pre hodnoty menšie ako 2^{64}. Nevie sa ani o žiadnom väčšom zloženom čísle, ktoré by tento test prehlásil za prvočíslo.

KAPITOLA 2

Deterministické testy s obmedzeným vstupom

V tejto časti sa zameriame na deterministické testy prvočíselnosti pre čísla menšie ako 2^{32}. Predchádzajúce testy sú veľmi efektívne a fungujú dobre pre veľké čísla. Čo by sa ale stalo, keby sme ohraničili možné vstupy zhora? Buď by sme mohli používať rovnaký test, ktorý ale nedokazuje prvočíselnosť, alebo by sme ho mohli modifikovať aby bol 100%-korektný a dokazoval prvočíselnosť. Najväčší problém predchádzajúcich testov je, že nie sú deterministické. Nemáme teda žiadnu záruku, ktoré bázy sa budú testovať pre rôzne volania daného testu. Keby sme tieto testy nejakým spôsobom determinizovali a zachovali ich korektnosť, dostali by sme test, ktorý je korektný a dokazuje prvočíselnosť. Existuje niekoľko spôsobov ako predchádzajúce pravdepodobnostné testy opraviť aby prvočíselnosť dokazovali. V tejto sekcii uvedieme niekoľko rôznych prístupov ako upraviť Miller-Rabinov test pre horné ohraničenie 2^{32}.

Najjednoduchší spôsob ako upraviť Miller-Rabinov test je povedať koľko rôznych báz treba vyskúšať aby sme mali istotu, že dané číslo je alebo nie je prvočíslo. Preto sme pre každé zložené číslo, ktoré nie je deliteľné 2,3,5,7 spočítali 128-bitový vektor, v ktorom sme mali na i-tej pozícii uloženú informáciu, či dane číslo je silné pseudoprvočíslo pri báze $(i + 2)$ alebo nie. Následne sme tieto dáta sekvenčne prešli a zrátali koľko čísiel je takých, že z 128 báz sú silné pseudoprvočísla pre j-ľubovoľných z nich. Výsledky ukazuje tabuľka 2.1.

Vďaka tabuľke 2.1 dostávame jednoduchý korektný test zobrazený algoritmom 2.1, ktorý stále nie je deterministicky.

Takýto test je síce korektný (lebo žiadne číslo z daného rozsahu nie je silné pseudoprvočíslo pre viac ako 70 báz) no nie je efektívny pretože treba až 71 kôl Miller-Rabinovho

X	Y	X	Y	X	Y	X	Y	X	Y
0	778293598	15	253	30	64	45	3	60	0
1	88530	16	235	31	53	46	2	61	0
2	18035	17	229	32	73	47	1	62	0
3	7780	18	189	33	60	48	2	63	0
4	4725	19	204	34	53	49	3	64	0
5	2508	20	159	35	46	50	1	65	0
6	1815	21	182	36	30	51	1	66	0
7	2165	22	163	37	38	52	1	67	0
8	1186	23	130	38	16	53	3	68	0
9	938	24	112	39	16	54	0	69	0
10	724	25	102	40	19	55	0	70	1
11	585	26	86	41	8	56	1	71	0
12	493	27	93	42	6	57	0	72	0
13	416	28	60	43	2	58	0	73	0
14	333	29	56	44	5	59	1	74	0

Tabuľka 2.1

Algoritmus 2.1

```
1  int prime(uint32_t x) {
2    if(x==2 || x==3 || x==5 || x==7) return true;
3    if(x%2==0 || x%3==0 || x%5==0 || x%7==0) return false;
4    if(x<121) return x!=1;
5    for(int i=0;i<71;++i) {
6      uint32_t base = 0;
7      while(used[base] || base==0) base = rand()%128+2;
8      used[base]=true;
9      if(!is_SPRP(x,base)) return false;
10   }
11   return true;
12 }
```

množina báz	**horný limit**	**autor**	**referencia**
$\{2,3,5,7\}$	118,670,087,467	PSW	[12] [10]
$\{2,7,61\}$	4,759,123,141	Jaeschke	[10]
M[1]	2^{64}	Sinclair	[4]

[1] $M = \{2,\ 325,\ 9375,\ 28178,\ 450775,\ 9780504,\ 1795265022\}$

Tabuľka 2.2

testu nehovoriac o kontrole opakovaní báz. Veľa rôznych ľudí sa už zaoberalo spôsobom ako zdeterminizovať tento test aby dokazoval prvočíselnosť. Tabuľka 2.2 sumarizuje doteraz objavené výsledky.

Ako si môžeme všimnúť, ak chceme korektný test, ktorý by zahŕňal všetky čísla reprezentované 32-bitovým vektorom, potrebujeme minimálne 3 kolá Rabin-Millera. V porovnaní s 71 kolovým testom je táto cesta efektívnejšia. Otázka znie, či sa to nedá

spraviť na menej kôl. Bohužiaľ, nie je známy žiadny test, ktorý by používal len jednu alebo dve bázy. Najlepší test, ktorý používa 2 bázy má horný limit 1 050 535 501. Preto treba zmeniť stratégiu.

2.1 Hashovanie

Odpoveďou pre 1-kolový test, či už je to Rabin-Miller, Euler alebo iné, je hashovanie. Princíp je rozdeliť si množinu čísiel, ktorých prvočíselnosť chceme dokazovať na N disjunktných množín. Dve čísla patria do rovnakej množiny práve vtedy, keď majú rovnaký hash. Následne stačí nájsť pre každú takúto množinu čísiel funkčnú bázu.

Poznámka 2.1.1. *Hovoríme že báza i je funkčná pre množinu M ak platí* $\forall x \in M : \ x \in \mathbb{P} \Leftrightarrow x$ *prejde Rabin-Millerov test pri báze* i

Algoritmus 2.2

```
uint32_t base[];
int hash(uint32_t x);
int prime(uint32_t x) {
  if(x==2 || x==3 || x==5 || x==7) return true;
  if(x%2==0 || x%3==0 || x%5==0 || x%7==0) return false;
  if(x<121) return x!=1;
  return is_SPRP(x,base[hash(x)]);
}
```

Tento test by bol deterministický a korektný, ak by sme poznali hashovaciu funkciu h a pole báz *base*. V takom prípade pamäťová zložitosť tohto testu by bola úmerná veľkosti poľa *base* a časová zložitosť by bola (jedno kolo Rabin-Millerovho testu + časová zložitosť hashu). Zvolili sme v metaalgoritme 2.2 Rabin-Millerov test, lebo z predchádzajúcich testov má najlepšie vlastnosti (pravdepodobnosť chyby $\leq \frac{1}{4}$) a najmenšiu časovú zložitosť.

2.2 Hľadanie vhodnej funkcie

Keďže nasledujúce algoritmy sú výpočtovo náročné, zvolili sme programovací jazyk `C++` kvôli rýchlosti.

Poznámka 2.2.1. *Nech* $M = \{x \mid 0 \leq x < 2^{32} \ \wedge \ (x, 210) = 1 \ \wedge \ x \notin \mathbb{P}\}$

2.2.1 Spôsob 1

Prvý spôsob hľadania bol veľmi jednoduchý. Algoritmus sa skladal z dvoch vnorených cyklov, ktoré iterovali cez všetky 32 bitové hodnoty a testovali ich správanie voči daným

bázam. Do poľa `witness` si potom zapísal počet operácii, ktoré boli potrebné na odhalenie, či dané číslo je alebo nie je prvočíslo. Na záver zo všetkých funkčných báz vybral tú, ktorá spravila minimálny počet operácii. Ako kontrolný algoritmus na zistenie či je číslo prvočíslo sme použili Eratostenovo sito.

```
int witness[1024][max_base];
for(uint32_t i=121;i>0;++i) {
  if(i%2==0 || i%3==0 || i%5==0 || i%7==0 || prime(i)) continue;
  for(long long j=0;j<max_base;++j) {
    //over či je báza dobrá pre danú triedu
    if(witness[hash(i)][j]==-1) continue;
    if(ops=is_SPRP(i,j+2)) witness[hash(i)][j]+=ops;
    else witness[hash(i)][j]=-1;
  }
}
```

Veľkosť použitej pamäte: veľkosť poľa `witness` + veľkosť pola, ktoré používa eratostenovo sito = 4·max_base KiB + 256 MiB.

Výhody: Algoritmus vypočíta bázy, ktoré spravia minimálny počet operácii spomedzi všetkých testovaných báz. Jeho pamäťová zložitosť je akceptovateľná.

Nevýhody: Nevýhod tejto metódy je veľa a jednou z nich je jej rýchlosť. Počas niekoľkých hodín bežania programu sa nám nepodarilo otestovať všetky 32 bitové čísla pri všetkých bázach z daného rozsahu. Aj z týchto dôvodov sme netestovali všetky 32 bitové hodnoty ale len tie, ktoré patria do množiny M. Čas behu programu bol aj tak neprijateľný. Ďalší problém tejto metódy je, že vie testovať len málo báz. Časová zložitosť je závislá lineárne od tejto premennej, takže zvýšením rozsahu báz sa algoritmus len spomalí, čo už je v tomto stave neprijateľné. Ak by sa ukázalo, že daná hashovacia funkcia nemá riešenie pre daný rozsah báz, bolo by treba vyskúšať novú hashovaciu funkciu, čo znamená spustiť už aj tak pomalý proces odznova. Toto bol hlavný dôvod, prečo sme upustili od tohto riešenia, keďže si nijakým spôsobom neukladá už vypočítané dáta. Napríklad vie, že číslo 47 sa správa pri bázach 2,3,5 nejakým spôsobom ale pri zmene hashovacej funkcie to zisťuje znova. Oprava tejto nevýhody viedla k nasledujúcemu riešeniu.

2.2.2 Spôsob 2

Tento spôsob je založený na myšlienke investovania nejakého času do predvýpočtu dát. Následne použitie predvypočítaných dát je veľmi rýchle a preto je táto metóda efektívnejšia ako predchádzajúca. Taktiež sme upustili od myšlienky nájsť bázu, ktorá spraví minimálny počet operácii a uspokojili sa s bázou, ktorá funguje. Vďaka tomu sme mohli vymeniť poradie cyklov zo spôsobu 1 a prerobiť algoritmus na 3 fázy.

1. fáza Táto fáza sa vykonáva len raz. Pri zmene hashovacej funkcie sa táto fáza už vykonávať nemusí (čo je rozdiel oproti prvému spôsobu, ktorý nebol odolný voči zmenám). Algoritmus pre každý prvok z množiny M vygeneruje 32-bitový vektor, v ktorom má na i-tom bite uloženú informáciu či báza $(i+2)$ vhodná pre toto číslo. Vygenerované dáta uloží do súboru, pretože ich veľkosť sa pohybuje okolo 3GB.

```
for(ull i=start;i<end;i+=2) {
  if(i%3==0 || i%5==0 || i%7==0 || prime(i)) continue;
  for(ull j=2;j<34;++j) {
    // baza je vhodna
    if(is_SPRP(j,i)==0) buffer[buffer_pos]|=(1<<(j-2));
  }
  ++buffer_pos;
  if(buffer_pos==buffer_size) {
    write_buffer(buffer,buffer_size,dest);
    buffer_pos=0;
  }
}
```

2. fáza Zvyšné 2 fázy sa vykonávajú pre každú testovanú hashovaciu funkciu. Pazravý algoritmus spracuje predvypočítané dáta nasledovne. Pre každé číslo x z množiny M nájde jeho prislúchajúci bitový vektor. Následne číslu x vypočíta jeho hashovaciu hodnotu a bitový vektor zahrnie k doterajšiemu riešeniu. Po skončení má uložené v poli `bases` odpovede, či pre dané triedy existuje funkčná báza z rozsahu $2-33$. Všetky triedy, ktorým nebola priradená báza vo fáze 2 postupujú do tretej fázy. Ak neurčených tried bolo viac ako $\frac{1}{4}$ zo všetkých, program vyhlási hashovaciu funkciu za zlú a treba skúšať druhú. Toto obmedzenie tam je z dôvodu obmedzenej pamäte.

```
//otvor súbor vygenerovaný z prvej fázy...
for(long long i=121;i<n;i+=2) {
  if(i%3==0 || i%5==0 || i%7==0 || prime(i)) continue;
  bases[hashh(i)]&=buffer[buffer_pos++];
  if(buffer_pos==buffer_size) {
    fread(buffer,4,buffer_size,f);
    buffer_pos=0;
  }
}
//trieda i má dobré bázy z rozsahu 2 ... 33 zapísane v poli bases[i]
```

3. fáza Algoritmus iteruje cez všetky hodnoty z množiny M a každej priradí jej hash. Hodnoty z tried, ktorým nebola určená báza, sú uložené do vectora. Následne sa pre každý vektor vypočíta vhodná báza.

```
for(x in M) {
  if(bases[hashh(x)]==0) vector[map[hashh(x)]].push_back(x)
}
for(int i=0;i<vector.size();++i) {
  nájdi vhodnú bázu pre daný vektor
}
```

Tento postup je efektívny pre hashovacie funkcie, ktorých triedy majú funkčné bázy položené veľmi nízko, pretože málo čísiel sa dostane do fáze 3. Hľadanie vhodných báz trvalo od 10 minút do 2 hodín čo je podľa nás prijateľný čas. Na ukážku uvádzame jeden výsledok vygenerovaný týmto riešením v algoritme 2.3.

Algoritmus 2.3

```
int hash(uint32_t x) {
  return (((long long)0xAFF7B4*x)>>7)&1023;
}
uint8_t base[] = {  17, 11,  6, 60,  7, 13, 11, 34, 13,  2,  3, 37, 13, 11, 38,  2,  7,105,
  2,  7, 42, 11,  7,  3,  6, 15, 53, 44,  6,  6,  5, 15, 54,  7, 35, 10, 10, 15, 10, 10, 17,
 17, 11, 10, 15, 43,  7,  5,  5,  3,  7, 43, 34,  2, 34,  2, 68, 53, 39, 10,  7,  6, 11,  2,
  5,  2,  7,  2,  6,  5, 15, 40,  3,  5,  5,  2,  2, 10, 47, 13,  7, 43,  6,  7,  5,  6,  6,
 13,  6, 35,  6, 15,  6, 13, 40, 10, 11,  2,  7,  2,  2,  3, 13,  3, 11, 15, 10,  5, 11, 14,
  7, 11, 47,  5,  2,  2,  6,  2,  5, 55,  6,  5,  7,  2,  6, 58, 35, 11,  5, 12, 17,  6, 10,
 12,  6,  6,  2, 53,  2,  2, 13,  5, 14,  7, 15,  6, 13, 62, 10,  6,  3,  7,  7,  3, 14,  5,
 14, 73, 15, 11, 11,  6,  5, 17, 10,  5,  3, 37, 51, 10,  7,  5, 38, 12,  5, 11,  5,  7,  6,
  5,  6, 40, 43, 57, 10, 13,  7, 15,  2, 10, 34,  7, 39, 10,  5,  3,  6, 13, 11,  5, 10, 43,
 10,  5,  3, 14,  5,  2,  5, 41,  5, 39, 46,  2, 10,  2,  5, 12,  3,  2,  2,  5, 15, 43, 17,
 41,  2, 13, 15, 38, 11, 11,  3, 34,  5,  6,  3,  7,  2, 37,  5,  6, 10, 17, 35,  2, 15,  6,
  7,  5,  3, 13, 13, 12, 34,  2, 12, 10, 15, 13,  2,  2, 34,  6,  6,  5,  2,  7, 13,  3,  6,
 11, 39, 42,  7,  2,  6, 39, 47,  3, 17,  5, 13,  7,  2, 47,  3,  7,  6, 11, 17, 37, 48,  7,
 37, 11,  7, 10,  3, 14, 39, 14, 15, 43, 17,  2, 12,  7, 13,  5,  3,  6, 34, 37,  3, 17, 13,
  2,  5, 10, 10, 44, 37,  2,  2, 10, 10,  7,  3,  7,  2,  7,  5, 43, 43, 11, 15, 51, 13, 17,
 10, 11,  2,  5, 34, 17,  2,  2, 42,  6,  6,  5, 47, 15,  2, 12,  7,  3, 10, 15,  3,  7, 12,
 12, 15, 43, 14,  7, 58, 13, 10,  6,  6, 38, 34,  5,  5, 13, 38,  6, 11, 10,  6,  7,  2, 55,
  2, 13,  5, 11, 44, 15, 17,  2, 40,  2, 15, 13,  6,  2,  3,  3,  3,  3,  6, 39,  5, 11, 17,
 37,  5,  7,  6, 10,  6, 12,  7,  5, 14, 10, 12, 71, 10, 35,  6, 11,  3,  2, 38,  3,  2, 34,
 10, 17, 42,  2, 12,  6,  6, 11, 40, 12, 10,  6, 10,  2,  3,  3, 56, 11,  7, 42,  2, 38, 12,
  2,  2, 13, 40, 12,  6,  5,  5, 59, 15, 38,  5,  5,  5,  7,  2, 10,  7,  2, 17, 10, 11,  6,
  6,  6,  2, 10,  6, 54,  2, 82,  3, 34, 14, 15, 44,  5, 46,  2, 13,  5, 12, 13, 11, 10, 39,
  5, 40,  3, 60,  3, 42, 11,  3, 46, 17,  3,  2, 37,  6, 42, 12, 14,  3, 12, 66, 13, 34,  7,
  3, 13,  3, 11,  2, 13, 12, 38, 34,  5, 40, 10, 14,  6, 14, 11, 38, 58,  2, 48,  5, 15,  5,
 73,  3, 37,  5, 11, 10,  5,  5, 13,  2, 10, 13, 34, 17,  3,  7, 47,  2,  2, 10, 15,  3,  3,
 13,  6, 34, 13, 10, 13,  3,  6, 41, 10,  6,  2,  6,  2,  6,  2,  6,  6, 37, 10, 44, 35, 13,
 51,  2,  7, 53,  5, 40,  5,  2, 37, 11, 15, 11, 13,  2,  5,  2,  6, 10, 17, 15, 43, 39, 17,
  2, 12, 10, 15, 17,  7, 13,  3,  7, 15, 37,  5, 15,  7,  6, 10, 51,  2,  2, 40, 61,  2, 13,
 13, 11,  2,  5, 34,  5,  5,  7,  2,  2,  2, 11,  3,  6, 13,  6, 17, 11, 10,  7, 46, 15,  7,
 14, 35, 11,  7, 10,  6, 11, 40, 11,  2, 39,  7,  6, 66,  5,  3,  6,  5, 11, 10,  2, 10,  7,
 13,  2, 45, 34,  6, 35,  2, 11,  5, 59, 75, 10, 17, 14, 17, 17, 17,  2, 11,  7, 10,  6, 11,
  6, 56, 34, 35, 11, 14, 12, 41, 40, 17, 40,  3, 11,  7, 37, 14,  7, 13,  7,  5,  2, 10,  6,
 39,  2,  7, 37, 35, 10,  5, 15,  2,  7, 38, 34, 11, 17,  5,  6, 10,  3,  6,  7,  7, 43, 14,
  2, 43,  3,  2, 47,  7, 35,  7,  3, 53,  2, 10, 10, 10, 60, 10,  6,  2,  6, 10,  5,  7, 57,
 53, 13,  3, 35, 38, 15, 42,  3,  3, 12,  2, 10,  3, 38, 54, 13, 10, 11,  7, 13,  7,  2, 12,
 39, 10, 54,  2, 12, 38, 10, 12, 12,  5, 15,  6, 10, 13,  5, 15, 10, 13,  6, 41, 40, 14, 12,
 10, 11, 40,  5, 11, 10,  2,  5,  2, 13,  6,  2, 13,  5,  2, 10, 15,  5,  5, 10, 34, 13,  2,
  5, 14,  5,  6,  5, 13,  3, 43,  6, 13, 11, 50,  3,  6,  6, 12, 15, 11, 37,  7, 69, 11, 14,
 14,  7, 43,  5, 35, 11, 35, 11, 11, 34, 34, 39, 14, 11,  2, 10, 53,  6, 11,  2, 11, 60, 39,
 11,  6, 15, 40, 17, 47, 34, 50,  7, 59, 47,  5, 13, 39,  5,  6, 53, 10, 14,  5, 51,  5,  7,
  5,  6, 77,  7, 12,  7, 42,  2,  5,  2,  6, 60, 10, 13, 10,  6, 47,  6, 15, 17, 10, 11, 10,
 12,  7,  7, 10, 17, 34,  5, 10,  7,  7,  2,  6, 10, 38,  2, 15,  6, 13,  7, 13,  2,  3, 13,
  5,  3, 17,  2,  5, 15, 11, 39,  7, 39, 10, 10,  2,  6, 13,  3,  5, 17,  6, 14, 10, 37, 44,
  3, 34,  5, 11,  7, 12,  2,  5,  3, 12,  3,  2,  3,133, 12,  2,  2,  2,  3, 34, 14, 41,  2,
 37, 11,  2,  6, 11,  6,  7, 15, 11, 35, 13,  6,  5,  2, 14,  7,  2 };
```

Algoritmus 2.4 BASE64 kódované bázy z algoritmu 2.3

```
EQsGPAcNCyINAgMlDQsmAgdpAgcqCwcDBg81LAYGBQ82ByMKCg8KChERCwoPKwcFBQMHKyICIgJE
NScKBwYLAgUCBwIGBQ8oAwUFAgIKLw0HKwYHBQYGDQYjBg8GDSgKCwIHAgIDDQMLDwoFCw4HCy8F
AgIGAgU3BgUHAgY6IwsFDBEGCgwGBgI1AgINBQ4HDwYNPgoGAwcHAw4FDkkPCwsGBREKBQMlMwoH
BSYMBQsFBwYFBigrOQoNBw8CCiIHJwoFAwYNCwUKKwoFAw4FAgUpBScuAgoCBQwDAgIFDysRKQIN
DyYLCwMiBQYDBwIlBQYKESMCDwYHBQMNDQwiAgwKDw0CAiIGBgUCBw0DBgsnKgcCBicvAxEFDQcC
LwMHBgsRJTAHJQsHCgMOJw4PKxECDAcNBQMGIiUDEQ0CBQoKLCUCAgoKBwMHAgcFKysLDzMNEQoL
AgUiEQICKgYGBS8PAgwHAwoPAwcMDA8rDgc6DQoGBiYiBQUNJgYLCgYHAjcCDQULLA8RAigCDw0G
AgMDAwMGJwULESUFBwYKBgwHBQ4KDEcKIwYLAwImAwIiChEqAgwGBgsoDAoGCgIDAzgLByoCJgwC
AgOoDAYFBTsPJgUFBQcCCgcCEQoLBgYGAgoGNgJSAyIODywFLgINBQwNCwonBSgDPAMqCwMuEQMC
JQYqDA4DDEINIgcDDQMLAg0MJiIFKAoOBg4LJjoCMAUPBUkDJQULCgUFDQIKDSIRAwcvAgIKDwMD
DQYiDQoNAwYpCgYCBgIGAgYGJQosIw0zAgc1BSgFAiULDwsNAgUCBgoRDysnEQIMCg8RBw0DBw8l
BQ8HBgozAgIoPQINDQsCBSIFBQcCAgILAwYNBhELCgcuDwcOIwsHCgYLKAsCJwcGQgUDBgULCgIK
Bw0CLSIGIwILBTtLChEOERERAgsHCgYLBjgiIwsODCkoESgDCwclDgcNBwUCCgYnAgclIwoFDwIH
JiILEQUGCgMGBwcrDgIrAwIvByMHAzUCCgoKPAoGAgYKBQc5NQ0DIyYPKgMDDAIKAyY2DQoLBw0H
AgwnCjYCDCYKDAwFDwYKDQUPCg0GKSgODAoLKAULCgIFAg0GAg0FAgoPBQUKIg0CBQ4FBgUNAysG
DQsyAwYGDA8LJQdFCw4OBysFIwsjCwsiIicOCwIKNQYLAgs8JwsGDygRLyIyBzsvBQ0nBQY1Cg4F
MwUHBQZNBwwHKgIFAgY8Cg0KBi8GDxEKCwoMBwcKESIFCgcHAgYKJgIPBg0HDQIDDQUDEQIFDwsn
BycKCgIGDQMFEQYOCiUsAyIFCwcMAgUDDAMCA4UMAgICAyIOKQIlCwIGCwYHDwsjDQYFAg4HAg==
```

Každá testovaná hashovacia funkcia, ktorá hashovala svoje hodnoty do 1024 prvkov a žiadnych 8000 po sebe idúcich prvkov nemalo rovnaký hash, mala riešenie, čo je zaujímavé. Preto sme sa zaoberali tým, akú minimálnu veľkosť hashovacej funkcie treba zvoliť, aby existovalo riešenie.

2.3 Analýza veľkosti hashovacej funkcie

Definícia 2.3.1. *Nech h je hashovacia funkcia $h: \{0,\dots,2^{32}-1\} \to X$ kde $X = \{0,\dots,s-1\}$. Funkciu h nazveme hashovacou funkciou integerov a veľkosťou hashovacej funkcie označíme hodnotu $|X|$ teda s. Množinu týchto funkcií označíme $\mathscr{H}_s$*

Definícia 2.3.2. *Nech h je hashovacia funkcia integerov rovnomerne náhodne zvolená z množiny $\mathscr{H}_s$. Takto zvolenú hashovaciu funkciu h označíme za náhodnú.*

Lema 2.3.1. *Nech $h \in \mathscr{H}_s$ je náhodná funkcia. Potom platí*

$$\forall x \in \{0,\dots,2^{32}-1\},\ \forall y \in \{0,\dots,s-1\}: \quad P[h(x)=y] = \frac{1}{s}$$

kde P je pravdepodobnostná miera.

Definícia 2.3.3. *Nech h je hashovacia funkcia integerov. Hovoríme, že funkcia h má riešenie pre rozsah báz r, ak existuje pole base veľkosti hashovacej funkcie a platí: $\forall i \in \{0\dots 2^{32}-1\}$ i prejde Rabin-Millerov test pri báze $\texttt{base}[h(i)] \Leftrightarrow i \in \mathbb{P}$ a $\forall j \quad 0 \leq j < |\texttt{base}| \quad 1 < \texttt{base}[j] < r$.*

Cieľ: V nasledujúcej sekcii budeme hľadať odpoveď na otázku: Ak h je náhodná hashovacia funkcia integerov, aká je pravdepodobnosť, že existuje riešenie pre túto funkciu? Najprv skúsime odhadnúť túto pravdepodobnosť teoreticky a potom sa budeme

snažiť naše výpočty overiť prakticky. Na základe týchto dát sa budeme snažiť nájsť čo najmenšiu hashovaciu funkciu s konkrétnym riešením, ako aj odhadnúť minimálnu veľkosť hashovacej funkcie.

2.3.1 Teoretický odhad (heuristický)

Pripomeňme ešte, že tieto odhady robíme vzhľadom na metaalgoritmus 2.2 kde sme uvažovali množinu $M = \{x \mid 0 \leq x < 2^{32} \ \wedge \ (x, 210) = 1 \ \wedge \ x \notin \mathbb{P} \}$

Označenie 2.3.1. *$OUT(i) = \{x \mid x \in M \ \wedge \ x$ je silné pseudoprvočíslo pri báze $i \}$. Veľkosť tejto množiny označíme $out(i) = |OUT(i)|$.*

$OUT(i)$ je vlastne množina všetkých zložených čísiel, ktoré oklamú bázu i v našom teste. Ak je i množina, tak triviálne to bude znamenať, že prvky danej množiny sú silné pseudoprvočísla pre každý prvok z množiny i.

Odhad: Treba podotknúť, že tento odhad je naozaj odhad, pretože porušuje niekoľko viet z pravdepodobnosti ako ukážeme nižšie. Navyše nie je jasné, ako veľmi sa odlišuje od reálnej hodnoty.

Ak je h náhodná hashovacia funkcia integerov s veľkosťou s, ktorá má riešenie pre rozsah báz r, potom každá jej trieda musí mať riešenie. Teda pre každú triedu hashovacej funkcie existuje funkčná báza. Pravdepodobnosť existencie funkčnej báze pre jednu triedu môžeme odvodiť nasledovne

$$p_i = \left(\frac{s-1}{s}\right)^{out(i)}$$

p_i udáva pravdepodobnosť, že báza i je vhodná pre danú triedu, lebo je to pravdepodobnosť udalosti, že všetky čísla ktoré "kazili" danú bázu sa zahashujú do inej triedy. My ale hľadáme pravdepodobnosť, že existuje aspoň jedno riešenie. Teda pravdepodobnosť komplementárnej udalosti k udalosti, že neexistuje riešenie teda všetky bázy nevyhovujú.

$$q = \prod_{2}^{r-1}(1 - p_i)$$

q udáva pravdepodobnosť, že všetky bázy z rozsahu sú nevyhovujúce. Pravdepodobnosť existencie riešenia pre jednu triedu je teda $Q = (1 - q)$. Hľadaný odhad pravdepodobnosti riešenia pre celú hashovaciu funkciu sa potom spočíta jednoducho. Funkcia má riešenie, keď jej všetky triedy majú riešenie teda

$$P[\text{funkcia h má riešenie}] = Q^s$$

Poznámka 2.3.1. *Ak sú dve udalosti nezávislé, potom sú aj komplementárne udalosti nezávislé. Ak máme dve nezávislé udalosti A, B potom $P[A \cap B] = P(A)P(B)$.*

Pravdepodobnosť q je vyrátaná s chybou. Udalosti ktoré prislúchajú k pravdepodobnostiam $(1 - p_i)$ a $(1 - p_j)$ nie sú nezávislé a vo výpočte sa predpokladá že sú. Ak by nezávislé boli, tak by boli nezávislé aj udalosti prislúchajúce k pravdepodobnostiam p_i a p_j. Lenže pravdepodobnosť, že pre danú triedu funguje báza i aj j je

$$x = \left(\frac{s-1}{s}\right)^{out(\{i,j\})}$$

čo je iná hodnota ako $p_i \cdot p_j$, pretože existujú čísla ktoré sú silné pseudoprvočísla pre obe tieto bázy.

Tento už aj tak nepresný odhad musíme nejakým spôsobom aproximovať, pretože na jeho výpočet potrebujeme poznať hodnoty $out(i)$ pre všetky i menšie ako r. Spôsobom 2, ktorý bol uvedení na hľadanie riešenia hashovacej funkcie sme rozšírili predvýpočet z 32 bitových vektorov na 128 bitové. Teda o každom čísle z množiny M sme mali informácie ako sa správa vzhľadom na bázy $2 \ldots 129$. Tento výpočet trval okolo 50 hodín (zdĺhavý výpočet bol spôsobený absenciou paralelizmu). Tento odhad potrebujeme počítať pre väčší rozsah báz ako 129, a preto treba nájsť spôsob aproximácie.

Aproximácia odhadu 1 Prvá možnosť ako tento odhad rozšíriť pre väčší počet báz je, že budeme predpokladať približne rovnakú distribúciu silných pseudoprvočísiel pre každú bázu. Teda každých po sebe idúcich 128 báz bude mať rovnako veľa out-ov ako prvá 128-mica. Formálne:

$$out((i-2)) = out((i-2) \mod 128)$$

Výpočet odhadu teda bude vyzerať

$$P[\text{funkcia h má riešenie}] = \left(1 - \left(\prod_{2}^{129}\left(1 - \left(\frac{s-1}{s}\right)^{out(i)}\right)\right)^{\frac{r}{128}}\right)^{s}$$

Aproximácia odhadu 2 Druhá možnosť ako aproximovať je nasledovná. Predpokladajme, že každá báza má rovnaký počet čísel ktoré ju kazia. $\forall x, y \in \mathbb{N},\ x \geq 2,\ \ y \geq 2\ out(x) = out(y)$. Tento počet určíme ako priemer out-ov spomedzi báz 2 až 129. Teda dostávame vzorec.

$$P[\text{funkcia h má riešenie}] = \left(1 - \left(1 - \left(\frac{s-1}{s}\right)^{avg}\right)^{r}\right)^{s}$$

Kde $avg = \frac{1}{128}\sum_{i=2}^{129} out(i) = 2470.9$

2.3.2 Praktický výpočet pravdepodobnosti

Zatiaľ nevieme ako presné sú tieto aproximácie odhadu. Preto sa potrebujeme pokúsiť prakticky spočítať pravdepodobnosť, že hashovacia funkcia má riešenie pre dané rozsahy s danými veľkosťami. Cieľom bude vygenerovať sériu grafov pre rôzne rozsahy báz, kde každý graf bude zobrazovať závislosť medzi pravdepodobnosťou existencie riešenia s rozsahom báz r pre náhodné hashovacie funkcie a veľkosťou hashovacej funkcie. Každý bod na týchto grafoch zostrojíme nasledovným spôsobom. Pre daný rozsah r a veľkosť s vygenerujeme 25 náhodných hashovacích funkcií. Pre každú z týchto funkcií sa pokúsime nájsť riešenie a zistíme počet funkcií ktoré riešenie mali. Z týchto dvoch hodnôt určíme pravdepodobnosť pre veľkosť s. Na základe týchto grafov budeme približne vedieť, ako sa odlišuje skutočná pravdepodobnosť riešenia od aproximácii.

V odhadoch vystupuje náhodná hashovacia funkcia. Z tohto dôvodu aj pri praktickom overovaní je nutné generovať náhodné hashovacie funkcie a to pomocou univerzálneho hashovania vektorov.

Definícia 2.3.4. *Nech existujú konštanty* $c_i \xleftarrow{\$} Z_n$ *pre* $i \in \{0, \ldots, m\}$ *kde* n *je veľkosť hashovacej funkcie. Nech* $H = (h_0 \ldots h_m)$ *je vektor hodnôt.*

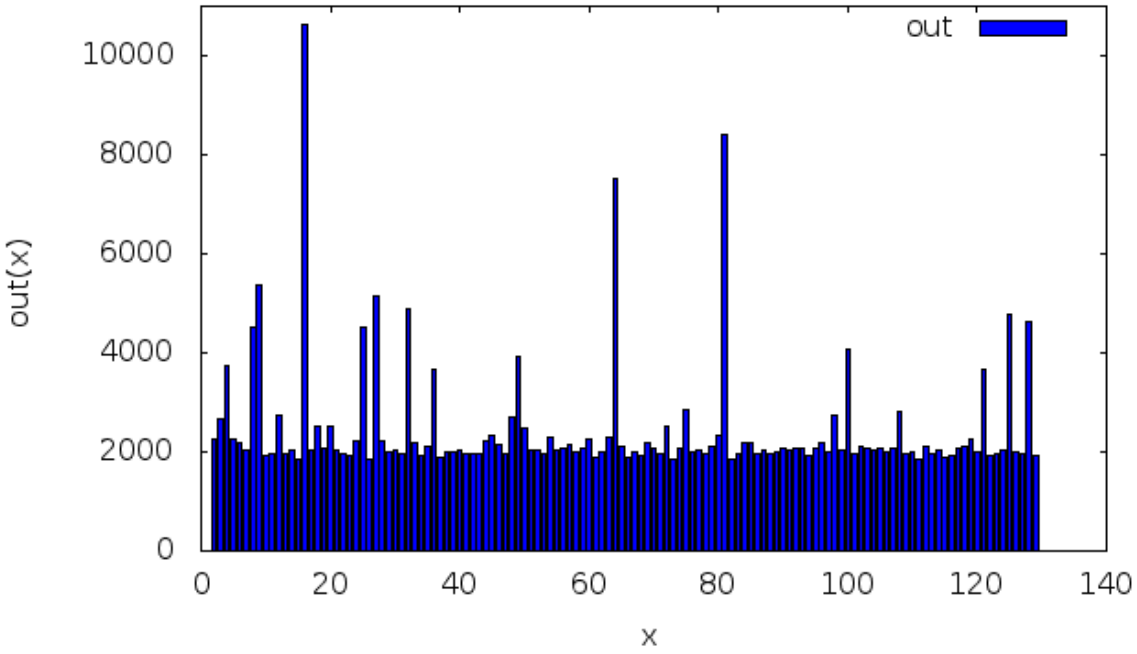
$$h(H) = \sum_{i=0}^{m} h_i c_i \qquad (\text{mod } n)$$

Potom funkciu h *s konštantami* c_i *nazveme univerzálne hashovanie vektora.*

Testované náhodné hashovacie funkcie budú fungovať na tomto princípe, pričom sa na každé číslo budeme pozerať ako vektor bitov.

Praktický výpočet 1

Prvý spôsob ako prakticky spočítať túto pravdepodobnosť spočíval v upravení spôsobu 2 zo sekcie 2.2 z predchádzajúcej časti. Investovali sme dva dni do upravenia predvypočítaného súboru tak, aby si pre každé číslo z množiny M pamätal 128 bitov informácii namiesto 32 bitov. Teda sme v konštantnom čase (ak nepočítame diskové operácie) vedeli povedať o každom čísle z množiny M, ako sa správa pri bázach 2-129. Veľkosť tohto súboru sa vyšplhala na 12 GiB, takže nebolo rozumné ho čítať viac než raz. Preto sme upravili fázu dva tak aby fungovala pre 25 funkcií naraz, a teda sa súbor čítala len jeden krát. Fáza 3 sa už vykonávala sekvenčne pre každú funkciu. Fáza 3 pôvodne fungovala tak, že si čísla patriace do neurčených tried uložila do vektorov. Vykonávať tento proces pre viac funkcií paralelne nebolo možné z dôvodu nedostatku pamäte. Tento sekvenčný beh trocha spomalil program, ale to nebol hlavný problém.



Graf 2.1: graf out(x)

Týmto spôsobom sme vygenerovali 8 hodnôt pre rozsah báz 128 pre veľkosť hashovacej funkcie medzi 700 a 1100. Skúšali sme taktiež generovať hodnoty pre rozsah báz 1024 ale nastali komplikácie. Prvý pokus bola veľkosť 1024 kde všetko prebehlo podľa plánu a 25 z 25 testovaných funkcií mali riešenie (to nie je nič nečakané, pretože obe aproximácie tam nadobúdajú hodnotu 1). Podľa predpočítaných odhadov sme odvodili druhú testovanú veľkosť 512, pretože tam odhady nadobúdali pravdepodobnosť od 0.86 po 1. V tomto bode malo 21 z 25 testovaných funkcií riešenie. Toto je stále vysoká pravdepodobnosť, pretože sme hľadali kde ten graf začína a ako sa potom správa. Preto sme uvážili, že väčšie hodnoty nemá zmysel testovať a skúsili sme veľkosť 448. Algoritmus 20 z testovaných funkcií zahodil, pretože po fáze 2 bolo viac ako $\frac{1}{4}$ tried neurčených. Po zmene tolerancie na $\frac{1}{2}$ s rizikom vyčerpania pamäti algoritmus spočítal 3 funkcie za 2.5 hodiny z ktorých mala jedna riešenie. Týmto tempom by sme sa k výsledku pre túto jednu veľkosť dopracovali za 21 hodín, pričom čas mal tendenciu rásť, pretože funkčné bázy mali vyššie hodnoty. Preto trebalo zmeniť riešenie.

Nevýhody tejto metódy spočívajú v náročnosti operácii. Čím je veľkosť hashovacej funkcie menšia, tým sa zvyšuje číselná hodnota funkčných báz. Preto je čoraz menej hodnôt určených vo fáze 2, dostavuje sa nedostatok pamäte a predvýpočet sa stáva zbytočným. Problém s pamäťou sa dá vyriešiť opakovaním fáze 3 pre každú triedu hashovej funkcie zvlášť, no tento postup vedie k spomaleniu algoritmu. Preto je toto riešenie nepraktické pre rozsah báz 4096 a treba zvýšiť výkon.

Praktický výpočet 2

Druhý spôsob je založený na paralelnom programovaní. Predchádzajúci spôsob sme prerobili do technológie CUDA, aby používal grafickú kartu na výpočty. Táto úprava výrazne urýchlila proces generovania.

Algoritmus 2.3.1.

Algoritmus má 2 fázy.

1. Algoritmus rozdelí množinu M na niekoľko súvislých úsekov, pričom každý úsek spracováva samostatné vlákno. Z každého úseku vyberie čísla, ktorých hash je medzi x a y kde x a y sú predom určené konštanty a uloží si túto hodnotu do dvojrozmerného pola. Následne pre každé pole z tohto 2D pola zavolá fázu 2.

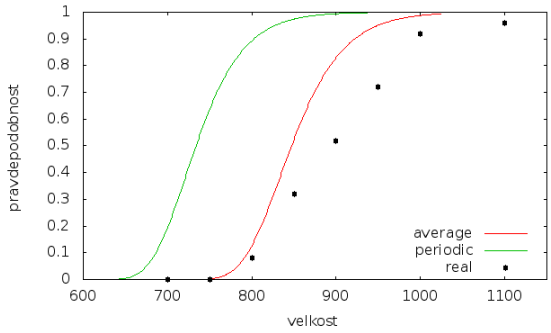
2. Vstupné pole sa rozdelí rovnomerne pre všetky thready. Každý thread testuje správanie svojho úseku na bázu i. Ak je niektoré číslo silné pseudoprvočíslo pri báze i zmení globálnú premennú na false a ostatné vlákna ukončia činnosť. Pokračuje sa bázou $i + 1$. Dochádza tu k takzvanému race-condition kedy môže niektoré vlákno prečítať hodnotu, ktorá sa vzápätí zmení, ale na efektivite a korektnosti to nič nemení takže sme to ignorovali.

Fáza 2 sa opakuje toľko krát, aby boli pokryté všetky triedy hashovacej funkcie. Keďže grafická karta má obmedzenú pamäť a chceli sme minimalizovať spúštanie fáze 1, ako kontrolný test sme nepoužili Eratostenovo sito lebo zaberá 256 MiB v pamäti. Namiesto neho sme použili už vypočítaný jedno kolový Rabin-Millerov test zobrazený algoritmom 2.3. Tento spôsob je oveľa rýchlejší ako prvý spôsob. Prvý spôsob pre rozsah báz 1024 a veľkosť funkcie 512 testoval hashovaciu funkciu v priemere 30 minút a tento spôsob približne 3 minúty.

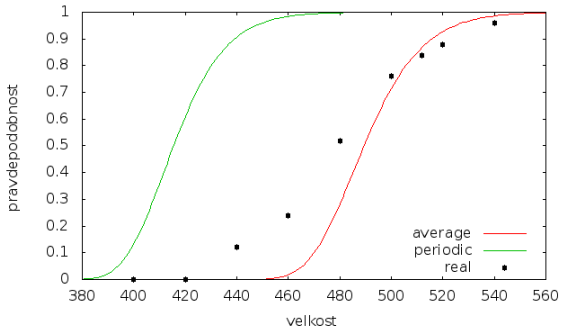
Body na grafoch 2.2, 2.3, 2.4 sme sa snažili voliť tak, aby sme zachytili celé správanie grafu voči aproximáciam 1 a 2. Ako si môžeme všimnúť, reálna pravdepodobnosť existencie riešenia pre náhodnú hashovaciu funkciu je väčšinou niekde medzi aproximáciou 1 (zelená krivka) a 2 (červená krivka). Pre malý rozsah báz je dokonca až pod aproximáciou 2.

2.3.3 Hľadanie minimálnej hashovacej funkcie

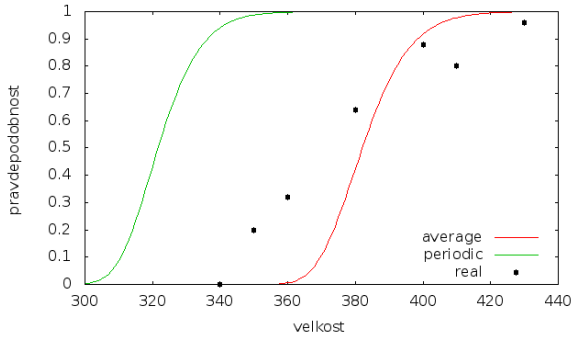
Predpokladali sme, že pre väčšie rozsahy báz sa táto pravdepodobnosť bude správať rovnako voči aproximáciam. Pri hľadaní finálneho minimálneho testu sme pre rozsah báz r odhadli veľkosť hashovacej funkcie tak, aby pravdepodobnosť existencie riešenia



Graf 2.2: rozsah báz 128



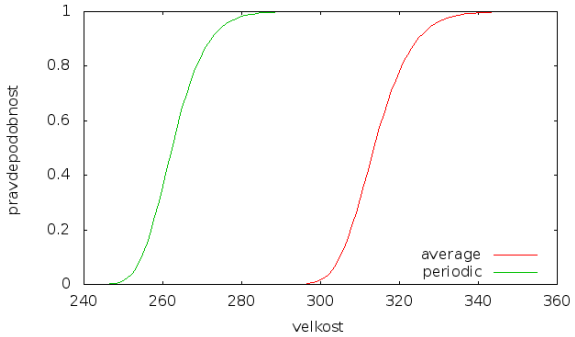
Graf 2.3: rozsah báz 1024



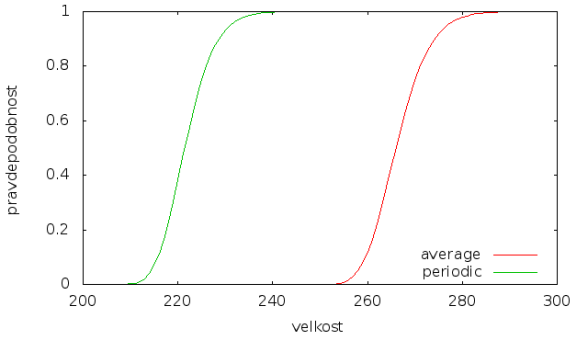
Graf 2.4: rozsah báz 4096

Na týchto grafoch os X zobrazuje veľkosť hashovacej funkcie. Os Y zobrazuje pravdepodobnosť existencie riešenia pre náhodnú hashovaciu funkciu s rozsahom báz, s veľkostou s. Zelená funkcia zobrazuje aproximáciu 1 a červená funkcia zobrazuje aproximáciu 2. Čierne body zobrazujú namerané hodnoty.

bola aspoň $\frac{1}{2}$. Na tieto odhady spe použili grafy 2.5, 2.6, 2.7, 2.8. Konkrétne bázy sme hľadali algoritmom 2.3.1.

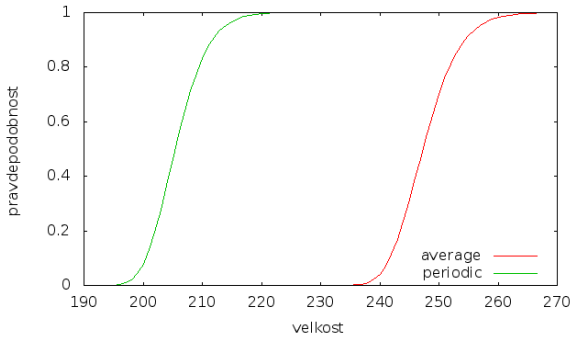


Graf 2.5: rozsah báz 16384

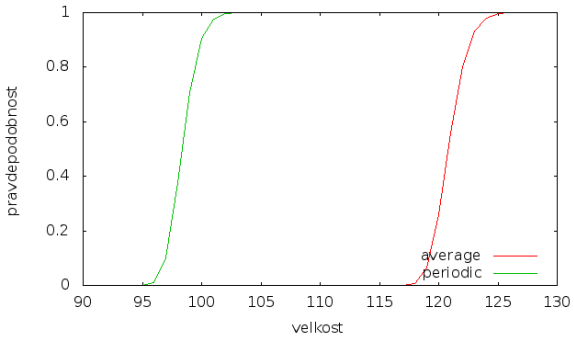


Graf 2.6: rozsah báz 65536

Skúsili sme teda pre rozsah báz 16 384 zvoliť veľkosť hashovacej funkcie 340. Podľa vygenerovaných grafov 2.5 sme očakávali, že ku každej hashovacej funkcii bude existovať riešenie. Zvolili sme teda hashovacie funkcie tvaru



Graf 2.7: rozsah báz 131072



Graf 2.8: rozsah báz 2^{32}

```
int hashh(long long x) {
      return ((x*c)>>k)%size
}
```

čo zodpovedá upravenému univerzálnemu hashovaniu integerov. Testovali sme niekoľko funkcií tohto tvaru a žiadna nemala riešenie v danom rozsahu. Pravdepodobne je to spôsobené tým, že veľa po sebe idúcich hodnôt má rovnakú hodnotu hashu a pseudoprvočísla sú distribuované viac náhodne.

Zvolili sme preto hashovaciu funkciu uvedenú v algoritme 2.5, ktorú sme našli v [1]. Na voľbe tejto funkcie príliš nezáleží a dala by sa použiť ľubovoľná podobná čo sa štatisticky dobre správa.

Algoritmus 2.5

```
int hashh(long long x) {
      x = ((x >> 16) ^ x) * 0x45d9f3b;
      x = ((x >> 16) ^ x) * 0x45d9f3b;
      x = ((x >> 16) ^ x);
      return x%256;
}
```

Táto hashovacia funkcia pre rozsah 16384 a veľkosť 340 riešenie mala. Pre túto funkciu sme našli riešenia s parametrami $r = 65536 \quad s = 256$ a $r = 131072 \quad s = 224$.

Najmenšie riešenie čo sme našli malo veľkosť 3808 bitov keďže každá z 224 konštánt sa dá kódovať 17timi bitmy. Spodný odhad veľkosti riešenia je 3200 bitov, pretože najväčší rozsah báz môžeme použiť 2^{32} ak nechceme implementovať 128 bitovú aritmetiku. Zvyšok vyplýva z obrázku 2.8. Rozdiel medzi našim odhadom a minimálnym je 608bitov. Menšie riešenie sme nehľadali, pretože riešenie veľkosti 3808 bitov sa hľadalo vyše 14hodín. Z tabuľky 2.3 a ostaných pozorovaní vyplýva, že vzťah medzi rozsahom báz a časom na nájdenie riešenia nie je lineárny, a preto by čas na hľadanie menšieho riešenia rástol veľmi rýchlo. Uspokojili sme sa preto s veľkosťou riešenia 3808 bitov.

Algoritmus 2.6 Bázy pre hashovaciu funkciu 2.5

```
uint16_t base[] = {  15591,  2018,   166,  7429,  8064, 16045, 10503,  4399,  1949,  1295,
 2776,  3620,   560,  3128,  5212,  2657,  2300,  2021,  4652,  1471,  9336,  4018,  2398,
20462, 10277,  8028,  2213,  6219,   620,  3763,  4852,  5012,  3185,  1333,  6227,  5298,
 1074,  2391,  5113,  7061,   803,  1269,  3875,   422,   751,   580,  4729, 10239,   746,
 2951,   556,  2206,  3778,   481,  1522,  3476,   481,  2487,  3266,  5633,   488,  3373,
 6441,  3344,    17, 15105,  1490,  4154,  2036,  1882,  1813,   467,  3307, 14042,  6371,
  658,  1005,   903,   737,  1887,  7447,  1888,  2848,  1784,  7559,  3400,   951, 13969,
 4304,   177,    41, 19875,  3110, 13221,  8726,   571,  7043,  6943,  1199,   352,  6435,
  165,  1169,  3315,   978,   233,  3003,  2562,  2994, 10587, 10030,  2377,  1902,  5354,
 4447,  1555,   263, 27027,  2283,   305,   669,  1912,   601,  6186,   429,  1930, 14873,
 1784,  1661,   524,  3577,   236,  2360,  6146,  2850, 55637,  1753,  4178,  8466,   222,
 2579,  2743,  2031,  2226,  2276,   374,  2132,   813, 23788,  1610,  4422,  5159,  1725,
 3597,  3366, 14336,   579,   165,  1375, 10018, 12616,  9816,  1371,   536,  1867, 10864,
  857,  2206,  5788,   434,  8085, 17618,   727,  3639,  1595,  4944,  2129,  2029,  8195,
 8344,  6232,  9183,  8126,  1870,  3296,  7455,  8947, 25017,   541, 19115,   368,   566,
 5674,   411,   522,  1027,  8215,  2050,  6544, 10049,   614,   774,  2333,  3007, 35201,
 4706,  1152,  1785,  1028,  1540,  3743,   493,  4474,  2521, 26845,  8354,   864, 18915,
 5465,  2447,    42,  4511,  1660,   166,  1249,  6259,  2553,   304,   272,  7286,    73,
 6554,   899,  2816,  5197, 13330,  7054,  2818,  3199,   811,   922,   350,  7514,  4452,
 3449,  2663,  4708,   418,  1621,  1171,  3471,    88, 11345,   412,  1559,   194 };
```

Algoritmus 2.7 BASE64 kódované bázy z algoritmu 2.6

```
EQsGPAcNCyINAgMlDQsmAgdpAgcqCwcDBg81LAYGBQ82ByMKCg8KChERCwoPKwcFBQMHKyICIgJE
NScKBwYLAgUCBwIGBQ8oAwUFAgIKLw0HKwYHBQYGDQYjBg8GDSgKCwIHAgIDDQMLDwoFCw4HCy8F
AgIGAgU3BgUHAgY6IwsFDBEGCgwGBgI1AgINBQ4HDwYNPgoGAwcHAw4FDkkPCwsGBREKBQMlMwoH
BSYMBQsFBwYFBigrOQoNBw8CCiIHJwoFAwYNCwUKKwoFAw4FAgUpBScuAgoCBQwDAgIFDysRKQIN
DyYLCwMiBQYDBwIlBQYKESMCDwYHBQMNDQwiAgwKDw0CAiIGBgUCBw0DBgsnKgcCBicvAxEFDQcC
LwMHBgsRJTAHJQsHCgMOJw4PKxECDAcNBQMGIiUDEQ0CBQoKLCUCAgoKBwMHAgcFKysLDzMNEQoL
AgUiEQICKgYGBS8PAgwHAwoPAwcMDA8rDgc6DQoGBiYiBQUNJgYLCgYHAjcCDQULLA8RAigCDw0G
AgMDAwMGJwULESUFBwYKBgwHBQ4KDEcKIwYLAwImAwIiChEqAgwGBgsoDAoGCgIDAzgLByoCJgwC
AgOoDAYFBTsPJgUFBQcCCgcCEQoLBgYGAgoGNgJSAyIODywFLgINBQwNCwonBSgDPAMqCwMuEQM=
```

počet bitov bázy	čas nájdenia riešenia
8	3 minúty
17	16 hodín

Tabuľka 2.3

Začínali sme so stavom kedy sme neboli schopný pre žiadnu hashovaciu funkciu nájsť riešenie. Postupnou analýzou problému a aplikovaním princípu neustáleho zlepšovania sme dospeli do stavu kedy, sme riešenie našli takmer okamžite.

KAPITOLA 3

Long long

Ako je to so 64 bitovými číslami? Dá sa spraviť podobná konštrukcia pre ne? 64 bitových čísiel je strašne veľa a už len preiterovať cez ne je časový problém. Navyše v predchádzajúcich spôsoboch sme mali problémy s pamäťou, ktoré sa dali vyriešiť na úkor času. Keby sme použili delenie 2, 3, 5, 7, na redukovanie tejto množiny rovnako ako sme to použili pri 32 bitových číslach stále by ich bolo $4.8 \cdot 10^9$ krát viac ako pri 32 bitovom prípade. Ak by sme nedelili prvými štyrmi prvočíslami ale prvými dvadsiatimi bolo by ich už "len" $2.5 \cdot 10^9$ krát viac. Touto cestou by sme možno množinu 64 bitových čísiel zredukovali ale triviálnych delení by bolo toľko, že by sa nám radšej oplatilo vykonať aspoň 1 dodatočný Miller-Rabin test.

3.1 2 bázový test

Jan Feistma [3] vypočítal a zverejnil všetky silné pseudoprvočísla pri báze 2 menšie ako 2^{64}. Vďaka týmto dátam môžeme efektívne nájsť deterministický 2 kolový test pre 64 bitové čísla. Kedže silných pseudoprvočísiel pri báze 2 menších ako 2^{64} je len 31 894 014, môžeme nájsť hashovaciu funkciu, ktorá vyvráti prvočíselnosť týchto čísiel.

Algoritmus 3.1 Metaalgoritmus pre 2 kolový test

```
uint64_t base[];
int hash(uint64_t x);
int prime(uint64_t x) {
  if(x==2 || x==3 || x==5 || x==7) return 1;
  if(x%2==0 || x<11) return 0;
  return is_SPRP(x,2) && is_SPRP(x,base[hash(x)]);
}
```

Na hľadanie vhodných báz pre konkrétnu hashovaciu funkciu sme použili CUDA algoritmus z predchádzajúcej časti, no namiesto hashovania čísel nesúdeliteľných s 2, 3,

báza	3	5	6	7	9	10
spsp(2,báza)	1501720	1330740	2080281	1363175	2542163	1505448
	11	12	13	14	15	
	1501710	1232021	1400025	1147899	1325227	

Tabuľka 3.1: Analýza počtu spsp pre 64 bitové čísla

5, 7, sme hashovali silné pseudoprvočísla pri báze 2. Týmto spôsobom sme našli bázy ktoré správne identifikovali všetky silné pseudoprvočísla pri báze 2 ako zložené čísla, no mohlo sa nám stať, že nejakému prvočíslu sa priradí báza ktorá s ním je súdeliteľná. To by znamenalo, že by sa vyhodnotilo ako zložené číslo, čo nie je v poriadku. Prvočísla väčšie ako najväčšia báza su automaticky v poriadku. Všetky prvočísla menšie ako maximálna báza sme hrubou silou skontrolovali, či dávaju správnu odpoveď na tento test. Keďže medzivýsledok binárneho umocňovania použitého v Miller-Rabinovom teste môže byť až 128 bitový, trebalo implementovať 128 bitovú aritmetiku. Použili sme aritmetiku z [18], kde je implementovaná Montgomeryho redukcia, čo znížilo zložitosť hľadania zvyšku po delení veľkým číslom.

Lema 3.1.1. *Ak je číslo n silné pseudoprvočíslo pri báze a tak je silné pseudoprvočíslo aj pri báze* $a^k \quad \forall k \geq 0$

Dôkaz. n je silné pseudoprvočíslo pri báze $a \Rightarrow$ podmienka (a) alebo (b) je splnená. $n - 1 = 2^s t$

(a) $a^t \equiv 1 \pmod{n} \Rightarrow a^{kt} \equiv 1^k \pmod{n} \Rightarrow (a^k)^t \equiv 1 \pmod{n}$

(b) $\exists i \in \{0, \ldots, s-1\} \quad a^{t2^i} \equiv -1 \pmod{n}$
$k = 2^j x$ kde x je nepárne

1 Nech $i \geq j$

$$\begin{aligned} (a^{2^j})^{t2^{i-j}} = a^{t2^i} \equiv -1 \pmod{n} \Rightarrow (a^{2^j})^{xt2^{i-j}} &\equiv (-1)^x \pmod{n} \\ \Rightarrow (a^k)^{t2^{i-j}} &\equiv -1 \pmod{n} \end{aligned}$$

2 Nech $i < j$

$$\begin{aligned} (a^{2^i})^t = a^{t2^i} \equiv -1 \pmod{n} \Rightarrow (a^{2^i})^{2^{j-i}xt} &\equiv (-1)^{2^{j-i}x} \pmod{n} \\ \Rightarrow (a^k)^t &\equiv 1 \pmod{n} \end{aligned}$$

a teda je splnená podmienka (a).

V tabuľke 3.1 neuvádzame hodnoty pre 2^k, pretože by skresľovali odhad. Podľa lemy 3.1.1 každé silné pseudoprvočíslo pri báze 2 je aj silné pseudoprvočíslo pri báze 2^k a teda sú tieto hodnoty známe. V odhade sme ich neuvažovali, pretože v porovnaní s ostatnými číslami je hodnôt 2^k zanedbateľne málo.

Na základe tabuľky 3.1 sme pomocou odhadov z predchádzajúcej časti odhadli veľkosť hashovacej funkcie na 524 288 pre bázy s rozsahom do 2048 a 262 144 pre bázy s rozsahom do 65 536. Hľadali sme veľkosti hashovacích funkcií len v tvare 2^k aby sa nemusela vykonávať pomalá inštrukcia modulo. Tieto odhady sú veľmi skreslené, keďže vznikli z aproximácii odhadu a tiež z málo vstupných dát. Podľa týchto informácii sme našli funkčné bázy pre hashovaciu funkciu 3.2 kde $s = 524\,288$ a $s = 262\,144$. Veľkosti týchto funkcii su 704 KiB a 512 KiB. Tieto veľkosti môžu na prvý pohľad pôsobiť kontraintuitívne, pretože hashovacie funkcie pre 32 bitové hodnoty mali len 4 KiB a predsa silných pseudoprvočísel pri báze 2 (32 miliónov) je oveľa menej ako 4 miliardy 32 bitových hodnôt. Problém je v tom, že pridanie dodatočnej bázy neredukuje množinu tak rýchlo. 64 bitových spsp$((2, i))$ je asi 10% z počtu spsp(2) pričom 32 bitových spsp(2) je asi $5 \cdot 10^{-5}$ oproti počtu 32 bitových čísiel. Z tohto vyplýva, že ak je číslo spsp(i) tak je aj spsp(j) s nezanedbateľnou pravdepodobnosťou. Veľkosť množiny závisí najmä od absolútneho počtu "outov". V prípade 1-kolového testu na 32 bitových číslach to bolo priemerne 2471 no v prípade 2 kolového testu 64 bitových čísel to je až 1.5 milióna.

Algoritmus 3.2 Hashovacia funkcia pre 64 bitové čísla

```
1 int hash(uint64_t x) {
2     x = ((x >> 32) ^ x) * 0x45d9f3b3335b369;
3     x = ((x >> 32) ^ x) * 0x3335b36945d9f3b;
4     x = ((x >> 32) ^ x);
5     return x%s;
6 }
```

Má zmysel hľadať jedno-kolový test? Podľa dát [3] existuje 32 miliónov silných pseudoprvočísiel pri báze 2. Za predpokladu, že pre každú bázu i existuje 32 miliónov silných pseudoprvočísiel, by podľa heuristicky odvodených odhadov náhodná hashovacia funkcia s rozsahom báz 2^{64} taká, že má riešenie s pravdepodobnosťou aspoň $\frac{1}{2}$, musela mať veľkosť aspoň 852110 (toto číslo je dosť nepresné, každopádne uloženie takejto tabuľky stojí 6.5 MiB). Ako sme sa v tejto práci presvedčili, nájsť tieto bázy je problém, pretože si to vyžaduje iterovať všetky 64 bitové čísla a roztriediť ich do tried podľa hodnoty hashovacej funkcie (žiadna trieda sa nebude zmestiť do pamäti celá). Navyše, pre každú triedu treba nájsť funkčnú bázu, čo znamená niekoľko krát iterovať tieto čísla.

KAPITOLA 4

Porovnanie testov

4.1 Teoretický odhad priemerného prípadu

Veta 4.1.1. *Nech $X : \Omega \to N$ kde $X(\Omega) = N$ je diskrétna náhodná premenná. Potom platí*

$$E(X) = \sum_{x \in X(\Omega)} xP[X = x] = \sum_{x>0} xP[X = x] = \sum_{x>0} P[X \geq x]$$

Dôkaz.

$$E(X) = \sum_{i>0} iP[X = i] = \sum_{i>0} \sum_{j=1}^{i} P[X = i] = \sum_{i} \sum_{j} (1 \leq j \leq i)P[X = i] =$$
$$= \sum_{j} \sum_{i} (1 \leq j \leq i)P[X = i] = \sum_{j \geq 1} \sum_{i \geq j} P[X = i] = \sum_{j \geq 1} P[X \geq j]$$

Nech $X : \Omega \to \{1, \ldots, k\}$ je diskrétna náhodná premenná reprezentujúca počet vykonaných kôl k-kolového Rabin-Millerovho testu. Nech $\{a_1, \ldots, a_k\}$ sú bázy k-kolového Rabin-Millerovho testu do ktorého vstupujú čísla z množiny M. Označme počet prvočísiel v množine M M_p. Potom strednú hodnotu počtu kôl tohto testu možno vypočítať nasledovne

$$A_0 = \emptyset$$
$$A_i = A_{i-1} \cup a_i \quad 0 < i \leq k$$
$$B_i = \{x \in M \ \wedge \ \text{x je silné pseudoprvočíslo pri bázach} A_i\}$$
$$B_i = \emptyset \quad i \geq k \ ^1$$
$$P[X \geq i] = \frac{M_p + |B_{i-1}|}{|M|}$$

[1] táto podmienka je implicitne splnená ak sa jedná o korektný k kolový test

Silných pseudoprvočísiel v 32 bitových číslach je podľa tabuľky 2.1 zanedbateľne málo v porovnaní s počtom 32 bitových čísiel a preto strednú hodnotu ovplyvňujú minimálne. Z tohto pozorovania vyplýva, že nezáleží na poradí testovaných báz. Preto môžeme strednú hodnotu počtu kôl vyjadriť ako

$$E(X) = \sum_{j=1}^{k} P[X \geq j] \doteq 1 + \sum_{j=1}^{k-1} \frac{M_p}{|M|} = 1 + \frac{(k-1) \cdot M_p}{|M|}$$

1-kolový test Tento test má na začiatku triviálne delenie preto sa silný test prvočíselnosti vykonáva len pre zlomok čísiel. Preto za M zvolíme množinu 32 bitových čísiel nesúdelitelných s číslami 2,3,5,7 a spočítame strednú hodnotu. Tú následne vynásobíme pravdepodobnosťou, že číslo nie je deliteľné číslami 2,3,5,7

$$E(X) \doteq 0.228$$

3-kolový test Aby sme boli objektívny uvažujeme aj tento test s triviálnym delením číslami 2,3,5,7 na začiatku (na korektnosť to nemá vplyv, ide len o rýchlosť). Preto

$$E(X) \doteq 0.322$$

A teda by mal byť 1 kolový test 1.41-krát rýchlejší v priemernom prípade za predpokladu, že je hashovacia funkcia dostatočne rýchla.

Definícia 4.1.1. $\pi(x) = |\{a \in \mathbb{P} \ \wedge \ a < x\}|$

Veta 4.1.2. *(Čebyševove nerovnosti) Existujú reálne konštanty c_1, c_2, že pre $x \geq 2$ platí:*

$$c_1 \frac{x}{\ln x} \leq \pi(x) \leq c_2 \frac{x}{\ln x}$$

Na teoretický odhad priemerného prípadu 64 bitových čísle potrebujeme poznať počet prvočísiel menších ako 2^{64}. Podľa [2] $\pi(10^{19}) = 234\ 057\ 667\ 276\ 344\ 607$ a $\pi(2 \cdot 10^{19}) = 460\ 637\ 655\ 126\ 005\ 490$. Na základe týchto hodnôt a vety 4.1.2 sme odhadli $\pi(2^{64}) \doteq 425\ 719\ 147\ 865\ 338\ 065$. Presná hodnota na tento odhad nie je nutná.

2-kolový test Keďže spsp(2) 64bitových čísiel je len 32 miliónov strednú hodnotu to ovplyvní až na 12 tom desatinnom mieste. Preto strednú hodnota počtu kôl môžeme aproximovať

$$E(X) \doteq 1 + \frac{1 \cdot \pi(2^{64})}{2^{64}} \doteq 1.023$$

7-kolový test

$$E(X) \doteq 1 + \frac{6 \cdot \pi(2^{64})}{2^{64}} \doteq 1.1384$$

A teda by mal byť 2 kolový test s hashovaním 1.113 krát rýchlejší ako 7-kolový Jim Sinclairov test v priemernom prípade.

4.2 Praktické porovnanie priemerného prípadu

4.2.1 32bit

Pomocou pseudonáhodného generátora čísel *Mersenne twister* (implementácia zo štandardnej knižnice `<random>` v `c++`) sme vygenerovali n čísiel s počiatočnou hodnotou j na ktorých sme spustili test prvočíselnosti. Výsledné časy pre rôzne n, j a rôzne testy zobrazuje tabuľka 4.1. V každom z testovaných prípadov bolo zhruba 4.7% prvočísiel, čo zodpovedá podielu prvočísiel v 32 bitových číslach.

test	$j = 3$ $n = 10^6$	$j = 2$ $n = 5 \cdot 10^6$	$j = 1$ $n = 10^7$
prime_1a	181	538	1039
prime_1b	168	526	1000
prime_2	215	654	1234
prime_3	213	752	1460
probab_prime_p	2244	10809	21427
bpsw_prp	2732	13237	27195
PrimeQ	4992	24804	49281

Tabuľka 4.1: Porovnanie priemerného prípadu 32 bitových testoch. Uvedené hodnoty sú časy behu programu v milisekundách.

prime_1a	1 kolový test s delením 2,3,5,7, s veľkosťou hashovacej funkcie $s = 224$
prime_1b	1 kolový test s delením 2,3,5,7, s veľkosťou hashovacej funkcie $s = 1024$ (algo. 2.3)
prime_2	2 kolový test s delením 2,3,5,7, s bázamy $\{2, 15\}$, s overením či číslo nie je jeden z 59 kontrapríkladov.
prime_3	3 kolový test s delením 2,3,5,7, s bázamy $\{2, 7, 61\}$
probab_prime_p	funkcia z knižnice GMP. volanie `mpz_probab_prime_p(x,25)`
bpsw_prp	BPSW test. implementácia pomocou GMP z [8]
PrimeQ	Mathematica, test prvočíselnosti
prime_2_64	(64bit) 2 kolový test s veľkostou hashovacej funkcie $s = 262144$
prime_sinclair	(64bit) 7 kolový test (pozri tabuľku 2.2)

4.2.2 64bit

Podobne ako pri 32 bitových testoch, aj tu sme na testovanie použili *Mersenne twister*, avšak jeho 64 bitovú inštanciu. Aj tu sa podiel prvočísiel blížil ku podielu prvočísiel v 64 bitových číslach pre dostatočne veľké hodnoty n. Teda môžeme týmto spôsobom testovať priemerný prípad. Výsledky ukazuje tabuľka 4.2.

test	$j = 3$ $n = 10^6$	$j = 2$ $n = 5 \cdot 10^6$	$j = 1$ $n = 10^7$
prime_2_64	1625	7649	15466
prime_sinclair	1924	9303	18338
probab_prime_p	3144	15496	30831
bpsw_prp	4549	22132	44539
PrimeQ	7940	39703	79202

Tabuľka 4.2: Porovnanie priemerného prípadu 64 bitových testoch. Uvedené hodnoty sú časy behu programu v milisekundách.

4.3 Teoretický odhad najhoršieho prípadu

32 bitové čísla Najhorší prípad nastane, ak ako vstup zvolíme prvočíslo. Preto 1-kolový test v najhoršom prípade spraví $\log n + (hash)$ operácii a 3-kolový test $3 \log n$ operácii. Teda teoreticky, najhorší prípad bude asymptoticky 3 krát lepší.

64 bitové čísla 2-kolový test v najhoršom prípade spraví $2 \log n + (hash)$ operácii a 7-kolový test $7 \log n$ operácii. Teda ak je hashovacia funkcia rýchla mal by byť 2-kolový test v najhorším prípade 3.5 krát rýchlejší

4.4 Praktické porovnanie najhoršieho prípadu

4.4.1 32bit

Najhorší prípad sa testoval pomocou *Mersenne twister*, ale testovalo sa len 1000 čísel pričom každé z čísiel sa otestovalo na prvočíselnosť 1000-krát. Výsledný čas sa potom normalizoval na jeden test. Najhorší prípad sa ťažko meria, pretože všetky testy sú veľmi rýchle. Navyše `mpz_probab_prime_p` nie je deterministický test, čo ovplyvňovalo meranie. Preto sú hodnoty v tabuľke 4.3 len orientačné a netreba ich brať veľmi seriózne. Rozdiel medzi `is_prime_i` a `mpz_probab_prime_p` a `mpz_bpsw_prp` je mnohonásobne väčší, takže to nie je chyba merania a jednokolové testy resp. 3 kolový test sú podstatne rýchlejšie.

prime_1a	**prime_1b**	**prime_2**	**prime_3**	**probab_prime_p**	**bpsw_prp**
2.369 μs	2.423 μs	3.045 μs	4.282 μs	58.198 μs	26.959 μs
2.413 μs	2.369 μs	3.090 μs	4.109 μs	58.763 μs	26.277 μs
2.360 μs	2.417 μs	3.033 μs	4.140 μs	59.822 μs	26.607 μs
2.393 μs	2.400 μs	2.977 μs	4.230 μs	58.942 μs	26.394 μs
2.412 μs	2.366 μs	2.977 μs	4.328 μs	57.655 μs	26.417 μs

Tabuľka 4.3

4.4.2 64bit

Najhorší prípad sa meral rovnako ako pri 32 bitových testoch. Jediná zmena bola použitie 64 bitovej verzie *Mersenne twister*. Výsledky zobrazuje tabuľka 4.4.

prime_2_64	**prime_sinclair**	**probab_prime_p**	**bpsw_prp**
15.755 μs	53.055 μs	147.543 μs	67.574 μs
15.623 μs	53.032 μs	147.100 μs	67.692 μs
15.469 μs	52.676 μs	147.906 μs	68.192 μs
15.587 μs	53.184 μs	149.042 μs	67.355 μs
15.557 μs	52.785 μs	148.276 μs	68.070 μs

Tabuľka 4.4

Záver

V tejto práci sme našli 1-kolový Rabin-Millerov test pre čísla z rozsahu 2 až 2^{32} a 2-kolový Rabin-Millerov test pre čísla z rozshau 2 až 2^{64}. Oba tieto typy testov, či už mali čo najmenej báz alebo nie, boli jednoznačne efektívnejšie v priemernom aj najhoršom prípade ako známe testy. Tiež sme heuristicky odvodili odhad minimálnej veľkosti týchto hashovacích funkcií. Na základe tohto heuristického odhadu sa dá približne odvodiť, že jednokolový test pre čísla z rozsahu 2 až 2^{64} pre rozsah báz 256 potreboval hashovaciu funkciu veľkosti $11 \cdot 10^6$. Ak by sa investoval dostatok času a ak sú naše heuristické odhady korektné, nebol by problém takýto test nájsť. V praxi totiž môžu byť testy s veľkými predpočítanými tabuľkami pomalšie, napríklad kvôli primalým lokálnym cache na procesore. Ďalej by sa dalo preskúmať túto metódu pre iné typy pseudoprvočísiel, napríklad Lucasove, a zistiť, čí náhodou pre ten prípad nie sú hashovacie funkcie menšie.

Literatúra

[1] *Hash function.* http://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key/12996028#12996028, . – Accessed: 2014-05-23

[2] *Prime values.* http://web.archive.org/web/20130516122549/http://primes.utm.edu/notes/md6-96.html, . – Accessed: 2014-05-10

[3] *Pseudoprimes Database Statistics.* http://web.archive.org/web/20140510203515/http://www.janfeitsma.nl/math/psp2/statistics, . – Accessed: 2014-05-10

[4] *The best known SPRP bases sets.* http://miller-rabin.appspot.com/, . – Accessed: 2014-05-10

[5] Alford, William R. ; Granville, Andrew ; Pomerance, Carl: There are infinitely many Carmichael numbers. In: *Annals of Mathematics* (1994), S. 703–722

[6] Arnault, François: The Rabin-Monier theorem for Lucas pseudoprimes. In: *Mathematics of Computation of the American Mathematical Society* 66 (1997), Nr. 218, S. 869–881

[7] Baillie, Robert ; Wagstaff, Samuel S.: Lucas pseudoprimes. In: *Mathematics of Computation* 35 (1980), Nr. 152, S. 1391–1417

[8] Cleaver, David: *Baillie-Pomerance-Selfridge-Wagstaff test implementation (mpz_prp.c).* http://sourceforge.net/projects/mpzprp/files/, . – Accessed: 2014-05-23

[9] Grantham, Jon: Frobenius pseudoprimes. In: *Mathematics of computation* 70 (2001), Nr. 234, S. 873–891

[10] Jaeschke, Gerhard: On strong pseudoprimes to several bases. In: *Mathematics of Computation* 61 (1993), Nr. 204, S. 915–926

[11] Monier, Louis: Evaluation and comparison of two efficient probabilistic primality testing algorithms. In: *Theoretical Computer Science* 12 (1980), Nr. 1, S. 97–108

[12] Pomerance, Carl ; Selfridge, John L. ; Wagstaff, Samuel S.: The pseudoprimes to $25 \cdot 10^9$. In: *Mathematics of Computation* 35 (1980), Nr. 151, S. 1003–1026

[13] Rabin, Michael O.: Probabilistic algorithm for testing primality. In: *Journal of number theory* 12 (1980), Nr. 1, S. 128–138

[14] Sleziak, Martin: *Teória čísel. Poznámky k prednáške*

[15] Solovay, Robert ; Strassen, Volker: A fast Monte-Carlo test for primality. In: *SIAM journal on Computing* 6 (1977), Nr. 1, S. 84–85

[16] Váňa, Tomáš: *Silné Pseudoprvočísla*, Univerzita Komenského v Bratislave, bakalárska práca, 2007

[17] Wagstaff Jr, Samuel S.: *Cryptanalysis of number theoretic ciphers*. CRC Press, 2002

[18] Warren Jr, Henry S.: Montgomery Multiplication. (2012)